🔥 PyTorch Course Summary

📝 Communication Standards

Tone & Language

- Use clear, concise technical language
- Maintain a professional yet approachable tone
- Explain complex concepts with practical examples
- Use appropriate mathematical notation when discussing algorithms
- Include visual explanations for complex neural network architectures

Documentation & References

- Reference PyTorch official documentation (pytorch.org)
- Cite academic papers for advanced concepts
- Link to relevant research papers on arXiv
- Include references to standard machine learning textbooks
- Document all data preprocessing steps

Response Structure

- Begin with a high-level overview
- Follow with detailed technical explanation
- Include code examples when applicable
- End with practical applications
- Provide performance metrics and benchmarks

💡 Problem-Solving Approach

Analysis Methods

1. Data Exploration

    - Statistical analysis
    - Visualization techniques
    - Distribution analysis
    - Missing data handling
    - Outlier detection

2. Model Selection

    - Use case evaluation
    - Architecture comparison
    - Performance requirements
    - Resource constraints
    - Scalability considerations

Solution Development

1. Iterative Process

    - Prototype development
    - Validation methods
    - Performance optimization
    - Error analysis
    - Model refinement

2. Implementation Strategy

    - Environment setup
    - Data pipeline creation
    - Model architecture design
    - Training workflow
    - Deployment planning

🎯 Domain Expertise

Primary Focus Areas

1. Deep Learning Fundamentals

    - Neural Network Architecture
    - Optimization Algorithms
    - Loss Functions
    - Backpropagation
    - Gradient Descent Variations

2. PyTorch Ecosystem

    - Tensor Operations
    - Autograd Mechanics
    - Dataset Management
    - Model Deployment
    - Distribution Strategies

Key Skills

1. Technical Competencies

    - Python Programming
    - Mathematical Foundations
    - Statistical Analysis
    - GPU Computing
    - Distributed Training

2. Applied Skills

    - Model Design
    - Hyperparameter Tuning
    - Performance Optimization
    - Debug & Troubleshooting
    - Production Deployment

🛠 Technical Implementation

Code Standards

1. Style Guidelines

    - Follow PEP 8 conventions
    - Use type hints
    - Implement proper error handling
    - Write comprehensive docstrings
    - Maintain consistent naming conventions

2. Best Practices

    - Modular architecture
    - Clean code principles
    - Version control workflow
    - Testing protocols
    - Documentation standards

🎓 Educational Approach

Teaching Methods

1. Theoretical Foundation

    - Mathematical concepts
    - Algorithm explanation
    - Architecture design
    - Performance analysis
    - Optimization techniques

2. Practical Application

    - Hands-on exercises
    - Real-world projects
    - Case studies
    - Performance profiling
    - Deployment scenarios

Learning Resources

1. Core Materials

  - Official PyTorch tutorials
  - Research papers
  - Technical blogs
  - Video lectures
  - Code repositories

2. Supplementary Resources

  - Practice datasets
  - Example projects
  - Community forums
  - Online courses
  - Reference implementations

🤝 Interaction Guidelines

User Support

1. Technical Assistance

  - Implementation guidance
  - Debugging help
  - Performance optimization
  - Best practices advice
  - Architecture reviews

2. Learning Support

  - Concept clarification
  - Resource recommendations
  - Progress tracking
  - Feedback provision
  - Mentorship guidance

📊 Quality Standards

Code Quality

1. Implementation Standards
    - Clean code principles
    - Performance optimization
    - Memory efficiency
    - GPU utilization
    - Scalability considerations
2. Testing Requirements
    - Unit tests
    - Integration tests
    - Performance benchmarks
    - Edge case handling
    - Error recovery

🔒 Security & Safety

Security Practices

1. Data Protection
    - Input validation
    - Secure data handling
    - Access control
    - Environment security
    - Dependency management
2. Model Security
    - Adversarial defense
    - Model robustness
    - Privacy preservation
    - Ethical considerations
    - Bias mitigation

🔄 Continuous Improvement

Development Areas

1. Technical Enhancement
    - Performance optimization
    - Feature implementation
    - Architecture updates
    - Tool integration
    - Documentation improvement
2. Learning Integration
    - New research findings
    - Community feedback
    - User suggestions
    - Industry trends
    - Best practices updates

Topic Categories

Part 1: Fundamentals (Sections 1-32)

- Python Basics
- Linear Algebra
- Calculus
- Probability & Statistics
- Machine Learning Basics

Part 2: PyTorch Foundations (Sections 33-64)

- Tensor Operations
- Autograd Mechanics
- Neural Network Basics
- Optimization Algorithms
- Loss Functions

Part 3: Deep Learning (Sections 65-96)

- Convolutional Networks
- Recurrent Networks
- Transformers
- Generative Models
- Transfer Learning

Part 4: Advanced Topics (Sections 97-128)

- Distributed Training
- Model Optimization
- Custom Extensions
- C++ Frontend
- Mobile Deployment

Part 5: Production (Sections 129-160)

- Model Serving
- Performance Tuning
- Scaling Strategies
- Monitoring
- Maintenance

Meta Information

Version: 2.0.0 Last Updated: 2025-03-05 19:37:43 UTC License: MIT

🚀 Quick Navigation

📚 Course Overview

This comprehensive PyTorch course covers:

- 🤖 Machine Learning fundamentals
- 🧠 Deep Learning with PyTorch
- 💻 Hands-on coding exercises
- 🛠 Practical implementations

📋 Prerequisites

- ✅ 3-6 months Python coding experience
- ✅ Basic understanding of programming concepts

🔗 Additional Resources

- 📖 PyTorch Documentation
- 📚 Learn PyTorch
- 💬 GitHub Discussions

📖 Course Content

Section 1: Main Topics

Key Topics:

- This comprehensive course will teach you the foundations of machine learning and deep learning using PyTorch
- PyTorch is a machine learning framework written in Python
- You'll learn machine learning by writing PyTorch code

▶ 📄 Click to view detailed content

```
This comprehensive course will teach you the foundations of machine learning and
deep learning
using PyTorch. PyTorch is a machine learning framework written in Python. You'll
learn machine
learning by writing PyTorch code. So when in doubt, run the provided code and
experiment.
Your teacher for this course is Daniel Bourke. Daniel is a machine learning
engineer and popular
course creator. So enjoy the course and don't watch the whole thing in one sitting.
Hello, welcome to the video. It's quite a big one. But if you've come here to learn
machine
learning and deep learning and PyTorch code, well, you're in the right place. Now,
this video and
tutorial is focused for beginners who have got about three to six months of Python
coding experience.
So we're going to cover a whole bunch of important machine learning concepts
by writing PyTorch code. Now, if you get stuck, you can leave a comment below or
post on the course
GitHub discussions page. And on GitHub is where you'll be able to find all the
materials that we cover,
as well as on learn pytorch.io. There's an online readable book version of this
```

course there.
But if you finish this video and you find that, hey, I would still like to learn more PyTorch.
I mean, you can't really cover all the PyTorch in a day that video titles just apply on words of
the length of video. That's an aside. There is five more chapters available at learn pytorch.io,
covering everything from transfer learning to model deployment to experiment tracking.
And all the videos to go with those are available at zero to mastery.io. But that's enough for me.
Having machine learning and I'll see you inside.
Hello, my name is Daniel and welcome to the deep learning with
PyTorch course. Now, that was too good not to watch twice. Welcome to the deep learning with
cools at fire PyTorch course. So this is very exciting. Are you going to see that animation
quite a bit because, I mean, it's fun and PyTorch's symbol is a flame because of torch.
But let's get into it. So naturally, if you've come to this course, you might have already
researched what is deep learning, but we're going to cover it quite briefly.
And just in the sense of how much you need to know for this course, because we're going to be
more focused on, rather than just definitions, we're going to be focused on getting practical
and seeing things happen. So if we define what machine learning is, because as we'll see in a
second, deep learning is a subset of machine learning. Machine learning is turning things
data, which can be almost anything, images, text, tables of numbers, video, audio files,
almost anything can be classified as data into numbers. So computers love numbers,
and then finding patterns in those numbers. Now, how do we find those patterns? Well,
the computer does this part specifically a machine learning algorithm or a deep learning
algorithm of things that we're going to be building in this course. How? Code and math. Now,
this course is code focused. I want to stress that before you get into it. We're focused on
writing code. Now, behind the scenes, that code is going to trigger some math to find patterns in
those numbers. If you would like to deep dive into the math behind the code, I'm going to be
linking extra resources for that. However, we're going to be getting hands on and writing lots of
code to do lots of this. And so if we keep going to break things down a little bit more,
machine learning versus deep learning, if we have this giant bubble here of artificial
intelligence, you might have seen something similar like this on the internet. I've just
copied that and put it into pretty colors for this course. So you've got this overarching
big bubble of the topic of artificial intelligence, which you could define as, again, almost anything

```
you want. Then typically, there's a subset within artificial intelligence, which is
known as machine
learning, which is quite a broad topic. And then within machine learning, you have
another topic
called deep learning. And so that's what we're going to be focused on working with
PyTorch,
writing deep learning code. But again, you could use PyTorch for a lot of different
machine
learning things. And truth be told, I kind of use these two terms interchangeably.
Yes, ML is the
broader topic and deep learning is a bit more nuanced. But again, if you want to
form your
own definitions of these, I'd highly encourage you to do so. This course is more
focused on,
rather than defining what things are, is seeing how they work. So this is what
we're focused on doing.
Just to break things down, if you're familiar with the fundamentals of machine
learning,
you probably understand this paradigm, but we're going to just rehash on it anyway.
So if we
consider traditional programming, let's say you'd like to write a computer program
that's enabled to,
or has the ability to reproduce your grandmother's favorite or famous roast chicken
dish. And so we
might have some inputs here, which are some beautiful vegetables, a chicken that
you've raised on the
farm. You might write down some rules. This could be your program, cut the
vegetables, season the
chicken, preheat the oven, cook the chicken for 30 minutes and add vegetables. Now,
it might not
be this simple, or it might actually be because your Sicilian grandmother is a
great cook. So she's
put things into an art now and can just do it step by step. And then those inputs
combined with
those rules makes this beautiful roast chicken dish. So that's traditional
programming. Now,
a machine learning algorithm typically takes some inputs and some desired outputs
and then
```

Section 2: Main Topics

Key Topics:

- So where in traditional program, we had to hand write all of these rules, the ideal machine learning algorithm will figure out this bridge between our inputs and our idealized output
- Now, in the machine learning sense, this is typically described as supervised learning, because you will have some kind of input with some kind of output, also known as features, and also known as labels

- And the machine learning algorithm's job is to figure out the relationships between the inputs or the features and the outputs or the label

▶ 📄 Click to view detailed content

```
figures out the rules. So the patterns between the inputs and the outputs. So where
in traditional
program, we had to hand write all of these rules, the ideal machine learning
algorithm will figure
out this bridge between our inputs and our idealized output. Now, in the machine
learning sense, this
is typically described as supervised learning, because you will have some kind of
input with
some kind of output, also known as features, and also known as labels. And the
machine learning
algorithm's job is to figure out the relationships between the inputs or the
features and the outputs
or the label. So if we wanted to write a machine learning algorithm to figure out
our Sicilian
grandmother's famous roast chicken dish, we would probably gather a bunch of inputs
of ingredients
such as these delicious vegetables and chicken, and then have a whole bunch of
outputs of the
finished product and see if our algorithm can figure out what we should do to go
from these
inputs to output. So that's almost enough to cover of the difference between
traditional programming
and machine learning as far as definitions go. We're going to get hands on encoding
these sort
of algorithms throughout the course. For now, let's go to the next video and ask
the question,
why use machine learning or deep learning? And actually, before we get there, I'd
like you to
think about that. So going back to what we just saw, the paradigm between
traditional programming
and machine learning, why would you want to use machine learning algorithms rather
than
traditional programming? So if you had to write all these rules, could that get
cumbersome?
So have a think about it and we'll cover it in the next video.
Welcome back. So in the last video, we covered briefly the difference between
traditional programming and machine learning. And again, I don't want to spend too
much time
on definitions. I'd rather you see this in practice. And I left you with the
question,
why would you want to use machine learning or deep learning? Well, let's think of a
good reason.
Why not? I mean, if we had to write all those handwritten rules to reproduce Alsace
and grandmother's
roast chicken dish all the time, that would be quite cumbersome, right? Well, let's
draw a line
on that. Why not? What's a better reason? And kind of what we just said, right? For
a complex
problem, can you think of all the rules? So let's imagine we're trying to build a
self-driving car.
```

Now, if you've learned to drive, you've probably done so in maybe 20 hours, 100 hours. But now,
I'll give you a task of writing down every single rule about driving. How do you back out of your
driveway? How do you turn left and go down the street? How do you park a reverse park? How do
you stop at an intersection? How do you know how fast to go somewhere? So we just listed half a
dozen rules. But you could probably go a fair few more. You might get into the thousands.
And so for a complex problem, such as driving, can you think of all the rules? Well, probably not.
So that's where machine learning and deep learning come in to help. And so this is a beautiful comment
I like to share with you on one of my YouTube videos is my 2020 machine learning roadmap.
And this is from Yashawing. I'm probably going to mispronounce this if I even try to.
But Yashawing says, I think you can use ML. So ML is machine learning. I'm going to use that
a lot throughout the course, by the way. ML is machine learning, just so you know.
For literally anything, as long as you can convert it into numbers, ah, that's what we said before,
machine learning is turning something into computer readable numbers. And then programming it to find
patterns, except with a machine learning algorithm, typically we write the algorithm and it finds
the patterns, not us. And so literally it could be anything, any input or output from the universe.
That's pretty darn cool about machine learning, right? But should you always use it just because
it could be used for anything? Well, I'd like to also introduce you to Google's number one rule
of machine learning. Now, if you can build a simple rule based system such as the step of five
rules that we had to map the ingredients to our Sicilian grandmothers roast chicken dish,
if you can write just five steps to do that, that's going to work every time, well, you should
probably do that. So if you can build a simple rule based system that doesn't require machine
learning, do that. And of course, maybe it's not so very simple, but maybe you can just write some
rules to solve the problem that you're working on. And this is from a wise software engineer,
which is, I kind of hinted at it before, rule one of Google's machine learning handbook. Now,
I'm going to highly recommend you read through that, but we're not going to go through that in
this video. So check that out. You can Google that otherwise the links will be where you get links.
So just keep that in mind, although machine learning is very powerful and very fun and very
excited, it doesn't mean that you should always use it. I know this is quite the thing to be saying
at the start of a deep learning machine learning course, but I just want you to keep in mind,

```
simple rule based systems are still good. Machine learning isn't a solve all for
everything.
Now, let's have a look at what deep learning is good for, but I'm going to leave
you on a
clip hammock because we're going to check this out in the next video. See you soon.
```

Section 3: Main Topics

Key Topics:

- In the last video, we familiarized ourselves with Google's number one rule of machine learning, which is basically if you don't need it, don't use it
- And with that in mind, what should we actually be looking to use machine learning or deep learning for
- But if you have a long, long list of rules, like the rules of driving a car, which could be hundreds, could be thousands, could be millions, who knows, that's where machine learning and deep learning may help

▶ 📄 Click to view detailed content

```
In the last video, we familiarized ourselves with Google's number one rule of
machine learning,
which is basically if you don't need it, don't use it. And with that in mind,
what should we actually be looking to use machine learning or deep learning for?
Well, problems with long lists of rules. So when the traditional approach fails to
remember the traditional approach is you have some sort of data input, you write a
list of rules for
that data to be manipulated in some way, shape, or form, and then you have the
outputs that you
know. But if you have a long, long list of rules, like the rules of driving a car,
which could be
hundreds, could be thousands, could be millions, who knows, that's where machine
learning and
deep learning may help. And it kind of is at the moment in the world of self-
driving cars,
machine learning and deep learning are the state of the art approach.
Continually changing environments. So whatever the benefits of deep learning is
that it can
keep learning if it needs to. And so it can adapt and learn to new scenarios. So if
you update the
data that your model was trained on, it can adjust to new different kinds of data
in the future.
So similarly to if you are driving a car, you might know your own neighborhood very
well.
But then when you go to somewhere you haven't been before, sure you can draw on the
foundations
of what you know, but you're going to have to adapt. How fast should you go? Where
```

should you
stop? Where should you park? These kinds of things. So with problems with long
lists of rules,
or continually changing environments, or if you had a large, large data set. And so
this is where
deep learning is flourishing in the world of technology. So let's give an example.
One of my
favorites is the food 101 data set, which you can search for online, which is
images of 101
different kinds of foods. Now we briefly looked at what a rule list might look like
for cooking
your grandmother's famous Sicilian roast chicken dish. But can you imagine if you
wanted to build
an app that could take photos of different food, how long your list of rules would
be to differentiate
101 different foods? It'd be so long. You need rule sets for every single one.
Let's just take
one food, for example. How do you write a program to tell what a banana looks like?
I mean you'd
have to code what a banana looks like, but not only a banana, what everything that
isn't a banana
looks like. So keep this in mind. What deep learning is good for? Problems with
long lists of rules,
continually changing environments, or discovering insights within large collections
of data.
Now, what deep learning is not good for? And I'm going to write typically here
because,
again, this is problem specific. Deep learning is quite powerful these days and
things might
change in the future. So keep an open mind, if there's anything about this course,
it's not for
me to tell you exactly what's what. It's for me to spark a curiosity into you to
figure out what's
what, or even better yet, what's not what. So when you need explainability, as
we'll see,
the patterns learned by a deep learning model, which is lots of numbers, called
weights and biases,
we'll have a look at that later on, are typically uninterpretable by a human. So
some of the times
deep learning models can have a million, 10 million, 100 million, a billion, some
models are getting
into the trillions of parameters. When I say parameters, I mean numbers or patterns
in data.
Remember, machine learning is turning things into numbers and then writing a
machine learning model
to find patterns in those numbers. So sometimes those patterns themselves can be
lists of numbers
that are in the millions. And so can you imagine looking at a list of numbers that
has a million
different things going on? That's going to be quite hard. I find it hard to
understand
three or four numbers, let alone a million. And when the traditional approach is a
better option,
again, this is Google's rule number one of machine learning. If you can do what you
need to do with
a simple rule based system, well, maybe you don't need to use machine learning or
deep learning.

```
Again, I'm going to use the deep learning machine learning terms interchangeably.
I'm not too concerned with definitions. You can form your own definitions, but just
so you know,
from my perspective, ML and deep learning are quite similar. When arrows are
unacceptable.
So since the outputs of a deep learning model aren't always predictable, we'll see
that deep
learning models are probabilistic. That means they're when they predict something,
they're making a
probabilistic bet on it. Whereas in a rule based system, you kind of know what the
outputs are
going to be every single time. So if you can't have errors based on probabilistic
errors,
well, then you probably shouldn't use deep learning and you'd like to go back to a
simple rule based
system. And then finally, when you don't have much data, so deep learning models
usually require a
fairly large amount of data to produce great results. However, there's a caveat
here, you know,
at the start, I said typically, we're going to see some techniques of how to get
great results
without huge amounts of data. And again, I wrote typically here because there are
techniques,
you can just research deep learning explainability. You're going to find a whole
bunch of stuff.
You can look up examples of when machine learning versus deep learning. And then
when arrows are
unacceptable, again, there are ways to make your model reproducible. So it predicts
you know what's
```

Section 4: Main Topics

Key Topics:

- Ah, we've got machine learning versus deep learning, and we're going to have a look at some different problem spaces in a second, and mainly breaking down in terms of what kind of data you have
- So in the last video, we covered a few things of what deep learning is good for and what deep learning is typically not good for
- So let's dive in to a little more of a comparison of machine learning versus deep learning

▶ 📄 Click to view detailed content

```
going to come out. So we do a lot of testing to verify this as well. And so what's
next? Ah,
we've got machine learning versus deep learning, and we're going to have a look at
```

some different
problem spaces in a second, and mainly breaking down in terms of what kind of data you have.
Not going to do this now prevent this video from getting too long. We'll cover all these
colorful beautiful pictures in the next video. Welcome back. So in the last video, we covered a
few things of what deep learning is good for and what deep learning is typically not good for.
So let's dive in to a little more of a comparison of machine learning versus deep learning. Again,
I'm going to be using these terms quite interchangeably. But there are some specific things that
typically you want traditional style of machine learning techniques versus deep learning. However,
this is constantly changing. So again, I'm not talking in absolutes here. I'm more just talking
in general. And I'll leave it to you to use your own curiosity to research the specific
differences between these two. But typically, for machine learning, like the traditional style of
algorithms, although they are still machine learning algorithms, which is kind of a little
bit confusing where deep learning and machine learning differ is you want to use traditional
machine learning algorithms on structured data. So if you have tables of numbers, this is what I
mean by structured rows and columns, structured data. And possibly one of the best algorithms
for this type of data is a gradient boosted machine, such as xg boost. This is an algorithm
that you'll see in a lot of data science competitions, and also used in production settings. When I
say production settings, I mean, applications that you may interact with on the internet,
or use on a day to day. So that's production. xg boost is typically the favorite algorithm for
these kinds of situations. So again, if you have structured data, you might look into xg boost
rather than building a deep learning algorithm. But again, the rules aren't set in stone. That's
where deep learning and machine learning is kind of an art kind of a science is that sometimes
xg boost is the best for structured data, but there might be exceptions to the rule. But for deep
learning, it is typically better for unstructured data. And what I mean by that is data that's kind
of all over the place. It's not in your nice, standardized rows and columns. So say you had
natural language such as this tweet by this person, whose name is quite similar to mine,
and has the same Twitter account as me. Oh, maybe I wrote that. How do I learn machine learning? What
you need to hear? Learn Python, learn math, start probability, software engineering, build.
What you need to do? Google it, go down the rabbit hole, resurfacing six to nine months,

and ring assess. I like that. Or if you had a whole bunch of texts such as the definition for
deep learning on Wikipedia, again, this is the reason why I'm not covering as many definitions
in this course is because look how simple you can look these things up. Wikipedia is going to
be able to define deep learning far better than what I can. I'm more focused on just getting involved
in working hands on with this stuff than defining what it is. And then we have
images. If we wanted to build a burger, take a photo app thing, you would work with image data,
which doesn't really have much of a structure. Although we'll see that there are ways for deep
learning that we can turn this kind of data to have some sort of structure through the beauty
of a tensor. And then we might have audio files such as if you were talking to your voice assistant.
I'm not going to say one because a whole bunch of my devices might go crazy if I say the name of
my voice assistant, which rhymes with I'm not even going to say that out loud. And so typically,
for unstructured data, you'll want to use a neural network of some kind. So structured data,
gradient boosted machine, or a random forest, or a tree based algorithm, such as extra boost,
and unstructured data, neural networks. So let's keep going. Let's have a look at some of the
common algorithms that you might use for structured data, machine learning versus unstructured data,
deep learning. So random forest is one of my favorites, gradient boosted models,
native base nearest neighbor, support vector machine, SVM, and then many more. But since
the advent of deep learning, these are often referred to as shallow algorithms. So deep learning,
why is it called deep learning? Well, as we'll see is that it can have many different layers
of algorithm, you might have an input layer, 100 layers in the middle, and then an output layer.
But we'll get hands on with this later on. And so common algorithms for deep learning and neural
networks, fully connected neural network, convolutional neural network, recurrent neural network,
transformers have taken over over the past couple years, and of course, many more. And the beautiful
thing about deep learning and neural networks is is almost as many problems that it can be applied
to is as many different ways that you can construct them. So this is why I'm putting all these
dot points on the page. And I can understand if you haven't had much experience of machine
learning or deep learning, this can be a whole bunch of information overload. But good news is
what we're going to be focused on building with PyTorch is neural networks, fully connected neural
networks and convolutional neural networks, the foundation of deep learning. But the excellent

Section 5: Main Topics

Key Topics:

- And again, part art, part science of machine learning and deep learning is depending on how you represent your problem, depending on what your problem is, many of the algorithms here and here can be used for both
- So I know I've just kind of bedazzled you and saying that, Oh, well, you kind of use these ones for deep learning, you kind of use these ones for machine learning
- So that's a little bit of confusion to machine learning

▶ 📄 Click to view detailed content

```
thing is, the exciting thing is, is that if we learn these foundational building
blocks,
we can get into these other styles of things here. And again, part art, part
science of machine
learning and deep learning is depending on how you represent your problem,
depending on what your
problem is, many of the algorithms here and here can be used for both. So I know
I've just kind of
bedazzled you and saying that, Oh, well, you kind of use these ones for deep
learning, you kind of
use these ones for machine learning. But depending on what your problem is, you can
also use both.
So that's a little bit of confusion to machine learning. But that's a fun part
about it too,
is use your curiosity to figure out what's best for whatever you're working on. And
with all this
talk about neural networks, how about in the next video, we cover what are neural
networks. Now,
I'd like you to Google this before we watch the next video, because it's going to
be hundreds of
definitions of what they are. And I'd like you to start forming your own definition
of what a
neural network is. I'll see you in the next video. Welcome back. In the last video,
I left you with
the cliffhanger of a question. What are neural networks? And I gave you the
challenge of
Googling that, but you might have already done that by the time you've got here.
Let's just do that together. If I type in what are neural networks, I've already
done this.
What are neural networks? Explain neural networks, neural network definition. There
are hundreds
of definitions of things like this online neural network in five minutes. Three
blue one brown.
I'd highly recommend that channel series on neural networks. That's going to be in
the
extracurricular stat quest is also amazing. So there's hundreds of different
```

definitions on here,
and you can read 10 of them, five of them, three of them, make your own definition.
But for the sake of this course, here's how I'm going to find neural networks.
So we have some data of whatever it is. We might have images of food. We might have tweets or natural language, and we might have speech. So these are some examples of inputs
for unstructured data, because they're not rows and columns. So these are the input data that
we have. And then how do we use them with a neural network? Well, before data can be used in a neural
network, it needs to be turned into numbers, because humans, we like looking at images of Raman and
spaghetti. We know that that's Raman. We know that that's spaghetti after we've seen it one or two
times. And we like reading good tweets, and we like listening to amazing music or hearing our
friend talk on the phone in audio file. However, before a computer understands what's going on
in these inputs, it needs to turn them into numbers. So this is what I call a numerical
encoding or a representation. And this numerical encoding, these square brackets indicate that
it's part of a matrix or a tensor, which we're going to get very hands on with throughout this
course. So we have our inputs, we've turned it into numbers, and then we pass it through a neural
network. And now this is a graphic for a neural network. However, the graphics for neural networks,
as we'll see, can get quite involved. But they all represent the same fundamentals. So if we go to
this one, for example, we have an input layer, then we have multiple hidden layers. However,
you define this, you can design these and how you want. Then we have an output layer. So our
inputs will go in some kind of data. The hidden layers will perform mathematical operations on the
input. So the numbers, and then we'll have an output. Oh, there's three blue one brown neural
networks from the ground up. Great video. Highly recommend you check that out. But then if we come
back to this, so we've got our inputs, we've turned it into numbers. And we've got our neural
networks that we put the input in. This is typically the input layer, hidden layer. This can be as
many different layers as you want, as many different, each of these little dots is called a node.
There's a lot of information here, but we're going to get hands-on with seeing what this looks
like. And then we have some kind of output. Now, which neural network should you use? Well,
you can choose the appropriate neural network for your problem, which could involve you
hand coding each one of these steps. Or you could find one that has worked on problems similar to
your own, such as for images, you might use a CNN, which is a convolutional neural network.
For natural language, you might use a transformer. For speech, you might also use a

```
transformer.
But fundamentally, they all follow the same principle of inputs, manipulation,
outputs.
And so the neural network will learn a representation on its own. We want to find
what it learns.
So it's going to manipulate these patterns in some way, shape, or form. And when I
say
learns representation, I'm going to also refer to it as learns patterns in the
data.
A lot of people refer to it as features. A feature may be the fact that the word do
comes out to how,
usually, in across a whole bunch of different languages. A feature can be almost
anything you
want. And again, we don't define this. The neural network learns these
representations,
patterns, features, also called weights on its own. And then where do we go from
there? Well,
we've got some sort of numbers, numerical encoding turned our data into numbers.
Our neural network
```

Section 6: Main Topics

Key Topics:

- We're going to see it in PyTorch code later on
- You can have, I put a s here because you can have one hidden layer, or the deep in deep learning comes from having lots of layers
- Now, from the next video, let's dive in briefly to different kinds of learning

▶ 📄 Click to view detailed content

```
has learned a representation that it thinks best represents the patterns in our
data.
And then it outputs those representation outputs, which we can use. And often
you'll
hear this referred to as features or weight matrix or weight tensor.
Learned representation is also another common one. There's a lot of different terms
for these
things. And then it will output. We can convert these outputs into human
understandable outputs.
So if we were to look at these, this could be, again, I said representations or
patterns that
are neural network learns can be millions of numbers. This is only nine. So imagine
if these
were millions of different numbers, I can barely understand the nine numbers that
is going on here.
So we need a way to convert these into human understandable terms. So for this
example,
```

we might have some input data, which are images of food. And then we want our neural network to
learn the representations between an image of ramen and an image of spaghetti.
And then eventually we'll take those patterns that it's learned and we'll convert them into
whether it thinks that this is an image of ramen or spaghetti. Or in the case of this tweet,
is this a tweet for a natural disaster or not a natural disaster? So our neural network has,
well, we've written code to turn this into numbers. Pass it through our neural network. Our neural
network has learned some kind of patterns. And then we ideally want it to represent this tweet
as not a disaster. And then we can write code to do each of these steps here. And the same thing
for these inputs going as speech, turning into something that you might say to your smart speaker,
which I'm not going to say because a whole bunch of my devices might go off. And so let's cover
the anatomy of neural networks. We've hinted at this a little bit already. But this is like
neural network anatomy 101. Again, this is highly customizable what this thing actually is. We're
going to see it in PyTorch code later on. But the data goes into the input layer. And in this case,
the number of units slash neurons slash nodes is two hidden layers. You can have, I put a s here
because you can have one hidden layer, or the deep in deep learning comes from having lots of
layers. So this is only showing four layers. You might have, well, this is three layers as well.
It might be very deep neural networks such as ResNet 152. This is 152 different layers.
So again, you can, or this is 34, because this is only ResNet 34. But ResNet 152 has 152 different
layers. So that's a common computer vision or a popular computer vision algorithm, by the way.
Lots of terms we're throwing out here. But with time, you'll start to become familiar with them.
So hidden layers can be almost as many as you want. We've only got pictured one here. And in this
case, there's three hidden units slash neurons. And then we have an output layer. So the outputs
learned representation or prediction probabilities from here, depending on how we set it up, which
again, we will see what these are later on. And in this case, it has one hidden unit. So two input,
three, one output, you can customize the number of these, you can customize how many layers there
are, you can customize what goes into here, you can customize what goes out of there. So now,
if we talk about the overall architecture, which is describing all of the layers combined. So that's,
when you hear neural network architecture, it talks about the input, the hidden layers,
which may be more than one, and the output layer. So that's a terminology for overall architecture.

```
Now, I say patterns is an arbitrary term. You can hear embedding embedding might
come from hidden
layers, weights, feature representation, feature vectors, all referring to similar
things. So,
again, how do we turn our data into some numerical form, build a neural network to
figure out patterns
to output some desired output that we want. And now to get more technical, each
layer is usually a
combination of linear, so straight lines, and nonlinear, non-straight functions. So
what I mean by that
is a linear function is a straight line, a nonlinear function is a non-straight
line.
If I asked you to draw whatever you want with unlimited straight lines and not
straight lines,
so you can use straight lines or curved lines, what kind of patterns could you
draw?
At a fundamental level, that is basically what a neural network is doing. It's
using a combination
of linear, straight lines, and not straight lines to draw patterns in our data.
We'll see what
this looks like later on. Now, from the next video, let's dive in briefly to
different kinds of
learning. So we've looked at what a neural network is, the overall algorithm, but
there are also
different paradigms of how a neural network learns. I'll see you in the next video.
Welcome back. We've discussed a brief overview of an anatomy of what a neural
network is,
but let's now discuss some learning paradigms. So the first one is supervised
learning,
and then we have unsupervised and self-supervised learning, and transfer learning.
Now supervised
learning is when you have data and labels, such as in the example we gave at the
start, which was
how you would build a neural network or a machine learning algorithm to figure out
the rules to
cook your Sicilian grandmother's famous roast chicken dish. So in the case of
supervised learning,
you'd have a lot of data, so inputs, such as raw ingredients as vegetables and
chicken,
and a lot of examples of what that inputs should ideally look like. Or in the case
of discerning
```

Section 7: Main Topics

Key Topics:

- photos between a cat and a dog, you might have a thousand photos of a cat and a thousand photos of a dog that you know which photos are cat and which photos are dog, and you pass those photos to a machine learning algorithm to discern
- So that's supervised learning, data and labels

- Unsupervised and self-supervised learning is you just have the data itself

▶ 📄 Click to view detailed content

```
photos between a cat and a dog, you might have a thousand photos of a cat and a
thousand photos
of a dog that you know which photos are cat and which photos are dog, and you pass
those photos
to a machine learning algorithm to discern. So in that case, you have data, the
photos, and the
labels, aka cat and dog, for each of those photos. So that's supervised learning,
data and labels.
Unsupervised and self-supervised learning is you just have the data itself.
You don't have any labels. So in the case of cat and dog photos, you only have the
photos.
You don't have the labels of cat and dog. So in the case of self-supervised
learning,
you could get a machine learning algorithm to learn an inherent representation of
what,
and when I say representation, I mean patterns and numbers, I mean weights, I mean
features,
a whole bunch of different names describing the same thing. You could get a self-
supervised
learning algorithm to figure out the fundamental patterns between a dog and a cat
image, but
it wouldn't necessarily know the difference between the two.
That's where you could come in later and go show me the patterns you've learned,
and it might show you the patterns and you could go, okay, the patterns that look
like this,
a dog and the patterns that look like that, a cat. So self-supervised and
unsupervised learning
learn solely on the data itself. And then finally, transfer learning is a very,
very
important paradigm in deep learning. It's taking the patterns that one model has
learned
of a data set and transferring it to another model, such in the case of if we were
trying to
build a supervised learning algorithm for discerning between cat and dog photos.
We might start with a model that has already learned patterns and images
and transfer those foundational patterns to our own model so that our model gets a
head start.
This is transfer learning is a very, very powerful technique, but as for this
course,
we're going to be writing code to focus on these two supervised learning and
transfer learning,
which are two of the most common paradigms or common types of learning in machine
learning
and deep learning. However, this style of code though can be adapted across
different learning
paradigms. Now, I just want to let you know there is one that I haven't mentioned
here,
which is kind of in its own bucket, and that is reinforcement learning. So I'll
leave this
as an extension if you wanted to look it up. But essentially, this is a good one.
That's a good photo, actually. So shout out to Katie Nuggets. The whole idea of
```

reinforcement
learning is that you have some kind of environment and an agent that does actions in that environment,
and you give rewards and observations back to that agent. So say, for example,
you wanted to teach your dog to urinate outside. Well, you would reward its actions of urinating
outside and possibly not reward its actions of urinating all over your couch. So reinforcement
learning is again, it's kind of in its own paradigm. This picture has a good explanation
between unsupervised learning, supervised learning to separate two different things,
and then reinforcement learning is kind of like that. But again, I will let you research the
different learning paradigms a little bit more in your own time. As I said, we're going to be
focused on writing code to do supervised learning and transfer learning,
specifically pytorch code.
Now with that covered, let's get a few examples of what is deep learning actually used for. And
before we get into the next video, I'm going to issue you a challenge to search this question
yourself and come up with some of your own ideas for what deep learning is currently used for.
So give that a shot and I'll see you in the next video. How'd you go? Did you do some research?
Did you find out what deep learning is actually used for? I bet you found a treasure trail of
things. And hey, I mean, if you're reading this course, chances are that you probably already
know some use cases for deep learning. You're like, Daniel, hurry up and get to the code. Well,
we're going to get there, don't you worry? But let's have a look at some things that deep
learning can be used for. But before, I just want to remind you of this comment. This is from
Yasha Sway on the 2020 machine learning roadmap video. I think you can use ML and remember,
ML is machine learning. And remember, deep learning is a part of ML for literally anything as long
as you can convert it into numbers and program it to find patterns. Literally, it could be anything,
any input or output from the universe. So that's a beautiful thing about machine learning is that
if you can encode it something into numbers, chances are you can build a machine learning
algorithm to find patterns in those numbers. Will it work? Well, again, that's the reason machine
learning and deep learning is part art, part science. A scientist would love to know that their
experiments would work. But an artist is kind of excited about the fact that, I don't know,
this might work, it might not. And so that's something to keep in mind. Along with the rule
number one of machine learning is if you don't need it, you don't use it. But if you do use it,
it can be used for almost anything. And let's get a little bit specific and find

```
out some deep
learning use cases. And I've put some up there for a reason because there are lots.
These are just
some that I interact with in my day to day life, such as recommendation, we've got
a programming
```

Section 8: Main Topics

Key Topics:

- That's all powered by deep learning
- Well, that's powered by deep learning as well
- But if I wanted to translate deep learning as epic to Spanish, it might come out as el aprendise, profando es ebiko

▶ 📄 Click to view detailed content

```
video, we've got a programming podcast, we got some jujitsu videos, we've got some
RuneScape
videos, a soundtrack from my favorite movie. Have you noticed, whenever you go to
YouTube,
you don't really search for things anymore. Well, sometimes you might, but the
recommendation
page is pretty darn good. That's all powered by deep learning. And in the last 10
years,
have you noticed that translation has got pretty good too? Well, that's powered by
deep learning
as well. Now, I don't have much hands on experience with this. I did use it when I
was in Japan.
I speak a very little amount of Japanese and even smaller amount of Mandarin. But
if I wanted to
translate deep learning as epic to Spanish, it might come out as el aprendise,
profando es ebiko.
Now, all of the native Spanish speakers watching this video can laugh at me because
that was a very
Australian version of saying deep learning is epic in Spanish. But that's so cool.
All the Google
Translate is now powered by deep learning. And the beautiful thing, if I couldn't
say it myself,
I could click this speaker and it would say it for me. So that speech recognition
that's powered
by deep learning. So if you were to ask your voice assistant who's the biggest big
dog of them all,
of course, they're going to say you, which is what I've set up, my voice assistant
to say.
That's part of speech recognition. And in computer vision, oh, look at this. You
see this? Where is
this photo from? This photo is from this person driving this car. Did a hit and run
```

on my car,
at the front of my house, my apartment building, my car was parked on the street, this car, the
trailer came off, ran into the back of my car, basically destroyed it, and then they drove off.
However, my next door neighbors security camera picked up on this car. Now, I became a detective
for a week, and I thought, hmm, if there was a computer vision algorithm built into that camera,
it could have detected when the car hit. I mean, it took a lot of searching to find it,
it turns out the car hit about 3.30am in the morning. So it's pitch black. And of course,
we didn't get the license plate. So this person is out there somewhere in the world after doing
a hit and run. So if you're watching this video, just remember computer vision might catch you one
day. So this is called object detection, where you would place a box around the area where the
pixels most represent the object that you're looking for. So for computer vision, we could
train an object detector to capture cars that drive past a certain camera. And then if someone
does a hit and run on you, you could capture it. And then fingers crossed, it's not too dark
that you can read the license plate and go, hey, excuse me, please, this person has hit my car
and wrecked it. So that's a very close to home story of where computer vision could be used.
And then finally, natural language processing. Have you noticed as well, your spam detector on
your email inbox is pretty darn good? Well, some are powered by deep learning, some not,
it's hard to tell these days what is powered by deep learning, what isn't. But natural language
processing is the process of looking at natural language text. So unstructured text. So whatever
you'd write an email in a story in a Wikipedia document and deciding or getting your algorithm
to find patterns in that. So for this example, you would find that this email is not spam.
This deep learning course is incredible. I can't wait to use what I've learned. Thank you so much.
And by the way, that is my real email. So if you want to email me, you can. And then this is spam.
Hey, Daniel, congratulations, you win a lot of money. Wow, I really like that a lot of money.
But somebody said, I don't think that this is real. So that would probably go to my spam inbox.
Now, with that being said, if we wanted to put these problems in a little bit more of a
classification, this is known as sequence to sequence because you put one sequence in
and get one sequence out. Same as this, you have a sequence of audio waves and you get some
text out. So sequence to sequence, sec to sec. This is classification slash regression. In this

```
case, the regression is predicting a number. That's what a regression problem is.
You would predict
the coordinates of where these box corners should be. So say this should be at
however many pixels
in from the X angle and however many pixels down from the Y angle, that's that
corner.
And then you would draw in between the corners. And then the classification part
would go,
Hey, this is that car that did a hit and run on us. And in this case, this is
classification.
Classification is predicting whether something is one thing or another, or perhaps
more than one
thing or another in the class of multi class classification. So this email is not
spam. That's
a class and this email is spam. So that's also a class. So I think we've only got
one direction
to go now that we've sort of laid the foundation for the course. And that is
Well, let's start talking about PyTorch. I'll see you in the next video.
Well, let's now cover some of the foundations of
PyTorch. But first, you might be asking, what is PyTorch? Well, of course, we could
just go to
our friend, the internet, and look up PyTorch.org. This is the homepage for
PyTorch.
```

Section 9: Main Topics

Key Topics:

- This course is not a replacement for everything on this homepage
- This should be your ground truth for everything PyTorch
- PyTorch

▶ 📄 Click to view detailed content

```
This course is not a replacement for everything on this homepage. This should be
your ground truth
for everything PyTorch. So you can get started. You've got a big ecosystem. You've
got a way to
set up on your local computer. You've got resources. You've got docs. PyTorch.
You've got the GitHub.
You've got search. You've got blog, everything here. This website should be the
place you're
visiting most throughout this course as we're writing PyTorch code. You're coming
here.
You're reading about it. You're checking things out. You're looking at examples.
But for the sake of this course, let's break PyTorch down. Oh, there's a little
flame animation
I just forgot about. What is PyTorch? I didn't sync up the animations. That's all
right. So
```

PyTorch is the most popular research deep learning framework. I'll get to that in a second.
It allows you to write fast deep learning code in Python. If you know Python, it's a very user-friendly
programming language. PyTorch allows us to write state-of-the-art deep learning code
accelerated by GPUs with Python. It enables you access to many pre-built deep learning models
from Torch Hub, which is a website that has lots of, if you remember, I said transfer learning is
a way that we can use other deep learning models to power our own. Torch Hub is a resource for that.
Same as Torch Vision.Models. We'll be looking at this throughout the course.
It provides an ecosystem for the whole stack of machine learning. From pre-processing data,
getting your data into tenses, what if you started with some images? How do you represent them as
numbers? Then you can build models such as neural networks to model that data. Then you can even
deploy your model in your application slash cloud, well, deploy your PyTorch model. Application slash
cloud will be depending on what sort of application slash cloud that you're using, but generally it
will run some kind of PyTorch model. And it was originally designed and used in-house by Facebook
slash meta. I'm pretty sure Facebook have renamed themselves meta now, but it is now open source
and used by companies such as Tesla, Microsoft and OpenAI. And when I say it is the most popular
deep learning research framework, don't take my word for it. Let's have a look at papers with code
dot com slash trends. If you're not sure what papers with code is, it is a website that tracks
the latest and greatest machine learning papers and whether or not they have code. So we have some
other languages here, other deep learning frameworks, PyTorch, TensorFlow, Jax is another one, MXNet,
paddle paddle, the original torch. So PyTorch is an evolution of torch written in Python,
CAF2, Mindspore. But if we look at this, when is this? Last date is December 2021. We have,
oh, this is going to move every time I move it. No. So I'll highlight PyTorch at 58% there.
So by far and large, the most popular research machine learning framework used to write the code
for state of the art machine learning algorithms. So this is browse state of the art papers with
code.com amazing website. We have semantic segmentation, image classification, object detection, image
generation, computer vision, natural language processing, medical, I'll let you explore this.
It's one of my favorite resources for staying up to date on the field. But as you see, out of the
65,000 papers with code that this website is tracked, 58% of them are implemented with PyTorch.
How cool is that? And this is what we're learning. So let's jump into there. Why PyTorch? Well,

```
other than the reasons that we just spoke about, it's a research favorite. This is
highlighting.
There we go. So there we go. I've highlighted it here. PyTorch, 58%, nearly 2,500
repos. If
you're not sure what a repo is, a repo is a place where you store all of your code
online.
And generally, if a paper gets published in machine learning, if it's fantastic
research,
it will come with code, code that you can access and use for your own applications
or your own
research. Again, why PyTorch? Well, this is a tweet from Francois Chale, who's the
author of
Keras, which is another popular deep learning framework. But with tools like Colab,
we're going
to see what Colab is in a second, Keras and TensorFlow. I've added in here and
PyTorch.
Virtually anyone can solve in a day with no initial investment problems that would
have
required an engineering team working for a quarter and $20,000 in hardware in 2014.
So this is just
to highlight how good the space of deep learning and machine learning tooling has
become. Colab,
Keras and TensorFlow are all fantastic. And now PyTorch is added to this list. If
you want to
check that out, there's Francois Chale on Twitter. Very, very prominent voice in
the machine learning
field. Why PyTorch? If you want some more reasons, well, have a look at this. Look
at all the
places that are using PyTorch. It's just coming up everywhere. We've got Andre
Kapathi here,
who's the director of AI at Tesla. So if we go, we could search this, PyTorch
at Tesla. We've got a YouTube talk there, Andre Kapathi, director of AI at Tesla.
And so Tesla are using PyTorch for the computer vision models of autopilot. So if
we go to videos
or maybe images, does it come up there? Things like this, a car detecting what's
going on in the scene.
Of course, there'll be some other code for planning, but I'll let you research
that.
When we come back here, OpenAI, which is one of the biggest open artificial
intelligence
research firms, open in the sense that they publish a lot of their research
methodologies,
however, recently there's been some debate about that. But if you go to openai.com,
```

Section 10: Main Topics

Key Topics:

- let's just say that they're one of the biggest AI research entities in the world, and they've standardized on PyTorch

- Presumably with PyTorch, because this blog post from January 2020 says that OpenAI is now standardized across PyTorch
- There's a repo called the incredible PyTorch, which collects a whole bunch of different projects that are built on top of PyTorch

▶ 📄 Click to view detailed content

```
let's just say that they're one of the biggest AI research entities in the world,
and they've standardized on PyTorch. So they've got a great blog, they've got great
research,
and now they've got OpenAI API, which is, you can use their API to access some of
the models
that they've trained. Presumably with PyTorch, because this blog post from January
2020 says
that OpenAI is now standardized across PyTorch. There's a repo called the
incredible PyTorch,
which collects a whole bunch of different projects that are built on top of
PyTorch.
That's the beauty of PyTorch is that you can build on top of it, you can build with
it
AI for AG, for agriculture. PyTorch has been used. Let's have a look. PyTorch in
agriculture.
There we go. Agricultural robots use PyTorch. This is a medium article.
It's everywhere. So if we go down here, this is using object detection. Beautiful.
Object detection to detect what kind of weeds should be sprayed with fertilizer.
This is just
one of many different things, so PyTorch on a big tractor like this. It can be used
almost
anywhere. If we come back, PyTorch builds the future of AI and machine learning at
Facebook,
so Facebook, which is also MetaAI, a little bit confusing, even though it says
MetaAI,
it's on AI.facebook.com. That may change by the time you watch this. They use
PyTorch in-house
for all of their machine learning applications. Microsoft is huge in the PyTorch
game.
It's absolutely everywhere. So if that's not enough reason to use PyTorch,
well, then maybe you're in the wrong course. So you've seen enough reasons of why
to use PyTorch.
I'm going to give you one more. That is that it helps you run your code, your
machine learning code
accelerated on a GPU. We've covered this briefly, but what is a GPU slash a TPU,
because this is more of a newer chip these days. A GPU is a graphics processing
unit,
which is essentially very fast at crunching numbers. Originally designed for video
games,
if you've ever designed or played a video game, you know that the graphics are
quite intense,
especially these days. And so to render those graphics, you need to do a lot of
numerical calculations.
And so the beautiful thing about PyTorch is that it enables you to leverage a GPU
through an
interface called CUDA, which is a lot of words I'm going to throw out you here, a
lot of acronyms
```

in the deep learning space, CUDA. Let's just search CUDA. CUDA toolkit. So CUDA is a parallel
computing platform and application programming interface, which is an API that allows software
to use certain types of graphics processing units for general purpose computing. That's what
we want. So PyTorch leverages CUDA to enable you to run your machine learning code on NVIDIA
GPUs. Now, there is also an ability to run your PyTorch code on TPUs, which is a tensor processing
unit. However, GPUs are far more popular when running various types of PyTorch code. So we're
going to focus on running our PyTorch code on the GPU. And to just give you a quick example,
PyTorch on TPU, let's see that. Getting started with PyTorch on cloud TPUs, there's plenty of
guys for that. But as I said, GPUs are going to be far more common in practice. So that's what
we're going to focus on. And with that said, we've said tensor processing unit. Now, the reason
why these are called tensor processing units is because machine learning and deep learning
deals a lot with tensors. And so in the next video, let's answer the question, what is a tensor?
But before I go through and answer that from my perspective, I'd like you to research this
question. So open up Google or your favorite search engine and type in what is a tensor and
see what you find. I'll see you in the next video. Welcome back. In the last video, I left you on
the cliffhanger question of what is a tensor? And I also issued you the challenge to research
what is a tensor. Because as I said, this course isn't all about telling you exactly what things
are. It's more so sparking a curiosity in you so that you can stumble upon the answers to these
things yourself. But let's have a look. What is a tensor? Now, if you remember this graphic,
there's a lot going on here. But this is our neural network. We have some kind of input,
some kind of numerical encoding. Now, we start with this data. In our case, it's unstructured data
because we have some images here, some text here, and some audio file here. Now, these necessarily
don't go in all at the same time. This image could just focus on a neural network specifically
for images. This text could focus on a neural network specifically for text. And this sound bite
or speech could focus on a neural network specifically for speech. However, the field is sort of also
moving towards building neural networks that are capable of handling all three types of inputs.
For now, we're going to start small and then build up the algorithms that we're going to focus on
are neural networks that focus on one type of data. But the premise is still the same. You have
some kind of input. You have to numerically encode it in some form, pass it to a

```
neural network
to learn representations or patterns within that numerical encoding, output some
form of
representation. And then we can convert that representation into things that humans
understand.
And you might have already seen these, and I might have already referenced the fact
that
these are tensors. So when the question comes up, what are tensors? A tensor could
be almost
anything. It could be almost any representation of numbers. We're going to get very
hands on with
```

Section 11: Main Topics

Key Topics:

- And that's actually the fundamental building block of PyTorch aside from neural network components is the torch dot tensor
- Now, a lot of those mathematical operations are taken care of by PyTorch behind the scenes
- What is machine learning

▶ 📄 Click to view detailed content

```
tensors. And that's actually the fundamental building block of PyTorch aside from
neural network
components is the torch dot tensor. We're going to see that very shortly. But this
is a very
important takeaway is that you have some sort of input data. You're going to
numerically encode
that data, turn it into a tensor of some kind. Whatever that kind is will depend on
the problem
you're working with. Then you're going to pass it to a neural network, which will
perform mathematical
operations on that tensor. Now, a lot of those mathematical operations are taken
care of by
PyTorch behind the scenes. So we'll be writing code to execute some kind of
mathematical
operations on these tensors. And then the neural network that we create, or the one
that's already
been created, but we just use for our problem, we'll output another tensor, similar
to the input,
but has been manipulated in a certain way that we've sort of programmed it to. And
then we can take
this output tensor and change it into something that a human can understand. So to
remove a lot
of the text around it, make it a little bit more clearer. If we were focusing on
building an image
```

classification model, so we want to classify whether this was a photo of Raman or spaghetti,
we would have images as input. We would turn those images into numbers, which are represented
by a tensor. We would pass that tensor of numbers to a neural network, or there might be lots of
tensors here. We might have 10,000 images. We might have a million images. Or in some cases,
if you're Google or Facebook, you might be working with 300 million or a billion images at a time.
The principle still stands that you encode your data in some form of numerical representation,
which is a tensor, pass that tensor, or lots of tensors to a neural network. The neural network
performs mathematical operations on those tensors, outputs a tensor, we convert that tensor into
something that we can understand as humans. And so with that being said, we've covered a lot of
the fundamentals. What is machine learning? What is deep learning? What is neural network? Well,
we've touched the surface of these things. You can get as deep as you like. We've covered
why use PyTorch. What is PyTorch? Now, the fundamental building block of deep learning
is tensors. We've covered that. Let's get a bit more specific in the next video
of what we're going to cover code-wise in this first module. I'm so excited we're going to start
codes in. I'll see you in the next video. Now it's time to get specific about what we're going to
cover code-wise in this fundamentals module. But I just want to reiterate the fact that
going back to the last video where I challenge you to look up what is a tensor, here's exactly
what I would do. I would come to Google. I would type in the question, what is a tensor? There we go.
What is a tensor in PyTorch? It knows Google knows that using that deep learning data that we want
to know what a tensor is in PyTorch. But a tensor is a very general thing. It's not
associated with just PyTorch. Now we've got tensor on Wikipedia. We've got tensor. This is probably
my favorite video on what is a tensor. By Dan Flesch. Flesch, I'm probably saying that wrong,
but good first name. This is going to be your extra curriculum for this video and the previous
video is to watch this on what is a tensor. Now you might be saying, well, what gives? I've come to
this course to learn PyTorch and all this guy's doing, all you're doing, Daniel, is just Googling
things when a question comes up. Why don't you just tell me what it is? Well, if I was to tell you
everything about deep learning and machine learning and PyTorch and what it is and what it's not,
that course would be far too long. I'm doing this on purpose. I'm searching questions like this on
purpose because that's exactly what I do day to day as a machine learning engineer. I write code
like we're about to do. And then if I don't know something, I literally go to

```
whatever search engine
I'm using, Google most of the time, and type in whatever error I'm getting or
PyTorch, what is
a tensor, something like that. So I want to not only tell you that it's okay to
search questions
like that, but it's encouraged. So just keep that in mind as we go through the
whole course,
you're going to see me do it a lot. Let's get into what we're going to cover. Here
we go.
Now, this tweet is from Elon Musk. And so I've decided, you know what, let's base
the whole
course on this tweet. We have learning MLDL from university, you have a little bit
of a small brain.
Online courses, well, like this one, that brain's starting to explode and you get
some little fireworks
from YouTube. Oh, you're watching this on YouTube. Look at that shiny brain from
articles. My goodness.
Lucky that this course comes in article format. If you go to learn pytorch.io, all
of the course
materials are in online book format. So we're going to get into this fundamental
section very
shortly. But if you want a reference, the course materials are built off this book.
And by the
time you watch this, there's going to be more chapters here. So we're covering all
the bases
here. And then finally, from memes, you would ascend to some godlike creature. I
think that's
hovering underwater. So that is the best way to learn machine learning. So this is
what we're
going to start with MLDL from university online courses, YouTube from articles from
memes. No,
```

Section 12: Main Topics

Key Topics:

- So now in this module, we are going to cover the pytorch basics and fundamentals, mainly dealing with tensors and tensor operations
- Then we're going to look at building and using pre-trained deep learning models, specifically neural networks
- We're going to see how we can make predictions with our model, because that's what deep learning and machine learning is all about, right, using patterns from the past to predict the future

▶ 📄 Click to view detailed content

no, no, no. But kind of here is what we're going to cover broadly. So now in this module,
we are going to cover the pytorch basics and fundamentals, mainly dealing with tensors and
tensor operations. Remember, a neural network is all about input tensors, performing operations on
those tensors, creating output operations. Later, we're going to be focused on pre-processing data,
getting it into tensors, so turning data from raw form, images, whatever, into a numerical
encoding, which is a tensor. Then we're going to look at building and using pre-trained deep
learning models, specifically neural networks. We're going to fit a model to the data. So we're
going to show our model or write code for our model to learn patterns in the data that we've
pre-processed. We're going to see how we can make predictions with our model, because that's
what deep learning and machine learning is all about, right, using patterns from the past to
predict the future. And then we're going to evaluate our model's predictions. We're going to learn
how to save and load our models. For example, if you wanted to export your model from where we're
working to an application or something like that. And then finally, we're going to see how we can
use a trained model to make predictions on our own data on custom data, which is very fun. And
how? Well, you can see that the scientist has faded out a little bit, but that's not really that true.
We're going to do it like cooks, not chemists. So chemists are quite precise. Everything has to be
exactly how it is. But cooks are more like, oh, you know what, a little bit of salt, a little bit of
butter. Does it taste good? Okay, well, then we're on. But machine learning is a little bit of both.
It's a little bit of science, a little bit of art. That's how we're going to do it. But
I like the idea of this being a machine learning cooking show. So welcome to cooking with machine
learning, cooking with PyTorch with Daniel. And finally, we've got a workflow here, which we have
a PyTorch workflow, which is one of many. We're going to kind of use this throughout the entire
course is step one, we're going to get our data ready. Step two, we're going to build a
pick a pre trained model to suit whatever problem we're working on. Step two point one,
pick a loss function and optimizer. Don't worry about what they are. We're going to cover them
soon. Step two point two, build a training loop. Now this is kind of all part of the parcel of
step two, hence why we've got two point one and two point two. You'll see what that means later on.
Number three, we're going to fit the model to the data and make a prediction. So say we're working

on image classification for Raman or spaghetti. How do we build a neural network or put our
images through that neural network to get some sort of idea of what's in an image? We'll see
how to do that. Well, the value weight our model to see if it's predicting BS or it's actually
going all right. Number five, we're going to improve through experimentation. That's another
big thing that you'll notice throughout machine learning throughout this course is that it's
very experimental part art, part science. Number six, save and reload your trained model. Again,
I put these with numerical order, but they can kind of be mixed and matched depending on where
you are in the journey. But numerical order is just easy to understand for now. Now we've got
one more video, maybe another one before we get into code. But in the next video, I'm going to
cover some very, very important points on how you should approach this course. I'll see you there.
Now you might be asking, how should I approach this course? You might not be asking, but we're
going to answer it anyway. How to approach this course? This is how I would recommend approaching
this course. So I'm a machine learning engineer day to day and learning machine learning to
coding machine learning, a kind of two different things. I remember when I first learned it was
kind of, you learned a lot of theory rather than writing code. So not to take away from the theory
of being important, this course is going to be focusing on writing machine learning specifically
PyTorch code. So the number one step to approaching this course is to code along. Now because this
course is focused on purely writing code, I will be linking extracurricular resources for you to
learn more about what's going on behind the scenes of the code. My idea of teaching is that if we
can code together, write some code, see how it's working, that's going to spark your curiosity to
figure out what's going on behind the scenes. So motto number one is if and out, run the code,
write it, run the code, see what happens. Number two, I love that. Explore an experiment again.
Approach this with the idea, the mind of a scientist and a chef or science and art. Experiment,
experiment, experiment. Try things with rigor like a scientist would, and then just try things
for the fun of it like a chef would. Number three, visualize what you don't understand. I can't
emphasize this one enough. We have three models so far. If and out, run the code, you're going to
hear me say this a lot. Experiment, experiment, experiment. And number three, visualize, visualize,
visualize. Why is this? Well, because we've spoken about machine learning and deep learning

```
deals with a lot of data, a lot of numbers. And so I find it that if I visualize
some numbers in
```

Section 13: Main Topics

Key Topics:

- Everyone is just on a different part of their learning journey
- If we go, have we got the book version of the course up here
- I'm not going to jump into them, but I would highly recommend don't just follow along with the course and code after I code

▶ 📄 Click to view detailed content

```
whatever form that isn't just numbers all over a page, I tend to understand it
better.
And there are some great extracurricular resources that I'm going to link that also
turn what we're
doing. So writing code into fantastic visualizations. Number four, ask questions,
including the dumb
questions. Really, there's no such thing as a dumb question. Everyone is just on a
different
part of their learning journey. And in fact, if you do have a quote unquote dumb
question,
it turns out that a lot of people probably have that one as well. So be sure to ask
questions.
I'm going to link a resource in a minute of where you can ask those questions, but
please, please, please ask questions, not only to the community, but to Google to
the internet
to wherever you can, or just yourself. Ask questions of the code and write code to
figure
out the answer to those questions. Number five, do the exercises. There are some
great exercises that I've created for each of the modules. If we go, have we got
the book version
of the course up here? We do. Within all of these chapters here, down the bottom is
going to be
exercises and extra curriculum. So we've got some exercises. I'm not going to jump
into them,
but I would highly recommend don't just follow along with the course and code after
I code.
Please, please, please give the exercises a go because that's going to stretch your
knowledge.
We're going to have a lot of practice writing code together, doing all of this
stuff here.
But then the exercises are going to give you a chance to practice what you've
learned.
And then of course, extra curriculum. Well, hey, if you want to learn more, there's
plenty of
```

opportunities to do so there. And then finally, number six, share your work. I can't emphasize
enough how much writing about learning deep learning or sharing my work through GitHub or
different code resources or with the community has helped with my learning. So if you learn
something cool about PyTorch, I'd love to see it. Link it to me somehow in the Discord chat
or on GitHub or whatever. There'll be links of where you can find me. I'd love to see it. Please
do share your work. It's a great way to not only learn something because when you share it, when
you write about it, it's like, how would someone else understand it? But it's also a great way to
help others learn too. And so we said how to approach this course. Now, let's go how not to
approach this course. I would love for you to avoid overthinking the process. And this is your brain,
and this is your brain on fire. So avoid having your brain on fire. That's not a good place to be.
We are working with PyTorch, so it's going to be quite hot. Just playing on words with the name
torch. But avoid your brain catching on fire. And avoid saying, I can't learn,
I've said this to myself lots of times, and then I've practiced it and it turns out I can
actually learn those things. So let's just draw a red line on there. Oh, I think a red line.
Yeah, there we go. Nice and thick red line. We'll get that out there. It doesn't really make sense
now that this says avoid and crossed out. But don't say I can't learn and prevent your brain from
catching on fire. Finally, we've got one more video that I'm going to cover before this one
gets too long of the resources for the course before we get into coding. I'll see you there.
Now, there are some fundamental resources that I would like you to be aware of before we
go any further in this course. These are going to be paramount to what we're working with.
So for this course, there are three things. There is the GitHub repo. So if we click this link,
I've got a pinned on my browser. So you might want to do the same while you're going through
the course. But this is Mr. D. Burks in my GitHub slash PyTorch deep learning. It is still a work
in progress at the time of recording this video. But by the time you go through it, it won't look
too much different, but there just be more materials. You'll have materials outline,
section, what does it cover? As you can see, some more are coming soon at the time of recording
this. So these will probably be done by the time you watch this exercise in extra curriculum.
There'll be links here. Basically, everything you need for the course will be in the GitHub repo.
And then if we come back, also on the GitHub repo, the same repo. So Mr. D. Burks slash PyTorch

```
deep learning. If you click on discussions, this is going to be the Q and A. This
is just the same
link here, the Q and A for the course. So if you have a question here, you can
click new discussion,
you can go Q and A, and then type in video, and then the title PyTorch
Fundamentals, and then go
in here. Or you could type in your error as well. What is N-DIM for a tensor? And
then in here,
you can type in some stuff here. Hello. I'm having trouble on video X, Y, Z. Put in
the name of the
video. So that way I can, or someone else can help you out. And then code, you can
go three
back ticks, write Python, and then you can go import torch, torch dot rand n, which
is going to
create a tensor. We're going to see this in a second. Yeah, yeah, yeah. And then if
you post that
```

Section 14: Main Topics

Key Topics:

- You could even include the error message, and then you can just click start discussion, and then someone, either myself or someone else from the course will be able to help out there
- There's nothing here yet because the course isn't out yet, but as you go through it, there will probably be more and more stuff here
- I've just got some issues here already about the fact that I need to record videos for the course

▶ 📄 Click to view detailed content

```
question, the formatting of the code is very helpful that we can understand what's
going on,
and what's going on here. So this is basically the outline of how I would ask a
question video.
This is going on. What is such and such for whatever's going on? Hello. This is
what I'm having
trouble with. Here's the code, and here's what's happening. You could even include
the error message,
and then you can just click start discussion, and then someone, either myself or
someone else from
the course will be able to help out there. And the beautiful thing about this is
that it's all in
one place. You can start to search it. There's nothing here yet because the course
isn't out yet,
but as you go through it, there will probably be more and more stuff here. Then if
you have any
```

issues with the code that you think needs fixed, you can also open a new issue there. I'll let you
read more into what's going on. I've just got some issues here already about the fact that I
need to record videos for the course. I need to create some stuff. But if you think there's
something that could be improved, make an issue. If you have a question about the course,
ask a discussion. And then if we come back to the keynote, we have one more resource. So that
was the course materials all live in the GitHub. The course Q&A is on the course
GitHub's discussions tab, and then the course online book. Now, this is a work of art.
This is quite beautiful. It is some code to automatically turn all of the materials from the
GitHub. So if we come into here code, if we click on notebook zero zero, this is going to sometimes
if you've ever worked with Jupiter notebooks on GitHub, they can take a while to load.
So all of the materials here automatically get converted into this book. So the beautiful
thing about the book is that it's got different headings here. It's all readable. It's all online.
It's going to have all the images there. And you can also search some stuff here,
PyTorch training steps, creating a training loop in PyTorch. Beautiful. We're going to see
this later on. So they're the three big materials that you need to be aware of, the three big resources
for this specific course materials on GitHub course Q&A course online book, which is
learn pytorch.io, simple URL to remember, all the materials will be there. And then
specifically for PyTorch or things PyTorch, the PyTorch website and the PyTorch forums.
So if you have a question that's not course related, but more PyTorch related, I'd highly
recommend you go to the PyTorch forums, which is available at discuss.pytorch.org. We've got a link
there. Then the PyTorch website, PyTorch.org, this is going to be your home ground for everything
PyTorch of course. We have the documentation here. And as I said, this course is not a replacement
for getting familiar with the PyTorch documentation. This, the course actually is built off all of
the PyTorch documentation. It's just organized in a slightly different way. So there's plenty of
amazing resources here on everything to do with PyTorch. This is your home ground. And you're
going to see me referring to this a lot throughout the course. So just keep these in mind, course
materials on GitHub, course discussions, learnpytorch.io. This is all for the course. And all things
PyTorch specific, so not necessarily this course, but just PyTorch in general, the PyTorch website
and the PyTorch forums. With that all being said, we've come so far. We've covered a lot already,
but guess what time it is? Let's write some code. I'll see you in the next video.
We've covered enough of the fundamentals so far. Well, from a theory point of view,

```
let's get into coding. So I'm going to go over to Google Chrome. I'm going to
introduce you to
the tool. One of the main tools we're going to be using for the entire course. And
that is Google
Colab. So the way I would suggest following along with this course is remember, one
of the major
ones is to code along. So we're going to go to colab.research.google. I've got a
typo here.
Classic. You're going to see me do lots of typos throughout this course.
Colab.research.google.com.
This is going to load up Google Colab. Now, you can follow along with what I'm
going to do,
but if you'd like to find out how to use Google Colab from a top-down perspective,
you can go through some of these. I'd probably recommend going through overview of
Collaboratory Features. But essentially, what Google Colab is going to enable us to
do is
create a new notebook. And this is how we're going to practice writing PyTorch
code.
So if you refer to the reference document of learnpytorch.io, these are actually
Colab notebooks just in book format, so online book format. So these are the basis
materials
for what the course is going to be. There's going to be more here, but every new
module,
we're going to start a new notebook. And I'm going to just zoom in here.
So this one, the first module is going to be zero, zero, because Python code starts
at zero,
zero. And we're going to call this PyTorch Fundamentals. I'm going to call mine
video,
just so we know that this is the notebook that I wrote through the video. And what
this is going
to do is if we click Connect, it's going to give us a space to write Python code.
So here we can go
print. Hello, I'm excited to learn PyTorch. And then if we hit shift and enter, it
comes out like
that. But another beautiful benefit of Google Colab are PS. I'm using the pro
version, which
```

Section 15: Main Topics

Key Topics:

- However, you do not have to use the paid version for this course
- Google Colab comes with a free version, which you'll be able to use to complete this course
- Now we're going to see this later on code that runs on the GPU is a lot faster in terms of compute time, especially for deep learning

▶ 📄 Click to view detailed content

costs about $10 a month or so. That price may be different depending on where you're from.
The reason I'm doing that is because I use Colab all the time. However, you do not have to use
the paid version for this course. Google Colab comes with a free version, which you'll be able
to use to complete this course. If you see it worthwhile, I find the pro version is worthwhile.
Another benefit of Google Colab is if we go here, we can go to runtime. Let me just show you that
again. Runtime, change runtime type, hardware accelerator. And we can choose to run our code
on an accelerator here. Now we've got GPU and TPU. We're going to be focused on using
GPU. If you'd like to look into TPU, I'll leave that to you. But we can click GPU, click save.
And now our code, if we write it in such a way, will run on the GPU. Now we're going to see this
later on code that runs on the GPU is a lot faster in terms of compute time, especially for deep
learning. So if we write here in a video SMI, we now have access to a GPU. In my case, I have a
Tesla P100. It's quite a good GPU. You tend to get the better GPUs. If you pay for Google Colab,
if you don't pay for it, you get the free version, you get a free GPU. It just won't be as fast as
the GPUs you typically get with the paid version. So just keep that in mind. A whole bunch of stuff
that we can do here. I'm not going to go through it all because there's too much. But we've covered
basically what we need to cover. So if we just come up here, I'm going to write a text cell. So
oo dot pytorch fundamentals. And I'm going to link in here resource notebook. Now you can come
to learn pytorch.io and all the notebooks are going to be in sync. So 00, we can put this in here.
Resource notebook is there. That's what this notebook is going to be based off. This one here.
And then if you have a question about what's going on in this notebook,
you can come to the course GitHub. And then we go back, back. This is where you can see what's
going on. This is pytorch deep learning projects as you can see what's happening. At the moment,
I've got pytorch course creation because I'm in the middle of creating it. But if you have a question,
you can come to Mr. D Burke slash pytorch deep learning slash discussions, which is this tab here,
and then ask a question by clicking new discussion. So any discussions related to this notebook,
you can ask it there. And I'm going to turn this right now. This is a code cell.
CoLab is basically comprised of code and text cells. I'm going to turn this into a text cell
by pressing command mm, shift and enter. Now we have a text cell. And then if we wanted another
code cell, we could go like that text code text code, yada, yada, yada. But I'm going to delete this.

And to finish off this video, we're going to import pytorch. So we're going to import torch.
And then we're going to print torch dot dot version. So that's another beautiful thing about Google
Colab is that it comes with pytorch pre installed and a lot of other common Python data science
packages, such as we could also go import pandas as PD, import NumPy as MP import mapplot lib
lib dot pyplot as PLT. This is Google Colab is by far the easiest way to get started with this
course. You can run things locally. If you'd like to do that, I'd refer to you to pytorch deep
learning is going to be set up dot MD, getting set up to code pytorch. We've just gone through
number one setting up with Google Colab. There is also another option for getting started locally.
Right now, this document's a work in progress, but it'll be finished by the time you watch this
video. This is not a replacement, though, for the pytorch documentation for getting set up
locally. So if you'd like to run locally on your machine, rather than going on Google Colab,
please refer to this documentation or set up dot MD here. But if you'd like to get started
as soon as possible, I'd highly recommend you using Google Colab. In fact, the entire course
is going to be able to be run through Google Colab. So let's finish off this video, make sure
we've got pytorch ready to go. And of course, some fundamental data science packages here.
Wonderful. This means that we have pytorch 1.10.0. So if your version number is far greater than this,
maybe you're watching this video a couple of years in the future, and pytorch is up to 2.11,
maybe some of the code in this notebook won't work. But 1.10.0 should be more than enough for
what we're going to do. And plus Q111, CU111, stands for CUDA version 11.1, I believe. And what
that would mean is if we came in here, and we wanted to install it on Linux, which is what
Colab runs on, there's Mac and Windows as well. We've got CUDA. Yeah. So right now, as of recording
this video, the latest pytorch build is 1.10.2. So you'll need at least pytorch 1.10 to complete
this course and CUDA 11.3. So that's CUDA toolkit. If you remember, CUDA toolkit is NVIDIA's
programming. There we go. NVIDIA developer. CUDA is what enables us to run our pytorch code on
NVIDIA GPUs, which we have access to in Google Colab. Beautiful. So we're set up ready to write code.

Section 16: Main Topics

Key Topics:

- Let's get started in the next video writing some pytorch code
- We've got access to pytorch
- And one last thing, how I'd recommend going through this course is in a split window fashion

▶ 📄 Click to view detailed content

```
Let's get started in the next video writing some pytorch code. This is so exciting.
I'll see you
there. So we've got set up. We've got access to pytorch. We've got a Google Colab
instance running
here. We've got a GPU because we've gone up to runtime, change runtime type,
hardware accelerator.
You won't necessarily need a GPU for this entire notebook, but I just wanted to
show you how to
get access to a GPU because we're going to be using them later on. So let's get rid
of this.
And one last thing, how I'd recommend going through this course is in a split
window fashion.
So for example, you might have the video where I'm talking right now and writing
code on the
left side, and then you might have another window over the other side with your own
Colab
window. And you can go new notebook, call it whatever you want, my notebook. You
could call it very
similar to what we're writing here. And then if I write code over on this side, on
this video,
you can't copy it, of course, but you'll write the same code here and then go on
and go on and
go on. And if you get stuck, of course, you have the reference notebook and you
have an
opportunity to ask a question here. So with that being said, let's get started. The
first thing
we're going to have a look at in PyTorch is an introduction to tenses. So tenses
are the main
building block of deep learning in general, or data. And so you may have watched
the video,
what is a tensor? For the sake of this course, tenses are a way to represent data,
especially
multi dimensional data, numeric data that is, but that numeric data represents
something else.
So let's go in here, creating tenses. So the first kind of tensor we're going to
create is
actually called a scalar. I know I'm going to throw a lot of different names of
things at you,
but it's important that you're aware of such nomenclature. Even though in PyTorch,
almost
everything is referred to as a tensor, there are different kinds of tenses. And
just to
exemplify the fact that we're using a reference notebook, if we go up here, we can
see we have
```

importing PyTorch. We've done that. Now we're up to introduction to tenses. We've got creating
tenses, and we've got scalar, etc, etc, etc. So this is what we're going to be working through.
Let's do it together. So scalar, the way to, oops, what have I done there? The way to create a
tensor in PyTorch, we're going to call this scalar equals torch dot tensor. And we're going to fill
it with the number seven. And then if we press or retype in scalar, what do we get back? Seven,
wonderful. And it's got the tensor data type here. So how would we find out about what torch dot
tensor actually is? Well, let me show you how I would. We go to torch dot tensor. There we go.
We've got the documentation. So this is possibly the most common class in PyTorch other than
one we're going to see later on that you'll use, which is torch dot nn. Basically, everything in
PyTorch works off torch dot tensor. And if you'd like to learn more, you can read through here.
In fact, I would encourage you to read through this documentation for at least 10 minutes
after you finish some videos here. So with that being said, I'm going to link that in here.
So PyTorch tensors are created using torch dot tensor. And then we've got that link there.
Oops, typos got law Daniel. Come on. They're better than this. No, I'm kidding. There's going to be
typos got law through the whole course. Okay. Now, what are some attributes of a scalar? So
some details about scalars. Let's find out how many dimensions there are. Oh, and by the way,
this warning, perfect timing. Google Colab will give you some warnings here, depending on whether
you're using a GPU or not. Now, the reason being is because Google Colab provides GPUs to you and
I for free. However, GPUs aren't free for Google to provide. So if we're not using a GPU, we can
save some resources, allow someone else to use a GPU by going to none. And of course, we can
always switch this back. So I'm going to turn my GPU off so that someone else out there,
I'm not using the GPU at the moment, they can use it. So what you're also going to see is if
your Google Colab instance ever restarts up here, we're going to have to rerun these cells. So if
you stop coding for a while, go have a break and then come back and you start your notebook again,
that's one downside of Google Colab is that it resets after a few hours. How many hours? I don't
know exactly. The reset time is longer if you have the pro subscription, but because it's a free
service and the way Google calculate usage and all that sort of stuff, I can't give a conclusive
evidence or conclusive answer on how long until it resets. But just know, if you come back, you might
have to rerun some of your cells and you can do that with shift and enter. So a

```
scalar has no
dimensions. All right, it's just a single number. But then we move on to the next
thing. Or actually,
if we wanted to get this number out of a tensor type, we can use scalar dot item,
this is going
to give it back as just a regular Python integer. Wonderful, there we go, the
number seven back,
get tensor back as Python int. Now, the next thing that we have is a vector. So
let's write
```

Section 17: Main Topics

Key Topics:

- That's a little bit confusing, but the thing you should remember in PyTorch is basically anytime you encode data into numbers, it's of a tensor data type
- So we're covering a fair bit of ground here, nice and quick, but that's going to be the teaching style of this course is we're going to get quite hands on and writing a lot of code and just interacting with it rather than continually going back over and discussing what's going on here
- PyTorch will do a lot of that behind the scenes

▶ 📄 Click to view detailed content

```
in here vector, which again is going to be created with torch dot tensor. But you
will also hear
the word vector used a lot too. Now, what is the deal? Oops, seven dot seven.
Google Colab's auto
complete is a bit funny. It doesn't always do the thing you want it to. So if we
see a vector,
we've got two numbers here. And then if we really wanted to find out what is a
vector.
So a vector usually has magnitude and direction. So what we're going to see later
on is, there we
go, magnitude, how far it's going and which way it's going. And then if we plotted
it, we've got,
yeah, a vector equals the magnitude would be the length here and the direction
would be where it's
pointing. And oh, here we go, scalar vector matrix tensor. This is what we're
working on as well.
So the thing about vectors, how they differ with scalars is how I just remember
them is
rather than magnitude and direction is a vector typically has more than one number.
So if we go vector and dim, how many dimensions does it have?
It has one dimension, which is kind of confusing. But when we see tensors with more
than one
dimension, it'll make sense. And another way that I remember how many dimensions
```

something
has is by the number of square brackets. So let's check out something else. Maybe we go vector
dot shape shape is two. So the difference between dimension. So dimension is like number of square
brackets. And when I say, even though there's two here, I mean number of pairs of closing square
brackets. So there's one pair of closing square brackets here. But the shape of the vector is two.
So we have two by one elements. So that means a total of two elements. Now if we wanted to step
things up a notch, let's create a matrix. So this is another term you're going to hear.
And you might be wondering why I'm capitalizing matrix. Well, I'll explain that in the second
matrix equals torch dot tensor. And we're going to put two square brackets here. You might be
thinking, what could the two square brackets mean? Or actually, that's a little bit of a challenge.
If one pair of square brackets had an endem of one, what will the endem be number of dimensions
of two square brackets? So let's create this matrix. Beautiful. So we've got another tensor here.
Again, as I said, these things have different names, like the traditional name of scalar,
vector matrix, but they're all still a torch dot tensor. That's a little bit confusing,
but the thing you should remember in PyTorch is basically anytime you encode data into numbers,
it's of a tensor data type. And so now how many n number of dimensions do you think a matrix has?
It has two. So there we go. We have two square brackets. So if we wanted to get matrix,
let's index on the zeroth axis. Let's see what happens there. Ah, so we get seven and eight.
And then we get off the first dimension. Ah, nine and 10. So this is where the square brackets,
the pairings come into play. We've got two square bracket pairings on the outside here.
So we have an endem of two. Now, if we get the shape of the matrix, what do you think the shape will be?
Ah, two by two. So we've got two numbers here by two. So we have a total of four elements in there.
So we're covering a fair bit of ground here, nice and quick, but that's going to be the
teaching style of this course is we're going to get quite hands on and writing a lot of code and
just interacting with it rather than continually going back over and discussing what's going on
here. The best way to find out what's happening within a matrix is to write more code that's similar
to these matrices here. But let's not stop at matrix. Let's upgrade to a tensor now. So I might
put this in capitals as well. And I haven't explained what the capitals mean yet, but we'll see that
in a second. So let's go torch dot tensor. And what we're going to do is this time,
we've done one square bracket pairing. We've done two square bracket pairings.

```
Let's do three
square bracket pairings and just get a little bit adventurous. All right. And so
you might be thinking
at the moment, this is quite tedious. I'm just going to write a bunch of random
numbers here. One,
two, three, three, six, nine, two, five, four. Now you might be thinking, Daniel,
you've said
tensors could have millions of numbers. If we had to write them all by hand, that
would be
quite tedious. And yes, you're completely right. The fact is, though, that most of
the time,
you won't be crafting tensors by hand. PyTorch will do a lot of that behind the
scenes. However,
it's important to know that these are the fundamental building blocks of the models
and the deep learning neural networks that we're going to be building. So tensor
capitals as well,
we have three square brackets. So, or three square bracket pairings. I'm just going
to refer to three
square brackets at the very start because they're going to be paired down here. How
many n dim or
number of dimensions do you think our tensor will have? Three, wonderful. And what
do you think the
shape of our tensor is? We have three elements here. We have three elements here,
three elements
here. And we have one, two, three. So maybe our tensor has a shape of one by three
by three.
Hmm. What does that mean? Well, we've got three by one, two, three. That's the
second square
```

Section 18: Main Topics

Key Topics:

- Ah, so that's the first dimension there or the zeroth dimension because we remember PyTorch is zero indexed
- In the last video, we covered the basic building blocks of data representation in deep learning, which is the tensor, or in PyTorch, specifically torch
- So I hope you gave that a shot because as you'll see throughout the course and your deep learning journey, a tensor can represent or can be of almost any shape and size and have almost any combination of numbers within it

▶ 📄 Click to view detailed content

```
bracket there by one. Ah, so that's the first dimension there or the zeroth
dimension because
we remember PyTorch is zero indexed. We have, well, let's just instead of talking
about it,
```

let's just get on the zeroth axis and see what happens with the zeroth dimension.
There we go.
Okay. So there's, this is the far left one, zero, which is very confusing because we've got a one
here, but so we've got, oops, don't mean that. What this is saying is we've got one three by three
shape tensor. So very outer bracket matches up with this number one here. And then this three
matches up with the next one here, which is one, two, three. And then this three matches up with
this one, one, two, three. Now, if you'd like to see this with a pretty picture, we can see it here.
So dim zero lines up. So the blue bracket, the very outer one, lines up with the one. Then dim
equals one, this one here, the middle bracket, lines up with the middle dimension here. And then
dim equals two, the very inner lines up with these three here. So again, this is going to take a lot
of practice. It's taken me a lot of practice to understand the dimensions of tensors. But
to practice, I would like you to write out your own tensor of, you can put however many square
brackets you want. And then just interact with the end dim shape and indexing, just as I've done
here, but you can put any combination of numbers inside this tensor. That's a little bit of practice
before the next video. So give that a shot and then we'll move on to the next topic.
I'll see you there. Welcome back. In the last video, we covered the basic building blocks of data
representation in deep learning, which is the tensor, or in PyTorch, specifically torch.tensor.
But within that, we had to look at what a scalar is. We had to look at what a vector is. We had to
look at a matrix. We had to look at what a tensor is. And I issued you the challenge to get as
creative as you like with creating your own tensor. So I hope you gave that a shot because as you'll
see throughout the course and your deep learning journey, a tensor can represent or can be of almost
any shape and size and have almost any combination of numbers within it. And so this is very important
to be able to interact with different tensors to be able to understand what the different names of
things are. So when you hear matrix, you go, oh, maybe that's a two dimensional tensor. When you
hear a vector, maybe that's a one dimensional tensor. When you hear a tensor, that could be any
amount of dimensions. And just for reference for that, if we come back to the course reference,
we've got a scalar. What is it? A single number, number of dimensions, zero. We've got a vector,
a number with direction, number of dimensions, one, a matrix, a tensor. And now here's another little
tidbit of the nomenclature of things, the naming of things. Typically, you'll see a variable name
for a scalar or a vector as a lowercase. So a vector, you might have a lowercase y

storing that
data. But for a matrix or a tensor, you'll often see an uppercase letter or variable in Python in
our case, because we're writing code. And so I am not exactly sure why this is, but this is just
what you're going to see in machine learning and deep learning code and research papers
across the board. This is a typical nomenclature. Scalars and vectors, lowercase, matrix and tensors,
uppercase, that's where that naming comes from. And that's why I've given the tensor uppercase here.
Now, with that being said, let's jump in to another very important concept with tensors.
And that is random tensors. Why random tensors? I'm just writing this in a code cell now.
I could go here. This is a comment in Python, random tensors. But we'll get rid of that. We could
just start another text cell here. And then three hashes is going to give us a heading, random tensors
there. Or I could turn this again into a markdown cell with command mm when I'm using Google Colab.
So random tensors. Let's write down here. Why random tensors? So we've done the tedious thing
of creating our own tensors with some numbers that we've defined, whatever these are. Again,
you could define these as almost anything. But random tensors is a big part in pytorch because
let's write this down. Random tensors are important because the way many neural networks learn is
that they start with tensors full of random numbers and then adjust those random numbers
to better represent the data. So seriously, this is one of the big concepts of neural networks.
I'm going to write in code here, which is this is what the tick is for. Start with random numbers.
Look at data, update random numbers. Look at data, update random numbers. That is the crux
of neural networks. So let's create a random tensor with pytorch. Remember how I said that
pytorch is going to create tensors for you behind the scenes? Well, this is one of the ways that
it does so. So we create a random tensor and we give it a size of random tensor of size or shape.
Pytorch use these independently. So size, shape, they mean the different versions of the same thing.
So random tensor equals torch dot rand. And we're going to type in here three, four. And the beautiful
thing about Google Colab as well is that if we wait long enough, it's going to pop up with the doc

---

Section 19: Main Topics

Key Topics:

- This is the PyTorch Fundamentals notebook
- So the takeaway from this video is that PyTorch enables you to create tensors quite easily with the random method

▶ 📄 Click to view detailed content

```
string of what's going on. I personally find this a little hard to read in Google Colab,
because you see you can keep going down there. You might be able to read that. But what can we do?
Well, we can go to torch dot rand. Then we go to the documentation. Beautiful. Now there's a whole
bunch of stuff here that you're more than welcome to read. We're not going to go through all that.
We're just going to see what happens hands on. So we'll copy that in here. And write this in notes,
torch random tensors. Done. Just going to make some code cells down here. So I've got some space.
I can get this a bit up here. Let's see what our random tensor looks like. There we go. Beautiful
of size three, four. So we've got three or four elements here. And then we've got three deep
here. So again, there's the two pairs. So what do you think the number of dimensions will be
for random tensor? And dim. Two beautiful. And so we have some random numbers here. Now the
beautiful thing about pie torch again is that it's going to do a lot of this behind the scenes. So
if we wanted to create a size of 10 10, in some cases, we won't want one dimension here. And then
it's going to go 10 10. And then if we check the number of dimensions, how many do you think it
will be now three? Why is that? Because we've got one 10 10. And then if we wanted to create 10 10 10.
What's the number of dimensions going to be? It's not going to change. Why is that?
We haven't run that cell yet, but we've got a lot of numbers here.
We can find out what 10 times 10 times 10 is. And I know we can do that in our heads, but
the beauty of collab is we've got a calculator right here. 10 times 10 times 10. We've got a
thousand elements in there. But sometimes tenses can be hundreds of thousands of elements or
millions of elements. But pie torch is going to take care of a lot of this behind the scenes. So
let's clean up a bit of space here. This is a random tensor. Random numbers beautiful of now
it's got two dimensions because we've got three by four. And if we put another one in the front
there, we're going to have how many dimensions three dimensions there. But again, this number
of dimensions could be any number. And what's inside here could be any number.
Let's get rid of that.
```

And let's get a bit specific because right now this is just a random tensor of whatever dimension.
How about we create a random tensor with similar shape to an image tensor. So a lot of the time
when we turn images, image size tensor, when we turn images into tenses, they're going to have,
let me just write it in code for you first, size equals a height, a width, and a number of color
channels. And so in this case, it's going to be height with color channels. And the color channels
are red, green, blue. And so let's create a random image tensor. Let's view the size of it or the
shape. And then random image size tensor will view the end dim. Beautiful. Okay, so we've got
torch size, the same size two, two, four, two, four, three, height, width, color channels. And we've got
three dimensions, one, four, height, width, color channels. Let's go and see an example of this. This
is the PyTorch Fundamentals notebook. If we go up to here, so say we wanted to encode this image
of my dad eating pizza with thumbs up of a square image of two, two, four by two, two, four.
This is an input. And if we wanted to encode this into tensor format, well, one of the ways of
representing an image tensor, very common ways is to split it into color channels because with
red, green, and blue, you can create almost any color you want. And then we have a tensor
representation. So sometimes you're going to see color channels come first. We can switch this
around and our code quite easily by going color channels here. But you'll also see color channels
come at the end. I know I'm saying a lot that we kind of haven't covered yet. The main takeaway
from here is that almost any data can be represented as a tensor. And one of the common ways to represent
images is in the format color channels, height, width, and how these values are will depend on
what's in the image. But we've done this in a random way. So the takeaway from this video is
that PyTorch enables you to create tensors quite easily with the random method. However, it is
going to do a lot of this creating tensors for you behind the scenes and why a random tensor is so
valuable because neural networks start with random numbers, look at data such as image tensors,
and then adjust those random numbers to better represent that data. And they repeat those steps
onwards and onwards and onwards. Let's finish this video here. I'm going to challenge for you
just to create your own random tensor of whatever size and shape you want. So you could have 5, 10,
10 here and see what that looks like. And then we'll keep coding in the next video.
I hope you took on the challenge of creating random tensor of your own size. And just a little
tidbit here. You might have seen me in the previous video. I didn't use the size parameter. But in

```
this case, I did here, you can go either way. So if we go torch dot rand size
equals, we put in a
```

Section 20: Main Topics

Key Topics:

- We've actually been using torch dot float the whole time, because that's whenever you create a tensor with pytorch, we're using a pytorch method, unless you explicitly define what the data type is, we'll see that later on, defining what the data type is, it starts off as torch float 32
- And if we just write in torch dot a range, we've got tensors of zero to nine, because it of course starts at zero index

▶ 📄 Click to view detailed content

```
tuple here of three three, we've got that tensor there three three. But then also
if we don't put
the size in there, it's the default. So it's going to create a very similar tensor.
So whether you
have this size or not, it's going to have quite a similar output depending on the
shape that you
put in there. But now let's get started to another kind of tensor that you might
see zeros and ones.
So say you wanted to create a tensor, but that wasn't just full of random numbers,
you wanted to create a tensor of all zeros. This is helpful for if you're creating
some form of
mask. Now, we haven't covered what a mask is. But essentially, if we create a
tensor of all zeros,
what happens when you multiply a number by zero? All zeros. So if we wanted to
multiply
these two together, let's do zeros times random tensor.
There we go, all zeros. So maybe if you're working with this random tensor and you
wanted to mask
out, say all of the numbers in this column for some reason, you could create a
tensor of zeros in
that column, multiply it by your target tensor, and you would zero all those
numbers. That's telling
your model, hey, ignore all of the numbers that are in here because I've zeroed
them out. And then
if you wanted to create a tensor of all ones, create a tensor of all ones, we can
go ones equals
torch dot ones, size equals three, four. And then if we have a look, there's
another parameter I
haven't showed you yet, but this is another important one is the D type. So the
default data type,
so that's what D type stands for, is torch dot float. We've actually been using
```

torch dot float
the whole time, because that's whenever you create a tensor with pytorch, we're using a pytorch
method, unless you explicitly define what the data type is, we'll see that later on, defining
what the data type is, it starts off as torch float 32. So these are float numbers. So that
is how you create zeros and ones zeros is probably I've seen more common than ones in use, but just
keep these in mind, you might come across them. There are lots of different methods to creating
tensors. And truth be told, like random is probably one of the most common, but you might see zeros
and ones out in the field. So now we've covered that. Let's move on into the next video, where
we're going to create a range. So have a go at creating a tensor full of zeros and whatever size
you want, and a tensor full of ones and whatever size you want. And I'll see you in the next video.
Welcome back. I hope you took on the challenge of creating a torch tensor of zeros of your
own size and ones of your own size. But now let's investigate how we might create a range of
tensors and tensors like. So these are two other very common methods of creating tensors.
So let's start by creating a range. So we'll first use torch dot range, because depending on
when you're watching this video, torch dot range may be still in play or it may be deprecated.
If we write in torch dot range right now with the pie torch version that I'm using, which is
torch dot version, which is torch or pie torch 1.10 point zero torch range is deprecated and
will be removed in a future release. So just keep that in mind. If you come across some code that's
using torch dot range, maybe out of whack. So the way to get around that is to fix that is to use
a range instead. And if we just write in torch dot a range, we've got tensors of zero to nine,
because it of course starts at zero index. If we wanted one to 10, we could go like this.
1, 2, 3, 4, 5, 6, 7, 8, 9, 10. And we can go zero, or we go 1, 2, 10, equals torch a range.
Wonderful. And we can also define the step. So let's let's type in some start and where can we
find the documentation on a range? Sometimes in Google Colab, you can press shift tab,
but I find that it doesn't always work for me. Yeah, you could hover over it, but we can also just
go torch a range and look for the documentation torch a range. So we've got start and step. Let's
see what all of these three do. Maybe we start at zero, and maybe we want it to go to a thousand,
and then we want a step of what should our step be? What's a fun number? 77. So it's not one to 10
anymore, but here we go. We've got start at zero, 77 plus 77 plus 77, all the way up to it finishes

```
at a thousand. So if we wanted to take it back to one to 10, we can go up here.
110, and the default
step is going to be one. Oops, we needed the end to be that it's going to finish at
end minus one.
There we go. Beautiful. Now we can also create tensors like. So creating tensors
like. So tensors
like is say you had a particular shape of a tensor you wanted to replicate
somewhere else, but you
didn't want to explicitly define what that shape should be. So what's the shape of
one to 10?
One to 10. Now if we wanted to create a tensor full of zeros that had the same
shape as this,
we can use tensor like or zeros like. So 10 zeros, zeros equals, I'm not even sure
if I'm
```

Section 21: Main Topics

Key Topics:

- float 32, even though we put none, this is because the default data type in pytorch, even if it's specified as none is going to come out as float 32
- Now this could be true, of course, we're going to set this as false
- Now, again, you won't necessarily always have to enter these when you're creating tensors, because pytorch does a lot of tensor creation behind the scenes for you

▶ 📄 Click to view detailed content

```
spelling zeros right then, zeros. Well, I might have a typo spelling zeros here,
but you get what
I'm saying is torch zeros. Oh, torch spell it like that. That's why I'm spelling it
like that.
Zeros like one to 10. And then the input is going to be one to 10. And we have a
look at 10 zeros.
My goodness, this is taking quite the while to run. This is troubleshooting on the
fly.
If something's happening like this, you can try to stop. If something was happening
like that,
you can click run and then stop. Well, it's running so fast that I can't click
stop. If you do also
run into trouble, you can go runtime, restart runtime. We might just do that now
just to show you.
Restart and run all is going to restart the compute engine behind the collab
notebook.
And run all the cells to where we are. So let's just see that we restart and run
runtime. If you're
getting errors, sometimes this helps. There is no set in stone way to troubleshoot
```

errors. It's
guess and check with this. So there we go. We've created 10 zeros, which is torch zeros like
our one to 10 tensor. So we've got zeros in the same shape as one to 10. So if you'd like to create
tensors, use torch arrange and get deprecated message. Use torch arrange instead for creating
a range of tensors with a start and end in a step. And then if you wanted to create tensors
or a tensor like something else, you want to look for the like method. And then you put an input,
which is another tensor. And then it'll create a similar tensor with whatever this method here
is like in that fashion or in the same shape as your input. So with that being said,
give that a try, create a range of tensors, and then try to replicate that range shape that you've
made with zeros. I'll see you in the next video. Welcome back. Let's now get into a very important
topic of tensor data types. So we've briefly hinted on this before. And I said that let's create
a tensor to begin with float 32 tensor. And we're going to go float 32 tensor equals torch
dot tensor. And let's just put in the numbers three, six, nine. If you've ever played need for
speed underground, you'll know where three, six, nine comes from. And then we're going to go
D type equals, let's just put none and see what happens, hey, float 32 tensor. Oh, what is the
data type? float 32, tensor dot D type. float 32, even though we put none, this is because
the default data type in pytorch, even if it's specified as none is going to come out as float 32.
What if we wanted to change that to something else? Well, let's type in here float 16.
And now we've got float 32 tensor. This variable name is a lie now because it's a float 16 tensor.
So we'll leave that as none. Let's go there. There's another parameter when creating tensors.
It's very important, which is device. So we'll see what that is later on. And then there's a
final one, which is also very important, which is requires grad equals false. Now this could be
true, of course, we're going to set this as false. So these are three of the most important parameters
when you're creating tensors. Now, again, you won't necessarily always have to enter these when
you're creating tensors, because pytorch does a lot of tensor creation behind the scenes for you.
So let's just write out what these are. Data type is what data type is the tensor, e.g. float 32,
or float 16. Now, if you'd like to look at what data types are available for pytorch tensors,
we can go torch tensor and write up the top unless the documentation changes. We have data types.
It's so important that data types is the first thing that comes up when you're creating a tensor.

```
So we have 32-bit floating point, 64-bit floating point, 16, 16, 32-bit complex.
Now,
the most common ones that you will likely interact with are 32-bit floating point
and 16-bit floating
point. Now, what does this mean? What do these numbers actually mean? Well, they
have to do with
precision in computing. So let's look up that. Precision in computing. Precision
computer science.
So in computer science, the precision of a numerical quantity, we're dealing with
numbers, right?
As a measure of the detail in which the quantity is expressed. This is usually
measured in bits,
but sometimes in decimal digits. It is related to precision in mathematics, which
describes the
number of digits that are used to express a value. So, for us, precision is the
numerical quantity,
is a measure of the detail, how much detail in which the quantity is expressed. So,
I'm not going
to dive into the background of computer science and how computers represent
numbers. The important
takeaway for you from this will be that single precision floating point is usually
called float
32, which means, yeah, a number contains 32 bits in computer memory. So if you
imagine, if we have
a tensor that is using 32 bit floating point, the computer memory stores the number
as 32 bits.
Or if it has 16 bit floating point, it stores it as 16 bits or 16 numbers
representing or 16.
I'm not sure if a bit equates to a single number in computer memory. But what this
means is that
a 32 bit tensor is single precision. This is half precision. Now, this means that
it's the default
of 32, float 32, torch dot float 32, as we've seen in code, which means it's going
to take up
```

Section 22: Main Topics

Key Topics:

- I'm spending a lot of time here, because I'm going to put a note here, note, tensor data types is one of the three big issues with pytorch and deep learning or not not issues, they're going to be errors that you run into and deep learning
- Three big errors, you'll run into with pytorch and deep learning
- So for example, you have one tensor that lives on a GPU for fast computing, and you have another tensor that lives on a CPU and you try to do something with them, while pytorch is going to throw you an error

▶ 📄 Click to view detailed content

a certain amount of space in computer memory. Now, you might be thinking, why would I do anything
other than the default? Well, if you'd like to sacrifice some detail in how your number is
represented. So instead of 32 bits, it's represented by 16 bits, you can calculate faster on numbers
that take up less memory. So that is the main differentiator between 32 bit and 16 bit. But if
you need more precision, you might go up to 64 bit. So just keep that in mind as you go forward.
Single precision is 32. Half precision is 16. What do these numbers represent? They represent
how much detail a single number is stored in memory. That was a lot to take in. But we're talking
about 10 to data types. I'm spending a lot of time here, because I'm going to put a note here,
note, tensor data types is one of the three big issues with pytorch and deep learning or
not not issues, they're going to be errors that you run into and deep learning. Three big
errors, you'll run into with pytorch and deep learning. So one is tensors, not right data type.
Two tensors, not right shape. We've seen a few shapes of four and three tensors, not on the right
device. And so in this case, if we had a tensor that was float 16 and we were trying to do computations
with a tensor that was float 32, we might run into some errors. And so that's the tensors not
being in the right data type. So it's important to know about the D type parameter here. And then
tensors not being the right shape. Well, that's once we get onto matrix multiplication, we'll see
that if one tensor is a certain shape and another tensor is another shape and those shapes don't
line up, we're going to run into shape errors. And this is a perfect segue to the device.
Device equals none. By default, this is going to be CPU. This is why we are using Google Colab
because it enables us to have access to, oh, we don't want to restart, enables us to have access
to a GPU. As I've said before, a GPU enables us. So we could change this to CUDA. That would be,
we'll see how to write device agnostic code later on. But this device, if you try to do
operations between two tensors that are not on the same device. So for example, you have one tensor
that lives on a GPU for fast computing, and you have another tensor that lives on a CPU and you
try to do something with them, while pytorch is going to throw you an error. And then finally,
this last requirement is grad is if you want pytorch to track the gradients, we haven't covered
what that is of a tensor when it goes through certain numerical calculations. This is a bit of
a bombardment, but I thought I'd throw these in as important parameters to be aware of since

we're discussing data type. And really, it would be reminiscent of me to discuss data type without
discussing not the right shape or not the right device. So with that being said, let's write down
here what device is your tensor on, and whether or not to track gradients with this tensor's
operations. So we have a float 32 tensor. Now, how might we change the tensor data type of this?
Let's create float 16 tensor. And we saw that we could explicitly write in float 16 tensor.
Or we can just type in here, float 16 tensor equals float 32 tensor dot type. And we're going to type
in torch dot float 16, why float 16, because well, that's how we define float 16, or we could use
half. So the same thing, these things are the same, let's just do half, or float 16 is more
explicit for me. And then let's check out float 16 tensor. Beautiful, we've converted our float
32 tensor into float 16. So that is one of the ways that you'll be able to tackle the tensors
not in the right data type issue that you run into. And just a little note on the precision
and computing, if you'd like to read more on that, I'm going to link this in here. And this is all
about how computers store numbers. So precision in computing. There we go. I'll just get rid of that.
Wonderful. So give that a try, create some tensors, research, or go to the documentation of torch
dot tensor and see if you can find out a little bit more about D type device and requires grad,
and create some tensors of different data types. Play around with whatever the ones you want here,
and see if you can run into some errors, maybe try to multiply two tensors together. So if you go
float 16 tensor times float 32 tensor, give that a try and see what happens. I'll see you in the next
video. Welcome back. In the last video, we covered a little bit about tensor data types,
as well as some of the most common parameters you'll see past to the torch dot tensor method.
And so I should do the challenge at the end of the last video to create some of your own tensors
of different data types, and then to see what happens when you multiply a float 16 tensor by a
float 32 tensor. Oh, it works. And but you've like Daniel, you said that you're going to have tensors
not the right data type. Well, this is another kind of gotcha or caveat of pie torch and deep
learning in general, is that sometimes you'll find that even if you think something may error
because these two tensors are different data types, it actually results in no error. But then

Section 23: Main Topics

Key Topics:

- Typo, of course, one of many in 32 tensor
- So if we run into one of the three big problems in deep learning and neural networks in general, especially with PyTorch, tensor's not the right data type, tensor's not the right shape or tensor's not on the right device

▶ 📄 Click to view detailed content

```
sometimes you'll have other operations that you do, especially training large
neural networks,
where you'll get data type issues. The important thing is to just be aware of the
fact that some
operations will run an error when your tensors are not in the right data type. So
let's try another
type. Maybe we try a 32 bit integer. So torch dot in 32. And we try to multiply
that by a float.
Wonder what will happen then? So let's go into 32 in 32 tensor equals torch dot
tensor. And we'll
just make it three. Notice that there's no floats there or no dot points to make it
a float.
Three, six, nine and D type can be torch in 32. And then in 32 tensor, what does
this look like?
Typo, of course, one of many in 32 tensor. So now let's go float 32 tensor and see
what happens.
Can we get pie torch to throw an error in 32 tensor?
Huh, it worked as well. Or maybe we go into 64. What happens here?
Still works. Now, see, this is again one of the confusing parts of doing tensor
operations.
What if we do a long tensor? Torch to long. Is this going to still work?
Ah, torch has no attribute called long. That's not a data type issue.
I think it's long tensor. Long tensor. Does this work? D type must be torch D type.
Torch long tensor. I could have sworn that this was torch dot tensor.
Oh, there we go. Torch dot long tensor. That's another word for 64 bit.
So what is this saying? CPU tensor. Okay, let's see. This is some troubleshooting
on the fly here.
Then we multiply it. This is a float 32 times a long. It works. Okay, so it's
actually a bit
more robust than what I thought it was. But just keep this in mind when we're
training models,
we're probably going to run into some errors at some point of our tensor's not
being the
right data type. And if pie torch throws us an error saying your tensors are in the
wrong data
type, well, at least we know now how to change that data type or how to set the
data type if we
need to. And so with that being said, let's just formalize what we've been doing a
fair bit already.
And that's getting information from tensors. So the three big things that we'll
```

want to get
from our tensors in line with the three big errors that we're going to face in neural networks and
deep lining is let's copy these down. Just going to get this, copy this down below. So if we want
to get some information from tensors, how do we check the shape? How do we check the data type?
How do we check the device? Let's write that down. So to get information from this, to get
D type or let's write data type from a tensor can use tensor dot D type. And let's go here to get
shape from a tensor can use tensor dot shape. And to get device from a tensor, which devices it on
CPU or GPU can use tensor dot device. Let's see these three in action. So if we run into one of
the three big problems in deep learning and neural networks in general, especially with PyTorch,
tensor's not the right data type, tensor's not the right shape or tensor's not on the right device.
Let's create a tensor and try these three out. We've got some tensor equals torch dot
rand and we'll create it a three four. Let's have a look at what it looks like.
There we go. Random numbers of shape three and four. Now let's find out some details about it.
Find out details about some tensor. So print or print some tensor.
And oops, didn't want that print. And let's format it or make an F string of shape of tensor.
Oh, let's do data type first. We'll follow that order.
Data type of tensor. And we're going to go, how do we do this? Some tensor dot what?
Dot d type. Beautiful. And then we're going to print tensors not in the right shape. So let's go
shape of tensor equals some tensor dot shape. Oh, I went a bit too fast, but we could also use
size. Let's just confirm that actually. We'll code that out together. From my experience,
some tensor dot size, and some tensor dot shape result in the same thing. Is that true? Oh, function.
Oh, that's what it is. Some tensor dot size is a function, not an attribute.
There we go. Which one should you use? For me, I'm probably more used to using shape. You may come
across dot size as well, but just realize that they do quite the same thing except one's a function
and one's an attribute. An attribute is written dot shape without the curly brackets. A function
or a method is with the brackets at the end. So that's the difference between these are attributes
here. D type size. We're going to change this to shape. Tensor attributes. This is what we're
getting. I should probably write that down. This is tensor attributes. That's the formal name for
these things. And then finally, what else do we want? Tensors, what device are we looking for?
Let's get rid of this, get rid of this. And then print f device tensor is on. By default,
our tensor is on the CPU. So some tensor dot device. There we go. So now we've got our tensor

```
here, some tensor. The data type is a torch float 32 because we didn't change it to
anything else.
And torch float 32 is the default. The shape is three four, which makes a lot of
sense because
```

Section 24: Main Topics

Key Topics:

- And the device tensor is on is the CPU, which is, of course, the default, unless we explicitly say to put it on another device, all of the tensors that we create will default to being on the CPU, rather than the GPU
- But see how to change the device a pytorch tensor is on
- And I alluded to the fact that these will help resolve three of the most common issues in building neural networks, deep learning models, specifically with pytorch

▶ 📄 Click to view detailed content

```
we passed in three four here. And the device tensor is on is the CPU, which is, of
course,
the default, unless we explicitly say to put it on another device, all of the
tensors that we
create will default to being on the CPU, rather than the GPU. And we'll see later
on how to put
tensors and other things in torch onto a GPU. But with that being said, give it a
shot,
create your own tensor, get some information from that tensor, and see if you can
change
these around. So see if you could create a random tensor, but instead of float 32,
it's a float 16.
And then probably another extracurricular, we haven't covered this yet. But see how
to change
the device a pytorch tensor is on. Give that a crack. And I'll see you in the next
video.
Welcome back. So in the last video, we had a look at a few tensor attributes,
namely the data
type of a tensor, the shape of a tensor, and the device that a tensor lives on. And
I alluded to
the fact that these will help resolve three of the most common issues in building
neural networks,
deep learning models, specifically with pytorch. So tensor has not been the right
data type,
tensor has not been the right shape, and tensor has not been on the right device.
So now let's
get into manipulating tensors. And what I mean by that, so let's just write here
the title,
manipulating tensors. And this is going to be tensor operations. So when we're
```

building neural
networks, neural networks are comprised of lots of mathematical functions that
pytorch code is going
to run behind the scenes for us. So let's go here, tensor operations include
addition,
subtraction, and these are the regular addition, subtraction, multiplication.
There's two types
of multiplication in that you'll typically see referenced in deep learning and
neural networks,
division, and matrix multiplication. And these, the ones here, so addition,
subtraction,
multiplication, division, your typical operations that you're probably familiar
with matrix multiplication.
The only different one here is matrix multiplication. We're going to have a look at
that in a minute.
But to find patterns in numbers of a data set, a neural network will combine these
functions
in some way, shape or form. So it takes a tensor full of random numbers, performs
some kind of
combination of addition, subtraction, multiplication, division, matrix
multiplication. It doesn't have
to be all of these. It could be any combination of these to manipulate these
numbers in some way
to represent a data set. So that's how a neural network learns is it will just
comprise these
functions, look at some data to adjust the numbers of a random tensor, and then go
from there. But
with that being said, let's look at a few of these. So we'll begin with addition.
First thing we need
to do is create a tensor. And to add something to a tensor, we'll just go torch
tensor. Let's go one,
two, three, add something to a tensor is tensor plus, we can use plus as the
addition operator,
just like in Python, tensor plus 10 is going to be tensor 11, 12, 13, tensor plus
100 is going to be
as you'd expect plus 100. Let's leave that as plus 10 and add 10 to it. And so you
might be
able to guess how we would multiply it by 10. So let's go multiply tensor by 10. We
can go tensor,
star, which are my keyboard shift eight, 10. We get 10, 10, 10. And because we
didn't reassign it,
our tensor is still 123. So if we go, if we reassign it here, tensor equals tensor
by 10,
and then check out tensor, we've now got 10 2030. And the same thing here, we'll
have 10 2030. But
then if we go back from the top, if we delete this reassignment, oh, what do we get
there, tensor
by 10. Oh, what's happened here? Oh, because we've got, yeah, okay, I see, tensor
by 10, tensor,
still 123. What should we try now? How about subtract subtract 10 equals tensor
minus 10.
And you can also use, well, there we go, one minus 10, eight minus 10, three minus
10.
You can also use like torch has inbuilt functions or pytorch. So try out pytorch
inbuilt functions. So torch dot mall is short for multiply. We can pass in our
tensor here,
and we can add in 10. That's going to multiply each element of tensor by 10. So

```
just taking
the original tensor that we created, which is 123. And performing the same thing as
this,
I would recommend where you can use the operators from Python. If for some reason,
you see torch
dot mall, maybe there's a reason for that. But generally, these are more
understandable if you
just use the operators, if you need to do a straight up multiplication, straight up
addition, or straight
up subtraction, because torch also has torch dot add, torch dot add, is it torch
dot add? It might
be torch dot add. I'm not sure. Oh, there we go. Yeah, torch dot add. So as I
alluded to before,
there's two different types of multiplication that you'll hear about element wise
and matrix
multiplication. We're going to cover matrix multiplication in the next video. As a
challenge,
though, I would like you to search what is matrix multiplication. And I think the
first website that
comes up, matrix multiplication, Wikipedia, yeah, math is fun. It has a great
guide. So before we
get into matrix multiplication, jump into math is fun to have a look at matrix
multiplying,
and have a think about how we might be able to replicate that in pie torch. Even if
you're not
```

Section 25: Main Topics

Key Topics:

- Yeah, let's write that two main ways of performing multiplication in neural networks and deep learning
- A few different options there, but let's look at what it looks like in pytorch code
- So that's what I'd encourage you to go through step by step and reproduce this a good challenge would be to reproduce this by hand with pytorch code

▶ 📄 Click to view detailed content

```
sure, just have a think about it. I'll see you in the next video. Welcome back. In
the last video,
we discussed some basic tensor operations, such as addition, subtraction,
multiplication,
element wise, division, and matrix multiplication. But we didn't actually go
through what matrix
multiplication is. So now let's start on that more particularly discussing the
difference between
element wise and matrix multiplication. So we'll come down here, let's write
another heading,
```

matrix multiplication. So there's two ways, or two main ways. Yeah, let's write that two main
ways of performing multiplication in neural networks and deep learning. So one is the simple
version, which is what we've seen, which is element wise multiplication. And number two is matrix
multiplication. So matrix multiplication is actually possibly the most common tensor operation you
will find inside neural networks. And in the last video, I issued the extra curriculum of having a
look at the math is fun dot com page for how to multiply matrices. So the first example they go
through is element wise multiplication, which just means multiplying each element by a specific
number. In this case, we have two times four equals eight, two times zero equals zero, two times one
equals two, two times negative nine equals negative 18. But then if we move on to matrix
multiplication, which is multiplying a matrix by another matrix, we need to do the dot product.
So that's something that you'll also hear matrix multiplication referred to as the dot product.
So these two are used interchangeably matrix multiplication or dot product. And if we just
look up the symbol for dot product, you'll find that it's just a dot. There we go, a heavy dot,
images. There we go, a dot B. So this is vector a dot product B. A few different options there,
but let's look at what it looks like in pytorch code. But first, there's a little bit of a
difference here. So how did we get from multiplying this matrix here of one, two, three, four, five,
six, times seven, eight, nine, 10, 11, 12? How did we get 58 there? Well, we start by going,
this is the difference between element wise and dot product, by the way, one times seven.
We'll record that down there. So that's seven. And then two times nine. So this is first row,
first column, two times nine is 18. And then three times 11 is 33. And if we add those up,
seven plus 18, plus 33, we get 58. And then if we were to do that for each other element that's
throughout these two matrices, we end up with something like this. So that's what I'd encourage
you to go through step by step and reproduce this a good challenge would be to reproduce this by
hand with pytorch code. But now let's go back and write some pytorch code to do both of these. So
I just want to link here as well, more information on multiplying matrices. So I'm going to turn
this into markdown. Let's first see element wise, element wise multiplication. We're going to start
with just a rudimentary example. So if we have our tensor, what is it at the moment? It's 123.
And then if we multiply that by itself, we get 149. But let's print something out so it looks a bit
prettier than that. So print, I'm going to turn this into a string. And then we do

that. So if we
print tensor times tensor, element wise multiplication is going to give us print equals. And then
let's do in here tensor times tensor. We go like that. Wonderful. So we get one times one
equals one, two times two equals four, three times three equals nine. Now for matrix multiplication,
pytorch stores matrix multiplication, similar to torch dot mall in the torch dot mat mall space,
which stands for matrix multiplication. So let's just test it out. Let's just true the exact
same thing that we did here, instead of element wise, we'll do matrix multiplication on our 123
tensor. What happens here? Oh my goodness, 14. Now why did we get 14 instead of 149? Can you guess
how we got to 14 or think about how we got to 14 from these numbers? So if we recall back,
we saw that for we're only multiplying two smaller tensors, by the way, 123. This example is with
a larger one, but the same principle applies across different sizes of tensors or matrices.
And when I say matrix multiplication, you can also do matrix multiplication between tensors.
And in our case, we're using vectors just to add to the confusion. But what is the difference
here between element wise and dot product? Well, we've got one main addition. And that is addition.
So if we were to code this out by hand, matrix multiplication by hand, we'd have recall that
the elements of our tensor are 123. So if we wanted to matrix multiply that by itself,
we'd have one times one, which is the equivalent of doing one times seven in this visual example.
And then we'd have plus, it's going to be two times two, two times two. What does that give us?
Plus three times three. What does it give us? Three times three. That gives us 14. So that's how
we got to that number there. Now we could do this with a for loop. So let's have a gaze at when I
say gaze, it means have a look. That's a Australian colloquialism for having a look. But I want to
show you the time difference in it might not actually be that big a difference if we do it by hand
versus using something like matmore. And that's another thing to note is that if PyTorch has a
method already implemented, chances are it's a fast calculating version of that method. So I know

Section 26: Main Topics

Key Topics:

- But once you start to build larger tensors, you might run into one of the most common errors in deep learning
- One of the most common errors in deep learning, we've already alluded to this as well, is shape errors
- So this is one of the most common errors that you're going to face in deep learning is that matrix one and matrix two shapes cannot be multiplied because it doesn't satisfy rule number one

▶ 📄 Click to view detailed content

```
for basic operators, I said it's usually best to just use this straight up basic
operator.
But for something like matrix multiplication or other advanced operators instead of
the basic
operators, you probably want to use the torch version rather than writing a for
loop, which is
what we're about to do. So let's go value equals zero. This is matrix
multiplication by hand. So
for I in range, len tensor, so for each element in the length of our tensor, which
is 123, we want to
update our value to be plus equal, which is doing this plus reassignment here. The
ith element in
each tensor times the ith element. So times itself. And then how long is this going
to take?
Let's now return the value. We should get 14, print 14. There we go. So 1.9
milliseconds on
whatever CPU that Google collab is using behind the scenes. But now if we time it
and use the torch
method torch dot matmore, it was tensor dot sensor. And again, we're using a very
small tensor. So
okay, there we go. It actually showed how much quicker it is, even with such a
small tensor.
So this is 1.9 milliseconds. This is 252 microseconds. So this is 10 times slower
using a for loop,
then pie torches vectorized version. I'll let you look into that if you want to
find out what
vectorization means. It's just a type of programming that rather than writing for
loops, because as
you could imagine, if this tensor was, let's say, had a million elements instead of
just three,
if you have to loop through each of those elements one by one, that's going to be
quite cumbersome.
So a lot of pie torches functions behind the scenes implement optimized functions
to perform
mathematical operations, such as matrix multiplication, like the one we did by
hand,
in a far faster manner, as we can see here. And that's only with a tensor of three
elements.
So you can imagine the speedups on something like a tensor with a million elements.
But with that being said, that is the crux of matrix multiplication. For a little
bit more,
I encourage you to read through this documentation here by mathisfun.com.
```

Otherwise,
let's look at a couple of rules that we have to satisfy for larger versions of matrix multiplication.
Because right now, we've done it with a simple tensor, only 123. Let's step things up a notch
in the next video. Welcome back. In the last video, we were introduced to matrix multiplication,
which although we haven't seen it yet, is one of the most common operations in neural networks.
And we saw that you should always try to use torches implementation of certain operations,
except if they're basic operations, like plus multiplication and whatnot,
because chances are it's a lot faster version than if you would do things by hand. And also,
it's a lot less code. Like compared to this, this is pretty verbose code compared to just a matrix
multiply these two tensors. But there's something that we didn't allude to in the last video.
There's a couple of rules that need to be satisfied when performing matrix multiplication.
It worked for us because we have a rather simple tensor. But once you start to build larger tensors,
you might run into one of the most common errors in deep learning. I'm going to write this down
actually here. This is one to be very familiar with. One of the most common errors in deep
learning, we've already alluded to this as well, is shape errors. So let's jump back to this in a
minute. I just want to write up here. So there are two rules that performing or two main rules
that performing matrix multiplication needs to satisfy. Otherwise, we're going to get an error.
So number one is the inner dimensions must match. Let's see what this means.
So if we want to have two tensors of shape, three by two, and then we're going to use the at symbol.
Now, we might be asking why the at symbol. Well, the at symbol is another, is a like an operator
symbol for matrix multiplication. So I just want to give you an example. If we go tensor at
at stands for matrix multiplication, we get tensor 14, which is exactly the same as what we got there.
Should you use at or should you use mat mall? I would personally recommend to use mat mall.
It's a little bit clearer at sometimes can get confusing because it's not as common as seeing
something like mat mall. So we'll get rid of that, but I'm just using it up here for brevity.
And then we're going to go three, two. Now, this won't work. We'll see why in a second.
But if we go two, three, at, and then we have three, two, this will work. Or, and then if we go
the reverse, say threes on the outside, twos here. And then we have twos on the inside and threes
on the outside, this will work. Now, why is this? Well, this is the rule number one. The inner
dimensions must match. So the inner dimensions are what I mean by this is let's create torch

```
round or create of size 32. And then we'll get its shape. So we have, so if we
created a tensor
like this, three, two, and then if we created another tensor, well, let me just
show you straight
up torch dot mat mall torch dot ran to watch this won't work. We'll get an error.
There we go. So
this is one of the most common errors that you're going to face in deep learning is
that matrix
one and matrix two shapes cannot be multiplied because it doesn't satisfy rule
number one.
The inner dimensions must match. And so what I mean by inner dimensions is this
dimension multiplied
```

Section 27: Main Topics

Key Topics:

- We'll go on with one of those common errors in deep learning shape errors
- Just watch what happens and we're going to replicate something like this in PyTorch code in the next video
- So this is what I've been alluding to as one of the most common errors in deep learning, and that is shape errors

▶ 📄 Click to view detailed content

```
by this dimension. So say we were trying to multiply three, two by three, two,
these are the inner
dimensions. Now this will work because why the inner dimensions match. Two, three
by three, two,
two, three by three, two. Now notice how the inner dimensions, inner, inner match.
Let's see what
comes out here. Look at that. And now this is where rule two comes into play. Two.
The resulting
matrix has the shape of the outer dimensions. So we've just seen this one two,
three at three, two,
which is at remember is matrix multiply. So we have a matrix of shape, two, three,
matrix multiply a matrix of three, two, the inner dimensions match. So it works.
The resulting shape
is what? Two, two. Just as we've seen here, we've got a shape of two, two. Now what
if we did
the reverse? What if we did this one that also will work? Three on the outside.
What do you think
is going to happen here? In fact, I encourage you to pause the video and give it a
go. So this
is going to result in a three three matrix. But don't take my word for it. Let's
have a look. Three,
put two on the inside and we'll put two on the inside here and then three on the
outside. What
```

does it give us? Oh, look at that. A three three. One, two, three. One, two, three. Now what if we
were to change this? Two and two. This can be almost any number you want. Let's change them both
to 10. What's going to happen? Will this work? What's the resulting shape going to be? So the
inner dimensions match? What's rule number two? The resulting matrix has the shape of the outer
dimension. So what do you think is going to be the shape of this resulting matrix multiplication?
Well, let's have a look. It's still three three. Wow. Now what if we go 10? 10 on the outside
and 10 and 10 on the inside? What do we get? Well, we get, I'm not going to count all of those,
but if we just go shape, we get 10 by 10. Because these are the two main rules of matrix multiplication
is if you're running into an error that the matrix multiplication can't work. So let's say this was
10 and this was seven. Watch what's going to happen? We can't multiply them because the inner
dimensions do not match. We don't have 10 and 10. We have 10 and seven. But then when we change
this so that they match, we get 10 and 10. Beautiful. So now let's create a little bit more of a
specific example. We'll create two tenses. We'll come down. Actually, to prevent this video from
being too long, I've got an error in the word error. That's funny. We'll go on with one of those
common errors in deep learning shape errors. We've just seen it, but I'm going to get a little bit
more specific with that shape error in the next video. Before we do that, have a look at matrix
multiplication. There's a website, my other favorite website. I told you I've got two. This is my
other one. Matrix multiplication dot XYZ. This is your challenge before the next video. Put in
some random numbers here, whatever you want, two, 10, five, six, seven, eight, whatever you want. Change
these around a bit, three, four. Well, that's a five, not a four. And then multiply and just watch
what happens. That's all I'd like you to do. Just watch what happens and we're going to replicate
something like this in PyTorch code in the next video. I'll see you there.
Welcome back. In the last video, we discussed a little bit more about matrix multiplication,
but we're not done there. We looked at two of the main rules of matrix multiplication,
and we saw a few errors of what happens if those rules aren't satisfied, particularly if the
inner dimensions don't match. So this is what I've been alluding to as one of the most common
errors in deep learning, and that is shape errors. Because neural networks are comprised of lots of
matrix multiplication operations, if you have some sort of tensor shape error somewhere
in your neural network, chances are you're going to get a shape error. So now let's investigate

```
how we can deal with those. So let's create some tenses, shapes for matrix
multiplication.
And I also showed you the website, sorry, matrix multiplication dot xyz. I hope you
had a go at
typing in some numbers here and visualizing what happens, because we're going to
reproduce
something very similar to what happens here, but with PyTorch code. Shapes for
matrix multiplication,
we have tensor a, let's create this as torch dot tensor. We're going to create a
tensor with
just the elements one, two, all the way up to, let's just go to six, hey, that'll
be enough. Six,
wonderful. And then tensor b can be equal to a torch tensor
of where we're going to go for this one. Let's go seven, 10, this will be a little
bit confusing
this one, but then we'll go eight, 11, and this will go up to 12, nine, 12. So it's
the same
sort of sequence as what's going on here, but they've been swapped around. So we've
got the
vertical axis here, instead of one, two, three, four, this is just seven, eight,
nine, 10, 11, 12.
But let's now try and perform a matrix multiplication. How do we do that?
Torch dot mat mall for matrix multiplication. PS torch also has torch dot mm, which
stands
for matrix multiplication, which is a short version. So I'll just write down here
so that you know
```

Section 28: Main Topics

Key Topics:

- This is literally how common matrix multiplications are in PyTorch is that they've made torch dot mm as an alias for mat mall
- But right now we've done it with pytorch code, which might be a little confusing
- But the reason why we're spending so much time on this is because as you'll see, as you get more and more into neural networks and deep learning, the matrix multiplication operation is one of the most or if not the most common

▶ 📄 Click to view detailed content

```
tensor a, tensor b. I'm going to write torch dot mm is the same as torch dot mat
mall. It's an alias
for writing less code. This is literally how common matrix multiplications are in
PyTorch
is that they've made torch dot mm as an alias for mat mall. So you have to type
four less characters
using torch dot mm instead of mat mall. But I like to write mat mall because it's a
```

little bit
like it explains what it does a little bit more than mm. So what do you think's going to happen
here? It's okay if you're not sure. But what you could probably do to find out is check the
shapes of these. Does this operation matrix multiplication satisfy the rules that we just
discussed? Especially this one. This is the main one. The inner dimensions must match.
Well, let's have a look, hey? Oh, no, mat one and mat two shapes cannot be multiplied.
Three by two and three by two. This is very similar to what we went through in the last video.
But now we've got some actual numbers there. Let's check the shape.
Oh, torch size three two. Torch size three two now. In the last video we created a random tensor
and we could adjust the shape on the fly. But these tensors already exist. How might we adjust
the shape of these? Well, now I'm going to introduce you to another very common operation or tensor
manipulation that you'll see. And that is the transpose. To fix our tensor shape issues,
we can manipulate the shape of one of our tensors using a transpose. And so, all right here,
we're going to see this anyway, but I'm going to define it in words. A transpose switches the
axes or dimensions of a given tensor. So let's see this in action. If we go, and the way to do it,
is you can go tensor b dot t. Let's see what happens. Let's look at the original tensor b as well.
So dot t stands for transpose. And that's a little bit hard to read, so we might do these on
different lines, tensor b. We'll get rid of that. So you see what's happened here. Instead of
tensor b, this is the original one. We might put the original on top. Instead of the original one
having seven, eight, nine, 10, 11, 12 down the vertical, the transpose has transposed it to seven,
eight, nine across the horizontal and 10, 11, 12 down here. Now, if we get the shape of this,
tensor b dot shape, let's have a look at that. Let's have a look at the original shape, tensor b dot
shape. What's happened? Oh, no, we've still got three, two. Oh, that's what I've missed out here.
I've got a typo. Excuse me. I thought I was, you think code that you've written is working,
but then you realize you've got something as small as just a dot t missing, and it throws off your
whole train of thought. So you're seeing these arrows on the fly here. Now, tensor b is this,
but its shape is torch dot size three, two. And if we try to matrix multiply three, two, and three,
two, tensor a and tensor b, we get an error. Why? Because the inner dimensions do not match.
But if we perform a transpose on tensor b, we switch the dimensions around. So now,
we perform a transpose with tensor b dot t, t's for transpose. We have, this is the important

```
point as well. We still have the same elements. It's just that they've been
rearranged. They've
been transposed. So now, tensor b still has the same information encoded, but
rearranged.
So now we have torch size two, three. And so when we try to matrix multiply these,
we satisfy the first criteria. And now look at the output of the matrix
multiplication of tensor a
and tensor b dot t transposed is three, three. And that is because of the second
rule of matrix
multiplication. The resulting matrix has the shape of the outer dimensions. So
we've got three,
two matrix multiply two, three results in a shape of three, three. So let's predify
some of this,
and we'll print out what's going on here. Just so we know, we can step through it,
because right now we've just got codal over the place a bit. Let's see here, the
matrix
multiplication operation works when tensor b is transposed. And in a second, I'm
going to
show you what this looks like visually. But right now we've done it with pytorch
code,
which might be a little confusing. And that's perfectly fine. Matrix multiplication
takes a
little while and a little practice. So original shapes is going to be tensor a dot
shape. Let's
see what this is. And tensor b equals tensor b dot shape. But the reason why we're
spending so
much time on this is because as you'll see, as you get more and more into neural
networks and
deep learning, the matrix multiplication operation is one of the most or if not the
most common.
Same shape as above, because we haven't changed tensor a shape, we've only changed
tensor b shape,
or we've transposed it. And then in tensor b dot transpose equals, we want tensor b
dot
t dot shape. Wonderful. And then if we print, let's just print out, oops,
print, I spelled the wrong word there, print. We want, what are we multiplying
here? This is
one of the ways, remember our motto of visualize, visualize, visualize, well, this
is how I visualize,
visualize, visualize things, shape, let's do the at symbol for brevity, tensor, and
let's get b dot
t dot shape. We'll put down our little rule here, inner dimensions must match. And
then print,
```

Section 29: Main Topics

Key Topics:

- Now, of course, you could rearrange this maybe transpose tensor a instead of tensor b, have a play around with it

- And then if we click multiply, this is what's happening behind the scenes with our pytorch code of matmore
- So let's have a look at some few PyTorch methods that are in built to do all of these

▶ 📄 Click to view detailed content

```
let's get the output output, I'll put that on a new line. The output is going to
equal
torch dot, or our outputs already here, but we're going to rewrite it for a little
bit of practice,
tensor a, tensor b dot t. And then we can go print output. And then finally, print,
let's get it on a
new line as well, the output shape, a fair bit going on here. But we're going to
step through it,
and it's going to help us understand a little bit about what's going on. That's the
data visualizes
motto. There we go. Okay, so the original shapes are what torch size three two, and
torch size three
two, the new shapes tensor a stays the same, we haven't changed tensor a, and then
we have tensor
b dot t is torch size two three, then we multiply a three by two by a two by three.
So the inner
dimensions must match, which is correct, they do match two and two. Then we have an
output of tensor
at 27, 30, 33, 61, 68, 75, etc. And the output shape is what the output shape is
the outer
dimensions three three. Now, of course, you could rearrange this maybe transpose
tensor a instead of
tensor b, have a play around with it. See if you can create some more errors trying
to multiply these
two, and see what happens if you transpose tensor a instead of tensor b, that's my
challenge. But
before we finish this video, how about we just recreate what we've done here with
this cool website
matrix multiplication. So what did we have? We had tensor a, which is one to six,
let's recreate
this, remove that, this is going to be one, two, three, four, five, six, and then
we want to increase
this, and this is going to be seven, eight, nine, 10, 11, 12. Is that the right way
of doing things?
So this is already transposed, just to let you know. So this is the equivalent of
tensor b
on the right here, tensor b dot t. So let me just show you, if we go tensor b dot
transpose,
which original version was that, but we're just passing in the transpose version to
our matrix
multiplication website. And then if we click multiply, this is what's happening
behind the
scenes with our pytorch code of matmore. We have one times seven plus two times 10.
Did you see
that little flippy thing that it did? That's where the 27 comes from. And then if
we come down here,
what's our first element? 27 when we matrix multiply them. Then if we do the same
```

thing,
the next step, we get 30 and 61, from a combination of these numbers, do it again,
33, 68, 95, from a combination of these numbers, again, and again, and finally we end up with
exactly what we have here. So that's a little bit of practice for you to go through is to create
some of your own tensors can be almost whatever you want. And then try to matrix multiply them
with different shapes. See what happens when you transpose and what different values you get.
And if you'd like to visualize it, you could write out something like this. That really
helps me understand matrix multiplication. And then if you really want to visualize it,
you can go through this website and recreate your target tensors in something like this.
I'm not sure how long you can go. But yeah, that should be enough to get started.
So give that a try and I'll see you in the next video.
Welcome back. In the last few videos, we've covered one of the most fundamental operations
in neural networks. And that is matrix multiplication. But now it's time to move on.
And let's cover tensor aggregation. And what I mean by that is finding the min, max, mean,
sum, et cetera, tensor aggregation of certain tensor values. So for whatever reason, you may
want to find the minimum value of a tensor, the maximum value, the mean, the sum, what's going on
there. So let's have a look at some few PyTorch methods that are in built to do all of these.
And again, if you're finding one of these values, it's called tensor aggregation because you're
going from what's typically a large amount of numbers to a small amount of numbers. So the min
of this tensor would be 27. So you're turning it from nine elements to one element, hence
aggregation. So let's create a tensor, create a tensor, x equals torch dot, let's use a range.
We'll create maybe a zero to 100 with a step of 10. Sounds good to me. And we can find the min
by going, can we do torch dot min? Maybe we can. Or we could also go
x dot min.
And then we can do the same, find the max torch dot max and x dot max. Now how do you think we
might get the average? So let's try it out. Or find the mean, find the mean torch dot mean
x. Oops, we don't have an x. Is this going to work? What's happened? Mean input data type
should be either floating point or complex D types got long instead. Ha ha. Finally,
I knew the error would show its face eventually. Remember how I said it right up here that
we've covered a fair bit already. But right up here, some of the most common errors that
you're going to run into is tensor is not the right data type, not the right shape. We've seen
that with matrix multiplication, not the right device. We haven't seen that yet.

```
But not the
right data type. This is one of those times. So it turns out that the tensor that
we created,
x is of the data type, x dot D type.
```

Section 30: Main Topics

Key Topics:

- So so far, we've seen two of the major errors in PyTorch is data type and shape issues
- And we also ran into one of the most common issues in pie torch and deep learning and neural networks in general

▶ 📄 Click to view detailed content

```
In 64, which is long. So if we go to, let's look up torch tensor.
This is where they're getting long from. We've seen long before is N64. Where's
that or long?
Yeah. So long tenter. That's what it's saying. And it turns out that the torch mean
function
can't work on tensors with data type long. So what can we do here? Well, we can
change
the data type of x. So let's go torch mean x type and change it to float 32. Or
before we do that,
if we go to torch dot mean, is this going to tell us that it needs a D type? Oh, D
type.
One option on the desired data type. Does it have float 32? It doesn't tell us. Ah,
so this is
another one of those little hidden things that you're going to come across. And you
only really
come across this by writing code is that sometimes the documentation doesn't really
tell you explicitly
what D type the input should be, the input tensor. However, we find out that with
this error message
that it should either be a floating point or a complex D type, not along. So we can
convert it
to torch float 32. So all we've done is gone x type as type float 32. Let's see
what happens here.
45 beautiful. And then the same thing, if we went, can we do x dot mean? Is that
going to work as well?
Oh, same thing. So if we go x dot type torch dot float 32, get the mean of that.
There we go.
So that is, I knew it would come up eventually. A beautiful example of finding the
right data
type. Let me just put a note here. Note the torch dot mean function requires a
tensor of float 32.
So so far, we've seen two of the major errors in PyTorch is data type and shape
```

issues. What's
another one that we said? Oh, some. So find the sum. Find the sum we want x dot sum or maybe we
just do torch dot sum first. Keep it in line with what's going on above and x dot sum.
Which one of these should you use like torch dot something x or x dot sum? Personally,
I prefer torch dot max, but you'll also probably see me at points right this. It really depends
on what's going on. I would say pick whichever style you prefer. And because behind the scenes,
they're calling the same methodology. Picture whichever style you prefer and stick with that
throughout your code. For now, let's leave it at that tensor aggregation. There's some
finding min max mean sum. In the next video, we're going to look at finding the positional
min and max, which is also known as arg max and arg min or vice versa. So actually, that's a
little bit of a challenge for the next video is see how you can find out what the positional
min and max is of this. And what I mean by that is which index does the max value occur at and
which index of this tensor does the min occur at? You'll probably want to look into the methods
arg min torch dot arg min for that one and torch dot arg max for that. But we'll cover that in the
next video. I'll see you there. Welcome back. In the last video, we learned all about tensor
aggregation. And we found the min the max the mean and the sum. And we also ran into one of the most
common issues in pie torch and deep learning and neural networks in general. And that was wrong
data types. And so we solved that issue by converting because some functions such as torch dot mean
require a specific type of data type as input. And we created our tensor here, which was of by
default torch in 64. However, torch dot mean requires torch dot float 32. We saw that in an error.
We fix that by changing the type of the inputs. I also issued you the challenge of finding
finding the positional min and max. And you might have found that you can use the
arg min for the minimum. Let's remind ourselves of what x is x. So this means at tensor index of
tensor x. If we find the argument, that is the minimum value, which is zero. So at index zero,
we get the value zero. So that's at zero there. Zero there. This is an index value. So this is
what arg min stands for find the position in tensor that has the minimum value with arg min.
And then returns index position of target tensor
where the minimum value occurs. Now, let's just change x to start from one,
just so there we go. So the arg min is still position zero, position zero. So this is an index
value. And then if we index on x at the zeroth index, we get one. So the minimum value in
x is one. And then the maximum, you might guess, is find the position in tensor

```
that has the maximum
value with arg max. And it's going to be the same thing, except it'll be the
maximum, which is,
which position index nine. So if we go zero, one, two, three, four, five, six,
seven, eight,
nine. And then if we index on x for the ninth element, we get 91 beautiful. Now
these two are
useful for if yes, you want to define the minimum of a tensor, you can just use
min. But if you
sometimes you don't want the actual minimum value, you just want to know where it
appears,
particularly with the arg max value. This is helpful for when we use the soft max
activation
function later on. Now we haven't covered that yet. So I'm not going to allude too
much to it.
```

Section 31: Main Topics

Key Topics:

- So reshaping is we saw before one of the most common errors in machine learning and deep learning is shape mismatches with matrices because they have to satisfy certain rules
- Because again, one of the number one issues in machine learning and deep learning is tensor shape issues

▶ 📄 Click to view detailed content

```
But just remember to find the positional min and max, you can use arg min and arg
max.
So that's all we need to cover with that. Let's keep going in the next video. I'll
see you then.
Welcome back. So we've covered a fair bit of ground. And just to let you know, I
took a little break
after going through all of these. And I'd just like to show you how I get back to
where I'm at,
because if we tried to just write x here and press shift and enter, because our
collab
was disconnected, it's now connecting because as soon as you press any button in
collab, it's
going to reconnect. It's going to try to connect, initialize, and then x is
probably not going to
be stored in memory anymore. So there we go. Name x is not defined. That's because
the collab
state gets reset if you take a break for a couple of hours. This is to ensure
Google can keep
providing resources for free. And it deletes everything to ensure that there's no
compute
```

resources that are being wasted. So to get back to here, I'm just going to go restart and run all.
You don't necessarily have to restart the notebook. You could also go, do we have run all? Yeah,
we could do run before. That'll run every cell before this. We could run after we could run the
selection, which is this cell here. I'm going to click run all, which is just going to go through
every single cell that we've coded above and run them all. However, it will also stop at the errors
where I've left in on purpose. So remember when we ran into a shape error? Well, because this error,
we didn't fix it. I left it there on purpose so that we could keep seeing a shape error. It's
going to stop at this cell. So we're going to have to run every cell after the error cell.
So see how it's going to run these now. They run fine. And then we get right back to where we were,
which was X. So that's just a little tidbit of how I get back into coding. Let's now cover reshaping,
stacking, squeezing, and unsqueezing. You might be thinking, squeezing and unsqueezing. What are
you talking about, Daniel? Well, it's all to do with tenses. And you're like, are we going to
squeeze our tenses? Give them a hug. Are we going to let them go by unsqueezing them?
Well, let's quickly define what these are. So reshaping is we saw before one of the most common
errors in machine learning and deep learning is shape mismatches with matrices because they
have to satisfy certain rules. So reshape reshapes an input tensor to a defined shape.
Now, we're just defining these things in words right now, but we're going to see it in code in
just a minute. There's also view, which is return a view of an input tensor of certain shape,
but keep the same memory as the original tensor. So we'll see what view is in a second.
Reshaping and view are quite similar, but a view always shares the same memory as the original
tensor. It just shows you the same tensor, but from a different perspective, a different shape.
And then we have stacking, which is combine multiple tensors on top of each other. This is a V stack
for vertical stack or side by side. H stack. Let's see what different types of torch stacks there are.
Again, this is how I research different things. If I wanted to learn something new, I would search
torch something stack concatenate a sequence of tensors along a new dimension. Okay. So maybe we
not H stack or V stack, we can just define what dimension we'd like to combine them on.
I wonder if there is a torch V stack. Torch V stack. Oh, there it is. And is there a torch H stack for
horizontal stack? There is a H stack. Beautiful. So we'll focus on just the plain stack. If you
want to have a look at V stack, it'll be quite similar to what we're going to do

```
with stack
and same with H stack. Again, this is just words for now. We're going to see the
code in a minute.
So there's also squeeze, which removes all one dimensions. I'm going to put one in
code,
dimensions from a tensor. We'll see what that looks like. And then there's
unsqueeze,
which adds a one dimension to our target tensor. And then finally, there's permute,
which is return
a view of the input with dimensions permuted. So swapped in a certain way. So a
fair few methods
here. But essentially the crust of all of these, the main point of all of these is
to manipulate
our tensors in some way to change their shape or change their dimension. Because
again, one of the
number one issues in machine learning and deep learning is tensor shape issues. So
let's start
off by creating a tensor and have a look at each of these. Let's create a tensor.
And then we're
going to just import torch. We don't have to, but this will just enable us to run
the notebook
directly from this cell if we wanted to, instead of having to run everything above
here. So let's
create another X torch dot a range because range is deprecated. I'm just going to
add a few code
cells here so that I can scroll and that's in the middle of the screen there.
Beautiful. So let's
just make it between one and 10 nice and simple. And then let's have a look at X
and X dot shape.
What does this give us? Okay, beautiful. So we've got the numbers from one to nine.
Our tensor is
```

Section 32: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
of shape torch size nine. Let's start with reshape. So how about we add an extra
dimension. So then
we have X reshaped equals X dot reshape. Now a key thing to keep in mind about the
reshape
is that the dimensions have to be compatible with the original dimensions. So we're
going to
change the shape of our original tensor with a reshape. And we try to change it
into the shape
one seven. Does that work with the number nine? Well, let's find out, hey, let's
check out X reshaped.
And then we'll look at X reshaped dot shape. What's this going to do? Oh, why do we
get an error there?
```

Well, it's telling us here, this is what pie torch is actually really good at is giving us
errors for what's going wrong. We have one seven is invalid for input size of nine.
Well, why is that? Well, we're trying to squeeze nine elements into a tensor of one
times seven into seven elements. But if we change this to nine, what do we get? Ah, so do you notice
what just happened here? We just added a single dimension. See the single square bracket with
the extra shape here. What if we wanted to add two? Can we do that? No, we can't. Why is that?
Well, because two nine is invalid for input size nine, because two times nine is what?
18. So we're trying to double the amount of elements without having double the amount of elements.
So if we change this back to one, what happens if we change these around nine one? What does this
do? Oh, a little bit different there. So now instead of adding one on the first dimension or
the zeroth dimension, because Python is zero indexed, we added it on the first dimension,
which is giving us a square bracket here if we go back. So we add it to the outside here,
because we've put the one there. And then if we wanted to add it on the inside,
we put the one on the outside there. So then we've got the torch size nine one. Now, let's try
change the view, change the view. So just to reiterate, the reshape has to be compatible
with the original size. So how about we change this to one to 10? So we have a size of 10,
and then we can go five, two, what happens there? Oh, it's compatible because five times two equals
10. And then what's another way we could do this? How about we make it up to 12? So we've got 12
elements, and then we can go three, four, a code cells taking a little while run here.
Then we'll go back to nine, just so we've got the original there.
Whoops, they're going to be incompatible. Oh, so this is another thing. This is good.
We're getting some errors on the fly here. Sometimes you'll get saved failed with Google
CoLab, and automatic saving failed. What you can do to fix this is just either keep coding,
keep running some cells, and CoLab will fix itself in the background, or restart the notebook,
close it, and open again. So we've got size nine, or size eight, sorry, incompatible.
But this is good. You're seeing the errors that come up on the fly, rather than me sort of just
telling you what the errors are, you're seeing them as they come up for me. I'm trying to live
code this, and this is what's going to happen when you start to use Google CoLab, and subsequently
other forms of Jupyter Notebooks. But now let's get into the view, so we can go z equals,
let's change the view of x. View will change it to one nine, and then we'll go z, and then z dot shape.
Ah, we get the same thing here. So view is quite similar to reshape. Remember,

```
though, that a
view shares the memory with the original tensor. So z is just a different view of
x. So z shares
the same memory as what x does. So let's exemplify this. So changing z changes x,
because a view of
a tensor shares the same memory as the original input. So let's just change z,
change the first
element by using indexing here. So we're targeting one, we'll set this to equal
five, and then we'll
see what z and x equal. Yeah, so see, we've got z, the first one here, we change
the first element,
the zero element to five. And the same thing happens with x, we change the first
element of z.
So because z is a view of x, the first element of x changes as well. But let's keep
going. How
about we stack some tenses on top of each other? And we'll see what the stack
function does in
torch. So stack tenses on top of each other. And I'll just see if I press command S
to save,
maybe we'll get this fixed. Or maybe it just will fix itself. Oh, notebook is
saved.
Unless you've made some extensive changes that you're worried about losing, you
could just
download this notebook, so file download, and upload it to collab. But usually if
you click yes,
it sort of resolves itself. Yeah, there we go. All changes saved. So that's
beautiful
troubleshooting on the fly. I like that. So x stack, let's stack some tenses
together,
equals torch stack. Let's go x x x, because if we look at what the doc string of
stack is,
will we get this in collab? Or we just go to the documentations? Yeah. So list, it
takes a list of
tenses and concatenates a sequence of tenses along a new dimension. And we define
the dimension,
the dimension by default is zero. That's a little bit hard to read for me. So
tenses,
```

Section 33: Main Topics

Key Topics:

- I mean, I still do it a lot of the time, even though I've written thousands of lines of machine learning code

▶ 📄 Click to view detailed content

```
dim equals zero. If we come into here, the default dimension is zero. Let's see
what happens when
```

we play around with the dimension here. So we've got four x's. And the first one, we'll just do it
by default, x stack. Okay, wonderful. So they're stacked vertically. Let's see what happens if we
change this to one. Oh, they rearranged a little and stack like that. What happens if we change it
to two? Does it have a dimension to? Oh, we can't do that. Well, that's because the original shape
of x is incompatible with using dimension two. So the only real way to get used to what happens
here by stacking them on top of each other is to play around with the different values for the
dimension. So dim zero, dim one, they look a little bit different there. Now they're on top of each
other. And so the first zero index is now the zeroth tensor. And then same with two being there,
three and so on. But we'll leave it at the default. And there's also v stack and h stack. I'll leave
that to you to to practice those. But I think from memory v stack is using dimension equals zero.
Or h stack is like using dimension equals one. I may have those back the front. You can correct me
if I'm wrong there. Now let's move on. We're going to now have a look at squeeze and unsqueeze.
So actually, I'm going to get you to practice this. So see if you can look up torch squeeze
and torch unsqueeze. And see if you can try them out. We've created a tensor here. We've used
reshape and view and we've used stack. The usage of squeeze and unsqueeze is quite similar. So give
that a go. And to prevent this video from getting too long, we'll do them together in the next video.
Welcome back. In the last video, I issued the challenge of trying out torch dot squeeze,
which removes all single dimensions from a target tensor. And how would you try that out? Well,
here's what I would have done. I'd go to torch dot squeeze and see what happens. Open up the
documentation. Squeeze input dimension returns a tensor with all the dimensions of input size
one removed. And does it have some demonstrations? Yes, it does. Wow. Okay. So you could copy this in
straight into a notebook, copy it here. But what I'd actually encourage you to do quite often is
if you're looking up a new torch method you haven't used, code all of the example by hand. And then
just practice what the inputs and outputs look like. So x is the input here. Check the size of x,
squeeze x, well, set the squeeze of x to y, check the size of y. So let's replicate something
similar to this. We'll go into here, we'll look at x reshaped and we'll remind ourselves of x reshaped
dot shape. And then how about we see what x reshaped dot squeeze looks like. Okay. What happened here?
Well, we started with two square brackets. And we started with a shape of one nine
and removes all single dimensions from a target tensor. And now if we call the squeeze method on

```
x reshaped, we only have one square bracket here. So what do you think the shape of
x reshaped dot
squeeze is going to be? We'll check the shape here. It's just nine. So that's the
squeeze method,
removes all single dimensions. If we had one one nine, it would remove all of the
ones. So it would
just end up being nine as well. Now, let's write some print statements so we can
have a little
pretty output. So previous tensor, this is what I like to do. This is a form of
visualize, visualize,
visualize. If I'm trying to get my head around something, I print out each
successive change
to see what's happening. That way, I can go, Oh, okay. So that's what it was there.
And then I
called that line of code there. Yes, it's a bit tedious. But you do this half a
dozen times, a
fair few times. I mean, I still do it a lot of the time, even though I've written
thousands of lines
of machine learning code. But it starts to become instinct after a while, you start
to go, Oh, okay,
I've got a dimension mismatch on my tensors. So I need to squeeze them before I put
them into a
certain function. For a little while, but with practice, just like riding a bike,
right? But that
try saying is like when you first start, you're all wobbly all over the place
having to look up
the documentation, not that there's much documentation for riding a bike, you just
kind of keep trying.
But that's the style of coding. I'd like you to adopt is to just try it first. Then
if you're stuck,
go to the documentation, look something up, print it out like this, what we're
doing,
quite cumbersome. But this is going to give us a good explanation for what's
happening. Here's our
previous tensor x reshaped. And then if we look at the shape of x reshaped, it's
one nine. And then
if we call the squeeze method, which removes all single dimensions from a target
tensor,
we have the new tensor, which is has one square bracket removed. And the new shape
is all single
dimensions removed. So it's still the original values, but just a different
dimension. Now,
let's do the same as what we've done here with unsqueeze. So we've given our
tensors a hug and
squeezed out all the single dimensions of them. Now we're going to unsqueeze them.
We're going to
take a step back and let them grow a bit. So torch unsqueeze adds a single
dimension
```

Section 34: Main Topics

Key Topics:

- Now that's another thing to note in PyTorch whenever it says dim, that's dimension as in this is a zeroth dimension, first dimension
- Remember, much of, and I'm going to spell color Australian style, much of deep learning is turning your data into numerical representations

▶ 📄 Click to view detailed content

```
to a target tensor at a specific dim dimension. Now that's another thing to note in
PyTorch whenever
it says dim, that's dimension as in this is a zeroth dimension, first dimension.
And if there
was more here, we'd go two, three, four, five, six, et cetera. Because why tensors
can have
unlimited dimensions. So let's go previous target can be excused. So we'll get this
squeezed version
of our tensor, which is x squeezed up here. And then we'll go print. The previous
shape
is going to be x squeezed dot shape. And then we're going to add an extra dimension
with unsqueeze.
There we go, x unsqueezed equals x squeezed. So our tensor before that we remove
the single
dimension. And we're going to put in unsqueeze, dim, we'll do it on the zeroth
dimension. And I
want you to have a think about what this is going to output even before we run the
code.
Just think about, because we've added an extra dimension on the zeroth dimension,
what's the new shape of the unsqueeze tensor going to be? So we're going to go x
unsqueezed.
And then we're going to go print, we'll get our new tensor shape, which is going to
be x unsqueezed
dot shape. All right, let's have a look. There we go. So there's our previous
tensor,
which is the squeezed version, just as a single dimension here. And then we have
our new tensor,
which with the unsqueeze method on dimension zero, we've added a square bracket on
the zeroth
dimension, which is this one here. Now what do you think's going to happen if I
change this to one?
Where's the single dimension going to be added? Let's have a look. Ah, so instead
of adding the
single dimension on the zeroth dimension, we've added it on the first dimension
here. It's quite
confusing because Python is zero index. So I kind of want to my brain's telling me
to say first,
but it's really the zeroth index here or the zeroth dimension. Now let's change
this back to
zero. But that's just another way of exploring things. Every time there's like a
parameter that
we have here, dim equals something like that could be shape, could be size,
whatever, try
changing the values. That's what I'd encourage you to do. And even write some print
code like
we've done here. Now there's one more we want to try out. And that's permute. So
torch dot permute
```

rearranges the dimensions of a target tensor in a specified order. So if we wanted to check out,
let's get rid of some of these extra tabs. Torch dot permute. Let's have a look. This one took me
a little bit of practice to get used to. Because again, working with zeroth dimensions, even though
it seems like the first one. So returns a view. Okay. So we know that a view shares the memory of
the original input tensor with its dimensions permuted. So permuted for me, I didn't really know
what that word meant. I just have mapped in my own memory that permute means rearrange dimensions.
So the example here is we start with a random tensor, we check the size, and then we'd have
torch permute. We're going to swap the order of the dimensions. So the second dimension is first,
the zeroth dimension is in the middle, and the first dimension is here. So these are dimension
values. So if we have torch random two, three, five, two, zero, one has changed this one to be
over here. And then zero, one is two, three, and now two, three there. So let's try something similar
to this. So one of the common places you'll be using permute, or you might see permute being
used is with images. So there's a data specific data format. We've kind of seen a little bit
before, not too much. Original equals torch dot rand size equals. So an image tensor,
we go height width color channels on the end. So I'll just write this down. So this is height
width color channels. Remember, much of, and I'm going to spell color Australian style,
much of deep learning is turning your data into numerical representations. And this is quite common
numerical representation of image data. You have a tensor dimension for the height, a tensor dimension
for the width, and a tensor dimension for the color channels, which is red, green, and blue,
because a certain number of red, green, and blue creates almost any color. Now, if we want to
permute this, so permute the original tensor to rearrange the axis or dimension, axis or dimension,
are kind of used in the same light for tensors or dim order. So let's switch the color channels
to be the first or the zeroth dimension. So instead of height width color channels,
it'll be color channels height width. How would we do that with permute? Let's give it a shot.
X permuted equals X original dot permute. And we're going to take the second dimension,
because this takes a series of dims here. So the second dimension is color channels. Remember,
zero, one, two. So two, we want two first, then we want the height, which is a zero. And then we
want the width, which is one. And now let's do this shifts, axis, zero to one, one to two,
and two to zero. So this is the order as well. This two maps to zero. This zero maps to the first

```
index. This one maps to this index. But that's enough talk about it. Let's see what
it looks like.
So print, previous shape, X original dot shape. And then we go here, print new
shape. This will
be the permuted version. We want X permuted dot shape. Let's see what this looks
like. Wonderful.
```

Section 35: Main Topics

Key Topics:

- They're helping us fix shape and dimension issues with our tensors, which is one of the most common issues in deep learning and neural networks
- And a view in PyTorch shares memory with that original tensor
- So if you've ever done indexing, indexing, with PyTorch is similar to indexing with NumPy

▶ 📄 Click to view detailed content

```
That's exactly what we wanted. So you see, let's just write a little note here. Now
this is
color channels, height, width. So the same data is going to be in both of these
tenses. So X
original X permuted, it's just viewed from a different point of view. Because
remember, a
permute is a view. And what did we discuss? A view shares the same memory as the
original tensor.
So X permuted will share the same place in memory as X original, even though it's
from a different
shape. So a little challenge before we move on to the next video for you, or before
you move
on to the next video, try change one of the values in X original. Have a look at X
original.
And see if that same value, it could be, let's get one of this zero, zero, get all
of the dimensions
here, zero. See what that is? Or can we get a single value maybe? Oops. Oh, no,
we'll need a zero
here, getting some practice on indexing here. Oh, zero, zero, zero. There we go.
Okay, so maybe
we set that to some value, whatever you choose, and see if that changes in X
permuted. So give
that a shot, and I'll see you in the next video. Welcome back. In the last video,
we covered
squeezing, unsqueezing, and permuting, which I'm not going to lie, these concepts
are quite a
lot to take in, but just so you're aware of them. Remember, what are they working
towards? They're
helping us fix shape and dimension issues with our tensors, which is one of the
```

```
most common
issues in deep learning and neural networks. And I usually do the little challenge
of changing a
value of X original to highlight the fact that permute returns a different view of
the original
tensor. And a view in PyTorch shares memory with that original tensor. So if we
change the value
at zero, zero, zero of X original to, in my case, 728218, it happens the same value
gets copied across
to X permuted. So with that being said, we looked at selecting data from tensors
here, and this is
using a technique called indexing. So let's just rehash that, because this is
another thing that
can be a little bit of a hurdle when first working with multi dimensional tensors.
So let's see how
we can select data from tensors with indexing. So if you've ever done indexing,
indexing,
with PyTorch is similar to indexing with NumPy. If you've ever worked with NumPy,
and you've done indexing, selecting data from arrays, NumPy uses an array as its
main data type,
PyTorch uses tensors. It's very similar. So let's again start by creating a tensor.
And again, I'm just going to add a few code cells here, so I can make my screen
right in the middle.
Now we're going to import torch. Again, we don't need to import torch all the time,
just so you can run the notebook from here later on. X equals torch dot. Let's
create a range again,
just nice and simple. This is how I like to work out the fundamentals too, is just
create the small
range, reshape it, and the reshape has to be compatible with the original
dimension. So we go
one, three, three, and why is this because torch a range is going to return us nine
values, because
it's from the start here to the end minus one, and then one times three times three
is what is
nine. So let's have a look x x dot shape. Beautiful. So we have one, two, three,
four, five, six,
seven, eight, nine of size one. So we have this is the outer bracket here, which is
going to contain
all of this. And then we have three, which is this one here, one, two, three. And
then we have three,
which is one, two, three. Now let's work with this. Let's index on our new tensor.
So let's see what
happens when we get x zero, this is going to index on the first bracket. So we get
this one here. So
we've indexed on the first dimension here, the zero dimension on this one here,
which is why we get
what's inside here. And then let's try again, let's index on the middle bracket. So
dimension
one. So we got to go x, and then zero, and then zero. Let's see what happens there.
Now is this the
same as going x zero, zero? It is, there we go. So it depends on what you want to
use. Sometimes
I prefer to go like this. So I know that I'm getting the first bracket, and then
the zeroth
version of that first bracket. So then we have these three values here. Now what do
you think
what's going to happen if we index on third dimension or the second dimension here?
```

```
Well,
let's find out. So let's index on the most in our bracket, which is last dimension.
So we have x zero, zero, zero. What numbers is going to give us back of x zero,
on the zero dimension gives us back this middle tensor. And then if x zero, zero
gives us back
the zeroth index of the middle tensor. If we go x zero, zero, zero is going to give
us the zeroth
tensor, the zeroth index, and the zeroth element. A lot to take in there. But what
we've done is
we've just broken it down step by step. We've got this first zero targets this
outer bracket
and returns us all of this. And then zero, zero targets this first because of this
first zero,
and then the zero here targets this. And then if we go zero, zero, zero, we target
this,
then we target this, and then we get this back because we are getting the zeroth
index here.
```

Section 36: Main Topics

Key Topics:

- So now let's move on to the next part, which is PyTorch tensors and NumPy
- PyTorch actually requires NumPy when you install PyTorch
- And because of this, PyTorch has functionality to interact with it

▶ 📄 Click to view detailed content

```
So if we change this to one, what do we get back? Two. And if we change these all
to one,
what will we get? This is a bit of trivia here, or a challenge. So we're going one,
one, one.
Let's see what happens. Oh, no, did you catch that before I ran the code? I did
that one quite
quickly. We have index one is out of bounds. Why is that? Well, because this
dimension is only one
here. So we can only index on the zero. That's where it gets a little bit confusing
because this
says one, but because it's only got zero dimension, we can only index on the zero
if to mention. But
what if we do 011? What does that give us? Five. Beautiful. So I'd like to issue
you the challenge
of how about getting number nine? How would you get number nine? So rearrange this
code to get
number nine. That's your challenge. Now, I just want to show you as well, is you
can use,
you can also use, you might see this, the semicolon to select all of a target
dimension. So let's say
we wanted to get all of the zeroth dimension, but the zero element from that. We
```

can get 123.
And then let's say we want to say get all values of the zeroth and first dimensions,
but only index one of the second dimension. Oh, that was a mouthful. But get all values of
zeroth and first dimensions, but only index one of second dimension. So let's break this
down step by step. We want all values of zeroth and first dimensions, but only index one of the
second dimension. We press enter, shift enter, 258. So what did we get there? 258. Okay. So we've
got all elements of the zeroth and first dimension, but then so which will return us this thing here.
But then we only want 258, which is the first element here of the second dimension, which is
this three there. So quite confusing. But with some practice, you can figure out how to select
almost any numbers you want from any kind of tensor that you have. So now let's try again,
get all values of the zero dimension, but only the one index value of the first and second
dimension. So what might this look like? Let's break it down again. So we come down here x,
and we're going to go all values of the zero dimension because zero comes first. And then we
want only the one index value of the first and only the one index value of the second.
What is this going to give us five? Oh, we selected the middle tensor. So really,
this line of code is exactly the same as this line of code here, except we've got the square
brackets on the outside here, because we've got this semicolon there. So if we change this to a zero,
we remove that. But because we've got the semicolon there, we've selected all the
dimensions. So we get back the square bracket there, something to keep in mind. Finally,
let's just go one more. So get index zero of zero and first dimension, and all values of second
dimension. So x zero, zero. So zero, the index of zero and first dimension, zero, zero,
and all values of the second dimension. What have we just done here? We've got tensor one,
two, three, lovely. This code again is equivalent to what we've done up here. This has a semicolon
on the end. But what this line explicitly says without the semicolon is, hey, give us all the
values on the remaining dimension there. So my challenge for you is to take this tensor that we
have got here and index on it to return nine. So I'll write down here, index on x to return nine.
So if you have a look at x, as well as index on x to return three, six, nine. So these values
here. So give those both a go and I'll see you in the next video. Welcome back. How'd you go?
Did you give the challenge ago? I finished the last video with issuing the challenge to index on
x to return nine and index on x to return three, six, nine. Now here's what I came up with. Again,

```
there's a few different ways that you could approach both of these. But this is
just what
I've found. So because x is one, three, three of size, well, that's his dimensions.
If we want to
select nine, we need zero, which is this first outer bracket to get all of these
elements. And
then we need two to select this bottom one here. And then we need this final two to
select the
second dimension of this bottom one here. And then for three, six, nine, we need
all of the
elements in the first dimension, all of the in the zeroth dimension, all of the
elements in the
first dimension. And then we get two, which is this three, six, nine set up here.
So that's how I
would practice indexing, start with whatever shape tensor you like, create it
something like this,
and then see how you can write different indexing to select whatever number you
pick.
So now let's move on to the next part, which is PyTorch tensors and NumPy. So NumPy
is a
popular scientific, very popular. PyTorch actually requires NumPy when you install
PyTorch. Popular
scientific Python numerical computing library, that's a bit of a mouthful. And
because of this,
PyTorch has functionality to interact with it. So quite often, you might start off
with,
let's change this into Markdown, you might start off with your data, because it's
numerical format,
you might start off with data in NumPy, NumPy array, want in PyTorch tensor.
Because your
```

Section 37: Main Topics

Key Topics:

- data might be represented by NumPy because it started in NumPy, but say you want to do some deep learning on it and you want to leverage PyTorch's deep learning capabilities, well, you might want to change your data from NumPy to a PyTorch tensor
- And PyTorch has a method to do this, which is torch from NumPy, which will take in an ND array, which is NumPy's main data type, and change it into a torch tensor
- And then if you want to go from PyTorch tensor to NumPy because you want to use some sort of NumPy method, well, the method to do this is torch dot tensor, and you can call dot NumPy on it

▶ 📄 Click to view detailed content

```
data might be represented by NumPy because it started in NumPy, but say you want to
do
some deep learning on it and you want to leverage PyTorch's deep learning
capabilities,
well, you might want to change your data from NumPy to a PyTorch tensor. And
PyTorch has a
method to do this, which is torch from NumPy, which will take in an ND array, which
is NumPy's
main data type, and change it into a torch tensor. We'll see this in a second. And
then if you want
to go from PyTorch tensor to NumPy because you want to use some sort of NumPy
method,
well, the method to do this is torch dot tensor, and you can call dot NumPy on it.
But this is all
just talking about in words, let's see it in action. So NumPy array to tensor.
Let's try this out
first. So we'll import torch so we can run this cell on its own, and then import
NumPy as np,
the common naming convention for NumPy, we're going to create an array in NumPy.
And we're
going to just put one to eight, a range. And then we're going to go tensor equals
torch from NumPy
because we want to go from NumPy array to a torch tensor. So we use from NumPy, and
then we pass
in array, and then we have array and tensor. Wonderful. So there's our NumPy array,
and our torch
tensor with the same data. But what you might notice here is that the D type for
the tensor is
torch dot float 64. Now why is this? It's because NumPy's default data type. Oh, D
type
is float 64. Whereas tensor, what have we discussed before? What's pytorch's
default data type?
float 64. Well, that's not pytorch's default data type. If we were to create torch,
a range,
1.0 to 8.0, by default, pytorch is going to create it in
float 32. So just be aware of that. If you are going from NumPy to pytorch, the
default NumPy
data type is float 64. And pytorch reflects that data type when you use the from
NumPy method.
I wonder if there's a D type. Can we go D type equals torch dot float 32? Takes no
keyword.
Okay. But how could we change the data type here? Well, we could go type torch
float 32.
Yeah, that will give us a tensor D type of float 32 instead of float 64. Beautiful.
I'll just keep
that there so you know, warning when converting from NumPy pytorch, pytorch
reflects NumPy's
default data type of float 64, unless specified. Otherwise, because what have we
discussed,
when you're trying to perform certain calculations, you might run into a data type
issue. So you might
need to convert the type from float 64 to float 32. Now, let's see what happens.
What do you think
will happen if we change the array? We change the value of an array. Well, let's
```

```
find out.
So change the value of array. The question is, what will this do to tensor? Because
we've used
the from NumPy method, do you think if we change the array, the tensor will change?
So let's try
this array equals array plus one. So we're just adding one to every value in the
array. Now,
what is the array and the tensor going to look like? Uh huh. So array, we only
change the first
value there. Oh, sorry, we change every value because we have one to seven. Now
it's two, three,
four, five, six, seven, eight. We change the value from the array. It doesn't
change the
value of the tensor. So that's just something to keep in mind. If you use from
NumPy, we get
a new tensor in memory here. So the original, the new tensor doesn't change if you
change the
original array. So now let's go from tensor to NumPy. If you wanted to go back to
NumPy,
tensor to NumPy array. So we'll start with a tensor. We could use the one we have
right now,
but we're going to create another one, but we'll create one of ones just for fun.
One rhymes with fun. NumPy tensor equals. How do we go to NumPy? Well, we have
torch dot tensor dot NumPy. So we just simply call NumPy on here. And then we have
tensor
and NumPy tensor. What data type do you think the NumPy tensor is going to have?
Because we've returned it to NumPy. Pi torches, default data type is
Flight 32. So if we change that to NumPy, what's going to be the D type of the
NumPy tensor?
NumPy tensor dot D type. It reflects the original D type of what you set the tensor
as. So just
keep that in mind. If you're going between PyTorch and NumPy, default data type of
NumPy is
float 64, whereas the default data type of PyTorch is float 32. So that may cause
some errors if
you're doing different kinds of calculations. Now, what do you think is going to
happen if we
went from our tensor to an array, if we change the tensor, change the tensor, what
happens to
NumPy tensor? So we get tensor equals tensor plus one. And then we go NumPy tensor.
Oh, we'll get tensor as well. So our tensor is now all twos because we added one to
the ones.
But our NumPy tensor remains the same. Remains unchanged. So this means they don't
share memory.
So that's how we go in between PyTorch and NumPy. If you'd like to look up more,
I'd encourage
you to go PyTorch and NumPy. So warm up NumPy, beginner. There's a fair few
tutorials here on
PyTorch because NumPy is so prevalent, they work pretty well together. So have a
look at that.
There's a lot going on there. There's a few more links, I'd encourage you to check
out,
```

Section 38: Main Topics

Key Topics:

- If you'd like to look that up, I'd encourage you to search PyTorch's reproducibility and see what you can find
- But what if you were trying to share this notebook with a friend, so say you went up share and you clicked the share link and you sent that to someone and you're like, hey, try out this machine learning experiment I did
- And pytorch comes the concept of a random seed

▶ 📄 Click to view detailed content

```
but we've covered some of the main ones that you'll see in practice. With that
being said,
let's now jump into the next video where we're going to have a look at the concept
of reproducibility.
If you'd like to look that up, I'd encourage you to search PyTorch's
reproducibility and see
what you can find. Otherwise, I'll see you in the next video. Welcome back. It's
now time for us
to cover the topic of reproducibility. If I could even spell it, that would be
fantastic.
Reproducibility. Trying to take the random out of random. So we've touched upon the
concept of
neural networks harnessing the power of randomness. And what I mean by that is we
haven't actually
built our own neural network yet, but we will be doing that. And we've created
tenses full of random
values. And so in short, how our neural network learns is start with random
numbers, perform tensor
operations, update random numbers to try and make them better representations of
the data. Again,
again, again, again, again. However, if you're trying to do reproducible
experiments, sometimes
you don't want so much randomness. And what I mean by this is if we were creating
random tensors,
from what we've seen so far is that every time we create a random tensor, let's
create one here,
torch dot rand, and we'll create it of three three. Every time we run this cell, it
gives us new numbers.
So 7 7 5 2. There we go. Rand again. Right. So we get a whole bunch of random
numbers here.
Every single time. But what if you were trying to share this notebook with a
friend,
so say you went up share and you clicked the share link and you sent that to
someone and you're like,
hey, try out this machine learning experiment I did. And you wanted a little less
randomness
because neural networks start with random numbers. How might you do that? Well,
```

let's
this write down to reduce the randomness in neural networks. And pytorch comes the concept of a
random seed. So we're going to see this in action. But essentially, let's write this down,
essentially what the random seed does is flavor the randomness. So because of how computers work,
they're actually not true randomness. And actually, there's arguments against this,
and it's quite a big debate in the computer science topic, whatnot, but I am not a computer
scientist, I am a machine learning engineer. So computers are fundamentally deterministic.
It means they run the same steps over and over again. So what the randomness we're doing here
is referred to as pseudo randomness or generated randomness. And the random seed,
which is what you see a lot in machine learning experiments, flavors that randomness. So let's
see it in practice. And at the end of this video, I'll give you two resources that I'd recommend
to learn a little bit more about the concept of pseudo randomness and reproducibility in pytorch.
Let's start by importing torch so you could start this notebook right from here. Create two random
tensors. We'll just call this random tensor a equals torch dot rand and we'll go three four
and we'll go random tensor b equals torch dot rand same size three four. And then if we have a
look at let's go print random tensor a print random tensor b. And then let's print to see if
they're equal anywhere random tensor a equals equals equals random tensor b. Now what do you
think this is going to do? If we have a look at one equals one, what does it return? True.
So this is comparison operator to compare two different tensors. We're creating two random
tensors here. We're going to have a look at them. We'd expect them to be full of random values.
Do you think any of the values in each of these random tensors is going to be equal to each other?
Well, there is a chance that they are, but it's highly unlikely. I'll be quite surprised if they are.
Oh, again, my connection might be a little bit. Oh, there we go. Beautiful. So we have tensor a
tensor of three four with random numbers. And we have tensor b of three four with random numbers.
So if we were, if I was to share this notebook with my friend or my colleague or even you,
if you ran this cell, you are going to get random numbers as well. And you have every chance of
replicating one of these numbers. But again, it's highly unlikely. So again, I'm getting that
automatic save failed. You might get that if your internet connection is dropping out, maybe that's
something going on with my internet connection. But again, as we've seen, usually this resolves
itself. If you try a few times, I'll just keep coding. If it really doesn't resolve itself,

```
you can go file is a download notebook or save a copy and drive download. You can
download the
notebook, save it to your local machine, re upload it to upload notebook and start
again in another
Google Colab instance. But there we go. It fixed itself. Wonderful troubleshooting
on the fly.
So the way we make these reproducible is through the concept of a random seed. So
let's have a
look at that. Let's make some random, but reproducible tenses. So import torch. And
we're going to
set the random seed by going torch dot manual seed random. Oh, we don't have random
set yet.
I'm going to set my random seed. You set the random seed to some numerical value.
42 is a common
one. You might see zero. You might see one, two, three, four. Essentially, you can
set it to whatever
you want. And each of these, you can think of 77, 100, as different flavors of
randomness. So
```

Section 39: Main Topics

Key Topics:

- We're telling pytorch a flavor our randomness with 42 torch manual seed
- So this is your extra curriculum for this, even if you don't understand what's going on in a lot of the code here, just be aware of reproducibility, because it's an important topic in machine learning and deep learning
- You can complete this whole course on the free tier as well

▶ 📄 Click to view detailed content

```
I like to use 42, because it's the answer to the universe. And then we go random
seed. And now
let's create some random tenses. Random tensor C with the flavor of our random
seed. Three,
four. And then we're going to go torch tensor D equals torch dot rand three, four.
Now, let's
see what happens. We'll print out random tensor C. And we'll print out random
tensor D. And then
we'll print out to see if they're equal anywhere. Random tensor C equals random
tensor D. So let's
find out what happens. Huh, what gives? Well, we've got randomness. We set the
random seed. We're
telling pytorch a flavor our randomness with 42 torch manual seed. Hmm, let's try
set the manual
seed each time we call a random method. We go there. Ah, much better. So now we've
got some
flavored randomness. So a thing to keep in mind is that if you want to use the
```

torch manual seed,
generally it only works for one block of code if you're using a notebook. So that's just
something to keep in mind. If you're creating random tensors, one after the other, we're using
assignment like this, you should use torch dot manual seed every time you want to call the rand
method or some sort of randomness. However, if we're using other torch processes, usually what
you might see is torch manual seed is set right at the start of a cell. And then a whole bunch
of code is done down here. But because we're calling subsequent methods here, we have to reset
the random seed. Otherwise, if we don't do this, we comment this line, it's going to flavor the
randomness of torch random tensor C with torch manual seed. But then random tensor D is just
going to have no flavor. It's not going to use a random seed. So we reset it there. Wonderful.
So I wonder, does this have a seed method? Let's go torch dot rand. Does this have seed?
Sometimes they have a seed method. Seed, no, it doesn't. Okay, that's all right.
The more you learn, but there's documentation for torch dot rand. And I said that I was going to
link at the end of this video. So the manual seed is a way to, or the random seed, but in
torch, it's called a manual seed is a way to flavor the randomness. So these numbers, as you see,
are still quite random. But the random seed just makes them reproducible. So if I was to share this
with you, if you had to run this block of code, ideally, you're going to get the same numerical
output here. So with that being said, I'd like to refer to you to the pie torch reproducibility
document, because we've only quite scratched the surface of this of reproducibility. We've covered
one of the main ones. But this is a great document on how to go through reproducibility in pie torch.
So this is your extra curriculum for this, even if you don't understand what's going on in a lot
of the code here, just be aware of reproducibility, because it's an important topic in machine
learning and deep learning. So I'll put this here, extra resources for reproducibility.
As we go pie torch randomness, we'll change this into markdown. And then finally, the concept
of a random seed is Wikipedia random seed. So random seeds quite a universal concept,
not just for pie torch, there's a random seed and NumPy as well. So if you'd like to see what
this means, yeah, initialize a pseudo random number generator. So that's a big word, pseudo random
number generator. But if you'd like to learn about more random number generation computing,
and what a random seed does is I'd refer to you to check out this documentation here.
Whoo, far out, we have covered a lot. But there's a couple more topics you should

```
really be aware
of to finish off the pie torch fundamentals. You got this. I'll see you in the next
video.
Welcome back. Now, let's talk about the important concept of running tenses or pie
torch objects. So running tenses and pie torch objects on GPUs and making faster
computations.
So we've discussed that GPUs, let me just scroll down a little bit here, GPUs equal
faster
computation on numbers. Thanks to CUDA plus NVIDIA hardware plus pie torch working
behind the
scenes to make everything hunky dory. Good. That's what hunky dory means, by the
way,
if you never heard that before. So let's have a look at how we do this. Now, we
first need to
talk about, let's go here one getting a GPU. There's a few different ways we've
seen one before.
Number one easiest is to use what we're using right now. Use Google Colab for a
free GPU.
But there's also Google Colab Pro. And I think there might even be, let's look up
Google Colab
Pro. Choose the best that's right for you. I use Google Colab Pro because I use it
almost every day.
So yeah, I pay for Colab Pro. You can use Colab for free, which is might be what
you're using.
There's also Colab Pro Plus, which has a lot more advantages as well. But Colab Pro
is giving me
faster GPUs, so access to faster GPUs, which means you spend less time waiting
while your code is running.
More memory, longer run time, so it'll last a bit longer if you leave it running
idle.
And then Colab Pro again is a step up from that. I personally haven't had a need
yet to use
Google Colab Pro Plus. You can complete this whole course on the free tier as well.
But as you start
to code more, as you start to run bigger models, as you start to want to compute
more, you might
```

Section 40: Main Topics

Key Topics:

- So one of my favorite posts for getting a GPU is, yeah, the best GPUs for deep learning in 2020, or something like this
- Deep learning
- This is, yeah, which GPUs to get for deep learning

▶ 📄 Click to view detailed content

want to look into something like Google Colab Pro. Or let's go here. Options to upgrade as well.
And then another way is use your own GPU. Now this takes a little bit of setup and requires
the investment of purchasing a GPU. There's lots of options. So one of my favorite posts for
getting a GPU is, yeah, the best GPUs for deep learning in 2020, or something like this.
What do we got? Deep learning? Tim Detmos. This is, yeah, which GPUs to get for deep learning?
Now, I believe at the time of this video, I think it's been updated since this date. So don't take
my word for it. But this is a fantastic blog post for figuring out what GPUs see this post
for what option to get. And then number three is use cloud computing. So such as
GCP, which is Google Cloud Platform AWS, which is Amazon Web Services or Azure.
These services, which is Azure is by Microsoft, allow you to rent computers on the cloud and access
them. So the first option using Google Colab, which is what we're using is by far the easiest
and free. So there's big advantages there. However, the downside is that you have to use a website
here, Google Colab, you can't run it locally. You don't get the benefit of using cloud computing,
but my personal workflow is I run basically all of my small scale experiments and things like
learning new stuff in Google Colab. And then if I want to upgrade things, run video experiments,
I have my own dedicated deep learning PC, which I have built with a big powerful GPU. And then
also I use cloud computing if necessary. So that's my workflow. Start with Google Colab.
And then these two, if I need to do some larger experiments. But because this is the beginning
of course, we can just stick with Google Colab for the time being. But I thought I'd make you aware
of these other two options. And if you'd like to set up a GPU, so four, two, three, PyTorch plus
GPU drivers, which is CUDA takes a little bit of setting up to do this, refer to PyTorch
setup documentation. So if we go to pytorch.org, they have some great setup guides here,
get started. And we have start locally. This is if you want to run on your local machine,
such as a Linux setup. This is what I have Linux CUDA 11.3. It's going to give you a
conda install command to use conda. And then if you want to use cloud partners, which is Alibaba
Cloud, Amazon Web Services, Google Cloud Platform, this is where you'll want to go. So I'll just link
this in here. But for this course, we're going to be focusing on using Google Colab. So now,
let's see how we might get a GPU in Google Colab. And we've already covered this, but I'm going to
recover it just so you know. We're going to change the runtime type. You can go in any notebook and

```
do this, runtime type, hardware accelerator, we can select GPU, click save. Now
this is going to
restart our runtime and connect us to our runtime, aka a Google compute instance
with a GPU. And so
now if we run NVIDIA SMI, I have a Tesla P100 GPU. So let's look at this Tesla P100
GPU. Do we have an image? Yeah, so this is the GPU that I've got running, not the
Tesla car,
the GPU. So this is quite a powerful GPU. That is because I have upgraded to Colab
Pro. Now,
if you're not using Colab Pro, you might get something like a Tesla K80, which is a
slightly
less powerful GPU than a Tesla P100, but still a GPU nonetheless and will still
work faster than
just running PyTorch code on the pure CPU, which is the default in Google Colab and
the default
in PyTorch. And so now we can also check to see if we have GPU access with PyTorch.
So let's go
here. This is number two now. Check for GPU access with PyTorch. So this is a
little command that's
going to allow us or tell us if PyTorch, just having the GPU here, this is by the
way, another
thing that Colab has a good setup with, is that all the connections between PyTorch
and the NVIDIA
GPU are set up for us. Whereas when you set it up on your own GPU or using cloud
computing,
there are a few steps you have to go through, which we're not going to cover in
this course.
I'd highly recommend you go through the getting started locally set up if you want
to do that,
to connect PyTorch to your own GPU. So let's check for the GPU access with PyTorch.
This is another advantage of using Google Colab. Almost zero set up to get started.
So import
torch and then we're going to go torch dot cuda dot is available. And remember,
cuda is
NVIDIA's programming interface that allows us to use GPUs for numerical computing.
There we go,
beautiful. So big advantage of Google Colab is we get access to a free GPU. In my
case, I'm paying
for the faster GPU, but in your case, you're more than welcome to use the free
version.
All that means it'll be slightly slower than a faster GPU here. And we now have
access to GPUs
with PyTorch. So there is one more thing known as device agnostic code. So set up
device agnostic
code. Now, this is an important concept in PyTorch because wherever you run
PyTorch, you might not
always have access to a GPU. But if there was access to a GPU, you'd like it to use
it if it's
available. So one of the ways that this is done in PyTorch is to set the device
variable. Now,
```

Section 41: Main Topics

Key Topics:

- If it's not available, if we don't have access to a GPU that PyTorch can use, just default to the CPU
- But as you upgrade your PyTorch experiments and machine learning experiments, you might have access to more than one GPU
- But final thing before we finish this video is if we go PyTorch device agnostic code, cuda semantics, there's a little section in here called best practices

▶ 📄 Click to view detailed content

```
really, you could set this to any variable you want, but you're going to see it
used as device
quite often. So cuda if torch dot cuda is available. Else CPU. So all this is going
to say, and we'll
see where we use the device variable later on is set the device to use cuda if it's
available. So
it is so true. If it's not available, if we don't have access to a GPU that PyTorch
can use,
just default to the CPU. So with that being said, there's one more thing. You can
also count the
number of GPUs. So this won't really apply to us for now because we're just going
to stick with
using one GPU. But as you upgrade your PyTorch experiments and machine learning
experiments,
you might have access to more than one GPU. So you can also count the devices here.
We have access to one GPU, which is this here. So the reason why you might want to
count the number
of devices is because if you're running huge models on large data sets, you might
want to run one
model on a certain GPU, another model on another GPU, and so on and so on. But
final thing before
we finish this video is if we go PyTorch device agnostic code, cuda semantics,
there's a little
section in here called best practices. This is basically what we just covered there
is setting
the device argument. Now this is using the arg pass, but so yeah, there we go.
args.device,
torch.device, cuda, args.device, torch.device, CPU. So this is one way to set it
from the Python
arguments when you're running scripts, but we're using the version of running it
through a notebook.
So check this out. I'll just link this here, device agnostic code. It's okay if
you're not sure
of what's going on here. We're going to cover it a little bit more later on
throughout the course,
but right here for PyTorch, since it's capable of running compute on the GPU or
CPU,
it's best practice to set up device agnostic code, e.g. run on GPU if available,
else default to CPU. So check out the best practices for using cuda, which is
namely setting up
device agnostic code. And let's in the next video, see what I mean about setting
```

our PyTorch tensors
and objects to the target device. Welcome back. In the last video, we checked out a few different
options for getting a GPU, and then getting PyTorch to run on the GPU. And for now we're using
Google Colab, which is the easiest way to get set up because it gives us free access to a GPU,
faster ones if you set up with Colab Pro, and it comes with PyTorch automatically set up to
use the GPU if it's available. So now let's see how we can actually use the GPU. So to do so,
we'll look at putting tensors and models on the GPU. So the reason we want our tensors slash models
on the GPU is because using GPU results in faster computations. And if we're getting our machine
learning models to find patterns and numbers, GPUs are great at doing numerical calculations.
And the numerical calculations we're going to be doing are tensor operations like we saw above.
So the tensor operations, well, we've covered a lot. Somewhere here, tensor operations,
there we go, manipulating tensor operations. So if we can run these computations faster,
we can discover patterns in our data faster, we can do more experiments, and we can work towards
finding the best possible model for whatever problem that we're working on. So let's see,
we'll create a tensor, as usual, create a tensor. Now the default is on the CPU.
So tensor equals torch dot tensor. And we'll just make it a nice simple one, one, two, three.
And let's write here, tensor not on GPU will print out tensor. And this is where we can use,
we saw this parameter before device. Can we pass it in here? Device equals CPU.
Let's see what this comes out with. There we go. So if we print it out, tensor 123 is on the CPU.
But even if we got rid of that device parameter, by default, it's going to be on the CPU. Wonderful.
So now PyTorch makes it quite easy to move things to, and I'm saying to for a reason,
to the GPU, or to, even better, the target device. So if the GPU is available, we use CUDA.
If it's not, it uses CPU. This is why we set up the device variable. So let's see, move tensor to GPU. If available,
tensor on GPU equals tensor dot two device. Now let's have a look at this, tensor on GPU.
So this is going to shift the tensor that we created up here to the target device.
Wonderful. Look at that. So now our tensor 123 is on device CUDA zero. Now this is the index of
the GPU that we're using, because we only have one, it's going to be at index zero. So later on,
when you start to do bigger experiments and work with multiple GPUs, you might have different tensors
that are stored on different GPUs. But for now, we're just sticking with one GPU, keeping it nice
and simple. And so you might have a case where you want to move, oh, actually, the reason why we
set up device agnostic code is again, this code would work if we run this,

```
regardless if we had,
so it won't error out. But regardless if we had a GPU or not, this code will work.
So whatever device
we have access to, whether it's only a CPU or whether it's a GPU, this tensor will
move to whatever
target device. But since we have a GPU available, it goes there. You'll see this a
lot. This two
```

Section 42: Main Topics

Key Topics:

- Remember the top three errors in deep learning or pytorch
- And with pytorch, number three is device issues
- That's a beautiful thing about pytorch is very helpful error messages

▶ 📄 Click to view detailed content

```
method moves tensors and it can be also used for models. We're going to see that
later on. So just
keep two device in mind. And then you might want to, for some computations, such as
using NumPy,
NumPy only works with the CPU. So you might want to move tensors back to the CPU,
moving tensors back
to the CPU. So can you guess how we might do that? It's okay if you don't know. We
haven't covered a
lot of things, but I'm going to challenge you anyway, because that's the fun part
of thinking
about something. So let's see how we can do it. Let's write down if tensor is on
GPU, can't transform
it to NumPy. So let's see what happens if we take our tensor on the GPU and try to
go NumPy.
What happens? Well, we get an error. So this is another huge error. Remember the
top three
errors in deep learning or pytorch? There's lots of them, but number one, shape
errors,
number two, data type issues. And with pytorch, number three is device issues. So
can't convert
CUDA zero device type tensor to NumPy. So NumPy doesn't work with the GPU. Use
tensor dot CPU
to copy the tensor to host memory first. So if we call tensor dot CPU, it's going
to bring our
target tensor back to the CPU. And then we should be able to use it with NumPy. So
to fix the GPU tensor with NumPy issue, we can first set it to the CPU. So tensor
back on CPU
equals tensor on GPU dot CPU. We're just taking what this said here. That's a
beautiful thing
about pytorch is very helpful error messages. And then we're going to go NumPy.
And then if we go tensor back on CPU, is this going to work? Let's have a look. Oh,
```

of course,
it's not because I typed it wrong. And I've typed it again twice. Third time, third time's a charm.
There we go. Okay, so that works because we've put it back to the CPU first before calling NumPy.
And then if we refer back to our tensor on the GPU, because we've reassociated this, again,
we've got typos galore classic, because we've reassigned tensor back on CPU, our tensor on
GPU remains unchanged. So that's the four main things about working with pytorch on the GPU.
There are a few more tidbits such as multiple GPUs, but now you've got the fundamentals. We're
going to stick with using one GPU. And if you'd like to later on once you've learned a bit more
research into multiple GPUs, well, as you might have guessed, pytorch has functionality for that too.
So have a go at getting access to a GPU using colab, check to see if it's available, set up device
agnostic code, create a few dummy tensors and just set them to different devices, see what happens
if you change the device parameter, run a few errors by trying to do some NumPy calculations
with tensors on the GPU, and then bring those tensors on the GPU back to NumPy and see what happens
there. So I think we've covered, I think we've reached the end of the fundamentals. We've covered
a fair bit. Introduction to tensors, the minmax, a whole bunch of stuff inside the introduction
to tensors, finding the positional minmax, reshaping, indexing, working with tensors and NumPy,
reproducibility, using a GPU and moving stuff back to the GPU far out. Now you're probably wondering,
Daniel, we've covered a whole bunch. What should I do to practice all this? Well, I'm glad you asked.
Let's cover that in the next video. Welcome back. And you should be very proud of your
self right now. We've been through a lot, but we've covered a whole bunch of PyTorch fundamentals.
These are going to be the building blocks that we use throughout the rest of the course.
But before moving on to the next section, I'd encourage you to try out what you've learned
through the exercises and extra curriculum. Now, I've set up a few exercises here based off
everything that we've covered. If you go into learn pytorch.io, go to the section that we're
currently on. This is going to be the case for every section, by the way. So just keep this in mind,
is we're working on PyTorch fundamentals. Now, if you go to the PyTorch fundamentals notebook,
this is going to refresh, but that if you scroll down to the table of contents at the bottom of
each one is going to be some exercises and extra curriculum. So these exercises here,
such as documentation reading, because a lot you've seen me refer to the PyTorch documentation

```
for almost everything we've covered a lot, but it's important to become familiar
with that.
So exercise number one is read some of the documentation. Exercise number two is
create a
random tensor with shape, seven, seven. Three, perform a matrix multiplication on
the tensor from two
with another random tensor. So these exercises are all based off what we've covered
here.
So I'd encourage you to reference what we've covered in whichever notebook you
choose,
could be this learn pytorch.io, could be going back through the one we've just
coded together
in the video. So I'm going to link this here, exercises, see exercises for this
notebook here.
So then how should you approach these exercises? So one way would be to just read
them here,
and then in collab we'll go file new notebook, wait for the notebook to load. Then
you could call this
zero zero pytorch exercises or something like that, and then you could start off by
importing
torch, and then away you go. For me, I'd probably set this up on one side of the
screen, this one
up on the other side of the screen, and then I just have the exercises here. So
number one,
```

Section 43: Main Topics

Key Topics:

- As we go throughout the course, these exercises are going to get a little bit more in depth as we've learned more
- This is the home for all of the course materials
- So pytorch fundamentals exercises

▶ 📄 Click to view detailed content

```
I'm not going to really write much code for that, but you could have documentation
reading here.
And then so this encourages you to read through torch.tensor and go through there
for 10 minutes or so. And then for the other ones, we've got create a random tensor
with shape
seven seven. So we just comment that out. So torch, round seven seven, and there we
go.
Some are as easy as that. Some are a little bit more complex. As we go throughout
the course,
these exercises are going to get a little bit more in depth as we've learned more.
But if you'd like an exercise template, you can come back to the GitHub. This is
the home for all
```

of the course materials. You can go into extras and then exercises. I've created templates for
each of the exercises. So pytorch fundamentals exercises. If you open this up, this is a template
for all of the exercises. So you see there, create a random tensor with shape seven seven.
These are all just headings. And if you'd like to open this in CoLab and work on it,
how can you do that? Well, you can copy this link here. Come to Google CoLab. We'll go file,
open notebook, GitHub. You can type in the link there. Click search. What's this going to do?
Boom. Pytorch fundamentals exercises. So now you can go through all of the exercises. This
will be the same for every module on the course and test your knowledge. Now it is open book. You
can use the notebook here, the ones that we've coded together. But I would encourage you to try
to do these things on your own first. If you get stuck, you can always reference back. And then
if you'd like to see an example solutions, you can go back to the extras. There's a solutions folder
as well. And that's where the solutions live. So the fundamental exercise solutions. But again,
I would encourage you to try these out, at least give them a go before having a look at the solutions.
So just keep that in mind at the end of every module, there's exercises and extra curriculum.
The exercises will be code based. The extra curriculum is usually like reading based.
So spend one hour going through the Pytorch basics tutorial. I recommend the quick start
in tensor sections. And then finally to learn more on how a tensor can represent data,
watch the video what's a tensor which we referred to throughout this. But massive effort on finishing
the Pytorch fundamentals section. I'll see you in the next section. Friends, welcome back to
the Pytorch workflow module. Now let's have a look at what we're going to get into.
So this is a Pytorch workflow. And I say a because it's one of many. When you get into
deep learning machine learning, you'll find that there's a fair few ways to do things. But here's
the rough outline of what we're going to do. We're going to get our data ready and turn it into
tensors because remember a tensor can represent almost any kind of data. We're going to pick or
build or pick a pre-trained model. We'll pick a loss function and optimize it. Don't worry if
you don't know what they are. We're going to cover this. We're going to build a training loop,
fit the model to make a prediction. So fit the model to the data that we have. We'll learn how
to evaluate our models. We'll see how we can improve through experimentation and we'll save
and reload our trained model. So if you wanted to export your model from a notebook and use it

```
somewhere else, this is what you want to be doing. And so where can you get help?
Probably the most
important thing is to follow along with the code. We'll be coding all of this
together.
Remember model number one. If and out, run the code. Try it for yourself. That's
how I learn best.
Is I write code? I try it. I get it wrong. I try again and keep going until I get
it right.
Read the doc string because that's going to show you some documentation about the
functions that
we're using. So on a Mac, you can use shift command and space in Google Colab or if
you're on a Windows
PC, it might be control here. If you're still stuck, try searching for it. You'll
probably come
across resources such as stack overflow or the PyTorch documentation. We've already
seen this
a whole bunch and we're probably going to see it a lot more throughout this entire
course actually
because that's going to be the ground truth of everything PyTorch. Try again. And
finally,
if you're still stuck, ask a question. So the best place to ask a question will be
at the PyTorch deep learning slash discussions tab. And then if we go to GitHub,
that's just under here. So Mr. Deeburg PyTorch deep learning. This is all the
course materials.
We see here, this is your ground truth for the entire course. And then if you have
a question,
go to the discussions tab, new discussion, you can ask a question there. And don't
forget to
please put the video and the code that you're trying to run. That way we can
reference
what's going on and help you out there. And also, don't forget, there is the book
version of the
course. So learn pytorch.io. By the time you watch this video, it'll probably have
all the chapters
here. But here's what we're working through. This is what the videos are based on.
All of this,
we're going to go through all of this. How fun is that? But this is just reference
material.
So you can read this at your own time. We're going to focus on coding together. And
speaking of coding.
```

Section 44: Main Topics

Key Topics:

- And now I'm going to title this 01 pytorch workflow
- Because in the course resources, we have the original notebook here, which is what this video notebook is going to be based off

- And then of course, you've got the book version of the notebook as well, which is just a different formatted version of this exact same notebook

▶ 📄 Click to view detailed content

```
Let's code. I'll see you over at Google Colab.
Oh, right. Well, let's get hands on with some code. I'm going to come over to
colab.research.google.com.
You may already have that bookmark. And I'm going to start a new notebook. So we're
going to do
everything from scratch here. We'll let this load up. I'm just going to zoom in a
little bit.
Beautiful. And now I'm going to title this 01 pytorch workflow. And I'm going to
put the video
ending on here so that you know that this notebook's from the video. Why is that?
Because in the
course resources, we have the original notebook here, which is what this video
notebook is going
to be based off. You can refer to this notebook as reference for what we're going
to go through.
It's got a lot of pictures and beautiful text annotations. We're going to be
focused on the
code in the videos. And then of course, you've got the book version of the notebook
as well,
which is just a different formatted version of this exact same notebook. So I'm
going to link
both of these up here. So let's write in here, pytorch workflow. And let's explore
an example,
pytorch end to end workflow. And then I'm going to put the resources. So ground
truth notebook.
We go here. And I'm also going to put the book version.
Book version of notebook. And finally, ask a question, which will be where at the
discussions
page. Then we'll go there. Beautiful. Let's turn this into markdown. So let's get
started. Let's
just jump right in and start what we're covering. So this is the trend I want to
start getting
towards is rather than spending a whole bunch of time going through keynotes and
slides,
I'd rather we just code together. And then we explain different things as they need
to be
explained because that's what you're going to be doing if you end up writing a lot
of pytorch is
you're going to be writing code and then looking things up as you go. So I'll get
out of these
extra tabs. I don't think we need them. Just these two will be the most important.
So what we're
covering, let's create a little dictionary so we can check this if we wanted to
later on.
So referring to our pytorch workflows, at least the example one that we're going to
go through,
which is just here. So we're going to go through all six of these steps, maybe a
little bit of
each one, but just to see it going from this to this, that's what we're really
focused on. And then
```

we're going to go through through rest the course like really dig deep into all of these. So what
we're covering number one is data preparing and loading. Number two is we're going to see how we
can build a machine learning model in pytorch or a deep learning model. And then we're going
to see how we're going to fit our model to the data. So this is called training. So fit is another
word. As I said in machine learning, there's a lot of different names for similar things,
kind of confusing, but you'll pick it up with time. So we're going to once we've trained a model,
we're going to see how we can make predictions and evaluate those predictions,
evaluating a model. If you make predictions, it's often referred to as inference. I typically
say making predictions, but inference is another very common term. And then we're going to look
at how we can save and load a model. And then we're going to put it all together. So a little bit
different from the visual version we have of the pytorch workflow. So if we go back to here,
I might zoom in a little. There we go. So we're going to focus on this one later on,
improve through experimentation. We're just going to focus on the getting data ready,
building a model, fitting the model, evaluating model, save and reload. So we'll see this one more,
like in depth later on, but I'll hint at different things that you can do
for this while we're working through this workflow. And so let's put that in here.
And then if we wanted to refer to this later, we can just go what we're covering.
Oh, this is going to connect, of course. Beautiful. So we can refer to this later on,
if we wanted to. And we're going to start by import torch. We're going to get pytorch ready
to go import nn. So I'll write a note here. And then we haven't seen this one before, but
we're going to see a few things that we haven't seen, but that's okay. We'll explain it as we go.
So nn contains all of pytorch's building blocks for neural networks.
And how would we learn more about torch nn? Well, if we just go torch.nn, here's how I'd
learn about it, pytorch documentation. Beautiful. Look at all these. These are the basic building
blocks for graphs. Now, when you see the word graph, it's referring to a computational graph,
which is in the case of neural networks, let's look up a photo of a neural network.
Images, this is a graph. So if you start from here, you're going to go towards the right.
There's going to be many different pictures. So yeah, this is a good one. Input layer. You have
a hidden layer, hidden layer to output layer. So torch and n comprises of a whole bunch of
different layers. So you can see layers, layers, layers. And each one of these, you can see input
layer, hidden layer one, hidden layer two. So it's our job as data scientists and machine

```
learning engineers to combine these torch dot nn building blocks to build things
such as these.
```

Section 45: Main Topics

Key Topics:

- Now, it might not be exactly like this, but that's the beauty of pytorch is that you can combine these in almost any different way to build any kind of neural network you can imagine
- And let's check our pytorch version
- Pytorch version torch dot version

▶ 📄 Click to view detailed content

```
Now, it might not be exactly like this, but that's the beauty of pytorch is that
you can
combine these in almost any different way to build any kind of neural network you
can imagine.
And so let's keep going. That's torch nn. We're going to get hands on with it,
rather than just talk about it. And we're going to need map plot lib because what's
our other
motto? Our data explorers motto is visualize, visualize, visualize. And let's check
our pytorch
version. Pytorch version torch dot version. So this is just to show you you'll need
at least this version. So 1.10 plus CUDA 111. That means that we've got CU stands
for CUDA.
That means we've got access to CUDA. We don't have a GPU on this runtime yet,
because we haven't gone to GPU. We might do that later.
So if you have a version that's lower than this, say 1.8, 0.0, you'll want pytorch
1.10 at least.
If you have a version higher than this, your code should still work. But that's
about enough
for this video. We've got our workflow ready to set up our notebook, our video
notebook.
We've got the resources. We've got what we're covering. We've got our dependencies.
Let's in the next one get started on one data, preparing and loading.
I'll see you in the next video.
Let's now get on to the first step of our pytorch workflow. And that is data,
preparing and loading.
Now, I want to stress data can be almost anything in machine learning.
I mean, you could have an Excel spreadsheet, which is rows and columns,
nice and formatted data. You could have images of any kind. You could have videos.
I mean,
YouTube has lots of data. You could have audio like songs or podcasts. You could
have even DNA
these days. Patents and DNA are starting to get discovered by machine learning. And
then, of course,
```

you could have text like what we're writing here. And so what we're going to be focusing on
throughout this entire course is the fact that machine learning is a game of two parts.
So one, get data into a numerical representation to build a model to learn patterns in that
numerical representation. Of course, there's more around it. Yes, yes, yes. I understand you can
get as complex as you like, but these are the main two concepts. And machine learning, when I say
machine learning, saying goes for deep learning, you need some kind of, oh, number a call. Number
a call. I like that word, number a call representation. Then you want to build a model to learn patterns
in that numerical representation. And if you want, I've got a nice pretty picture that describes that
machine learning a game of two parts. Let's refer to our data. Remember, data can be almost
anything. These are our inputs. So the first step that we want to do is create some form
of numerical encoding in the form of tenses to represent these inputs, how this looks will be
dependent on the data, depending on the numerical encoding you choose to use. Then we're going to
build some sort of neural network to learn a representation, which is also referred to as
patterns features or weights within that numerical encoding. It's going to output that
representation. And then we want to do something without representation, such as in the case of
this, we're doing image recognition, image classification, is it a photo of Raman or spaghetti?
Is this tweet spam or not spam? Is this audio file saying what it says here? I'm not going to say
this because my audio assistant that's also named to this word here is close by and I don't want it
to go off. So this is our game of two parts. One here is convert our data into a numerical
representation. And two here is build a model or use a pre trained model to find patterns in
that numerical representation. And so we've got a little stationary picture here, turn data into
numbers, part two, build a model to learn patterns in numbers. So with that being said,
now let's create some data to showcase this. So to showcase this, let's create some known
data using the linear regression formula. Now, if you're not sure what linear regression is,
or the formula is, let's have a look linear regression formula. This is how I'd find it.
Okay, we have some fancy Greek letters here. But essentially, we have y equals a function of x
and b plus epsilon. Okay. Well, there we go. A linear regression line has the equation in the
form of y equals a plus bx. Oh, I like this one better. This is nice and simple. We're going to
start from as simple as possible and work up from there. So y equals a plus bx,

```
where x is the
explanatory variable, and y is the dependent variable. The slope of the line is b.
And the
slope is also known as the gradient. And a is the intercept. Okay, the value of
when y
when x equals zero. Now, this is just text on a page. This is formula on a page.
You know how I
like to learn things? Let's code it out. So let's write it here. We'll use a linear
regression formula
to make a straight line with known parameters. I'm going to write this down because
parameter
is a common word that you're going to hear in machine learning as well. So a
parameter is
something that a model learns. So for our data set, if machine learning is a game
of two parts,
we're going to start with this. Number one is going to be done for us, because
we're going to
start with a known representation, a known data set. And then we want our model to
learn that
```

Section 46: Main Topics

Key Topics:

- Well, it's because typically X in machine learning you'll find is a matrix or a tensor
- Of course, we know what the relationship is between X and Y because we've coded this formula here
- That is the whole premise of machine learning

▶ 📄 Click to view detailed content

```
representation. This is all just talk, Daniel, let's get into coding. Yes, you're
right. You're
right. Let's do it. So create known parameters. So I'm going to use a little bit
different
names to what that Google definition did. So weight is going to be 0.7 and bias is
going to be 0.3.
Now weight and bias are another common two terms that you're going to hear in
neural networks.
So just keep that in mind. But for us, this is going to be the equivalent of our
weight will be B
and our bias will be A. But forget about this for the time being. Let's just focus
on the code.
So we know these numbers. But we want to build a model that is able to estimate
these numbers.
How? By looking at different examples. So let's create some data here. We're going
to create a
range of numbers. Start equals zero and equals one. We're going to create some
```

numbers between
zero and one. And they're going to have a gap. So the step the gap is going to be 0.02.
Now we're going to create an X variable. Why is X a capital here?
Well, it's because typically X in machine learning you'll find is a matrix or a tensor.
And if we remember back to the fundamentals, a capital represents a matrix or a tensor
and a lowercase represents a vector. But now case it's going to be a little confusing because
X is a vector. But later on, X will start to be a tensor and a matrix. So for now,
we'll just keep the capital, not capital notation.
We're going to create the formula here, which is remember how I said our weight is in this case,
the B and our bias is the A. So we've got the same formula here. Y equals weight times X plus
bias. Now let's have a look at these different numbers. So we'll view the first 10 of X and we'll
view the first 10 of Y. We'll have a look at the length of X and we'll have a look at the length of
Y. Wonderful. So we've got some values here. We've got 50 numbers of each. This is a little
confusing. Let's just view the first 10 of X and Y first. And then we can have a look at the
length here. So what we're going to be doing is building a model to learn some values,
to look at the X values here and learn what the associated Y value is and the relationship
between those. Of course, we know what the relationship is between X and Y because we've
coded this formula here. But you won't always know that in the wild. That is the whole premise of
machine learning. This is our ideal output and this is our input. The whole premise of machine
learning is to learn a representation of the input and how it maps to the output. So here are our
input numbers and these are our output numbers. And we know that the parameters of the weight and
bias are 0.7 and 0.3. We could have set these to whatever we want, by the way. I just like the
number 7 and 3. You could set these to 0.9, whatever, whatever. The premise would be the same.
So, oh, and what I've just done here, I kind of just coded this without talking.
But I just did torch a range and it starts at 0 and it ends at 1 and the step is 0.02. So there
we go, 000 by 0.02, 04. And I've unsqueezed it. So what does unsqueezed do? Removes the extra
dimensions. Oh, sorry, ads are extra dimension. You're getting confused here. So if we remove that,
we get no extra square bracket. But if we add unsqueeze, you'll see that we need this later on
for when we're doing models. Wonderful. So let's just leave it at that. That's enough for this
video, we've got some data to work with. Don't worry if this is a little bit confusing for now,
we're going to keep coding on and see what we can do to build a model to infer patterns in this

```
data. But right now, I want you to have a think, this is tensor data, but it's just
numbers on a
page. What might be a better way to hint, this is a hint by the way, visualize it.
What's our
data explorer's motto? Let's have a look at that in the next video. Welcome back.
In the last
video, we created some numbers on a page using the linear regression formula with
some known
parameters. Now, there's a lot going on here, but that's all right. We're going to
keep building
upon what we've done and learn by doing. So in this video, we're going to cover one
of the most
important concepts in machine learning in general. So splitting data into training
and test sets.
One of the most important concepts in machine learning in general. Now, I know I've
said this
already a few times. One of the most important concepts, but truly, this is
possibly, in terms
of data, this is probably the number one thing that you need to be aware of. And if
you've come
from a little bit of a machine learning background, you probably well and truly
know all about this.
But we're going to recover it anyway. So let's jump in to some pretty pictures. Oh,
look at that
one speaking of pretty pictures. But that's not what we're focused on now. We're
looking at the
three data sets. And I've written down here possibly the most important concept in
machine
learning, because it definitely is from a data perspective. So the course
materials,
imagine you're at university. So this is going to be the training set. And then you
have the
```

Section 47: Main Topics

Key Topics:

- So say you're trying to learn something at university or through this course, you might have all of the materials, which is your training set
- Now, let's just see if you're learning the course materials well
- And this is to see if you've gone through the entire course materials, and you've learned some things

▶ 📄 Click to view detailed content

```
practice exam, which is the validation set. Then you have the final exam, which is
the test set.
And the goal of all of this is for generalization. So let's step back. So say
```

you're trying to learn
something at university or through this course, you might have all of the materials, which is your
training set. So this is where our model learns patterns from. And then to practice what you've
done, you might have a practice exam. So the mid semester exam or something like that. Now,
let's just see if you're learning the course materials well. So in the case of our model,
we might tune our model on this plastic exam. So we might find that on the validation set,
our model doesn't do too well. And we adjusted a bit, and then we retrain it, and then it does
better. Before finally, at the end of semester, the most important exam is your final exam. And
this is to see if you've gone through the entire course materials, and you've learned some things.
Now you can adapt to unseen material. And that's a big point here. We're going to see this in
practice is that when the model learns something on the course materials, it never sees the validation
set or the test set. So say we started with 100 data points, you might use 70 of those data points
for the training material. You might use 15% of those data points, so 15 for the practice.
And you might use 15 for the final exam. So this final exam is just like if you're at university
learning something is to see if, hey, have you learned any skills from this material at all?
Are you ready to go into the wild into the quote unquote real world? And so this final exam is to
test your model's generalization, because it's never seen this data is, let's define generalization
is the ability for a machine learning model or a deep learning model to perform well on data it
hasn't seen before, because that's our whole goal, right? We want to build a machine learning model
on some training data that we can deploy in our application or production setting. And then
more data comes in that it hasn't seen before. And it can make decisions based on that new data
because of the patterns it's learned in the training set. So just keep this in mind,
three data sets training validation test. And if we jump in to the learn pytorch book,
we've got split data. So we're going to create three sets. Or in our case, we're only going to
create two or training in a test. Why is that? Because you don't always need a validation set.
There is often a use case for a validation set. But the main two that are always used is the training
set and the testing set. And how much should you split? Well, usually for the training set,
you'll have 60 to 80% of your data. If you do create a validation set, you'll have somewhere
between 10 and 20. And if you do create a testing set, it's a similar split to the validation set,

you'll have between 10 and 20%. So training, always testing always validation often, but
not always. So with that being said, I'll let you refer to those materials if you want. But now
let's create a training and test set with our data. So we saw before that we have 50 points,
we have X and Y, we have one to one ratio. So one value of X relates to one value of Y.
And we know that the split now for the training set is 60 to 80%. And the test set is 10 to 20%.
So let's go with the upper bounds of each of these, 80% and 20%, which is a very common split,
actually 80, 20. So let's go create a train test split. And we're going to go train split.
We'll create a number here so we can see how much. So we want an integer of 0.8, which is 80%
of the length of X. What does that give us? Train split should be about 40 samples. Wonderful.
So we're going to create 40 samples of X and 40 samples of Y. Our model will train on those 40
samples to predict what? The other 10 samples. So let's see this in practice. So X train,
Y train equals X. And we're going to use indexing to get all of the samples up until the train
split. That's what this colon does here. So hey, X up until the train split, Y up until the train
split, and then for the testing. Oh, thanks for that. Auto correct cola, but didn't actually need that
one. X test. Y test equals X. And then we're going to get everything from the train split onwards.
So the index onwards, that's what this notation means here. And Y from the train split onwards as
well. Now, there are many different ways to create a train and test split. Ours is quite simple here,
but that's because we're working with quite a simple data set. One of the most popular methods
that I like is scikit learns train test split. We're going to see this one later on. It adds a
little bit of randomness into splitting your data. But that's for another video, just to make you
aware of it. So let's go length X train. We should have 40 training samples to
how many testing samples length X test and length Y test. Wonderful 40 40 10 10 because we have
training features, training labels, testing features, testing labels. So essentially what we've
created here is now a training set. We've split our data. Training set could also be referred to
as training split yet another example of where machine learning has different names for different

Section 48: Main Topics

Key Topics:

- One of your biggest, biggest, biggest hurdles in machine learning will be creating proper training and test sets

▶ 📄 Click to view detailed content

```
things. So set split same thing training split test split. This is what we've
created. Remember,
the validation set is used often, but not always because our data set is quite
simple. We're just
sticking with the necessities training and test. But keep this in mind. One of your
biggest,
biggest, biggest hurdles in machine learning will be creating proper training and
test sets. So
it's a very important concept. With that being said, I did issue the challenge in
the last video
to visualize these numbers on a page. We haven't done that in this video. So let's
move towards
that next. I'd like you to think of how could you make these more visual? Right.
These are just
numbers on a page right now. Maybe that plot lib can help. Let's find out. Hey,
hey, hey, welcome
back. In the last video, we split our data into training and test sets. And now
later on,
we're going to be building a model to learn patterns in the training data to relate
to the
testing data. But as I said, right now, our data is just numbers on a page. It's
kind of
hard to understand. You might be able to understand this, but I prefer to get
visual. So let's write
this down. How might we better visualize our data? And I'm put a capital here. So
we're grammatically
correct. And this is where the data Explorers motto comes in. Visualize, visualize,
visualize.
Ha ha. Right. So if ever you don't understand a concept, one of the best ways to
start
understanding it more for me is to visualize it. So let's write a function to do
just that.
We're going to call this plot predictions. We'll see why we call it this later on.
That's the
benefit of making these videos is that I've got a plan for the future. Although it
might seem
like I'm winging it, there is a little bit of behind the scenes happening here. So
we'll have
the train data, which is our X train. And then we'll have the train labels, which
is our Y train.
And we'll also have the test data. Yeah, that's a good idea. X test. And we'll also
have the test
labels, equals Y test. Excuse me. I was looking at too many X's there. And then the
predictions.
And we'll set this to none, because we don't have any predictions yet. But as you
might have guessed,
we might have some later on. So we'll put a little doc string here, so that we're
```

being nice and
Pythonic. So plots training data, test data, and compares predictions. Nice and simple.
Nothing too outlandish. And then we're going to create a figure. This is where map plot lib comes
in. Plot figure. And we'll go fig size equals 10, seven, which is my favorite hand in poker.
And we'll plot the training data in blue also happens to be a good dimension for a map plot.
Plot dot scatter. Train data. Creating a scatter plot here. We'll see what it does in a second.
Color. We're going to give this a color of B for blue. That's what C stands for in map plot lib
scatter. We'll go size equals four and label equals training data. Now, where could you find
information about this scatter function here? We've got command shift space. Is that going to
give us a little bit of a doc string? Or sometimes if command not space is not working,
you can also hover over this bracket. I think you can even hover over this.
There we go. But this is a little hard for me to read. Like it's there, but it's got a lot going
on. X, Y, S, C, C map. I just like to go map plot lib scatter. There we go. We've got a whole
bunch of information there. A little bit easier to read for me here. And then you can see some
examples. Beautiful. So now let's jump back into here. So in our function plot predictions,
we've taken some training data, test data. We've got the training data plotting in blue. What
color should we use for the testing data? How about green? I like that idea. Plot.scatter.
Test data. Green's my favorite color. What's your favorite color? C equals G. You might be
able to just plot it in your favorite color here. Just remember though, it'll be a little bit
different from the videos. And then we're going to call this testing data. So just the exact same
line is above, but with a different set of data. Now, let's check if there are predictions. So
are there predictions? So if predictions is not none, let's plot the predictions, plot the
predictions, if they exist. So plot scatter test data. And why are we plotting the test data?
Remember, what is our scatter function? Let's go back up to here. It takes in x and y. So
our predictions are going to be compared to the testing data labels. So that's the whole
game that we're playing here. We're going to train our model on the training data.
And then to evaluate it, we're going to get our model to predict the y values
as with the input of x test. And then to evaluate our model, we compare how good our models
predictions are. In other words, predictions versus the actual values of the test data set.
But we're going to see this in practice. Rather than just talk about it. So let's do our predictions
in red. And label equals predictions. Wonderful. So let's also show the legend,

```
because, I mean,
we're legends. So we could just put in a mirror here. Now I'm kidding. Legend is
going to show
our labels on the map plot. So prop equals size and prop stands for properties.
Well,
```

Section 49: Main Topics

Key Topics:

- So the whole idea of what we're going to be doing with our machine learning model is we don't actually really need to build a machine learning model for this
- We could do other things, but machine learning is fun
- But now let's get into building our first PyTorch model

▶ 📄 Click to view detailed content

```
it may or may not. I just like to think it does. That's how I remember it. So we
have a beautiful
function here to plot our data. Should we try it out? Remember, we've got hard
coded inputs here,
so we don't actually need to input anything to our function. We've got our train
and test data
ready to go. If in doubt, run the code, let's check it out. Did we make a mistake
in our plot
predictions function? You might have caught it. Hey, there we go. Beautiful. So
because we don't
have any predictions, we get no red dots. But this is what we're trying to do.
We've got a simple
straight line. You can't get a much more simple data set than that. So we've got
our training data
in blue, and we've got our testing data in green. So the whole idea of what we're
going to be doing
with our machine learning model is we don't actually really need to build a machine
learning
model for this. We could do other things, but machine learning is fun. So we're
going to take
in the blue dots. There's quite a pattern here, right? This is the relationship we
have an x value
here, and we have a y value. So we're going to build a model to try and learn the
pattern
of these blue dots, so that if we fed our model, our model, the x values of the
green dots,
could it predict the appropriate y values for that? Because remember, these are the
test data set.
So pass our model x test to predict y test. So blue dots as input, green dots as
the ideal output.
This is the ideal output, a perfect model would have red dots over the top of the
```

green dots. So
that's what we will try to work towards. Now, we know the relationship between x and y.
How do we know that? Well, we set that up above here. This is our weight and bias.
We created that line y equals weight times x plus bias, which is the simple version of the
linear regression formula. So mx plus c, you might have heard that in high school algebra,
so gradient plus intercept. That's what we've got. With that being said,
let's move on to the next video and build a model. Well, this is exciting. I'll see you there.
Welcome back. In the last video, we saw how to get visual with our data. We followed the data
explorer's motto of visualize, visualize, visualize. And we've got an idea of the training data that
we're working with and the testing data that we're trying to build a model to learn the patterns
in the training data, essentially this upwards trend here, to be able to predict the testing data.
So I just want to give you another heads up. I took a little break after the recording last
video. And so now my colab notebook has disconnected. So I'm going to click reconnect.
And my variables here may not work. So this is what might happen on your end. If you take a break
from using Google Colab and come back, if I try to run this function, they might have been saved,
it looks like they have. But if not, you can go restart and run all. This is typically one of the
most helpful troubleshooting steps of using Google Colab. If a cell, say down here isn't working,
you can always rerun the cells above. And that may help with a lower cell here, such as if this
function wasn't instantiated because this cell wasn't run, and we couldn't run this cell here,
which calls this function here, we just have to rerun this cell above so that we can run this one.
But now let's get into building our first PyTorch model. We're going to jump straight into the code.
So our first PyTorch model. Now this is very exciting.
Let's do it. So we'll turn this into Markdown. Now we're going to create a linear regression model.
So look at linear regression formula again, we're going to create a model that's essentially going
to run this computation. So we need to create a model that has a parameter for A, a parameter for B,
and in our case it's going to be weight and bias, and a way to do this forward computation.
What I mean by that, we're going to see with code. So let's do it. We'll do it with pure PyTorch.
So create a linear regression model class. Now if you're not experienced with using Python classes,
I'm going to be using them throughout the course, and I'm going to call this one linear regression
model. If you haven't dealt with Python classes before, that's okay. I'm going to be explaining
what we're doing as we're doing it. But if you'd like a deeper dive, I'd recommend

```
you to real Python
classes. OOP in Python three. That's a good rhyming. So I'm just going to link this
here.
Because we're going to be building classes throughout the course,
I'd recommend getting familiar with OOP, which is object oriented programming, a
little bit of a
mouthful, hence the OOP in Python. To do so, you can use the following resource
from real Python.
But when I'm going to go through that now, I'd rather just code it out and talk it
out while we
do it. So we've got a class here. Now the first thing you might notice is that the
class inherits
from nn.module. And you might be wondering, well, what's nn.module? Well, let's
write down here,
almost everything in PyTorch inherits from nn.module. So you can imagine nn.module
as the
Lego building bricks of PyTorch model. And so nn.module has a lot of helpful
inbuilt things that's
```

Section 50: Main Topics

Key Topics:

- going to help us build our PyTorch models
- And of course, how could you learn more about it
- module, PyTorch

▶ 📄 Click to view detailed content

```
going to help us build our PyTorch models. And of course, how could you learn more
about it?
Well, you could go nn.module, PyTorch. Module. Here we go. Base class for all
neural network
modules. Wonderful. Your models should also subclass this class. So that's what
we're building. We're
building our own PyTorch model. And so the documentation here says that your models
should
also subclass this class. And another thing with PyTorch, this is what makes it, it
might seem very
confusing when you first begin. But modules can contain other modules. So what I
mean by being a
Lego brick is that you can stack these modules on top of each other and make
progressively more
complex neural networks as you go. But we'll leave that for later on. For now,
we're going to start
with something nice and simple. And let's clean up our web browser. So we're going
to create a
constructor here, which is with the init function. It's going to take self as a
parameter. If you're
```

not sure of what's going on here, just follow along with the code for now. And I'd encourage you
to read this documentation here after the video. So then we have super dot init. I know when I
first started learning this, I was like, why do we have to write a knit twice? And then what's
super and all that jazz. But just for now, just take this as being some required Python syntax.
And then we have self dot weights. So that means we're going to create a weights parameter. We'll
see why we do this in a second. And to create that parameter, we're going to use nn dot parameter.
And just a quick reminder that we imported nn from torch before. And if you remember,
nn is the building block layer for neural networks. And within nn, so nn stands for neural network
is module. So we've got nn dot parameter. Now, we're going to start with random parameters.
So torch dot rand n. One, we're going to talk through each of these in a second. So I'm also
going to put requires, requires grad equals true. We haven't touched any of these, but that's okay.
D type equals torch dot float. So let's see what nn parameter tells us. What do we have here?
A kind of tensor that is to be considered a module parameter. So we've just created a module
using nn module. Parameters are torch tensor subclasses. So this is a tensor in itself
that have a very special property when used with modules. When they're assigned as a module
attribute, they are automatically added to the list of its parameters. And we'll appear e g
in module dot parameters iterator. Oh, we're going to see that later on. Assigning a tensor
doesn't have such effect. So we're creating a parameter here. Now requires grad. What does that
mean? Well, let's just rather than just try to read the doc string collab, let's look it up.
nn dot parameter. What does it say requires grad optional. If the parameter requires gradient.
Hmm. What does requires gradient mean? Well, let's come back to that in a second. And then
for now, I just want you to think about it. D type equals torch dot float. Now,
the data type here torch dot float is, as we've discussed before, is the default
for pytorch to watch dot float. This could also be torch dot float 32. So we're just going to
leave it as torch float 32, because pytorch likes to work with flight 32. Now, do we have
this by default? We do. So we don't necessarily have to set requires grad equals true. So just
keep that in mind. So now we've created a parameter for the weights. We also have to create a parameter
for the bias. Let's finish creating this. And then we'll write the code, then we'll talk about it.
So rand n. Now requires grad equals true. And d type equals torch dot float. There we go.
And now we're going to write a forward method. So forward method to define the

```
computation
in the model. So let's go def forward, which self takes in a parameter x, which is
data,
which X is expected to be of type torch tensor. And it returns a torch dot tensor.
And then we go
here. And so we say X, we don't necessarily need this comment. I'm just going to
write it anyway.
X is the input data. So in our case, it might be the training data. And then from
here, we want
it to return self dot weights times X plus self dot bias. Now, where have we seen
this before?
Well, this is the linear regression formula. Now, let's take a step back into how
we created our data.
And then we'll go back through and talk a little bit more about what's going on
here.
So if we go back up to our data, where did we create that? We created it here. So
you see how
we've created known parameters, weight and bias. And then we created our y
variable, our target,
using the linear regression formula, wait times X plus bias, and X were a range of
numbers.
So what we've done with our linear regression model that we've created from
scratch,
if we go down here, we've created a parameter, weights. This could just be weight,
if we wanted to.
We've created a parameter here. So when we created our data, we knew what the
parameters weight and
bias were. The whole goal of our model is to start with random numbers. So these
are going to be
random parameters. And to look at the data, which in our case will be the training
samples,
and update those random numbers to represent the pattern here. So ideally, our
model, if it's
```

Section 51: Main Topics

Key Topics:

- learning correctly, will take our weight, which is going to be a random value, and our bias, which is going to be a random value
- So that's the premise of machine learning
- And so what this is going to do is when we run computations using this model here, pytorch is going to keep track of the gradients of our weights parameter and our bias parameter

▶ 📄 Click to view detailed content

```
learning correctly, will take our weight, which is going to be a random value, and
our bias,
which is going to be a random value. And it will run it through this forward
calculation,
which is the same formula that we use to create our data. And it will adjust the
weight and bias
to represent as close as possible, if not perfect, the known parameters. So that's
the premise of
machine learning. And how does it do this? Through an algorithm called gradient
descent. So I'm just
going to write this down because we've talked a lot about this, but I'd like to
just tie it together
here. So what our model does, so start with random values, weight and bias, look at
training data,
and adjust the random values to better represent the, or get closer to the ideal
values. So the
weight and bias values we use to create the data. So that's what it's going to do.
It's going to
start with random values, and then continually look at our training data to see if
it can adjust
those random values to be what would represent this straight line here. Now, how
does it do so?
How does it do so? Through two main algorithms. So one is gradient descent, and two
is back
propagation. So I'm going to leave it here for the time being, but we're going to
continue talking
about this gradient descent is why we have requires grad equals true. And so what
this is going to
do is when we run computations using this model here, pytorch is going to keep
track of the gradients
of our weights parameter and our bias parameter. And then it's going to update them
through a
combination of gradient descent and back propagation. Now, I'm going to leave this
as extracurricular
for you to look through and gradient descent and back propagation. I'm going to add
some
resources here. There will also be plenty of resources in the pytorch workflow
fundamentals
book chapter on how these algorithms work behind the scenes. We're going to be
focused on the code,
the pytorch code, to trigger these algorithms behind the scenes. So pytorch, lucky
for us,
has implemented gradient descent and back propagation for us. So we're writing the
higher level code
here to trigger these two algorithms. So in the next video, we're going to step
through this a
little bit more, and then further discuss some of the most useful and required
modules of pytorch,
particularly an N and a couple of others. So let's leave it there, and I'll see you
in the next video.
Welcome back. In the last video, we covered a whole bunch in creating our first
pytorch model
that inherits from nn.module. We talked about object oriented programming and how a
lot of
pytorch uses object oriented programming. I can't say that. I might just say OOP
for now.
```

What I've done since last video, though, is I've added two resources here for gradient descent
and back propagation. These are two of my favorite videos on YouTube by the channel three blue
one brown. So this is on gradient descent. I would highly recommend watching this entire series,
by the way. So that's your extra curriculum for this video, in particular, and for this course overall
is to go through these two videos. Even if you're not sure entirely what's happening,
you will gain an intuition for the code that we're going to be writing with pytorch.
So just keep that in mind as we go forward, a lot of what pytorch is doing behind the scenes for us
is taking care of these two algorithms for us. And we also created two parameters here in our model
where we've instantiated them as random values. So one parameter for each of the ones that we use,
the weight and bias for our data set. And now I want you to keep in mind that we're working
with a simple data set here. So we've created our known parameters. But in a data set that you
haven't created by yourself, you've maybe gathered that from the internet, such as images,
you won't be necessarily defining these parameters. Instead, another module from nn will define the
parameters for you. And we'll work out what those parameters should end up being. But since we're
working with a simple data set, we can define our two parameters that we're trying to estimate.
This is a key point here is that our model is going to start with random values. That's the
annotation I've added here. Start with a random weight value using torch random. And then we've
told it that it can update via gradient descent. So pytorch is going to track the gradients of
this parameter for us. And then we've told it that the D type we want is float 32. We don't
necessarily need these two set explicitly, because a lot of the time the default in pytorch is to
set these two requires grad equals true and d type equals torch dot float. Does that for us
behind the scenes? But just to keep things as fundamental and as straightforward as possible,
we've set all of this explicitly. So let's jump into the keynote. I'd just like to explain
what's going on one more time in a visual sense. So here's the exact code that we've
just written. I've just copied it from here. And I've just made it a little bit more colorful.
But here's what's going on. So when you build a model in pytorch, it subclasses the nn.modgable
class. This contains all the building blocks for neural networks. So our class of model, subclasses
nn.modgable. Now, inside the constructor, we initialize the model parameters. Now, as we'll see,

```
later on with bigger models, we won't necessarily always explicitly create the
weights and biases.
```

Section 52: Main Topics

Key Topics:

- So this, in turn, means that pytorch behind the scenes will track all of the gradients for these parameters here for use with torch
- auto grad module of pytorch is what implements gradient descent
- Now, a lot of this will happen behind the scenes for when we write our pytorch training code

▶ 📄 Click to view detailed content

```
We might initialize whole layers. Now, this is a concept we haven't touched on yet,
but
we might initialize a list of layers or whatever we need. So basically, what
happens in here is that
we create whatever variables that we need for our model to use. And so these could
be different
layers from torch.nn, single parameters, which is what we've done in our case, hard
coded values,
or even functions. Now, we've explicitly set requires grad equals true for our
model parameters.
So this, in turn, means that pytorch behind the scenes will track all of the
gradients
for these parameters here for use with torch.auto grad. So torch.auto grad module
of pytorch is what
implements gradient descent. Now, a lot of this will happen behind the scenes for
when we write
our pytorch training code. So if you'd like to know what's happening behind the
scenes,
I'd highly recommend you checking out these two videos, hence is why I've linked
them here.
Oh, and for many pytorch.nn modules requires grad is true is set by default.
Finally, we've got a forward method. Now, any subclass of nn.modgable, which is
what we've done,
requires a forward method. Now, we can see this in the documentation. If we go
torch
dot nn.modgable.
Click on module. Do we have forward?
Yeah, there we go. So forward, we've got a lot of things built into an nn.modgable.
So you see here, this is a subclass of an nn.modgable. And then we have forward.
So forward is what defines the computation performed at every call. So if we were
to call linear regression model and put some data through it, the forward method is
the
operation that this module does that this model does. And in our case, our forward
```

```
method is
the linear regression function. So keep this in mind, any subclass of nn.modgable
needs to
override the forward method. So you need to define a forward method if you're going
to subclass
nn.modgable. We'll see this very hands on. But for now, I believe that's enough
coverage of what
we've done. If you have any questions, remember, you can ask it in the discussions.
We've got a
fair bit going on here. But I think we've broken it down a fair bit. The next step
is for us to,
I know I mentioned this in a previous video is to cover some PyTorch model building
essentials.
But we're going to cover a few more of them. We've seen some already. But the next
way to really
start to understand what's going on is to check the contents of our model, train
one, and make
some predictions with it. So let's get hands on with that in the next few videos.
I'll see you there.
Welcome back. In the last couple of videos, we stepped through creating our first
PyTorch model.
And it looks like there's a fair bit going on here. But some of the main takeaways
is that almost
every model in PyTorch inherits from nn.modgable. And if you are going to inherit
from nn.modgable,
you should override the forward method to define what computation is happening in
your model.
And for later on, when our model is learning things, in other words, updating its
weights and
bias values from random values to values that better fit the data, it's going to do
so via
gradient descent and back propagation. And so these two videos are some extra
curriculum
for what's happening behind the scenes. But we haven't actually written any code
yet to trigger
these two. So I'll refer back to these when we actually do write code to do that.
For now,
we've just got a model that defines some forward computation. But speaking of
models, let's have
a look at a couple of PyTorch model building essentials. So we're not going to
write too much
code for this video, and it's going to be relatively short. But I just want to
introduce you to some
of the main classes that you're going to be interacting with in PyTorch. And we've
seen
some of these already. So one of the first is torch.nn. So contains all of the
building blocks
for computational graphs. Computational graphs is another word for neural networks.
Well, actually computational graphs is quite general. I'll just write here, a
neural network
can be considered a computational graph. So then we have torch.nn.parameter. We've
seen this.
So what parameters should our model try and learn? And then we can write here often
a PyTorch
layer from torch.nn will set these for us. And then we've got torch.nn.module,
which is
what we've seen here. And so torch.nn.module is the base class for all neural
```

```
network modules.
If you subclass it, you should overwrite forward, which is what we've done here.
We've created our
own forward method. So what else should we cover here? We're going to see these
later
on, but I'm going to put it here, torch.optim. This is where the optimizers in
PyTorch live.
They will help with gradient descent. So optimizer, an optimizer is, as we've said
before,
that our model starts with random values. And it looks at training data and adjusts
the random
values to better represent the ideal values. The optimizer contains algorithm
that's going to
optimize these values, instead of being random, to being values that better
represent our data.
So those algorithms live in torch.optim. And then one more for now, I'll link to
extra resources.
And we're going to cover them as we go. That's how I like to do things, cover them
as we need them.
So all nn.module. So this is the forward method. I'm just going to explicitly say
here that all
nn.module subclasses require you to overwrite forward. This method defines what
happens
in the forward computation. So in our case, if we were to pass some data to our
linear regression
```

Section 53: Main Topics

Key Topics:

- PyTorch is central neural network building modules
- And if you'd like more, one of my favorite resources is the PyTorch cheat sheet
- As I said, this course is not a replacement for the documentation

▶ 📄 Click to view detailed content

```
model, the forward method would take that data and perform this computation here.
And as your models get bigger and bigger, ours is quite straightforward here.
This forward computation can be as simple or as complex as you like, depending on
what you'd
like your model to do. And so I've got a nice and fancy slide here, which basically
reiterates
what we've just discussed. PyTorch is central neural network building modules.
So the module torch.nn, torch.nn.module, torch.optim, torch.utils.dataset. We
haven't actually talked
about this yet. And I believe there's one more data loader. We're going to see
these two later on.
But these are very helpful when you've got a bit more of a complicated data set. In
our case,
```

we've got just 50 integers for our data set. We've got a simple straight line. But when we need
to create more complex data sets, we're going to use these. So this will help us build models.
This will help us optimize our models parameters. And this will help us load data. And if you'd
like more, one of my favorite resources is the PyTorch cheat sheet. Again, we're referring
back to the documentation. See, all of this documentation, right? As I said, this course is
not a replacement for the documentation. It's just my interpretation of how one should best
become familiar with PyTorch. So we've got imports, the general import torch from torch.utils.dataset
data loader. Oh, did you look at that? We've got that mentioned here, data, data set data loader.
And torch, script and jit, neural network API. I want an X. I'll let you go through here.
We're covering some of the most fundamental ones here. But there's, of course, PyTorch is
quite a big library. So some extra curricula for this video would be to go through this for
five to 10 minutes and just read. You don't have to understand them all. We're going to start to
get more familiar with all of these. We're not all of them because, I mean, that would require
making videos for the whole documentation. But a lot of these through writing them via code.
So that's enough for this video. I'll link this PyTorch cheat sheet in the video here.
And in the next video, how about we, we haven't actually checked out what happens if we do
create an instance of our linear regression model. I think we should do that. I'll see you there.
Welcome back. In the last video, we covered some of the PyTorch model building essentials. And look,
I linked a cheat sheet here. There's a lot going on. There's a lot of text going on in the page.
Of course, the reference material for here is in the Learn PyTorch book. PyTorch model building
essentials under 0.1, which is the notebook we're working on here. But I couldn't help myself.
I wanted to add some color to this. So before we inspect our model, let's just add a little bit
of color to our text on the page. We go to whoa. Here's our workflow. This is what we're covering
in this video, right? Or in this module, 0.1. But to get data ready, here are some of the most
important PyTorch modules. Torchvision.transforms. We'll see that when we cover computer vision later
on. Torch.utils.data.data set. So that's if we want to create a data set that's a little bit
more complicated than because our data set is so simple, we haven't used either of these
data set creator or data loader. And if we go build a picker model, well, we can use torch.nn.
We've seen that one. We've seen torch.nn.module. So in our case, we're building a

```
model. But if we
wanted a pre-trained model, well, there's some computer vision models that have
already been
built for us in torchvision.models. Now torchvision stands for PyTorch's computer
vision
module. So we haven't covered that either. But this is just a spoiler for what's
coming on
later on. Then if the optimizer, if we wanted to optimize our model's parameters to
better
represent a data set, we can go to torch.optim. Then if we wanted to evaluate the
model,
well, we've got torch metrics for that. We haven't seen that, but we're going to be
hands-on with all of these later on. Then if we wanted to improve through
experimentation,
we've got torch.utils.tensorboard. Hmm. What's this? But again, if you want more,
there's some at the PyTorch cheat sheet. But now this is just adding a little bit
of color
and a little bit of code to our PyTorch workflow. And with that being said, let's
get a little bit
deeper into what we've built, which is our first PyTorch model. So checking the
contents of our
PyTorch model. So now we've created a model. Let's see what's inside. You might
already be able
to guess this by the fact of what we've created in the constructor here in the init
function.
So what do you think we have inside our model? And how do you think we'd look in
that? Now,
of course, these are questions you might not have the answer to because you've
just, you're like,
Daniel, I'm just starting to learn PyTorch. I don't know these, but I'm asking you
just to start
thinking about these different things, you know? So we can check out our model
parameters or what's
inside our model using, wait for it, dot parameters. Oh, don't you love it when
things are nice and
simple? Well, let's check it out. Hey, well, first things we're going to do is
let's create a random
seed. Now, why are we creating a random seed? Well, because recall, we're creating
these parameters
with random values. And if we were to create them with outer random seed, we would
get different
values every time. So for the sake of the educational sense, for the sake of this
video,
```

Section 54: Main Topics

Key Topics:

- Model zero, because it's going to be the zeroth model, the first model that we've ever created in this whole course, how amazing linear regression model, which is what our class is called

- So this is the essence of what our machine learning models and deep learning models are going to do
- And so of course, we're not going to do them all by hand

▶ 📄 Click to view detailed content

```
we're going to create a manual seed here, torch dot manual seed. I'm going to use
42 or maybe 43,
I could use 43 now 42 because I love 42. It's the answer to the universe. And we're
going to create
an instance of the model that we created. So this is a subclass of an end up
module.
So let's do it. Model zero, because it's going to be the zeroth model, the first
model that
we've ever created in this whole course, how amazing linear regression model, which
is what
our class is called. So we can just call it like that. That's all I'm doing, just
calling this class.
And so let's just see what happens there. And then if we go model zero, what does
it give us? Oh,
linear regression. Okay, it doesn't give us much. But we want to find out what's
going on in here.
So check out the parameters. So model zero dot parameters. What do we get from
this? Oh, a generator.
Well, let's turn this into a list that'll be better to look at. There we go. Oh,
how exciting is that?
So parameter containing. Look at the values tensor requires grad equals true
parameter containing
wonderful. So these are our model parameters. So why are they the values that they
are? Well,
it's because we've used torch rand n. Let's see what happens if we go, let's just
create torch dot
rand n one, what happens? We get a value like that. And now if we run this again,
we get the same values. But if we run this again, so keep this in one two, three,
four,
five, actually, that's, wow, that's pretty cool that we got a random value that was
all in order,
four in a row. Can we do it twice in a row? Probably not. Oh, we get it the same
one. Now,
why is that? Oh, we get a different one. Did we just get the same one twice? Oh, my
gosh,
we got the same value twice in a row. You saw that. You saw that. That's
incredible. Now,
the reason why we get this is because this one is different every time because
there's no random
seed. Watch if we put the random seed here, torch dot manual seed, 42, 3, 3, 6, 7,
what happens?
3, 3, 6, 7, what happens? 3, 3, 6, 7. Okay. And what if we commented out the random
seed
here, initialized our model, different values, two, three, five, two, three, four,
five, it must
like that value. Oh, my goodness. Let me know if you get that value, right? So if
we keep going,
we get different values every single time. Why is this? Why are we getting
```

different values
every single time? You might be, Daniel, you sound like a broken record, but I'm trying to
really drive home the fact that we initialize our models with random parameters. So this is the
essence of what our machine learning models and deep learning models are going to do. Start with
random values, weights and bias. Maybe we've only got two parameters here, but the future models
that we build might have thousands. And so of course, we're not going to do them all by hand.
We'll see how we do that later on. But for now, we start with random values. And our ideal model
will look at the training data and adjust these random values. But just so that we can get
reproducible results, I'll get rid of this cell. I've set the random seed here. So you should be
getting similar values to this. If you're not, because there's maybe some sort of pytorch update
and how the random seeds calculated, you might get slightly different values. But for now,
we'll use torch.manualc.42. And I want you to just be aware of this can be a little bit confusing.
If you just do the list of parameters, for me, I understand it better if I list the name parameters.
So the way we do that is with model zero, and we call state dict on it. This is going to give us
our dictionary of the parameters of our model. So as you can see here, we've got weights,
and we've got bias, and they are random values. So where did weights and bias come from? Well,
of course, they came from here, weights, bias. But of course, as well up here,
we've got known parameters. So now our whole goal is what? Our whole goal is to build code,
or write code, that is going to allow our model to look at these blue dots here,
and adjust this weight and bias value to be weights as close as possible to weight and bias.
Now, how do we go from here and here to here and here? Well, we're going to see that in future
videos, but the closer we get these values to these two, the better we're going to be able to
predict and model our data. Now, this principle, I cannot stress enough, is the fundamental
entire foundation, the fundamental foundation. Well, good description, Daniel. The entire
foundation of deep learning, we start with some random values, and we use gradient descent and
back propagation, plus whatever data that we're working with to move these random values as close
as possible to the ideal values. And in most cases, you won't know what the ideal values are.
But in our simple case, we already know what the ideal values are.
So just keep that in mind going forward. The premise of deep learning is to start with random
values and make them more representative closer to the ideal values. With that being said,

```
let's try and make some predictions with our model as it is. I mean, it's got
random values.
```

Section 55: Main Topics

Key Topics:

- In the last video, we checked out the internals of our first PyTorch model
- And we also discussed the entire premise of deep learning is to start with random numbers and slowly progress those towards more ideal numbers, slightly less random numbers based on the data
- Because remember again, another premise of a machine learning model is to take some features as input and make some predictions close to some sort of labels

▶ 📄 Click to view detailed content

```
How do you think the predictions will go? So I think in the next video, we'll make
some predictions
on this test data and see what they look like. I'll see you there. Welcome back. In
the last
video, we checked out the internals of our first PyTorch model. And we found out
that because we're
creating our model with torch dot or the parameters of our model, with torch dot
rand, they begin as
random variables. And we also discussed the entire premise of deep learning is to
start with random
numbers and slowly progress those towards more ideal numbers, slightly less random
numbers based
on the data. So let's see, before we start to improve these numbers, let's see what
their predictive
power is like right now. Now you might be able to guess how well these random
numbers will be
able to predict on our data. You're not sure what that predicting means? Let's have
a look. So making
predictions using torch dot inference mode, something we haven't seen. But as
always, we're going to
discuss it while we use it. So to check our models predictive power, let's see how
well
it predicts Y test based on X test. Because remember again, another premise of a
machine
learning model is to take some features as input and make some predictions close to
some sort of
labels. So when we pass data through our model, it's going to run it through the
forward method.
So here's where it's a little bit confusing. We defined a forward method and it
takes X as input.
Now I've done a little X, but we're going to pass it in a large X as its input. But
```

the reason why I've
done a little X is because oftentimes in pytorch code, you're going to find all over the internet
is that X is quite common, commonly used in the forward method here, like this as the input data.
So I've just left it there because that's what you're going to find quite often.
So let's test it out. We haven't discussed what inference mode does yet, but we will make predictions
with model. So with torch dot inference mode, let's use it. And then we will discuss what's going
on. Y threads equals a model zero X test. So that's all we're doing. We're passing the X test data
through our model. Now, when we pass this X test in here, let's remind ourselves of what X test is.
X test 10 variables here. And we're trying to our ideal model will predict the exact values of Y test.
So this is what our model will do if it's a perfect model. It will take these X test values as input,
and it will return these Y test values as output. That's an ideal model. So the predictions are the
exact same as the test data set. How do you think our model will go considering it's starting with
random values as its parameters? Well, let's find out, hey. So what can that Y threads?
Oh, what's happened here? Not implemented error. Ah, this is an error I get quite often in Google
Colab when I'm creating a high-torch model. Now, it usually happens. I'm glad we've stumbled upon
this. And I think I know the fix. But if not, we might see a little bit of troubleshooting in this
video is that when we create this, if you see this not implemented error, right, it's saying that
the Ford method. Here we go. Ford implemented. There we go. It's a little bit of a rabbit hole
this not implemented area. I've come across it a fair few times and it took me a while to figure
out that for some reason the spacing. So in Python, you know how you have space space and that defines
a function space space. There's another thing there and another line there. For some reason,
if you look at this line in my notebook, and by the way, if you don't have these lines or if you
don't have these numbers, you can go into tools, settings, editor, and then you can define them here.
So show line numbers, show notation guides, all that sort of jazz there. You can customize what's
going on. But I just have these two on because I've run into this error a fair few times. And so
it's because that this Ford method is not in line with this bracket here. So we need to highlight
this and click shift tab, move it over. So now you see that it's in line here. And then if we run
this, won't change any output there. See, that's the hidden gotcha. Is that when we ran this before,
it found no error. But then when we run it down here, it works. So just keep that in mind. I'm
really glad we stumbled upon that because indentation errors, not implemented

```
errors,
one of the most common errors you'll find in PyTorch, or in, well, when you're
writing PyTorch
code in Google CoLab, I'm not sure why, but it just happens. So these are our
models predictions
so far by running the test data through our models Ford method that we defined. And
so if
we look at Y test, are these close? Oh my gosh, they are shocking. So why don't we
visualize them?
Plot predictions. And we're going to put in predictions equals Y threads.
Let's have a look. Oh my goodness. All the way over here. Remember how we discussed
before
that an ideal model will have, what, red dots on top of the green dots because our
ideal model
will be perfectly predicting the test data. So right now, because our model is
initialized with
random parameters, it's basically making random predictions. So they're extremely
far from where
our ideal predictions are, is that we'll have some training data. And our model
predictions,
```

Section 56: Main Topics

Key Topics:

- That's something called transfer learning

▶ 📄 Click to view detailed content

```
when we first create our model will be quite bad. But we want to write some code
that will
hopefully move these red dots closer to these green dots. I'm going to see how we
can do that in
later videos. But we did one thing up here, which we haven't discussed, which is
with torch dot
inference mode. Now this is a context manager, which is what happens when we're
making predictions.
So making predictions, another word for predictions is inference torch uses
inference. So I'll try
to use that a bit more, but I like to use predictions as well. We could also just
go
Y preds equals model zero dot X test. And we're going to get quite a similar
output.
Right. But I've put on inference mode because I want to start making that a habit
for later on,
when we make predictions, put on inference mode. Now why do this? You might notice
something different.
What's the difference here between the outputs? Y preds equals model. There's no
inference mode
here, no context manager. Do you notice that there's a grad function here? And we
```

don't need to go
into discussing what exactly this is doing here. But do you notice that this one is lacking that
grad function? So do you remember how behind the scenes I said that pie torch does a few things
with requires grad equals true, it keeps track of the gradients of different parameters so that
they can be used in gradient descent and back propagation. Now what inference mode does is it
turns off that gradient tracking. So it essentially removes all of the, because when we're doing
inference, we're not doing training. So we don't need to keep track of the gradient. So we don't
need to do to keep track of how we should update our models. So inference mode disables all of the
useful things that are available during training. What's the benefit of this? Well, it means that
pie torch behind the scenes is keeping track of less data. So in turn, it will, with our small
data set, it probably won't be too dramatic. But with a larger data set, it means that your
predictions will potentially be a lot faster because a whole bunch of numbers aren't being
kept track of or a whole bunch of things that you don't need during prediction mode or inference
mode. That's why it's called inference mode. I'm not being saved to memory. If you'd like to
learn more about this, you go pie torch inference mode Twitter. I just remember to search for Twitter
because they did a big tweet storm about it. Here we go. So oh, this is another thing that we can
cover. I'm going to copy this in here. But there's also a blog post about what's going on behind
the scenes. Long story short, it makes your code faster. Want to make your inference code and pie
torch run faster? Here's a quick thread on doing exactly that. And that's what we're doing. So
I'm going to write down here. See more on inference mode here.
And I just want to highlight something as well is that they referenced torch no grad with the
torch inference mode context manager. Inference mode is fairly new in pie torch. So you might
see a lot of code existing pie torch code with torch dot no grad. You can use this as well.
Why? Preds equals model zero. And this will do much of the same as what inference mode is doing.
But inference mode has a few things that are advantages over no grad, which are discussed in
this thread here. But if we do this, we get very similar output to what we got before.
Grad function. But as you'll read in here and in the pie torch documentation, inference mode is
the favored way of doing inference for now. I just wanted to highlight this. So you can also do
something similar with torch dot no grad. However, inference mode is preferred. Alrighty. So I'm
just going to comment this out. So we just have one thing going on there. The main

```
takeaway
from this video is that when we're making predictions, we use the context manager
torch
dot inference mode. And right now, because our models variables or internal
parameters are
randomly initialized, our models predictions are as good as random. So they're
actually not too far
off where our values are. At least the red dots aren't like scattered all over
here. But in the
upcoming videos, we're going to be writing some pie torch training code to move
these values
closer to the green dots by looking at the training data here. So with that being
said,
I'll see you in the next video. Friends, welcome back. In the last video, we saw
that our model
performs pretty poorly. Like, ideally, these red dots should be in line with these
green dots.
And we know that because why? Well, it's because our model is initialized with
random parameters.
And I just want to put a little note here. You don't necessarily have to initialize
your model
with random parameters. You could initialize it with this could be zero. Yeah,
these two values,
weights can bias could be zero and you could go from there. Or you could also use
the parameters
from another model. But we're going to see that later on. That's something called
transfer learning.
That's just a little spoiler for what's to come. And so we've also discussed that
an ideal model
will replicate these known parameters. So in other words, start with random unknown
parameters,
these two values here. And then we want to write some code for our model to move
towards estimating
the ideal parameters here. Now, I just want to be explicit here and write down some
intuition
```

Section 57: Main Topics

Key Topics:

- We're about to get into training our very first machine learning model
- And so if we go pytorch loss functions, we're going to see that pytorch has a fair few loss functions built in
- We're learning something fun

▶ 📄 Click to view detailed content

before we jump into the training code. But this is very exciting. We're about to get into
training our very first machine learning model. So what's right here, the whole idea of training
is for a model to move from some unknown parameters, these may be random to some known parameters.
Or in other words, from a poor representation, representation of the data to a better representation
of the data. And so in our case, would you say that our models representation of the green dots
here with this red dots, is that a good representation? Or is that a poor representation?
I mean, I don't know about you, but I would say that to me, this is a fairly poor representation.
And one way to measure the representation between your models outputs, in our case, the red dots,
the predictions, and the testing data, is to use a loss function. So I'm going to write
this down here. This is what we're moving towards. We're moving towards training, but we need a
way to measure how poorly our models predictions are doing. So one way to measure how poor or how
wrong your models predictions are, is to use a loss function. And so if we go pytorch loss
functions, we're going to see that pytorch has a fair few loss functions built in. But the essence
of all of them is quite similar. So just wait for this to load my internet's going a little bit
slow today, but that's okay. We're not in a rush here. We're learning something fun.
If I search here for loss, loss functions, here we go. So yeah, this is torch in N. These are the
basic building blocks for graphs, whole bunch of good stuff in here, including loss functions.
Beautiful. And this is another thing to note as well, another one of those scenarios where
there's more words for the same thing. You might also see a loss function referred to as a criterion.
There's another word called cost function. So I might just write this down so you're aware of it.
Yeah, cost function versus loss function. And maybe some formal definitions about what all of these
are. Maybe they're used in different fields. But in the case of we're focused on machine learning,
right? So I'm just going to go note, loss function may also be called cost function or criterion in
different areas. For our case, we're going to refer to it as a loss function. And let's
just formally define a loss function here, because we're going to go through a fair few steps in
the upcoming videos. So this is a warning, nothing we can't handle. But I want to put some formal
definitions on things. We're going to see them in practice. That's what I prefer to do,
rather than just sit here defining stuff. This lecture has already had enough text on the page.

So hurry up and get into coding Daniel. A loss function is a function to measure how wrong your
models predictions are to the ideal outputs. So lower is better. So ideally, think of a measurement,
how could we measure the difference between the red dots and the green dots? One of the
simplest ways to do so would be just measure the distance here, right? So if we go, let's just
estimate this is 035 to 0.8. They're abouts. So what's the difference there? About 0.45.
Then we could do the same again for all of these other dots, and then maybe take the average of that.
Now, if you've worked with loss functions before, you might have realized that I've just
reproduced mean absolute error. But we're going to get to that in a minute. So we need a loss
function. I'm going to write down another little dot point here. This is just setting up intuition.
Things we need to train. We need a loss function. This is PyTorch. And this is machine learning
in general, actually. But we're focused on PyTorch. We need an optimizer. What does the optimizer do?
Takes into account the loss of a model and adjusts the model's parameters. So the parameters recall
our weight and bias values. Weight and biases. We can check those or bias. We can check those by
going model dot parameter or parameters. But I also like, oh, that's going to give us a generator,
isn't it? Why do we not define the model yet? What do we call our model? Oh, model zero. Excuse me.
I forgot where. I'm going to build a lot of models in this course. So we're giving them numbers.
Modeled up parameters. Yeah, we've got a generator. So we'll turn that into a list.
But model zero, if we want to get them labeled, we want state dict here.
There we go. So our weight is this value. That's a random value we've set. And there's the bias.
And now we've only got two parameters for our model. So it's quite simple. However, the principles
that we're learning here are going to be the same principles, taking a loss function,
trying to minimize it, so getting it to lower. So the ideal model will predict exactly what our
test data is. And an optimizer will take into account the loss and will adjust a model's parameter.
And our case weights and bias to be, let's finish this definition takes into account the
loss of a model and adjust the model's parameters, e.g. weight and bias, in our case, to improve the
loss function. And specifically, for PyTorch, we need a training loop and a testing loop.
Now, this is what we're going to work towards building throughout the next couple of videos.
We're going to focus on these two first, the loss function and optimizer. There's the formal
definition of those. You're going to find many different definitions. That's how I'm going to

Section 58: Main Topics

Key Topics:

- I mean, the whole course is fun
- But this is really exciting because training your first machine learning model seems a little bit like magic, but it's even more fun when you're writing the code yourself what's going on behind the scenes
- That is the whole idea of a training loop in PyTorch, or an optimization loop in PyTorch

▶ 📄 Click to view detailed content

```
find them. Loss function measures how wrong your model's predictions are, lower is
better,
optimizer takes into account the loss of your model. So how wrong it is, and starts
to move
these two values into a way that improves where these red dots end up. But these,
again, these
principles of a loss function and an optimizer can be for models with two
parameters or models
with millions of parameters, can be for computer vision models, or could be for
simple models like
ours that predict the dots on a straight line. So with that being said, let's jump
into the next
video. We'll start to look a little deeper into loss function, row problem, and an
optimizer.
I'll see you there. Welcome back. We're in the exciting streak of videos coming up
here. I mean,
the whole course is fun. Trust me. But this is really exciting because training
your first machine
learning model seems a little bit like magic, but it's even more fun when you're
writing the code
yourself what's going on behind the scenes. So we discussed that the whole concept
of training
is from going unknown parameters, random parameters, such as what we've got so far
to parameters that better represent the data. And we spoke of the concept of a loss
function.
We want to minimize the loss function. That is the whole idea of a training loop in
PyTorch,
or an optimization loop in PyTorch. And an optimizer is one of those ways that can
nudge the parameters of our model. In our case, weights or bias towards values
rather than just
being random values like they are now towards values that lower the loss function.
And if we
lower the loss function, what does a loss function do? It measures how wrong our
models
predictions are compared to the ideal outputs. So if we lower that, well, hopefully
we move
```

these red dots towards the green dots. And so as you might have guessed, PyTorch has some built
in functionality for implementing loss functions and optimizers. And by the way, what we're covering
so far is in the train model section of the PyTorch workflow fundamentals, I've got a little
nice table here, which describes a loss function. What does it do? Where does it live in PyTorch?
Common values, we're going to see some of these hands on. If you'd like to read about it,
of course, you have the book version of the course here. So loss functions in PyTorch,
I'm just in docstorch.nn. Look at this. Look at all these loss functions. There's far too many
for us to go through all in one hit. So we're just going to focus on some of the most common ones.
Look at that. We've got about what's our 15 loss functions, something like that? Well, truth be
told is that which one should use? You're not really going to know unless you start to work hands
on with different problems. And so in our case, we're going to be looking at L1 loss. And this is
an again, once more another instance where different machine learning libraries have different names
for the same thing, this is mean absolute error, which we kind of discussed in the last video,
which is if we took the distance from this red dot to this green dot and say at 0.4, they're about
0.4, 0.4, and then took the mean, well, we've got the mean absolute error. But in PyTorch,
they call it L1 loss, which is a little bit confusing because then we go to MSE loss,
which is mean squared error, which is L2. So naming conventions just takes a little bit of getting
used to this is a warning for you. So let's have a look at the L1 loss function. Again,
I'm just making you aware of where the other loss functions are. We'll do with some binary
cross entropy loss later in the course. And maybe even is that categorical cross entropy?
We'll see that later on. But all the others will be problem specific. For now, a couple of loss
functions like this, L1 loss, MSE loss, we use for regression problems. So that's predicting a number.
Cross entropy loss is a loss that you use with classification problems. But we'll see those hands
on later on. Let's have a look at L1 loss. So L1 loss creates a criterion. As I said, you might
hear the word criterion used in PyTorch for a loss function. I typically call them loss functions.
The literature typically calls it loss functions. That measures the mean absolute error. There we
go. L1 loss is the mean absolute error between each element in the input X and target Y. Now,
your extracurricular measure might have guessed is to read through the documentation for the
different loss functions, especially L1 loss. But for the sake of this video, let's

```
just implement
it for ourselves. Oh, and if you want a little bit of a graphic, I've got one here.
This is where
we're up to, by the way, picking a loss function optimizer for step two. This is a
fun part, right?
We're getting into training a model. So we've got mean absolute error. Here's that
graph we've
seen before. Oh, look at this. Okay. So we've got the difference here. I've
actually measured
this before in the past. So I kind of knew what it was. Mean absolute error is if
we repeat for
all samples in our set that we're working with. And if we take the absolute
difference between
these two dots, well, then we take the mean, we've got mean absolute error. So MAE
loss equals
torch mean we could write it out. That's the beauty of pine torch, right? We could
write this out.
Or we could use the torch and N version, which is recommended. So let's jump in.
There's a colorful
```

Section 59: Main Topics

Key Topics:

- Our objective for training a machine learning model will be two
- Often picking a loss function and optimizer and pytorch come as part of the same package because they work together
- So again, pytorch has torch

▶ 📄 Click to view detailed content

```
slide describing what we're about to do. So let's go set up a loss function. And
then we're also
going to put in here, set up an optimizer. So let's call it loss FN equals NN dot
L1 loss.
Simple as that. And then if we have a look at what's our loss function, what does
this say?
Oh my goodness. My internet is going quite slow today.
It's raining outside. So there might be some delays somewhere. But that's right.
Gives us a
chance to sit here and be mindful about what we're doing. Look at that. Okay. Loss
function.
L1 loss. Beautiful. So we've got a loss function. Our objective for training a
machine learning
model will be two. Let's go back. Look at the colorful graphic will be to minimize
these
distances here. And in turn, minimize the overall value of MAE. That is our goal.
If our red dots line up with our green dots, we will have a loss value of zero, the
```

ideal point
for a model to be. And so let's go here. We now need an optimizer. As we discussed before,
the optimizer takes into account the loss of a model. So these two work in tandem.
That's why I've put them as similar steps if we go back a few slides.
So this is why I put these as 2.1. Often picking a loss function and optimizer and pytorch
come as part of the same package because they work together. The optimizer's objective is to
give the model values. So parameters like a weight and a bias that minimize the loss function.
They work in tandem. And so let's see what an optimizer optimizes. Where might that be?
What if we search here? I typically don't use this search because I prefer just using Google
search. But does this give us optimizer? Hey, there we go. So again, pytorch has torch.optim
which is where the optimizers are. Torch.optim. Let me put this link in here.
This is another bit of your extracurricular. If you want to read more about different optimizers
in pytorch, as you might have guessed, they have a few. Torch.optim is a package implementing
various optimization algorithms. Most commonly used methods are already supported and the interface
is general enough so that more sophisticated ones can also be easily integrated into the future.
So if we have a look at what algorithms exist here, again, we're going to throw a lot of names
at you. But in the literature, a lot of them that have made it into here are already good working
algorithms. So it's a matter of picking whichever one's best for your problem. How do you find that
out? Well, SGD, stochastic gradient descent, is possibly the most popular. However, there are
some iterations on SGD, such as Adam, which is another one that's really popular. So again,
this is one of those other machine learning is part art, part science is trial and error of
figuring out what works best for your problem for us. We're going to start with SGD because
it's the most popular. And if you were paying attention to a previous video, you might have
seen that I said, look up gradient descent, wherever we got this gradient descent. There we go.
So this is one of the main algorithms that improves our models. So gradient descent and back
propagation. So if we have a look at this stochastic gradient descent, bit of a tongue twister,
is random gradient descent. So that's what stochastic means. So basically, our model
improves by taking random numbers, let's go down here, here, and randomly adjusting them
so that they minimize the loss. And once how optimizer, that's right here, once how optimizer
torch dot opt in, let's implement SGD, SGD stochastic gradient descent. We're going to write this here,
stochastic gradient descent. It starts by randomly adjusting these values. And once

```
it's found
some random values or random steps that have minimized the loss value, we're going
to see
this in action later on, it's going to continue adjusting them in that direction.
So say it says,
oh, weights, if I increase the weights, it reduces the loss. So it's going to keep
increasing the
weights until the weights no longer reduce the loss. Maybe it gets to a point at
say 0.65.
If you increase the weights anymore, the loss is going to go up. So the optimizer
is like,
well, I'm going to stop there. And then for the bias, the same thing happens. If it
decreases the
bias and finds that the loss increases, well, it's going to go, well, I'm going to
try increasing
the bias instead. So again, one last summary of what's going on here, a loss
function measures
how wrong our model is. And the optimizer adjust our model parameters, no matter
whether there's
two parameters or millions of them to reduce the loss. There are a couple of things
that
an optimizer needs to take in. It needs to take in as an argument, params. So this
is if we go to
SGD, I'm just going to link this as well. SGD, there's the formula of what SGD
does. I look at this
and I go, hmm, there's a lot going on here. And take me a while to understand that.
So I like to
see it in code. So we need params. This is short for what parameters should I
optimize as an optimizer.
And then we also need an LR, which stands for, I'm going to write this in a
comment, LR equals
learning rate, possibly the most, oh, I didn't even type rate, did I possibly the
most important
hyper parameter you can set? So let me just remind you, I'm throwing lots of words
out here, but I'm
kind of like trying to write notes about what we're doing. Again, we're going to
see these in action
```

Section 60: Main Topics

Key Topics:

- So learning rate equals possibly the most important learning hyper parameter
- I don't need learning there, do I
- And a hyper parameter is a value that us as a data scientist or a machine learning engineer set ourselves, you can set

▶ 📄 Click to view detailed content

in a second. So check out our models and parameters. So a parameter is a value that the model sets
itself. So learning rate equals possibly the most important learning hyper parameter. I don't
need learning there, do I? Hyper parameter. And a hyper parameter is a value that us as a data scientist
or a machine learning engineer set ourselves, you can set. So the learning rate is, in our case,
let's go 0.01. You're like, Daniel, where did I get this value from? Well, again, these type of
values come with experience. I think it actually says it in here, LR, LR 0.1. Yeah, okay, so the
default is 0.1. But then if we go back to Optim, I think I saw it somewhere. Did I see it somewhere?
0.0? Yeah, there we go. Yeah, so a lot of the default settings are pretty good in torch optimizers.
However, the learning rate, what does it actually do? We could go 0.01. These are all common values
here. Triple zero one. I'm not sure exactly why. Oh, model, it's model zero. The learning rate says
to our optimizer, yes, it's going to optimize our parameters here. But the higher the learning
rate, the more it adjusts each of these parameters in one hit. So let's say it's 0.01. And it's going
to optimize this value here. So it's going to take that big of a step. If we changed it to here,
it's going to take a big step on this three. And if we changed it to all the way to the end 0.01,
it's only going to change this value. So the smaller the learning rate, the smaller the change
in the parameter, the larger the learning rate, the larger the change in the parameter.
So we've set up a loss function. We've set up an optimizer. Let's now move on to the next step
in our training workflow. And that's by building a training loop. Far out. This is exciting. I'll
see you in the next video. Welcome back. In the last video, we set up a loss function. And we set
up an optimizer. And we discussed the roles of each. So loss function measures how wrong our model
is. The optimizer talks to the loss function and goes, well, if I change these parameters a certain
way, does that reduce the loss function at all? And if it does, yes, let's keep adjusting them in
that direction. If it doesn't, let's adjust them in the opposite direction. And I just want to show
you I added a little bit of text here just to concretely put down what we were discussing.
Inside the optimizer, you'll often have to set two parameters, params and lr, where params is
the model parameters you'd like to optimize for an example, in our case, params equals our model
zero parameters, which were, of course, a weight and a bias. And the learning rate, which is lr
in optimizer, lr stands for learning rate. And the learning rate is a hyper parameter. Remember,

a hyper parameter is a value that we the data scientist or machine learning engineer sets,
whereas a parameter is what the model sets itself defines how big or smaller optimizer changes
the model parameters. So a small learning rate, so the smaller this value results in small
changes, a large learning rate results in large changes. So another question might be,
well, very valid question. Hey, I put this here already, is which loss function and which optimizer
should I use? So this is another tough one, because it's problem specific. But with experience
and machine learning, I'm showing you one example here, you'll get an idea of what works for your
particular problem for a regression problem, like ours, a loss function of l1 loss, which is mai
and pytorch. And an optimizer like torch dot opt in slash s gd like sarcastic gradient descent
will suffice. But for a classification problem, we're going to see this later on.
Not this one specifically, whether a photo is a cat of a dog, that's just an example of a binary
classification problem, you might want to use a binary classification loss. But with that being
said, we now are moving on to, well, here's our whole goal is to reduce the MAE of our model.
Let's get the workflow. We've done these two steps. Now we want to build a training loop. So
let's get back into here. There's going to be a fair few steps going on. We've already covered
a few, but hey, nothing we can't handle together. So building a training loop in pytorch.
So I thought about just talking about what's going on in the training loop, but we can talk
about the steps after we've coded them. How about we do that? So we want to build a training loop
and a testing loop. How about we do that? So a couple of things we need in a training loop.
So there's going to be a fair few steps here if you've never written a training loop before,
but that is completely fine because you'll find that the first couple of times that you write this,
you'll be like, oh my gosh, there's too much going on here. But then when you have practice,
you'll go, okay, I see what's going on here. And then eventually you'll write them with your
eyes closed. I've got a fun song for you to help you out remembering things. It's called the
unofficial pytorch optimization loop song. We'll see that later on, or actually, I'll probably leave
that as an extension, but you'll see that you can also functionize these things, which we will do
later in the course so that you can just write them once and then forget about them. But we're
going to write it all from scratch to begin with so we know what's happening. So we want to,

Section 61: Main Topics

Key Topics:

- Many people, one of the main things I get asked from machine learning is how do I learn machine learning if I didn't do math
- Well, the beautiful thing about PyTorch is that it implements a lot of the math of back propagation
- So these two algorithms drive the majority of our learning

▶ 📄 Click to view detailed content

```
or actually step zero, is loop through the data. So we want to look at the data
multiple times
because our model is going to, at first, start with random predictions on the data,
make some
predictions. We're trying to improve those. We're trying to minimize the loss to
make those
predictions. We do a forward pass. So forward pass. Why is it called a forward
pass? So this
involves data moving through our model's forward functions. Now that I say
functions because there
might be plural, there might be more than one. And the forward method recall, we
wrote in our model
up here. Ford. A forward pass is our data going through this function here. And if
you want to
look at it visually, let's look up a neural network graphic. Images, a forward pass
is just
data moving from the inputs to the output layer. So starting here input layer
moving through the
model. So that's a forward pass, also called forward propagation. Another time
we'll have
more than one name is used for the same thing. So we'll go back down here, forward
pass. And
I'll just write here also called forward propagation, propagation. Wonderful. And
then we need to
calculate the loss. So forward pass. Let me write this. To calculate or to make
predictions,
make predictions on data. So calculate the loss, compare forward pass predictions.
Oh, there's
an undergoing in the background here of my place. We might be in for a storm.
Perfect time to write
code, compare forward pass predictions to ground truth labels. We're going to see
all this in code
in a second, calculate the loss. And then we're going to go optimise a zero grad.
We haven't
spoken about what this is, but that's okay. We're going to see that in a second.
I'm not going to
put too much there. Loss backward. We haven't discussed this one either. There's
probably three
steps that we haven't really discussed. We've discussed the idea behind them, but
```

not too much
in depth. Optimise our step. So this one is loss backwards is move backwards. If the forward pass
is forwards, like through the network, the forward pass is data goes into out. The backward pass
data goes, our calculations happen backwards. So we'll see what that is in a second. Where were
we over here? We've got too much going on. I'm getting rid of these moves backwards through
the network to calculate the gradients. Oh, oh, the gradients of each of the parameters
of our model with respect to the loss. Oh my gosh, that is an absolute mouthful,
but that'll do for now. Optimise a step. This is going to use the optimiser to adjust our
model's parameters to try and improve the loss. So remember how I said in a previous video
that I'd love you to watch the two videos I linked above. One on gradient descent and one
on back propagation. If you did, you might have seen like there's a fair bit of math going on in
there. Well, that's essentially how our model goes from random parameters to better parameters,
using math. Many people, one of the main things I get asked from machine learning is how do I
learn machine learning if I didn't do math? Well, the beautiful thing about PyTorch is that it
implements a lot of the math of back propagation. So this is back propagation. I'm going to write
this down here. This is an algorithm called back, back propagation, hence the loss backward. We're
going to see this in code in a second, don't you worry? And this is gradient descent. So these
two algorithms drive the majority of our learning. So back propagation, calculate the gradients of
the parameters of our model with respect to the loss function and optimise our step,
we'll trigger code to run gradient descent, which is to minimise the gradients because what is a
gradient? Let's look this up. What is a gradient? I know we haven't written a code yet, but we're
going to do that. Images. Gradient, there we go. Changing y, changing x. Gradient is from high
school math. Gradient is a slope. So if you were on a hill, let's find a picture of a hill.
Picture of a hill. There we go. This is a great big hill. So if you were on the top of this hill,
and you wanted to get to the bottom, how would you get to the bottom? Well, of course, you just
walked down the hill. But if you're a machine learning model, what are you trying to do? Let's
imagine your loss is the height of this hill. You start off with your losses really high, and you
want to take your loss down to zero, which is the bottom, right? Well, if you measure the gradient
of the hill, the bottom of the hill is in the opposite direction to where the gradient is steep.
Does that make sense? So the gradient here is an incline. We want our model to move

```
towards the
gradient being nothing, which is down here. And you could argue, yeah, the
gradient's probably
nothing up the top here, but let's just for argument's sake say that we want to get
to the
bottom of the hill. So we're measuring the gradient, and one of the ways an
optimisation algorithm
works is it moves our model parameters so that the gradient equals zero, and then
if the gradient
of the loss equals zero, while the loss equals zero two. So now let's write some
code. So we're
going to set up a parameter called or a variable called epochs. And we're going to
start with one,
even though this could be any value, let me define these as we go. So we're going
to write code to
```

Section 62: Main Topics

Key Topics:

- And I know you could argue that, hey, our machine learning parameters of model zero, or our model parameters, model zero aren't actually parameters, because we've set them
- So pytorch models have a couple of different modes
- So what does train mode do in a pytorch model

▶ 📄 Click to view detailed content

```
do all of this. So epochs, an epoch is one loop through the data dot dot dot. So
epochs, we're
going to start with one. So one time through all of the data, we don't have much
data. And so
for epoch, let's go this, this is step zero, zero, loop through the data. By the
way, when I say
loop through the data, I want you to do all of these steps within the loop. And do
dot dot dot
loop through the data. So for epoch in range epochs, even though it's only going to
be one,
we can adjust this later. And because epochs, we've set this ourselves, it is a,
this is a hyper parameter, because we've set it ourselves. And I know you could
argue that,
hey, our machine learning parameters of model zero, or our model parameters, model
zero aren't
actually parameters, because we've set them. But in the models that you build in
the future,
they will likely be set automatically rather than you setting them explicitly like
we've done when
we created model zero. And oh my gosh, this is taking quite a while to run. That's
```

all right.
We don't need it to run fast. We just, we need to write some more code, then you'll come on.
There's a step here I haven't discussed either. Set the model to training mode. So pytorch models
have a couple of different modes. The default is training mode. So we can set it to training
mode by going like this. Train. So what does train mode do in a pytorch model? My goodness.
Is there a reason my engineer is going this slide? That's all right. I'm just going to
discuss this with talking again list. Train mode. Train mode in pytorch sets. Oh, there we go.
Requires grad equals true. Now I wonder if we do with torch dot no grad member no grad is similar
to inference mode. Will this adjust? See, I just wanted to take note of requires grad equals
true. Actually, what I might do is we do this in a different cell. Watch this. This is just going
to be rather than me just spit words at you. I reckon we might be able to get it work in doing
this. Oh, that didn't list the model parameters. Why did that not come out? Model zero dot eval.
So there's two modes of our mode and train mode model dot eval parameters. Hey, we're experimenting
together on the fly here. And actually, this is what I want you to do is I want you to experiment
with different things. It's not going to say requires grad equals false. Hmm. With torch dot no
grad. Model zero dot parameters. I don't know if this will work, but it definitely works behind
the scenes. And what I mean by works behind the scenes are not here. It works behind the scenes
when calculations have been made, but not if we're trying to explicitly print things out.
Well, that's an experiment that I thought was going to work and it didn't work. So train
mode in pytorch sets all parameters that require gradients to require gradients.
So do you remember with the picture of the hill? I spoke about how we're trying to minimize the
gradient. So the gradient is the steepness of the hill. If the height of the hill is a loss function
and we want to take that down to zero, we want to take the gradient down to zero. So same thing
with the gradients of our model parameters, which are here with respect to the loss function,
we want to try and minimize that gradient. So that's gradient descent is take that gradient down to
zero. So model dot train. And then there's also model zero dot a vowel. So turns off gradient
tracking. So we're going to see that later on. But for now, I feel like this video is getting far
too long. Let's finish the training loop in the next video. I'll see you there.
Friends, welcome back. In the last video, I promised a lot of code, but we didn't get there. We
discussed some important steps. I forgot how much behind the scenes there is to apply towards training

```
loop. And I think it's important to spend the time that we did discussing what's
going on,
because there's a fair few steps. But once you know what's going on, I mean, later
on, we don't
have to write all the code that we're going to write in this video, you can
functionize it. We're
going to see that later on in the course, and it's going to run behind the scenes
for us. But we're
spending a fair bit of time here, because this is literally the crux of how our
model learns. So
let's get into it. So now we're going to implement the forward pass, which involves
our model's
forward function, which we defined up here. When we built our model, the forward
pass runs through
this code here. So let's just write that. So in our case, because we're training,
I'm just
going to write here. This is training. We're going to see dot of our later on.
We'll talk
about that when it comes. Let's do the forward pass. So the forward pass, we want
to pass data
through our model's forward method. We can do this quite simply by going y pred. So
y predictions,
because remember, we're trying to use our ideal model is using x test to predict y
test
on our test data set. We make predictions on our test data set. We learn on our
training data set.
So we're passing, which is going to get rid of that because we don't need that. So
we're
passing our model x train and model zero is going to be our current model. There we
go. So we learn
```

Section 63: Main Topics

Key Topics:

- And in our case, we're going to set loss equal to our loss function, which is L one loss in PyTorch, but it is MAE
- But in the case of staying true to the documentation, let's do inputs first and then targets next for the rest of the course
- And now the reason why it accumulates, that's pretty deep in the pytorch documentation

▶ 📄 Click to view detailed content

```
patterns on the training data to evaluate our model on the test data. Number two,
where we are.
So we have to calculate the loss. Now, in a previous video, we set up a loss
```

function.
So this is going to help us calculate the what what kind of loss are we using? We want to calculate
the MAE. So the difference or the distance between our red dot and a green dot. And the formula would
be the same if we had 10,000 red dots and 10,000 green dots, we're calculating how far they are
apart. And then we're taking the mean of that value. So let's go back here. So calculate the loss.
And in our case, we're going to set loss equal to our loss function, which is L one loss in
PyTorch, but it is MAE. Y-pred and Y-train. So we're calculating the difference between our models
predictions on the training data set and the ideal training values. And if you want to go into
torch dot NN loss functions, that's going to show you the order because sometimes this confuses me
to what order the values go in here, but it goes prediction first, then labels and I may be wrong
there because I get confused here. My dyslexia kicks in, but I'm pretty sure it's predictions first,
then actual labels. Do we have an example of where it's used? Yeah, import first, target next.
So there we go. And truth be told, because it's mean absolute error, it shouldn't actually matter
too much. But in the case of staying true to the documentation, let's do inputs first and then
targets next for the rest of the course. Then we're going to go optimizer zero grad. Hmm,
haven't discussed this one, but that's okay. I'm going to write the code and then I'm going to
discuss what it does. So what does this do? Actually, before we discuss this, I'm going to write
these two steps because they kind of all work together. And it's a lot easier to discuss what
optimizer zero grad does in the context of having everything else perform back propagation
on the loss with respect to the parameters of the model. Back propagation is going to take
the loss value. So lost backward, I always say backwards, but it's just backward. That's the code
there. And then number five is step the optimizer. So perform gradient descent. So optimizer dot
step. Oh, look at us. We just wrote the five major steps of a training loop. Now let's discuss
how all of these work together. So it's kind of strange, like the ordering of these, you might
think, Oh, what should I do the order? Typically the forward pass and the loss come straight up.
Then there's a little bit of ambiguity around what order these have to come in. But the optimizer
step should come after the back propagation. So I just like to keep this order how it is because
this works. Let's just keep it that way. But what happens here? Well, it also is a little bit
confusing in the first iteration of the loop because we've got zero grad. But what happens here is

```
that the optimizer makes some calculations in how it should adjust model parameters
with regards to
the back propagation of the loss. And so by default, these will by default, how the
optimizer
changes will accumulate through the loop. So we have to zero them above in step
three
for the next iteration of the loop. So a big long comment there. But what this is
saying is,
let's say we go through the loop and the optimizer chooses a value of one, change
it by one. And
then it goes through a loop again, if we didn't zero it, if we didn't take it to
zero, because
that's what it is doing, it's going one to zero, it would go, okay, next one, two,
three, four,
five, six, seven, eight, all through the loop, right? Because we're looping here.
If this was
10, it would accumulate the value that it's supposed to change 10 times. But we
want it to start
again, start fresh each iteration of the loop. And now the reason why it
accumulates, that's
pretty deep in the pytorch documentation. From my understanding, there's something
to do with
like efficiency of computing. If you find out what the exact reason is, I'd love to
know.
So we have to zero it, then we perform back propagation. If you recall, back
propagation is
discussed in here. And then with optimizer step, we form gradient descent. So the
beauty of pytorch,
this is the beauty of pytorch, is that it will perform back propagation, we're
going to have a
look at this in second, and gradient descent for us. So to prevent this video from
getting too long,
I know we've just written code, but I would like you to practice writing a training
loop
yourself, just write this code, and then run it and see what happens. Actually, you
can comment
this out, we're going to write the testing loop in a second. So your extra
curriculum for this
video is to, one, rewrite this training loop, is to, two, sing the pytorch
optimization loop
song, let's go into here. If you want to remember the steps, well, I've got a song
for you. This is
the training loop song, we haven't discussed the test step, but maybe you could try
this yourself.
So this is an old version of the song, actually, I've got a new one for you. But
let's sing this
together. It's training time. So we do the forward pass, calculate the loss,
optimise a zero grad,
loss backwards, optimise a step, step, step. Now you only have to call optimise a
step once,
this is just for jingle purposes. But for test time, let's test with torch no grad,
do the forward
```

Section 64: Main Topics

Key Topics:

- And if you want the video version of it, well, you're just going to have to search unofficial pytorch optimisation loop song
- In the last few videos, we've been discussing the steps in a training loop in pytorch
- Essentially, it sets up a whole bunch of settings behind the scenes in our model parameters so that they can track the gradients and do a whole bunch of learning behind the scenes with these functions down here

▶ 📄 Click to view detailed content

```
pass, calculate the loss, watch it go down, down, down. That's from my Twitter, but this is a way
that I help myself remember the steps that are going on in the code here. And if you want the
video version of it, well, you're just going to have to search unofficial pytorch optimisation loop
song. Oh, look at that, who's that guy? Well, he looks pretty cool. So I'll let you check that
out in your own time. But for now, go back through the training loop steps. I've got a colorful
graphic coming up in the next video, we're going to write the testing steps. And then we're going
to go back one more time and talk about what's happening in each of them. And again, if you'd
like some even more extra curriculum, don't forget the videos I've shown you on back propagation
and gradient descent. But for now, let's leave this video here. I'll see you in the next one.
Friends, welcome back. In the last few videos, we've been discussing the steps in a training
loop in pytorch. And there's a fair bit going on. So in this video, we're going to step back
through what we've done just to recap. And then we're going to get into testing. And it's nice
and early where I am right now. The sun's about to come up. It's a very, very beautiful morning
to be writing code. So let's jump in. We've got a little song here for what we're doing in the
training steps. For an epoch in a range, comodel.train, do the forward pass, calculate the loss of the
measure zero grad, last backward of the measure step step step. That's the little jingle I use to
remember the steps in here, because the first time you write it, there's a fair bit going on.
But subsequent steps and subsequent times that you do write it, you'll start to memorize this.
```

And even better later on, we're going to put it into a function so that we can just call it
over and over and over and over again. With that being said, let's jump in to a colorful slide,
because that's a lot of code on the page. Let's add some color to it, understand what's happening.
That way you can refer to this and go, Hmm, I see what's going on now. So for the loop, this is why
it's called a training loop. We step through a number of epochs. One epoch is a single forward
pass through the data. So pass the data through the model for a number of epochs. Epox is a
hyper parameter, which means you could set it to 100, you could set it to 1000, you could set it
to one as we're going to see later on in this video. We skip this step with the colors, but
we put the model in we call model.train. This is the default mode that the model is in.
Essentially, it sets up a whole bunch of settings behind the scenes in our model parameters so that
they can track the gradients and do a whole bunch of learning behind the scenes with these
functions down here. PyTorch does a lot of this for us. So the next step is the forward pass.
We perform a forward pass on the training data in the training loop. This is an important note.
In the training loop is where the model learns patterns on the training data. Whereas in the
testing loop, we haven't got to that yet is where we evaluate the patterns that our model has learned
or the parameters that our model has learned on unseen data. So we pass the data through the model,
this will perform the forward method located within the model object. So because we created
a model object, you can actually call your models whatever you want, but it's good practice to
you'll often see it just called model. And if you remember, we'll go back to the code.
We created a forward method in our model up here, which is this, because our linear regression model,
class, subclasses, nn.module, we need to create our own custom forward method. So that's why it's
called a forward pass is because not only does it, well, the technical term is forward propagation.
So if we have a look at a neural network picture, forward propagation just means going through
the network from the input to the output, there's a thing called back propagation, which we're going
to discuss in a second, which happens when we call loss.backward, which is going backward through
the model. But let's return to our colorful slide. We've done the forward pass, call a forward method,
which performs some calculation on the data we pass it. Next is we calculate the loss value,
how wrong the model's predictions are. And this will depend on what loss function you use,
what kind of predictions your model is outputting, and what kind of true values you

```
have.
But that's what we're doing here. We're comparing our model's predictions on the
training data
to what they should ideally be. And these will be the training labels. The next
step, we zero
the optimizer gradients. So why do we do this? Well, it's a little confusing for
the first epoch in
the loop. But as we get down to optimizer dot step here, the gradients that the
optimizer
calculates accumulate over time so that for each epoch for each loop step, we want
them to go back
to zero. And now the exact reason behind why the optimizer accumulates gradients is
buried somewhere
within the pie torch documentation. I'm not sure of the exact reason from memory.
It's because of
compute optimization. It just adds them up in case you wanted to know what they
were. But if
```

Section 65: Main Topics

Key Topics:

- Now we want it in machine learning
- What's it going to give us in machine learning, a gradient is a derivative of a function that has more than one input variable
- And then we take a learning step

▶ 📄 Click to view detailed content

```
you find out exactly, I'd love to know. Next step is to perform back propagation on
the loss function.
That's what we're calling loss. backward. Now back propagation is we compute the
gradient of
every parameter with requires grad equals true. And if you recall, we go back to
our code.
We've set requires grad equals true for our parameters. Now the reason we've set
requires
grad equals true is not only so back propagation can be performed on it. But let me
show you what
the gradients look like. So let's go loss function curve. That's a good idea. So
we're looking for
so we're looking for some sort of convex curve here. There we go. L two loss. We're
using L one loss
at the moment. Is there a better one here? All we need is just a nice looking
curve. Here we go.
So this is why we keep track of the gradients behind the scenes. Pie torch is going
to create
some sort of curve for all of our parameters that looks like this. Now this is just
```

a 2d plot.
So the reason why we're just using an example from Google images is one, because you're going to
spend a lot of your time Googling different things. And two, in practice, when you have your own
custom neural networks, right now we only have two parameters. So it's quite easy to visualize a
loss function curve like this. But when you have say 10 million parameters, you basically can't
visualize what's going on. And so pie torch again will take care of these things behind the scenes.
But what it's doing is when we say requires grad pie torch is going to track the gradients
of each of our parameters. And so what we're trying to do here with back propagation and
subsequently gradient descent is calculate where the lowest point is. Because this is a loss function,
this is MSC loss, we could trade this out to be MAE loss in our case or L1 loss for our specific
problem. But this is some sort of parameter. And we calculate the gradients because what is the
gradient? Let's have a look. What is a gradient? A gradient is an inclined part of a road or railway.
Now we want it in machine learning. What's it going to give us in machine learning, a gradient
is a derivative of a function that has more than one input variable. Okay, let's dive in a little
deeper. See, here's some beautiful loss landscapes. We're trying to get to the bottom of here. This
is what gradient descent is all about. So oh, there we go. So this is a cost function, which is also a
loss function. We start with a random initial variable. What have we done? We started with a
random initial variable. Right? Okay. And then we take a learning step. Beautiful. This is W. So
this could be our weight parameter. Okay, we're connecting the dots here. This is exciting.
We've got a lot of tabs here, but that's all right. We'll bring this all together in a second.
And what we're trying to do is come to the minimum. Now, why do we need to calculate the gradients?
Well, the gradient is what? Oh, value of weight. Here we go. This is even better.
I love Google images. So this is our loss. And this is a value of a weight. So we calculate the
gradients. Why? Because the gradient is the slope of a line or the steepness. And so if we
calculate the gradient here, and we find that it's really steep right up the top of this,
this incline, we might head in the opposite direction to that gradient. That's what gradient
descent is. And so if we go down here, now, what are these step points? There's a little thing that
I wrote down in the last video at the end of the last video I haven't told you about yet,
but I was waiting for a moment like this. And if you recall, I said kind of all of these three steps
optimizes zero grad loss backward, optimizes step are all together. So we calculate

```
the
gradients because we want to head in the opposite direction of that gradient to get
to a gradient
value of zero. And if we get to a gradient value of zero with a loss function,
well, then the loss
is also zero. So that's why we keep track of a gradient with requires grad equals
true.
And again, PyTorch does a lot of this behind the scenes. And if you want to dig
more into
what's going on here, I'm going to show you some extra resources for back
propagation,
which is calculating this gradient curve here, and gradient descent, which is
finding the bottom
of it towards the end of this video. And again, if we started over this side, we
would just go
in the opposite direction of this. So maybe this is a positive gradient here, and
we just go in the
opposite direction here. We want to get to the bottom. That is the main point of
gradient descent.
And so if we come back, I said, just keep this step size in mind here. If we come
back to where
we created our loss function and optimizer, I put a little tidbit here for the
optimizer.
Because we've written a lot of code, and we haven't really discussed what's going
on, but
I like to do things on the fly as we need them. So inside our optimizer, we'll have
main two
parameters, which is params. So the model parameters you'd like to optimize,
params equals model zero dot parameters in our case. And then PyTorch is going to
create
something similar to this curve, not visually, but just mathematically behind the
scenes for
```

Section 66: Main Topics

Key Topics:

- That's the beauty of PyTorch
- Then within the optimizer, once we've told it what parameters to optimize, we have the learning rate
- So the learning rate is another hyper parameter that defines how big or small the optimizer changes the parameters with each step

▶ 📄 Click to view detailed content

```
every parameter. Now, this is a value of weight. So this would just be potentially
the weight
parameter of our network. But again, if you have 10 million parameters, there's no
```

way you could
just create all of these curves yourself. That's the beauty of PyTorch. It's doing
this behind the
scenes through a mechanism called torch autograd, which is auto gradient
calculation. And there's
beautiful documentation on this. If you'd like to read more on how it works, please
go through
that. But essentially behind the scenes, it's doing a lot of this for us for each
parameter.
That's the optimizer. Then within the optimizer, once we've told it what parameters
to optimize,
we have the learning rate. So the learning rate is another hyper parameter that
defines how big or
small the optimizer changes the parameters with each step. So a small learning rate
results in
small changes, whereas a large learning rate is in large changes. And so if we look
at this
curve here, we might at the beginning start with large steps, so we can get closer
and closer to
the bottom. But then as we get closer and closer to the bottom, to prevent stepping
over to this
side of the curve, we might do smaller and smaller steps. And the optimizer in
PyTorch,
there are optimizers that do that for us. But there is also another concept called
learning
rate scheduling, which is, again, something if you would like to look up and do
more. But
learning rate scheduling essentially says, hey, maybe start with some big steps.
And then as we
get closer and closer to the bottom, reduce how big the steps are that we take.
Because if you've
ever seen a coin, coin at the back of couch. This is my favorite analogy for this.
If you've ever
tried to reach a coin at the back of a couch, like this excited young chap, if
you're reaching
towards the back of a couch, you take quite big steps as you say your arm was over
here,
you would take quite big steps until you get to about here. And in the closer you
get to the coin,
the smaller and smaller your steps are. Otherwise, what's going to happen? The coin
is going to be
lost. Or if you took two small steps, you'd never get to the coin. It would take
forever to get there.
So that's the concept of learning rate. If you take two big steps, you're going to
just end up
over here. If you take two small steps, it's going to take you forever to get to
the bottom here.
And this bottom point is called convergence. That's another term you're going to
come across. I
know I'm throwing a lot of different terms at you, but that's the whole concept of
the learning
rate. How big is your step down here? In gradient descent. Gradient descent is
this. Back propagation
is calculating these derivative curves or the gradient curves for each of the
parameters in our
model. So let's get out of here. We'll go back to our training steps. Where were
we? I think we're

up to back propagation. Have we done backward? Yes. So the back propagation is where we do the
backward steps. So the forward pass, forward propagation, go from input to output. Back propagation,
we take the gradients of the loss function with respect to each parameter in our model
by going backwards. That's what happens when we call loss.backward. PyTorch does that for us
behind the scenes. And then finally, step number five is step the optimizer. We've kind of discussed
that. As I said, if we take a step, let's get our loss curve back up. Loss function curve.
Doesn't really matter what curve we use. The optimizer step is taking a step this way to try
and optimize the parameters so that we can get down to the bottom here. And I also just noted
here that you can turn all of this into a function so we don't necessarily have to remember to
write these every single time. The ordering of this, you'll want to do the forward pass first.
And then calculate the loss because you can't calculate the loss unless you do the forward pass.
I like this ordering here of these three as well. But you also want to do the optimizer step
after the loss backward. So this is my favorite ordering. It works. If you like this ordering,
you can take that as well. With that being said, I think this video has gotten long enough.
In the next video, I'd like to step through this training loop one epoch at a time so that we can
see, I know I've just thrown a lot of words at you that this optimizer is going to try and
optimize our parameters each step. But let's see that in action how our parameters of our model
actually change every time we go through each one of these steps. So I'll see you in the next video.
Let's step through our model. Welcome back. And we've spent a fair bit of time on the training loop
and the testing loop. Well, we haven't even got to that yet, but there's a reason behind this,
because this is possibly one of the most important things aside from getting your data ready,
which we're going to see later on in PyTorch deep learning is writing the training loop,
because this is literally like how your model learns patterns and data. So that's why we're
spending a fair bit of time on here. And we'll get to the testing loop, because that's how you
evaluate the patterns that your model has learned from data, which is just as important as learning
the patterns themselves. And following on from the last couple of videos, I've just linked some

Section 67: Main Topics

Key Topics:

- Remember, this course focuses on writing PyTorch code
- But if you'd like to dive into what math PyTorch is triggering behind the scenes, I'd highly recommend these two videos
- But with experience over time, you work with machine learning problems, you write a lot of code, you get an idea of what works and what doesn't with your particular problem set

▶ 📄 Click to view detailed content

```
YouTube videos that I would recommend for extra curriculum for back propagation,
which is what happens when we call loss stop backward down here. And for the
optimizer step,
gradient descent is what's happening there. So I've linked some extra resources for
what's going
on behind the scenes there from a mathematical point of view. Remember, this course
focuses on
writing PyTorch code. But if you'd like to dive into what math PyTorch is
triggering behind the
scenes, I'd highly recommend these two videos. And I've also added a note here as
to which
loss function and optimizer should I use, which is a very valid question. And
again,
it's another one of those things that's going to be problem specific. But with
experience over time,
you work with machine learning problems, you write a lot of code, you get an idea
of what works
and what doesn't with your particular problem set. For example, like a regression
problem,
like ours, regression is again predicting a number. We use MAE loss, which PyTorch
causes
L1 loss. You could also use MSE loss and an optimizer like torch opt-in stochastic
gradient
descent will suffice. But for classification, you might want to look into a binary
classification,
a binary cross entropy loss, but we'll look at a classification problem later on in
the course.
For now, I'd like to demonstrate what's going on in the steps here. So let's go
model zero.
Let's look up the state dict and see what the parameters are for now.
Now they aren't the original ones I don't think. Let's re-instantiate our model so
we get
re new parameters. Yeah, we recreated it here. I might just get rid of that. So
we'll rerun our
model code, rerun model state dict. And we will create an instance of our model and
just make
sure your parameters should be something similar to this. If it's not exactly like
```

that, it doesn't
matter. But yeah, I'm just going to showcase you'll see on my screen what's going on anyway.
State dict 3367 for the weight and 012888 for the bias. And again, I can't stress enough. We've
only got two parameters for our model and we've set them ourselves future models that you build
and later ones in the course will have much, much more. And we won't actually explicitly set any
of them ourselves. We'll check out some predictions. They're going to be terrible because we're using
random parameters to begin with. But we'll set up a new loss function and an optimizer. Optimizer
is going to optimize our model zero parameters, the weight and bias. The learning rate is 0.01,
which is relatively large step. That would be a bit smaller. Remember, the larger the learning rate,
the bigger the step, the more the optimizer will try to change these parameters every step.
But let's stop talking about it. Let's see it in action. I've set a manual seed here too, by the way,
because the optimizer steps are going to be quite random as well, depending on how the models
predictions go. But this is just to try and make it as reproduces possible. So keep this in mind,
if you get different values to what we're going to output here from my screen to your screen,
don't worry too much. What's more important is the direction they're going. So ideally,
we're moving these values here. This is from we did one epoch before. We're moving these values
closer to the true values. And in practice, you won't necessarily know what the true values are.
But that's where evaluation of your model comes in. We're going to cover that when we write a
testing loop. So let's run one epoch. Now I'm going to keep that down there. Watch what happens.
We've done one epoch, just a single epoch. We've done the forward pass. We've calculated the loss.
We've done optimizer zero grad. We've performed back propagation. And we've stepped the optimizer.
What is stepping the optimizer do? It updates our model parameters to try and get them further
closer towards the weight and bias. If it does that, the loss will be closer to zero. That's what
it's trying to do. How about we print out the loss at the same time. Print loss and the loss.
Let's take another step. So the loss is 0301. Now we check the weights and the bias. We've changed
again three, three, four, four, five, one, four, eight, eight. We go again. The loss is going down.
Check it. Hey, look at that. The values are getting closer to where they should be if over so slightly.
Loss went down again. Oh my goodness, this is so amazing. Look, we're training our,
let's print this out in the same cell. Print our model. State dict. We're training our first
machine learning model here, people. This is very exciting, even if it's only step

```
by step and it's
only a small model. This is very important. Loss is going down again. Values are
getting closer to
where they should be. Again, we won't really know where they should be in real
problems, but for
now we do. So let's just get excited. The real way to sort of measure your model's
progress and
practice is a lower loss value. Remember, lower is better. A loss value measures
how wrong your
model is. We're going down. We're going in the right direction. So that's what I
meant by,
as long as your values are going in the similar direction. So down, we're writing
similar code
here, but if your values are slightly different in terms of the exact numbers,
don't worry too
much because that's inherent to the randomness of machine learning, because the
steps that the
optimizer are taking are inherently random, but they're sort of pushed in a
direction.
So we're doing gradient descent here. This is beautiful. How low can we get the
loss? How about
```

Section 68: Main Topics

Key Topics:

- We've measured the gradient pytorch has done that behind the scenes for us
- Thank you pytorch
- We're training our first machine learning model

▶ 📄 Click to view detailed content

```
we try to get to 0.1? Look at that. We're getting close to 0.1. And then, I mean,
we don't have to
do this hand by hand. The bias is getting close to where it exactly should be.
We're below 0.1.
Beautiful. So that was only about, say, 10 passes through the data, but now you're
seeing it in
practice. You're seeing it happen. You're seeing gradient descent. Let's go
gradient descent work
in action. We've got images. This is what's happening. We've got our cost function.
J is
another term for cost function, which is also our loss function. We start with an
initial weight.
What have we done? We started with an initial weight, this value here. And what are
we doing?
We've measured the gradient pytorch has done that behind the scenes for us. Thank
you pytorch.
And we're taking steps towards the minimum. That's what we're trying to do. If we
```

minimize the
gradient of our weight, we minimize the cost function, which is also a loss function. We could
keep going here for hours and get as long as we want. But my challenge for you, or actually,
how about we make some predictions with our model we've got right now? Let's make some predictions.
So with torch dot inference mode, we'll make some predictions together. And then I'm going
to set you a challenge. How about you run this code here for 100 epochs after this video,
and then you make some predictions and see how that goes. So why preds? Remember how
poor our predictions are? Why preds new equals, we just do the forward pass here. Model zero
on the test data. Let's just remind ourselves quickly of how poor our previous predictions were.
Plot predictions, predictions equals y. Do we still have this saved? Why preds?
Hopefully, this is still saved. There we go. Shocking predictions, but we've just done 10 or so
epochs. So 10 or so training steps have our predictions. Do they look any better? Let's run
this. We'll copy this code. You know my rule. I don't really like to copy code, but in this case,
I just want to exemplify a point. I like to write all the code myself. What do we got? Why preds
new? Look at that. We are moving our predictions close at the red dots closer to the green dots.
This is what's happening. We're reducing the loss. In other words, we're reducing the difference
between our models predictions and our ideal outcomes through the power of back propagation
and gradient descent. So this is super exciting. We're training our first machine learning model.
My challenge to you is to run this code here. Change epochs to 100. See how low you can get this
loss value and run some predictions, plot them. And I think it's time to start testing. So give
that a go yourself, and then we'll write some testing code in the next video. I'll see you there.
Welcome back. In the last video, we did something super excited. We saw our loss go down. So the
loss is remember how different our models predictions are to what we'd ideally like them. And we saw
our model update its parameters through the power of back propagation and gradient descent, all
taken care of behind the scenes for us by PyTorch. So thank you, PyTorch. And again, if you'd like
some extra resources on what's actually happening from a math perspective for back propagation and
gradient descent, I would refer to you to these. Otherwise, this is also how I learn about things.
Gradient descent. There we go. How does gradient descent work? And then we've got back propagation.
And just to reiterate, I am doing this and just Googling these things because that's what you're
going to do in practice. You're going to come across a lot of different things that

```
aren't
covered in this course. And this is seriously what I do day to day as a machine
learning engineer
if I don't know what's going on. Just go to Google, read, watch a video, write some
code,
and then I build my own intuition for it. But with that being said, I also issued
you the challenge
of trying to run this training code for 100 epochs. Did you give that a go? I hope
you did. And
how low did your loss value? Did the weights and bias get anywhere close to where
they should have
been? How do the predictions look? Now, I'm going to save that for later on,
running this code for
100 epochs. For now, let's write some testing code. And just a note, you don't
necessarily have to
write the training and testing loop together. You can functionize them, which we
will be doing later
on. But for the sake of this intuition, building and code practicing and first time
where we're
writing this code together, I'm going to write them together. So testing code, we
call model.ofour,
what does this do? So this turns off different settings in the model not needed for
evaluation
slash testing. This can be a little confusing to remember when you're writing
testing code. But
we're going to do it a few times until it's habit. So just make it a habit. If
you're training your
model, call model dot train to make sure it's in training mode. If you're testing
or evaluating
your model. So that's what a vowel stands for evaluate, call model dot a vowel. So
it turns off
different settings in the model not needed for evaluation. So testing, this is
things like drop
out. We haven't seen what drop out is slash batch norm layers. But if we go into
torch dot
and end, I'm sure you'll come across these things in your future machine learning
endeavors. So drop
```

Section 69: Main Topics

Key Topics:

- So we discussed that if parameters in our model have requires grad equals true, which is the default for many different parameters in pytorch, pytorch will behind the scenes keep track of the gradients of our model and use them in lost up backward and optimizer step for back propagation and gradient descent
- However, we only need those two back propagation and gradient descent during training because that is when our model is learning

- So we don't need to do any learning when we're testing

▶ 📄 Click to view detailed content

```
out drop out layers. There we go. And batch norm. Do we have batch batch norm?
There we go. If you'd
like to work out what they are, feel free to check out the documentation. Just take
it from me for
now that model of our turns off different settings not needed for evaluation and
testing. Then we
set up with torch dot inference mode, inference mode. So what does this do? Let's
write down here.
So this turns off gradient tracking. So as we discussed, if we have parameters in
our model,
and it turns off actually a few more things and a couple more things behind the
scenes,
these are things again, not needed for testing. So we discussed that if parameters
in our model
have requires grad equals true, which is the default for many different parameters
in pytorch,
pytorch will behind the scenes keep track of the gradients of our model and use
them in
lost up backward and optimizer step for back propagation and gradient descent.
However,
we only need those two back propagation and gradient descent during training
because that
is when our model is learning. When we are testing, we are just evaluating the
parameters the patterns
that our model has learned on the training data set. So we don't need to do any
learning
when we're testing. So we turn off the things that we don't need. And is this going
to have
the correct spacing for me? I'm not sure we'll find out. So we still do the forward
pass
in testing mode, do the forward pass. And if you want to look up torch inference
mode,
just go torch inference mode. There's a great tweet about it that pytorch did,
which explains
what's happening. I think we've covered this before, but yeah, want to make your
inference
code and pytorch run faster. Here's a quick thread on doing exactly that. So
inference
mode is torch no grad. Again, you might see torch no grad. I think I'll write that
down just to
let you know. But here's what's happening behind the scenes. A lot of optimization
code,
which is beautiful. This is why we're using pytorch so that our code runs nice and
far.
Let me go there. You may also see with torch dot no grad in older pytorch code. It
does
similar things, but inference mode is the faster way of doing things according to
the thread.
And according to there's a blog post attached to there as well, I believe.
So you may also see torch dot no grad in older pytorch code, which would be valid.
But again,
```

inference mode is the better way of doing things. So do forward pass. So let's get our model. We
want to create test predictions here. So we're going to go model zero. There's a lot of code
going on here, but I'm going to just step by step it in a second. We'll go back through it all.
And then number two is calculate the loss. Now we're doing the test predictions here,
calculate the loss test predictions with model zero. So now we want to calculate the what we want
to calculate the test loss. So this will be our loss function, the difference between the test
pred and the test labels. That's important. So for testing, we're working with test data,
for training, we're working with training data. Model learns patterns on the training data,
and it evaluates those patterns that it's learned, the different parameters on the testing data. It
has never seen before, just like in a university course, you'd study the course materials, which
is the training data, and you'd evaluate your knowledge on materials you'd hopefully never
seen before, unless you sort of were friends with your professor, and they gave you the exam before
the actual exam that would be cheating right. So that's a very important point for the test data
set. Don't let your model see the test data set before you evaluate it. Otherwise, you'll get
poor results. And that's putting it out what's happening. Epoch, we're going to go Epoch,
and then I will introduce you to my little jingle to remember all of these steps because
there's a lot going on. Don't you worry. I know there's a lot going on, but again, with practice,
we're going to know what's happening here. Like it's the back of our hand. All right.
So do we need this? Oh, yeah, we could say that. Oh, no, we don't need test here. Loss. This is
loss, not test. Print out what's happening. Okay. And we don't actually need to do this
every epoch. We could just go say if epoch divided by 10 equals zero, print out what's happening.
Let's do that rather than clutter everything up, print it out, and we'll print out this.
So let's just step through what's happening. We've got 100 epochs. That's what we're about to run,
100 epochs. Our model is trained for about 10 so far. So it's got a good base. Maybe we'll just
get rid of that base. Start a new instance of our model. So we'll come right back down.
So our model is back to randomly initialized parameters, but of course, randomly initialized
flavored with a random seed of 42. Lovely, lovely. And so we've got our training code here. We've
discussed what's happening there. Now, we've got our testing code. We call model dot eval,
which turns off different settings in the model, not needed for evaluation slash

```
testing. We call
with torch inference mode context manager, which turns off gradient tracking and a
couple more
things behind the scenes to make our code faster. We do the forward pass. We do the
test predictions.
We pass our model, the test data, the test features to calculate the test
predictions.
```

Section 70: Main Topics

Key Topics:

- But the beauty of PyTorch is you can use basic Python printing statements to see what's happening with your model
- This is, I can't stress enough, like what we are doing here is going to be very similar throughout the entire rest of the course for training more and more models
- So this step that we've done here for training our model and evaluating it is seriously like the fundamental steps of deep learning with PyTorch is training and evaluating a model

▶ 📄 Click to view detailed content

```
Then we calculate the loss using our loss function. We can use the same loss
function that we used
for the training data. And it's called the test loss, because it's on the test data
set.
And then we print out what's happening, because we want to know what's happening
while our
model's training, we don't necessarily have to do this. But the beauty of PyTorch
is you can
use basic Python printing statements to see what's happening with your model. And
so,
because we're doing 100 epochs, we don't want to clutter up everything here. So
we'll just
print out what's happening every 10th epoch. Again, you can customize this as much
as you like
what's printing out here. This is just one example. If you had other metrics here,
such as calculating
model accuracy, we might see that later on, hint hint. We might print out our model
accuracy.
So this is very exciting. Are you ready to run 100 epochs? How low do you think our
loss can go?
This loss was after about 10. So let's just save this here. Let's give it a go.
Ready?
Three, two, one. Let's run. Oh my goodness. Look at that. Waits. Here we go. Every
10 epochs
were printing out what's happening. So the zero epoch, we started with losses 312.
```

Look at it go
down. Yes, that's what we want. And our weights and bias, are they moving towards our ideal weight
and bias values of 0.7 and 0.3? Yes, they're moving in the right direction here. The loss is
going down. Epoch 20, wonderful. Epoch 30, even better. 40, 50, going down, down, down. Yes,
this is what we want. This is what we want. Now, we're predicting a straight line here. Look how
low the loss gets. After 100 epochs, we've got about three times less than what we had before.
And then we've got these values are quite close to where they should be, 0.5629, 0.3573. We'll make
some predictions. What do they look like? Why preds new? This is the original predictions
with random values. And if we make why preds new, look how close it is after 100 epochs.
Now, what's our, do we print out the test loss? Oh no, we're printing out loss as well.
Let's get rid of that. I think this is this. Yeah, that's this statement here. Our code would have
been a much cleaner if we didn't have that, but that's all right. Life goes on. So our test loss,
because this is the test predictions that we're making, is not as low as our training loss.
I wonder how we could get that lower. What do you think we could do? We just trained it for
longer. And what happened? How do you think you could get these red dots to line up with these
green dots? Do you think you could? So that's my challenge to you for the next video.
Think of something that you could do to get these red dots to match up with these green dots,
maybe train for longer. How do you think you could do that? So give that a shot. And I'll see
in the next video, we'll review what our testing code is doing. I'll see you there.
Welcome back. In the last video, we did something super exciting. We trained our model for 100 epochs
and look how good the predictions got. But I finished it off with challenging you to see if you could
align the red dots with the green dots. And it's okay if you're not sure how the best way to do
that. That's what we're here for. We're here to learn what are the best way to do these things
together. But you might have had the idea of potentially training the model for a little bit
longer. So how could we do that? Well, we could just rerun this code. So the model is going to
remember the parameters that it has from what we've done here. And if we rerun it, well, it's going
to start from where it finished off, which is already pretty good for our data set. And then
it's going to try and improve them even more. This is, I can't stress enough, like what we are
doing here is going to be very similar throughout the entire rest of the course for training more
and more models. So this step that we've done here for training our model and

```
evaluating it
is seriously like the fundamental steps of deep learning with PyTorch is training
and evaluating
a model. And we've just done it. Although I'll be it to predict some red dots and
green dots.
That's all right. So let's try to line them up, hey, red dots onto green dots. I
reckon if we
train it for another 100 epochs, we should get pretty darn close. Ready? Three,
two, one. I'm
going to run this cell again. Runs really quick because our data's nice and simple.
But
look at this, lastly, we started 0244. Where do we get down to? 008. Oh my
goodness. So we've
improved it by another three X or so. And now this is where our model has got
really good.
On the test loss, we've gone from 00564. We've gone down to 005. So almost 10X
improvement there.
And so we make some more predictions. What are our model parameters? Remember the
ideal ones here.
We won't necessarily know them in practice, but because we're working with a simple
data set,
we know what the ideal parameters are. Model zero state dig weights. These are what
they
previously were. What are they going to change to? Oh, would you look at that? Oh,
06990. Now, again, if yours are very slightly different to mine, don't worry too
much. That is
the inherent randomness of machine learning and deep learning. Even though we set a
manual seed,
```

Section 71: Main Topics

Key Topics:

- Of course, we could just create a model and set the parameters ourselves manually
- We just wrote some machine learning code to do it for us with the power of back propagation and gradient descent
- We might use a different learning rate

▶ 📄 Click to view detailed content

```
it may be slightly different. The direction is more important. So if your number
here is not
exactly what mine is, it should still be quite close to 0.7. And the same thing
with this one.
If it's not exactly what mine is, don't worry too much. The same with all of these
loss values
as well. The direction is more important. So we're pretty darn close. How do these
```

predictions
look? Remember, these are the original ones. We started with random. And now we've trained a model.
So close. So close to being exactly that. So a little bit off. But that's all right. We could
tweak a few things to improve this. But I think that's well and truly enough for this example
purpose. You see what's happened. Of course, we could just create a model and set the parameters
ourselves manually. But where would be the fun in that? We just wrote some machine learning code
to do it for us with the power of back propagation and gradient descent. Now in the last video,
we wrote the testing loop. We discussed a few other steps here. But now let's go over it with
a colorful slide. Hey, because I mean, code on a page is nice, but colors are even nicer. Oh,
we haven't done this. We might set up this in this video too. But let's just discuss what's going on.
Create an empty list for storing useful value. So this is helpful for tracking model progress.
How can we just do this right now? Hey, we'll go here and we'll go.
So what did we have? Epoch count equals that. And then we'll go
lost values. So why do we keep track of these? It's because
if we want to monitor our models progress, this is called tracking experiments. So track
different values. If we wanted to try and improve upon our current model with a future model. So
our current results, such as this, if we wanted to try and improve upon it, we might build an
entire other model. And we might train it in a different setup. We might use a different learning
rate. We might use a whole bunch of different settings, but we track the values so that we
can compare future experiments to past experiments, like the brilliant scientists that we are.
And so where could we use these lists? Well, we're calculating the loss here. And we're calculating
the test loss here. So maybe we each time append what's going on here as we do a status update.
So epoch count dot append, and we're going to go a current epoch. And then we'll go loss values
dot append, a current loss value. And then we'll do test loss values dot append, the current test
loss values. Wonderful. And now let's re-instantiate our model so that it starts from fresh. So this
is just create another instance. So we're just going to re-initialize our model parameters to
start from zero. If we wanted to, we could functionize all of this so we don't have to
go right back up to the top of the code. But just for demo purposes, we're doing it how we're doing
it. And I'm going to run this for let's say 200 epochs, because that's what we ended up doing,
right? We ran it for 200 epochs, because we did 100 epochs twice. And I want to show you something
beautiful, one of the most beautiful sites in machine learning. So there we go, we

```
run it for
200 epochs, we start with a fairly high training loss value and a fairly high test
loss value. So
remember, what is our loss value? It's ma e. So if we go back, yeah, this is what
we're measuring
for loss. So this means for the test loss on average, each of our dot points here,
the red
predictions are 0.481. That's the average distance between each dot point. And then
ideally, what
are we doing? We're trying to minimize this distance. That's the ma e. So the mean
absolute error.
And we get it right down to 0.05. And if we make predictions, what do we have here,
we get very
close to the ideal weight and bias, make our predictions, have a look at the new
predictions.
Yeah, very small distance here. Beautiful. That's a low loss value.
Ideally, they'd line up, but we've got as close as we can for now. So this is one
of the most
beautiful sites in machine learning. So plot the loss curves. So let's make a plot,
because what
we're doing, we were tracking the value of epoch count, loss values and test loss
values.
Let's have a look at what these all look like. So epoch count goes up, loss values
ideally go down.
So we'll get rid of that. We're going to create a plot p l t dot plot. We're going
to step back
through the test loop in a second with some colorful slides, label equals train
loss.
And then we're going to go plot. You might be able to tell what's going on here.
Test loss
values. We're going to visualize it, because that's the data explorer's motto,
right, is visualize,
visualize, visualize. This is equals. See, collab does this auto correct. That
doesn't really work
very well. And I don't know when it does it and why it doesn't. And we got, I know,
we didn't,
we didn't say loss value. So that's a good auto correct. Thank you, collab.
So training and loss and test loss curves. So this is another term you're going to
come across
often is a loss curve. Now you might be able to think about a loss curve. If we're
doing a loss
curve, and it's starting at the start of training, what do we want that curve to
do?
```

Section 72: Main Topics

Key Topics:

- So what we're doing here is our loss values are still on PyTorch, and they can't be because mapplotlib works with NumPy
- One of the most beautiful sides in machine learning is a declining loss curve

- And of course, lower is better

▶ 📄 Click to view detailed content

```
What do we want our loss value to do? We want it to go down. So what should an
ideal loss
curve look like? Well, we're about to see a couple. Let's have a look. Oh, what do
we got wrong?
Well, we need to, I'll turn it into NumPy. Is this what we're getting wrong? So why
is this wrong?
Loss values. Why are we getting an issue? Test loss values.
Ah, it's because they're all tens of values. So I think we should, let's,
I might change this to NumPy. Oh, can I just do that? If I just call this as a
NumPy array,
we're going to try and fix this on the fly. People, NumPy array, we'll just turn
this into a NumPy
array. Let's see if we get NumPy. I'm figuring these things out together. NumPy as
NumPy,
because mapplotlib works with NumPy. Yeah, there we go. So can we do loss values?
Maybe
I'm going to try one thing, torch dot tensor, loss values, and then call
CPU dot NumPy. See what happens here.
There we go. Okay, so let's just copy this. So what we're doing here is
our loss values are still on PyTorch, and they can't be because mapplotlib works
with
NumPy. And so what we're doing here is we're converting our loss values of the
training loss
to NumPy. And if you call from the fundamental section, we call CPU and NumPy, I
wonder if we
can just do straight up NumPy, because we're not working on there. Yeah, okay, we
don't need
CPU because we're not working on the GPU yet, but we might need that later on.
Well, this work.
Beautiful. There we go. One of the most beautiful sides in machine learning is a
declining loss
curve. So this is how we keep track of our experiments, or one way, quite
rudimentary. We'd like to
automate this later on. But I'm just showing you one way to keep track of what's
happening.
So the training loss curve is going down here. The training loss starts at 0.3, and
then it goes
right down. The beautiful thing is they match up. If there was a two bigger
distance behind the
train loss and the test loss, or sorry, between, then we're running into some
problems. But if they
match up closely at some point, that means our model is converging and the loss is
getting as
close to zero as it possibly can. If we trained for longer, maybe the loss will go
almost basically
to zero. But that's an experiment I'll leave you to try to train that model for
longer.
Let's just step back through our testing loop to finish off this video. So we did
that. We created
empty lists for strong useful values, storing useful values, strong useful values.
Told the
```

model what we want to evaluate or that we want to evaluate. So we put it in an evaluation mode.
It turns off functionality used for training, but not evaluations, such as drop out and batch
normalization layers. If you want to learn more about them, you can look them up in the documentation.
Turn on torch inference mode. So this is for faster performance. So we don't necessarily need this,
but it's good practice. So I'm going to say that yes, turn on torch inference mode. So this
disables functionality such as gradient tracking for inference. Gradient tracking is not needed
for inference only for training. Now we pass the test data through the model. So this will call
the models implemented forward method. The forward pass is the exact same as what we did in the
training loop, except we're doing it on the test data. So big notion there, training loop,
training data, testing loop, testing data. Then we calculate the test loss value,
how wrong the models predictions are on the test data set. And of course, lower is better.
And finally, we print out what's happening. So we can keep track of what's going on during
training. We don't necessarily have to do this. You can customize this print value to print out
almost whatever you want, because it's pie torches, basically very beautifully interactive with pure
Python. And then we keep track of the values of what's going on on epochs and train loss and test
loss. We could keep track of other values here. But for now, we're just going, okay, what's the loss
value at a particular epoch for the training set? And for the test set. And of course, all of this
could be put into a function. And that way we won't have to remember these steps off by heart.
But the reason why we've spent so much time on this is because we're going to be using this
training and test functionality for all of the models that we build throughout this course.
So give yourself a pat in the back for getting through all of these videos. We've written a lot
of code. We've discussed a lot of steps. But if you'd like a song to remember what's happening,
let's finish this video off with my unofficial PyTorch optimization loop song.
So for an epoch in a range, go model dot train, do the forward pass, calculate the loss, optimize
a zero grad, loss backward, optimize a step, step, step. No, you only have to call this once.
But now let's test, go model dot eval with torch inference mode, do the forward pass,
calculate the loss. And then the real song goes for another epoch because you keep going back
through. But we finish off with print out what's happening. And then of course, we evaluate what's
going on. With that being said, it's time to move on to another thing. But if you'd like to review

```
what's happening, please, please, please try to run this code for yourself again
and check out the
```

Section 73: Main Topics

Key Topics:

- Oh, by the way, if you want to link to all of the extra curriculum, just go to the book version of the course
- So what we might cover in this video is saving a model in PyTorch
- So now let's see how we can save our models in PyTorch

▶ 📄 Click to view detailed content

```
slides and also check out the extra curriculum. Oh, by the way, if you want to link
to all
of the extra curriculum, just go to the book version of the course. And it's all
going to be in here.
So that's there ready to go. Everything I link is extra curriculum will be in the
extra curriculum
of each chapter. I'll see you in the next video. Welcome back. In the last video,
we saw how to
train our model and evaluate it by not only looking at the loss metrics and the
loss curves,
but we also plotted our predictions and we compared them. Hey, have a go at these
random
predictions. Quite terrible. But then we trained a model using the power of back
propagation and
gradient descent. And now look at our predictions. They're almost exactly where we
want them to be.
And so you might be thinking, well, we've trained this model and it took us a while
to
write all this code to get some good predictions. How might we run that model
again? So I've took
in a little break after the last video, but now I've come back and you might notice
that my Google
Colab notebook has disconnected. So what does this mean if I was to run this? Is it
going to work?
I'm going to connect to a new Google Colab instance. But will we have all of the
code that we've run
above? You might have already experienced this if you took a break before and came
back to the
videos. Ah, so plot predictions is no longer defined. And do you know what that
means? That
means that our model is also no longer defined. So we would have lost our model. We
would have
lost all of that effort of training. Now, luckily, we didn't train the model for
too long. So we can
```

just go run time, run all. And it's going to rerun all of the previous cells and be quite quick.
Because we're working with a small data set and using a small model. But we've been through all
of this code. Oh, what have we got wrong here? Model zero state dict. Well, that's all right.
This is good. We're finding errors. So if you want to as well, you can just go run after. It's going
to run all of the cells after. Beautiful. And we come back down. There's our model training.
We're getting very similar values to what we got before. There's the lost curves. Beautiful.
Still going. Okay. Now our predictions are back because we've rerun all the cells and we've got
our model here. So what we might cover in this video is saving a model in PyTorch. Because if
we're training a model and you get to a certain point, especially when you have a larger model,
you probably want to save it and then reuse it in this particular notebook itself. Or you might
want to save it somewhere and send it to your friend so that your friend can try it out. Or you
might want to use it in a week's time. And if Google Colab is disconnected, you might want to
be able to load it back in somehow. So now let's see how we can save our models in PyTorch. So
I'm going to write down here. There are three main methods you should know about
for saving and loading models in PyTorch because of course with saving comes loading. So we're
going to over the next two videos discuss saving and loading. So one is torch.save. And as you might
guess, this allows you to save a PyTorch object in Python's pickle format. So you may or may not
be aware of Python pickle. There we go. Python object serialization. There we go. So we've got
the pickle module implements a binary protocols or implements binary protocols for serializing
and deserializing a Python object. So serializing means I understand it is saving and deserializing
means that it's loading. So this is what PyTorch uses behind the scenes, which is from pure Python.
So if we go back here in Python's pickle format, number two is torch.load, which you might be able
to guess what that does as well, allows you to load a saved PyTorch object. And number three is
also very important is torch.nn.module.loadStatedict. Now what does this allow you to do? Well,
this allows you to load a model's saved dictionary or save state dictionary. Yeah, that's what we'll
call it. Save state dictionary. Beautiful. And what's the model state dict? Well, let's have a look,
model zero dot state dict. The beauty of PyTorch is that it stores a lot of your model's important
parameters in just a simple Python dictionary. Now it might not be that simple because our model,
again, only has two parameters. In the future, you may be working with models with millions of

```
parameters. So looking directly at the state deck may not be as simple as what
we've got here.
But the principle is still the same. It's still a dictionary that holds the state
of your model.
And so I've got these three methods I want to show you where from because this is
going to be
your extra curriculum, save and load models, your extra curriculum for this video.
If we go into here, this is a very, very, very important piece of PyTorch
documentation,
or maybe even a tutorial. So your extra curriculum for this video is to go through
it.
Here we go. We've got torch, save, torch, load, torch, module, state deck. That's
where, or load
state deck, that's where I've got the three things that we've just written down.
And there's a fair
few different pieces of information. So what is a state deck? So in PyTorch, the
learnable
parameters, i.e. the weights and biases of a torch and end module, which is our
model.
```

Section 74: Main Topics

Key Topics:

- So the recommended way of saving and loading a PyTorch model is by saving its state deck
- So PyTorch save and load code
- So if we go saving our PyTorch model

▶ 📄 Click to view detailed content

```
Remember, our model subclasses and end module are contained in the model's
parameters. Access
with model.parameters, a state deck is simply a Python dictionary object that maps
each layer
to its parameter tensor. That's what we've seen. And so then if we define a model,
we can initialize the model. And if we wanted to print the state decked, we can use
that.
The optimizer also has a state deck. So that's something to be aware of. You can go
optimizer.state
deck. And then you get an output here. And this is our saving and loading model for
inference. So
inference, again, is making a prediction. That's probably what we want to do in the
future at some
point. For now, we've made predictions right within our notebook. But if we wanted
to use our model
outside of our notebook, say in an application, or in another notebook that's not
this one,
```

you'll want to know how to save and load it. So the recommended way of saving and loading a
PyTorch model is by saving its state deck. Now, there is another method down here,
which is saving and loading the entire model. So your extracurricular for this lesson,
we're going to go through the code to do this. But your extracurricular is to read all of the
sections in here, and then figure out what the pros and cons are of saving and loading the entire
model versus saving and loading just the state deck. So that's a challenge for you for this video.
I'm going to link this in here. And now let's write some code to save our model.
So PyTorch save and load code. Code tutorial plus extracurricular. So if we go
saving our PyTorch model. So what might we want? What do you think the save parameter takes?
If we have torch.save, what do you think it takes inside it? Well, let's find out together.
Hey, so let's import part lib. We're going to see why in a second. This is Python's
module for dealing with writing file paths. So if we wanted to save something to this is Google
Colab's file section over here. But just remember, if we do save this from within Google Colab,
the model will disappear if our Google Colab notebook instance disconnects. So I'll show you
how to download it from Google Colab if you want. Google Colab also has a way save from Google Colab
Google Colab to Google Drive to save it to your Google Drive if you wanted to. But I'll leave you
to look at that on your own if you like. So we're first going to create a model directory.
So create models directory. So this is going to help us create a folder over here called models.
And of course, we could create this by hand by adding a new folder here somewhere. But I like
to do it with code. So model path, we're going to set this to path, which is using the path library
here to create us a path called models. Simple. We're just going to save all of our models to
models to the models file. And then we're going to create model path, we're going to make that
directory model path dot mkdir for make directory. We're going to set parents to equals true.
And we're also going to set exist okay equals to true. That means if it already existed,
it won't throw us an error. It will try to create it. But if it already exists, it'll just recreate
the parents directory or it'll leave it there. It won't error out on us. We're also going to
create a model save path. This way, we can give our model a name. Right now, it's just model zero.
We want to save it under some name to the models directory. So let's create the model name.
Model name equals 01. I'm going to call it 01 for the section. That way, if we have more models
later on the course, we know which ones come from where you might create your own naming
convention, model workflow, pytorch workflow, model zero dot pth. And now this is

```
another
important point. Pytorch objects usually have the extension dot pth for pytorch or
dot pth.
So if we go in here, and if we look up dot pth, yeah, a common convention is to
save models
using either a dot pth or dot pth file extension. I'll let you choose which one you
like. I like
dot pth. So if we go down here dot pth, they both result in the same thing. You
just have to remember
to make sure you write the right loading path and right saving path. So now we're
going to create
our model save path, which is going to be our model path. And because we're using
the path lib,
we can use this syntax that we've got here, model path slash model name. And then
if we just print out
model save path, what does this look like? There we go. So it creates a supposic
path
using the path lib library of models slash 01 pytorch workflow model zero dot pth.
We haven't
saved our model there yet. It's just got the path that we want to save our model
ready. So if we
refresh this, we've got models over here. Do we have anything in there? No, we
don't yet. So now
is our step to save the model. So three is save the model state dict. Why are we
saving the state
dict? Because that's the recommended way of doing things. If we come up here,
saving and loading the
model for inference, save and load the state dict, which is recommended. We could
also save the entire
model. But that's part of your extra curriculum to look into that. So let's use
some syntax. It's
```

Section 75: Main Topics

Key Topics:

- Now, of course, we've saved a model
- So we'll come back here and we'll go loading a pytorch model

▶ 📄 Click to view detailed content

```
quite like this torch dot save. And then we pass it an object. And we pass it a
path of where to
save it. We already have a path. And good thing is we already have a model. So we
just have to call
this. Let's try it out. So let's go print f saving model to and we'll put in the
path here.
Model save path. I like to print out some things here and there that way. We know
what's going on.
```

And I don't need that capital. Why do I? Getting a little bit trigger happy here with the typing.
So torch dot save. And we're going to pass in the object parameter here. And if we looked up torch
save, we can go. What does this code take? So torch save object f. What is f? A file like object.
Okay. Or a string or OS path like object. Beautiful. That's what we've got. A path like
object containing a file name. So let's jump back into here. The object is what? It's our model zero
dot state dict. That's what we're saving. And then the file path is model save path. You ready?
Let's run this and see what happens. Beautiful. Saving model to models. So it's our model path.
And there's our model there. So if we refresh this, what do we have over here?
Wonderful. We've saved our trained model. So that means we could potentially if we wanted to,
you could download this file here. That's going to download it from Google CoLab to your local
machine. That's one way to do it. But there's also a guide here to save from Google Collaboratory
to Google Drive. That way you could use it later on. So there's many different ways.
The beauty of pie torches is flexibility. So now we've got a saved model. But let's just check
using our LS command. We're going to check models. Yeah, let's just check models. This is going to
check here. So this is list. Wonderful. There's our 01 pie torch workflow model zero dot pth. Now,
of course, we've saved a model. How about we try loading it back in and seeing how it works. So if
you want to challenge, read ahead on the documentation and try to use torch dot load to bring our model
back in. See what happens. I'll see in the next video. Welcome back. In the last video, we wrote
some code here to save our pie torch model. I'm just going to exit out of this couple of things
that we don't need just to clear up the screen. And now we've got our dot pth file, because remember
dot pth or dot pth is a common convention for saving a pie torch model. We've got it saved there,
and we didn't necessarily have to write all of this path style code. But this is just handy for
later on if we wanted to functionize this and create it in say a save dot pie file over here,
so that we could just call our save function and pass it in a file path where we wanted to save
like a directory and a name, and then it'll save it exactly how we want it for later on.
But now we've got a saved model. I issued a challenge of trying to load that model in.
So do we have torch dot load in here? Did you try that out? We've got, oh, we've got a few options
here. Wonderful. But we're using one of the first ones. So let's go back up here. If we wanted to
check the documentation for torch dot load, we've got this option here, load. What happens? Loads

```
and objects saved with torch dot save from a file. Torch dot load uses Python's
unpickling
facilities, but treat storages which underlie tenses specially. They are firstly
serialized
on the CPU, and then I moved the device they were saved from. Wonderful. So this is
moved to the
device. If later on when we're using a GPU, this is just something to keep in mind.
We'll see that
when we start to use a CPU and a GPU. But for now, let's practice using the torch
dot load method
and see how we can do it. So we'll come back here and we'll go loading a pytorch
model.
And since we, she's going to start writing here, since we saved our models state
debt,
so just the dictionary of parameters from a model, rather than
the entire model, we'll create a new instance of our model class and load the state
deck,
load the saved state deck. That's better state deck into that.
Now, this is just words on a page. Let's see this in action. So to load in a state
deck,
which is what we say, we didn't save the entire model itself, which is one option.
That's extra curriculum, but we saved just the model state deck. So if we remind
ourselves what
model zero dot state deck looks like, we saved just this. So to load this in, we
have to
instantiate a new class or a new instance of our linear regression model class. So
to load in a
saved state deck, we have to instantiate a new instance of our model class. So
let's call this
loaded model zero. I like that. That way we can differentiate because it's still
going to be the
same parameters as model zero, but this way we know that this instance is the
loaded version,
not just the version we've been training before. So we'll create a new version of
it here,
linear regression model. This is just the code that we wrote above, linear
regression model.
And then we're going to load the saved state deck of model zero. And so this will
update the new
```

Section 76: Main Topics

Key Topics:

- So that's saving and loading a model in pytorch
- Over the past few videos, we've covered a whole bunch of ground in a pytorch workflow, starting with data, then building a model
- We then learned how to save and load a model in pytorch

▶ 📄 Click to view detailed content

instance with updated parameters. So let's just check before we load it, we haven't written any
code to actually load anything. What does loaded model zero? What does the state deck look like here?
It won't have anything. It'll be initialized with what?
Oh, loaded. That's what I called it loaded. See how it's initialized with random parameters.
So essentially all we're doing when we load a state dictionary into our new instance of our
model is that we're going, hey, take the saved state deck from this model and plug it into this.
So let's see what happens when we do that. So loaded model zero. Remember how I said there's
a method to also be aware of up here, which is torch nn module dot load state deck. And because
our model is a what, it's a subclass of torch dot nn dot module. So we can call load state deck
on our model directly or on our instance. So recall linear regression model is a subclass
of nn dot module. So let's call in load state deck. And this is where we call the torch dot load
method. And then we pass it the model save path. Is that what we call it? Because torch dot load,
it takes in F. So what's F a file like object or a string or a OS path like object. So that's
why we created this path like object up here. Model save path. So all we're doing here,
we're creating a new instance, linear regression model, which is a subclass of nn dot module.
And then on that instance, we're calling in load state deck of torch dot load model save path.
Because what's saved at the model save path, our previous models state deck, which is here.
So if we run this, let's see what happens. All keys match successfully. That is beautiful.
And so see the values here, loaded state deck of model zero. Well, let's check the loaded version
of that. We now have wonderful, we have the exact same values as above. But there's a little
way that we can test this. So how about we go make some predictions. So make some predictions.
Just to make sure with our loaded model. So let's put it in a valve mode. Because when you make
predictions, you want it in evaluation mode. So it goes a little bit faster. And we want to
also use inference mode. So with torch dot inference mode for making predictions. We want to write
this loaded model preds, we're going to make some predictions on the test data as well. So loaded
model zero, we're going to forward pass on the X test data. And then we can have a look at the
loaded model preds. Wonderful. And then to see if the two models are the same, we can compare
loaded model preds with original model preds. So why preds? These should be equivalent equals
equals loaded model preds. Do we have the same thing? False, false, false, what's

going on here?
Why preds? How much different are they? Oh, where's that happened? Have we made some
model preds with this yet? So how about we make some model preds? This is troubleshooting on
the fly team. So let's go model zero dot eval. And then with torch dot inference mode,
this is how we can check to see that our two models are actually equivalent. Why preds equals,
I have a feeling why preds actually save somewhere else equals model zero. And then we pass it the
X test data. And then we might move this above here. And then have a look at what why preds equals.
Do we get the same output? Yes, we should. Wonderful. Okay, beautiful. So now we've covered
saving and loading models or specifically saving the models state deck. So we saved it here with
this code. And then we loaded it back in with load state deck plus torch load. And then we
checked to see by testing equivalents of the predictions of each of our models. So the original
one that we trained here, model zero, and the loaded version of it here. So that's saving and
loading a model in pytorch. There are a few more things that we could cover. But I'm going to leave
that for extra curriculum. We've covered the two main things or three main things. One, two, three.
If you'd like to read more, I'd highly encourage you to go through and read this tutorial here.
But with that being said, we've covered a fair bit of ground over the last few videos. How about
we do a few videos where we put everything together just to reiterate what we've done.
I think that'll be good practice. I'll see you in the next video.
Welcome back. Over the past few videos, we've covered a whole bunch of ground in a pytorch
workflow, starting with data, then building a model. Well, we split the data, then we built a
model. We looked at the model building essentials. We checked the contents of our model. We made
some predictions with a very poor model because it's based off random numbers. We spent a whole
bunch of time figuring out how we could train a model. We figured out what the loss function is.
We saw an optimizer. We wrote a training and test loop. We then learned how to save and load a
model in pytorch. So now I'd like to spend the next few videos putting all this together. We're
not going to spend as much time on each step, but we're just going to have some practice together
so that we can reiterate all the things that we've done. So putting it all together,
let's go back through the steps above and see it all in one place. Wonderful.

Section 77: Main Topics

Key Topics:

- We're going to cover those throughout the course
- So let's start by importing pytorch
- And we're going to check out pytorch version

▶ 📄 Click to view detailed content

```
So we're going to start off with 6.1 and we'll go have a look at our workflow. So
6.1 is data,
but we're going to do one step before that. And I'm just going to get rid of this
so we have a bit
more space. So we've got our data ready. We've turned it into tenses way back at
the start.
Then we built a model and then we picked a loss function and an optimizer. We built
a training
loop. We trained our model. We made some predictions. We saw that they were better.
We evaluated our
model. We didn't use torch metrics, but we got visual. We saw our red dots starting
to line up
with the green dots. We haven't really improved through experimentation. We did a
little bit of
it though, as in we saw that if we trained our model for more epochs, we got better
results.
So you could argue that we have done a little bit of this, but there are other ways
to experiment.
We're going to cover those throughout the course. And then we saw how to save and
reload a trained
model. So we've been through this entire workflow, which is quite exciting,
actually.
So now let's go back through it, but we're going to do it a bit quicker than what
we've done before,
because I believe you've got the skills to do so now. So let's start by importing
pytorch.
So you could start the code from here if you wanted to. And that plot live. And
actually,
if you want, you can pause this video and try to recode all of the steps that we've
done
by putting some headers here, like data, and then build a model and then train the
model,
save and load a model, whatever, and try to code it out yourself. If not, feel free
to follow along
with me and we'll do it together. So import torch from torch import. Oh, would help
if I could spell
torch import and n because we've seen that we use an n quite a bit. And we're going
to also
import map plot live because we like to make some plots because we like to get
visual.
Visualize visualize visualize as PLT. And we're going to check out pytorch version.
That way we know if you're on an older version, some of the code might not work
```

here. But if you're
on a newer version, it should work. If it doesn't, let me know. There we go. 1.10. I'm using 1.10
for this. By the time you watch this video, there may be a later version out. And we're also going
to let's create some device agnostic code. So create device agnostic code, because I think we're
up to this step now. This means if we've got access to a GPU, our code will use it for potentially
faster computing. If no GPU is available, the code will default to using CPU. We don't necessarily
need to use a GPU for our particular problem that we're working on right now because it's a small
model and it's a small data set, but it's good practice to write device agnostic code. So that
means our code will use a GPU if it's available, or a CPU by default, if a GPU is not available.
So set up device agnostic code. We're going to be using a similar setup to this throughout the
entire course from now on. So that's why we're bringing it back. CUDA is available. So remember
CUDA is NVIDIA's programming framework for their GPUs, else use CPU. And we're going to print
what device are we using? Device. So what we might do is if we ran this, it should be just a CPU
for now, right? Yours might be different to this if you've enabled a GPU, but let's change this
over to use CUDA. And we can do that if you're using Google Colab, we can change the runtime type
by selecting GPU here. And then I'm going to save this, but what's going to happen is it's
going to restart the runtime. So we're going to lose all of the code that we've written above.
How can we get it all back? Well, we can go. Run all. This is going to run all of the cells
above here. They should all work and it should be quite quick because our model and data aren't
too big. And if it all worked, we should have CUDA as our device that we can use here. Wonderful.
So the beauty of Google Colab is that they've given us access to on a video GPU. So thank you,
Google Colab. Just once again, I'm paying for the paid version of Google Colab. You don't have to.
The free version should give you access to a GPU, or be it it might not be as a later version as
GPU as the pro versions give access to. But this will be more than enough for what we're about to
recreate. So I feel like that's enough for this video. We've got some device agnostic code ready
to go. And for the next few videos, we're going to be rebuilding this except using device agnostic
code. So give it a shot yourself. There's nothing in here that we haven't covered before. So I'll
see you in the next video. Let's create some data. Welcome back. In the last video, we set up some
device agnostic code and we got ready to start putting everything we've learned together.

```
So now let's continue with that. We're going to recreate some data. Now we could
just copy this
code, but we're going to write it out together so we can have some practice
creating a dummy data
set. And we want to get to about this stage in this video. So we want to have some
data that we can
```

Section 78: Main Topics

Key Topics:

- There's a lot of things in machine learning that are quite flexible
- It's tenses because we use PyTorch to create it
- So building a PyTorch linear model

▶ 📄 Click to view detailed content

```
plot so that we can build a model to once again, learn on the blue dots to predict
the green dots.
So we'll come down here data. I'm going to get out of this as well so that we have
a bit more room.
Let's now create some data using the linear regression formula of y equals weight
times
features plus bias. And you may have heard this as y equals mx plus c or mx plus b
or something like
that, or you can substitute these for different names. Images when I learned this
in high school,
it was y equals mx plus c. Yours might be slightly different. Yeah, bx plus a.
That's what they use
here. A whole bunch of different ways to name things, but they're all describing
the same thing.
So let's see this in code rather than formulaic examples. So we're going to create
our weight,
which is 0.7 and a bias, which is 0.3. These are the values we previously used for
a challenge you
could change these to 0.1 maybe and 0.2. These could be whatever values you'd like
to set them as.
So weight and bias, the principle is going to be the same thing. We're going to try
and build a
model to estimate these values. So we're going to start at 0 and we're going to end
at 1.
So we can just create a straight line and we're going to fill in those between 0
and 1 with a
step of 0.02. And now we'll create the x and y features x and y, which is features
and labels
actually. So x is our features and y are our labels. x equals torch dot a range and
x is a
capital Y is that because typically x is a feature matrix. Even though ours is just
a vector now,
```

we're going to unsqueeze this so we don't run into dimensionality issues later on.
You can check this for yourself without unsqueeze, errors will pop up and y equals weight times
x plus bias. You see how we're going a little bit faster now? This is sort of the pace that we're
going to start going for things that we've already covered. If we haven't covered something, we'll
slow down, but if we have covered something, I'm going to step it through. We're going to start
speeding things up a little. So if we get some values here, wonderful. We've got some x values
and they correlate to some y values. We're going to try and use the training values of x to predict
the training values of y and subsequently for the test values. Oh, and speaking of training and test
values, how about we split the data? So let's split the data. Split data. So we'll create the
train split equals int 0.8. We're going to use 80%, which is where 0.8 comes from,
for the length of x. So we use 80% of our samples for the training, which is a typical
training and test split, 80, 20. They're abouts. You could use like 70, 30. You could use 90, 10.
It all depends on how much data you have. There's a lot of things in machine learning that are
quite flexible. Train split, we're going to index on our data here so that we can create our splits.
Google Colab auto corrected my code in a non-helpful way just then. And we're going to do the
opposite split for the testing data. Now let's have a look at the lengths of these. If my calculations
are correct, we should have about 40 training samples and 10 testing samples. And again, this
may change in the future. When you work with larger data sets, you might have 100,000 training
samples and 20,000 testing samples. The ratio will often be quite similar. And then let's plot
what's going on here. So plot the data and note, if you don't have the plot predictions
function loaded, this will error. So we can just run plot predictions here if we wanted to. And
we'll pass it in X train, Y train, X test, Y test. And this should come up with our
plot. Wonderful. So we've just recreated the data that we've been previously using. We've got
blue dots to predict green dots. But if this function errors out because you've started the notebook
from here, right from this cell, and you've gone down from there, just remember, you'll just have
to go up here and copy this function. We don't have to do it because we've run all the cells,
but if you haven't run that cell previously, you could put it here and then run it, run it,
and we'll get the same outcome here. Wonderful. So what's next? Well, if we go back to our workflow,
we've just created some data. And have we turned it into tenses yet? I think it's just still, oh,
yeah, it is. It's tenses because we use PyTorch to create it. But now we're up to building or

```
picking a model. So we've built a model previously. We did that back in build
model. So you could
refer to that code and try to build a model to fit the data that's going on here.
So that's
your challenge for the next video. So building a PyTorch linear model. And why do
we call it linear?
Because linear refers to a straight line. What's nonlinear? Non-straight. So I'll
see you in the
next video. Give it a shot before we get there. But we're going to build a PyTorch
linear model.
Welcome back. We're going through some steps to recreate everything that we've
done. In the last
video, we created some dummy data. And we've got a straight line here. So now by
the workflow,
we're up to building a model or picking a model. In our case, we're going to build
one
```

Section 79: Main Topics

Key Topics:

- And I've put building a PyTorch linear model here
- module because why a lot of PyTorch models, subclass, and then module
- Remember, nn in PyTorch stands for neural network

▶ 📄 Click to view detailed content

```
to suit our problem. So we've got some linear data. And I've put building a PyTorch
linear model
here. I issued you the challenge of giving it a go. You could do exactly the same
steps that
we've done in build model. But I'm going to be a little bit cheeky and introduce
something
new here. And that is the power of torch.nn. So let's see it. What we're going to
do is we're
going to create a linear model by subclassingnn.module because why a lot of PyTorch
models,
subclass, and then module. So class linear regression, what should we call this
one?
Linear regression model v2. How about that? And we'll subclassnn.module. So much
similar code to
what we've been writing so far. Or when we first created our linear regression
model.
And then we're going to put the standard constructor code here, def init underscore
underscore.
And it's going to take as an argument self. And then we're going to call super dot
another
underscore init underscore underscore brackets. But we're going to instead of if
you recall above
```

back in the build model section, we initialized these parameters ourselves. And I've been hinting
at in the past in videos we've seen before that oftentimes you won't necessarily initialize the
parameters yourself. You'll instead initialize layers that have the parameters in built in those
layers. We still have to create a forward method. But what we're going to see is how we can use our
torch linear layer to do these steps for us. So let's write the code and then we'll step through it.
So we'll go usenn.linear because why we're building linear regression model and our data is linear.
And in the past, our previous model has implemented linear regression formula. So for creating the
model parameters. So we can go self dot linear layer equals. So this is constructing a variable
that this class can use self linear layer equals nn dot linear. Remember, nn in PyTorch stands for
neural network. And we have in features as one of the parameters and out features as another
parameter. This means we want to take as input of size one and output of size one. Where does that
come from? Well, if we have a look at x train and y train, we have one value of x. Maybe there's
too many here. x five will be the first five five and five. So recall, we have one value of x
equates to one value of y. So that means within this linear layer, we want to take as one feature
x to output one feature y. And we're using just one layer here. So the input and the output shapes
of your model in features, out features, what data goes in and what data comes out. These values
will be highly dependent on the data that you're working with. And we're going to see different
data or different examples of input features and output features all throughout this course. So
but that is what's happening. We have one in feature to one out feature. Now what's happening
inside nn.linear. Let's have a look torch and then linear. We go the documentation
applies a linear transformation to the incoming data. Where have we seen this before?
y equals x a t plus b. Now they're using different letters, but we've got the same formula as
what's happening up here. Look at the same formula as our data. Wait times x plus bias. And then if
we look up linear regression formula once again, linear regression formula. We've got this formula
here. Now again, these letters can be replaced by whatever letters you like. But this linear layer
is implementing the linear regression formula that we created in our model before. So it's
essentially doing this part for us. And behind the scenes, the layer creates these parameters for us.
So that's a big piece of the puzzle of pie torch is that as I've said, you won't always be
initializing the parameters your model yourself. You'll generally initialize layers. And then you'll

```
use those layers in some Ford computation. So let's see how we could do that. So
we've got a linear
layer which takes us in features one and out features one. What should we do now?
Well, because
we've subclassed nn.module we need to override the Ford method. So we need to tell
our model
what should it do as the Ford computation. And in here it's going to take itself as
input,
as well as x, which is conventional for the input data. And then we're just going
to return
here, self dot linear layer x. Right. And actually, we might use some typing here
to say that this
should be a torch tensor. And it's also going to return a torch dot tensor. That's
using Python's
type ins. So this is just saying, hey, X should be a torch tensor. And I'm going to
return you a
torch tensor, because I'm going to pass x through the linear layer, which is
expecting one in feature
and one out feature. And it's going to this linear transform. That's another word
for it. Again,
pytorch and machine learning in general has many different names of the same thing.
I would call
this linear layer. I'm going to write here, also called linear transform, probing
layer,
fully connected layer, dense layer, intensive flow. So a whole bunch of different
names for
the same thing, but they're all implementing a linear transform. They're all
implementing a
version of linear regression y equals x, a ranspose plus b, in features, out
features,
wonderful. So let's see this in action. So we're going to go set the manual seed so
we can
get reproducibility as well, torch dot manual seed. And we're going to set model
one equals
linear regression. This is model one, because we've already got model zero, linear
regression
```

Section 80: Main Topics

Key Topics:

- This goes behind the scenes of how PyTorch creates its different layers
- So this style of what's going on here is how you're going to see the majority of your PyTorch deep learning models created using pre-existing layers from the torch
- So for all of the common layers in deep learning, because that's what neural networks are, they're layers of different mathematical transformations, PyTorch has a lot of pre-built implementations

▶ 📄 Click to view detailed content

```
V two, and we're going to check model one, and we're going to check its state
dictionary,
state dict. There we go. What do we have inside this ordered dict? Has that not
created anything
for us? Model one, dot state dinked, ordered dink. We haven't got anything here in
the regression
model V two. Ideally, this should be outputting a weight and a bias. Yeah,
variables, weight,
and bias. Let's dig through our code line by line and see what we've got wrong. Ah,
did you notice
this? The init function so the constructor had the wrong amount of underscores. So
it was never
actually constructing this linear layer troubleshooting on the fly team. There we
go. Beautiful. So we
have a linear layer, and we have it is created for us inside a weight and a bias.
So effectively,
we've replaced the code we wrote above for build model, initializing a weight and
bias parameter
with the linear layer. And you might be wondering why the values are slightly
different, even though
we've used the manual seed. This goes behind the scenes of how PyTorch creates its
different
layers. It's probably using a different form of randomness to create different
types of
variables. So just keep that in mind. And to see this in action, we have a
conversion here.
So this is what's going on. We've converted, this is our original model class,
linear regression.
We initialize our model parameters here. We've got a weight and a bias. But
instead, we've
swapped this in our linear regression model V2. This should be V2 to use linear
layer. And then
in the forward method, we had to write the formula manually here when we initialize
the parameters
manually. But because of the power of torch.nn, we have just passed it through the
linear layer,
which is going to perform some predefined forward computation in this layer. So
this
style of what's going on here is how you're going to see the majority of your
PyTorch
deep learning models created using pre-existing layers from the torch.nn module. So
if we go back
into torch.nn, torch.nn, we have a lot of different layers here. So we have
convolutional layers,
pooling layers, padding layers, normalization, recurrent, transformer, linear,
we're using a
linear layer, dropout, et cetera, et cetera. So for all of the common layers in
deep learning,
because that's what neural networks are, they're layers of different mathematical
transformations,
PyTorch has a lot of pre-built implementations. So that's a little bit of a sneaky
trick that
I've done to alter our model. But we've still got basically the exact same model as
we had before.
```

So what's next? Well, it's to train this model. So let's do that in the next video.
Welcome back. So in the last video, we built a PyTorch linear model, nice and simple using a
single nn.linear layer with one in feature, one out feature. And we over read the forward method
of nn.module using the linear layer that we created up here. So what's going to happen is when we do
the forward parser on our model, we're going to put some data in and it's going to go through
this linear layer, which behind the scenes, as we saw with torch and n linear,
behind the scenes, it's going to perform the linear regression formula here. So y equals x,
a t plus b. But now case, we've got weight and bias. So let's go back. It's now time to write
some training code. But before we do, let's set the model to use the target device. And so in
our case, we've got a device of CUDA. But because we've written device agnostic code, if we didn't
have access to a CUDA device, a GPU, our default device would be a CPU. So let's check the model
device. We can do that first up here, check the model current device, because we're going to use
the GPU here, or we're going to write device agnostic code. That's better to say device agnostic code.
That's the proper terminology device. What device are we currently using? This is the CPU, right?
So by default, the model will end up on the CPU. But if we set it to model one call dot two device,
what do you think it's going to do now? If our current target device is CUDA, we've seen what
two does in the fundamental section, two is going to send the model to the GPU memory. So now let's
check whether parameters of our model live dot device. If we send them to the device previously,
it was the CPU, it's going to take a little bit longer while the GPU gets fired up and goes,
PyTorch goes, Hey, I'm about to send you this model. You ready for it? Boom, there we go. Wonderful.
So now our model is on the device or the target device, which is CUDA. And if CUDA wasn't available,
the target device would be CPU. So this would just come out just exactly how we've got it here.
But with that being said, now let's get on to some training code. And this is the fun part.
What do we have to do? We've already seen this for training. I'm just going to clear up our
workspace a little bit here. For training, we need, this is part of the PyTorch workflow,
we need a loss function. What does a loss function do? Measures how wrong our model is,
we need an optimizer, we need a training loop and a testing loop. And the optimizer, what does that
do? Well, it optimizes the parameters of our model. So in our case, model one dot state dig,
what do we have? So we have some parameters here within the linear layer, we have a weight,

```
and we have a bias. The optimizer is going to optimize these random parameters so
that they
```

Section 81: Main Topics

Key Topics:

- Well, pytorch offers a lot of optimizers in torch dot opt in SGD
- LR, which stands for learning rate
- In other words, how big of a step will our optimizer change our parameters with every iteration, a smaller learning rate

▶ 📄 Click to view detailed content

```
hopefully reduce the loss function, which remember the loss function measures how
wrong our model
is. So in our case, because we're working with the regression problem, let's set up
the loss
function. And by the way, all of these steps are part of the workflow. We've got
data ready,
we've built or picked a model, we're using a linear model. Now we're up to here 2.1
pick a loss
function and an optimizer, we're going to do build a training loop in the same
session,
because you know what, we're getting pretty darn good at this, loss function equals
what?
Well, we're going to use l one loss. So let's set that up and then dot l one loss,
which is the
same as ma and if we wanted to set up our optimizer, what optimizer could we use?
Well, pytorch offers
a lot of optimizers in torch dot opt in SGD. That's stochastic gradient descent,
because remember
gradient descent is the algorithm that optimizes our model parameters. Adam is
another popular option.
For now, we're going to stick with SGD. LR, which stands for learning rate. In
other words,
how big of a step will our optimizer change our parameters with every iteration, a
smaller
learning rate. So such as 0001 will be a small step. And then a large learning
rate, such as 0.1
will be a larger step. Too big of a step. Our model learns too much. And it
explodes too small of a
step. Our model never learns anything. But oh, we actually have to pass params
first. I forgot
about that. I got ahead of myself with a learning rate. Params is the parameters
we'd like our
optimizer to optimize. So in our case, it's model one dot parameters, because model
one is our current
```

target model. Beautiful. So we've got a loss function and an optimizer. Now, let's write a training
loop. So I'm going to set torch manual seeds so we can try and get as reproducible as results as
possible. Remember, if you get different numbers to what I'm getting, don't worry too much if they're
not exactly the same, the direction is more important. So that means if my loss function is
getting smaller, yours should be getting smaller too. Don't worry too much if your fourth decimal
place isn't the same as what my values are. So we have a training loop ready to be written here.
Epox, how many should we do? Well, we did 200 last time and that worked pretty well. So let's do 200
again. Did you go through the extra curriculum yet? Did you watch the video for the unofficial
PyTorch optimization loop song yet? This one here, listen to the unofficial PyTorch optimization
loop song. If not, it's okay. Let's sing it together. So for an epoch in range, epochs, we're going to
go through the song in a second. We're going to set the model to train. In our case, it's model one,
model to train. Now, step number one is what? Do the forward pass. This is where we calculate
the predictions. So we calculate the predictions by passing the training data through our model.
And in our case, because the forward method in model one implements the linear layer,
this data is going to go through the linear layer, which is torch.nn.linear and go through
the linear regression formula. And then we calculate the loss, which is how wrong our models predictions
are. So the loss value equals loss fn. And here we're going to pass in y-pred and y-train.
Then what do we do? We zero the optimizer, optimizer zero grad, which because by default,
the optimizer is going to accumulate gradients behind the scenes. So every epoch, we want to
reduce those back to zero. So it starts from fresh. We're going to perform back propagation here,
back propagation, by calling loss, stop backwards. If the forward pass goes forward through the
network, the backward pass goes backwards through the network, calculating the gradients for the
loss function with respect to each parameter in the model. So optimizer step, this next part,
is going to look at those gradients and go, you know what? Which way should I optimize the parameters?
So because the optimizer is optimizing the model parameters, it's going to look at the
loss and go, you know what? I'm going to adjust the weight to be increased. And I'm going to lower
the bias and see if that reduces the loss. And then we can do testing. We can do both of these in
the same hit. Now we are moving quite fast through this because we spent a whole bunch of time
discussing what's going on here. So for testing, what do we do? We set the model

```
into evaluation
mode. That's going to turn off things like dropout and batch normalization layers.
We don't have any
of that in our model for now, but just it's good practice to always call a vowel
whenever you're
doing testing. And same with inference mode. We don't need to track gradients and a
whole bunch of
other things PyTorch does behind the scenes when we're testing or making
predictions. So we use
the inference mode context manager. This is where we're going to create test pred,
which is going
to be our test predictions, because here we're going to pass the test data
features, forward
pass through our model. And then we can calculate the test loss, which is our loss
function. And we're
going to compare the test pred to Y test. Wonderful. And then we can print out
what's happening.
So what should we print out? How about if epoch divided by 10 equals zero. So every
10 epochs,
let's print something out, print. We'll do an F string here, epoch is epoch. And
then we'll go
```

Section 82: Main Topics

Key Topics:

- Oh, of course
- Yes, of course, that's what's happened
- So remember, one of the biggest issues with pytorch aside from shape errors is that you should have your data or all of the things that you're computing with on the same device

▶ 📄 Click to view detailed content

```
loss, which is the training loss, and just be equal to the loss. And then we'll go
test loss is
equal to test loss. So do you think this will work? It's okay if you're not sure.
But let's find
out together, hey, oh, we've got a, we need a bracket there. Oh my goodness, what's
going on?
Run time error. Expected all tenses to be on the same device. Oh, of course. Do you
know what's
happening here? But we found at least two devices, CUDA and CPU. Yes, of course,
that's what's happened.
So what have we done? Up here, we put our model on the GPU. But what's going on
here? Our data?
Has our data on the GPU? No, it's not. By default, it's on the CPU. So we haven't
written device
```

agnostic code for our data. So let's write it here, put data on the target device.
Device agnostic code for data. So remember, one of the biggest issues with pytorch aside from
shape errors is that you should have your data or all of the things that you're computing with
on the same device. So that's why if we set up device agnostic code for our model,
we have to do the same for our data. So now let's put X train to device. Y train equals Y train
to device. This is going to create device agnostic code. In our case, it's going to use CUDA because
we have access to a CUDA device. But if we don't, this code will still work. It will still default
to CPU. So this is good. I like that we got that error because that's the sum of the things you're
going to come across in practice, right? So now let's run this. What's happening here?
Hey, look at that. Wonderful. So our loss starts up here nice and high. And then it starts to go
right down here for the training data. And then the same for the testing data. Beautiful.
Right up here. And then all the way down. Okay. So this looks pretty good on the test data set. So
how can we check this? How can we evaluate our model? Well, one way is to check its state
deck. So state decked. What do we got here? What are our weight and bias? Oh my gosh, so close.
So we just set weight and bias before to be 0.7 and 0.3. So this is what our model has estimated
our parameters to be based on the training data. 0.6968. That's pretty close to 0.7,
nearly perfect. And the same thing with the bias 0.3025 versus the perfect value is 0.93. But remember,
in practice, you won't necessarily know what the ideal parameters are. This is just to exemplify
what our model is doing behind the scenes. It's moving towards some ideal representative
parameters of whatever data we're working with. So in the next video, I'd like you to give it a go
of before we get to the next video, make some predictions with our model and plot them on the
original data. How close to the green dots match up with the red dots? And you can use this plot
predictions formula or function that we've been using in the past. So give that a go and I'll
see you in the next video. But congratulations. Look how quickly we just trained a model using
the steps that we've covered in a bunch of videos so far and device agnostic code. So good.
I'll see you soon. In the last video, we did something very, very exciting. We worked through
training an entire neural network. Some of these steps took us an hour or so worth of videos to
go back through before. But we coded that in one video. So you're ready listening the song just to
remind ourselves of what's going on. For an epoch in a range, call model dot train, do the forward
pass, calculate the loss, optimizer zero grad, loss backward, optimizer step, step,

```
step, let's
test, come on a dot eval with torch inference mode, do the forward pass, calculate
the loss,
print out what's happening. And then we do it again, again, again, for another
epoch in a range.
Now I'm kidding. We'll just leave it there. We'll just leave it there. But that's
the
unofficial pytorch optimization loop song. We created some device agnostic code so
that we could
make the calculations on the same device as what our model is because the models
also using device
agnostic code. And so now we've got to evaluate our models. We've looked at the
loss and the test
lost here. And we know that our models loss is going down. But what does this
actually equate to
when it makes predictions? That's what we're most interested in, right? And we've
looked at the
parameters. They're pretty close to the ideal parameters. So at the end of last
video, I issued
you the challenge to making and evaluating predictions to make some predictions and
plot them. I hope
you gave it a shot. Let's see what it looks like together. Hey, so turn the model
into evaluation
mode. Why? Because every time we're making predictions or inference, we want our
model to be in a
vowel mode. And every time we're training, we want our model to be in training
mode. And then we're
going to make predictions on the test data, because we train on the train data, and
we evaluate our
model on the test data data that our model has never actually seen, except for when
it makes
predictions. With torch inference mode, we turn on inference mode whenever we make
inference or
predictions. So we're going to set Y threads equal to model one, and the test data
goes in here.
Let's have a look at what the Y threads look like. Wonderful. So we've got a tensor
here. It shows
```

Section 83: Main Topics

Key Topics:

- Oh, of course
- It uses matplotlib, of course, and matplotlib works with NumPy, not pytorch
- So of course, we're running into another error down here, because we just said that our predictions are on the CUDA device

▶ 📄 Click to view detailed content

```
us that they're still on the device CUDA. Why is that? Well, that's because
previously we set the
model one to the device, the target device, the same with the test data. So
subsequently,
our predictions are also on the CUDA device. Now, let's bring in the plot
predictions function here.
So check out our model predictions visually. We're going to adhere to the data
explorer's motto
of visualize visualize visualize plot predictions. And predictions are going to be
set to
equals Y threads. And let's have a look. How good do these look? Oh, no.
Oh, we've got another error type error. Can't convert CUDA device type tensor to
NumPy.
Oh, of course. Look what we've done. So our plot predictions function, if we go
back up,
where did we define that? What does our plot predictions function use? It uses
matplotlib,
of course, and matplotlib works with NumPy, not pytorch. And NumPy is CPU based. So
of course,
we're running into another error down here, because we just said that our
predictions are on the CUDA
device. They're not on the CPU. They're on a GPU. So it's giving us this helpful
information here.
Use tensor dot CPU to copy the tensor to host memory first. So this is our tensor.
Let's call
dot CPU and see what happens then. Is that going to go to CPU? Oh, my goodness.
Look at that.
Look at that. Go the linear layer. The red dots, the predictions are basically on
top of the testing
data. That is very exciting. Now again, you may not get the exact same numbers
here, and that is
perfectly fine. But the direction should be quite similar. So your red dots should
be basically on
top of the green dots, if not very slightly off. But that's okay. That's okay. We
just want to focus
on the direction here. So thanks to the power of back propagation here and gradient
descent,
our models random parameters have updated themselves to be as close as possible to
the ideal parameters.
And now the predictions are looking pretty darn good for what we're trying to
predict.
But we're not finished there. We've just finished training this model. What would
happen if our
notebook disconnected right now? Well, that wouldn't be ideal, would it? So in the
next part,
we're going to move on to 6.5, saving, and loading a trained model. So I'm going to
give you a
challenge here as well, is to go ahead and go back and refer to this code here,
saving model
in PyTorch, loading a PyTorch model, and see if you can save model one, the state
dictionary of
model one, and load it back in and get something similar to this. Give that a shot,
and I'll see you
in the next video. Welcome back. In the last video, we saw the power of the
torch.nn.linear layer,
and back propagation and gradient descent. And we've got some pretty darn good
```

predictions
out of our model. So that's very exciting. Congratulations. You've now trained two machine
learning models. But it's not over yet. We've got to save and load our trained model. So I
issued you the challenge in the last video to try and save and load the model yourself. I hope
you gave that a go. But we're going to do that together in this video. So we're going to start
by importing path because we would like a file path to save our model to. And the first step we're
going to do is create models directory. We don't have to recreate this because I believe we already
have one. But I'm going to put the code here just for completeness. And this is just so if you
didn't have a models directory, this would create one. So model path is going to go to path
models. And then we'd like to model path dot maker, we're going to call maker for make directory.
We'll set parents equal to true. And if it exists, okay, that'll also be true. So we won't get an error.
Oh my gosh, Google collab. I didn't want that. We won't get an error if it already exists.
And two, we're going to create a model save path. So if you recall that pytorch objects in general
have the extension of what? There's a little pop quiz before we get to the end of this sentence.
So this is going to be pytorch workflow for this module that we're going through. This one here,
chapter 01 pytorch workflow model one. And they usually have the extension dot PT for pytorch or
PT H for pytorch as well. I like PT H. But just remember, sometimes you might come across slightly
different versions of that PT or PT H. And we're going to create the model save name or the save
path. It's probably a better way to do it is going to be model path. And then we can use because we're
using the path lib module from Python, we can save it under model name. And so if we look at this,
what do we get model save path? We should get Oh, path is not defined. Oh, too many capitals here,
Daniel. The reason why I'm doing these in capitals is because oftentimes hyper parameters such as epochs
in machine learning are set as hyper parameters LR could be learning rate. And then you could have
as well model name equals Yeah, yeah, yeah. But that's just a little bit of nomenclature trivia for
later on. And model save path, we've done that. Now we're going to save the model state dictionary
rather than the whole model, save the model state deck, which you will find the pros and cons of
in where in the pytorch documentation for saving and loading model, which was a little bit of extra

Section 84: Main Topics

Key Topics:

- And the file path we're going to use is, of course, the model save path, which we've seen here is a POSIX path
- This one for us from the workflow we did before up here, saving a model in PyTorch, loading a PyTorch model
- And now the one we've got, of course, model one is the one that we've just saved

▶ 📄 Click to view detailed content

```
curriculum for a previous video. But let's have a look at our model save path will
print it out.
And we'll go torch save, we'll set the object that we're trying to save to equal
model one dot state
deck, which is going to contain our trained model parameters. We can inspect what's
going on in here,
state deck. They'll show us our model parameters. Remember, because we're only
using a single linear
layer, we only have two parameters. But in practice, when you use a model with
maybe hundreds of layers
or tens of millions of parameters, viewing the state deck explicitly, like we are
now,
might not be too viable of an option. But the principle still remains a state deck
contains
all of the models trained or associated parameters, and what state they're in. And
the file path we're
going to use is, of course, the model save path, which we've seen here is a POSIX
path. Let's save
our model. Wonderful saving model to this file path here. And if we have a look at
our folder,
we should have two saved models now, beautiful to save models. This one for us from
the workflow
we did before up here, saving a model in PyTorch, loading a PyTorch model. And now
the one we've got,
of course, model one is the one that we've just saved. Beautiful. So now let's load
a model. We're
going to do both of these in one video. Load a PyTorch model. You know what,
because we've had a
little bit of practice so far, and we're going to pick up the pace. So let's go
loaded, let's call
it, we'll create a new instance of loaded model one, which is, of course, our
linear regression model
V2, which is the version two of our linear regression model class, which
subclasses, what?
Subclasses and n.module. So if we go back here up here to where we created it. So
linear regression
model V2 uses a linear layer rather than the previous iteration of linear
regression model,
which we created right up here. If we go up to here, which explicitly defined the
```

parameters,
and then implemented a linear regression formula in the forward method, the difference between
what we've got now is we use PyTorch's pre-built linear layer, and then we call that linear layer
in the forward method, which is probably the far more popular way of building PyTorch models,
is stacking together pre-built NN layers, and then calling them in some way in the forward method.
So let's load it in. So we'll create a new instance of linear regression model V2, and now what do
we do? We've created a new instance, I'm just going to get out of this, make some space for us.
We want to load the model state deck, the saved model one state deck, which is the state deck that
we just saved beforehand. So we can do this by going loaded model one, calling the load state
decked method, and then passing it torch dot load, and then the file path of where we saved that
PyTorch object before. But the reason why we use the path lib is so that we can just call model
save path in here. Wonderful. And then let's check out what's going on. Or actually, we need to
put the target model or the loaded model to the device. The reason being is because we're doing all
our computing with device agnostic code. So let's send it to the device. And I think that'll be about
it. Let's see if this works. Oh, there we go. Linear regression model V2 in features one,
out features one, bias equals true. Wonderful. Let's check those parameters. Hey, next loaded model
one dot parameters. Are they on the right device? Let's have a look. Beautiful. And let's just check
the loaded state dictionary of loaded model one. Do we have the same values as we had previously?
Yes, we do. Okay. So to conclusively make sure what's going on, let's evaluate the loaded model.
Evaluate loaded model, loaded model one. What do we do for making predictions? Or what do we do to
evaluate? We call dot a vowel. And then if we're going to make some predictions, we use torch
inference mode with torch inference mode. And then let's create loaded model one, threads
equals loaded model one. And we'll pass it the test data. And now let's check for a quality
between Y threads, which is our previous model one preds that we made up here, Y threads.
And we're going to compare them to the fresh loaded model one preds. And should they be the same?
Yes, they are beautiful. And we can see that they're both on the device CUDA. How amazing is that? So
I want to give you a big congratulations, because you've come such a long way. We've gone through
the entire PyTorch workflow from making data, preparing and loading it to building a model.
All of the steps that come in building a model, there's a whole bunch there, making predictions,

```
training a model, we spent a lot of time going through the training steps. But
trust me, it's
worth it, because we're going to be using these exact steps all throughout the
course. And in fact,
you're going to be using these exact steps when you build PyTorch models after this
course. And
then we looked at how to save a model so we don't lose all our work, we looked at
loading a model,
and then we put it all together using the exact same problem, but in far less time.
And as you'll
see later on, we can actually make this even quicker by functionalizing some of the
code we've already
written. But I'm going to save that for later. I'll see you in the next video,
where I'm just
```

Section 85: Main Topics

Key Topics:

- going to show you where you can find some exercises and all of the extra curriculum I've been talking about throughout this section 01 PyTorch workflow
- In the last video, we finished up putting things together by saving and loading our trained model, which is super exciting, because let's come to the end of the PyTorch workflow section
- So within the book version of the course materials, which is at learnpytorch

▶ 📄 Click to view detailed content

```
going to show you where you can find some exercises and all of the extra curriculum
I've been talking
about throughout this section 01 PyTorch workflow. I'll see you there. Welcome
back. In the last
video, we finished up putting things together by saving and loading our trained
model, which is
super exciting, because let's come to the end of the PyTorch workflow section. So
now, this section
is going to be exercises and extra curriculum, or better yet, where you can find
them. So I'm
going to turn this into markdown. And I'm going to write here for exercises and
extra curriculum.
Refer to. So within the book version of the course materials, which is at
learnpytorch.io,
we're in the 01 section PyTorch workflow fundamentals. There'll be more here by the
time you watch
this video likely. And then if we go down here, at the end of each of these
sections, we've got
the table of contents over here. We've got exercises and extra curriculum. I listed
```

a bunch of things
throughout this series of 01 videos, like what's gradient descent and what's back
propagation. So
I've got plenty of resources to learn more on that. There's the loading and saving
PyTorch
documentation. There's the PyTorch cheat sheet. There's a great article by Jeremy
Howard for a
deeper understanding of what's going on in torch.nn. And there's, of course, the
unofficial PyTorch
optimization loop song by yours truly, which is a bit of fun. And here's some
exercises. So
the exercises here are all based on the code that we wrote throughout section 01.
So there's
nothing in the exercises that we haven't exactly covered. And if so, I'll be sure
to put a note
in the exercise itself. But we've got create a straight line data set using the
linear regression
formula. And then build a model by subclassing and end up module. So for these
exercises, there's an
exercise notebook template, which is, of course, linked here. And in the PyTorch
deep learning
GitHub, if we go into here, and then if we go into extras, and if we go into
exercises, you'll
find all of these templates here. They're numbered by the same section that we're
in. This is PyTorch
workflow exercises. So if you wanted to complete these exercises, you could click
this notebook
here, open in Google CoLab. I'll just wait for this to load. There we go. And you
can start to
write some code here. You could save a copy of this in your own Google Drive and go
through this.
It's got some notes here on what you should be doing. You can, of course, refer to
the text-based
version of them. They're all here. And then if you want an example of what some
solutions look
like, now, please, I can't stress enough that I would highly, highly recommend
trying the exercises
yourself. You can use the book that we've got here. This is just all the code from
the videos.
You can use this. You can use, I've got so many notebooks here now, you can use all
of the code
that we've written here to try and complete the exercises. But please give them a
go yourself.
And then if you go back into the extras folder, you'll also find solutions. And
this is just one
example solutions for section 01. But I'm going to get out of that so you can't
cheat and look
at the solutions first. But there's a whole bunch of extra resources all contained
within
the PyTorch deep loaning repo, extras, exercises, solutions, and they're also in
the book version
of the course. So I'm just going to link this in here. I'm going to put this right
at the bottom
here. Wonderful. But that is it. That is the end of the section 01 PyTorch
workflow.
So exciting. We went through basically all of the steps in a PyTorch workflow,
getting data ready, turning into tenses, build or pick a model, picking a loss

```
function on an
optimizer. We built a training loop. We fit the model to the data. We made a
prediction.
We evaluated our model. We improved through experimentation by training for more
epochs.
We'll do more of this later on. And we saved and reload our trained model.
But that's going to finish 01. I will see you in the next section. Friends, welcome
back.
We've got another very exciting module. You ready? Neural network classification
with
PyTorch. Now combining this module once we get to the end with the last one, which
was
regression. So remember classification is predicting a thing, but we're going to
see this in a second.
And regression is predicting a number. Once we've covered this, we've covered two
of the
the biggest problems in machine learning, predicting a number or predicting a
thing.
So let's start off with before we get into any ideas or code, where can you get
help?
First things first is follow along with the code. If you can, if in doubt, run the
code.
Try it for yourself. Write the code. I can't stress how important this is.
If you're still stuck, press shift, command, and space to read the doc string of
any of the
functions that we're running. If you are on Windows, it might be control. I'm on a
Mac, so I put command
here. If you're still stuck, search for your problem. If an error comes up, just
copy and paste
that into Google. That's what I do. You might come across resources like Stack
Overflow or,
of course, the PyTorch documentation. We'll be referring to this a lot again
throughout this
section. And then finally, oh wait, if you're still stuck, try again. If in doubt,
run the code.
And then finally, if you're still stuck, don't forget, you can ask a question. The
best place to
```

Section 86: Main Topics

Key Topics:

- do so will be on the course GitHub, which will be at the discussions page, which is linked here
- If we load this up, there's nothing here yet, because as I record these videos, the course hasn't launched yet, but press new discussion
- Finally, don't forget that this notebook that we're about to go through is based on chapter two of the Zero to Mastery Learn PyTorch for deep learning, which is

neural network classification with PyTorch

▶ 📄 Click to view detailed content

```
do so will be on the course GitHub, which will be at the discussions page, which is
linked here.
If we load this up, there's nothing here yet, because as I record these videos, the
course
hasn't launched yet, but press new discussion. Talk about what you've got. Problem
with XYZ.
Let's go ahead. Leave a video number here and a timestamp, and that way, we'll be
able to help
you out as best as possible. So video number, timestamp, and then your question
here, and you
can select Q&A. Finally, don't forget that this notebook that we're about to go
through is based
on chapter two of the Zero to Mastery Learn PyTorch for deep learning, which is
neural network
classification with PyTorch. All of the text-based code that we're about to write
is here. That
was a little spoiler. And don't forget, this is the home page. So my GitHub repo
slash PyTorch
deep learning for all of the course materials, everything you need will be here. So
that's very
important. How can you get help? But this is the number one. Follow along with the
code and try
to write it yourself. Well, with that being said, when we're talking about
classification,
what is a classification problem? Now, as I said, classification is one of the main
problems of
machine learning. So you probably already deal with classification problems or
machine learning
powered classification problems every day. So let's have a look at some examples.
Is this email
spam or not spam? Did you check your emails this morning or last night or whenever?
So chances are
that there was some sort of machine learning model behind the scenes. It may have
been a neural
network. It may have not that decided that some of your emails won't spam. So to
Daniel,
at mrdberg.com, hey, Daniel, this steep learning course is incredible. I can't wait
to use what
I've learned. Oh, that's such a nice message. If you want to send that email
directly to me,
you can. That's my actual email address. But if you want to send me this email,
well, hopefully
my email, which is hosted by some email service detects this as spam because
although that is a
lot of money and it would be very nice, I think if someone can't spell too well,
are they really
going to pay me this much money? So thank you email provider for classifying this
as spam. And now
because this is one thing or another, not spam or spam, this is binary
classification. So in this
case, it might be one here and this is a zero or zero or one. So one thing or
```

another, that's binary
classification. If you can split it into one thing or another, binary classification. And then we
have an example of say we had the question, we asked our photos app on our smartphone or whatever
device you're using. Is this photo of sushi steak or pizza? We wanted to search our photos for every
time we've eaten sushi or every time we've eaten steak or every time we've eaten pizza far out
and this looks delicious. But this is multi class classification. Now, why is this? Because we've
got more than two things. We've got 123. And now this could be 10 different foods. It could be 100
different foods. It could be 1000 different categories. So the image net data set, which is a popular
data set for computer vision, image net, we go to here, does it say 1000 anywhere, 1k or 1000?
No, it doesn't. But if we go image net 1k, download image net data, maybe it's here.
It won't say it, but you just, oh, there we go, 1000 object classes. So this is multi class
classification because it has 1000 classes, that's a lot, right? So that's multi class classification,
more than one thing or another. And finally, we might have multi label classification,
which is what tags should this article have when I first got into machine learning, I got these
two mixed up a whole bunch of times. Multi class classification has multiple classes such as sushi
steak pizza, but assigns one label to each. So this photo would be sushi in an ideal world. This is
steak and this is pizza. So one label to each. Whereas multi label classification means you could
have multiple different classes. But each of your target samples such as this Wikipedia article,
what tags should this article have? It may have more than one label. It might have three labels,
it might have 10 labels. In fact, what if we went to the Wikipedia page for deep learning Wikipedia
and does it have any labels? Oh, there we go. Where was that? I mean, you can try this yourself.
This is just the Wikipedia page for deep learning. There is a lot, there we go categories deep
learning, artificial neural networks, artificial intelligence and emerging technologies. So that
is an example. If we wanted to build a machine learning model to say, read all of the text in
here and then go tell me what are the most relevant categories to this article? It might come up
with something like these. In this case, because it has one, two, three, four, it has multiple labels
rather than just one label of deep learning, it could be multi label classification. So we'll go
back. But there's a few more. These will get you quite far in the world of classification.
So let's dig a little deeper on binary versus multi class classification. You may have already

```
experienced this. So in my case, if I search on my phone in the photos app for
photos of a dog,
it might come here. If I search for photos of a cat, it might come up with this.
But if I wanted
```

Section 87: Main Topics

Key Topics:

- In other words, because remember, machine learning models, neural networks love to have numerical inputs
- Well, of course, we're going to be part cook, part chemist, part artist, part science
- Well, let's break it down to inputs, some kind of machine learning algorithm, and then outputs

▶ 📄 Click to view detailed content

```
to train an algorithm to detect the difference between photos of these are my two
dogs.
Aren't they cute? They're nice and tired and they're sleeping like a person. This
is seven.
Number seven, that's her name. And this is Bella. This is a cat that me and my
partner rescued.
And so I'm not sure what this cat's name is actually. So I'd love to give it a
name, but I can't.
So binary classification, if we wanted to build an algorithm, we wanted to feed it,
say, 10,000
photos of dogs and 10,000 photos of cats. And then we wanted to find a random image
on the
internet and pass it through to our model and say, hey, is this a dog or is this a
cat? It would
be binary classification because the options are one thing or another dog or cat.
But then for
multi-class classification, let's say we've been working on a farm and we've been
taking some photos
of chickens because they groovy, right? Well, we updated our model and added some
chicken photos
in there. We would now be working with a multi-class classification problem because
we've got more
than one thing or another. So let's jump in to what we're going to cover. This is
broadly,
by the way, because this is just text on a page. You know, I like to just write
code of what we're
actually doing. So we're going to look at the architecture of a neural network
classification
model. We're going to check what the input shapes and output shapes of a
classification model are
features and labels. In other words, because remember, machine learning models,
```

neural networks
love to have numerical inputs. And those numerical inputs often come in tenses. Tenses have different
shapes, depending on what data you're working with. We're going to see all of this in code, creating
custom data to view, fit and predict on. We're going to go back through our steps in modeling.
We covered this a fair bit in the previous section, but creating a model for neural network classification.
It's a little bit different to what we've done, but not too out landishly different. We're going to
see how we can set up a loss function and an optimizer for a classification model. We'll
recreate a training loop and a evaluating loop or a testing loop. We'll see how we can save and
load our models. We'll harness the power of nonlinearity. Well, what does that even mean? Well,
if you think of what a linear line is, what is that? It's a straight line. So you might be
able to guess what a nonlinear line looks like. And then we'll look at different classification
evaluation methods. So ways that we can evaluate our classification models. And how are we going
to do all of this? Well, of course, we're going to be part cook, part chemist, part artist, part
science. But for me, I personally prefer the cook side of things because we're going to be cooking
up lots of code. So in the next video, before we get into coding, let's do a little bit more on
what are some classification inputs and outputs. I'll see you there. Welcome back. In the last
video, we had a little bit of a brief overview of what a classification problem is. But now,
let's start to get more hands on by discussing what the actual inputs to a classification problem
look like and the outputs look like. And so let's say we had our beautiful food photos from before,
and we were trying to build this app here called maybe food vision to understand what
foods are in the photos that we take. And so what might this look like? Well, let's break it down
to inputs, some kind of machine learning algorithm, and then outputs. In this case,
the inputs we want to numerically represent these images in some way, shape or form. Then we want
to build a machine learning algorithm. Hey, one might actually exist. We're going to see this later
on in the transfer learning section for our problem. And then we want some sort of outputs. And in
the case of food vision, we want to know, okay, this is a photo of sushi. And this is a photo of
steak. And this is a photo of pizza. You could get more hands on and technical and complicated, but
we're just going to stick with single label multi class classification. So it could be a sushi photo,
it could be a steak photo, or it could be a pizza photo. So how might we numerically represent
these photos? Well, let's just say we had a function in our app that every photo

```
that gets taken
automatically gets resized into a square into 224 width and 224 height. This is
actually quite a
common dimensionality for computer vision problems. And so we've got the width
dimension, we've got
the height, and then we've got this C here, which isn't immediately recognizable.
But in the case
of pictures, they often get represented by width, height color channels. And the
color channels is
red, green and blue, which is each pixel in this image has some value of red, green
or blue, that
makes whatever color is displayed here. And this is one way that we can numerically
represent an
image by taking its width, its height and color channels, and whatever number makes
up this
particular image. We're going to see this later on when we work with computer
vision problems.
So we create a numerical encoding, which is the pixel values here. Then we import
the pixel values
of each of these images into a machine learning algorithm, which is often already
exists. And if
it doesn't exist for our particular problem, hey, well, we're learning the skills
to build them now,
```

Section 88: Main Topics

Key Topics:

- we could use pytorch to build a machine learning algorithm for this
- Well, in this case, these are prediction probabilities, which the outputs of machine learning models are never actually discrete, which means it is definitely pizza
- That's the whole idea of machine learning, is that if you adjust the algorithm, if you adjust the data, you can improve your predictions

▶ 📄 Click to view detailed content

```
we could use pytorch to build a machine learning algorithm for this. And then
outputs, what might
these look like? Well, in this case, these are prediction probabilities, which the
outputs of
machine learning models are never actually discrete, which means it is definitely
pizza.
It will give some sort of probability value between zero and one for say the closer
to one,
the more confident our model is that it's going to be pizza. And the closer to zero
is means that,
hey, this photo of pizza, let's say this one, and we're trying to predict sushi.
Well,
```

it doesn't think that it's sushi. So it's giving it quite a low value here. And
then the same for
steak, but it's really high, the value here for pizza. We're going to see this
hands on. And then
it's the opposite here. So it might have got this one wrong. But with more training
and more data,
we could probably improve this prediction. That's the whole idea of machine
learning,
is that if you adjust the algorithm, if you adjust the data, you can improve your
predictions. And so
the ideal outputs that we have here, this is what our models going to output. But
for our case of
building out food vision, we want to bring them back to. So we could just put all
of these numbers
on the screen here, but that's not really going to help people. We want to put out
labels of what's
going on here. So we can write code to transfer these prediction probabilities into
these labels
too. And so how did these labels come about? How do these predictions come about?
Well,
it comes from looking at lots of different samples. So this loop, we could keep
going,
improve these, find the ones where it's wrong, add more images here, train the
model again,
and then make our app better. And so if we want to look at this from a shape
perspective,
we want to create some tenses for an image classification example. So we're
building food vision.
We've got an image again, this is just reiterating on some of the things that we've
discussed.
We've got a width of 224 and a height of 224. This could be different. This could
be 300, 300.
This could be whatever values that you decide to use. Then we numerically encoded
in some way,
shape or form. We use this as the inputs to our machine learning algorithm, because
of what?
Computers and machine learning algorithms, they love numbers. They can find
patterns in here
that we couldn't necessarily find. Or maybe we could, if you had a long enough
time,
but I'd rather write an algorithm to do it for me. Then it has some outputs,
which comes in the formal prediction probabilities, the closer to one, the more
confident model is
and saying, hey, I'm pretty damn confident that this is a photo of sushi. I don't
think it's a
photo of steak. So I'm giving that zero. It might be a photo of pizza, but I don't
really think so.
So I'm giving it quite a low prediction probability. And so if we have a look at
what the shapes are
for our tenses here, if this doesn't make sense, don't worry. We're going to see
the code to do
all of this later on. But for now, we're just focusing on a classification input
and output.
The big takeaway from here is numerical encoding, outputs and numerical encoding.
But we want to
change these numerical codings from the outputs to something that we understand,
say the word sushi.

```
But this tensor may be batch size. We haven't seen what batch size is. That's all
right. We're
going to cover it. Color channels with height. So this is represented as a tensor
of dimensions.
It could be none here. None is a typical value for a batch size, which means it's
blank. So when
we use our model and we train it, all the code that we write with pytorch will fill
in this behind
the scenes. And then we have three here, which is color channels. And we have 224,
which is the width.
And we have 224 as well, which is the height. Now there is some debate in the field
on the ordering.
We're using an image as our particular example here on the ordering of these
shapes. So say,
for example, you might have height width color channels, typically width and height
come together
in this order. Or they're just side by side in the tensor in terms of their whether
dimension
appears. But color channels sometimes comes first. That means after the batch size
or at the end here.
But pytorch, the default for now is color channels with height, though you can
write code to change
this order because tenses are quite flexible. And so or the shape could be 32 for
the batch size,
three, two, two, four, two, two, four, because 32 is a very common batch size. And
you don't believe me?
Well, let's go here. Yarn LeCoon 32 batch size. Now what is a batch size? Great
tweet. Just keep
this in mind for later on. Training with large mini batches is bad for your health.
More importantly,
it's bad for your test error. Friends don't let friends use mini batches larger
than 32. So this
is quite an old tweet. However, it still stands quite true. Because like today,
it's 2022 when
I'm recording these videos, there are batch sizes a lot larger than 32. But 32
works pretty darn
well for a lot of problems. And so this means that if we go back to our slide, that
if we use
a batch size of 32, our machine learning algorithm looks at 32 images at a time.
Now why does it do
this? Well, because sadly, our computers don't have infinite compute power. In an
ideal world,
```

Section 89: Main Topics

Key Topics:

- Now, of course, as you could imagine, these might change depending on the problem you're working with

- And by the way, this has all come from, if we go to the book version of the course, we've got what is a classification problem
- So all of this text is available at learnpytorch

▶ 📄 Click to view detailed content

```
we look at thousands of images at a time, but it turns out that using a multiple of eight here
is actually quite efficient. And so if we have a look at the output shape here, why is it three?
Well, because we're working with three different classes, one, two, three. So we've got shape equals
three. Now, of course, as you could imagine, these might change depending on the problem you're working
with. So say if we just wanted to predict if a photo was a cat or a dog, we still might have this
same representation here because this is the image representation. However, the shape here
may be two, or will be two because it's cat or dog, rather than three classes here, but a little
bit confusing as well with binary classification, you could have the shape just being one here.
But we're going to see this all hands on. Just remember, the shapes vary with whatever problem
you're working on. The principle of encoding your data as a numerical representation stays the same
for the inputs. And the outputs will often be some form of prediction probability based on whatever
class you're working with. So in the next video, right before we get into coding, let's just discuss
the high level architecture of a classification model. And remember, architecture is just like
the schematic of what a neural network is. I'll see you there. Welcome back. In the last video,
we saw some example classification inputs and outputs. The main takeaway that the inputs to a
classification model, particularly a neural network, want to be some form of numerical
representation. And the outputs are often some form of prediction probability. So let's discuss
the typical architecture of a classification model. And hey, this is just going to be text
on a page, but we're going to be building a fair few of these. So we've got some hyper parameters
over here. We've got binary classification. And we've got multi class classification. Now,
there are some similarities between the two in terms of what problem we're working with.
But there also are some differences here. And by the way, this has all come from, if we go
to the book version of the course, we've got what is a classification problem. And we've got
architecture of a classification neural network. So all of this text is available at learnpytorch.io
```

and in section two. So we come back. So the input layer shape, which is typically decided by the parameter in features, as you can see here, is the same of number of features.
So if we were working on a problem, such as we brought it to predict whether someone had
heart disease or not, we might have five input features, such as one for age, a number for age,
it might be in my case, 28, sex could be male, height, 180 centimeters. If I've been growing
overnight, it's really close to 177. Wait, well, it depends on how much I've eaten, but it's around
about 75 kilos and smoking status, which is zero. So it could be zero or one, because remember,
we want numerical representation. So for sex, it could be zero for males, one for female,
height could be its number, weight could be its number as well. All of these numbers could be
more, could be less as well. So this is really flexible. And it's a hyper parameter. Why? Because
we decide the values for each of these. So in the case of our image prediction problem,
we could have in features equals three for number of color channels. And then we go
hidden layers. So there's the blue circle here. I forgot that this was all timed and colorful.
But let's just discuss hidden layers. Each of these is a layer and n dot linear and n dot linear
and n dot relu and n dot linear. So that's the kind of the syntax you'll see in PyTorch for a
layer is nn dot something. Now, there are many different types of layers in this in PyTorch.
If we go torch and n, basically everything in here is a layer in a neural network. And then if we
look up what a neural network looks like, neural network, recall that all of these are different
layers of some kind of mathematical operation. Input layer, hidden layer, you could have as
many hidden layers as you want. Do we have ResNet architecture? The ResNet architecture,
some of them have 50 layers. Look at this. Each one of these is a layer. And this is only the
34 layer version. I mean, there's ResNet 152, which is 152 layers. We're not at that yet.
But we're working up the tools to get to that stage. Let's come back to here. The neurons per
hidden layer. So we've got these, out features, the green circle, the green square. Now, this is,
if we go back to our neural network picture, this is these. Each one of these little things
is a neuron, some sort of parameter. So if we had 100, what would that look like? Well,
we'd have a fairly big graphic. So this is why I like to teach with code because you could customize
this as flexible as you want. So behind the scenes, PyTorch is going to create 100 of these little
circles for us. And within each circle is what? Some sort of mathematical operation.
So if we come back, what do we got next? Output layer shape. So this is how many

```
output features
we have. So in the case of binary classification is one, one class or the other.
We're going to
see this later on. Multi-class classification is you might have three output
features,
one per class, e.g., one for food, person or dog, if you're building a food, person
or dog,
```

Section 90: Main Topics

Key Topics:

- Relu, which is a rectified linear unit, but can be many others because PyTorch, of course, has what
- So for binary classification, we might use binary cross entropy loss in PyTorch, and for multi-class classification, we might just use cross entropy rather than binary cross entropy
- Another common option is the atom optimizer, and of course, the torch

▶ 📄 Click to view detailed content

```
image classification model. Hidden layer activation, which is, we haven't seen
these yet.
Relu, which is a rectified linear unit, but can be many others because PyTorch, of
course, has what?
Has a lot of non-linear activations. We're going to see this later on. Remember,
I'm kind of planting
the seed here. We've seen what a linear line is, but I want you to imagine what a
non-linear line is.
It's going to be a bit of a superpower for our classification problem. What else do
we have?
Output activation. We haven't got that here, but we'll also see this later on,
which could be
sigmoid for, which is generally sigmoid for binary classification, but softmax for
multi-class
classification. A lot of these things are just names on a page. We haven't seen
them yet.
I like to teach them as we see them, but this is just a general overview of what
we're going to
cover. Loss function. What loss function or what does a loss function do? It
measures how
wrong our model's predictions are compared to what the ideal predictions are. So
for binary
classification, we might use binary cross entropy loss in PyTorch, and for multi-
class
classification, we might just use cross entropy rather than binary cross entropy.
Get it?
```

Binary classification? Binary cross entropy? And then optimizer. SGD is stochastic gradient descent.
We've seen that one before. Another common option is the atom optimizer, and of course,
the torch.optim package has plenty more options. So this is an example multi-class classification
problem. This network here. Why is that? And we haven't actually seen an end up sequential,
but as you could imagine, sequential stands for it just goes through each of these steps.
So multi-class classification, because it has three output features, more than one thing or
another. So three for food, person or dog, but going back to our food vision problem,
we could have the input as sushi, steak, or pizza. So we've got three output features,
which would be one prediction probability per class of image. We have three classes, sushi,
steak, or pizza. Now, I think we've done enough talking here, and enough just pointing to text
on slides. How about in the next video? Let's code. I'll see you in Google CoLab.
Welcome back. Now, we've done enough theory of what a classification problem is, what the inputs
and outputs are and the typical architecture. Let's get in and write some code. So I'm going to
get out of this, and going to go to colab.research.google.com, so we can start writing some PyTorch code.
I'm going to click new notebook. We're going to start exactly from scratch. I'm going to name this
section two, and let's call it neural network classification with PyTorch. I'm going to put
underscore video, because I'll just show you, you'll see this in the GitHub repo. But for all the
video notebooks, the ones that I write code during these videos that you're watching, the exact code
is going to be saved on the GitHub repo under video notebooks. So there's 00, which is the
fundamentals, and there's the workflow underscore video. But the reference notebook with all the
pretty pictures and stuff is in the main folder here. So PyTorch classification that I pi and b
are actually, maybe we'll just rename it that PyTorch classification. But we know it's with
neural networks. PyTorch classification. Okay, and let's go here. We'll add a nice title. So O2,
neural network classification with PyTorch. And so we'll remind ourselves, classification is a
problem of predicting whether something is one thing or another. And there can be multiple
things as the options, such as email, spam or not spam, photos of dogs or cats or pizza or
sushi or steak. Lots of talk about food. And then I'm just going to link in here, this resource,
because this is the book version of the course. These are what the videos are based off. So book
version of this notebook. And then all the resources are in here. All other resources

```
in the GitHub, and then stuck. Ask a question here, which is under the discussions
tab. We'll
copy that in here. That way we've got everything linked and ready to go. But as
always, what's our
first step in our workflow? This is a little test. See if you remember. Well, it's
data, of course,
because all machine learning problems start with some form of data. We can't write
a machine
learning algorithm to learn patterns and data that doesn't exist. So let's do this
video. We're
going to make some data. Of course, you might start with some of your own that
exists. But for now,
we're going to focus on just the concepts around the workflow. So we're going to
make our own
custom data set. And to do so, I'll write the code first, and then I'll show you
where I get it from.
We're going to import the scikit loan library. One of the beautiful things about
Google Colab
is that it has scikit loan available. You're not sure what scikit loan is. It's a
very popular
machine learning library. PyTorch is mainly focused on deep learning, but scikit
loan is
focused on a lot of things around machine learning. So Google Colab, thank you for
having scikit
loan already installed for us. But we're going to import the make circles data set.
And rather
than talk about what it does, let's see what it does. So make 1000 samples. We're
going to go N
samples equals 1000. And we're going to create circles. You might be wondering why
circles. Well,
we're going to see exactly why circles later on. So X and Y, we're going to use
this variable.
How would you say nomenclature as capital X and Y. Why is that? Because X is
typically a matrix
```

Section 91: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
features and labels. So let's go here. Mate circles. And we're going to make N
samples. So 1000 different
samples. We're going to add some noise in there. Just put a little bit of
randomness. Why not?
You can increase this as you want. I found that 0.03 is fairly good for what we're
doing. And
then we're going to also pass in the random state variable, which is equivalent to
sitting a random
or setting a random seed. So we're flavoring the randomness here. Wonderful. So now
```

let's
have a look at the length of X, which should be what? And length of Y. Oh, we don't have Y
underscore getting a bit trigger happy with this keyboard here. 1000. So we have 1000 samples of
X caught with 1000 or paired with 1000 samples of Y features labels. So let's have a look at the
first five of X. So print first five samples of X. And then we'll put in here X. And we can index
on this five because we're adhering to the data, explorer's motto of visualize visualize visualize
first five samples of Y. And then we're going to go why same thing here.
Wonderful. Let's have a look. Maybe we'll get a new line in here. Just so
looks a bit better. Wonderful. So numerical. Our samples are already numerical. This is one of
the reasons why we're creating our own data set. We'll see later on how we get non numerical data
into numbers. But for now, our data is numerical, which means we can learn it with our model or
we can build a model to learn patterns in here. So this sample has the label of one. And this
sample has the label of one as well. Now, how many features do we have per sample? If I highlight
this line, how many features is this? It would make it a bit easier if there was a comma here,
but we have two features of X, which relates to one label of Y. And so far, we've only seen,
let's have a look at all of Y. We've got zero on one. So we've got two classes. What does this
mean? Zero or one? One thing or another? Well, it looks like binary classification to me,
because we've got only zero or only one. If there was zero, one, two, it would be
multi class classification, because we have more than two things. So let's X out of this.
Let's keep going and do a little bit more data exploration. So how about we make a data frame?
With pandas of circle data. There is truly no real definite way of how to explore data.
For me, I like to visualize it multiple different ways, or even look at random samples. In the case
of large data sets, such as images or text or whatnot. If you have 10 million samples, perhaps
visualizing them one by one is not the best way to do so. So random can help you out there.
So we're going to create a data frame, and we can insert a dictionary here. So I'm going to call
the features in this part of X, X1, and these are going to be X2. So let's say I'll write some code
to index on this. So everything in the zero index will be X1. And everything in the first index,
there we go, will be X2. Let me clean up this code. This should be on different lines,
enter. And then we've got, let's put in the label as Y. So this is just a dictionary here.
So X1 key to X0. X2, a little bit confusing because of zero indexing, but X feature one,
X feature two, and the label is Y. Let's see what this looks like. We'll look at

```
the first 10 samples.
Okay, beautiful. So we've got X1, some numerical value, X2, another numerical
value, correlates
to or matches up with label zero. But then this one, 0442208, and negative that
number matches up
with label zero. So I can't tell what the patterns are just looking at these
numbers. You might be
able to, but I definitely can't. We've got some ones. All these numbers look the
same to me. So
what can we do next? Well, how about we visualize, visualize, visualize, and
instead of just numbers
in a table, let's get graphical this time, visualize, visualize, visualize. So
we're going to bring in
our friendly mapplotlib, import mapplotlib, which is a very powerful plotting
library. I'm just
going to add some cells here. So we've got some space, mapplotlib.pyplot as PLT.
That's right.
We've got this plot.scatter. We're going to do a scatterplot equals X. And we want
the first index.
And then Y is going to be X as well. So that's going to appear on the Y axis. And
then we want to
color it with labels. We're going to see what this looks like in a second. And then
the color map,
C map stands for color map is going to be plot dot color map PLT. And then red,
yellow, blue,
one of my favorite color outputs. So let's see what this looks like. You ready?
Ah, there we go. There's our circles. That's a lot better for me. So what do you
think we're
going to try and do here? If this is our data and we're working on classification,
we're trying to predict if something is one thing or another. So our problem is we
want to
try and separate these two circles. So say given a number here or given two numbers
and X one
and an X two, which are coordinates here, we want to predict the label. Is it going
to be a blue
dot or is it going to be a red dot? So we're working with binary classification. So
we have
```

Section 92: Main Topics

Key Topics:

- This is a common thing that you'll also hear in machine learning
- So if you'd like to learn more about some data sets that you can have a look in here and potentially practice on with neural networks or other forms of machine learning models from scikit-learn, check out this scikit-learn
- I know this is a pie-torch course

▶ 📄 Click to view detailed content

one thing or another. Do we have a blue dot or a red dot? So this is going to be our toy data here.
And a toy problem is, let me just write this down. This is a common thing that you'll also
hear in machine learning. Note, the data we're working with is often referred to as a toy data set,
a data set that is small enough to experiment on, but still sizable enough to practice the
fundamentals. And that's what we're really after in this notebook is to practice the fundamentals
of neural network classification. So we've got a perfect data set to do this. And by the way,
we've got this from scikit-learn. So this little function here made all of these samples for us.
And how could you find out more about this function here? Well, you could go scikit-learn
classification data sets. There are actually a few more in here that we could have done.
I just like the circle one. Toy data sets, we saw that. So this is like a toy box of different
data sets. So if you'd like to learn more about some data sets that you can have a look in here
and potentially practice on with neural networks or other forms of machine learning models from
scikit-learn, check out this scikit-learn. I can't speak highly enough. I know this is a pie-torch
course. We're not focused on this, but they kind of all come together in terms of the machine
learning and deep learning world. You might use something from scikit-learn, like we've done here,
to practice something. And then you might use pie-torch for something else, like what we're
doing here. Now, with that being said, what are the input and output shapes of our problem?
Have a think about that. And also have a think about how we'd split this into training and test.
So give those a go. We covered those concepts in some previous videos,
but we'll do them together in the next video. I'll see you there.
Welcome back. In the last video, we made some classification data so that we can
practice building a neural network in pie-torch to separate the blue dots from the red dots.
So let's keep pushing forward on that. And I'll just clean up here a little bit,
but where are we in our workflow? What have we done so far? Well, we've got our data ready a
little bit. We haven't turned it into tenses. So let's do that in this video, and then we'll
keep pushing through all of these. So in here, I'm going to make this heading 1.1. Check input
and output shapes. The reason we're focused a lot on input and output shapes is why,
because machine learning deals a lot with numerical representations as tenses. And input and output
shapes are some of the most common errors, like if you have a mismatch between your input and
output shapes of a certain layer of an output layer, you're going to run into a lot of errors

```
there. So that's why it's good to get acquainted with whatever data you're using,
what are the
input shapes and what are the output shapes you'd like. So in our case, we can go x
dot shape
and y dot shape. So we're working with NumPy arrays here if we just look at x.
That's what the
make circles function is created for us. We've got an array, but as our workflow
says,
we'd like it in tenses. If we're working with PyTorch, we want our data to be
represented as
PyTorch tenses of that data type. And so we've got a shape here, we've got a
thousand samples,
and x has two features, and y has no features. It's just a single number. It's a
scalar. So it
doesn't have a shape here. So there's a thousand samples of y, thousand samples of
x, two samples
of x equals one y label. Now, if you're working with a larger problem, you might
have a thousand
samples of x, but x is represented by 128 different numbers, or 200 numbers, or as
high as you want,
or just 10 or something like that. So just keep in mind that this number is quite
flexible of how
many features represent a label. Why is the label here? But let's keep going. So
view the first
example of features and labels. So let's make it explicit with what we've just been
discussing.
We'll write some code to do so. We'll get the first sample of x, which is the zero
index,
and we'll get the first sample of y, which is also the zero index. We could get
really anyone
because they're all of the same shape. But print values for one sample of x. What
does this equal?
X sample, and the same for y, which is y sample. And then we want to go print f
string for one
sample of x. We'll get the shape here. X sample dot shape, and the same for y, and
then we'll get
y sample dot shape. Beautiful. What's this going to do? Well, we've got one sample
of x. So this
sample here of these numbers, we've got a lot going on here. 75424625 and 0231
48074. I mean,
you can try to find some patterns in those. If you do, all the best here, and the
same for y. So this
is, we have the y sample, this correlates to a number one, a label of one. And then
we have
shapes for one sample of x, which is two. So we have two features for y. It's a
little bit confusing
here because y is a scalar, which doesn't actually have a shape. It's just one
value. So for me,
in terms of speaking this, teaching it out loud, we'll be two features of x trying
to predict
```

Section 93: Main Topics

Key Topics:

- And for this, we need to import torch, get pytorch and we'll check the torch version
- Just make sure we can import pytorch
- However, pytorch, the default type is float 32

▶ 📄 Click to view detailed content

```
one number for y. And so let's now create another heading, which is 1.2. Let's get
our data into
tenses, turn data into tenses. We have to convert them from NumPy. And we also want
to create
train and test splits. Now, even though we're working with a toy data set here, the
principle
of turning data into tenses and creating train and test splits will stay around for
almost any
data set that you're working with. So let's see how we can do that. So we want to
turn data
into tenses. And for this, we need to import torch, get pytorch and we'll check the
torch version.
It has to be at least 1.10. And I might just put this down in the next cell. Just
make sure we can
import pytorch. There we go, 1.10 plus CUDA 111. If your version is higher than
that, that is okay.
The code below should still work. And if it doesn't, let me know. So x equals torch
dot
from NumPy. Why are we doing this? Well, it's because x is a NumPy array. And if we
go x dot,
does it have a d type attribute float 64? Can we just go type or maybe type? Oh,
there we go.
NumPy and DRA. We can just go type x. NumPy and DRA. So we want it in a torch
tensor. So we're
going to go from NumPy. We saw this in the fundamental section. And then we're
going to change it into
type torch dot float. A float is an alias for float 32. We could type the same
thing. These two are
equivalent. I just going to type torch float for writing less code. And then we're
going to go
the same with why torch from NumPy. Now, why do we turn it into a torch float?
Well, that's
because if you recall, the default type of NumPy arrays is, if we go might just put
out this in
a comma x dot D type is float 64. There we go. However, pytorch, the default type
is float 32.
So we're changing it into pytorch's default type. Otherwise, if we didn't have this
little
section of code here dot type torch dot float, our tensors would be of float 64 as
well. And that
may cause errors later on. So we're just going for the default data type within
pytorch. And so
now let's have a look at the first five values of x and the first five values of y.
What do we
have? Beautiful. We have tensor data types here. And now if we check the data type
of x and we
```

```
check the data type of y, what do we have? And then one more, we'll just go type x.
So we have
our data into tensors. Wonderful. But now so it's torch dot tensor. Beautiful. But
now we would like
training and test sets. So let's go split data into training and test sets. And a
very, very popular
way to split data is a random split. So before I issued the challenge of how you
would split this
into a training and test set. So because these data points are kind of scattered
all over the
place, we could split them randomly. So let's see what that looks like. To do so,
I'm going to
use our faithful scikit learn again. Remember how I said scikit learn has a lot of
beautiful methods
and functions for a whole bunch of different machine learning purposes. Well, one
of them is
for a train test split. Oh my goodness, pytorch I didn't want auto correct there.
Train test split.
Now you might be able to guess what this does. These videos are going to be a
battle between me and
code labs auto correct. Sometimes it's good. Other times it's not. So we're going
to set this code
up. I'm going to write it or we're going to write it together. So we've got x train
for our training
features and X tests for our testing features. And then we also want our training
labels and
our testing labels. That order is the order that train test split works in. And
then we have train
test split. Now if we wrote this function and we wanted to find out more, I can
press command
ship space, which is what I just did to have this. But truly, I don't have a great
time reading all
of this. You might. But for me, I just like going train test split. And possibly
one of the first
functions that appears, yes, is scikit learn. How good is that? So scikit learn dot
model selection
dot train test split. Now split arrays or matrices into random train and test
subsets. Beautiful.
We've got a code example of what's going on here. You can read what the different
parameters do.
But we're going to see them in action. This is just another example of where
machine learning
libraries such as scikit learn, we've used matplotlib, we've used pandas, they all
interact
together to serve a great purpose. But now let's pass in our features and our
labels.
This is the order that they come in, by the way. Oh, and we have the returns
splitting. So the
order here, I've got the order goes x train x test y train y test took me a little
while to
remember this order. But once you've created enough training test splits with this
function,
you kind of know this off by heart. So just remember features first train first and
then labels.
And we jump back in here. So I'm going to put in the test size parameter of 0.2.
This is percentage wise. So let me just write here 0.2 equals 20% of data will be
test.
```

```
And 80% will be train. If we wanted to do a 50 50 split, that kind of split doesn't
usually
```

Section 94: Main Topics

Key Topics:

- However, because we are using scikit learn, setting torch dot manual seed will only affect pytorch code rather than scikit learn code
- So in the next video, we've now got training and test sets, we've started to move through our beautiful pytorch workflow here
- In fact, all models in PyTorch subclass and end up module

▶ 📄 Click to view detailed content

```
happen, but you could go 0.5. But the test size says, hey, how big and percentage
wise do you want
your test data to be? And so behind the scenes train test split will calculate
what's 20% of
our x and y samples. So we'll see how many there is in a second. But let's also put
a random state
in here. Because if you recall back in the documentation, train test split splits
data
randomly into random train and test subsets. And random state, what does that do
for us? Well,
this is a random seed equivalent of very similar to torch dot manual seed. However,
because we are
using scikit learn, setting torch dot manual seed will only affect pytorch code
rather than
scikit learn code. So we do this so that we get similar random splits. As in, I get
a similar
random split to what your random split is. And in fact, they should be exactly the
same. So let's
run this. And then we'll check the length of x train. And length of x test. So if
we have 1000
total samples, and I know that because above in our make circles function, we said
we want
1000 samples, that could be 10,000, that could be 100. That's the beauty of
creating your own
data set. And we have length y train. If we have 20% testing values, how many
samples are going
to be dedicated to the test sample, 20% of 1000 years, 200, and 80%, which is
because training is
going to be training here. So 100 minus 20% is 80%. So 80% of 1000 years, let's
find out.
Run all beautiful. So we have 800 training samples, 200 testing samples. This is
going to be the
data set that we're going to be working with. So in the next video, we've now got
```

training and
test sets, we've started to move through our beautiful pytorch workflow here. We've got our
data ready, we've turned it into tenses, we've created a training and test split. Now it's time
to build or pick a model. So I think we're still in the building phase. Let's do that in the next
video. Welcome back. In the last video, we split our data into training and test sets. And because
we did 80 20 split, we've got about 800 samples to train on, and 200 samples to test on. Remember,
the training set is so that the model can learn patterns, patterns that represent this data set
here, the circles data set, red dots or blue dots. And the test data set is so that we can
evaluate those patterns. And I took a little break before, but you can tell that because my
notebook is disconnected. But if I wanted to reconnect it, what could I do? We can go here,
run time, run before that's going to run all of the cells before. It shouldn't take too long
because we haven't done any large computations. But this is good timing because we're up to part
two, building a model. And so there's a fair few steps here, but nothing that we haven't covered
before, we're going to break it down. So let's build a model to classify our blue and red dots.
And to do so, we want to tenses. I want to not tenses. That's all right. So let me just make
some space here. There we go. So number one, let's set up device agnostic code. So we get in the
habit of creating that. So our code will run on an accelerator. I can't even spell accelerator.
It doesn't matter. You know what I mean? GPU. If there is one. Two. What should we do next?
Well, we should construct a model. Because if we want to build a model, we need a model.
Construct a model. And we're going to go by subclassing and then dot module.
Now we saw this in the previous section, we subclassed and then module. In fact, all models
in PyTorch subclass and end up module. And let's go define loss function and optimizer.
And finally, good collabs auto correct is not ideal. And then we'll create a training
and test loop. Though this will probably be in the next section. We'll focus on building a model
here. And of course, all of these steps are in line with what they're in line with this.
So we don't have device agnostic code here, but we're just going to do it enough so that we have
a habit. These are the main steps. Pick or build a pre-trained model, suit your problem, pick a
loss function and optimizer, build a training loop. So let's have a look. How can we start this off?
So we will import PyTorch. And and then we've already done this, but we're going to do it anyway
for completeness, just in case you wanted to run your code from here, import and

```
then. And we're
going to make device agnostic code. So we'll set the device equal to CUDA if torch
dot CUDA
is available else CPU, which will be the default. The CPU is the default. If
there's no GPU,
which means that CUDA is available, all of our PyTorch code will default to using
the CPU
device. Now we haven't set up a GPU yet so far. You may have, but as you see, my
target device is
currently CPU. How about we set up a GPU? We can go into here runtime change
runtime type
GPU. And I'm going to click save. Now this is going to restart the runtime and
reconnect.
So once it reconnects beautiful, we could actually just run this code cell here.
This is going to set up the GPU device, but because we're only running this cell,
if we were to just
set up X train, we've not been defined. So because we restarted our runtime, let's
run all or we can
```

Section 95: Main Topics

Key Topics:

- So almost all models in pytorch, subclass, and then got module because there's some great things that it does for us behind the scenes

▶ 📄 Click to view detailed content

```
just run before. So this is going to rerun all of these cells here. And do we have
X train now?
Let's have a look. Wonderful. Yes, we do. Okay, beautiful. So we've got device
agnostic code.
In the next video, let's get on to constructing a model. I'll see you there.
Welcome back. In the last video, we set up some device agnostic code. So this is
going to come in
later on when we send our model to the target device, and also our data to the
target device.
This is an important step because that way, if someone else was able to run your
code or you
were to run your code in the future, because we've set it up to be device agnostic,
quite a fault it will run on the CPU. But if there's an accelerator present,
well, that means that it might go faster because it's using a GPU rather than just
using a CPU.
So we're up to step two here construct a model by subclassing and in module. I
think we're going to
write a little bit of text here just to plan out the steps that we're doing. Now
we've
set up device agnostic code. Let's create a model that we're going to break it
down. We've got some
```

sub steps up here. We're going to break it down even this one down into some sub-sub steps. So number
one is we're going to subclass and then got module. And a reminder here, I want to make some space,
just so we're coding in about the middle of the page. So almost all models in pytorch,
subclass, and then got module because there's some great things that it does for us behind the
scenes. And step two is we're going to create two and then dot linear layers. And we want these
that are capable to handle our data. So that are capable of handling the shapes of our data.
Step three, we want to define or defines a forward method. Why do we want to define a forward method?
Well, because we're subclassing an end dot module, right? And so the forward method defines a forward
method that outlines the forward pass or forward computation of the model. And number four, we want
to instantiate, well, this doesn't really have to be the part of creating it, but we're going to do
anyway, and instantiate an instance of our model class and send it to the target device. So I'm
going to be a couple of little different steps here, but nothing too dramatic that we haven't really
covered before. So let's go number one, construct a model that subclasses an end dot module.
So I'm going to code this all out. Well, we're going to code this all out together. And then we'll
go back through and discuss it, and then maybe draw a few pictures or something to check out
what's actually happening. So circle model V one, because we're going to try and split some circles,
red and blue circles. This is our data up here. This is why it's called circle model, because we're
trying to separate the blue and red circle using a neural network. So we've subclassed an end dot
module. And when we create a class in Python, we'll create a constructor here, a net, and then
put in super dot underscore a net. And then inside the constructor, we're going to create our
layers. So this is number two, create two and then linear layers, capable of handling the shapes
of our data. So I'm going to write this down here to create two, two, and then dot linear layers, capable
of handling the shapes of our data. And so if we have a look at X train, what are the shapes here?
What's the input shape? Because X train is our features, right? Now features are going to go
into our model. So we have 800 training samples. This is the first number here of size two each.
So 800 of these and inside each is two numbers. Again, depending on the data set you're working
with, your features may be 100 in length, a vector of 100, or maybe a different size tensor all
together, or there may be millions. It really depends on what data set you're working with.
Because we're working with a simple data set, we're going to focus on that. But the

```
principal
is still the same. You need to define a neural network layer that is capable of
handling your
input features. So we're going to make layer one equals an n dot linear. And then
if we wanted
to find out what's going on an n n dot linear, we could run shift command space on
my computer,
because it's a Mac, maybe shift control space if you're on Windows. So we're going
to define the
n features. What would n features be here? Well, we just decided that our X has two
features.
So n features are going to be two. And now what is the out features? This one is a
little bit tricky.
So in our case, we could have out features equal to one if we wanted to just pass a
single linear
layer, but we want to create two linear layers here. So why would out features be
one? Well,
that's because if we have a look at the first sample of Y train, we would want us
to input,
or maybe we'll look at the first five. We want to map one sample of X to one sample
of Y and Y
has a shape of one. Oh, well, really, it's nothing because it's a scalar, but we
would still put
one here so that it outputs just one number. But we're going to change this up.
We're going to put
```

Section 96: Main Topics

Key Topics:

- I don't know enough about computer hardware to know exactly why that's the case, but that's just a rule of thumb in machine learning
- But truly, this is what the majority of building machine learning models in PyTorch is going to look like

▶ 📄 Click to view detailed content

```
it into five and we're going to create a second layer. Now, this is an important
point of joining
together neural networks in features here. What do you think the in features of our
second layer is
going to be? If we've produced an out feature of five here, now this number is
arbitrary. We could
do 128. We could do 256. Generally, it's multiples of 8, 64. We're just doing five
now because we're
keeping it nice and simple. We could do eight multiples of eight is because of the
efficiency
of computing. I don't know enough about computer hardware to know exactly why
```

that's the case,
but that's just a rule of thumb in machine learning. So the in features here has to match up with the
out features of a previous layer. Otherwise, we'll get shape mismatch errors. And so let's go here
out features. So we're going to treat this as the output layer. So this is the out features equals
one. So takes in two features and upscales to five features. So five numbers. So what this does,
what this layer is going to do is take in these two numbers of X, perform an end up linear. Let's
have a look at what equation it does. An end up linear is going to perform this function here
on the inputs. And it's going to upscale it to five features. Now, why would we do that? Well,
the rule of thumb here, because this is denoted as a hidden unit, or how many hidden neurons there
are. The rule of thumb is that the more hidden features there are, the more opportunity our model
has to learn patterns in the data. So to begin with, it only has two numbers to learn patterns on,
but at when we upscale it to five, it has five numbers to learn patterns on. Now, you might think,
why don't we just go straight to like 10,000 or something? But there is like an upper limit here
to sort of where the benefits start to trail off. We're just using five because it keeps it nice
and simple. And then the in features of the next layer is five, so that these two line up. We're
going to map this out visually in a moment, but let's keep coding. We've got in features two for
X. And now this is the output layer. So takes in five features from previous layer and outputs
a single feature. And now this is same shape. Same shape as why. So what is our next step? We
want to define a Ford method, a Ford computation of Ford pass. So the Ford method is going to
define the Ford computation. And as an input, it's going to take X, which is some form of data.
And now here's where we can use layer one and layer two. So now let's just go return.
Or we'll put a note here of what we're doing. Three, we're going to go define a Ford method
that outlines the Ford pass. So Ford, and we're going to return. And here's some notation we
haven't quite seen yet. And then we're going to go self layer two. And inside the brackets we'll
have self layer one inside those brackets. We're going to have X. So the way this goes is X goes
into layer one. And then the output of layer one goes into layer two. So whatever data we have,
so our training data, X train goes into layer one performs the linear calculation here. And then
it goes into layer two. And then layer two is going to output, go to the output. So X is the input,
layer one computation layer two output. So we've done that. Now let's do step four, which is

```
instantiate an instance of our model class. And send it to the target device. So
this is our model
class, circle model V zero. We're just going to create a model because it's the
first model we've
created up this section. Let's call it model zero. And we're going to go circle
model V one. And then
we're going to go to two. And we're going to pass in device, because that's our
target device.
Let's now have a look at model zero. And then Oh, typo. Yeah, classic.
What did we get wrong here? Oh, did we not pass in self self? Oh, there we go.
Little typo classic. But the beautiful thing about creating a class here is that we
could put
this into a Python file, such as model dot pi. And then we wouldn't necessarily
have to rewrite
this all the time, we could just call it. And so let's just check what the vice
it's on.
So target device is CUDA, because we've got a GPU, thank you, Google Colab. And
then if we wanted
to, let's go next model zero dot parameters, we'll call the parameters, and then
we'll go device.
CUDA beautiful. So that means our models parameters are on the CUDA device. Now
we've covered enough
code in here for this video. So if you want to understand it a little bit more, go
back through
it. But we're going to come back in the next video and make it a little bit more
visual. So I'll see
you there. Welcome back. In the last video, we did something very, very exciting.
We created our
first multi layer neural network. But right now, this is just code on a page. But
truly, this is
what the majority of building machine learning models in PyTorch is going to look
like. You're
going to create some layers, or a simple or as complex as you like. And then you're
going to
use those layers in some form of Ford computation to create the forward pass. So
let's make this a
```

Section 97: Main Topics

Key Topics:

- If we go over to the TensorFlow playground, and now TensorFlow is another deep learning framework similar to PyTorch, it just allows you to write code such as this, to build neural networks, fit them to some sort of data to find patterns and data, and then use those machine learning models in your applications
- And maybe we put the learning rate, we've seen the learning rate to 0

▶ 📄 Click to view detailed content

```
little bit more visual. If we go over to the TensorFlow playground, and now
TensorFlow is another
deep learning framework similar to PyTorch, it just allows you to write code such
as this,
to build neural networks, fit them to some sort of data to find patterns and data,
and then use those machine learning models in your applications. But let's create
this. Oh,
by the way, this is playground.tensorFlow.org. This is a neural network that we can
train in
the browser if we really wanted to. So that's pretty darn cool. But we've got a
data set here,
which is kind of similar to the data set that we're working with. We have a look at
our circles one.
Let's just say it's close enough. It's circular. That's what we're after. But if we
increase this,
we've got five neurons now. We've got two features here, X1 and X2. Where is this
reminding you of what's happening? There's a lot of things going on here that we
haven't covered
yet, but don't worry too much. We're just focused on this neural network here. So
we've got some
features as the input. We've got five hidden units. This is exactly what's going on
with the model
that we just built. We pass in X1 and X2, our values. So if we go back to our data
set,
these are X1 and X2. We pass those in. So we've got two input features. And then we
pass them to a
hidden layer, a single hidden layer, with five neurons. What have we just built? If
we come down
into here to our model, we've got in features two, out features five. And then that
feeds into
another layer, which has in features five and out features one. So this is the
exact same model
that we've built here. Now, if we just turn this back to linear activation, because
we're sticking
with linear for now, we'll have a look at different forms of activation functions
later on. And maybe
we put the learning rate, we've seen the learning rate to 0.01. We've got epochs
here, got classification.
And we're going to try and fit this neural network to this data. Let's see what
happens.
Oh, the test loss, it's sitting about halfway 0.5. So about 50% loss. So if we only
have two
classes and we've got a loss of 50%, what does that mean? Well, the perfect loss
was zero.
And the worst loss was one. Then we just divide one by two and get 50%. But we've
only got two
classes. So that means if our model was just randomly guessing, it would get a loss
of about 0.5,
because you could just randomly guess whatever data point belongs to blue or orange
in this case.
So in a binary classification problem, if you have the same number of samples in
each class,
in this case, blue dots and orange dots, randomly guessing will get you about 50%.
Just like tossing
a coin, toss a coin 100 times and you get about 50 50 might be a little bit
different, but it's
```

```
around about that over the long term. So we've just fit for 3000 epochs. And we're
still not getting
any better loss. Hmm. I wonder if that's going to be the case for our neural
network. And so to
draw this in a different way, I'm going to come to a little tool called fig jam,
which is just a
whiteboard that we can put shapes on and it's based on the browser. So this is
going to be nothing
fancy. It's going to be a simple diagram. Say this is our input. And I'm going to
make this green
because my favorite color is green. And then we're going to have, let's make some
different
colored dots. I want a blue dot here. So this can be dot one, and dot two, I'll put
another dot
here. I'll zoom out a little so we have a bit more space. Well, maybe that was too
much. 50%
looks all right. So let me just move this around, move these up a little. So we're
building a neural
network here. This is exactly what we just built. And so we'll go here. Well, maybe
we'll put this
as input X one. So this will make a little bit more sense. And then we'll maybe we
can copy this.
Now this is X two. And then we have some form of output. Let's make this one. And
we're going to
color this orange. So output. Right. So you can imagine how we got connected dots
here.
They will connect these. So our inputs are going to go through all of these. I
wonder if I can
draw here. Okay, this is going to be a little bit more complex, but that's all
right. So this
is what we've done. We've got two input features here. And if we wanted to keep
drawing these,
we could all of these input features are going to go through all of these hidden
units that we
have. I just drew the same arrow twice. That's okay. But this is what's happening
in the forward
computation method. It can be a little bit confusing for when we coded it out. Why
is that? Well,
from here, it looks like we've only got an input layer into a single hidden layer
in the blue.
And an output layer. But truly, this is the same exact shape. You get the point.
And then all of
these go to the output. But we're going to see this computationally later on. So
whatever data set
you're working with, you're going to have to manufacture some form of input layer.
Now this
may be you might have 10 of these if you have 10 features. Or four of them if you
have four
```

Section 98: Main Topics

Key Topics:

- So this is, of course, going to override our previous model zero

▶ 📄 Click to view detailed content

```
features. And then if you wanted to adjust these, well, you could increase the
number of hidden
units or the number of out features of a layer. What just has to match up is that
the layer it's
going into has to have a similar shape as the what's coming out of here. So just
keep that in mind
as you're going on. And in our case, we only have one output. So we have the output
here,
which is why. So this is a visual version. We've got the TensorFlow playground. You
could play
around with that. You can change this to increase. Maybe you want five hidden
layers with five neurons
in each. This is a fun way to explore. This is a challenge, actually, go to
playground.tensorflow.org,
replicate this network and see if it fits on this type of data. What do you think,
will it?
Well, we're going to have to find out in the next few videos. So I'm going to show
you in the
next video another way to create the network that we just created. This one here
with even less
code than what we've done before. I'll see you there. Welcome back. In the last
video, what we
discussed, well, actually, in the previous video to last, we coded up this neural
network here,
circle model V zero. By subclassing an end or module, we created two linear layers,
which are
capable of handling the shape of our data in features two because why we have two X
features.
Out features were upscaling the two features to five so that it gives our network
more of a
chance to learn. And then because we've upscaled it to five features, the next
subsequent layer
has to be able to handle five features as input. And then we have one output
feature because that's
the same shape as our Y here. Then we got a little bit visual by using the
TensorFlow
playground. Did you try out that challenge, make five in layers with five neurons?
Did it work?
And then we also got a little bit visual in Figma as well. This is just another way
of
visualizing different things. You might have to do this a fair few times when you
first start
with neural networks. But once you get a bit of practice, you can start to infer
what's going on
through just pure code. So now let's keep pushing forward. How about we replicate
this
with a simpler way? Because our network is quite simple, that means it only has two
layers.
That means we can use. Let's replicate the model above using nn.sequential. And I'm
going to code
this out. And then we can look up what nn.sequential is. But I think you'll be able
```

to comprehend what's
happening by just looking at it. So nn, which is torch.nn. We can do torch.nn, but we've already
imported nn. We're going to call nn.sequential. And then we're going to go nn.linear. And what
was the in features of our nn.linear? Well, it was two because we have two in features. And then
we're going to replicate the same out features. Remember, we could customize this to whatever we
want 10, 100, 128. I'm going to keep it at five, nice and simple. And then we go nn.linear. And
the in features of this next layer is going to be five because the out features of the previous
layer was five. And then finally, the out features here is going to be one because we want one y
value to our two x features. And then I'm going to send that to the device. And then I'm going to
have a look at model zero. So this is, of course, going to override our previous model zero. But
have a look. The only thing different is that this is from the circle model V zero class. We
subclassed an n dot module. And the only difference is the name here. This is just from sequential.
And so can you see what's going on here? So as you might have guessed sequential,
it implements most of this code for us behind the scenes. Because we've told it that it's going
to be sequential, it's just going to go, hey, step the code through this layer, and then step
the code through this layer. And outputs basically the same model, rather than us creating our own
forward method, you might be thinking, Daniel, why don't you show us this earlier? That looks
like such an easy way to create a neural network compared to this. Well, yes, you're 100% right.
That is an easier way to create a neural network. However, the benefit of subclassing, and that's
why I started from here, is that when you have more complex operations, such as things you'd
like to construct in here, and a more complex forward pass, it's important to know how to
build your own subclasses of nn dot module. But for simple straightforward stepping through
each layer one by one, so this layer first, and then this layer, you can use nn dot sequential.
In fact, we could move this code up into here. So we could do this self dot, we'll call this
two linear layers equals nn dot sequential. And we could have layer one, we could go self,
self dot layer one. And or actually, we'll just recode it, we'll go nn dot linear. So it's so
it's the same code is what we've got below in features. If I could type that'll be great,
n features equals two, out features equals five. And then we go nn dot linear. And then we go
n features equals what equals five, because it has to line up out features equals one.

```
And then we've got two linear layers. And then if we wanted to get rid of this,
return
```

Section 99: Main Topics

Key Topics:

- But this is the flexibility of PyTorch
- So of course, if we have a look at our model zero dot state deck, oh, this will be a good experiment
- So this is for the first of the zeroth layer, these two here with the zero dot, and then the one dot weight is four, of course, the first index layer

▶ 📄 Click to view detailed content

```
to linear layers, and we'll pass it X remake it. There we go. Well, because we've
created these as
well, let's get rid of that. Beautiful. So that's the exact same model, but just
using nn dot
sequential. Now I'm going to get rid of this so that our code is not too verbose.
That means a lot
going on. But this is the flexibility of PyTorch. So just keep in mind that there's
a fair few ways
to make a model. The simplest is probably sequential. And then subclassing is this
is a little bit
more complicated than what we've got. But this can extend to handle lot more
complex neural networks,
which you'll likely have to be building later on. So let's keep pushing forward.
Let's see what
happens if we pass some data through here. So we'll just rerun this cell to make
sure we've got
our model zero instantiated. We'll make some predictions with the model. So of
course, if we
have a look at our model zero dot state deck, oh, this will be a good experiment.
So look at this.
So we have weight, a weight tensor, a bias tensor, a weight tensor, and a bias
tensor. So this is
for the first of the zeroth layer, these two here with the zero dot, and then the
one dot weight is
four, of course, the first index layer. Now have a look inside here. Now you see
how out features
is five. Well, that's why our bias parameter has five values here. And the same
thing for this weight
value here. And the weight value here, why is this have 10 samples? One, two,
three, four, five, six,
seven, eight, nine, 10, because two times five equals 10. So this is just with a
simple two layer
network, look at all the numbers that are going on behind the scenes. Imagine
```

coding all of these
by hand. Like there's something like 20 numbers or something here. Now we've only done two layers
here. Now the beauty of this is that in the previous section, we created all of the weight
and biases using an end dot parameter and random values. You'll notice that these are all random
two. Again, if yours are different to mine, don't worry too much, because they're going to be started
randomly and we haven't set a random seed. But the thing to note here is that PyTorch is creating
all of these parameters for us behind the scenes. And now when we do back propagation and gradient
descent, when we code our training loop, the optimizer is going to change all of these values
ever so slightly to try and better fit or better represent the data so that we can split our two
circles. And so you can imagine how verbose this could get if we had say 50 layers with 128 different
features of each. So let's change this up, see what happens. Watch how quickly the numbers get
out of hand. Look at that. We just changed one value and look how many parameters our model has.
So you might be able to calculate that by hand, but I personally don't want to. So we're going to
let PyTorch take care of a lot of that for us behind the scenes. So for now we're keeping it simple,
but that's how we can crack our models open and have a look at what's going on. Now that was a
little detour. It's time to make some predictions with random numbers. I just wanted to highlight
the fact that our model is in fact instantiated with random numbers here. So the untrained threads
model zero, we're going to pass in X test. And of course, we have to send the test data to the
device. Otherwise, if it's on a different device, we'll get errors because PyTorch likes to make
calculations on the same device. So we'll go print. Let's do a nice print statement of length of
predictions. We're going to go length or then untrained threads, we'll pass that in there.
And then we'll go, oh no, we need to squiggle. And then we'll go shape. Shape is going to be
untrained spreads dot shape. So this is again, following the data explorer's motto of visualize,
visualize, visualize. And sometimes print is one of the best ones to do so. So length of test samples,
you might already know this, or we've already checked this together, haven't we? X test.
And then we're going to get the shape, which is going to be X test dot shape. Wonderful. And then
we're going to print. What's our little error here? Oh no, collabs tricking me. So let's go first
10 predictions. And we're going to go untrained threads. So how do you think these predictions will
fare? They're doing it with random numbers. And what are we trying to predict again? Well,

```
we're trying to predict whether a dot is a red dot or a blue dot or zero or one.
And then we'll go
first 10 labels is going to be, we'll get this on the next line. And we'll go Y
test.
Beautiful. So let's have a look at this untrained predictions. So we have length of
predictions
is 200. Length of test samples is 200. But the shapes are different. What's going
on here?
Y test. And let's have a look at X test. Oh, well, I better just have a look at Y
test.
Why don't we have a two there? Oh, I've done X test dot shape. Oh, let's test
samples. That's
okay. And then the predictions are one. Oh, yes. So Y test. Let's just check the
first 10 X test.
So a little bit of clarification needed here with your shapes. So maybe we'll get
this over here
because I like to do features first and then labels. What did we miss here? Oh, X
test 10
```

Section 100: Main Topics

Key Topics:

- And n dot sequential is a simpler way of creating a pytorch model
- And then inside our model, pytorch has behind the scenes created us some weight and bias tensors for each of our layers with regards to the shapes that we've set
- And so the handy thing about this is that if we got quite ridiculous with our layers, pytorch would still do the same thing behind the scenes, create a whole bunch of random numbers for us

▶ 📄 Click to view detailed content

```
and Y test. See, we're troubleshooting on the fly here. This is what you're going
to do with
a lot of your code. So there's our test values. There's the ideal labels. But our
predictions,
they don't look like our labels. What's going on here? We can see that they're on
the CUDA device,
which is good. We said that. We can see that they got gradient tracking. Oh, we
didn't with
touch. We didn't do inference mode here. That's a poor habit of us. Excuse me.
Let's inference
mode this. There we go. So you notice that the gradient tracking goes away there.
And so our
predictions are nowhere near what our test labels are. But also, they're not even
in the same like
```

ball park. Like these are whole numbers, one or zero. And these are all floats between one and
zero. Hmm. So maybe rounding them. Will that do something? So where's our threads here? So
we go torch dot round. What happens there? Oh, they're all zero. Well, our model is probably
going to get about 50% accuracy. Why is that? Because all the predictions look like they're
going to be zero. And they've only got two options, basically head or tails. So when we create our
model and when we evaluate it, we want our predictions to be in the same format as our labels. But
we're going to cover some steps that we can take to do that in a second. What's important to take
away from this is that there's another way to replicate the model we've made above using
nn dot sequential. We've just replicated the same model as what we've got here. And n dot
sequential is a simpler way of creating a pytorch model. But it's limited because it literally
just sequentially steps through each layer in order. Whereas in here, you can get as creative as you
want with the forward computation. And then inside our model, pytorch has behind the scenes
created us some weight and bias tensors for each of our layers with regards to the shapes that
we've set. And so the handy thing about this is that if we got quite ridiculous with our layers,
pytorch would still do the same thing behind the scenes, create a whole bunch of random numbers for
us. And because our numbers are random, it looks like our model isn't making very good predictions.
But we're going to fix this in the next few videos when we move on to
fitting the model to the data and making a prediction. But before we do that, we need to
pick up a loss function and an optimizer and build a training loop. So let's get on to these two things.
Welcome back. So over the past few videos, we've been setting up a classification model to deal
with our specific shape of data. Now recall, depending on the data set that you're working
with will depend on what layers you use for now we're keeping it simple and n dot linear is one
of the most simple layers in pytorch. We've got two input features, we're upscaling that to five
output features. So we have five hidden units, and then we have one output feature. And that's in line
with the shape of our data. So two features of x equals one number for y. So now let's continue
on modeling with where we're up to. We have build or pick a model. So we've built a model. Now we
need to pick a loss function and optimizer. We're getting good at this. So let's go here,
set up loss function and optimizer. Now here comes the question. If we're working on classification
previously, we used, let's go to the next one, and an dot L one loss for regression, which is

```
MAE mean absolute error, just a heads up that won't necessarily work with a
classification problem.
So which loss function or optimizer should you use? So again, this is problem
specific. But with
a little bit of practice, you'll get used to using different ones. So for example,
for regression,
you might want, which is regressions predicting a number. And I know it can get
fusing because
it looks like we're predicting a number here, we are essentially predicting a
number. But this
relates to a class. So for regression, you might want MAE or MSE, which is mean
absolute
absolute error, or mean squared error. And for classification, you might want
binary cross entropy
or categorical cross entropy, which is sometimes just referred to as cross entropy.
Now, where would
you find these things out? Well, through the internet, of course. So you could go,
what is
binary cross entropy? I'm going to leave you this for your extra curriculum to read
through this.
We've got a fair few resources here. Understanding binary cross entropy slash log
loss
by Daniel Godoy. Oh, yes. Great first name, my friend. This is actually the article
that I
would recommend to if you want to learn what's going on behind the scenes through
binary cross
entropy. For now, there's a lot of math there. We're going to be writing code to
implement this. So
PyTorch has done this for us. Essentially, what does a loss function do? Let's
remind ourselves.
Go down here. As a reminder, the loss function measures how wrong your models'
predictions are.
So I also going to leave a reference here to I've got a little table here in the
book version of
this course. So 0.2 neural network classification with PyTorch set up loss function
and optimizer.
So we've got some example loss functions and optimizers here, such as stochastic
gradient
descent or SGD optimizer, atom optimizer is also very popular. So I've got problem
type here,
and then the PyTorch code that we can implement this with. We've got binary cross
entropy loss.
```

Section 101: Main Topics

Key Topics:

- Then trust me, when I first started using PyTorch, I got a little bit confused about why they have two here, but we're going to explore that anyway
- Luckily PyTorch does this for us, but logit is kind of confusing in deep learning

- So if we go what is a logit in deep learning, it kind of means a different thing

▶ 📄 Click to view detailed content

```
We've got cross entropy loss, mean absolute error, MAE, mean squared error, MSE. So
you want to use
these two for regression. There are other different loss functions you could use,
but these are some
of the most common. That's what I'm focusing on, the most common ones that are
going to get you
through a fair few problems. We've got binary classification, multi-class
classification. What
are we working with? We're working with binary classification. So we're going to
look at torch.nn
BCE, which stands for binary cross entropy, loss with logits. What the hell is a
logit?
And BCE loss. Now this is confusing. Then trust me, when I first started using
PyTorch,
I got a little bit confused about why they have two here, but we're going to
explore that anyway.
So what is a logit? So if you search what is a logit, you'll get this and you'll
get statistics
and you'll get the log odds formula. In fact, if you want to read more, I would
highly encourage it.
So you could go through all of this. We're going to practice writing code for it
instead.
Luckily PyTorch does this for us, but logit is kind of confusing in deep learning.
So if we go
what is a logit in deep learning, it kind of means a different thing. It's kind of
just a name of what
yeah, there we go. What is the word logits in TensorFlow? As I said, TensorFlow is
another
deep learning framework. So let's close this. What do we got? We've got a whole
bunch of
definitions here. Logits layer. Yeah. This is one of my favorites. In context of
deep learning,
the logits layer means the layer that feeds into the softmax. So softmax is a form
of activation.
We're going to see all of this later on because this is just words on a page right
now. Softmax
or other such normalization. So the output of the softmax are the probabilities for
the
classification task and its input is the logit's layer. Whoa, there's a lot going
on here. So let's
just take a step back and get into writing some code. And for optimizers, I'm just
going to complete
this. And for optimizers, two of the most common and useful are SGD and Adam.
However, PyTorch
has many built in options. And as you start to learn more about the world of
machine learning,
you'll find that if you go to torch.optim or torch.nn. So if we have.nn, what do we
have in here?
Loss functions. There we go. Beautiful. That's what we're after. L1 loss, which is
MAE,
MSC loss, cross entropy loss, CTC loss, all of these different types of loss here
```

will depend
on the problem you're working on. But I'm here to tell you that for regression and classification,
two of the most common of these. See, this is that confusion again. BCE loss, BCE with
logit's loss. What the hell is a logit? My goodness. Okay, that's enough. And Optim,
these are different optimizers. We've got probably a dozen or so here. Algorithms.
Add a delta, add a grad. Adam, this can be pretty full on when you first get started. But for now,
just stick with SGD and the atom optimizer. They're two of the most common. Again, they may not
perform the best on every single problem, but they will get you fairly far just knowing those.
And then you'll pick up some of these extra ones as you go. But let's just get rid of all of,
maybe we'll, so I'll put this in here, this link. So we'll create our loss function. For the loss
function, we're going to use torch.nn.bce with logit's loss. This is exciting. For more on what
binary cross entropy, which is BCE, a lot of abbreviations in machine learning and deep learning
is check out this article. And then for a definition on what a logit is, we're going to see a
logit in a second in deep learning. Because again, deep learning is one of those fields,
a machine learning, which likes to be a bit rebellious, you know, likes to be a bit different
from the pure mathematics type of fields and statistics in general. It's this beautiful
gestaltism and for different optimizers, see torch dot opt in. But we've covered a few of these
things before. And finally, I'm going to put up here, and then for some common choices of loss
functions and optimizers. Now, don't worry too much. This is why I'm linking all of these extra
resources. A lot of this is covered in the book. So as we just said, set up loss function,
optimizer, we just talked about these things. But I mean, you can just go to this book website
and reference it. Oh, we don't want that. We want this link. Come on, I knew you can't even
copy and paste. How are you supposed to code? I know I've been promising code this whole time,
so let's write some. So let's set up the loss function. What did we say it was? We're going to
call it L O double S F N for loss function. And we're going to call B C E with logit's loss. So B
C E with logit's loss. This has the sigmoid activation function built in. And we haven't covered what
the sigmoid activation function is, but we are going to don't you worry about that built in.
In fact, if you wanted to learn what the sigmoid activation function is, how could you find out
sigmoid activation function? But we're going to see it in action. Activation functions in neural
networks. This is the beautiful thing about machine learning. There's so much stuff

```
out there.
People have written some great articles. You've got formulas here. PyTorch has
implemented that
behind the scenes for us. So thank you, PyTorch. But if you recall, sigmoid
activation function
```

Section 102: Main Topics

Key Topics:

- So sigmoid torch dot sigmoid and pytorch
- So these two are going to work in tandem again, when we write our training loop, and we'll set our learning rate to 0
- Okay, let's see if we can implement something similar to that just using pure pytorch

▶ 📄 Click to view detailed content

```
built in, where did we discuss the architecture of a classification network? What
do we have here?
Right back in the zeroth chapter of this little online book thing that we heard
here. Binary
classification. We have output activation. Oh, oh, look at that. So sigmoid torch
dot sigmoid and
pytorch. All right. And then for multi class classification, we need the softmax.
Okay. Names
on a page again, but this is just a reference table so we can keep coming back to.
So let's just
keep going with this. I just want to highlight the fact that nn dot BCE loss also
exists. So
this requires BCE loss equals requires inputs to have gone through the sigmoid
activation function
prior to input to BCE loss. And so let's look up the documentation. I'm going to
comment that
out because we're going to stick with using this one. Now, why would we stick with
using this one?
Let's check out the documentation, hey, torch dot nn. And I realized this video is
all over the
place, but we're going to step back through BCE loss with logits. Did I even say
this right?
With logits loss. So with I got the width around the wrong way. So let's check this
out. So this
loss combines a sigmoid layer with the BCE loss in one single class. So if we go
back to the code,
BCE loss is this. So if we combined an n dot sequential, and then we passed in an n
dot sigmoid,
and then we went and then dot BCE loss, we'd get something similar to this. But if
we keep reading
```

in the documentation, because that's just I just literally read that it combines sigmoid with BCE
loss. But if we go back to the documentation, why do we want to use it? So this version is more
numerically stable than using a plain sigmoid by a BCE loss, followed by a BCE loss. As by
combining the operations into one layer, we take advantage of the log sum x trick for numerical
stability, beautiful. So if we use this log function, loss function for our binary cross entropy,
we get some numeric stability. Wonderful. So there's our loss function. We've got the sigmoid
activation function built in. And so we're going to see the difference between them later on,
like in the flesh, optimizer, we're going to choose, hmm, let's stick with SGD, hey,
old faithful stochastic gradient descent. And we have to set the parameters here, the parameters
parameter params equal to our model parameters would be like, hey, stochastic gradient descent,
please update. If we get another code cell behind here, please update our model parameters model
with respect to the loss, because we'd like our loss function to go down. So these two are going
to work in tandem again, when we write our training loop, and we'll set our learning rate to 0.1.
We'll see where that gets us. So that's what the optimizer is going to do. It's going to optimize
all of these parameters for us, which is amazing. And the principal would be the same, even if there
was 100 layers here, and 10,000, a million different parameters here. So we've got a loss function,
we've got an optimizer. And how about we create an evaluation metric. So let's calculate
accuracy at the same time. Because that's very helpful with classification problems is accuracy.
Now, what is accuracy? Well, we could look up formula for accuracy. So true positive over true
positive plus true negative times 100. Okay, let's see if we can implement something similar to that
just using pure pytorch. Now, why would we want accuracy? Because the accuracy is out of 100
examples. What percentage does our model get right? So for example, if we had a coin toss,
and we did 100 coin tosses in our model predicted heads 99 out of 100 times, and it was right
every single time, it might have an accuracy of 99%, because it got one wrong. So 99 out of 100,
it gets it right. So dev accuracy FN accuracy function, we're going to pass it y true. So
remember, any type of evaluation function or loss function is comparing the predictions to
the ground truth labels. So correct equals, this is going to see how many of our y true
or y threads are correct. So torch equal stands for, hey, how many of these samples y true are
equal to y pred? And then we're going to get the sum of that, and we need to get

```
the item from
that because we want it as a single value in Python. And then we're going to
calculate the
accuracy, ACC stands for accuracy, equals correct, divided by the length of samples
that we have
as input. And then we're going to times that by 100, and then return the accuracy.
Wonderful.
So we now have an accuracy function, we're going to see how all the three of these
come into play
when we write a training loop, which we might as we get started on the next few
videos, hey,
I'll see you there. Welcome back. In the last video, we discussed some different
loss function
options for our classification models. So we learned that if we're working with
binary cross
entropy or binary classification problems, we want to use binary cross entropy. And
pie torch
has two different times of binary cross entropy, except one is a bit more
numerically stable.
That's the BCE with logit's loss, because it has a sigmoid activation function
built in.
So that's straight from the pie to its documentation. And that for optimizer wise,
we have a few
different choices as well. So if we check out this section here on the pie torch
book, we have a
few different loss functions and optimizers for different problems and the pie
torch code that
```

Section 103: Main Topics

Key Topics:

- So if we go into this stack overflow answer, we saw machine learning, what is a logit
- That's the exciting part of machine learning
- So that's the definition of a logit in machine learning and deep learning

▶ 📄 Click to view detailed content

```
we can implement. But the premise is still the same across the board of different
problems.
The loss function measures how wrong our model is. And the goal of the optimizer is
to optimize
the model parameters in such a way that the loss function goes down. And we also
implemented our
own accuracy function metric, which is going to evaluate our models predictions
using accuracy
as an evaluation metric, rather than just loss. So let's now work on training a
```

model.
So what should we do first? Well, do you remember the steps in a pie torch training loop?
So to train our model, we're going to need to build a training loop. So if you watch the video
on the pie torch, so if you can Google unofficial pie torch song, you should find my, there we go,
the unofficial pie torch optimization loop song. We're not going to watch that. That's going to
be a little tidbit for the steps that we're going to code out. But that's just a fun little jingle
to remember these steps here. So if we go into the book section, this is number three train model,
exactly where we're up to here. But we have pie torch training loop steps. Remember, for an
epoch in a range, do the forward pass, calculate the loss, optimizer zero grand, loss backward,
optimizer step, step, step. We keep singing this all day. You could keep reading those steps all
day, but it's better to code them. But let's write this out. So forward pass to calculate the loss,
three, optimizer zero grad, four. What do we do? Loss backward. So back propagation,
I'll just write that up in here back propagation. We've linked to some extra resources. If you'd
like to find out what's going on in back propagation, we're focused on code here, and then gradient
descent. So optimizer step. So build a training loop with the following steps. However, I've kind
of mentioned a few things that need to be taken care of before we talk about the forward pass.
So we've talked about logits. We looked up what the hell is a logit. So if we go into this stack
overflow answer, we saw machine learning, what is a logit? How about we see that? We need to
do a few steps. So I'm going to write this down. Let's get a bit of clarity about us, Daniel.
We're kind of all over the place at the moment, but that's all right. That's the exciting part
of machine learning. So let's go from going from raw logits to prediction probabilities
to prediction labels. That's what we want. Because to truly evaluate our model, we want to
so let's write in here our model outputs going to be raw logit. So that's the definition of a
logit in machine learning and deep learning. You might read some few other definitions, but for us,
the raw outputs of our model, model zero are going to be referred to as logits. So then model zero,
so whatever comes out of here are logits. So we can convert these logits into prediction probabilities
by passing them to some kind of activation function, e.g. sigmoid for binary cross entropy
and softmax for multi-class classification. I've got binary class e-fication. I have to
sound it out every time I spell it for binary classification. So class e-fication. So we're

```
going to see multi-class classification later on, but we want some prediction
probabilities.
We're going to see what they look like in a minute. So we want to go from logits to
prediction
probabilities to prediction labels. Then we can convert our model's prediction
probabilities to
prediction labels by either rounding them or taking the argmax.
So round is for binary classification and argmax will be for the outputs of the
softmax activation
function, but let's see it in action first. So I've called the logits are the raw
outputs of our
model with no activation function. So view the first five outputs of the forward
pass
on the test data. So of course, our model is still instantiated with random values.
So we're
going to set up a variable here, y logits, and model zero, we're going to pass at
the test data.
So x test, not text, two device, because our model is currently on our CUDA device
and we need
our test data on the same device or target device. Remember, that's why we're
writing device
agnostic codes. So this would work regardless of whether there's a GPU active or
not. Let's have
a look at the logits. Oh, okay. Right now, we've got some positive values here. And
we can see that
they're on the CUDA device. And we can see that they're tracking gradients. Now,
ideally,
we would have run torch dot inference mode here, because we're making predictions.
And the rule
of thumb is whenever you make predictions with your model, you turn it into a vowel
mode.
We just have to remember to turn it back to train when we want to train and you run
torch dot
inference mode. So we get a very similar set up here. We just don't have the
gradients being
tracked anymore. Okay. So these are called logits. The logits are the raw outputs
of our model,
without being passed to any activation function. So an activation function is
something a little
separate from a layer. So if we come up here, we've used layer. So in the neural
networks that we
start to build and the ones that you'll subsequently build are comprised of layers
and activation
functions, we're going to make the concept of an activation function a little bit
more clear later
on. But for now, just treat it all as some form of mathematical operation. So if we
were to pass
data through this model, what is happening? Well, it's going through the linear
layer. Now recall,
```

Section 104: Main Topics

Key Topics:

- So we need this in the same format as this, which is not of course

▶ 📄 Click to view detailed content

```
we've seen this a few times now torch and then linear. If we pass data through a
linear layer,
it's applying the linear transformation on the incoming data. So it's performing
this
mathematical operation behind the scenes. So why the output equals the input x
multiplied by a
weight tensor a this could really be w which is transposed so that this is doing a
dot product
plus a bias term here. And then if we jump into our model state deck, we've got
weight
and we've got bias. So that's the formula that's happening in these two layers. It
will be different
depending on the layer that we choose. But for now, we're sticking with linear. And
so that the
raw output of our data going through our two layers, the logits is going to be this
information
here. However, it's not in the same format as our test data. And so if we want to
make a comparison
to how good our model is performing, we need apples to apples. So we need this in
the same format
as this, which is not of course. So we need to go to a next step. Let's use the
sigmoid. So use the
sigmoid activation function on our model logits. So why are we using sigmoid? Well,
recall in a
binary classification architecture, the output activation is the sigmoid function
here. So now
let's jump back into here. And we're going to create some predprobs. And what this
stands for
on our model logits to turn them into prediction probabilities, probabilities. So
why predprobs
equals torch sigmoid, why logits? And now let's have a look at why predprobs. What
do we get from
this? Oh, when we still get numbers on a page, goodness gracious me. But the
important point
now is that they've gone through the sigmoid activation function, which is now we
can pass these
to a torch dot round function. Let's have a look at this torch dot round. And what
do we get?
Predprobs. Oh, the same format as what we've got here. Now you might be asking
like, why don't we
just put torch dot round here? Well, that's a little, this step is required to, we
can't just do it on
the raw logits. We need to use the sigmoid activation function here to turn it into
prediction
probabilities. And now what is a prediction probability? Well, that's a value
usually between 0 and 1
for how likely our model thinks it's a certain class. And in the case of binary
cross entropy,
these prediction probability values, let me just write this out in text. So for our
```

prediction
probability values, we need to perform a range style rounding on them. So this is a decision
boundary. So this will make more sense when we go why predprobs, if it's equal to 0.5 or greater
than 0.5, we set y equal to one. So y equal one. So class one, whatever that is, a red dot or a
blue dot, and then why predprobs, if it is less than 0.5, we set y equal to zero. So this is class
zero. You can also adjust this decision boundary. So say, if you wanted to increase this value,
so anything over 0.7 is one. And below that is zero. But generally, most commonly, you'll find
it split at 0.5. So let's keep going. Let's actually see this in action. So how about we
recode this? So find the predicted probabilities. And so we want no, sorry, we want the predicted
labels, that's what we want. So when we're evaluating our model, we want to convert the outputs of
our model, the outputs of our model are here, the logits, the raw outputs of our model are
logits. And then we can convert those logits to prediction probabilities using the sigmoid function
on the logits. And then we want to find the predicted labels. So we go raw logits output of our model,
prediction probabilities after passing them through an activation function, and then prediction labels.
This is the steps we want to take with the outputs of our model. So find the predicted labels.
Let's go in here a little bit different to our regression problem previously, but nothing we can't
handle. Torch round, we're going to go y-pred-probs. So I like to name it y-pred-probs for prediction
probabilities and y-preds for prediction labels. Now let's go in full if we wanted to. So y-pred
labels equals torch dot round torch dot sigmoid. So sigmoid activation function for binary cross
entropy and model zero x test dot two device. Truly this should be within inference mode code,
but for now we'll just leave it like this to have a single example of what's going on here.
Now I just need to count one, two, there we want. That's where we want the index. We just want it
on five examples. So check for equality. And we want print torch equal. We're going to check
y-pred's dot squeeze is equal to y-pred labels. So just we're doing the exact same thing. And we
need squeeze here to get rid of the extra dimension that comes out. You can try doing this without
squeeze. So get rid of extra dimension once again. We want y-pred's dot squeeze. Fair bit of code
there, but this is what's happened here. We create y-pred's. So we turn the y-pred
probes into y-pred's. And then we just do the full step over again. So we make predictions with
our model, we get the raw logits. So this is logits to pred probes to pred labels. So the raw
outputs of our model are logits. We turn the logits into prediction probabilities

```
using torch
sigmoid. And we turn the prediction probabilities into prediction labels using
torch dot round.
And we fulfill this criteria here. So everything above 0.5. This is what torch dot
round does.
```

Section 105: Main Topics

Key Topics:

- They're in the same format, but of course they're not the same values because this model is using random weights to make predictions
- And how can we check that, of course, while we can type in device, we can run that cell
- Do we have the PyTorch

▶ 📄 Click to view detailed content

```
Turns it into a 1. Everything below 0.5 turns it into a 0. The predictions right
now are going to
be quite terrible because our model is using random numbers. But y-pred's found
with the steps above
is the same as y-pred labels doing the more than one hit. Thanks to this check for
equality using
torch equal y-pred's dot squeeze. And we just do the squeeze to get rid of the
extra dimensions.
And we have out here some labels that look like our actual y-test labels. They're
in the same format,
but of course they're not the same values because this model is using random
weights to make predictions.
So we've done a fair few steps here, but I believe we are now in the right space to
start building
a training a test loop. So let's write that down here 3.2 building a training and
testing loop.
You might want to have a go at this yourself. So we've got all the steps that we
need to do the
forward pass. But the reason we've done this step here, the logits, then the pred
probes and the
pred labels, is because the inputs to our loss function up here, this requires, so
BCE with
logits loss, requires what? Well, we're going to see that in the next video, but
I'd encourage
you to give it a go at implementing these steps here. Remember the jingle for an
epoch in a range,
do the forward pass, calculate the loss, which is BC with logits loss, optimise a
zero grad,
which is this one here, last backward, optimise a step, step, step. Let's do that
together in the
```

next video. Welcome back. In the last few videos, we've been working through creating a model for
a classification problem. And now we're up to training a model. And we've got some steps here,
but we started off by discussing the concept of logits. Logits are the raw output of the model,
whatever comes out of the forward functions of the layers in our model. And then we discussed how
we can turn those logits into prediction probabilities using an activation function,
such as sigmoid for binary classification, and softmax for multi class classification.
We haven't seen softmax yet, but we're going to stick with sigmoid for now because we have
binary classification data. And then we can convert that from prediction probabilities
to prediction labels. Because remember, when we want to evaluate our model, we want to compare
apples to apples. We want our models predictions to be in the same format as our test labels.
And so I took a little break after the previous video. So my collab notebook has once again
disconnected. So I'm just going to run all of the cells before here. It's going to reconnect up
here. We should still have a GPU present. That's a good thing about Google collab is that if you
change the runtime type to GPU, it'll save that wherever it saves the Google collab notebook,
so that when you restart it, it should still have a GPU present. And how can we check that,
of course, while we can type in device, we can run that cell. And we can also check
Nvidia SMI. It'll tell us if we have an Nvidia GPU with CUDA enabled ready to go.
So what's our device? CUDA. Wonderful. And Nvidia SMI. Excellent. I have a Tesla P100 GPU.
Ready to go. So with that being said, let's start to write a training loop. Now we've done this before,
and we've got the steps up here. Do the forward pass, calculate the loss. We've spent enough on
this. So we're just going to start jumping into write code. There is a little tidbit in this one,
though, but we'll conquer that when we get to it. So I'm going to set a manual seed,
torch top manual seed. And I'm going to use my favorite number 42. This is just to ensure
reproducibility, if possible. Now I also want you to be aware of there is also another
form of random seed manual seed, which is a CUDA random seed. Do we have the PyTorch?
Yeah, reproducibility. So torch dot CUDA dot manual seed dot seed. Hmm. There is a CUDA
seed somewhere. Let's try and find out. CUDA. I think PyTorch have just had an upgrade to
their documentation. Seed. Yeah, there we go. Okay. I knew it was there. So torch dot CUDA
dot manual seed. So if we're using CUDA, we have a CUDA manual seed as well. So let's see what
happens if we put that to watch that CUDA dot manual seed 42. We don't necessarily

```
have to put
these. It's just to try and get as reproducible as numbers as possible on your
screen and my screen.
Again, what is more important is not necessarily the numbers exactly being the same
lining up
between our screens. It's more so the direction of which way they're going. So
let's set the number
of epochs. We're going to train for 100 epochs. epochs equals 100. But again, as
you might have
guessed, the CUDA manual seed is for if you're doing operations on a CUDA device,
which in our
case, we are. Well, then perhaps we'd want them to be as reproducible as possible.
So speaking of
CUDA devices, let's put the data to the target device because we're working with or
we're writing
data agnostic code here. So I'm going to write x train y train equals x train two
device,
comma y train dot two device, that'll take care of the training data. And I'm going
to do the
same for the testing data equals x test two device. Because if we're going to run
our model
on the CUDA device, we want our data to be there too. And the way we're writing our
code,
```

Section 106: Main Topics

Key Topics:

- This is a testament to the Pythonic nature of PyTorch as well
- For some reason, you stumble across some pytorch code that's using BCE loss, not BCE with logits loss

▶ 📄 Click to view detailed content

```
our code is going to be device agnostic. Have I said that enough yet? So let's also
build our
training and evaluation loop. Because we've covered the steps in here before, we're
going to start
working a little bit faster through here. And don't worry, I think you're up to it.
So for an epoch
in a range of epochs, what do we do? We start with training. So let me just write
this.
Training model zero dot train. That's the model we're working with. We call the
train,
which is the default, but we're going to do that anyway. And as you might have
guessed,
the code that we're writing here is, you can functionize this. So we're going to do
this later
on. But just for now, the next couple of videos, the next module or two, we're
```

going to keep
writing out the training loop in full. So this is the part, the forward pass, where there's a
little bit of a tidbit here compared to what we've done previously. And that is because we're
outputting raw logits here, if we just pass our data straight to the model. So model zero
x train. And we're going to squeeze them here to get rid of an extra dimension. You can try to
see what the output sizes look like without squeeze. But we're just going to call squeeze
here. Remember, squeeze removes an extra one dimension from a tensor. And then to convert it
into prediction labels, we go torch dot round. And torch dot sigmoid, because torch dot sigmoid
is an activation function, which is going to convert logits into what convert the logits
into prediction probabilities. So why logits? And I'm just going to put a note here. So this
is going to go turn logits into pred probes into pred labels. So we've done the forward pass.
So that's a little tidbit there. We could have done all of this in one step, but I'll show you
why we broke this apart. So now we're going to calculate loss slash accuracy. We don't necessarily
have to calculate the accuracy. But we did make an accuracy function up here. So that we can
calculate the accuracy during training, we could just stick with only calculating the loss. But
sometimes it's cool to visualize different metrics loss plus a few others while your model is training.
So let's write some code to do that here. So we'll start off by going loss equals loss
f n and y logits. Ah, here's the difference of what we've done before. Previously in the notebook
zero one, up to zero two now, we passed in the prediction right here. But because what's our
loss function? Let's have a look at our loss function. Let's just call that see what it returns.
BCE with logits loss. So the BCE with logits expects logits as input. So as you might have guessed,
loss function without logits. If we had nn dot BCE loss, notice how we haven't got with logits.
And then we called loss f n, f n stands for function, by the way, without logits. What do we get?
So BCE loss. So this loss expects prediction probabilities as input. So let's write some code
to differentiate between these two. As I said, we're going to be sticking with this one.
Why is that because if we look up torch BCE with logits loss, the documentation states that it's
more numerically stable. So this loss combines a sigmoid layer and the BCE loss into one single
class, and is more numerically stable. So let's come back here, we'll keep writing some code.
And the accuracy is going to be accuracy f n. So our accuracy function, there's a little bit of a

```
difference here is why true equals y train for the training data. So this will be
the training
accuracy. And then we have y pred equals y pred. So this is our own custom accuracy
function
that we wrote ourselves. This is a testament to the Pythonic nature of PyTorch as
well.
We've just got a pure Python function that we've slotted into our training loop,
which is essentially what the loss function is behind the scenes.
Now, let's write here, and then dot BCE with logits loss expects raw logits. So the
raw output
of our model as input. Now, what if we were using a BCE loss on its own here? Well,
let's just write
some code for that. So let's call loss function. And then we want to pass in y
pred. Or we can
just go why or torch sigmoid. So why would we pass in torch sigmoid on the logits
here? Because
remember, calling torch dot sigmoid on our logits turns our logits into prediction
probabilities.
And then we would pass in y train here. So if this was BCE loss expects this
expects prediction
probabilities as input. So does that make sense? That's the difference between with
logits. So
our loss function requires logits as input. Whereas if we just did straight up BCE
loss,
we need to call torch dot sigmoid on the logits because it expects prediction
probabilities as
input. Now, I'm going to comment that out because our loss function is BCE with
logits loss. But
just keep that in mind. For some reason, you stumble across some pytorch code
that's using BCE loss,
not BCE with logits loss. And you find that torch dot sigmoid is calling here, or
you come across
some errors, because your inputs to your loss function are not what it expects. So
with that
being said, we can keep going with our other steps. So we're up to optimizer zero
grad. So
optimizer dot zero grad. Oh, this is step three, by the way. And what's after this?
Once we've
zero grad the optimizer, we do number four, which is loss backward. We can go last
backward. And then
```

Section 107: Main Topics

Key Topics:

- And I'm singing the unofficial pytorch optimization loop song there
- And of course, we're going to compute the test logits, because logits are the raw output of our model with no modifications

- Well, just if we've done before, and we're going to go loss FN test logits, because our loss function, we're using what we're using BCE with logits loss, expects logits as input, where do we find that out in the documentation, of course, then we come back here, test logits, we're going to compare that to the Y test labels

▶ 📄 Click to view detailed content

```
we go what's next? Optimizer step step step. So optimizer dot step. And I'm singing
the unofficial
pytorch optimization loop song there. This is back propagation. Calculate the
gradients with respect
to all of the parameters in the model. And the optimizer step is update the
parameters to reduce
the gradients. So gradient descent, hence the descent. Now, if we want to do
testing,
well, we know what to do here now, we go model zero, what do we do? We call model
dot of
al when we're testing. And if we're making predictions, that's what we do when we
test,
we make predictions on the test data set, using the patterns that our model has
learned on the
training data set, we turn on inference mode, because we're doing inference, we
want to do the
forward pass. And of course, we're going to compute the test logits, because logits
are the raw output
of our model with no modifications. X test dot squeeze, we're going to get rid of
an extra one
dimension there. Then we create the test pred, which is we have to do a similar
calculation to
what we've done here for the test pred, which is torch dot round. For our binary
classification,
we want prediction probabilities, which we're going to create by calling the
sigmoid function
on the test logits, prediction probabilities that are 0.5 or above to go to one,
and prediction
probabilities under 0.5 to go to level zero. So two is calculate the test loss,
test loss
slash accuracy. How would we do this? Well, just if we've done before, and we're
going to go
loss FN test logits, because our loss function, we're using what we're using BCE
with logits loss,
expects logits as input, where do we find that out in the documentation, of course,
then we come back here, test logits, we're going to compare that to the Y test
labels.
And then for the test accuracy, what are we going to do? We're going to call
accuracy FN
on Y true equals Y test, and Y pred equals test pred. Now you might be thinking,
why did I switch
up the order here for these? Oh, and by the way, this is important to know with
logits loss.
So with these loss functions, the order here matters of which way you put in your
parameters.
So predictions come first, and then true labels for our loss function. You might be
wondering why I've done it the reverse for our accuracy function, Y true and Y
```

pred. That's just
because I like to be confusing. Well, not really. It's because if we go to scikit-learn, I base a
lot of my structured code of how scikit-learn structures things. The scikit-learn metrics accuracy
score goes Y true Y pred. So I base it off that order, because the scikit-learn metrics package
is very helpful. So I've based our metric evaluation metric function off this one. Whereas PyTorch's
loss function does it in the reverse order, and it's important to get these in the right order.
Exactly why they do it in that order. I couldn't tell you why. And we've got one final step, which
is to print out what's happening. So how about we go, we're doing a lot of epochs here, 100 epochs.
So we'll divide the epoch by 10 to print out every epoch or every 10th epoch, sorry. And we have a
couple of different metrics that we can print out this time. So we're going to print out the epoch
number epoch. And then we're going to print out the loss. So loss, how many decimal points?
We'll go point five here. This is going to be the training loss. We'll also do the accuracy,
which will be the training accuracy. We could write trainiac here for our variable to be a little bit,
make them a little bit more understandable. And then we go here, but we're just going to leave
it as loss and accuracy for now, because we've got test loss over here, test loss. And we're
going to do the same five decimal points here. And then we're going to go test accuracy as well.
Test act dot, we'll go to for the accuracy. And because it's accuracy, we want a percentage. This
is the percent out of 100 guesses. What's the percentage that our model gets right on the training
data and the testing data, as long as we've coded all the functions correctly. Now,
we've got a fair few steps here. My challenge to you is to run this. And if there are any errors,
try to fix them. No doubt there's probably one or two or maybe more that we're going to have to
fix in the next video. But speaking of next videos, I'll see you there. Let's train our first
classification model. Well, this is very exciting. I'll see you soon.
Welcome back. In the last video, we wrote a mammoth amount of code, but nothing that we
can't handle. We've been through a lot of these steps. We did have to talk about a few tidbits
between using different loss functions, namely the BCE loss, which is binary cross entropy loss,
and the BCE with logit's loss. We discussed that the BCE loss in PyTorch expects prediction
probabilities as input. So we have to convert our model's logits. Logits are the raw output of the
model to prediction probabilities using the torch dot sigmoid activation function. And if we're using
BCE with logits loss, it expects raw logits as input as sort of the name hints at. And so we

```
just pass it straight away the raw logits. Whereas our own custom accuracy function
compares labels
to labels. And that's kind of what we've been stepping through over the last few
videos,
is going from logits to predprobs to pred labels, because that's the ideal output
of our model is
```

Section 108: Main Topics

Key Topics:

- Our model doesn't seem to be learning anything
- So turn to investigate why our model is not learning
- It looks like our model isn't learning anything

▶ 📄 Click to view detailed content

```
some kind of label that we as humans can interpret. And so let's keep pushing
forward. You may have
already tried to run this training loop. I don't know if it works. We wrote all
this code to get
them in the last video. And it's probably an error somewhere. So you ready? We're
going to train
our first classification model together for 100 epochs. If it all goes to plan in
three, two,
one, let's run. Oh my gosh, it actually worked the first time. I promise you, I
didn't change
anything in here from the last video. So let's inspect what's going on. It trains
pretty fast.
Why? Well, because we're using a GPU, so it's going to be accelerated as much as it
can anyway.
And our data set is quite small. And our network is quite small. So you won't
always
get networks training this fast. They did 100 epochs in like a second. So the loss.
Oh,
0.69973. It doesn't go down very much. The accuracy even starts high and then goes
down.
What's going on here? Our model doesn't seem to be learning anything. So what would
an ideal
accuracy be? An ideal accuracy is 100. And what's an ideal loss value? Well, zero,
because lower
is better for loss. Hmm, this is confusing. And now if we go, have a look at our
blue and red
dots. Where's our data? So I reckon, do we still have a data frame here? How many
samples do we
have of each? Let's inspect. Let's do some data analysis. Where do we create a data
frame here?
Now, circles, do we still have this instantiated circles dot label dot? We're going
to call on
```

pandas here, value counts. Is this going to output how many of each? Okay. Wow, we've got 500 of
class one and 500 of class zero. So we have 500 red dots and blue dots, which means we have a
balanced data set. So if we're getting, we're basically trying to predict heads or tails here.
So if we're getting an accuracy of under 50%, or about 50%, if you rounded it up.
Our model is basically doing as well as guessing. Well, what gives? Well, I think we should get
visual with this. So let's make some predictions with our model, because these are just numbers
on the page. It's hard to interpret what's going on. But our intuition now is because we have 500
samples of each, or in the case of the training data set, we have 400 of each because we have
800 samples in the training data set. And we have in the testing data set, we have 200 total
samples. So we have 100 of each. We're basically doing a coin flip here. Our model is as good as
guessing. So turn to investigate why our model is not learning. And one of the ways we can do
that is by visualizing our predictions. So let's write down here from the metrics. It looks like
our model isn't learning anything. So to inspect it, let's make some predictions and make them
visual. And we're right down here. In other words, visualize, visualize, visualize. All right.
So we've trained a model. We've at least got the structure for the training code here.
But this is the right training code. We've written this code before. So you know that this set up
for training code does allow a model to train. So there must be something wrong with either
how we've built our model, the data set. But let's keep going and investigate together.
So to do so, I've got a function that I've pre-built earlier. Did I mention that we're learning side
by side of a machine learning cooking show? So this is an ingredient I prepared earlier,
a part of a dish. So to do so, we're going to import a function called plot decision,
or maybe I'll turn this into code, plot decision boundary.
Welcome to the cooking show, cooking with machine learning. What model will we cook up today?
So if we go to pytorch deep learning, well, it's already over here, but this is the home repo for
the course, the link for this will be scattered everywhere. But there's a little function here
called helper functions dot py, which I'm going to fill up with helper functions throughout the
course. And this is the one I'm talking about here, plot decision boundary. Now we could just
copy this into our notebook, or I'm going to write some code to import this programmatically,
so we can use other functions from in here. Here's our plot predictions function that we made in
the last section, zero one, but this plot decision boundary is a function that I

```
got inspired by
to create from madewithml.com. Now this is another resource, a little bit of an
aside,
I highly recommend going through this by Goku Mohandas. It gives you the
foundations of neural
networks and also ml ops, which is a field, which is based on getting your neural
networks and machine
learning models into applications that other people can use. So I can't recommend
this resource
enough. So please, please, please check that out if you want another resource for
machine learning,
but this is where this helper function came from. So thank you, Goku Mohandas. I've
made a little
bit of modifications for this course, but not too many. So we could either copy
that, paste it in
here, or we could write some code to import it for us magically, or using the power
of the internet,
right, because that's what we are. We're programmers, we're machine learning
engineers, we're data
scientists. So from pathlib, so the request module in Python is a module that
allows you to make
requests, a request is like going to a website, hey, I'd like to get this code from
you, or this
```

Section 109: Main Topics

Key Topics:

- So download helper functions from learn pytorch repo
- If we go back, this is just pytorch deep learning the repo for this course slash helper functions
- So this code is basically saying hey requests, get the information that's at this link here, which is of course, all of this code here, which is a Python script

▶ 📄 Click to view detailed content

```
information from you, can you please send it to me? So that's what that allows us
to do,
and pathlib, we've seen pathlib before, but it allows us to create file parts.
Because why? Well,
we want to save this helper function dot pi script to our Google collab files. And
so we can do this
with a little bit of code. So download helper functions from learn pytorch repo. If
it's not
already downloaded. So let's see how we can do that. So we're going to write some
if else code to
check to see if the path of helper functions dot pi already exist, we don't want to
download it again.
```

So at the moment, it doesn't exist. So this if statement is going to return false. So let's just
print out what it does if it returns true helper functions dot pi already exists. We might we could
even probably do a try and accept looping about if else will help us out for now. So if it exists
else, print downloading helper functions dot pi. So ours doesn't exist. So it's going to make a
request or let's set up our request request dot get. And here's where we can put in a URL. But we
need the raw version of it. So this is the raw version. If we go back, this is just pytorch deep
learning the repo for this course slash helper functions. If I click raw, I'm going to copy that.
Oh, don't want to go in there want to go into request get type that in this has to be in a
string format. So we get the raw URL. And then we're going to go with open, we're going to open
a file called helper functions dot pi. And we're going to set the context to be right binary,
which is wb as file F is a common short version of writing file. Because we're going to call
file dot write, and then request dot content. So this code is basically saying hey requests,
get the information that's at this link here, which is of course, all of this code here,
which is a Python script. And then we're going to create a file called helper functions dot pi,
which gives us write permissions. We're going to name it F, which is short for file. And then
we're going to call on it file dot write the content of the request. So instead of talking
through it, how about we see it in action? We'll know if it works if we can from helper functions
import plot predictions, we're going to use plot predictions later on, as well as plot decision
boundary. So plot predictions we wrote in the last section. Wonderful. I'm going to write here,
downloading helper functions dot pi did at work. We have helper functions dot pi. Look at that,
we've done it programmatically. Can we view this in Google column? Oh my goodness, yes we can.
And look at that. So this may evolve by the time you do the course, but these are just some general
helper functions rather than writing all of this out. If you would like to know what's going on
in plot decision boundary, I encourage you to read through here. And what's going on,
you can step by step at yourself. There is nothing here that you can't tackle yourself. It's all
just Python code, no secrets just Python code. We've got we're making predictions with a
PyTorch model. And then we're testing for multi class or binary. So we're going to get out of that.
But now let's see the ultimate test is if the plot decision boundary function works. So again,
we could discuss plot decision boundary of the model. We could discuss what it does

```
behind the scenes
to the cows come home. But we're going to see it in real life here. I like to get
visual.
So fig size 12, six, we're going to create a plot here, because we are adhering to
the data
explorer's motto of visualize visualize visualize. And we want to subplot because
we're going to
compare our training and test sets here, train. And then we're going to go PLT, or
actually we'll
plot the first one, plot decision boundary. Now, because we're doing a training
plot here,
we're going to pass in model zero and X train and Y train. Now, this is the order
that the
parameters go in. If we press command shift space, I believe Google collab, if it's
working with me,
we'll put up a doc string. There we go, plot decision boundary. Look at the inputs
that it
takes model, which is torch and end up module. And we've got X, which is our X
value, which is a
torch tensor, and Y, which is our torch tensor value here. So that's for the
training data.
Now, let's do the same for the testing data, plot dot subplot. This is going to be
one, two,
two for the index. This is just number of rows of the plot, number of columns. And
this is the
index. So this plot will appear on the first slot. We're going to see this anyway.
Anything
below this code will appear on the second slot, PLT dot title. And we're going to
call this one
test. Then we're going to call plot decision boundary. If this works, this is going
to be some
serious magic. I love visualization functions in machine learning. Okay, you ready?
Three,
two, one, let's check it out. How's our model doing? Oh, look at that. Oh, now it's
clear.
So behind the scenes, this is the plots that plot decision boundary is making. Of
course,
this is the training data. This is the testing data, not as many dot points here,
but the same
```

Section 110: Main Topics

Key Topics:

- So I want you to have a think about this, even if you're completely new to deep learning, can we
- This is similar to what we've been doing with PyTorch
- PyTorch is essentially just a collection of Python scripts that we're using to build neural networks

▶ 📄 Click to view detailed content

```
sort of line of what's going on. So this is the line that our model is trying to
draw through the
data. No wonder it's getting about 50% accuracy and the loss isn't going down. It's
just trying
to split the data straight through the middle. It's drawing a straight line. But
our data is
circular. Why do you think it's drawing a straight line? Well, do you think it has
anything to do
with the fact that our model is just made with using pure linear layers? Let's go
back to our model.
What's it comprised on? Just a couple of linear layers. What's a linear line? If we
look up linear
line, is this going to work with me? I don't actually think it might. There we go.
Linear line,
all straight lines. So I want you to have a think about this, even if you're
completely
new to deep learning, can we? You can answer this question. Can we ever separate
this circular data
with straight lines? I mean, maybe we could if we drew straight lines here, but
then trying to
curve them around. But there's an easier way. We're going to see that later on. For
now,
how about we try to improve our model? So the model that we built, we've got 100
epochs.
I wonder if our model will improve if we trained it for longer. So that's a little
bit of a challenge
before the next video. See if you can train the model for 1000 epochs. Does that
improve the
results here? And if it doesn't improve the results here, have a think about why
that might be.
I'll see you in the next video. Welcome back. In the last video, we wrote some code
to download
a series of helper functions from our helper functions dot pi. And later on, you'll
see why
this is quite standard practice as you write more and more code is to write some
code, store them
somewhere such as a Python script like this. And then instead of us rewriting
everything that we
have and helper functions, we just import them and then use them later on. This is
similar to
what we've been doing with PyTorch. PyTorch is essentially just a collection of
Python scripts
that we're using to build neural networks. Well, there's a lot more than what we've
just done.
I mean, we've got one here, but PyTorch is a collection of probably hundreds of
different
Python scripts. But that's beside the point. We're trying to train a model here to
separate
blue and red dots. But our current model is only drawing straight lines. And I got
you to
have a think about whether our straight line model, our linear model could ever
separate this data.
Maybe it could. And I issued the challenge to see if it could if you trained for
1000 epochs.
```

So did it improve at anything? Is the accuracy any higher? Well, speaking of training for more
epochs, we're up to section number five, improving a model. This is from a model perspective. So now
let's discuss some ways. If you were getting results after you train a machine learning model or a
deep learning model, whatever kind of model you're working with, and you weren't happy with those
results. So how could you go about improving them? So this is going to be a little bit of an overview
of what we're going to get into. So one way is to add more layers. So give the model more chances
to learn about patterns in the data. Why would that help? Because if our model currently has two
layers, model zero dot state dinked. Well, we've got however many numbers here, 20 or so. So this
is zero flayer. This is the first layer. If we had 10 of these, well, we'd have 10 times the
amount of parameters to try and learn the patterns in this data, a representation of this data.
Another way is to add more hidden units. So what I mean by that is we created this model here,
and each of these layers has five hidden units. The first one outputs, out features equals five,
and this one takes in features equals five. So we could go from, go from five hidden units to
10 hidden units. The same principle as above applies here is that the more parameters our model has
to represent our data, the potentially now I say potentially here because some of these things
might not necessarily work. So our data sets quite simple. So maybe if we added too many layers,
our models trying to learn things that are too complex, it's trying to adjust too many numbers
for the data set that we have the same thing for more hidden units. What other options do we
have? Well, we could fit for longer, give the model more of a chance to learn because every epoch
is one pass through the data. So maybe 100 times looking at this data set wasn't enough.
So maybe you could fit for 1000 times, which was the challenge. Then there's change in the
activation functions, which we're using sigmoid at the moment, which is generally the activation
function you use for a binary classification problem. But there are also activation functions
you can put within your model. Hmm, there's a little hint that we'll get to that later.
Then there's change the learning rate. So the learning rate is the amount the optimizer will
adjust these every epoch. And if it's too small, our model might not learn anything because it's
taking forever to change these numbers. But if also on the other side of things, if the learning
rate is too high, these updates might be too large. And our model might just explode. There's an

```
actual problem in machine learning called exploding gradient problem, where the
numbers just get
```

Section 111: Main Topics

Key Topics:

- So we're going to have a look at some options here, add more layers and fit for longer, maybe changing the learning rate
- TensorBoard is a tool or a utility from PyTorch, which helps you to monitor experiments
- So remember how I said multiples of eight are pretty good generally in deep learning

▶ 📄 Click to view detailed content

```
too large. On the other side, there's also a vanishing gradients problem, where the
gradients
just go basically to zero too quickly. And then there's also change the loss
function. But I feel
like for now, sigmoid and binary cross entropy, pretty good, pretty standard. So
we're going to
have a look at some options here, add more layers and fit for longer, maybe
changing the learning
rate. But let's just add a little bit of color to what we've been talking about.
Right now,
we've fit the model to the data and made a prediction. I'm just going to step
through this.
Where are we up to? We've done this, we've done this, we've done these two, we've
built a training
loop, we've fit the model to the data, made a prediction, we've evaluated our model
visually,
and we're not happy with that. So we're up to number five, we're going to improve
through
experimentation. We don't need to use TensorBoard just yet, we're going to talk
about this as our
high level. TensorBoard is a tool or a utility from PyTorch, which helps you to
monitor experiments.
We'll see that later on. And then we'll get to this, we won't save our model until
we've got one
that we're happy with. And so if we look at what we've just talked about improving
a model from a
model's perspective, let's talk about the things we've talked about with some color
this time. So
say we've got a model here, this isn't the exact model that we're working with, but
it's similar
structure. We've got one, two, three, four layers, we've got a loss function BC
```

with Logit's loss,
we've got an optimizer, optimizer stochastic gradient descent, and if we did write some training code,
this is 10 epochs. And then the testing code here, I've just cut it out because it wouldn't fit on
the slide. Then if we wanted to go to a larger model, let's add some color here so we can highlight
what's happening, adding layers. Okay, so this one's got one, two, three, four, five, six layers.
And we've got another color here, which is I'd say this is like a little bit of a greeny blue
increase the number of hidden units. Okay, so the hidden units are these features here.
We've gone from 100 to 128 to 128. Remember, the out features of a previous layer have to line up
with the in features of a next layer. Then we've gone to 256. Wow. So remember how I said multiples
of eight are pretty good generally in deep learning? Well, this is where these numbers come from.
And then what else do we have change slash add activation functions? We haven't seen this before
and end up relu. If you want to jump ahead and have a look at what and end up relu is,
how would you find out about it? Well, I just Google and end up relu. But we're going to have
a look at what this is later on. We can see here that this one's got one, but this larger model has
some relu's scattered between the linear layers. Hmm, maybe that's a hint. If we combine a linear
layer with a relu, what's a relu layer? I'm not going to spoil this. We're going to find out
later on change the optimization function. Okay. So we've got SGD. Do you recall how I said
Adam is another popular one that works fairly well across a lot of problems as well. So Adam
might be a better option for us here. The learning rate as well. So maybe this learning rate was a
little too high. And so we've divided it by 10. And then finally, fitting for longer. So instead
of 10 epochs, we've gone to 100. So how about we try to implement some of these with our own model
to see if it improves what we've got going on here? Because frankly, like, this isn't
satisfactory. We're trying to build a neural network here. Neural networks are supposed to be
these models that can learn almost anything. And we can't even separate some blue dots from
some red dots. So in the next video, how about we run through writing some code to do some of
these steps here? In fact, if you want to try yourself, I'd highly encourage that. So I'd start
with trying to add some more layers and add some more hitting units and fitting for longer. You can
keep all of the other settings the same for now. But I'll see you in the next video. Welcome back.
In the last video, we discussed some options to improve our model from a model perspective. And

```
namely, we're trying to improve it so that the predictions are better, so that the
patterns it
learns better represent the data. So we can separate blue dots from red dots. And
you might be wondering
why we said from a model perspective here. So let me just write these down. These
options are all
from a models perspective, because they deal directly with the model, rather than
the data.
So there's another way to improve a models results is if the model was sound
already,
in machine learning and deep learning, you may be aware that generally if you have
more data samples,
the model learns or gets better results because it has more opportunity to learn.
There's a few
other ways to improve a model from a data perspective, but we're going to focus on
improving a model
from a models perspective. So, and because these options are all values we as
machine learning
engineers and data scientists can change, they are referred to as hyper parameters.
So a little bit of an important distinction here. Parameters are the numbers within
a model.
The parameters here, like these values, the weights and biases are parameters,
```

Section 112: Main Topics

Key Topics:

- Hyper parameters are what we as machine learning engineers and data scientists, such as adding more layers, more hidden units, fitting for longer number of epochs, activation functions, learning rate, loss functions are hyper parameters because they're values that we can change
- But generally, when you're doing machine learning experiments, you'd only like to change one value at a time and track the results
- So that's called experiment tracking and machine learning

▶ 📄 Click to view detailed content

```
are the values a model updates by itself. Hyper parameters are what we as machine
learning
engineers and data scientists, such as adding more layers, more hidden units,
fitting for longer
number of epochs, activation functions, learning rate, loss functions are hyper
parameters because
they're values that we can change. So let's change some of the hyper parameters of
our model.
So we'll create circle model v1. We're going to import from nn.module as well. We
could write this
```

model using nn.sequential, but we're going to subclass nn.module for practice.
Why would we use nn.sequential? Well, because as you'll see, our model is not too complicated,
but we subclass nn.module. In fact, nn.sequential. So if we write here, nn.sequential is also a
version of nn.module. But we subclass nn.module here for one for practice and for later on,
if we wanted to, or if you wanted to make more complex models, you're going to see a subclass
of nn.module a lot in the wild. So the first change we're going to update is the number
of hidden units. So out features, I might write this down before we do it. Let's try and improve
our model by adding more hidden units. So this will go from five and we'll increase it to 10.
And we want to increase the number of layers. So we want to go from two to three. We'll add an
extra layer and then increase the number of epochs. So we're going to go from 100 to 1,000. Now,
what can you, we're going to put on our scientist hats for a second. What would be the problem with
the way we're running this experiment? If we're doing all three things in one hit, why might that
be problematic? Well, because we might not know which one offered the improvement if there is
any improvement or degradation. So just to keep in mind going forward, I'm just doing this as an
example of how we can change all of these. But generally, when you're doing machine learning
experiments, you'd only like to change one value at a time and track the results. So that's called
experiment tracking and machine learning. We're going to have a look at experiment tracking a
little later on in the course, but just keep that in mind. A scientist likes to change one
variable of what's going on so that they can control what's happening. But we're going to
create this next layer here layer two. And of course, it takes the same number of out features as
in features as the previous layer. This is two because why our X train has. Let's look at just
the first five samples has two features. So now we're going to create self layer three, which
equals an n dot linear. The in features here is going to be 10. Why? Because the layer above
has out features equals 10. So what we've changed here so far is we've got hidden units previously
in the zero of this model was five. And now we've got a third layout, which previously before was
two. So these are two of our main changes here. And out features equals one, because why? Let's
have a look at speaking of why. Our why is just one number. So remember the shapes, the input and
output shapes of a model is one of the most important things in deep learning. We're going to see
different values for the shapes later on. But because we're working with this data set, we're

```
focused on two in features and one out feature. So now that we've got our layers
prepared,
what's next? Well, we have to override the forward method, because every subclass
of
an n dot module has to implement a forward method. So what are we going to do here?
Well, we could,
let me just show you one option. We could go z, which would be z for logits. Logits
is actually
represented by z, fun fact. But you could actually put any variable here. So this
could be x one,
or you could reset x if you wanted to. I just look putting a different one because
it's a little
less confusing for me. And then we could go update z by going self layer two. And
then the,
because z above is the output of layer one, it now goes into here. And then if we
go z,
again, equals self layer three, what's this going to take? It's going to take z
from above.
So this is saying, hey, give me x, put it through layer one, assign it to z. And
then
create a new variable z or override z with self layer two with z from before as the
input. And
then we've got z again, the output of layer two has the input for layer three. And
then we could
return z. So that's just passing our data through each one of these layers here.
But a way that
you can leverage speedups in PyTorch is to call them all at once. So layer three,
and we're going
to put self dot layer two. And this is generally how I'm going to write them. But
it also behind
the scenes, because it's performing all the operations at once, you leverage
whatever speed
ups you can get. Oh, this should be layer one. So it goes in order here. So what's
happening?
Well, it's computing the inside of the brackets first. So layer one, x is going
through layer one.
And then the output of x into layer one is going into layer two. And then the same
again,
for layer three. So this way, this way of writing operations, leverages, speed ups,
where possible
behind the scenes. And so we've done our Ford method there. We're just passing our
data through
layers with an extra hidden units, and an extra layer overall. So now let's create
an instance of
```

Section 113: Main Topics

Key Topics:

- Is that the same LR we use before learning rate
- Oh, potentially that our learning rate may be too big

▶ 📄 Click to view detailed content

```
circle model v one, which we're going to set to model one. And we're going to write
circle model
v one. And we're going to send it to the target device, because we like writing
device agnostic code.
And then we're going to check out model one. So let's have a look at what's going
on there.
Beautiful. So now we have a three layered model with more hidden units. So I wonder
if we trained
this model for longer, are we going to get improvements here? So my challenge to
you is we've already
done these steps before. We're going to do them over the next couple of videos for
completeness.
But we need to what create a loss function. So I'll give you a hint. It's very
similar to the one
we've already used. And we need to create an optimizer. And then once we've done
that, we need to
write a training and evaluation loop for model one. So give that a shot. Otherwise,
I'll see you
in the next video. We'll do this all together. Welcome back. In the last video, we
subclassed
nn.module to create circle model V one, which is an upgrade on circle model V zero.
In the
fact that we added more hidden units. So from five to 10. And we added a whole
extra layer.
And we've got an instance of it ready to go. So we're up to in the workflow. We've
got our data.
Well, we haven't changed the data. So we've built our new model. We now need to
pick a loss function.
And I hinted at before that we're going to use the same loss function as before.
The same optimizer. You might have already done all of these steps. So you may know
whether this
model works on our data set or not. But that's what we're going to work towards
finding out in
this video. So we've built our new model. Now let's pick a loss function and
optimizer. We could
almost do all of this with our eyes closed now, build a training loop, fit the
model to the data,
make a prediction and evaluate the model. We'll come back here. And let's set up a
loss function.
And by the way, if you're wondering, like, why would adding more features here,
we've kind of
hinted at this before. And why would an extra layer improve our model? Well, again,
it's back
to the fact that if we add more neurons, if we add more hidden units, and if we add
more layers,
it just gives our model more numbers to adjust. So look at what's going on here,
layer one,
layer two. Look how many more we have compared to model zero dot state date.
We have all of these. This is model zero. And we just upgraded it. Look how many
more we have
from just adding an extra layer and more hidden units. So now we have our optimizer
can change
these values to hopefully create a better representation of the data we're trying
to fit.
```

```
So we just have more opportunity to learn patterns in our target data set. So
that's the theory
behind it. So let's get rid of ease. Let's create a loss function. What are we
going to use? Well,
we're going to use nn dot BCE with logit's loss. And our optimizer is going to be
what? We're
going to keep that as the same as before, torch dot opt in dot SGD. But we have to
be aware that
because we're using a new model, we have to pass in params of model one. These are
the parameters
we want to optimize. And the LR is going to be 0.1. Is that the same LR we use
before learning
rate? 0.1. Oh, potentially that our learning rate may be too big. 0.1. Where do we
create our
optimizer? So we've written a lot of code here. Optimizer. There we go. 0.1. That's
all right.
So we'll keep it at 0.1 just to keep as many things the same as possible. So we're
going to set up
torch dot manual seed 42 to make training as reproducible as possible torch dot
CUDA dot manual
seed 42. Now, as I said before, don't worry too much if your numbers aren't exactly
the same as mine.
The direction is more important, whether it's good or bad direction. So now let's
set up epochs.
We want to train for longer this time as well. So 1000 epochs. This is one of our
three improvements
that we're trying to do. Adding more hidden units, increase the number of layers
and increase the
number of epochs. So we're going to give our model 1000 looks at the data to try
and improve
its patterns. So put data on the target device. We want to write device agnostic
code. And yes,
we've already done this, but we're going to write it out again for practice because
even though we
could functionize a lot of this, it's good while we're in still the foundation
stages to practice
what's going on here, because I want you to be able to do this with your eyes
closed before we
start to functionize it. So put the training data and the testing data to the
target device,
whatever it is, CPU or GPU. And then we're going to, well, what's our song? For an
epoch in range.
Let's loop through the epochs. We're going to start off with training. What do we
do for training? Well,
we set model one to train. And then what's our first step? Well, we have to forward
pass. What's
our outputs of the model? Well, the raw outputs of a model are logits. So model
one, we're going
to pass it the training data. We're going to squeeze it so that we get rid of an
extra one
```

Section 114: Main Topics

Key Topics:

- Remember machine learning has a lot of different names for the same thing
- Of course, this is going to be the training loss
- Now, of course, this is going to be the training accuracy

▶ 📄 Click to view detailed content

```
dimension. If you don't believe me that we would like to get rid of that one
dimension,
try running the code without that dot squeeze. And why pred equals torch dot round.
And torch dot sigmoid, why we're calling sigmoid on our logits to go from logits to
prediction
probabilities to prediction labels. And then what do we do next? Well, we calculate
the loss
slash accuracy to here. And remember, accuracy is optional, but loss is not
optional. So we're
going to pass in here, our loss function is going to take in. I wonder if it'll
work with just straight
up why pred? I don't think it will because we're using we need logits in here. Why
logits and why
train? Because why? Oh, Google collab correcting the wrong thing. We have why
logits because we're
using BCE with logits loss here. So let's keep pushing forward. We want our
accuracy now,
which is our accuracy function. And we're going to pass in the order here, which is
the reverse
of above, a little confusing, but I've kept the evaluation function in the same
order as
scikit loan. Why pred equals y pred? Three, we're going to zero the gradients of
the optimizer,
optimizer zero grad. And you might notice that we've started to pick up the pace a
little.
That is perfectly fine. If I'm typing too fast, you can always slow down the video,
or you could just watch what we're doing and then code it out yourself afterwards,
the code resources will always be available. We're going to take the last backward
and perform back propagation. The only reason we're going faster is because we've
covered
these steps. So anything that we sort of spend time here, we've covered in a
previous video,
optimizer step. And this is where the adjustments to all of our models parameters
are going to take
place to hopefully create a better representation of the data. And then we've got
testing. What's
the first step that we do in testing? Well, we call model one dot a vowel to put it
in evaluation
mode. And because we're making predictions, we're going to turn on torch inference
mode
predictions. I call them predictions. Some other places call it inference.
Remember machine learning has a lot of different names for the same thing.
Forward pass. So we're going to create the test logits here. Equals model one X
test.
And we're going to squeeze them because we won't don't want the extra one
dimension. Just going to
```

add some code cells here so that we have more space and I'm typing in the middle of the screen.
Then I'm going to put in test pred here. How do we get from logits to predictions? Well,
we go torch dot round. And then we go torch dot sigmoid y sigmoid because we're working with a
binary classification problem. And to convert logits from a binary classification problem
to prediction probabilities, we use the sigmoid activation function. And then we're going to
calculate the loss. So how wrong is our model on the test data? So test last equals loss function.
We're going to pass it in the test logits. And then we're going to pass it in Y test for the ideal
labels. And then we're going to also calculate test accuracy. And test accuracy is going to
take in Y true equals Y test. So the test labels and Y pred equals test pred. So the test predictions
test predictions here. And our final step is to print out what's happening. So print out what's
happening. Oh, every tutorial needs a song. If I could, I'd teach everything with song.
Song and dance. So because we're training for 1000 epochs, how about every 100 epochs we print
out something. So print f string, and we're going to write epoch in here. So we know what epoch our
models on. And then we're going to print out the loss. Of course, this is going to be the training
loss. Because the test loss has test at the front of it. And then accuracy here. Now, of course,
this is going to be the training accuracy. We go here. And then we're going to pipe. And we're
going to print out the test loss. And we want the test loss here. We're going to take this to five
decimal places. Again, when we see the printouts of the different values, do not worry too much
about the exact numbers on my screen appearing on your screen, because that is inherent to the
randomness of machine learning. So have we got the direction is more important? Have we got,
we need a percentage sign here, because that's going to be a bit more complete for accuracy.
Have we got any errors here? I don't know. I'm just, we've just all coded this free hand,
right? There's a lot of code going on here. So we're about to train our next model,
which is the biggest model we've built so far in this course, three layers, 10 hidden units on
each layer. Let's see what we've got. Three, two, one, run. Oh, what? What? A thousand epochs,
an extra hidden layer, more hidden units. And we still, our model is still basically a coin toss.
50%. Now, this can't be for real. Let's plot the decision boundary.
Plot the decision boundary. To find out, let's get a bit visual. Plot figure, actually, to prevent us
from writing out all of the plot code, let's just go up here, and we'll copy this. Now, you know,
I'm not the biggest fan of copying code. But for this case, we've already written

```
it. So there's
nothing really new here to cover. And we're going to just change this from model
zero to model one,
because why it's our new model that we just trained. And so behind the scenes, plot
decision
```

Section 115: Main Topics

Key Topics:

- This is the learn pytorch
- io book pytorch workflow fundamentals
- Because right now it seems like it's not learning anything at all

▶ 📄 Click to view detailed content

```
boundary is going to make predictions with the target model on the target data set
and put it
into a nice visual representation for us. Oh, I said nice visual representation.
What does this
look like? We've just got a coin toss on our data set. Our model is just again,
it's trying
to draw a straight line to separate circular data. Now, why is this? Our model is
based on linear,
is our data nonlinear? Hmm, maybe I've revealed a few of my tricks. I've done a
couple of reveals
over the past few videos. But this is still quite annoying. And it can be fairly
annoying
when you're training models and they're not working. So how about we verify that
this model
can learn anything? Because right now it's just basically guessing for our data
set.
So this model looks a lot like the model we built in section 01. Let's go back to
this.
This is the learn pytorch.io book pytorch workflow fundamentals. Where did we
create a model model
building essentials? Where did we build a model? Linear regression model? Yeah,
here. And then
dot linear. But we built this model down here. So all we've changed from 01 to here
is we've added
a couple of layers. The forward computation is quite similar. If this model can
learn something
on a straight line, can this model learn something on a straight line? So that's my
challenge to you
is grab the data set that we created in this previous notebook. So data, you could
just
reproduce this in exact data set. And see if you can write some code to fit the
model that we built
here. This one here on the data set that we created in here. Because I want to
```

verify that
this model can learn anything. Because right now it seems like it's not learning anything at all.
And that's quite frustrating. So give that a shot. And I'll see you in the next video.
Welcome back. In the past few videos, we've tried to build a model to separate the blue from red
dots yet. Our previous efforts have proven futile, but don't worry. We're going to get there. I promise
you we're going to get there. And I may have a little bit of inside information here. But we're
going to build a model to separate these blue dots from red dots, a fundamental classification model.
And we tried a few things in the last couple of videos, such as training for longer, so more epochs.
We added another layer. We increased the hidden units because we learned of a few methods to
improve a model from a model perspective, such as upgrading the hyperparameters, such as number
of layers, more hidden units, fitting for longer, changing the activation functions,
changing the learning rate, we haven't quite done that one yet, and changing the loss function.
One way that I like to troubleshoot problems is I'm going to put a subheading here, 5.1.
We're going to prepare or preparing data to see if our model can fit a straight line.
So one way to troubleshoot, this is my trick for troubleshooting problems, especially neural
networks, but just machine learning in general, to troubleshoot a larger problem is to test out
a smaller problem. And so why is this? Well, because we know that we had something working
in a previous section, so 01, PyTorch, workflow fundamentals, we built a model here that worked.
And if we go right down, we know that this linear model can fit a straight line. So we're going
to replicate a data set to fit a straight line to see if the model that we're building here
can learn anything at all, because right now it seems like it can't. It's just tossing a coin
displayed between our data here, which is not ideal. So let's make some data. But yeah, this is the,
let's create a smaller problem, one that we know that works, and then add more complexity to try
and solve our larger problem. So create some data. This is going to be the same as notebook 01.
And I'm going to set up weight equals 0.7 bias equals 0.3. We're going to move quite quickly
through this because we've seen this in module one, but the overall takeaway from this is we're
going to see if our model works on any kind of problem at all, or do we have something fundamentally
wrong, create data. We're going to call it x regression, because it's a straight line, and we
want it to predict a number rather than a class. So you might be thinking, oh, we might have to change

```
a few things of our model architecture. Well, we'll see that in a second dot
unsqueeze. And we're
going to go on the first dimension here or dim equals one. And why regression,
we're going to use
the linear regression formula as well, wait times x, x regression, that is, because
we're working
with a new data set here, plus the bias. So this is linear regression formula.
Without epsilon. So it's a simplified version of linear regression, but the same
formula that we've seen in a previous section. So now let's check the data. Nothing
we really haven't covered here, but we're going to do a sanity check on it to make
sure that we're dealing with what we're dealing with.
```

Section 116: Main Topics

Key Topics:

- Because it's all about the data and machine learning
- To download it from the course GitHub, and we imported plot predictions from it
- So later on, if you're working with a big machine learning data set, you'd probably start with a smaller portion of that data set first

▶ 📄 Click to view detailed content

```
What we're dealing with is not just a load of garbage. Because it's all about the
data and machine learning. I can't stress to you enough. That's the data explorer's
motto is to visualize, visualize, visualize. Oh, what did we get wrong here?
Unsqueeze. Did you notice that typo? Why didn't you say something? I'm kidding.
There we go. Okay, so we've got 100 samples of x. We've got a different step size
here, but that's all right. Let's have a little bit of fun with this. And we've got
one x-value, which is, you know, a little bit more.
One x value per y value is a very similar data set to what we use before. Now, what
do we do once we have a data set? Well, if we haven't already got training and test
splits, we better make them. So create train and test splits.
And then we're going to go train split. We're going to use 80% equals int 0.8 times
the length of, or we could just put 100 in there.
But we're going to be specific here. And then we're going to go x train regression,
y train regression equals. What are these equal? Well, we're going to go on x
regression.
And we're going to index up to the train split on the x. And then for the y, y
regression, we're going to index up to the train split.
Wonderful. And then we can do the same on the test or creating the test data.
Nothing really new here that we need to discuss. We're creating training and test
sets. What do they do for each of them?
Well, the model is going to hopefully learn patterns in the training data set that
is able to model the testing data set. And we're going to see that in a second.
So if we check the length of each, what do we have? Length x train regression. We
might just check x train x test regression. What do we have here?
And then we're going to go length y train regression. Long variable names here.
```

Excuse me for that. But we want to keep it separate from our already existing x and y data. What values do we have here?
80, 20, 80, 20, beautiful. So 80 training samples to 100 testing samples. That should be enough. Now, because we've got our helper functions file here. And if you don't have this, remember, we wrote some code up here before to where is it?
To download it from the course GitHub, and we imported plot predictions from it. Now, if we have a look at helper functions.py, it contains the plot predictions function that we created in the last section, section 0.1. There we go. Plot predictions.
So we're just running this exact same function here, or we're about to run it. It's going to save us from re typing out all of this. That's the beauty of having a helper functions.py file.
So if we come down here, let's plot our data to visually inspected. Right now, it's just numbers on a page. And we're not going to plot really any predictions because we don't have any predictions yet.
But we'll pass in the train data is equal to X train regression. And then the next one is the train labels, which is equal to Y train regression.
And then we have the test data, which is equal to X test regression. And then we have the test labels. Now, I think this should be labels too. Yeah, there we go. Y test progression might be proven wrong as we try to run this function.
Okay, there we go. So we have some training data and we have some testing data. Now, do you think that our model model one, we have a look what's model one could fit this data.
Does it have the right amount of in and out features? We may have to adjust these slightly. So I'd like you to think about that. Do we have to change the input features to our model for this data set?
And do we have to change the out features of our model for this data set? We'll find out in the next video.
Welcome back. We're currently working through a little side project here, but really the philosophy of what we're doing. We just created a straight line data set because we know that we've built a model in the past back in section 01 to fit a straight line data set.
And why are we doing this? Well, because the model that we've built so far is not fitting or not working on our circular data set here on our classification data set.
And so one way to troubleshoot a larger problem is to test out a smaller problem first. So later on, if you're working with a big machine learning data set, you'd probably start with a smaller portion of that data set first.
Likewise, with a larger machine learning model, instead of starting with a huge model, you'll start with a small model.
So we're taking a step back here to see if our model is going to learn anything at all on a straight line data set so that we can improve it for a non-straight line data set.
And there's another hint. Oh, we're going to cover it in a second. I promise you. But let's see how now we can adjust model one to fit a straight line.
And I should do the question at the end of last video. Do we have to adjust the parameters of model one in any way shape or form to fit this straight line data?
And you may have realized or you may not have that our model one is set up for our classification data, which has two X input features.

Section 117: Main Topics

Key Topics:

- There's only one value per, let's remind ourselves, this is input and output shapes, one of the most fundamental things in machine learning and deep learning
- And of course, the second layer, the number of features here has to line up with the out features of the previous layer
- Of course, we could make this number quite large

▶ 📄 Click to view detailed content

```
Whereas this data, if we go X train regression, how many input features do we have?
We just get the first sample.
There's only one value. Or maybe we get the first 10. There's only one value per,
let's remind ourselves, this is input and output shapes, one of the most
fundamental things in machine learning and deep learning.
And trust me, I still get this wrong all the time. So that's why I'm harping on
about it. We have one feature per one label. So we have to adjust our model
slightly.
We have to change the end features to be one instead of two. The out features can
stay the same because we want one number to come out.
So what we're going to do is code up a little bit different version of model one.
So same architecture as model one. But using NN dot sequential, we're going to do
the faster way of coding a model here.
Let's create model two and NN dot sequential. The only thing that's going to change
is the number of input features.
So this will be the exact same code as model one. And the only difference, as I
said, will be features or in features is one. And then we'll go out features equals
10.
So 10 hidden units in the first layer. And of course, the second layer, the number
of features here has to line up with the out features of the previous layer.
This one's going to output 10 features as well. So we're scaling things up from one
feature to 10 to try and give our model as much of a chance or as many parameters
as possible.
Of course, we could make this number quite large. We could make it a thousand
features if we want. But there is an upper bound on these things.
And I'm going to let you find those in your experience as a machine learning
engineer and a data scientist.
But for now, we're keeping it nice and small. So we can run as many experiments as
possible. Beautiful. Look at that. We've created a sequential model. What happens
with NN dot sequential?
Data goes in here, passes through this layer. Then it passes through this layer.
Then it passes through this layer. And what happens when it goes through the layer?
It triggers the layers forward method, the internal forward method. In the case of
NN dot linear, we've seen it. It's got the linear regression formula.
So if we go NN dot linear, it performs this mathematical operation, the linear
transformation. But we've seen that before. Let's keep pushing forward.
Let's create a loss and an optimizer loss and optimize. We're going to work through
our workflow. So loss function, we have to adjust this slightly.
We're going to use the L1 loss because why we're dealing with a regression problem
here rather than a classification problem. And our optimizer, what can we use for
our optimizer?
How about we bring in just the exact same optimizer SGD that we've been using for
our classification data. So model two dot params or parameters.
```

```
Always get a little bit confused. And we'll give it an LR of 0.1 because that's
what we've been using so far. This is the params here.
So we want our optimizer to optimize our model two parameters here with a learning
rate of 0.1. The learning rate is what?
The amount each parameter will be or the multiplier that will be applied to each
parameter each epoch.
So now let's train the model. Do you think we could do that in this video? I think
we can. So we might just train it on the training data set and then we can evaluate
it on the test data set separately.
So we'll set up both manual seeds, CUDA and because we've set our model to the
device up here. So it should be on the GPU or whatever device you have active.
So set the number of epochs. How many epochs should we set? Well, we set a thousand
before, so we'll keep it at that.
epochs equals a thousand. And now we're getting really good at this sort of stuff
here. Let's put our data. Put the data on the target device.
And I know we've done a lot of the similar steps before, but there's a reason for
that. I've kept all these in here because I'd like you to buy the end of this
course is to sort of know all of this stuff off by heart.
And even if you don't know it all off my heart, because trust me, I don't, you know
where to look.
So X train regression, we're going to send this to device. And then we're going to
go Y train regression, just a reminder or something to get you to think while we're
writing this code.
What would happen if we didn't put our data on the same device as a model? We've
seen that error come up before, but what happens?
Well, I've just kind of given away, haven't you Daniel? Well, that was a great
question. Our code will air off.
Oh, well, don't worry. There's plenty of questions I've been giving you that I
haven't given the answer to yet.
Device a beautiful. We've got a device agnostic code for the model and for the
data. And now let's loop through epochs.
So train. We're going to for epoch in range epochs for an epoch in a range. Do the
forward pass.
Calculate the loss. So Y pred equals model two. This is the forward pass. X train
regression.
It's all going to work out hunky Dory because our model and our data are on the
same device loss equals what we're going to bring in our loss function.
Then we're going to compare the predictions to Y train regression to the Y labels.
What do we do next?
```

Section 118: Main Topics

Key Topics:

- And of course, we could do some testing here
- And of course, we could shorten this by making these a function
- So that means our model must be learning something

▶ 📄 Click to view detailed content

Optimize a zero grad. Optimize a dot zero grad. We're doing all of this with our comments. Look at us go.
Loss backward and what's next? Optimize a step, step, step. And of course, we could do some testing here.
Testing. We'll go model two dot a vowel. And then we'll go with torch dot inference mode.
We'll do the forward pass. We'll create the test predictions equals model two dot X test regression.
And then we'll go the test loss equals loss FN on the test predictions and versus the Y test labels.
Beautiful. Look at that. We've just done an optimization loop, something we spent a whole hour on before, maybe even longer, in about ten lines of code.
And of course, we could shorten this by making these a function. But we're going to see that later on.
I'd rather us give a little bit of practice while this is still a bit fresh. Print out what's happening.
Let's print out what's happening. What should we do? So because we're training for a thousand epochs, I like the idea of printing out something every 100 epochs.
That should be about enough of a step. Epoch. What do we got? We'll put in the epoch here with the F string and then we'll go to loss, which will be loss.
And maybe we'll get the first five of those five decimal places that is. We don't have an accuracy, do we?
Because we're working with regression. And we'll get the test loss out here. And that's going to be.5F as well.
Beautiful. Have we got any mistakes? I don't think we do. We didn't even run this code cell before. We'll just run these three again, see if we got...
Look at that. Oh my goodness. Is our loss... Our loss is going down.
So that means our model must be learning something.
Now, what if we adjusted the learning rate here? I think if we went 0.01 or something, will that do anything?
Oh, yes. Look how low our loss gets on the test data set. But let's confirm that. We've got to make some predictions.
Well, maybe we should do that in the next video. Yeah, this one's getting too long. But how good's that?
We created a straight line data set and we've created a model to fit it. We set up a loss and an optimizer already.
And we put the data on the target device. We trained and we tested so our model must be learning something.
But I'd like you to give a shot at confirming that by using our plot predictions function.
So make some predictions with our trained model. Don't forget to turn on inference mode. And we should see some red dots here fairly close to the green dots on the next plot.
Give that a shot and I'll see you in the next video.
Welcome back. In the last video, we did something very exciting. We solved a smaller problem that's giving us a hint towards our larger problem.
So we know that the model that we've previously been building, model two, has the capacity to learn something.
Now, how did we know that? Well, it's because we created this straight line data set. We replicated the architecture that we used for model one.
Recall that model one didn't work very well on our classification data. But with a little bit of an adjustment such as changing the number of in features.
And not too much different training code except for a different loss function because, well, we use MAE loss with regression data.
And we changed the learning rate slightly because we found that maybe our model could learn a bit better.

```
And again, I'd encourage you to play around with different values of the learning
rate. In fact, anything that we've changed, try and change it yourself and just see
what happens.
That's one of the best ways to learn what goes on with machine learning models.
But we trained for the same number of epochs. We set up device agnostic code. We
did a training and testing loop.
Look at this looks. Oh, my goodness. Well done. And our loss went down.
So, hmm. What does that tell us? Well, it tells us that model two or the specific
architecture has some capacity to learn something.
So we must be missing something. And we're going to get to that in a minute, I
promise you.
But we're just going to confirm that our model has learned something and it's not
just numbers on a page going down by getting visual.
So turn on. We're going to make some predictions and plot them. And you may have
already done this because I issued that challenge at the last of at the end of the
last video.
So turn on evaluation mode. Let's go model two dot eval. And let's make
predictions, which are also known as inference.
And we're going to go with torch dot inference mode inference mode with torch dot
inference mode.
Make some predictions. We're going to save them as why preds and we're going to use
model two and we're going to pass it through ex test regression.
This should all work because we've set up device agnostic code, plot data and
predictions.
To do this, we can of course use our plot predictions function that we imported via
our helper functions dot pi function.
The code for that is just a few cells above if you'd like to check that out.
But let's set up the train data here. Train data parameter, which is x train
regression.
And my goodness. Google collab. I'm already typing fast enough. You don't have to
slow me down by giving me the wrong auto corrects train label equals y train
regression.
And then we're going to pass in our test data equals ex test regression.
```

Section 119: Main Topics

Key Topics:

- That's the advantage of being the host of this machine learning cooking show
- You could get a little bit better, of course, get the red dots on top of the green dots
- Now, this is one of the beautiful things about machine learning

▶ 📄 Click to view detailed content

```
And then we're going to pass in test labels, which is why test regression got too
many variables going on here. My goodness gracious.
We could have done better with naming, but this will do for now is why preds.
And then if we plot this, what does it look like? Oh, no, we got an error.
Now secretly, I kind of knew that that was coming ahead of time. That's the
```

advantage of being the host of this machine learning cooking show. So type error. How do we fix this?
Remember how I asked you in one of the last videos what would happen if our data wasn't on the same device as our model? Well, we get an error, right? But this is a little bit different as well.
We've seen this one before. We've got CUDA device type tensa to NumPy. Where is this coming from? Well, because our plot predictions function uses mapplotlib.
And behind the scenes, mapplotlib references NumPy, which is another numerical computing library. However, NumPy uses a CPU rather than the GPU.
So we have to call dot CPU, this helpful message is telling us, call tensa dot CPU before we use our tensors with NumPy. So let's just call dot CPU on all of our tensor inputs here and see if this solves our problem.
Wonderful. Looks like it does. Oh my goodness. Look at those red dots so close.
Well, okay. So this just confirms our suspicions. What we kind of already knew is that our model did have some capacity to learn.
It's just the data set when we changed the data set it worked. So, hmm. Is it our data that our model can't learn on? Like this circular data, or is the model itself?
Remember, our model is only comprised of linear functions. What is linear? Linear is a straight line, but is our data made of just straight lines?
I think it's got some nonlinearities in there. So the big secret I've been holding back will reveal itself starting from the next video. So if you want a head start of it, I'd go to torch and end.
And if we have a look at the documentation, we've been speaking a lot about linear functions. What are these nonlinear activations? And I'll give you another spoiler.
We've actually seen one of these nonlinear activations throughout this notebook.
So go and check that out. See what you can infer from that. And I'll see you in the next video. Let's get started with nonlinearities. Welcome back.
In the last video, we saw that the model that we've been building has some potential to learn. I mean, look at these predictions. You could get a little bit better, of course, get the red dots on top of the green dots.
But we're just going to leave that the trend is what we're after. Our model has some capacity to learn, except this is straight line data.
And we've been hinting at it a fair bit is that we're using linear functions. And if we look up linear data, what does it look like?
Well, it has a quite a straight line. If we go linear and just search linear, what does this give us? Linear means straight. There we go, straight.
And then what happens if we search for nonlinear? I kind of hinted at this as well.
Nonlinear. Oh, we get some curves. We get curved lines.
So linear functions. Straight. Nonlinear functions. Hmm.
Now, this is one of the beautiful things about machine learning. And I'm not sure about you, but when I was in high school, I kind of learned a concept called line of best fit, or y equals mx plus c, or
y equals mx plus b. And it looks something like this. And then if you wanted to go over these, you use quadratic functions and a whole bunch of other stuff.
But one of the most fundamental things about machine learning is that we build neural networks and deep down neural networks are just a combination.
It could be a large combination of linear functions and nonlinear functions.
So that's why in torch.nn, we have nonlinear activations and we have all these other different types of layers. But essentially, what they're doing deep down is combining straight lines with, if we go back up to our data, non straight lines.
So, of course, our model didn't work before because we've only given it the power to use linear lines. We've only given it the power to use straight lines.
But our data is what? It's curved. Although it's simple, we need nonlinearity to be able to model this data set.
And now, let's say we were building a pizza detection model. So let's look up some images of pizza, one of my favorite foods, images.

```
Pizza, right? So could you model pizza with just straight lines?
You're thinking, Daniel, you can't be serious. A computer vision model doesn't look
for just straight lines in this. And I'd argue that, yes, it does, except we also
add some curved lines in here.
That's the beauty of machine learning. Could you imagine trying to write the rules
of an algorithm to detect that this is a pizza? Maybe you could put in, oh, it's a
curve here.
And if you see red, no, no, no, no. Imagine if you're trying to do a hundred
different foods. Your program would get really large. Instead, we give our machine
learning models, if we come down to the model that we created.
We give our deep learning models the capacity to use linear and nonlinear
functions. We haven't seen any nonlinear layers just yet.
Or maybe we've hinted at some, but that's all right. So we stack these on top of
each other, these layers.
```

Section 120: Main Topics

Key Topics:

- This is going to follow you out throughout all of machine learning and deep learning, nonlinearity
- Or in machine learning terms, an infinite amount, but really it is finite
- By infinite in machine learning terms, this is a technicality

▶ 📄 Click to view detailed content

```
And then the model figures out what patterns in the data it should use, what lines
it should draw to draw patterns to not only pizza, but another food such as sushi.
If we wanted to build a food image classification model, it would do this. The
principle remains the same. So the question I'm going to pose to you, we'll get out
of this, is, we'll come down here.
We've unlocked the missing piece or about to. We're going to cover it over the next
couple of videos, the missing piece of our model.
And this is a big one. This is going to follow you out throughout all of machine
learning and deep learning, nonlinearity.
So the question here is, what patterns could you draw if you were given an infinite
amount of straight and non straight lines?
Or in machine learning terms, an infinite amount, but really it is finite. By
infinite in machine learning terms, this is a technicality.
It could be a million parameters. It could be as we've got probably a hundred
parameters in our model.
So just imagine a large amount of straight and non straight lines, an infinite
amount of linear and nonlinear functions.
You could draw some pretty intricate patterns, couldn't you? And that's what gives
machine learning and especially neural networks the capacity to not only fit a
straight line here, but to separate two different circles.
But also to do crazy things like drive a self-driving car, or at least power the
vision system of a self-driving car.
```

Of course, after that, you need some programming to plan what to actually do with what you see in an image.
But we're getting ahead of ourselves here. Let's now start diving into nonlinearity.
And the whole idea here is combining the power of linear and nonlinear functions.
Straight lines and non straight lines. Our classification data is not comprised of just straight lines. It's circles, so we need nonlinearity here.
So recreating nonlinear data, red and blue circles. We don't need to recreate this, but we're going to do it anyway for completeness.
So let's get a little bit of a practice. Make and plot data. This is so that you can practice the use of nonlinearity on your own.
And that plot little bit dot pie plot as PLT. We're going to go a bit faster here because we've covered this code above.
So import make circles. We're just going to recreate the exact same circle data set that we've created above.
Number of samples. We'll create a thousand. And we're going to create x and y equals what?
Make circles. Pass it in number of samples. Beautiful.
Colab, please. I wonder if I can turn off autocorrect and colab. I'm happy to just see all of my errors in the flesh. See? Look at that. I don't want that.
I want noise like that. Maybe I'll do that in the next video. We're not going to spend time here looking around how to do it.
We can work that out on the fly later. For now, I'm too excited to share with you the power of nonlinearity.
So here, x, we're just going to plot what's going on. We've got two x features and we're going to color it with the flavor of y because we're doing a binary classification.
And we're going to use one of my favorite C maps, which is color map. And we're going to go PLT dot CM for C map and red blue.
What do we get?
Okay, red circle, blue circle. Hey, is it the same color as what's above? I like this color better.
Did we get that right up here?
Oh, my goodness. Look how much code we've written. Yeah, I like the other blue. I'm going to bring this down here.
It's all about aesthetics and machine learning. It's not just numbers on a page, don't you? How could you be so crass? Let's go there.
Okay, that's better color red and blue. That's small lively, isn't it? So now let's convert to train and test.
And then we can start to build a model with nonlinearity. Oh, this is so good.
Okay, convert data to tenses and then to train and test splits. Nothing we haven't covered here before.
So import torch, but it never hurts to practice code, right? Import torch from sklearn dot model selection.
Import train test split so that we can split our red and blue dots randomly. And we're going to turn data into tenses.
And we'll go X equals torch from NumPy and we'll pass in X here. And then we'll change it into type torch dot float.
Why do we do this? Well, because, oh, my goodness, autocorrect. It's getting the best of me here.
You know, watching me live code this stuff and battle with autocorrect. That's what this whole course is.
And we're really teaching pie torch. Am I just battling with Google collab's autocorrect?
We are turning it into torch dot float with a type here because why NumPy's default, which is what makes circles users behind the scenes.
NumPy is actually using a lot of other machine learning libraries, pandas, built on

```
NumPy, scikit learn, does a lot of NumPy.
Matplotlib, NumPy. That's just showing there. What's the word? Is it ubiquitous,
ubiquity? I'm not sure, maybe.
If not, you can correct me. The ubiquity of NumPy.
And test sets, but we're using pie torch to leverage the power of autograd, which
is what powers our gradient descent.
And the fact that it can use GPUs.
So we're creating training test splits here with train test split X Y.
And we're going to go test size equals 0.2. And we're going to set random.
Random state equals 42. And then we'll view our first five samples. Are these going
to be?
Tenses. Fingers crossed. We haven't got an error. Beautiful. We have tenses here.
```

Section 121: Main Topics

Key Topics:

- But PyTorch implements some of the most common layers that you would have as hidden layers
- And of course, if you wanted to, you could look up what is a nonlinear function
- Whereas neural networks and machine learning models can work with numbers that are in hundreds of dimensions, impossible for us humans to visualize, but since computers love numbers, it's a piece of cake to them

▶ 📄 Click to view detailed content

```
Okay. Now we're up to the exciting part. We've got our data set back.
I think it's time to build a model with nonlinearity.
So if you'd like to peek ahead, check out TorchNN again. This is a little bit of a
spoiler.
Go into the nonlinear activation. See if you can find the one that we've already
used. That's your challenge.
Can you find the one we've already used? And go into here and search what is our
nonlinear function.
So give that a go and see what comes up. I'll see you in the next video.
Welcome back. Now put your hand up if you're ready to learn about nonlinearity.
And I know I can't see your hands up, but I better see some hands up or I better
feel some hands up
because my hands up because nonlinearity is a magic piece of the puzzle that we're
about to learn about.
So let's title this section building a model with nonlinearity.
So just to re-emphasize linear equals straight lines and in turn nonlinear equals
non-straight lines.
And I left off the end of the last video, giving you the challenge of checking out
the TorchNN module,
looking for the nonlinear function that we've already used.
Now where would you go to find such a thing and oh, what do we have here? Nonlinear
```

activations.
And there's going to be a fair few things here, but essentially all of the modules within TorchNN
are either some form of layer in a neural network if we recall.
Let's go to a neural network. We've seen the anatomy of a neural network.
Generally you'll have an input layer and then multiple hidden layers and some form of output layer.
Well, these multiple hidden layers can be almost any combination of what's in TorchNN.
And in fact, they can almost be any combination of function you could imagine.
Whether they work or not is another question.
But PyTorch implements some of the most common layers that you would have as hidden layers.
And they might be pooling layers, padding layers, activation functions.
And they all have the same premise. They perform some sort of mathematical operation on an input.
And so if we look into the nonlinear activation functions, you might have find an n dot sigmoid.
Where have we used this before? There's a sigmoid activation function in math terminology.
It takes some input x, performs this operation on it.
And here's what it looks like if we did it on a straight line, but I think we should put this in practice.
And if you want an example, well, there's an example there.
All of the other nonlinear activations have examples as well.
But I'll let you go through all of these in your own time.
Otherwise we're going to be here forever.
And then dot relu is another common function.
We saw that when we looked at the architecture of a classification network.
So with that being said, how about we start to code a classification model with nonlinearity.
And of course, if you wanted to, you could look up what is a nonlinear function.
If you wanted to learn more, nonlinear means the graph is not a straight line.
Oh, beautiful. So that's how I'd learn about nonlinear functions.
But while we're here together, how about we write some code.
So let's go build a model with nonlinear activation functions.
And just one more thing before, just to re-emphasize what we're doing here.
Before we write this code, I've got, I just remembered, I've got a nice slide, which is the question we posed in the previous video, the missing piece, nonlinearity.
But the question I want you to think about is what could you draw if you had an unlimited amount of straight,
in other words, linear, and non-straight, nonlinear line.
So we've seen previously that we can build a model, a linear model to fit some data that's in a straight line, linear data.
But when we're working with nonlinear data, well, we need the power of nonlinear functions.
So this is circular data. And now, this is only a 2D plot, keep in mind there.
Whereas neural networks and machine learning models can work with numbers that are in hundreds of dimensions,
impossible for us humans to visualize, but since computers love numbers, it's a piece of cake to them.
So from torch, import, and then we're going to create our first neural network with nonlinear activations.
This is so exciting. So let's create a class here.
We'll create circle model. We've got circle model V1 already. We're going to create circle model V2.

```
And we'll inherit from an end dot module. And then we'll write the constructor,
which is the init function,
and we'll pass in self here. And then we'll go self, or super sorry, too many S
words.
Dot underscore underscore init, underscore. There we go. So we've got the
constructor here.
And now let's create a layer one, self dot layer one equals just the same as what
we've used before. And then dot linear.
We're going to create this quite similar to the model that we've built before,
except with one added feature.
And we're going to create in features, which is akin to the number of X features
that we have here.
Again, if this was different, if we had three X features, we might change this to
three.
But because we're working with two, we'll leave it as that. We'll keep out features
as 10, so that we have 10 hidden units.
And then we'll go layer two, and then dot linear. Again, these values here are very
customizable because why, because they're hyper parameters.
So let's line up the out features of layer two, and we'll do the same with layer
three.
Because layer three is going to take the outputs of layer two. So it needs in
features of 10.
```

Section 122: Main Topics

Key Topics:

- And then put the learning rate to 0
- But the main points here are have to learning rate of 0

▶ 📄 Click to view detailed content

```
And we want layer three to be the output layer, and we want one number as output,
so we'll set one here.
Now, here's the fun part. We're going to introduce a nonlinear function. We're
going to introduce the relu function.
Now, we've seen sigmoid. Relu is another very common one. It's actually quite
simple.
But let's write it out first, and then dot relu.
So remember, torch dot nn stores a lot of existing nonlinear activation functions,
so that we don't necessarily have to code them ourselves.
However, if we did want to code a relu function, let me show you. It's actually
quite simple.
If we dive into nn dot relu, or relu, however you want to say it, I usually say
relu, applies the rectified linear unit function element wise.
So that means element wise on every element in our input tensor.
And so it stands for rectified linear unit, and here's what it does. Basically, it
takes an input.
If the input is negative, it turns the input to zero, and it leaves the positive
```

inputs how they are.
And so this line is not straight.
Now, you could argue, yeah, well, it's straight here and then straight there, but this is a form of a nonlinear activation function.
So it goes boom, if it was linear, it would just stay straight there like that.
But let's see it in practice. Do you think this is going to improve our model?
Well, let's find out together, hey, forward, we need to implement the forward method.
And here's what we're going to do. Where should we put our nonlinear activation functions?
So I'm just going to put a node here. Relu is a nonlinear activation function.
And remember, wherever I say function, it's just performing some sort of operation on a numerical input.
So we're going to put a nonlinear activation function in between each of our layers.
So let me show you what this looks like, self dot layer three.
We're going to start from the outside in self dot relu, and then we're going to go self dot layer two.
And then we're going to go self dot relu.
And then there's a fair bit going on here, but nothing we can't handle layer one.
And then here's the X.
So what happens is our data goes into layer one, performs a linear operation with an end up linear.
Then we pass the output of layer one to a relu function.
So we, where's relu up here, we turn all of the negative outputs of our model of our of layer one to zero,
and we keep the positives how they are.
And then we do the same here with layer two.
And then finally, the outputs of layer three stay as they are. We've got out features there.
We don't have a relu on the end here, because we're going to pass the outputs to the sigmoid function later on.
And if we really wanted to, we could put self dot sigmoid here equals an end dot sigmoid.
But I'm going to, that's just one way of constructing it.
We're just going to apply the sigmoid function to the logits of our model, because what are the logits, the raw output of our model.
And so let's instantiate our model. This is going to be called model three, which is a little bit confusing, but we're up to model three,
which is circle model V two, and we're going to send that to the target device.
And then let's check model three. What does this look like?
Wonderful. So it doesn't actually show us where the relu's appear, but it just shows us what are the parameters of our circle model V two.
Now, I'd like you to have a think about this. And my challenge to you is to go ahead and see if this model is capable of working on our data, on our circular data.
So we've got the data sets ready. You need to set up some training code.
My challenge to you is write that training code and see if this model works.
But we're going to go through that over the next few videos.
And also, my other challenge to you is to go to the TensorFlow Playground and recreate our neural network here.
You can have two hidden layers. Does this go to 10? Well, it only goes to eight.
We'll keep this at five.
So build something like this. So we've got two layers with five. It's a little bit different to ours because we've got two layers with 10.
And then put the learning rate to 0.01. What do we have? 0.1 with stochastic gradient descent.

```
We've been using 0.1, so we'll leave that. So this is the TensorFlow Playground.
And then change the activation here. Instead of linear, which we've used before,
change it to relu, which is what we're using.
And press play here and see what happens. I'll see you in the next video.
Welcome back. In the last video, I left off leaving the challenge of recreating
this model here.
It's not too difficult to do. We've got two hidden layers and five neurons. We've
got our data set, which looks kind of like ours.
But the main points here are have to learning rate of 0.1, which is what we've been
using.
But to change it from, we've previously used a linear activation to change it from
linear to relu, which is what we've got set up here in the code.
Now, remember, relu is a popular and effective nonlinear activation function.
And we've been discussing that we need nonlinearity to model nonlinear data.
And so that's the crux to what neural networks are.
Artificial neural networks, not to get confused with the brain neural networks, but
who knows?
This might be how they work, too. I don't know. I'm not a neurosurgeon or a
neuroscientist.
```

Section 123: Main Topics

Key Topics:

- And if we change the learning rate, maybe a little lower, let's see what happens
- See, that's the power of changing the learning rate
- So see how the learning rate is much smaller

▶ 📄 Click to view detailed content

```
Artificial neural networks are a large combination of linear.
So this is straight and non-straight nonlinear functions, which are potentially
able to find patterns in data.
And so for our data set, it's quite small. It's just a blue and a red circle.
But this same principle applies for larger data sets and larger models combined
linear and nonlinear functions.
So we've got a few tabs going on here. Let's get rid of some. Let's come back to
here.
Did you try this out? Does it work? Do you think it'll work?
I don't know. Let's find out together. Ready? Three, two, one.
Look at that.
Almost instantly the training loss goes down to zero and the test loss is basically
zero as well. Look at that.
That's amazing. We can stop that there. And if we change the learning rate, maybe a
little lower, let's see what happens.
It takes a little bit longer to get to where it wants to go to.
See, that's the power of changing the learning rate. Let's make it really small.
What happens here?
So that was about 300 epochs. The loss started to go down.
```

If we change it to be really small, oh, we're getting a little bit of a trend.
Is it starting to go down? We're already surpassed the epochs that we had.
So see how the learning rate is much smaller? That means our model is learning much slower.
So this is just a beautiful visual way of demonstrating different values of the learning rate.
We could sit here all day and that might not get to lower, but let's increase it by 10x.
And that was over 1,000 epochs and it's still at about 0.5, let's say.
Oh, we got a better. Oh, we're going faster already.
So not even at 500 or so epochs, we're about 0.4.
That's the power of the learning rate. We'll increase it by another 10x.
We'll reset. Start again. Oh, would you look at that much faster this time.
That is beautiful. Oh, there's nothing better than watching a loss curve go down.
In the world of machine learning, that is. And then we reset that again.
And let's change it right back to what we had. And we get to 0 in basically under 100 epochs.
So that's the power of the learning rate, little visual representation.
Working on learning rates, it's time for us to build an optimizer and a loss function.
So that's right here. We've got our nonlinear model set up loss and optimizer.
You might have already done this because the code, this is code that we've written before,
but we're going to redo it for completeness and practice.
So we want a loss function. We're working with logits here and we're working with binary cross entropy.
So what loss do we use?
Binary cross entropy. Sorry, we're working with a binary classification problem.
Blue dots or red dots, torch dot opt in.
What are some other binary classification problems that you can think of?
We want model three dot parameters.
They're the parameters that we want to optimize this model here.
And we're going to set our LR to 0.1, just like we had in the TensorFlow playground.
Beautiful. So some other binary classification problems I can think of would be email.
Spam or not spam credit cards.
So equals fraud or not fraud.
What else? You might have insurance claims.
Equals who's at fault or not at fault.
If someone puts in a claim speaking about a car crash, whose fault was it?
Was the person submitting the claim? Were they at fault?
Or was the person who was also mentioned in the claim? Are they not at fault?
So there's many more, but they're just some I can think of up the top of my head.
But now let's train our model with nonlinearity.
Oh, we're on a roll here.
Training a model with nonlinearity.
So we've seen that if we introduce a nonlinear activation function within a model, remember this is a linear activation function, and if we train this, the loss doesn't go down.
But if we just adjust this to add a relu in here, we get the loss going down.
So hopefully this replicates with our pure PyTorch code.
So let's do it, hey?
So we're going to create random seeds.
Because we're working with CUDA, we'll introduce the CUDA random seed as well.
Torch.manual seed. Again, don't worry too much if your numbers on your screen aren't exactly what mine are.

```
That's due to the inherent randomness of machine learning.
In fact, stochastic gradient descent stochastic again stands for random.
And we're just setting up the seeds here so that they can be as close as possible.
But the direction is more important.
So if my loss goes down, your loss should also go down on target device.
And then we're going to go Xtrain.
So this is setting up device agnostic code. We've done this before.
But we're going to do it again for completeness.
Just to practice every step of the puzzle. That's what we want to do.
We want to have experience. That's what this course is. It's a momentum builder.
So that when you go to other repos and machine learning projects that use PyTorch,
you can go, oh, does this code set device agnostic code?
What problem are we working on? Is it binary or multi-class classification?
So let's go loop through data.
Again, we've done this before, but we're going to set up the epochs.
Let's do 1000 epochs. Why not?
So we can go for epoch in range epochs.
What do we do here? Well, we want to train. So this is training code.
We set our model model three dot train.
And I want you to start to think about how could we functionalize this training
code?
We're going to start to move towards that in a future video.
```

Section 124: Main Topics

Key Topics:

- That's from my unofficial pytorch song
- And this one is a little bit backwards compared to pytorch, but we pass in the y training labels first
- Then we're going to perform back propagation pytorch is going to take care of that for us by calling loss backwards

▶ 📄 Click to view detailed content

```
So one is forward pass. We've got the logits. Why the logits?
Well, because the raw output of our model without any activation functions towards
the final layer.
Classified as logits or called logits.
And then we create y-pred as in prediction labels by rounding the output of torch
dot sigmoid of the logits.
So this is going to take us from logits to prediction probabilities to prediction
labels.
And then we can go to, which is calculate the loss.
That's from my unofficial pytorch song. Calculate the last.
We go loss equals loss FN y logits.
Because remember, we've got BCE with logits loss and takes in logits as first
input.
```

And that's going to calculate the loss between our models, logits and the y training labels.
And we will go here, we'll calculate accuracy using our accuracy function.
And this one is a little bit backwards compared to pytorch, but we pass in the y training labels first.
But it's constructed this way because it's in the same style as scikit line.
Three, we go optimizer zero grad. We zero the gradients of the optimizer so that it can start from fresh.
Calculating the ideal gradients every epoch.
So it's going to reset every epoch, which is fine.
Then we're going to perform back propagation pytorch is going to take care of that for us by calling loss backwards.
And then we will perform gradient descent. So step the optimizer to see how we should improve our model parameters.
So optimizer dot step.
Oh, and I want to show you speaking of model parameters. Let's check our model three dot state dig.
So the relu activation function actually doesn't have any parameters.
So you'll notice here, we've got weight, we've got bias of layer one, layer two, and a layer three.
So the relu function here doesn't have any parameters to optimize. If we go nn dot relu.
Does it say what it implements? There we go.
So it's just the maximum of zero or x. So it takes the input and takes the max of zero or x.
And so when it takes the max of zero or x, if it's a negative number, zero is going to be higher than a negative number.
So that's why it zeroes all of the negative inputs.
And then it leaves the positive inputs how they are because the max of a positive input versus zero is the positive input.
So this has no parameters to optimize. That's why it's so effective because you think about it.
Every parameter in our model needs some little bit of computation to adjust.
And so the more parameters we add to our model, the more compute that is required.
So generally, the kind of trade-off in machine learning is that, yes, more parameters have more of an ability to learn, but you need more compute.
So let's go model three dot a vowel. And we're going to go with torch dot inference mode.
If I could spell inference, that'd be fantastic. We're going to do what? We're going to do the forward pass.
So test logits equals model three on the test data.
And then we're going to calculate the test pred labels by calling torch dot round on torch dot sigmoid on the test logits.
And then we can calculate the test loss. How do we do that?
And then we can also calculate the test accuracy. I'm just going to give myself some more space here.
So I can code in the middle of the screen equals accuracy function on what we're going to pass in y true equals y test.
We're going to pass in y true equals y test. And then we will pass in y pred equals test pred.
Beautiful. A final step here is to print out what's happening.
Now, this will be very important because one, it's fun to know what your model is doing.
And two, if our model does actually learn, I'd like to see the loss values go down and the accuracy values go up.
As I said, there's nothing much more beautiful in the world of machine learning than watching a loss function go down or a loss value go down and watching a loss

```
curve go down.
So let's print out the current epoch and then we'll print out the loss, which will
just be the training loss.
And we'll take that to four decimal places. And then we'll go accuracy here.
And this will be a and we'll take this to two decimal places and we'll put a little
percentage sign there and then we'll break it up by putting in the test loss here
and we'll put in the test loss.
Because remember our model learns patterns on the training data set and then
evaluates those patterns on the test data set.
So, and we'll pass in test act here and no doubt there might be an error or two
within all of this code, but we're going to try and run this because we've seen
this code before, but I think we're ready.
We're training our first model here with non-linearities built into the model.
You ready? Three, two, one, let's go.
Oh, of course. Module torch CUDA has no attribute manuals are just a typo standard
man you out.
There we go. Have to sound that out.
Another one. What do we get wrong here? Oh, target size must be same as input size.
Where did it mess up here?
What do we get wrong? Test loss, test logits on Y test. Hmm.
So these two aren't matching up. Model three X test and Y test. What's the size of?
So let's do some troubleshooting on the fly. Hey, not everything always works out
as you want.
So length of X test, we've got a shape issue here. Remember how I said one of the
most common issues in deep learning is a shape issue?
```

Section 125: Main Topics

Key Topics:

- You could play around with the learning rate, regularization
- We could functionalize this, of course, Model 3 and then pass in X test
- I actually really liked that we did because then we got to troubleshoot a shape error on the fly because that's one of the most common issues you're going to come across in deep learning

▶ 📄 Click to view detailed content

```
We've got the same shape here.
Let's check test logits dot shape and Y test dot shape. We'll print this out.
So 200. Oh, here's what we have to do. That's what we missed dot squeeze. Oh, see
how I've been hinting at the fact that we needed to call dot squeeze.
So this is where the discrepancy is. Our test logits dot shape. We've got an extra
dimension here.
And what are we getting here? A value error on the target size, which is a shape
mismatch.
So we've got target size 200 must be the same input size as torch size 201.
So did we squeeze this? Oh, that's why the training worked. Okay, so we've missed
```

this.
Let's just get rid of this. So we're getting rid of the extra one dimension by using squeeze, which is the one dimension here.
We should have everything lined up. There we go. Okay. Look at that. Yes.
Now accuracy has gone up, albeit not by too much. It's still not perfect.
So really we'd like this to be towards 100% lost to be lower. But I feel like we've got a better performing model. Don't you?
Now that is the power of non linearity. All we did was we added in a relu layer or just two of them.
Relu here, relu here. But what did we do? We gave our model the power of straight lines. Oh, straight linear of straight lines and non straight lines.
So it can potentially draw a line to separate these circles.
So in the next video, let's draw a line, plot our model decision boundary using our function and see if it really did learn anything.
I'll see you there.
Welcome back. In the last video, we trained our first model, and as you can tell, I've got the biggest smile on my face,
but we trained our first model that harnesses both the power of straight lines and non straight lines or linear functions and non linear functions.
And by the 1000th epoch, we look like we're getting a bit better results than just pure guessing, which is 50%.
Because we have 500 samples of red dots and 500 samples of blue dots. So we have evenly balanced classes.
Now, we've seen that if we added a relu activation function with a data set similar to ours with a TensorFlow playground, the model starts to fit.
But it doesn't work with just linear. There's a few other activation functions that you could play around with here.
You could play around with the learning rate, regularization. If you're not sure what that is, I'll leave that as extra curriculum to look up.
But we're going to retire the TensorFlow program for now because we're going to go back to writing code.
So let's get out of that. Let's get out of that. We now have to evaluate our model because right now it's just numbers on a page.
So let's write down here 6.4. What do we like to do to evaluate things? It's visualize, visualize, visualize.
So evaluating a model trained with nonlinear activation functions.
And we also discussed the point that neural networks are really just a big combination of linear and nonlinear functions trying to draw patterns in data.
So with that being said, let's make some predictions with our Model 3, our most recently trained model.
We'll put it into a Val mode and then we'll set up inference mode.
And then we'll go yprads equals torch dot round and then torch dot sigmoid.
We could functionalize this, of course, Model 3 and then pass in X test.
And you know what? We're going to squeeze these here because we ran into some troubles in the previous video.
I actually really liked that we did because then we got to troubleshoot a shape error on the fly because that's one of the most common issues you're going to come across in deep learning.
So yprads, let's check them out and then let's check out y test.
You want y test 10.
So remember, when we're evaluating predictions, we want them to be in the same format as our original labels.
We want to compare apples to apples.
And if we compare the format here, do these two things look the same?
Yes, they do. They're both on CUDA and they're both floats.
We can see that it's got this one wrong.
Whereas the other ones look pretty good. Hmm, this might look pretty good if we

```
visualize it.
So now let's, you might have already done this because I issued the challenge of
plotting the decision boundaries.
Plot decision boundaries and let's go PLT dot figure and we're going to set up the
fig size to equal 12.6 because, again, one of the advantages of hosting a machine
learning cooking show is that you can code ahead of time.
And then we can go PLT dot title is train.
And then we're going to call our plot decision boundary function, which we've seen
before.
Plot decision boundary.
And we're going to pass this one in.
We could do model three, but we could also pass it in our older models to model one
that doesn't use it on the reality.
In fact, I reckon that'll be a great comparison.
So we'll also create another plot here for the test data and this will be on index
number two.
So remember, subplot is a number of rows, number of columns, index where the plot
appears.
We'll give this one a title.
Plot dot title.
This will be test and Google Colab.
I didn't want that.
As I said, this course is also a battle between me and Google Colab's autocorrect.
So we're going model three and we'll pass in the test data here.
And behind the scenes, our plot decision boundary function will create a beautiful
graphic for us,
```

Section 126: Main Topics

Key Topics:

- Maybe you lower the learning rate because right now we've got 0
- As I potentially add more layers, I maybe increase the number of hidden units, and then if we needed to fit for longer and maybe lower the learning rate to 0
- So it looks like PyTorch's default data type for integers is in 64

▶ 📄 Click to view detailed content

```
perform some predictions on the X, the features input, and then we'll compare them
with the Y values.
Let's see what's going on here.
Oh, look at that.
Yes, our first nonlinear model.
Okay, it's not perfect, but it is certainly much better than the models that we had
before.
Look at this.
Model one has no linearity.
Model one equals no nonlinearity.
```

I've got double negative there.
Whereas model three equals has nonlinearity.
So do you see the power of nonlinearity or better yet the power of linearity or linear straight lines with non straight lines?
So I feel like we could do better than this, though.
Here's your challenge is to can you improve model three to do better?
What did we get?
79% accuracy to do better than 80% accuracy on the test data.
I think you can.
So that's the challenge.
And if you're looking for hints on how to do so, where can you look?
Well, we've covered this improving a model.
So maybe you add some more layers, maybe you add more hidden units.
Maybe you fit for longer.
Maybe you if you add more layers, you put a relio activation function on top of those as well.
Maybe you lower the learning rate because right now we've got 0.1.
So give this a shot, try and improve it.
I think you can do it.
But we're going to push forward.
That's going to be your challenge for some extra curriculum.
I think in the next section, we've seen our nonlinear activation functions in action.
Let's write some code to replicate them.
I'll see you there.
Welcome back.
In the last video, I left off with the challenge of improving model three to do better than
80% accuracy on the test data.
I hope you gave it a shot.
But here are some of the things I would have done.
As I potentially add more layers, I maybe increase the number of hidden units, and then if we needed to fit for longer and maybe lower the learning rate to 0.01.
But I'll leave that for you to explore because that's the motto of the data scientists, right?
Is to experiment, experiment, experiment.
So let's go in here.
We've seen our nonlinear activation functions in practice.
Let's replicate them.
So replicating nonlinear activation functions.
And remember neural networks rather than us telling the model what to learn.
We give it the tools to discover patterns in data.
And it tries to figure out the best patterns on its own.
And what are these tools?
That's right down here.
We've seen this in action.
And these tools are linear and nonlinear functions.
So a neural network is a big stack of linear and nonlinear functions.
For us, we've only got about four layers or so, four or five layers.
But as I said, other networks can get much larger.
But the premise remains.
Some form of linear and nonlinear manipulation of the data.
So let's get out of this.
Let's make our workspace a little bit more cleaner.
Replicating nonlinear activation functions.
So let's create a tensor to start with.
Everything starts from the tensor.

And we'll go A equals torch A range.
And we're going to create a range from negative 10 to 10 with a step of one.
And we can set the D type here to equal torch dot float 32.
But we don't actually need to.
That's going to be the default.
So if we set A here, A dot D type.
Then we've got torch float 32 and I'm pretty sure if we've got rid of that.
Oh, we've got torch in 64.
Why is that happening?
Well, let's check out A.
Oh, it's because we've got integers as our values because we have a step as one.
If we turn this into a float, what's going to happen?
We get float 32.
But we'll keep it.
Otherwise, this is going to be what?
About a hundred numbers?
Yeah, no, that's too many.
Let's keep it at negative 10 to 10 and we'll set the D type here to torch float 32.
Beautiful.
So it looks like PyTorch's default data type for integers is in 64.
But we're going to work with float 32 because float 32, if our data wasn't float 32 with
the functions we're about to create, we might run into some errors.
So let's visualize this data.
I want you to guess, is this a straight line or non-straight line?
You've got three seconds.
One, two, three.
Straight line.
There we go.
We've got negative 10 to positive 10 up here or nine.
Close enough.
And so how would we turn this straight line?
If it's a straight line, it's linear.
How would we perform the relu activation function on this?
Now, we could of course call torch relu on A.
Actually, let's in fact just plot this.
PLT dot plot on torch relu.
What does this look like?
Boom, there we go.
But we want to replicate the relu function.
So let's go nn dot relu.
What does it do?
We've seen this before.
So we need the max.
We need to return based on an input.
We need the max of zero and x.
So let's give it a shot.
We'll come here.
Again, we need more space.
There can never be enough code space here.
I like writing lots of code.
I don't know about you.
But let's go relu.
We'll take an input x, which will be some form of tensor.
And we'll go return torch dot maximum.
I think you could just do torch dot max.
But we'll try maximum.

```
Torch dot tensor zero.
So the maximum is going to return the max between whatever this is.
One option and whatever the other option is.
So inputs must be tensors.
So maybe we could just give a type hint here that this is torch dot tensor.
```

Section 127: Main Topics

Key Topics:

- Of course, you could do more complicated activation functions or layers and whatnot
- But it's a lot easier to use PyTorch's layers because we're building neural networks here like Lego bricks, stacking together these layers in some way, shape, or form
- And because they're a part of PyTorch, we know that they've been error-tested and they compute as fast as possible behind the scenes and use GPU and get a whole bunch of benefits

▶ 📄 Click to view detailed content

```
And this should return a tensor too.
Return torch dot tensor.
Beautiful.
You're ready to try it out.
Let's see what our relu function does.
Relu A.
Wonderful.
It looks like we got quite a similar output to before.
Here's our original A.
So we've got negative numbers.
There we go.
So recall that the relu activation function turns all negative numbers into zero
because it takes the maximum between zero and the input.
And if the input's negative, well then zero is bigger than it.
And it leaves all of the positive values as they are.
So that's the beauty of relu.
Quite simple, but very effective.
So let's plot relu activation function.
Our custom one.
We will go PLT dot plot.
We'll call our relu function on A.
Let's see what this looks like.
Oh, look at us go.
Well done.
Just the exact same as the torch relu function.
```

Easy as that.
And what's another nonlinear activation function that we've used before?
Well, I believe one of them is if we go down to here, what did we say before?
Sigmoid.
Where is that?
Where are you, Sigmoid?
Here we go.
Hello, Sigmoid.
Oh, this has got a little bit more going on here.
One over one plus exponential of negative x.
So Sigmoid or this little symbol for Sigmoid of x, which is an input.
We get this.
So let's try and replicate this.
I might just bring this one in here.
Right now, let's do the same for Sigmoid.
So what do we have here?
Well, we want to create a custom Sigmoid.
And we want to have some sort of input, x.
And we want to return one divided by, do we have the function in Sigmoid?
One divided by one plus exponential.
One plus torch dot exp for exponential on negative x.
And we might put the bottom side in brackets so that it does that operation.
I reckon that looks all right to me.
So one divided by one plus torch exponential of negative x.
Do we have that?
Yes, we do.
Well, there's only one real way to find out.
Let's plot the torch version of Sigmoid.
Torch dot Sigmoid and we'll pass in x.
See what happens.
And then, oh, we have a.
My bad.
A is our tensor.
What do we get?
We get a curved line.
Wonderful.
And then we go plt dot plot.
And we're going to use our Sigmoid function on a.
Did we replicate torch's Sigmoid function?
Yes, we did.
Ooh, now.
See, this is what's happening behind the scenes with our neural networks.
Of course, you could do more complicated activation functions or layers and whatnot.
And you can try to replicate them.
In fact, that's a great exercise to try and do.
But we've essentially across the videos and the sections that we've done, we've replicated our linear layer.
And we've replicated the relu.
So we've actually built this model from scratch, or we could if we really wanted to.
But it's a lot easier to use PyTorch's layers because we're building neural networks here like Lego bricks,
stacking together these layers in some way, shape, or form.
And because they're a part of PyTorch, we know that they've been error-tested and they compute as fast as possible
behind the scenes and use GPU and get a whole bunch of benefits.

```
PyTorch offers a lot of benefits by using these layers rather than writing them
ourselves.
And so this is what our model is doing.
It's literally like to learn these values and decrease the loss function and
increase the accuracy.
It's combining linear layers and nonlinear layers or nonlinear functions.
Where's our relu function here?
A relu function like this behind the scenes.
So just combining linear and nonlinear functions to fit a data set.
And that premise remains even on our small data set and on very large data sets and
very large models.
So with that being said, I think it's time for us to push on.
We've covered a fair bit of code here.
But we've worked on a binary classification problem.
Have we worked on a multi-class classification problem yet?
Do we have that here? Where's my fun graphic?
We have multi-class classification.
I think that's what we cover next.
We're going to put together all of the steps in our workflow that we've covered for
binary classification.
But now let's move on to a multi-class classification problem.
If you're with me, I'll see you in the next video.
Welcome back.
In the last few videos we've been harnessing the power of nonlinearity.
Specifically non-straight line functions and we replicated some here.
And we learned that a neural network combines linear and nonlinear functions to
find patterns in data.
And for our simple red versus blue dots, once we added a little bit of
nonlinearity,
we found the secret source of to start separating our blue and red dots.
And I also issued you the challenge to try and improve this and I think you can do
it.
So hopefully you've given that a go.
But now let's keep pushing forward.
We're going to reiterate over basically everything that we've done,
except this time from the point of view of a multi-class classification problem.
So I believe we're up to section eight, putting it all together with a multi-class
classification problem.
Beautiful.
And recall the difference between binary classification equals one thing or another
such as cat versus dog.
If you were building a cat versus dog image classifier, spam versus not spam for
say emails that were spam or not spam or
even internet posts on Facebook or Twitter or one of the other internet services.
And then fraud or not fraud for credit card transactions.
And then multi-class classification is more than one thing or another.
So we could have cat versus dog versus chicken.
So I think we've got all the skills to do this.
```

Section 128: Main Topics

Key Topics:

- And of course, torch
- Of course, you can set it whatever number you want, but I like 42
- Oh, and we need a comma here, of course

▶ 📄 Click to view detailed content

```
Our architecture might be a little bit different for a multi-class classification
problem.
But we've got so many building blocks now.
It's not funny.
Let's clean up this and we'll add some more code cells and just to reiterate.
So we've gone over nonlinearity.
The question is what could you draw if you had an unlimited amount of straight
linear and non-straight,
nonlinear lines, I believe you could draw some pretty intricate patterns.
And that is what our neural networks are doing behind the scenes.
And so we also learned that if we wanted to just replicate some of these nonlinear
functions,
some of the ones that we've used before, we could create a range.
Linear activation is just the line itself.
And then if we wanted to do sigmoid, we get this curl here.
And then if we wanted to do relu, well, we saw how to replicate the relu function
as one.
These both are nonlinear.
And of course, torch.nn has far more nonlinear activations where they came from
just as it has far more different layers.
And you'll get used to these with practice.
And that's what we're doing here.
So let's go back to the keynote.
So this is what we're going to be working on.
Multi-class classification.
So there's one of the big differences here.
We use the softmax activation function versus sigmoid.
There's another big difference here.
Instead of binary cross entropy, we use just cross entropy.
But I think most of it's going to stay the same.
We're going to see this in action in a second.
But let's just describe our problem space.
Just to go visual, we've covered a fair bit here.
Well done, everyone.
So binary versus multi-class classification.
Binary one thing or another.
Zero or one.
Multi-class could be three things.
Could be a thousand things.
Could be 5,000 things.
Could be 25 things.
So more than one thing or another.
But that's the basic premise we're going to go with.
Let's create some data, hey?
8.1.
Creating a 20 multi-class data set.
And so to create our data set, we're going to import our dependencies.
We're going to re-import torch, even though we already have it.
```

Just for a little bit of completeness.
And we're going to go map plotlib.
So we can plot, as always, we like to get visual where we can.
Visualize, visualize, visualize.
We're going to import from scikitlearn.datasets.
Let's get make blobs.
Now, where would I get this from?
SKlearn.datasets.
What do we get?
20 data sets.
Do we have classification?
20 data sets.
Do we have blobs?
If we just go make scikitlearn.
Classification data sets.
What do we get?
Here's one option.
There's also make blobs.
Beautiful.
Make blobs.
This is a code for that.
So let's just copy this in here.
And make blobs.
We're going to see this in action anyway.
Make blobs.
As you might have guessed, it makes some blobs for us.
I like blobs.
It's a fun word to say.
Blobs.
So we want train test split because we want to make a data set and then we want to split
it into train and test.
Let's set the number of hyper parameters.
So set the hyper parameters for data creation.
Now I got these from the documentation here.
Number of samples.
How many blobs do we want?
How many features do we want?
So say, for example, we wanted two different classes.
That would be binary classification.
Say, for example, you wanted 10 classes.
You could set this to 10.
And we're going to see what the others are in practice.
But if you want to read through them, you can well and truly do that.
So let's set up.
We want num classes.
Let's double what we've been working with.
We've been working with two classes, red dots or blue dots.
Let's step it up a notch.
We'll go to four classes.
Watch out, everyone.
And we're going to go number of features will be two.
So we have the same number of features.
And then the random seed is going to be 42.
You might be wondering why these are capitalized.
Well, generally, if we do have some hyper parameters that we say set at the start
of a notebook,

```
you'll find it's quite common for people to write them as capital letters just to
say
that, hey, these are some settings that you can change.
You don't have to, but I'm just going to introduce that anyway because you might
stumble upon it yourself.
So create multi-class data.
We're going to use the make blobs function here.
So we're going to create some x blobs, some feature blobs and some label blobs.
Let's see what these look like in a second.
I know I'm just saying blobs a lot.
But we pass in here, none samples.
How many do we want?
Let's create a thousand as well.
That could really be a hyper parameter, but we'll just leave that how it is for
now.
Number of features is going to be num features.
Centres equals num classes.
So we're going to create four classes because we've set up num classes equal to
four.
And then we're going to go center standard deviation.
We'll give them a little shake up, add a little bit of randomness in here.
Give the clusters a little shake up.
We'll mix them up a bit.
Make it a bit hard for our model.
But we'll see what this does in a second.
Random state equals random seed, which is our favorite random seed 42.
Of course, you can set it whatever number you want, but I like 42.
Oh, and we need a comma here, of course.
Beautiful.
Now, what do we have to do here?
Well, because we're using scikit-learn and scikit-learn leverages NumPy.
So let's turn our data into tenses.
Turn data into tenses.
And how do we do that?
```

Section 129: Main Topics

Key Topics:

- And we'll turn it into torch dot float because remember NumPy defaults as float 64, whereas PyTorch likes float 32
- Oh, of course we got something wrong
- Building a multi-class classification model in PyTorch

▶ 📄 Click to view detailed content

```
Well, we grab x blob and we call torch from NumPy from NumPy.
If I could type, that would be fantastic.
```

That's all right.
We're doing pretty well today.
Haven't made too many typos.
We did make a few in a couple of videos before, but hey.
I'm only human.
So we're going to torch from NumPy and we're going to pass in the y blob.
And we'll turn it into torch dot float because remember NumPy defaults as float 64, whereas
PyTorch likes float 32.
So split into training and test.
And we're going to create x blob train y or x test.
x blob test.
We'll keep the blob nomenclature here.
y blob train and y blob test.
And here's again where we're going to leverage the train test split function from scikit-learn.
So thank you for that scikit-learn.
x blob and we're going to pass the y blob.
So features, labels, x is the features, y are the labels.
And a test size, we've been using a test size of 20%.
That means 80% of the data will be for the training data.
That's a fair enough split with our data set.
And we're going to set the random seed to random seed because generally normally train test split is random,
but because we want some reproducibility here, we're passing random seeds.
Finally, we need to get visual.
So let's plot the data.
Right now we've got a whole bunch of code and a whole bunch of talking, but not too much visuals going on.
So we'll write down here, visualize, visualize, visualize.
And we can call in plot.figure.
What size do we want?
I'm going to use my favorite hand in poker, which is 10-7, because it's generally worked out to be a good plot size.
In my experience, anyway, we'll go x blob.
And we want the zero index here, and then we'll grab x blob as well.
And you might notice that we're visualizing the whole data set here.
That's perfectly fine.
We could visualize, train and test separately if we really wanted to, but I'll leave that as a level challenge to you.
And we're going to go red, yellow, blue.
Wonderful.
What do we get wrong?
Oh, of course we got something wrong.
Santa STD, did we spell center wrong?
Cluster STD.
That's what I missed.
So, cluster STD.
Standard deviation.
What do we get wrong?
Random seed.
Oh, this needs to be random state.
Oh, another typo.
You know what?
Just as I said, I wasn't getting too many typos.
I'll get three.
There we go.

Look at that.
Our first multi-class classification data set.
So if we set this to zero, what does it do to our clusters?
Let's take note of what's going on here, particularly the space between all of the dots.
Now, if we set this cluster STD to zero, what happens?
We get dots that are really just, look at that.
That's too easy.
Let's mix it up, all right?
Now, you can pick whatever value you want here.
I'm going to use 1.5, because now we need to build a model that's going to draw some lines between these four colors.
Two axes, four different classes.
But it's not going to be perfect because we've got some red dots that are basically in the blue dots.
And so, what's our next step?
Well, we've got some data ready.
It's now time to build a model.
So, I'll see you in the next video.
Let's build our first multi-class classification model.
Welcome back.
In the last video, we created our multi-class classification data set,
using scikit-learn's make-blobs function.
And now, why are we doing this?
Well, because we're going to put all of what we've covered so far together.
But instead of using binary classification or working with binary classification data,
we're going to do it with multi-class classification data.
So, with that being said, let's get into building our multi-class classification model.
So, we'll create a little heading here.
Building a multi-class classification model in PyTorch.
And now, I want you to have a think about this.
We spent the last few videos covering non-linearity.
Does this data set need non-linearity?
As in, could we separate this data set with pure straight lines?
Or do we need some non-straight lines as well?
Have a think about that.
It's okay if you're not sure, we're going to be building a model to fit this data anyway,
or draw patterns in this data anyway.
And now, before we get into coding a model,
so for multi-class classification, we've got this.
For the input layer shape, we need to define the in features.
So, how many in features do we have for the hidden layers?
Well, we could set this to whatever we want, but we're going to keep it nice and simple for now.
For the number of neurons per hidden layer, again, this could be almost whatever we want,
but because we're working with a relatively small data set,
we've only got four different classes, we've only got a thousand data points,
we'll keep it small as well, but you could change this.
Remember, you can change any of these because they're hyper parameters.
For the output layer shape, well, how many output features do we want?
We need one per class, how many classes do we have?
We have four clusters of different dots here, so we'll need four output features.
And then if we go back, we have an output activation of softmax, we haven't seen

```
that yet,
and then we have a loss function, rather than binary cross entropy, we have cross
entropy.
And then optimizer as well is the same as binary classification, two of the most
common
are SGDs, stochastic gradient descent, or the atom optimizer,
but of course, the torch.optim package has many different options as well.
So let's push forward and create our first multi-class classification model.
```

Section 130: Main Topics

Key Topics:

- Now, of course, if you don't, you can go change runtime type, select GPU here, that will restart the runtime, you'll have to run all of the code that's before this cell as well, but I'm going to be using a GPU
- So I'm going to call my class here blob model, and it's going to, of course, inherit from nn
- And then, of course, we've got output features, which is also an int

▶ 📄 Click to view detailed content

```
First, we're going to create, we're going to get into the habit of creating
device agnostic code, and we'll set the device here, equals CUDA,
nothing we haven't seen before, but again, we're doing this to put it all together,
so that we have a lot of practice.
Is available, else CPU, and let's go device.
So we should have a GPU available, beautiful CUDA.
Now, of course, if you don't, you can go change runtime type, select GPU here,
that will restart the runtime, you'll have to run all of the code that's before
this cell as well,
but I'm going to be using a GPU.
You don't necessarily need one because our data set's quite small,
and our models aren't going to be very large, but we set this up so we have device
agnostic code.
And so let's build a multi-class classification model.
Look at us go, just covering all of the foundations of classification in general
here,
and we now know that we can combine linear and non-linear functions to create
neural networks that can find patterns in almost any kind of data.
So I'm going to call my class here blob model, and it's going to, of course,
inherit from nn.module, and we're going to upgrade our class here.
We're going to take some inputs here, and I'll show you how to do this.
If you're familiar with Python classes, you would have already done stuff like
this,
but we're going to set some parameters for our models,
because as you write more and more complex classes, you'll want to take inputs
```

here.
And I'm going to pre-build the, or pre-set the hidden units parameter to eight.
Because I've decided, you know what, I'm going to start off with eight hidden units,
and if I wanted to change this to 128, I could.
But in the constructor here, we've got some options.
So we have input features.
We're going to set these programmatically as inputs to our class when we instantiate it.
The same with output features as well.
And so here, we're going to call self.
Oh, no, super.
Sorry.
I always get this mixed up dot init.
And underscore underscore.
Beautiful.
So we could do a doc string here as well.
So let's write in this.
Initializes multi-class classification.
If I could spell class e-fication model.
Oh, this is great.
And then we have some arcs here.
This is just a standard way of writing doc strings.
If you want to find out, this is Google Python doc string guide.
There we go.
Google Python style guide.
This is where I get mine from.
You can scroll through this.
This is just a way to write Python code.
Yeah, there we go.
So we've got a little sentence saying what's going on.
We've got arcs.
We've got returns and we've got errors if something's going on.
So I highly recommend checking that out.
Just a little tidbit.
So this is if someone was to use our class later on.
They know what the input features are.
Input features, which is an int, which is number of input features to the model.
And then, of course, we've got output features, which is also an int.
Which is number of output features of the model.
And we've got the red line here is telling us we've got something wrong, but that's okay.
And then the hidden features.
Oh, well, this is number of output classes for the case of multi-class classification.
And then the hidden units.
Int and then number of hidden units between layers and then the default is eight.
Beautiful.
And then under that, we'll just do that.
Is that going to fix itself?
Yeah, there we go.
We could put in what it returns.
Returns, whatever it returns.
And then an example use case, but I'll leave that for you to fill out.
If you like.
So let's instantiate some things here.
What we might do is write self dot linear layer stack.

```
Self dot linear layer stack.
And we will set this as nn dot sequential.
Ooh, we haven't seen this before.
But we're just going to look at a different way of writing a model here.
Previously, when we created a model, what did we do?
Well, we instantiated each layer as its own parameter here.
And then we called on them one by one, but we did it in a straightforward fashion.
So that's why we're going to use sequential here to just step through our layers.
We're not doing anything too fancy, so we'll just set up a sequential stack of
layers here.
And recall that sequential just steps through, passes the data through each one of
these layers one by one.
And because we've set up the parameters up here, input features can equal to input
features.
And output features, what is this going to be?
Is this going to be output features or is this going to be hidden units?
It's going to be hidden units because it's not the final layer.
We want the final layer to output our output features.
So input features, this will be hidden units because remember the subsequent layer
needs to line up with the previous layer.
Output features, we're going to create another one that outputs hidden units.
And then we'll go in n.linear in features equals hidden units because it takes the
output features of the previous layer.
So as you see here, the output features of this feeds into here.
The output features of this feeds into here.
And then finally, this is going to be our final layer.
We'll do three layers.
Output features equals output features.
Wonderful. So how do we know the values of each of these?
Well, let's have a look at xtrain.shape and ytrain.shape.
So in the case of x, we have two input features.
```

Section 131: Main Topics

Key Topics:

- And then, of course, we're going to send this to device
- And, of course, a training loop
- Where do we find loss functions in PyTorch

▶ 📄 Click to view detailed content

```
And in the case of y, well, this is a little confusing as well because y is a
scalar.
But what do you think the values for y are going to be?
Well, let's go NP. Or is there torch.unique? I'm not sure. Let's find out together,
hey?
Torch unique.
Zero on one, ytrain. Oh, we need y blob train. That's right, blob.
```

I'm too used to writing blob.
And we need blob train, but I believe it's the same here.
And then blob.
There we go. So we have four classes.
So we need an output features value of four.
And now if we wanted to add nonlinearity here, we could put it in between our layers here like this.
But I asked the question before, do you think that this data set needs nonlinearity?
Well, let's leave it in there to begin with.
And one of the challenges for you, oh, do we need commerce here?
I think we need commerce here.
One of the challenges for you will be to test the model with nonlinearity and without nonlinearity.
So let's just leave it in there for the time being.
What's missing from this?
Well, we need a forward method.
So def forward self X. What can we do here?
Well, because we've created this as a linear layer stack using nn.sequential, we can just go return linear layer stack and pass it X.
And what's going to happen?
Whatever input goes into the forward method is just going to go through these layers sequentially.
Oh, we need to put self here because we've initialized it in the constructor.
Beautiful.
And now let's create an instance of blob model and send it to the target device.
We'll go model four equals blob model.
And then we can use our input features parameter, which is this one here.
And we're going to pass it a value of what?
Two.
And then output features. Why? Because we have two X features.
Now, the output feature is going to be the same as the number of classes that we have for.
If we had 10 classes, we'd set it to 10.
So we'll go four.
And then the hidden units is going to be eight by default.
So we don't have to put this here, but we're going to put it there anyway.
And then, of course, we're going to send this to device.
And then we're going to go model four.
What do we get wrong here?
Unexpected keyword argument output features.
Do we spell something wrong?
No doubt. We've got a spelling mistake.
Output features. Output features.
Oh, out features.
Ah, that's what we needed. Out features, not output.
I've got a little confused there.
Okay.
There we go. Okay, beautiful.
So just recall that the parameter here for an end up linear.
Did you pick up on that?
Is out features not output features.
Output features, a little confusing here, is our final layout output layers number of features there.
So we've now got a multi-class classification model that lines up with the data that we're using.
So the shapes line up. Beautiful.

```
Well, what's next?
Well, we have to create a loss function. And, of course, a training loop.
So I'll see you in the next few videos. And let's do that together.
Welcome back. In the last video, we created our multi-class classification model.
And we did so by subclassing an end up module.
And we set up a few parameters for our class constructor here.
So that when we made an instance of the blob model, we could customize the input
features.
The output features. Remember, this lines up with how many features X has.
And the output features here lines up with how many classes are in our data.
So if we had 10 classes, we could change this to 10. And it would line up.
And then if we wanted 128 hidden units, well, we could change that.
So we're getting a little bit more programmatic with how we create models here.
And as you'll see later on, a lot of the things that we've built in here can also
be functionalized in a similar matter.
But let's keep pushing forward. What's our next step?
If we build a model, if we refer to the workflow, you'd see that we have to create
a loss function.
And an optimizer for a multi-class classification model.
And so what's our option here for creating a loss function?
Where do we find loss functions in PyTorch? I'm just going to get out of this.
And I'll make a new tab here. And if we search torch.nn
Because torch.nn is the basic building box for graphs. In other words, neural
networks.
Where do we find loss functions? Hmm, here we go. Beautiful.
So we've seen that L1 loss or MSE loss could be used for regression, predicting a
number.
And I'm here to tell you as well that for classification, we're going to be looking
at cross entropy loss.
Now, this is for multi-class classification. For binary classification, we work
with BCE loss.
And of course, there's a few more here, but I'm going to leave that as something
that you can explore on your own.
Let's jump in to cross entropy loss.
So what do we have here? This criterion computes. Remember, a loss function in
PyTorch is also referred to as a criterion.
You might also see loss function referred to as cost function, C-O-S-T.
But I call them loss functions. So this criterion computes the cross entropy loss
between input and target.
Okay, so the input is something, and the target is our target labels.
It is useful when training a classification problem with C classes. There we go.
So that's what we're doing. We're training a classification problem with C classes,
C is a number of classes.
```

Section 132: Main Topics

Key Topics:

- Machine learning mastery is also another fantastic website
- But of course, within the torch dot opt in module, you will find a lot more different optimizers

- And we'll set the learning rate to 0

▶ 📄 Click to view detailed content

```
If provided the optional argument, weight should be a 1D tensor assigning a weight
to each of the classes.
So we don't have to apply a weight here, but why would you apply a weight? Well, it
says, if we look at weight here,
this is particularly useful when you have an unbalanced training set. So just keep
this in mind as you're going forward.
If you wanted to train a dataset that has imbalanced samples, in our case we have
the same number of samples for each class,
but sometimes you might come across a dataset with maybe you only have 10 yellow
dots.
And maybe you have 500 blue dots and only 100 red and 100 light blue dots.
So you have an unbalanced dataset. So that's where you can come in and have a look
at the weight parameter here.
But for now, we're just going to keep things simple. We have a balanced dataset,
and we're going to focus on using this loss function.
If you'd like to read more, please, you can read on here. And if you wanted to find
out more, you could go, what is cross entropy loss?
And I'm sure you'll find a whole bunch of loss functions. There we go. There's the
ML cheat sheet. I love that.
The ML glossary, that's one of my favorite websites. Towards data science, you'll
find that website, Wikipedia.
Machine learning mastery is also another fantastic website. But you can do that all
in your own time.
Let's code together, hey. We'll set up a loss function. Oh, and one more resource
before we get into code is that we've got the architecture,
well, the typical architecture of a classification model. The loss function for
multi-class classification is cross entropy or torch.nn.cross entropy loss.
Let's code it out. If in doubt, code it out. So create a loss function for multi-
class classification.
And then we go, loss fn equals, and then dot cross entropy loss. Beautiful. And
then we want to create an optimizer.
Create an optimizer for multi-class classification. And then the beautiful thing
about optimizers is they're quite flexible.
They can go across a wide range of different problems. So the optimizer. So two of
the most common, and I say most common because they work quite well.
Across a wide range of problems. So that's why I've only listed two here. But of
course, within the torch dot opt in module, you will find a lot more different
optimizers.
But let's stick with SGD for now. And we'll go back and go optimizer equals torch
dot opt in for optimizer SGD for stochastic gradient descent.
The parameters we want our optimizer to optimize model four, we're up to our fourth
model already. Oh my goodness.
Model four dot parameters. And we'll set the learning rate to 0.1. Of course, you
could change the learning rate if you wanted to.
In fact, I'd encourage you to see what happens if you do because why the learning
rate is a hyper parameter.
I'm better at writing code than I am at spelling. You can change. Wonderful. So
we've now got a loss function and an optimizer for a multi class classification
problem.
What's next? Well, we could start to build.
Building a training loop. We could start to do that, but I think we have a look at
```

```
what the outputs of our model are.
So more specifically, so getting prediction probabilities for a multi class pie
torch model.
So my challenge to you before the next video is to have a look at what happens if
you pass x blob test through a model.
And remember, what is a model's raw output? What is that referred to as?
Oh, I'll let you have a think about that before the next video. I'll see you there.
Welcome back. In the last video, we created a loss function and an optimizer for
our multi class classification model.
And recall the loss function measures how wrong our model's predictions are.
And the optimizer optimizer updates our model parameters to try and reduce the
loss.
So that's what that does. And I also issued the challenge of doing a forward pass
with model four, which is the most recent model that we created.
And oh, did I just give you some code that wouldn't work? Did I do that on purpose?
Maybe, maybe not, you'll never know.
So if this did work, what are the raw outputs of our model? Let's get some raw
outputs of our model.
And if you recall, the raw outputs of a model are called logits.
So we got a runtime error expected. All tensors to be on the same device are of
course. Why did this come up?
Well, because if we go next model for dot parameters, and if we check device, what
happens here?
Oh, we need to bring this in. Our model is on the CUDA device, whereas our data is
on the CPU still.
Can we go X? Is our data a tensor? Can we check the device parameter of that? I
think we can.
I might be proven wrong here. Oh, it's on the CPU. Of course, we're getting a
runtime error.
Did you catch that one? If you did, well done. So let's see what happens.
But before we do a forward pass, how about we turn our model into a vowel mode to
make some predictions with torch dot inference mode?
We'll make some predictions. We don't necessarily have to do this because it's just
tests, but it's a good habit.
Oh, why prads? Equals, what do we get? Why prads? And maybe we'll just view the
first 10.
What do we get here? Oh, my goodness. How much are numbers on a page? Is this the
same format as our data or our test labels?
Let's have a look. No, it's not. Okay. Oh, we need why blob test. Excuse me.
```

Section 133: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
We're going to make that mistake a fair few times here. So we need to get this into
the format of this. Hmm.
How can we do that? Now, I want you to notice one thing as well is that we have one
value here per one value, except that this is actually four values.
```

Now, why is that? We have one, two, three, four. Well, that is because we set the out features up here. Our model outputs four features per sample.
So each sample right now has four numbers associated with it. And what are these called? These are the logits.
Now, what we have to do here, so let's just write this down in order to evaluate and train and test our model.
We need to convert our model's outputs, outputs which are logits to prediction probabilities, and then to prediction labels.
So we've done this before, but for binary classification. So we have to go from logits to predprobs to pred labels.
All right, I think we can do this. So we've got some logits here. Now, how do we convert these logits to prediction probabilities?
Well, we use an activation function. And if we go back to our architecture, what's our output activation here?
For a binary classification, we use sigmoid. But for multi-class classification, these are the two main differences between multi-class classification and binary classification.
One uses softmax, one uses cross entropy. And it's going to take a little bit of practice to know this off by heart.
It took me a while, but that's why we have nice tables like this. And that's why we write a lot of code together.
So we're going to use a softmax function here to convert out logits. Our models raw outputs, which is this here, to prediction probabilities.
And let's see that. So convert our models, logit outputs to prediction probabilities.
So let's create why predprobs. So I like to call prediction probabilities predprobs for short.
So torch dot softmax. And then we go why logits. And we want it across the first dimension.
So let's have a look. If we print why logits, we'll get the first five values there. And then look at the conversion here.
Why logits? Oh, why predprobs? That's what we want to compare. Predprobs. Five. Let's check this out.
Oh, what did we get wrong here? Why logits? Do we have why logits? Oh, no. We should change this to why logits, because really that's the raw output of our model here.
Why logits? Let's rerun that. Check that. We know that these are different to these, but we ideally want these to be in the same format as these, our test labels.
These are our models predictions. And now we should be able to convert. There we go. Okay, beautiful. What's happening here? Let's just get out of this.
And we will add a few code cells here. So we have some space. Now, if you wanted to find out what's happening with torch dot softmax, what could you do?
We could go torch softmax. See what's happening. Softmax. Okay, so here's the function that's happening. We replicated some nonlinear activation functions before.
So if you wanted to replicate this, what could you do? Well, if in doubt, code it out. You could code this out. You've got the tools to do so.
We've got softmax to some X input takes the exponential of X. So torch exponential over the sum of torch exponential of X. So I think you could code that out if you wanted to.
But let's for now just stick with what we've got. We've got some logits here, and we've got some softmax, some logits that have been passed through the softmax function.
So that's what's happened here. We've passed our logits as the input here, and it's gone through this activation function.
These are prediction probabilities. And you might be like, Daniel, these are still

```
just numbers on a page. But you also notice that none of them are negative.
Okay, and there's another little tidbit about what's going on here. If we sum one
of them up, let's get the first one.
Will this work? And if we go torch dot sum, what happens?
Ooh, they all sum up to one. So that's one of the effects of the softmax function.
And then if we go torch dot max of Y-pred probes.
So this is a prediction probability.
For multi class, you'll find that for this particular sample here, the 0th sample,
this is the maximum number. And so our model, what this is saying is our model is
saying, this is the prediction probability.
This is how much I think it is class 0. This number here, it's in order. This is
how much I think it is class 1. This is how much I think it is class 2.
This is how much I think it is class 3. And so we have one value for each of our
four classes, a little bit confusing because it's 0th indexed.
But the maximum value here is this index. And so how would we get the particular
index value of whatever the maximum number is across these values?
Well, we can take the argmax and we get tensor 1. So for this particular sample,
this one here, our model, and these guesses or these predictions aren't very good.
Why is that? Well, because our model is still just predicting with random numbers,
we haven't trained it yet. So this is just random output here, basically.
But for now, the premise still remains that our model thinks that for this sample
using random numbers, it thinks that index 1 is the right class or class number 1
for this particular sample.
And then for this next one, what's the maximum number here? I think it would be the
0th index and the same for the next one. What's the maximum number here?
```

Section 134: Main Topics

Key Topics:

- But of course, these numbers are going to change once we've trained our model
- 5, create a training loop, and testing loop for a multi-class pytorch model
- Let's build our first training and testing loop for a multi-class pytorch model, and I'll give you a little hint

▶ 📄 Click to view detailed content

```
Well, it would be the 0th index as well. But of course, these numbers are going to
change once we've trained our model.
So how do we get the maximum index value of all of these? So this is where we can
go, convert our model's prediction probabilities to prediction labels.
So let's do that. We can go ypreds equals torch dot argmax on ypredprobs. And if we
go across the first dimension as well. So now let's have a look at ypreds.
Do we have prediction labels in the same format as our ylob test? Beautiful. Yes,
we do. Although many of them are wrong, as you can see, ideally they would line up
with each other.
But because our model is predicting or making predictions with random numbers, so
they haven't been our model hasn't been trained. All of these are basically random
```

outputs.
So hopefully once we train our model, it's going to line up the values of the predictions are going to line up with the values of the test labels.
But that is how we go from our model's raw outputs to prediction probabilities to prediction labels for a multi-class classification problem.
So let's just add the steps here, logits, raw output of the model, predprobs, to get the prediction probabilities, use torch dot softmax or the softmax activation function, pred labels, take the argmax of the prediction probabilities.
So we're going to see this in action later on when we evaluate our model, but I feel like now that we know how to go from logits to prediction probabilities to pred labels, we can write a training loop.
So let's set that up. 8.5, create a training loop, and testing loop for a multi-class pytorch model. This is so exciting.
I'll see you in the next video. Let's build our first training and testing loop for a multi-class pytorch model, and I'll give you a little hint.
It's quite similar to the training and testing loops we've built before, so you might want to give it a shot. I think you can.
Otherwise, we'll do it together in the next video.
Welcome back. In the last video, we covered how to go from raw logits, which is the output of the model, the raw output of the model for a multi-class pytorch model.
Then we turned our logits into prediction probabilities using torch.softmax, and then we turn those prediction probabilities into prediction labels by taking the argmax, which returns the index of where the maximum value occurs in the prediction probability.
So for this particular sample, with these four values, because it outputs four values, because we're working with four classes, if we were working with 10 classes, it would have 10 values, the principle of these steps would still be the same.
So for this particular sample, this is the value that's the maximum, so we would take that index, which is 1. For this one, the index 0 has the maximum value.
For this sample, same again, and then same again, I mean, these prediction labels are just random, right? So they're quite terrible.
But now we're going to change that, because we're going to build a training and testing loop for our multi-class model.
Let's do that. So fit the multi-class model to the data.
Let's go set up some manual seeds.
Torch dot manual seed, again, don't worry too much if our numbers on the page are not exactly the same. That's inherent to the randomness of machine learning.
We're setting up the manual seeds to try and get them as close as possible, but these do not guarantee complete determinism, which means the same output.
But we're going to try. The direction is more important.
Set number of epochs. We're going to go epochs. How about we just do 100? I reckon we'll start with that. We can bump it up to 1000 if we really wanted to.
Let's put the data to the target device. What's our target device? Well, it doesn't really matter because we've set device agnostic code.
So whether we're working with a CPU or a GPU, our code will use whatever device is available. I'm typing blog again.
So we've got x blob train, y blob train. This is going to go where? It's going to go to the device.
And y blob train to device. And we're going to go x blob test. And then y blob test equals x blob test to device.
Otherwise, we'll get device issues later on, and we'll send this to device as well.
Beautiful. Now, what do we do now? Well, we loop through data.
Loop through data. So for an epoch in range epochs for an epoch in a range.
Epox. I don't want that auto correct. Come on, Google Colab. Work with me here.
We're training our first multi-class classification model. This is serious business. No, I'm joking. It's actually quite fun.

```
So model four dot train. And let's do the forward pass. I'm not going to put much
commentary here because we've been through this before.
But what are the logits? The logits are raw outputs of our model. So we'll just go
x blob train.
And x test. I didn't want that. X blob train. Why did that do that? I need to turn
off auto correct in Google Colab. I've been saying it for a long time.
Y pred equals torch dot softmax. So what are we doing here? We're going from logits
to prediction probabilities here.
So torch softmax. Y logits. Across the first dimension. And then we can take the
argmax of this and dim equals one.
In fact, I'm going to show you a little bit of, oh, I've written blog here. Maybe
auto correct would have been helpful for that.
```

Section 135: Main Topics

Key Topics:

- Of course

▶ 📄 Click to view detailed content

```
A little trick. You don't actually have to do the torch softmax. The logits. If you
just took the argmax of the logits is a little test for you.
Just take the argmax of the logits. And see, do you get the same similar outputs as
what you get here?
So I've seen that done before, but for completeness, we're going to use the softmax
activation function because you'll often see this in practice.
And now what do we do? We calculate the loss. So the loss FM. We're going to use
categorical cross entropy here or just cross entropy loss.
So if we check our loss function, what do we have? We have cross entropy loss.
We're going to compare our models, logits to y blob train.
And then what are we going to do? We're going to calculate the accuracy because
we're working with the classification problem.
It'd be nice if we had accuracy as well as loss. Accuracy is one of the main
classification evaluation metrics.
y pred equals y pred. y pred. And now what do we do? Well, we have to zero grab the
optimizer. Optimizer zero grad.
Then we go loss backward. And then we step the optimizer. Optimizer step, step,
step.
So none of these steps we haven't covered before. We do the forward pass. We
calculate the loss and any evaluation metric we choose to do so.
We zero the optimizer. We perform back propagation on the loss. And we step the
optimizer.
The optimizer will hopefully behind the scenes update the parameters of our model
to better represent the patterns in our training data.
And so we're going to go testing code here. What do we do for testing code? Well,
or inference code.
We set our model to a vowel mode.
That's going to turn off a few things behind the scenes that our model doesn't need
such as dropout layers, which we haven't covered.
```

But you're more than welcome to check them out if you go torch and end.
Dropout layers. Do we have dropout? Dropout layers. Beautiful. And another one that it turns off is match norm.
Beautiful. And also you could search this. What does model dot a vowel do?
And you might come across stack overflow question. One of my favorite resources.
So there's a little bit of extra curriculum. But I prefer to see things in action.
So with torch inference mode, again, this turns off things like gradient tracking and a few more things.
So we get as fast as code as possible because we don't need to track gradients when we're making predictions.
We just need to use the parameters that our model has learned.
We want X blob test to go to our model here for the test logits. And then for the test preds, we're going to do the same step as what we've done here.
We're going to go torch dot softmax on the test logits across the first dimension.
And we're going to call the argmax on that to get the index value of where the maximum prediction probability value occurs.
We're going to calculate the test loss or loss function. We're going to pass in what the test logits here.
Then we're going to pass in why blob test compare the test logits behind the scenes.
Our loss function is going to do some things that convert the test logits into the same format as our test labels and then return us a value for that.
Then we'll also calculate the test accuracy here by passing in the why true as why blob test.
And we have the y pred equals y pred.
Wonderful.
And then what's our final step?
Well, we want to print out what's happening because I love seeing metrics as our model trains.
It's one of my favorite things to watch.
If we go if epoch, let's do it every 10 epochs because we've got 100 so far.
It equals zero.
Let's print out a nice f string with epoch.
And then we're going to go loss.
What do we put in here?
We'll get our loss value, but we'll take it to four decimal places and we'll get the training accuracy, which will be acc.
And we'll take this to two decimal places and we'll get a nice percentage sign there.
And we'll go test loss equals test loss and we'll go there.
And finally, we'll go test act at the end here, test act.
Now, I'm sure by now we've written a fair few of these.
You're either getting sick of them or you're like, wow, I can actually do the steps through here.
And so don't worry, we're going to be functionalizing all of this later on,
but I thought I'm going to include them as much as possible so that we can practice as much as possible together.
So you ready?
We're about to train our first multi-class classification model.
In three, two, one, let's go.
No typos.
Of course.
What do we get wrong here?
Oh, this is a fun error.
Runtime error.
NLL loss for reduced CUDA kernel to the index not implemented for float.
Okay, that's a pretty full on bunch of words there.

```
I don't really know how to describe that.
But here's a little hint.
We've got float there.
So we know that float is what?
Float is a form of data.
It's a data type.
So potentially because that's our hint.
We said not implemented for float.
So maybe we've got something wrong up here.
Our data is of the wrong type.
Can you see anywhere where our data might be the wrong type?
Well, let's print it out.
Where's our issue here?
Why logits?
Why blob train?
Okay.
Why blob train?
And why logits?
What does why blob train look like?
Why blob train?
Okay.
And what's the D type here?
Float.
Okay.
So it's not implemented for float.
Hmm.
```

Section 136: Main Topics

Key Topics:

- So if we run this again and now this one took me a while to find and I want you to know that, that behind the scenes, even though, again, this is a machine learning cooking show, it still takes a while to troubleshoot code and you're going to come across this
- So we turned our labels here into float, which generally is okay in PyTorch
- We're slowly working through all of the errors in deep learning here

▶ 📄 Click to view detailed content

```
Maybe we have to turn them into a different data type.
What if we went type torch long tensor?
What happens here?
Expected all tensors to be on the same device but found at least two devices.
Oh, my goodness.
What do we got wrong here?
Type torch long tensor.
```

Friends.
Guess what?
I found it.
And so it was to do with this pesky little data type issue here.
So if we run this again and now this one took me a while to find and I want you to know that,
that behind the scenes, even though, again, this is a machine learning cooking show,
it still takes a while to troubleshoot code and you're going to come across this.
But I thought rather than spend 10 minutes doing it in a video, I'll show you what I did.
So we went through this and we found that, hmm, there's something going on here.
I don't quite know what this is.
And that seems quite like a long string of words, not implemented for float.
And then we looked back at the line where it went wrong.
And so that we know that maybe the float is hinting at that one of these two tensors is of the wrong data type.
Now, why would we think that it's the wrong data type?
Well, because anytime you see float or int or something like that, it generally hints at one of your data types being wrong.
And so the error is actually right back up here where we created our tensor data.
So we turned our labels here into float, which generally is okay in PyTorch.
However, this one should be of type torch dot long tensor, which we haven't seen before.
But if we go into torch long tensor, let's have a look torch dot tensor.
Do we have long tensor?
Here we go.
64 bit integer signed.
So why do we need torch dot long tensor?
And again, this took me a while to find.
And so I want to express this that in your own code, you probably will butt your head up against some issues and errors that do take you a while to find.
And data types is one of the main ones.
But if we look in the documentation for cross entropy loss, the way I kind of found this out was this little hint here.
The performance of the criteria is generally better when the target contains class indices, as this allows for optimized computation.
But I read this and it says target contains class indices.
I'm like, hmm, alza indices already, but maybe they should be integers and not floats.
But then if you actually just look at the sample code, you would find that they use d type equals torch dot long.
Now, that's the thing with a lot of code around the internet is that sometimes the answer you're looking for is a little bit buried.
But if in doubt, run the code and butt your head up against a wall for a bit and keep going.
So let's just rerun all of this and see do we have an error here?
Let's train our first multi-class classification model together.
No arrows, fingers crossed.
But what did we get wrong here?
OK, so we've got different size.
We're slowly working through all of the errors in deep learning here.
Value error, input batch size 200 to match target size 200.
So this is telling me maybe our test data, which is of size 200, is getting mixed up with our training data, which is of size 800.
So we've got test loss, the test logits, model four.
What's the size?

```
Let's print out print test logits dot shape and wine blob test.
So troubleshooting on the fly here, everyone.
What do we got?
Torch size 800.
Where are our test labels coming from?
Wine blob test equals, oh, there we go.
Ah, did you catch that before?
Maybe you did, maybe you didn't.
But I think we should be right here.
Now if we just comment out this line, so we've had a data type issue and we've had
a shape issue.
Two of the main and machine learning, oh, and again, we've had some issues.
Wine blob test.
What's going on here?
I thought we just changed the shape.
Oh, no, we have to go up and reassign it again because now this is definitely why
blob, yes.
Let's rerun all of this, reassign our data.
We are running into every single error here, but I'm glad we're doing this because
otherwise you might not see how to
troubleshoot these type of things.
So the size of a tensor much match the size.
Oh, we're getting the issue here.
Test spreads.
Oh, my goodness.
We have written so much code here.
Test spreads.
So instead of wire spread, this should be test spreads.
Fingers crossed.
Are we training our first model yet or what?
There we go.
Okay, I'm going to printing out some stuff.
I don't really want to print out that stuff.
I want to see the loss go down, so I'm going to.
So friends, I hope you know that we've just been through some of the most
fundamental troubleshooting steps.
And you might say, oh, Daniel, there's a cop out because you're just coding wrong.
And in fact, I code wrong all the time.
But we've now worked out how to troubleshoot them shape errors and data type
errors.
But look at this.
After all of that, thank goodness.
Our loss and accuracy go in the directions that we want them to go.
So our loss goes down and our accuracy goes up.
Beautiful.
So it looks like that our model is working on a multi-class classification data
set.
So how do we check that?
Well, we're going to evaluate it in the next step by visualize, visualize,
visualize.
```

Section 137: Main Topics

Key Topics:

- But we also butted our heads up against two of the most common issues in machine learning and deep learning in general
- And trust me, you're going to run across many of them in your own deep learning and machine learning endeavors
- But we can further evaluate this by making and evaluating predictions with a PyTorch multi-class model

▶ 📄 Click to view detailed content

```
So you might want to give that a shot.
See if you can use our plot decision boundary function.
We'll use our model to separate the data here.
So it's going to be much the same as what we did for binary classification.
But this time we're using a different model and a different data set.
I'll see you there.
Welcome back.
In the last video, we went through some of the steps that we've been through before
in terms of training and testing a model.
But we also butted our heads up against two of the most common issues in machine
learning and deep learning in general.
And that is data type issues and shape issues.
But luckily we were able to resolve them.
And trust me, you're going to run across many of them in your own deep learning and
machine learning endeavors.
So I'm glad that we got to have a look at them and sort of I could show you what I
do to troubleshoot them.
But in reality, it's a lot of experimentation.
Run the code, see what errors come out, Google the errors, read the documentation,
try again.
But with that being said, it looks like that our model, our multi-class
classification model has learned something.
The loss is going down, the accuracy is going up.
But we can further evaluate this by making and evaluating predictions with a
PyTorch multi-class model.
So how do we make predictions?
We've seen this step before, but let's reiterate.
Make predictions, we're going to set our model to what mode, a vowel mode.
And then we're going to turn on what context manager, inference mode.
Because we want to make inference, we want to make predictions.
Now what do we make predictions on?
Or what are the predictions? They're going to be logits because why?
They are the raw outputs of our model.
So we'll take model four, which we just trained and we'll pass it the test data.
Well, it needs to be blob test, by the way.
I keep getting that variable mixed up.
We just had enough problems with the data, Daniel.
We don't need any more. You're completely right.
I agree with you.
But we're probably going to come across some more problems in the future.
Don't you worry about that.
```

So let's view the first 10 predictions.
Why logits? What do they look like?
All right, just numbers on the page. They're raw logits.
Now how do we go from go from logits to prediction probabilities?
How do we do that?
With a multi-class model, we go y-pred-probs equals torch.softmax on the y-logits.
And we want to do it across the first dimension.
And what do we have when we go pred-probs?
Let's go up to the first 10.
Are we apples to apples yet?
What does our y-blog test look like?
We're not apples to apples yet, but we're close.
So these are prediction probabilities.
You'll notice that we get some fairly different values here.
And remember, the one closest to one here, the value closest to one,
which looks like it's this, the index of the maximum value
is going to be our model's predicted class.
So this index is index one.
And does it correlate here? Yes.
One, beautiful.
Then we have index three, which is the maximum value here.
Three, beautiful.
And then we have, what do we have here?
Index two, yes.
Okay, wonderful.
But let's not step through that.
We're programmers.
We can do this with code.
So now let's go from pred-probs to pred-labels.
So y-pred-equals, how do we do that?
Well, we can do torch.argmax on the y-pred-probs.
And then we can pass dim equals one.
We could also do it this way.
So y-pred-probs call dot-argmax.
There's no real difference between these two.
But we're just going to do it this way, called torch.argmax.
y-pred-es, let's view the first 10.
Are we now comparing apples to apples when we go y-blob test?
Yes, we are.
Have a go at that.
Look, one, three, two, one, zero, three, one, three, two, one, zero, three.
Beautiful.
Now, we could line these up and look at and compare them all day.
I mean, that would be fun.
But I know what something that would be even more fun.
Let's get visual.
So plot dot figure.
And we're going to go fig size equals 12.6, just because the beauty of this
being a cooking show is I kind of know what ingredients work from ahead of time.
Despite what you saw in the last video with all of that trouble shooting.
But I'm actually glad that we did that because seriously.
Shape issues and data type issues.
You're going to come across a lot of them.
The two are the main issues I troubleshoot, aside from device issues.
So let's go x-blob train and y-blob train.
And we're going to do another plot here.
We're going to get subplot one, two, two.

```
And we're going to do this for the test data.
Test and then plot decision boundary.
Plot decision boundary with model four on x-blob test and y-blob test as well.
Let's see this.
Did we train a multi-class?
Oh my goodness.
Yes, we did.
Our code worked faster than I can speak.
Look at that beautiful looking plot.
We've separated our data almost as best as what we could.
Like there's some here that are quite inconspicuous.
And now what's the thing about these lines?
With this model have worked, I posed the question a fair few videos ago,
whenever we created our multi-class model that could we separate this data
without nonlinear functions.
So how about we just test that?
Since we've got the code ready, let's go back up.
We've got nonlinear functions here.
We've got relu here.
So I'm just going to recreate our model there.
So I just took relu out.
That's all I did.
```

Section 138: Main Topics

Key Topics:

- And in fact, PyTorch makes it so easy to add nonlinearities to our model, we might as well have them in so that our model can use it if it needs it and if it doesn't need it, well, hey, it's going to build a pretty good model as we saw before if we included the nonlinearities in our model
- So, just keep that in mind, and the beauty of PyTorch is that it allows us to create models with linear and non-linear functions quite flexibly
- But, of course, there were a few tough samples, which I mean a little bit hard

▶ 📄 Click to view detailed content

```
Commented it out, this code will still all work.
Or fingers crossed it will.
Don't count your chickens before they hatch.
Daniel, come on.
We're just going to rerun all of these cells.
All the code's going to stay the same.
All we did was we took the nonlinearity out of our model.
Is it still going to work?
Oh my goodness.
It still works.
```

Now why is that?
Well, you'll notice that the lines are a lot more straighter here.
Did we get different metrics?
I'll leave that for you to compare.
Maybe these will be a little bit different.
I don't think they're too far different.
But that is because our data is linearly separable.
So we can draw straight lines only to separate our data.
However, a lot of the data that you deal with in practice
will require linear and nonlinear.
Hence why we spent a lot of time on that.
Like the circle data that we covered before.
And let's look up an image of a pizza.
If you're building a food vision model to take photos of food
and separate different classes of food,
could you do this with just straight lines?
You might be able to, but I personally don't think
that I could build a model to do such a thing.
And in fact, PyTorch makes it so easy to add nonlinearities
to our model, we might as well have them in
so that our model can use it if it needs it
and if it doesn't need it, well, hey,
it's going to build a pretty good model as we saw before
if we included the nonlinearities in our model.
So we could uncomment these and our model is still
going to perform quite well.
That is the beauty of neural networks,
is that they decide the numbers that should
represent outdated the best.
And so, with that being said, we've evaluated our model,
we've trained our multi-class classification model,
we've put everything together, we've gone from binary
classification to multi-class classification.
I think there's just one more thing that we should cover
and that is, let's go here, section number nine,
a few more classification metrics.
So, as I said before, evaluating a model,
let's just put it here, to evaluate our model,
our classification models, that is,
evaluating a model is just as important as training a model.
So, I'll see you in the next video.
Let's cover a few more classification metrics.
Welcome back.
In the last video, we evaluated our
multi-class classification model visually.
And we saw that it did pretty darn well,
because our data turned out to be linearly separable.
So, our model, even without non-linear functions,
could separate the data here.
However, as I said before, most of the data that you deal with
will require some form of linear and non-linear function.
So, just keep that in mind, and the beauty of PyTorch is
that it allows us to create models with linear
and non-linear functions quite flexibly.
So, let's write down here.
If we wanted to further evaluate our classification models,
we've seen accuracy.

```
So, accuracy is one of the main methods
of evaluating classification models.
So, this is like saying, out of 100 samples,
how many does our model get right?
And so, we've seen our model right now
is that testing accuracy of nearly 100%.
So, it's nearly perfect.
But, of course, there were a few tough samples,
which I mean a little bit hard.
Some of them are even within the other samples,
so you can forgive it a little bit here
for not being exactly perfect.
What are some other metrics here?
Well, we've also got precision,
and we've also got recall.
Both of these will be pretty important
when you have classes with different amounts of values in them.
So, precision and recall.
So, accuracy is pretty good to use when you have balanced classes.
So, this is just text on a page for now.
F1 score, which combines precision and recall.
There's also a confusion matrix,
and there's also a classification report.
So, I'm going to show you a few code examples
of where you can access these,
and I'm going to leave it to you as extra curriculum
to try each one of these out.
So, let's go into the keynote.
And by the way, you should pay yourself on the back here
because we've just gone through all of the PyTorch workflow
for a classification problem.
Not only just binary classification,
we've done multi-class classification as well.
So, let's not stop there, though.
Remember, building a model,
evaluating a model is just as important as building a model.
So, we've been through non-linearity.
We've seen how we could replicate non-linear functions.
We've talked about the machine learning explorer's motto,
visualize, visualize, visualize.
Machine learning practitioners motto is experiment, experiment, experiment.
I think I called that the machine learning or data scientist motto.
Same thing, you know?
And steps in modeling with PyTorch.
We've seen this in practice,
so we don't need to look at these slides.
I mean, they'll be available on the GitHub if you want them,
but here we are.
Some common classification evaluation methods.
So, we have accuracy.
There's the formal formula if you want,
but there's also code, which is what we've been focusing on.
So, we wrote our own accuracy function, which replicates this.
By the way, Tp stands for not toilet paper,
it stands for true positive,
Tn is true negative, false positive, Fp, false negative, Fn.
And so, the code, we could do torch metrics.
```

```
Oh, what's that?
But when should you use it?
The default metric for classification problems.
Note, it is not the best for imbalanced classes.
So, if you had, for example,
1,000 samples of one class,
so, number one, label number one,
but you had only 10 samples of class zero.
So, accuracy might not be the best to use for then.
```

Section 139: Main Topics

Key Topics:

- Torchmetrics is a PyTorch-like library
- So we've built our own accuracy function, but the beauty of using torchmetrics is that it uses PyTorch-like code
- If you want access to a lot of PyTorch metrics, see torchmetrics

▶ 📄 Click to view detailed content

```
For imbalanced data sets,
you might want to look into precision and recall.
So, there's a great article called,
I think it's beyond accuracy, precision and recall,
which gives a fantastic overview of, there we go.
This is what I'd recommend.
There we go, by Will Coestron.
So, I'd highly recommend this article as some extra curriculum here.
See this article for when to use precision recall.
We'll go there.
Now, if we look back,
there is the formal formula for precision,
true positive over true positive plus false positive.
So, higher precision leads to less false positives.
So, if false positives are not ideal,
you probably want to increase precision.
If false negatives are not ideal,
you want to increase your recall metric.
However, you should be aware that there is such thing as a precision recall trade-
off.
And you're going to find this in your experimentation.
Precision recall trade-off.
So, that means that, generally, if you increase precision,
you lower recall.
And, inversely, if you increase precision, you lower recall.
So, check out that, just to be aware of that.
But, again, you're going to learn this through practice of evaluating your models.
```

```
If you'd like some code to do precision and recall,
you've got torchmetrics.precision, or torchmetrics.recall,
as well as scikit-learn.
So scikit-learn has implementations for many different classification metrics.
Torchmetrics is a PyTorch-like library.
And then we have F1 score, which combines precision and recall.
So, it's a good combination if you want something in between these two.
And then, finally, there's a confusion matrix.
I haven't listed here a classification report, but I've listed it up here.
And we can see a classification report in scikit-learn.
Classification report.
Classification report kind of just puts together all of the metrics that we've
talked about.
And we can go there.
But I've been talking a lot about torchmetrics.
So let's look up torchmetrics' accuracy.
Torchmetrics.
So this is a library.
I don't think it comes with Google Colab at the moment,
but you can import torchmetrics, and you can initialize a metric.
So we've built our own accuracy function, but the beauty of using torchmetrics
is that it uses PyTorch-like code.
So we've got metric, preds, and target.
And then we can find out what the value of the accuracy is.
And if you wanted to implement your own metrics, you could subclass the metric
class here.
But let's just practice this.
So let's check to see if I'm going to grab this and copy this in here.
If you want access to a lot of PyTorch metrics, see torchmetrics.
So can we import torchmetrics?
Maybe it's already in Google Colab.
No, not here.
But that's all right.
We'll go pip install torchmetrics.
So Google Colab has access to torchmetrics.
And that's going to download from torchmetrics.
It shouldn't take too long.
It's quite a small package.
Beautiful.
And now we're going to go from torchmetrics import accuracy.
Wonderful.
And let's see how we can use this.
So setup metric.
So we're going to go torchmetric underscore accuracy.
We could call it whatever we want, really.
But we need accuracy here.
We're just going to set up the class.
And then we're going to calculate the accuracy of our multi-class model by calling
torchmetric accuracy.
And we're going to pass it Y threads and Y blob test.
Let's see what happens here.
Oh, what did we get wrong?
Runtime error.
Expected all tensors to be on the same device, but found at least two devices.
Oh, of course.
Now, remember how I said torchmetrics implements PyTorch like code?
Well, let's check what device this is on.
```

```
Oh, it's on the CPU.
So something to be aware of that if you use torchmetrics, you have to make sure
your metrics
are on the same device by using device agnostic code as your data.
So if we run this, what do we get?
Beautiful.
We get an accuracy of 99.5%, which is in line with the accuracy function that we
coded ourselves.
So if you'd like a lot of pre-built metrics functions, be sure to see either
scikit-learn for any of these or torchmetrics for any PyTorch like metrics.
But just be aware, if you use the PyTorch version, they have to be on the same
device.
And if you'd like to install it, what do we have?
Where's the metrics?
Module metrics?
Do we have classification?
There we go.
So look how many different types of classification metrics there are from
torchmetrics.
So I'll leave that for you to explore.
The resources for this will be here.
This is an extracurricular article for when to use precision recall.
And another extracurricular would be to go through the torchmetrics module for 10
minutes
and have a look at the different metrics for classification.
So with that being said, I think we've covered a fair bit.
But I think it's also time for you to practice what you've learned.
So let's cover some exercises in the next video.
I'll see you there.
Welcome back.
In the last video, we looked at a few more classification metrics, a little bit of
a high
level overview for some more ways to evaluate your classification models.
And I linked some extracurricular here that you might want to look into as well.
But we have covered a whole bunch of code together.
But now it's time for you to practice some of this stuff on your own.
And so I have some exercises prepared.
Now, where do you go for the exercises?
Well, remember on the learnpytorch.io book, for each one of these chapters, there's
a section.
Now, just have a look at how much we've covered.
If I scroll, just keep scrolling.
Look at that.
We've covered all of that in this module.
That's a fair bit of stuff.
But down the bottom of each one is an exercises section.
```

Section 140: Main Topics

Key Topics:

- And then, of course, extracurricular

- But, of course, you can just find it on the learnpytorch
- And so if you'd like a template of the exercise code, you can go to the PyTorch deep learning repo

▶ 📄 Click to view detailed content

```
So all exercises are focusing on practicing the code in the sections above, all of
these
sections here.
I've got number one, two, three, four, five, six, seven.
Yeah, seven exercises, nice, writing plenty of code.
And then, of course, extracurricular.
So these are some challenges that I've mentioned throughout the entire section zero
two.
But I'm going to link this in here.
Exercises.
But, of course, you can just find it on the learnpytorch.io book.
So if we come in here and we just create another heading.
Exercises.
And extracurricular.
And then we just write in here.
See exercises and extracurricular.
Here.
And so if you'd like a template of the exercise code, you can go to the PyTorch
deep learning
repo.
And then within the extras folder, we have exercises and solutions.
You might be able to guess what's in each of these exercises.
We have O2 PyTorch classification exercises.
This is going to be some skeleton code.
And then, of course, we have the solutions as well.
Now, this is just one form of solutions.
But I'm not going to look at those because I would recommend you looking at the
exercises
first before you go into the solutions.
So we have things like import torch.
Set up device agnostic code.
Create a data set.
Turn data into a data frame.
And then et cetera, et cetera.
For the things that we've done throughout this section.
So give that a go.
Try it on your own.
And if you get stuck, you can refer to the notebook that we've coded together.
All of this code here.
You can refer to the documentation, of course.
And then you can refer to as a last resort, the solutions notebooks.
So give that a shot.
And congratulations on finishing.
Section 02 PyTorch classification.
Now, if you're still there, you're still with me.
Let's move on to the next section.
We're going to cover a few more things of deep learning with PyTorch.
I'll see you there.
```

Hello, and welcome back.
We've got another section.
We've got computer vision and convolutional neural networks with.
PyTorch.
Now, computer vision is one of my favorite, favorite deep learning topics.
But before we get into the materials, let's answer a very important question.
And that is, where can you get help?
So, first and foremost, is to follow along with the code as best you can.
We're going to be writing a whole bunch of PyTorch computer vision code.
And remember our motto.
If and out, run the code.
See what the inputs and outputs are of your code.
And that's try it yourself.
If you need the doc string to read about what the function you're using does,
you can press shift command and space in Google CoLab.
Or it might be control if you're on Windows.
Otherwise, if you're still stuck, you can search for the code that you're running.
You might come across stack overflow or the PyTorch documentation.
We've spent a bunch of time in the PyTorch documentation already.
And we're going to be referencing a whole bunch in the next module in section three.
We're up to now.
If you go through all of these four steps, the next step is to try it again.
If and out, run the code.
And then, of course, if you're still stuck, you can ask a question on the PyTorch deep learning repo.
Discussions tab.
Now, if we open this up, we can go new discussion.
And you can write here section 03 for the computer vision.
My problem is, and then in here, you can write some code.
Be sure to format it as best you can.
That way it'll help us answer it.
And then go, what's happening here?
Now, why do I format the code in these back ticks here?
It's so that it looks like code and that it's easier to read when it's formatted on the GitHub discussion.
Then you can select a category.
If you have a general chat, an idea, a poll, a Q&A, or a show and tell of something you've made,
or what you've learned from the course.
For question and answering, you want to put it as Q&A.
Then you can click start discussion.
And it'll appear here.
And that way, they'll be searchable and we'll be able to help you out.
So I'm going to get out of this.
And oh, speaking of resources, we've got the PyTorch deep learning repo.
The links will be where you need the links.
All of the code that we're going to write in this section is contained within the section 3 notebook.
PyTorch computer vision.
Now, this is a beautiful notebook annotated with heaps of text and images.
You can go through this on your own time and use it as a reference to help out.
If you get stuck on any of the code we write through the videos, check it out in this notebook because it's probably here somewhere.
And then finally, let's get out of these.
If we come to the book version of the course,
this is learnpytorch.io.

```
We've got home.
This will probably be updated by the time you look at that.
But we have section 03, which is PyTorch computer vision.
It's got all of the information about what we're going to cover in a book format.
And you can, of course, skip ahead to different subtitles.
See what we're going to write here.
All of the links you need and extra resources will be at learnpytorch.io.
And for this section, it's PyTorch computer vision.
With that being said, speaking of computer vision, you might have the question,
what is a computer vision problem?
Well, if you can see it, it could probably be phrased at some sort of computer
vision problem.
That's how broad computer vision is.
So let's have a few concrete examples.
We might have a binary classification problem,
```

Section 141: Main Topics

Key Topics:

- And so our machine learning model may take in the pixels of an image and understand the different patterns that go into what a steak looks like and the same thing with a pizza
- So behind the scenes, Nutrify is using the pixels of an image and then running them through a machine learning model and classifying it first, whether it's food or not food
- So potentially, you could design a machine learning model to find this certain type of car

▶ 📄 Click to view detailed content

```
such as if we wanted to have two different images.
Is this photo of steak or pizza?
We might build a model that understands what steak looks like in an image.
This is a beautiful dish that I cooked, by the way.
This is me eating pizza at a cafe with my dad.
And so we could have binary classification, one thing or another.
And so our machine learning model may take in the pixels of an image
and understand the different patterns that go into what a steak looks like
and the same thing with a pizza.
Now, the important thing to note is that we won't actually be telling our model
what to learn.
It will learn those patterns itself from different examples of images.
Then we could step things up and have a multi-class classification problem.
You're noticing a trend here.
We've covered classification before, but classification can be quite broad.
```

It can be across different domains, such as vision or text or audio.
But if we were working with multi-class classification for an image problem,
we might have, is this a photo of sushi, steak or pizza?
And then we have three classes instead of two.
But again, this could be 100 classes, such as what Nutrify uses,
which is an app that I'm working on.
We go to Nutrify.app.
This is bare bones at the moment.
But right now, Nutrify can classify up to 100 different foods.
So if you were to upload an image of food, let's give it a try.
Nutrify, we'll go into images, and we'll go into sample food images.
And how about some chicken wings?
What does it classify this as?
Chicken wings. Beautiful.
And then if we upload an image of not food, maybe.
Let's go to Nutrify.
This is on my computer, by the way.
You might not have a sample folder set up like this.
And then if we upload a photo of a Cybertruck, what does it say?
No food found.
Please try another image.
So behind the scenes, Nutrify is using the pixels of an image
and then running them through a machine learning model
and classifying it first, whether it's food or not food.
And then if it is food, classifying it as what food it is.
So right now it works for 100 different foods.
So if we have a look at all these, it can classify apples,
artichokes, avocados, barbecue sauce.
Each of these work at different levels of performance,
but that's just something to keep in mind of what you can do.
So the whole premise of Nutrify is to upload a photo of food
and then learn about the nutrition about it.
So let's go back to our keynote.
What's another example?
Well, we could use computer vision for object detection,
where you might answer the question is,
where's the thing we're looking for?
So for example, this car here, I caught them on security camera,
actually did a hit and run on my new car,
wasn't that much of an expensive car, but I parked it on the street
and this person, the trailer came off the back of their car
and hit my car and then they just picked the trailer up
and drove away.
But I went to my neighbor's house and had to look at their security footage
and they found this car.
So potentially, you could design a machine learning model
to find this certain type of car.
It was an orange jute, by the way, but the images were in black and white
to detect to see if it ever recognizes a similar car
that comes across the street and you could go,
hey, did you crash into my car the other day?
I didn't actually find who it was.
So sadly, it was a hit and run.
But that's object detection, finding something in an image.
And then you might want to find out
whether the different sections in this image.
So this is a great example at what Apple uses on their devices,

```
iPhones and iPads and whatnot, to segregate or segment
the different sections of an image, so person one, person two,
skin tones, hair, sky, original.
And then it enhances each of these sections in different ways.
So that's a practice known as computational photography.
But the whole premise is how do you segment different portions of an image?
And then there's a great blog post here
that talks about how it works and what it does
and what kind of model that's used.
I'll leave that as extra curriculum if you'd like to look into it.
So if you have these images, how do you enhance the sky?
How do you make the skin tones look how they should?
How do you remove the background if you really wanted to?
So all of this happens on device.
So that's where I got that image from, by the way.
Semantic Mars.
And this is another great blog, Apple Machine Learning Research.
So to keep this in mind, we're about to see another example for computer vision,
which is Tesla Computer Vision.
A lot of companies have websites such as Apple Machine Learning Research
where they share a whole bunch of what they're up to in the world of machine
learning.
So in Tesla's case, they have eight cameras on each of their self-driving cars
that fuels their full self-driving beta software.
And so they use computer vision to understand what's going on in an image
and then plan what's going on.
So this is three-dimensional vector space.
And what this means is they're basically taking these different viewpoints
from the eight different cameras, feeding them through some form of neural network,
and turning the representation of the environment around the car into a vector.
So a long string of numbers.
And why will it do that?
Well, because computers understand numbers far more than they understand images.
```

Section 142: Main Topics

Key Topics:

- So Tesla uses PyTorch
- So the exact same code that we're writing, Tesla uses similar PyTorch code to, of course, they write PyTorch code to suit their problem
- But nonetheless, they use PyTorch code to train their machine learning models that power their self-driving software

▶ 📄 Click to view detailed content

```
So we might be able to recognize what's happening here.
But for a computer to understand it, we have to turn it into vector space.
```

```
And so if you want to have a look at how Tesla uses computer vision,
so this is from Tesla's AI Day video.
I'm not going to play it all because it's three hours long,
but I watched it and I really enjoyed it.
So there's some information there.
And there's a little tidbit there.
If you go to two hours and one minute and 31 seconds on the same video,
have a look at what Tesla do.
Well, would you look at that? Where have we seen that before?
That's some device-agnostic code, but with Tesla's custom dojo chip.
So Tesla uses PyTorch.
So the exact same code that we're writing,
Tesla uses similar PyTorch code to, of course,
they write PyTorch code to suit their problem.
But nonetheless, they use PyTorch code to train their machine learning models
that power their self-driving software.
So how cool is that?
And if you want to have a look at another example,
there's plenty of different Tesla self-driving videos.
So, oh, we can just play it right here.
I was going to click the link.
So look, this is what happens.
If we have a look in the environment,
Tesla, the cameras, understand what's going on here.
And then it computes it into this little graphic here
on your heads-up display in the car.
And it kind of understands, well, I'm getting pretty close to this car.
I'm getting pretty close to that car.
And then it uses this information about what's happening,
this perception, to plan where it should drive next.
And I believe here it ends up going into it.
It has to stop.
Yeah, there we go.
Because we've got a stop sign.
Look at that.
It's perceiving the stop sign.
It's got two people here.
It just saw a car drive pass across this street.
So that is pretty darn cool.
That's just one example of computer vision, one of many.
And how would you find out what computer vision can be used for?
Here's what I do.
What can computer vision be used for?
Plenty more resources.
So, oh, there we go.
27 most popular computer vision applications in 2022.
So we've covered a fair bit there.
But what are we going to cover specifically with PyTorch code?
Well, broadly, like that.
We're going to get a vision data set to work with using torch vision.
So PyTorch has a lot of different domain libraries.
Torch vision helps us deal with computer vision problems.
And there's existing data sets that we can leverage to play around with computer
vision.
We're going to have a look at the architecture of a convolutional neural network,
also known as a CNN with PyTorch.
We're going to look at an end-to-end multi-class image classification problem.
```

So multi-class is what?
More than one thing or another?
Could be three classes, could be a hundred.
We're going to look at steps at modeling with CNNs in PyTorch.
So we're going to create a convolutional network with PyTorch.
We're going to pick a last function and optimize it to suit our problem.
We're going to train a model, training a model a model.
A little bit of a typo there.
And then we're going to evaluate a model, right?
So we might have typos with our text, but we'll have less typos in the code.
And how are we going to do this?
Well, we could do it cook, so we could do it chemis.
Well, we're going to do it a little bit of both.
Part art, part science.
But since this is a machine learning cooking show, we're going to be cooking up lots of code.
So in the next video, we're going to cover the inputs and outputs of a computer vision problem.
I'll see you there.
So in the last video, we covered what we're going to cover, broadly.
And we saw some examples of what computer vision problems are.
Essentially, anything that you're able to see, you can potentially turn into a computer vision problem.
And we're going to be cooking up lots of machine learning, or specifically pie torch, computer vision code.
You see I fixed that typo.
Now let's talk about what the inputs and outputs are of a typical computer vision problem.
So let's start with a multi-classification example.
And so we wanted to take photos of different images of food and recognize what they were.
So we're replicating the functionality of Nutrify.
So take a photo of food and learn about it.
So we might start with a bunch of food images that have a height and width of some sort.
So we have width equals 224, height equals 224, and then they have three color channels.
Why three?
Well, that's because we have a value for red, green and blue.
So if we look at this up, if we go red, green, blue image format.
So 24-bit RGB images.
So a lot of images or digital images have some value for a red pixel, a green pixel and a blue pixel.
And if you were to convert images into numbers, they get represented by some value of red, some value of green and some value of blue.
That is exactly the same as how we'd represent these images.
So for example, this pixel here might be a little bit more red, a little less blue, and a little less green because it's close to orange.
And then we convert that into numbers.
So what we're trying to do here is essentially what we're trying to do with all of the data that we have with machine learning is represented as numbers.
So the typical image format to represent an image because we're using computer vision.
So we're trying to figure out what's in an image.

Section 143: Main Topics

Key Topics:

- And this would be the inputs to our machine learning algorithm
- And then you might fashion this machine learning algorithm to output the exact shapes that you want
- Now it might not always get it right because after all, that's what machine learning is

▶ 📄 Click to view detailed content

```
The typical way to represent that is in a tensor that has a value for the height,
width and color channels.
And so we might numerically encode these.
In other words, represent our images as a tensor.
And this would be the inputs to our machine learning algorithm.
And in many cases, depending on what problem you're working on, an existing
algorithm already exists for many of the most popular computer vision problems.
And if it doesn't, you can build one.
And then you might fashion this machine learning algorithm to output the exact
shapes that you want.
In our case, we want three outputs.
We want one output for each class that we have.
We want a prediction probability for sushi.
We want a prediction probability for steak.
And we want a prediction probability for pizza.
Now in our case, in this iteration, looks like our model got one of them wrong
because the highest value was assigned to the wrong class here.
So for the second image, it assigned a prediction probability of 0.81 for sushi.
Now, keep in mind that you could change these classes to whatever your particular
problem is.
I'm just simplifying this and making it three.
You could have a hundred.
You could have a thousand.
You could have five.
It's just, it depends on what you're working with.
And so we might use these predicted outputs to enhance our app.
So if someone wants to take a photo of their plate of sushi, our app might say,
hey, this is a photo of sushi.
Here's some information about those, the sushi rolls or the same for steak, the
same for pizza.
Now it might not always get it right because after all, that's what machine
learning is.
It's probabilistic.
So how would we improve these results here?
Well, we could show our model more and more images of sushi steak and pizza
so that it builds up a better internal representation of said images.
So when it looks at images it's never seen before or images outside its training
data set,
it's able to get better results.
```

But just keep in mind this whole process is similar no matter what computer vision problem you're working with.
You need a way to numerically encode your information.
You need a machine learning model that's capable of fitting the data
in the way that you would like it to be fit in our case classification.
You might have a different type of model if you're working with object detection,
a different type of model if you're working with segmentation.
And then you need to fashion the outputs in a way that best suit your problem as well.
So let's push forward.
Oh, by the way, the model that often does this is a convolutional neural network.
In other words, a CNN.
However, you can use many other different types of machine learning algorithms here.
It's just that convolutional neural networks typically perform the best with image data.
Although with recent research, there is the transformer architecture or deep learning model
that also performs fairly well or very well with image data.
So just keep that in mind going forward.
But for now we're going to focus on convolutional neural networks.
And so we might have input and output shapes because remember one of the chief machine learning problems
is making sure that your tensor shapes line up with each other, the input and output shapes.
So if we encoded this image of stake here, we might have a dimensionality of batch size
with height color channels.
And now the ordering here could be improved.
It's usually height then width.
So alphabetical order.
And then color channels last.
So we might have the shape of none, two, two, four, two, four, three.
Now where does this come from?
So none could be the batch size.
Now it's none because we can set the batch size to whatever we want, say for example 32.
Then we might have a height of two to four and a width of two to four and three color channels.
Now height and width are also customizable.
You might change this to be 512 by 512.
What that would mean is that you have more numbers representing your image.
And in sense would take more computation to figure out the patterns because there is simply more information encoded in your image.
But two, two, four, two, four is a common starting point for images.
And then 32 is also a very common batch size, as we've seen in previous videos.
But again, this could be changed depending on the hardware you're using, depending on the model you're using.
You might have a batch size to 64.
You might have a batch size of 512.
It's all problem specific.
And that's this line here.
These will vary depending on the problem you're working on.
So in our case, our output shape is three because we have three different classes for now.
But again, if you have a hundred, you might have an output shape of a hundred.
If you have a thousand, you might have an output shape of a thousand.

```
The same premise of this whole pattern remains though.
Numerically encode your data, feed it into a model, and then make sure the output
shape fits your specific problem.
And so, for this section, Computer Vision with PyTorch, we're going to be building
CNNs to do this part.
We're actually going to do all of the parts here, but we're going to focus on
building a convolutional neural network
to try and find patterns in data, because it's not always guaranteed that it will.
Finally, let's look at one more problem.
Say you had grayscale images of fashion items, and you have quite small images.
They're only 28 by 28.
The exact same pattern is going to happen.
You numerically represent it, use it as inputs to a machine learning algorithm,
```

Section 144: Main Topics

Key Topics:

- and then hopefully your machine learning algorithm outputs the right type of clothing that it is
- So hopefully our machine learning algorithm can recognize what's going on in these images
- Well, because you come across a lot of different representations of data full stop, but particularly image data in PyTorch and other libraries, many libraries expect color channels last

▶ 📄 Click to view detailed content

```
and then hopefully your machine learning algorithm outputs the right type of
clothing that it is.
In this case, it's a t-shirt.
But I've got dot dot dot here because we're going to be working on a problem that
uses ten different types of items of clothing.
And the images are grayscale, so there's not much detail.
So hopefully our machine learning algorithm can recognize what's going on in these
images.
There might be a boot, there might be a shirt, there might be pants, there might be
a dress,
etc, etc.
But we numerically encode our images into dimensionality of batch size, height with
color channels.
This is known as NHWC, or number of batches, or number of images in a batch, height
with C, or color channels.
This is color channels last.
Why am I showing you two forms of this?
Do you notice color channels in this one is color channels first?
So color channels height width?
```

```
Well, because you come across a lot of different representations of data full stop,
but particularly image data in PyTorch and other libraries,
many libraries expect color channels last.
However, PyTorch currently at the time of recording this video may change in the
future,
defaults to representing image data with color channels first.
Now this is very important because you will get errors if your dimensionality is in
the wrong order.
And so there are ways to go in between these two, and there's a lot of debate of
which format is the best.
It looks like color channels last is going to win over the long term, just because
it's more efficient,
but again, that's outside the scope, but just keep this in mind.
We're going to write code to interact between these two, but it's the same data
just represented in different order.
And so we could rearrange these shapes to how we want color channels last or color
channels first.
And once again, the shapes will vary depending on the problem that you're working
on.
So with that being said, we've covered the input and output shapes.
How are we going to see them with code?
Well, of course we're going to be following the PyTorch workflow that we've done.
So we need to get our data ready, turn it into tenses in some way, shape or form.
We can do that with taught division transforms.
Oh, we haven't seen that one yet, but we will.
We can use torchutilsdata.datasetutils.data.data loader.
We can then build a model or pick a pre-trained model to suit our problem.
We've got a whole bunch of modules to help us with that, torchNN module,
torchvision.models.
And then we have an optimizer and a loss function.
We can evaluate the model using torch metrics, or we can code our own metric
functions.
We can of course improve through experimentation, which we will see later on,
which we've actually done that, right?
We've done improvement through experimentation.
We've tried different models, we've tried different things.
And then finally, we can save and reload our trained model if we wanted to use it
elsewhere.
So with that being said, we've covered the workflow.
This is just a high-level overview of what we're going to code.
You might be asking the question, what is a convolutional neural network, or a CNN?
Let's answer that in the next video.
I'll see you there.
Welcome back.
In the last video, we saw examples of computer vision input and output shapes.
And we kind of hinted at the fact that convolutional neural networks are deep
learning models, or CNNs,
that are quite good at recognizing patterns in images.
So we left off the last video with the question, what is a convolutional neural
network?
And where could you find out about that?
What is a convolutional neural network?
Here's one way to find out.
And I'm sure, as you've seen, there's a lot of resources for such things.
A comprehensive guide to convolutional neural networks.
Which one of these is the best?
Well, it doesn't really matter.
```

```
The best one is the one that you understand the best.
So there we go.
There's a great video from Code Basics.
I've seen that one before, simple explanation of convolutional neural network.
I'll leave you to research these things on your own.
And if you wanted to look at images, there's a whole bunch of images.
I prefer to learn things by writing code.
Because remember, this course is code first.
As a machine learning engineer, 99% of my time is spent writing code.
So that's what we're going to focus on.
But anyway, here's the typical architecture of a CNN.
In other words, a convolutional neural network.
If you hear me say CNN, I'm not talking about the news website.
In this course, I'm talking about the architecture convolutional neural network.
So this is some PyTorch code that we're going to be working towards building.
But we have some hyperparameters slash layer types here.
We have an input layer.
So we have an input layer, which takes some in channels, and an input shape.
Because remember, it's very important in machine learning and deep learning to line
up your
input and output shapes of whatever model you're using, whatever problem you're
working with.
Then we have some sort of convolutional layer.
Now, what might happen in a convolutional layer?
Well, as you might have guessed, as what happens in many neural networks, is that
the layers
perform some sort of mathematical operation.
Now, convolutional layers perform convolving window operation across an image or
across
a tensor.
And discover patterns using, let's have a look, actually.
Let's go, nn.com2d.
There we go.
This is what happens.
So the output of our network equals a bias plus the sum of the weight tensor over
the
convolutional channel out, okay, times input.
Now, if you want to dig deeper into what is actually going on here, you're more
than welcome to
```

Section 145: Main Topics

Key Topics:

- You will not only start to learn to build them, you will just start to learn to use them, as we'll see later on in the transfer learning section of the course
- And then I'm going to name this one, this is going to be 03 PyTorch computer vision

- So just so it has the video tag, because if we go in here, if we go video notebooks of the PyTorch deep learning repo, the video notebooks are stored in here

▶ 📄 Click to view detailed content

```
do that.
But we're going to be writing code that leverages the torch nn.com2d.
And we're going to fix up all of these hyperparameters here so that it works with
our problem.
Now, what you need to know here is that this is a bias term.
We've seen this before.
And this is a weight matrix.
So a bias vector typically and a weight matrix.
And they operate over the input.
But we'll see these later on with code.
So just keep that in mind.
This is what's happening.
As with every layer in a neural network, some form of operation is happening on our
input
data.
These operations happen layer by layer until eventually, hopefully, they can be
turned into
some usable output.
So let's jump back in here.
Then we have an hidden activation slash nonlinear activation because why do we use
nonlinear
activations?
Well, it's because if our data was nonlinear, non-straight lines, we need the help
of straight
and non-straight lines to model it, to draw patterns in it.
Then we typically have a pooling layer.
And I want you to take this architecture.
I've said typical here for a reason because these type of architectures are
changing all
the time.
So this is just one typical example of a CNN.
It's about as basic as a CNN as you can get.
So over time, you will start to learn to build more complex models.
You will not only start to learn to build them, you will just start to learn to use
them,
as we'll see later on in the transfer learning section of the course.
And then we have an output layer.
So do you notice the trend here?
We have an input layer and then we have multiple hidden layers that perform some
sort of mathematical
operation on our data.
And then we have an output slash linear layer that converts our output into the
ideal shape
that we'd like.
So we have an output shape here.
And then how does this look in process?
While we put in some images, they go through all of these layers here because we've
used
an end up sequential.
And then hopefully this forward method returns x in a usable status or usable state
```

that
we can convert into class names.
And then we could integrate this into our computer vision app in some way, shape or form.
And here's the asterisk here.
Note, there are almost an unlimited amount of ways you could stack together a convolutional
neural network.
This slide only demonstrates one.
So just keep that in mind, only demonstrates one.
But the best way to practice this sort of stuff is not to stare at a page.
It's to if and out, code it out.
So let's code, I'll see you in Google CoLab.
Welcome back.
Now, we've discussed a bunch of fundamentals about computer vision problems and convolutional
neural networks.
But rather than talk to more slides, well, let's start to code them out.
I'm going to meet you at colab.research.google.com.
She's going to clean up some of these tabs.
And I'm going to start a new notebook.
And then I'm going to name this one, this is going to be 03 PyTorch computer vision.
And I'm going to call mine video.
So just so it has the video tag, because if we go in here, if we go video notebooks of
the PyTorch deep learning repo, the video notebooks are stored in here.
They've got the underscore video tag.
So the video notebooks have all of the code I write exactly in the video.
But there are some reference notebooks to go along with it.
Let me just write a heading here, PyTorch computer vision.
And I'll put a resource here, see reference notebook.
Now, of course, this is the one that's the ground truth.
It's got all of the code that we're going to be writing.
I'm going to put that in here.
Explain with text and images and whatnot.
And then finally, as we got see reference online book.
And that is at learnpytorch.io at section number three, PyTorch computer vision.
I'm going to put that in there.
And then I'm going to turn this into markdown with command mm.
Beautiful.
So let's get started.
I'm going to get rid of this, get rid of this.
How do we start this off?
Well, I believe there are some computer vision libraries that you should be aware of.
Computer vision libraries in PyTorch.
So this is just going to be a text based cell.
But the first one is torch vision, which is the base domain library for PyTorch computer vision.
So if we look up torch vision, what do we find?
We have torch vision 0.12.
That's the version that torch vision is currently up to at the time of recording this.
So in here, this is very important to get familiar with if you're working on computer vision problems.
And of course, in the documentation, this is just another tidbit.

```
We have torch audio for audio problems.
We have torch text for text torch vision, which is what we're working on torch rack
for recommendation systems
torch data for dealing with different data pipelines torch serve, which is for
serving PyTorch models
and PyTorch on XLA.
So I believe that stands for accelerated linear algebra devices.
You don't have to worry about these ones for now.
We're focused on torch vision.
However, if you would like to learn more about a particular domain, this is where
you would go to learn more.
So there's a bunch of different stuff that's going on here.
Transforming and augmenting images.
So fundamentally, computer vision is dealing with things in the form of images.
Even a video gets converted to an image.
We have models and pre-trained weights.
So as I referenced before, you can use an existing model that works on an existing
computer vision problem for your own problem.
```

Section 146: Main Topics

Key Topics:

- We're going to cover that in section, I think it's six, for transfer learning
- So PyTorch is really, really good for computer vision
- All of the links for these, by the way, is in the book version of the course PyTorch Computer Vision

▶ 📄 Click to view detailed content

```
We're going to cover that in section, I think it's six, for transfer learning.
And then we have data sets, which is a bunch of computer vision data sets, utils,
operators, a whole bunch of stuff here.
So PyTorch is really, really good for computer vision.
I mean, look at all the stuff that's going on here.
But that's enough talking about it.
Let's just put it in here.
Torch vision. This is the main one.
I'm not going to link to all of these.
All of the links for these, by the way, is in the book version of the course
PyTorch Computer Vision.
And we have what we're going to cover.
And finally, computer vision libraries in PyTorch.
Torch vision, data sets, models, transforms, et cetera.
But let's just write down the other ones.
So we have torch vision, not data sets, something to be aware of.
So get data sets and data loading functions for computer vision here.
Then we have torch vision.
```

And from torch vision, models is get pre-trained computer vision.
So when I say pre-trained computer vision models, we're going to cover this more in transfer learning, as I said.
Pre-trained computer vision models are models that have been already trained on some existing vision data
and have trained weights, trained patterns that you can leverage for your own problems,
that you can leverage for your own problems.
Then we have torch vision.transforms.
And then we have functions for manipulating your vision data, which is, of course, images to be suitable for use with an ML model.
So remember, what do we have to do when we have image data or almost any kind of data?
For machine learning, we have to prepare it in a way so it can't just be pure images, so that's what transforms help us out with.
Transforms helps to turn our image data into numbers so we can use it with a machine learning model.
And then, of course, we have some, these are the torch utils.
This is not vision specific, it's entirety of PyTorch specific, and that's data set.
So if we wanted to create our own data set with our own custom data, we have the base data set class for PyTorch.
And then we have finally torch utils data.
These are just good to be aware of because you'll almost always use some form of data set slash data loader with whatever PyTorch problem you're working on.
So this creates a Python iterable over a data set.
Wonderful.
I think these are most of the libraries that we're going to be using in this section.
Let's import some of them, hey, so we can see what's going on.
Let's go import PyTorch.
Import PyTorch.
So import torch.
We're also going to get NN, which stands for neural network.
What's in NN?
Well, in NN, of course, we have lots of layers, lots of loss functions, a whole bunch of different stuff for building neural networks.
We're going to also import torch vision.
And then we're going to go from torch vision.
Import data sets because we're going to be using data sets later on to get a data set to work with from torch vision.
Well, import transforms.
You could also go from torch vision dot transforms import to tensor.
This is one of the main ones you'll see for computer vision problems to tensor.
You can imagine what it does.
But let's have a look.
Transforms to tensor.
Transforming and augmenting images.
So look where we are.
We're in pytorch.org slash vision slash stable slash transforms.
Over here.
So we're in the torch vision section.
And we're just looking at transforming and augmenting images.
So transforming.
What do we have?
Transforms are common image transformations of our and the transforms module.
They can be trained together using compose.

```
Beautiful.
So if we have two tensor, what does this do?
Convert a pill image on NumPy and the array to a tensor.
Beautiful.
That's what we want to do later on, isn't it?
Well, this is kind of me giving you a spoiler is we want to convert our images into
tensors so that we can use those with our models.
But there's a whole bunch of different transforms here and actually one of your
extra curriculum is to be to read through each of these packages for 10 minutes.
So that's about an hour of reading, but it will definitely help you later on if you
get familiar with using the pytorch documentation.
After all, this course is just a momentum builder.
We're going to write heaves of pytorch code.
But fundamentally, you'll be teaching yourself a lot of stuff by reading the
documentation.
Let's keep going with this.
Where were we up to?
When we're getting familiar with our data, mapplotlib is going to be fundamental
for visualization.
Remember, the data explorer's motto, visualize, visualize, visualize, become one
with the data.
So we're going to import mapplotlib.pyplot as PLT.
And then finally, let's check the versions.
So print torch.version or underscore, underscore version and print torch vision.
So by the time you watch this, there might be a newer version of each of these
modules out.
If there's any errors in the code, please let me know.
But this is just a bare minimum version that you'll need to complete this section.
I believe at the moment, Google Colab is running 1.11 for torch and maybe 1.10.
We'll find out in a second.
It just connected.
So we're importing pytorch.
Okay, there we go.
So my pytorch version is 1.10 and it's got CUDA available and torch vision is 0.11.
So just make sure if you're running in Google Colab, if you're running this at a
later date,
you probably have at minimum these versions, you might even have a later version.
So these are the minimum versions required for this upcoming section.
```

Section 147: Main Topics

Key Topics:

- So we've covered the base computer vision libraries in pytorch
- So in the last video, we covered some of the fundamental computer vision libraries in pytorch
- And then of course, we've got torch utils dot data dot data set, which is the base data set class for pytorch and data loader, which creates a Python irritable over a data set

▶ 📄 Click to view detailed content

```
So we've covered the base computer vision libraries in pytorch.
We've got them ready to go.
How about in the next video, we cover getting a data set.
I'll see you there.
Welcome back.
So in the last video, we covered some of the fundamental computer vision libraries
in pytorch.
The main one being torch vision and then modules that stem off torch vision.
And then of course, we've got torch utils dot data dot data set, which is the base
data set class for pytorch
and data loader, which creates a Python irritable over a data set.
So let's begin where most machine learning projects do.
And that is getting a data set, getting a data set.
I'm going to turn this into markdown.
And the data set that we're going to be used to demonstrating some computer vision
techniques is fashion amnest.
Which is a take of the data set we'll be using is fashion amnest, which is a take
on the original amnest data set,
amnest database, which is modified national institute of standards and technology
database, which is kind of like the hello world
in machine learning and computer vision, which is these are sample images from the
amnest test data set,
which are grayscale images of handwritten digits.
So this, I believe was originally used for trying to find out if you could use
computer vision at a postal service
to, I guess, recognize post codes and whatnot.
I may be wrong about that, but that's what I know.
Yeah, 1998.
So all the way back at 1998, how cool is that?
So this was basically where convolutional neural networks were founded.
I'll let you read up on the history of that.
But neural network started to get so good that this data set was quite easy for
them to do really well.
And that's when fashion amnest came out.
So this is a little bit harder if we go into here.
This is by Zalando research fashion amnest.
And it's of grayscale images of pieces of clothing.
So like we saw before the input and output, what we're going to be trying to do is
turning these images of clothing into numbers
and then training a computer vision model to recognize what the different styles of
clothing are.
And here's a dimensionality plot of all the different items of clothing.
Visualizing where similar items are grouped together, there's the shoes and
whatnot.
Is this interactive?
Oh no, it's a video.
Excuse me.
There we go.
To serious machine learning researchers.
We are talking about replacing amnest.
Amnest is too easy.
Amnest is overused.
Amnest cannot represent modern CV tasks.
So even now fashion amnest I would say has also been pretty much sold, but it's a
```

good way to get started.
Now, where could we find such a data set?
We could download it from GitHub.
But if we come back to the taught division documentation, have a look at data sets.
We have a whole bunch of built-in data sets.
And remember, this is your extra curricular to read through these for 10 minutes or so each.
But we have an example.
We could download ImageNet if we want.
We also have some base classes here for custom data sets.
We'll see that later on.
But if we scroll through, we have image classification data sets.
Caltech 101.
I didn't even know what all of these are.
There's a lot here.
CFAR 100.
So that's an example of 100 different items.
So that would be a 100 class, multi-class classification problem.
CFAR 10 is 10 classes.
We have amnest.
We have fashion amnest.
Oh, that's the one we're after.
But this is basically what you would do to download a data set from taughtvision.datasets.
You would download the data in some way, shape, or form.
And then you would turn it into a data loader.
So ImageNet is one of the most popular or is probably the gold standard data set for computer vision evaluation.
It's quite a big data set.
It's got millions of images.
But that's the beauty of taught vision is that it allows us to download example data sets
that we can practice on.
I don't even perform research on from a built-in module.
So let's now have a look at the fashion amnest data set.
How might we get this?
So we've got some example code here, or this is the documentation.
taughtvision.datasets.fashion amnest.
We have to pass in a root.
So where do we want to download the data set?
We also have to pass in whether we want the training version of the data set
or whether we want the testing version of the data set.
Do we want to download it?
Yes or no?
Should we transform the data in any way shape or form?
So we're going to be downloading images through this function call or this class call.
Do we want to transform those images in some way?
What do we have to do to images before we can use them with a model?
We have to turn them into a tensor, so we might look into that in a moment.
And target transform is do we want to transform the labels in any way shape or form?
So often the data sets that you download from taughtvision.datasets
are pre formatted in a way that they can be quite easily used with PyTorch.
But that won't always be the case with your own custom data sets.
However, what we're about to cover is just important to get an idea of what the computer vision workflow is.

```
And then later on you can start to customize how you get your data in the right
format to be used with the model.
Then we have some different parameters here and whatnot.
Let's just rather than look at the documentation, if and down, code it out.
```

Section 148: Main Topics

Key Topics:

- This is what I was talking about how PyTorch defaults with a lot of transforms to CHW

▶ 📄 Click to view detailed content

```
So we'll be using fashion MNIST and we'll start by, I'm going to just put this
here, from taughtvision.datasets.
And we'll put the link there and we'll start by getting the training data.
Set up training data.
I'm just going to make some code cells here so that I can code in the middle of the
screen.
Set up training data. Training data equals data sets dot fashion MNIST.
Because recall, we've already from taughtvision.
We don't need to import this again, I'm just doing it for demonstration purposes,
but from taughtvision import data sets
so we can just call data sets dot fashion MNIST.
And then we're going to type in root.
See how the doc string comes up and tells us what's going on.
I personally find this a bit hard to read in Google Colab, so if I'm looking up the
documentation,
I like to just go into here.
But let's code it out.
So root is going to be data, so where to download data to.
We'll see what this does in a minute.
Then we're going to go train.
We want the training version of the data set.
So as I said, a lot of the data sets that you find in taughtvision.datasets
have been formatted into training data set and testing data set already.
So this Boolean tells us do we want the training data set?
So if that was false, we would get the testing data set of fashion MNIST.
Do we want to download it?
Do we want to download?
Yes, no.
So yes, we do. We're going to set that to true.
Now what sort of transform do we want to do?
So because we're going to be downloading images and what do we have to do to our
images
to use them with a machine-loading model, we have to convert them into tensors.
So I'm going to pass the transform to tensor, but we could also just go
torchvision.transforms.to tensor.
```

That would be the exact same thing as what we just did before.
And then the target transform, do we want to transform the labels?
No, we don't.
We're going to see how they come, or the target, sorry.
High torch, this is another way, another naming convention.
Often uses target for the target that you're trying to predict.
So using data to predict the target, which is I often use data to predict a label.
They're the same thing.
So how do we want to transform the data?
And how do we want to transform the labels?
And then we're going to do the same for the test data.
So we're going to go data sets.
You might know what to do here.
It's going to be the exact same code as above, except we're going to change one line.
We want to store it in data.
We want to download the training data set as false because we want the testing version.
Do we want to download it?
Yes, we do.
Do we want to transform it the data?
Yes, we do, we want to use to tensor to convert our image data to tensors.
And do we want to do a target transform?
Well, no, we don't.
We want to keep the label slash the targets as they are.
Let's see what happens when we run this.
Oh, downloading fashion, Evan is beautiful.
So this is going to download all of the labels.
What do we have?
Train images, train labels, lovely, test images, test labels, beautiful.
So that's how quickly we can get a data set by using torch vision data sets.
Now, if we have a look over here, we have a data folder because we set the root to be
data.
Now, if we look what's inside here, we have fashion MNIST, exactly what we wanted.
Then we have the raw, and then we have a whole bunch of files here, which torch vision has
converted into data sets for us.
So let's get out of that.
And this process would be much the same if we used almost any data set in here.
They might be slightly different depending on what the documentation says and depending
on what the data set is.
But that is how easy torch vision data sets makes it to practice on example computer vision
data sets.
So let's go back.
Let's check out some parameters or some attributes of our data.
How many samples do we have?
So we'll check the lengths.
So we have 60,000 training examples and 10,000 testing examples.
So what we're going to be doing is we're going to be building a computer vision model to
find patterns in the training data and then use those patterns to predict on the test
data.
And so let's see a first training example.

```
See the first training example.
So we can just index on the train data.
Let's get the zero index and then we're going to have a look at the image and the
label.
Oh my goodness.
A whole bunch of numbers.
Now you see what the two tensor has done for us?
So we've downloaded some images and thanks to this torch vision transforms to
tensor.
How would we find the documentation for this?
Well, we could go and see what this does transforms to tensor.
We could go to tensor.
There we go.
What does this do?
Convert a pill image.
So that's Python image library image on NumPy array to a tensor.
This transform does not support torch script.
So converts a pill image on NumPy array height with color channels in the range 0
to 255
to a torch float tensor of shape.
See here?
This is what I was talking about how PyTorch defaults with a lot of transforms to
CHW.
So color channels first height then width in that range of zero to one.
So typically red, green and blue values are between zero and 255.
```

Section 149: Main Topics

Key Topics:

- However, some other machine learning libraries prefer height, width, then color channels
- So if our machine learning model predicted nine or class nine, we can convert that to ankle boot using this attribute of the train data
- And part of becoming one with the data is, of course, checking the input output shapes of it

▶ 📄 Click to view detailed content

```
But neural networks like things between zero and one.
And in this case, it is now in the shape of color channels first, then height, then
width.
However, some other machine learning libraries prefer height, width, then color
channels.
Just keep that in mind.
We're going to see this in practice later on.
So we've got an image.
```

```
We've got a label.
Let's check out some more details about it.
Remember how we discussed?
Oh, there's our label, by the way.
So nine, we can go traindata.classes, find some information about our class names.
Class names.
Beautiful.
So number nine would be 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
So this particular tensor seems to relate to an ankle boot.
How would we find that out?
Well, one second.
I'm just going to show you one more thing, class to IDX.
Let's go traindata.class to IDX.
What does this give us?
Class to IDX.
This is going to give us a dictionary of different labels and their corresponding
index.
So if our machine learning model predicted nine or class nine, we can convert that
to
ankle boot using this attribute of the train data.
There are more attributes that you can have a look at if you like.
You can go traindata.dot, then I just push tab to find out a bunch of different
things.
You can go data.
That'll be the images, and then I believe you can also go targets.
So targets, that's all the labels, which is one big long tensor.
Now let's check the shape.
Check the shape of our image.
So image.shape and label.shape.
What are we going to get from that?
Oh, label doesn't have a shape.
Why is that?
Well, because it's only an integer.
So oh, beautiful.
Look at that.
So our image shape is we have a color channel of one.
So let me print this out in something prettier, print image shape, which is going
to be image
shape.
Remember how I said it's very important to be aware of the input and output shapes
of
your models and your data.
It's all part of becoming one with the data.
So that is what our image shape is.
And then if we go next, this is print image label, which is label, but we'll index
on
class names for label.
And then we'll do that wonderful.
So our image shape is currently in the format of color channels height width.
We got a bunch of different numbers that's representing our image.
It's black and white.
It only has one color channel.
Why do you think it only has one color channel?
Because it's black and white, so if we jump back into the keynote, fashion, we've
already
discussed this, grayscale images have one color channel.
So that means that for black, the pixel value is zero.
```

And for white, it's some value for whatever color is going on here.
So if it's a very high number, say it's one, it's going to be pure white.
If it's like 0.001, it might be a faint white pixel.
But if it's exactly zero, it's going to be black.
So color images have three color channels for red, green and blue, grayscale have one
color channel.
But I think we've done enough of visualizing our images as numbers.
How about in the next video, we visualize our image as an image?
I'll see you there.
Welcome back.
So in the last video, we checked the input output shapes of our data, and we downloaded
the fashion MNIST data set, which is comprised of images or grayscale images of T-shirts,
trousers, pullovers, dress, coat, sandal, shirt, sneaker, bag, ankle boot.
Now we want to see if we can build a computer vision model to decipher what's going on in
fashion MNIST.
So to separate, to classify different items of clothing based on their numerical representation.
And part of becoming one with the data is, of course, checking the input output shapes
of it.
So this is a fashion MNIST data set from Zalando Research.
Now if you recall, why did we look at our input and output shapes?
Well, this is what we looked at before.
We have 28 by 28 grayscale images that we want to represent as a tensor.
We want to use them as input into a machine learning algorithm, typically a computer vision
algorithm, such as a CNN.
And we want to have some sort of outputs that are formatted in the ideal shape that we'd
like.
So in our case, we have 10 different types of clothing.
So we're going to have an output shape of 10, but our input shape is what?
So by default, PyTorch turns tensors into color channels first.
So we have an input shape of none, one, 28, 28.
So none is going to be our batch size, which of course we can set that to whatever we'd
like.
Now input shape format is in NCHW, or in other words, color channels first.
But just remember, if you're working with some other machine learning libraries, you
may want to use color channels last.
So let's have a look at where that might be the case.
We're going to visualize our images.
So I make a little heading here, 1.2.
Now this is all part of becoming one with the data.
In other words, understanding its input and output shapes, how many samples there are,
what they look like, visualize, visualize, visualize.
Let's import mapplotlib.
I'm just going to add a few code cells here, import mapplotlib.pyplot as PLT.
Now let's create our image and label is our train data zero, and we're going to print
the image shape so we can understand what inputs are going into our mapplotlib

```
function.
And then we're going to go plot.imshow, and we're going to pass in our image and
see
what happens, because recall what does our image look like, image?
```

Section 150: Main Topics

Key Topics:

- Now as I said, this is one of the most common errors in machine learning is a shape issue

▶ 📄 Click to view detailed content

```
Our image is this big tensor of numbers.
And we've got an image shape, 128, 128.
Now what happens if we call plot.imshow?
What happens there?
Oh, we get an error in valid shape, 128, 128 for image data.
Now as I said, this is one of the most common errors in machine learning is a shape
issue.
So the shape of your input tensor doesn't match the expected shape of that tensor.
So this is one of those scenarios where our data format, so color channels first,
doesn't
match up with what mapplotlib is expecting.
So mapplotlib expects either just height and width, so no color channel for gray
style
images, or it also expects the color channels to be last.
So we'll see that later on, but for grayscale, we can get rid of that extra
dimension by
passing in image.squeeze.
So do you recall what squeeze does?
It's going to remove that singular dimension.
If we have a look at what goes on now, beautiful, we get an ankle boot.
Well, that's a very pixelated ankle boot, but we're only dealing with 28 by 28
pixels,
so not a very high definition image.
Let's add the title to it.
We're going to add in the label.
Beautiful.
So we've got the number nine here.
So where if we go up to here, that's an ankle boot.
Now let's plot this in grayscale.
How might we do that?
We can do the same thing.
We can go plotplt.imshow.
We're going to pass in image.squeeze.
And we're going to change the color map, C map equals gray.
So in mapplotlib, if you ever have to change the colors of your plot, you want to
```

look
into the C map property or parameter, or sometimes it's also shortened to just C.
But in this case, M show is C map, and we want to plot title, and we're going to pull
it in class names and the label integer here.
So we have a look at it now.
We have an ankle boot, and we can remove the accesses to if we wanted to plot.access,
and turn that off.
That's going to remove the access.
So there we go.
That's the type of images that we're dealing with.
But that's only a singular image.
How about we harness the power of randomness and have a look at some random images from
our data set?
So how would we do this?
Let's go plot more images.
We'll set a random seed.
So you and I are both looking at as similar as possible images, 42.
Now we'll create a plot by calling plot.figure, and we're going to give it a size.
We might create a nine by nine grid.
So we want to see nine random images from our data set.
So rows, calls, or sorry, maybe we'll do four by four.
That'll give us 16.
We're going to go four i in range, and we're going to go one to rows times columns plus
one.
So we can print i.
What's that going to give us?
We want to see 16 images.
Oh, they're about.
So 16 random images, but used with a manual C to 42 of our data set.
This is one of my favorite things to do with any type of data set that I'm looking
at, whether it be text, image, audio, doesn't matter.
I like to randomly have a look at a whole bunch of samples at the start so that I can
become one with the data.
With that being said, let's use this loop to grab some random indexes.
We can do so using tortures, rand, int, so random integer between zero and length of
the training data.
This is going to give us a random integer in the range of zero and however many training
samples we have, which in our case is what, 60,000 or thereabouts.
So we want to create the size of one, and we want to get the item from that so that we
have a random index.
What is this going to give us?
Oh, excuse me, maybe we print that out.
There we go.
So we have random images.
Now, because we're using manual seed, it will give us the same numbers every time.
So we have three, seven, five, four, two, three, seven, five, four, two.
And then if we just commented out the random seed, we'll get different numbers every time.
But this is just to demonstrate, we'll keep the manual seed there for now.

```
You can comment that out if you want different numbers or different images,
different indexes
each time.
So we'll create the image and the label by indexing on the training data at the
random
index that we're generating.
And then we'll create our plot.
So Fig or we'll add a subplot, Fig add subplot, and we're going to go rows, calls,
I.
So at the if index, we're going to add a subplot.
Remember, we set rows and columns up to here.
And then we're going to go PLT dot in show, we're going to show what we're going to
show
our image, but we have to squeeze it to get rid of that singular dimension as the
color
channel.
Otherwise, we end up with an issue with map plot lib.
We're going to use a color map of gray.
So it looks like the image we plotted above.
And then for our title, it's going to be our class names indexed with our label.
And then we don't want the accesses because that's going to clutter up our plot.
Let's see what this looks like.
Oh my goodness, look at that.
It worked first.
Go.
Usually visualizations take a fair bit of trial and error.
So we have ankle boots, we have shirts, we have bags, we have ankle boots, sandal,
shirt,
pull over.
```

Section 151: Main Topics

Key Topics:

- But the whole premise of building machine learning models to do this would be could you write a program that would take in the shapes of these images and figure out, write a rule-based program that would go, hey, if it's looked like a rectangle with a buckle in the middle, it's probably a bag
- I mean, you probably could after a while, but I prefer to write machine learning algorithms to figure out patterns and data
- So right now, our data is in the form of PyTorch data sets

▶ 📄 Click to view detailed content

```
Oh, do you notice something about the data set right now, pull over and shirt?
To me, they look quite similar.
Do you think that will cause an issue later on when our model is trying to predict
```

```
between
a pull over and a shirt?
How about if we look at some more images?
We'll get rid of the random seed so we can have a look at different styles.
So have a sandal ankle boot coat, t-shirt, top, shirt, oh, is that a little bit
confusing
that we have a class for t-shirt and top and shirt?
Like I'm not sure about you, but what's the difference between a t-shirt and a
shirt?
This is just something to keep in mind as a t-shirt and top, does that look like it
could
be maybe even a dress?
Like the shape is there.
So this is just something to keep in mind going forward.
The chances are if we get confused on our, like you and I looking at our data set,
if
we get confused about different samples and what they're labeled with, our model
might
get confused later on.
So let's have a look at one more and then we'll go into the next video.
So we have sneaker, trouser, shirt, sandal, dress, pull over, bag, bag, t-shirt,
oh, that's
quite a difficult one.
It doesn't look like there's even much going on in that image.
But the whole premise of building machine learning models to do this would be could
you
write a program that would take in the shapes of these images and figure out, write
a rule-based
program that would go, hey, if it's looked like a rectangle with a buckle in the
middle,
it's probably a bag?
I mean, you probably could after a while, but I prefer to write machine learning
algorithms
to figure out patterns and data.
So let's start moving towards that.
We're now going to go on figuring out how we can prepare this data to be loaded
into
a model.
I'll see you there.
All right, all right, all right.
So we've got 60,000 images of clothing that we'd like to build a computer vision
model
to classify into 10 different classes.
And now that we've visualized a fair few of these samples, do you think that we
could
model these with just linear lines, so straight lines, or do you think we'll need a
model
with nonlinearity?
So I'm going to write that down.
So do you think these items of clothing images could be modeled with pure linear
lines, or
do you think we'll need nonlinearity?
Don't have to answer that now.
We could test that out later on.
You might want to skip ahead and try to build a model yourself with linear lines or
nonlinearities.
We've covered linear lines and nonlinearities before, but let's now start to
```

prepare our
data even further to prepare data loader.
So right now, our data is in the form of PyTorch data sets.
So let's have a look at it.
Same data.
There we go.
So we have data set, which is of fashion MNIST.
And then if we go test data, we see a similar thing except we have a different number of
data points.
We have the same transform on each, we've turned them into tenses.
So we want to convert them from a data set, which is a collection of all of our data,
into a data loader.
Paul, that a data loader, turns our data set into a Python iterable.
So I'm going to turn this into Markdown, beautiful.
More specifically, specific Galilee, can I spell right?
I don't know, we want to just code right, we're not here to learn spelling.
We want to turn our data into batches, or mini batches.
Why would we do this?
Well, we may get away with it by building a model to look at all 60,000 samples of our
current data set, because it's quite small.
It's only comprised of images of 28 by 28 pixels.
And when I say quite small, yes, 60,000 images is actually quite small for a deep learning
scale data set.
Modern data sets could be in the millions of images.
But if our computer hardware was able to look at 60,000 samples of 28 by 28 at one time,
it would need a fair bit of memory.
So we have RAM space up here, we have GPU memory, we have compute memory.
But chances are that it might not be able to store millions of images in memory.
So what you do is you break a data set from say 60,000 into groups of batches or mini
batches.
So we've seen batch size before, why would we do this?
Well, one, it is more computationally efficient, as in your computing hardware may not be able
to look store in memory at 60,000 images in one hit.
So we break it down to 32 images at a time.
This would be batch size of 32.
Now again, 32 is a number that you can change.
32 is just a common batch size that you'll see with many beginner style problems.
As you go on, you'll see different batch sizes.
This is just to exemplify the concept of mini batches, which is very common in deep learning.
And why else would we do this?
The second point or the second main point is it gives our neural network more chances
to update its gradients per epoch.
So what I mean by this, this will make more sense when we write a training loop.
But if we were to just look at 60,000 images at one time, we would per epoch.
So per iteration through the data, we would only get one update per epoch across our entire
data set.

```
Whereas if we look at 32 images at a time, our neural network updates its internal
states,
```

Section 152: Main Topics

Key Topics:

- So yeah, large-scale machine learning, mini batch gradient descent, mini batch gradient descent
- And this principle, by the way, preparing a data loader goes the same for not only images, but for text, for audio, whatever sort of data you're working with, mini batches will follow you along or batches of data will follow you along throughout a lot of different deep learning problems
- This is some extra curriculum for you too, by the way, is to read this data page torch utils not data because no matter what problem you're going with with deep learning or pytorch, you're going to be working with data

▶ 📄 Click to view detailed content

```
its weights, every 32 images, thanks to the optimizer.
This will make a lot more sense once we write our training loop.
But these are the two of the main reasons for turning our data into mini batches in
the
form of a data loader.
Now if you'd like to learn more about the theory behind this, I would highly
recommend
looking up Andrew Org mini batches.
There's a great lecture on that.
So yeah, large-scale machine learning, mini batch gradient descent, mini batch
gradient
descent.
Yeah, that's what it's called mini batch gradient descent.
If you look up some results on that, you'll find a whole bunch of stuff.
I might just link this one, I'm going to pause that, I'm going to link this in
there.
So for more on mini batches, see here.
Now to see this visually, I've got a slide prepared for this.
So this is what we're going to be working towards.
There's our input and output shapes.
We want to create batch size of 32 across all of our 60,000 training images.
And we're actually going to do the same for our testing images, but we only have
10,000
testing images.
So this is what our data set's going to look like, batched.
So we're going to write some code, namely using the data loader from
torch.util.data.
```

We're going to pass it a data set, which is our train data.
We're going to give it a batch size, which we can define as whatever we want.
For us, we're going to use 32 to begin with.
And we're going to set shuffle equals true if we're using the training data.
Why would we set shuffle equals true?
Well, in case our data set for some reason has order, say we had all of the pants
images
in a row, we had all of the T-shirt images in a row, we had all the sandal images
in
a row.
We don't want our neural network to necessarily remember the order of our data.
We just want it to remember individual patterns between different classes.
So we shuffle up the data, we mix it, we mix it up.
And then it looks something like this.
So we might have batch number zero, and then we have 32 samples.
Now I ran out of space when I was creating these, but we got, that was fun up to
32.
So this is setting batch size equals 32.
So we look at 32 samples per batch.
We mix all the samples up, and we go batch, batch, batch, batch, batch, and we'll
have,
however many batches we have, we'll have number of samples divided by the batch
size.
So 60,000 divided by 32, what's that, 1800 or something like that?
So this is what we're going to be working towards.
I did want to write some code in this video, but I think to save it getting too
long, we're
going to write this code in the next video.
If you would like to give this a go on your own, here's most of the code we have to
do.
So there's the train data loader, do the same for the test data loader.
And I'll see you in the next video, and we're going to batchify our fashion MNIST
data set.
Welcome back.
In the last video, we had a brief overview of the concept of mini batches.
And so rather than our computer looking at 60,000 images in one hit, we break
things down.
We turn it into batches of 32.
Again, the batch size will vary depending on what problem you're working on.
But 32 is quite a good value to start with and try out.
And we do this for two main reasons, if we jump back to the code, why would we do
this?
It is more computationally efficient.
So if we have a GPU with, say, 10 gigabytes of memory, it might not be able to
store all
60,000 images in one hit.
In our data set, because it's quite small, it may be hour or two, but it's better
practice
for later on to turn things into mini batches.
And it also gives our neural network more chances to update its gradients per
epoch,
which will make a lot more sense once we write our training loop.
But for now, we've spoken enough about the theory.
Let's write some code to do so.
So I'm going to import data loader from torch dot utils dot data, import data
loader.
And this principle, by the way, preparing a data loader goes the same for not only

```
images,
but for text, for audio, whatever sort of data you're working with, mini batches
will
follow you along or batches of data will follow you along throughout a lot of
different deep
learning problems.
So set up the batch size hyper parameter.
Remember, a hyper parameter is a value that you can set yourself.
So batch size equals 32.
And it's practice.
You might see it typed as capitals.
You won't always see it, but you'll see it quite often a hyper parameter typed as
capitals.
And then we're going to turn data sets into iterables.
So batches.
So we're going to create a train data loader here of our fashion MNIST data set.
We're going to use data loader.
We're going to see what the doc string is.
Or actually, let's look at the documentation torch data loader.
This is some extra curriculum for you too, by the way, is to read this data page
torch
utils not data because no matter what problem you're going with with deep learning
or pytorch,
you're going to be working with data.
So spend 10 minutes just reading through here.
I think I might have already assigned this, but this is just so important that it's
worth
going through again.
Read through all of this.
```

Section 153: Main Topics

Key Topics:

- So this is going to tell us how many batches there are, batches of, which of course is batch size
- Now of course, the number of batches we have will change if we change the batch size

▶ 📄 Click to view detailed content

```
Even if you don't understand all of it, what's going on, it's just it helps you
know where
to look for certain things.
So what does it take?
Data loader takes a data set.
We need to set the batch size to something is the default of one.
That means that it would create a batch of one image at a time in our case.
```

```
Do we want to shuffle it?
Do we want to use a specific sampler?
There's a few more things going on.
Number of workers.
Number of workers stands for how many cores on our machine do we want to use to
load data?
Generally the higher the better for this one, but we're going to keep most of these
as
the default because most of them are set to pretty good values to begin with.
I'll let you read more into the other parameters here.
We're going to focus on the first three data set batch size and shuffle true or
false.
Let's see what we can do.
So data set equals our train data, which is 60,000 fashion MNIST.
And then we have a batch size, which we're going to set to our batch size hyper
parameter.
So we're going to have a batch size of 32.
And then finally, do we want to shuffle the training data?
Yes, we do.
And then we're going to do the same thing for the test data loader, except we're
not
going to shuffle the test data.
Now, you can shuffle the test data if you want, but in my practice, it's actually
easier
to evaluate different models when the test data isn't shuffled.
So you shuffle the training data to remove order.
And so your model doesn't learn order.
But for evaluation purposes, it's generally good to have your test data in the same
order
because our model will never actually see the test data set during training.
We're just using it for evaluation.
So the order doesn't really matter to the test data loader.
It's just easier if we don't shuffle it, because then if we evaluate it multiple
times, it's
not been shuffled every single time.
So let's run that.
And then we're going to check it out, our train data loader and our test data
loader.
Beautiful.
Instances of torch utils data, data loader, data loader.
And now let's check out what we've created, hey, I always like to print different
attributes
of whatever we make, check out what we've created.
This is all part of becoming one with the data.
So print F, I'm going to go data loaders, and then pass in, this is just going to
output
basically the exact same as what we've got above.
This data loader.
And we can also see what attributes we can get from each of these by going train
data
loader.
I don't need caps lock there, train data loader, full stop.
And then we can go tab.
We've got a whole bunch of different attributes.
We've got a batch size.
We've got our data set.
Do we want to drop the last as in if our batch size overlapped with our 60,000
```

samples?
Do we want to get rid of the last batch?
Say for example, the last batch only had 10 samples.
Do we want to just drop that?
Do we want to pin the memory that's going to help later on if we wanted to load our data faster?
A whole bunch of different stuff here.
If you'd like to research more, you can find all the stuff about what's going on here in
the documentation.
But let's just keep pushing forward.
What else do we want to know?
So let's find the length of the train data loader.
We will go length train data loader.
So this is going to tell us how many batches there are, batches of, which of course is batch
size.
And we want print length of test data loader.
We want length test data loader batches of batch size dot dot dot.
So let's find out some information.
What do we have?
Oh, there we go.
So just we're seeing what we saw before with this one.
But this is more interesting here.
Length of train data loader.
Yeah, we have about 1,875 batches of 32.
So if we do 60,000 training samples divided by 32, yeah, it comes out to 1,875.
And if we did the same with 10,000 for testing samples of 32, it comes out at 313.
This gets rounded up.
So this is what I meant, that the last batch will have maybe not 32 because 32 doesn't
divide evenly into 10,000, but that's okay.
And so this means that our model is going to look at 1,875 individual batches of 32 images, rather than just one big batch of 60,000 images.
Now of course, the number of batches we have will change if we change the batch size.
So we have 469 batches of 128.
And if we reduce this down to one, what do we get?
We have a batch per sample.
So 60,000 batches of 1, 10,000 batches of 1, we're going to stick with 32.
But now let's visualize.
So we've got them in train data loader.
How would we visualize a batch or a single image from a batch?
So let's show a sample.
I'll show you how you can interact with a data loader.
We're going to use randomness as well.
So we'll set a manual seed and then we'll get a random index, random idx equals torch
rand int.
We're going to go from zero to length of train features batch.
Oh, where did I get that from?
Excuse me.
Getting ahead of myself here.
I want to check out what's inside the training data loader.
We'll check out what's inside the training data loader because the test data load is

```
going to be similar.
So we want the train features batch.
```

Section 154: Main Topics

Key Topics:

- So I say features as in the images themselves and the train labels batch is going to be the labels of our data set or the targets in pytorch terminology
- When starting to build a series of machine learning modeling experiments, it's best practice to start with a baseline model

▶ 📄 Click to view detailed content

```
So I say features as in the images themselves and the train labels batch is going
to be
the labels of our data set or the targets in pytorch terminology.
So next idar data loader.
So because our data loader has 1875 batches of 32, we're going to turn it into an
iterable
with ita and we're going to get the next batch with next and then we can go here
train features
batch.shape and we'll get train labels batch.shape.
What do you think this is going to give us?
Well, there we go.
Look at that.
So we have a tensor.
Each batch we have 32 samples.
So this is batch size and this is color channels and this is height and this is
width.
And then we have 32 labels associated with the 32 samples.
Now where have we seen this before, if we go back through our keynote input and
output
shapes.
So we have shape equals 32, 28, 28, 1.
So this is color channels last, but ours is currently in color channels first.
Now again, I sound like a broken record here, but these will vary depending on the
problem
you're working with.
If we had larger images, what would change or the height and width dimensions would
change.
If we had color images, the color dimension would change, but the premise is still
the
same.
We're turning our data into batches so that we can pass that to a model.
Let's come back.
Let's keep going with our visualization.
So we want to visualize one of the random samples from a batch and then we're going
```

to
go image label equals train features batch and we're going to get the random IDX from
that and we'll get the train labels batch and we'll get the random IDX from that.
So we're matching up on the, we've got one batch here, train features batch, train labels
batch and we're just getting the image and the label at a random index within that batch.
So excuse me, I need to set this equal there.
And then we're going to go PLT dot in show, what are we going to show?
We're going to show the image but we're going to have to squeeze it to remove that singular
dimension and then we'll set the C map equal to gray and then we'll go PLT dot title, we'll
set the title which is going to be the class names indexed by the label integer and then
we can turn off the accesses.
You can use off here or you can use false, depends on what you'd like to use.
Let's print out the image size because you can never know enough about your data and
then print, let's also get the label, label and label shape or label size.
Our label will be just a single integer so it might not have a shape but that's okay.
Let's have a look.
Oh, bag.
See, look, that's quite hard to understand.
I wouldn't be able to detect that that's a bag.
Can you tell me that you could write a program to understand that?
That just looks like a warped rectangle to me.
But if we had to look at another one, we'll get another random, oh, we've got a random
seed so it's going to produce the same image each time.
So we have a shirt, okay, a shirt.
So we see the image size there, 128, 28.
Now, recall that the image size is, it's a single image so it doesn't have a batch dimension.
So this is just color channels height width.
We'll go again, label four, which is a coat and we could keep doing this to become more
and more familiar with our data.
But these are all from this particular batch that we created here, coat and we'll do one
more, another coat.
We'll do one more just to make sure it's not a coat.
There we go.
We've got a bag.
Beautiful.
So we've now turned our data into data loaders.
So we could use these to pass them into a model, but we don't have a model.
So I think it's time in the next video, we start to build model zero.
We start to build a baseline.
I'll see you in the next video.
Welcome back.
So in the last video, we got our data sets or our data set into data loaders.
So now we have 1,875 batches of 32 images off of the training data set rather than 60,000
in a one big data set.

```
And we have 13 or 313 batches of 32 for the test data set.
Then we learned how to visualize it from a batch.
And we saw that we have still the same image size, one color channel, 28, 28.
All we've done is we've turned them into batches so that we can pass them to our
model.
And speaking of model, let's have a look at our workflow.
Where are we up to?
Well, we've got our data ready.
We've turned it into tensors through a combination of torch vision transforms,
torch utils data
dot data set.
We didn't have to use that one because torch vision dot data sets did it for us
with the
fashion MNIST data set, but we did use that one.
We did torch utils dot data, the data loader to turn our data sets into data
loaders.
Now we're up to building or picking a pre-trained model to suit your problem.
So let's start simply.
Let's build a baseline model.
And this is very exciting because we're going to build our first model, our first
computer
vision model, albeit a baseline, but that's an important step.
So I'm just going to write down here.
When starting to build a series of machine learning modeling experiments, it's best
practice
to start with a baseline model.
I'm going to turn this into markdown.
A baseline model.
```

Section 155: Main Topics

Key Topics:

- We go nn flatten, flatten in pytorch, what does it do
- And of course, all nn
- We're going to have an input shape, which we'll use a type hint, which will take an integer because remember, input shape is very important for machine learning models

▶ 📄 Click to view detailed content

```
So a baseline model is a simple model.
You will try and improve upon with subsequent models, models slash experiments.
So you start simply, in other words, start simply and add complexity when necessary
because
neural networks are pretty powerful, right?
And so they have a tendency to almost do too well on our data set.
That's a concept known as overfitting, which we'll cover a little bit more later.
```

But we built a simple model to begin with, a baseline.
And then our whole goal will be to run experiments, according to the workflow, improve through
experimentation.
Again, this is just a guide.
It's not set in stone, but this is the general pattern of how things go.
Get data ready, build a model, fit the model, evaluate, improve the model.
So the first model that we build is generally a baseline.
And then later on, we want to improve through experimentation.
So let's start building a baseline.
But I'm going to introduce to you a new layer that we haven't seen before.
That is creating a flatten layer.
Now what is a flatten layer?
Well, this is best seen when we code it out.
So let's create a flatten model, which is just going to be nn.flatten.
And where could we find the documentation for this?
We go nn flatten, flatten in pytorch, what does it do?
Flattens a continuous range of dims into a tensor, for use with sequential.
So there's an example there, but I'd rather, if and doubt, code it out.
So we'll create the flatten layer.
And of course, all nn.flatten or nn.modules could be used as a model on their own.
So we're going to get a single sample.
So x equals train features batch.
Let's get the first one, zero.
What does this look like?
So it's a tensor, x, maybe we get the shape of it as well, x shape.
What do we get?
There we go.
So that's the shape of x.
Keep that in mind when we pass it through the flatten layer.
Do you have an inkling of what flatten might do?
So our shape to begin with is what, 128, 28.
Now let's flatten the sample.
So output equals, we're going to pass it to the flatten model, x.
So this is going to perform the forward pass internally on the flatten layer.
So perform forward pass.
Now let's print out what happened.
Print, shape before flattening equals x dot shape.
And we're going to print shape after flattening equals output dot shape.
So we're just taking the output of the flatten model and printing its shape here.
Oh, do you notice what happened?
Well we've gone from 128, 28 to 1784.
Wow what does the output look like?
Output.
Oh, the values are now in all one big vector and if we squeeze that we can remove the extra
dimension.
So we've got one big vector of values.
Now where did this number come from?
Well, if we take this and this is what shape is it?
We've got color channels.
We've got height.
We've got width and now we've flattened it to be color channels, height, width.
So we've got one big feature vector because 28 by 28 equals what?
We've got one value per pixel, 784.
One value per pixel in our output vector.
Now where did we see this before?

```
If we go back to our keynote, if we have a look at Tesla's takes eight cameras and
then
it turns it into a three dimensional vector space, vector space.
So that's what we're trying to do here.
We're trying to encode whatever data we're working with in Tesla's case.
They have eight cameras.
Now theirs has more dimensions than ours because they have the time aspect because
they're
dealing with video and they have multiple different camera angles.
We're just dealing with a single image here.
But regardless, the concept is the same.
We're trying to condense information down into a single vector space.
And so if we come back to here, why might we do this?
Well, it's because we're going to build a baseline model and we're going to use a
linear
layer as the baseline model.
And the linear layer can't handle multi dimensional data like this.
We want it to have a single vector as input.
Now this will make a lot more sense after we've coded up our model.
Let's do that from torch import and then we're going to go class, fashion, amnest,
model
V zero.
We're going to inherit from an end dot module.
And inside here, we're going to have an init function in the constructor.
We're going to pass in self.
We're going to have an input shape, which we'll use a type hint, which will take an
integer
because remember, input shape is very important for machine learning models.
We're going to define a number of hidden units, which will also be an integer, and
then we're
going to define our output shape, which will be what do you think our output shape
will
be?
How many classes are we dealing with?
We're dealing with 10 different classes.
So our output shape will be, I'll save that for later on.
I'll let you guess for now, or you might already know, we're going to initialize
it.
And then we're going to create our layer stack.
self.layer stack equals nn.sequential, recall that sequential, whatever you put
inside sequential,
if data goes through sequential, it's going to go through it layer by layer.
So let's create our first layer, which is going to be nn.flatten.
So that means anything that comes into this first layer, what's going to happen to
it?
It's going to flatten its external dimensions here.
So it's going to flatten these into something like this.
So we're going to flatten it first, flatten our data.
Then we're going to pass in our linear layer.
```

Section 156: Main Topics

Key Topics:

- One of the biggest errors that you're going to face in machine learning is different tensor shape mismatches

▶ 📄 Click to view detailed content

```
And we're going to have how many n features this is going to be input shape,
because we're
going to define our input shape here.
And then we're going to go out features, equals hidden units.
And then we're going to create another linear layer here.
And we're going to set up n features, equals hidden units.
Why are we doing this?
And then out features equals output shape.
Why are we putting the same out features here as the n features here?
Well, because subsequent layers, the input of this layer here, its input shape has
to
line up with the output shape of this layer here.
Hence why we use out features as hidden units for the output of this nn.linear
layer.
And then we use n features as hidden units for the input value of this hidden layer
here.
So let's keep going.
Let's go def.
We'll create the forward pass here, because if we subclass nn.module, we have to
override
the forward method.
The forward method is going to define what?
It's going to define the forward computation of our model.
So we're just going to return self.layer stack of x.
So our model is going to take some input, x, which could be here, x.
In our case, it's going to be a batch at a time, and then it's going to pass each
sample
through the flatten layer.
It's going to pass the output of the flatten layer to this first linear layer, and
it's
going to pass the output of this linear layer to this linear layer.
So that's it.
Our model is just two linear layers with a flatten layer.
The flatten layer has no learnable parameters.
Only these two do.
And we have no nonlinearities.
So do you think this will work?
Does our data set need nonlinearities?
Well, we can find out once we fit our model to the data, but let's set up an
instance
of our model.
So torch dot manual seed.
Let's go set up model with input parameters.
So we have model zero equals fashion MNIST model, which is just the same class that
we
wrote above.
And here's where we're going to define the input shape equals 784.
```

Where will I get that from?
Well, that's here.
That's 28 by 28.
So the output of flatten needs to be the input shape here.
So we could put 28 by 28 there, or we're just going to put 784 and then write a comment
here.
This is 28 by 28.
Now if we go, I wonder if nn.linear will tell us, nn.linear will tell us what it expects
as in features.
Size of each input sample, shape, where star means any number of dimensions, including
none in features, linear weight, well, let's figure it out.
Let's see what happens if in doubt coded out, hey, we'll see what we can do.
In units equals, let's go with 10 to begin with.
How many units in the hidden layer?
And then the output shape is going to be what?
Output shape is length of class names, which will be 1 for every class.
Beautiful.
And now let's go model zero.
We're going to keep it on the CPU to begin with.
We could write device-agnostic code, but to begin, we're going to send it to the CPU.
I might just put that up here, actually, to CPU.
And then let's have a look at model zero.
Wonderful.
So we can try to do a dummy forward pass and see what happens.
So let's create dummy x equals torch, rand, we'll create it as the same size of image.
Just a singular image.
So this is going to be a batch of one, color channel one, height 28, height 28.
And we're going to go model zero and pass through dummy x.
So this is going to send dummy x through the forward method.
Let's see what happens.
Okay, wonderful.
So we get an output of 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 logits.
Beautiful.
That's exactly what we want.
We have one logit value per class that we have.
Now what would happen if we got rid of flatten?
Then we ran this, ran this, ran this.
What do we get?
Oh, mat one and mat two shapes cannot be multiplied.
So we have 28 by 28 and 7.
Okay, what happens if we change our input shape to 28?
We're getting shape mismatches here.
What happens here?
Oh, okay, we get an interesting output, but this is still not the right shape, is it?
So that's where the flatten layer comes in.
What is the shape of this?
Oh, we get 1, 1, 28, 10.
Oh, so that's why we put in flatten so that it combines it into a vector.
So we get rid of this, see if we just leave it in this shape?
We get 28 different samples of 10, which is not what we want.
We want to compress our image into a singular vector and pass it in.

```
So let's reinstanceuate the flatten layer and let's make sure we've got the right
input
shape here, 28 by 28, and let's pass it through, torch size 110.
That's exactly what we want, 1 logit per class.
So this could be a bit fiddly when you first start, but it's also a lot of fun once
you
get it to work.
And so just keep that in mind, I showed you what it looks like when you have an
error.
One of the biggest errors that you're going to face in machine learning is
different tensor
shape mismatches.
So just keep in mind the data that you're working with and then have a look at the
documentation
for what input shape certain layers expect.
So with that being said, I think it's now time that we start moving towards
training
our model.
I'll see you in the next video.
```

Section 157: Main Topics

Key Topics:

- So these are of course initialized with random values, but the whole premise of deep learning and machine learning is to pass data through our model and use our optimizer to update these random values to better represent the features in our data
- The whole premise of it is to, or the whole fun, the whole magic behind machine learning is that it figures out what features to learn
- Well, if we go into learnpytorch

▶ 📄 Click to view detailed content

```
Welcome back.
In the last video, we created model zero, which is going to be our baseline model
for
our computer vision problem of detecting different types of clothing in 28 by 28
gray scale
images.
And we also learned the concept of making sure our or we rehashed on the concept of
making sure our input and output shapes line up with where they need to be.
We also did a dummy forward pass with some dummy data.
This is a great way to troubleshoot to see if your model shapes are correct.
If they come out correctly and if the inputs are lining up with where they need to
be.
And just to rehash on what our model is going to be or what's inside our model, if
```

we check
model zero state dict, what we see here is that our first layer has a weight
tensor.
It also has a bias and our next layer has a weight tensor and it also has a bias.
So these are of course initialized with random values, but the whole premise of
deep learning
and machine learning is to pass data through our model and use our optimizer to
update
these random values to better represent the features in our data.
And I keep saying features, but I just want to rehash on that before we move on to
the
next thing.
Featuring data could be almost anything.
So for example, the feature of this bag could be that it's got a rounded handle at
the
top.
It has a edge over here.
It has an edge over there.
Now, we aren't going to tell our model what features to learn about the data.
The whole premise of it is to, or the whole fun, the whole magic behind machine
learning
is that it figures out what features to learn.
And so that is what the weights and bias matrices or tensors will represent is
different features
in our images.
And there could be many because we have 60,000 images of 10 classes.
So let's keep pushing forward.
It's now time to set up a loss function and an optimizer.
Speaking of optimizers, so 3.1 set up loss optimizer and evaluation metrics.
Now recall in notebook two, I'm going to turn this into markdown.
We created, oh, I don't need an emoji there.
So this is, by the way, we're just moving through this workflow.
We've got our data ready into tensors.
We've built a baseline model.
It's now time to pick a loss function and an optimizer.
So we go back to Google Chrome.
That's right here.
Loss function.
What's our loss function going to be?
Since we're working with multi-class data, our loss function will be NN dot cross
entropy
loss.
And our optimizer, we've got a few options here with the optimizer, but we've had
practice
in the past with SGD, which stands for stochastic gradient descent and the atom
optimizer.
So our optimizer, let's just stick with SGD, which is kind of the entry level
optimizer
torch opt in SGD for stochastic gradient descent.
And finally, our evaluation metric, since we're working on a classification
problem, let's
use accuracy as our evaluation metric.
So recall that accuracy is a classification evaluation metric.
Now, where can we find this?
Well, if we go into learnpytorch.io, this is the beauty of having online reference
material.
In here, neural network classification with PyTorch, in this notebook, section 02,

```
we
created, do we have different classification methods?
Yes, we did.
So we've got a whole bunch of different options here for classification evaluation
metrics.
We've got accuracy, precision, recall, F1 score, a confusion matrix.
Now we have some code that we could use.
If we wanted to use torch metrics for accuracy, we could.
And torch metrics is a beautiful library that has a lot of evaluation.
Oh, it doesn't exist.
What happened to torch metrics?
Maybe I need to fix that.
Link.
Torch metrics has a whole bunch of different PyTorch metrics.
So very useful library.
But we also coded a function in here, which is accuracy FN.
So we could copy this, straight into our notebook here.
Or I've also, if we go to the PyTorch deep learning GitHub, I'll just bring it over
here.
I've also put it in helper functions.py.
And this is a script of common functions that we've used throughout the course,
including
if we find accuracy function here.
Calculate accuracy.
Now, how would we get this helper functions file, this Python file, into our
notebook?
One way is to just copy the code itself, straight here.
But let's import it as a Python script.
So import request, and we're going to go from pathlib import path.
So we want to download, and this is actually what you're going to see, very common
practice
in larger Python projects, especially deep learning and machine learning projects,
is
different functionality split up in different Python files.
And that way, you don't have to keep rewriting the same code over and over again.
Like you know how we've written a training and testing loop a fair few times?
Well, if we've written it once and it works, we might want to save that to a.py
file so
we can import it later on.
So let's now write some code to import this helper functions.py file into our
notebook
here.
So download helper functions from learn pytorch repo.
So we're going to check if our helper functions.py, if this already exists, we
don't want
to download it.
So we'll print helper functions.py already exists, skipping download, skipping
download
.dot.
And we're going to go else here.
If it doesn't exist, so we're going to download it, downloading helper
functions.py.
And we're going to create a request here with the request library equals
request.get.
Now here's where we have to pass in the URL of this file.
It's not this URL here.
```

Section 158: Main Topics

Key Topics:

- Of course, we could have used torch metrics as well
- Of course, you can customize helper functions
- And then I'm going to set the learning rate here

▶ 📄 Click to view detailed content

```
When dealing with GitHub, to get the actual URL to the files, many files, you have
to
click the raw button.
So I'll just go back and show you, click raw here.
And we're going to copy this raw URL.
See how it's just text here?
This is what we want to download into our co-lab notebook.
And we're going to write it in there, request equals request.get.
And we're going to go with open, and here's where we're going to save our helper
functions
.py.
We're going to write binary as file, F is for file.
We're going to go F.write, request.content.
So what this is saying is Python is going to create a file called helper
functions.py
and give it write binary permissions as F, F is for file, short for file.
And then we're going to say F.write, request, get that information from helper
functions
.py here, and write your content to this file here.
So let's give that a shot.
Beautiful.
So downloading helper functions.py, let's have a look in here.
Do we have helper functions.py?
Yes, we do.
Wonderful.
We can import our accuracy function.
Where is it?
There we go.
Import accuracy function.
So this is very common practice when writing lots of Python code is to put helper
functions
into.py scripts.
So let's import the accuracy metric.
Accuracy metric from helper functions.
Of course, we could have used torch metrics as well.
That's another perfectly valid option, but I just thought I'd show you what it's
like
to import your own helper function script.
Of course, you can customize helper functions.py to have whatever you want in
there.
```

```
So see this?
We've got from helper functions, import accuracy function.
What's this saying?
Could not be resolved.
Is this going to work?
It did.
And where you can go accuracy function, do we get a doc string?
Hmm.
Seems like colab isn't picking things up, but that's all right.
It looks like it still worked.
We'll find out later on if it actually works when we train our model.
So set up loss function and optimizer.
So I'm going to set up the loss function equals nn dot cross entropy loss.
And I'm going to set up the optimizer here as we discussed before as torch dot opt-
in
dot SGD for stochastic gradient descent.
The parameters I want to optimize are the parameters from model zero, our baseline
model,
which we had a look at before, which are all these random numbers.
We'd like our optimizer to tweak them in some way, shape, or form to better
represent our
data.
And then I'm going to set the learning rate here.
How much should they be tweaked each epoch?
I'm going to set it to 0.1.
Nice and high because our data set is quite simple.
It's 28 by 28 images.
There are 60,000 of them.
But again, if this doesn't work, we can always adjust this and experiment,
experiment,
experiment.
So let's run that.
We've got a loss function.
Is this going to give me a doc string?
There we go.
So calculates accuracy between truth and predictions.
Now, where does this doc string come from?
Well, let's have a look, hope of functions.
That's what we wrote before.
Good on us for writing good doc strings, accuracy function.
Well, we're going to test all these out in the next video when we write a training
loop.
So, oh, actually, I think we might do one more function before we write a training
loop.
How about we create a function to time our experiments?
Yeah, let's give that a go in the next video.
I'll see you there.
Welcome back.
In the last video, we downloaded our helper functions.py script and imported our
accuracy
function that we made in notebook two.
But we could really beef this up, our helper functions.py file.
We could put a lot of different helper functions in there and import them so we
didn't have
to rewrite them.
That's just something to keep in mind for later on.
But now, let's create a function to time our experiments.
```

```
So creating a function to time our experiments.
So one of the things about machine learning is that it's very experimental.
You've probably gathered that so far.
So let's write here.
So machine learning is very experimental.
Two of the main things you'll often want to track are, one, your model's
performance
such as its loss and accuracy values, et cetera.
And two, how fast it runs.
So usually you want a higher performance and a fast model, that's the ideal
scenario.
However, you could imagine that if you increase your model's performance, you might
have
a bigger neural network.
It might have more layers.
It might have more hidden units.
It might degrade how fast it runs because you're simply making more calculations.
So there's often a trade-off between these two.
And how fast it runs will really be important if you're running a model, say, on
the internet
or say on a dedicated GPU or say on a mobile device.
So these are two things to really keep in mind.
So because we're tracking our model's performance with our loss value and our
accuracy function,
let's now write some code to check how fast it runs.
And I did on purpose above, I kept our model on the CPU.
So we're also going to compare later on how fast our model runs on the CPU versus
how
fast it runs on the GPU.
So that's something that's coming up.
Let's write a function here.
We're going to use the time module from Python.
So from time it, import the default timer, as I'm going to call it timer.
So if we go Python default timer, do we get the documentation for, here we go, time
it.
So do we have default timer, wonderful.
So the default timer, which is always time.perf counter, you can read more about
Python timing
```

Section 159: Main Topics

Key Topics:

- And then of course, we could add more there for the arguments, but that's a quick one liner
- Since I use Google Colab Pro, completely unnecessary for the course, but I just found it worth it for how much I use Google Colab, I get longer idle timeouts, so that means that my Colab notebook will stay persistent for a longer time

- But of course overnight it's going to disconnect, so I click reconnect, and then if I want to get back to wherever we were, because we downloaded some data from torchvision

▶ 📄 Click to view detailed content

```
functions in here.
But this is essentially just going to say, hey, this is the exact time that our
code
started.
And then we're going to create another stop for when our code stopped.
And then we're going to compare the start and stop times.
And that's going to basically be how long our model took to train.
So we're going to go def print train time.
This is just going to be a display function.
So start, we're going to get the float type hint, by the way, start an end time.
So the essence of this function will be to compare start and end time.
And we're going to set the torch or the device here, we'll pass this in as torch
dot device.
And we're going to set that default to none, because we want to compare how fast
our model
runs on different devices.
So I'm just going to write a little doc string here, prints, difference between
start and
end time.
And then of course, we could add more there for the arguments, but that's a quick
one liner.
Tell us what our function does.
So total time equals end minus start.
And then print, we're going to write here train time on, whichever device we're
using
might be CPU, might be GPU.
Total time equals, we'll go to three and we'll say seconds, three decimal places
that is
and return total time.
Beautiful.
So for example, we could do start time equals timer, and then end time equals
timer.
And then we can put in here some code between those two.
And then if we go print train, oh, maybe we need a timer like this, we'll find out
if
and out code it out, you know, we'll see if it works.
Start time and end equals end time and device equals.
We're running on the CPU right now, CPU, let's see if this works, wonderful.
So it's a very small number here.
So train time on CPU, very small number, because the start time is basically on
this
exact line, comment basically it takes no time to run, then end time is on here, we
get
3.304 times 10 to the power of negative five.
So quite a small number, but if we put some modeling code in here, it's going to
measure
the start time of this cell, it's going to model our code in there, then we have
the
```

end time, and then we find out how long our model took the train.
So with that being said, I think we've got all of the pieces of the puzzle for creating
some training and testing functions.
So we've got a loss function, we've got an optimizer, we've got a valuation metric, we've
got a timing function, we've got a model, we've got some data.
How about we train our first baseline computer vision model in the next video?
I'll see you there.
Good morning.
Well might not be morning wherever you are in the world.
It's nice and early here, I'm up recording some videos, because we have a lot of momentum
going with this, but look at this, I took a little break last night, I have a runtime
disconnected, but this is just what's going to happen if you're using Google Colab.
Since I use Google Colab Pro, completely unnecessary for the course, but I just found it worth
it for how much I use Google Colab, I get longer idle timeouts, so that means that my
Colab notebook will stay persistent for a longer time.
But of course overnight it's going to disconnect, so I click reconnect, and then if I want to
get back to wherever we were, because we downloaded some data from torchvision.datasets, I have
to rerun all of these cells.
So a nice shortcut, we might have seen this before, is to just come down to where we were,
and if all the code above works, oh there we go, I wrote myself some notes of where we're
up to.
Let's go run before, so this is just going to run all the cells above, and we're up
to here, 3.3 creating a training loop, and training a model on batches of data.
So that's going to be a little bit interesting, and I wrote myself another reminder here, this
is a little bit of behind the scenes, the optimise will update a model's parameters
once per batch rather than once per epoch.
So let's hold myself to that note, and make sure I let you know.
So we're going to make another title here.
Let's go creating a training loop, and training a model on batches of data.
So something a little bit different to what we may have seen before if we haven't created
batches of data using data loader, and recall that just up above here, we've got something
like 1800 there, there we go.
So we've split our data into batches, rather than our model looking at 60,000 images of
fashion MNIST data at one time, it's going to look at 1875 batches of 32, so 32 images
at the time, of the training data set, and 313 batches of 32 of the test data set.
So let's go to training loop and train our first model.
So I'm going to write out a few steps actually, because we have to do a little bit differently
to what we've done before.
So one, we want to loop through epochs, so a number of epochs.
Loop through training batches, and by the way, you might be able to hear some birds singing,

```
the sun is about to rise, I hope you enjoy them as much as I do.
So we're going to perform training steps, and we're going to calculate calculate
the
train loss per batch.
So this is going to be one of the differences between our previous training loops.
```

Section 160: Main Topics

Key Topics:

- Wonderful, four, we're going to, of course, print out what's happening
- You may have seen the unofficial PyTorch optimization loop theme song
- And we're going to time it all for fun, of course, because that's what our timing function is for

▶ 📄 Click to view detailed content

```
And this is going to, after number two, we're going to loop through the testing
batches.
So we'll train and evaluate our model at the same step, or same loop.
And we're going to perform testing steps.
And then we're going to calculate the test loss per batch as well, per batch.
Wonderful, four, we're going to, of course, print out what's happening.
You may have seen the unofficial PyTorch optimization loop theme song.
And we're going to time it all for fun, of course, because that's what our timing
function
is for.
So let's get started.
There's a fair few steps here, but nothing that we can't handle.
And remember the motto, if and out, code it out.
Well, there's another one, if and out, run the code, but we haven't written any
code to
run just yet.
So we're going to import TQDM for a progress bar.
If you haven't seen TQDM before, it's a very good Python progress bar that you can
add
with a few lines of code.
So this is just the GitHub.
It's open source software, one of my favorite pieces of software, and it's going to
give
us a progress bar to let us know how many epochs our training loop has gone
through.
It doesn't have much overhead, but if you want to learn more about it, please refer
to the TQDM GitHub.
However, the beautiful thing is that Google CoLab has TQDM built in because it's so
good
and so popular.
So we're going to import from TQDM.auto.
```

So there's a few different types of TQDM progress bars.auto is just going to recognize what
compute environment we're using.
And it's going to give us the best type of progress bar for what we're doing.
So for example, Google CoLab is running a Jupyter Notebook behind the scenes.
So the progress bar for Jupyter Notebooks is a little bit different to Python scripts.
So now let's set the seed and start the timer.
We want to write all of our training loop in this single cell here.
And then once it starts, once we run this cell, we want the timer to start so that we
can time how long the entire cell takes to run.
So we'll go train time start on CPU equals, we set up our timer before, beautiful.
Now we're going to set the number of epochs.
Now we're going to keep this small for faster training time so we can run more experiments.
So we'll keep this small for faster training time.
That's another little tidbit.
Do you notice how quickly all of the cells ran above?
Well, that's because we're using a relatively small data set.
In the beginning, when you're running experiments, you want them to run quite quickly so that
you can run them more often.
So you can learn more about your data so that you can try different things, try different
models.
So this is why we're using number of epochs equals three.
We start with three so that our experiment runs in 30 seconds or a minute or so.
That way, if something doesn't work, we haven't wasted so much time waiting for a model to
train.
Later on, we could train it for 100 epochs if we wanted to.
So we're going to create a training and test loop.
So for epoch in TQDM range epochs, let's get this going.
So for TQDM to work, we just wrap our iterator with TQDM and you'll see later on how this
tracks the progress.
So I'm going to put out a little print statement here.
We'll go epoch.
This is just going to say what epoch we're on.
We'll go here.
That's something that I like to do quite often is put little print statements here and there
so that we know what's going on.
So let's set up the training.
We're going to have to instantiate the train loss.
We're going to set that to zero to begin with.
And we're going to cumulatively add some values to the train loss here and then we'll
see later on how this accumulates and we can calculate the training loss per batch.
Let's what we're doing up here, calculate the train loss per batch.
And then finally, at the end of the loop, we will divide our training loss by the number
of batches so we can get the average training loss per batch and that will give us the training
loss per epoch.
Now that's a lot of talking.

```
If that doesn't make sense, remember.
But if and out, code it out.
So add a loop to loop through the training batches.
So because our data is batchified now and I've got a crow or maybe a cooker bar
sitting
on the roof across from my apartment, it's singing its song this morning, lovely.
So we're going to loop through our training batch data.
So I've got four batch, comma x, y, because remember our training batches come in
the
form of X.
So that's our data or our images and why, which is label.
You could call this image label or target as part of which would, but it's
convention
to often call your features X and your labels Y.
We've seen this before in we're going to enumerate the train data loader as well.
We do this so we can keep track of the number of batches we've been through.
So that will give us batch there.
I'm going to set model zero to training mode because even though that's the
default, we
just want to make sure that it's in training mode.
Now we're going to do the forward pass.
If you remember, what are the steps in apply to our optimization loop?
We do the forward pass.
We calculate the loss of the minus zero grad, last backwards, up to minus a step,
step,
step.
So let's do that.
```

Section 161: Main Topics

Key Topics:

- And of course, later on, you might be thinking, then you'll, how come we haven't functionalized this training loop already
- Well, that's because we like to practice writing PyTorch code, right
- So print looked at, and of course you can adjust this to whatever you would like

▶ 📄 Click to view detailed content

```
Hey, model zero, we'll put the features through there and then we're going to
calculate the
loss.
We've been through these steps before.
So we're not going to spend too much time on the exact steps here, but we're just
going
to practice writing them out.
And of course, later on, you might be thinking, then you'll, how come we haven't
functionalized
```

```
this training loop already?
We've seemed to write the same generic code over and over again.
Well, that's because we like to practice writing PyTorch code, right?
We're going to functionalize them later on.
Don't you worry about that.
So here's another little step that we haven't done before is we have the training
loss.
And so because we've set that to zero to begin with, we're going to accumulate the
training
loss values every batch.
So we're going to just add it up here.
And then later on, we're going to divide it by the total number of batches to get
the
average loss per batch.
So you see how this loss calculation is within the batch loop here?
So this means that one batch of data is going to go through the model.
And then we're going to calculate the loss on one batch of data.
And this loop is going to continue until it's been through all of the batches in
the train
data loader.
So 1875 steps or whatever there was.
So accumulate train loss.
And then we're going to optimize a zero grad, optimizer dot zero grad.
And then number four is what?
Loss backward.
Loss backward.
We'll do the back propagation step.
And then finally, we've got number five, which is optimizer step.
So this is where I left my little note above to remind me and to also let you know,
highlight
that the optimizer will update a model's parameters once per batch rather than once
per epoch.
So you see how we've got a for loop inside our epoch loop here.
So the batch loop.
So this is what I meant that the optimizer, this is one of the advantages of using
mini
batches is not only is it more memory efficient because we're not loading 60,000
images into
memory at a time.
We are updating our model's parameters once per batch rather than waiting for it to
see
the whole data set with every batch.
Our model is hopefully getting slightly better.
So that is because the optimizer dot step call is within the batch loop rather than
the
epoch loop.
So let's now print out what's happening.
Print out what's happening.
So if batch, let's do it every 400 or so batches because we have a lot of batches.
We don't want to print out too often, otherwise we'll just fill our screen with
numbers.
That might not be a bad thing, but 400 seems a good number.
That'll be about five printouts if we have 2000 batches.
So print looked at, and of course you can adjust this to whatever you would like.
That's the flexibility of PyTorch, flexibility of Python as well.
So looked at how many samples have we looked at?
So we're going to take the batch number, multiply it by X, the length of X is going
```

to be 32 because that is our batch size.
Then we're going to just write down here the total number of items that we've got now
of data set, and we can access that by going train data loader dot data set.
So that's going to give us length of the data set contained within our train data loader,
which is you might be able to guess 60,000 or should be.
Now we have to, because we've been accumulating the train loss, this is going to be quite
high because we've been adding every single time we've calculated the loss, we've been
adding it to the train loss, the overall value per batch.
So now let's adjust if we wanted to find out, see how now we've got this line, we're outside
of the batch loop.
We want to adjust our training loss to get the average training loss per batch per epoch.
So we're coming back to the epoch loop here.
A little bit confusing, but you just line up where the loops are, and this is going to
help you figure out what context you're computing in.
So now we are in the epoch loop.
So divide total train loss by length of train data loader, oh, this is so exciting, training
our biggest model yet.
So train loss equals or divide equals, we're going to reassign the train loss, we're going
to divide it by the length of the train data loader.
So why do we do this?
Well, because we've accumulated the train loss here for every batch in the train data
loader, but we want to average it out across how many batches there are in the train data
loader.
So this value will be quite high until we readjust it to find the average loss per epoch, because
we are in the epoch loop.
All right, there are a few steps going on, but that's all right, we'll figure this out,
or what should happening in a minute, let's code up the testing loop.
So testing, what do we have to do for testing?
Well, let's set up a test loss variable.
Why don't we do accuracy for testing as well?
Did we do accuracy for training?
We didn't do accuracy for training, but that's all right, we'll stick to doing accuracy for
testing.
We'll go model zero dot eval, we'll put it in evaluation mode, and we'll turn on our
inference mode context manager with torch dot inference mode.
Now we'll do the same thing for x, y in test data loader, we don't need to keep track
of the batches here again in the test data loader.

Section 162: Main Topics

Key Topics:

- And then we'll get the test loss, of course, test loss and we'll go, we'll get that to four decimal places as well

▶ 📄 Click to view detailed content

```
So we'll just loop through x, so features, images, and labels in our test data
loader.
We're going to do the forward pass, because the test loop, we don't have an
optimization
step, we are just passing our data through the model and evaluating the patterns it
learned
on the training data.
So we're going to pass in x here.
This might be a little bit confusing, let's do this x test, y test.
That way we don't get confused with our x above for the training set.
Now we're going to calculate the loss, a cum, relatively might small that wrong app
to
sound that out.
What do we have here?
So we've got our test loss variable that we just assigned to zero above, just up
here.
So we're going to do test loss plus equals.
We're doing this in one step here.
Test spread, y test.
So we're comparing our test prediction to our y test labels, our test labels.
Now we're going to back out of the for loop here, because that's all we have to do,
the
forward pass and calculate the loss for the test data set.
Oh, I said we're going to calculate the accuracy.
Silly me.
So calculate accuracy.
Let's go test act.
And we've got plus equals.
We can bring out our accuracy function here.
That's what we downloaded from our helper functions dot pi before, y true equals y
test.
And then y pred equals test, pred dot arg max, dim equals one.
Why do we do this?
Well, because recall that the outputs of our model, the raw outputs of our model
are going
to be logits and our accuracy function expects our true labels and our predictions
to be
in the same format.
If our test pred is just logits, we have to call arg max to find the logit value
with
the highest index, and that will be the prediction label.
And so then we're comparing labels to labels.
That's what the arg max does here.
```

So we can back out of the batch loop now, and we're going to now calculate Cal queue
length, the test loss, average per batch.
So let's go here, test loss, divide equals length test data loader.
So because we were in the context of the loop here of the batch loop, our test lost and
test accuracy values are per batch and accumulated every single batch.
So now we're just dividing them by how many batches we had, test data loader, and the
same thing for the accuracy, calculate the ACK or test ACK average per batch.
So this is giving us test loss and test accuracy per epoch, test ACK divided equals length,
test data loader, wonderful, we're so close to finishing this up.
And now we'll come back to where's our epoch loop.
We can, these lines are very helpful in Google CoLab, we scroll down.
I believe if you want them, you can go settings or something like that, yeah, settings.
That's where you can get these lines from if you don't have them.
So print out what's happening.
We are going to print f equals n, let's get the train loss in here.
Ten loss and we'll print that to four decimal places.
And then we'll get the test loss, of course, test loss and we'll go, we'll get that to four
decimal places as well.
And then we'll get the test ACK, test accuracy, we'll get that to four decimal places as well.
For f, wonderful.
And then finally, one more step, ooh, we've written a lot of code in this video.
We want to calculate the training time because that's another thing that we want to track.
We want to see how long our model is taken to train.
So train time end on CPU is going to equal the timer and then we're going to get the
total train time model zero so we can set up a variable for this so we can compare our
modeling experiments later on.
We're going to go print train time, start equals train time, start on CPU and equals
train time end on CPU.
And finally, the device is going to be string next model zero dot parameters.
So we're just, this is one way of checking where our model zero parameters live.
So beautiful, all right.
Have we got enough brackets there?
I don't think we do.
Okay.
There we go.
Whoo.
I'll just show you what the output of this is.
So next, model zero dot parameters, what does this give us?
Oh, can we go device here?
Oh, what do we have here?
Model zero dot parameters.
I thought this was a little trick.
And then if we go next parameter containing.
I thought we could get device, oh, there we go.
Excuse me.
That's how we get it.

```
That's how we get the device that it's on.
So let me just turn this.
This is what the output of that's going to be CPU.
That's what we're after.
So troubleshooting on the fly here.
Hopefully all of this code works.
So we went through all of our steps.
We're looping through epochs at the top level here.
We looped through the training batches, performed the training steps.
So our training loop, forward pass, loss calculation, optimizer zero grad, loss
backwards, calculate
the loss per batch, accumulate those.
We do the same for the testing batches except without the optimizer steps and print
out
what's happening and we time it all for fun.
A fair bit going on here, but if you don't think there's any errors, give that a
go, run
that code.
I'm going to leave this one on a cliffhanger and we're going to see if this works
in the
next video.
I'll see you there.
Welcome back.
The last video was pretty full on.
We did a fair few steps, but this is all good practice.
```

Section 163: Main Topics

Key Topics:

- The best way to learn PyTorch code is to write more PyTorch code
- Oh, of course we did
- They should be in the same realm as mine, but due to inherent randomness of machine learning, even if we set the manual seed might be slightly different

▶ 📄 Click to view detailed content

```
The best way to learn PyTorch code is to write more PyTorch code.
So did you try it out?
Did you run this code?
Did it work?
Did we probably have an error somewhere?
Well, let's find out together.
You ready?
Let's train our biggest model yet in three, two, one, bomb.
Oh, of course we did.
What do we have?
What's going on?
```

Indentation error.
Ah, classic.
So print out what's happening.
Do we not have an indent there?
Oh, is that not in line with where it needs to be?
Excuse me.
Okay.
Why is this not in line?
So this is strange to me, enter.
How did this all get off by one?
I'm not sure, but this is just what you'll face.
Like sometimes you'll write this beautiful code that should work, but the main error
of your entire code is that it's off by a single space.
I'm not sure how that happened, but we're just going to pull this all into line.
We could have done this by selecting it all, but we're going to do it line by line just
to make sure that everything's in the right order, beautiful, and we print out what's
happening.
Three, two, one, round two.
We're going.
Okay.
So this is the progress bar I was talking about.
Look at that.
How beautiful is that?
Oh, we're going quite quickly through all of our samples.
I need to talk faster.
Oh, there we go.
We've got some good results.
We've got the tests, the train loss, the test loss and the test accuracy is pretty darn
good.
Oh my goodness.
This is a good baseline already, 67%.
So this is showing us it's about seven seconds per iteration.
Remember TQDM is tracking how many epochs.
We're going through.
So we have three epochs and our print statement is just saying, hey, we've looked at zero
out of 60,000 samples and we looked at 12,000 out of 60,000 samples and we finished on
an epoch two because it's zero indexed and we have a train loss of 0.4550 and a test
loss 476 and a test accuracy 834265 and a training time about just over 21 seconds or
just under 22.
So keep in mind that your numbers may not be the exact same as mine.
They should be in the same realm as mine, but due to inherent randomness of machine learning,
even if we set the manual seed might be slightly different.
So don't worry too much about that and what I mean by in the same realm, if your accuracy
is 25 rather than 83, well then probably something's wrong there.
But if it's 83.6, well then that's not too bad.
And the same with the train time on CPU, this will be heavily dependent, how long it takes

to train will be heavily dependent on the hardware that you're using behind the scenes.
So I'm using Google Colab Pro.
Now that may mean I get a faster CPU than the free version of Google Colab.
It also depends on what CPU is available in Google's computer warehouse where Google
Colab is hosting of how fast this will be.
So just keep that in mind.
If your time is 10 times that, then there's probably something wrong.
If your time is 10 times less than that, well, hey, keep using that hardware because that's
pretty darn good.
So let's keep pushing forward.
This will be our baseline that we try to improve upon.
So we have an accuracy of 83.5 and we have a train time of 20 or so seconds.
So we'll see what we can do with a model on the GPU later and then also later on a convolutional neural network.
So let's evaluate our model where we up to what we just did.
We built a training loop.
So we've done that.
That was a fair bit of code.
But now we're up to we fit the model to the data and make a prediction.
Let's do these two combined, hey, we'll evaluate our model.
So we'll come back.
Number four is make predictions and get model zero results.
Now we're going to create a function to do this because we want to build multiple models
and that way we can, if we have, say, model 0123, we can pass it to our function to evaluate
that model and then we can compare the results later on.
So that's something to keep in mind.
If you're going to be writing a bunch of code multiple times, you probably want to functionize it and we could definitely do that for our training and last loops.
But we'll see that later on.
So let's go deaf of our model.
So evaluate a given model, we'll pass it in a model, which will be a torch dot nn dot
module, what of type.
And we'll pass it in a data loader, which will be of type torch dot utils dot data dot
data loader.
And then we'll pass in the loss function so that it can calculate the loss.
We could pass in an evaluation metric if we wanted to track that too.
So this will be torch nn dot module as well.
And then, oh, there we go.
Speaking of an evaluation function, let's pass in our accuracy function as well.
And I don't want L, I want that.
So we want to return a dictionary containing the results of model predicting on data loader.
So that's what we want.
We're going to return a dictionary of model results.
That way we could call this function multiple times with different models and different
data loaders and then compare the dictionaries full of results depending on which model we
passed in here.
So let's set up loss and accuracy equals zero, zero, we'll start those off.

```
We'll go, this is going to be much the same as our testing loop above, except it's
going
```

Section 164: Main Topics

Key Topics:

- The test data loader, of course, because we want to evaluate it on the test data set

▶ 📄 Click to view detailed content

```
to be functionalized and we're going to return a dictionary.
So we'll turn on our context manager for inferencing with torch dot inference mode.
Now we're going to loop through the data loader and we'll get the x and y values.
So the x will be our data, the y will be our ideal labels, we'll make predictions
with
the model.
In other words, do the forward pass.
So we'll go y pred equals model on x.
Now we don't have to specify what model it is because we've got the model parameter
up
here.
So we're starting to make our functions here or this function generalizable.
So it could be used with almost any model and any data loader.
So we want to accumulate the loss and accuracy values per batch because this is
within the
batch loop here per batch.
And then we're going to go loss plus equals loss function, we'll pass it in the y
pred
and the y the true label and we'll do the same with the accuracy.
So except this time we use our accuracy function, we'll send in y true equals y and
y pred equals
y pred dot argmax because the raw outputs of our model are logits.
And if we want to convert them into labels, we could take the softmax for the
prediction
probabilities, but we could also take the argmax and just by skipping the softmax
step, the
argmax will get the index where the highest value load it is, dim equals one.
And then we're going to make sure that we're still within the context manager here.
So with torch inference mode, but outside the loop.
So that'll be this line here.
We're going to scale the loss and act to find the average loss slash act per batch.
So loss will divide and assign to the length of the data loader.
So that'll divide and reassign it to however many batches are in our data loader
that we
pass into our of our model function, then we'll do the same thing for the accuracy
here.
Length data loader, beautiful.
```

```
And now we're going to return a dictionary here.
So return, we can return the model name by inspecting the model.
We get an attribute of the model, which is its class name.
I'll show you how you can do that.
So this is helpful to track if you've created multiple different models and given
them different
class names, you can access the name attribute.
So this only works when model was created with a class.
So you just have to ensure that your models have different class names.
If you want to do it like that, because we're going to do it like that, we can set
the model
name to be its class name.
We'll get the model loss, which is just this value here.
After it's been scaled, we'll turn it into a single value by taking dot item.
And then we'll go model dot act, or we'll get model underscore act for the models
accuracy.
We'll do the same thing here.
Act.
I don't think we need to take the item because accuracy comes back in a different
form.
We'll find out, if in doubt, code it out.
So calculate model zero results on test data set.
And I want to let you know that you can create your own functions here to do almost
whatever
you want.
I've just decided that this is going to be helpful for the models and the data that
we're building.
But keep that in mind that your models, your data sets might be different and will
likely
be different in the future.
So you can create these functions for whatever use case you need.
Model zero results equals a vowel model.
So we're just going to call our function that we've just created here.
Model is going to equal model zero.
The data loader is going to equal what?
The test data loader, of course, because we want to evaluate it on the test data
set.
And we're going to send in our loss function, which is loss function that we
assigned above
just before our training loop.
If we come up here, our loss function is up here, and then if we go back down, we
have
our accuracy function is equal to our accuracy function.
We just pass another function in there, beautiful.
And let's see if this works.
Model zero results.
Did you see any typos likely or errors in our code?
How do you think our model did?
Well, let's find out.
Oh, there we go.
We got model accuracy.
Can you see how we could reuse this dictionary later on?
So if we had model one results, model two results, we could use these dictionaries
and compare
them all together.
So we've got our model name.
Our version zero, the model has an accuracy of 83.42 and a loss of 0.47 on the test
```

```
data
loader.
Again, your numbers may be slightly different.
They should be in the same realm.
But if they're not the exact same, don't worry too much.
If they're 20 accuracy points less and the loss is 10 times higher, then you should
probably
go back through your code and check if something is wrong.
And I believe if we wanted to do a progress bar here, could we do that?
TQDM.
Let's have a look, eh?
Oh, look at that progress bar.
That's very nice.
So that's nice and quick because it's only on 313 batches.
It goes quite quick.
So now, what's next?
Well, we've built model one, we've got a model zero, sorry, I'm getting ahead
myself.
We've got a baseline here.
We've got a way to evaluate our model.
What's our workflow say?
So we've got our data ready.
We've done that.
We've picked or built a model.
We've picked a loss function.
We've built an optimizer.
We've created a training loop.
```

Section 165: Main Topics

Key Topics:

- That will PyTorch will check if there's a GPU available with CUDA and it's not
- So no matter what hardware our system is running, PyTorch leverages it
- That is another advantage of using a relatively small data set that is already saved on PyTorch data sets

▶ 📄 Click to view detailed content

```
We've fit the model to the data.
We've made a prediction.
We've evaluated the model using loss and accuracy.
We could evaluate it by making some predictions, but we'll save that for later on
as in visualizing
some predictions.
I think we're up to improving through experimentation.
So let's give that a go, hey?
Do you recall that we trained model zero on the CPU?
```

How about we build model one and start to train it on the GPU?
So in the next section, let's create number five, is set up device agnostic code.
So we've done this one together for using a GPU if there is one.
So my challenge to you for the next video is to set up some device agnostic code.
So you might have to go into CoLab if you haven't got a GPU active, change runtime type
to GPU, and then because it might restart the runtime, you might have to rerun all of
the cells above so that we get our helper functions file back and the data and whatnot.
So set up some device agnostic code and I'll see you in the next video.
How'd you go?
You should give it a shot, did you set up some device agnostic code?
I hope you gave it a go, but let's do it together.
This won't take too long.
The last two videos have been quite long.
So if I wanted to set device agnostic code, I want to see if I have a GPU available, do
I?
I can check it from the video SMI.
That fails because I haven't activated a GPU in CoLab yet.
I can also check here, torch CUDA is available.
That will PyTorch will check if there's a GPU available with CUDA and it's not.
So let's fix these two because we want to start using a GPU and we want to set up device
agnostic code.
So no matter what hardware our system is running, PyTorch leverages it.
So we're going to select GPU here, I'm going to click save and you'll notice that our Google
CoLab notebook will start to reset and we'll start to connect.
There we go.
We've got a GPU on the back end, Python, three Google Compute Engine back end GPU.
Do we have to reset this?
NVIDIA SMI, wonderful, I have a Tesla T4 GPU with 16 gigabytes of memory, that is wonderful.
And now do we have a GPU available?
Oh, torch is not defined.
Well, do you notice the numbers of these cells?
One, two, that means because we've reset our runtime to have a GPU, we have to rerun
all the cells above.
So we can go run before, that's going to run all the cells above, make sure that we download
the data, make sure that we download the helper functions file, we go back up, we should see
our data may be downloading.
It shouldn't take too long.
That is another advantage of using a relatively small data set that is already saved on PyTorch
data sets.
Just keep in mind that if you use a larger data set and you have to re-download it into
Google Colab, it may take a while to run, and if you build bigger models, they may take
a while to run.
So just keep that in mind for your experiments going forward, start small, increase when

```
necessary.
So we'll re-run this, we'll re-run this, and finally we're going to, oh, there we
go,
we've got a GPU, wonderful, but we'll write some device-agnostic code here, set up
device-agnostic
code.
So import-torch, now realistically you quite often do this at the start of every
notebook,
but I just wanted to highlight how we might do it if we're in the middle, and I
wanted
to practice running a model on a CPU only before stepping things up and going to a
GPU.
So device equals CUDA, this is for our device-agnostic code, if torch dot CUDA is
available, and it
looks like this is going to return true, else use the CPU.
And then we're going to check device, wonderful, CUDA.
So we've got some device-agnostic code ready to go, I think it's time we built
another
model.
And I asked the question before, do you think that the data set that we're working
with
requires nonlinearity?
So the shirts, and the bags, and the shoes, do we need nonlinear functions to model
this?
Well it looks like our baseline model without nonlinearities did pretty well at
modeling
our data, so we've got a pretty good test accuracy value, so 83%, so out of 100
images
it predicts the right one, 83% of the time, 83 times out of 100, it did pretty well
without
nonlinearities.
Why don't we try a model that uses nonlinearities and it runs on the GPU?
So you might want to give that a go, see if you can create a model with nonlinear
functions,
try nn.relu, run it on the GPU, and see how it goes, otherwise we'll do it together
in
the next video, I'll see you there.
Hello everyone, and welcome back, we are making some terrific progress, let's see
how far
we've come, we've got a data set, we've prepared our data loaders, we've built a
baseline model,
and we've trained it, evaluated it, now it's time, oh, and the last video we set up
device
diagnostic code, but where are we in our little framework, we're up to improving
through experimentation,
and quite often that is building a different model and trying it out, it could be
using
more data, it could be tweaking a whole bunch of different things.
So let's get into some coding, I'm going to write it here, model one, I believe
we're
up to section six now, model one is going to be building a better model with
nonlinearity,
```

Section 166: Main Topics

Key Topics:

- so I asked you to do the challenge in the last video to give it a go, to try and build a model with nonlinearity, I hope you gave it a go, because if anything that this course, I'm trying to impart on you in this course, it's to give things a go, to try things out because that's what machine learning and coding is all about, trying things out, giving it a go, but let's write down here, we learned about the power of nonlinearity in notebook O2, so if we go to the learnpytorch

▶ 📄 Click to view detailed content

```
so I asked you to do the challenge in the last video to give it a go, to try and
build
a model with nonlinearity, I hope you gave it a go, because if anything that this
course,
I'm trying to impart on you in this course, it's to give things a go, to try things
out
because that's what machine learning and coding is all about, trying things out,
giving it
a go, but let's write down here, we learned about the power of nonlinearity in
notebook
O2, so if we go to the learnpytorch.io book, we go to section number two, we'll
just wait
for this to load, and then if we come down here, we can search for nonlinearity,
the missing
piece nonlinearity, so I'm going to get this and just copy that in there, if you
want to
see what nonlinearity helps us do, it helps us model nonlinear data, and in the
case of
a circle, can we model that with straight lines, in other words, linear lines?
All linear means straight, nonlinear means non-straight, and so we learned that
through
the power of linear and nonlinear functions, neural networks can model almost any
kind
of data if we pair them in the right way, so you can go back through and read that
there,
but I prefer to code things out and try it out on our data, so let's create a model
with
nonlinear and linear layers, but we also saw that our model with just linear layers
can
model our data, it's performing quite well, so that's where the experimentation
side of
things will come into play, sometimes you won't know what a model will do, whether
it
will work or won't work on your data set, but that is where we try different things
out, so we come up here, we look at our data, hmm, that looks actually quite linear
to
me as a bag, like it's just some straight lines, you could maybe model that with
```

just
straight lines, but there are some things which you could potentially classify as
nonlinear
in here, it's hard to tell without knowing, so let's give it a go, let's write a
nonlinear
model which is going to be quite similar to model zero here, except we're going to
intersperse
some relu layers in between our linear layers, so recall that relu is a nonlinear
activation
function, and relu has the formula, if something comes in and it's a negative
value, relu is
going to turn that negative into a zero, and if something is positive, relu is just
going
to leave it there, so let's create another class here, fashion MNIST model V1, and
we're
going to subclass from nn.module, beautiful, and then we're going to initialize our
model,
it's going to be quite the same as what we created before, we want an input shape,
that's
going to be an integer, and then we want a number of hidden units, and that's going
to be an int here, and then we want an output shape, int, and I want to stress as
well that
although we're creating a class here with these inputs, classes are as flexible as
functions,
so if you need different use cases for your modeling classes, just keep that in
mind that
you can build that functionality in, self dot layer stack, we're going to spell
layer stack
correctly, and we're going to set this equal to nn dot sequential, because we just
want
a sequential set of layers, the first one's going to be nn dot flatten, which is
going
to be flatten inputs into a single vector, and then we're going to go nn dot
linear,
because we want to flatten our stuff because we want it to be the right shape, if
we don't
flatten it, we get shape issues, input shape, and then the out features of our
linear layer
is going to be the hidden units, hidden units, I'm just going to make some code
cells here
so that my code goes into the middle of the screen, then here is where we're going
to
add a nonlinear layer, so this is where we're going to add in a relu function, and
where
might we put these? Well, generally, you'll have a linear function followed by a
nonlinear
function in the construction of neural networks. However, neural networks are as
customizable
as you can imagine, whether they work or not is a different question. So we'll go
output
shape here, as the out features, oh, do we miss this one up? Yes, we did. This
needs
to be hidden units. And why is that? Well, it's because the output shape of this
linear
layer here needs to match up with the input shape of this linear layer here. The
relu
layer won't change the shape of our data. And you could test that out by printing

```
the
different shapes if you'd like. And then we're going to finish off with another
nonlinear
layer at the end. Relu. Now, do you think that this will improve our model's
results
or not? Well, it's hard to tell without trying it out, right? So let's continue
building
our model. We have to override the forward method. Self X is going to be, we'll
give
a type in here, this is going to be a torch tensor as the input. And then we're
just going
to return what's happening here, we go self dot layer stack X. So that just means
that
X is going to pass through our layer stack here. And we could customize this, we
could
try it just with one nonlinear activation. This is actually our previous network,
just
with those commented out. All we've done is added in two relu functions. And so I'm
going to run that beautiful. And so what should we do next? Well, we shouldn't
stand
```

Section 167: Main Topics

Key Topics:

- And the learning rate, we're just going to keep it the same as our previous model
- The accuracy function is of a course going to be measuring our models accuracy

▶ 📄 Click to view detailed content

```
shaded but previously we ran our last model model zero on if we go parameters. Do
we run
this on the GPU or the CPU? On the CPU. So how about we try out our fashion MNIST
model
or V one running on the device that we just set up which should be CUDA. Wonderful.
So
we can instantiate. So create an instance of model one. So we want model one or
actually
we'll set up a manual seed here so that whenever we create a new instance of a
model, it's
going to be instantiated with random numbers. We don't necessarily have to set a
random
seed, but we do so anyway so that our values are quite similar on your end and my
end input
shape is going to be 784. Where does that come from? Well, that's because this is
the
output of the flatten layer after our 28 by 28 image goes in. Then we're going to
set
up the hidden units. We're going to use the same number of hidden units as before,
```

which
is going to be 10. And then the output shape is what? We need one value, one output
neuron
for each of our classes. So length of the class names. And then we're going to send
this to the target device so we can write send to the GPU if it's available. So now
that we've set up device agnostic code in the last video, we can just put two
device
instead of hard coding that. And so if we check, so this was the output for model
zero's device,
let's now check model one's device, model one parameters, and we can check where
those
parameters live by using the device attribute. Beautiful. So our model one is now
living
on the GPU CUDA at index zero. Index zero means that it's on the first GPU that we
have
available. We only have one GPU available. So it's on this Tesla T for GPU. Now,
we've
got a couple more things to do. Now that we've created another model, we can
recreate if
we go back to our workflow, we've just built a model here. What do we have to do
after
we built a model? We have to instantiate a loss function and an optimizer. Now
we've
done both of those things for model zero. So that's what we're going to do in the
next
video. But I'd like you to go ahead and try to create a loss function for our model
and
optimizer for model one. The hint is that they can be the exact same loss function
and
optimizer as model zero. So give that a shot and I'll see you in the next video.
Welcome
back. In the last video, we created another model. So we're continuing with our
modeling
experiments. And the only difference here between fashion MNIST model V1 and V0 is
that
we've added in nonlinear layers. Now we don't know for now we could think or guess
whether
they would help improve our model. And with practice, you can start to understand
how
different functions will influence your neural networks. But I prefer to, if in
doubt, code
it out, run lots of different experiments. So let's continue. We now have to create
a loss function, loss, optimizer, and evaluation metrics. So we've done this for
model zero.
So we're not going to spend too much time explaining what's going on here. And
we've
done this a fair few times now. So from helper functions, which is the script we
downloaded
before, we're going to import our accuracy function. And we're going to set up a
loss
function, which is we're working with multi class classification. So what loss
function
do we typically use? And then dot cross entropy loss. And as our optimizer is going
to be
torch dot opt in dot SGD. And we're going to optimize this time. I'll put in the
params
keyword here, model one dot parameters. And the learning rate, we're just going to

```
keep
it the same as our previous model. And that's a thing to keep a note for your
experiments.
When you're running fair few experiments, you only really want to tweak a couple of
things
or maybe just one thing per experiment, that way you can really narrow down what
actually
influences your model and what improves it slash what doesn't improve it. And a
little
pop quiz. What does a loss function do? This is going to measure how wrong our
model is.
And what does the optimizer do? Tries to update our models parameters to reduce the
loss. So that's what these two functions are going to be doing. The accuracy
function is
of a course going to be measuring our models accuracy. We measure the accuracy
because that's
one of the base classification metrics. So we'll run this. Now what's next? We're
getting
quite good at this. We've picked a loss function and an optimizer. Now we're going
to build
a training loop. However, we spent quite a bit of time doing that in a previous
video.
If we go up here, that was our vowel model function. Oh, that was helpful. We
turned it
into a function. How about we do the same with these? Why don't we make a function
for
our training loop as well as our testing loop? So I think you can give this a go.
We're going
to make a function in the next video for training. We're going to call that train
step. And
we'll create a function for testing called test step. Now they'll both have to take
in
some parameters. I'll let you figure out what they are. But otherwise, we're going
to code
that up together in the next video. So I'll see you there.
So we've got a loss function ready and an optimizer. What's our next step? Well,
it's
```

Section 168: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
to create training and evaluation loops. So let's make a heading here. We're going
to
call this functionizing training and evaluation or slash testing loops because
we've written
similar code quite often for training and evaluating slash testing our models. Now
we're
```

going to start moving towards functionizing code that we've written before because that's
not only a best practice, it helps reduce errors because if you're writing a training
loop all the time, we may get it wrong. If we've got one that works for our particular
problem, hey, we might as well save that as a function so we can continually call that
over and over and over again. So how about we, and this is going to be very rare that
I'm going to allow you to do this is that is we're going to copy this training and you
might have already attempted to create this. That is the function called, let's create
a function for one training loop. And we're going to call this train step. And we're going
to create a function for the testing loop. You're going to call this test step. Now these
are just what I'm calling them. You can call them whatever you want. I just understand
it quite easily by calling it train step. And then we can for each epoch in a range,
we call our training step. And then the same thing for each epoch in a range, we can call
a testing step. This will make a lot more sense once we've coded it out. So let's put
the training code here. To functionize this, let's start it off with train step. Now what
parameters should our train step function take in? Well, let's think about this. We
need a model. We need a data loader. We need a loss function. And we need an optimizer.
We could also put in an accuracy function here if we wanted to. And potentially it's
not here, but we could put in what target device we'd like to compute on and make our
code device agnostic. So this is just the exact same code we went through before. We
loop through a data loader. We do the forward pass. We calculate the loss. We accumulate
it. We zero the optimizer. We perform backpropagation in respect to the loss with the parameters
of the model. And then we step the optimizer to hopefully improve the parameters of our
model to better predict the data that we're trying to predict. So let's craft a train
step function here. We'll take a model, which is going to be torch nn.module, type hint.
And we're going to put in a data loader, which is going to be of type torch utils dot data
dot data loader. Now we don't necessarily need to put this in these type hints, but
they're relatively new addition to Python. And so you might start to see them more and
more. And it also just helps people understand what your code is expecting. So the loss
function, we're going to put in an optimizer torch dot opt in, which is a type optimizer.
We also want an accuracy function. We don't necessarily need this either. These are

```
a
lot of nice to habs. The first four are probably the most important. And then the
device. So
torch is going to be torch dot device equals device. So we'll just hard code that
to be
our already set device parameter. And we'll just write in here, performs training
step
with model, trying to learn on data loader. Nice and simple, we could make that
more
explanatory if we wanted to, but we'll leave it at that for now. And so right at
the start,
we're going to set up train loss and train act equals zero zero. We're going to
introduce
accuracy here. So we can get rid of this. Let's just go through this line by line.
What
do we need to do here? Well, we've got four batch XY in enumerate train data
loader. But
we're going to change that to data loader up here. So we can just change this to
data
loader. Wonderful. And now we've got model zero dot train. Do we want that? Well,
no,
because we're going to keep this model agnostic, we want to be able to use any
model with this
function. So let's get rid of this model dot train. We are missing one step here is
put data on target device. And we could actually put this model dot train up here.
Put model
into training mode. Now, this will be the default for the model. But just in case
we're
going to call it anyway, model dot train, put data on the target device. So we're
going
to go XY equals X dot two device, Y dot two device. Wonderful. And the forward
pass, we
don't need to use model zero anymore. We're just going to use model that's up here.
The
loss function can stay the same because we're passing in a loss function up there.
The train
loss can be accumulated. That's fine. But we might also accumulate now the train
accuracy,
limit loss, and accuracy per batch. So train act equals or plus equals our accuracy
function
on Y true equals Y and Y pred equals Y pred. So the outputs here, Y pred, we need
to take
because the raw outputs, outputs, the raw logits from the model, because our
accuracy
function expects our predictions to be in the same format as our true values. We
need
to make sure that they are we can call the argmax here on the first dimension. This
is
going to go from logits to prediction labels. We can keep the optimizer zero grab
the same
because we're passing in an optimizer up here. We can keep the loss backwards
because the
```

Section 169: Main Topics

Key Topics:

- I hope you did because that's the best way to practice PyTorch code is to write more pytorch code
- Because we need a model and we need data, the data loader is going to be, of course, the test data load here, torch dot utils dot data dot data loader

▶ 📄 Click to view detailed content

```
loss is just calculated there. We can keep optimizer step. And we could print out
what's
happening. But we might change this up a little bit. We need to divide the total
train loss
and accuracy. I just want to type in accuracy here because now we've added in
accuracy metric
act. So train act divided equals length train data loader. Oh, no, sorry. We can
just use
the data loader here, data loader, data loader. And we're not going to print out
per batch
here. I'm just going to get rid of this. We'll make at the end of this step, we
will make
our print out here, print. Notice how it's at the end of the step because we're
outside
the for loop now. So we're going to here, we're accumulating the loss on the
training
data set and the accuracy on the training data set per batch. And then we're
finding
out at the end of the training steps. So after it's been through all the batches in
the data loader, we're finding out what the average loss is per batch. And the
average
accuracy is per batch. And now we're going to go train loss is going to be the
train
loss on 0.5. And then we're going to go train act is going to be train act. And
we're going
to set that to 0.2 F. Get that there, percentage. Wonderful. So if all this works,
we should
be able to call our train step function and pass it in a model, a data loader, a
loss
function, an optimizer, an accuracy function and a device. And it should
automatically
do all of these steps. So we're going to find that out in a later video. In the
next video,
we're going to do the same thing we've just done for the training loop with the
test step.
But here's your challenge for this video is to go up to the testing loop code we
wrote
before and try to recreate the test step function in the same format that we've
done here. So
give that a go. And I'll see you in the next video. Welcome back. In the last
```

```
video, we
functionalized our training loop. So now we can call this train step function. And
instead
of writing all this training loop code again, well, we can train our model through
the art
of a function. Now let's do the same for our testing loop. So I issued you the
challenge
in the last video to give it a go. I hope you did because that's the best way to
practice
PyTorch code is to write more pytorch code. Let's put in a model, which is going to
be
torch and then dot module. And we're going to put in a data loader. Because we need
a
model and we need data, the data loader is going to be, of course, the test data
load
here, torch dot utils dot data dot data loader. And then we're going to put in a
loss function,
which is going to be torch and end up module as well. Because we're going to use an
end
up cross entropy loss. We'll see that later on. We're going to put in an accuracy
function.
We don't need an optimizer because we're not doing any optimization in the testing
loop.
We're just evaluating. And the device can be torch dot device. And we're going to
set
that as a default to the target device parameter. Beautiful. So we'll put a little
doctoring
here. So performs a testing loop step on model going over data loader. Wonderful.
So now
let's set up a test loss and a test accuracy, because we'll measure test loss and
accuracy
without testing loop function. And we're going to set the model into, I'll just put
a comment
here, put the model in a vowel mode. So model dot a vowel, we don't have to use any
underscore
here as in model zero, because we have a model coming in the top here. Now, what
should we
do? Well, because we're performing a test step, we should turn on inference mode.
So
turn on inference mode, inference mode context manager. Remember, whenever you're
performing
predictions with your model, you should put it in model dot a vowel. And if you
want as
many speedups as you can get, make sure the predictions are done within the
inference
mode. Because remember, inference is another word for predictions within the
inference
mode context manager. So we're going to loop through our data loader for X and Y in
data
loader. We don't have to specify that this is X test. For Y test, we could if we
wanted
to. But because we're in another function here, we can just go for X, Y in data
loader,
we can do the forward pass. After we send the data to the target device, target
device,
so we're going to have X, Y equals X dot two device. And the same thing with Y,
we're
```

```
just doing best practice here, creating device agnostic code. Then what should we
do? Well,
we should do the thing that I said before, which is the forward pass. Now that our
data
and model be on the same device, we can create a variable here test pred equals
model, we're
going to pass in X. And then what do we do? We can calculate the loss. So to
calculate
the loss slash accuracy, we're going to accumulate it per batch. So we'll set up
test loss equals
loss function. Oh, plus equals loss function. We're going to pass it in test pred
and Y,
which is our truth label. And then the test act where you will accumulate as well,
using
our accuracy function, we'll pass in Y true equals Y. And then Y pred, what do we
have
```

Section 170: Main Topics

Key Topics:

- This is important because if we want to adapt a value created inside the context manager, we have to modify it still with inside that context manager, otherwise pytorch will throw an error
- Did you create a training loop or a PyTorch optimization loop using our training step function and a test step function

▶ 📄 Click to view detailed content

```
to do to Y pred? Well, our test pred, we have to take the argmax to convert it
from.
So this is going to outputs raw logits. Remember, a models raw output is referred
to as logits.
And then here, we have to go from logits to prediction labels. Beautiful. Oh,
little typo
here. Did you catch that one? Tab, tab. Beautiful. Oh, look how good this function
is looking.
Now we're going to adjust the metrics. So adjust metrics and print out. You might
notice
that we're outside of the batch loop here, right? So if we draw down from this line
for
and we write some code here, we're still within the context manager. This is
important because
if we want to adapt a value created inside the context manager, we have to modify
it
still with inside that context manager, otherwise pytorch will throw an error. So
try to write
this code if you want outside the context manager and see if it still works. So
```

```
test loss, we're
going to adjust it to find out the average test loss and test accuracy per batch
across
a whole step. So we're going to go length data loader. Now we're going to print out
what's happening. Print out what's happening. So test loss, which we put in here,
well,
we're going to get the test loss. Let's get this to five decimal places. And then
we're
going to go test act. And we will get that to two decimal places. You could do this
as
many decimal as you want. You could even times it by 100 to get it in proper
accuracy format.
And we'll put a new line on the end here. Wonderful. So now it looks like we've got
functions.
I haven't run this cell yet for a training step and a test step. So how do you
think we
could replicate if we go back up to our training loop that we wrote before? How do
you think
we could replicate the functionality of this, except this time using our functions?
Well,
we could still use this for epoch and TQDM range epochs. But then we would just
call
our training step for this training code, our training step function. And we would
call
our testing step function, passing in the appropriate parameters for our testing
loop.
So that's what we'll do in the next video. We will leverage our two functions,
train
step and test step to train model one. But here's your challenge for this video.
Give
that a go. So use our training step and test step function to train model one for
three
epochs and see how you go. But we'll do it together in the next video. Welcome
back.
How'd you go? Did you create a training loop or a PyTorch optimization loop using
our training
step function and a test step function? Were there any errors? In fact, I don't
even know.
But how about we find out together? Hey, how do we combine these two functions to
create
an optimization loop? So I'm going to go torch dot manual seed 42. And I'm going to
measure
the time of how long our training and test loop takes. This time we're using a
different
model. So this model uses nonlinearities and it's on the GPU. So that's the main
thing
we want to compare is how long our model took on CPU versus GPU. So I'm going to
import
from time it, import default timer as timer. And I'm going to start the train time.
Train
time start on GPU equals timer. And then I'm just right here, set epochs. I'm going
to
set epochs equal to three, because we want to keep our training experiments as
close
to the same as possible. So we can see what little changes do what. And then it's
create
a optimization and evaluation loop using train step and test step. So we're going
```

```
to loop
through the epochs for epoch in TQDM. So we get a nice progress bar in epochs. Then
we're
going to print epoch. A little print out of what's going on. Epoch. And we'll get a
new
line. And then maybe one, two, three, four, five, six, seven, eight or something
like
that. Maybe I'm miscounted there. But that's all right. Train step. What do we have
to
do for this? Now we have a little doc string. We have a model. What model would we
like
to use? We'd like to use model one. We have a data loader. What data loader would
we
like to use? Well, we'd like to use our train data loader. We also have a loss
function,
which is our loss function. We have an optimizer, which is our optimizer. And we
have an accuracy
function, which is our accuracy function. And oops, forgot to put FM. And finally,
we have
a device, which equals device, but we're going to set that anyway. So how beautiful
is that
for creating a training loop? Thanks to the code that we've functionalized before.
And
just recall, we set our optimizer and loss function in a previous video. You could
bring
these down here if you really wanted to, so that they're all in one place, either
way
up. But we can just get rid of that because we've already set it. Now we're going
to do
the same thing for our test step. So what do we need here? Let's check the doc
string.
We could put a little bit more information in this doc string if we wanted to to
really
make our code more reusable, and so that if someone else was to use our code, or
even
us in the future knows what's going on. But let's just code it out because we're
just
still fresh in our minds. Model equals model one. What's our data loader going to
be for
```

Section 171: Main Topics

Key Topics:

- But we'll see a more of an example of that later on in the course
- So that's a big thing about machine learning is that it uses randomness

▶ 📄 Click to view detailed content

the test step? It's going to be our test data loader. Then we're going to set in a
loss
function, which is going to be just the same loss function. We don't need to use an
optimizer
here because we are only evaluating our model, but we can pass in our accuracy
function.
Accuracy function. And then finally, the device is already set, but we can just
pass
it in anyway. Look at that. Our whole optimization loop in a few lines of code.
Isn't that beautiful?
So these functions are something that you could put in, like our helper functions
dot
pi. And that way you could just import it later on. And you don't have to write
your
training loops all over again. But we'll see a more of an example of that later on
in
the course. So let's keep going. We want to measure the train time, right? So we're
going to create, once it's been through these steps, we're going to create train
time end
on CPU. And then we're going to set that to the timer. So all this is going to do
is
measure at value in time, once this line of code is run, it's going to run all of
these
lines of code. So it's going to perform the training and optimization loop. And
then it's
going to, oh, excuse me, this should be GPU. It's going to measure a point in time
here.
So once all this codes run, measure a point in time there. And then finally, we can
go
total train time for model one is equal to print train time, which is our function
that
we wrote before. And we pass it in a start time. And it prints the difference
between
the start and end time on a target device. So let's do that. Start equals what?
Train
time start on GPU. The end is going to be train time end on GPU. And the device is
going
to be device. Beautiful. So are you ready to run our next modeling experiment model
one?
We've got a model running on the GPU, and it's using nonlinear layers. And we want
to
compare it to our first model, which our results were model zero results. And we
have total
train time on model zero. Yes, we do. So this is what we're going for. Does our
model
one beat these results? And does it beat this result here? So three, two, one, do
we
have any errors? No, we don't. Okay. Train step got an unexpected keyword loss. Oh,
did
you catch that? I didn't type in loss function. Let's run it again. There we go.
Okay, we're
running. We've got a progress bar. It's going to output at the end of each epoch.
There
we go. Training loss. All right. Test accuracy, training accuracy. This is so
exciting. I
love watching neural networks train. Okay, we're improving per epoch. That's a good

sign.
But we've still got a fair way to go. Oh, okay. So what do we have here? Well, we didn't
beat our, hmm, it looks like we didn't beat our model zero results with the nonlinear
layers. And we only just slightly had a faster training time. Now, again, your numbers might
not be the exact same as what I've got here. Right? So that's a big thing about machine
learning is that it uses randomness. So your numbers might be slightly different. The direction
should be quite similar. And we may be using different GPUs. So just keep that in mind.
Right now I'm using a new video, SMI. I'm using a Tesla T4, which is at the time of
recording this video, Wednesday, April 20, 2022 is a relatively fast GPU for making
inference. So just keep that in mind. Your GPU in the future may be different. And your
CPU that you run may also have a different time here. So if these numbers are like 10
times higher, you might want to look into seeing if your code is there's some error.
If they're 10 times lower, well, hey, you're running it on some fast hardware. So it looks
like my code is running on CUDA slightly faster than the CPU, but not dramatically faster.
And that's probably akin to the fact that our data set isn't too complex and our model
isn't too large. What I mean by that is our model doesn't have like a vast amount of
layers. And our data set is only comprised of like, this is the layers our model has.
And our data set is only comprised of 60,000 images that are 28 by 28. So as you can imagine,
the more parameters in your model, the more features in your data, the higher this time
is going to be. And you might sometimes even find that your model is faster on CPU. So
this is the train time on CPU. You might sometimes find that your model's training
time on a CPU is in fact faster for the exact same code running on a GPU. Now, why might
that be? Well, let's write down this here. Let's go note. Sometimes, depending on your
data slash hardware, you might find that your model trains faster on CPU than GPU. Now,
why is this? So one of the number one reasons is that one, it could be that the overhead
for copying data slash model to and from the GPU outweighs the compute benefits offered
by the GPU. So that's probably one of the number one reasons is that you have to, for
data to be processed on a GPU, you have to copy it because it is by default on the CPU.
If you have to copy it to that GPU, you have some overhead time for doing that copy into
the GPU memory. And then although the GPU will probably compute faster on that data

Section 172: Main Topics

Key Topics:

- Usually if you're using a GPU like a fairly modern GPU, it will be faster at computing, deep learning or running deep learning algorithms than your general CPU
- And so if you'd like a great article on how to get the most out of your GPUs, it's a little bit technical, but this is something to keep in mind as you progress as a machine learning engineer is how to make your GPUs go burr
- Making deep learning go burr as in your GPU is going burr because it's running so fast from first principles

▶ 📄 Click to view detailed content

```
once it's there, you still have that back and forth of going between the CPU and
the
GPU. And the number two reason is that the hardware you're using has a better CPU
in
terms of compute capability than the GPU. Now, this is quite a bit rarer. Usually
if
you're using a GPU like a fairly modern GPU, it will be faster at computing, deep
learning
or running deep learning algorithms than your general CPU. But sometimes these
numbers
of compute time are really dependent on the hardware that you're running. So you'll
get
the biggest benefits of speedups on the GPU when you're running larger models,
larger
data sets, and more compute intensive layers in your neural networks. And so if
you'd like
a great article on how to get the most out of your GPUs, it's a little bit
technical,
but this is something to keep in mind as you progress as a machine learning
engineer is
how to make your GPUs go burr. And I mean that burr from first principles. There we
go. Making deep learning go burr as in your GPU is going burr because it's running
so
fast from first principles. So this is by Horace He who works on PyTorch. And it's
great. It talks about compute as a first principle. So here's what I mean by
copying
memory and compute. There might be a fair few things you're not familiar with here,
but that's okay. But just be aware bandwidth. So bandwidth costs are essentially
the cost
paid to move data from one place to another. That's what I was talking about
copying stuff
from the CPU to the GPU. And then also there's one more, where is it overhead?
Overhead is
basically everything else. I called it overhead. There are different terms for
```

different things.
This article is excellent. So I'm going to just copy this in here. And you'll find this
in the resources, by the way. So for more on how to make your models compute faster,
see here. Lovely. So right now our baseline model is performing the best in terms of results.
And in terms of, or actually our model computing on the GPU is performing faster than our CPU.
Again yours might be slightly different. For my case, for my particular hardware, CUDA
is faster. Except model zero, our baseline is better than model one. So what's to do
next? Well, it's to keep experimenting, of course. I'll see you in the next video. Welcome
back. Now, before we move on to the next modeling experiment, let's get a results dictionary
for our model one, a model that we trained on. So just like we've got one for model zero,
let's create one of these for model one results. And we can create that without a vowel model
function. So we'll go right back down to where we were. I'll just get rid of this cell.
And let's type in here, get model one results dictionary. This is helpful. So later on,
we can compare all of our modeling results, because they'll all be in dictionary format.
So we're going to model one results equals a vowel model on a model equals model one.
And we can pass in a data loader, which is going to be our test data loader. Then we
can pass in a loss function, which is going to equal our loss function. And we can pass
in our accuracy function equals accuracy function. Wonderful. And then if we check out our model
one results, what do we get? Oh, no, we get an error. Do we get the code right? That looks
right to me. Oh, what does this say runtime error expected all tensors to be on the same
device, but found at least two devices, CUDA and CPU. Of course. So why did this happen?
Well, let's go back up to our of our model function, wherever we defined that. Here we
go. Ah, I see. So this is a little gotcha in pytorch or in deep learning in general. There's
a saying in the industry that deep learning models fail silently. And this is kind of
one of those ones. It's because our data and our model are on different devices. So remember
how I said the three big errors are shape mismatches with your data and your model device
mismatches, which is what we've got so far. And then data type mismatches, which is if
your data is in the wrong data type to be computed on. So what we're going to have to
do to fix this is let's bring down our vowel model function down to where we were. And

```
just like we've done in our test step and train step functions, where we've created
device agnostic data here, we've sent our data to the target device, we'll do that
exact
same thing in our vowel model function. And this is just a note for going forward.
It's
always handy to where you can create device agnostic code. So we've got our new of
our
model function here for x, y in our data loader. Let's make our data device
agnostic. So just
like our model is device agnostic, we've sent it to the target device, we will do
the same
here, x dot two device, and then y dot two device. Let's see if that works. We will
just rerun this cell up here. I'll grab this, we're just going to write the exact
same
code as what we did before. But now it should work because we've sent our, we could
actually
also just pass in the target device here, device equals device. That way we can
pass
in whatever device we want to run it on. And we're going to just add in device
here,
device equals device. And let's see if this runs correctly. Beautiful. So if we
compare
```

Section 173: Main Topics

Key Topics:

- That's due to the inherent randomness of machine learning and deep learning
- We have an input layer, just like any other deep learning model
- And you'll notice that a lot of the code is quite similar to the code that we've been writing before for other PyTorch models

▶ 📄 Click to view detailed content

```
this to our model zero results, it looks like our baseline's still out in front.
But that's
okay. We're going to in the next video, start to step things up a notch and move on
to convolutional
neural networks. This is very exciting. And by the way, just remember, if your
numbers
here aren't exactly the same as mine, don't worry too much. If they're out
landishly different,
just go back through your code and see if it's maybe a cell hasn't been run
correctly
or something like that. If there are a few decimal places off, that's okay. That's
due
to the inherent randomness of machine learning and deep learning. But with that
being said,
```

I'll see you in the next video. Let's get our hands on convolutional neural
networks.
Welcome back. In the last video, we saw that our second modeling experiment, model
one,
didn't quite beat our baseline. But now we're going to keep going with modeling
experiments.
And we're going to move on to model two. And this is very exciting. We're going to
build
a convolutional neural network, which are also known as CNN. CNNs are also known as
com net. And CNNs are known for their capabilities to find patterns in visual data.
So what are
we going to do? Well, let's jump back into the keynote. We had a look at this slide
before
where this is the typical architecture of a CNN. There's a fair bit going on here,
but
we're going to step through it one by one. We have an input layer, just like any
other
deep learning model. We have to input some kind of data. We have a bunch of hidden
layers
in our case in a convolutional neural network, you have convolutional layers. You
often have
hidden activations or nonlinear activation layers. You might have a pooling layer.
You
generally always have an output layer of some sort, which is usually a linear
layer. And
so the values for each of these different layers will depend on the problem you're
working
on. So we're going to work towards building something like this. And you'll notice
that
a lot of the code is quite similar to the code that we've been writing before for
other
PyTorch models. The only difference is in here is that we're going to use different
layer types. And so if we want to visualize a CNN in a colored block edition, we're
going
to code this out in a minute. So don't worry too much. We have a simple CNN. You
might
have an input, which could be this image of my dad eating some pizza with two
thumbs
up. We're going to preprocess that input. We're going to, in other words, turn it
into
a tensor in red, green and blue for an image. And then we're going to pass it
through a
combination of convolutional layers, relu layers and pooling layers. Now again,
this
is a thing to note about deep learning models. I don't want you to get too bogged
down in
the order of how these layers go, because they can be combined in many different
ways.
In fact, research is coming out almost every day, every week about how to best
construct
these layers. The overall principle is what's more important is how do you get your
inputs
into an idolized output? That's the fun part. And then of course, we have the
linear output
layer, which is going to output however many classes or value for however many
classes
that we have in the case of classification. And then if you want to make your CNN

```
deeper,
this is where the deep comes from deep learning, you can add more layers. So the
theory behind
this, or the practice behind this, is that the more layers you add to your deep
learning
model, the more chances it has to find patterns in the data. Now, how does it find
these patterns?
Well, each one of these layers here is going to perform, just like what we've seen
before,
a different combination of mathematical operations on whatever data we feed it. And
each subsequent
layer receives its input from the previous layer. In this case, there are some
advanced
networks that you'll probably come across later in your research and machine
learning
career that use inputs from layers that are kind of over here or the way down here
or
something like that. They're known as residual connections. But that's beyond the
scope of
what we're covering for now. We just want to build our first convolutional neural
network.
And so let's go back to Google Chrome. I'm going to show you my favorite website to
learn
about convolutional neural networks. It is the CNN explainer website. And this is
going
to be part of your extra curriculum for this video is to spend 20 minutes clicking
and
going through this entire website. We're not going to do that together because I
would
like you to explore it yourself. That is the best way to learn. So what you'll
notice up
here is we have some images of some different sort. And this is going to be our
input. So
let's start with pizza. And then we have a convolutional layer, a relu layer, a
conv
layer, a relu layer, max pool layer, com to relu to com to relu to max pool to this
architecture is a convolutional neural network. And it's running live in the
browser. And
so we pass this image, you'll notice that it breaks down into red, green and blue.
And
then it goes through each of these layers and something happens. And then finally,
we
have an output. And you notice that the output has 10 different classes here,
because we
have one, two, three, four, five, six, seven, eight, nine, 10, different classes of
image
```

Section 174: Main Topics

Key Topics:

- And of course, we could change this if we had 100 classes, we might change this to 100
- And yes, of course, there is if this is the first time you ever seen this
- But essentially, what's happening is a kernel, which is also known as a filter, is going over our image pixel values, because of course, they will be in the format of a tensor

▶ 📄 Click to view detailed content

```
in this demo here. And of course, we could change this if we had 100 classes, we
might
change this to 100. But the pieces of the puzzle here would still stay quite the
same.
And you'll notice that the class pizza has the highest output value here, because
our
images of pizza, if we change to what is this one, espresso, it's got the highest
value there. So this is a pretty well performing convolutional neural network. Then
we have
a sport car. Now, if we clicked on each one of these, something is going to happen.
Let's
find out. We have a convolutional layer. So we have an input of an image here that
64
64 by three. This is color channels last format. So we have a kernel. And this
kernel, this
is what happens inside a convolutional layer. And you might be going, well, there's
a lot
going on here. And yes, of course, there is if this is the first time you ever seen
this.
But essentially, what's happening is a kernel, which is also known as a filter, is
going
over our image pixel values, because of course, they will be in the format of a
tensor. And
trying to find small little intricate patterns in that data. So if we have a look
here, and
this is why it's so valuable to go through this and just play around with it, we
start
in a top left corner, and then slowly move along, you'll see on the output on the
right
hand side, we have another little square. And do you notice in the middle all of
those
numbers changing? Well, that is the mathematical operation that's happening as a
convolutional
layer convolves over our input image. How cool is that? And you might be able to
see on the
output there that there's some slight values for like, look around the headlight
here. Do
you notice on the right how there's some activation? There's some red tiles there?
Well, that
just means that potentially this layer or this hidden unit, and I want to zoom out
for
a second, is we have 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 hidden units. Each one of these
is
going to learn a different feature about the data. And now the beauty of deep
```

learning,
but also one of the curses of deep learning is that we don't actually control what each
one of these learns. The magic of deep learning is that it figures it out itself what is
best to learn. We go into here, notice that each one we click on has a different representation
on the right hand side. And so this is what's going to happen layer by layer as it goes
through the convolutional neural network. And so if you want to read about what is a convolutional
neural network, you can go through here. But we're going to replicate this exact neural
network here with PyTorch code. That's how I'd prefer to learn it. But if you want the
intuition behind it, the math behind it, you can check out all of these resources here.
That is your extra curriculum for this video. So we have an input layer, we have a convolutional
layer, you can see how the input gets modified by some sort of mathematical operation, which
is of course, the convolutional operation. And we have there all different numbers finding
different patterns and data. This is a really good example here. You notice that the outputs
eyes slightly changes, that'll be a trend throughout each layer. And then we can understand
the different hyper parameters, but I'm going to leave this for you to explore on your own.
In the next video, we're going to start to write PyTorch code to replicate everything
that's going on here. So I'm going to link this in here to find out what's happening
inside CNN. See this website here. So join me in the next video. This is super exciting.
We're going to build our first convolutional neural network for computer vision. I'll see
you there. Welcome back. In the last video, we went briefly through the CNN explainer
website, which is my favorite resource for learning about convolutional neural networks.
And of course, we could spend 20 minutes clicking through everything here to find out what's
going on with a convolutional neural network, or we could start to code one up. So how about
we do that? Hey, if and down, code it out. So we're going to create a convolutional neural
network. And what I'm going to do is I'm going to build this, or we're going to build this
model together in this video. And then because it's going to use layers or PyTorch layers
that we haven't looked at before, we're going to spend the next couple of videos stepping
through those layers. So just bear with me, as we code this entire model together, we'll
go break it down in subsequent videos. So let's build our first convolutional neural

```
network. That's a mouthful, by the way, I'm just going to probably stick to saying
CNN.
Fashion MNIST, we're up to model V2. We're going to subclass nn.module, as we
always do
when we're building a PyTorch model. And in here, we're going to say model
architecture
that replicates the tiny VGG. And you might be thinking, where did you get that
from, Daniel?
Model from CNN explainer website. And so oftentimes, when convolutional neural
networks or new
types of architecture come out, the authors of the research paper that present the
model
get to name the model. And so that way, in the future, you can refer to different
types of
model architectures with just a simple name, like tiny VGG. And people kind of know
what's going on.
So I believe somewhere on here, it's called tiny VGG, tiny VGG. We have nothing.
Yeah,
```

Section 175: Main Topics

Key Topics:

- Well, one, you could go, of course, to the PyTorch documentation, PyTorch, and then com 2d

▶ 📄 Click to view detailed content

```
there we go. In tiny VGG. And do we have more than one tiny, tiny, yeah, tiny VGG.
And if we
look up VGG, conv net, VGG 16 was one of the original ones, VGG, very deep
convolutional neural
networks of VGG net. There's also ResNet, which is another convolutional neural
network.
You can also, I don't want to give you my location, Google, you can go popular CNN
architectures. And this will give you a fair few options. Lynette is one of the
first AlexNet,
ZF net, whole bunch of different resources. And also, how could you find out more
about a
convolutional neural network? What is a convolutional neural network? You can go
through that. But
let's stop that for a moment. Let's code this one up together. So we're going to
initialize our
class here, def init. We're going to pass it in an input shape, just like we often
do.
We're going to put in a number of hidden units, which is an int. And we're going to
put in an
output shape, which is an int. Wonderful. So nothing to outlandish that we haven't
seen before there.
```

And we're going to go super dot init to initialize our initializer for lack of a better way of
putting it. Now, we're going to create our neural network in a couple of blocks this time. And
you might often hear in when you learn more about convolutional neural networks, or I'll just tell
you that things are referred to are often referred to as convolutional blocks. So if we go back to
our keynote, this here, this combination of layers might be referred to as a convolutional block.
And a convolutional block, a deeper CNN, might be comprised of multiple convolutional blocks.
So to add to the confusion, a block is comprised of multiple layers. And then an overall architecture
is comprised of multiple blocks. And so the deeper and deeper your models get, the more blocks
it might be comprised of, and the more layers those blocks may be comprised of within them.
So it's kind of like Lego, which is very fun. So let's put together an an ensequential.
Now, the first few layers here that we're going to create in conv block one, uh,
nn.com 2d. Oh, look at that. Us writing us our first CNN layer. And we have to define something
here, which is in channels. So this channels refers to the number of channels in your visual data.
And we're going to put in input shape. So we're defining the input shape. This is going to be
the first layer in our model. The input shape is going to be what we define when we instantiate
this class. And then the out channels. Oh, what's the out channels going to be? Well, it's going
to be hidden units, just like we've done with our previous models. Now the difference here
is that in nn.com 2d, we have a number of different hyper parameters that we can set.
I'm going to set some pretty quickly here, but then we're going to step back through them,
not only in this video, but in subsequent videos. We've got a fair bit going on here.
We've got in channels, which is our input shape. We've got out channels, which are our hidden units.
We've got a kernel size, which equals three. Or this could be a tuple as well, three by three.
But I just like to keep it as three. We've got a stride and we've got padding. Now,
because these are values, we can set ourselves. What are they referred to as?
Let's write this down. Values, we can set ourselves in our neural networks.
In our nn's neural networks are called hyper parameters. So these are the hyper parameters
of nn.com 2d. And you might be thinking, what is 2d for? Well, because we're working with
two-dimensional data, our images have height and width. There's also com 1d for one-dimensional data,
3d for three-dimensional data. We're going to stick with 2d for now.
And so what do each of these hyper parameters do? Well, before we go through what each one of them
do, we're going to do that when we step by step through this particular layer. What we've just done

```
is we've replicated this particular layer of the CNN explainer website. We've still
got the
relu. We've still got another conv and a relu and a max pool and a conv and a relu
and a
conv and a relu and a max pool. But this is the block I was talking about. This is
one block here
of this neural network, or at least that's how I've broken it down. And this is
another block.
You might notice that they're comprised of the same layers just stacked on top of
each other.
And then we're going to have an output layer. And if you want to learn about where
the hyper
parameters came from, what we just coded, where could you learn about those? Well,
one, you could
go, of course, to the PyTorch documentation, PyTorch, and then com 2d. You can read
about it there.
There's the mathematical operation that we talked about or briefly stepped on
before,
or touched on, stepped on. Is that the right word? So create a conv layer. It's
there.
But also this is why I showed you this beautiful website so that you can read about
these
hyper parameters down here. Understanding hyper parameters. So your extra
curriculum for this
video is to go through this little graphic here and see if you can find out what
padding means,
what the kernel size means, and what the stride means. I'm not going to read
through this for you.
You can have a look at this interactive plot. We're going to keep coding because
that's what
```

Section 176: Main Topics

Key Topics:

- Now, of course, we can change all of these values later on, but just bear with me while we set them how they are
- And of course, this can be a tuple as well
- And so as our data, this is a trend in all of deep learning, actually

▶ 📄 Click to view detailed content

```
we're all about here. If and out, code it out. So we're going to now add a relu
layer.
And then after that, we're going to add another conv 2d layer. And the in channels
here is going
to be the hidden units, because we're going to take the output size of this layer
and use it as
```

the input size to this layer. We're going to keep going here. Out channels equals hidden units again
in this case. And then the kernel size is going to be three as well. Stride will be one. Padding
will be one. Now, of course, we can change all of these values later on, but just bear with me
while we set them how they are. We'll have another relu layer. And then we're going to finish off
with a nn max pool 2d layer. Again, the 2d comes from the same reason we use comf2d. We're working
with 2d data here. And we're going to set the kernel size here to be equal to two. And of course,
this can be a tuple as well. So it can be two two. Now, where could you find out about nn max
pool 2d? Well, we go nn max pool 2d. What does this do? applies a 2d max pooling over an input
signal composed of several input planes. So it's taking the max of an input. And we've got some
parameters here, kernel size, the size of the window to take the max over. Now, where have we
seen a window before? I'm just going to close these. We come back up. Where did we see a window?
Let's dive into the max pool layer. See where my mouse is? Do you see that two by two? Well,
that's a window. Now, look at the difference between the input and the output. What's happening?
Well, we have a tile that's two by two, a window of four. And the max, we're taking the max of that
tile. In this case, it's zero. Let's find the actual value. There we go. So if you look at those
four numbers in the middle inside the max brackets, we have 0.07, 0.09, 0.06, 0.05. And the max of
all those is 0.09. And you'll notice that the input and the output shapes are different. The
output is half the size of the input. So that's what max pooling does, is it tries to take the max
value of whatever its input is, and then outputs it on the right here. And so as our data,
this is a trend in all of deep learning, actually. As our image moves through, this is what you'll
notice. Notice all the different shapes here. Even if you don't completely understand what's going
on here, you'll notice that the two values here on the left start to get smaller and smaller as
they go through the model. And what our model is trying to do here is take the input and learn a
compressed representation through each of these layers. So it's going to smoosh and smoosh and
smoosh trying to find the most generalizable patterns to get to the ideal output. And that
input is eventually going to be a feature vector to our final layer. So a lot going on there,
but let's keep coding. What we've just completed is this first block. We've got a cons layer,
a relu layer, a cons layer, a relu layer, and a max pool layer. Look at that, cons layer,
relu layer, cons layer, relu layer, max pool. Should we move on to the next block?

```
We can do this
one a bit faster now because we've already coded the first one. So I'm going to do
nn.sequential as
well. And then we're going to go nn.com2d. We're going to set the in channels. What
should the
in channels be here? Well, we're going to set it to hidden units as well because
our network is
going to flow just straight through all of these layers. And the output size of
this is going to
be hidden units. And so we want the in channels to match up with the previous
layers out channels.
So then we're going to go out channels equals hidden units as well. We're going to
set the
kernel size, kernel size equals three, stride equals one, padding equals one, then
what comes
next? Well, because the two blocks are identical, the con block one and com two, we
can just go
the exact same combination of layers. And then relu and n.com2d in channels equals
hidden units.
Out channels equals, you might already know this, hidden units. Then we have kernel
size
equals three, oh, 32, don't want it that big, stride equals one, padding equals
one,
and what comes next? Well, we have another relu layer, relu, and then what comes
after that?
We have another max pool. And then max pool 2d, kernel size equals two, beautiful.
Now,
what have we coded up so far? We've got this block, number one, that's what this
one on the inside
here. And then we have com two, relu two, com two, relu two, max pool two. So we've
built these
two blocks. Now, what do we need to do? Well, we need an output layer. And so what
did we do before
when we made model one? We flattened the inputs of the final layer before we put
them to the last
linear layer. So flatten. So this is going to be the same kind of setup as our
classifier layer.
Now, I say that on purpose, because that's what you'll generally hear the last
output layer
in a classification model called is a classifier layer. So we're going to have
these two layers
are going to be feature extractors. In other words, they're trying to learn the
patterns that
```

Section 177: Main Topics

Key Topics:

- Now, that is actually very common practice in machine learning is to find some sort of architecture that someone has found to work on some sort of problem and replicate it with code and see if it works on your own problem

- And of course, we're going to send this model to the device
- Of course, typo

▶ 📄 Click to view detailed content

```
best represent our data. And this final layer is going to take those features and
classify them
into our target classes. Whatever our model thinks best suits those features, or
whatever our model
thinks those features that it learned represents in terms of our classes. So let's
code it out.
We'll go down here. Let's build our classifier layer. This is our biggest neural
network yet.
You should be very proud. We have an end of sequential again. And we're going to
pass in
an end of flatten, because the output of these two blocks is going to be a multi-
dimensional tensor,
something similar to this size 131310. So we want to flatten the outputs into a
single feature
vector. And then we want to pass that feature vector to an nn.linear layer. And
we're going to
go in features equals hidden units times something times something. Now, the reason
I do this is
because we're going to find something out later on, or time zero, just so it
doesn't error. But
sometimes calculating what you're in features needs to be is quite tricky. And I'm
going to
show you a trick that I use later on to figure it out. And then we have out
features relates
to our output shape, which will be the length of how many classes we have, right?
One value for
each class that we have. And so with that being said, let's now that we've defined
all of the
components of our tiny VGG architecture. There is a lot going on, but this is the
same methodology
we've been using the whole time, defining some components, and then putting them
together to
compute in some way in a forward method. So forward self X. How are we going to do
this?
Are we going to set X is equal to self, comp block one X. So X is going to go
through comp block one,
it's going to go through the comp 2D layer, relu layer, comp 2D layer, relu layer,
max pool layer,
which will be the equivalent of an image going through this layer, this layer, this
layer,
this layer, this layer, and then ending up here. So we'll set it to that. And then
we can print out
X dot shape to get its shape. We'll check this later on. Then we pass X through
comp block two,
which is just going to go through all of the layers in this block, which is
equivalent to
the output of this layer going through all of these layers. And then because we've
constructed a
classifier layer, we're going to take the output of this block, which is going to
be here, and we're
```

going to pass it through our output layer, or what we've termed it, our classifier layer. I'll just
print out X dot shape here, so we can track the shape as our model moves through the architecture.
X equals self dot classifier X. And then we're going to return X. Look at us go. We just built
our first convolutional neural network by replicating what's on a CNN explainer website.
Now, that is actually very common practice in machine learning is to find some sort of architecture
that someone has found to work on some sort of problem and replicate it with code and see if it
works on your own problem. You'll see this quite often. And so now let's instantiate a model.
Go torch dot manual C. We're going to instantiate our first convolutional neural network.
Model two equals fashion amnest. We will go model V two. And we are going to set the input shape.
Now, what will the input shape be? Well, I'll come to the layer up here. The input shape
is the number of channels in our images. So do we have an image ready to go image shape?
This is the number of color channels in our image. We have one. If we had color images,
we would set the input shape to three. So the difference between our convolutional neural network,
our CNN, tiny VGG, and the CNN explainer tiny VGG is that they are using color images. So
their input is three here. So one for each color channel, red, green and blue. Whereas we have
black and white images. So we have only one color channel. So we set the input shape to one.
And then we're going to go hidden units equals 10, which is exactly the same as what tiny VGG
has used. 10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. So that sets the hidden units value of each of our
layers. That's the power of creating an initializer with hidden units. And then finally, our output
shape is going to be what we've seen this before. This is going to be the length of our class names,
one value for each class in our data set. And of course, we're going to send this model to the
device. We're going to hit shift and enter. Oh, no, what did we get wrong? Out channels,
output shape. Where did I spell wrong? Out channels, out channels, out channels. I forgot an L.
Of course, typo. Oh, kernel size and other typo. Did you notice that?
Kernel size, kernel size, kernel size, kernel size. Where did we spell this wrong? Oh, here.
Kernel size. Are there any other typos? Probably.
A beautiful. There we go. Okay, what have we got? Initializing zero, obtenses and non-op.
Oh, so we've got an issue here and error here because I've got this. But this is just to,
there's a trick to calculating this. We're going to cover this in another video. But
pay yourself on the back. We've written a fair bit of code here. This is a

```
convolutional neural
network that replicates the tiny VGG architecture on the CNN explainer website.
Now, don't forget,
```

Section 178: Main Topics

Key Topics:

- Now, I'm super stoked because in the last video, we coded together our first ever convolutional neural network in PyTorch
- We replicated the tiny VGG architecture from the CNN explainer website, my favorite place for learning about CNNs in the browser
- Well, of course, we have the documentation and then comp2d

▶ 📄 Click to view detailed content

```
your extra curriculum is to go through this website for at least 20 minutes and
read about
what's happening in our models. We're focused on code here. But this is
particularly where you
want to pay attention to. If you read through this understanding hyper parameters
and play around
with this, the next couple of videos will make a lot more sense. So read about
padding,
read about kernel size and read about stride. I'll see you in the next video. We're
going to go
through our network step by step. Welcome back. Now, I'm super stoked because in
the last video,
we coded together our first ever convolutional neural network in PyTorch. So well
done. We
replicated the tiny VGG architecture from the CNN explainer website, my favorite
place for learning
about CNNs in the browser. So now we introduced two new layers that we haven't seen
before,
conv2d and maxpool2d. But they all have the same sort of premise of what we've been
doing so far
is that they're trying to learn the best features to represent our data in some
way, shape or form.
Now, in the case of maxpool2d, it doesn't actually have any learnable parameters.
It just takes
the max, but we're going to step through that later on. Let's use this video to
step through
and then conv2d. We're going to do that with code. So I'll make a new heading here.
7.1
stepping through and then conv2d. Beautiful. Now, where could we find out what's
going on
in an end comp2d? Well, of course, we have the documentation and then comp2d. We've
got PyTorch.
```

So if you want to learn the mathematical operation that's happening, we have this value here, this
operation here. Essentially, it's saying the output is equal to the bias term times something,
plus the sum of the weight times something times the input. So do you see how just the weight
matrix, the weight tensor and the bias value, manipulating our input in some way equals the output?
Now, if we map this, we've got batch size, channels in, height, width, channels out, out, out,
et cetera, et cetera. But we're not going to focus too much on this. If you'd like to
read more into that, you can. Let's try it with code. And we're going to reproduce this particular
layer here, the first layer of the CNN explainer website. And we're going to do it with a dummy input.
In fact, that's one of my favorite ways to test things. So I'm just going to link here the documentation.
See the documentation for an end comp2d here. And if you'd like to read through more of this,
of course, this is a beautiful place to learn about what's going on.
There's the shape how to calculate the shape, height out, width out, et cetera. That's very
helpful if you need to calculate input and output shapes. But I'll show you my trick for doing so
later on. We have here, let's create some dummy data. So I'm going to set torch manual seed. We
need it to be the same size as our CNN explainer data. So 64, 64, 3. But we're going to do it
pie torch style. This is color channels last. We're going to do color channels first. So how
about we create a batch of images, we're going to be writing torch dot rand n. And we're going to
pass in size equals 32, three, 64, 64. And then we're going to create a singular image by taking
the first of that. So image is zero. Now, let's get the image batch shape. Because a lot of
machine learning, as I've said before, and deep learning is making sure your data has the right
shape. So let's check images dot shape. And let's check single image shape. We're going to go test
image dot shape. And finally, we're going to print, what does the test image look like?
We'll get this on a new line, hey, new line test image, this is of course not going to be an
actual image is just going to be a collection of random numbers. And of course, that is what
our model is currently comprised of model two, if we have a look at what's on the insides,
we are going to see a whole bunch of random numbers. Look at all this. What do we have?
We scroll up is going to give us a name for something. We have comp block two, two, we have a weight,
we have a bias, keep going up, we go right to the top, we have another weight, keep going down,
we have a bias, a weight, et cetera, et cetera. Now, our model is comprised of random numbers,

```
and what we are trying to do is just like all of our other models is pass data in
and adjust the
random numbers within these layers to best represent our data. So let's see what
happens
if we pass some random data through one of our comp2d layers. So let's go here,
we're going to
create a single comp2d layer. So comp layer equals, what is it equal? And then
comp2d,
and we're going to set the in channels is equal to what? Oh, revealed the answer
too quickly.
Three. Why is it three? Well, it's because the in channels is the same number of
color channels
as our images. So if we have a look at our test image shape, what do we have?
Three, it has three
color channels. That is the same as the value here, except the order is reversed.
This is color
channels last, pytorch defaults to color channels first. So, or for now it does, in
the future this
may change. So just keep that in mind. So out channels equals 10. This is
equivalent to the
```

Section 179: Main Topics

Key Topics:

- Now, of course, you could find that out by reading through the documentation here
- Now, the beauty of PyTorch is it does all of this behind the scenes for us
- That's the beauty of deep learning is that it learns how to best represent our data, hopefully, on its own by looking at more data

▶ 📄 Click to view detailed content

```
number of hidden units we have. One. Oh, I don't want that one just yet. One, two,
three, four,
five, six, seven, eight, nine, 10. So we have that 10 there. So we have 10 there.
And then we have
kernel size. Oh, what is the kernel? Well, it's not KFC. I can tell you that. And
then we have
stride. And then we have padding. We're going to step through these in a second.
But let's check
out the kernel. And this kernel can also be three by three. But it's a shortcut to
just type in three.
So that's what it actually means. If you just type in a single number, it's
equivalent to typing in
a tuple. Now, of course, you could find that out by reading through the
documentation here.
But where did I get that value? Well, let's dive into this beautiful website. And
let's see what
```

happening. So we have a kernel here, which is also called a filter. So the thing I'm talking about
is this little square here, this kernel. Oh, we can see the weights there at the top. This is how
beautiful this website is. So if we go over there, this is what's going to happen. This is a convolution.
It starts with this little square, and it moves pixel by pixel across our image. And you'll notice
that the output is creating some sort of number there. And you'll notice in the middle, we have a
mathematical operation. This operation here is what's happening here. I wait times the input.
That's what we've got there. Now, the beauty of PyTorch is it does all of this behind the
scenes for us. So again, if you'd like to dig more into the mathematical operation behind the
scenes, you've got the resource here. And you've also got plenty of other resources online. We're
going to focus on code for now. So if we keep doing this across our entire image, we get this
output over here. So that's the kernel. And now where did I get three by three from? Well, look at
this. One, two, three, one, two, three, one, two, three, three by three, we have nine squares. Now,
if we scroll down, this was your extracurricular for the last video, understanding hyperparameters.
What happens if we change the kernel size to three by three? Have a look at the red square on the
left. Now, if we change it to two by two, it changed again. Three by three. This is our kernel,
or also known as a filter, passing across our image, performing some sort of mathematical
operation. And now the whole idea of a convolutional layer is to try and make sure that this kernel
performs the right operation to get the right output over here. Now, what do these kernels learn?
Well, that is entirely up to the model. That's the beauty of deep learning is that it
learns how to best represent our data, hopefully, on its own by looking at more data. And then so
if we jump back in here, so that's the equivalent of setting kernel size three by three. What if
we set the stride equal to one? Have we got this in the right order? It doesn't really matter.
Let's go through stride next. If we go to here, what does stride say? Stride of the convolution
of the convolving kernel. The default is one. Wonderful. Now, if we set the stride,
or if we keep it at one, it's a default one, it's going to hop over, watch the red square on the
left. It's going to hop over one pixel at a time. So the convolution, the convolving, happens one
pixel at a time. That's what the stride sets. Now, watch what happens when I change the stride
value to the output shape. Wow. Do you notice that it went down? So we have here, the kernel size
is still the same. But now we're jumping over two pixels at a time. Notice how on the left,

```
two pixels become available. And then if I jump over again, two pixels. So the
reason why the
output compresses is because we're skipping some pixels as we go across the image.
And now this
pattern happens throughout the entire network. That's one of the reasons why you
see the size
of our input or the size of each layer go down over time. What our convolutional
layer is doing,
and in fact, a lot of deep learning neural networks do, is they try to compress the
input
into some representation that best suits the data. Because it would be no point of
just memorizing
the exact patterns, you want to compress it in some way. Otherwise, you just might
as well move
your input data around. You want to learn generalizable patterns that you can move
around. And so we
keep going. We've got padding equals zero. Let's see what happens here. If we
change the padding
value, what happens? Up, down. Notice the size here. Oh, we've added two extra
pixels around the
edge. Now if we go down, one extra pixel. Now if we go zero, now why might we do
that?
If we add some padding on the end, well, that's so that our kernel can operate on
what's going on
here in the corner. In case there's some information on the edges of our image.
Then you might be
thinking, Daniel, there's a whole bunch of values here. How do we know what to set
them?
Well, you notice that I've just copied exactly what is going on here.
There's a three by three kernel. There's no padding on the image. And the stride is
just going
one by one. And so that's often very common in machine learning, is that when
you're just getting
started and you're not sure what values to set these values to, you just copy some
existing
```

Section 180: Main Topics

Key Topics:

- Of course we get a shape error
- One of the most common issues of machine learning and deep learning
- Now, just keep in mind that if you're running this layer and then com2d on a pytorch version, that is, I believe they fixed this or they changed it in pytorch

▶ 📄 Click to view detailed content

values from somewhere and see if it works on your own problem. And then if it doesn't, well,
you can adjust them. So let's see what happens when we do that. So pass the data through
the convolutional layer. So let's see what happens. Conv output equals conv layer.
Let's pass it our test image. And we'll check the conv output. What happens?
Oh no, we get an error. Of course we get a shape error. One of the most common issues of machine
learning and deep learning. So this is saying that our input for the conv layer expects a four
dimensional tensor, except it got a three dimensional input size of 364 64. Now, how do we add an
extra dimension to our test image? Let's have a look. How would we add a batch dimension over on
the left here? We can go unsqueeze zero. So now we have a four dimensional tensor. Now, just keep
in mind that if you're running this layer and then com2d on a pytorch version, that is, I believe
they fixed this or they changed it in pytorch. What am I on? I think this Google collab instance is
on 1.10. I think you might not get this error if you're running 1.11. So just keep that in mind.
Like this should work if you're running 1.11. But if it doesn't, you can always unsqueeze here.
And let's see what happens. Look at that. We get another tensor output. Again,
this is just all random numbers though, because our test image is just random numbers. And our
conv layer is instantiated with random numbers. But we'll set the manual seed here. Now, if our
numbers are different to what's, if your numbers are different to what's on my screen, don't worry
too much. Why is that? Because our conv layer is instantiated with random numbers. And our test
image is just random numbers as well. What we're paying attention to is the input and output shapes.
Do you see what just happened? We put our input image in there with three channels.
And now because we've set out channels to be 10, we've got 10. And we've got 62, 62. And this is
just the batch size. It just means one image. So essentially our random numbers, our test image,
have gone through the convolutional layer that we created, have gone through this mathematical
operation with regards to all the values that we've set, we've put the weight tensor, well,
actually PyTorch created that for us. PyTorch has done this whole operation for us. Thank you,
PyTorch. It's gone through all of these steps across. You could code this all by hand if you want,
but it's a lot easier and simpler to use a PyTorch layer. And it's done this. And now it's
created this output. Now, whatever this output is, I don't know, it is random numbers, but this
same process will happen if we use actual data as well. So let's see what happens if we change
the values kernel size we increase. Notice how our output has gotten smaller because we're using

```
a bigger kernel to convolve across the image. What if we put this to three, three
back to what it
was and stride of two? What do you think will happen? Well, our output size
basically halves
because we're skipping two pixels at a time. We'll put that back to one. What do
you think will
happen if we set padding to one? 64, 64. We get basically the same size because
we've added an
extra pixel around the edges. So you can play around with this. And in fact, I
encourage you to
do this is what we just did. Padding one, we just added an extra dummy zero pixel
around the edges.
So practice with this, see what happens as you pass our test image, random numbers,
through a conv 2d layer with different values here. What do you think will happen
if you change
this to 64? Give that a shot and I'll see you in the next video. Who's ready to
step through
the nn max pool 2d layer? Put your hand up. I've got my hand up. So let's do it
together, hey,
we've got 7.2. Now you might have already given this a shot yourself. Stepping
through
nn max pool 2d. And this is this is what I do for a lot of different concepts that
I haven't
gone through before is I just write some test code and see what the inputs and
outputs are.
And so where could we find out about max pool 2d? Well, of course, we've got the
documentation.
I'm just going to link this in here. Max pool 2d. In the simplest case, the output
value of
layer with input size nchw output nch out w out. By the way, this is number of
batches,
color channels, height, width. And this is the output of that layer. And kernel
size, which is
a parameter up here, k h k w can be precisely described as out is going to be the
max of some
value, depending on the kernel size and the stride. So let's have a look at that in
practice.
And of course, you can read further through the documentation here. I'll just grab
the link for
this actually. So it's here. Wonderful. And let's now first try it with our test
image that we
created above. So just highlight what the test image is. A bunch of random numbers
in the same
shape as what a single image would be if we were to replicate the image size of the
CNN explainer.
By the way, we'll have a look at a visual in a second of max pool here. But you can
go through
that on your own time. Let's if in doubt, code it out. So we're going to print out
the original
```

Section 181: Main Topics

Key Topics:

- Now, if you're using a later version of PyTorch, you might not get an error if you only use a three dimensional image tensor and pass it through a comp layer
- And of course, our conv layer is going to be instantiated with random numbers

▶ 📄 Click to view detailed content

```
image shape without unsqueezed dimension. Because recall that we had to add an
extra dimension to
pass it through our com2d layer. Now, if you're using a later version of PyTorch,
you might not
get an error if you only use a three dimensional image tensor and pass it through a
comp layer.
So we're going to pass it in test image, original shape, test image dot shape. So
this is just going
to tell us what the line of code in the cell above tells us. But that's fine. I
like to make
pretty printouts, you know, test image with unsqueezed dimension. So this is just
going to be our test
image. And we're going to see what happens when we unsqueeze a dimension, unsqueeze
on zero
for dimension. That is about to say first, but it's the zero. Now we're going to
create a sample
nn max pool 2d layer. Because remember, even layers themselves in torch dot nn are
models
of their own accord. So we can just create a single, this is like creating a single
layer model here.
We'll set the kernel size equal to two. And recall, if we go back to CNN explainer,
kernel size equal to two results in a two by two square, a two by two kernel that's
going to
convolve over our image, like so. And this is an example input, an example output.
And you can see
the operation that max pooling does here. So just keep that in mind as we pass some
sample data
through our max pool layer. And now let's pass data through it. I actually will
pass it through
just the conv layer first, through just the conv layer. Because that's sort of how
you might stack
things, you might put a convolutional layer and then a max pool layer on top of
that convolutional
layer. So test image through conv. We'll create a variable here, equals our conv
layer.
Is going to take as an input, our test image dot unsqueeze on the zero dimension
again.
Beautiful. Now we're going to print out the shape here. This is just highlighting
how I
like to troubleshoot things is I do one step, print the shape, one step, print the
shape,
see what is happening as our data moves through various layers. So test image
through conv.shape,
we'll see what our conv layer does to the shape of our data. And then we're going
to pass data through
max pool layer, which is the layer we created a couple of lines above this one
```

here.
So let's see what happens. Test image through current type at the moment through conv and max
pool. So quite a long variable name here, but this is to help us avoid confusion of what's
going on. So we go test image through conv. So you notice how we're taking the output of our
convolutional layer, this here, and we're passing it through our max pool layer, which has another
typo. Wonderful. And finally, we'll print out the shape, shape after going through conv layer
and max pool layer. What happens here? So we want test image through conv and max pool.
Let's see how our max pool layer manipulates our test images shape. You ready? Three, two,
one, let's go. What do we get? Okay. So we have the test image original shape,
recall that our test image is just a collection of random numbers. And of course, our conv layer
is going to be instantiated with random numbers. And max pool actually has no parameters. It just
takes the maximum of a certain range of inner tensor. So when we unsqueeze the test image as the
input, we get an extra dimension here. When we pass it through our conv layer. Oh, where did this
64 come from? 164 64 64 64. Let's go back up to our conv layer. Do you notice how that we get the
64 there because we changed the out channels value? If we change this back to 10, like what's in the
CNN explainer model? One, two, three, four, five, six, seven, eight, nine, 10. What do you think will
happen there? Well, we get a little highlight here. 10. Then we keep going. I'll just get rid of
this extra cell. We don't need to check the version anymore. We'll check the test image
shapes still three 64 64. But then as we pass it through the conv layer here, we get a different
size now. So it originally had three channels as the input for color channels, but we've upscaled
it to 10 so that we have 10 hidden units in our layer. And then we have 64 64. Now, again,
these shapes will change if we change the values of what's going on here. So we might put padding
to zero. What happens there? Instead of 64 64, we get 62 62. And then what happens after we pass
it through the conv layer and then through the max pool layer? We've got 110 64 64. And now we have
110 32 32. Now, why is that? Well, let's go back into the CNN explainer, jump into this max pool
layer here. Maybe this one because it's got a bit more going on. Do you notice on the left here is
the input? And we've got a two by two kernel here. And so the max pooling layer, what it does is it
takes the maximum of whatever the input is. So you'll notice the input is 60 60 in this case.
Whereas the output over here is 30 30. Now, why is that? Well, because the max operation here is
reducing it from section of four numbers. So let's get one with a few different

```
numbers.
There we go. That'll do. So it's taking it from four numbers and finding the
maximum value within
```

Section 182: Main Topics

Key Topics:

- So as we've discussed before, what deep learning neural network is trying to do or in this case, a CNN is take some input data and figure out what features best represent whatever the input data is and compress them into a feature vector that is going to be our output
- And now, of course, you could customize this value here
- That's part of the experimental nature of machine learning, but we're going to keep it at two for now

▶ 📄 Click to view detailed content

```
those four numbers here. Now, why would it do that? So as we've discussed before,
what deep learning
neural network is trying to do or in this case, a CNN is take some input data and
figure out
what features best represent whatever the input data is and compress them into a
feature vector
that is going to be our output. Now, the reason being for that is because you could
consider it
from a neural networks perspective is that intelligence is compression. So you're
trying to
compress the patterns that make up actual data into a smaller vector space, go from
a higher
dimensional space to a smaller vector space in terms of dimensionality of a tensor.
But still,
this smaller dimensionality space represents the original data and can be used to
predict on future
data. So that's the idea behind Max Paul is, hey, if we've got these learned
features from our
convolutional layers, will the patterns, will the most important patterns stay
around if we just
take the maximum of a certain section? So do you notice how the input here, we
still have,
you can still see the outline of the car here, albeit a little bit more pixelated,
but just by taking the max of a certain region, we've got potentially the most
important feature
of that little section. And now, of course, you could customize this value here. If
when we
create our max pool layer, you could increase the kernel size to four by four. What
do you think
```

will happen if we can increase it to four? So here, we've got a two by two kernel. If we increase it
to four by four, what happens? Ah, do you notice that we've gone from 62 to 15, we've essentially
divided our feature space by four, we've compressed it even further. Now, will that work? Well,
I'm not sure. That's part of the experimental nature of machine learning, but we're going to
keep it at two for now. And so this is with our tensor here 6464. But now let's do the same as
what we've done above, but we'll do it with a smaller tensor so that we can really visualize
things. And we're going to just replicate the same operation that's going on here. So let's go here,
we'll create another random tensor. We'll set up the manual seed first. And we're going to create
a random tensor with a similar number of dimensions. Now, recall dimensions don't tell you, so this
is a dimension 1364 64. That is a dimension. The dimensions can have different values within
themselves. So we want to create a four dimensional tensor to our images. So what that means is,
let me just show you it's way easy to explain things when we've got code is torch dot rand n.
And we're going to set it up as size equals one, one, two, two. We can have a look at this random
tensor. It's got four dimensions. One, two, three, four. So you could have a batch size,
color channels, and height width, a very small image, but it's a random image here. But this is
quite similar to what we've got going on here, right? Four numbers. Now, what do you think will
happen if we create a max pool layer, just like we've done above, create a max pool layer. So we
go max pool layer, just repeating the code that we have in the cell above, that's all right,
a little bit of practice. Kernel size equals two. And then we're going to pass the random tensor
through the max pool layer. So we'll go max pool tensor equals max pool layer. And we're going
to pass it in the random tensor. Wonderful. And then we can print out some shapes and print
out some tenses. As we always do to visualize, visualize, visualize. So we're going to write in
here max pool tensor on a new line. We'll get in the max pool tensor. We'll see what this looks
like. And we'll also print out max pool tensor shape. And we can probably print out random tensor
itself, as well as its shape as well. We'll get the shape here, dot shape. And we'll do the same
for the random tensor. So print, get a new line, random tensor, new line, random tensor. And then
we'll get the shape. Random tensor shape, random tensor. Oh, a lot of coding here. That's, that's
the fun part about machine learning, right? You get to write lots of code. Okay. So we're
visualizing what's going on with our random tensor. This is what's happening within

```
the max pool layer.
We've seen this from a few different angles now. So we have a random tensor of
numbers,
and we've got a size here. But the max pool tensor, once we pass our random tensor,
through the max pool layer, what happens? Well, we have 0.3367, 1288, 2345, 2303.
Now,
what's the max of all these? Well, it takes the max here is 3367. Oh, and we've got
the random
tensor down there. We don't want that. And see how we've reduced the shape from two
by two to one
by one. Now, what's going on here? Just for one last time to reiterate, the
convolutional layer
is trying to learn the most important features within an image. So if we jump into
here,
now, what are they? Well, we don't decide what a convolutional layer learns. It
learns these
features on its own. So the convolutional layer learns those features. We pass them
through a
relu nonlinear activation in case our data requires nonlinear functions. And then
we pass
those learned features through a max pool layer to compress them even further. So
the convolutional
layer can compress the features into a smaller space. But the max pooling layer
really compresses
```

Section 183: Main Topics

Key Topics:

- And we've just covered we've broken it down over the last few videos and rebuilt it ourselves with a few lines of PyTorch code
- So that's just goes to show how powerful PyTorch is and how far the deep learning field has come

▶ 📄 Click to view detailed content

```
them. So that's the entire idea. One more time, we start with some input data. We
design a neural
network, in this case, a convolutional neural network, to learn a compressed
representation
of what our input data is, so that we can use this compressed representation to
later on make
predictions on images of our own. And in fact, you can try that out if you wanted
to click here
and add your own image. So I'd give that a go. That's your extension for this
video. But now we've
stepped through the max pool 2D layer and the conv 2D layer. I think it's time we
started to try
```

and use our tiny VGG network. This is your challenge is to create a dummy tensor and pass it through
this model. Pass it through its forward layer and see what happens to the shape of your dummy tensor
as it moves through conv block 1 and conv block 2. And I'll show you my trick to calculating
the in features here for this final layer, which is equivalent to this final layer here.
I'll see you in the next video.
Over the last few videos, we've been replicating the tiny VGG architecture
from the CNN explainer website. And I hope you know that this is this actually quite exciting
because years ago, this would have taken months of work. And we've just covered we've broken it
down over the last few videos and rebuilt it ourselves with a few lines of PyTorch code.
So that's just goes to show how powerful PyTorch is and how far the deep learning field has come.
But we're not finished yet. Let's just go over to our keynote. This is what we've done.
CNN explainer model. We have an input layer. We've created that. We have com2d layers.
We've created those. We have relo activation layers. We've created those.
And finally, we have pulling layers. And then we finish off with an output layer.
But now let's see what happens when we actually pass some data through this entire model.
And as I've said before, this is actually quite a common practice is you replicate a model
that you found somewhere and then test it out with your own data. So we're going to start off
by using some dummy data to make sure that our model works. And then we're going to pass through.
Oh, I've got another slide for this. By the way, here's a breakdown of torch and
N com2d. If you'd like to see it in text form, nothing here that we really haven't discussed before, but
this will be in the slides if you would like to see it. Then we have a video animation.
We've seen this before, though. And plus, I'd rather you go through the CNN explainer website
on your own and explore this different values rather than me just keep talking about it.
Here's what we're working towards doing. We have a fashion MNIST data set. And we have
our inputs. We're going to numerically encode them. We've done that already. Then we have our
convolutional neural network, which is a combination of convolutional layers, nonlinear activation
layers, pooling layers. But again, these could be comprised in many different ways, shapes and
forms. In our case, we've just replicated the tiny VGG architecture. And then finally,
we want to have an output layer to predict what class of clothing a particular input image is.
And so let's go back. We have our CNN model here. And we've got model two. So let's just practice
a dummy forward pass here. We're going to come back up a bit to where we were. We'll make sure

```
we've got model two. And we get an error here because I've times this by zero. So
I'm going to
just remove that and keep it there. Let's see what happens if we create a dummy
tensor and pass it
through here. Now, if you recall what our image is, do we have image? This is a
fashion MNIST
image. So I wonder if we can go plot dot M not M show image. And I'm going to
squeeze that.
And I'm going to set the C map equal to gray. So this is our current image.
Wonderful.
So there's our current image. So let's create a tensor. Or maybe we just try to
pass this through
the model and see what happens. How about we try that model image? All right, we're
going to try
the first pass forward pass. So pass image through model. What's going to happen?
Well, we get an
error. Another shape mismatch. We've seen this before. How do we deal with this?
Because what
is the shape of our current image? 128, 28. Now, if you don't have this image
instantiated,
you might have to go back up a few cells. Where did we create image? I'll just find
this. So
just we created this a fairly long time ago. So I'm going to probably recreate it
down the
bottom. My goodness, we've written a lot of code. Well, don't do us. We could
create a dummy tensor
if we wanted to. How about we do that? And then if you want to find, oh, right back
up here,
we have an image. How about we do that? We can just do it with a dummy tensor.
That's fine.
We can create one of the same size. But if you have image instantiated, you can try
that out.
So there's an image. Let's now create an image that is, or a random tensor, that is
the same
shape as our image. So rand image tensor equals what torch dot rand n. And we're
going to pass in
size equals 128, 28. Then if we get rand image tensor,
we check its shape. What do we get? So the same shape as our test image here,
```

Section 184: Main Topics

Key Topics:

- Again, we're going through all the three major issues in deep learning

▶ 📄 Click to view detailed content

```
but it's just going to be random numbers. But that's okay. We just want to
highlight a point
here of input and output shapes. We want to make sure our model works. Can our
```

```
random image tensor
go all the way through our model? That's what we want to find out. So we get an
error here.
We have four dimensions, but our image is three dimensions. How do we add an extra
dimension
for batch size? Now you might not get this error if you're running a later version
of pie torch.
Just keep that in mind. So unsqueeze zero. Oh, expected all tensors to be on the
same device,
but found at least two devices. Again, we're going through all the three major
issues in deep
learning. Shape mismatch, device mismatch, data type mismatch. So let's put this on
the device,
two target device, because we've set up device agnostic code.
That one and that two shapes cannot be multiplied. Oh, but we can output here.
That is very exciting. So what I might do is move this a couple of cells up so that
we can
tell what's going on. I'm going to delete this cell. So where do these shapes come
from?
Well, we printed out the shapes there. And so this is what's happened when our,
I'll just create our random tensor. I'll bring our random tensor up a bit too.
Let's bring this up.
There we go. So we pass our random to image tensor through our model, and we've
made sure it's
got four dimensions by unsqueeze zero. And we make sure it's on the same device as
our model,
because our model has been sent to the GPU. And this is what happens as we pass our
random
image tensor. We've got 12828 instead of previously we've seen 6464.3, which is
going to clean this
up a bit. And we get different shapes here. So you'll notice that as our input, if
it was 6464.3
goes through these layers, it gets shaped into different values. Now this is going
to be universal
across all of the different data sets you work on, you will be working with
different shapes.
So it's important to, and also quite fun, to troubleshoot what shapes you need to
use for
your different layers. So this is where my trick comes in. To find out the shapes
for different
layers, I often construct my models, how we've done here, as best I can with the
information
that I've got, such as replicating what's here. But I don't really know what the
output
shape is going to be before it goes into this final layer. And so I recreate the
model as best
I can. And then I pass data through it in the form of a dummy tensor in the same
shape as my
actual data. So we could customize this to be any shape that we wanted. And then I
print the
shapes of what's happening through each of the forward past steps. And so if we
pass it through
this random tensor through the first column block, it goes through these layers
here. And then it
outputs a tensor with this size. So we've got 10, because that's how many output
channels we've
set. And then 14, 14, because our 2828 tensor has gone through a max pool 2d layer
```

```
and gone through
a convolutional layer. And then it goes through the next block, column block two,
which is because
we've put it in the forward method here. And then it outputs the shape. And if we
go back down,
we have now a shape of one 10, seven, seven. So our previous tensor, the output of
column block one,
has gone from 1414 to seven seven. So it's been compressed. So let me just write
this down here,
output shape of column block one, just so we get a little bit more information.
And I'm just going to copy this, put it in here, that will become block two.
And then finally, I want to know if I get an output shape of classifier.
So if I rerun all of this, I don't get an output shape of classifier. So my model
is running into
trouble. Once it gets to, so I get the output of conv block one, I don't get an
output of classifier.
So this is telling me that I have an issue with my classifier layer. Now I know
this, but I'm
not. Now I know this because, well, I've coded this model before, and the in
features here,
we need a special calculation. So what is going on with our shapes?
Mat one and mat two shapes cannot be multiplied. So do you see here, what is the
rule of matrix
multiplication? The inner dimensions here have to match. We've got 490. Where could
that number
have come from? And we've got 10 times 10. Now, okay, I know I've set hidden units
to 10.
So maybe that's where that 10 came from. And what is the output layer of the output
shape of conv
block two? So if we look, we've got the output shape of conv block two. Where does
that go?
The output of conv block two goes into our classifier model. And then it gets
flattened.
So that's telling us something there. And then our NN linear layer is expecting the
output of
the flatten layer as it's in features. So this is where my trick comes into play. I
pass the
output of conv block two into the classifier layer. It gets flattened. And then
that's what
my NN not linear layer is expecting. So what happens if we flatten this shape here?
Do we get
this value? Let's have a look. So if we go 10 times seven times seven, 490. Now,
where was this 10?
Well, that's our hidden units. And where were these sevens? Well, these sevens are
the output
```

Section 185: Main Topics

Key Topics:

- And if we look where we are at the PyTorch workflow, we've got our data ready

- And we'll use a learning rate of 0

▶ 📄 Click to view detailed content

```
of conv block two. So that's my trick. I print the shapes of previous layers and
see whether or
not they line up with subsequent layers. So if we go time seven times seven, we're
going to have
hidden units equals 10 times seven times seven. Where do we get the two sevens?
Because that is
the output shape of conv block two. Do you see how this can be a little bit hard to
calculate ahead
of time? Now, you could calculate this by hand if you went into n conv 2d. But I
prefer to write
code to calculate things for me. You can calculate that value by hand. If you go
through,
H out W out, you can add together all of the different parameters and multiply them
and divide
them and whatnot. You can calculate the input and output shapes of your
convolutional layers.
You're more than welcome to try that out by hand. But I prefer to code it out. If
and out code it
out. Now, let's see what happens if we run our random image tensor through our
model. Now,
do you think it will work? Well, let's find out. All we've done is we've added this
little line
here, times seven times seven. And we've calculated that because we've gone, huh,
what if we pass a
tensor of this dimension through a flattened layer? And what is our rule of matrix
multiplication?
The inner dimensions here must match. And why do we know that these are matrices?
Well,
mat one and mat two shapes cannot be multiplied. And we know that inside a linear
layer
is a matrix multiplication. So let's now give this a go. We'll see if it works.
Oh, ho ho. Would you look at that? That is so exciting. We have the output shape of
the classifier
is one and 10. We have a look, we have one number one, two, three, four, five, six,
seven, eight,
nine, 10, one number for each class in our data set. Wow. Just like the CNN explain
a website,
we have 10 outputs here. We just happen to have 10 classes as well. Now, this
number again could be
whatever you want. It could be 100, could be 30, could be three, depending on how
many classes
you have. But we have just figured out the input and output shapes of each layer in
our model.
So that's very exciting. I think it's now time we've passed a random tensor
through. How about we
pass some actual data through our model? In the next video, let's use our train and
test step
functions to train our first convolutional neural network. I'll see you there.
Well, let's get ready to train our first CNN. So what do we need? Where are we up
to in the
workflow? Well, we've built a model and we've stepped through it. We know what's
```

going on,
but let's really see what's going on by training this CNN or see if it trains because we don't
always know if it will on our own data set, which is of fashion MNIST. So we're going to set up a
loss function and optimizer for model two. And just as we've done before, model two, turn that
into markdown. I'll just show you the workflow again. So this is what we're doing. We've got some
inputs. We've got a numerical encoding. We've built this architecture and hopefully it helps us
learn or it helps us make a predictive model that we can input images such as grayscale images of
clothing and predict. And if we look where we are at the PyTorch workflow, we've got our data ready.
We've built our next model. Now here's where we're up to picking a loss function and an optimizer.
So let's do that, hey, loss function, or we can do evaluation metrics as well. So set up loss
function slash eval metrics slash optimizer. And we want from helper functions, import accuracy
function, we don't need to reimport it, but we're going to do it anyway for completeness. Loss
function equals nn dot cross entropy loss, because we are working with a multi class classification
problem. And the optimizer, we're going to keep the same as what we've used before, torch dot
opt in SGD. And we'll pass it in this time, the params that we're trying to optimize are the
parameters of model two parameters. And we'll use a learning rate of 0.1. Run that. And just
to reiterate, here's what we're trying to optimize model two state dig. We have a lot of random
weights in model two. Have a look at all this. There's the bias, there's the weight. We're going
to try and optimize these to help us predict on our fashion MNIST data set. So without any further
ado, let's in the next video, go to the workflow, we're going to build our training loop. But thanks
to us before, we've now got functions to do this for us. So if you want to give this a go,
use our train step and test step function to train model two. Try that out. And we'll do it
together in the next video. We're getting so close to training our model. Let's write some code to
train our first thing in that model. Training and testing, I'm just going to make another heading
here. Model two, using our training and test functions. So we don't have to rewrite all of the
steps in a training loop and a testing loop, because we've already created that functionality
before through our train step function. There we go. Performs the training, or this should be
performs a training step with model trying to learn on data loader. So let's set this up.
We're going to set up torch manual seed 42, and we can set up a CUDA manual seed as well.

Section 186: Main Topics

Key Topics:

- And then of course, the device is going to be equal to the device
- Oh, of course
- So you can see here all the functions that are being called behind the scenes from PyTorch

▶ 📄 Click to view detailed content

```
Just to try and make our experiments as reproducible as possible, because we're
going to be using
CUDA, we're going to measure the time because we want to compare our models, not
only their
performance in evaluation metrics, but how long they take to train from time it,
because there's
no point having a model that performs really, really well, but takes 10 times
longer to train.
Well, maybe there is, depending on what you're working on. Model two equals timer,
and we're going to train and test model, but the time is just something to be aware
of,
is that usually a better performing model will take longer to train. Not always the
case, but
just something to keep in mind. So for epoch in, we're going to use TQDM to measure
the progress.
We're going to create a range of epochs. We're just going to train for three
epochs,
keeping our experiment short for now, just to see how they work, epoch, and we're
going to
print a new line here. So for an epoch in a range, we're going to do the training
step,
which is our train step function. The model is going to be equal to model two,
which is our
convolutional neural network, our tiny VGG. The data loader is just going to be
equal to the
train data loader, the same one we've used before. The loss function is going to be
equal to the
loss function that we've set up above, loss FN. The optimizer as well is going to
be
the optimizer in our case, stochastic gradient descent, optimizer equals optimizer,
then we set up the accuracy function, which is going to be equal to our accuracy
function,
and the device is going to be the target device. How easy was that? Now we do the
same for the
train or the testing step, sorry, the model is going to be equal to model two, and
then the data
loader is going to be the test data loader, and then the loss function is going to
```

be our same
our same loss function. And then we have no optimizer for this, we're just going to pass in the
accuracy function here. And then of course, the device is going to be equal to the device.
And then what do we do now? Well, we can measure the end time so that we know how long the code
here took to run. So let's go train time end for model two. This will be on the GPU, by the way,
but this time it's using a convolutional neural network. And the total train time,
total train time for model two is going to be equal to print train time, our function that we
created before as well, to help us measure start and end time. So we're going to pass in train
to time start model two, and then end is going to be train time end model two. And then we're going
to print out the device that it's using as well. So you're ready? Are you ready to train our first
convolutional neural network? Hopefully this code works. We've created these functions before,
so it should be all right. But if and out, code it out, if and out, run the code, let's see what
happens. Oh my goodness. Oh, of course. Oh, we forgot to comment out the output shapes.
So we get a whole bunch of outputs for our model, because what have we done? Back up here,
we forgot to. So this means every time our data goes through the forward pass, it's going to
be printing out the output shapes. So let's just comment out these. And I think this cell is going
to take quite a long time to run because it's got so many printouts. Yeah, see, streaming output
truncated to the last 5,000 lines. So we're going to try and stop that. Okay, there we go.
Beautiful. That actually worked. Sometimes it doesn't stop so quickly. So we're going to rerun
our fashion MSV to model cell so that we comment out these print lines. And then we'll just rerun
these cells down here. Just go back through fingers crossed, there's no errors. And we'll train our
model again. Beautiful. Not as many printouts this time. So here we go. Our first CNN is training.
How do you think it'll go? Well, that's what we have printouts, right? So we can see the progress.
So you can see here all the functions that are being called behind the scenes from PyTorch. So
thank you to PyTorch for that. There's our, oh, our train step function was in there.
Train step. Wonderful. Beautiful. So there's epoch zero. Oh, we get a pretty good test accuracy.
How good is that? Test accuracy is climbing as well. Have we beaten our baseline? We're looking at
about 14 seconds per epoch here. And then the final epoch. What do we finish at? Oh, 88.5. Wow.
In 41.979 or 42 there about seconds. Again, your mileage may vary. Don't worry too much if these
numbers aren't exactly the same on your screen and same with the training time

```
because we might
be using slightly different hardware. What GPU do I have today? I have a Tesla P100
GPU. You might
not have the same GPU. So the training time, if this training time is something
like 10 times
higher, you might want to look into what's going on. And if these values are like
10% lower or 10%
higher, you might want to see what's going on with your code as well. But let's now
calculate
our Model 2 results. I think it is the best performing model that we have so far.
Let's get
a results dictionary. Model 2 results is so exciting. We're learning the power of
convolutional neural
networks. Model 2 results equals a vowel model. And this is a function that we've
created before.
```

Section 187: Main Topics

Key Topics:

- And then, oops, excuse me, typo, our loss function will be, of course, our loss function
- But let's make sure by comparing, this is another important part of machine learning experiments is comparing the results across your experiments
- We could of course change the kernel size, change the padding, change the max pool

▶ 📄 Click to view detailed content

```
So returns a dictionary containing the results of a model predicting on data
loader.
So now let's pass in the model, which will be our trained model to, and then we'll
pass in the
data loader, which will be our test data loader. And then, oops, excuse me, typo,
our loss function
will be, of course, our loss function. And the accuracy function will be accuracy
function.
And the device is already set, but we can reset anyway, device equals device. And
we'll check
out the Model 2 results. Make some predictions. Oh, look at that. Model accuracy
88. Does that
beat our baseline? Model 0 results. Oh, we did beat our baseline with a
convolutional neural network.
All right. So I feel like that's, uh, that's quite exciting. But now let's keep
going on. And, uh,
let's start to compare the results of all of our models. I'll see you in the next
video.
```

Welcome back. Now, in the last video, we trained our first convolutional neural network. And
from the looks of things, it's improved upon our baseline. But let's make sure by comparing,
this is another important part of machine learning experiments is comparing the results
across your experiments. So and training time. Now, we've done that in a way where we've got
three dictionaries here of our model zero results, model one results, model two results. So how
about we create a data frame comparing them? So let's import pandas as PD. And we're going to
compare results equals PD dot data frame. And because our model results dictionaries, uh,
all have the same keys. Let's pass them in as a list. So model zero results, model one results,
and model two results to compare them. Wonderful. And what it looks like when we compare the results.
All righty. So recall our first model was our baseline V zero was just two linear layers.
And so we have an accuracy of 83.4 and a loss of 0.47. The next model was we trained on the GPU
and we introduced nonlinearities. So we actually found that that was worse off than our baseline.
But then we brought in the big guns. We brought in the tiny VGG architecture from the CNN explainer
website and trained our first convolutional neural network. And we got the best results so far.
But there's a lot more experiments that we could do. We could go back through our
tiny VGG and we could increase the number of hidden units. Where do we create our model up here?
We could increase this to say 30 and see what happens. That would be a good experiment to
try. And if we found that nonlinearities didn't help with our second model, we could comment out
the relu layers. We could of course change the kernel size, change the padding, change the max
pool. A whole bunch of different things that we could try here. We could train it for longer.
So maybe if we train it for 10 epochs, it would perform better. But these are just things to
keep in mind and try out. I'd encourage you to give them a go yourself. But for now, we've kept
all our experiments quite the same. How about we see the results we add in the training time?
Because that's another important thing that we've been tracking as well. So we'll add
training time to results comparison. So the reason why we do this is because
if this model is performing quite well, even compared to our CNN, so a difference in about
5% accuracy, maybe that's tolerable in the space that we're working, except that this model
might actually train and perform inference 10 times faster than this model. So that's just
something to be aware of. It's called the performance speed trade off. So let's add another column
here, compare results. And we're going to add in, oh, excuse me, got a little error

```
there. That's
all right. Got trigger happy on the shift and enter. Training time equals, we're
going to add in,
we've got another list here is going to be total train time for model zero, and
total train time
for model one, and total train time for model two. And then we have a look at our
how compare results dictionary, or sorry, compare results data frame. Wonderful. So
we see, and
now this is another thing. I keep stressing this to keep in mind. If your numbers
aren't exactly
of what I've got here, don't worry too much. Go back through the code and see if
you've set up
the random seeds correctly, you might need a koodle random seed. We may have missed
one of those.
If your numbers are out landishly different to these numbers, then you should go
back through
your code and see if there's something wrong. And again, the training time will be
highly
dependent on the compute environment you're using. So if you're running this
notebook locally,
you might get faster training times. If you're running it on a different GPU to
what I have,
NVIDIA SMI, you might get different training times. So I'm using a Tesla P100,
which is quite a fast
GPU. But that's because I'm paying for Colab Pro, which generally gives you faster
GPUs.
And model zero was trained on the CPU. So depending on what compute resource Google
allocates to you
with Google Colab, this number might vary here. So just keep that in mind. These
values training
time will be very dependent on the hardware you're using. But if your numbers are
dramatically
different, well, then you might want to change something in your code and see
what's going on.
And how about we finish this off with a graph? So let's go visualize our model
results. And while
we're doing this, have a look at the data frame above. Is the performance here 10
seconds longer
```

Section 188: Main Topics

Key Topics:

- And the whole point of making a machine learning model on computer vision data is to be able to visualize predictions
- This is one of my favorite steps after training a machine learning model

▶ 📄 Click to view detailed content

training time worth that extra 5% of the results on the accuracy? Now in our case, we're using a
relatively toy problem. What I mean by toy problem is quite a simple data set to try and test this
out. But in your practice, that may be worth doing. If your model takes longer to train,
but gets quite a bit better performance, it really depends on the problem you're working with.
Compare results. And we're going to set the index as the model name, because I think that's
what we want our graph to be, not the model name. And then we're going to plot, we want to compare
the model accuracy. And we want to plot, the kind is going to be equal to bar h, horizontal bar chart.
We've got p x label, we're going to get accuracy as a percentage. And then we're going to go py label.
This is just something that you could share. If someone was asking, how did your modeling
experiments go on fashion MNIST? Well, here's what I've got. And then they ask you, well,
what's the fashion MNIST model V2? Well, you could say that's a convolutional neural network that
trained, that's replicates the CNN explainer website that trained on a GPU. How long did that
take to train? Well, then you've got the training time here. We could just do it as a vertical bar
chart. I did it as horizontal so that this looks a bit funny to me. So horizontal like that.
So the model names are over here. Wonderful. So now I feel like we've got a trained model.
How about we make some visual predictions? Because we've just got numbers on a page here,
but our model is trained on computer vision data. And the whole point of making a machine
learning model on computer vision data is to be able to visualize predictions. So let's give
that a shot, hey, in the next video, we're going to use our best performing model, fashion MNIST
model V2 to make predictions on random samples from the test data set. You might want to give
that a shot, make some predictions on random samples from the test data set, and plot them out with
their predictions as the title. So try that out. Otherwise, we'll do it together in the next video.
In the last video, we compared our models results. We tried three experiments. One was a basic linear
model. One was a linear model with nonlinear activations. And fashion MNIST model V2 is a
convolutional neural network. And we saw that from an accuracy perspective, our convolutional neural
network performed the best. However, it had the longest training time. And I just want to exemplify
the fact that the training time will vary depending on the hardware that you run on. We spoke about
this in the last video. However, I took a break after finishing the last video, reran all of the

```
cells that we've written, all of the code cells up here by coming back to the
notebook and going
run all. And as you'll see, if you compare the training times here to the last
video, we get
some different values. Now, I'm not sure exactly what hardware Google collab is
using behind the
scenes. But this is just something to keep in mind, at least from now on, we know
how to track
our different variables, such as how long our model takes to train and what its
performance
values are. But it's time to get visual. So let's create another heading, make and
evaluate. This
is one of my favorite steps after training a machine learning model. So make and
evaluate random
predictions with the best model. So we're going to follow the data explorer's model
of getting
visual visual visual or visualize visualize visualize. Let's make a function called
make
predictions. And it's going to take a model, which will be a torch and end module
type.
It's also going to take some data, which can be a list. It'll also take a device
type,
which will be torch dot device. And we'll set that by default to equal the default
device that
we've already set up. And so what we're going to do is create an empty list for
prediction
probabilities. Because what we'd like to do is just take random samples from the
test data set,
make predictions on them using our model, and then plot those predictions. We want
to visualize
them. And so we'll also turn our model into evaluation mode, because if you're
making predictions with
your model, you should turn on evaluation mode. We'll also switch on the inference
mode context
manager, because predictions is another word for inference. And we're going to loop
through
for each sample in data. Let's prepare the sample. So this is going to take in
a single image. So we will unsqueeze it, because we need to add a batch size
dimension
on the sample, we'll set dim equals to zero, and then we'll pass that to the
device. So
add a batch dimension, that's with the unsqueeze, and pass to target device. That
way, our data and
model are on the same device. And we can do a forward pass. Well, we could actually
up here go
model dot two device. That way we know that we've got device agnostic code there.
Now let's do the forward pass, forward pass model outputs raw logits. So recall
that if we have a
linear layer at the end of our model, it outputs raw logits. So pred logit for a
single sample is
going to equal model. We pass the sample to our target model. And then we're going
to get the
prediction probability. How do we get the prediction probability? So we want to go
from
logit to prediction probability. Well, if we're working with a multi class
classification problem,
```

Section 189: Main Topics

Key Topics:

- It's a five, which is, of course, class names will index on that

▶ 📄 Click to view detailed content

```
we're going to use the softmax activation function on our pred logit. And we're
going to squeeze
it so it gets rid of an extra dimension. And we're going to pass in dim equals
zero. So that's going
to give us our prediction probability for a given sample. Now let's also turn our
prediction
probabilities into prediction labels. So get pred. Well, actually, I think we're
just going
to return the pred probes. Yeah, let's see what that looks like, because we've got
a
an empty list up here for pred probes. So for matplotlib, we're going to have to
use our data
on the CPU. So let's make sure it's on the CPU, because matplotlib doesn't work
with the GPU.
So get pred prob off GPU for further calculations. So we're just hard coded in here
to make sure
that our prediction probabilities off the GPU. So pred probs, which is our list up
here. We're
going to append the pred prob that we just calculated. But we're going to put it on
the CPU. And then
let's go down here. And we're going to. So if we've done it right, we're going to
have a list of
prediction probabilities relating to particular samples. So we're going to stack
the pred probs
to turn list into a tensor. So this is only one way of doing things. There are many
different ways
that you could make predictions and visualize them. I'm just exemplifying one way.
So we're
going to torch stack, which is just going to say, hey, concatenate everything in
the list to a
single tensor. So we might need to tab that over, tab, tab. Beautiful. So let's try
this function
in action and see what happens. I'm going to import random. And then I'm going to
set the random
seed to 42. And then I'm going to create test samples as an empty list, because we
want an empty
or we want a list of test samples to iterate through. And I'm going to create test
labels also as an
empty list. So that remember, when we are evaluating predictions, we want to
compare them to the
ground truth. So we want to get some test samples. And then we want to get their
actual labels so
that when our model makes predictions, we can compare them to their actual labels.
```

So for sample,
comma label, in, we're going to use random to sample the test data. Now note that this is not
the test data loader. This is just test data. And we're going to set k equals to nine. And recall,
if you want to have a look at test data, what do we do here? We can just go test data,
which is our data set, not converted into a data loader yet. And then if we wanted to get the first
10 samples, can we do that? Only one element tensors can be converted into Python scalars. So if we
get the first zero, and maybe we can go up to 10. Yeah, there we go. And what's the shape of this?
Tuple has no object shape. Okay, so we need to go image label equals that. And then can we check
the shape of the image label? Oh, because the labels are going to be integers.
Wonderful. So that's not the first 10 samples, but that's just what we get if we iterate through
the test data, we get an image tensor, and we get an associated label. So that's what we're doing
with this line here, we're just randomly sampling nine samples. And this could be any number you
want. I'm going to use nine, because this is a spoiler for later on, we're going to create a
three by three plot. So that just nine is just a fun number. So get some random samples from the
test data set. And then we can go test samples dot append sample. And we will go test labels dot
append label. And then let's go down here, view the first, maybe we go first sample shape.
So test samples zero dot shape.
And then if we get test samples, zero, we're going to get a tensor of image values. And then
if we wanted to plot that, can we go PLT, M show, C map, equals gray. And we may have to squeeze
this, I believe, to remove the batch tensor. Let's see what happens batch dimension. There we go.
Beautiful. So that's to me, a shoe, a high heel shoe of some sort. If we get the title,
PLT dot title, test labels, let's see what this looks like. It's a five, which is, of course,
class names will index on that. Sandal. Okay, beautiful. So we have nine random samples,
nine labels that are associated with that sample. Now let's make some predictions.
So make predictions.
And this is one of my favorite things to do. I can't stress it enough is to randomly pick data
samples from the test data set and predict on them and do it over and over again to see
what the model is doing. So not only at the start of a problem, I'll just get the prediction
probabilities here. We're going to call our make predictions function. So not only at the start of
a problem should you become one with the data, even after you've trained a model, you'll want to
further become one with the data, but this time become one with your models predictions on the

```
data and see what happens. So view the first two prediction probabilities list. So
we're just
using our make predictions function that we created before, passing at the model,
the train model
to, and we're passing at the data, which is the test samples, which is this list
that we just
created up here, which is comprised of random samples from the test data set.
Wonderful. So
```

Section 190: Main Topics

Key Topics:

- Because we've got nine random samples, you could, of course, change this to however many you want

▶ 📄 Click to view detailed content

```
let's go. Pred probes. Oh, we don't want to view them all. That's going to give us
Oh, we want to the prediction probabilities for a given sample. And so how do we
convert
prediction probabilities into labels? Because if we're trying to, if we have a look
at test
labels, if we're trying to compare apples to apples, when we're evaluating our
model, we want to,
we can't really necessarily compare the prediction probabilities straight to the
test labels. So we
need to convert these prediction probabilities into prediction labels. So how can
we do that?
Well, we can use argmax to take whichever value here, the index, in this case, this
one,
the index of whichever value is the highest of these prediction probabilities. So
let's see that
in action. Convert prediction probabilities to labels. So we'll go pred classes
equals
pred probes, and we'll get the argmax across the first dimension. And now let's
have a look at the
pred classes. Wonderful. So are they in the same format as our test labels? Yes,
they are. So if
you'd like to go ahead, in the next video, we're going to plot these and compare
them. So we're
going to write some code to create a mapplotlib plotting function that's going to
plot nine
different samples, along with their original labels, and their predicted label. So
give that a shot,
we've just written some code here to make some predictions on random samples. If
you'd like them
to be truly random, you can comment out the seed here, but I've just kept the seed
at 42. So that
```

our random dot sample selects the same samples on your end and on my end. So in the next video,
let's plot these. Let's now continue following the data explorer's motto of visualize visualize
visualize. We have some prediction classes. We have some labels we'd like to compare them to.
You can compare them visually. It looks like our model is doing pretty good. But let's,
since we're making predictions on images, let's plot those images along with the predictions.
So I'm going to write some code here to plot the predictions. I'm going to create a matplotlib
figure. I'm going to set the fig size to nine and nine. Because we've got nine random samples,
you could, of course, change this to however many you want. I just found that a three by three
plot works pretty good in practice. And I'm going to set n rows. So for my matplotlib plot, I want
three rows. And I want three columns. And so I'm going to enumerate through the samples in test
samples. And then I'm going to create a subplot for each sample. So create a subplot. Because this
is going to create a subplot because it's within the loop. Each time it goes through a new sample,
create a subplot of n rows and calls. And the index it's going to be on is going to be i plus
one, because it can't start at zero. So we just put i plus one in there. What's going on here?
Enumerate. Oh, excuse me. In enumerate, wonderful. So now we're going to plot the target image.
We can go plot dot in show, we're going to get sample dot squeeze. Because we need to remove the
batch dimension. And then we're going to set the C map is equal to gray. What's this telling me
up here? Oh, no, that's correct. Next, we're going to find the prediction label in text form,
because we don't want it in a numeric form, we could do that. But we want to look at things
visually with human readable language, such as sandal for whatever class sandal is, whatever number
class that is. So we're going to set the pred label equals class names. And we're going to index
using the pred classes I value. So right now we're going to plot our sample. We're going to find
its prediction. And now we're going to get the truth label. So we also want this in text form.
And what is the truth label going to be? Well, the truth label is we're going to have to index
using class names and index on that using test labels I. So we're just matching up our indexes
here. Finally, we're going to create a title, create a title for the plot. And now here's what I like
to do as well. If we're getting visual, well, we might as well get really visual, right? So I
think we can change the color of the title text, depending if the prediction is right or wrong.
So I'm going to create a title using an F string, pred is going to be a pred label,

```
and truth label. We could even plot the prediction probabilities here if we wanted
to. That might
be an extension that you might want to try. And so here we're going to check for
equality
between pred and truth and change color of title text. So what I mean by this, it's
going to be a
lot easier to explain if we just if and doubt coded out. So if the pred label
equals the truth
label, so they're equal, I want the plot dot title to be the title text. But I want
the font size,
well, the font size can be the same 10. I want the color to equal green. So if
they're so green text,
if prediction, same as truth, and else I'm going to set the plot title to have
title text font
size equals 10. And the color is going to be red. So does that make sense? All
we're doing is we're
enumerating through our test samples that we got up here, test samples that we
found randomly from
the test data set. And then each time we're creating a subplot, we're plotting our
image,
```

Section 191: Main Topics

Key Topics:

- See how, for me, I much appreciate, like, I much prefer visualizing things numbers on a page look good, but there's something, there's nothing quite like visualizing your machine learning models predictions, especially when it gets it right
- We're up to a very exciting point in evaluating our machine learning model
- And if you recall, if we go back to section two of the lone pytorch

▶ 📄 Click to view detailed content

```
we're finding the prediction label by indexing on the class names with our pred
classes value,
we're getting the truth label, and we're creating a title for the plot that
compares the pred label
to the truth. And we're changing the color of the title text, depending if the pred
label is
correct or not. So let's see what happens. Did we get it right? Oh, yes, we did.
Oh, I'm going to
do one more thing. I want to turn off the accesses, just so we get more real
estate. I love these
kind of plots. It helps that our model got all of these predictions right. So look
at this,
pred sandal, truth, sandal, pred trouser, truth trouser. So that's pretty darn
good, right? See how,
for me, I much appreciate, like, I much prefer visualizing things numbers on a page
```

look good,
but there's something, there's nothing quite like visualizing your machine learning models
predictions, especially when it gets it right. So how about we select some different random samples
up here, we could functionize this as well to do like all of this code in one hit, but that's all
right. We'll be a bit hacky for now. So this is just going to randomly sample with no seed at all.
So your samples might be different to mine, nine different samples. So this time we have an ankle
boot, we'll make some predictions, we'll just step through all of this code here. And oh,
there we go. It got one wrong. So all of these are correct. But this is more interesting as
well is where does your model get things wrong? So it predicted address, but this is a coat.
Now, do you think that this could be potentially address? To me, I could see that as being addressed.
So I kind of understand where the model's coming from there. Let's make some more random predictions.
We might do two more of these before we move on to the next video.
Oh, all correct. We're interested in getting some wrong here. So our model seems to be too good.
All correct again. Okay, one more time. If we don't get any wrong, we're going on to the next
video. But this is just really, oh, there we go. Too wrong. Beautiful. So predicted address,
and that's a shirt. Okay. I can kind of see where the model might have stuffed up there.
It's a little bit long for a shirt for me, but I can still understand that that would be a shirt.
And this is a pullover, but the truth is a coat. So maybe, maybe there's some issues with the labels.
And that's probably what you'll find in a lot of data sets, especially quite large ones.
Just with a sheer law of large numbers, there may be some truth labels in your data sets that
you work with that are wrong. And so that's why I like to see, compare the models predictions
versus the truth on a bunch of random samples to go, you know what, is our models results
better or worse than they actually are. And that's what visualizing helps you do is figure out,
you know what, our model is actually, it says it's good on the accuracy. But when we visualize
the predictions, it's not too good. And vice versa, right? So you can keep playing around with this,
try, look at some more random samples by running this again. We'll do one more for good luck.
And then we'll move on to the next video. We're going to go on to another way. Oh,
see, this is another example. Some labels here could be confusing. And speaking of confusing,
well, that's going to be a spoiler for the next video. But do you see how the prediction is a
t-shirt top, but the truth is a shirt? To me, that label is kind of overlapping. Like, I don't know,

```
what's the difference between a t-shirt and a shirt? So that's something that
you'll find
as you train models is maybe your model is going to tell you about your data as
well.
And so we hinted that this is going to be confused. The model is confused between
t-shirt top and
shirt. How about we plot a confusion matrix in the next video? I'll see you there.
We're up to a very exciting point in evaluating our machine learning model.
And that is visualizing, visualizing, visualizing. And we saw that in the previous
video, our model
kind of gets a little bit confused. And in fact, I would personally get confused at
the difference
between t-shirt slash top and a shirt. So these kind of insights into our model
predictions
can also give us insights into maybe some of our labels could be improved. And
another way to
check that is to make a confusion matrix. So let's do that, making a confusion
matrix for further
prediction evaluation. Now, a confusion matrix is another one of my favorite ways
of evaluating
a classification model, because that's what we're doing. We're doing multi class
classification.
And if you recall, if we go back to section two of the lone pytorch.io book,
and then if we scroll down, we have a section here, more classification evaluation
metrics.
So accuracy is probably the gold standard of classification evaluation.
There's precision, there's recall, there's F1 score, and there's a confusion matrix
here.
So how about we try to build one of those? I want to get this and copy this.
So, and write down a confusion matrix is a fantastic way of evaluating your
classification models
visually. Beautiful. So we're going to break this down. First of all, we need to
plot a
confusion matrix. We need to make predictions with our trained model on the test
data set.
Number two, we're going to make a confusion matrix. And to do so, we're going to
leverage
```

Section 192: Main Topics

Key Topics:

- So recall that torch metrics we've touched on this before is a great package torch metrics for a whole bunch of evaluation metrics of machine learning models in pytorch flavor
- And do you notice how this is quite similar to the pytorch documentation
- Well, that's the beautiful thing about torch metrics is that it's created with pytorch in mind

▶ 📄 Click to view detailed content

```
torch metrics tricks have to figure out how to spell metrics and confusion matrix.
So recall
that torch metrics we've touched on this before is a great package torch metrics
for a whole
bunch of evaluation metrics of machine learning models in pytorch flavor. So if we
find we've
got classification metrics, we've got audio image detection. Look how this is
beautiful,
a bunch of different evaluation metrics. And if we go down over here, we've got
confusion
matrix. So I only touched on five here, but or six. But if you look at torch
metrics, they've got,
how many is that about 25 different classification metrics? So if you want some
extra curriculum,
you can read through these. But let's go to confusion matrix. And if we look at
some code here,
we've got torch metrics, confusion matrix, we need to pass in number of classes. We
can
normalize if we want. And do you notice how this is quite similar to the pytorch
documentation?
Well, that's the beautiful thing about torch metrics is that it's created with
pytorch in mind.
So let's try out if you wanted to try it out on some
tester code, you could do it here. But since we've already got some of our own
code,
let's just bring in this. And then number three is to plot it. We've got another
helper package here,
plot the confusion matrix using ML extend. So this is another one of my favorite
helper
libraries for machine learning things. It's got a lot of functionality that you can
code up
yourself, but you often find yourself coding at a few too many times, such as
plotting a confusion
matrix. So if we look up ML extend plot confusion matrix, this is a wonderful
library. I believe it was
it was created by Sebastian Rushka, who's a machine learning researcher and also
author of
a great book. There he is. Yeah, this is a side note machine learning with pytorch
and
scikit loan. I just got this book it just got released in the start of 2022. And
it's a great
book. So that's a little side note for learning more about machine learning with
pytorch and scikit
loan. So shout out to Sebastian Rushka. Thank you for this package as well. This is
going to
just help us plot a confusion matrix like this. So we'll have our predicted labels
on the bottom
and our true labels on the side here. But we can just copy this code in here.
Link sorry, and then confusion matrix, we can copy that in here. The thing is that
torch
metrics doesn't come with Google Colab. So if you're using Google Colab, I think ML
extend does,
but we need a certain version of ML extend that Google Colab doesn't yet have yet.
So we actually
```

need version 0.19.0. But we're going to import those in a second. Let's first make some predictions
across our entire test data set. So previously, we made some predictions only on nine random samples.
So random sample, we selected nine. You could, of course, change this number to make it on more.
But this was only on nine samples. Let's write some code to make predictions across our entire
test data set. So import tqdm.auto for progress bar tracking.
So tqdm.auto. We don't need to re-import it. I believe we've already got it above, but I'm just
going to do it anyway for completeness. And so we're going to make, this is step one, above,
make predictions, make predictions with trained model. Our trained model is model two. So let's
create an empty predictions list. So we can add our predictions to that. We're going to set our
model into evaluation mode. And we're going to set with torch inference mode as our context manager.
And then inside that, let's just build the same sort of code that we used for our testing loop,
except this time we're going to append all of our predictions to a list.
So we're going to iterate through the test data loader. And we can give our tqdm description.
We're going to say making predictions dot dot dot. You'll see what that looks like in a minute.
And here we are going to send the data and targets to target device. So x, y equals x
to device and y to device. Wonderful. And we're going to do the forward pass.
So we're going to create y logit. Remember, the raw outputs of a model with a linear layer at the
end are referred to as logits. And we don't need to calculate the loss, but we want to turn predictions
from logits to prediction probabilities to prediction labels. So we'll set here y pred equals torch
dot softmax. You could actually skip the torch softmax step if you wanted to and just take the
argmax of the logits. But we will just go from prediction probabilities to pred labels for completeness.
So squeeze and we're going to do it across the first dimension or the zeroth dimension. And then
we'll take the argmax of that across the first dimension as well. And a little tidbit. If you
take different dimensions here, you'll probably get different values. So just check the inputs
and outputs of your code to make sure you're using the right dimension here. And so let's go
put predictions on CPU for evaluation. Because if we're going to plot anything, that plot lib will
want them on the CPU. So we're going to append our predictions to y preds, y pred dot CPU.
Beautiful. And because we're going to have a list of different predictions, we can use concatenate
a list of predictions into a tensor. So let's just print out y preds. And so I can show you what
it looks like. And then if we go y pred tensor, this is going to turn our list of predictions

Section 193: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
into a single tensor. And then we'll go y pred tensor. And we'll view the first 10.
Let's see if this
works. So making predictions. Oh, would you look at that? Okay, so yeah, here's our
list of
predictions. A big list of tensors. Right, we don't really want it like that. So if
we get rid of
that, and there's our progress bar, it's going through each batch in the test data
load, so there's
313 batches of 32. So if we comment out print y preds, this line here torch dot cat
y preds is
going to turn this these tensors into a single tensor, or this list of tensors into
a single
tensor concatenate. Now, if we have a look, there we go, beautiful. And if we have
a look at the
whole thing, we're making predictions every single time here, but that's all right.
They are pretty
quick. There we go. One big long tensor. And then if we check length y pred tensor,
there should be
one prediction per test sample. 10,000 beautiful. So now we're going to, we need to
install torch
metrics because torch metrics doesn't come with Google Colab at the time of
recording. So let
me just show you if we tried to import torch metrics. It doesn't, it might in the
future, so just keep
that in mind, it might come with Google Colab because it's a pretty useful package.
But let's
now install see if required packages are installed. And if not, install them. So
we'll just install
torch metrics. We'll finish off this video by trying to import. We'll set up a try
and accept
loop. So Python is going to try import torch metrics and ML extend. I write it like
this,
because you may already have to which metrics and ML extend if you're running this
code on a local
machine. But if you're running it in Google Colab, which I'm sure many of you are,
we are
going to try and import it anyway. And if it doesn't work, we're going to install
it.
So ML extend, I'm just going to check the version here because we need version for
our plot confusion
matrix function. This one, we need version 0.19.0 or higher. So I'm just going to
write a little
statement here. Assert int ML extend dot version. So if these two, if this
condition in the try
```

loop is or try block is accepted, it will skip the next step dot split. And I'm just going to check
the first index string equals is greater than or equal to 19. Otherwise, I'm going to return an
error saying ML extend version should be 0.19.0 or higher. And so let me just show you what this
looks like. If we run this here, string and int, did I not turn it into a string? Oh, excuse me.
There we go. And I don't need that bracket on the end. There we go. So that's what I'm saying.
So this is just saying, hey, the version of ML extend that you have should be 0 or should be
19 or higher. Because right now Google Colab by default has 14, this may change in the future.
So let's finish off this accept block. If the above condition fails, which it should,
we are going to pip install. So we're going to install this into Google Colab torch metrics.
We're going to do it quietly. And we're also going to pass the U tag for update ML extend.
So import torch metrics, ML extend afterwards, after it's been installed and upgraded. And print,
we're going to go ML extend version, going to go ML extend underscore version. And let's see what
happens if we run this. So we should see, yeah, some installation happening here. This is going
to install torch metrics. Oh, do we not have ML extend the upgraded version? Let's have a look.
We may need to restart our Google Colab instance. Ah, okay, let's take this off. Quiet.
Is this going to tell us to restart Google Colab?
Well, let's restart our runtime. After you've run this cell, if you're using Google Colab,
you may have to restart your runtime to reflect the fact that we have the updated version of ML
extend. So I'm going to restart my runtime now. Otherwise, we won't be able to plot our confusion
matrix. We need 0.19.0. And I'm going to run all of these cells. So I'm going to pause the video
here, run all of the cells by clicking run all. Note, if you run into any errors, you will have
to run those cells manually. And then I'm going to get back down to this cell and make sure that I
have ML extend version 0.1.9. I'll see in a few seconds.
I'm back. And just a little heads up. If you restart your runtime and click run all,
your Colab notebook will stop running cells if it runs into an error. So this is that error we
found in a previous video where our data and model were on different devices. So to skip past that,
we can just jump to the next cell and we can click run after. There we go. And it's going to run all
of the cells after for us. It's going to retrain our models. Everything's going to get rerun.
And then we're going to come right back down to where we were before trying to install the
updated version of ML extend. I'm going to write some more code while our code is

```
running import
ML extend. And then I'm going to just make sure that we've got the right version
here. You may
require a runtime restart. You may not. So just try to see after you've run this
install of
torch metrics and upgrade of ML extend. See if you can re import ML extend. And if
you have the
```

Section 194: Main Topics

Key Topics:

- And of course, we can get the targets, which is the labels
- PyTorch calls labels targets

▶ 📄 Click to view detailed content

```
version 0.19.0 or above, we should be able to run the code. Yeah, there we go.
Wonderful.
ML extend 0.19.0. And we've got ML extend version, assert, import. Beautiful. So
we've got a lot
of extra code here. In the next video, let's move forward with creating a confusion
matrix.
I just wanted to show you how to install and upgrade some packages in Google Colab
if you
don't have them. But now we've got predictions across our entire test data set. And
we're going
to be moving towards using confusion matrix function here to compare our
predictions versus the target
data of our test data set. So I'll see in the next video, let's plot a confusion
matrix.
Welcome back. In the last video, we wrote a bunch of code to import some extra
libraries that we
need for plotting a confusion matrix. This is really helpful, by the way. Google
Colab comes
with a lot of prebuilt installed stuff. But definitely later on down the track,
you're going to need
to have some experience installing stuff. And this is just one way that you can do
it. And we also
made predictions across our entire test data set. So we've got 10,000 predictions
in this tensor.
And what we're going to do with a confusion matrix is confirm or compare these
predictions
to the target labels in our test data set. So we've done step number one. And we've
prepared
ourselves for step two and three, by installing torch metrics, and installing ML
extend or the
later version of ML extend. So now let's go through step two, making a confusion
matrix,
```

and step three plotting that confusion matrix. This is going to look so good. I love how good
confusion matrix is look. So because we've got torch metrics now, we're going to import the
confusion matrix class. And from our ML extend, we're going to go into the plotting module,
and import plot confusion matrix. Recall that the documentation for both of these are
within torch metrics here, and within ML extend here. Let's see what they look like. So number two
is set up confusion matrix instance, and compare predictions to targets. That's what evaluating a
model is, right? Comparing our models predictions to the target predictions. So I'm going to set
up a confusion matrix under the variable conf mat, then I'm going to call the confusion matrix class
from torch metrics. And to set up an instance of it, I need to pass in the number of classes that
we have. So because we have 10 classes, they are all contained within class names. Recall that
class names is a list of all of the different classes that we're working with. So I'm just going
to pass in the number of classes as the length of our class names. And then I can use that
conf mat instance, confusion matrix instance, to create a confusion matrix tensor by passing
into conf mat, which is what I've just created up here. Conf mat, just like we do with our loss
function, I'm going to pass in preds equals our Y pred tensor, which is just above Y pred tensor
that we calculated all of the predictions on the test data set. There we go. That's our preds.
And our target is going to be equal to test data dot targets. And this is our test data data set
that we've seen before. So if we go test data and press tab, we've got a bunch of different
attributes, we can get the classes. And of course, we can get the targets, which is the labels.
PyTorch calls labels targets. I usually refer to them as labels, but the target is the test data
target. So we want to compare our models predictions on the test data set to our test data targets.
And so let's keep going forward. We're up to step number three now. So this is going to create
our confusion matrix tensor. Oh, let's see what that looks like, actually. Conf mat tensor.
Oh, okay. So we've got a fair bit going on here. But let's turn this into a pretty version of this.
So along the bottom is going to be our predicted labels. And along the side here is going to be
our true labels. But this is where the power of ML extend comes in. We're going to plot our
confusion matrix. So let's create a figure and an axes. We're going to call the function plot
confusion matrix that we've just imported above. And we're going to pass in our conf mat equals
our conf mat tensor. But because we're working with map plot lib, it'll want it as

```
NumPy.
So I'm just going to write here, map plot lib likes working with NumPy. And we're
going to
pass in the class names so that we get labels for each of our rows and columns.
Class names,
this is just a list of our text based class names. And then I'm going to set the
fig size
to my favorite hand and poker, which is 10, seven. Also happens to be a good
dimension for
Google Colab. Look at that. Oh, that is something beautiful to see. Now a confusion
matrix. The
ideal confusion matrix will have all of the diagonal rows darkened with all of the
values
and no values here, no values here. Because that means that the predicted label
lines up with the
true label. So in our case, we have definitely a very dark diagonal here. But let's
dive into
some of the highest numbers here. It looks like our model is predicting shirt when
the true label
is actually t shirt slash top. So that is reflective of what we saw before. Do we
still have that
image there? Okay, we don't have an image there. But in a previous video, we saw
that when we plotted
```

Section 195: Main Topics

Key Topics:

- And of course, vice versa
- And you've got, of course, more in torch metrics
- So if you've been through the other parts of the course, you definitely have

▶ 📄 Click to view detailed content

```
our predictions, the model predicted t shirt slash top when it was actually a
shirt. And of course,
vice versa. So what's another one here? Looks like our model is predicting shirt
when it's
actually a coat. And now this is something that you can use to visually inspect
your data to see
if the the errors that your model is making make sense from a visual perspective.
So it's getting
confused by predicting pull over when the actual label is coat, predicting pull
over when the
actual label is shirt. So a lot of these things clothing wise and data wise may in
fact look
quite the same. Here's a relatively large one as well. It's predicting sneaker when
it should be
an ankle boot. So it's confusing two different types of shoes there. So this is
```

just a way to
further evaluate your model and start to go. Hmm, maybe our labels are a little bit confusing.
Could we expand them a little bit more? So keep that in mind, a confusion matrix is one of the
most powerful ways to visualize your classification model predictions. And a really, really, really
helpful way of creating one is to use torch metrics confusion matrix. And to plot it,
you can use plot confusion matrix from ML extend. However, if you're using Google Colab for these,
you may need to import them or install them. So that's a confusion matrix. If you'd like
more classification metrics, you've got them here. And you've got, of course, more in torch
metrics. So give that a look. I think in the next video, we've done a fair bit of evaluation.
Where are we up to in our workflow? I believe it's time we saved and loaded our best trained model.
So let's give that a go. I'll see you in the next video.
In the last video, we created a beautiful confusion matrix with the power of torch metrics
and ML extend. But now it's time to save and load our best model. Because if we, if we evaluated it,
our convolutional neural network and go, you know what, this model is pretty good. Let's export
it to a file so we can use it somewhere else. Let's see how we do that. And by the way, if we go into
our keynote, we've got a value at model torch metrics. We've been through this a fair few times
now. We've improved through experimentation. We haven't used tensor board yet, but that'll be
in a later video and save and reload your trained model. So here's where we're up to. If we've gone
through all these steps enough times and we're like, you know what, let's save our model so we
can use it elsewhere. And we can reload it in to make sure that it's, it's saved correctly.
Let's go through with this step. We want number 11. We're going to go save and load
best performing model. You may have already done this before. So if you've been through the other
parts of the course, you definitely have. So if you want to give that a go, pause the video now
and try it out yourself. I believe we did it in notebook number one. We have here we go,
saving and loading a pie torch model. You can go through this section of section number one
on your own and see if you can do it. Otherwise, let's code it out together. So I'm going to start
from with importing path from path lib, because I like to create a model directory path.
So create model directory path. So my model path is going to be set equal to path. And I'm going
to save it to models. This is where I want to, I want to create file over here called models
and save my models to their model path dot MKD for make directory parents. Yes, I wanted to make

```
the parent directories if they don't exist and exist. Okay. Also equals true. So if
we try to
create it, but it's already existing, we're not going to get an error. That's fine.
And next,
we're going to create a model save path. Just going to add some code cells here. So
we have
more space. Let's pass in here a model name. Going to set this equal to, since
we're on section three,
I'm going to call this O three pie torch, computer vision, model two is our best
model. And I'm going
to save it to PTH for pie torch. You can also save it to dot PT. I like to use PTH.
And we're
going to go model save path equal model path slash model name. So now if we have a
look at this,
we're going to have a path called model save path. But it's going to be a POSIX
path in models
O three pie torch computer vision, model two dot PTH. And if we have a look over
here,
we should have, yeah, we have a models directory now. That's not going to have
anything in it at
the moment. We've got our data directory that we had before there's fashion MNIST.
This is a good
way to start setting up your directories, break them down data models, helper
function files,
etc. But let's keep going. Let's save, save the model state dict. We're going to go
print,
saving model to just going to give us some information about what's happening.
Model save
path. And we can save a model by calling torch dot save. And we pass in the object
that we want
to save using the object parameter, OBJ. When we get a doc string there, we're
going to go model
two, we want to save the state dict, recall that the state dict is going to be our
models what
our models learned parameters on the data set, so that all the weights and biases
and all that
```

Section 196: Main Topics

Key Topics:

- Of course, you can customize what the name is, where you save it, et cetera, et cetera
- There's also a link to the pytorch documentation would highly recommend that
- And of course, we're going to evaluate it on the same test data set that we've been using test data loader

▶ 📄 Click to view detailed content

sort of jazz. Beautiful. So when we first created model two, these were all random numbers. They've
been or since we trained model two on our training data, these have all been updated to represent
the training images. And we can leverage these later on, as you've seen before, to make predictions.
So I'm not going to go through all those, but that's what we're saving. And the file path is
going to be our model save path. So let's run this and see what happens. Beautiful. We're saving our
model to our model directory. And now let's have a look in here. Do we have a model? Yes, we do.
Beautiful. So that's how quickly we can save a model. Of course, you can customize what the name is,
where you save it, et cetera, et cetera. Now, let's see what happens when we load it in.
So create a new instance, because we only saved the state dict of model two,
we need to create a new instance of our model two, or how it was created, which was with
our class fashion MNIST V two. If we saved the whole model, we could just import it to a new
variable. But I'll let you read back more on that on the different ways of saving a model in here.
There's also a link to the pytorch documentation would highly recommend that. But let's see it in
action, we need to create a new instance of our fashion MNIST model V two, which is our convolution
or neural network. So I'm going to set the manual seed. That way when we create a new instance,
it's instantiated with the same random numbers. So we're going to set up loaded model two,
equals fashion MNIST V two. And it's important here that we set it up with the same parameters
as our original saved model. So fashion MNIST V two. Oh, we've got a typo here.
I'll fashion MNIST model V two. Wonderful. So the input shape is going to be one,
because that is the number of color channels in our test, in our images, test image dot shape.
Do we still have a test image should be? Oh, well, we've created a different one, but our image size,
our image shape is 12828 image shape for color channels height width. Then we create it with
hidden units, we use 10 for hidden units. So we can just set that here. This is important,
they just have to otherwise if the shapes aren't the same, what are we going to get? We're going
to get a shape mismatch error. And our output shape is what is also going to be 10 or
length of class names. If you have the class names variable instantiated, that is. So we're
going to load in the saved state dict, the one that we just saved. So we can go loaded model two,
dot load state dict. And we can pass in torch dot load in here. And the file that we want to load
or the file path is model save path up here. This is why I like to just save my path variables
to a variable so that I can just use them later on, instead of re typing out this

all the time,
which is definitely prone to errors. So we're going to send the model to the target device.
Loaded model two dot two device. Beautiful. Let's see what happens here.
Wonderful. So let's now evaluate the loaded model. So evaluate loaded model. The results
should be very much the same as our model two results. So model two results.
So this is what we're looking for. We want to make sure that our saved model saved these results
pretty closely. Now I say pretty closely because you might find some discrepancies in this lower
these lower decimals here, just because of the way files get saved and something gets lost,
et cetera, et cetera. So that's just to do with precision and computing. But as long as the first
few numbers are quite similar, well, then we're all gravy. So let's go torch manual seed.
Remember, evaluating a model is almost as well is just as important as training a model. So this
is what we're doing. We're making sure our model save correctly. Before we deployed it, if it didn't
if we deployed it, it didn't save correctly. Well, then we'd get our we would get less than ideal
results, wouldn't we? So model equals loaded model two, we're going to use our same
of our model function, by the way. And of course, we're going to evaluate it on the same test data
set that we've been using test data loader. And we're going to create a loss function or just
put in our loss function that we've created before. And our accuracy function is the accuracy
function we've been using throughout this notebook. So now let's check out loaded model two results.
They should be quite similar to this one. We're going to make some predictions. And then if we go
down, do we have the same numbers? Yes, we do. So we have five, six, eight, two, nine, five, six,
eight, two, nine, wonderful. And three, one, three, five, eight, three, one, three, five, eight,
beautiful. It looks like our loaded model gets the same results as our previously trained model
before we even saved it. And if you wanted to check if they were close, you can also use torch
dot is close, check if model results, if you wanted to check if they were close programmatically,
that is, because we just looked at these visually, check if model results are close to each other.
Now we can go torch is close, we're going to pass in torch dot tensor, we have to turn these

---

Section 197: Main Topics

Key Topics:

- Let's in the next video, I think that's enough code for this section, section three, pytorch computer vision
- Look how much computer vision pytorch code we've written together
- We checked out computer vision libraries and pytorch, the main one being torch vision

▶ 📄 Click to view detailed content

```
values into a tensor. We're going to go model two results. And we'll compare the
model loss.
How about we do that? We want to make sure the loss values are the same. Or very
close,
that is with torch dot is close. Torch dot tensor model. Or we want this one to be
loaded model two
results. Model loss. Another bracket on the end there. And we'll see how close they
are true,
wonderful. Now, if this doesn't return true, you can also adjust the tolerance
levels in here.
So we go atal equals, this is going to be the absolute tolerance. So if we do one
to the negative
eight, it's saying like, Hey, we need to make sure our results are basically the
same up to eight
decimal points. That's probably quite low. I would say just make sure they're at
least within two.
But if you're getting discrepancies here between your saved model and your loaded
model, or sorry,
this model here, the original one and your loaded model, if they are quite large,
so they're like
more than a few decimal points off in this column or even here, I'd go back through
your code and
make sure that your model is saving correctly, make sure you've got random seeds
set up. But
if they're pretty close, like in terms of within three or two decimal places of
each other,
well, then I'd say that's that's close enough. But you can also adjust the
tolerance level here
to check if your model results are close enough, programmatically. Wow, we have
covered a fair bit
here. We've gone through this entire workflow for a computer vision problem. Let's
in the next
video, I think that's enough code for this section, section three, pytorch computer
vision. I've got
some exercises and some extra curriculum lined up for you. So let's have a look at
those in the
next video. I'll see you there. My goodness. Look how much computer vision pytorch
code
we've written together. We started off right up the top. We looked at the reference
notebook and
the online book. We checked out computer vision libraries and pytorch, the main one
being torch
vision. Then we got a data set, namely the fashion MNIST data set. There are a
bunch more data sets
that we could have looked at. And in fact, I'd encourage you to try some out in the
```

torch vision
dot data sets, use all of the steps that we've done here to try it on another data set. We repaired
our data loaders. So turned our data into batches. We built a baseline model, which is an important
step in machine learning, because the baseline model is usually relatively simple. And it's going
to serve as a baseline that you're going to try and improve upon through just go back to the keynote
through various experiments. We then made predictions with model zero. We evaluated it.
We timed our predictions to see if running our models on the GPU was faster when we learned that
sometimes a GPU won't necessarily speed up code if it's a relatively small data set because of the
overheads between copying data from CPU to GPU. We tried a model with non-linearity and we saw that
it didn't really improve upon our baseline model. But then we brought in the big guns, a convolutional
neural network, replicating the CNN explainer website. And by gosh, didn't we spend a lot of time
here? I'd encourage you as part of your extra curriculum to go through this again and again.
I still even come back to refer to it too. I referred to it a lot making the materials for this
video section and this code section. So be sure to go back and check out the CNN explainer website
for more of what's going on behind the scenes of your CNNs. But we coded one using pure pytorch.
That is amazing. We compared our model results across different experiments. We found that our
convolutional neural network did the best, although it took a little bit longer to train. And we also
learned that the training time values will definitely vary depending on the hardware you're using.
So that's just something to keep in mind. We made an evaluated random predictions with our best
model, which is an important step in visualizing, visualizing, visualizing your model's predictions,
because you could get evaluation metrics. But until you start to actually visualize what's going on,
well, in my case, that's how I best understand what my model is thinking. We saw a confusion
matrix using two different libraries torch metrics and ML extend a great way to evaluate
your classification models. And we saw how to save and load the best performing model to file
and made sure that the results of our saved model weren't too different from the model that
we trained within the notebook. So now it is time I'd love for you to practice what
you've gone through. This is actually really exciting now because you've gone through an end-to-end
computer vision problem. I've got some exercises prepared. If you go to the learn pytorch.io website
in section 03, scroll down. You can read through all of this. This is all the materials that we've
just covered in pure code. There's a lot of pictures in this notebook too that are

```
helpful to learn
things what's going on. We have some exercises here. So all of the exercises are
focused on
practicing the code and the sections above. We have two resources. We also have
some extra
curriculum that I've put together. If you want an in-depth understanding of what's
going on
behind the scenes in the convolutional neural networks, because we've focused a lot
on code,
I'd highly recommend MIT's induction to deep computer vision lecture. You can spend
10 minutes
```

Section 198: Main Topics

Key Topics:

- clicking through the different options in the pytorch vision library, torch vision, look up most common convolutional neural networks in the torch vision model library, and then for a larger number of pre-trained pytorch computer vision models, and if you get deeper into computer vision, you're probably going to run into the torch image models library, otherwise known as 10, but I'm going to leave that as extra curriculum
- Again, it's at learn pytorch
- Now this is in the pytorch deep learning repo, extras exercises number three

▶ 📄 Click to view detailed content

```
clicking through the different options in the pytorch vision library, torch vision,
look up most
common convolutional neural networks in the torch vision model library, and then
for a larger number
of pre-trained pytorch computer vision models, and if you get deeper into computer
vision,
you're probably going to run into the torch image models library, otherwise known
as 10,
but I'm going to leave that as extra curriculum. I'm going to just link this
exercises section
here. Again, it's at learn pytorch.io in the exercises section. We come down. There
we go.
But there is also resource here, an exercise template notebook. So we've got one,
what are
three areas in industry where computer vision is being currently used. Now this is
in the
pytorch deep learning repo, extras exercises number three. I've put out some
template code here
for you to fill in these different sections. So some of them are code related. Some
```

of them
are just text based, but they should all be able to be completed by referencing what we've gone
through in this notebook here. And just as one more, if we go back to pytorch deep learning,
this will probably be updated by the time you get here, you can always find the exercise in
extra curriculum by going computer vision, go to exercise in extra curriculum, or if we go into
the extras file, and then we go to solutions. I've now also started to add video walkthroughs
of each of the solutions. So this is me going through each of the exercises myself and coding
them. And so you'll get to see the unedited videos. So they're just one long live stream.
And I've done some for O2, O3, and O4, and there will be more here by the time you watch this video.
But if you'd like to see how I figure out the solutions to the exercises, you can watch those
videos and go through them yourself. But first and foremost, I would highly recommend trying out
the exercises on your own first. And then if you get stuck, refer to the notebook here,
refer to the pytorch documentation. And finally, you can check out what I would have coded as a
potential solution. So there's number three, computer vision, exercise solutions. So congratulations
on going through the pytorch computer vision section. I'll see you in the next section. We're
going to look at pytorch custom data sets, but no spoilers. I'll see you soon.
Hello, hello, hello, and welcome to section number four of the Learn pytorch for deep learning course.
We have custom data sets with pytorch. Now, before we dive into what we're going to cover,
let's answer the most important question. Where can you get help? Now, we've been through this
a few times now, but it's important to reiterate. Follow along with the code as best you can. We're
going to be writing a bunch of pytorch code. Remember the motto, if and out, run the code.
That's in line with try it for yourself. If you'd like to read or read the doxtring,
you can press shift command plus space in Google Colab. Or if you're on Windows, command might
be control. Then if you're still stuck, you can search for it. Two of the resources you will
probably come across is stack overflow or the wonderful pytorch documentation, which we've
had a lot of experience with so far. Then, of course, try again, go back through your code,
if and out, code it out, or if and out, run the code. And then finally, if you're still stuck,
ask a question on the pytorch deep learning discussions GitHub page. So if I click this link,
we come to Mr. D Burke slash pytorch deep learning, the URL is here. We've seen this before. If you
have a trouble or a problem with any of the course, you can start a discussion and

```
you can
select the category, general ideas, polls, Q and A, and then we can go here, video,
put the video number in. So 99, for example, my code doesn't do what I'd like it
to. So say
your problem and then come in here, write some code here, code here, and then my
question is
something, something, something, click start discussion, and then we can help out.
And then if
we come back to the discussions, of course, you can search for what's going on. So
if you have an
error and you feel like someone else might have seen this error, you can, of
course, search it
and find out what's happening. Now, I just want to highlight again, the resources
for this course
are at learn pytorch.io. We are up to section four. This is a beautiful online book
version of
all the materials we are going to cover in this section. So spoiler alert, you can
use this as a
reference. And then, of course, in the GitHub, we have the same notebook here,
pytorch custom
data sets. This is the ground truth notebook. So check that out if you get stuck.
So I'm just
going to exit out of this. We've got pytorch custom data sets at learn pytorch.io.
And then,
of course, the discussions tab for the Q&A. Now, if we jump back to the keynote,
what do we have?
We might be asking, what is a custom data set? Now, we've built a fair few pytorch
deeplining
neural networks so far on various data sets, such as fashion MNIST. But you might
be wondering,
hey, I've got my own data set, or I'm working on my own problem. Can I build a
model with pytorch
to predict on that data set? And the answer is yes. However, you do have to go
through a few
pre processing steps to make that data set compatible with pytorch. And that's what
we're
```

Section 199: Main Topics

Key Topics:

- And so I'd like to highlight the pytorch domain libraries
- And of course, the bonus is torch data
- As I said, we want to load these images into PyTorch so that we can build a model

▶ 📄 Click to view detailed content

```
going to be covering in this section. And so I'd like to highlight the pytorch
domain libraries.
```

Now, we've had a little bit of experience before with torch vision, such as if we wanted to classify
whether a photo was a pizza, steak, or sushi. So a computer vision image classification problem.
Now, there's also text, such as if these reviews are positive or negative. And you can use torch
text for that. But again, these are only just one problem within the vision space within the text
space. I want you to just understand that if you have any type of vision data, you probably
want to look into torch vision. And if you have any kind of text data, you probably want to look
into torch text. And then if you have audio, such as if you wanted to classify what song was playing,
this is what Shazam does, it uses the input sound of some sort of music, and then runs a neural network
over it to classify it to a certain song, you can look into torch audio for that. And then if you'd
like to recommend something such as you have an online store, or if your Netflix or something
like that, and you'd like to have a homepage that updates for recommendations, you'd like to look
into torch rec, which stands for recommendation system. And so this is just something to keep in mind.
Because each of these domain libraries has a data sets module that helps you work with different
data sets from different domains. And so different domain libraries contain data loading functions
for different data sources. So torch vision, let's just go into the next slide, we have problem space
vision for pre built data sets, so existing data sets like we've seen with fashion MNIST,
as well as functions to load your own vision data sets, you want to look into torch vision
dot data sets. So if we click on this, we have built in data sets, this is the pie torch documentation.
And if we go here, we have torch audio, torch text, torch vision, torch rec, torch data. Now,
at the time of recording, which is April 2022, this is torch data is currently in beta. But it's
going to be updated over time. So just keep this in mind, updated over time to add even more ways
to load different data resources. But for now, we're just going to get familiar with torch vision
data sets. If we went into torch text, there's another torch text dot data sets. And then if we
went into torch audio, we have torch audio dot data sets. And so you're noticing a trend here
that depending on the domain you're working in, whether it be vision, text, audio, or your data
is recommendation data, you'll probably want to look into its custom library within pie torch.
And of course, the bonus is torch data. It contains many different helper functions for loading data,
and is currently in beta as of April 2022. So 2022. So the by the time you watch this torch data
may be out of beta. And then that should be something that's extra curriculum on

```
top of what we're
going to cover in this section. So let's keep going. So this is what we're going to
work towards
building food vision mini. So we're going to load some data, namely some images of
pizza,
sushi, and steak from the food 101 data set, we're going to build an image
classification model,
such as the model that might power a food vision recognition app or a food image
recognition app.
And then we're going to see if it can classify an image of pizza as pizza, an image
of sushi as sushi,
and an image of steak as steak. So this is what we're going to focus on. We want to
load,
say we had images existing already of pizza, sushi, and steak, we want to write
some code
to load these images of food. So our own custom data set for building this food
vision mini model,
which is quite similar to if you go to this is the project I'm working on
personally,
neutrify.app. This is a food image recognition model. Here we go. So it's still a
work in progress as
I'm going through it, but you can upload an image of food and neutrify will try to
classify
what type of food it is. So do we have steak? There we go. Let's upload that.
Beautiful steak.
So we're going to be building a similar model to what powers neutrify. And then
there's the
macro nutrients for the steak. If you'd like to find out how it works, I've got all
the links here,
but that's at neutrify.app. So let's keep pushing forward. We'll go back to the
keynote.
This is what we're working towards. As I said, we want to load these images into
PyTorch so that
we can build a model. We've already built a computer vision model. So we want to
figure out
how do we get our own data into that computer vision model. And so of course we'll
be adhering
to our PyTorch workflow that we've used a few times now. So we're going to learn
how to load a
data set with our own custom data rather than an existing data set within PyTorch.
We'll see how
we can build a model to fit our own custom data set. We'll go through all the steps
that's involved
in training a model such as picking a loss function and an optimizer. We'll build a
training loop.
We'll evaluate our model. We'll improve through experimentation. And then we can
see save and reloading
our model. But we're also going to practice predicting on our own custom data,
which is a very,
very important step whenever training your own models. So what we're going to cover
broadly,
```

Section 200: Main Topics

Key Topics:

- we're going to get a custom data set with PyTorch
- But I like to treat machine learning as a little bit of an art, so we're going to be cooking up lots of code
- Welcome back to the PyTorch cooking show

▶ 📄 Click to view detailed content

```
we're going to get a custom data set with PyTorch. As we said, we're going to
become one with the
data. In other words, preparing and visualizing it. We'll learn how to transform
data for use with
a model, very important step. We'll see how we can load custom data with pre-built
functions
and our own custom functions. We'll build a computer vision model, aka food vision
mini,
to classify pizza, steak, and sushi images. So a multi-class classification model.
We'll compare
models with and without data augmentation. We haven't covered that yet, but we will
later on.
And finally, we'll see how we can, as I said, make predictions on custom data. So
this means
data that's not within our training or our test data set. And how are we going to
do it? Well,
we could do it cooks or chemists. But I like to treat machine learning as a little
bit of an art,
so we're going to be cooking up lots of code. With that being said, I'll see you in
Google Colab.
Let's code. Welcome back to the PyTorch cooking show. Let's now learn how we can
cook up some
custom data sets. I'm going to jump into Google Colab. So
colab.research.google.com.
And I'm going to click new notebook. I'm just going to make sure this is zoomed in
enough for
the video. Wonderful. So I'm going to rename this notebook 04 because we're up to
section 04.
And I'm going to call it PyTorch custom data sets underscore video because this is
going to be one
of the video notebooks, which has all the code that I write during the videos,
which is of course
contained within the video notebooks folder on the PyTorch deep learning repo. So
if you'd like
the resource or the ground truth notebook for this, I'm going to just put a heading
here.
04 PyTorch custom data sets video notebook, make that bigger, and then put
resources.
So book version of the course materials for 04. We'll go there, and then we'll go
ground truth
version of notebook 04, which will be the reference notebook that we're going to
use
for this section. Come into PyTorch custom data sets. And then we can put that in
there.
```

Wonderful. So the whole synopsis of this custom data sets section is we've used some data sets
with PyTorch before, but how do you get your own data into PyTorch? Because that's what you
want to start working on, right? You want to start working on problems of your own. You want to
come into any sort of data that you've never worked with before, and you want to figure out how do
you get that into PyTorch. So one of the ways to do so is via custom data sets. And then I want
to put a note down here. So we're going to go zero section zero is going to be importing
PyTorch and setting up device agnostic code. But I want to just stress here that domain libraries.
So just to reiterate what we went through last video. So depending on what you're working on,
whether it be vision, text, audio, recommendation, something like that, you'll want to look into
each of the PyTorch domain libraries for existing data loader or data loading functions and
customizable data loading functions. So just keep that in mind. We've seen some of them. So if we
go torch vision, which is what we're going to be looking at, torch vision, we've got data sets,
and we've got documentation, we've got data sets for each of the other domain libraries here as
well. So if you're working on a text problem, it's going to be a similar set of steps to what
we're going to do with our vision problem when we build food vision mini. What we have is a data
set that exists somewhere. And what we want to do is bring that into PyTorch so we can build a
model with it. So let's import the libraries that we need. So we're going to import torch and
we'll probably import an N. So we'll import that from PyTorch. And I'm just going to check the
torch version here. So note, we need PyTorch 1.10.0 plus is required for this course. So if you're
using Google Colab at a later date, you may have a later version of PyTorch. I'm just going to
show you what version I'm using. Just going to let this load. We're going to get this ready.
We're going to also set up device agnostic code right from the start this time because this is
best practice with PyTorch. So this way, if we have a CUDA device available, our model is going
to use that CUDA device. And our data is going to be on that CUDA device. So there we go. Wonderful.
We've got PyTorch 1.10.0 plus CUDA. 111. Maybe that's 11.1. So let's check if CUDA.is available.
Now, I'm using Google Colab. We haven't set up a GPU yet. So it probably won't be available yet.
Let's have a look. Wonderful. So because we've started a new Colab instance, it's going to use
the CPU by default. So how do we change that? We come up to runtime, change runtime type. I'm going
to go hard there accelerator GPU. We've done this a few times now. I am paying for

```
Google Colab Pro.
So one of the benefits of that is that it our Google Colab reserves faster GPUs for
you. You do
don't need Google Colab Pro. As I've said to complete this course, you can use the
free version,
but just recall Google Colab Pro tends to give you a better GPU just because GPUs
aren't free.
Wonderful. So now we've got access to a GPU CUDA. What GPU do I have?
Nvidia SMI. I have a Tesla P100 with 16 gigabytes of memory, which will be more
than enough for
```

Section 201: Main Topics

Key Topics:

- But now, oh, and also, if you go to loan pytorch
- But now we're going to write some code, because this data set, the smaller version that I've created is on the pytorch deep learning repo, under data
- And this, of course, will depend on where your data set lives, what you'd like to do

▶ 📄 Click to view detailed content

```
the problem that we're going to work on in this video. So I believe that's enough
to cover for
the first coding video. Let's in the next section, we are working with custom
datasets after all.
Let's in the next video. Let's get some data, hey.
Now, as I said in the last video, we can't cover custom datasets without some data.
So let's get
some data and just remind ourselves what we're going to build. And that is food
vision mini.
So we need a way of getting some food images. And if we go back to Google Chrome,
torch vision datasets has plenty of built-in datasets. And one of them is the food
101 dataset.
Food 101. So if we go in here, this is going to take us to the original food 101
website.
So food 101 is 101 different classes of food. It has a challenging dataset of 101
different
food categories with 101,000 images. So that's a quite a beefy dataset. And so for
each class,
250 manually reviewed test images are provided. So we have per class, 101 classes,
250 testing
images, and we have 750 training images. Now, we could start working on this entire
dataset
straight from the get go. But to practice, I've created a smaller subset of this
dataset,
and I'd encourage you to do the same with your own problems. Start small and
upgrade when necessary.
```

```
So I've reduced the number of categories to three and the number of images to 10%.
Now, you could reduce this to an arbitrary amount, but I've just decided three is
enough to begin with
and 10% of the data. And then if it works, hey, you could upscale that on your own
accord.
And so I just want to show you the notebook that I use to create this dataset and
as extra curriculum,
you could go through this notebook. So if we go into extras, 04 custom data
creation,
this is just how I created the subset of data. So making a dataset to use with
notebook number
four, I created it in custom image data set or image classification style. So we
have a top level
folder of pizza, steak, and sushi. We have a training directory with pizza, steak,
and sushi
images. And we have a test directory with pizza, steak, and sushi images as well.
So you can go
through that to check it out how it was made. But now, oh, and also, if you go to
loan pytorch.io
section four, there's more information here about what food 101 is. So get data.
Here we go.
There's all the information about food 101. There's some resources, the original
food 101 data set,
torch vision data sets, food 101, how I created this data set, and actually
downloading the data.
But now we're going to write some code, because this data set, the smaller version
that I've created
is on the pytorch deep learning repo, under data. And then we have pizza, steak,
sushi.zip.
Oh, this one is a little spoiler for one of the exercises for this section. But
you'll see that
later. Let's go in here. Let's now write some code to get this data set from
GitHub,
pizza, steak, sushi.zip. And then we'll explore it, we'll become one with the data.
So I just want to write down here, our data set is a subset of the food 101 data
set.
Food 101 starts with 101 different classes of food. So we could definitely build
computer
vision models for 101 classes, but we're going to start smaller. Our data set
starts with three
classes of food, and only 10% of the images. So what's right here? And 1000 images
per class,
which is 750 training, 250 testing. And we have about 75 training images per class,
and about 25 testing images per class. So why do this? When starting out ML
projects,
it's important to try things on a small scale and then increase the scale when
necessary.
The whole point is to speed up how fast you can experiment.
Because there's no point trying to experiment on things that if we try to train on
100,000
images to begin with, our models might train take half an hour to train at a time.
So at the
beginning, we want to increase the rate that we experiment at. And so let's get
some data.
We're going to import requests so that we can request something from GitHub to
download this
URL here. Then we're also going to import zip file from Python, because our data is
```

```
in the form
of a zip file right now. Then we're going to get path lib, because I like to use
paths whenever
I'm dealing with file paths or directory paths. So now let's set up a path to a
data folder.
And this, of course, will depend on where your data set lives, what you'd like to
do. But I
typically like to create a folder over here called data. And that's just going to
store all of my
data for whatever project I'm working on. So data path equals path data. And then
we're going to go
image path equals data path slash pizza steak sushi. That's how we're going to have
images
from those three classes. Pizza steak and sushi are three of the classes out of the
101 in food
101. So if the image folder doesn't exist, so if our data folder already exists, we
don't want to
redownload it. But if it doesn't exist, we want to download it and unzip it. So if
image path
is der, so we want to print out the image path directory already exists skipping
download.
And then if it doesn't exist, we want to print image path does not exist, creating
one. Beautiful.
And so we're going to go image path dot mk der to make a directory. We want to make
its parents
```

Section 202: Main Topics

Key Topics:

- And then the whole premise of this entire section will be loading this data of just images into PyTorch so that we can build a computer vision model on it
- Who knows wherever your data is, but you'll want to write code to load it from here into PyTorch

▶ 📄 Click to view detailed content

```
if we need to. So the parent directories and we want to pass exist, okay, equals
true. So we don't
get any errors if it already exists. And so then we can write some code. I just
want to show you
what this does if we run it. So our target directory data slash pizza steak sushi
does not exist.
It's creating one. So then we have now data and inside pizza steak sushi.
Wonderful. But we're
going to fill this up with some images so that we have some data to work with. And
then the whole
premise of this entire section will be loading this data of just images into
```

PyTorch so that we
can build a computer vision model on it. But I just want to stress that this step will be very
similar no matter what data you're working with. You'll have some folder over here or maybe it'll
live on the cloud somewhere. Who knows wherever your data is, but you'll want to write code to
load it from here into PyTorch. So let's download pizza steak and sushi data. So I'm going to use
width. I'll just X over here. So we have more screen space with open. I'm going to open the data
path slash the file name that I'm trying to open, which will be pizza steak sushi dot zip. And I'm
going to write binary as F. So this is essentially saying I'm doing this in advance because I know
I'm going to download this folder here. So I know the the file name of it, pizza steak sushi dot zip.
I'm going to download that into Google collab and I want to open it up. So request equals request
dot get. And so when I want to get this file, I can click here. And then if I click download,
it's going to what do you think it's going to do? Well, let's see. If I wanted to download it
locally, I could do that. And then I could come over here. And then I could click upload if I
wanted to. So upload the session storage. I could upload it from that. But I prefer to write code
so that I could just run this cell over again and have the file instead of being download to
my local computer. It just goes straight into Google collab. So to do that, we need the URL
from here. And I'm just going to put that in there. It needs to be as a string.
Excuse me. I'm getting trigger happy on the shift and enter. Wonderful. So now I've got a request
to get the content that's in here. And GitHub can't really show this because this is a zip file
of images, spoiler alert. Now let's keep going. We're going to print out that we're downloading
pizza, stake and sushi data dot dot dot. And then I'm going to write to file the request dot content.
So the content of the request that I just made to GitHub. So that's request is here.
Using the Python request library to get the information here from GitHub. This URL could be
wherever your file has been stored. And then I'm going to write the content of that request
to my target file, which is this. This here. So if I just copy this, I'm going to write the data
to here data path slash pizza, stake sushi zip. And then because it's a zip file, I want to unzip it.
So unzip pizza, stake sushi data. Let's go with zip file. So we imported zip file up there,
which is a Python library to help us deal with zip files. We're going to use zip file dot zip
file. We're going to pass it in the data path. So just the path that we did below,
data path slash pizza, stake sushi dot zip. And this time, instead of giving it right permissions,

```
so that's what wb stands for, stands for right binary. I'm going to give it read
permissions.
So I want to read this target file instead of writing it. And I'm going to go as
zip ref.
We can call this anything really, but zip ref is kind of, you'll see this a lot in
different Python examples. So we're going to print out again. So unzipping pizza,
stake,
and sushi data. Then we're going to go zip underscore ref dot extract all. And
we're going to go image
path. So what this means is it's taking the zip ref here. And it's extracting all
of the
information that's within that zip ref. So within this zip file, to the image path,
which is what we created up here. So if we have a look at image path, let's see
that.
Image path. Wonderful. So that's where all of the contents of that zip file are
going to go
into this file. So let's see it in action. You're ready. Hopefully it works. Three,
two, one, run.
File is not a zip file. Oh, no, what do we get wrong? So did I type this wrong?
Got zip data path. Oh, we got the zip file here. Pizza, stake, sushi, zip, read
data path.
Okay, I found the error. So this is another thing that you'll have to keep in mind.
And I believe we've covered this before, but I like to keep the errors in these
videos so that
you can see where I get things wrong, because you never write code right the first
time.
So we have this link in GitHub. We have to make sure that we have the raw link
address. So if I
come down to here and copy the link address from the download button, you'll notice
a slight
difference if we come back into here. So I'm just going to copy that there. So if
we step
```

Section 203: Main Topics

Key Topics:

- D Burke pytorch deep learning, we have raw instead of blob
- We'll be loading a target data set and then writing code to convert whatever the format the data set is in into tenses for PyTorch
- In the last video, we wrote some code to download a target data set, our own custom data set from the PyTorch deep learning data directory

▶ 📄 Click to view detailed content

```
through this GitHub, Mr. D Burke pytorch deep learning, we have raw instead of
blob. So that
is why we've had an error is that our code is correct. It's just downloading the
```

wrong data.
So let's change this to the raw. So just keep that in mind, you must have raw here.
And so let's see if this works.
Do we have the correct data? Oh, we might have to delete this. Oh, there we go.
Test. Beautiful. Train. Pizza steak sushi. Wonderful. So it looks like we've got some data. And if we
open this up, what do we have? We have various JPEGs. Okay. So this is our testing data. And if
we click on there, we've got an image of pizza. Beautiful. So we're going to explore this a
little bit more in the next video. But that is some code that we've written to download data sets
or download our own custom data set. Now, just recall that we are working specifically on a pizza
steak and sushi problem for computer vision. However, our whole premise is that we have some
custom data. And we want to convert these. How do we get these into tenses? That's what we want
to do. And so the same process will be for your own problems. We'll be loading a target data set
and then writing code to convert whatever the format the data set is in into tenses for PyTorch.
So I'll see you in the next video. Let's explore the data we've downloaded.
Welcome back. In the last video, we wrote some code to download a target data set, our own custom
data set from the PyTorch deep learning data directory. And if you'd like to see how that
data set was made, you can go to PyTorch deep learning slash extras. It's going to be in the
custom data creation notebook here for 04. So I've got all the code there. All we've done is take
data from the food 101 data set, which you can download from this website here, or from torch
vision. So if we go to torch vision, food 101. We've got the data set built into PyTorch there.
So I've used that data set from PyTorch and broken it down from 101 classes to three classes so that
we can start with a small experiment. So there we go. Get the training data, data sets food 101,
and then I've customized it to be my own style. So if we go back to CoLab, we've now got
pizza steak sushi, a test folder, which will be our testing images, and a train folder,
which will be our training images. This data is in standard image classification format. But we'll
cover that in a second. All we're going to do in this video is kick off section number two,
which is becoming one with the data, which is one of my favorite ways to refer to data preparation
and data exploration. So we're coming one with the data. And I'd just like to show you one of my
favorite quotes from Abraham loss function. So if I had eight hours to build a machine learning model,
I'd spend the first six hours preparing my data set. And that's what we're going to do. Abraham
loss function sounds like he knows what is going on. But since we've just downloaded some data,

```
let's explore it. Hey, and we'll write some code now to walk through each of the
directories. How
you explore your data will depend on what data you've got. So we've got a fair few
different
directories here with a fair few different folders within them. So how about we
walk through each
of these directories and see what's going on. If you have visual data, you probably
want to
visualize an image. So we're going to do that in the second two, write a little doc
string for
this helper function. So walks through the path, returning its contents. Now, just
in case you didn't
know Abraham loss function does not exist as far as I know. But I did make up that
quote. So we're
going to use the OS dot walk function, OS dot walk. And we're going to pass it in a
dirt path. And
what does walk do? We can get the doc string here. Directory tree generator. For
each directory
in the directory tree rooted at the top, including top itself, but in excluding dot
and dot dot,
yields a three tuple, derpath, der names, and file names. You can step through this
in the Python
documentation, if you'd like. But essentially, it's just going to go through our
target directory,
which in this case will be this one here. And walk through each of these
directories printing out
some information about each one. So let's see that in action. This is one of my
favorite things to do
if we're working with standard image classification format data. So there are lane,
length,
der names, directories. And let's go land, land, file names. We say at length, like
I've got the
G on the end, but it's just land images in, let's put in here, derpath. So a little
bit confusing
if you've never used walk before, but it's so exciting to see all of the
information in all
of your directories. Oh, we didn't read and run it. Let's check out function now
walk through der.
And we're going to pass it in the image path, which is what? Well, it's going to
show us.
How beautiful. So let's compare what we've got in our printout here. There are two
directories
and zero images in data, pizza, steak sushi. So this one here, there's zero images,
but there's
two directories test and train wonderful. And there are three directories in data,
pizza, steak, sushi,
test. Yes, that looks correct. Three directories, pizza, steak, sushi. And then we
have zero
```

Section 204: Main Topics

Key Topics:

- But the premise remains, we're going to be writing code to get our data here into tenses for use with PyTorch

▶ 📄 Click to view detailed content

```
directories and 19 images in pizza, steak, sushi, slash test, steak. We have a look
at this. So that
means there's 19 testing images for steak. Let's have a look at one of them. There
we go. Now,
again, these are from the food 101 data set, the original food 101 data set, which
is just a whole
bunch of images of food, 100,000 of them. There's some steak there. Wonderful. And
we're trying to
build a food vision model to recognize what is in each image. Then if we jump down
to here,
we have three directories in the training directory. So we have pizza, steak,
sushi. And then we have
75 steak images, 72 sushi images and 78 pizza. So slightly different, but very much
the same
numbers. They're not too far off each other. So we've got about 75 or so training
images,
and we've got about 25 or so testing images per class. Now these were just randomly
selected
from the food 101 data set 10% of three different classes. So let's keep pushing
forward. And we're
going to set up our training and test parts. So I just want to show you, we'll just
set up this,
and then I'll just show you the standard image classification setup, image
path.train. And we're
going to go tester. So if you're working on image classification problem, we want
to set this up
as test. And then if we print out the trainer and the tester, this is what we're
going to be
trying to do. We're going to write some code to go, Hey, look at this path for our
training images.
And look at this path for our testing images. And so this is the standard image
classification
data format is that you have your overall data set folder. And then you have a
training folder
dedicated to all of the training images that you might have. And then you have a
testing folder
dedicated to all of the testing images that you might have. And you could have a
validation
data set here as well if you wanted to. But to label each one of these images, the
class name
is the folder name. So all of the pizza images live in the pizza directory, the
same for steak,
and the same for sushi. So depending on your problem, your own data format will
depend on
whatever you're working on, you might have folders of different text files or
folders of
different audio files. But the premise remains, we're going to be writing code to
get our data here
into tenses for use with PyTorch. And so where does this come from? This image data
classification
```

format. Well, if we go to the torch vision dot data sets documentation, as you start to work
with more data sets, you'll start to realize that there are standardized ways of storing
specific types of data. So if we come down to here, base classes for custom data sets,
we'll be working towards using this image folder data set. But this is a generic data
loader where the images are arranged in this way by default. So I've specifically formatted our data
to mimic the style that this pre built data loading function is for. So we've got a root directory
here in case of we were classifying dog and cat images, we have root, then we have a dog folder,
then we have various images. And the same thing for cat, this would be dog versus cat. But the only
difference for us is that we have food images, and we have pizza steak sushi. If we wanted to use the
entire food 101 data set, we would have 101 different folders of images here, which is totally
possible. But to begin with, we're keeping things small. So let's keep pushing forward. As I said,
we're dealing with a computer vision problem. So what's another way to explore our data,
other than just walking through the directories themselves. Let's visualize an image, hey? But
we've done that before with just clicking on the file. How about we write some code to do so.
We'll replicate this but with code. I'll see you in the next video.
Welcome back. In the last video, we started to become one with the data. And we learned that we
have about 75 images per training class and about 25 images per testing class. And we also learned
that the standard image classification data structure is to have the steak images within the steak
folder of the training data set and the same for test, and the pizza images within the pizza
folder, and so on for each different image classification class that we might have.
So if you want to create your own data set, you might format it in such a way that your training
images are living in a directory with their classification name. So if you wanted to classify
photos of dogs and cats, you might create a training folder of train slash dog train slash
cat, put images of dogs in the dog folder, images of cats in the cat folder, and then the same for
the testing data set. But the premise remains, I'm going to sound like a broken record here.
We want to get our data from these files, whatever files they may be in, whatever data structure
they might be in, into tenses. But before we do that, let's keep becoming one with the data.
And we're going to visualize an image. So visualizing an image, and you know how much I love randomness.
So let's select a random image from all of the files that we have in here. And let's plot it,
hey, because we could just click through them and visualize them. But I like to do

```
things with
code. So specifically, let's let's plan this out. Let's write some code to number
one is get all
```

Section 205: Main Topics

Key Topics:

- So let's import random, because machine learning is all about harnessing the power of randomness

▶ 📄 Click to view detailed content

```
of the image paths. We'll see how we can do that with the path path lib library. We
then want to
pick a random image path using we can use Python's random for that. Python's random
dot choice will
pick a single image random dot choice. Then we want to get the image class name.
And this is where
part lib comes in handy. Class name, recall that whichever target image we pick,
the class name will
be whichever directory that it's in. So in the case of if we picked a random image
from this directory,
the class name would be pizza. So we can do that using, I think it's going to be
path lib dot path.
And then we'll get the parent folder, wherever that image lives. So the parent
image parent
folder that parent directory of our target random image. And we're going to get the
stem of that.
So we have stem, stem is the last little bit here. Number four, what should we do?
Well,
we want to open the image. So since we're working with images, let's open the image
with Python's pill, which is Python image library, but we'll actually be pillow. So
if we go Python
pillow, a little bit confusing when I started to learn about Python image
manipulation. So pillow
is a friendly pill for, but it's still called pill. So just think of pillow as a
way to process
images with Python. So pill is the Python imaging library by Frederick Lund. And so
Alex Clark and
contributors have created pillow. So thank you, everyone. And let's go to number
five. What do
we want to do as well? We want to, yeah, let's get some metadata about the image.
We'll then show
the image and print metadata. Wonderful. So let's import random, because machine
learning is all
about harnessing the power of randomness. And I like to use randomness to explore
data as well
as model it. So let's set the seed. So we get the same image on both of our ends.
```

So random dot seed.
I'm going to use 42. You can use whatever you'd like. But if you'd like to get the same image as me,
I'd suggest using 42 as well. Now let's get all the image paths. So we can do this because our image
path list, we want to get our image path. So recall that our image path
is this. So this folder here, I'm just going to close all this. So this is our image path,
this folder here, you can also go copy path if you wanted to, we're just going to get something
very similar there. That's going to error out. So I'll just comment that. So it doesn't error.
That's our path. But we're going to keep it in the POSIX path format. And we can go list. Let's
create a list of image path dot glob, which stands for grab. I don't actually know what glob stands
for. But to me, it's like glob together. All of the images that are all of the files that suit
a certain pattern. So glob together for me means stick them all together. And you might be able
to correct me if I've got the wrong meaning there. I'd appreciate that. And so we're going to pass
in a certain combination. So we want star slash star. And then we want star dot jpg. Now why are
we doing this? Well, because we want every image path. So star is going to be this first
directory here. So any combination, it can be train or test. And then this star means anything for
what's inside tests. And let's say this first star is equal to test. This second star is equal to
anything here. So it could be any of pizza, steak or sushi. And then finally, this star,
let's say it was test pizza. This star is anything in here. And that is before dot jpg.
So it could be any one of these files here. Now this will make more sense once we print it out.
So image path list, let's have a look. There we go. So now we've got a list of every single image
that's within pizza steak sushi. And this is just another way that I like to visualize data is to
just get all of the paths and then randomly visualize it, whether it be an image or text or
audio, you might want to randomly listen to it. Recall that each each of the domain libraries have
different input and output methods for different data sets. So if we come to torch vision, we have
utils. So we have different ways to draw on images, reading and writing images and videos. So we
could load an image via read image, we could decode it, we could do a whole bunch of things.
I'll let you explore that as extra curriculum. But now let's select a random image from here
and plot it. So we'll go number two, which was our step up here, pick a random image. So pick a
random image path. Let's get rid of this. And so we can go random image path equals random
dot choice, harness the power of randomness to explore our data. Let's get a random

```
image from
image path list, and then we'll print out random image path, which one was our
lucky image that
we selected. Beautiful. So we have a test pizza image is our lucky random image.
And
because we've got a random seed, it's going to be the same one each time. Yes, it
is.
And if we comment out the random seed, we'll get a different one each time. We've
got a stake
image. We've got another stake image. Another stake image. Oh, three in a row, four
in a row.
Oh, pizza. Okay, let's keep going. So we'll get the image class
```

Section 206: Main Topics

Key Topics:

- We could also open up the image with pytorch here
- Oh, we are well on the way to creating our own PyTorch custom data set
- Because what's one of the main errors in machine learning and deep learning

▶ 📄 Click to view detailed content

```
from the path name. So the image class is the name of the directory, because our
image data is
in standard image classification format, where the image is stored. So let's do
that image class
equals random image path dot parent dot stem. And then we're going to print image
class. What do we
get? So we've got pizza. Wonderful. So the parent is this folder here. And then the
stem is the end
of that folder, which is pizza. Beautiful. Well, now what are we up to now? We're
working with
images. Let's open up the image so we can open up the image using pill. We could
also open up the
image with pytorch here. So with read image, but we're going to use pill to keep
things a little
bit generic for now. So open image, image equals image. So from pill import image,
and the image
class has an open function. And we're just going to pass it in here, the random
image path. Note
if this is corrupt, if your images corrupt, this may error. So then you could
potentially use this
to clean up your data set. I've imported a lot of images with image dot open of our
target data
set here. I don't believe any of them are corrupt. But if they are, please let me
know. And we'll find
out later on when our model tries to train on it. So let's print some metadata. So
when we open our
```

image, we get some information from it. So let's go our random image path is what? Random image path.
We're already printing this out, but we'll do it again anyway. And then we're going to go the image
class is equal to what will be the image class. Wonderful. And then we can print out, we can get
some metadata about our images. So the image height is going to be IMG dot height. We get that
metadata from using the pill library. And then we're going to print out image width. And we'll get
IMG dot width. And then we'll print the image itself. Wonderful. And we can get rid of this,
and we can get rid of this. Let's now have a look at some random images from our data set.
Lovely. We've got an image of pizza there. Now I will warn you that the downsides of working with
food data is it does make you a little bit hungry. So there we've got some sushi. And then we've got
some more sushi. Some steak. And we have a steak, we go one more for good luck. And we finish off
with some sushi. Oh, that could be a little bit confusing to me. I thought that might be steak
to begin with. And this is the scene. Now we'll do one more. Why it's important to sort of visualize
your images randomly, because you never know what you're going to come across. And this way,
once we visualize enough images, you could do this a hundred more times. You could do this
20 more times until you feel comfortable to go, Hey, I feel like I know enough about the data now.
Let's see how well our model goes on this sort of data. So I'll finish off on this steak image.
And now I'll set your little challenge before the next video is to visualize an image like we've
done here. But this time do it with matplotlib. So try to visualize an image with matplotlib.
That's your little challenge before the next video. So give that a go. We want to do a random
image as well. So quite a similar set up to this. But instead of printing out things like this,
we want to visualize it using matplotlib. So try that out and we'll do it together in the next video.
Oh, we are well on the way to creating our own PyTorch custom data set. We've started to
become one with the data. But now let's continue to visualize another image. I set you the challenge
in the last video to try and replicate what we've done here with the pill library with matplotlib.
So now let's give it a go. Hey, and why use matplotlib? Well, because matplotlib and I'm going to
import numpy as well, because we're going to have to convert this image into an array. That was a
little trick that I didn't quite elaborate on. But I hope you tried to decode it out and figure
it out from the errors you received. But matplotlib is one of the most fundamental data science
libraries. So you're going to see it everywhere. So it's just important to be aware

```
of how to plot
images and data with matplotlib. So turn the image into an array. So we can go
image as array. And
I'm going to use the numpy method NP as array. We're going to pass it in the image,
recall that
the image is the same image that we've just set up here. And we've already opened
it with pill.
And then I'm going to plot the image. So plot the image with matplotlib.
plt.figure.
And then we can go fig size equals 10, seven. And then we're going to go plt.im
show image as
array, pass it in the array of numbers. I'm going to set the title here as an f
string. And then
I'm going to pass in image class, equals image class. Then I'm going to pass in
image shape. So
we can get the shape here. Now this is another important thing to be aware of of
your different
datasets when you're exploring them is what is the shape of your data? Because
what's one of the
main errors in machine learning and deep learning? It's shape mismatch issues. So
if we know the
```

Section 207: Main Topics

Key Topics:

- But pytorch recall is default if we put the color channels at the start color channels first
- But for now pytorch defaults to color channels first
- But now we want to write code to turn all of these images into pytorch tenses

▶ 📄 Click to view detailed content

```
shape of our data where we can start to go, okay, I kind of understand what shape I
need my model
layers to be in what what shape I need my other data to be in. And I'm going to
turn the axes off
here. Beautiful. So look at what we've got. Now I've just thrown this in here
without really
explaining it. But we've seen this before in the computer vision section. As our
image shape is
512 3063. Now the dimensions here are height is 512 pixels. The width is 306
pixels. And it has
three color channels. So what format is this? This is color channels last, which is
the default
for the pill library. There's also the default for map plot lib. But pytorch recall
is default
if we put the color channels at the start color channels first. Now there is a lot
```

of debate as
I've said over which is the best order. It looks like it's leading towards going towards this. But
for now pytorch defaults to color channels first. But that's okay. Because we can manipulate these
dimensions to what we need for whatever code that we're writing. And the three color channels is what
red, green and blue. So if you combine red, green and blue in some way, shape or form,
you get the different colors here that represent our image. And so if we have a look at our image
as a ray. Our image is in numerical format. Wonderful. So okay. We've got one way to do this for
one image. I think we start moving towards scaling this up to do it for every image in our data
folder. So let's just finish off this video by visualizing one more image. What do we get? Same
premise. The image is now as an array, different numerical values. We've got a delicious looking
pizza here of shave 512 512 with color channels last. And we've got the same thing up here. So
that is one way to become one with the data is to visualize different images, especially random
images. You could do the same thing visualizing different text samples that you're working with
or listening to different audio samples. It depends what domain you're working in. So now in the
next video, let's start working towards turning all of the images in here. Now that we visualize
some of them and become one with the data, we've seen that the shapes are varying in terms of
height and width. But they all look like they have three color channels because we have color images.
But now we want to write code to turn all of these images into pytorch tenses.
So let's start moving towards that. I'll see you in the next video.
Hello and welcome back. In the last video, we converted an image to a NumPy array.
And we saw how an image can be represented as an array. But what if we'd like to get this image
from our custom data set over here, pizza steak sushi into pytorch? Well, let's cover that in
this video. So I'm going to create a new heading here. And it's going to be transforming data.
And so what we'd like to do here is I've been hinting at the fact the whole time is we want
to get our data into tensor format, because that is the data type that pytorch accepts.
So let's write down here before we can use our image data with pytorch. Now this goes for images,
other vision data, it goes for text, it goes to audio, basically whatever kind of data set you're
working with, you need some way to turn it into tenses. So that's step number one. Turn your target
data into tenses. In our case, it's going to be a numerical representation of our images.
And number two is turn it into a torch dot utils dot data dot data set. So recall from a previous
video that we've used the data set to house all of our data in tensor format. And

```
then subsequently,
we've turned our data sets, our pytorch data sets into torch dot utils dot data dot
data loader.
And a data loader creates an iterable or a batched version of our data set. So for
short, we're going
to call these data set and data loader. Now, as I discussed previously, if we go to
the pytorch
documentation torch vision for torch vision, this is going to be quite similar for
torch audio torch
text, torch rec torch data eventually when it comes out of beta, there are
different ways to
create such data sets. So we can go into the data sets module, and then we can find
built-in data
sets, and then also base classes for custom data sets. But if we go into here,
image folder,
there's another parameter I'd like to show you, and this is going to be universal
across many of
your different data types is the transform parameter. Now, the transform parameter
is
a parameter we can use to pass in some transforms on our data. So when we load our
data sets from an
image folder, it performs a transform on those data samples that we've sent in here
as the target
data folder. Now, this is a lot more easier to understand through illustration,
rather than just
talking about it. So let's create a transform. And the main transform we're going
to be doing is
transforming our data, and we're turning it into tenses. So let's see what that
looks like. So we're
going to just going to re import all of the main libraries that we're going to use.
So from torch
utils dot data, let's import data loader. And we're going to import from torch
vision. I'm going to
import data sets. And I'm also going to import transforms. Beautiful. And I'm going
to create
```

Section 208: Main Topics

Key Topics:

- It's essentially turning your pytorch code into a Python script

▶ 📄 Click to view detailed content

```
another little heading here, this is going to be 3.1, transforming data with torch
vision dot
transform. So the main transform we're looking to here is turning out images from
JPEGs.
If we go into train, and then we go into any folder, we've got JPEG images.
And we want to turn these into tensor representation. So there's some pizza there.
```

We'll get out of this. Let's see what we can do. How about we create a transform here,
write a transform for image. And let's start off by calling it data transform.
And I'm going to show you how we can combine a few transforms together. If you want to
combine transforms together, you can use transforms dot compose. You can also use
an n dot sequential to combine transforms. But we're going to stick with transforms dot
compose for now. And it takes a list. And so let's just write out three transforms to begin with.
And then we can talk about them after we do so. So we want to resize our images
to 6464. Now, why might we do this? Well, do you recall in the last section computer vision,
we use the tiny VGG architecture. And what size were the images that the tiny VGG architecture took?
Well, we replicated the CNN website version or the CNN explainer website version, and they took
images of size 6464. So perhaps we want to leverage that computer vision model later on.
So we're going to resize our images to 6464. And then we're going to create another transform.
And so this is, I just want to highlight how transforms can help you manipulate your data in a
certain way. So if we wanted to flip the images, which is a form of data augmentation, in other
words, artificially increasing the diversity of our data set, we can flip the images randomly on
the horizontal. So transforms dot random horizontal flip. And I'm going to put a probability in here
of p equals 0.5. So that means 50% of the time, if an image goes through this transform pipeline,
it will get flipped on the horizontal axis. As I said, this makes a lot more sense when we
visualize it. So we're going to do that very shortly. And finally, we're going to turn the image into
a torch tensor. So we can do this with transforms dot to tensor. And now where might you find such
transforms? So this transform here says to tensor, if we have a look at the doc string,
we got convert a pill image, which is what we're working with right now, or a NumPy array to a
tensor. This transform does not support torch script. If you'd like to find out what that is,
I'd like to read the documentation for that. It's essentially turning your pytorch code into a
Python script. It converts a pill image or a NumPy array from height with color channels in the range
0 to 255, which is what our values are up here. They're from 0 to 255, red, green and blue,
to a torch float tensor of shape color channels height width in the range 0 to 1. So it will
take our tensor values here or our NumPy array values from 0 to 255 and convert them into a torch
tensor in the range 0 to 1. We're going to see this later on in action. But this is our first
transform. So we can pass data data through that. In fact, I'd encourage you to try that out.

```
See what happens when you pass in data transform. What happens when you pass it in
our image as a
ray? Image as a ray. Let's see what happens. Hey, oh, image should be pill image
got class NumPy
array. What if we just pass in our straight up image? So this is a pill image.
There we go.
Beautiful. So if we look at the shape of this, what do we get?
3 64 64. There's 64. And if what if we wanted to change this to 224, which is
another common value for
computer vision models to 24 to 24. Do you see how powerful this is? This little
transforms
module, the torch vision library will change that back to 64 64. And then if we
have a look at what
D type of our transform tensor is, we get torch float 32. Beautiful. So now we've
got a way to
transform our images into tensors. And so, but we're still only doing this with one
image.
How about we progress towards doing it for every image in our data folder here?
But before we do that, I'd like to visualize what this looks like. So in the next
video,
let's write some code to visualize what it looks like to transform multiple images
at a time.
And I think it'd be a good idea to compare the transform that we're doing to the
original image.
So I'll see you in the next video. Let's write some visualization code.
Let's now follow our data explorer's motto of visualizing our transformed images.
So we saw what it looks
like to pass one image through a data transform. And if we wanted to find more
documentation on
torch vision transforms, where could we go? There is a lot of these. So
transforming and augmenting
images, this is actually going to be your extra curriculum for this video. So
transforms are
common image transformations available in the transforms module. They can be
chained together
using compose, which is what we've already done. Beautiful. And so if you'd like to
go through all
of these, there's a whole bunch of different transforms that you can do, including
some data
augmentation transforms. And then if you'd like to see them visually, I'd encourage
you to check
out illustration of transforms. But let's write some code to explore our own
transform visually
```

Section 209: Main Topics

Key Topics:

- We want our images as pytorch tenses

▶ 📄 Click to view detailed content

first. So I'll leave this as a link. So I'm going up here, right here, transforms
help you get your images ready to be used with a model slash perform data
augmentation.
Wonderful. So we've got a way to turn images into tenses. That's what we want for
our model.
We want our images as pytorch tenses. The same goes for any other data type that
you're working
with. But now I'd just like to visualize what it looks like if we plot a number of
transformed
images. So we're going to make a function here that takes in some image paths, a
transform,
a number of images to transform at a time and a random seed here, because we're
going to harness
the power of randomness. And sometimes we want to set the seed. Sometimes we don't.
So we have
an image path list that we've created before, which is just all of the image paths
that we have
of our data set. So data, pizza, steak sushi. Now how about we select some random
image paths
and then take the image from that path, run it through our data transform, and then
compare the
original image of what it looks like and the transformed image and what that looks
like.
Let's give it a try, hey? So I'm going to write a doc string of what this does,
and then selects random images from a path of images and loads slash transforms
them,
then plots the original verse, the transformed version. So that's quite a long doc
string,
but that'll be enough. We can put in some stuff for the image paths, transforms,
and seed. We'll
just code this out. Let's go random seed, we'll create the seed. Maybe we do it if
seed, random seed.
Let's put that, and we'll set seed to equal none by default. That way we can, we'll
see if this works,
hey, if in doubt, coded out random image paths, and then we're going to go random
sample from the
image paths and the number of sample that we're going to do. So random sample is
going to, this will
be a list on which part in here that this is a list. So we're going to randomly
sample
k, which is going to be n. So three images from our image path list. And then we're
going to go for
image path, we're going to loop through the randomly sampled image parts. You know
how much I love
harnessing the power of randomness for visualization. So for image path in random
image paths, let's
open up that image using pill image dot open image path as f. And then we're going
to create a
figure and an axes. And we're going to create a subplot with my plot lib. So
subplots. And we
want it to create one row. So it goes n rows and calls. One row and n calls equals
two. And then
on the first or the zeroth axis, we're going to plot the original image. So in
show, we're just
going to pass it straight in f. And then if we want to go x zero, we're going to
set the title. So

```
set title, we're going to set it to be the original. So we'll create this as an f
string, original,
and then new line will create a size variable. And this is going to be f dot size.
So we're just
getting the size attribute from our file. So we'll keep going, and we'll turn off
the axes here.
So axis, and we're going to set that to false. Now let's transform on the first
axes plot. We're
going to transform and plot target image. This is so that our images are going to
be side by side,
the original and the transformed version. So there's one thing that we're going to
have to do. I'll
just, I'll code it out in a wrong way first. I think that'll be a good way to
illustrate what's
going on. f. So I'm just going to put a note here. Note, we will need to change
shape for
matplotlib, because we're going to come back here. Because what does this do? What
have we
noticed that our transform does? If we check the shape here, oh, excuse me, it
converts our image
to color channels first. Whereas matplotlib prefers color channels last. So just
keep that
in mind for when we're going forward. This code, I'm writing it, it will error on
purpose. So
transformed image. And then we're going to go axe one as well. We're going to set
the title,
which is going to be transformed. And then we'll create a new line and we'll say
size is going to be
transformed image dot shape. Or probably a bit of, yeah, we could probably go shape
here. And then
finally, we're going to go axe one, we're going to turn the axis, we're going to
set that to false.
You can also set it to off. So you could write false, or you could write off, you
might see that
different versions of that somewhere. And I'm going to write a super title here,
which we'll see what
this looks like class is going to be image path. So we're getting the target image
path. And we're
just going to get the attribute or the parent attribute, and then the stem
attribute from that,
just like we did before, to get the class name. And then I'm going to set this to a
larger font
size, so that we make some nice looking plots, right? If we're going to visualize
our data,
we might as well make our plots visually appealing. So let's plot some transformed
data or transformed
images. So image paths, we're going to set this to image part list, which is just
the variable we
```

Section 210: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
have down below, which is the part list, a list containing all of our image paths.
Our transform,
we're going to set our transform to be equal to our data transform. So this just
means that if
we pass the transform in, our image is going to go through that transform, and then
go through all
of these is going to be resized, it's going to be randomly horizontally flipped,
and it's going to
be converted to a tensor. And then so we're going to set that data transfer there
or data transform,
sorry, and is going to be three. So we plot three images, and we'll set the seed to
42 to begin with.
Let's see if this works. Oh, what did we get wrong? We have invalid shape. As I
said, I love seeing
this error, because we have seen this error many times, and we know what to do with
it. We know that
we have to rearrange the shapes of our data in some way, shape or form. Wow, I said
shape a lot
there. That's all right. Let's go here, permute. This is what we have to do. We
have to permute,
we have to swap the order of the axes. So right now, our color channels is first.
So we have to
bring this color channel axis or dimension to the end. So we need to shuffle these
across. So 64
into here, 64 into here, and three on the end. We need to, in other words, turn it
from color
channels first to color channels last. So we can do that by permuting it to have
the first
axis come now in the zero dimension spot. And then number two was going to be in
the first
dimension spot. And then number zero was going to be at the back end. So this is
essentially going
from C H W, and we're just changing the order to be H W C. So the exact same data
is going to be
within that tensor. We're just changing the order of the dimensions. Let's see if
this works.
Look at that. Oh, I love seeing some manipulated data. We have a class of pizza and
the original
image is there, and it's 512 by 512. But then we've resized it using our transform.
Notice that
it's a lot more pixelated now, but that makes sense because it's only 64 64 pixels.
Now, why
might we do such a thing? Well, one, if is this image still look like that? Well,
to me, it still
does. But the most important thing will be does it look like that to our model?
Does it still look
like the original to our model? Now 64 by 64, there is less information encoded in
this image.
So our model will be able to compute faster on images of this size. However, we may
lose
some performance because not as much information is encoded as the original image.
Again, the size
of an image is something that you can control. You can set it to be a hyper
parameter. You can
```

```
tune the size to see if it improves your model. But I've just decided to go 60 64
64 3 in line
with the CNN explainer website. So a little hint, we're going to be re replicating
this model that
we've done before. Now you notice that our images are now the same size 64 64 3 as
what the CNN
explainer model uses. So that's where I've got that from. But again, you could
change this to
size to whatever you want. And we see, oh, we've got a stake image here. And you
notice that our
image has been flipped on the horizontal. So the horizontal access, our image has
just been flipped
same with this one here. So this is the power of torch transforms. Now there are a
lot more
transforms, as I said, you can go through them here to have a look at what's going
on. Illustrations
of transforms is a great place. So there's resize, there's center crop, you can
crop your
images, you can crop five different locations, you can do grayscale, you can change
the color,
a whole bunch of different things. I'd encourage you to check this out. That's your
extra curriculum
for this video. But now that we've visualized a transform, this is what I hinted at
before that
we're going to use this transform for when we load all of our images in, using into
a torch
data set. So I just wanted to make sure that they had been visualized first. We're
going to use our
data transform in the next video when we load all of our data using a torch vision
dot data sets
helper function. So let's give that a go. I'll see you in the next video.
Have a look at that beautiful plot. We've got some original images and some
transformed
images. And the beautiful thing about our transformed images is that they're in
tensor format,
which is what we need for our model. That's what we've been slowly working towards.
We've got a data set. And now we've got a way to turn it into tensors ready for a
model. So
let's just visualize what another, I'll turn the seed off here so we can look at
some more random
images. There we go. Okay, so we've got stake pixelated because we're downsizing
64, 64, 3.
Same thing for this one. And it's been flipped on the horizontal. And then same
thing for this
pizza image and we'll do one more to finish off. Wonderful. So that is the premise
of transforms
turning our images into tensors and also manipulating those images if we want to.
So let's get rid of this. I'm going to make another heading. We're up to section or
part four now.
```

Section 211: Main Topics

Key Topics:

- PyTorch likes to use target, I like to use label, but that's okay
- So this transform is going to run our images, whatever images are loaded from these folders, through this transform that we've created here, it's going to resize them, randomly flip them on the horizontal, and then turn them into tenses, which is exactly how we want them for our PyTorch models
- This is one of the benefits of using a pytorch prebuilt data loader, is that or data set loader is that it comes with a fair few attributes

▶ 📄 Click to view detailed content

```
And this is going to be option one. So loading image data using image folder. And
now I'm going
to turn that into markdown. And so let's go torch vision data sets. So recall how
each one of the
torch vision domain libraries has its own data sets module that has built in
functions for
helping you load data. In this case, we have an image folder. And there's a few
others here if
you'd like to look into those. But an image folder, this class is going to help us
load in data that
is in this format, the generic image classification format. So this is a prebuilt
data sets function.
Just like there's prebuilt data sets, we can use prebuilt data set functions. Now
option two
later on, this is a spoiler, is we're going to create our own custom version of a
data set loader.
But we'll see that in a later video. So let's see how we can use image folder to
load all of our
custom data, our custom images into tensors. So this is where the transform is
going to come in
helpful. So let's write here, we can load image classification data using, let's
write this,
let's write the full path name, torch vision dot data sets dot image folder. Put
that in there,
beautiful. And so let's just start it out, use image folder to create data sets.
Now in a previous
video, I hinted at the fact that we can pass a transform to our image folder class.
That's going
to be right here. So let's see what that looks like in practice. So from torch
vision, I'm going
to import data sets, because that's where the image folder module lives. And then
we can go train
data equals data sets dot image folder. And we're going to pass in the root, which
is our train
der, because we're going to do it for the training directory first. And then we're
going to pass
in a transform, which is going to be equal to our data transform. And then we're
going to pass in
a target transform, but we're going to leave this as none, which is the default, I
believe,
```

we go up to here. Yeah, target transform is optional. So what this means is this is going to be a
transform for the data. And this is going to be a transform for the label slash target.
PyTorch likes to use target, I like to use label, but that's okay. So this means that we don't need
a target transform, because our labels are going to be inferred by the target directory where the
images live. So our pizza images are in this directory, and they're going to have pizza as the label,
because our data set is in standard image classification format. Now, if your data set wasn't in a
standard image classification format, you might use a different data loader here. A lot of them
will have a transform for the data. So this transform is going to run our images, whatever images are
loaded from these folders, through this transform that we've created here, it's going to resize them,
randomly flip them on the horizontal, and then turn them into tenses, which is exactly how we
want them for our PyTorch models. And if we wanted to transform the labels in some way, shape or form,
we could pass in a target transform here. But in our case, we don't need to transform the labels.
So let's now do the same thing for the test data. And so that's why I wanted to visualize
our transforms in the previous videos, because otherwise we're just passing them in as a transform.
So really, what's going to happen behind the scenes is all of our images are going to go
through these steps. And so that's what they're going to look like when we turn them into a data
set. So let's create the test data here or the test data set. The transform, we're going to
transform the test data set in the same way we've transformed our training data set. And we're
just going to leave that like that. So let's now print out what our data sets look like,
train data, and test data. Beautiful. So we have a data set, a torch data set,
which is an image folder. And we have number of data points. This is going to be for the training
data set. We have 225. So that means about 75 images per class. And we have the root location,
which is the folder we've loaded them in from, which is our training directory. We've set these
two up before, trained and tester. And then we have a transform here, which is a standard transform,
a resize, followed by random horizontal flip, followed by two tensor. Then we've got basically
the same output here for our test directory, except we have less samples there. So let's get a few
little attributes from the image folder. This is one of the benefits of using a pytorch prebuilt
data loader, is that or data set loader is that it comes with a fair few attributes. So we could
go to the documentation, find this out from in here, inherits from data set folder, keep digging

```
into there, or we could just come straight into Google collab. Let's go get class
names as a list.
Can we go train data dot and then press tab? Beautiful. So we've got a fair few
things here
that are attributes. Let's have a look at classes. This is going to give us a list
of the class names,
class names. This is very helpful later on. So we've got pizza steak sushi. We're
trying to
do everything with code here. So if we have this attribute of train data dot
classes,
```

Section 212: Main Topics

Key Topics:

- And then of course, if you'd like to explore more attributes, you can go train data dot, and then we've got a few other things, functions, images, loader, samples, targets
- So that means that our data, our custom data set, this is so exciting, is now compatible to be used with a pytorch model

▶ 📄 Click to view detailed content

```
we can use this list later on for when we plot images straight from our data set,
or make predictions on them and we want to label them. You can also get class names
as a dictionary,
map to their integer index, that is, so we can go train data dot and press tab.
We've got class
to ID X. Let's see what this looks like. Class decked. Wonderful. So then we've got
our string
class names mapped to their integer. So we've got pizza is zero, steak is one,
sushi is two. Now,
this is where the target transform would come into play. If you wanted to transform
those
these labels here in some way, shape or form, you could pass a transform into here.
And then if we keep going, let's check the lengths of what's going on. Check the
lengths
of our data set. So we've seen this before, but this is going to just give us how
many samples
that we have length, train data, length, test data, beautiful. And then of course,
if you'd like
to explore more attributes, you can go train data dot, and then we've got a few
other things,
functions, images, loader, samples, targets. If you wanted to just see the images,
you can go dot
samples. If you wanted to see just the labels, you can go dot targets. This is
going to be all
of our labels. Look at that. And I believe they're going to be an order. So we're
```

going to have
zero, zero, zero, one, one, one, two, two, and then if we wanted to have a look, let's say we have a
look at the first sample, hey, we have data, pizza, steak sushi, train, pizza. There's the image path,
and it's a label zero for pizza. Wonderful. So now we've done that. How about we, we've been
visualizing this whole time. So let's keep up that trend. And let's visualize a sample and a label
from the train data data set. So in this video, we've used image folder to load our images
into tenses. And because our data is already in standard image classification format,
we can use one of torch vision dot data sets prebuilt functions.
So let's do some more visualization in the next video. I'll see you there.
Welcome back. In the last video, we used data sets dot image folder to turn all of our
image data into tenses. And we did that with the help of our data transform, which is a little
pipeline up here to take in some data, or specifically an image, resize it to a value that we've set in
our k6464 randomly flip it along the horizontal. We don't necessarily need this, but I've just put
that in there to indicate what happens when you pass an image through a transforms pipeline.
And then most importantly, we've turned our images into a torch tensor. So that means that our data,
our custom data set, this is so exciting, is now compatible to be used with a pytorch model.
So let's keep pushing forward. We're not finished yet. We're going to visualize some samples
from the train data data set. So let's, how can we do this? Let's get, we can index on the train data
data set to get a single image and a label. So if we go, can we do train data zero? What does that
give us? Okay, so this is going to give us an image tensor. And it's associated label. In this
case, it's an image of pizza, because why it's associated label is pizza. So let's take the zero
zero. So this is going to be our image. And the label is going to be train data zero. And we're
just going to get the first index item there, which is going to be one. And then if we have a look
at them separately, image and label, beautiful. So now one of our target images is in tensor format,
exactly how we want it. And it's label is in numeric format as well, which is also exactly how
we want it. And then if we wanted to convert this back to a non label, we can go class names
and index on that. And we see pizza. And I mean, non label is in non numeric, we can get it back
to string format, which is human understandable. We can just index on class names. So let's print
out some information about what's going on here. Print F, we're going to go image tensor.
I love F strings if you haven't noticed yet. Image tensor. And we're going to set in

```
new line, we're going to pass it in our image, which is just the image that we've
got here.
Then we'll print in some more information about that. This is still all becoming
one with the
data right where we're slowly finding out information about our data set so that if
errors arise later
on, we can go, hmm, our image or we're getting a shape error. And I know our images
are of this
shape or we're getting a data type error, which is why I've got the dot D type
here. And that
might be why we're getting a data type issue. So let's do one more with the image
label,
label, oh, well, actually, we'll do one more. We'll do print, we'll get the label
data type as well.
Label, this will be important to take note of later on. Type, as I said, three big
issues.
Shape mismatch, device mismatch, and data type mismatch. Can we get the type of our
label?
Beautiful. So we've got our image tensor and we've got its shape. It's of torch
size 36464.
That's exactly how we want it. The data type is torch float 32, which is the
default data type
```

Section 213: Main Topics

Key Topics:

- in PyTorch
- So we're still adhering to our PyTorch workflow here
- So if we tried to load 100,000 images into that whilst also computing on them with a PyTorch model, potentially we're going to run out of memory and run into issues

▶ 📄 Click to view detailed content

```
in PyTorch. Our image label is zero and the label data type is of integer. So let's
try and plot
this and see what it looks like, hey, using matplotlib. So first of all, what do we
have to do? Well,
we have to rearrange the order of dimensions. In other words, matplotlib likes
color channels
last. So let's see what looks this looks like. We'll go image per mute. We've done
this before,
image.permute 120 means we're reordering the dimensions. Zero would usually be
here,
except that we've taken the zero dimension, the color channels and put it on the
end
and shuffled the other two forward. So let's now print out different shapes. I love
printing
```

out the change in shapes. It helps me really understand what's going on. Because sometimes
I look at a line like this and it doesn't really help me. But if I print out something of what
the shapes were originally and what they changed to, well, hey, that's a big help. That's what
Jupiter notebooks are all about, right? So this is going to be color channels first, height,
width. And depending on what data you're using, if you're not using images, if you're using text,
still knowing the shape of your data is a very good thing. We're going to go image per mute.shape
and this should be everything going right is height with color channels on the end here.
And we're just going to plot the image. You can never get enough plotting practice.
Plot the image. You're going to go PLT dot figure, we'll pass in fig size equals 10, 7.
And then we're going to PLT dot in show. We'll pass in the permuted image,
image underscore permutes, and then we'll turn off the axes. And we will set the title to be
class names. And we're going to index on the label, just as we did before. And we're going to set
the font size equal to 14. So it's nice and big. Here we go. Beautiful. There is our image of pizza.
It is very pixelated because we're going from about 512 as the original size 512 by 512 to 64,
64. I would encourage you to try this out. Potentially, you could use a different image here. So we've
indexed on sample zero. Maybe you want to change this to just be a random image and go through these
steps here. And then if you'd like to see different transforms, I'd also encourage you to try
changing this out, our transform pipeline here, maybe increase the size and see what it looks
like. And if you're feeling really adventurous, you can go into torch vision and look at the
transforms library here and then try one of these and see what it does to our images.
But we're going to keep pushing forward. We are going to look at another way. Or actually,
I think for completeness, let's now turn, we've got a data set. We want to, we wrote up here before
that we wanted to turn our images into a data set, and then subsequently a torch utils data
data loader. So we've done this before, by batching our images, or batching our data that we've
been working with. So I'd encourage you to give this a shot yourself. Try to go through the next
video and create a train data loader using our train data, wherever that is train data,
and a test data loader using our test data. So give that a shot and we'll do it together in the
next video. We'll turn our data sets into data loaders. Welcome back. How'd you go? In the last
video, I issued you the challenge to turn our data sets into data loaders. So let's do that
together now. I hope you gave it a shot. That's the best way to practice. So turn

```
loaded images
into data loaders. So we're still adhering to our PyTorch workflow here. We've got
a custom
data set. We found a way to turn it into tenses in the form of data sets. And now
we're going to
turn it into a data loader. So we can turn our data sets into iterables or batchify
our data.
So let's write down here, a data loader is going to help us turn our data sets into
iterables.
And we can customize the batch size, write this down. So our model can see batch
size
images at a time. So this is very important. As we touched on in the last section
computer vision,
we create a batch size because if we had 100,000 images, chances are if they were
all in one data
set, there's 100,000 images in the food 101 data set. We're only working with about
200.
If we try to load all 100,000 in one hit, chances are our hardware may run out of
memory. And so
that's why we matchify our images. So if we have a look at this, NVIDIA SMI, our
GPU only has 16
gigabytes. I'm using a Tesla T4 right now, well, has about 15 gigabytes of memory.
So if we tried
to load 100,000 images into that whilst also computing on them with a PyTorch
model,
potentially we're going to run out of memory and run into issues. So instead, we
can turn them
into a data loader so that our model looks at 32 images at a time and can leverage
all of the
memory that it has rather than running out of memory. So let's turn our train and
test data sets
into data loaders, turn train and test data sets into data loaders. Now, this is
not just for image
data. This is for all kinds of data in PyTorch. Images, text, audio, you name it.
So import data
loader, then we're going to create a train data loader. We're going to set it equal
to data loader.
```

Section 214: Main Topics

Key Topics:

- If you're running it on dedicated deep learning hardware, you may even have even more, right
- Now, of course, this would change if we set, oh, we didn't even set this to the batch size parameter batch size

▶ 📄 Click to view detailed content

We're going to pass in a data set. So let's set this to train data. Let's set the batch size.
What should we set the batch size to? I'm going to come up here and set a laser capital variable.
I'm going to use 32 because 32 is a good batch size. So we'll go 32 or actually,
let's start small. Let's just start with a batch size of one and see what happens.
Batch size one, number of workers. So this parameter is going to be, this is an important one. I'm going
to, I potentially have covered it before, but I'm going to introduce it again. Is this going to be
how many cores or how many CPU cores that is used to load your data? So the higher the better usually
and you can set this via OS CPU count, which will count how many CPUs your compute hardware has.
So I'll just show you how this works. Import OS and this is a Python OS module. We can do
CPU count to find out how many CPUs our Google Colab instance has. Mine has two,
your number may vary, but I believe most Colab instances have two CPUs. If you're running this on
your local machine, you may have more. If you're running it on dedicated deep learning hardware,
you may even have even more, right? So generally, if you set this to one, it will use one CPU core,
but if you set it to OS dot CPU count, it will use as many as possible. So we're just going to
leave this as one right now. You can customize this to however you want. And I'm going to shuffle
the training data because I don't want my model to recognize any order in the training data. So I'm
going to mix it up. And then I'm going to create the test data loader. Data set equals test data.
And batch size equals one, num workers, I'm going to set this to equal one as well. Again,
you can customize each of these, their hyper parameters to whatever you want. Number of workers
generally the more the better. And then I'm going to set shuffle equals false for the test data so
that if we want to evaluate our models later on, our test data set is always in the same order.
So now let's have a look at train data loader, see what happens. And test data loader.
Wonderful. So we get two instances of torch utils dot data dot data loader. And now we can
see if we can visualize something from the train data loader, as well as the test data loader.
I actually maybe we just visualize something from one of them. So we're not just double
handling everything. We get a length here. Wonderful. Because we're using a batch size of one,
our lengths of our data loaders are the same as our data sets. Now, of course, this would change
if we set, oh, we didn't even set this to the batch size parameter batch size. Let's come down
here and do the same here batch size. So we'll watch this change. If we wanted to look at 32
images at a time, we definitely could do that. So now we have eight batches,

```
because 22, 225
divided by 32 equals roughly eight. And then 75 divided by 32 also equals roughly
three. And
remember, these numbers are going to be rounded if there are some overlaps. So
let's get rid of,
we'll change this back to one. And we'll keep that there. We'll get rid of these
two.
And let's see what it looks like to plot an image from our data loader. Or at least
have a look at it.
Check out the shapes. That's probably the most important point at this time. We've
already
plotted in our things. So let's iterate through our train data loader. And we'll
grab the next one.
We'll grab the image and the label. And we're going to print out here. So batch
size will now be one.
You can change the batch size if you like. This is just again, another way of
getting familiar
with the shapes of our data. So image shape. Let's go image dot shape. And we're
going to
write down here. This shape is going to be batch size. This is what our data loader
is going to
add to our images is going to add a batch dimension, color channels, height, width.
And then print.
Let's check out that label shape. Same thing with the labels. It's going to add a
batch
dimension. Label. And let's see what happens. Oh, we forgot the end of the bracket.
Beautiful.
So we've got image shape. Our label shape is only one because we have a batch size
of one.
And so now we've got batch size one, color channels three, height, width. And if we
change this to
32, what do you think's going to happen? We get a batch size of 32, still three
color channels,
still 64, still 64. And now we have 32 labels. So that means within each batch, we
have 32
images. And we have 32 labels. We could use this with a model. I'm going to change
this back to one.
And I think we've covered enough in terms of loading our data sets. How cool is
this?
We've come a long way. We've downloaded a custom data set. We've loaded it into a
data set using
image folder turned it into tenses using our data transform and now batchified our
custom data set
in data loaders. We've used these with models before. So if you wanted to, you
could go right
ahead and build a convolutional neural network to try and find patterns in our
image tenses.
But in the next video, let's pretend we didn't have this data loader,
```

Section 215: Main Topics

Key Topics:

- So the whole goal of this video is to start writing a function or a class that's capable of loading data from here into Tensor format, capable of being used with the PyTorch's data loader class, like we've done here
- But at the base level of PyTorch is torchutils
- And another pro is that you're not limited to PyTorch pre-built data set functions

▶ 📄 Click to view detailed content

```
this image folder class available to us. How could we load our image data set so
that it's
compatible? Like our image data set here, how could we replicate this image folder
class?
So that we could use it with a data loader. Because data load is part of torch
utils.data,
you're going to see these everywhere. Let's pretend we didn't have the torch
vision.data sets
image folder helper function. And we'll see in the next video, how we can replicate
that functionality.
I'll see you there. Welcome back. So over the past few videos, we've been working
out how to get
how to get our data from our data folder, pizza, steak, and sushi. We've got images
of different
food data here. And we're trying to get it into Tensor format. So we've seen how to
do that
with an existing data loader helper function or data set function in image folder.
However,
what if image folder didn't exist? And we need to write our own custom data loading
function.
Now the premise of this is although it does exist, it's going to be good practice
because you might
come across a case where you're trying to use a data set where a prebuilt function
doesn't exist.
So let's replicate the functionality of image folder by creating our own data
loading class.
So we want a few things. We want to be able to get the class names as a list from
our loaded data.
And we want to be able to get our class names as a dictionary as well. So the whole
goal of this
video is to start writing a function or a class that's capable of loading data from
here into
Tensor format, capable of being used with the PyTorch's data loader class, like
we've done here. So we
want to create a data set. Let's start it off. We're going to create another
heading here. This is
going to be number five, option two, loading image data with a custom data set. So
we want a few
functionality steps here. Number one is one, two, be able to load images from file
to one,
two, be able to get class names from the data set, and three, one, two, be able to
get classes
as dictionary from the data set. And so let's briefly discuss the pros and cons of
creating
your own custom data set. We saw option one was to use a pre-existing data set
```

loader helping
function from torch vision. And it's going to be quite similar if we go torch
vision data sets.
Quite similar if you're using other domain libraries here, there we're going to be
data
loading utilities. But at the base level of PyTorch is torchutils.data.dataset. Now
this is
the base data set class. So we want to build on top of this to create our own image
folder loading
class. So what are the pros and cons of creating your own custom data set? Well,
let's discuss some
pros. So one pro would be you can create a data set out of almost anything as long
as you write
the right code to load it in. And another pro is that you're not limited to PyTorch
pre-built
data set functions. A couple of cons would be that even though this is to point
number one.
So even though you could create a data set out of almost anything, it doesn't mean
that it will
automatically work. It will work. And of course, you can verify this through
extensive testing,
seeing if your model actually works, if it actually loads data in the way that you
want it. And another
con is that using a custom data set requires us to write more code. So often
results in us
writing more code, which could be prone to errors or performance issues. So
typically if
something makes it into the PyTorch standard library or the PyTorch domain
libraries,
if functionality makes it into here, it's generally been tested many, many times.
And it can kind of
be verified that it works quite well with, or if you do use it, it works quite
well. Whereas if
we write our own code, sure, we can test it ourselves, but it hasn't got the
robustness to begin with,
that is, we could fix it over time, as something that's included in say the PyTorch
standard library.
Nonetheless, it's important to be aware of how we could create such a custom data
set.
So let's import a few things that we're going to use. We'll import OS, because
we're going to be
working with Python's file system over here. We're going to import path lib,
because we're going to
be working with file paths. We'll import torch, we don't need to again, but I'm
just doing this
for completeness. We're going to import image from pill, the image class, because
we want to be
opening images. I'm going to import from torch utils dot data. I'm going to import
data set,
which is the base data set. And as I said over here, we can go to data sets, click
on torch utils
data dot data set. This is an abstract class representing a data set. And you'll
find that this
data set links to itself. So this is the base data set class. Many of the data sets
in PyTorch,
the prebuilt functions, subclass this. So this is what we're going to be doing it.
And as a few notes here, all subclasses should overwrite get item. And you should

```
optionally
overwrite land. These two methods, we're going to see this in a future video. For
now, we're just
we're just setting the scene here. So from torch vision, we're going to import
transforms, because
we want to not only import our images, but we want to transform them into tenses.
And from the
```

Section 216: Main Topics

Key Topics:

- Of course, you could look this up in the Python documentation

▶ 📄 Click to view detailed content

```
Python's typing module, I'm going to import tuple dict and list. So we can put type
hints
when we create our class and loading functions. Wonderful. So this is our instance
of torch vision
dot data sets image folder, torch vision dot data sets dot image folder. Let's have
a look
at the train data. So we want to write a function that can replicate getting the
classes from a
particular directory, and also turning them into an index or dictionary that is. So
let's build
a helper function to replicate this functionality here. In other words, I'd like to
write a helper
function that if we pass it in a file path, such as pizza steak sushi or this data
folder,
it's going to go in here. And it's going to return the class names as a list. And
it's also going
to turn them into a dictionary, because it's going to be helpful for later on when
we'd like to access
the classes and the class to ID X. And if we really want to completely recreate
image folder,
well, image folder has this functionality. So we'd like that too. So this is just a
little high level
overview of what we're going to be doing. I might link in here that we're going to
subclass this.
So all custom data sets in pie torch, often subclass this. So here's what we're
going to be doing.
Over the next few videos, we want to be able to load images from a file. Now you
could replace
images with whatever data that you're working with the same premise will be here.
You want to be
able to get the class names from the data set and want to be able to get classes as
a dictionary
from the data set. So we're going to map our samples, our image samples to that
class name
```

by just passing a file path to a function that we're about to write. And some pros and cons of
creating a custom data set. We've been through that. Let's in the next video, start coding up a
helper function to retrieve these two things from our target directory. In the last video,
we discussed the exciting concept of creating a custom data set. And we wrote down a few things
that we want to get. We discussed some pros and cons. And we learned that many custom data sets
inherit from torch dot utils dot data data set. So that's what we'll be doing later on. In this
video, let's focus on writing a helper function to recreate this functionality. So I'm going to
title this 5.1, creating a helper function to get class names. I'm going to turn this into
markdown. And if I go into here, so we want to function to let's write down some steps and then
we'll code it out. So we'll get the class names, we're going to use OS dot scanner. So it's going
to scanner directory to traverse a target directory. And ideally, the directory is in standard image
classification format. So just like the image folder class, our custom data class is going to
require our data already be formatted. In the standard image classification format, such as
train and test for training and test images, and then images for a particular class are in a
particular directory. So let's keep going. And number two, what else do we want it to do? We want
it to raise an error if the class names aren't found. So if this happens, there might be,
we want this to enter the fact that there might be something wrong with the directory structure.
And number three, we also want to turn the class names into our dict and a list and return them.
Beautiful. So let's get started. Let's set up the path directory
for the target directory. So our target directory is going to be what the directory we want to load
directory, if I could spell, we want to load our data from, let's start with the training
der, just for an example. So target directory, what do we get? So we're just going to use the
training folder as an example to begin with. And we'll go print target der, we'll put in the target
directory, just want to exemplify what we're doing. And then we're going to get the class names
from the target directory. So I'll show you the functionality of our scanner. Of course,
you could look this up in the Python documentation. So class names found, let's set this to be sorted.
And then we'll get the entry name, entry dot name for entry in list. So we're going to get OS list
scanner of the image path slash target directory. Let's see what happens when we do this.
Target directory have we got the right brackets here.
Now, is this going to work? Let's find out. Oh, image path slash target directory.

```
What do we get wrong? Oh, we don't need the image path there. Let's put, let's just
put target
directory there. There we go. Beautiful. So we set up our target directory as been
the training
to. And so if we just go, let's just do list. What happens if we just run this
function here?
Oh, a scanner. Yeah, so there we go. So we have three directory entries. So this is
where we're
getting entry dot name for everything in the training directory. So if we look in
the training
directory, what do we have train? And we have one entry for pizza, one entry for
sushi, one entry
for steak. Wonderful. So now we have a way to get a list of class names. And we
could quite easily
turn this into a dictionary, couldn't we? Which is exactly what we want to do. We
want to recreate
```

Section 217: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
this, which we've done. And we want to recreate this, which is also done. So now
let's take this
functionality here. And let's turn that into a function. All right, what can we do?
What do we
call this? I'm going to call this def fine classes. And I'm going to say that it
takes in a directory
which is a string. And it's going to return. This is where I imported typing from
Python type and
imported tuple. And I'm going to return a list, which is a list of strings and a
dictionary,
which is strings map to integers. Beautiful. So let's keep going. We want to, we
want this
function to return given a target directory, we want it to return these two things.
So we've seen
how we can get a list of the directories in a target directory by using OS scanner.
So let's
write finds the classes are the class folder names in a target directory.
Beautiful. And we know
that it's going to return a list and a dictionary. So let's do step number one, we
want to get the
class names by scanning the target directory. We'll go classes, just we're going to
replicate the
functionality we've done about, but for any given directory here. So classes equals
sorted entry
dot name for entry in OS scanner. And we're going to pass at the target directory.
If entry dot is
dirt, we're just going to make sure it's a directory as well. And so if we just
```

return classes and see
what happens. So find classes, let's pass it in our target directory, which is our training directory.
What do we get? Beautiful. So we need to also return class to ID X. So let's keep going. So number
two is let's go raise an error. If class names could not be found. So if not classes, let's say
raise file, we're going to raise a file not found error. And then let's just write in here F
couldn't find any classes in directory. So we're just writing some error checking code here.
So if we can't find a class list within our target directory, we're going to raise this
error and say couldn't find any classes in directory, please check file structure. And there's another
checkup here that's going to help us as well to check if the entry is a directory. So finally,
let's do number three. What do we want to do? So we want to create a dictionary of index labels.
So computers, why do we do this? Well, computers prefer numbers rather than strings as labels. So we
can do this, we've already got a list of classes. So let's just create class to ID X equals class
name, I for I class name in enumerate classes. Let's see what this looks like.
So we go class names, and then class to ID X, or we can just return it actually. Do we spell
enumerate role? Yes, we did. So what this is going to do is going to map a class name to an integer
or to I for I class name in enumerate classes. So it's going to go through this, and it's going
to go for I. So the first one zero is going to be pizza. Ideally, one will be steak,
two will be sushi. Let's see how this goes. Beautiful. Look at that. We've just replicated
the functionality of image folder. So now we can use this helper function in our own custom
data set, find classes to traverse through a target directory, such as train, we could do the
same for test if we wanted to to. And that way, we've got a list of classes. And we've also got
a dictionary mapping those classes to integers. So now let's in the next video move towards sub
classing torch utils dot data dot data set. And we're going to fully replicate image folder. So I'll see you there.
In the last video, we wrote a great helper function called find classes that takes in a target
directory and returns a list of classes and a dictionary mapping those class names to an integer.
So let's move forward. And this time, we're going to create a custom data set. To replicate
image folder. Now we don't necessarily have to do this, right, because image folder already exists.
And if something already exists in the pie torch library, chances are it's going to be tested well,
it's going to work efficiently. And we should use it if we can. But if we needed some custom
functionality, we can always build up our own custom data set by sub classing torch

```
dot utils
dot data data set. Or if a pre built data set function didn't exist, well, we're
probably going
to want to subclass torch utils data dot data set anyway. And if we go into the
documentation here,
there's a few things that we need to keep in mind when we're creating our own
custom data set.
All data sets that represent a map from keys to data samples. So that's what we
want to do.
We want to map keys, in other words, targets or labels to data samples, which in
our case are
food images. So we should subclass this class here. Now to note, all subclasses
should overwrite
get item. So get item is a method in Python, which is going to get an item or get a
sample,
supporting fetching a data sample for a given key. So for example, if we wanted to
get sample
number 100, this is what get item should support and should return us sample number
100.
And subclasses could also optionally override land, which is the length of a data
set. So return
the size of the data set by many sampler implementations and the default options of
data
```

Section 218: Main Topics

Key Topics:

- Now, of course, these attributes will differ depending on your data set
- This is so exciting, because when you work with prebuilt data sets, it's pretty cool in machine learning

▶ 📄 Click to view detailed content

```
loader, because we want to use this custom data set with data loader later on. So
we should keep
this in mind when we're building our own custom subclasses of torch utils data data
set. Let's see
this hands on, we're going to break it down. It's going to be a fair bit of code,
but that's all right.
Nothing that we can't handle. So to create our own custom data set, we want to
number one,
first things first is we're going to subclass subclass torch dot utils dot data dot
data set.
Two, what do we want to do? We want to init our subclass with target directory. So
the directory
we'd like to get data from, as well as a transform, if we'd like to transform our
data. So just like
when we used image folder, we could pass a transform to our data set, so that we
```

could transform the
data that we were loading. We want to do the same thing. And we want to create
several attributes.
Let's write them down here. We want paths, which will be the parts of our images.
What else do
we want? We want transform, which will be the transform we'd like to use. We want
classes,
which is going to be a list of the target classes. And we want class to ID X, which
is going to be
a dict of the target classes, mapped to integer labels. Now, of course, these
attributes will
differ depending on your data set. But we're replicating image folder here. So
these are just
some of the things that we've seen that come with image folder. But regardless of
what data set
you're working with, there are probably some things that you want to cross them
universal.
You probably want all the paths of where your data is coming from, the transforms
you'd like to
perform on your data, what classes you're working with, and a map of those classes
to an index.
So let's keep pushing forward. We want to create a function to load images, because
after all,
we want to open some images. So this function will open an image. Number five, we
want to
overwrite the LAN method to return the length of our data set. So just like it said
in the documentation,
if you subclass using torch.utils.data, the data set, you should overwrite get
item,
and you should optionally overwrite LAN. So we're going to, instead of optionally,
we are going to
overwrite length. And number six, we want to overwrite the get item method to
return a given sample
when passed an index. Excellent. So we've got a fair few steps here. But if they
don't make
sense now, it's okay. Let's code it out. Remember our motto, if and doubt, code it
out. And if
and doubt, run the code. So we're going to write a custom data set. This is so
exciting, because
when you work with prebuilt data sets, it's pretty cool in machine learning. But
when you can write
code to create your own data sets, and that's, well, that's magic. So number one is
we're going to,
or number zero is we're going to import torch utils data set, we don't have to
rewrite this,
we've already imported it, but we're going to do it anyway for completeness. Now
step number one
is to subclass it subclass torch utils data, the data set. So just like when we
built a model,
we're going to subclass and in module, but in this time, we're going to call us our
class
image folder custom. And we're going to inherit from data set. This means that all
the functionality
that's contained within torch utils data data set, we're going to get for our own
custom class.
Number two, let's initialize. So we're going to initialize
our custom data set. And there's a few things that we'd like, and into our subclass

```
with the
target directory, the directory we'd like to get data from, as well as the
transform if we'd
like to transform our data. So let's write a knit function, a knit, and we're going
to go self,
target, and target is going to be a string. And we're going to set a transform
here,
we'll set it equal to none. Beautiful. So this way we can pass in a target
directory of images
that we'd like to load. And we can also pass in a transform, just similar to the
transforms that
we've created previously. So now we're up to number three, which is create several
attributes. So
let's see what this looks like, create class attributes. So we'll get all of the
image paths.
So we can do this just like we've done before, self paths equals list, path lib dot
path,
because what's our target directory going to be? Well, I'll give you a spoiler
alert,
it's going to be a path like the test directory, or it's going to be the train
directory.
Because we're going to use this once for our test directory and our train
directory,
just like we use the original image folder. So we're going to go through the target
directory
and find out all of the paths. So this is getting all of the image paths that
support
or that follow the file name convention of star star dot jpg. So if we have a look
at this,
we passed in the test folder. So test is the folder star would mean any of these
123 pizza
steak sushi, that's the first star, then slash would go into the pizza directory.
The star here
would mean any of the file combinations here that end in dot jpg. So this is
getting us a list of
all of the image paths within a target directory. In other words, within the test
directory and
within the train directory, when we call these two separately. So let's keep going,
we've got all
```

Section 219: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
of the image parts, what else did we have to do? We want to create transforms. So
let's set up
transforms, self dot transforms equals transform. Oh, we'll just call that
transform actually,
```

set up transform equals transform. So we're going to get this from here. And I put it as
none because it transform can be optional. So let's create classes and class to ID X attributes,
which is the next one on our list, which is here classes and class to ID X. Now, lucky us,
in the previous video, we created a function to return just those things. So let's go self dot
classes and self dot class to ID X equals find classes. And we're going to pass in the target
der or the target der from here. Now, what's next? We've done step number three, we need
number four is create a function to load images. All right, let's see what this looks like. So
number four, create a function to load images. So let's call it load image. And we're going to
pass in self. And we'll also pass in an index. So the index of the image we'd like to load.
And this is going to return an image dot image. So where does that come from? Well, previously,
we imported from pill. So we're going to use Python image library or pillow to import our
images. So we're going to give on a file path from here, such as pizza, we're going to import
it with the image class. And we can do that using, I believe it's image dot open. So let's give that
a try. I'll just write a note in here, opens an image via a path and returns it. So let's write
image path equals self. This is why we got all of the image paths above. So self dot paths. And
we're going to index it on the index. Beautiful. And then let's return image dot open image path.
So we're going to get a particular image path. And then we're just going to open it.
So now we're up to step number five, override the land method to return the length of our data set.
This is optional, but we're going to do it anyway. So overwrite.
Len. So this just wants to return how many samples we have in our data set. So let's write that
def, Len. So if we call Len on our data set instance, it's going to return just how many numbers there
are. So let's write this down. Returns the total number of samples. And this is just going to be
simply return length or Len of self dot paths. So for our target directory, if it was the training
directory, we'd return the number of image paths that this code has found out here. And same for the
test directory. So next, I'm going to go number six is we want to overwrite, we put this up here,
the get item method. So this is required if we want to subclass torch utils data data set. So
this is in the documentation here. All subclasses should override get item. So we want get item to,
if we pass it an index to our data set, we want it to return that particular item. So let's see
what this looks like. Override the get item method to return our particular sample.
And now this method is going to leverage get item, all of the code that we've

```
created above.
So this is going to go take in self, which is the class itself. And it's going to
take in an index,
which will be of an integer. And it's going to return a tuple of torch dot tensor
and an integer,
which is the same thing that gets returned when we index on our training data. So
if we have a
look image label equals train data, zero, get item is going to replicate this. We
pass it an index here.
Let's check out the image and the label. This is what we have to replicate. So
remember train
data was created with image folder from torch vision dot data sets. And so we will
now get item
to return an image and a label, which is a tuple of a torch tensor, where the image
is of a tensor
here. And the label is of an integer, which is the label here, the particular index
as to which
this image relates to. So let's keep pushing forward. I'm going to write down here,
returns one sample
of data, data and label, X and, or we'll just go XY. So we know that it's a tuple.
Beautiful.
So let's set up the image. What do we want the image to be? Well, this is where
we're going to
call on our self dot load image function, which is what we've created up here. Do
you see the
customization capabilities of creating your own class? So we've got a fair bit of
code here,
right? But essentially, all we're doing is we're just creating functions that is
going to help us
load our images into some way, shape or form. Now, again, I can't stress this
enough, regardless
of the data that you're working on, the pattern here will be quite similar. You'll
just have to
change the different functions you use to load your data. So let's load an image of
a particular
index. So if we pass in an index here, it's going to load in that image. Then what
do we do? Well,
we want to get the class name, which is going to be self dot paths. And we'll get
the index here,
and we can go parent dot name. So this expects path in format data,
folder slash class name slash image dot JPG. That's just something to be aware of.
And the class
```

Section 220: Main Topics

Key Topics:

- So we're going to create a transform here so that we can transform our images raw jpeg images into tenses, because that's the whole goal of importing data into pytorch

- But now we're using the to transform transform from pytorch or torch visions dot transforms

▶ 📄 Click to view detailed content

```
ID X is going to be self dot class to ID X. And we will get the class name here.
So now we have an image by loading in the image here. We have a class name by
because our data
is going to be or our data is currently in standard image classification format.
You may have to
change this depending on the format your data is in, we can get the class name from
that,
and we can get the class ID X by indexing on our attribute up here, our dictionary
of class names
to indexes. Now we have one small little step. This is transform if necessary. So
remember our
transform parameter up here. If we want to transform our target image, well, let's
put in if self dot
transform if the transform exists, let's pass the image through that transform,
transform image
and then we're going to also return the class ID X. So do you notice how we've
returned a
tuple here? This is going to be a torch tensor. If our transform exists and the
class ID X is also
going to be returned, which is what we want here, X and Y, which is what gets
returned here,
image as a tensor label as an integer. So return data label X, Y, and then if the
transform doesn't
exist, let's just return image class ID X, return untransformed image and label.
Beautiful. So
that is a fair bit of code there. So you can see the pro of subclassing torch utils
data that data
set is that we can customize this in almost any way we wanted to to load whatever
data that we're
working with, well, almost any data. However, because we've written so much code,
this may be
prone to errors, which we're going to find out in the next video to see if it
actually works.
But essentially, all we've done is we've followed the documentation here torch dot
utils data
dot data set to replicate the functionality of an existing data loader function,
namely image folder.
So if we scroll back up, ideally, if we've done it right, we should be able to
write code like this,
passing in a root directory, such as a training directory, a particular data
transform.
And we should get very similar instances as image folder, but using our own custom
data set class.
So let's try that out in the next video. So now we've got a custom image folder
class
that replicates the functionality of the original image folder, data loader class,
or data set class, that is, let's test it out. Let's see if it works on our own
custom data.
So we're going to create a transform here so that we can transform our images raw
jpeg images into tenses,
```

because that's the whole goal of importing data into pytorch. So let's set up a train transforms
compose. We're going to set it to equal to transforms dot compose. And I'm going to pass in a list here,
that it's going to be transforms, we're going to resize it to 6464. Whatever the image size will
reduce it down to 6464. Then we're going to go transforms dot random horizontal flip. We don't
need to necessarily flip them, but we're going to do it anyway, just to see if it works. And then
let's put in here transforms dot to tensor, because our images are getting opened as a pill image,
using image dot open. But now we're using the to transform transform from pytorch or torch
visions dot transforms. So I'll just put this here. From torch vision dot transforms, that way you
know where importing transforms there. And let's create one for the test data set as well, test
transforms, we'll set this up. Oh, excuse me, I need to just go import transforms. And let's go
transforms dot compose. And we'll pass in another list, we're going to do the exact same as above,
we'll set up resize, and we'll set the size equal to 6464. And then transforms, we're going to go
dot to tensor, we're going to skip the data augmentation for test data. Because typically,
you don't manipulate your test data in terms of data augmentation, you just convert it into a
tensor, rather than manipulate its orientation, shape, size, etc, etc. So let's run this.
And now let's see how image folder custom class works. Test out image folder custom.
Let's go, we'll set up the train data custom is equal to image folder custom. And then we'll set up
the target, which is equal to the training directory. And then we'll pass in the transform,
which is equal to the train transforms, which we just created above train transforms. And then
we're going to, I think that's all we need, actually, we only had two parameters that we're not going
to use a target transform, because our labels, we've got to help a function to transform our labels.
So test data custom is going to be image folder custom. And I'm going to set up the target to be
equal to the test directory. And the transform is going to be the test transforms from the cell
above there. And what's co lab telling me there? Oh, I'm going to set that up. Did we spell
something? Oh, we spelled it wrong train transforms. There we go. Beautiful. Now let's have a look at
our train data and test data custom. See if it worked. What do we have? Or we have an image folder
custom. Well, it doesn't give us as much rich information as just checking it out as it does
for the train data. But that's okay. We can still inspect these. So this is our original one made

```
with image folder. And we've got now train data custom and test data custom. Let's
see if we can
```

Section 221: Main Topics

Key Topics:

- And so the takeaways from this is that whatever data you have, PyTorch gives you a base data set class to inherit from
- So that's going to help you work with your own custom data sets in PyTorch
- We've seen analytically that our custom data set is quite similar to the original PyTorch, torch vision dot data sets image folder data set

▶ 📄 Click to view detailed content

```
get some information from there. So let's check the original length of the train
data and see if
we can use the land method on our train data custom. Did that work? Wonderful. Now
how about we do it
for the original test data made with image folder and our custom version made with
test data or
image folder custom. Beautiful. That's exactly what we want. And now let's have a
look at the
train data custom. Let's see if the classes attribute comes up. Dot classes. And
we'll just leave that
there. We'll do the class dot ID X. Yes, it is. So this attribute here is I wonder
if we get
information from Google co lab loading. What do we get? Oh, classes to ID X classes
load image
paths transform. So if we go back up here, all these attributes are from here paths
transform
classes class to ID X as well as load image. So this is all coming from the code
that we wrote
our custom data set class. So let's keep pushing forward. Let's have a look at the
class to ID X.
Do we get the same as what we wanted before? Yes, we do beautiful a dictionary
containing our
string names and the integer associations. So let's now check for equality. We can
do this by going
check for equality between original image folder data set and image folder custom
data set. Now
we've kind of already done that here, but let's just try it out. Let's go print.
Let's go train
data custom dot classes. Is that equal to train? Oh, I don't want three equals
train data. The
original one classes and also print. Let's do test data custom dot classes. Is this
equal to
test data? The original one classes. True and true. Now you could try this out. In
```

fact,
it's a little exercise to try it out to compare the others. But congratulations to us, we have
replicated the main functionality of the image folder data set class. And so the takeaways from
this is that whatever data you have, PyTorch gives you a base data set class to inherit from.
And then you can write a function or a class that somehow interacts with whatever data you're
working with. So in our case, we load in an image. And then you, as long as you override the land
method and the get item method and return some sort of values, well, you can create your own
data set loading function. How beautiful is that? So that's going to help you work with your own
custom data sets in PyTorch. So let's keep pushing forward. We've seen analytically that
our custom data set is quite similar to the original PyTorch, torch vision dot data sets
image folder data set. But you know what I like to do? I like to visualize things. So let's in
the next video, create a function to display some random images from our trained data custom class.
It's time to follow the data explorer's motto of visualize, visualize, visualize. So let's
create another section. I'm going to write here a title called create a function to display random
images. And sure, we've, we've had a look at the different attributes of our custom data set.
We see that it gives back a list of different class names. We see that the lengths are similar
to the original, but there's nothing quite like visualizing some data. So let's go in here. We're
going to write a function, a helper function. So step number one, we need to take in a data set.
So one of the data sets that we just created, whether it be trained data custom or trained data.
And a number of other parameters, such as class names and how many images to visualize. And then
step number two is to prevent the display getting out of hand. Let's cap the number of
images to see at 10. Because look, if our data set is going to be thousands of images and we want
to put in a number of images to look at, let's just make sure it's the maximum is 10. That should
be enough. So we'll set the random seed for reproducibility. Number four is, let's get a list of random
samples. So we want random sample indexes, don't just get rid of this s from what do we want it from
from the target data set. So we want to take in a data set, and we want to count the number of
images we're seeing, we want to set a random seed. And do you see how much I use randomness here to
really get an understanding of our data? I really, really, really love harnessing the power of
randomness. So we want to get a random sample of indexes from all of our data set. And then we're

```
going to set up a matplotlib plot. Then we want to loop through the random sample
images.
And plot them with matplotlib. And then as a side to this one, step seven is we
need to make sure
the dimensions of our images line up with matplotlib. So matplotlib needs a height
width color channels.
All right, let's take it on, hey? So number one is create a function to take in a
data set.
So we're going to call this def, let's call it def display random images going to
be one of our
helper functions. We've created a few type of functions like this. But let's take
in a data set,
which is torch utils of type that is of type data set. Then we're going to take in
classes,
which is going to be a list of different strings. So this is going to be our class
names for
whichever data set we're using. I'm going to set this equal to none. And then we're
going to take in
```

Section 222: Main Topics

Key Topics:

- So you can customize the beautiful thing about Python and PyTorch, as you can customize these display functions in any way you see fit
- So if we recall by default, pytorch is going to turn our image dimensions into what color channels first, however, matplotlib prefers color channels last

▶ 📄 Click to view detailed content

```
n, which is the number of images we'd like to plot. And I'm going to set this to 10
by default. So
we can see 10 images at a time, 10 random images, that is, do we want to display
the shape? Let's
set that equal to true, so that we can display what the shape of the images,
because we're passing
it through our transform as it goes into a data set. So we want to see what the
shape of our
images are just to make sure that that's okay. And we can also let's set up a seed,
which is
going to be an integer, and we'll set that to none to begin with as well. Okay, so
step number two,
what do we have above? We have to prevent the display getting out of hand, let's
cap the number
of images to see at 10. So we've got n is by default, it's going to be 10, but
let's just make
sure that it stays there. Adjust display, if n is too high. So if n is greater than
10,
```

let's just readjust this, let's set n equal to 10, and display shape, we'll turn off the
display shape, because if we have 10 images, our display may get out of hand. So just print out
here for display purposes, and shouldn't be larger than 10, setting to 10, and removing
shape display. Now I only know this because I've had experience cooking this dish before.
In other words, I've written this type of code before. So you can customize the beautiful thing
about Python and PyTorch, as you can customize these display functions in any way you see fit.
So step number three, what are we doing? Set the random seed for reproducibility. Okay,
set the seed. So if seed, let's set random dot seed equal to that seed value, and then we can keep
and then we can keep going. So number four is let's get some random sample indexes. So we can do
that by going get random sample indexes, which is step number four here. So we've got a target
data set that we want to inspect. We want to get some random samples from that. So let's create a
random samples IDX list. And I'm going to randomly sample from a length of our data set, or sorry,
a range of the length of our data set. And I'll show you what this means in a second.
And the K, excuse me, have we got enough brackets there? I always get confused with the brackets.
The K is going to be n. So in this case, I want to randomly sample 10 images from the length of
our data set or 10 indexes. So let's just have a look at what this looks like. We'll put in here,
our train data custom here. So this is going to take a range of the length of our train data
custom, which is what 225. We looked at that before, just up here, length of this. So between zero
and 255, we're going to get 10 indexes if we've done this correctly. Beautiful. So there's 10
random samples from our train data custom, or 10 random indexes, that is. So we're up to step number
five, which was loop through the random sample images or indexes. Let's create this to indexes,
indexes and plot them with matplotlib. So this is going to give us a list here.
So let's go loop through random indexes and plot them with matplotlib. Beautiful. So for
i tug sample in enumerate, let's enumerate through the random, random samples, idx list.
And then we're going to go tug image and tug label, because all of the samples in our target
data set are in the form of tuples. So we're going to get the target image and the target label,
which is going to be data set tug sample. We'll take the index. So it might be one of these values
here. We'll index on that. And the zero index will be the image. And then we'll go on the data set as
well. We'll take the tug sample index. And then the index number one will be the label of our target

```
sample. And then number seven, oh, excuse me, we've missed a step. That should be
number six.
Did you catch that? Number five is setup plot. So we can do this quite easily by
going plot
figure. This is so that each time we iterate through another sample, we're going to
have
quite a big figure here. So we set up the plot outside the loop so that we can add
a plot to this
original plot here. And now this is number seven, where we make sure the dimensions
of our images
line up with matplotlib. So if we recall by default, pytorch is going to turn our
image dimensions into
what color channels first, however, matplotlib prefers color channels last. So
let's go adjust,
tensor dimensions for plotting. So let's go tag image. Let's call this tag image
adjust equals
tag image dot commute. And we're going to alter the order of the indexes. So this
is going to go
from color channels or the dimensions that is height width. And we're going to
change this width,
if I could spell, to height width color channels. Beautiful. That one will probably
catch you off
guard a few times. But we've seen it a couple of times now. So we're going to keep
going with this
plot adjusted samples. So now we can add a subplot to our matplotlib plot. And we
want to create,
we want one row of n images, this will make a lot more sense when we visualize it.
And then for
the index, we're going to keep track of i plus one. So let's keep going. So then
we're going to go
```

Section 223: Main Topics

Key Topics:

- So this is the inbuilt pytorch image folder
- And we can still use it with PyTorch's data loader

▶ 📄 Click to view detailed content

```
plot in show. And I'm going to go tug image adjust. So I'm going to plot this image
here. And then
let's turn off the axis. And we can go if the classes variable exists, which is up
here, a list
of classes, let's adjust the title of the plot to be the particular index in the
class list. So
title equals f class. And then we're going to put in here classes. And we're going
to index on that
with the target label index, which is going to come from here. Because that's going
```

to be a new
numerical format. And then if display shape, let's set the title equal to title plus f. We're going
to go new line shape. This is going to be the shape of the image, tug image adjust dot shape.
And then we'll set the title to PLT dot title. So you see how if we have display shape, we're
just adjusting the title variable that we created here. And then we're putting the title onto the
plot. So let's see how this goes. That is quite a beautiful function. Let's pass in one of our
data sets and see what it looks like. Let's plot some random images. So which one should we start
with first? So let's display random images from the image folder created data sets. So this is the
inbuilt pytorch image folder. Let's go display random images, the function we just created above.
We're going to pass in the train data. And then we can pass in the number of images. Let's have
a look at five. And the classes is going to be the class names, which is just a list of our
different class names. And then we can set the seed, we want it to be random. So we'll just set
the seed to equal none. Oh, doesn't that look good? So this is from our original train data
made with image folder. So option number one up here, option one, there we go. And we've
passed in the class name. So this is sushi resize to 64, 64, three, same with all of the others,
but from different classes. Let's set the seed to 42, see what happens. I get these images,
we got a sushi, we got a pizza, we got pizza, sushi pizza. And then if we try a different one,
we just go none. We get random images again, wonderful. Now let's write the same code,
but this time using our train data custom data set. So display random images from the image folder
custom data set. So this is the one that we created display random images. I'm going to pass
in train data custom, our own data set. Oh, this is exciting. Let's set any equal to 10 and just see
see how far we can go with with our plot. Or maybe we set it to 20 and just see if our
code for adjusting the plot makes sense. Class names and seed equals, I'm going to put in 42 this time.
There we go. For display purposes, and shouldn't be larger than 10 setting to 10 and removing shape
display. So we have a stake image, a pizza image, pizza, steak pizza, pizza, pizza, pizza, steak,
pizza. If we turn off the random seed, we should get another 10 random images here.
Beautiful. Look at that. Steak, steak, sushi, pizza, steak, sushi class. I'm reading out
the different things here. Pizza, pizza, pizza, pizza. Okay. So it looks like our custom data set
is working from both a qualitative standpoint, looking at the different images and a quantitative.
How about we change it to five and see what it looks like? Do we have a different

```
shape? Yes,
we do the same shape as above. Wonderful. Okay. So we've got train data custom.
And we've got train data, which is made from image folder. But the premises remain,
we've built up
a lot of different ideas. And we're looking at things from different points of
view. We are
getting our data from the folder structure here into tensor format. So there's
still one more
step that we have to do. And that's go from data set to data loader. So in the next
video,
let's see how we can turn our custom loaded images, train data custom, and test
data custom
into data loaders. So you might want to go ahead and give that a try yourself.
We've done it before
up here. Turn loaded images into data loaders. We're going to replicate the same
thing as we did
in here for our option number two, except this time we'll be using our custom data
set.
I'll see you in the next video. I'll take some good looking images and even better
that they're
from our own custom data set. Now we've got one more step. We're going to turn our
data set into a
data loader. In other words, we're going to batchify all of our images so they can
be used with the
model. And I gave you the challenge of trying this out yourself in the last video.
So I hope
you gave that a go. But let's see what that might look like in here. So I'm going
to go 5.4.
Let's go. What should we call this? So turn custom loaded images into data loaders.
So this
is just goes to show that we can write our own custom data set class. And we can
still use it
with PyTorch's data loader. So let's go from utils torch dot utils that is utils
dot data import
data loader. We'll get that in here. We don't need to do that again, but I'm just
doing it for
completeness. So we're going to set this to train data loader custom. And I'm going
to create an
```

Section 224: Main Topics

Key Topics:

- And of course, numb workers, we could also set this numb workers equals zero or OS dot CPU count
- And so that's just exactly what pytorch is done with taught vision dot data sets dot image folder

▶ 📄 Click to view detailed content

instance of data loader here. And then inside I'm going to pass the data set, which is going to be
train data custom. I'm just going to set a universal parameter here in capitals for batch size equals
32. Because we can come down here, we can set the batch size, we're going to set this equal to 32.
Or in other words, the batch size parameter we set up there, we can set the number of workers
here as well. If you set to zero, let's go see what the default is actually torch utils data loader.
What's the default for number of workers? Zero. Okay, beautiful. And recall that number of workers
is going to set how many cores load your data with a data loader. And generally higher is better.
But you can also experiment with this value and see what value suits your model and your
hardware the best. So just keep in mind that number of workers is going to alter how much
compute your hardware that you're running your code on uses to load your data. So by default,
it's set to zero. And then we're going to shuffle the training data. Wonderful. And let's do the
same for the test data loader. We'll create test data loader custom. And I'm going to create a
new instance. So let me make a few code cells here of data loader, and create a data set or pass
in the data set parameter as the test data custom. So again, these data sets are what we've created
using our own custom data set class. I'm going to set the batch size equal to batch size. And
let's set the number workers equal to zero. In a previous video, we've also set it to CPU count.
You can also set it to one. You can hard code it to four all depends on what hardware you're using.
I like to use OPA OS dot CPU count. And then we're not going to shuffle the test data.
False. Beautiful. And let's have a look at what we get here. Train data loader custom and test
data loader custom. And actually, I'm just going to reset this instead of being OOS CPU count.
I'm going to put it back to zero, just so we've got it in line with the one above.
And of course, numb workers, we could also set this numb workers equals zero or OS dot CPU count.
And then we could come down here and set this as numb workers and numb workers.
And let's have a look to see if it works. Beautiful. So we've got two instances of utils.data.data
loader. Now, let's just get a single sample from the train data loader here, just to make sure the
image shape and batch size is correct. Get image and label from custom data loader. We want image
custom. And I'm going to go label custom equals next. And I'm going to iter over the train data
loader custom. And then let's go print out the shapes. We want image custom dot shape and label
custom. Do we get a shape here? Beautiful. There we go. So we have shape here of 32,

```
because that is our batch size. Then we have three color channels, 64, 64, which is
in line with
what? Which is in line with our transform that we set all the way up here.
Transform. We transform
our image. You may want to change that to something different depending on the
model you're using,
depending on how much data you want to be comprised within your image. Recall,
generally a larger
image size encodes more information. And this is all coming from our original image
folder
custom data set class. So look at us go. And I mean, this is a lot of code here or
a fair bit of
code, right? But you could think of this as like you write it once. And then if
your data set continues
to be in this format, well, you can use this over and over again. So you might put
this, this image
folder custom into a helper function file over here, such as data set dot pie or
something like
that. And then you could call it in future code instead of rewriting it all the
time. And so that's
just exactly what pytorch is done with taught vision dot data sets dot image
folder. So we've
got some shapes here. And if we wanted to change the batch size, what do we do? We
just change it
like that 64. Remember, a good batch size is also a multiple of eight, because
that's going to help
out computing. And batch size equals one. We get a batch size equal of one. We've
been through a
fair bit. But we've covered a very important thing. And that is loading your own
data with a custom
data set. So generally, you will be able to load your own data with an existing
data loading function
or data set function from one of the torch domain libraries, such as torch audio,
torch text,
torch vision, torch rack. And later on, when it's out of beta, torch data. But if
you need to create
your own custom one, while you can subclass torch dot utils dot data, dot data set,
and then add
your own functionality to it. So let's keep pushing forward. Previously, we touched
a little bit on
transforming data. And you may have heard me say that torch vision transforms can
be used for data
augmentation. And if you haven't, that is what the documentation says here. But
data augmentation
is manipulating our images in some way, shape or form, so that we can artificially
increase the
diversity of our training data set. So let's have a look at that more in the next
video. I'll see you
```

Section 225: Main Topics

Key Topics:

- And that is one used to recently train pytorch torch vision image models to state of the art levels
- So let's take a look at one particular type of data augmentation, used to train pytorch vision models to state of the art levels
- So this is a recent blog post by the pytorch team, how to train state of the art models, which is what we want to do, state of the art means best in business, otherwise known as soda

▶ 📄 Click to view detailed content

```
there. Over the last few videos, we've created functions and classes to load in our
own custom
data set. And we learned that one of the biggest steps in loading a custom data set
is transforming
your data, particularly turning your target data into tenses. And we also had a
brief look at the
torch vision transforms module. And we saw that there's a fair few different ways
that we can
transform our data. And that one of the ways that we can transform our image data
is through
augmentation. And so if we went into the illustration of transforms, let's have a
look at all the
different ways we can do it. We've got resize going to change the size of the
original image.
We've got center crop, which will crop. We've got five crop. We've got grayscale.
We've got random
transforms. We've got Gaussian blur. We've got random rotation, random caffeine,
random crop.
We could keep going. And in fact, I'd encourage you to check out all of the
different options here.
But oh, there's auto augment. Wonderful. There's random augment. This is what I was
hinting at.
Data augmentation. Do you notice how the original image gets augmented in different
ways here?
So it gets artificially changed. So it gets rotated a little here. It gets dark and
a little
here or maybe brightened, depending how you look at it, it gets shifted up here.
And then the colors
kind of change here. And so this process is known as data augmentation, as we've
hinted at.
And we're going to create another section here, which is number six, other forms of
transforms.
And this is data augmentation. So how could you find out about what data
augmentation is?
Well, you could go here. What is data augmentation? And I'm sure there's going to
be plenty of
resources here. Wikipedia. There we go. Data augmentation in data analysis are
techniques
used to increase the amount of data by adding slightly modified copies of already
existing data
or newly created synthetic data from existing data. So I'm going to write down
here,
```

data augmentation is the process of artificially adding diversity to your training data.
Now, in the case of image data, this may mean applying various image transformations to the
training images. And we saw a whole bunch of those in the torch vision transformed package.
But now let's have a look at one type of data augmentation in particular. And that is trivial
augment. But just to illustrate this, I've got a slide here ready to go. We've got what is data
augmentation. And it's looking at the same image, but from different perspectives. And we do this,
as I said, to artificially increase the diversity of a data set. So if we imagine our original
images over here on the left, and then if we wanted to rotate it, we could apply a rotation
transform. And then if we wanted to shift it on the vertical and the horizontal axis,
we could apply a shift transform. And if we wanted to zoom in on the image, we could apply
a zoom transform. And there are many different types of transforms. As I've got a note here,
there are many different kinds of data augmentation, such as cropping, replacing,
shearing. And this slide only demonstrates a few. But I'd like to highlight another type of data
augmentation. And that is one used to recently train pytorch torch vision image models to state
of the art levels. So let's take a look at one particular type of data augmentation,
used to train pytorch vision models to state of the art levels.
Now, just in case you're not sure why we might do this, we would like to increase
the diversity of our training data so that our images become harder for our model to learn. Or
it gets a chance to view the same image from different perspectives so that when you use your
image classification model in practice, it's seen the same sort of images, but from many different
angles. So hopefully it learns patterns that are generalizable to those different angles.
So this practice, hopefully, results in a model that's more generalizable to unseen data.
And so if we go to torch vision, state of the art, here we go. So this is a recent blog post
by the pytorch team, how to train state of the art models, which is what we want to do,
state of the art means best in business, otherwise known as soda. You might see this acronym quite
often using torch visions latest primitives. So torch vision is the package that we've been
using to work with vision data. And torch vision has a bunch of primitives, which are,
in other words, functions that help us train really good performing models. So blog post here.
And if we jump into this blog post and if we scroll down, we've got some improvements here.
So there's an original ResNet 50 model. ResNet 50 is a common computer vision architecture.

```
So accuracy at one. So what do we have? Well, let's just say they get a boost in
what the previous
results were. So if we scroll down, there is a type of data augmentation here. So
if we add up
all of the improvements that they used, so there's a whole bunch here. Now, as your
extra curriculum,
I'd encourage you to look at what the improvements are. You're not going to get
them all the first
go, but that's all right. Blog posts like this come out all the time and the
recipes are continually
changing. So even though I'm showing you this now, this may change in the future.
So I just
scroll down to see if this table showed us what the previous results were. Doesn't
look like it does.
```

Section 226: Main Topics

Key Topics:

- So there's a bunch of different things such as learning rate optimization, training for longer
- That is from the PyTorch torch vision transforms library
- And it was used trivial augment to train the latest state of the art vision models in the PyTorch torch vision models library or models repository

▶ 📄 Click to view detailed content

```
Oh, no, there's the baseline. So 76 and with all these little additions, it got
right up to nearly
81. So nearly a boost of 5% accuracy. And that's pretty good. So what we're going
to have a look
at is trivial augment. So there's a bunch of different things such as learning rate
optimization,
training for longer. So these are ways you can improve your model. Random erasing
of image data,
label smoothing, you can add that as a parameter to your loss functions, such as
cross entropy loss,
mix up and cut mix, weight decay tuning, fixed res mitigations, exponential moving
average,
which is EMA, inference resize tuning. So there's a whole bunch of different recipe
items here,
but we're going to focus on what we're going to break it down. Let's have a look at
trivial
augment. So we'll come in here. Let's look at trivial augment. So if we wanted to
look at
trivial augment, can we find it in here? Oh, yes, we can. It's right here. Trivial
augment.
So as you'll see, if you pass an image into trivial augment, it's going to change
```

it in a few
different ways. So if we go into here, let's write that down. So let's see this in action on some
of our own data. So we'll import from torch vision, import transforms. And we're going to create a
train transform, which is equal to transforms dot compose. We'll pass it in there. And this is
going to be very similar to what we've done before in terms of composing a transform. What do we
want to do? Well, let's say we wanted to resize one of our images or an image going through this
transform. Let's change its size to 224224, which is a common size in image classification. And
then it's going to go through transforms. We're going to pass in trivial augment wide. And there's
a parameter here, which is number of magnitude bins, which is basically a number from 0 to 31,
31 being the max of how intense you want the augmentation to happen. So say we, we only put this as
5, our augmentation would be of intensity from 0 to 5. And so in that case, the maximum wouldn't
be too intense. So if we put it to 31, it's going to be the max intensity. And what I mean by intensity
is say this rotation, if we go on a scale of 0 to 31, this may be a 10, whereas 31 would be
completely rotating. And same with all these others, right? So the lower this number, the less the
maximum up a bound of the applied transform will be. Then if we go transforms dot to tensor,
wonderful. So there we've just implemented trivial augment. How beautiful is that? That is from
the PyTorch torch vision transforms library. We've got trivial augment wide. And it was used
trivial augment to train the latest state of the art vision models in the PyTorch torch vision
models library or models repository. And if you wanted to look up trivial augment, how could you
find that? You could search it. Here is the paper if you'd like to read it. Oh, it's implemented.
It's actually a very, very, I would say, let's just say trivial augment. I didn't want to say
simple because I don't want to downplay it. Trivial augment leverages the power of randomness
quite beautifully. So I'll let you read more on there. I would rather try it out on our data
and visualize it first. Test transform. Let's go transforms compose. And you might have the
question of which transforms should I use with my data? Well, that's the million dollar question,
right? That's the same thing as asking, which model should I use for my data? There's a fair
few different answers there. And my best answer will be try out a few, see what work for other
people like we've done here by finding that trivial augment worked well for the PyTorch team.
Try that on your own problems. If it works well, excellent. If it doesn't work well,

```
well, you can always excuse me. We've got a spelling mistake. If it doesn't work
well,
well, you can always set up an experiment to try something else. So let's test out
our
augmentation pipeline. So we'll get all the image paths. We've already done this,
but we're
going to do it anyway. Again, just to reiterate, we've covered a fair bit here. So
I might just
rehash on a few things. We're going to get list, image path, which is our, let me
just show you
our image path. We just want to get all of the images within this file.
So we'll go image path dot glob, glob together all the files and folders that match
this pattern.
And then if we check, what do we get? We'll check the first 10. Beautiful. And then
we can
leverage our function from the four to plot some random images, plot random images.
We'll pass in or plot transformed random transformed images. That's what we want.
Let's see what it looks like when it goes through our trivial augment. So image
paths,
equals image part list. This is a function that we've created before, by the way,
transform equals
train transform, which is the transform we just created above that contains trivial
augment.
And then we're going to put n equals three for five images. And we'll do seed
equals none
to plot. Oh, sorry, n equals three for three images, not five. Beautiful. And we'll
set the
seed equals none, by the way. So look at this. We've got class pizza. Now trivial
augment,
it resized this. Now, I'm not quite sure what it did to transform it per se. Maybe
it got a little
bit darker. This one looks like it's been the colors have been manipulated in some
way, shape,
```

Section 227: Main Topics

Key Topics:

- And of course, you can read a little bit more in the documentation, or sorry, in the paper here
- I've just highlighted trivial augment because it's what the PyTorch team have used in their most recent blog post for their training recipe to train state-of-the-art vision models
- In the last video, we covered how the PyTorch team used trivial augment wide, which is the latest state-of-the-art in data augmentation at the time of recording this video to train their latest state-of-the-art computer vision models that are within torch vision

▶ 📄 Click to view detailed content

```
or form. And this one looks like it's been resized and not too much has happened to
that one from
my perspective. So if we go again, let's have a look at another three images. So
trivial augment
works. And what I said before, it harnesses the power of randomness. It kind of
selects randomly
from all of these other augmentation types, and applies them at some level of
intensity.
So all of these ones here, trivial augment is just going to select summit random,
and then
apply them some random intensity from zero to 31, because that's what we've set on
our data.
And of course, you can read a little bit more in the documentation, or sorry, in
the paper here.
But I like to see it happening. So this one looks like it's been cut off over here
a little bit.
This one again, the colors have been changed in some way, shape, or form. This
one's been darkened.
And so do you see how we're artificially adding diversity to our training data set?
So instead
of all of our images being this one perspective like this, we're adding a bunch of
different
angles and telling our model, hey, you got to try and still learn these patterns,
even if they've
been manipulated. So we'll try one more of these. So look at that one. That's
pretty
manipulated there, isn't it? But it's still an image of stake. So that's what we're
trying to
get our model to do is still recognize this image as an image of stake, even though
it's been
manipulated a bit. Now, will this work or not? Hey, it might, it might not, but
that's all the
nature of experimentation is. So play around. I would encourage you to go in the
transforms
documentation like we've just done, illustrations, change this one out, trivial
augment wine,
for another type of augmentation that you can find in here, and see what it does to
some of
our images randomly. I've just highlighted trivial augment because it's what the
PyTorch team have
used in their most recent blog post for their training recipe to train state-of-
the-art vision
models. So speaking of training models, let's move forward and we've got to build
our first model
for this section. I'll see you in the next video.
Welcome back. In the last video, we covered how the PyTorch team used trivial
augment
wide, which is the latest state-of-the-art in data augmentation at the time of
recording this
video to train their latest state-of-the-art computer vision models that are within
torch vision. And we saw how easily we could apply trivial augment thanks to torch
vision
dot transforms. And we'll just see one more of those in action, just to highlight
what's going on.
```

So it doesn't look like much happened to that image when we augmented, but we see this one has
been moved over. We've got some black space there. This one has been rotated a little,
and now we've got some black space there. But now's time for us to build our first
computer vision model on our own custom data set. So let's get started. We're going to go model zero.
We're going to reuse the tiny VGG architecture, which we covered in the computer vision section.
And the first experiment that we're going to do, we're going to build a baseline,
which is what we do with model zero. We're going to build it without data augmentation.
So rather than use trivial augment, which we've got up here, which is what the PyTorch team used
to train their state-of-the-art computer vision models, we're going to start by training our
computer vision model without data augmentation. And then so later on, we can try one to see
with data augmentation to see if it helps or doesn't. So let me just put a link in here,
CNN explainer. This is the model architecture that we covered in depth in the last section.
So we're not going to go spend too much time here. All you have to know is that we're going
to have an input of 64, 64, 3 into multiple different layers, such as convolutional layers,
relio layers, max pool layers. And then we're going to have some output layer that suits the
number of classes that we have. In this case, there's 10 different classes, but in our case,
we have three different classes, one for pizza, steak, and sushi. So let's replicate the tiny VGG
architecture from the CNN explainer website. And this is going to be good practice, right?
We're not going to spend too much time referencing their architecture. We're going to spend more
time coding here. But of course, before we can train a model, what do we have to do? Well,
let's go 7.1. We're going to create some transforms and loading data. We're going to load data for
model zero. Now, we could of course use some of the variables that we already have loaded. But
we're going to recreate them just to practice. So let's create a simple transform. And what is
our whole premise of loading data for model zero? We want to get our data from the data folder,
from pizza, steak sushi, from the training and test folders, from their respective folders,
we want to load these images and turn them into tenses. Now we've done this a few times now.
And one of the ways that we can do that is by creating a transform equals transforms dot compose.
And we're going to pass in, let's resize it. So transforms dot resize, we're going to resize our
images to be the same size as the tiny VGG architecture on the CNN explainer website. 64
64 three. And then we're also going to pass in another transform to tensor. So that

```
our
images get resized to 64 64. And then they get converted into tenses. And
particularly,
```

Section 228: Main Topics

Key Topics:

- So batchify it so that we can use it with a pytorch model
- And I know we've already coded this up before, but it's good practice to see what it's like to build pytorch models from scratch, create tiny VGG model class

▶ 📄 Click to view detailed content

```
these values within that tensor are going to be between zero and one. So there's
our transform.
Now we're going to load some data. If you want to pause the video here and try to
load it yourself,
I'd encourage you to try out option one, loading image data using the image folder
class,
and then turn that data set, that image folder data set into a data loader. So
batchify it so
that we can use it with a pytorch model. So give that a shot. Otherwise, let's go
ahead and do
that together. So one, we're going to load and transform data. We've done this
before,
but let's just rehash on it what we're doing. So from torch vision import data
sets, then we're
going to create the train data simple. And I call this simple because we're going
to use at first
a simple transform, one with no data augmentation. And then later on for another
modeling experiment,
we're going to create another transform one with data augmentation. So let's put
this here
data sets image folder. And let's go the route equals the training directory. And
then the
transform is going to be what? It's going to be our simple transform that we've got
above.
And then we can put in test data simple here. And we're going to create data sets
dot image
folder. And then we're going to pass in the route as the test directory. And we'll
pass in the
transform is going to be the simple transform again above. So we're performing the
same
transformation here on our training data, and on our testing data. Then what's the
next step
we can do here? Well, we can to turn the data sets into data loaders. So let's try
it out.
First, we're going to import OS, then from torch dot utils dot data, we're going to
```

```
import data
loader. And then we're going to set up batch size and number of workers. So let's
go batch size.
We're going to use a batch size of 32 for our first model.
Numb workers, which will be the number of excuse me, got a typo up here classic
number of workers,
which will be the what the number of CPU cores that we dedicate towards loading our
data.
So let's now create the data loaders. We're going to create train data loader
simple,
which will be equal to data loader. And the data set that goes in here will be
train data
simple. Then we can set the batch size equal to the batch size parameter that we
just created,
or hyper parameter that is, recall a hyper parameter is something that you can set
yourself. We
would like to shuffle the training data. And we're going to set numb workers equal
to numb workers.
So in our case, how many calls does Google Colab have? Let's just run this. Find
out how many
numb workers there are. I think there's going to be two CPUs. Wonderful. And then
we're going to do
the same thing for the test data loader. Test data loader simple. We're going to go
data loader.
We'll pass in the data set here, which is going to be the test data simple. And
then we're going
to go batch size equals batch size. We're not going to shuffle the test data set.
And then the
numb workers will just set it to the same thing as we've got above. Beautiful. So I
hope you gave
that a shot, but now do you see how quickly we can get our data loaded if it's in
the right format?
I know we spent a lot of time going through all of these steps over multiple videos
and
writing lots of code, but this is how quickly we can get set up to load our data.
We create a
simple transform, and then we load in and transform our data at the same time. And
then we turn the
data sets into data loaders just like this. Now we're ready to use these data
loaders with a model.
So speaking of models, how about we build the tiny VGG architecture in the next
video? And in
fact, we've already done this in notebook number three. So if you want to refer
back to the model
that we built there, right down here, which was model number two, if you want to
refer back to
this section and give it a go yourself, I'd encourage you to do so. Otherwise,
we'll build tiny VGG
architecture in the next video. Welcome back. In the last video, we got set up
starting to get
ready to model our first custom data set. And I issued you the challenge to try and
replicate
the tiny VGG architecture from the CNN explainer website, which we covered in
notebook number
three. But now let's see how fast we can do that together. Hey, I'm going to write
down here section
seven point two. And I know we've already coded this up before, but it's good
```

```
practice to see what
it's like to build pytorch models from scratch, create tiny VGG model class. So the
model is going
to come from here. Previously, we created our model, there would have been one big
change from
the model that we created in section number three, which is that our model in
section number three
used black and white images. But now the images that we have are going to be color
images. So
there's going to be three color channels rather than one. And there might be a
little bit of a
trick that we have to do to find out the shape later on in the classifier layer.
But let's get
started. We've got class tiny VGG, we're going to inherit from nn.module. This is
going to be
the model architecture copying tiny VGG from CNN explainer. And remember that it's
a it's
```

Section 229: Main Topics

Key Topics:

- quite a common practice in machine learning to find a model that works for a problem similar to yours and then copy it and try it on your own problem
- Of course, we could calculate them by hand by looking up the formula for input and output shapes of convolutional layers

▶ 📄 Click to view detailed content

```
quite a common practice in machine learning to find a model that works for a
problem similar to
yours and then copy it and try it on your own problem. So I only want two
underscores there.
We're going to initialize our class. We're going to give it an input shape, which
will be an int.
We're going to say how many hidden units do we want, which will also be an int. And
we're going
to have an output shape, which will be an int as well. And it's going to return
something none
of type none. And if we go down here, we can initialize it with super dot
underscore init.
Beautiful. And now let's create the first COM block. So COM block one, which we'll
recall
will be this section of layers here. So COM block one, let's do an nn.sequential to
do so.
Now we need com relu com relu max pool. So let's try this out. And then com to D.
The in channels is going to be the input shape of our model. The input shape
parameter.
```

The out channels is going to be the number of hidden units we have, which is from
Oh, I'm gonna just put enter down here input shape hidden units. We're just getting those
to there. Let's set the kernel size to three, which will be how big the convolving window will be
over our image data. There's a stride of one and the padding equals one as well. So these are the
similar parameters to what the CNN explainer website uses. And we're going to go and then
relu. And then we're going to go and then com to D. And I want to stress that even if someone
else uses like certain values for these, you don't have to copy them exactly. So just keep that in
mind. You can try out various values of these. These are all hyper parameters that you can set
yourself. Hidden units, out channels, equals hidden units as well. Then we're going to go kernel
size equals three stride equals one. And we're going to put padding equals one as well.
Then we're going to have another relu layer. And I believe I forgot my comma up here.
Another relu layer here. And we're going to finish off
with an N dot max pool 2D. And we're going to put in the kernel size.
These equals two and the stride here equals two. Wonderful. So oh, by the way, for max
pool 2D, the default stride value is same as the kernel size. So let's have a go here.
What can we do now? Well, we could just replicate this block as block two. So how about we copy this
down here? We've already had enough practice writing this sort of code. So we're going to
go comp block two, but we need to change the input shape here. The input shape of this block
two is going to receive the output shape here. So we need to line those up. This is going to be
hidden units. Hidden units. And I believe that's all we need to change there. Beautiful. So let's
create the classifier layer. And the classifier layer recall is going to be this output layer
here. So we need at some point to add a linear layer. That's going to have a number of outputs
equal to the number of classes that we're working with. And in this case, the number of classes is
10. But in our case, our custom data set, we have three classes, pizza, steak, sushi. So let's
create a classifier layer, which will be an end sequential. And then we're going to pass in an end
dot flatten to turn the outputs of our convolutional blocks into feature vector into a feature vector
site. And then we're going to have an end dot linear. And the end features, do you remember my
trick for calculating the shape in features? I'm going to put hidden units here for the time being.
Out features is going to be output shape. So I put hidden units here for the time being because
we don't quite yet know what the output shape of all of these operations is going to be. Of course,

```
we could calculate them by hand by looking up the formula for input and output
shapes of convolutional
layers. So the input and output shapes are here. But I prefer to just do it
programmatically and let
the errors tell me where I'm wrong. So we can do that by doing a forward pass. And
speaking of a
forward pass, let's create a forward method, because every time we have to subclass
an end
dot module, we have to override the forward method. We've done this a few times.
But as you can see,
I'm picking up the pace a little bit because you've got this. So let's pass in the
conv block one,
we're going to go X, then we're going to print out x dot shape. And then we're
going to reassign
X to be self.com block two. So we're passing it through our second block of
convolutional layers,
print X dot shape to check the shape here. Now this is where our model will
probably error
is because the input shape here isn't going to line up in features, hidden units,
because we've
passed all of the output of what's going through comp block one, comp block two to
a flatten layer,
because we want a feature vector to go into our nn.linear layer, our output layer,
which has an
out features size of output shape. And then we're going to return X. So I'm going
to print x dot
shape here. And I just want to let you in on one little secret as well. We haven't
covered this
before, but we could rewrite this entire forward method, this entire stack of code,
```

Section 230: Main Topics

Key Topics:

- Now this topic is beyond the scope of this course, essentially, all you need to know is that operator fusion behind the scenes speeds up how your GPU performs computations
- This is operator fusion, the most important optimization in deep learning compilers
- So I will link this, making deep learning go bur from first principles by Horace Hare, a great blog post that I really like, right here

▶ 📄 Click to view detailed content

```
by going return self dot classifier, and then going from the outside in. So we
could pass in
comp block two here, comp block two, and then self comp block one, and then X on
the inside.
```

So that is essentially the exact same thing as what we've done here, except this is going to
benefits from operator fusion. Now this topic is beyond the scope of this course,
essentially, all you need to know is that operator fusion behind the scenes speeds up
how your GPU performs computations. So all of these are going to happen in one step,
rather than here, we are reassigning X every time we make a computation through these layers.
So we're spending time going from computation back to memory, computation back to memory,
whereas this kind of just chunks it all together in one hit. If you'd like to read
more about this, I'd encourage you to look up the blog post, how to make your GPUs go
bur from first principles, and bur means fast. That's why I love this post, right?
Because it's half satire, half legitimately, like GPU computer science. So if you go in here,
yeah, here's what we want to avoid. We want to avoid all of this transportation between
memory and compute. And then if we look in here, we might have operator fusion. There we go.
This is operator fusion, the most important optimization in deep learning compilers. So
I will link this, making deep learning go bur from first principles by Horace Hare,
a great blog post that I really like, right here. So if you'd like to read more on that,
it's also going to be in the extracurricular section of the course. So don't worry, it'll be there.
Now, we've got a model. Oh, where do we, where do we forget a comma? Right here, of course we did.
And we've got another, we forgot another comma up here. Did you notice these?
Beautiful. Okay. So now we can create our model by going torch or an instance of the tiny VGG
to see if our model holds up. Let's create model zero equals tiny VGG. And I'm going to pass in
the input shape. What is the input shape? It's going to be the number of color channels of our
image. So number of color channels in our image data, which is three, because we have color images.
And then we're going to put in hidden units, equals 10, which will be the same number of
hidden units as the tiny VGG architecture. One, two, three, four, five, six, seven, eight, nine,
10. Again, we could put in 10, we could put in 100, we could put in 64, which is a good multiple
of eight. So let's just leave it at 10 for now. And then the output shape is going to be what?
It's going to be the length of our class names, because we want one hidden unit or one output unit
per class. And then we're going to send it to the target device, which is of course CUDA. And then
we can check out our model zero here. Beautiful. So that took a few seconds, as you saw there,
to move to the GPU memory. So that's just something to keep in mind for when you build
large neural networks and you want to speed up their computation, is to use operator fusion

```
where you can, because as you saw, it took a few seconds for our model to just move
from the CPU,
which is the default to the GPU. So we've got our architecture here. But of course,
we know that
this potentially is wrong. And how would we find that out? Well, we could find the
right hidden
unit shape or we could find that it's wrong by passing some dummy data through our
model. So
that's one of my favorite ways to troubleshoot a model. Let's in the next video
pass some dummy
data through our model and see if we've implemented the forward pass correctly. And
also check the
input and output shapes of each of our layers. I'll see you there. In the last
video, we replicated
the tiny VGG architecture from the CNN explainer website, very similar to the model
that we built
in section 03. But this time, we're using color images instead of grayscale images.
And we did
it quite a bit faster than what we previously did, because we've already covered
it, right?
And you've had some experience now building pilotage models from scratch.
So we're going to pick up the pace when we build our models. But let's now go and
try a dummy
forward pass to check that our forward method is working correctly and that our
input and output
shapes are correct. So let's create a new heading. Try a forward pass on a single
image. And this
is one of my favorite ways to test the model. So let's first get a single image.
Get a single
image. We want an image batch. Maybe we get an image batch, get a single image
batch, because
we've got images that are batches already image batch. And then we'll get a label
batch. And we'll
go next, it a train data loader. Simple. That's the data loader that we're working
with for now.
And then we'll check image batch dot shape and label batch dot shape.
Wonderful. And now let's see what happens. Try a forward pass.
Oh, I spelled single wrong up here. Try a forward pass. We could try this on a
single image trying
it on a same batch will result in similar results. So let's go model zero. And
we're just going to
pass it in the image batch and see what happens. Oh, no. Of course, we get that
input type,
torch float tensor and wait type torch CUDA float tensor should be the same or
input should be.
```

Section 231: Main Topics

Key Topics:

- So this is on the CPU, the image batch, whereas our model is, of course, on the target device
- It's on the CUDA device, of course
- Of course we do, because we've now got different shapes

▶ 📄 Click to view detailed content

```
So we've got tensors on a different device, right? So this is on the CPU, the image
batch,
whereas our model is, of course, on the target device. So we've seen this error a
number of times.
Let's see if this fixes it. Oh, we get an other error. And we kind of expected this
type of error.
We've got runtime error amount one and mat two shapes cannot be multiplied. 32. So
that looks
like the batch size 2560 and 10. Hmm, what is 10? Well, recall that 10 is the
number of hidden
units that we have. So this is the size here. That's 10 there. So it's trying to
multiply
a matrix of this size by this size. So 10 has got something going on with it. We
need to get
these two numbers, the middle numbers, to satisfy the rules of matrix
multiplication,
because that's what happens in our linear layer. We need to get these two numbers
the same.
And so our hint and my trick is to look at the previous layer. So if that's our
batch size,
where does this value come from? Well, could it be the fact that a tensor of this
size goes
through the flatten layer? Recall that we have this layer up here. So we've printed
out the shape
here of the conv block, the output of conv block one. Now this shape here is the
output of conv
block two. So we've got this number, the output of conv block one, and then the
output of conv
block two. So that must be the input to our classifier layer. So if we go 10 times
16 times 16,
what do we get? 2560. Beautiful. So we can multiply our hitting units 10 by 16 by
16, which is the
shape here. And we get 2560. Let's see if that works. We'll go up here, times 16
times 16.
And let's see what happens. We'll rerun the model, we'll rerun the image batch, and
then we'll pass
it. Oh, look at that. Our model works. Or the shapes at least line up. We don't
know if it works
yet. We haven't started training yet. But this is the output size. We've got the
output. It's on
the CUDA device, of course. But we've got 32 samples with three numbers in each.
Now these are going
to be as good as random, because we haven't trained our model yet. We've only
initialized it here
with random weights. So we've got 32 or a batch worth of random predictions on 32
images.
So you see how the output shape here three corresponds to the output shape we set
```

up here.
Output shape equals length class names, which is exactly the number of classes that we're dealing
with. But I think our number is a little bit different to what's in the CNN explainer 1616.
How did they end up with 1313? You know what? I think we got one of these numbers wrong,
kernel size, stride, padding. Let's have a look. Jump into here. If we wanted to truly replicate it,
is there any padding here? I actually don't think there's any padding here. So what if we go back
here and see if we can change this to zero and change this to zero? Zero. I'm not sure if this
will work, by the way. If it doesn't, it's not too bad, but we're just trying to line up the shapes
with the CNN explainer to truly replicate it. So the output of the COM Block 1 should be 30-30-10.
What are we working with at the moment? We've got 32-32-10. So let's see if removing the padding
from our convolutional layers lines our shape up with the CNN explainer. So I'm going to rerun
this, rerun our model. I've set the padding to zero on all of our padding hyper parameters.
Oh, and we get another error. We get another shape error. Of course we do,
because we've now got different shapes. Wow, do you see how often that these errors come up?
Trust me, I spend a lot of time troubleshooting these shape errors. So we now have to line up
these shapes. So we've got 13-13-10. Now does that equal 16-90? Let's try it out. 13-13-10.
16-90. Beautiful. And do our shapes line up with the CNN explainer? So we've got 30-30-10.
Remember, these are in PyTorch. So color channels first, whereas this is color channels last. So
yeah, we've got the output of our first COM Block is lining up here. That's correct.
And then same with the second block. How good is that? We've officially replicated the CNN explainer
model. So we can take this value 13-13-10 and bring it back up here. 13-13-10. Remember,
hidden units is 10. So we're just going to multiply it by 13-13. You could calculate
these shapes by hand, but my trick is I like to let the error codes give me a hint of where to go.
And boom, there we go. We get it working again. Some shape troubleshooting on the fly. So now
we've done a single forward pass on the model. We can kind of verify that our data at least flows
through it. What's next? Well, I'd like to show you another little package that I like to use
to also have a look at the input and output shapes of my model. And that is called Torch Info. So
you might want to give this a shot before we go into the next video. But in the next video,
we're going to see how we can use Torch Info to print out a summary of our model. So we're
going to get something like this. So this is how beautifully easy Torch Info is to

```
use. So
give that a shot, install it into Google CoLab and run it in a cell here. See if
you can get
```

Section 232: Main Topics

Key Topics:

- But I'd like to introduce to you one of my favorite packages for finding out information from a PyTorch model
- And of course, you could get this type of output from almost any PyTorch model
- And of course, we could change these values here if we wanted to, 24 to 24

▶ 📄 Click to view detailed content

```
something similar to this output for our model zero. And I'll see you in the next
video. We'll try
that together. In the last video, we checked our model by doing a forward pass on a
single batch.
And we learned that our forward method so far looks like it's intact and that we
don't get any
shape errors as our data moves through the model. But I'd like to introduce to you
one of my
favorite packages for finding out information from a PyTorch model. And that is
Torch Info.
So let's use Torch Info to get an idea of the shapes going through our model. So
you know how
much I love doing things in a programmatic way? Well, that's what Torch Info does.
Before,
we used print statements to find out the different shapes going through our model.
And I'm just going to comment these out in our forward method so that when we run
this later on
during training, we don't get excessive printouts of all the shapes. So let's see
what Torch Info
does. And in the last video, I issued a challenge to give it a go. It's quite
straightforward of
how to use it. But let's see it together. This is the type of output we're looking
for from our
tiny VGG model. And of course, you could get this type of output from almost any
PyTorch model.
But we have to install it first. And as far as I know, Google CoLab doesn't come
with Torch Info
by default. Now, you might as well try this in the future and see if it works. But
yeah, I don't
get this module because my Google CoLab instance doesn't have an install. No
problem with that.
Let's install Torch Info here. Install Torch Info and then we'll import it if it's
available.
```

So we're going to try and import Torch Info. If it's already installed, we'll import it.
And then if it doesn't work, if that try block fails, we're going to run pip install Torch Info.
And then we will import Torch Info. And then we're going to run down here from Torch Info,
import summary. And then if this all works, we're going to get a summary of our model. We're going
to pass it in model zero. And we have to put in an input size here. Now that is an example of the
size of data that will flow through our model. So in our case, let's put in an input size of 1,
3, 64, 64. So this is an example of putting in a batch of one image. You could potentially
put in 32 here if you wanted, but let's just put in a batch of a singular image. And of course,
we could change these values here if we wanted to, 24 to 24. But what you might notice is that if
it doesn't get the right input size, it produces an error. There we go. So just like we got before
when we printed out our input sizes manually, we get an error here. Because what Torch Info
behind the scenes is going to do is it's going to do a forward pass on whichever model you pass
it with an input size of whichever input size you give it. So let's put in the input size that
our model was built for. Wonderful. So what Torch Info gives us is, oh, excuse me, we didn't
comment out the printouts before. So just make sure we've commented out these printouts in the
forward method of our 20 VGG class. So I'm just going to run this, then we run that, run that,
just to make sure everything still works. We'll run Torch Info. There we go. So no printouts
from our model, but this is, look how beautiful this is. I love how this prints out. So we have
our tiny VGG class, and then we can see it's comprised of three sequential blocks. And then
inside those sequential blocks, we have different combinations of layers. We have some conv layers,
some relu layers, some max pool layers. And then the final layer is our classification layer
with a flatten and a linear layer. And we can see the shapes changing throughout our model.
As our data goes in and gets manipulated by the various layers. So are these in line with
the CNN explainer? So if we check this last one, we've already verified this before.
And we also get some other helpful information down here, which is total params. So you can see
that each of these layers has a different amount of parameters to learn. Now, recall that a parameter
is a value such as a weight or a bias term within each of our layers, which starts off as a random
number. And the whole goal of deep learning is to adjust those random numbers to better represent
our data. So in our case, we have just over 8000 total parameters. Now this is

```
actually quite small.
In the future, you'll probably play around with models that have a million
parameters or more.
And models now are starting to have many billions of parameters. And we also get
some
information here, such as how much the model size would be. Now this would be very
helpful,
depending on where we had to put our model. So what you'll notice is that as a
model gets larger,
as more layers, it will have more parameters, more weights and bias terms that can
be adjusted
to learn patterns and data. But its input size and its estimated total size would
definitely get
bigger as well. So that's just something to keep in mind if you have size
constraints in terms of
storage in your future applications. So ours is under a megabyte, which is quite
small. But you
```

Section 233: Main Topics

Key Topics:

- It should work with most of your PyTorch models
- And of course, for the train step and for the test step, each of them respectively are going to take a training data loader
- Of course, we want them to be respectively taken their own data loader

▶ 📄 Click to view detailed content

```
might find that some models in the future get up to 500 megabytes, maybe even over
a gigabyte.
So just keep that in mind for going forward. And that's the crux of torch info, one
of my
favorite packages, just gives you an idea of the input and output shapes of each of
your layers.
So you can use torch info wherever you need. It should work with most of your
PyTorch models.
Just be sure to pass it in the right input size. You can also use it to verify like
we did before,
if the input and output shapes are correct. So check that out, big shout out to
Tyler Yup,
and everyone who's created the torch info package. Now in the next video, let's
move towards training
our tiny VGG model. We're going to have to create some training and test functions.
If you want to
jump ahead, we've already done this. So I encourage you to go back to section 6.2
in the
functionalizing training and test loops. And we're going to build functions very
```

similar to this,
but for our custom data set. So if you want to replicate these functions in this notebook,
give that a go. Otherwise, I'll see you in the next video and we'll do it together.
How'd you go? Did you give it a shot? Did you try replicating the train step and the test step
function? I hope you did. Otherwise, let's do that in this video, but this time we're going to do
it for our custom data sets. And what you'll find is not much, if anything, changes, because
we've created our train and test loop functions in such a way that they're generic. So we want
to create a train step function. And by generic, I mean they can be used with almost any model and
data loader. So train step is takes in a model and data loader and trains the model on the data
loader. And we also want to create another function called test step, which takes in
a model and a data loader and other things and evaluates the model on the data loader. And of course,
for the train step and for the test step, each of them respectively are going to take a training
data loader. I just might make this a third heading so that our outline looks nice, beautiful.
Section seven is turning out to be quite a big section. Of course, we want them to be
respectively taken their own data loader. So train takes in the train data loader, test takes in the
test data loader. Without any further ado, let's create the train step function. Now we've seen
this one in the computer vision section. So let's see what we can make here. So we need a train
step, which is going to take in a model, which will be a torch and then dot module. And we want
it also to take in a data loader, which will be a torch dot utils dot data dot data loader.
And then it's going to take in a loss function, which is going to be a torch and then
dot module as well. And then it's going to take in an optimizer, which is going to be torch
opt in dot optimizer. Wonderful. And then what do we do? What's the first thing that we do in
a training step? Well, we put the model in train mode. So let's go model dot train.
Then what shall we do next? Well, let's set up some evaluation metrics, one of them being loss
and one of them being accuracy. So set up train loss and train accuracy values. And we're going
to accumulate these per batch because we're working with batches. So we've got train loss
and train act equals zero, zero. Now we can loop through our data loader. So let's write loop through
data loader. And we'll loop through each of the batches in this because we've batchified our
data loader. So for batch x, y, in enumerate data loader, we want to send the data to the target
device. So we could even put that device parameter up here. Device equals device. We'll set that

```
to device by default. And then we can go x, y equals x dot two device. And y dot
two device.
Beautiful. And now what do we do? Well, remember the pie torch, the unofficial pie
torch optimization
song, we do the forward pass. So y pred equals model om x. And then number two is
we calculate the
last. So calculate the loss. Let's go loss equals loss function. And we're going to
pass it in
y pred y. We've done this a few times now. So that's why we're doing it a little
bit faster.
So I hope you noticed that the things that we've covered before, I'm stepping up
the pace a bit.
So it might be a bit of a challenge, but that's all right, you can handle it. And
then, so that's
accumulating the loss. So we're starting from zero up here. And then each batch,
we're doing a forward
pass, calculating the loss, and then adding it to the overall train loss. And so
we're going to
optimize a zero grad. So zero, the gradients of the optimizer for each new batch.
And then we're
going to perform back propagation. So loss backwards. And then five, what do we do?
Optimize a step,
step, step. Wonderful. Look at that. Look at us coding a train loop in a minute or
so.
Now, let's calculate the accuracy and accumulate it. Calculate the, you notice that
we don't have
an accuracy function here. That's because accuracy is quite a straightforward
metric to calculate.
So we'll first get the, the y pred class, because this is going to output model
logits.
```

Section 234: Main Topics

Key Topics:

- Well, of course, we do the forward pass, forward pass

▶ 📄 Click to view detailed content

```
As we've seen before, the raw output of a model is logits. So to get the class,
we're going to take
the arg max torch dot softmax. So we'll get the prediction probabilities of y pred,
which is the
raw logits, what we've got up here, across dimension one, and then also across
dimension one here.
Beautiful. So that should give us the labels. And then we can find out if this is
wrong by
checking it later on. And then we're going to create the accuracy by taking the y
pred class,
checking for a quality with the right labels. So this is going to give us how many
```

of these
values equal true. And we want to take the sum of that, take the item of that, which is just a
single integer. And then we want to divide it by the length of y pred. So we're just getting the
total number that are right, and dividing it by the length of samples. So that's the formula for
accuracy. Now we can come down here outside of the batch loop, we know that because we've got this
helpful line drawn here. And we can go adjust metrics to get the average loss and accuracy
per batch. So we're going to set train loss is equal to train loss, divided by the length of
the data loader. So the number of batches in total. And the train accuracy is the train
act, divided by the length of the data loader as well. So that's going to give us the average
loss and average accuracy per epoch across all batches. So train act. Now that's a pretty good
looking function to me for a train step. Do you want to take on the test step? So pause the video,
give it a shot, and you'll get great inspiration from this notebook here. Otherwise, we're going
to do it together in three, two, one, let's do the test step. So create a test step function.
So we want to be able to call these functions in an epoch loop. And that way, instead of writing
out training and test code for multiple different models, we just write it out once, and we can
call those functions. So let's create def test step, we're going to do model, which is going to be
if I could type torch and then module. And then we're going to do data loader,
which is torch utils dot data, that data loader, capital L there. And then we're going to just
pass in a loss function here, because we don't need an optimizer for the test function. We're
not trying to optimize anything, we're just trying to evaluate how our model did on the training
dataset. And let's put in the device here, why not? That way we can change the device if we need
to. So put model in a val mode, because we're going to be evaluating or we're going to be testing.
Then we can set up test loss and test accuracy values. So test loss and test act. We're going
to make these zero, we're going to accumulate them per batch. But before we go through the batch,
let's turn on inference mode. So this is behind the scenes going to take care of a lot of pie torch
functionality that we don't need. That's very helpful during training, such as tracking gradients.
But during testing, we don't need that. So loop through data loader or data batches.
And we're going to go for batch x, y in enumerate data loader. You'll notice that above, we didn't
actually use this batch term here. And we probably won't use it here either. But I just like to go
through and have that there in case we wanted to use it anyway. So send data to the

```
target device.
So we're going to go x, y equals x dot two device. And same with y dot two device.
Beautiful. And
then what do we do for an evaluation step or a test step? Well, of course, we do
the forward pass,
forward pass. And we're going to, let's call these test pred logits and get the raw
outputs of our
model. And then we can calculate the loss on those raw outputs, calculate the loss.
We get the loss
is equal to loss function on test pred logits versus y. And then we're going to
accumulate the
loss. So test loss plus equals loss dot item. Remember, item just gets a single
integer from
whatever term you call it on. And then we're going to calculate the accuracy. Now
we can do this
exactly how we've done for the training data set or the training step. So test pred
labels,
we're going to, you don't, I just want to highlight the fact that you actually
don't need to take
the softmax here, you could just take the argmax directly from this. The reason why
we take the
softmax. So you could do the same here, you could just directly take the argmax of
the logits. The
reason why we get the softmax is just for completeness. So if you wanted the
prediction probabilities,
you could use torch dot softmax on the prediction logits. But it's not 100%
necessary to get the
same values. And you can test this out yourself. So try this with and without the
softmax and
see if you get the same results. So we're going to go test accuracy. Plus equals,
now we'll just
create our accuracy calculation on the fly test pred labels. We'll check for
equality on the y,
then we'll get the sum of that, we'll get the item of that, and then we'll divide
that by the
length of the test pred labels. Beautiful. So it's going to give us accuracy per
batch. And so now
we want to adjust the metrics to get average loss and accuracy per batch. So test
loss equals
```

Section 235: Main Topics

Key Topics:

- In the last video, I issued you the challenge to combine our train step function, as well as our test step function together in their own function so that we could just call one function that calls both of these and train a model and evaluate it, of course

▶ 📄 Click to view detailed content

```
test loss divided by length of the data loader. And then we're going to go test,
ac equals test, act divided by length of the data loader. And then finally, we're
going to
return the test loss, not lost, and test accuracy. Look at us go. Now, in previous
videos, that took
us, or in previous sections, that took us a fairly long time. But now we've done it
in about 10
minutes or so. So give yourself a pat in the back for all the progress you've been
making.
But now let's in the next video, we did this in the computer vision section as
well. We created,
do we create a train function? Oh, no, we didn't. But we could. So let's create a
function to
functionize this. We want to train our model. I think we did actually. Deaf train,
we've done
so much. I'm not sure what we've done. Oh, okay. So looks like we might not have.
But in the next
video, give yourself this challenge, create a function called train that combines
these two
functions and loops through them both with an epoch range. So just like we've done
here in the
previous notebook, can you functionize this? So just this step here. So you'll need
to take in a
number of epochs, you'll need to take in a train data loader and a test data
loader, a model, a
loss function, an optimizer, and maybe a device. And I think you should be pretty
on your way to
all the steps we need for train. So give that a shot. But in the next video, we're
going to create
a function that combines train step and test step to train a model. I'll see you
there.
How'd you go? In the last video, I issued you the challenge to combine our train
step function,
as well as our test step function together in their own function so that we could
just call
one function that calls both of these and train a model and evaluate it, of course.
So let's now do that together. I hope you gave it a shot. That's what it's all
about. So we're
going to create a train function. Now the role of this function is going to, as I
said, combine
train step and test step. Now we're doing all of this on purpose, right, because we
want to not
have to rewrite all of our code all the time. So we want to be functionalizing as
many things as
possible, so that we can just import these later on, if we wanted to train more
models and just
leverage the code that we've written before, as long as it works. So let's see if
it does,
we're going to create a train function. I'm going to first import TQDM, TQDM.auto,
because I'd like to get a progress bar while our model is training. There's nothing
quite like
watching a neural network train. So step number one is we need to create a train
function that takes
in various model parameters, plus optimizer, plus data loaders, plus a loss
function. A whole
bunch of different things. So let's create def train. And I'm going to pass in a
```

```
model here,
which is going to be torch and then dot module. You'll notice that the inputs of
this are going
to be quite similar to our train step and test step. I don't actually need that
there.
So we also want a train data loader for the training data, torch dot utils dot data
dot data
loader. And we also want a test data loader, which is going to be torch dot utils
dot data
dot data loader. And then we want an optimizer. So the optimizer will only be used
with our
training data set, but that's okay. We can take it as an input of the miser. And
then we want a
loss function. This will generally be used for both our training and testing step.
Because that's
what we're combining here. Now, since we're working with multi class
classification,
I'm going to set our loss function to be a default of an n dot cross entropy loss.
Then I'm going to get epochs. I'm going to set five, we'll train for five epochs by
default.
And then finally, I'm going to set the device equal to the device. So what do we
get wrong here?
That's all right. We'll just keep coding. We'll ignore these little red lines. If
they
stay around, we'll come back to them. So step number two, I'm going to create. This
is a step
you might not have seen, but I'm going to create an empty results dictionary. Now,
this is going
to help us track our results. Do you recall in a previous notebook, we outputted a
model dictionary
for how a model went. So if we look at model one results, yeah, we got a dictionary
like this.
So I'd like to create one of these on the fly, but keep track of the result every
epoch. So what
was the loss on epoch number zero? What was the accuracy on epoch number three? So
we'll show you
how I'll do that. We can use a dictionary and just update that while our model
trains.
So results, I want to keep track of the train loss. So we're going to set that
equal to an empty
list and just append to it. I also want to keep track of the train accuracy. We'll
set that as
an empty list as well. I also want to keep track of the test loss. And I also want
to keep track
of the test accuracy. Now, you'll notice over time that these, what you can track
is actually
very flexible. And what your functions can do is also very flexible. So this is not
the gold
```

Section 236: Main Topics

Key Topics:

- And of course, you can code that out
- So the data loader here is of course going to be the train data loader
- And the train function, of course, is going to call out our train step function and our test step function

▶ 📄 Click to view detailed content

```
standard of doing anything by any means. It's just one way that works. And you'll
probably find in
the future that you need different functionality. And of course, you can code that
out. So let's
now loop through our epochs. So for epoch in TQDM, let's create a range of our
epochs above.
And then we can set the train loss. Have I missed a comma up here somewhere?
Type annotation not supported for that type of expression. Okay, that's all right.
We'll just leave
that there. So we're going to go train loss and train act, recall that our train
step function
that we created in the previous video, train step returns our train loss and train
act. So as I
said, I want to keep track of these throughout our training. So I'm going to get
them from train
step. Then for each epoch in our range of epochs, we're going to pass in our model
and perform a
training step. So the data loader here is of course going to be the train data
loader. The
loss function is just going to be the loss function that we pass into the train
function.
And then the optimizer is going to be the optimizer. And then the device is going
to be device.
Beautiful. Look at that. We just performed a training step in five lines of code.
So let's keep pushing forward. It's telling us we've got a whole bunch of different
things here.
Epox is not defined. Maybe we just have to get rid of this. We can't have the type
annotation here.
And that'll that'll stop. That'll stop Google Colab getting angry at us. If it does
anymore,
I'm just going to ignore it for now. Epox. Anyway, we'll leave it at that. We'll
find out if there's
an error later on. Test loss. You might be able to find it before I do. So test
step. We're going
to pass in the model. We're going to pass in a data loader. Now this is going to be
the test data
loader. Look at us go. Grading training and test step functions, loss function. And
then we don't
need an optimizer. We're just going to pass in the device. And then behind the
scenes,
both of these functions are going to train and test our model. How cool is that? So
still within
the loop. This is important. Within the loop, we're going to have number four is
we're going to
print out. Let's print out what's happening. Print out what's happening. We can go
print.
And we'll do a fancy little print statement here. We'll get the epoch. And then we
```

will get
the train loss, which will be equal to the train loss. We'll get that to, let's go four decimal places. How about that? And then we'll get the train accuracy, which is going to be the
train act. We'll get that to four, maybe three decimal of four, just for just so it looks nice.
It looks aesthetic. And then we'll go test loss. We'll get that coming out here. And we'll pass
in the test loss. We'll get that to four decimal places as well. And then finally, we'll get the
test accuracy. So a fairly long print statement here. But that's all right. We'd like to see how
our model is doing while it's training. Beautiful. And so again, still within the epoch, we want to
update our results dictionary so that we can keep track of how our model performed over time.
So let's pass in results. We want to update the train loss. And so this is going to be this.
And then we can append our train loss value. So this is just going to expend the list in here
with the train loss value, every epoch. And then we'll do the same thing on the train accuracy,
append train act. And then we'll do the same thing again with test loss dot append test loss.
And then we will finally do the same thing with the test accuracy test accuracy. Now,
this is a pretty big function. But this is why we write the code now so that we can use it
multiple times later on. So return the field results at the end of the epoch. So outside the
epochs loop. So our loop, we're outside it now. Let's return results. Now, I've probably got an
error somewhere here and you might be able to spot it. Okay, train data loader. Where do we get
that invalid syntax? Maybe up here, we don't have a comma here. Was that the issue the whole time?
Wonderful. You might have seen that I'm completely missed that. But we now have a train function
to train our model. And the train function, of course, is going to call out our train step
function and our test step function. So what's left to do? Well, nothing less than train and
evaluate model zero. So our model is way back up here. How about in the next video, we leverage
our functions, namely just the train function, because it's going to call our train step function
and our test step function and train our model. So I'm going to encourage you to give that a go.
You're going to have to go back to the workflow. Maybe you'll maybe already know this.
So what have we done? We've got our data ready and we turned it into tenses using a combination
of these functions. We've built and picked a model while we've built a model, which is the
tiny VGG architecture. Have we created a loss function yet? I don't think we have or an optimizer.
I don't think we've done that yet. We've definitely built a training loop though.

```
We aren't using torch metrics. We're just using accuracy, but we could use this if
we want.
```

Section 237: Main Topics

Key Topics:

- Now, if we refer back to the PyTorch workflow, I issued you the challenge in the last video to try and create a loss function and an optimizer
- And then, of course, we're going to send the target model to the target device
- Now, of course, the optimizer is one of the hyper parameters that you can set for your model, and a hyper parameter being a value that you can set yourself

▶ 📄 Click to view detailed content

```
We haven't improved through experimentation yet, but we're going to try this later
on and
then save and reload the model. We've seen this before. So I think we're up to
picking a loss
function and an optimizer. So give that a shot. In the next video, we're going to
create a loss
function and an optimizer and then leverage the functions we've spent in the last
two videos
creating to train our first model model zero on our own custom data set. This is
super exciting.
I'll see you in the next video.
Who's ready to train and evaluate model zero? Put your hand up.
I definitely am. So let's do it together. We're going to start off section 7.7 and
we're going
to put in train and evaluate model zero, our baseline model on our custom data set.
Now,
if we refer back to the PyTorch workflow, I issued you the challenge in the last
video to try and
create a loss function and an optimizer. I hope you gave that a go, but we've
already built a
training loop. So we're going to leverage our training loop functions, namely
train, train step
and test step. All we need to do now is instantiate a model, choose a loss function
and an optimizer
and pass those values to our training function. So let's do that. All right, this
is so exciting.
Let's set the random seeds. I'm going to set torch manual seed 42 and torch cuda
manual seed 42.
Now remember, I just want to highlight something. I read an article the other day
about not using
random seeds. The reason why we are using random seeds is for educational purposes.
So to try and
get our numbers on my screen and your screen as close as possible, but in practice,
```

you quite
often don't use random seeds all the time. The reason why is because you want your models performance
to be similar regardless of the random seed that you use. So just keep that in mind going forward.
We're using random seeds to just exemplify how we can get similar numbers on our page. But
ideally, no matter what the random seed was, our models would go in the same direction.
That's where we want our models to eventually go. But we're going to train for five epochs.
And now let's create a recreate an instance of tiny VGG. We can do so because we've created the
tiny VGG class. So tiny VGG, which is our model zero. We don't have to do this, but we're going
to do it any later. So we've got all the code in one place, tiny VGG. What is our input shape
going to be? That is the number of color channels of our target images. And because we're dealing
with color images, we have an input shape of three. Previously, we used an input shape of one to
deal with grayscale images. I'm going to set hidden units to 10 in line with the CNN explainer website.
And the output shape is going to be the number of classes in our training data set. And then,
of course, we're going to send the target model to the target device. So what do we do now?
Well, we set up a loss function and an optimizer, loss function, and optimizer.
So our loss function is going to be because we're dealing with multiclass classification,
and then cross entropy, if I could spell cross entropy loss. And then we're going to have an
optimizer. This time, how about we mix things up? How about we try the atom optimizer? Now,
of course, the optimizer is one of the hyper parameters that you can set for your model,
and a hyper parameter being a value that you can set yourself. So the parameters that we want to
optimize are our model zero parameters. And we're going to set a learning rate of 0.001. Now,
recall that you can tweet this learning rate, if you like, but I believe, did I just see that
the default learning rate of atom is 0.001? Yeah, there we go. So Adam's default learning rate is
one to the power of 10 to the negative three. And so that is a default learning rate for Adam.
And as I said, oftentimes, different variables in the pytorch library, such as optimizers,
have good default values that work across a wide range of problems. So we're just going to stick
with the default. If you want to, you can experiment with different values of this.
But now let's start the timer, because we want to time our models.
We're going to import from time it. We want to get the default timer class. And I'm going to
import that as timer, just so we don't have to type out default timer. So the start time is going
to be timer. This is going to just put a line in the sand of what the start time is

```
at this
particular line of code. It's going to measure that. And then we're going to train
model zero.
Now this is using, of course, our train function. So let's write model zero
results, and then
they wrote model one, but we're not up to there yet. So let's go train model equals
model zero.
And this is just the training function that we wrote in a previous video. And the
train data
is going to be our train data loader. And we've got train data loader simple,
because we're not
using data augmentation for model one. And then our test data loader is going to be
our test data
loader simple. And then we're going to set our optimizer, which is equal to the
optimizer we just
created. Friendly atom optimizer. And the loss function is going to be the loss
function that
```

Section 238: Main Topics

Key Topics:

- And of course, we could train our model for longer if we wanted to
- So that's why we're only training for five, maybe later on you train for 10, 20, tweak the learning rate, do a whole bunch of different things
- Of course, we'd like this number to go higher, and maybe it would if it trained for longer

▶ 📄 Click to view detailed content

```
we just created, which is an n cross entropy loss. Finally, we can send in epochs
is going to be
num epochs, which is what we set at the start of this video to five. And of course,
we could train
our model for longer if we wanted to. But the whole idea of when you first start
training a model
is to keep your experiments quick. So that's why we're only training for five,
maybe later on you
train for 10, 20, tweak the learning rate, do a whole bunch of different things.
But let's go
down here, let's end the timer, see how long our models took to train, and the
timer and print out
how long it took. So in a previous section, we created a helper function for this.
We're just going to simplify it in this section. And we're just going to print out
how long the
training time was. Total training time. Let's go n time minus start time. And then
we're going to go
point, we'll take it to three decimal places, hey, seconds, you ready to train our
```

first model,
our first convolutional neural network on our own custom data set on pizza, stake and sushi
images. Let's do it. You're ready? Three, two, one, no errors. Oh, there we go. Okay,
should this be trained data loader? Did you notice that? What is our trained data
taker's input? Oh, we're not getting a doc string. Oh, there we go. We want trained data
loader, data loader, and same with this, I believe. Let's try again. Beautiful. Oh, look at that
lovely progress bar. Okay, how's our model is training quite fast? Okay. All right, what do we
get? So we get an accuracy on the training data set of about 40%. And we get an accuracy on the
test data set of about 50%. Now, what's that telling us? It's telling us that about 50% of the time
our model is getting the prediction correct. But we've only got three classes. So even if our model
was guessing, it would get things right 33% of the time. So even if you just guessed pizza every
single time, because we only have three classes, if you guessed pizza every single time, you get
a baseline accuracy of 33%. So our model isn't doing too much better than our baseline accuracy.
Of course, we'd like this number to go higher, and maybe it would if it trained for longer.
So I'll let you experiment with that. But if you'd like to see some different methods of
improving a model, recall back in section number O two, we had an improving a model section,
improving a model. Here we go. So here's some things you might want to try.
We can improve a model by adding more layers. So if we come back to our tiny VGG architecture,
right up here, we're only using two convolutional blocks. Perhaps you wanted to add in a convolutional
block three. You can also add more hidden units. Right now we're using 10 hidden units. You might
want to double that and see what happens. Fitting for longer. This is what we just spoke about.
So right now we're only fitting for five epochs. So if you maybe wanted to try double that again,
and then even double that again, changing the activation functions. So maybe relu is not the
ideal activation function for our specific use case. Change the learning rate. We've spoken
about that before. So right now our learning rate is 0.001 for Adam, which is the default.
But perhaps there's a better learning rate out there. Change the loss function. This is probably not
in our case, not going to help too much because cross entropy loss is a pretty good loss for
multi class classification. But these are some things that you could try these first three,
especially. You could try quite quickly. You could try doubling the layers. You could try
adding more hidden units. And you could try fitting for longer. So I'd give that a shot.

```
But in the next video, we're going to take our model zero results, which is a
dictionary or at
least it should be. And we're going to plot some loss curves. So this is a good way
to inspect how
our model is training. Yes, we've got some values here. Let's plot these in the
next video. I'll see you there.
In the last video, we trained our first convolutional neural network on custom
data. So you should be
very proud of that. That is no small feat to take our own data set of whatever we
want
and train apply to its model on it. However, we did find that it didn't perform as
well as we'd
like it to. We also highlighted a few different things that we could try to do to
improve it.
But now let's plot our models results using a loss curve. So I'm going to write
another heading
down here. We'll go, I believe we're up to 7.8. So plot the loss curves of model
zero. So what
is a loss curve? So I'm going to write down here, a loss curve is a way of tracking
your models
progress over time. So if we just looked up Google and we looked up loss curves,
oh, there's a great guide by the way. I'm going to link this. But I'd rather if and
doubt code it
out than just look at guides. Yeah, loss curves. So yeah, loss over time. So
there's our loss value
on the left. And there's say steps, which is epochs or batches or something like
that.
Then we've got a whole bunch of different loss curves over here. Essentially, what
we want it
to do is go down over time. So that's the idea loss curve. Let's go back down here.
```

Section 239: Main Topics

Key Topics:

- And of course, these lists would be longer if we train for more epochs
- And then of course, if the accuracy is going higher, then the loss is going to come down

▶ 📄 Click to view detailed content

```
And a good guide for different loss curves can be seen here. We're not going to go
through that
just yet. Let's focus on plotting our own models, loss curves, and we can inspect
those.
Let's get the model keys. Get the model zero results keys. I'm going to type in
model zero
results dot keys because it's a dictionary. Let's see if we can write some code to
plot these
```

values here. So yeah, over time. So we have one value for train loss, train,
act, test loss, and test act for every epoch. And of course, these lists would be longer if we
train for more epochs. But let's just how about we create a function called def plot loss curves,
which will take in a results dictionary, which is of string and a list of floats. So this just
means that our results parameter here is taking in a dictionary that has a string as a key.
And it contains a list of floats. That's what this means here. So let's write a doc string
plots training curves of a results dictionary. Beautiful. And so we're in this section of our
workflow, which is kind of like a, we're kind of doing something similar to TensorBoard, what it
does. I'll let you look into that if you want to. Otherwise, we're going to see it later on.
But we're really evaluating our model here. Let's write some plotting code. We're going to use map plot
lib. So we want to get the lost values of the results dictionary. So this is training and test.
Let's set loss equal to results train loss. So this is going to be the loss on the training
data set. And then we'll create the test loss, which is going to be, well, index on the results
dictionary and get the test loss. Beautiful. Now we'll do the same and we'll get the accuracy.
Get the accuracy values of the results dictionary. So training and test.
Then we're going to go accuracy equals results. This will be the training accuracy train
act and accuracy. Oh, we'll call this test accuracy actually test accuracy equals results test act.
Now let's create a number of epochs. So we want to figure out how many epochs we did. We can do
that by just counting the length of this value here. So figure out how many epochs there were.
So we'll set epochs equal to a range because we want to plot it over time. Our models results
over time. That's that's the whole idea of a loss curve. So we'll just get the the length of
our results here. And we'll get the range. So now we can set up a plot.
Let's go PLT dot figure. And we'll set the fig size equal to something nice and big because
we're going to do four plots. We want one for maybe two plots, one for the loss, one for the accuracy.
And then we'll go plot the loss. PLT dot subplot. We're going to create one row, two columns,
and index number one. We want to put PLT dot plot. And here's where we're going to plot the
training loss. So we get that a label of train loss. And then we'll add another plot with epochs
and test loss. The label here is going to be test loss. And then we'll add a title, which will be
loss PLT. Let's put a label on the X, which will be epochs. So we know how many steps we've done.
This plot over here, loss curves, it uses steps. I'm going to use epochs. They mean almost the

```
same thing. It depends on what scale you'd like to see your loss curves. We'll get
a legend as well
so that we are the labels appear. Now we're going to plot the accuracy. So PLT dot
subplot.
Let's go one, two, and then index number two that this plot's going to be on PLT
dot plot.
We're going to go epochs accuracy. And the label here is going to be train
accuracy.
And then we'll get on the next plot, which is actually going to be on the same
plot.
We'll put the test accuracy. That way we have the test accuracy and the training
accuracy side
by side, test accuracy same with the train loss and train, sorry, test loss. And
then we'll give
our plot a title. This plot is going to be accuracy. And then we're going to give
it an
X label, which is going to be epochs as well. And then finally, we'll get the plot,
but legend,
a lot of plotting code here. But let's see what this looks like. Hey, if we've done
it all right,
we should be able to pass it in a dictionary just like this and see some nice plots
like this.
Let's give it a go. And I'm going to call plot loss curves. And I'm going to pass
in model 0 results.
All righty then. Okay. So that's not too bad. Now, why do I say that? Well, because
we're
looking here for mainly trends, we haven't trained our model for too long.
Quantitatively, we know
that our model hasn't performed at the way we'd like it to do. So we'd like the
accuracy on both
the train and test data sets to be higher. And then of course, if the accuracy is
going higher,
then the loss is going to come down. So the ideal trend for a loss curve is to go
down from
the top left to the bottom right. In other words, the loss is going down over time.
So that's,
the trend is all right here. So potentially, if we train for more epochs, which I'd
encourage
you to give it a go, our model's loss might get lower. And the accuracy is also
trending in the
```

Section 240: Main Topics

Key Topics:

- So this is Google's testing and debugging and machine learning guide
- So in our case, if we go back to our loss curves, of course, we want this to be lower, and we want our accuracy to be higher
- And so two of the biggest problems in machine learning is trying to underfitting

▶ 📄 Click to view detailed content

```
right way. Our accuracy, we want it to go up over time. So if we train for more
epochs, these curves
may continue to go on. Now, they may not, they, you never really know, right? You
can guess these
things. But until you try it, you don't really know. So in the next video, we're
going to have a
look at some different forms of loss curves. But before we do that, I'd encourage
you to go through
this guide here, interpreting loss curves. So I feel like if you just search out
loss curves,
you're going to find Google's guide, or you could just search interpreting loss
curves.
Because as you'll see, there's many different ways that loss curves can be
interpreted. But the ideal
trend is for the loss to go down over time, and metrics like accuracy to go up over
time.
So in the next video, let's cover a few different forms of loss curves, such as the
ideal loss
curve, what it looks like when your model's underfitting, and what it looks like
when your
model's overfitting. And if you'd like to have a primer on those things, I'd read
through this
guide here. Don't worry too much if you're not sure what's happening. We're going
to cover a bit
more about loss curves in the next video. I'll see you there. In the last video, we
looked at our
model's loss curves, and also the accuracy curves. And a loss curve is a way to
evaluate a model's
performance over time, such as how long it was training for. And as you'll see, if
you Google
some images of loss curves, you'll see many different types of loss curves. They
come in all
different shapes and sizes. And there's many different ways to interpret loss
curves. So
this is Google's testing and debugging and machine learning guide. So I'm going to
set this as
actually curriculum for this section. So we're up to number eight. Let's have a
look at what should
an ideal loss curve look like. So we'll just link that in there. Now, loss curve,
I'll just
rewrite here, is a loss curve is, I'll just make some space. A loss curve is one of
the most
helpful ways to troubleshoot a model. So the trend of a loss curve, you want it to
go down over time,
and the trend typically of an evaluation metric, like accuracy, you want it to go
up over time.
So let's go into the keynote, loss curves. So a way to evaluate your model's
performance over time.
These are three of the main different forms of loss curve that you'll face. But
again,
there's many different types as mentioned in here, interpreting loss curves.
Sometimes you get it
going all over the place. Sometimes your loss will explode. Sometimes your metrics
will be
```

contradictory. Sometimes your testing loss will be higher than your training loss. We'll have a
look at what that is. Sometimes your model gets stuck. In other words, the loss doesn't reduce.
Let's have a look at some loss curves here in the case of underfitting, overfitting, and just
right. So this is the Goldilocks zone. Underfitting is when your model's loss on the training and
test data sets could be lower. So in our case, if we go back to our loss curves, of course,
we want this to be lower, and we want our accuracy to be higher. So from our perspective,
it looks like our model is underfitting. And we would probably want to train it for longer,
say, 10, 20 epochs to see if this train continues. If it keeps going down, it may stop underfitting.
So underfitting is when your loss could be lower. Now, the inverse of underfitting is called
overfitting. And so two of the biggest problems in machine learning is trying to
underfitting. So in other words, make your loss lower and also reduce overfitting. These are
both active areas of research because you always want your model to perform better,
but you also want it to perform pretty much the same on the training set as it does the test set.
And so overfitting would be when your training loss is lower than your testing loss. And why
would this be overfitting? So it means overfitting because your model is essentially learning the
training data too well. And that means the loss goes down on the training data set,
which is typically a good thing. However, this learning is not reflected in the testing data set.
So your model is essentially memorizing patterns in the training data set that don't
generalize well to the test data set. So this is where we come to the just right curve is that we
want, ideally, our training loss to reduce as much as our test loss. And quite often, you'll find
that the loss is slightly lower on the training set than it is on the test set. And that's just
because the model is exposed to the training data, and it's never seen the test data before.
So it might be a little bit lower on the training data set than on the test data set.
So underfitting, the model's loss could be lower. Overfitting, the model is learning the training
data too well. Now, this would be equivalent to say you were studying for a final exam,
and you just memorize the course materials, the training set. And when it came time to the final
exam, because you don't even memorize the course materials, you couldn't adapt those skills to
questions you hadn't seen before. So the final exam would be the test set. So that's overfitting.
The train loss is lower than the test loss. And just right, ideally, you probably won't see
loss curves this exact smooth. I mean, they might be a little bit jumpy. Ideally, your training loss

Section 241: Main Topics

Key Topics:

- And of course, there's more combinations of these
- They don't always work as with many things in machine learning
- Use transfer learning

▶ 📄 Click to view detailed content

```
and test loss go down at a similar rate. And of course, there's more combinations
of these. If
you'd like to see them, check out the Google's loss curve guide that you can check
that out there.
That's some extra curriculum. Now, you probably want to know how do you deal with
underfitting
and overfitting? Let's look at a few ways. We'll start with overfitting.
So we want to reduce overfitting. In other words, we want our model to perform just
as
well on the training data set as it does on the test data set. So one of the best
ways to
reduce overfitting is to get more data. So this means that our training data set
will be larger.
Our model will be exposed to more examples. And with us, in theory, it doesn't
always work.
These all come with a caveat, right? They don't always work as with many things in
machine learning.
So get more data, give your model more chance to learn patterns, generalizable
patterns in a
data set. You can use data augmentation. So make your models training data set
harder to learn.
So we've seen a few examples of data augmentation. You can get better data. So not
only more data,
perhaps the data that you're using isn't that the quality isn't that good. So if
you enhance the
quality of your data set, your model may be able to learn better, more
generalizable patterns and
in turn reduce overfitting. Use transfer learning. So we're going to cover this in
a later section
of the course. But transfer learning is taking one model that works, taking its
patterns that it's
learned and applying it to your own data set. So for example, I'll just go into the
Torch Vision
models library. Many of these models in here in Torch Vision, the models module,
have already
been trained on a certain data set and such as ImageNet. And you can take the
weights or the
patterns that these models have learned. And if they work well on an ImageNet data
set, which is
```

millions of different images, you can adjust those patterns to your own problem. And oftentimes
that will help with overfitting. If you're still overfitting, you can try to simplify your model.
Usually this means taking away things like extra layers, taking away more hidden units. So say you
had 10 layers, you might reduce it to five layers. Why does this? What's the theory behind this?
Well, if you simplify your model and take away complexity from your model, you're kind of telling
your model, hey, use what you've got. And you're going to have to, because you've only got five
layers now, you're going to have to make sure that those five layers work really well, because
you've no longer got 10. And the same for hidden units. Say you started with 100 hidden units per
layer, you might reduce that to 50 and say, hey, you had 100 before. Now use those 50 and make your
patterns generalizable. Use learning rate decay. So the learning rate is how much your optimizer
updates your model's weight every step. So learning rate decay is to decay the learning rate
over time. So you might look this up, you can look this up, go high torch, learning rate,
scheduling. So what this means is you want to decrease your learning rate over time.
Now, I know I'm giving you a lot of different things here, but you've got this keynote as a
reference. So you can come across these over time. So learning rate scheduling. So we might look
into here, do we have schedule, scheduler, beautiful. So this is going to adjust the learning rate
over time. So for example, at the start of when a model is training, you might want a higher learning
rate. And then as the model starts to learn patterns more and more and more, you might want to reduce
that learning rate over time so that your model doesn't update its patterns too much
in later epochs. So that's the concept of learning rate scheduling. At the closer you get to convergence,
the lower you might want to set your learning rate, think of it like this. If you're reaching
for a coin at the back of a couch, can we get an image of that coin at back of couch?
Images. So if you're trying to reach a coin in the cushions here, so the closer you get to that coin,
at the beginning, you might take big steps. But then the closer you get to that coin, the smaller
the step you might take to pick that coin out. Because if you take a big step when you're really
close to the coin here, the coin might fall down the couch. The same thing with learning rate decay.
At the start of your model training, you might take bigger steps as your model works its way
down the loss curve. But then you get closer and closer to the ideal position on the loss curve.
You might start to lower and lower that learning rate until you get right very

```
close to the end
and you can pick up the coin. Or in other words, your model can converge. And then
finally, use
early stopping. So if we go into an image, is there early stopping here? Early
stopping.
Loss curves early stopping. So what this means is that you stop. Yeah, there we go.
So there's
heaps of different guides early stopping with PyTorch. Beautiful. So what this
means is before
your testing error starts to go up, you keep track of your model's testing error.
And then you stop
your model from training or you save the weight or you save the patterns where your
model's loss
was the lowest. So then you could just set your model to train for an infinite
amount of training
```

Section 242: Main Topics

Key Topics:

- You can again tweak the learning rate
- Perhaps your learning rate is too high to begin with and your model doesn't learn very well
- So you can adjust the learning rate again, just like we discussed with reaching for that coin at the back of a couch

▶ 📄 Click to view detailed content

```
steps. And as soon as the testing error starts to increase for say 10 steps in a
row, you go back
to this point here and go, I think that was where our model was the best. And the
testing
error started to increase after that. So we're going to save that model there
instead of the model
here. So that's the concept of early stopping. So that's dealing with overfitting.
There are
other methods to deal with underfitting. So recall underfitting is when we have a
loss that isn't as
low as we'd like it to be. Our model is not fitting the data very well. So it's
underfitting.
So to reduce underfitting, you can add more layers slash units to your model.
You're trying to
increase your model's ability to learn by adding more layers or units. You can
again tweak the
learning rate. Perhaps your learning rate is too high to begin with and your model
doesn't learn
very well. So you can adjust the learning rate again, just like we discussed with
reaching for
```

that coin at the back of a couch. If your model is still underfitting, you can train for longer. So
that means giving your model more opportunities to look at the data. So more epochs, that just
means it's got looking at the training set over and over and over and over again and trying to
learn those patterns. However, you might find again, if you try to train for too long, your testing
error will start to go up. Your model might start overfitting if you train too long. So machine
learning is all about a balance between underfitting and overfitting. You want your model to fit quite
well. And so this is a great one. So you want your model to start fitting quite well. But then if you
try to reduce underfitting too much, you might start to overfit and then vice versa, right? If
you try to reduce overfitting too much, your model might underfit. So this is one of the
most fun dances in machine learning, the balance between overfitting and underfitting.
Finally, you might use transfer learning. So transfer learning helps with overfitting and
underfitting. Recall transfer learning is using a model's learned patterns from Ron problem and
adjusting them to your own. We're going to see this later on in the course. And then finally,
use less regularization. So regularization is holding your model back. So it's trying
to prevent overfitting. So if you do too much preventing of overfitting, in other words,
regularizing your model, you might end up underfitting. So if we go back, we have a look at the ideal
curves, underfitting. If you try to prevent underfitting too much, so increasing your model's
capability to learn, you might end up overfitting. And if you try to prevent overfitting too much,
you might end up underfitting. We are going for the just right section. And this is going to be a
balance between these two throughout your entire machine learning career. In fact, it's probably
the most prevalent area of research is trying to get models not to underfit, but also not to
overfit. So keep that in mind. A loss curve is a great way to evaluate your model's performance
over time. And a lot of what we do with the loss curves is try to work out whether our model is
underfitting or overfitting, and we're trying to get to this just right curve. We might not get
exactly there, but we want to keep trying getting as close as we can. So with that being said,
let's now build another model in the next video. And we're going to try a method to try and see if
we can use data augmentation to prevent our model from overfitting. Although that experiment
doesn't sound like the most ideal one we could do right now, because it looks like our model is
underfitting. So with your knowledge of what you've just learned in the previous

```
video,
how to prevent underfitting, what would you do to increase this model's capability
of learning
patterns in the training data set? Would you train it for longer? Would you add
more layers?
Would you add more hidden units? Have a think and we'll start building another
model in the next video.
Welcome back. In the last video, we covered the important concept of a loss curve
and how it can
give us information about whether our model is underfitting. In other words, our
model's loss
could be lower or whether it's overfitting. In other words, the training loss is
lower than the test
loss or far lower than the validation loss. That's another thing to note here is
that I put training
and test sets here. You could also do this with a validation data set and that the
just right,
the Goldilocks zone, is when our training and test loss are quite similar over
time.
Now, there was a fair bit of information in that last video, so I just wanted to
highlight
that you can get this all in section 04, which is the notebook that we're working
on. And then
if you come down over here, if we come to section 8, watch an ideal loss curve look
like we've got
underfitting, overfitting, just right, how to deal with overfitting. We've got a
few options here.
We've got how to deal with underfitting and then we've got a few options there. And
then if we
wanted to look for more, how to deal with overfitting. You could find a bunch of
resources here and then
how to deal with underfitting. You could find a bunch of resources here as well. So
that is a
very fine line, very fine balance that you're going to experience throughout all of
your
machine learning career. But it's time now to move on. We're going to move on to
creating
```

Section 243: Main Topics

Key Topics:

- So this is in line with our PyTorch workflow, trying a model and trying another one and trying another one, so and so over again
- We're going to use the trivial augment data augmentation, create training transform, which is, as we saw in a previous video, what PyTorch the PyTorch team have recently used to train their state-of-the-art computer vision models

- Our models aren't going to be learning any generalizable patterns on the test data set, which is why we focus our data augmentations on the training data set

▶ 📄 Click to view detailed content

```
another model, which is tiny VGG, with data augmentation this time. So if we go
back to the slide,
data augmentation is one way of dealing with overfitting. Now, it's probably not
the most
ideal experiment that we could take because our model zero, our baseline model,
looks like it's
underfitting. But data augmentation, as we've seen before, is a way of manipulating
images
to artificially increase the diversity of your training data set without collecting
more data.
So we could take our photos of pizza, sushi, and steak and randomly rotate them 30
degrees
and increase diversity forces a model to learn or hopefully learn. Again, all of
these come with
a caveat of not always being the silver bullet to learn more generalizable
patterns. Now,
I should have spelled generalizable here rather than generalization, but similar
thing.
Let's go here. Let's create to start off with, we'll just write down.
Now let's try another modeling experiment. So this is in line with our PyTorch
workflow,
trying a model and trying another one and trying another one, so and so over again.
This time,
using the same model as before, but with some slight data augmentation.
Oh, maybe we're not slight. That's probably not the best word. We'll just say with
some data
augmentation. And if we come down here, we're going to write section 9.1. We need
to first
create a transform with data augmentation. So we've seen what this looks like
before.
We're going to use the trivial augment data augmentation, create training
transform,
which is, as we saw in a previous video, what PyTorch the PyTorch team have
recently used
to train their state-of-the-art computer vision models. So train transform trivial.
This is what I'm going to call my transform. And I'm just going to from Torch
Vision import
transforms. We've done this before. We don't have to re-import it, but I'm going to
do it anyway,
just to show you that we're re-importing or we're using transforms. And we're going
to compose
a transform here. Recall that transforms help us manipulate our data. So we're
going to transform
our images into size 64 by 64. Then we're going to set up a trivial augment
transforms,
just like we did before, trivial augment wide. And we're going to set the number of
magnitude
bins here to be 31, which is the default here, which means we'll randomly use some
data augmentation
on each one of our images. And it will be applied at a magnitude of 0 to 31, also
```

```
randomly selected.
So if we lower this to five, the upper bound of intensity of how much that data
augmentation is
applied to a certain image will be less than if we set it to say 31. Now, our final
transform
here is going to be too tensor because we want our images in tensor format for our
model.
And then I'm going to create a test transform. I'm going to call this simple, which
is just going
to be transforms dot compose. And all that it's going to have, oh, I should put a
list here,
all that it's going to have, we'll just make some space over here, is going to be
transforms.
All we want to do is resize the image size equals 64 64. Now we don't apply data
augmentation
to the test data set, because we only just want to evaluate our models on the test
data set.
Our models aren't going to be learning any generalizable patterns on the test data
set,
which is why we focus our data augmentations on the training data set. And I've
just readjusted
that. I don't want to do that. Beautiful. So we've got a transform ready. Now let's
load some data
using those transforms. So we'll create train and test data sets and data loaders
with data augmentation. So we've done this before. You might want to try it out on
your own. So
pause the video if you'd like to test it out. Create a data set and a data loader
using these
transforms here. And recall that our data set is going to be creating a data set
from pizza,
steak and sushi for the train and test folders. And that our data loader is going
to be batchifying
our data set. So let's turn our image folders into data sets. Data sets, beautiful.
And I'm going
to write here train data augmented just so we know that it's it's been augmented.
We've got a few
of similar variable names throughout this notebook. So I just want to be as clear
as possible. And
I'm going to use, I'll just re import torch vision data sets. We've seen this
before, the image
folder. So rather than our use our own custom class, we're going to use the
existing image folder
class that's within torch vision data sets. And we have to pass in here a root. So
I'll just get
the doc string there, root, which is going to be equal to our trainer, which recall
is the path
to our training directory. Got that saved. And then I'm going to pass in here, the
transform is going
to be train transform trivial. So our training data is going to be augmented.
Thanks to this
transform here, and transforms trivial augment wide. You know where you can find
more about
trivial augment wide, of course, in the pie torch documentation, or just searching
transforms
trivial augment wide. And did I spell this wrong? trivial. Oh, train train
transform. I spelled
that wrong. Of course I did. So test data, let's create this as test data simple,
equals data sets
dot image folder. And the root D is going to be here the test directory. And the
transform is
just going to be what the test transform simple. Beautiful. So now let's turn these
data sets
into data loaders. So turn our data sets into data loaders. We're going to import
os,
```

Section 244: Main Topics

Key Topics:

- We could of course train this model for longer if we really wanted to by increasing the number of epochs
- Loss function often as well in PyTorch is called criterion
- We're going to set the learning rate to zero zero one, which is the default for the atom optimizer in PyTorch

▶ 📄 Click to view detailed content

```
I'm going to set the batch size here to equal to 32. The number of workers that are
going to
load our data loaders, I'm going to set this to os dot CPU count. So there'll be
one worker
per CPU on our machine. I'm going to set here the torch manual seed to 42, because
we're going to
shuffle our training data. Train data loader, I'm going to call this augmented
equals data loader.
Now I just want to I don't need to re import this, but I just want to show you
again from
torch dot utils. You can never have enough practice right dot data. Let's import
data loader.
So that's where we got the data loader class from. Now let's go train data
augmented. We'll
pass in that as the data set. And I'll just put in here the parameter name for
completeness.
That's our data set. And then we want to set the batch size, which is equal to
batch size.
I'm going to set shuffle equal to true. And I'm going to set num workers equal to
num workers.
Beautiful. And now let's do that again with the test data loader that this time
test data
loader. I'm going to call this test data loader simple. We're not using any data
augmentation on the test data set, just turning our images, our test images into
tenses.
The data set here is going to be test data simple. Going to pass in the batch size
equal to batch
size. So both our data loaders will have a batch size of 32. Going to keep shuffle
```

on false.
And num workers, I'm going to set to num workers. Look at us go. We've already got a data set
and a data loader. This time, our data loader is going to be augmented for the training data set.
And it's going to be nice and simple for the test data set. So this is really similar,
this data loader to the previous one we made. The only difference in this modeling experiment
is that we're going to be adding data augmentation, namely trivial augment wide.
So with that being said, we've got a data set, we've got a data loader. In the next video,
let's construct and train model one. In fact, you might want to give that a go. So you can use
our tiny VGG class to make model one. And then you can use our train function to train a new
tiny VGG instance with our training data loader augmented and our test data loader simple.
So give that a go and we'll do it together in the next video. I'll see you there.
Now that we've got our data sets and data loaders with data augmentation ready,
let's now create another model. So 9.3, we're going to construct and train model one.
And this time, I'm just going to write what we're going to doing, going to be doing sorry.
This time, we'll be using the same model architecture, but we're changing the data here.
Except this time, we've augmented the training data. So we'd like to see how this performs
compared to a model with no data augmentation. So that was our baseline up here. And that's what
you'll generally do with your experiments. You'll start as simple as possible and introduce
complexity when required. So create model one and send it to the target device, that is,
to the target device. And because of our helpful selves previously, we can create a manual seed
here, torch.manualseed. And we can create model one, leveraging the class that we created before.
So although we built tiny VGG from scratch in this video, in this section, sorry, in subsequent
coding sessions, because we've built it from scratch once and we know that it works, we can
just recreate it by calling the class and passing in different variables here. So let's get the
number of classes that we have in our train data augmented classes. And we're going to send it
to device. And then if we inspect model one, let's have a look. Wonderful. Now let's keep going.
We can also leverage our training function that we did. You might have tried this before.
So let's now train our model. She's going to put here. Wonderful. Now we've got a model and
data loaders. Let's create what do we have to do? We have to create a loss function and an optimizer
and call upon our train function that we created earlier to train and evaluate our model. Beautiful.
So I'm going to set the random seeds, torch dot manual seeds, and torch dot CUDA,

```
because we're
going to be using CUDA. Let's set the manual seed here 42. I'm going to set the
number of epochs.
We're going to keep many of the parameters the same. Set the number of epochs, num
epochs equals
five. We could of course train this model for longer if we really wanted to by
increasing the
number of epochs. But now let's set up the loss function. So loss FN equals NN
cross entropy loss.
Don't forget this just came into mind. Loss function often as well in PyTorch is
called
criterion. So the criterion you're trying to reduce. But I just like to call it
loss function.
And then we're going to have optimizer. Let's use the same optimizer we use before
torch dot
opt in dot atom. Recall SGD and atom are two of the most popular optimizers. So
model one dot
parameters. Then the parameters we're going to optimize. We're going to set the
learning rate to
zero zero one, which is the default for the atom optimizer in PyTorch. Then we're
going to start
the timer. So from time it, let's import the default timer as timer. And we'll go
start time
equals timer. And then let's go train model one. How can we do this? Well, we're
going to get a
```

Section 245: Main Topics

Key Topics:

- So if we go back to section four of the LearnPyTorch
- Could we use transfer learning

▶ 📄 Click to view detailed content

```
results dictionary as model one results. We're going to call upon our train
function. Inside our
train function, we'll pass the model parameter as model one. For the train data
loader parameter,
we're going to pass in train data loader augmented. So our augmented training data
loader.
And for the test data loader, we can pass in here test data loader. Simple. Then we
can write our
optimizer, which will be the atom optimizer. Our loss function is going to be an n
cross entropy
loss, what we've created above. And then we can set the number of epochs is going
to be equal to
num epochs. And then if we really wanted to, we could set the device equal to
device, which will
```

be our target device. And now let's end the timer and print out how long it took.
Took n time equals timer. And we'll go print total training time for model one is going to be
n time minus start time. And oh, it would help if I could spell, we'll get that to three decimal
places. And that'll be seconds. So you're ready? We look how quickly we built a training pipeline
for model one. And look how big easily we created it. So go ask for coding all of that stuff up
before. Let's train our second model, our first model using data augmentation. You're ready? Three,
two, one, let's go. No errors. Beautiful. We're going nice and quick here.
So oh, about just over seven seconds. So what what GPU do I have currently?
Just keep this in mind that I'm using Google Colab Pro. So I get preference in terms of
allocating a faster GPU. Your model training time may be longer than what I've got, depending on the
GPU. It also may be faster, again, depending on the GPU. But we get about seven seconds, but it looks
like our model with data augmentation didn't perform as well as our model without data augmentation.
Hmm. So how long did our model before without data augmentation take the train? Oh, just over seven
seconds as well. But we got better results in terms of accuracy on the training and test data sets
for model zero. So maybe data augmentation doesn't help in our case. And we kind of hinted at that
because the loss here was already going down. We weren't really overfitting yet. So recall that data
augmentation is a way to help with overfitting generally. So maybe that wasn't the best step to
try and improve our model. But let's nonetheless keep evaluating our model. In the next video,
we're going to plot the loss curves of model one. So in fact, you might want to give that a go.
So we've got a function plot loss curves, and we've got some results in a dictionary format.
So try that out, plot the loss curves, and see what you see. Let's do it together in the next video.
I'll see you there.
In the last video, we did the really exciting thing of training our first model with data
augmentation. But we also saw that quantitatively, it looks like that it didn't give us much improvement.
So let's keep evaluating our model here. I'm going to make a section. Recall that one of my
favorite ways or one of the best ways, not just my favorite, to evaluate the performance of a
model over time is to plot the loss curves. So a loss curve helps you evaluate your model's performance
over time. And it will also give you a great visual representation or a visual way to see if
your model is underfitting or overfitting. So let's plot the loss curves of model one results and see
what happens. We're using this function we created before. And oh my goodness, is that going in the
right direction? It looks like our test loss is going up here. Now, is that where

```
we want it to go?
Remember the ideal direction for a loss curve is to go down over time because loss
is measuring
what? It's measuring how wrong our model is. And the accuracy curve looks like it's
all over the
place as well. I mean, it's going up kind of, but maybe we don't have enough time
to measure these
things. So an experiment that you could do is train both of our models model zero
and model one
for more epochs and see if these loss curves flatten out. So I'll pose you the
question,
is our model underfitting or overfitting right now or both? So if we want to have a
look at the
loss curves, our just right is for the loss that is, this is not for accuracy, this
is for loss over
time, we want it to go down. So for me, our model is underfitting because our loss
could be lower,
but it also looks like it's overfitting as well. So it's not doing a very good job
because our test
loss is far higher than our training loss. So if we go back to section four of the
LearnPyTorch.io
book, what should an ideal loss curve look like? I'd like you to start thinking of
some ways
that we could deal with overfitting of our model. So could we get more data? Could
we simplify it?
Could we use transfer learning? We're going to see that later on, but you might
want to jump
ahead and have a look. And if we're dealing with underfitting, what are some other
things that we
could try with our model? Could we add some more layers, potentially another
convolutional block?
Could we increase the number of hidden units per layer? So if we've got currently
10 hidden units
per layer, maybe you want to increase that to 64 or something like that? Could we
train it for
longer? That's probably one of the easiest things to try with our current training
functions. We
```

Section 246: Main Topics

Key Topics:

- Then, of course, there are tools to do this, such as PyTorch plus TensorBoard
- So I'll link to this, PyTorch TensorBoard
- We're going to see this in a later section of the course

▶ 📄 Click to view detailed content

could train for 20 epochs. So have a go at this, reference this, try out some experiments with,
see if you can get these loss curves more towards the ideal shape. And in the next video, we're going
to keep pushing forward. We're going to compare our model results. So we've done two experiments.
Let's now see them side by side. We've looked at our model results individually,
and we know that they could be improved. But a good way to compare all of your experiments
is to compare your model's results side by side. So that's what we're going to do in the next video.
I'll see you there. Now that we've compared our models loss curves on their own individually,
how about we compare our model results to each other? So let's have a look at comparing our model
results. And so I'm going to write a little note here that after evaluating our modeling
experiments on their own, it's important to compare them to each other. And there's a few
different ways to do this. There's a few different ways to do this. Number one is hard coding.
So like we've done, we've written functions, we've written helper functions and whatnot,
and manually plotted things. So I'm just going to write in here, this is what we're doing.
Then, of course, there are tools to do this, such as PyTorch plus TensorBoard. So I'll link to this,
PyTorch TensorBoard. We're going to see this in a later section of the course. TensorBoard is a
great resource for tracking your experiments. If you'd like to jump forward and have a look at what
that is in the PyTorch documentation, I'd encourage you to do so. Then another one of my favorite
tools is weights and biases. So these are all going to involve some code as well, but they help out
with automatically tracking different experiments. So weights and biases is one of my favorite,
and you've got platform for experiments. That's what you'll be looking at. So if you run multiple
experiments, you can set up weights and biases pretty easy to track your different model hub
parameters. So PyTorch, there we go. Import weights and biases, start a new run on weights and biases.
You can save the learning rate value and whatnot, go through your data and just log everything there.
So this is not a course about different tools. We're going to focus on just pure PyTorch,
but I thought I'd leave these here anyway, because you're going to come across them
eventually, and MLflow is another one of my favorites as well. We have ML tracking,
projects, models, registry, all that sort of stuff. If you'd like in to look into
more ways to track your experiments, there are some extensions. But for now, we're going to stick
with hard coding. We're just going to do it as simple as possible to begin with. And if we wanted
to add other tools later on, we can sure do that. So let's create a data frame for each of our model

results. We can do this because our model results recall are in the form of dictionaries. So model
zero results. But you can see what we're doing now by hard coding this, it's quite cumbersome.
Can you imagine if we had say 10 models or even just five models, we'd have to really
write a fair bit of code here for all of our dictionaries and whatnot, whereas these tools
here help you to track everything automatically. So we've got a data frame here. Model zero results
over time. These are our number of epochs. We can notice that the training loss starts to go down.
The testing loss also starts to go down. And the accuracy on the training and test data set starts
to go up. Now, those are the trends that we're looking for. So an experiment you could try would
be to train this model zero for longer to see if it improved. But we're currently just interested
in comparing results. So let's set up a plot. I want to plot model zero results and model one
results on the same plot. So we'll need a plot for training loss. We'll need a plot for training
accuracy, test loss and test accuracy. And then we want two separate lines on each of them. One
for model zero and one for model one. And this particular pattern would be similar regardless if
we had 10 different experiments, or if we had 10 different metrics we wanted to compare,
you generally want to plot them all against each other to make them visual. And that's what tools
such as weights and biases, what TensorBoard, and what ML flow can help you to do. I'm just
going to get out of that, clean up our browser. So let's set up a plot here. I'm going to use
matplotlib. I'm going to put in a figure. I'm going to make it quite large because we want four
subplots, one for each of the metrics we want to compare across our different models. Now,
let's get number of epochs. So epochs is going to be length, or we'll turn it into a range, actually,
range of Len model zero DF. So that's going to give us five. Beautiful range between zero and five.
Now, let's create a plot for the train loss. We want to compare the train loss across model zero
and the train loss across model one. So we can go PLT dot subplot. Let's create a plot with two
rows and two columns. And this is going to be index number one will be the training loss.
We'll go PLT dot plot. I'm going to put in here epochs and then model zero DF. Inside here,
I'm going to put train loss for our first metric. And then I'm going to label it with model zero.

Section 247: Main Topics

Key Topics:

- Then we wanted to upload an image and have it be classified by our pytorch model

▶ 📄 Click to view detailed content

```
So we're comparing the train loss on each of our modeling experiments. Recall that
model zero was
our baseline model. And that was tiny VGG without data augmentation. And then we
tried out model one,
which was the same model. But all we did was we added a data augmentation transform
to our training
data. So PLT will go x label. They both used the same test data set and PLT dot
legend. Let's see
what this looks like. Wonderful. So there's our training loss across two different
models.
So we notice that model zero is trending in the right way. Model one kind of
exploded on epoch
number that would be zero, one, two, or one, depending how you're counting. Let's
just say epoch number
two, because that's easier. The loss went up. But then it started to go back down.
So again,
if we continued training these models, we might notice that the overall trend of
the loss is
going down on the training data set, which is exactly what we'd like. So let's now
plot,
we'll go the test loss. So I'm going to go test loss here. And then I'm going to
change this.
I believe if I hold control, or command, maybe, nope, or option on my Mac keyboard,
yeah, so it might be a different key on Windows. But for me, I can press option and
I can get a
multi cursor here. So I'm just going to come back in here. And that way I can
backspace there
and just turn this into test loss. Wonderful. So I'm going to put this as test loss
as the title.
And I need to change the index. So this will be index one, index two, index three,
index four.
Let's see what this looks like. Do we get the test loss? Beautiful. That's what we
get.
However, we noticed that model one is probably overfitting at this stage. So maybe
the data
augmentation wasn't the best change to make to our model. Recall that even if you
make a change
to your model, such as preventing overfitting or underfitting, it won't always
guarantee that
the change takes your model's evaluation metrics in the right direction. Ideally,
loss is going
from top left to bottom right over time. So looks like model zero is winning out
here at the moment
```

on the loss front. So now let's plot the accuracy for both training and test. So I'm going to change
this to train. I'm going to put this as accuracy. And this is going to be index number three on the
plot. And do we save it as, yeah, just act? Wonderful. So I'm going to option click here on my Mac.
This is going to be train. And this is going to be accuracy here. And then I'll change this one to
accuracy. And then I'm going to change this to accuracy. And this is going to be plot number four,
two rows, two columns, index number four. And I'm going to option click here to have two cursors,
test, act. And then I'll change this to test, act. And I'm going to get rid of the legend here.
It takes a little bit to plot because we're doing four graphs in one hit. Wonderful. So that's
comparing our models. But do you see how we could potentially functionalize this to plot, however,
many model results that we have? But if we had say another five models, we did another five
experiments, which is actually not too many experiments on a problem, you might find that
sometimes you do over a dozen experiments for a single modeling problem, maybe even more.
These graphs can get pretty outlandish with all the little lines going through. So that's
again what tools like TensorBoard, weights and biases and MLflow will help with. But if we have
a look at the accuracy, it seems that both of our models are heading in the right direction.
We want to go from the bottom left up in the case of accuracy. But the test accuracy that's training,
oh, excuse me, is this not training accuracy? I messed up that. Did you catch that one?
So training accuracy, we're heading in the right direction, but it looks like model one is
yeah, still overfitting. So the results we're getting on the training data set
aren't coming over to the testing data set. And that's what we really want our models to shine
is on the test data set. So metrics on the training data set are good. But ideally,
we want our models to perform well on the test data set data it hasn't seen before.
So that's just something to keep in mind. Whenever you do a series of modeling experiments,
it's always good to not only evaluate them individually, evaluate them against each other.
So that way you can go back through your experiments, see what worked and what didn't.
If you were to ask me what I would do for both of these models, I would probably train them for
longer and maybe add some more hidden units to each of the layers and see where the results go from
there. So give that a shot. In the next video, let's see how we can use our trained models to
make a prediction on our own custom image of food. So yes, we used a custom data set of
pizza steak and sushi images. But what if we had our own, what if we finished this model training

```
and we decided, you know what, this is a good enough model. And then we deployed it
to an app like
neutrify dot app, which is a food recognition app that I'm personally working on.
Then we wanted to
upload an image and have it be classified by our pytorch model. So let's give that
a shot, see how
```

Section 248: Main Topics

Key Topics:

- Now we're going to move on to one of the most exciting parts of deep learning
- So we could replicate a similar process to this using our trained PyTorch model, or be it
- So this image is on the course github

▶ 📄 Click to view detailed content

```
we can use our trained model to predict on an image that's not in our training data
and not in our
testing data. I'll see you in the next video. Welcome back. In the last video, we
compared our
modeling experiments. Now we're going to move on to one of the most exciting parts
of deep learning.
And that is making a prediction on a custom image. So although we've trained a
model on custom data,
how do you make a prediction on a sample slash image in our case? That's not in
either
the training or testing data set. So let's say you were building a food recognition
app,
such as neutrify, take a photo of food and learn about it. You wanted to use
computer vision to
essentially turn foods into QR codes. So I'll just show you the workflow here. If
we were to upload
this image of my dad giving two thumbs up for a delicious pizza. And what does
neutrify predicted
as pizza? Beautiful. So macaronutrients that you get some nutrition information and
then the time
taken. So we could replicate a similar process to this using our trained PyTorch
model, or be it.
It's not going to be too great of results or performance because we've seen that we
could
improve our models, but based on the accuracy here and based on the loss and
whatnot. But let's just
see what it's like, the workflow. So the first thing we're going to do is get a
custom image.
Now we could upload one here, such as clicking the upload button in Google Colab,
choosing an image
```

and then importing it like that. But I'm going to do so programmatically, as you've seen before.
So let's write some code in this video to download a custom image. I'm going to do so using requests
and like all good cooking shows, I've prepared a custom image for us. So custom image path. But
again, you could use this process that we're going to go through with any of your own images
of pizza, steak or sushi. And if you wanted to train your own model on another set of custom data,
the workflow will be quite similar. So I'm going to download a photo called pizza dad,
which is my dad, two big thumbs up. And so I'm going to download it from github. So this image is
on the course github. And let's write some code to download the image. If it doesn't already exist
in our Colab instance. So if you wanted to upload a single image, you could click with this button.
Just be aware that like all of our other data, it's going to disappear if Colab disconnects.
So that's why I like to write code. So we don't have to re upload it every time.
So if not custom image path is file, let's open a request here or open a file going to open up
the custom image path with right binary permissions as F short for file. And then when downloading,
this is because our image is stored on github. When downloading an image or when downloading
from github in general, you typically want the raw link need to use the raw file link.
So let's write a request here equals request dot get. So if we go to the pytorch deep learning
repo, then if we go into, I believe it might be extras, not in extras, it's going to be in images,
that would make a lot more sense. Wouldn't it Daniel? Let's get O for pizza dad.
So if we have a look, this is pytorch deep learning images, O for pizza dad. There's a big version
of the image there. And then if we click download, just going to give us the raw link. Yeah, there we
go. So that's the image. Hey dad, how you doing? Is that pizza delicious? It looks like it.
Let's see if our model can get this right. What do you think? Will it? So of course, we want
our model to predict pizza for this image because it's got a pizza in it. So custom image path,
we're going to download that. I've just put in the raw URL above. So notice the raw
github user content. That's from the course github. Then I'm going to go f dot right. So file,
write the request content. So the content from the request, in other words, the raw file from
github here. Similar workflow for if you were getting another image from somewhere else on
the internet and else if it is already downloaded, let's just not download it. So print f custom image
path already exists skipping download. And let's see if this works or run the code. So downloading
data o four pizza dad dot jpeg. And if we go into here, we refresh. There we go. Beautiful. So our

```
data or our custom image, sorry, is now in our data folder. So if we click on this,
this is inside
Google CoLab now. Beautiful. We got a big nice big image there. And there's a nice
big pizza there.
So we're going to be writing some code over the next few videos to do the exact
same process as
what we've been doing to import our custom data set for our custom image. What do
we still have to
do? We still have to turn it into tenses. And then we have to pass it through our
model. So let's see
what that looks like over the next few videos. We are up to one of the most
exciting parts of
building dev learning models. And that is predicting on custom data in our case, a
custom image of
a photo of my dad eating pizza. So of course, we're training a computer vision
model on here on
pizza steak and sushi. So hopefully the ideal result for our model to predict on
this image
```

Section 249: Main Topics

Key Topics:

- Let's figure out how we can get our image, our custom image, our singular image into Tensor form, loading in a custom image with pytorch, creating another section here
- It will just really help you familiarize yourself with all the functions of PyTorch domain libraries
- We can read an image into PyTorch using and go with that

▶ 📄 Click to view detailed content

```
will be pizza. So let's keep going. Let's figure out how we can get our image, our
custom image,
our singular image into Tensor form, loading in a custom image with pytorch,
creating another
section here. So I'm just going to write down here, we have to make sure our custom
image is in the
same format as the data our model was trained on. So namely, that was in Tensor
form with data type
torch float 32. And then of shape 64 by 64 by three. So we might need to change the
shape of our
image. And then we need to make sure that it's on the right device. Command MM,
beautiful. So let's
see what this looks like. Hey, so if I'm going to import torch vision. Now the
package you use to
load your data will depend on the domain you're in. So let's open up the torch
```

vision documentation.
We can go to models. That's okay. So if we're working with text, you might want to look in
here for some input and output functions, so some loading functions, torch audio, same thing.
Torch vision is what we're working with. Let's click into torch vision. Now we want to look into
reading and writing images and videos because we want to read in an image, right? We've got a
custom image. We want to read it in. So this is part of your extracurricular, by the way, to go
through these for at least 10 minutes each. So spend an hour if you're going through torch vision.
You could do the same across these other ones. It will just really help you familiarize yourself
with all the functions of PyTorch domain libraries. So we want to look here's some options for video.
We're not working with video. Here's some options for images. Now what do we want to do? We want
to read in an image. So we've got a few things here. Decode image. Oh, I've skipped over one.
We can write a JPEG if we wanted to. We can encode a PNG. Let's jump into this one. Read image.
What does it do? Read the JPEG or PNG into a three-dimensional RGB or grayscale tensor.
That is what we want. And then optionally converts the image to the desired format.
The values of the output tensor are you int eight. Okay. Beautiful. So let's see what this looks like.
Okay. Mode. The read mode used optionally for converting the image. Let's see what we can do
with this. I'm going to copy this in. So I'll write this down. We can read an image into PyTorch using
and go with that. So let's see what this looks like in practice. Read in custom image. I can't
explain to you how much I love using deep learning models to predict on custom data. So custom image.
We're going to call it you int eight because as we read from the documentation here,
it reads it in you int eight format. So let's have a look at what that looks like rather than
just talking about it. Torch vision.io. Read image. What's our target image path?
Well, we've got custom image path up here. This is why I like to do things programmatically.
So if our collab notebook reset, we could just run this cell again,
get our custom image and then we've got it here. So custom image you int eight. Let's see what this
looks like. Oh, what did we get wrong? Unable to cast Python instance. Oh, does it need to be a
string expected a value type of string or what found POSIX path? So this the path needs to be a
string. Okay. If we have a look at our custom image path, what did we get wrong? Oh, we've got a
POSIX path. So let's convert this custom image path into a string and see what happens. Look at that.
That's how image in integer form. I wonder if this is plotable. Let's go PLT dot M show custom image
you int eight. Maybe we get a dimensionality problem here in valid shape. Okay.

```
Let's
some permute it, permute, and we'll go one, two, zero. Is this going to plot? It's
a fairly big image.
There we go. Two thumbs up. Look at us. So that is the power of torch vision.io. I
owe stands for
input output. We were just able to read in our custom image. Now, how about we get
some metadata
about this? Let's go. We'll print it up here, actually. I'll keep that there
because that's
fun to plot it. Let's find the shape of our data, the data type. And yeah, we've
got it in Tensor
format, but it's you int eight right now. So we might have to convert that to float
32. We want
to find out its shape. And we need to make sure that if we're predicting on a
custom image,
the data that we're predicting on the custom image needs to be on the same device
as our model.
So let's print out some info. Print. Let's go custom image Tensor. And this is
going to be a new line.
And then we will go custom image you int eight. Wonderful. And then let's go custom
image
shape. We will get the shape parameter custom image shape or attribute. Sorry. And
then we also
want to know the data type custom image data type. But we have a kind of an inkling
because the
documentation said it would be you int eight, you int eight, and we'll go D type.
Let's have a look.
What do we have? So there's our image Tensor. And it's quite a big image. So custom
image shape.
So what was our model trained on? Our model was trained on images of 64 by 64. So
this image
```

Section 250: Main Topics

Key Topics:

- Now, I want to just highlight something about the importance of different data types and shapes and whatnot and devices, three of the biggest errors in deep learning

▶ 📄 Click to view detailed content

```
encodes a lot more information than what our model was trained on. So we're going
to have to
change that shape to pass it through our model. And then we've got an image data
type here or
Tensor data type of torch you int eight. So maybe that's going to be some errors
for us later on.
So if you want to go ahead and see if you can resize this Tensor to 64 64 using a
```

torch transform
or torch vision transform, I'd encourage you to try that out. And if you know how to change a
torch tensor from you int eight to torch float 32, give that a shot as well. So let's try
make a prediction on our image in the next video. I'll see you there.
In the last video, we loaded in our own custom image and got two big thumbs up from my dad,
and we turned it into a tensor. So we've got a custom image tensor here. It's quite big though,
and we looked at a few things of what we have to do before we pass it through our model.
So we need to make sure it's in the data type torch float 32, shape 64, 64, 3, and on the right
device. So let's make another section here. We'll go 11.2 and we'll call it making a prediction on a
custom image with a pie torch model with a trained pie torch model. And albeit, our models aren't
quite the level we would like them at yet. I think it's important just to see what it's like to
make a prediction end to end on some custom data, because that's the fun part, right? So try to make
a prediction on an image. Now, I want to just highlight something about the importance of different
data types and shapes and whatnot and devices, three of the biggest errors in deep learning.
In let's see what happens if we try to predict on you int eight format. So we'll go model one
dot eval and with torch dot inference mode. Let's make a prediction. We'll pass it through our model
one. We could use model zero if we wanted to here. They're both performing pretty poorly anyway.
Let's send it to the device and see what happens. Oh, no. What did we get wrong here?
Runtime error input type. Ah, so we've got you int eight. So this is one of our first errors
that we talked about. We need to make sure that our custom data is of the same data type that
our model was originally trained on. So we've got torch CUDA float tensor. So we've got an issue
here. We've got a you into eight image data or image tensor trying to be predicted on by a model
with its data type of torch CUDA float tensor. So let's try fix this by loading the custom image
and convert to torch dot float 32. So one of the ways we can do this is we'll just recreate the
custom image tensor. And I'm going to use torch vision dot IO dot read image. We don't have to
fully reload our image, but I'm going to do it anyway for completeness and a little bit of practice.
And then I'm going to set the type here with the type method to torch float 32. And then
let's just see what happens. We'll go custom image. Let's see what this looks like. I wonder if our
model will work on this. Let's just try again, we'll bring this up, copy this down to make a
prediction and custom image dot two device. Our image is in torch float 32 now.

```
Let's see what
happens. Oh, we get an issue. Oh my goodness, that's a big matrix. Now I have a
feeling that
that might be because our image, our custom image is of a shape that's far too
large. Custom image
dot shape. What do we get? Oh my gosh, 4000 and 3,024. And do you notice as well
that our values
here are between zero and one, whereas our previous images, do we have an image?
There we go. That
our model was trained on what between zero and one. So how could we get these
values to be between
zero and one? Well, one of the ways to do so is by dividing by 255. Now, why would
we divide by 255?
Well, because that's a standard image format is to store the image tensor values in
values from
zero to 255 for red, green and blue color channels. So if we want to scale them, so
this is what I
meant by zero to 255, if we wanted to scale these values to be between zero and
one, we can divide
them by 255. Because that is the maximum value that they can be. So let's see what
happens if we do
that. Okay, we get our image values between zero and one. Can we plot this image?
So plt dot m
show, let's plot our custom image. We got a permute it. So it works nicely with
mapplotlib.
What do we get here?
Beautiful. We get the same image, right? But it's still quite big. Look at that.
We've got a pixel
height of or image height of almost 4000 pixels and a width of over 3000 pixels. So
we need to do
some adjustments further on. So let's keep going. We've got custom image to device.
We've got an
error here. So this is a shape error. So what can we do to transform our image
shape? And you
might have already tried this. Well, let's create a transform pipeline to transform
our image shape.
So create transform pipeline or composition to resize the image. Because remember,
what are we
trying to do? We're trying to get our model to predict on the same type of data it
was trained on.
```

Section 251: Main Topics

Key Topics:

- Of course, that's what we forgot here

▶ 📄 Click to view detailed content

```
So let's go custom image transform is transforms dot compose. And we're just going
to, since our
image is already of a tensor, let's do transforms dot resize, and we'll set the
size to the same shape
that our model was trained on, or the same size that is. So let's go from torch
vision. We don't
have to rewrite this. It's already imported. But I just want to highlight that
we're using the
transforms package. We'll run that. There we go. We've got a transform pipeline.
Now let's see what
happens when we transform our target image, transform target image. What happens?
Custom image
transformed. I love printing the inputs and outputs of our different pipelines
here. So let's pass
our custom image that we've just imported. So custom image transform, our custom
image is recall
of shape. Quite large. We're going to pass it through our transformation pipeline.
And let's
print out the shapes. Let's go original shape. And then we'll go custom image dot
shape. And then
we'll go print transformed shape is going to be custom image underscore transformed
dot shape.
Let's see the transformation. Oh, would you look at that? How good we've gone from
quite a large image
to a transformed image here. So it's going to be squished and squashed a little. So
that's what
happens. Let's see what happens when we plot our transformed image. We've gone from
4000 pixels
on the height to 64. And we've gone from 3000 pixels on the height to 64. So this
is what our
model is going to see. Let's go custom image transformed. And we're going to
permute it to be 120.
Okay, so quite pixelated. Do you see how this might affect the accuracy of our
model?
Because we've gone from custom image, is this going to, oh, yeah, we need to plot
dot image.
So we've gone from this high definition image to an image that's of far lower
quality here.
And I can kind of see myself that this is still a pizza, but I know that it's a
pizza. So just
keep this in mind going forward is that another way that we could potentially
improve our model's
performance if we increased the size of the training image data. So instead of 64
64, we might want
to upgrade our models capability to deal with images that are of 224 224. So if we
have a look
at what this looks like 224 224. Wow, that looks a lot better than 64 64. So that's
something that
you might want to try out later on. But we're going to stick in line with the CNN
explainer model.
How about we try to make another prediction? So since we transformed our
image to be the same size as the data our model was trained on. So with torch
inference mode,
let's go custom image pred equals model one on custom image underscore transformed.
Does it work now? Oh my goodness, still not working expected all tensors on the
same device. Of course,
that's what we forgot here. Let's go to device. Or actually, let's leave that error
there. And
we'll just copy this code down here. And let's put this custom image transform back
on the right
device and see if we finally get a prediction to happen with our model. Oh, we
still get an error.
Oh my goodness, what's going on here? Oh, we need to add a batch size to it. So I'm
just gonna write
up here. This will error. No batch size. And this will error. Image not on right
device. And then
let's try again, we need to add a batch size to our image. So if we look at custom
image transformed
dot shape, recall that our images that passed through our model had a batch
dimension. So this
is another place where we get shape mismatch issues is if our model, because what's
going on
in neural network is a lot of tensor manipulation. If the dimensions don't line up,
we want to perform
matrix multiplication and the rules. If we don't play to the rules, the matrix
multiplication will
fail. So let's fix this by adding a batch dimension. So we can do this by going a
custom image transformed.
Let's unsqueeze it on the first dimension and then check the shape. There we go. We
add a single batch.
So that's what we want to do when we make a prediction on a single custom image. We
want to pass it to
our model as an image or a batch of one sample. So let's finally see if this will
work.
Let's just not comment what we'll do. This, or maybe we'll try anyway, this should
work.
Added a batch size. So do you see the steps we've been through so far? And we're
just going to
unsqueeze this. Unsqueeze on the zero dimension to add a batch size. Oh, it didn't
error. Oh my
goodness. It didn't error. Have a look at that. Yes, that's what we want. We get a
prediction
load it because the raw outputs of our model, we get a load it value for each of
our custom classes.
So this could be pizza. This could be steak. And this could be sushi, depending on
the order of
our classes. Let's just have a look. Class to IDX. Did we not get that? Class
names.
Beautiful. So pizza steak sushi. We've still got a ways to go to convert this into
that.
But I just want to highlight what we've done. So note, to make a prediction on a
custom image,
we had to. And this is something you'll have to keep in mind for almost all of your
custom data.
It needs to be formatted in the same way that your model was trained on. So we had
to load the image
```

Section 252: Main Topics

Key Topics:

- So make sure you take care of the three big pie torch and deep learning errors
- So the first dimension of this tensor will be the inner brackets, of course

▶ 📄 Click to view detailed content

```
and turn it into a tensor. We had to make sure the image was the same data type as
the model.
So that was torch float 32. And then we had to make sure the image was the same
shape as the data
the model was trained on, which was 64, 64, three with a batch size. So that was
one,
three, 64, 64. And excuse me, this should actually be the other way around. This
should be color
channels first, because we're dealing with pie torch here. 64. And then finally, we
had to make
sure the image was on the same device as our model. So they are three of the big
ones that we've
talked about so much the same data type or data type mismatch will result in a
bunch of issues.
Shape mismatch will result in a bunch of issues. And device mismatch will also
result in a bunch
of issues. If you want these to be highlighted, they are in the learn pie torch.io
resource. We have
putting things together. Where do we have it? Oh, yeah, no, it's in the main
takeaway section,
sorry, predicting on your own custom data with a trained model as possible, as long
as you format
the data into a similar format to what the model was trained on. So make sure you
take care of the
three big pie torch and deep learning errors. Wrong data types, wrong data shapes,
and wrong
devices, regardless of whether that's images or audio or text, these three will
follow you around.
So just keep them in mind. But now we've got some code to predict on custom images,
but it's kind
of all over the place. We've got about 10 coding cells here just to make a
prediction on a custom
image. How about we functionize this and see if it works on our pizza dad image.
I'll see you in the
next video. Welcome back. We're now well on our way to making custom predictions on
our own custom
image data. Let's keep pushing forward. In the last video, we finished off getting
some raw model
logits. So the raw outputs from our model. Now, let's see how we can convert these
logits into
prediction labels. Let's write some code. So convert logits to prediction labels.
Or let's go
convert logits. Let's first convert them to prediction probabilities.
Probabilities.
So how do we do that? Let's go custom image pred probes equals torch dot softmax
to convert our custom image pred across the first dimension. So the first dimension
of this tensor
```

will be the inner brackets, of course. So just this little section here. Let's see what these
look like. This will be prediction probabilities. Wonderful. So you'll notice that these are quite
spread out. Now, this is not ideal. Ideally, we'd like our model to assign a fairly large
prediction probability to the target class, the right target class that is. However, since our model
when we trained it isn't actually performing that all that well. The prediction probabilities
are quite spread out across all of the classes. But nonetheless, we're just highlighting what
it's like to predict on custom data. So now let's convert the prediction probabilities
to prediction labels. Now, you'll notice that we used softmax because why we are working with
multi class classification data. And so we can get the custom image pred labels, the integers,
by taking the argmax of the prediction probabilities, custom image pred probes across the first
dimension as well. So let's go custom image pred labels. Let's see what they look like.
Zero. So the index here with the highest value is index number zero. And you'll notice that it's
still on the coded device. So what would happen if we try to index on our class names with
the custom image pred labels? Or maybe that doesn't need to be a plural. Oh, there we go. We get pizza.
But you might also have to change this to the CPU later on. Otherwise, you might run into some
errors. So just be aware of that. So you notice how we just put it to the CPU. So we get pizza. We
got a correct prediction. But this is as good as guessing in my opinion, because these are kind
of spread out. Ideally, this value would be higher, maybe something like 0.8 or above for our pizza
dad image. But nonetheless, our model is getting two thumbs up even on this 64 by 64 image. But
that's a lot of code that we've written. Let's functionize it. So we can just pass in a file path
and get a custom prediction from it. So putting custom image prediction together.
Let's go building a function. So we want the ideal outcome is, let's plot our image as well.
Ideal outcome is a function where we plot or where we pass an image path to and have our model predict
on that image and plot the image plus the prediction. So this is our ideal outcome. And I think I'm
going to issue this as a challenge. So give that a go, put all of our code above together. And you'll
just have to import the image, you'll have to process it and whatnot. I know I said we were going
to build a function in this video, but we're going to say that to the next video. I'd like
you to give that a go. So start from way back up here, import the image via torture vision.io read
image, format it using what we've done, change the data type, change the shape, change the device,

```
and then plot the image with its prediction as the title. So give that a go and
we'll do it
together in the next video. How'd you go? I just realized I had a typo in the
previous cell,
```

Section 253: Main Topics

Key Topics:

▶ 📄 Click to view detailed content

```
but that's all right. Did you give it a shot? Did you put together the custom image
prediction
in a function format? I'd love it if you did. But if not, that's okay. Let's keep
going. Let's see
what that might look like. And there are many different ways that you could do
this. But
here's one of the ways that I've thought of. So we want to function that's going to
pred and plot a target image. We wanted to take in a torch model. And so that's
going to be ideally
a trained model. We wanted to also take in an image path, which will be of a
string. It can
take in a class names list so that we can index it and get the prediction label in
string format.
So let's put this as a list of strings. And by default, this can equal none. Just
in case we
just wanted the prediction, it wants to take in a transform so that we can pass it
in some form of
transform to transform the image. And then it's going to take in a device, which
will be by default
the target device. So let's write a little doc string here, makes a prediction on a
target image
with a trained model and plots the image and prediction. Beautiful. Now what do we
have to do
first? Let's load in the image. Load in the image just like we did before with
torch vision. So
target image equals torch vision.io dot read image. And we'll go string on the
image path,
which will be the image path here. And we convert it to a string just in case it
doesn't get passed
in as a string. And then let's change it into type torch float 32. Because we want
to make sure that
our custom image or our custom data is in the same type as what we trained our
model on. So now
let's divide the image pixel values by 255 to get them between zero or to get them
between zero
one as a range. So we can just do this by target image equals target image divided
by 255. And we
could also just do this in one step up here 255. But I've just put it out there
just to let you know
```

that, hey, read image imports image data as between zero and 255. So our model prefers numbers
between zero and one. So let's just scale it there. Now we want to transform our data if necessary.
In our case, it is, but it won't always be. So we want this function to be pretty generic
predomplot image. So if the transform exists, let's set the target image to the transform,
or we'll pass it through the transform that is wonderful. And the transform we're going to get
from here. Now what's left to do? Well, let's make sure the model is on the target device.
It might be by default, but if we're passing in a device parameter, we may as well make sure the
model is there too. And now we can make a prediction. So let's turn on a vowel slash inference mode
and make a prediction with our model. So model, we call a vowel mode, and then with torch dot
inference mode, because we're making a prediction, we want to turn our model into inference mode,
or put it in inference mode context. Let's add an extra dimension to the image. Let's go target
image. We could do this step above, actually, but we're just going to do it here. From kind of
remembering things on the fly here of what we need to do, we're adding a, this is, let's write
this down, this is the batch dimension. e g our model will predict on batches of one x image.
So we're just unsqueezing it to add an extra dimension at the zero dimension space,
just like we did in a previous video. Now let's make a prediction
on the image with an extra dimension. Otherwise, if we don't have that extra dimension, we saw
that we get a shape issue. So right down here, target image pred. And remember, this is going
to be the raw model outputs, raw logit outputs. We're going to target image pred. And yeah,
I believe that's all we need for the prediction. Oh wait, there was one more thing, two device.
Me too. Also make sure the target image is on the right device. Beautiful. So fair
few steps here, but nothing we can't handle. All we're really doing is replicating what we've done
for batches of images. But we want to make sure that if someone passed any image to our
pred and plot image function, that we've got functionality in here to handle that image.
And do we get this? Oh, we want just target image to device. Did you catch that error?
So let's keep going. Now let's convert the logits. Our models raw logits. Let's convert those
to prediction probabilities. This is so exciting. We're getting so close to making a function
to predict on custom data. So we'll set this to target image pred probes, which is going to be
torch dot softmax. And we will pass in the target image pred here. We want to get the softmax of
the first dimension. Now let's convert our prediction probabilities, which is what we get in the line

```
above. We want to convert those to prediction labels. So let's get the target image
pred labels
labels equals torch dot argmax. We want to get the argmax of, or in other words,
the index,
which is the maximum value from the pred probes of the first dimension as well. Now
what should we
return here? Well, we don't really need to return anything. We want to create a
plot. So let's plot
```

Section 254: Main Topics

Key Topics:

- One of the most fun things to do when building deep learning models

▶ 📄 Click to view detailed content

```
the image alongside the prediction and prediction probability. Beautiful. So plot
dot in show,
what are we going to pass in here? We're going to pass in here our target image.
Now we have to
squeeze this, I believe, because we've added an extra dimension up here. So we'll
squeeze it to
remove that batch size. And then we still have to permute it because map plot lib
likes images
in the format color channels last one, two, zero. So remove batch dimension.
And rearrange shape to be hc hwc. That is color channels last. Now if the class
names parameter
exists, so we've passed in a list of class names, this function is really just
replicating
everything we've done in the past 10 cells, by the way. So right back up here,
we're replicating
all of this stuff in one function. So pretty large function, but once we've written
it,
we can pass in our images as much as we like. So if class names exist, let's set
the title
to our showcase that class name. So the pred is going to be class names. Let's
index on that
pred image, or target image pred label. And this is where we'll have to put it to
the CPU,
because if we're using a title with map plot lib, map plot lib cannot handle things
that are on
the GPU. This is why we have to put it to the CPU. And then I believe that should
be enough for
that. Let's add a little line in here, so that we can have it. Oh, I've missed
something.
An outside bracket there. Wonderful. Let's add the prediction probability, because
that's always
fun to see. So we want target image pred probs. And we want to get the maximum pred
problem from
```

that. And we'll also put that on the CPU. And I think we might get this three decimal places.
Now this is saying, oh, pred labels, we don't need that. We need just non plural, beautiful. Now,
if the class names doesn't exist, let's just set the title equal to f f string, we'll go pred,
target image pred label. Is Google Colab still telling me this is wrong?
Target image pred label. Oh, no, we've still got the same thing. It just hasn't caught up with me,
and I'm coding a bit fast here. And then we'll pass in the prob, which will be just the same as
above. I could even copy this in. Beautiful. And let's now set the title to the title. And we
and we will turn the axes off. PLT axes false. Fair bit of code there. But this is going to be a
super exciting moment. Let's see what this looks like. When we pass it in a target image and a
target model, some class names, and a transform. Are you ready? We've got our transform ready,
by the way, it's back up here. Custom image transform. It's just going to resize our image.
So let's see. Oh, this file was updated remotely or in another tab. Sometimes this happens, and
usually Google Colab sorts itself out, but that's all right. It doesn't affect our code for now.
Pred on our custom image. Are you ready? Save failed. Would you like to override? Yes, I would.
So you might see that in Google Colab. Usually it fixes itself. There we go. Save successfully.
Pred and plot image. I was going to say, Google Colab, don't fail me now. We're about to predict
on our own custom data. Using a model trained on our own custom data. Image part. Let's pass in
custom image path, which is going to be the path to our pizza dad image. Let's go class names,
equals class names, which is pizza, steak, and sushi. We'll pass in our transform to convert our
image to the right shape and size custom image transform. And then finally, the target device is
going to be device. Are you ready? Let's make a prediction on custom data. One of my favorite
things. One of the most fun things to do when building deep learning models. Three, two, one.
How did it go? Oh, no. What did we get wrong? CPU. Okay. Such a so close, but yet so far.
Has no attribute CPU. Oh, maybe we need to put this to CPU. That's where I got the square bracket
wrong. So that's what we needed to change. We needed to because this is going to be potentially
on the GPU. Tag image pred label. We need to put it on the CPU. We need to do that. Why?
Because this is going to be the title of our map plot lib plot. And map plot lib doesn't interface
too well with data on a GPU. Let's try it again. Three, two, one, running. Oh, look at that.
Prediction on a custom image. And it gets it right. Two thumbs up. I didn't plan this. Our model is

```
performing actually quite poorly. So this is as good as a guess to me. You might
want to try this
on your own image. And in fact, if you do, please share it with me. I would love to
see it. But
you could potentially try this with another model. See what happens? Steak. Okay,
there we go. So
even though model one performs worse quantitatively, it performs better
qualitatively. So that's the
power of a visualize, visualize, visualize. And if we use model zero, also, which
isn't performing
too well, it gets it wrong with a prediction probability of 0.368, which isn't too
high either.
So we've talked about a couple of different ways to improve our models. Now we've
even
got a way to make predictions on our own custom images. So give that a shot. I'd
love to see
your custom predictions, upload an image here if you want, or download it into
Google Colab using
```

Section 255: Main Topics

Key Topics:

- But of course, there are a fair few ways that we could improve our models performance
- And then a lot of machine learning is dealing with the balance between overfitting and underfitting
- So much of the research and machine learning is actually dedicated towards this balance

▶ 📄 Click to view detailed content

```
code that we've used before. But we've come a fairly long way. I feel like we've
covered enough
for custom data sets. Let's summarize what we've covered in the next video. And
I've got a bunch
of exercises and extra curriculum for you. So this is exciting stuff. I'll see you
in the next video.
In the last video, we did the very exciting thing of making a prediction on our own
custom
image, although it's quite pixelated. And although our models performance
quantitatively didn't
turn out to be too good qualitatively, it happened to work out. But of course,
there are a fair few
ways that we could improve our models performance. But the main takeaway here is
that we had to do
a bunch of pre processing to make sure our custom image was in the same format as
```

what our model
expected. And this is quite a lot of what I do behind the scenes for Nutrify. If you upload an
image here, it gets pre processed in a similar way to go through our image classification model
to output a label like this. So let's get out of this. To summarize, I've got a colorful slide here,
but we've already covered this predicting on custom data. These are three things to make sure of,
regardless of whether you're using images, text or audio, make sure your data is in the right
data type. In our case, it was torch float 32. Make sure your data is on the same device as the model.
So we had to put our custom image to the GPU, which was where our model also lived. And then we had
to make sure our data was in the correct shape. So the original shape was 64, 64, 3. Actually,
this should be reversed, because it was color channels first. But the same principle remains here.
We had to add a batch dimension and rearrange if we needed. So in our case, we used images of this
shape batches first color channels first height width. But depending on your problem will depend
on your shape, depending on the device you're using will depend on where your data and your
model lives. And depending on the data type you're using will depend on what you're using for torch
float 32 or something else. So let's summarize. If we go here main takeaways, you can read through
these, but some of the big ones are pie torch has many built in functions to deal with all kinds
of data from vision to text to audio to recommendation systems. So if we look at the pie torch docs,
you're going to become very familiar with these over time. We've got torch audio data,
torch text, torch vision is what we practiced with. And we've got a whole bunch of things here for
transforming and augmenting images, data sets, utilities, operators, and torch data is currently
in beta. But this is just something to be aware of later on. So it's a prototype library right now,
but by the time you watch this, it might be available. But it's another way of loading data.
So just be aware of this for later on. And if we come back to up here, if applied to watch built
in data loading functions, don't suit your requirements, you can write your own custom
data set classes by subclassing torch dot utils dot data dot data set. And we saw that way back
up here in option number two. Option two, here we go, loading image data with a custom data set,
wrote plenty of code to do that. And then a lot of machine learning is dealing with the
balance between overfitting and underfitting. We've got a whole section in the book here to
check out what an ideal loss curve should look like and how to deal with overfitting,

```
how to deal with underfitting. It's it is a fine line. So much of the research and
machine
learning is actually dedicated towards this balance. And then three big things for
being aware of
when you're predicting on your own custom data, wrong data types, wrong data
shapes,
and wrong devices. This will follow you around, as I said, and we saw that in
practice to get our
own custom image ready for a trained model. Now, we have some exercises here. If
you'd like
the link to it, you can go to loan pytorch.io section number four exercises, and of
course,
extra curriculum. A lot of the things I've mentioned throughout the course that
would be a good
resource to check out contained in here. But the exercises, this is this is your
time to shine,
your time to practice. Let's go back to this notebook, scroll right down to the
bottom.
Look how much code we've written. Goodness me, exercises for all exercises and
extra curriculum.
See here, turn that into markdown. Wonderful. And so if we go in here, you've got a
couple of
resources. There's an exercise template notebook for number four, and example
solutions for notebook
number four, which is what we're working on now. So of course, I'd encourage you to
go through the
pytorch custom data sets exercises template first. Try to fill out all of the code
here on your own.
So we've got some questions here. We've got some dummy code. We've got some
comments.
So give that a go. Go through this. Use this book resource to reference. Use all
the code
we've written. Use the documentation, whatever you want. But try to go through this
on your own.
And then if you get stuck somewhere, you can look at an example solution that I
created,
which is here, pytorch custom data sets exercise solutions. And just be aware that
this is just
one way of doing things. It's not necessarily the best. It's just a way to
reference what
```

Section 256: Main Topics

Key Topics:

- So this is me live streaming the whole thing, writing a bunch of pytorch code
- Oh, by the way, this is in the extras exercises tab of the pytorch deep learning repo
- So that has been pytorch custom data sets

▶ 📄 Click to view detailed content

```
you're writing to what I would do. And there's actually now live walkthroughs of
the solutions,
errors and all on YouTube. So if you go to this video, which is going to mute. So
this is me
live streaming the whole thing, writing a bunch of pytorch code. If you just keep
going through all
of that, you'll see me writing all of the solutions, running into errors, trying
different things,
et cetera, et cetera. But that's on YouTube. You can check that out on your own
time. But I feel
like we've covered enough exercises. Oh, by the way, this is in the extras
exercises tab
of the pytorch deep learning repo. So extras exercises and solutions that are
contained in there.
Far out. We've covered a lot. Look at all that. So that has been pytorch custom
data sets.
I will see you in the next section. Holy smokes. That was a lot of pytorch code.
But if you're still hungry for more, there is five more chapters available at
learnpytorch.io,
which cover transfer learning, my favorite topic, pytorch model experiment
tracking,
pytorch paper replicating, and pytorch model deployment. How do you get your model
into the
hands of others? And if you'd like to learn in this video style, the videos for
those chapters
are available at zero to mastery.io. But otherwise, happy machine learning. And
I'll see you next time.
```

Styled with CSS variables for easy theme customization