Tutorial 7

To complete the Activities and Tasks, please read all of the content we have provided.

1. Introduction to the Colab
2. Introduction to Neural Networks (NNs)
3. It is a lot of reading for NNs, but it is necessary to complete the tutorial.
4. Building the Neural Networks (NNs) using PyTorch
5. Read the comments in the code. They meant to explain some concept/information.

Introduction to the Colab

Colab is a Python development environment that runs in the browser using Google Cloud.

Google Colaboratory is a free online cloud-based Jupyter Notebook environment that allows us to train our machine learning and deep learning models on CPUs, GPUs, and TPUs.

It does not matter which computer you have, what its configuration is, and how ancient it might be. You can still use Google Colab! All you need is a **Google account and a web browser**. And here's the cherry on top – you get access to GPUs like Tesla K80 and even a TPU for free!

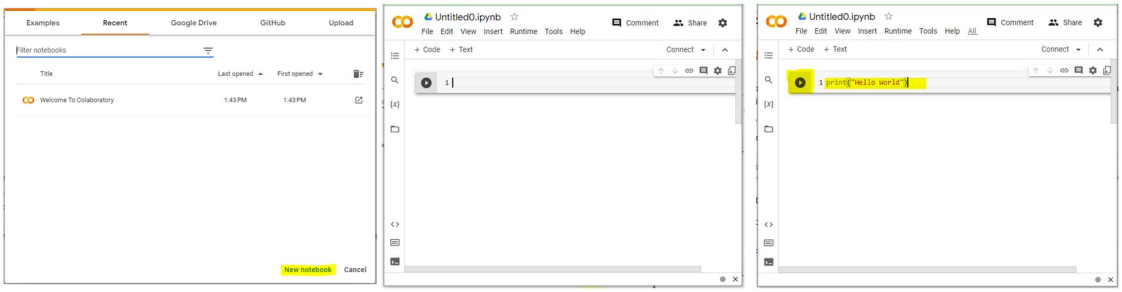
TPUs are much more expensive than GPU, and you can use them for free on Colab. It's worth repeating again and again – it's an offering like no other.

Activity-I

Create a Colab Notebook. Execute the steps below to create a Colab Notebook. The ECT for this activity is 60 minutes.

As mentioned above, you should have a **Google account and a web browser.**

- Step 1: Sign in to your Gmail account in any web browser (Google Chrome is recommended).
- Step 2: Go to Google Colab using the link: https://colab.research.google.com/. You'll get the screen below when you open Colab.
- Step 3: Click the "NEW NOTEBOOK" button to create a new Colab notebook. You can also upload your local notebook to Colab by clicking the upload button.
- Step 4: You will be taken to the new Colab Notebook. Refer to the picture below for Colab Notebook.
- Step 5: Enter the following command in the cell and click on the run. **`print("Hello World")`**
- Step 6: Include the screen capture of the Colab notebook in the Tutorial Completion Document.



You can rename your notebook by clicking **"Untitled0.ipynb"**and changing it to anything you want. Changing according to the project you are working on is a good practice.

To print "Hello World", just hover the mouse over [ ] and press the play button to the upper left. Or press shift-enter to execute.

```
print("Hello World")
```

```
Hello World
```

Screenshot

Functions, Conditionals, and Iteration

Let's create a Python function, and call it from a loop.

```
def HelloWorldXY(x, y):
  if (x < 10):
    print("Hello World, x was < 10")
  elif (x < 20):
    print("Hello World, x was >= 10 but < 20")
  else:
    print("Hello World, x was >= 20")
  return x + y

for i in range(8, 25, 5):  # i=8, 13, 18, 23 (start, stop, step)
  print("--- Now running with i: {}".format(i))
  r = HelloWorldXY(i,i)
  print("Result from HelloWorld: {}".format(r))
```

```
--- Now running with i: 8
Hello World, x was < 10
Result from HelloWorld: 16
--- Now running with i: 13
Hello World, x was >= 10 but < 20
Result from HelloWorld: 26
--- Now running with i: 18
Hello World, x was >= 10 but < 20
Result from HelloWorld: 36
--- Now running with i: 23
Hello World, x was >= 20
Result from HelloWorld: 46
```

```
print(HelloWorldXY(1,2))
```

```
Hello World, x was < 10
3
```

Why are GPUs or TPUs necessary for training Deep Learning models?

CPUs, GPUs and TPUs

CPU

- Central Processing Unit abbreviation CPU, is the electronic circuitry, which work as a brains of the computer that perform the basic arithmetic, logical, control and input/output operations specified by the instructions of a computer program.
- CPU can handle **tens of operation per cycle**
- A processor designed to solve every computational problem in a general fashion. The cache and memory design is designed to be optimal for any general programming problem

GPU

- GPU, the Graphics Processing Unit is a specialized electronic circuit designed to render 2D and 3D graphics together with a CPU. GPU also known as Graphics Card in the Gamer's culture. Now GPU are being harnessed more broadly to accelerate computational workloads in areas such as financial modeling, cutting-edge scientific research, deep learning, analytics and oil and gas exploration etc.
- GPU can handle **tens of thousands of operation per cycle**
- A processor designed to accelerate the rendering of graphics.

TPU

- Tensor Processing Unit abbreviation TPU is a custom-built integrated circuit developed specifically for machine learning and tailored for TensorFlow, Google's open-source machine learning framework. TPU's have been powering Google data centers since 2015, however Google still uses CPUs and GPUs for other types of machine learning.
- TPU can handle **upto 128000 operations per cycle**
- A co-processor designed to accelerate deep learning tasks develop using TensorFlow (a programming framework); Compilers have not been developed for TPU which could be used for general purpose programming; hence, it requires significant effort to do general programming on TPU

Deep Learning requires a lot of hardware. We will come back to this once we understand **What is Deep Learning?**

Colab Specifics

When a new Colab Notebook is created, a **"Welcome To Colaboratory"** notebook is automatically created. It has a handful of information if you want to explore Colab Notebook. However, you do not need to go through it for this tutorial.

Colab is a virtual machine you can access directly. To run commands at the VM's terminal, prefix the line with an exclamation point (!).

```
print("\nDoing $ls on filesystem")
!ls -l
!pwd
```

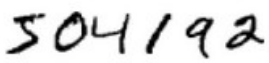
```
Doing $ls on filesystem
total 2628
drwxr-xr-x 3 root root    4096 Apr 18 02:01 data
-rw-r--r-- 1 root root 2680839 Apr 18 02:36 model.pth
drwxr-xr-x 1 root root    4096 Apr 14 13:35 sample_data
/content
```

```
print("Install numpy")  # Just for test, numpy is actually
preinstalled in all Colab instances
!pip install numpy
```

```
Install numpy
Looking in indexes: https://pypi.org/simple, https://us-
python.pkg.dev/colab-wheels/public/simple/
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-
packages (1.22.4)
```

Introduction to Neural Networks (NNs)

The human visual system is one of the wonders of the world. Consider the following sequence of handwritten digits:



Most people effortlessly recognize those digits as 504192. That ease is deceptive. In each hemisphere of our brain, humans have a primary visual cortex, also known as V1, containing 140 million neurons, with tens of billions of connections between them. And yet human vision involves not just V1, but an entire series of visual cortices - V2, V3, V4, and V5 - doing progressively more complex image processing. We carry in our heads a supercomputer, tuned by evolution over hundreds of millions of years, and superbly adapted to understand the visual world. Recognizing handwritten digits isn't easy. Rather, we humans are stupendously, astoundingly good at making sense of what our eyes show us. But nearly all that work is done unconsciously. And so we don't usually appreciate how tough a problem our visual systems solve.

The difficulty of visual pattern recognition becomes apparent if you attempt to write a computer program to recognize digits like those above. What seems easy when we do it ourselves suddenly becomes extremely difficult. Simple intuitions about how we recognize shapes - "a 9 has a loop at the top, and a vertical stroke in the bottom right" - turn out to be not so simple to express algorithmically. When you try to make such rules precise, you quickly get lost in a morass of exceptions and caveats and special cases. It seems hopeless.

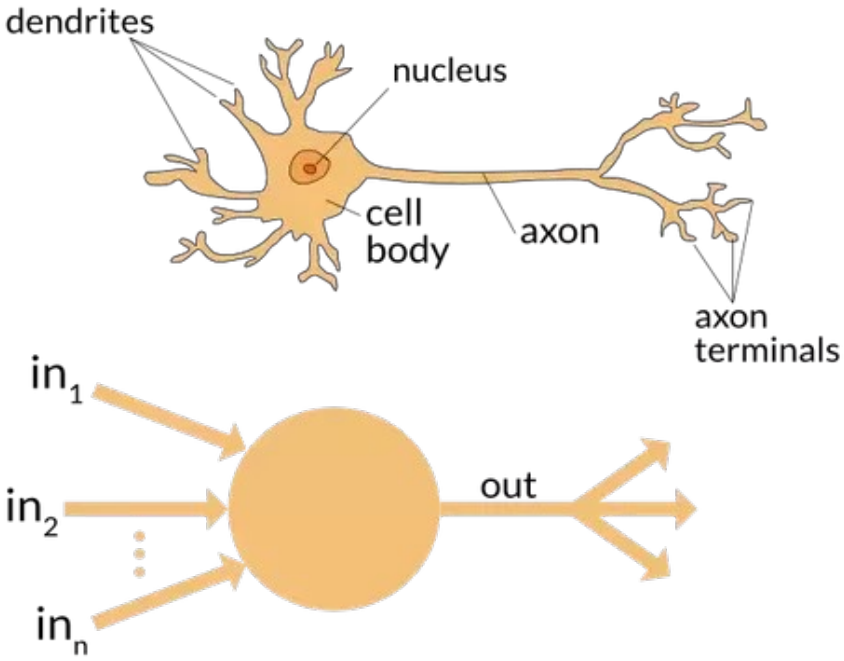
Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples,

and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten digits. Furthermore, by increasing the number of training examples, the network can learn more about handwriting, and so improve its accuracy. So while I've shown just 100 training digits above, perhaps we could build a better handwriting recognizer by using thousands or even millions or billions of training examples.

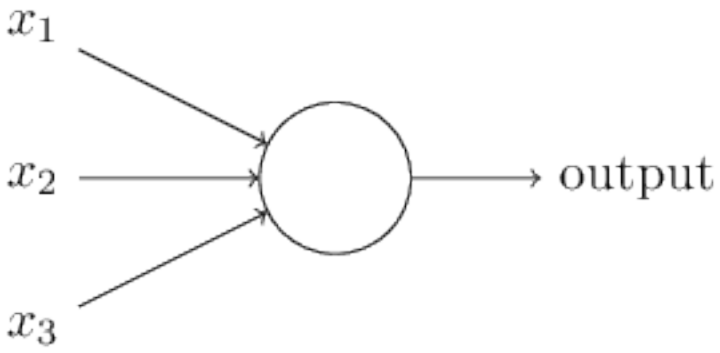
Here, we will discuss key concepts regarding neural networks, including two essential types of artificial neurons, the **perceptron** and the **sigmoid** neuron.

Perceptrons

What is a neural network? To get started, I'll explain a type of artificial neuron called a perceptron. Perceptrons were developed in the 1950s and 1960s by the scientist Frank Rosenblatt, inspired by earlier work by Warren McCulloch and Walter Pitts. McCulloch and Pitts realized that a simplified model of a real neuron could be represented using simple addition and thresholding, as shown in below image



So how do perceptrons work? A perceptron takes several binary inputs, x1,x2,..., and produces a single binary output:



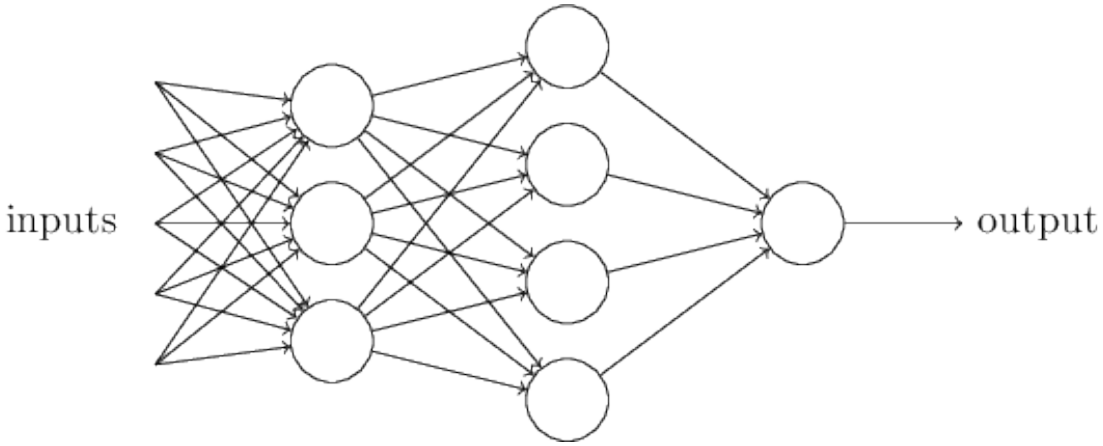
In the example shown the perceptron has three inputs, x1,x2,x3. In general it could have more or fewer inputs. Rosenblatt proposed a simple rule to compute the output. He introduced weights, w1,w2,..., real numbers expressing the importance of the respective inputs to the output. The neuron's output, 0 or 1, is determined by whether the weighted sum $\sum_j w_j x_j$ is less than or greater than some threshold value. Just like the weights, the threshold is a real number which is a parameter of the neuron. To put it in more precise algebraic terms:

\begin{eqnarray} \mbox{output} & = & \left{ \begin{array}{ll} 0 & \mbox{if } \sum_j w_j x_j \leq \mbox{ threshold} \ 1 & \mbox{if } \sum_j w_j x_j > \mbox{ threshold} \end{array} \right. \tag{1}\end{eqnarray}

That's all there is to how a perceptron works!

That's the basic mathematical model. A way you can think about the perceptron is that it's a device that makes decisions by weighing up evidence. By varying the weights and the threshold, we can get different models of decision-making.

Obviously, the perceptron isn't a complete model of human decision-making! But what the example illustrates is how a perceptron can weigh up different kinds of evidence in order to make decisions. And it should seem plausible that a complex network of perceptrons could make quite subtle decisions:



In this network, the first column of perceptrons - what we'll call the first layer of perceptrons - is making three very simple decisions, by weighing the input evidence. What about the perceptrons in the second layer? Each of those perceptrons is making a decision by weighing up the results from the first layer of decision-making. In this way a perceptron in the second layer can make a decision at a more complex and more abstract level than perceptrons in the first layer. And even more complex decisions can be made by the perceptron in the third layer. In this way, a many-layer network of perceptrons can engage in sophisticated decision making.

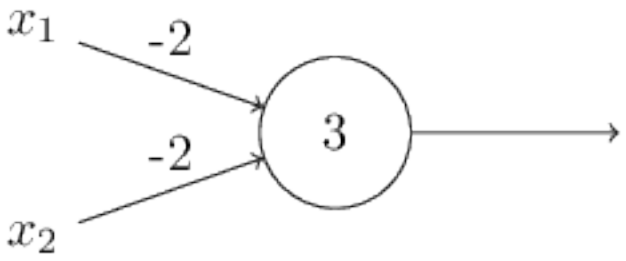
Incidentally, when I defined perceptrons I said that a perceptron has just a single output. In the network above the perceptrons look like they have multiple outputs. In fact, they're still single output. The multiple output arrows are merely a useful way of indicating that the output from a perceptron is being used as the input to several other perceptrons. It's less unwieldy than drawing a single output line which then splits.

Let's simplify the way we describe perceptrons. The condition $\begin{eqnarray} \sum_j w_j x_j > \mbox{threshold} \end{eqnarray}$ is cumbersome, and we can make two notational changes to simplify it. The first change is to write $\Sigma j w j x j$ as a dot product, $w \cdot x \equiv \Sigma j w j x j$, where w and x are vectors whose components are the weights and inputs, respectively. The second change is to move the threshold to the other side of the inequality, and to replace it by what's known as the perceptron's bias, $b \equiv -\text{threshold}$. Using the bias instead of the threshold, the perceptron rule can be rewritten:

$\begin{eqnarray} \mbox{output} = \left\{ \begin{array}{ll} 0 & \mbox{if } w\cdot x + b \leq 0 \\ 1 & \mbox{if } w\cdot x + b > 0 \end{array} \right. \tag{2}\end{eqnarray}$

You can think of the bias as a measure of how easy it is to get the perceptron to output a 1. Or to put it in more biological terms, the bias is a measure of how easy it is to get the perceptron to fire. For a perceptron with a really big bias, it's extremely easy for the perceptron to output a 1. But if the bias is very negative, then it's difficult for the perceptron to output a 1.

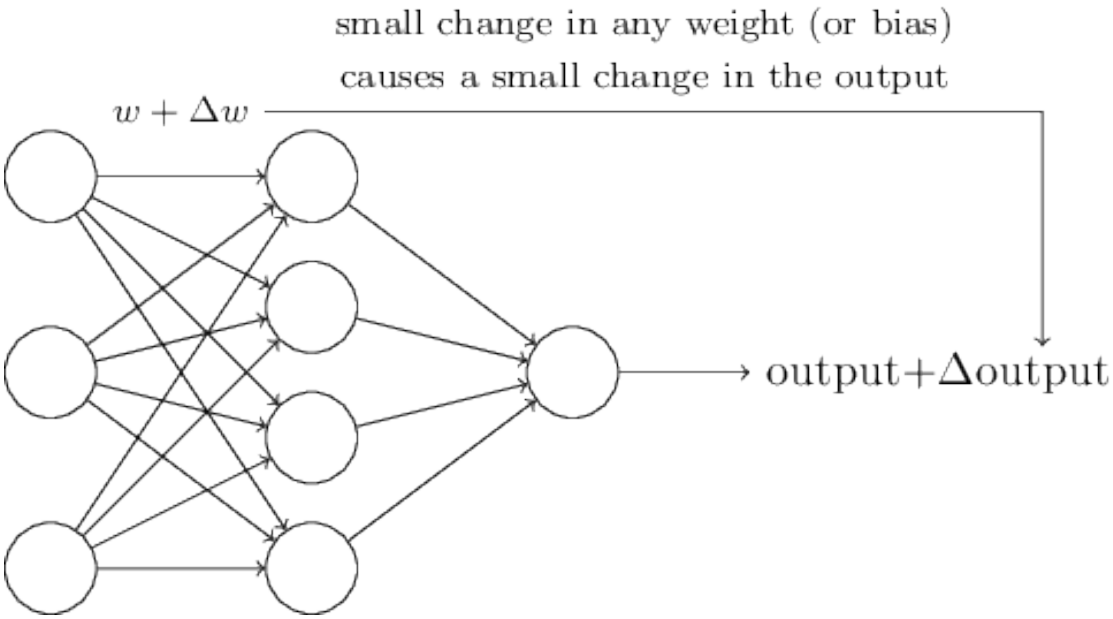
I've described perceptrons as a method for weighing evidence to make decisions. Another way perceptrons can be used is to compute the elementary logical functions we usually think of as underlying computation, functions such as AND, OR, and NAND. For example, suppose we have a perceptron with two inputs, each with weight −2, and an overall bias of 3. Here's our perceptron:



Then we see that input 00 produces output 1, since $(-2)*0+(-2)*0+3=3$ is positive. Here, I've introduced the $*$ symbol to make the multiplications explicit. Similar calculations show that the inputs 01 and 10 produce output 1. But the input 11 produces output 0, since $(-2)*1+(-2)*1+3=-1$ is negative. And so our perceptron implements a NAND gate!

Sigmoid neurons

Learning algorithms sound terrific. But how can we devise such algorithms for a neural network? Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem. For example, the inputs to the network might be the raw pixel data from a scanned, handwritten image of a digit. And we'd like the network to learn weights and biases so that the output from the network correctly classifies the digit. To see how learning might work, suppose we make a small change in some weight (or bias) in the network. What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network. As we'll see in a moment, this property will make learning possible. Schematically, here's what we want (obviously this network is too simple to do handwriting recognition!):



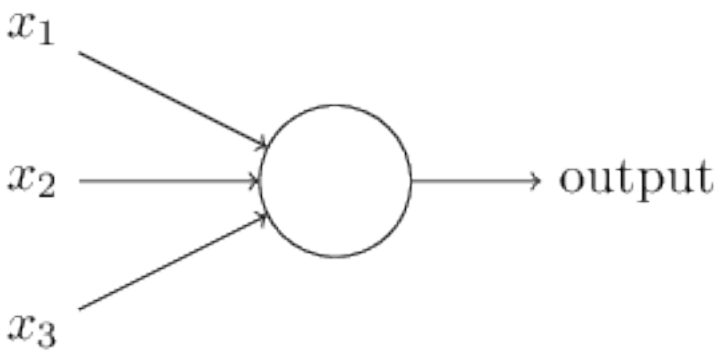
If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. For example, suppose the network was mistakenly classifying an image as an "8" when it should be a "9". We could figure out how to make a small change in the weights and biases so the network gets a little closer to classifying the image as a "9". And then we'd repeat this, changing the weights and biases over and over to produce better and better output. The network would be learning.

The problem is that this isn't what happens when our network contains perceptrons. In fact, a small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip, say from 0 to 1.

That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way. So while your "9" might now be classified correctly, the behaviour of the network on all the other images is likely to have completely changed in some hard-to-control way. That makes it difficult to see how to gradually modify the weights and biases so that the network gets closer to the desired behaviour. Perhaps there's some clever way of getting around this problem. But it's not immediately obvious how we can get a network of perceptrons to learn.

We can overcome this problem by introducing a new type of artificial neuron called a ***sigmoid neuron***. Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output. That's the crucial fact which will allow a network of sigmoid neurons to learn.

Okay, let me describe the sigmoid neuron. We'll depict sigmoid neurons in the same way we depicted perceptrons:

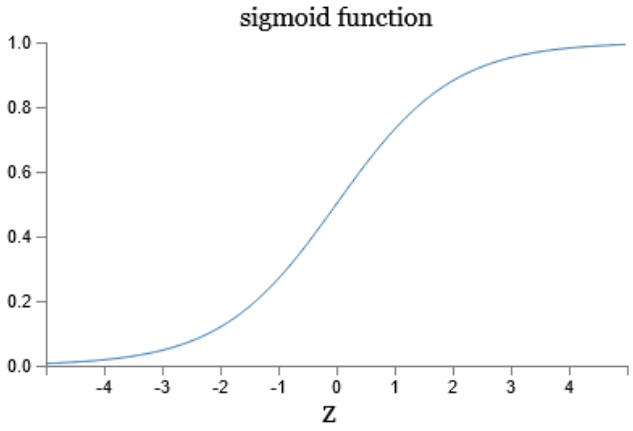


Just like a perceptron, the sigmoid neuron has inputs, x1,x2,.... But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1. So, for instance, 0.638... is a valid input for a sigmoid neuron. Also just like a perceptron, the sigmoid neuron has weights for each input, w1,w2,..., and an overall bias, b. But the output is not 0 or 1. Instead, it's σ(w·x+b), where σ is called the **sigmoid function**, and is defined by: \begin{eqnarray} \sigma(z) \equiv \frac{1}{1+e^{-z}}. \tag{3}\end{eqnarray}

Incidentally, σ is sometimes called the **logistic function,** and this new class of neurons called logistic neurons. It's useful to remember this terminology, since these terms are used by many people working with neural nets. However, we'll stick with the sigmoid terminology.

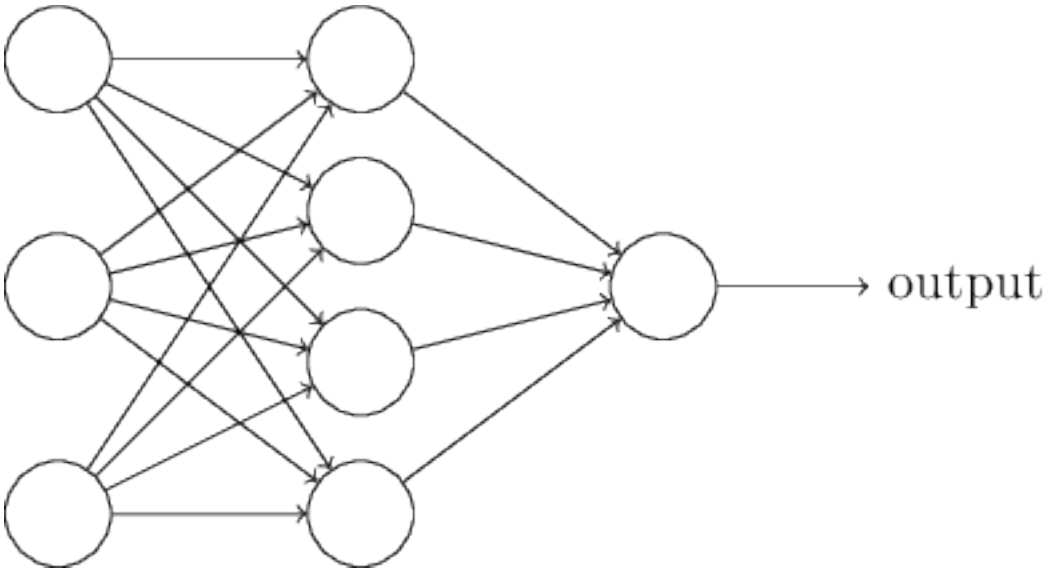
To put it all a little more explicitly, the output of a sigmoid neuron with inputs x1,x2,..., weights w1,w2,..., and bias b is \begin{eqnarray} \frac{1}{1+\exp(-\sum_j w_j x_j-b)}. \tag{4}\end{eqnarray}

What about the algebraic form of σ? How can we understand that? In fact, the exact form of σ isn't so important - what really matters is the shape of the function when plotted. Here's the shape:

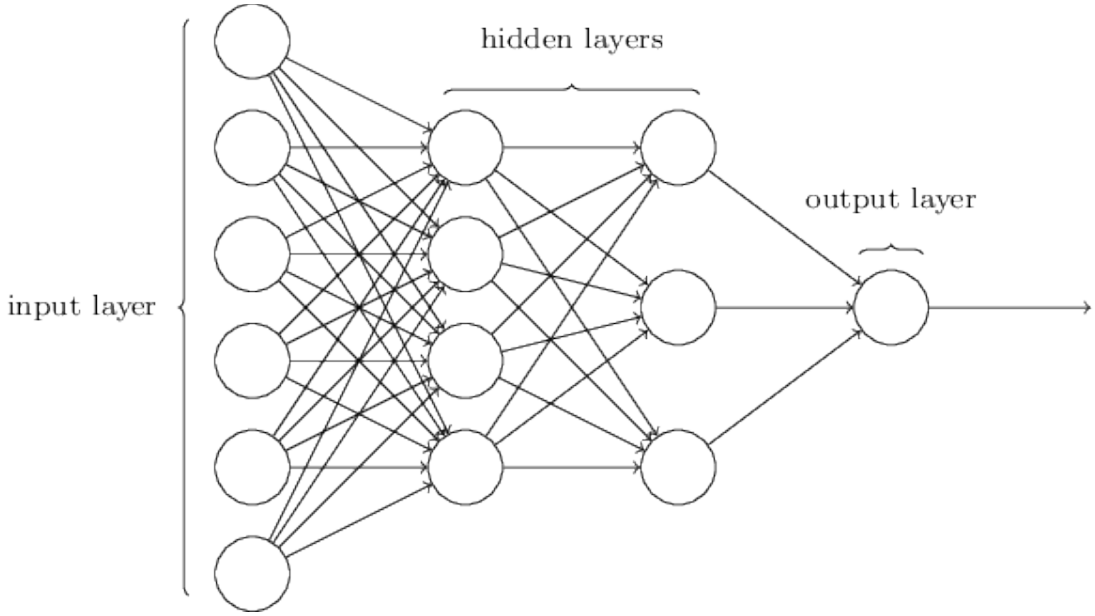


The architecture of neural networks

Before moving to neural networks, it helps to explain some terminology that lets us name different parts of a network. Suppose we have the network:



As mentioned earlier, the leftmost layer in this network is called the input layer, and the neurons within the layer are called input neurons. The rightmost or output layer contains the output neurons, or, as in this case, a single output neuron. The middle layer is called a hidden layer, since the neurons in this layer are neither inputs nor outputs. The term "hidden" perhaps sounds a little mysterious - the first time I heard the term I thought it must have some deep philosophical or mathematical significance - but it really means nothing more than "not an input or an output". The network above has just a single hidden layer, but some networks have multiple hidden layers. For example, the following four-layer network has two hidden layers:



Somewhat confusingly, and for historical reasons, such multiple layer networks are sometimes called *multilayer perceptrons* or MLPs, despite being made up of sigmoid

neurons, not perceptrons. I'm not going to use the MLP terminology in this tutorial, since I think it's confusing, but wanted to warn you of its existence.

The design of the input and output layers in a network is often straightforward. For example, suppose we're trying to determine whether a handwritten image depicts a "9" or not. A natural way to design the network is to encode the intensities of the image pixels into the input neurons. If the image is a 64 by 64 greyscale image, then we'd have 4,096=64×64 input neurons, with the intensities scaled appropriately between 0 and 1. The output layer will contain just a single neuron, with output values of less than 0.5 indicating "input image is not a 9", and values greater than 0.5 indicating "input image is a 9 ".

While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers. In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb. Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets. For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network.

Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer. Such networks are called **feedforward neural networks**. This means there are no loops in the network - information is always fed forward, never fed back. If we did have loops, we'd end up with situations where the input to the σ

function depended on the output. That'd be hard to make sense of, and so we don't allow such loops.

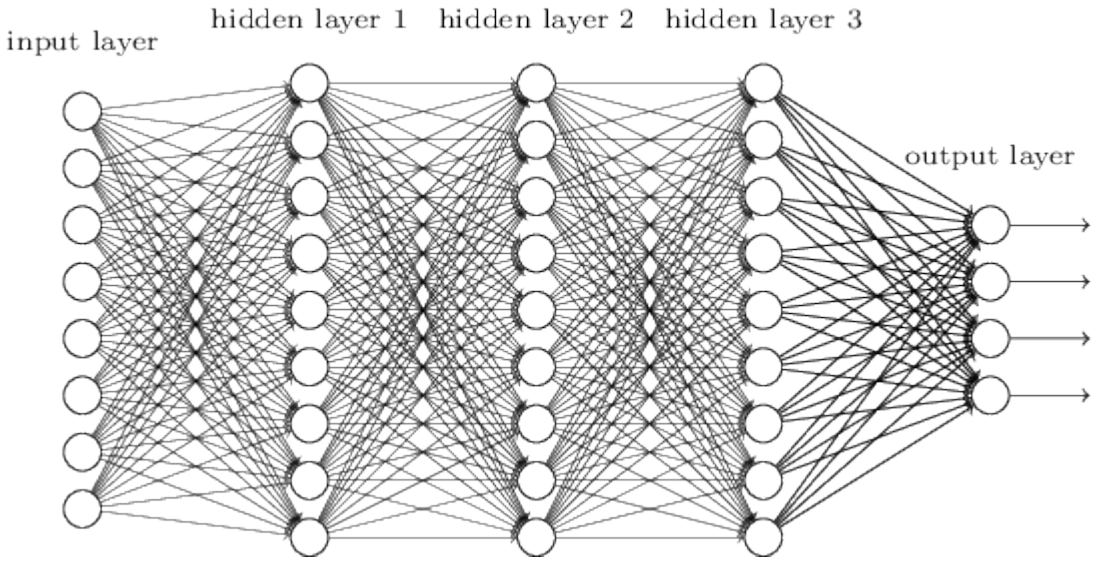
However, there are other models of artificial neural networks in which feedback loops are possible. These models are called recurrent neural networks. The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent. That firing can stimulate other neurons, which may fire a little while later, also for a limited duration. That causes still more neurons to fire, and so over time we get a cascade of neurons firing. Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously.

Recurrent neural nets have been less influential than feedforward networks, in part because the learning algorithms for recurrent nets are (at least to date) less powerful. But recurrent networks are still extremely interesting. They're much closer in spirit to how our brains work than feedforward networks. And it's possible that recurrent networks can solve important problems which can only be solved with great difficulty by feedforward networks. However, to limit our scope, in this tutorial we're going to concentrate on the more widely-used feedforward networks.

Deep Learning

It is a subfield of Machine Learning, inspired by the biological neurons of a brain, and translating that to artificial neural networks with representation learning.

One way of understanding Deep learning is a **deep neural network** with more than one hidden layer. In deep neural network, every neuron in each layer is connected to all neurons in the following layer. In the above example, we have seen two hidden layers that help solve the problem. However, we might need more than two layers for a complex problem. Below is a simple example of a deep neural network.



Why are GPUs or TPUs necessary for training Deep Learning models?

The answer is simple, deep learning is an algorithm – a software construct. We define an artificial neural network in our favorite programming language which would then be converted into a set of commands that run on the computer.

If you would have to guess which components of neural network do you think would require intense hardware resource, what would be your answer?

A few candidates from top of my mind are:

```
Preprocessing input data
Training the deep learning model
Storing the trained deep learning model
Deployment of the model
```

Among all these, **training the deep learning model** is the most intensive task. Lets see in detail why is this so.

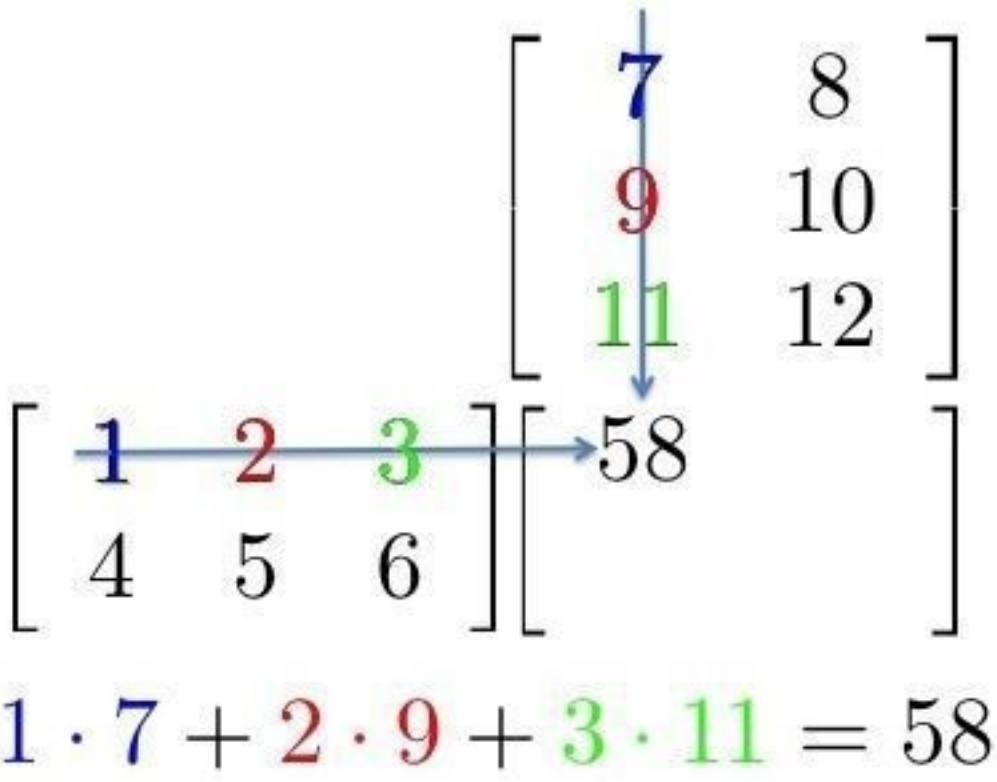
Training a deep learning model

When you train a deep learning model, two main operations are performed:

```
Forward Pass
Backward Pass
```

In forward pass, input is passed through the neural network and after processing the input, an output is generated. Whereas in backward pass, we update the weights of neural network on the basis of error we get in forward pass.

Both of these operations are essentially matrix multiplications. A simple matrix multiplication can be represented by the image below



Here, we can see that each element in one row of first array is multiplied with one column of second array. So in a neural network, we can consider first array as input to the neural network, and the second array can be considered as weights of the network.

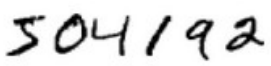
This seems to be a simple task. Now just to give you a sense of what kind of scale deep learning – VGG16 (a convolutional neural network of 16 hidden layers which is frequently used in deep learning applications) has ~140 million parameters; aka weights and biases. Now think of all the matrix multiplications you would have to do to pass just one input to this network! It would take years to train this kind of systems if we take traditional approaches.

We saw that the computationally intensive part of neural network is made up of multiple matrix multiplications. So how can we make it faster?

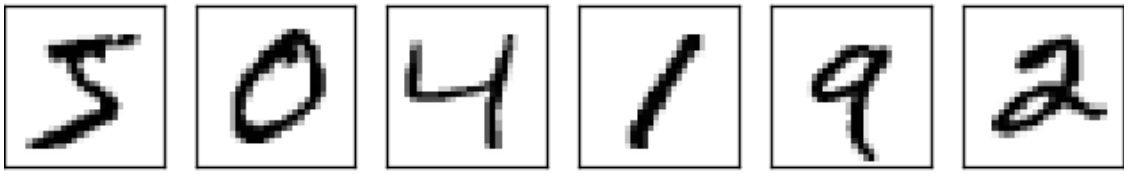
We can simply do this by doing all the operations at the same time instead of doing it one after the other. This is in a nutshell why we use GPU (graphics processing units) instead of a CPU (central processing unit) for training a neural network.

A simple network to classify handwritten digits

Having defined neural networks, let's return to handwriting recognition. We can split the problem of recognizing handwritten digits into two sub-problems. First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit. For example, we'd like to break the image



into six separate images,



We humans solve this segmentation problem with ease, but it's challenging for a computer program to correctly break up the image. Once the image has been segmented, the program then needs to classify each individual digit. So, for instance, we'd like our program to recognize that the first digit above is a 5.

We'll focus on writing a program to solve the second problem, that is, classifying individual digits. We do this because it turns out that the segmentation problem is not so difficult to solve, once you have a good way of classifying individual digits. There are many approaches to solving the segmentation problem.

Activity-II

Reading and understanding the **"Introduction to Neural Networks (NNs)"** content, which is the basis for the next module in the tutorial, Do not skip the content. The ECT for this activity is 180 minutes.

Building the Neural Networks (NNs) using PyTorch

PyTorch

PyTorch is an open-source deep learning framework that's known for its flexibility and ease-of-use. This is enabled in part by its compatibility with the popular Python high-level programming language favored by machine learning developers and data scientists.

What is PyTorch?

PyTorch is a fully featured framework for building deep learning models, which is a type of machine learning that's commonly used in applications like image recognition and language processing. Written in Python, it's relatively easy for most machine learning developers to learn and use. PyTorch is distinctive for its excellent support for GPUs and its use of reverse-mode auto-differentiation, which enables computation graphs to be

modified on the fly. This makes it a popular choice for fast experimentation and prototyping.

Why PyTorch?

PyTorch is the work of developers at Facebook AI Research and several other labs. The framework combines the efficient and flexible GPU-accelerated backend libraries from Torch with an intuitive Python frontend that focuses on rapid prototyping, readable code, and support for the widest possible variety of deep learning models. Pytorch lets developers use the familiar imperative programming approach, but still output to graphs. It was released to open source in 2017, and its Python roots have made it a favorite with machine learning developers.

PyTorch: Tensors

Numpy is a great framework, but it cannot utilize GPUs to accelerate its numerical computations. For modern deep neural networks, GPUs often provide speedups of 50x or greater, so unfortunately numpy won't be enough for modern deep learning.

Here we introduce the most fundamental PyTorch concept: **the Tensor.** A PyTorch Tensor is conceptually identical to a numpy array: a Tensor is an n-dimensional array, and PyTorch provides many functions for operating on these Tensors. Behind the scenes, Tensors can keep track of a computational graph and gradients, but they're also useful as a generic tool for scientific computing.

Also unlike numpy, PyTorch Tensors can utilize GPUs to accelerate their numeric computations. To run a PyTorch Tensor on GPU, you simply need to specify the correct device.

PyTorch: `nn`

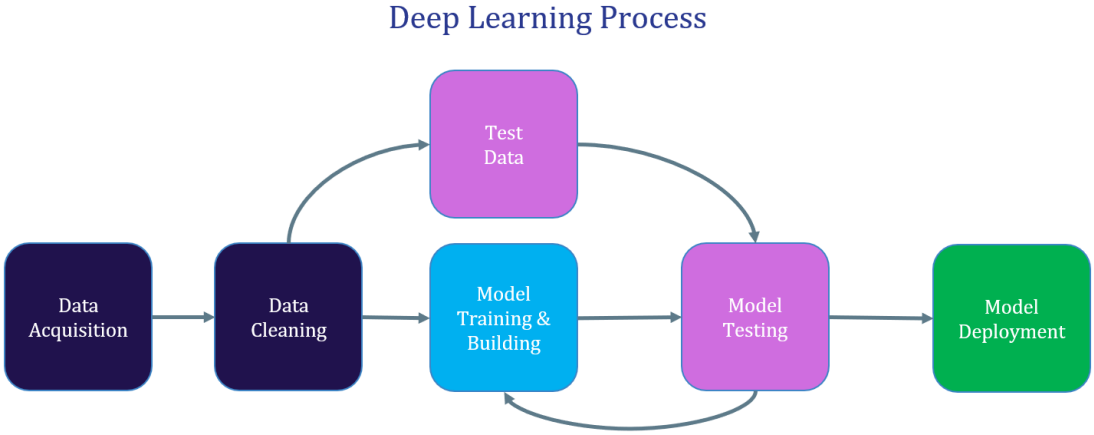
When building neural networks we frequently think of arranging the computation into layers, some of which have learnable parameters which will be optimized during learning.

In PyTorch, the `nn` package serves this same purpose. The `nn` package defines a set of Modules, which are roughly equivalent to neural network layers. A Module receives input Tensors and computes output Tensors, but may also hold internal state such as Tensors containing learnable parameters. The `nn` package also defines a set of useful loss functions that are commonly used when training neural networks.

In this tutorial we use the `nn` package to implement our network:

Similar to the machine learning models we have practiced in previous tutorials, the deep learning model involves four steps.

1. Preprocessing the data (input)
2. Training the model with Training dataset
3. Testing the model with the Test dataset
4. Evaluating the model for performance



Activity-III

Practicing and understanding the "**Building the Neural Networks (NNs) using PyTorch**" content, which is necessary to finish the **TASK**, The ECT for this activity is 180 minutes.

Let's create a neural network with PyTorch library to classify hand written digits(MNIST dataset).

MNIST Dataset

The MNIST database (Modified National Institute of Standards and Technology database) is a large collection of handwritten digits. This is a dataset of 60,000 28x28 `(Width x Height)` grayscale images of the 10 digits, along with a test set of 10,000 images.

Reference: https://paperswithcode.com/dataset/mnist

Step1: Import the necessary libraries

```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
```

Get Device for Training

We want to be able to train our model on a hardware accelerator like the GPU or MPS, if available. Let's check to see if `torch.cuda` or `torch.backends.mps` are available, otherwise we use the CPU.

```
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
Using cpu device
```

Step2: Read or Load the input data

https://pytorch.org/docs/stable/generated/torch.rand.html?highlight=torch+rand

```
X = torch.rand(1, 28, 28, device=device)
```

Step3: Creating a Model

We define our neural network by subclassing `nn.Module`, and initialize the neural network layers in `__init__`. Every `nn.Module` subclass implements the operations on input data in the `forward` method.

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512), # Input layer
            nn.ReLU(),
            nn.Linear(512, 512), # Hidden Layer
            nn.ReLU(),
            nn.Linear(512, 10), # Output Layer
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

We create an instance of `NeuralNetwork`, and move it to the `device`, and print its structure.

```
model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Step4: Traing the model with input data

To use the model, we pass it the input data. This executes the model's `forward`, along with some background operations. Do not call `model.forward()` directly!

Calling the model on the input returns a 2-dimensional tensor with dim=0 corresponding to each output of 10 raw predicted values for each class, and dim=1 corresponding to the individual values of each output.

```
logits = model(X)
```

Step5: Predecting the output

We get the prediction probabilities by passing it through an instance of the `nn.Softmax`module.

```
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

```
Predicted class: tensor([4])
```

Model Layers

Let's break down the layers in the MNIST model. To illustrate it, we will take a sample minibatch of 3 images of size 28x28 and see what happens to it as we pass it through the network.

```
input_image = torch.rand(3,28,28)
print(input_image.size())
```

```
torch.Size([3, 28, 28])
```

`nn.Flatten`

We initialize the `nn.Flatten` layer to convert each 2D 28x28 image into a contiguous array of 784 pixel values (the minibatch dimension (at dim=0) is maintained).

```
flatten = nn.Flatten()
flat_image = flatten(input_image)
print(flat_image.size())
```

```
torch.Size([3, 784])
```

`nn.Linear`

The linear layer is a module that applies a linear transformation on the input using its stored weights and biases.

```
layer1 = nn.Linear(in_features=28*28, out_features=20)
hidden1 = layer1(flat_image)
print(hidden1.size())
```

```
torch.Size([3, 20])
```

`nn.ReLU`

Non-linear activations are what create the complex mappings between the model's inputs and outputs. They are applied after linear transformations to introduce nonlinearity, helping neural networks learn a wide variety of phenomena.

In this model, we use `nn.ReLU` between our linear layers, but there's other activations to introduce non-linearity in your model.

```
print(f"Before ReLU: {hidden1}\n\n")
hidden1 = nn.ReLU()(hidden1)
print(f"After ReLU: {hidden1}")
```

```
Before ReLU: tensor([[ 0.1669, -0.0065,  0.4009,  0.5661,  0.4405, -
0.2603, -0.0608,  0.2876,
          0.6532, -0.6559, -0.4407, -0.2309,  0.1032,  0.2033,
0.1740, -0.7042,
          0.3114,  0.3388, -0.0429, -0.0150],
        [ 0.1650,  0.0568,  0.1223,  0.4750,  0.6952, -0.3338,
0.1217, -0.0136,
          0.7450, -0.7327, -0.4172, -0.5211,  0.3408,  0.3747,
0.0968, -0.4921,
         -0.0494,  0.2923, -0.0322, -0.1361],
        [ 0.6344, -0.0568,  0.0151,  0.6426,  0.2484, -0.6081, -
0.0105, -0.0236,
          0.5583, -0.6851, -0.7161, -0.1594,  0.3168,  0.0012,
0.0451, -0.4680,
         -0.1064,  0.4006, -0.1546,  0.0454]],
grad_fn=<AddmmBackward0>)


After ReLU: tensor([[0.1669, 0.0000, 0.4009, 0.5661, 0.4405, 0.0000,
0.0000, 0.2876, 0.6532,
         0.0000, 0.0000, 0.0000, 0.1032, 0.2033, 0.1740, 0.0000,
0.3114, 0.3388,
         0.0000, 0.0000],
        [0.1650, 0.0568, 0.1223, 0.4750, 0.6952, 0.0000, 0.1217,
0.0000, 0.7450,
         0.0000, 0.0000, 0.0000, 0.3408, 0.3747, 0.0968, 0.0000,
0.0000, 0.2923,
         0.0000, 0.0000],
        [0.6344, 0.0000, 0.0151, 0.6426, 0.2484, 0.0000, 0.0000,
0.0000, 0.5583,
         0.0000, 0.0000, 0.0000, 0.3168, 0.0012, 0.0451, 0.0000,
0.0000, 0.4006,
         0.0000, 0.0454]], grad_fn=<ReluBackward0>)
```

`nn.Sequential`

`nn.Sequential` is an ordered container of modules. The data is passed through all the modules in the same order as defined. You can use sequential containers to put together a quick network like `seq_modules`.

```
seq_modules = nn.Sequential(
    flatten,
    layer1,
    nn.ReLU(),
    nn.Linear(20, 10)
)
input_image = torch.rand(3,28,28)
logits = seq_modules(input_image)
```

`nn.Softmax`

The last linear layer of the neural network returns `logits` - raw values in [-infty, infty] - which are passed to the `nn.Softmax` module. The logits are scaled to values [0, 1] representing the model's predicted probabilities for each class. `dim` parameter indicates the dimension along which the values must sum to 1.

```
softmax = nn.Softmax(dim=1)
pred_probab = softmax(logits)
```

```
pred_probab
```

```
tensor([[0.0948, 0.0976, 0.1363, 0.0998, 0.0811, 0.1048, 0.0876,
0.0889, 0.1345,
         0.0745],
        [0.0939, 0.0985, 0.1110, 0.1155, 0.0849, 0.0985, 0.1006,
0.1060, 0.1220,
         0.0690],
        [0.0967, 0.0959, 0.1172, 0.1117, 0.0810, 0.0987, 0.0958,
0.0979, 0.1310,
         0.0741]], grad_fn=<SoftmaxBackward0>)
```

Model Parameters

Many layers inside a neural network are `parameterized`, i.e. have associated weights and biases that are optimized during training. Subclassing nn.Module automatically tracks all fields defined inside your model object, and makes all parameters accessible using your model's `parameters()` or `named_parameters()` methods.

In this example, we iterate over each parameter, and print its size and a preview of its values.

```
print(f"Model structure: {model}\n\n")

for name, param in model.named_parameters():
    print(f"Layer: {name} | Size: {param.size()} | Values :
{param[:2]} \n")
```

```
Model structure: NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
```

```
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)


Layer: linear_relu_stack.0.weight | Size: torch.Size([512, 784]) |
Values : tensor([[ 0.0088, -0.0354, -0.0083,  ...,  0.0196,  0.0340, -
0.0107],
        [ 0.0323,  0.0039,  0.0197,  ...,  0.0149,  0.0335, -0.0242]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.0.bias | Size: torch.Size([512]) | Values :
tensor([-0.0198,  0.0101], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.weight | Size: torch.Size([512, 512]) |
Values : tensor([[ 0.0252, -0.0008,  0.0131,  ...,  0.0020,  0.0031,
0.0023],
        [ 0.0226, -0.0136,  0.0385,  ..., -0.0126,  0.0370, -0.0203]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.2.bias | Size: torch.Size([512]) | Values :
tensor([0.0346, 0.0091], grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.weight | Size: torch.Size([10, 512]) |
Values : tensor([[ 0.0087,  0.0369, -0.0012,  ...,  0.0114,  0.0004, -
0.0081],
        [ 0.0375,  0.0182,  0.0311,  ...,  0.0100, -0.0303,  0.0360]],
       grad_fn=<SliceBackward0>)

Layer: linear_relu_stack.4.bias | Size: torch.Size([10]) | Values :
tensor([ 0.0077, -0.0015], grad_fn=<SliceBackward0>)
```

Like linear and logistic regression, we use "model" to create the neural network model.

Now, we do the same for all 9 digits

Step1: Import the necessary libraries

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

# Get Device for Training
device = (
```

```
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
Using cpu device
```

Step2: Read or Load the input data

```
# Download training data from open datasets.
training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)
```

Print and check the data

```
# Create data loaders.
train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

for X, y in test_dataloader:
    print(f"Shape of X [N, C, H, W]: {X.shape}")
    print(f"Shape of y: {y.shape} {y.dtype}")
    break
```

```
Shape of X [N, C, H, W]: torch.Size([64, 1, 28, 28])
Shape of y: torch.Size([64]) torch.int64
```

- N: number of images in the **batch** (We will talk about it in a minute)
- C: number of channels of the image (ex: 3 for RGB, 1 for grayscale...)
- H: height of the image
- W: width of the image

```
import plotly.express as px

images, labels = next(iter(train_dataloader)) # Load all images and
their labels
```

```
# print a single image in train dataset
px.imshow(images[0].reshape(28,28), color_continuous_scale="gray")

# print the label for the above image from train dataset
print(labels[0]) # Print
```

```
tensor(5)
```

Step3: Creating a Model

```
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_model = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_model(x)
        return logits

model = NeuralNetwork().to(device)
print(model)
```

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_model): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Step4: Traing the model with traing data

Here we need to talk about few model parameters to improve training.

Loss Function:

A loss function is a function that compares the target and predicted output values; measures how well the neural network models the training data. When training, we aim to minimize this loss between the predicted and target outputs.

Stochastic Gradient Descent:

Stochastic Gradient Descent, or SGD for short, is an **optimization** algorithm used to train machine learning algorithms, most notably artificial neural networks used in deep learning.

The job of the algorithm is to find a set of internal model parameters that perform well against some performance measure such as logarithmic loss or mean squared error.

Optimization is a type of searching process and you can think of this search as learning. The optimization algorithm is called “gradient descent“, where “gradient” refers to the calculation of an error gradient or slope of error and “descent” refers to the moving down along that slope towards some minimum level of error.

The algorithm is iterative. This means that the search process occurs over multiple discrete steps, each step hopefully slightly improving the model parameters.

Each step involves using the model with the current set of internal parameters to make predictions on some samples, comparing the predictions to the real expected outcomes, calculating the error, and using the error to update the internal model parameters.

This update procedure is different for different algorithms, but in the case of artificial neural networks, the backpropagation update algorithm is used.

Before we dive into batches and epochs, let’s take a look at what we mean by sample.

What Is a Sample?

A sample is a single row of data.

It contains inputs that are fed into the algorithm and an output that is used to compare to the prediction and calculate an error.

A training dataset is comprised of many rows of data, e.g. many samples. A sample may also be called an instance, an observation, an input vector, or a feature vector.

Now that we know what a sample is, let's define a batch.

What Is a Batch?

The batch size is a hyperparameter that defines the number of samples to work through before updating the internal model parameters.

Think of a batch as a for-loop iterating over one or more samples and making predictions. At the end of the batch, the predictions are compared to the expected output variables and an error is calculated. From this error, the update algorithm is used to improve the model, e.g. move down along the error gradient.

A training dataset can be divided into one or more batches.

When all training samples are used to create one batch, the learning algorithm is called batch gradient descent. When the batch is the size of one sample, the learning algorithm is called stochastic gradient descent. When the batch size is more than one sample and less than the size of the training dataset, the learning algorithm is called mini-batch gradient descent.

```
Batch Gradient Descent. Batch Size = Size of Training Set
Stochastic Gradient Descent. Batch Size = 1
Mini-Batch Gradient Descent. 1 < Batch Size < Size of Training Set
```

In the case of mini-batch gradient descent, popular batch sizes include 32, 64, and 128 samples. You may see these values used in models in the literature and in tutorials.

What Is an Epoch?

The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset.

One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches. For example, as above, an epoch that has one batch is called the batch gradient descent learning algorithm.

You can think of a for-loop over the number of epochs where each loop proceeds over the training dataset. Within this for-loop is another nested for-loop that iterates over each batch of samples, where one batch has the specified “batch size” number of samples.

The number of epochs is traditionally large, often hundreds or thousands, allowing the learning algorithm to run until the error from the model has been sufficiently minimized. You may see examples of the number of epochs in the literature and in tutorials set to 10, 100, 500, 1000, and larger.

It is common to create line plots that show epochs along the x-axis as time and the error or skill of the model on the y-axis. These plots are sometimes called learning curves. These plots can help to diagnose whether the model has over learned, under learned, or is suitably fit to the training dataset.

Worked Example

Finally, let’s make this concrete with a small example.

Assume you have a dataset with 200 samples (rows of data) and you choose a batch size of 5 and 1,000 epochs.

This means that the dataset will be divided into 40 batches, each with five samples. The model weights will be updated after each batch of five samples.

This also means that one epoch will involve 40 batches or 40 updates to the model.

With 1,000 epochs, the model will be exposed to or pass through the whole dataset 1,000 times. That is a total of 40,000 batches during the entire training process.

```python
loss_fn = nn.CrossEntropyLoss() # Loss Function
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3) #Optimizer
batch_size = 64
epochs=5

size = len(train_dataloader.dataset)
model.train()

for e in range(epochs):
  for batch, (X, y) in enumerate(train_dataloader):
    X, y = X.to(device), y.to(device)

    # Compute prediction error
    pred = model(X)
    loss = loss_fn(pred, y)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if batch % 100 == 0:
      loss, current = loss.item(), (batch + 1) * len(X)
      print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

```
loss: 2.302791  [   64/60000]
loss: 2.295357  [ 6464/60000]
loss: 2.295498  [12864/60000]
loss: 2.280337  [19264/60000]
loss: 2.276518  [25664/60000]
loss: 2.276347  [32064/60000]
loss: 2.268280  [38464/60000]
loss: 2.269387  [44864/60000]
loss: 2.251013  [51264/60000]
loss: 2.251429  [57664/60000]
loss: 2.251251  [   64/60000]
loss: 2.240841  [ 6464/60000]
loss: 2.252542  [12864/60000]
loss: 2.212824  [19264/60000]
loss: 2.221383  [25664/60000]
loss: 2.222712  [32064/60000]
loss: 2.203198  [38464/60000]
loss: 2.222115  [44864/60000]
loss: 2.184353  [51264/60000]
loss: 2.179831  [57664/60000]
loss: 2.178211  [   64/60000]
loss: 2.161520  [ 6464/60000]
loss: 2.187181  [12864/60000]
loss: 2.110797  [19264/60000]
```

```
loss: 2.133548  [25664/60000]
loss: 2.134501  [32064/60000]
loss: 2.096858  [38464/60000]
loss: 2.138575  [44864/60000]
loss: 2.073570  [51264/60000]
loss: 2.059387  [57664/60000]
loss: 2.054154  [   64/60000]
loss: 2.024892  [ 6464/60000]
loss: 2.070829  [12864/60000]
loss: 1.939289  [19264/60000]
loss: 1.975628  [25664/60000]
loss: 1.973789  [32064/60000]
loss: 1.913971  [38464/60000]
loss: 1.985039  [44864/60000]
loss: 1.881910  [51264/60000]
loss: 1.854367  [57664/60000]
loss: 1.845438  [   64/60000]
loss: 1.794210  [ 6464/60000]
loss: 1.864704  [12864/60000]
loss: 1.673980  [19264/60000]
loss: 1.711768  [25664/60000]
loss: 1.706231  [32064/60000]
loss: 1.632226  [38464/60000]
loss: 1.740378  [44864/60000]
loss: 1.599673  [51264/60000]
loss: 1.560951  [57664/60000]
```

Step5: Testing or Evaluating the model with testing data

First, we will test with the single image then we will go for the entire dataset.

```
# Load all test images and their labels
test_imgs, test_labels = next(iter(test_dataloader))
```

Case1: Where model prediction is correct

```
# print a single image and it's label
img_0 = test_imgs[0].reshape(28,28)
px.imshow(img_0, color_continuous_scale="gray")

# Print actual label
label_0 = test_labels[0]
print(label_0)

tensor(7)

img_0 = img_0.reshape(1,28,28)

# Print predicted label
pred= model(img_0)
pred_probab = nn.Softmax(dim=1)(pred)
```

```
img_0_pred_label = pred_probab.argmax(1)
print(img_0_pred_label)
```

```
tensor([7])
```

Case2: Where model prediction is wrong

```
# print a single image and it's label
img_63 = test_imgs[63].reshape(28,28)
px.imshow(img_63, color_continuous_scale="gray")
```

```
# Print actual label
label_63 = test_labels[63]
print(label_63)
```

```
tensor(3)
```

```
img_63 = img_63.reshape(1,28,28)
```

```
# Print predicted label
pred= model(img_63)
pred_probab = nn.Softmax(dim=1)(pred)
img_63_pred_label = pred_probab.argmax(1)
print(img_63_pred_label)
```

```
tensor([2])
```

Testing with the entire dataset

```
size = len(test_dataloader.dataset)
num_batches = len(test_dataloader)
model.eval()
test_loss, correct = 0, 0
with torch.no_grad():
  for X, y in test_dataloader:
    X, y = X.to(device), y.to(device)
    pred = model(X)
    test_loss += loss_fn(pred, y).item()
    correct += (pred.argmax(1) == y).type(torch.float).sum().item()
test_loss /= num_batches
correct /= size
print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

```
Test Error:
 Accuracy: 73.9%, Avg loss: 1.535577
```

Step6: Save the model

```
torch.save(model.state_dict(), "model.pth")
print("Saved PyTorch Model State to model.pth")
```

```
Saved PyTorch Model State to model.pth
```

Step7: Load the model and use it

```
model = NeuralNetwork()
model.load_state_dict(torch.load("model.pth"))
```

```
<All keys matched successfully>
```

```
# print a single image and it's label
img_1 = test_imgs[1].reshape(28,28)
px.imshow(img_1, color_continuous_scale="gray")
```

```
# Print actual label
label_1 = test_labels[1]
print(label_1)
```

```
tensor(2)
```

```
img_1 = img_1.reshape(1,28,28)
```

```
# Print predicted label
pred= model(img_1)
pred_probab = nn.Softmax(dim=1)(pred)
img_1_pred_label = pred_probab.argmax(1)
print(img_1_pred_label)
```

```
tensor([3])
```

TASK I

The ECT for this TASK is 180 minutes.

a. Report the results after increasing the batch size to 32 and the number of epochs to 10. In the **Tutorial Completion Document**, record your observations (e.g., a change in accuracy).

```
# Import the necessary libraries
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

# Get Device for Training
device = (
    "cuda"
    if torch.cuda.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

```
# Read or Load the input data
# Download training data from open datasets.
training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor(),
)

# Creating a Model
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_model = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_model(x)
        return logits

model = NeuralNetwork().to(device)

# Traing the model with traing data
loss_fn = nn.CrossEntropyLoss() # Loss Function
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3) #Optimizer
print("Using batch size 32 and the number of epochs 10 \n \n")
batch_size = 32
epochs=10

train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

size = len(train_dataloader.dataset)
```

```
model.train()

for t in range(epochs):
  for batch, (X, y) in enumerate(train_dataloader):
    X, y = X.to(device), y.to(device)

    # Compute prediction error
    pred = model(X)
    loss = loss_fn(pred, y)

    # Backpropagation
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if batch % 100 == 0:
      loss, current = loss.item(), (batch + 1) * len(X)
      # print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

# Testing or Evaluating the model with testing data
size = len(test_dataloader.dataset)
num_batches = len(test_dataloader)
model.eval()
test_loss, correct = 0, 0
with torch.no_grad():
  for X, y in test_dataloader:
    X, y = X.to(device), y.to(device)
    pred = model(X)
    test_loss += loss_fn(pred, y).item()
    correct += (pred.argmax(1) == y).type(torch.float).sum().item()
test_loss /= num_batches
correct /= size
print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

```
Using cpu device
Using batch size 32 and the number of epochs 10


Test Error:
 Accuracy: 89.4%, Avg loss: 0.383376
```

Description

The model created above is a neural network that has three fully connected linear layers with ReLU activation functions between each of them. The input layer is a Flatten() layer which takes in an input of 28x28 image pixels and flattens it to a 1-dimensional tensor of size 784. This flattened input is passed through two hidden layers, each with 512 neurons

and ReLU activation function. The output layer has 10 neurons, which correspond to the 10 possible digit classes in the MNIST dataset.

The model is trained on the MNIST dataset using the stochastic gradient descent (SGD) optimizer and cross-entropy loss function. The training is performed for 10 epochs with a batch size of 32. After training, the model is evaluated on the test dataset, which also contains images of handwritten digits. The test error is calculated and reported as an accuracy of 89.4% with an average loss of 0.383376. This means that the model is able to correctly classify 89.4% of the images in the test dataset.

b. Construct a new model with an additional hidden layer containing the input and output neurons listed below.

- input layer = 28*28 - 512
- First hidden layer = 512 - 256
- Second hidden layer = 256 - 128
- Final layer = 128 -10

Train the new model with the same MNIST data and document any observations (e.g., accuracy change) in the **Tutorial Completion Document**.

```
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

# Get Device for Training
device = (
    "cuda"
    if torch.cuda.is_available()
    else "cpu"
)
print(f"Using {device} device")

# Read or Load the input data
# Download training data from open datasets.
training_data = datasets.MNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor(),
)

# Download test data from open datasets.
test_data = datasets.MNIST(
    root="data",
    train=False,
    download=True,
```

```
    transform=ToTensor(),
)

# Creating a Model
# Define model
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_model = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 256),
            nn.ReLU(),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_model(x)
        return logits

model = NeuralNetwork().to(device)

# Train the model with training data
loss_fn = nn.CrossEntropyLoss() # Loss Function
optimizer = torch.optim.SGD(model.parameters(), lr=1e-3) # Optimizer
print("Training the model...\n")
batch_size = 64
epochs = 5
train_dataloader = DataLoader(training_data, batch_size=batch_size)
test_dataloader = DataLoader(test_data, batch_size=batch_size)

size = len(train_dataloader.dataset)
model.train()

for t in range(epochs):
    for batch, (X, y) in enumerate(train_dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
```

```
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

# Evaluate the model with test data
size = len(test_dataloader.dataset)
num_batches = len(test_dataloader)
model.eval()
test_loss, correct = 0, 0
with torch.no_grad():
    for X, y in test_dataloader:
        X, y = X.to(device), y.to(device)
        pred = model(X)
        test_loss += loss_fn(pred, y).item()
        correct += (pred.argmax(1) ==
y).type(torch.float).sum().item()
test_loss /= num_batches
correct /= size
print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

```
Using cpu device
Training the model...

loss: 2.311015  [   64/60000]
loss: 2.319871  [ 6464/60000]
loss: 2.305876  [12864/60000]
loss: 2.292568  [19264/60000]
loss: 2.311316  [25664/60000]
loss: 2.319512  [32064/60000]
loss: 2.297016  [38464/60000]
loss: 2.301329  [44864/60000]
loss: 2.297458  [51264/60000]
loss: 2.281071  [57664/60000]
loss: 2.302502  [   64/60000]
loss: 2.309677  [ 6464/60000]
loss: 2.298776  [12864/60000]
loss: 2.284657  [19264/60000]
loss: 2.303187  [25664/60000]
loss: 2.309987  [32064/60000]
loss: 2.287924  [38464/60000]
loss: 2.295477  [44864/60000]
loss: 2.289294  [51264/60000]
loss: 2.273911  [57664/60000]
loss: 2.293485  [   64/60000]
loss: 2.298637  [ 6464/60000]
loss: 2.290488  [12864/60000]
loss: 2.275564  [19264/60000]
```

```
loss: 2.293750  [25664/60000]
loss: 2.299189  [32064/60000]
loss: 2.276587  [38464/60000]
loss: 2.288345  [44864/60000]
loss: 2.278847  [51264/60000]
loss: 2.264109  [57664/60000]
loss: 2.282489  [   64/60000]
loss: 2.285666  [ 6464/60000]
loss: 2.279644  [12864/60000]
loss: 2.263458  [19264/60000]
loss: 2.281197  [25664/60000]
loss: 2.285528  [32064/60000]
loss: 2.261348  [38464/60000]
loss: 2.278698  [44864/60000]
loss: 2.264370  [51264/60000]
loss: 2.249504  [57664/60000]
loss: 2.267586  [   64/60000]
loss: 2.268415  [ 6464/60000]
loss: 2.264083  [12864/60000]
loss: 2.245564  [19264/60000]
loss: 2.263866  [25664/60000]
loss: 2.267205  [32064/60000]
loss: 2.239755  [38464/60000]
loss: 2.264356  [44864/60000]
loss: 2.243555  [51264/60000]
loss: 2.227903  [57664/60000]
Test Error:
 Accuracy: 52.6%, Avg loss: 2.240703
```

Description

This PyTorch model is a neural network designed to classify handwritten digits from the MNIST dataset. It's built using PyTorch's neural network module, which provides classes for building and training neural networks.

Firstly, the code sets the device on which the model will be trained, either "cuda" if it's available or the CPU if it's not. Then, it reads and transforms the MNIST dataset using PyTorch's built-in functions to create the training and testing datasets.

The neural network model is defined by creating a NeuralNetwork class that inherits from nn.Module. The model consists of several fully connected layers with ReLU activation functions, which use the flatten layer to transform the input images into a 1D tensor, and the final layer outputs a tensor of size 10, representing the 10 possible classes.

The model is trained on the training data using the stochastic gradient descent (SGD) optimizer with a learning rate of 1e-3 and cross-entropy loss function. The training loop runs for a specified number of epochs, with each epoch consisting of multiple batches. The model is set to train mode using the model.train() method.

Finally, the model is evaluated on the test data using the same cross-entropy loss function. The test loop runs through all the test data in batches and calculates the accuracy of the model. The final test error and accuracy are printed to the console.

Overall, this PyTorch model provides a basic implementation of a neural network for image classification tasks, and demonstrates how to use PyTorch's built-in functions for data loading, model building, training, and evaluation.

References

1. https://iq.opengenus.org/cpu-vs-gpu-vs-tpu/
2. https://www.quora.com/What-is-the-difference-between-GPUs-CPUs-and-TPUs
3. https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/
4. https://www.youtube.com/watch?v=inN8seMm7UI
5. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/