

Google AI for JavaScript developers with TensorFlow.js

Course Introduction

Artificial intelligence and machine learning technologies are revolutionizing the world at an accelerated pace. Combine the power of these technologies with JavaScript, the most popular programming language in the world, and you can supercharge your websites and applications to act intelligently!

In this chapter, I will introduce you to TensorFlow.js, Google's machine learning library for JavaScript developers. You will understand why it makes sense to perform machine learning using JavaScript and get a broad idea of what you will be able to do at the end of the course. You'll also explore the TensorFlow.js community and get an idea of the type of projects that are being currently developed using this library.

Documentacion de TensorFlow

<https://www.tensorflow.org/?hl=es-419>

<https://js.tensorflow.org/api/latest/>

repositorio de modelos entrenados: <https://www.tensorflow.org/hub?hl=es>

What is TensorFlow.js

TensorFlow.js is Google's Machine Learning Library for JavaScript, and it's just like the original version of TensorFlow that came out in Python, but is aimed at folk who code in JavaScript or Node.js instead.

The screenshot shows the TensorFlow.js homepage. At the top, there's a navigation bar with links for 'Install' (with a download icon), 'Learn', 'API', 'Resources', 'Community', and 'Why TensorFlow'. To the right of the navigation is a search bar with a magnifying glass icon and a 'Language' dropdown menu. Below the navigation, a banner for 'For JavaScript' has tabs for 'Overview', 'Tutorials', 'Guide', 'Models', 'Demos', and 'API'. A call-to-action button 'Join Challenge' is visible. The main content area features a large orange graphic on the left and a central illustration of a computer monitor displaying a grid of data points, with various colored lines (blue, orange, grey) connecting them to the screen, symbolizing machine learning models. Text on the page includes: 'TensorFlow.js is a library for machine learning in JavaScript', 'Develop ML models in JavaScript, and use ML directly in the browser or in Node.js.', 'See tutorials', 'See models', 'See demos', and descriptive text for each: 'Tutorials show you how to use TensorFlow.js with complete, end-to-end examples.', 'Pre-trained, out-of-the-box models for common use cases.', and 'Live demos and examples run in your browser using TensorFlow.js.'

Why Perform Machine Learning in JavaScript?

First, it enables more developers to use machine learning without having to learn a new language, given that around 70% of developers use JavaScript already in production. Furthermore, developers can use their creations in even more places than ever before, as JavaScript can run pretty much anywhere.

From the client-side in the browser to server-side via Node.js, mobile native to desktop native apps, and even IoT devices like a Raspberry Pi, JavaScript is the only language that has this flexibility of execution.

Coding your creations in JavaScript allows easy shareability with the reach and scale of the web to billions of users around the world along with a number of benefits that are impossible to achieve server-side as you will learn later in the course, like privacy, as no data needs to be sent to the cloud.

Get empowered to make web apps with superpowers that your customers will love

By the end of this course, you will understand how to create next-generation web apps that can give you superpowers vs. traditional websites, allowing you to offer your customers capabilities that traditional web engineering could not.

This course assumes no prior knowledge of the machine learning industry, and will kick things off by demystifying artificial intelligence and machine learning, and then swiftly move on to learn how to be productive using TensorFlow.js through a variety of hands-on examples.

From off the shelf solutions you can use in minutes to roll your own custom code, you will have a fundamental understanding across the spectrum enabling you to be productive and then go further in the world of machine learning.

Machine learning is already fast impacting every major industry out there, and the pace of innovation in this space will continue to rise in the years to come, with the potential to influence every business you have ever worked with.

So now is the perfect time to learn more about it and take your first steps — no matter what background you may be from — to provide you with competitive skills for your future career. More customers than ever are requesting or expecting smarter systems powered by machine learning every day as it becomes the new normal.

Course overview:

Chapter 1: Course Introduction

- Explore machine learning through the lens of JavaScript.
- Understand what you can do by the end of the course and how others are using TensorFlow.js

Chapter 2: Introduction to Machine Learning and TensorFlow.js

- Explore how Artificial Intelligence, Machine Learning, and Deep Learning relate to each other.
- Understand how machines learn and explore the various types of machine learning techniques that started it all
- Discover the advantages of using TensorFlow.js, Google's machine learning library for JavaScript developers.
- Understand the three ways to use machine learning:
 1. Using pre-made and off-the-shelf models
 2. Creating custom models
 3. Transfer learning

Chapter 3: Using Pre-made Models in TensorFlow.js

- Understand what pre-made models are.
- Explore using common pre-made models in TensorFlow.js from object detection to human pose estimation and more.
- Learn how to select the right model for a given task based on key criteria such as speed, memory usage, and file size.

- Implement a smart security camera project by using a pre-made object detection model that can detect when an object of interest is in view.
- Learn to work with Tensors - the fundamental data structure of machine learning.
- Understand what a raw TensorFlow.js model consists of.
- Implement loading a raw TensorFlow.js model, and send data into the model using Tensors to get predictions that you can actually use and interpret.
- Implement loading a more advanced raw model from TensorFlow Hub that can detect human poses that will require significant pre and post processing.

Chapter 4: Writing Custom Models in TensorFlow.js

- Understand when rolling your own models makes sense
- Learn about how to gather datasets that can be split for training, testing, and validation.
- Understand the importance of using clean and unbiased data to train a model
- Learn about Perceptrons / Neurons - the basic building blocks of machine learning.
- Explore using Neurons to perform linear regression
- Implement defining and training your first model for linear regression to solve a real problem using a single perceptron.
- Go deeper and implement a multi-layered perceptron model that performs classification.
- Discuss other ML architectures and their applications and understand the basics of Convolutional Neural Networks for image recognition.

Chapter 5: Understand Transfer Learning - Retrain Existing Models

- Explore the concept of Transfer Learning in ML.
- Develop your own simple version of the popular Teachable machine website to retrain models live in the browser to recognize new objects that it has never seen before!

Chapter 6: Reuse Machine Learning Models Created in Python

- Understand how to reuse models made in Python using TensorFlow.js.
- Reuse a pre-existing Python model in TensorFlow.js for comment spam detection.
- Customize your model to detect custom edge cases.
- Discuss which Python models are suitable for conversion.

Chapter 7: To the Future and Beyond

- Explore what other areas of Machine Learning exist that you may want to pursue after the course along with further learning resources that may be of interest to continue your TensorFlow.js journey.
- Connect with the global TensorFlow.js community to get further support and inspiration now that you have the knowledge to use ML in JS.

Introduction to ML and [TensorFlow.js](#)

What You Will Learn

By the end of the chapter, you will be able to:

- Differentiate between traditional programming paradigms and the machine learning paradigm
- Compare and contrast AI, ML, and Deep Learning
- Describe the three types of machine learning – supervised, unsupervised, and reinforcement learning
- Explain the process of training a typical machine learning system
- Define features or attributes of data used in machine learning
- Define regression problems with examples
- Define classification problems with examples
- Explain the differences between regression and classification problems
- Describe what TensorFlow.js is
- Enumerate the advantages of performing machine learning using TensorFlow.js in JavaScript environments
- Explain the three ways of performing machine learning – using pre-made models, using transfer learning, and rolling out custom models

The **Introduction to ML and TensorFlow.js** chapter lays the foundation for what you will be learning throughout this course. In this chapter, you will

first address the elephant in the room by discussing the difference between artificial intelligence, machine learning, and deep learning, ensuring a level playground for those taking this course standalone. Once you have understood these concepts, we will demystify machine learning by discussing the three types of ML – supervised, unsupervised, and reinforcement with special focus on supervised examples in this course.

Next, you will see how a typical ML system is trained in a way any JS engineer could grasp. Then you will learn what TensorFlow.js (TFJS) is using simple graphs and charting to see how they can be used to separate data (classification), or draw a line of best fit (regression) and explore the history of the TensorFlow.js library along with some of the benefits of performing ML in JavaScript.

Towards the end of this chapter, you will learn the three ways you can use or create models with TFJS, namely, use pre-made models, transfer learning, and write your own from a blank canvas.

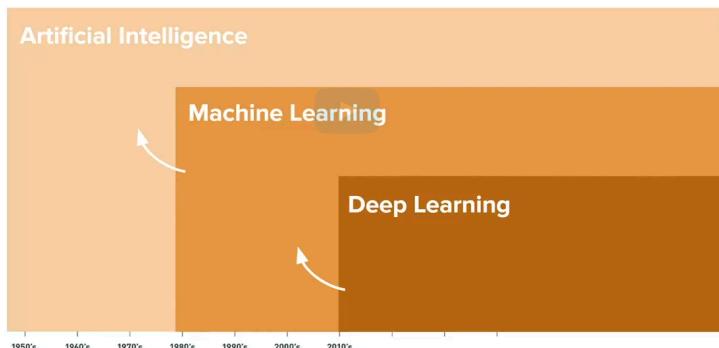
Artificial Intelligence, Machine Learning, and Deep Learning

Summary

(AI + ML + DL)

These concepts go back quite some time!

Deep learning drives machine learning, which then ultimately can enable artificial intelligence.



Core Concepts of AI and Machine Learning

1. The Hierarchy of Definitions

- **Artificial Intelligence (AI):** The broad science of making machines smart, or "human intelligence exhibited by machines." Currently, we use **Narrow AI**, which focuses on systems that perform specific tasks (like medical imaging or text routing) as well as or better than human experts.
- **Machine Learning (ML):** An approach to achieving AI. It is the implementation of programs that learn from data to find patterns. Instead of hard-coding rules, you feed the system data, and it learns to classify information it hasn't seen before.
- **Deep Learning:** A specific technique within ML that uses **Deep Neural Networks**. These are layers of code that loosely mimic the human brain. The deeper the network, the more complex the patterns it can recognize (e.g., from simple lines to human faces).

2. Traditional Programming vs. Machine Learning

- **Traditional Programming:** Relies on manual "if-then" logic. If the problem changes (like a spammer changing their keywords), the code breaks, and the programmer must manually update it.
- **ML Approach:** You use the same code but feed it different data. To switch from a "cat recognizer" to a "dog recognizer," you don't change the code; you simply provide different training images.

3. Common Use Cases

- **Computer Vision:** * *Image Recognition*: Identifying *what* is in an image.
 - *Object Detection*: Identifying *what, where*, and *how many* objects are in an image.
- **Linear Regression**: Predicting a numerical value based on another (e.g., predicting house prices based on square footage).
- **Natural Language Processing (NLP)**: Understanding human language to detect spam, perform sentiment analysis (positive/negative), or summarize long texts.
- **Generative/Creative AI**: Models that can create entirely new content, such as generating realistic human faces that do not exist or turning a voice into a musical instrument.

4. Key Benefits of ML

- **Speed & Reliability**: It is much faster to train a model with images than to spend weeks trying to code manual rules for edge detection or color filtering.
- **Personalization**: Existing solutions can be quickly adapted for different users or industries just by changing the training dataset.
- **Solving "Unsolvable" Problems**: Tasks that humans do intuitively (like facial recognition) are nearly impossible to translate into manual code but are relatively simple for ML systems to figure out.

5. The Modern Revolution

- **Hardware Evolution**: These concepts have existed since the 1950s, but they are only viable now because high-power hardware (RAM, CPUs, and **GPUs**) has become affordable.
- **The ML Revolution**: We are entering an era of innovation that is moving faster than the Industrial or Digital Revolutions, with the potential to influence every major industry.

Let's go through the key concepts and definitions discussed in the video.



Artificial Intelligence (AI)

A broad term that refers to human intelligence exhibited by machines. It is the science of making things smart.

Narrow AI

A type of artificial intelligence that can perform one (or a few) defined tasks as well or better than a human expert. Common use cases could be classifying text in user contact forms and automatically routing those messages to the respective sub-teams in a company, or accurately detecting tumors in medical images.



Machine Learning (ML)

Machine learning is a subfield of artificial intelligence. It is the implementation of the actual program that learns from prior experience to find patterns in a dataset. This paradigm contrasts traditional programming, where you hand-code rules to process data and arrive at answers.

Consider the use case where you want to automate spam detection. A traditional programming approach would be to formulate explicit rules for the program. In contrast, a machine learning program would learn by looking at millions of comments that have already been labeled **spam** or **not spam**. The system would then attempt to categorize new (and previously unseen) comments as **spam** or **not spam** based on what it has learned by finding what words are statistically significant.

Traditional Programming

Write a computer program with **explicit rules** to follow

```
if email contains V!agra  
    then mark is-spam;  
  
if email contains ...  
  
if email contains ...
```

Machine Learning Programs

Write a computer program to **learn from examples**

```
try to classify some emails;  
change self to reduce errors;  
repeat;
```

Figure 2.1.1: Traditional Programming vs. Machine Learning Programs

Additionally, you can reuse the model by feeding different training data to address a new use case! It is the reusability that makes machine learning so powerful since you do not need to change the code each time you change the application.

Advantages of using Machine Learning

In contrast to traditional programming, where programmers think logically and mathematically, machine learning involves training systems that make observations about complex data. The systems use statistics to update their understanding of the hidden patterns in the data. This shift in thinking provides machine learning several advantages.

- **Decrease in programming time:** Consider the problem of image recognition. While a traditional programmer might spend weeks coding the different explicit rules needed to recognize an image, they might end up with an algorithm that works only in certain situations while failing in others. You can solve the same problem in a fraction of the time required by using machine learning, where many sample images of the object are fed to a model to get reliable answers.
- **Customization:** You can customize machine learning models for diverse groups of users. A machine learning model that recognizes one type of object can easily be retrained with sample images of another kind, allowing the programmer to develop solutions at super speed!
- **Complex Problems:** Consider solving problems that require face recognition or text toxicity detection. These are complex problems to code since they require defining hundreds or

thousands of rules, yet these problems are relatively simple for machine learning models to figure out.



Key Concept 3

Deep Learning

Deep Learning is a machine learning algorithm that involves Deep Neural Networks (DNN) code structures arranged in layers that mimic how scientists believe the human brain works by learning patterns of patterns as you go further down the layers.

For instance, a deep neural network may learn to recognize the content of images as shown in figure 2.1.2.

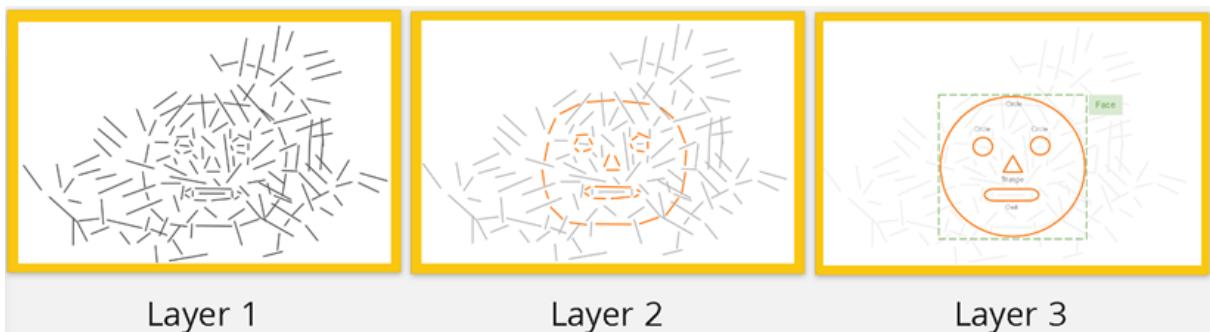


Figure 2.1.2: Example of a Deep Neural Network extracting key features in an image. Deeper layers build on knowledge found in the previous layers to find more complex connections between the data.

Layer 1: The first layer may detect only lines and edges.

Layer 2: The second layer may combine the data from the first layer to detect shapes at a deeper level.

Layer 3: The third layer might combine shapes to detect objects by understanding their relative positions to each other!

Relationship between AI, ML, and DL

Artificial intelligence, the process of enabling human intelligence in systems, has a subset called machine learning which is the actual implementation of a program that can learn from data. Deep learning is a class of algorithms that can drive a machine learning program.

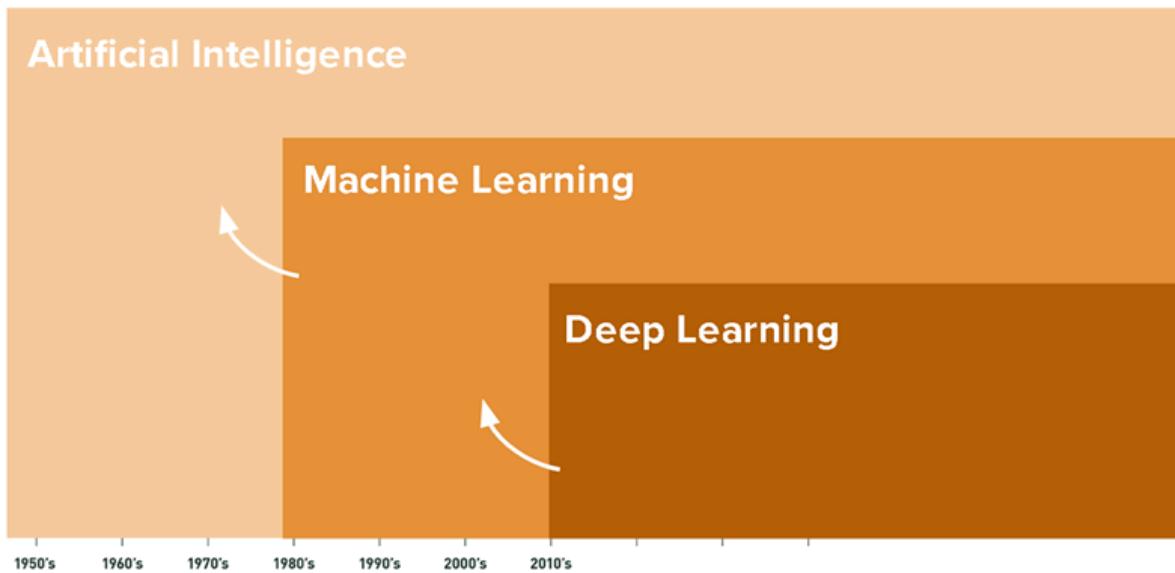


Figure 2.1.3: Relationship between AI, ML, and DL

Machine Learning Applications

Los enunciados de nivel 2 podrían ser creados por proveedores de cursos en el futuro.



Application

Machine learning applications extend beyond spam detection to other fields. Here's a list of some common use cases for your quick reference.

Use Case	Example/Application
Computer Vision	Object Detection: Identifying the location of objects in an image Image Recognition: Detect what exists in an image.

Numerical Prediction	Regression: Predicting a numerical value from other numerical input values, for instance, predicting housing prices based on square footage.
Natural Language	Text Toxicity/Sentiment: Detecting spam in comments and toxicity in posts, understanding sentiments in posts, summarizing texts, answering questions from complex texts, and translating between languages.
Audio	Speech Recognition: Speech recognition, digital assistants, and Web Speech APIs.
Generative	Style Transfer/Creative: Generating realistic images of human faces or transforming human voices into musical instruments

Demystifying Machine Learning

You learned that a machine learning model is essentially an algorithm that transforms numerical inputs into numerical outputs used in applications. For instance, a machine learning model that recognizes types of flowers may take in numerical representations of the flower's color and the stem length as inputs and produce a score of confidence that classifies the flower as a particular type. You can then decide what to do with this output in your application.

You need to consider that all machine learning models are initially **untrained** and there are several ways to train them. Based on the training method, you can classify them into the following forms of machine learning:

- Supervised learning
- Unsupervised learning
- Reinforcement learning



Supervised Learning

Machine learning models are trained in supervised learning using **labeled** examples or data. The model is provided with the **ground truth** for each example from which it will learn. In other words, each example consists of the data and the label that tells the model what the correct output for the data is. Supervised learning aims to learn from the examples and generalizes its learning to data it hasn't seen before. A model performs well only when you provide **diverse and well-labeled data** for training.

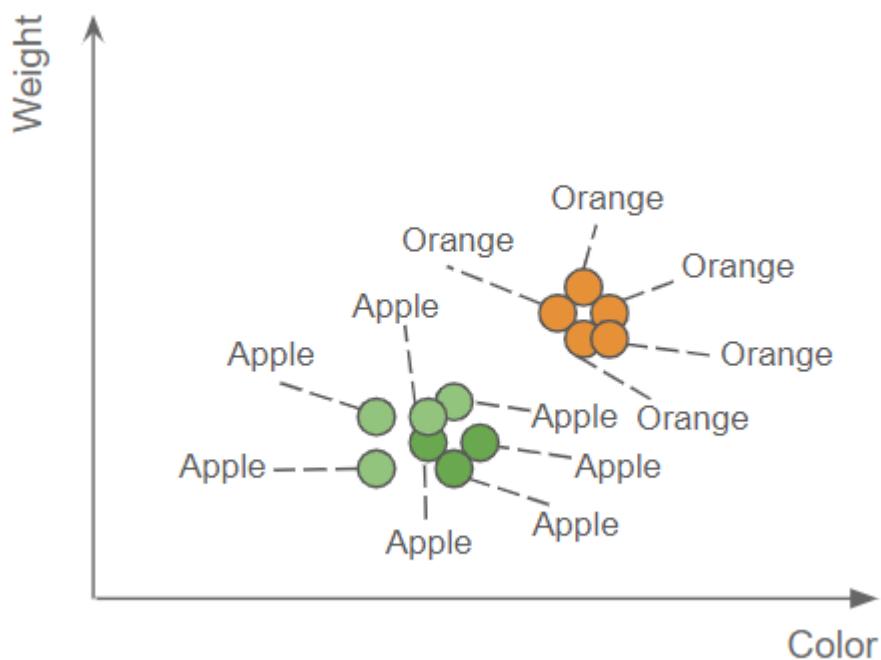


Figure 2.2.1: Scatter Plot of Fruit Data Points

Let's understand how you can use **Supervised Learning** to train a model to categorize fruits into apples or oranges.

Step 1: Provide numerical input data representing fruits and their labels (See Figure 2.2.1: Each data point has the weight and color of fruit along with the ground truth or the label of the fruit).

Step 2: Train the model using a machine learning algorithm to learn a line that can separate the two types of fruit.

Step 3: Have the model classify a fruit that it has not seen before as an apple or an orange by predicting what side of the line it will be.

How do you think the model would perform?

The model will initially, and frequently be wrong but will use the labels provided during training to correct itself and arrive at the right answer. The model will also perform only as well as the data used to train it. For instance, if you train it using only green apples as examples, the model will only recognize **green apples as apples** and may struggle with red or yellow ones. The more diverse and labeled your data, the better your model will perform!



Unsupervised Learning

In unsupervised learning, the example data provided to the model is **unlabeled**. You may roughly know the number of classes you expect to discover from the data. The model attempts to find related data and clusters them together. Unsupervised learning is especially useful when dealing with datasets that are difficult to label. An everyday use case for unsupervised learning is recommendation engines commonly

found in retail websites, where the system clusters people with similar shopping habits to recommend new products.

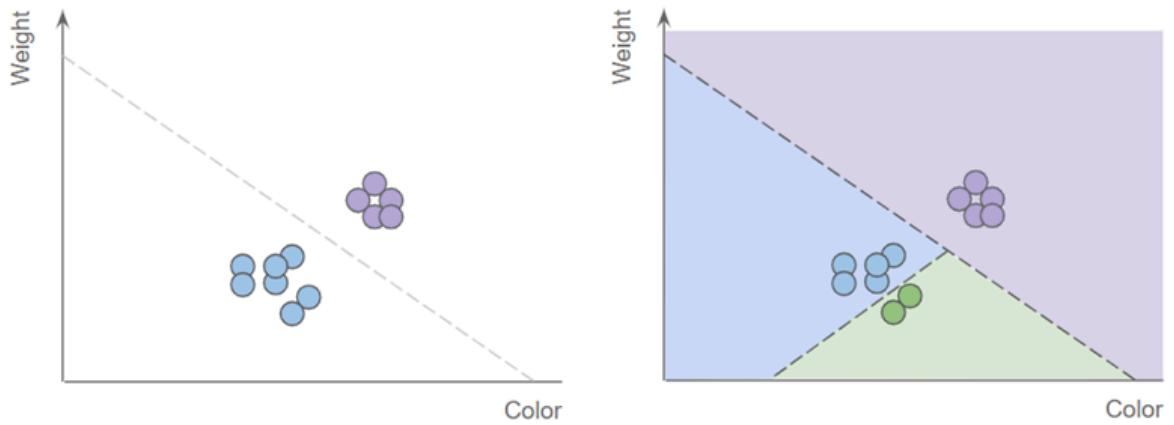


Figure 2.2.2: Clustering in Unsupervised Learning



Reinforcement Learning

Reinforcement learning is a relatively new area of machine learning where the model takes actions to achieve a **goal** while maximizing a **reward** that is defined. This system relies on **trial** and **error** with accurate “trials” providing rewards. The system undergoes many iterations to find a combination of rules that achieves the best results. The applications of reinforcement learning include gaming, robotics, and scientific research.



Applications

Type of Machine Learning	Example / Application
Supervised Learning	Classifying Objects: Spam detection, Text categorization, Object recognition
Unsupervised Learning	Clustering: Product Recommendation Engines

**Reinforce
ment
Learning**

Game Development and Robotics

How to Train ML Systems

Supervised learning involves feeding labeled data to the machine learning model. Let's start by understanding the **features** or **attributes** of data while exploring two types of supervised learning problems.

1. Classification Problems

Consider a scenario where you want to classify fruits into apples or oranges based on their color and weight. The information you provide about the fruits, i.e., the color and the weight, are called **features** or **attributes** of your data.

Plotting the weight and color of apples (red and green dots) and oranges (orange dots)

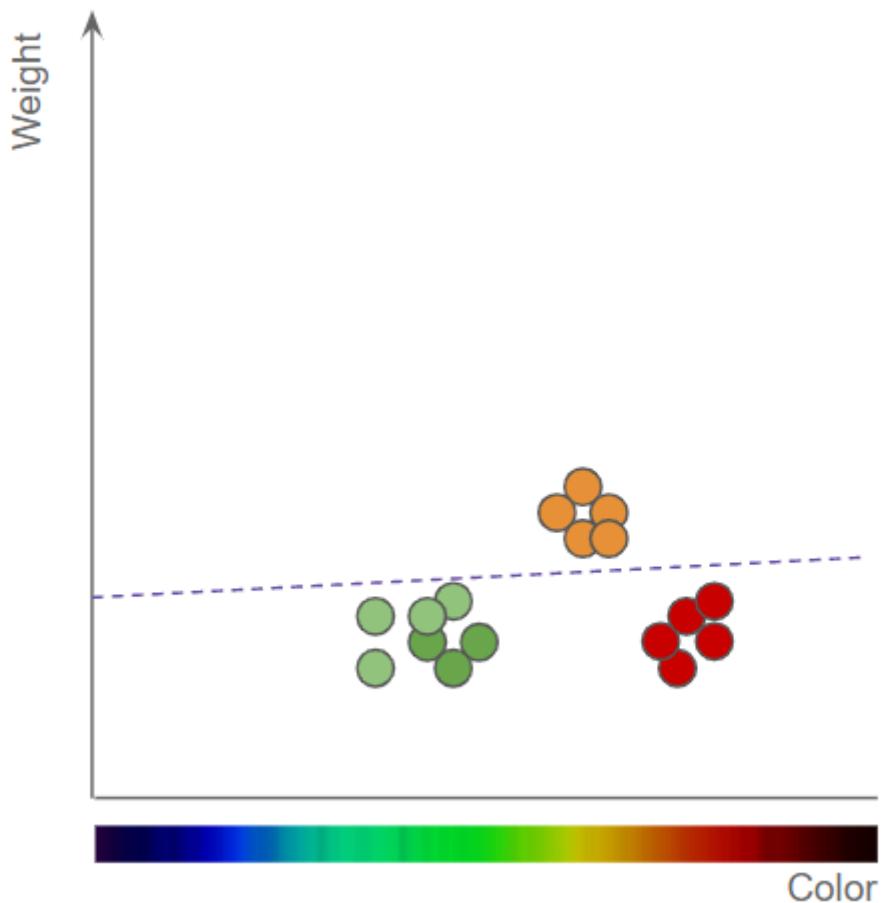


Figure 2.3.1: Scatter Chart Plotting of the Color and Weight of Fruits

When you plot the weight (y-axis) and color values (x-axis) of apples and oranges (see Figure 2.3.1), you observe that apples and oranges cluster at different weight values since oranges are typically heavier. The apples themselves cluster at different horizontal positions based on their colors. Note the dotted line that separates oranges from apples. A fruit that falls above this line is likely an orange, while a fruit below must be an apple. A machine that can define the equation of this line will be able to classify previously unseen fruits into apples or oranges.

Selecting the Most Appropriate Features:

Your input data might have multiple possible features. For instance, in addition to color and weight, the ripeness and the number of seeds could also be features. Note the ripeness and the number of seeds are features that do not help in classifying fruits as apples or oranges. Observe figure 2.3.2 and think for a moment whether you can draw a line that allows one to easily separate apples from oranges.

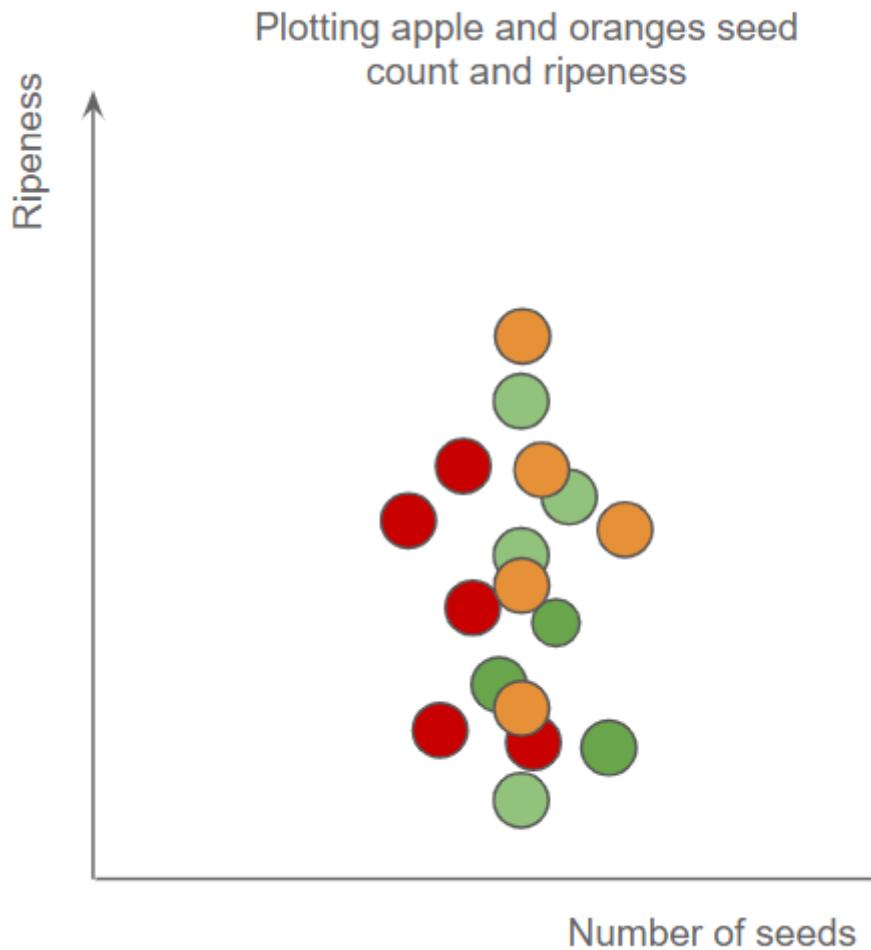


Figure 2.3.2: Scatter Chart Plotting of the Ripeness and Number of Seeds of Fruits

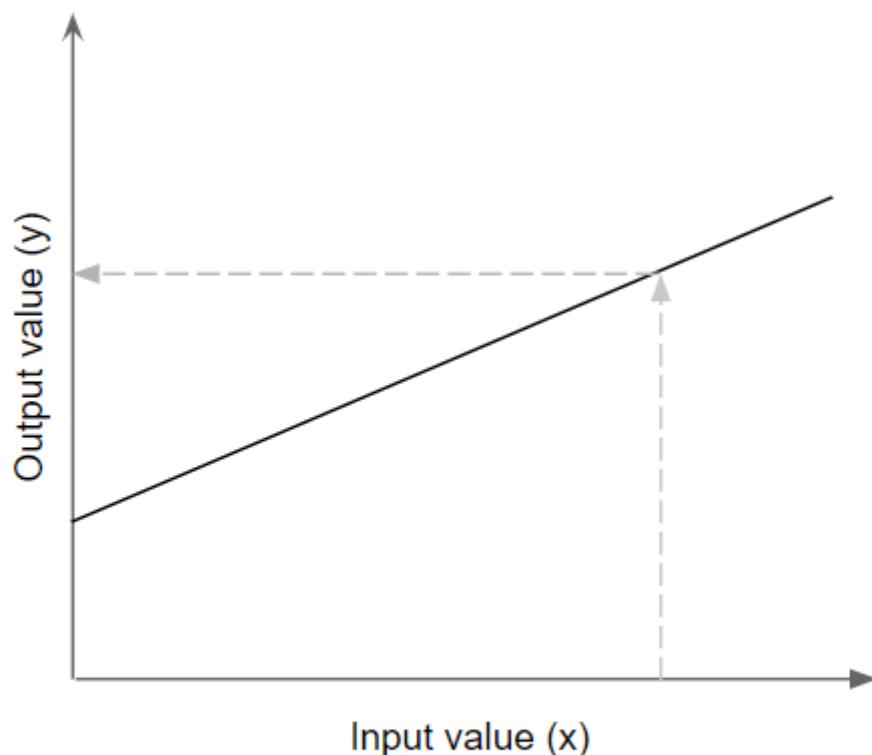
What is a Classification Problem?

The example of the apples and oranges is a supervised learning **classification** problem, where the model attempts to figure out what is the thing represented by the inputs provided. It predicts a specific class, for example, an apple or an orange from a number of known possible classes.

2. Regression Problem

Suppose you want to predict a house's price based on the area of the house. The **square footage** could be an appropriate **feature** or **attribute** of your data.

Consider a model that predicts housing prices based on the square footage. The square footage is an input feature that the model analyzes to predict the required output. When you graph the input feature against the output (see Figure 2.3.3), you see a linear relationship between the two, as indicated by the sloped line. The line is used to predict the house price on the y-axis based on the input square footage on the x-axis.



Regression problem

(What's house price given size of house?)

Figure 2.3.3: The Regression Model

Selecting the Most Appropriate Features

You need to keep in mind that not all features of data are useful. The number of bathrooms could also be an appropriate feature, but the number of a particular type of plant or the number of blades of grass in the garden could be features that do not have anything to do with the price of the house.

What is a Regression Problem?

The housing price prediction model is solving a regression problem, where you are trying to predict an output number – house price, from some other input numbers – square footage.

Features and Dimensions

- The regression model used for predicting the house price took in one input data feature, i.e., the square footage. We represented this in a graph that showed the square footage on the x-axis and the price on the y-axis. Whereas, the classification model used for classifying fruits took in two features of input data, i.e., the weight and the color of the fruit. Refer to figure 2.3.4 to understand the difference between classification and regression problems.

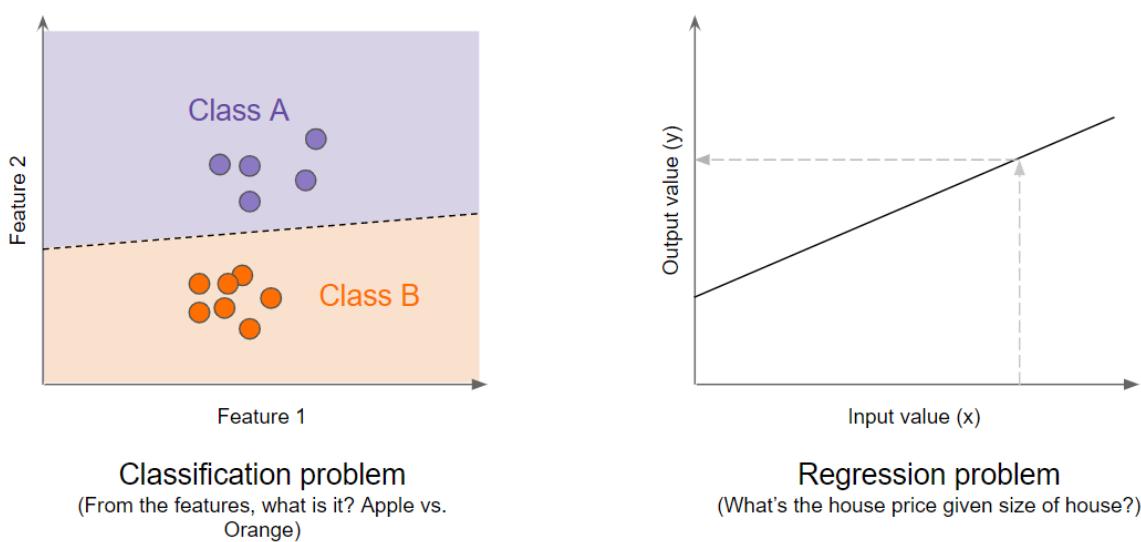


Figure 2.3.3: Classification vs. Regression Models

- Consider a regression model that uses the square footage, number of bedrooms, and the number of bathrooms as features. Here you have three features, and you will need a 3-dimensional graph to plot the features against the price of the house.
- Consider a classification model that classifies images. Each pixel in an image is stored as a number and is a distinct feature in itself. Depending on the size of the picture, there can be thousands or even millions of input features. While humans may find it impossible to visualize beyond three dimensions, computers can efficiently deal with these types of data.

Whether it is classifying fruits using a few input features or classifying images with thousands of features, it is crucial to have data representing all the situations you may encounter in the real world. Failure to do this may cause your machine learning model to fail since the real world is filled with **edge** cases where classes overlap. These edge cases have the potential to introduce data bias in your model.



Key Concepts

Data:

Data is the information provided to, processed, and analyzed by a machine learning model. Input data can be images, tabular data, numbers, text, sensor recordings, sound samples, etc.

Features/Attributes:

Features are characteristics or attributes of the input data used to train an ML system. They are the properties of the things you are trying to learn.

Regression Model:

A regression model predicts a number based on numerical inputs. The output is the equation of a line, where the line itself is used to predict an output number based on the input feature values.

Classification Model:

A classification model predicts discrete classes or labels from several possible known classes or labels. The output is a line or a plane that separates different classes.

What is TensorFlow.js



TensorFlow is an open-sourced Machine Learning library developed by Google.

TensorFlow.js

TensorFlow.js is the JavaScript specific version of TensorFlow that emerged in 2018 due to the demand for a production-quality JS library aligned with the original implementation. Currently, TensorFlow.js is being used by companies, from startups to large multinationals, and individuals from academics to hobbyists.

Advantages of TensorFlow.js

Traditionally, developers performed machine learning with languages focused on the backend. These languages, such as Python, run on servers in the cloud outside the web browser or client's machine. The emergence of TensorFlow.js, however, has allowed practitioners to deploy machine learning at the browser level on the client-side. This scenario equates to a broader reach, privacy (as no data is sent to server for classification), instant access, reduced costs, and faster development cycles.

Developers can now run TensorFlow.js in any environment where JavaScript can run as shown in figure 2.4.1.



Figure 2.4.1: Environments Supporting TensorFlow.js

Note: One of the challenges you need to consider with TensorFlow.js is the type of hardware the client will be using to execute a model. The hardware changes from user to user, which affects the speed at which any given model can run.

TensorFlow.js Application Programming Interfaces (APIs)

Developers work with two TensorFlow.js APIs. These can be visualized as sitting below the ready-to-use pre-made models already available which are built upon these APIs (Refer to Figure 2.4.2).

- The Layers API is a high-level API that allows developers to make custom models without dealing with the mathematics involved in machine learning. This API is analogous to the Keras API made available in the Python form of TensorFlow.

- The Ops or Core API is a lower-level API that allows developers to work with mathematics such as linear algebra.

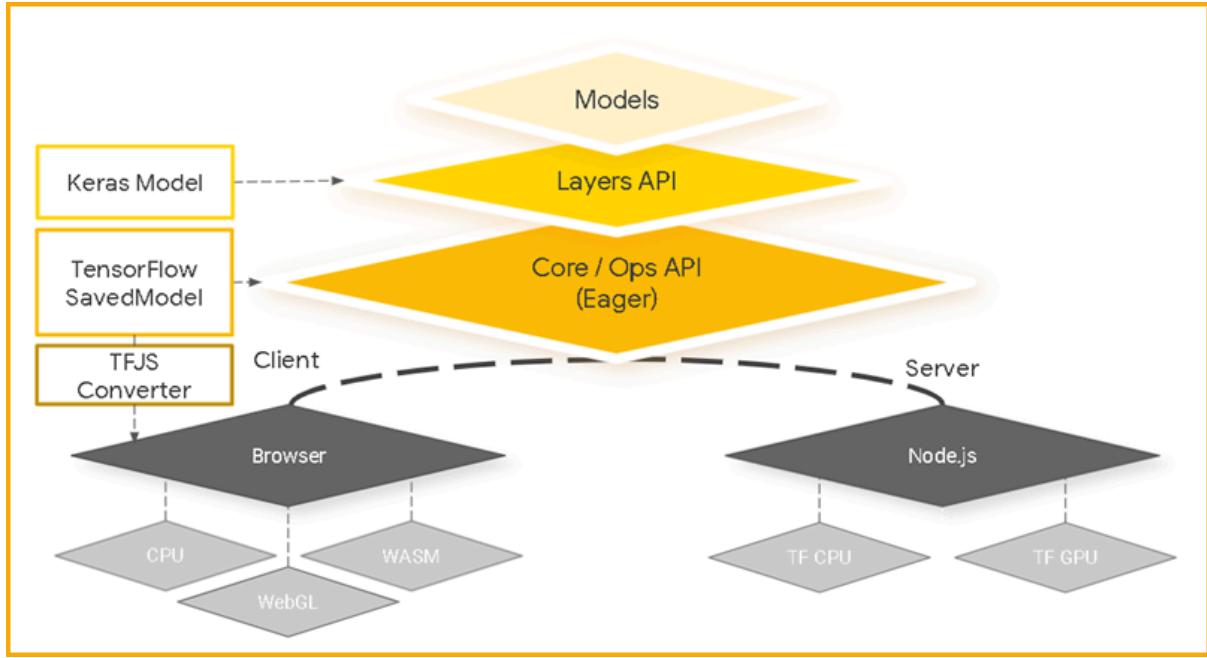


Figure 2.4.2: TensorFlow.js APIs

TensorFlow.js and Client-side Environments

TensorFlow.js can execute on several backends or hardware at the browser level as shown in figure 2.4.3.

Backends - hardware execution

Highly optimized JS runtime for CPU/GPU

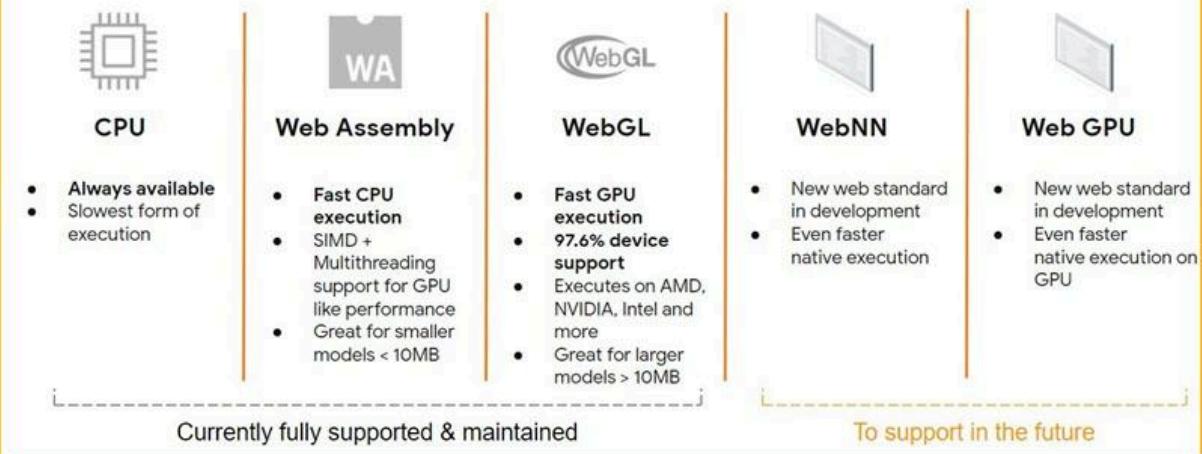


Figure 2.4.3: Backends - Hardware Execution

Node.js is a version of JavaScript that runs server-side. Node.js supports the same TensorFlow CPU and GPU bindings as Python. In fact both Python and Node.js server side versions of TensorFlow simply wrap around a C++ core that TensorFlow itself is written in. Using TensorFlow.js on the server-side with Node.js will enable developers to leverage the CUDA and AVX acceleration to perform comparably or faster than Python. Node.js also supports ingestion of Keras and TensorFlow saved models (that were exported from the Python version of TensorFlow). This feature allows Python developers to integrate with web teams that use Node.js with ease and for web teams to take advanced models and use them without any need for conversion. Developers can also convert Python models to the TensorFlow.js form for execution in the web browser using the TensorFlow.js command line converter.

Client-side superpowers of TensorFlow.js

- **Greater Privacy**

Client-side models maintain user data privacy since no data is sent to a third-party server for classification.

- **Lower Latency**

Any application with stringent latency requirements needs to run on the client-side, and TensorFlow.js provides this superpower to developers. JS has direct access to the device sensors such as mic, cam, accelerometer, and more, and since there is no round trip time to the server, TensorFlow.js allows models to run much faster than server-side models that have latency to speak to a server that could be thousands of miles away.

- **Lower Cost**

Since there is no data sent to a server, there is less bandwidth cost and hardware server costs with TensorFlow.js. Developers just need to pay for hosting website assets and model files instead of handling the cost of running a dedicated server with expensive Graphics Cards, RAM, and Processors.

- **Higher Interactivity**

Web technology has evolved to handle rich formats such as Web XR/WebGL, etc. It supports mature graphics and data visualization libraries and significantly decreases the time spent on coding while enhancing interactivity.

- **Greater Reach and Scale**

TensorFlow.js runs on the browser and has the natural advantage of ‘Zero Install.’ Developers do not need to go through multiple installation steps that may be error-prone and time-consuming. Researchers can also get more eyes on their cutting-edge research and see it used in novel ways across

industries with more people being able to access it than ever before via the web.

Advantages of Server-side TensorFlow.js with Node.js

- Developers can use TensorFlow saved models without conversion. This feature, in turn, enables them to run larger models than those that run on the client-side or use cutting edge research that has not yet been converted.
- Currently, around 68% of developers already use JavaScript in production, and now they can deploy server-side machine learning models without requiring them to learn another language.
- The Node.js implementation of TensorFlow wraps around the same C++ bindings as the Python implementation. Developers attain the same server-side hardware acceleration with the Node.js implementation compared to the Python version. In addition, the just-in-time compiler in JavaScript provides a performance boost in Node.js, especially if a model requires pre or post-processing of data that can lead to a 2x speed improvement over Python code.

3 Ways to Use Machine Learning:

TensorFlow.js allows developers to create machine learning models in three distinct ways. Developers may use pre-made models, retrain existing models with new data, or roll out their custom models. Let's review these models in the Key Concepts section.

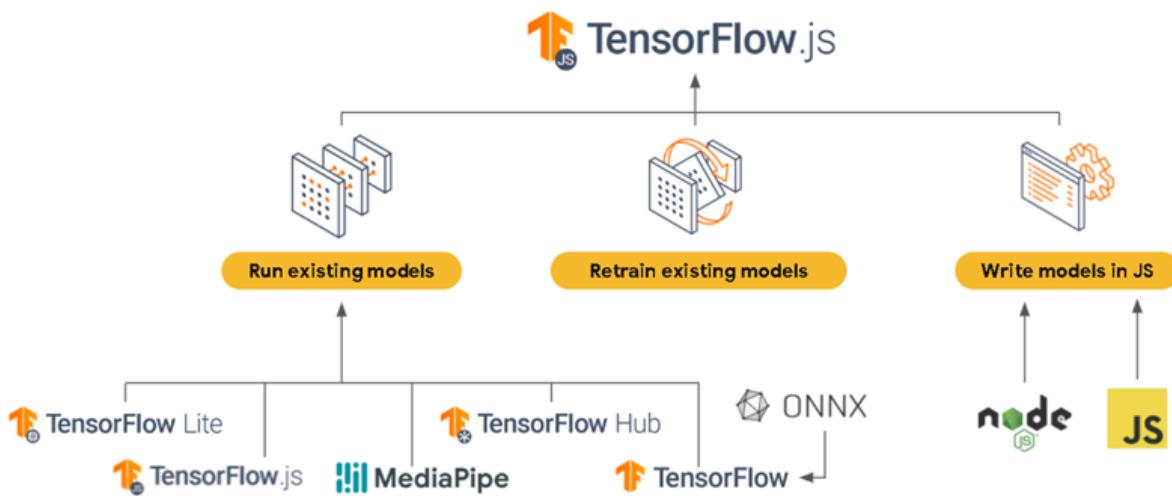


Figure: 2.5.1: Different Ways of Using Machine Learning in TensorFlow.js

Key Concepts

Pre-Made Models

- Pre-made models have already been trained on a scenario and developers can reuse them for similar use cases.
- These models include state-of-the-art models that have been trained on vast amounts of data and are robust. A few models also have Model Cards that

show how a model performs, the data used to train it, and known biases that developers need to know.

- Using pre-made models saves developers a massive amount of time and money since others have gathered the data, and development costs are minimal by comparison.

Pre-Trained Models

Continually optimizing or expanding our collection



Vision

- + Image Classification
- + **Object Detection**



Human Body

- + Body Segmentation
- + Pose Estimation
- + Face Landmark Detection
- + Hand Pose Estimation



Text

- + Text Toxicity
- + Sentence Encoding
- + BERT Q&A
- + Conversation Intent Detection



Sound

- + Speech Command Recognition



Other

- + KNN Classifier

tensorflow.org/js/models

Transfer Learning

- Transfer learning involves retraining pre-made models with new data to extend their use cases to similar domains. For instance, an image recognition model trained to recognize a set of images such as cats can be retrained with additional data to identify

new images such as dogs, for example. Just like you saw in Teachable Machine.

Custom Models

- TensorFlow.js APIs can be used to roll out custom models from a blank canvas.
- Custom models are needed when there are no existing models for a use case or when current models are not fast enough for a particular use case.

Command Line Conversion

- TensorFlow.js supports the execution of other forms of TensorFlow models through the command line converter. This feature is helpful for developers who want to use research that has been done on other platforms.



Did You Know?

MediaPipe: MediaPipe is a C++ framework for building applied ML pipelines that some researchers choose to use.

TensorFlow Hub: TensorFlow Hub is a repository of trained machine learning models that can be reused.

TensorFlow Lite: TensorFlow Lite is Google's machine learning framework that focuses on mobile native and IoT devices. While TensorFlow.js can also run on these platforms, TFLite is highly optimized for these two platforms specifically.

ONNX: The Open Neural Network Exchange (ONNX) is an open-source artificial intelligence ecosystem of technology companies and research organizations. ONNX establishes open standards for representing machine learning algorithms in a way agnostic to the library it was written in.

Benchmarking Model

Inference Speed

The inference speed is the time required by the model to provide an output prediction once it has received new data as input. This is measured in milliseconds (ms) or frames per second (FPS).

Benchmarking Model Inference Speed

Use the following code snippet to calculate a model's inference speed.

```
async function calculateInferenceSpeed() {  
  
    // Record timestamp before model execution  
  
    const timeStart = performance.now();  
  
    // Execute model  
  
    let results = await useSomeModel();  
  
    // Record timestamp after model execution  
  
    const timeEnd = performance.now();  
  
    // Calculate time taken for model execution  
  
    const timeTaken = timeEnd - timeStart;  
  
    console.log(`Time taken ${timeTaken} ms. `);  
  
    // Convert ms to FPS  
  
    console.log(`Frames per sec: ${1000 / timeTaken}`);  
  
}
```

File Size

A model's raw file size is measured in megabytes, and this parameter is one of the factors that determine how fast your web page loads. Web pages load faster with smaller models. Check the model's documentation to see if the file size has been recorded. Alternatively, you may choose to benchmark the model's file size yourself using Chrome Developer Tools.

Benchmarking: Model File Size

Steps

1. To open the 'Developer Console':
 1. Press **F12** while on the required page or press **Shift + Ctrl + I** (Chrome on Windows/Linux) or **Shift + Ctrl + J** (Firefox on Windows) or **Command + Option + J** (Firefox on OS X). Alternatively **right-click** the mouse on the web page and select Inspect, or open the **Chrome Menu** (upper right-hand corner of the browser tab) and select **More Tools > Developer Tools**.
 2. Select the **Network Tab** in the Developer Console (Chrome), the Web Console from the Web Developer submenu in the Firefox Menu, or the Tools menu if you are on the menu bar or are using Mac OS X.
1. Select the **Disable Cache** option.
2. Refresh the page manually or press **Ctrl + F5** or **Ctrl + R**.

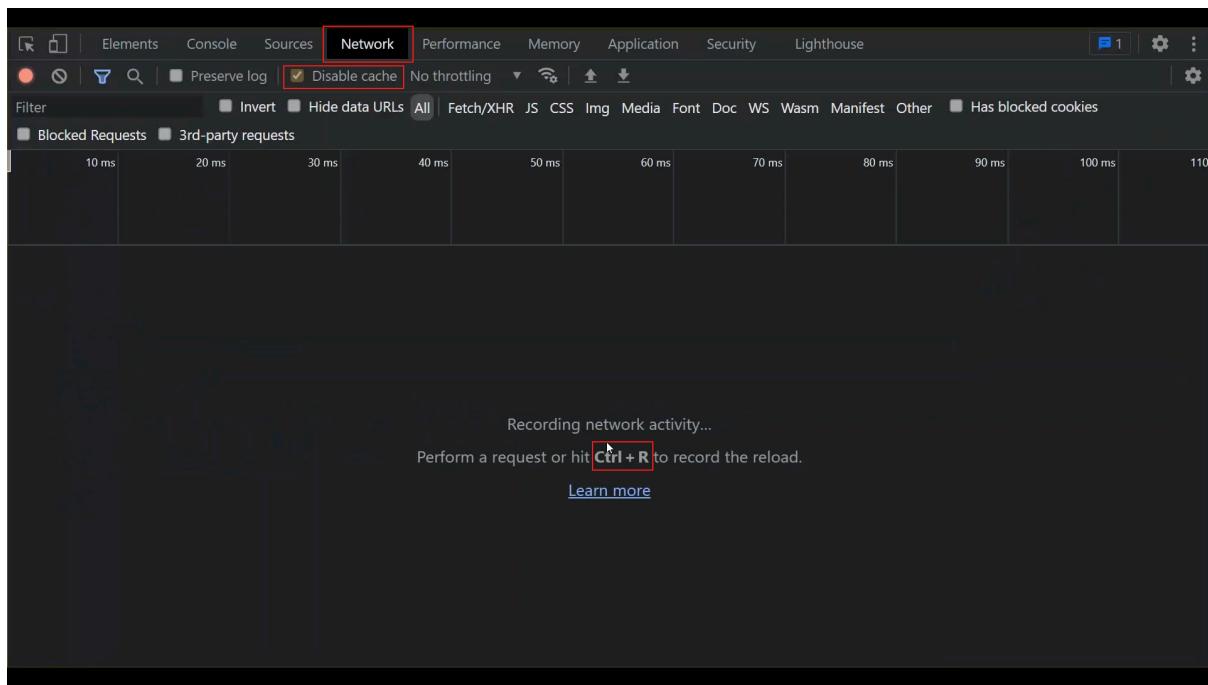


Figure: 3.2.1: The Developer Console - Network Tab

4. Once the web page has reloaded, filter the results for **.json** and **.bin** files. TensorFlow models are saved as **.json** files and associated binary files with the **.bin** extension.
5. Add up these file sizes to get the total model file size.

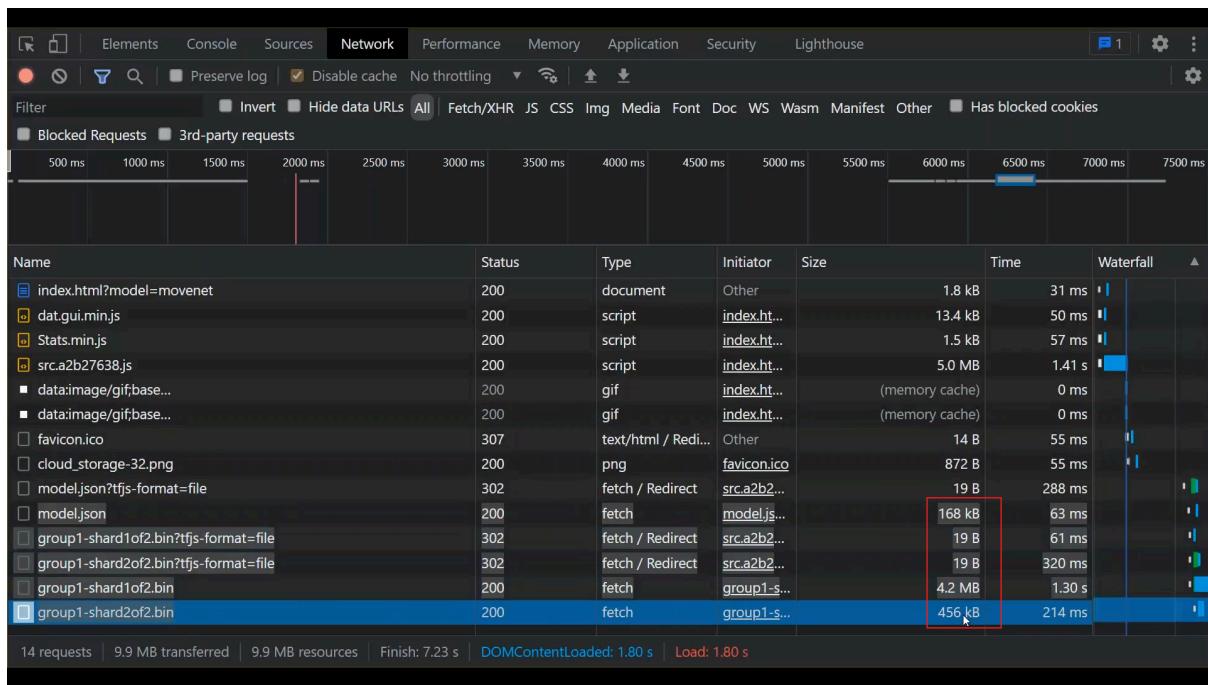


Figure 3.2.2: Calculating Model File Size from .json and .bin files - here the total is about 4.8 MB in size.

Model RAM Usage

Modern websites are designed responsively, keeping in mind the user's expected working environment. Understanding a model's runtime memory usage or the amount of RAM it needs to run on a machine is essential when choosing the model to use.

Benchmarking Model RAM Usage

Steps

1. Navigate to the Memory Tab in the Chrome Developer Console.
2. Take a snapshot.

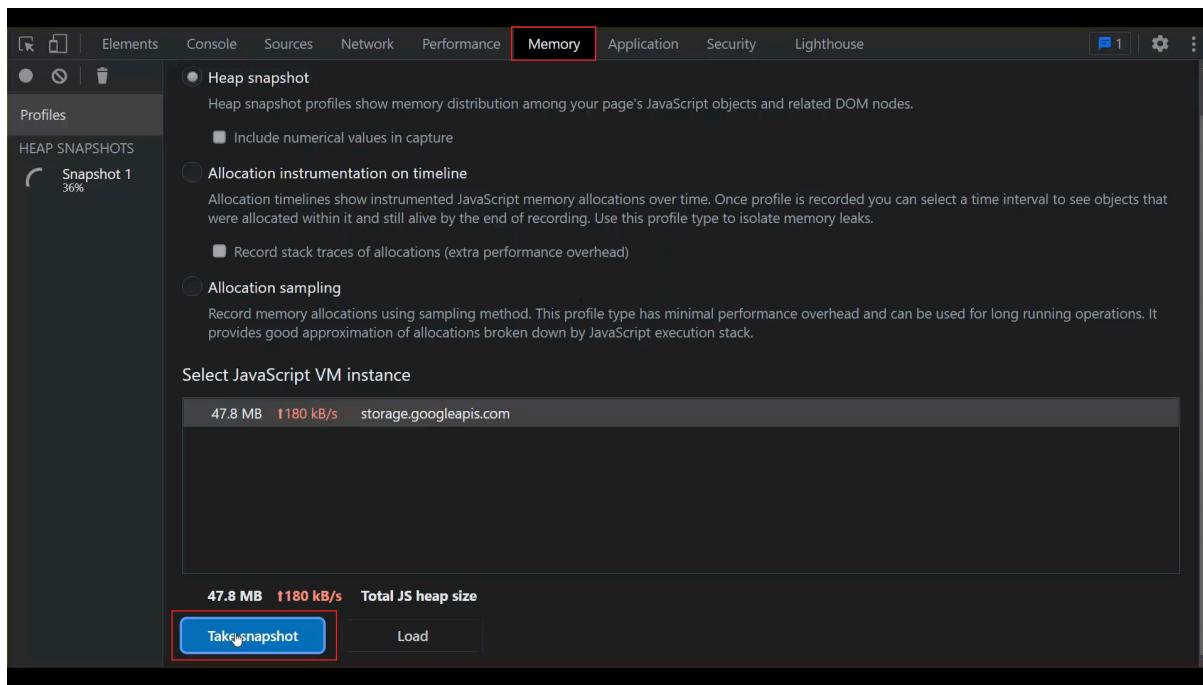


Figure 3.2.3: Developer Console - Memory Tab

3. Select **Statistics** from the Summary drop-down menu.

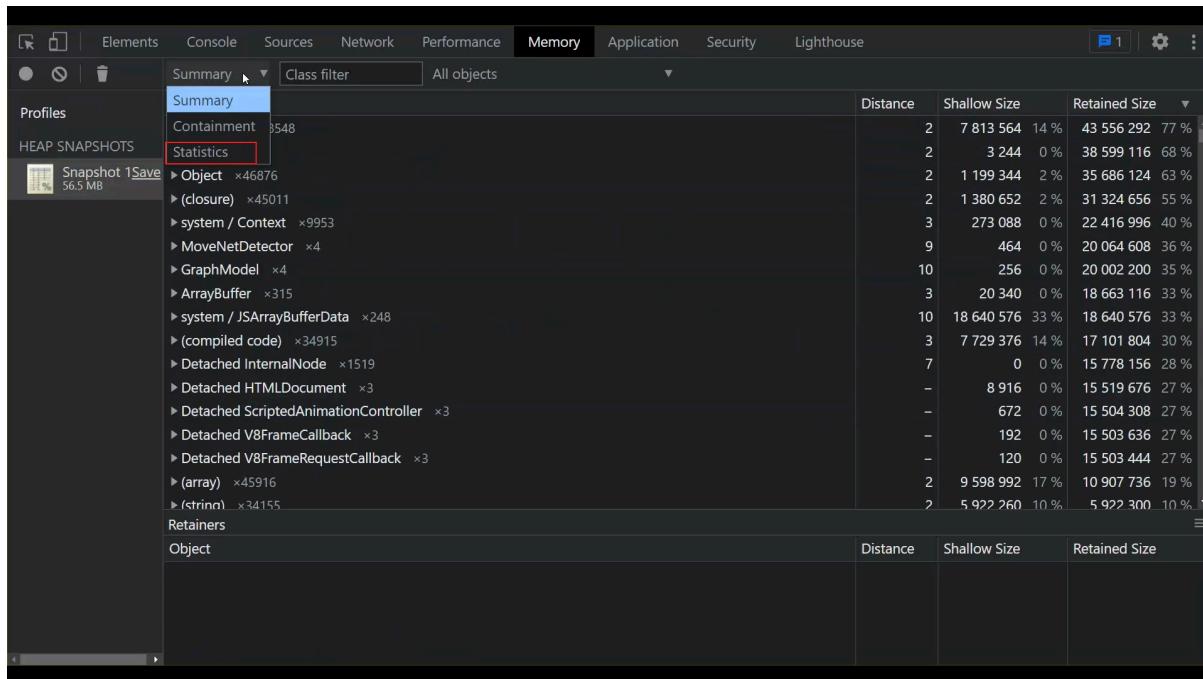


Figure 3.2.4: Developer Console - Memory Tab Snapshot

4. View the total RAM used and its distribution for different tasks.

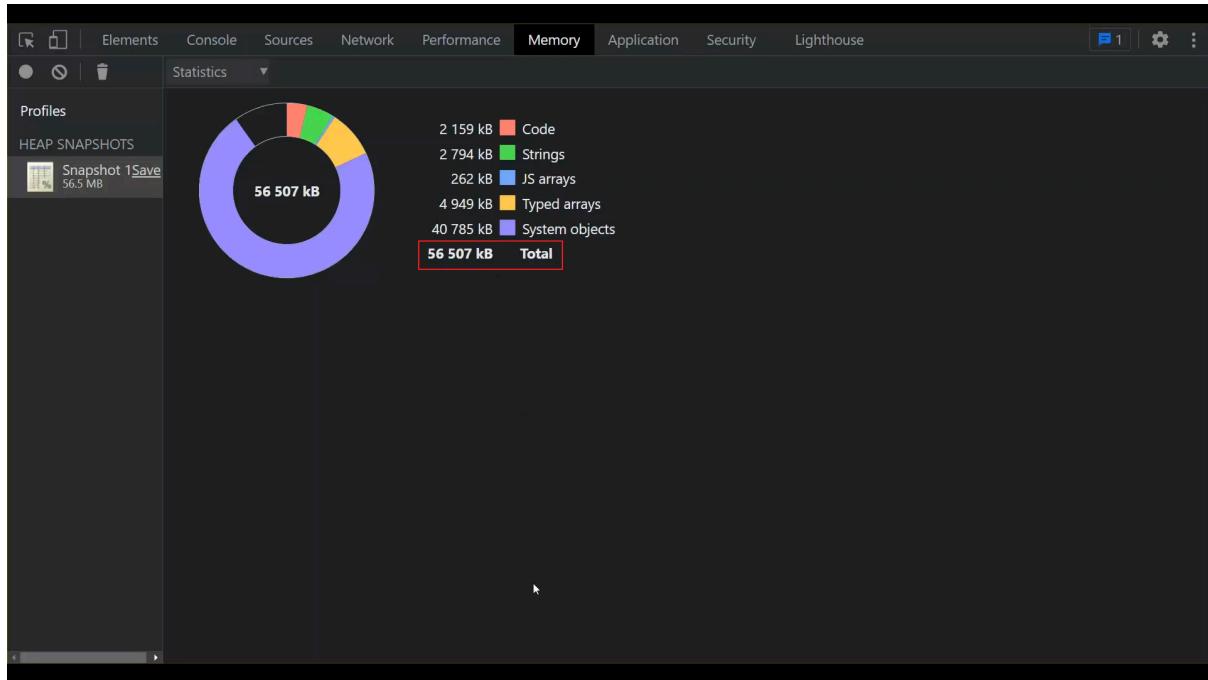


Figure 3.2.5: Developer Console - Memory Tab Statistics View. Here you see about 56.5MB of RAM is used when running this web app.

Conclusion

A thorough investigation needs to be conducted into a client's requirements to balance factors such as model speed, accuracy, size, and the client's working environment. Armed with this knowledge, you can select the best possible model to solve the problem at hand and even suggest enhancements that your client hasn't considered based on the chosen model's capabilities.

Tensors In, Tensors Out

TensorFlow is actually named after these fundamental data structures known as Tensors since they are essential to how data is stored and flows through machine learning models.

Tensors

Tensors are the primary data structures in TensorFlow programs. They are similar in structure to arrays, but can have multiple dimensions. Unlike arrays, tensors do not have mixed data types. Machine Learning (ML) models take tensors as inputs, manipulate the tensors, and provide tensors as outputs.

A tensor is an object of the tensor class, which is a class that has many useful functions. These functions assist in transformations and mathematical functions that can be performed on such tensors.

Tensors Overview

Consider an array of numbers: [1, 1, 2, 3, 5, 8]. We would define this in vanilla JavaScript as follows.

```
let values = [1, 1, 2, 3, 5, 8];
```

The TensorFlow.js API allows us to convert this simple array of numbers into a tensor as follows:

```
let tensor = tf.tensor([1, 1, 2, 3, 5, 8]);
```

We have created a ‘tensor’ object that contains numerical data from the ‘values’ array.

Tensors: Terminology

The following terms are often used when you work with tensors:

1. Data Type (DType)
2. Shape
3. Rank / Axis
4. Size

Note: ML practitioners refer to the dimensionality of tensors in terms of ‘Rank’. The number of ‘Axes’ in a tensor also corresponds to the number of dimensions or rank of the tensor.

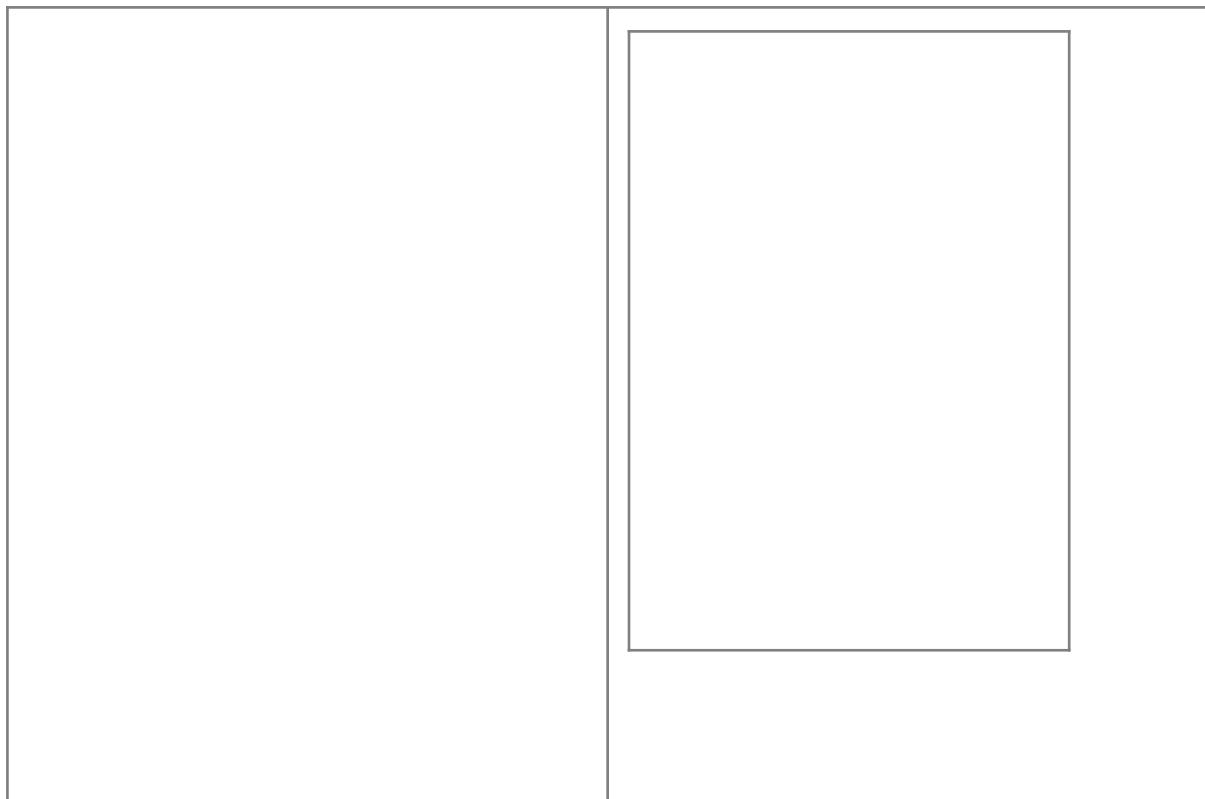
Remember that Tensors are data structures that typically contain only numerical values.

Let us explore different types of tensors and their relevant terminology.

Types of Tensors

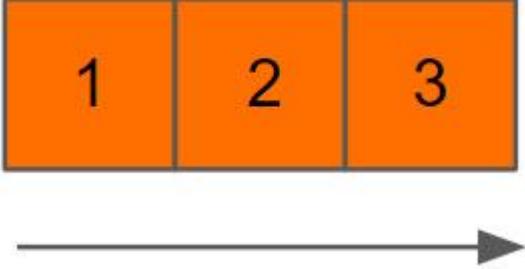
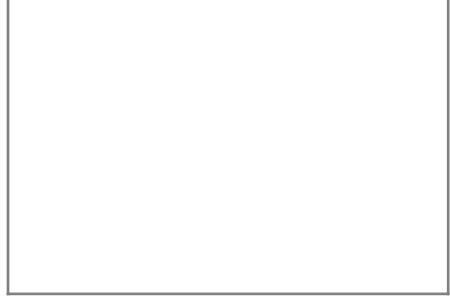
0 Dimensional Tensor	
A scalar: A single numerical value	Dimension/Rank/Axes

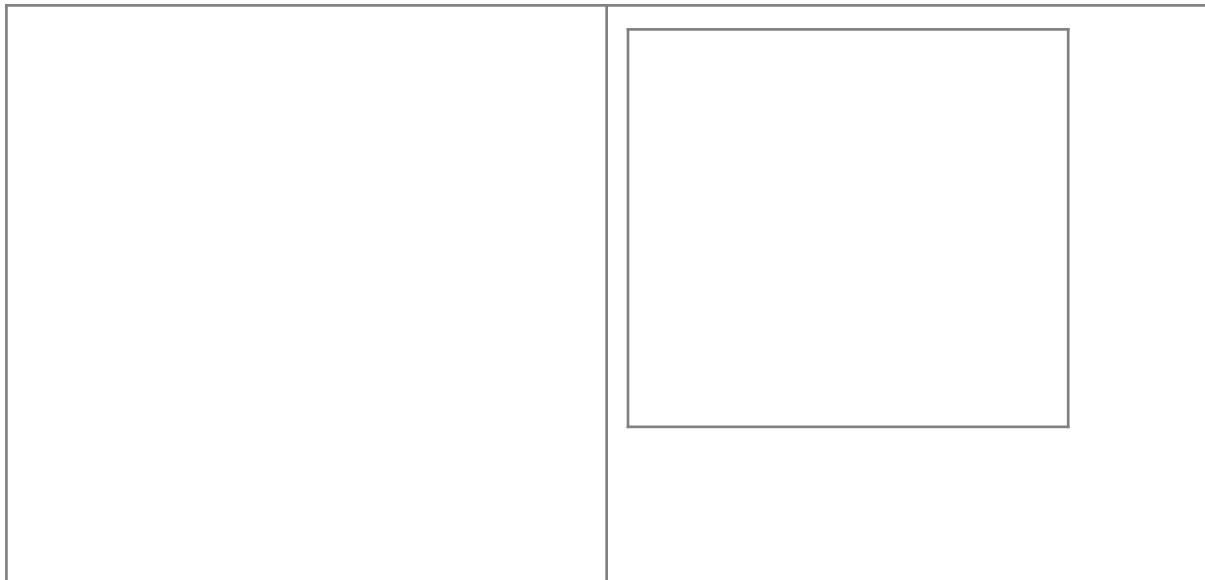
	<p>A scalar has 0 dimensions, has rank 0, and has 0 axes.</p>
Example: The number 6	<p>Visualization</p> <p>A bucket of memory containing a single number</p>  <p>Figure 3.4.1 - 0D Tensor</p>
Use Case: Any binding that stores a single numerical value.	<pre>let value = 6; let tensor = tf.scalar(6);</pre>



Note: In addition to the ‘tf.tensor()’ method, tensors may also be created by specifying the number of dimensions explicitly: tf.tensor1d(), tf.tensor2d(), tf.tensor3d(), etc which is preferred as easier to read and understand.

1-Dimensional Tensor	
A vector is a list of values.	Dimension/Rank/Axes A 1-dimensional tensor is a rank 1 tensor and has 1 axis. Note: The word ‘dimensions’ is also applied to data dimensionality too. An array of 3 values could represent a 3 dimensional value. Do not confuse the ‘ dimensionality ’

	<p>of the data with the ‘dimensionality’ of the tensor. 3 dimensional data can be stored in a 1 dimensional Tensor.</p>
Example: [1, 2, 3]	Visualization An array or list of numbers in memory is arranged in 1 dimension. 
Use Case: Tensor storing the ‘x’, ‘y’, and ‘z’ coordinates of a point.	



2 Dimensional Tensor	
An array of arrays or a matrix is a data structure with numbers arranged in rows and columns.	Dimension/Rank/Axes A 2-Dimensional Tensor is a rank 2 tensor and has 2 axes.

Example:

```
[  
    [5, 0, 7],  
  
    [8, 0, 3],  
  
    [0, 2, 9]  
]
```

Visualization

Numbers are arranged in rows and columns,
i.e., 2 dimensions.

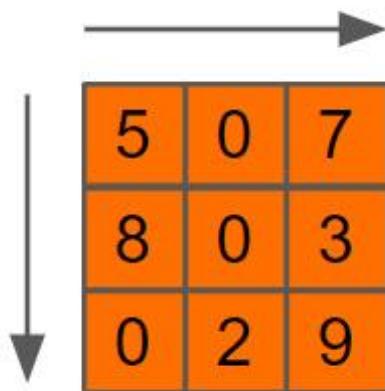
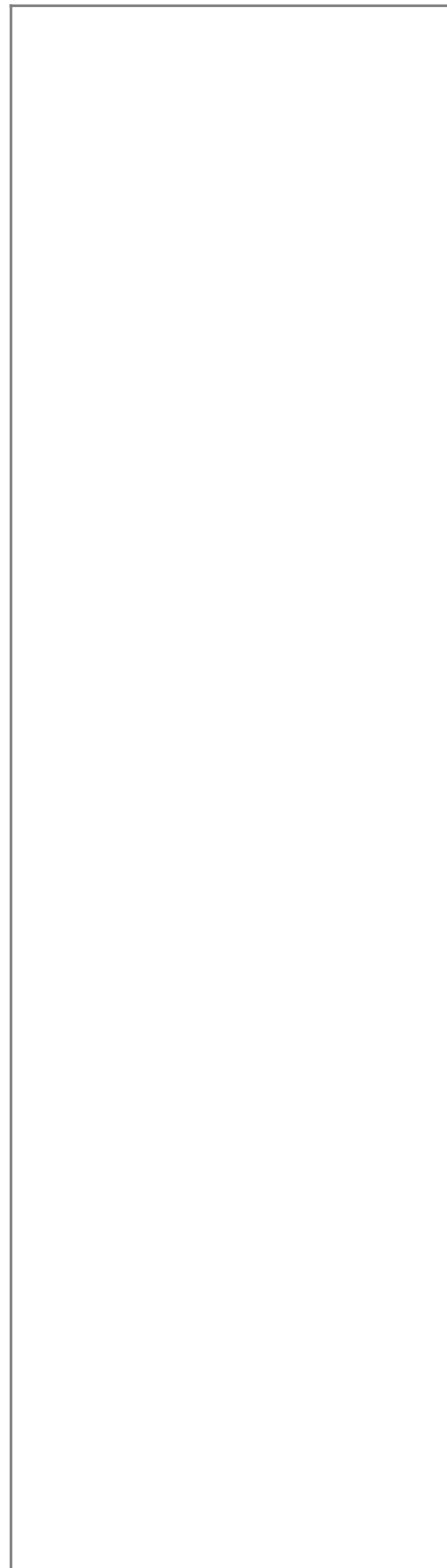
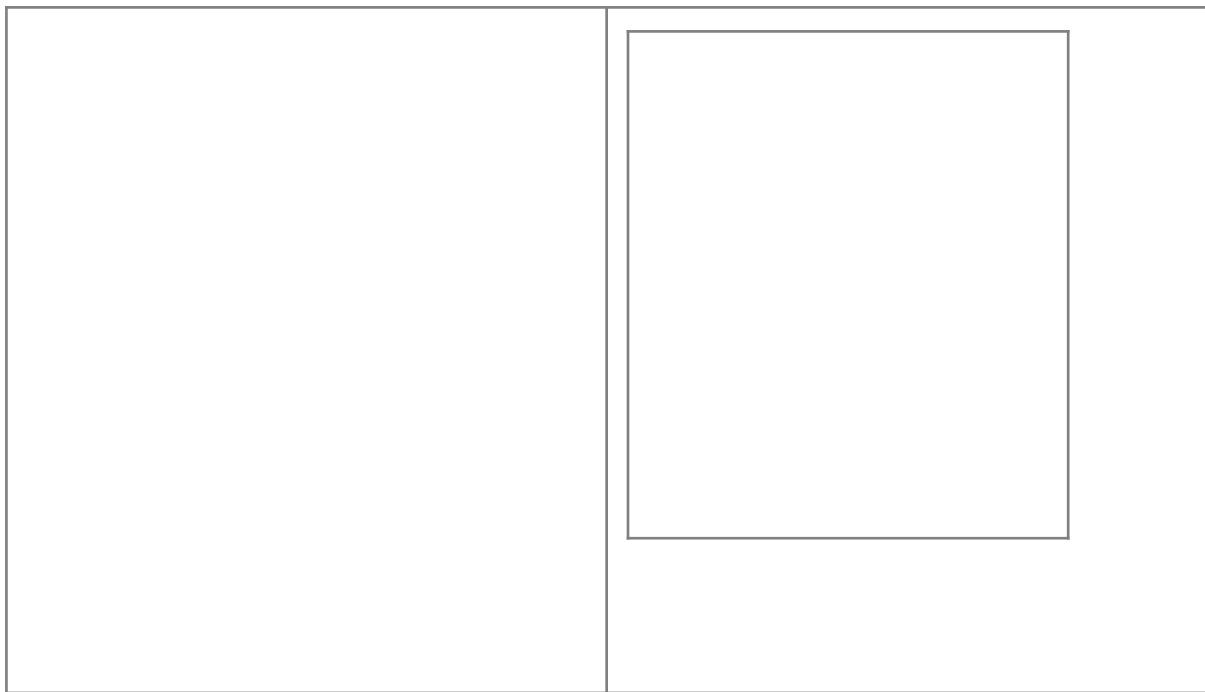


Figure 3.4.3 - 2D Tensor

Use Case: A greyscale image. Every pixel in the image would have a value from 0 to 255 representing different shades of grey the computer could draw arranged in rows and columns just like you see here.





3-Dimensional Tensor

For a 3D tensor you have an array of 2D tensors stacked on each other as shown on the right.

Dimension/Rank/Axes

A 3-Dimensional Tensor is a rank 3 tensor and has 3 axes.

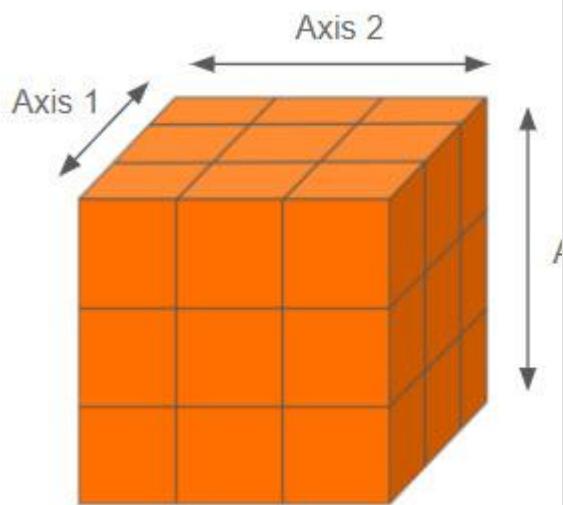


Figure 3.4.4 - 3D Tensor

Example:

```
[  
  [  
    [1, 2, 3], [4, 5, 6], [7, 8,  
     9]  
  
    ],  
    [  
      [0, 0, 0], [0, 0, 0], [0, 0,  
      0]  
    ],  
    [  
      [2, 4, 6], [8, 6, 4], [2, 4,  
      6]  
    ]  
];
```

Visualization

Numbers are arranged in stacks of layers. In this example each layer consists of 3x3 numbers that are stacked on top of each other as shown. This could represent the different R, G, B layers of a full color image.

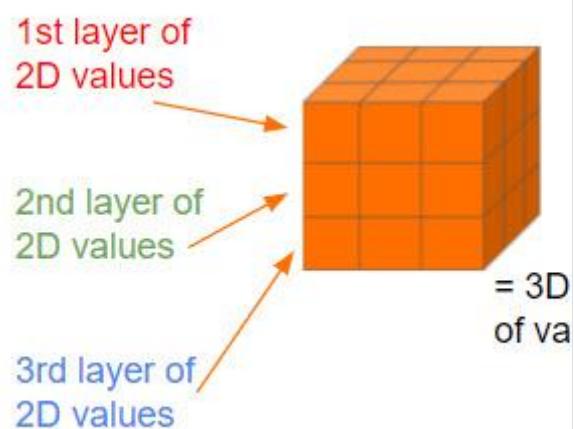
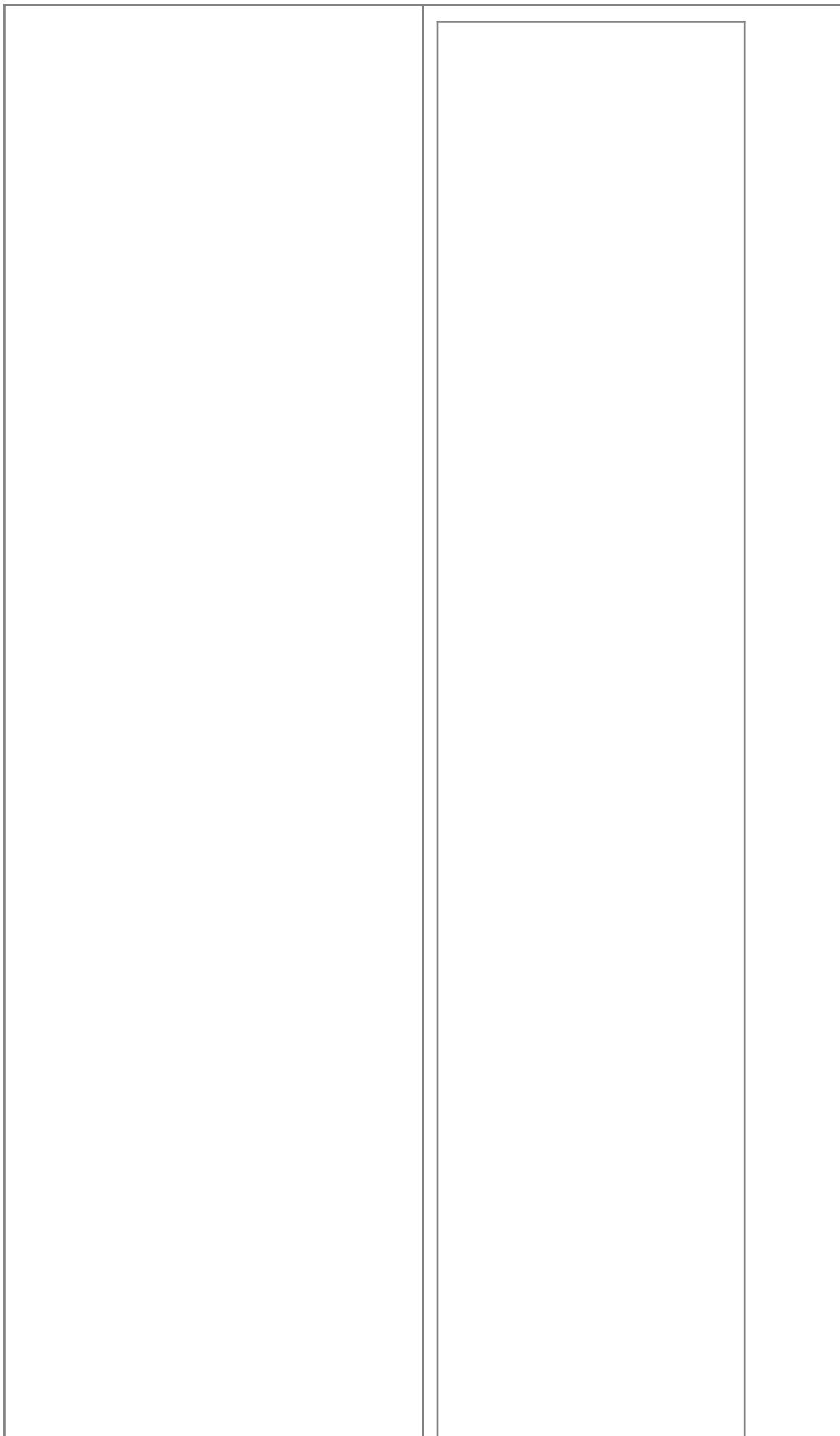
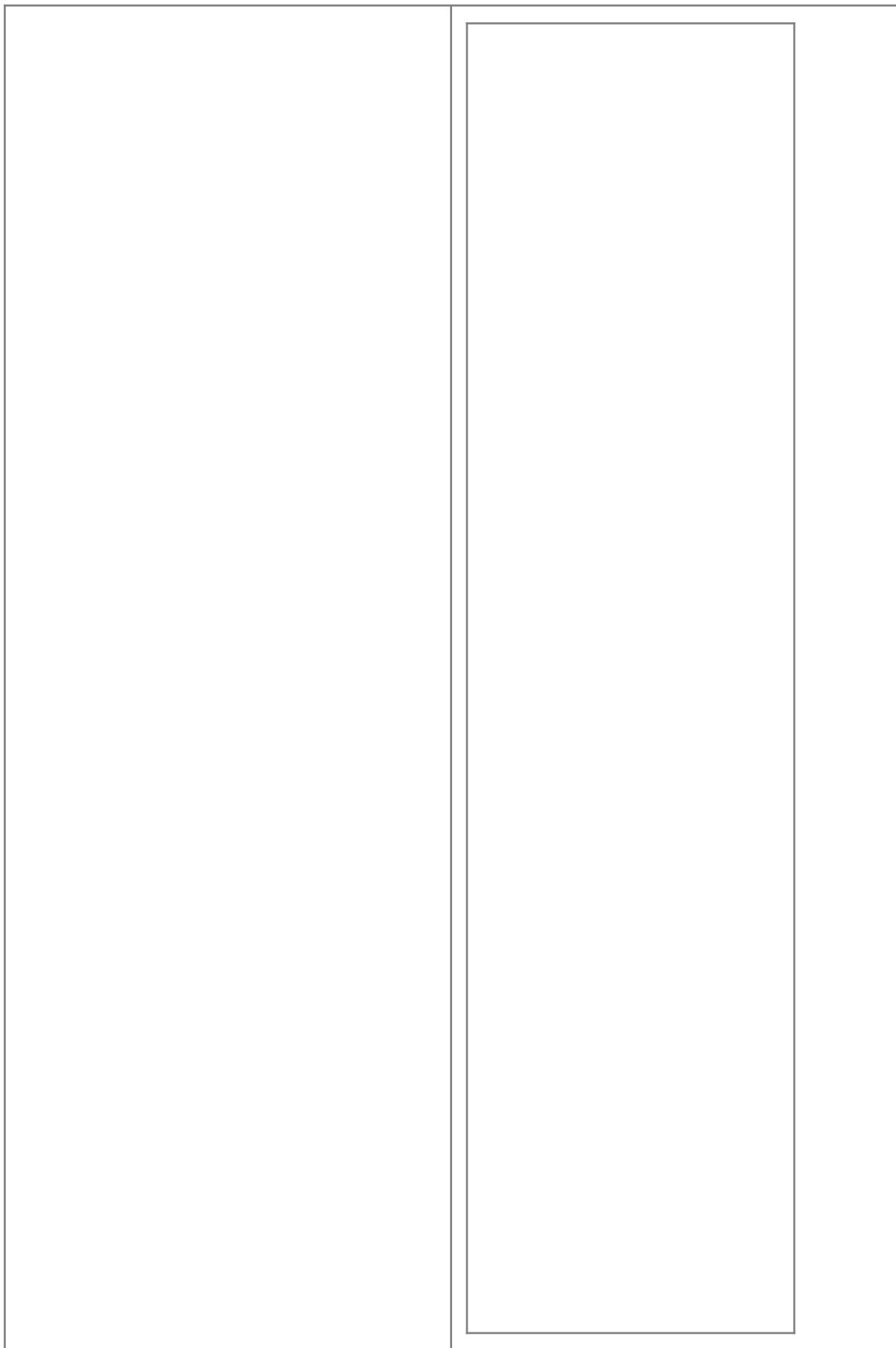
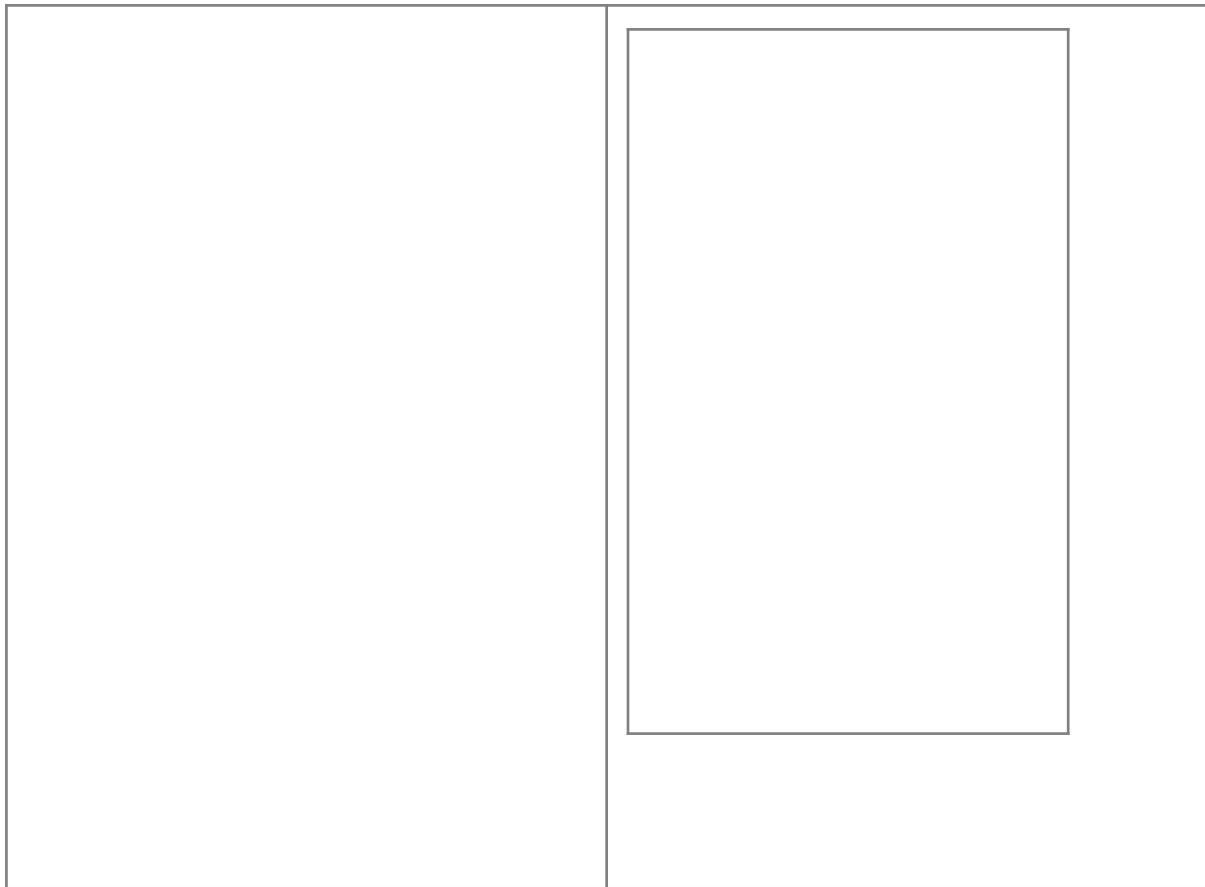


Figure 3.4.5 - 3D Tensor: Full-Color Image

Use Case: A regular full-color RGB image has three layers of pixels representing the three color channels, with each of the channel's pixels arranged in rows and columns.







4-Dimensional Tensor	
Essentially a stack of 3 dimensional tensors.	Dimension/Rank/Axes A 4-Dimensional Tensor is a rank 4 tensor and has 4 axes.

Use Case: Videos are 4-dimensional since they are a series of RGB images over time.

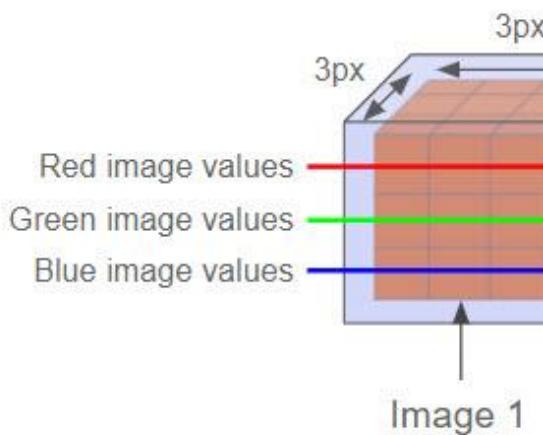
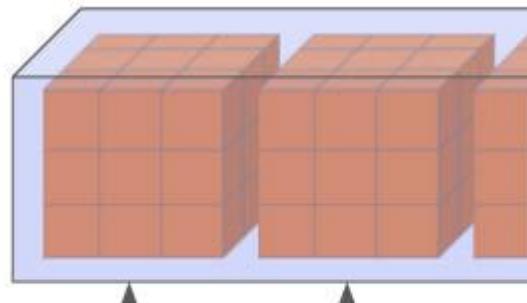


Figure 3.4.6 - 4D Tensor: Video

Visualization



An array of 3D arrays of v
== 4D array == rank 4 te

Figure 3.4.7 - 4D Tensor

5 and 6 Dimensional Tensors

5D Tensors - Use Case 1: A batch of videos

5D Tensors - Use Case 2: The digital Minecraft world is represented by 3d pixel equivalents called voxels. Each voxel has an RGB value stored in 1 dimension, then 'x', 'y', and 'z' position values stored in dimensions 2 to 4, and a 'time' value stored in the 5th dimension.

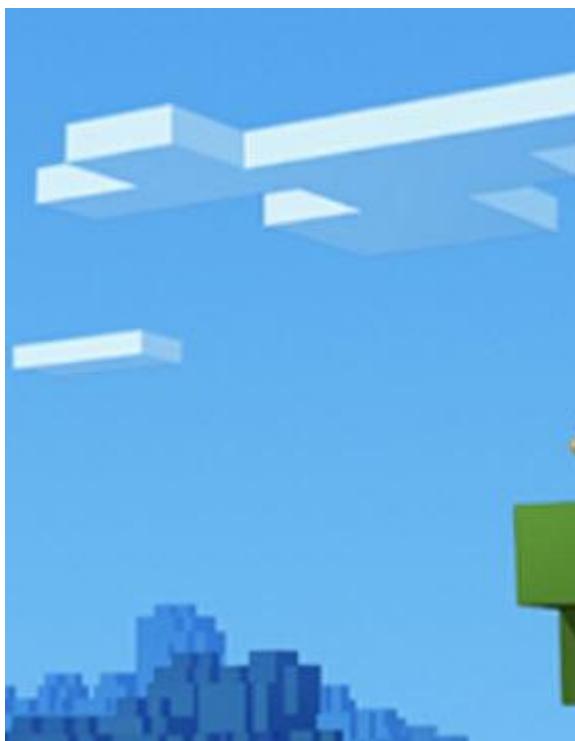


Figure 3.4.8 - Voxels

6D Tensors - Use Case: Batch of Voxel Animations

Dimension/Rank/Axes

A 5-dimensional tensor is a rank 5 tensor and has 5 axes.

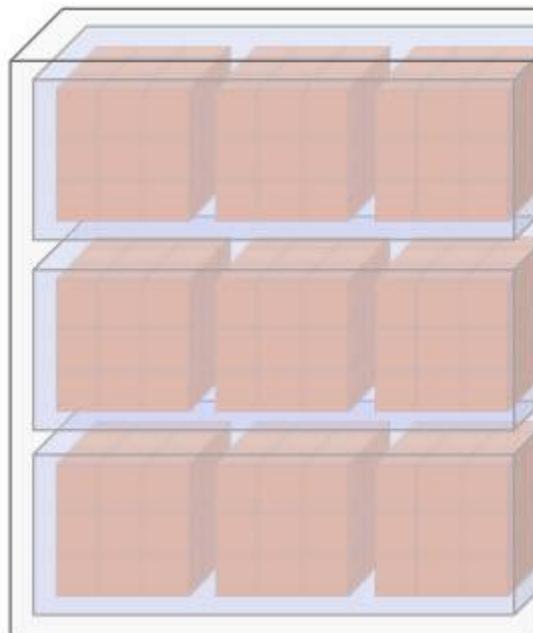


Figure 3.4.9 - 5D Tensor: Batch of Videos

A 6-dimensional tensor is a rank 6 tensor and has 6 axes.

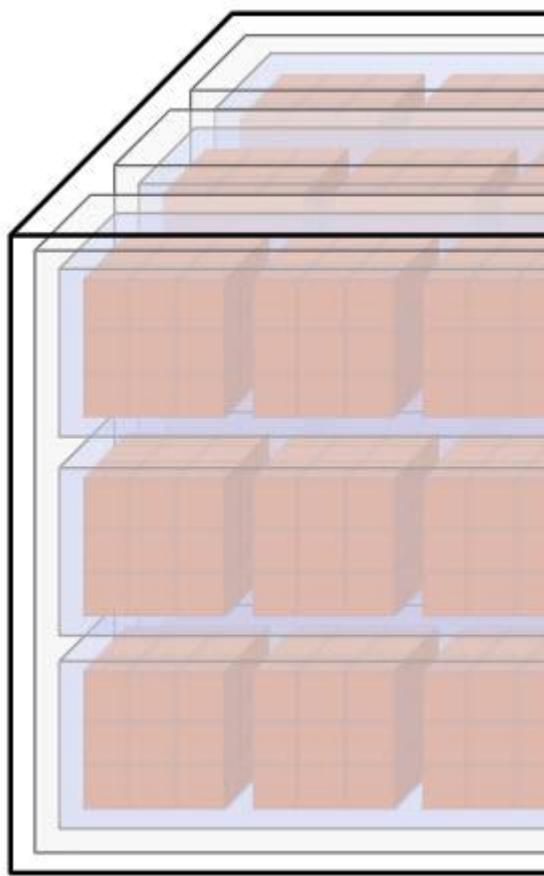


Figure 3.4.10 - 6D Tensor: Batch of Voxel Animations

Note: Six is the highest tensor rank that TensorFlow.js supports.

Tensors: Data Types, Shapes, and Lengths

Tensor Data Types

A tensor's data type refers to the type of data that the tensor will store. This could be integers or floating-point numbers (numbers with decimal points like "0.2").

The size of memory that is used to store a number depends on its data type, as can be seen in Table 3.4.1

Data Type	Memory used in bits to store 1 number of this data type	Range of numbers storable (assuming unsigned)
int-8 (char)	8	0 to 255
int-16 (short)	16	0 to 65,535
int-32 (long)	32	0 to 4,294,967,295

Table 3.4.1: Data Types, Memory Usage, and Range

Since ML models deal with millions of numbers, having data structures with only one type of numbers helps memory efficiency and enhances performance. In addition, having only one data type in an array makes it more rapidly accessible.

Tensor Shape

The shape of a tensor defines the number of elements present along each axis of the tensor. It is the representation of a tensor's length in each dimension.

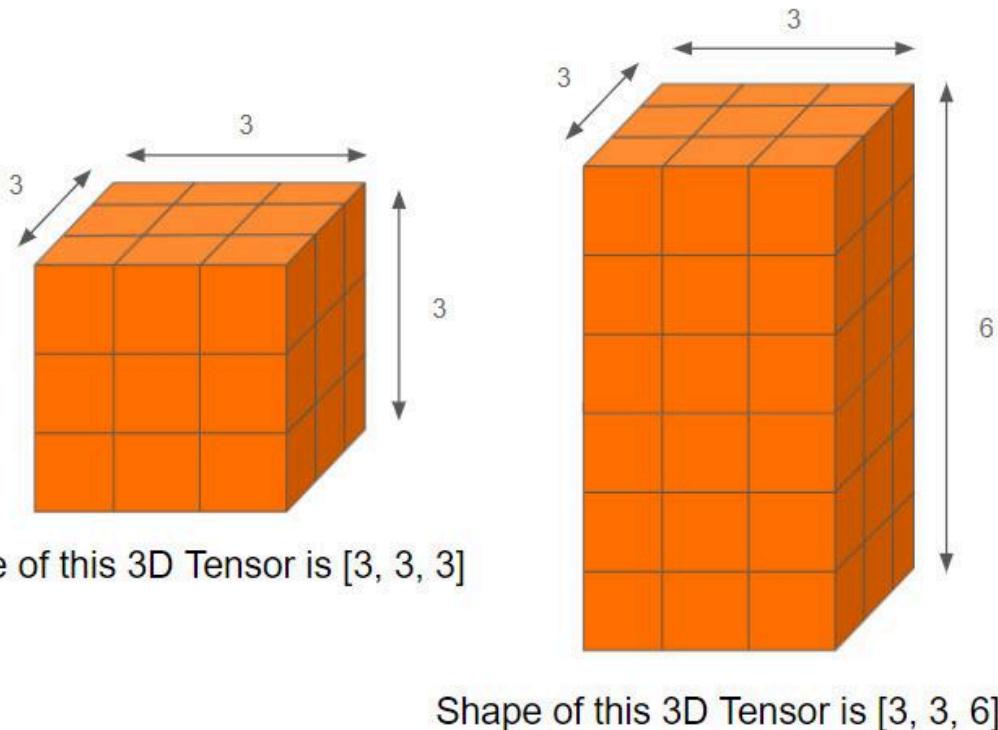
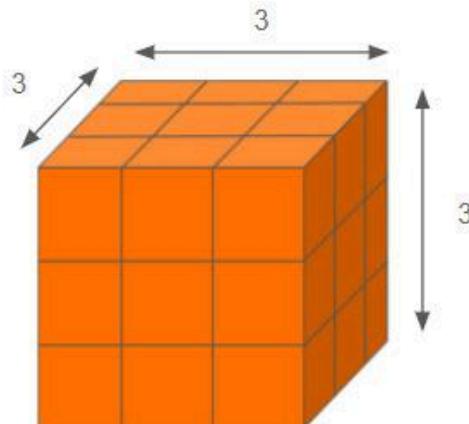


Figure 3.4.11 - Tensor Shapes

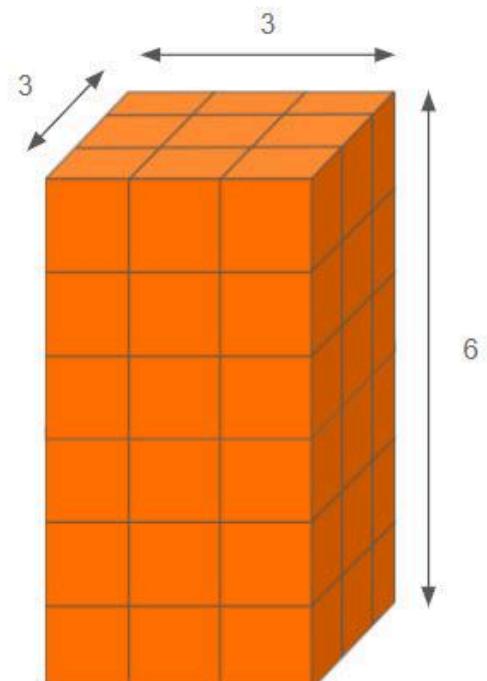
Tensor Size

Finally, size is just the total number of items in the tensor.

If you know the shape of the Tensor you can multiply the numbers to get its size.



Size of this 3D Tensor is 27



Size of this 3D Tensor is 54

Figure 3.4.12

Tensor Operations

Tensors are objects with many built-in operations. Go ahead and experiment with the code provided below in order to understand a few basic tensor operations.

```
/**  
 * Line below creates a 2D tensor.  
 * Printing its values at this stage would give you:  
 * Rank: 2, shape: [2,3], values: [[1, 2, 3], [4, 5, 6]]  
 **/  
let tensor = tf.tensor2d([[1, 2, 3], [4, 5, 6]]);  
  
// Create a Tensor holding a single scalar value:  
let scalar = tf.scalar(2);  
  
// Multiply all values in tensor by the scalar value 2.  
let newTensor = tensor.mul(scalar);  
  
// Values of newTensor would be: [[2, 4, 6], [8, 10, 12]]  
newTensor.print();  
  
// You can even change the shape of the Tensor.  
// This would convert the 2D tensor to a 1D version with  
// 6 elements instead of a 2D 2 by 3 shape.  
let reshaped = tensor.reshape([6]);
```

Note: We use methods such as ‘tensor.mul()’ for multiplication instead of regular JavaScript because we can leverage the faster execution of the graphics card or other backends that TensorFlow.js supports. These backends perform calculations in parallel and are much faster than vanilla JavaScript operations.

Memory Management

Unlike regular JavaScript bindings, you need to dispose of tensors manually. If you skip this step, you could cause a memory leak that could slow down, or even crash your computer. The TensorFlow.js API provides special disposal functions such as `tf.dispose` and `tf.tidy`, which you will explore later in this course.

Using Raw TFJS Pre-Trained Models

Raw models may still be in development or are developed to solve less common problems where few examples exist. Obtaining comprehensive documentation for raw models may not always be possible.

Types of TensorFlow.js Supported Models

Layers Model

Graph Model

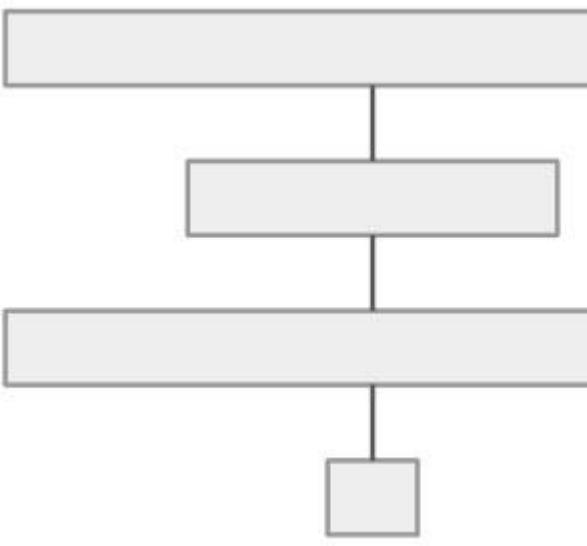


Figure 3.5.1: Layers Model Representation

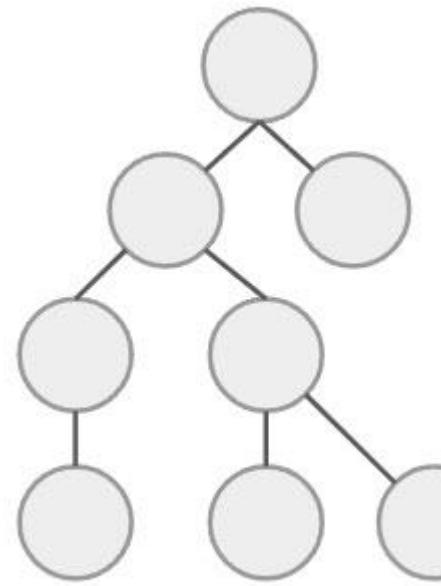


Figure 3.5.2 - Graph Model Representation

Layers models:

- Retain their higher-level building blocks for ease of use and debugging; They are easier to inspect and understand at runtime.
- Are not optimized and therefore may run slower when compared with graph models.

Graph models:

- Are highly optimized and can combine multiple operations into one batch for faster execution.
- Are not very easy to understand when unwrapped.

Layers models and the Graphs models are distinct formats, so we need distinct functions to load them.

TensorFlow.js Raw Model Files

TensorFlow models are saved as two different types of files: a **model.json** file and one or many **.bin** files.

model.json	.bin
The model.json file contains metadata regarding the model's type, architecture, and configuration details.	<ul style="list-style-type: none">• The .bin files store all the trained weights the model has learned. The original .bin file is broken down into shards as follows: shard1ofN.bin.....shardNofN.bin• Each shard is less than 4MB in size. This allows the browser to download multiple files simultaneously to speed up page load time.
All model files are hosted on a web server or a Content Delivery Network (CDN).	
Note: All files are stored in one directory by default. If you need to change the location of the binary files, you need to edit the model.json file to specify the new location.	

Understanding Models

TensorFlow.js provides us information on loaded models.

The `model.summary()` method provides information about:

- The model layers.
- The output shapes at each layer.
- The number of parameters in each layer.
- The total number of parameters.
- The number of trainable and non-trainable parameters.

Layer (type) 1	Output shape 2	Param # #
<code>dense_input (InputLayer)</code>	<code>[null,1]</code> 4	<code>0</code>
<code>dense (Dense)</code>	<code>[null,1]</code>	<code>2</code>
Total params: 2 3		
Trainable params: 2		
Non-trainable params: 0		

Figure 3.5.3 - Model Summary Sample

Figure 3.5.3 displays the following information about a model:

1. The model's first layer is an input layer with output shape (null, 1) and 0 parameters. This is a special layer since it consists of

only input values. It has no trainable parameters. The ‘1’ in the output shape of the input layer indicates that it expects a single number as input. **Note:** The input layer and shape may not always be present in a model’s summary since they depend on how the model was trained, in which case check the documentation or ask the developer for details about expected input shapes.

2. The model’s second layer in this case is the final layer. Hence its output shape is the shape of the tensor that the model outputs after it has processed the input data. The output layer’s shape is (null, 1), and the ‘1’ represents the number of outputs expected from the model.
3. The ‘null’ parameter in both shapes refers to the ‘batch’ size. If you want to classify ‘n’ input examples in one go (i.e., as a batch), you can simply pass n examples and they will be accepted as input. TensorFlow.js performs these predictions in one go for efficiency. Passing inputs as batches necessitate adding another dimension to the input tensor. So a 1D input tensor needs to be stored as a 2D tensor to be used in a batch.
4. The total number of parameters gives you an idea of the complexity of the model. More parameters typically means more memory will be needed to execute and also more processing power too.

Shape Mismatch

Should there be a mismatch between the expected shapes and the provided shape, TensorFlow.js throws up a ‘shape error’ on the console. This is a common occurrence when using tensors in ML.

```
✖ ►Uncaught (in promise) Error: Error when checking : expected      runtime.js:747  
dense_input to have shape [null,1] but got array with shape [1,2].  
    at PL (training.js:339)  
    at t.n.predict (training.js:1106)  
    at t.n.predict (models.js:777)  
    at loadModel (?editor_console=true:148)
```

Normalization

‘Normalization’ is a pre-processing tool that changes the values of a dataset such that they end up on a common scale. Suppose your input data has two features. The values of one feature range from -100 to 100, while the values of the other feature range from 0 to 1000. The two sets of values cannot be compared easily. For example, 50 would be considered ‘high’ for the first feature, while the same value would be on the lower end for the second feature.

Normalization can be performed on this data to ensure that each feature value is a number between 0 and 1. This makes it more efficient for the machine learning model to process the data and learn from it.



Key Concept 1

The Normalization Process

1. Calculate the difference between the maximum and minimum values in your dataset. In the example provided earlier, the range for the first set is 200 ($100 - (-100)$), and the range for the second set is 1000 ($1000 - 0$).

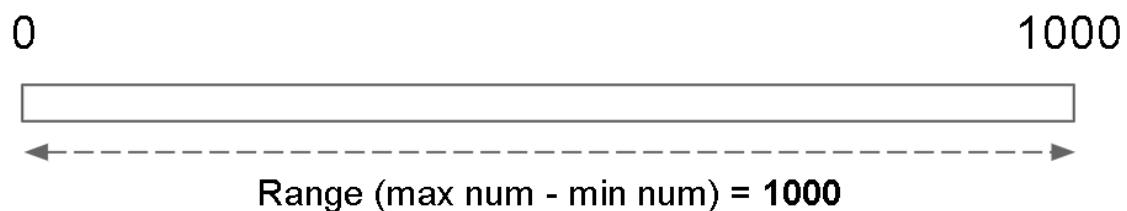
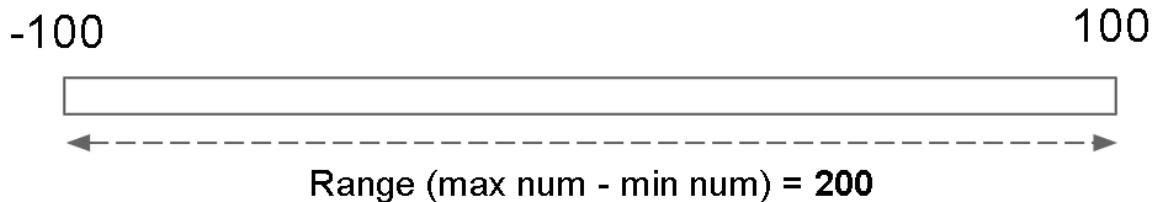


Figure 3.6.1.4: Normalization 1

2. Subtract the minimum value in your dataset from each value. In Figure 3.6.5, a random value (50) is chosen for representation.



$$\text{Range} = 200$$
$$50 - (-100) = 150$$



$$\text{Range} = 1000$$
$$50 - 0 = 50$$

Figure 3.6.1.5: Normalization 2

3. Divide the resulting value from the previous step (step 2) by the range of the dataset. This operation always returns a number between 0 and 1, regardless of the input values and the range.



Range = 200
 $50 - (-100) = 150$
Normalized: $150 / 200 = 0.75$



Range = 1000
 $50 - 0 = 50$
Normalized: $50 / 1000 = 0.05$

Figure 3.6.1.6: Normalization 3

Normalization Formula

$$\begin{aligned}
 X(\text{normalized}) &= \frac{X - \min(X)}{\max(X) - \min(X)}
 \end{aligned}$$

What's particularly useful about this normalized representation, is that often your original input image may be a different size altogether than what was input into the model.

Currently, the input to the model was 192 by 192 pixels, but if the original image was of the same aspect ratio (also a square in this case), maybe 1024 by 1024 pixels, as you have the normalized coordinates being returned, you can multiply the normalized y value by 1024 and the x value

by 1024 instead to find the true position in that larger image too! This is very useful and convenient when you want to draw something on the original image you had in the correct position.

Do take note however if the original image was a different aspect ratio, then you would need to scale values accordingly in each dimension to account for that.

TensorFlow Hub

Introduction to TensorFlow Hub

<https://www.tensorflow.org/hub?hl=es>

TensorFlow Hub is a repository of machine learning models. It is a ‘model garden’, a website where developers worldwide upload their raw saved models, which can then be used as needed. While there is a wide variety of models available, some of them may:

- Not have comprehensive documentation
- Be of varying quality
- Be at different stages of production
- Not have an easy-to-use interface

Note: In addition to actually using the model for inference, you need to perform two steps:

- **Pre-processing:** You need to write code to transform the input data into a form that can be sent to a model. We explore one way of doing this in this lesson.
- **Post-processing:** After the model performs inference, you need to write additional code that transforms and uses the model output somehow.

Writing Custom Models

Training / Testing / Validation Datasets:

Before training a machine learning model, you need to find data from which the model can learn. In supervised learning, you provide this data as labeled data. Each data entry consists of features or attributes that have a corresponding example answer or label. The model learns from such features and labels so that it can predict the answers for data that it has not seen during training.

For instance, consider a model that is trained to classify fruits. The data includes the ‘weight’ and ‘color’ of fruits as features and the labels ‘apples’ and ‘oranges’ as answers. Once the model is trained, it attempts to classify fruits that it has not seen during training as apples or oranges. Since apples and oranges occur in a wide variety of sizes and colors, you need to provide thousands of examples to train the model. This lesson explores how you collect, split, and clean the available data used to train the model.

Training Data and Testing Data

Engineers do not always use all the available data to train the model. In some use cases, they use 80% of the data to train the model (the ‘training data’) and set aside 20% to test the trained model (the ‘test data’).

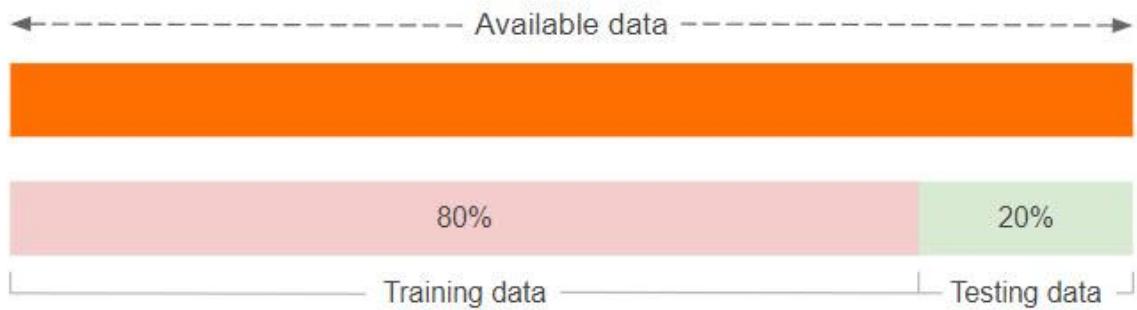


Figure: 4.2.1. Training vs. Testing Datasets

Once the model is trained on the training data, engineers evaluate its performance on the test data and fine-tune the model’s parameters to improve its performance. This process is repeated until they find the model that works best.

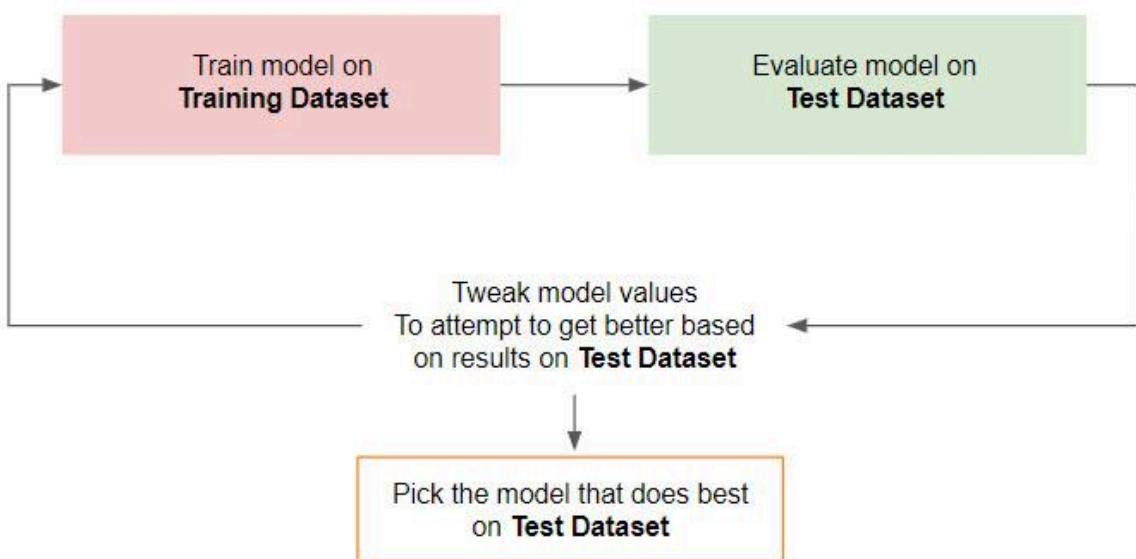


Figure 4.2.2. Training and Testing Datasets - Workflow

A disadvantage of this approach is that the model may adapt itself to simply perform well on the testing data too as it is tweaked based on the testing data that it also sees. If it learns the training data perfectly, this is called ‘overfitting,’ a phenomenon wherein the model performs exceptionally well on the data it has trained on but does not generalize well.



Key Concept 2

Training, Testing, and Validation Data

Instead of splitting your data into training and testing sets, you may randomly split the available data into 3 sets to avoid the prior situation:

- Training dataset
- Testing dataset
- Validation dataset

The size of the splits will vary according to how much data you have available. A typical distribution is shown in Figure 4.2.3.

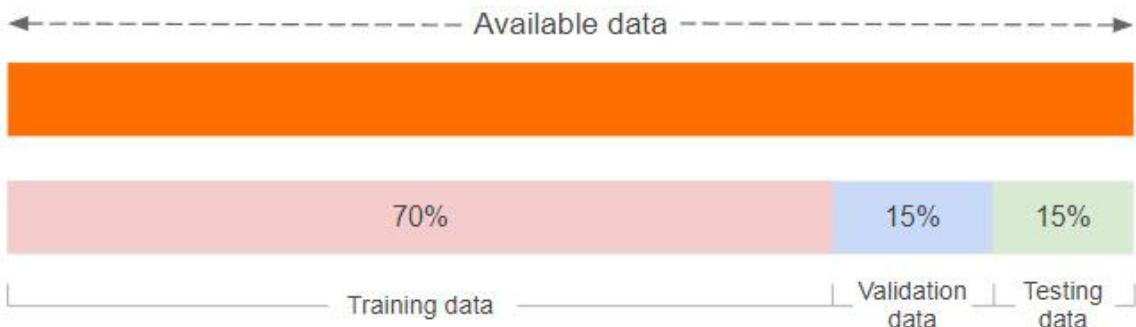


Figure 4.2.3. Training vs. Testing vs. Validation Datasets

When you have a three-way split, you start by training the model using the training set. Then, you use the validation set to evaluate the model's performance until you get the best performance. You then use the test set to see how well your selected model works but the model never updates itself based on this test set, as illustrated below:

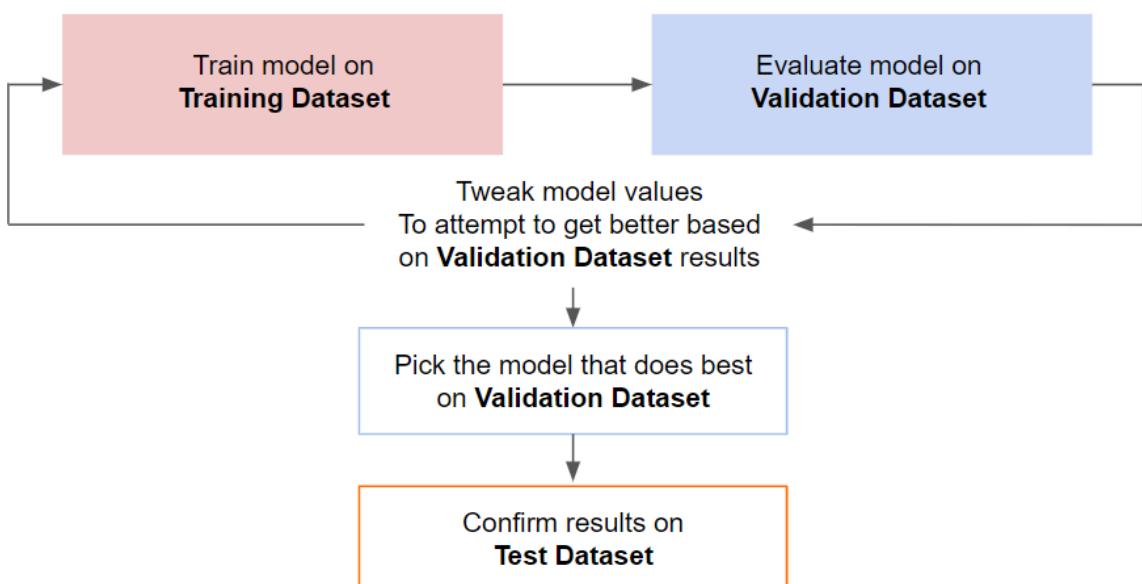


Figure 4.2.4. Training, Testing, and Validation Datasets - Workflow

The advantage of this approach is that you can avoid overfitting because the model has never seen the test set examples and cannot learn from them either, so it is a good indication of how your model may perform in the real world.

Remember, the goal of training the model is to predict the correct answer for all the examples it could possibly encounter in the real world, not just the ones on which it has been trained!

Collecting Data for Machine Learning

Once you know what data you need to collect, there are several sources and methods from which you can obtain data. You can:

- Obtain data manually from your existing data.
- Obtain data manually by recording and collecting data on your own to start a new data collection yourself.
- Re-use existing data from other sources.
- Use a third party to collect data.

Method	Example Sources	Example use case	Features
Use your own existing data	Database or server logs	You own a recruiting website with registered users. Its database contains years of experience, city,	Data Quality: High Cost: Low

		and current salaries of each registered user.	Time to Obtain: Short
Collect new data	Spreadsheets or databases populated from questionnaires or user inputs	You own a website with many users, but you were unable to / could not record the features required to train the new model. You now start recording the required data once you know what you need.	Data Quality: High Cost: Low - Medium Time to Obtain: Long
Re-use existing data from other sources	Open Source Datasets from Companies, Governments, Scientific communities, Kaggle, DataHub.io, and more Sites such as datasetsearch.research.google.com	MNIST dataset - a collection of handwritten digits from 0 - 9 that people can use to recognize numbers in handwriting.	Data Quality: Variable Cost: Low - Medium Time to Obtain: Short
Use a third party to collect data	Spreadsheets or Databases populated from questionnaires	You need images of cat breeds to create the ultimate cat classifier. You hire a company to take photos of cats for you worldwide with correct labels	Data Quality: High Cost: High

			Time to Obtain: Variable
--	--	--	------------------------------------

Quantity of Data

Since a model has to generalize to all possible real-world scenarios, you will usually need thousands or even millions of examples as input data to train a machine learning model. The more granular your data is, the more likely your model is able to perform accurate classification or predictions later on.

Refining Data

When you are dealing with millions of examples of data, you will frequently come across erroneous data that could disrupt the training process.

For example, the data in Figure 4.2.5 has the following errors:

- The price of the house in row 2 is provided in British pounds instead of dollars.
- The house type in row 3 is written in uppercase letters.
- The size of the house in the last row is not filled.
- The number of bedrooms in the last row is not an integer.

Ensure that your data is free of such errors before you start training your model as this could throw the learning process off.

Price	Size (sqft)	House Type	Bedrooms
500000	500	Detached	4
£739802	70	Detached	3
1000000	950	DETACHED	3
1400000	null	Apartment	2.5

Figure 4.2.5. Housing Prices Dataset with Erroneous Data

Feature Selection

Choosing the right features goes a long way in making your model more efficient and small. As you saw in a previous lesson, while classifying fruits, you selected the ‘weight’ and ‘color’ of the fruit as features and discarded ‘ripeness’ and ‘number of seeds’ since they were not useful. Similarly, to predict the price of a house, the ‘size’ may be a more effective feature than the house type.

Bias

You should be extra careful while collecting data and defining the features and attributes you will use to train the model in order to avoid introducing ‘bias’ in your system. Biased models may have far-reaching consequences, including amplification of negative stereotypes. For instance, if you were developing a speech recognition model for English and used people from only one geographic location as your input data, the model could be biased against inputs from people who live elsewhere. Often this is not intentional but you should look out for these edge cases when gathering data as it can

be hard to spot when everything is working with the 10 people who are working on the project. Test with real users before any official launch and iterate as needed to refine the model if you find issues collecting more data for potentially overlooked areas or maybe even creating multiple models for different situations if needed.

Perceptrons and Neurons

Biological and Artificial Neurons

The human brain has around 86 billion neurons connected in a biological neural network. A neuron's inputs receive stimuli and the outputs of those neurons may drive the inputs of several other neurons that are attached to it. For example, when you recognize a cat, specific groups of neurons get excited or activated. How active the neurons get depends on how confidently they correctly identified the features of a cat they have learnt to look out for. This broad process comprises the following steps:

1. Different clusters of neurons may get excited when they see certain features like eyes, fur, whiskers etc. You can think of this as having voting power for different features by the lower level neurons in the network.
2. If all these are active at the same time, the higher level neurons that sample the outputs of those lower level neurons may then also get activated resulting in your brain identifying the animal. You can then determine if you observed a cat or a dog for example based on what features were present and how confident they were (see Figure 4.3.1.1.).

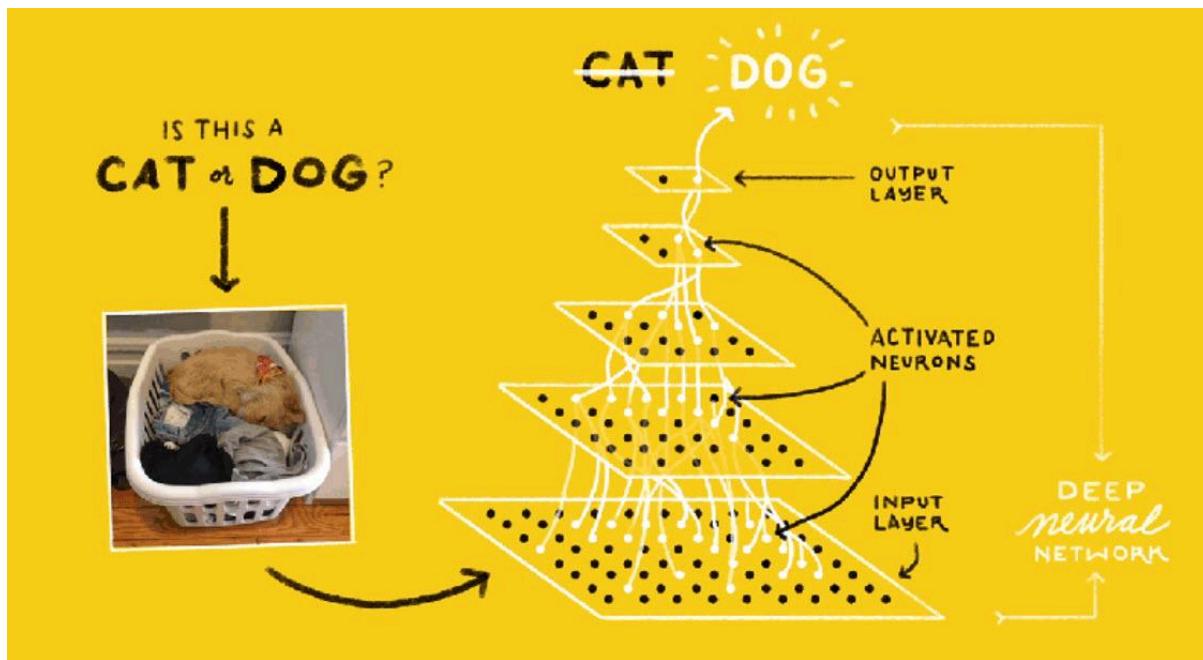


Figure 4.3.1.1. How the Brain Recognizes Objects

Artificial Neural Networks

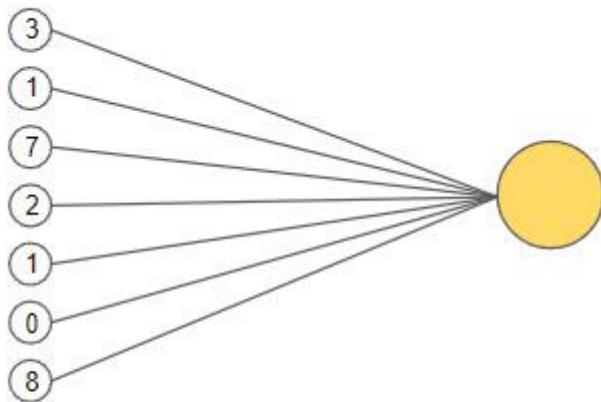
Artificial neural networks are loosely based on how we believe the brain is structured. Just as the brain looks at an object or an animal's features and then predicts what that object or animal is, an ML model is trained on features and calculates the probabilities for its predictions using mathematics instead of biology. When these artificial neurons are connected in multiple layers, they are called **deep neural networks**.

Artificial Neurons and Perceptrons

The artificial neuron is the basic unit or element of a neural network. One type of artificial neuron is the **perceptron**. The perceptron is the basic building block of ML that takes in input data and transforms inputs into some output number if it crosses some threshold for activation.

Figure 4.3.1.2 shows a single perceptron receiving multiple numerical inputs.

- These numbers are the sample data from which you want your model to learn.
- Each input represents a feature for an item in your data.
- This example has seven input values, but if you use a 100x100 grayscale image, there will be 10,000 input values as each pixel would be a feature!
- A single neuron can have as many inputs as needed to sample all the input data.



Inputs

(numbers from your sample data you want to learn from eg pixels in an image)

Figure 4.3.1.2. Single Perceptron with Multiple Inputs

Figure 4.3.1.3 shows the “weight” given to each of the input signals.

- Observe that the thicker lines carry greater weight than the thinner ones in this diagram.
- Heavier weights contribute more to the value that is calculated in the perceptron.
- The weights allow the neuron to amplify or suppress the value of each input based on the importance of the input.
- Before the perceptron is trained, the weights are randomly chosen to start with. So outputs will also seem pretty random at first before training.

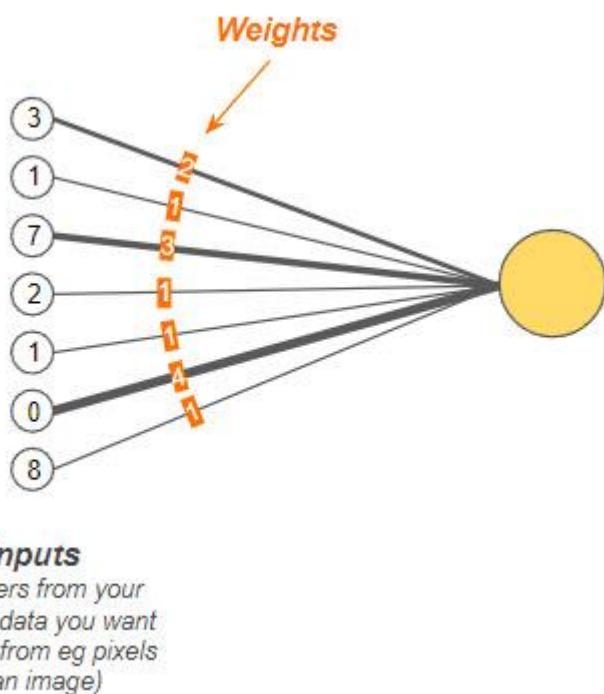


Figure 4.3.1.3. Single Perceptron with Multiple Inputs and Weights

Figure 4.3.1.4 shows a new parameter called the 'bias' added to the model.

- In this case, the 'bias' is randomly chosen to be 3. It is also randomly initialized at the start.
- The bias can be changed during the learning process if needed.

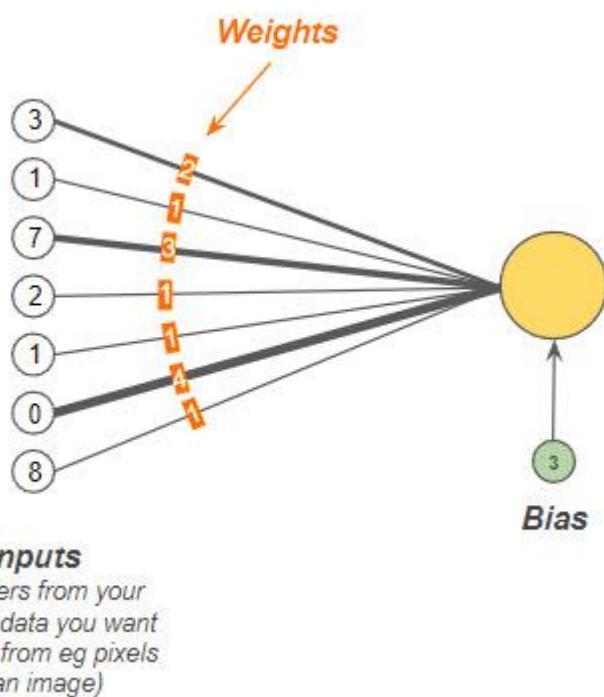


Figure 4.3.1.4. Single Perceptron with Multiple Inputs, Weights and Biases

Figure 4.3.1.5. shows the first half of the perceptron.

- Each input value is multiplied by its corresponding weight.
- The resulting products of all the input values and their weights are summed together.
- The bias value is added to the sum obtained in the previous step.
- The final value (42) is shown in the perceptron in the yellow circle, if you were to calculate the multiplications of inputs, sum them, and then add the bias.

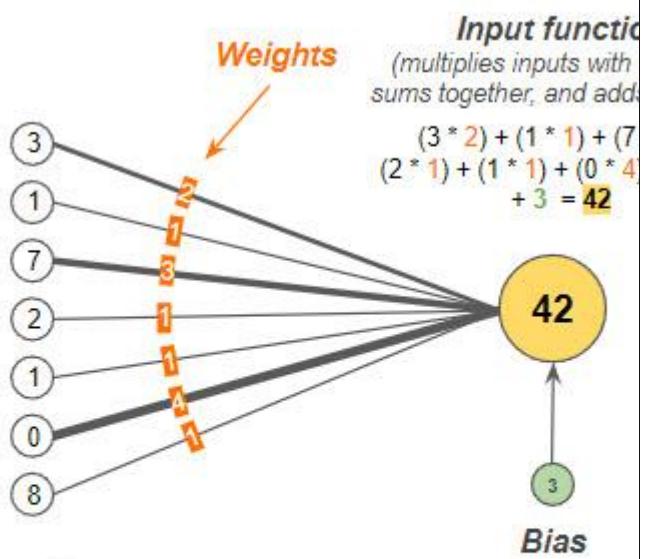


Figure 4.3.1.5. Single Perceptron Input Function

Figure 4.3.1.6 shows the activation function, which is the final part of the perceptron.

The activation function activates the neuron to provide an output number if the total calculated is greater than a threshold.

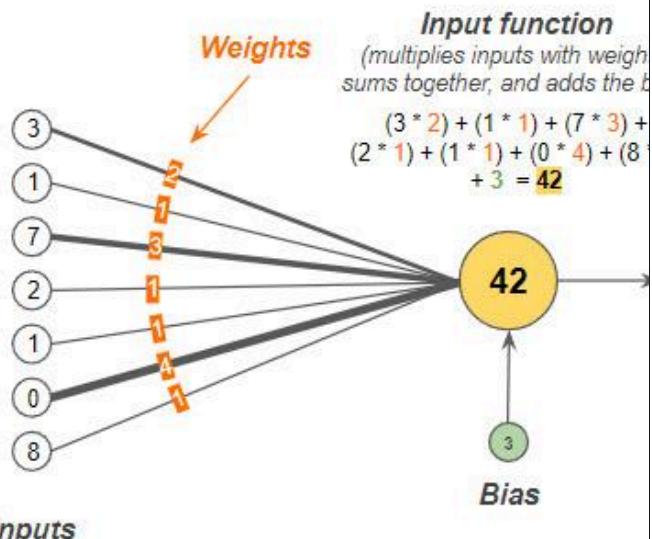


Figure 4.3.1.6. Single Perceptron Activation Function

Activation Functions

Activation functions define when a neuron activates to produce an output.

Example

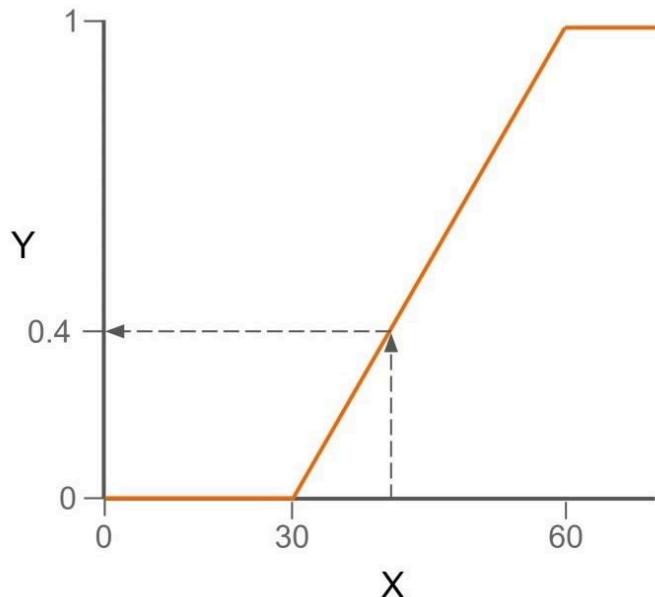


Figure 4.3.1.7

Consider the graph shown in Figure 4.3.1.7:

- For all values of 'x' that are less than 30, the 'y' output is 0.
- For 'x' values that increase from 30 to 60, the 'y' output steadily increases between 0 and 1.
- For all 'x' values above 60, the 'y' output remains at 1.

If your 'x' value were to be 42, the 'y' output would be roughly 0.4 in this graph. The neuron will now activate based on the threshold you have decided for the model with 0.4 as an output of the neuron.

There is a range of distinct activation functions available, some suited to particular tasks. The chart below shows some common ones:

Name	Plot	Equation
Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$
ArcTan		$f(x) = \tan^{-1}(x)$
Rectified Linear Unit (ReLU) ^[2]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) ^[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) ^[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$

Figure 4.3.1.8. Activation Functions

Rectified Linear Unit (ReLU)

ReLU is an example of a popular activation function. The function returns 0 if 'x' is less than 0, but returns x in all other cases with no upper limit, it just passes through numbers that are above 0. ReLU is an example of a non linear activation function as it is not a single straight line like the identity function shown above.

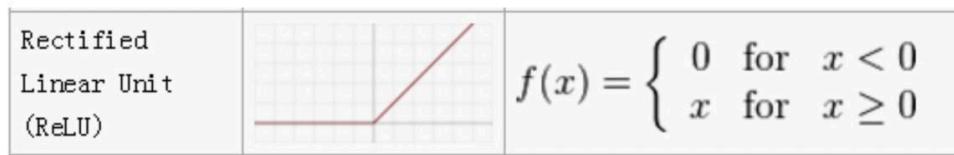


Figure 4.3.1.9. The ReLU Activation Function

Putting it all together:

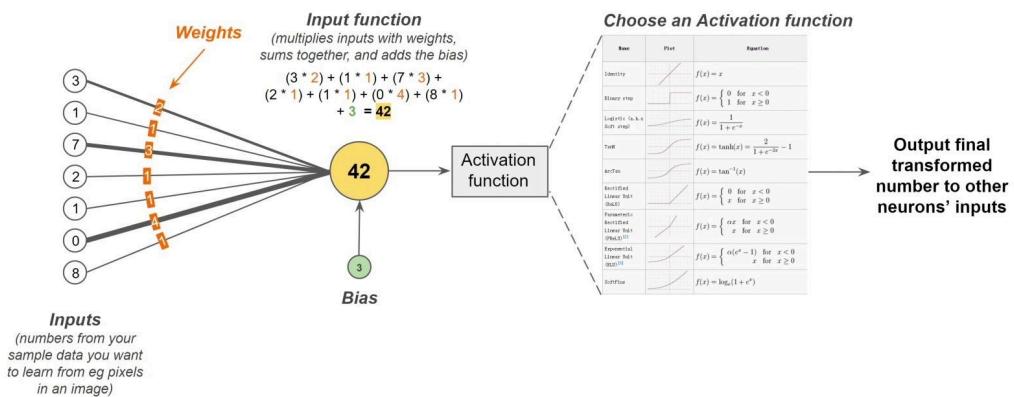


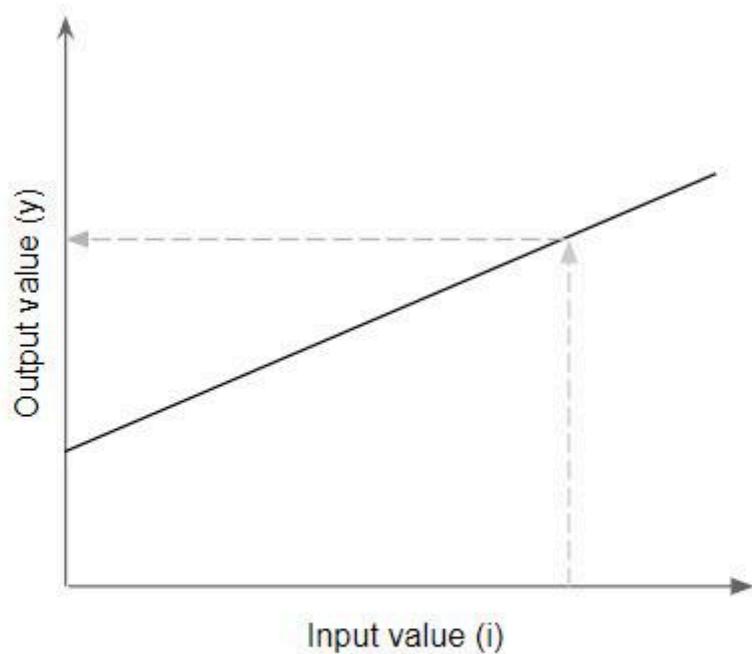
Figure 4.3.1.10. Complete Perceptron Diagram

In a neuron:

- Inputs are multiplied by their corresponding weights, the totals are added, and a bias is added to arrive at a grand total.
- The grand total is sent through an activation function that transforms the number into an output value.
- In the case of a **perceptron**, the activation function outputs either a 1 or a 0 based on some defined threshold. A perceptron, essentially, turns inputs into binary outputs.

Training Neurons:

You now understand how neurons learn from the training data passed to them.



Regression problem
(What's house price given size of house?)

Figure 4.3.2.1. House Size vs. House Price

Figure 4.3.2.1 depicts a graph showing the linear relationship between the size of a house and its price. Using linear regression, a model can calculate the equation of a straight line like the one shown here to predict the price of a house given its size.

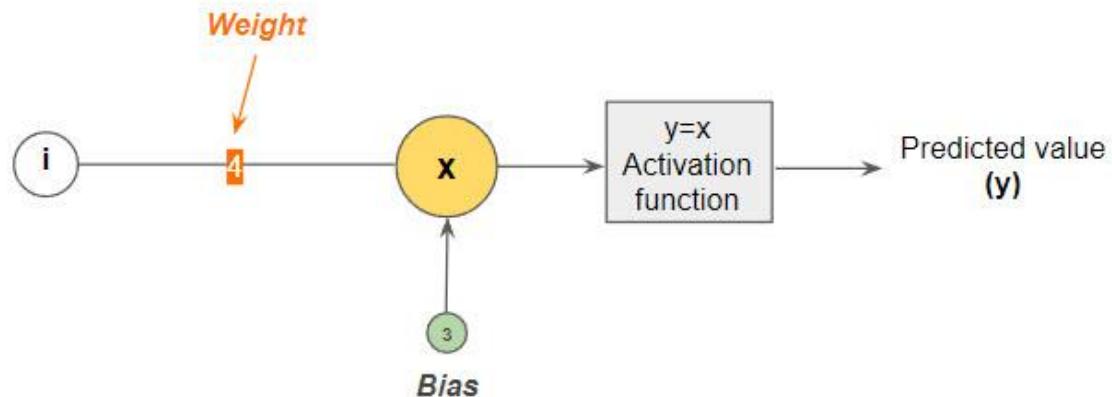


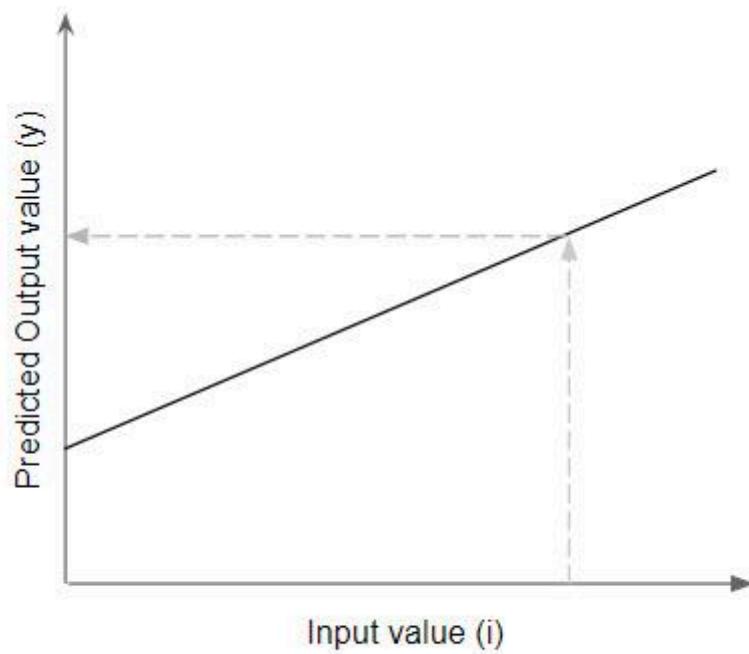
Figure 4.3.2.2. Model of a Single Neuron

Consider a model with a single neuron that can predict a house price given its size (Figure 4.3.2.2).

- The neuron takes in one input ‘i’. This is the size of a particular house.
- Since there is one input, it has a single corresponding weight ‘w’ and a bias ‘b.’
- The value of ‘x’ is calculated by multiplying the input value with the weight and adding the bias to the product.
- ‘x’ is then passed through an activation function to produce output ‘y’.

In this case, the activation function is the ‘identity function,’ defined as ‘y’ = ‘x.’ The final predicted price is ‘y’.

Notice that the final value ‘y’ is actually calculated by multiplying the input ‘i’ with weight ‘w’ and adding bias ‘b’ (as seen in Figure 4.3.2.3).



$$y = w^*i + b$$



Mathematically the same as
equation of straight line!

Figure 4.3.2.3. Equation of a Straight Line

This is mathematically identical to the equation of a straight line! All that the neuron needs to do now is to find the values for the weight and bias that best fit the training data in order to predict house prices.

How Neurons Learn

So you have collected the following training data for your regression model.

Input	Output
1	2
2	4
3	6
4	8
5	10

Table 4.3.2.1.

The values in Table 4.3.2.1. Represent could something meaningful, such as the size of the house or the number of bedrooms as the inputs and the price of the house as an output. In this example case however, observe that the output is arrived at by simply doubling the input. The model needs to figure out this relationship during training by itself.

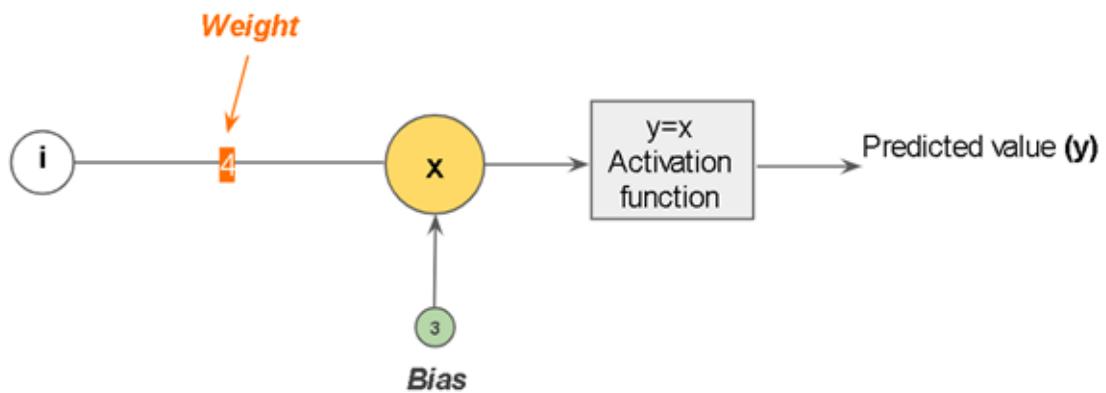


Figure 4.3.2.4. Model of a Single Neuron

Initially, the model has no idea of the relationship between the input data and the answers. It starts by assigning a random value to the weight and bias. In figure 4.3.2.4., the values 4 and 3 have been assigned to the ‘weight’ and ‘bias,’ respectively.

Using these weights and biases, the neuron calculates the following values for the input values:

Input	Output	Prediction
1	2	7
2	4	11
3	6	15
4	8	19
5	10	23

Table 4.3.2.2.

Notice that the model could not predict even one output accurately. The difference between the predicted output and the actual answer is the error. The neuron now needs to adjust the values of the weights and bias to decrease the error, and this process is called training.

Input	Output	Prediction	Error
1	2	7	+5
2	4	11	+7
3	6	15	+9
4	8	19	+11
5	10	23	+13

Table 4.3.2.3

The measure of how wrong a model was for a current batch of predictions is defined by a ‘loss’ function. A commonly used loss function is the **Mean Squared Error (MSE)**. This is calculated by squaring each error and finding the mean of all the squared errors. In this case below, you will find the MSE to be 89:

Input	Output	Prediction	Error	Squared Error
1	2	7	+5	+25

2	4	11	+7	+49
3	6	15	+9	+81
4	8	19	+11	+121
5	10	23	+13	+169
Total				445
MSE				445/5 = 89

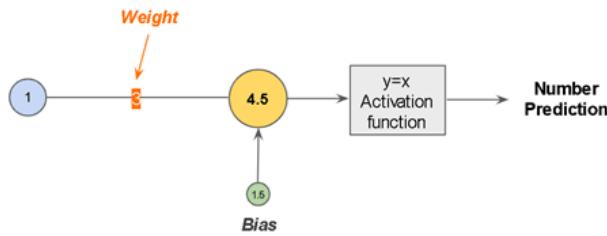
Table 4.3.2.4.

Since the MSE is a positive number, you need to adjust your weights and biases by making them smaller. The **learning rate** defines the magnitude of the change. The learning rate describes **how much** you will change the weights and biases to get better predictions. If the learning rate is too large, the model could never find a solution. If the learning rate is too low, the model could take a very long time to find a solution.

The model now needs to figure out how to adjust the weights and biases to get better predictions. This is done by a method called **backpropagation**, which takes into account the loss and learning rate and figures out how much to change the weights and biases using complex mathematics which you do not need to know to appreciate what is going on. Essentially though, if the loss is positive, the value of the weights and biases would be reduced and vice versa.

Training in Action

Sum of Squared Error: 111.25
MSE Loss for this batch: 22.25



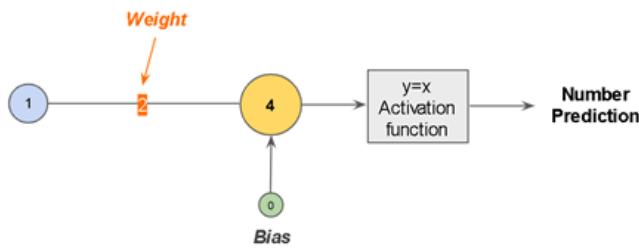
Input	Output	Prediction	Error	SE
1	2	4.5	+2.5	6.25
2	4	7.5	+3.5	12.25
3	6	10.5	+4.5	20.25
4	8	13.5	+5.5	30.25
5	10	16.5	+6.5	42.25
				111.25

Figure 4.3.2.5. Mean Squared Error with Incorrect Weight and Bias

Your model initially set the value of the weight to 4 and the bias to 3 (see Figure 4.3.2.4.) and obtained an MSE of 89 (see Table 4.3.2.4.). Since this is a large positive value, using the learning rate and backpropagation, the model may now decide to reduce the weight by 1 to 3 and the bias by 1.5 to 1.5 (see Figure 4.3.2.5.). The MSE calculated with the new weight and bias is 22.25, which is an improvement, but still higher than 0 so we can do this all again.

Using a single neuron

MSE Loss for this batch: 0



Input	Output	Prediction	Error
1	2	2	0
2	4	4	0
3	6	6	0
4	8	8	0
5	10	10	0

Figure 4.3.2.6. Mean Squared Error with Correct Weight and Bias

The backpropagation algorithm is now repeated, and the weight and bias are set to 2 and 0, respectively if you reduce by the same numbers as before. The MSE calculated with the new weight and bias is now 0, which means the neuron perfectly represents the input data. It has learned the equation of the line represented by the input data!

Note that you will rarely arrive at a loss of 0 when training real world models. If you do, it usually means the model has overfitted to the training data and will not generalize well to new data that it has not seen before.

Another point to consider is that the outputs are rarely exact multiples of inputs and will not fall perfectly on a single line -most organic data is somewhat noisy and has some randomness to it.

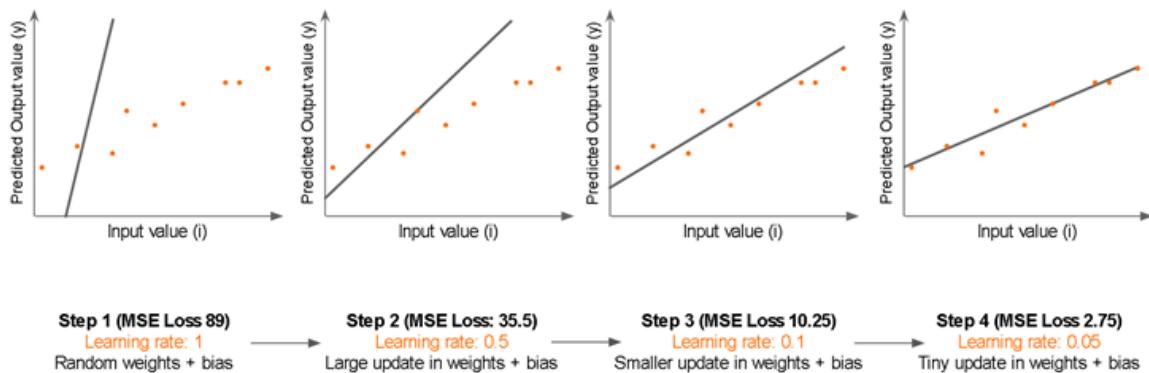


Figure 4.3.2.7. How a Regression Model Learns

Figure 4.3.2.7. depicts a more realistic solution to a real-world regression problem. Though the input values have a linear relationship with the output values, they do not fall on a single line. The model attempts to find the line that best fits the data with minimal loss. This happens over multiple steps, with the weights and biases adjusted according to the loss and the learning rate. Note in this example that the learning rate decreases as the loss decreases. This helps avoid overshooting the optimal values for the weights and biases.

Note: Once a model is trained, it will always provide an output, no matter what input value you provide. If you test the model with an input value outside the training data range, the model will, at best, provide an educated guess. You should be cautious while using predictions in such cases as they could be very wrong from the real data if you were to find some.

Loss Function

A measure used to calculate how wrong a model's prediction is, given a set of input values and their correct outputs or answers.

Mean Squared Error

A method that calculates the loss function by squaring the loss obtained for each input (difference between predicted and actual output), adding the squared losses, and finding their mean.

Backpropagation

An algorithm that uses the ‘loss’ and the ‘learning rate’ to figure out how to change the weights and biases during each training update step.

Learning Rate

A parameter defined by the machine learning engineer that determines the magnitude by which the weights and biases of a model should change. Too high a learning rate and the model may overshoot the optimal values for a line of best fit, and too low a learning rate may take the model too long to find a solution.

Implementing a Neuron for Linear Regression - Analyzing Data

You want to build a custom machine learning model that estimates house prices in a certain city. Your customers have provided data in the form of a table that contains the size in square feet, number of bedrooms, and the house value in dollars for 2000 houses. In this lesson, you analyze the existing data.

Example Data

Size (Sqft)	Bedrooms	Value
3225	3	262,300
3789	3	270,000
3636	3	262,000
3109	3	259,700
3836	3	259,000
3510	3	258,800
3849	3	258,600
3619	3	258,500
3619	3	258,500
3619	3	258,500
3858	3	258,400
3858	3	258,400

3762	3	258,200
3660	3	257,400
3510	3	257,200

Analyzing Data

1. **Check for missing values:** If you have a small quantity of data, you can visually check for missing values. You may also write code to show maximum, minimum, or null values. A number of third-party libraries also exist that can perform this task for you. [Danfo.js](#) is one such open-source library that is used to manipulate and process data that aims to replicate the famous Pandas library from Python but in JavaScript!
 2. **Visualize Data:** Your input data has two features: the size of the house in square feet, and the number of bedrooms, arranged in two columns. You can visualize the correlation between each feature and the house price to see how well they relate.
- a. **Size:** Figure 4.4.1.1 shows a scatter plot where the house sizes are plotted against the house prices.

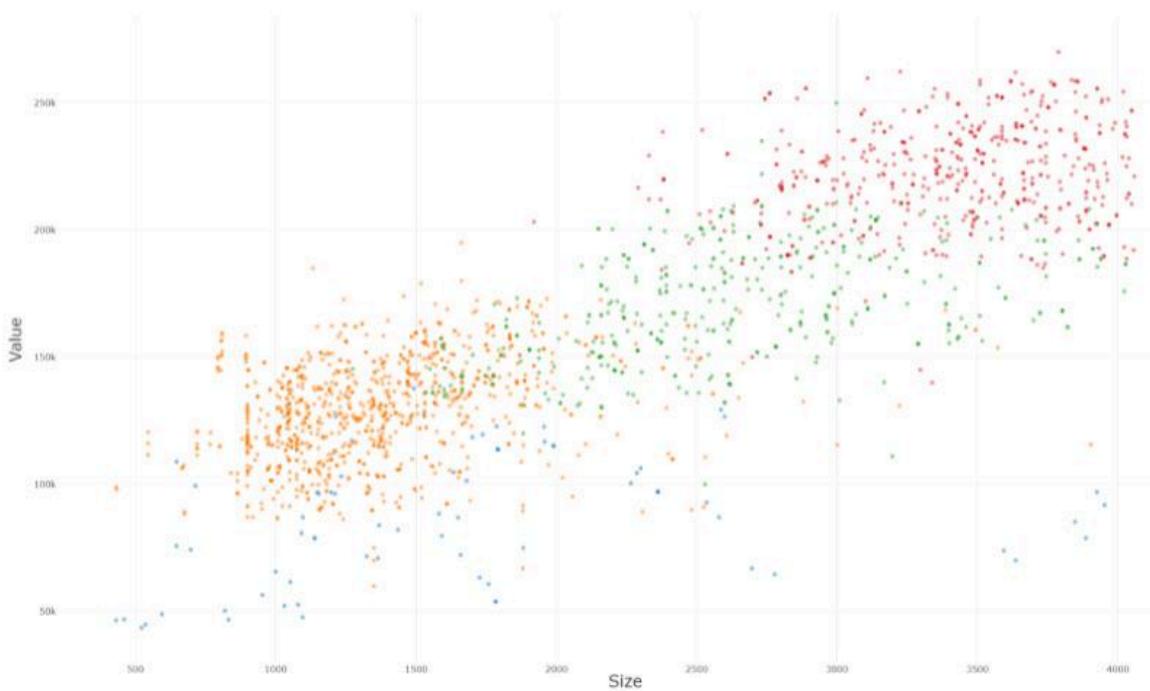


Figure 4.4.1.1. Scatter Plot of House Size vs. House Price

Note that as the size of the houses increases, their prices also increase more or less in a linear fashion. Figure 4.4.1.2 also highlights a few outliers that do not fit this pattern.

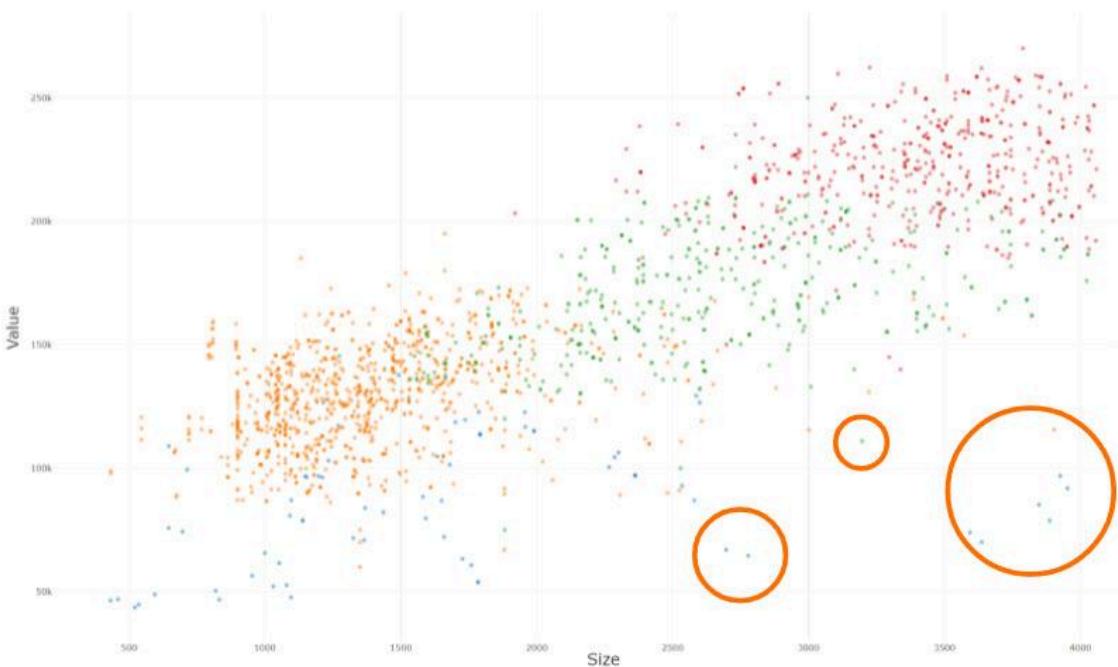


Figure 4.4.1.2. Scatter Plot of House Size vs. House Price with Outliers

Ignoring the outliers, the house size and the house price can be correlated and approximated with a straight line (see Figure 4.4.1.3).

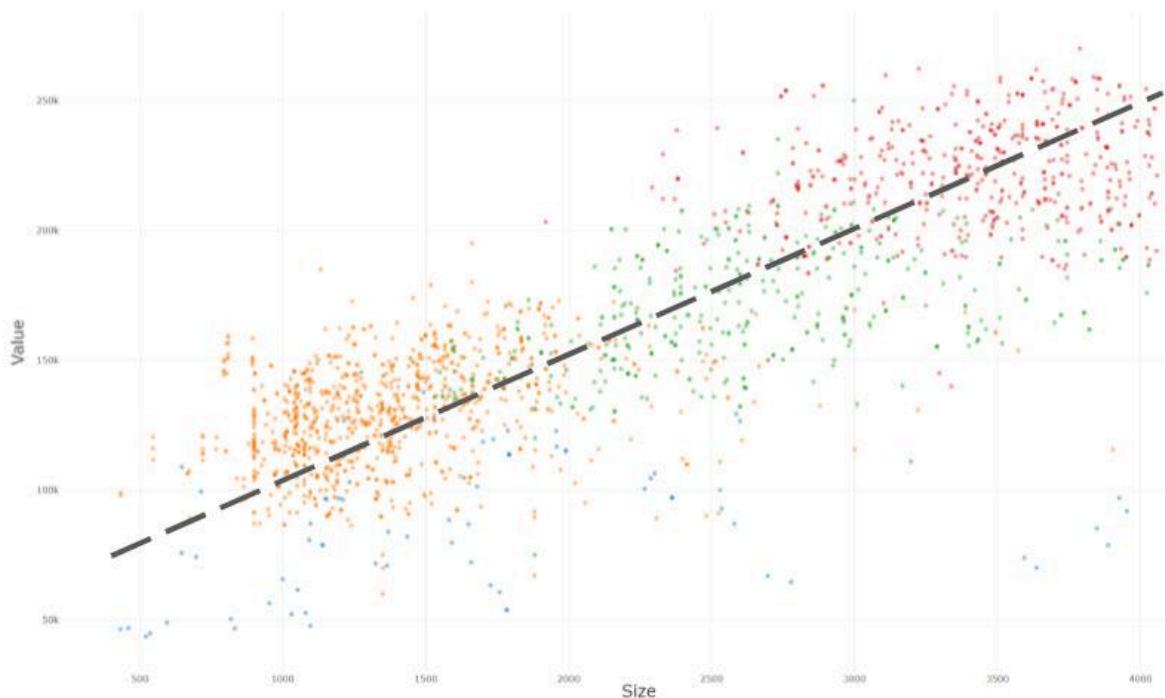


Figure 4.4.1.3. Linear Relationship between House Size and House Price

b. **Bedrooms:** Figure 4.4.1.4 shows a scatter plot where the number of bedrooms is plotted against the house prices instead.

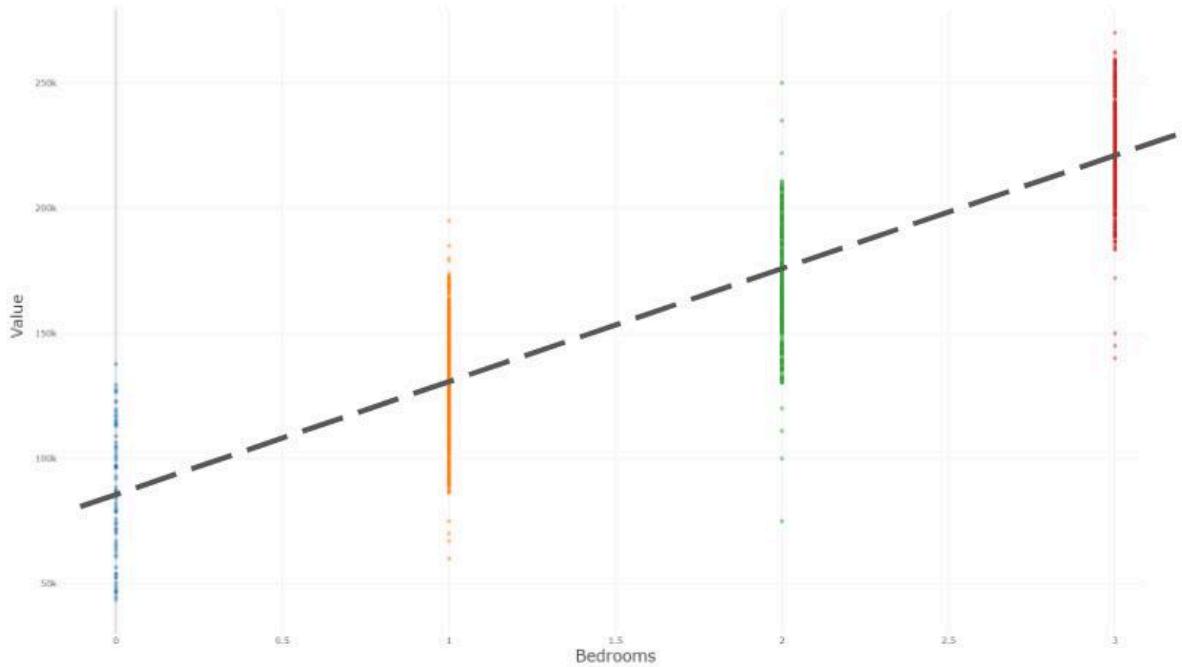


Figure 4.4.1.4. Scatter Plot of Bedrooms vs. House Price

Observe that for each bedroom that is added, the house values increase by about \$100,000.

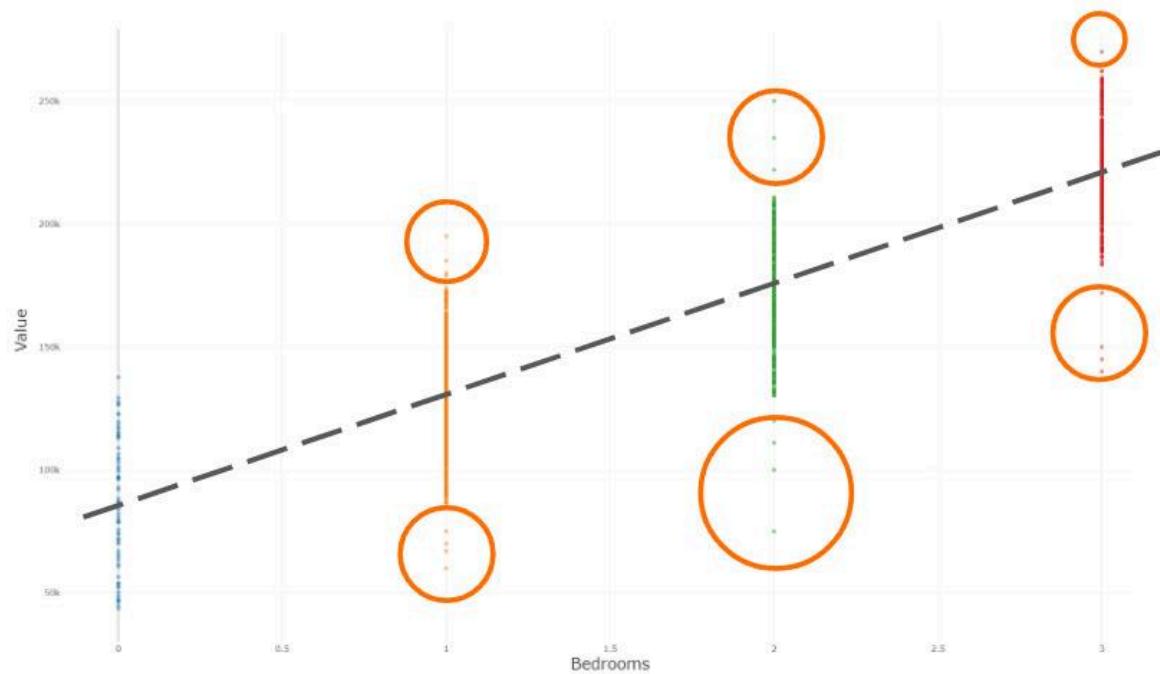


Figure 4.4.2.5. Scatter Plot of Bedrooms vs. House Price with Outliers

The Size vs. Value scatterplot and the Bedrooms vs. Value scatterplot both have a few outliers (see Figure 4.4.2.5) that do not fit a linear pattern. You may consider dropping these data points from your dataset to achieve better predictions in the vast majority of use cases.

Since there is a strong linear correlation between each input feature and the house price, you can develop a model with a single neuron to perform a linear regression. In the next couple of lessons, you will implement a neuron for linear regression that can use to learn from and fit the data you saw in this section.

Implementing a Neuron for Linear Regression - Importing and Normalizing Data

Define the Normalization Function

1. One way to normalize a set of values is to ensure all numbers are in the range of 0 and 1.
2. Start by defining a function ‘normalize’ that takes the tensor whose values you wish to normalize as a parameter along with optional ‘min’ and ‘max’ parameters. The ‘min’ and ‘max’ parameters are used to store the minimum and maximum values of the tensor. These values are used during normalization for efficiency if they are already known to remove the need to calculate them every time.
3. TensorFlow.js has a ‘tf.tidy’ function that cleans up any Tensors defined within the function that are not returned. This saves you the task of manually disposing of each Tensor later on. Take note, the function you pass to the ‘tf.tidy’ function should **not be an**

asynchronous function, else it will not work as expected. If you have async code you will need to dispose of tensors manually in that code.

4. Define the normalization function as provided in the script.js normalization code:

script.js - Normalization Code

```
// Function to take a Tensor and normalize values
// with respect to each column of values contained in that Tensor.
function normalize(tensor, min, max) {
  const result = tf.tidy(function() {
    // Find the minimum value contained in the Tensor.
    const MIN_VALUES = min || tf.min(tensor, 0);

    // Find the maximum value contained in the Tensor.
    const MAX_VALUES = max || tf.max(tensor, 0);

    // Now subtract the MIN_VALUE from every value in the Tensor
    // And store the results in a new Tensor.
    const TENSOR_SUBTRACT_MIN_VALUE = tf.sub(tensor, MIN_VALUES);

    // Calculate the range size of possible values.
    const RANGE_SIZE = tf.sub(MAX_VALUES, MIN_VALUES);
    // Calculate the adjusted values divided by the range size as a new Tensor.
    const NORMALIZED_VALUES = tf.div(TENSOR_SUBTRACT_MIN_VALUE, RANGE_SIZE);

    return {NORMALIZED_VALUES, MIN_VALUES, MAX_VALUES};
  });
  return result;
}
```

Code explanation:

- Find the minimum value contained in the tensor:

```
const MIN_VALUES = min || tf.min(tensor, 0);
```

This line of code stores the ‘min’ value if a tensor of minimum value is passed, or it calculates the minimum value in the input tensor using the ‘tf.min’ function. Note that the ‘tf.min’ function takes two parameters. The first parameter is the tensor whose minimum value you wish to calculate, while the second parameter specifies the axis of the tensor from which the minimum value should be calculated. Since your input tensor is a 2d tensor, you need to calculate the minimum values for the

size (first column) and the number of bedrooms (second column) respectively. Passing axis 0 as the second argument instructs TensorFlow to find the minimum value for each column and not for each row or the tensor as a whole.

- Find the maximum value contained in the tensor.

```
const MAX_VALUES = max || tf.max(tensor, 0);
```

The ‘tf.max’ function is used to find the maximum value contained in a tensor instead.

- Subtract the minimum value from every value in the tensor and store the result in a new tensor. This new tensor now contains the adjusted values:

```
const TENSOR_SUBTRACT_MIN_VALUE = tf.sub(tensor,  
MIN_VALUES);
```

The ‘tf.sub’ function is used to subtract the minimum values tensor, which is a 1d tensor, from your input tensor, which is a 2d tensor.

- Calculate the range size of values in the original tensor by subtracting the maximum value from the minimum value for each feature.

```
const RANGE_SIZE = tf.sub(MAX_VALUES, MIN_VALUES);
```

- Divide the adjusted values by the range size and store the result in a new tensor. This tensor will have all values in the range 0 to 1.

```
const NORMALIZED_VALUES =
```

```
tf.div(TENSOR_SUBTRACT_MIN_VALUE, RANGE_SIZE);
```

- Return the normalized values, the minimum, and the maximum values. Storing the minimum and maximum values will save you time when you need to make new predictions.

```
return {NORMALIZED_VALUES, MIN_VALUES,  
MAX_VALUES};
```

Understanding ‘tf.min’ & ‘tf.max’

Tensor2D:

1st value represents
size of house,

2nd value represents
bedrooms

```
[  
  [550, 1],  
  [700, 1],  
  [1000, 2],  
  [1500, 3],  
]
```

Tensor1D:

1st value represents
Min size of house,

2nd value represents
Min bedrooms

$\xrightarrow{\text{tf.min(tensor, 0)}}$ [550, 1]

Figure 4.4.2.2. Understanding ‘tf.min’ and ‘tf.max’

In Figure 4.4.2.2, the ‘tf.min’ function is applied to a tensor 2d with the axis parameter specified to 0. This function produces a tensor 1d, with the first value being the smallest house (in the first column), and the second value being the lowest number of bedrooms (in the second column). The ‘tf.max’ function performs the same function but finds the maximum values instead.

Understanding ‘tf.sub’

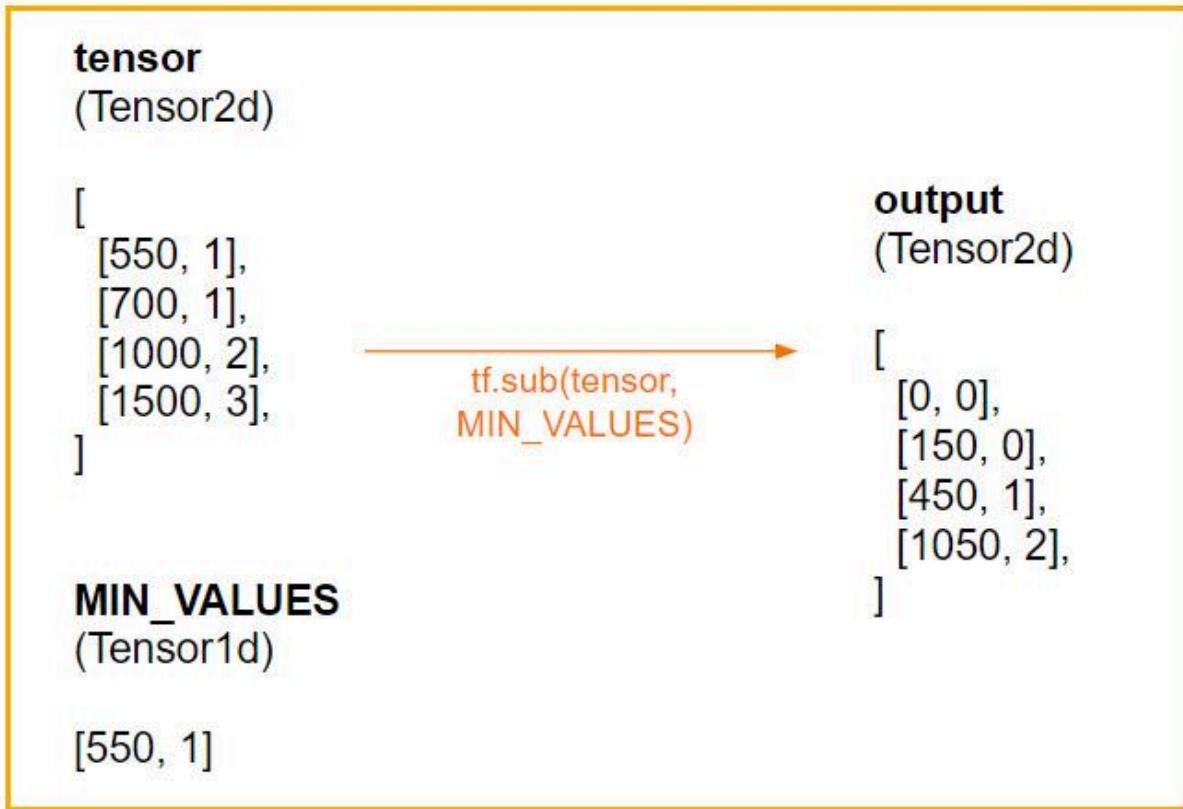


Figure 4.4.2.3. Understanding ‘tf.sub’

In Figure 4.4.2.3, the tensor [550, 1] is subtracted from an example 2D Tensor. Note that 550 is subtracted only from the first column of the tensor 2d, while 1 is subtracted only from the second column. The resulting numbers are populated in a new tensor 2d.

Use the Normalization Function

Next, call the normalize function on your inputs and store results in a constant called FEATURE_RESULTS which you can then print out to see the output and then dispose of the original INPUTS_TENSOR.

```

// Normalize all input feature arrays and then
// dispose of the original non normalized Tensors.
const FEATURE_RESULTS = normalize(INPUTS_TENSOR);
console.log('Normalized Values:');
FEATURE_RESULTS.NORMALIZED_VALUES.print();

console.log('Min Values:');
FEATURE_RESULTS.MIN_VALUES.print();

console.log('Max Values:');
FEATURE_RESULTS.MAX_VALUES.print();

INPUTS_TENSOR.dispose();

```

Code explanation:

Inspect the normalized tensor values that are returned to ensure there are no mistakes in the normalization function implementation. The console output shows only the first three and the last three entries of the normalized tensor as there are thousands of rows of data. Note that the normalized tensor's values for size and number of bedrooms are within the range 0 and 1.

```

// Normalize all input feature arrays and then
// dispose of the original non normalized Tensors.
const FEATURE_RESULTS = normalize(INPUTS_TENSOR);
console.log('Normalized Values:');
FEATURE_RESULTS.NORMALIZED_VALUES.print(); → Normalized Values:
Tensor
[[0.7700579, 1],
 [0.9255583, 1],
 [0.8833747, 1],
 ...,
 [0.0077199, 0],
 [0, 0],
 [0.0449407, 0]]

console.log('Min Values:');
FEATURE_RESULTS.MIN_VALUES.print(); → Min Values:
Tensor
[432, 0]

console.log('Max Values:');
FEATURE_RESULTS.MAX_VALUES.print(); → Max Values:
Tensor
[4059, 3]

INPUTS_TENSOR.dispose();

```

Figure 4.4.2.4. Inspect Normalized Tensor Values

Finally you clean up the tensors that were not in the ‘tf.tidy’ function and dispose of the INPUT_TENSOR.

In the next unit, you will use this pre-processed data and feed it into your custom model for predictions.

Model Creation and Training

Now that you have imported and pre-processed the data, it’s time to build and train the model.

Define the Model Architecture

1. You need to first call ‘tf.sequential’ to create a new sequential model skeleton to build upon. A sequential model indicates a model with layers that are executed one after the other. The outputs of one layer become the inputs of the next layer.

```
const model = tf.sequential();
```

2. Next, you add a layer to your model. Use ‘model.add’ to add a layer.

```
model.add(tf.layers.dense({inputShape: [2], units: 1}));
```

- a. The ‘dense’ term indicates that each neuron in the layer is fully connected to all the inputs.
- b. The ‘inputShape’ refers to the number of input features in each example in your data. In this case, it is two, since the inputs are house size and number of bedrooms.
- c. The ‘units’ refers to the number of neurons in the layer, which is one in this model. Since this layer is dense, the

neuron is connected to each of the two inputs. The neuron also allocates a weight randomly to each input to start.

- d. Note that you have not added an activation function in this model and so the output from the neuron is not modified in any way and instead just passes through.
3. You can print a summary of your model using 'model.summary()'. Observe that the model has three trainable parameters (two weights - one for house size and one for the number of bedrooms - and one bias). The summary also shows that the model produces an output with a single value.

```
// Now actually create and define model architecture.
const model = tf.sequential();

// We will use one dense layer with 1 neuron (units) and an input of
// 2 input feature values (representing house size and number of rooms)
model.add(tf.layers.dense({inputShape: [2], units: 1}));

model.summary(); →
train();
```

Layer (type)	Output shape	Param #
dense_Dense1 (Dense)	[null,1]	3
Total params: 3		
Trainable params: 3		
Non-trainable params: 0		

Figure 4.3.1 Linear Regression: Model Summary

4. Finally, you will call a train() function once the model is created, which you will define in the next unit.

Compile and Train the Model

1. The train function gets the model to adjust the weights and bias to learn from the input data. Start by defining an asynchronous function called train().

```
async function train() {...}
```

Training the model involves the following steps:

- a. **Define the learning rate:** This determines the magnitude of the change for the weights and bias. Set too high a number and you may overshoot the optimal values, and set too low a number, and the model may take too long to predict an answer.

```
const LEARNING_RATE = 0.01;
```

- b. **Compile the model:** During compilation, you define a few key parameters that the model uses while training. These include:

- i. *Optimizer:* This is a method used to adjust and find the optimal values for the weights and biases. In your model, you use the ‘tf.train.sgd’ function and pass the learning rate to this function. SGD stands for Stochastic Gradient Descent, which is a mathematical algorithm used to update the weights.

```
optimizer: tf.train.sgd(LEARNING_RATE),
```

- ii. *Loss:* The optimizer uses the values obtained from the ‘loss function’ and the learning rate to optimally adjust the weights and biases. For this model, use the Mean Squared Error loss function by just adding ‘meanSquaredError’ as a string to the loss property.

```
loss: 'meanSquaredError'
```

- c. **Train the model:** You need to call the ‘model.fit()’ function asynchronously to train your model on the input data. This call takes in three objects - the input tensor, an output tensor, and an optional object - to define a few additional parameters.

- i. Your input tensor is called FEATURE_RESULTS.NORMALIZED_VALUES.
- ii. Your output tensor is called OUTPUTS_TENSOR.
- iii. You can define the following additional parameters in the ‘model.fit’ function:
 1. validationSplit: This sets aside a portion of the training data that is not used in training but is used

after training is complete to see how well the model performs. The percentage of the data that is used for this split depends on the number of examples in your training data. In this case you can set it to 0.15, which represents 15%.

2. **shuffle**: This randomly shuffles the data in the input and output tensors each time the model goes through the dataset. This is to prevent the model from learning anything from the order in which the data is presented. Note that shuffling preserves the input-output correlation.
 3. **batchSize**: This parameter specifies the number of examples a model has to go through before it calculates the loss and optimizes the weights and biases. If you set the batch size as 1, the model will update the weights and biases for each example, a process that can take a long time.
 4. **epochs**: An epoch is completed when your model goes through all your training examples once. If you set the number of epochs as 10, the model goes through each of your training examples 10 times.
- d. **Dispose of Tensors**: Use ‘tf.dispose’ to clean up any remaining tensors.
 - e. **Print results of training**: You can access the ‘results.history.loss’ array and select the last item in the array to access last loss value record and then return its square root.

```
console.log("Average error loss: " +  
Math.sqrt(results.history.loss[results.history.loss.length - 1]));
```

You can likewise access the validation loss to see how well the model has performed on unseen data as well.

- f. **Evaluate Model:** Once you have completed training the model, you call the evaluate() function, which you will define in the next unit.

Batch Size and Epochs:

You have 2,000 rows of inputs in your dataset. If your model is training for 10 epochs, the model is learning from 20,000 examples. However, since your batch size is 64, the weights and biases are only updated each time 64 examples are processed. This results in the weights and biases being updated a total of 312 times (approximately 20,000/64). The optimal number of epochs and batch sizes is usually arrived at after some experimentation.

```
async function train() {
  const LEARNING_RATE = 0.01; // Choose a learning rate that's suitable for the data we are using.

  // Compile the model with the defined learning rate and specify a loss function to use.
  model.compile({
    optimizer: tf.train.sgd(LEARNING_RATE),
    loss: 'meanSquaredError'
  });

  // Finally do the training itself.
  let results = await model.fit(FEATURE_RESULTS.NORMALIZED_VALUES, OUTPUTS_TENSOR, {
    validationSplit: 0.15, // Take aside 15% of the data to use for validation testing.
    shuffle: true, // Ensure data is shuffled in case it was in an order
    batchSize: 64, // As we have a lot of training data, batch size is set to 64.
    epochs: 10 // Go over the data 10 times!
  });

  OUTPUTS_TENSOR.dispose();
  FEATURE_RESULTS.NORMALIZED_VALUES.dispose();

  console.log("Average error loss: " + Math.sqrt(results.history.loss[results.history.loss.length - 1]));
  console.log("Average validation error loss: " +
    Math.sqrt(results.history.val_loss[results.history.val_loss.length - 1]));

  evaluate(); // Once trained evaluate the model.
}
```

You should now see the following output on your console. Note that since the weights and biases are chosen randomly at the start, you will get different values for the loss each time you execute this code. One of the factors you need to evaluate while using a machine learning model is whether the model can predict better than existing solutions. If this is not

the case, you may need to explore new features or collect more data to improve your model.

```
Average error loss: 18325.97544470689
Average validation error loss: 35396.365011111378
```

Figure 4.3.2. Average error loss and Average validation error loss

Evaluate the Model

1. You now define an evaluate function to assess your model on data it has not seen before. Since you need to define new tensors in this function, wrap your prediction code within a ‘tf.tidy’ function.
2. To predict the house price for new input, you need to normalize the input using the minimum and maximum values that you have stored previously. In this case, the input example is a house of size 750 square feet with 1 bedroom.

```
let newInput = normalize(tf.tensor2d([[750, 1]]),
FEATURE_RESULTS.MIN_VALUES, FEATURE_RESULTS.MAX_VALUES);
```

3. Once you have normalized the new input, call the model.predict function to run inference and produce a new tensor as the output.

```
let output = model.predict(newInput.NORMALIZED_VALUES);
```

4. Once you get your prediction, dispose of all the tensors that still exist. You can also log the number of tensors left in memory to ensure you have disposed of all.

```
FEATURE_RESULTS.MIN_VALUES.dispose();
FEATURE_RESULTS.MAX_VALUES.dispose();
console.log(tf.memory().numTensors);
```

```
function evaluate() {
  // Predict answer for a single piece of data.
  tf.tidy(function() {
    let newInput = normalize(tf.tensor2d([[750, 1]]), FEATURE_RESULTS.MIN_VALUES, FEATURE_RESULTS.MAX_VALUES);

    let output = model.predict(newInput.NORMALIZED_VALUES);
    output.print();
  });

  // Finally when you no longer need to make any more predictions,
  // clean up the remaining Tensors.
  FEATURE_RESULTS.MIN_VALUES.dispose();
  FEATURE_RESULTS.MAX_VALUES.dispose();
  model.dispose();

  console.log(tf.memory().numTensors);
}
```

Store the Model

You may choose to store the model to your computer or to local storage in the browser.

Download model files to your computer:

```
await model.save('downloads://my-model');
```

Save to local storage for offline access!

```
await model.save('localStorage://demo/new modelName');
```

Figure 4.3.3. Download and Save Model

Finding the Limits of a Single Neuron

A model with a single neuron may accurately predict an output from inputs with single or even multiple features as long as there is a linear relationship between the inputs and the outputs. Many real-world problems, however, have inputs and outputs that are not linearly related. Datasets that have an exponential relationship between inputs and outputs cannot be approximated using a single straight line. In this and the subsequent

lesson, you will explore models that can deal with data in which there is a non-linear relationship between input and output as shown below.

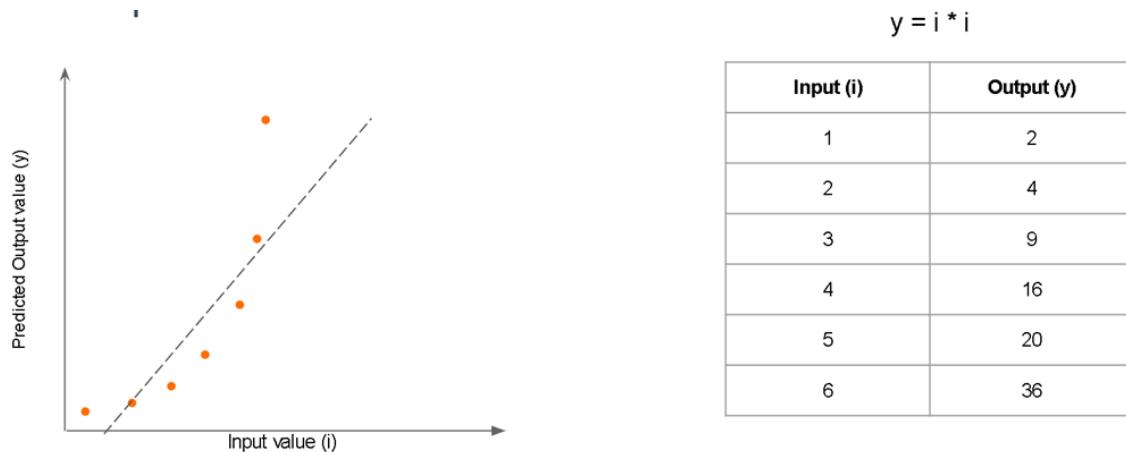


Figure 4.5.1.1. Scatter Plot showing an Exponential Relationship

1. El Problema: Linealidad vs. Complejidad

Una neurona artificial individual es, por definición, un **clasificador o regresor lineal**. Esto significa que su capacidad de predicción se limita a trazar una **línea recta** a través de los datos.

En el proyecto actual, estamos intentando predecir una relación exponencial ($y = x^2$):

- **Realidad:** Los datos forman una curva que asciende cada vez más rápido.
- **Intento de la neurona:** Intenta "ajustar" una línea recta lo más cerca posible de todos los puntos a la vez.

2. Hallazgos del Experimento (Análisis de Resultados)

Al ejecutar el código proporcionado, observarás lo siguiente:

- **Error Residual Alto:** El error promedio (Loss) se estanca en un valor alto (alrededor de 29 o más). Esto ocurre porque no existe ninguna línea recta en el universo que pueda tocar todos los puntos de una parábola.
- **Predictión Inexacta:** Cuando pedimos el cuadrado de 7, el modelo devuelve un valor erróneo (como 22 o 84) en lugar de **49**. El modelo

"adivina" un punto en su línea recta, pero esa línea está muy lejos de la curva real.

- **Estancamiento del Aprendizaje:** Aunque aumentes las épocas a 1000 o reduzcas el *Learning Rate*, el error dejará de bajar. Has alcanzado el **Límite de capacidad** del modelo.

3. Factores Críticos Identificados

1. **Arquitectura insuficiente:** Una sola neurona solo tiene un peso (w) y un sesgo (b). Solo puede aprender la pendiente y la altura de una línea ($y = mx + b$).
2. **Falta de Funciones de Activación No Lineales:** En una sola neurona de regresión, el resultado es puramente lineal. No hay nada que permita "doblar" la línea para convertirla en curva.
3. **Escasez de Datos:** Al entrenar con solo 20 ejemplos, el modelo tiene poca información, aunque en este caso el problema principal es la estructura, no la cantidad de datos.

4. Conclusión y Recomendación

El modelo actual sufre de **Underfitting** (subajuste) severo. No es que el modelo no haya "estudiado" lo suficiente; es que no tiene la "inteligencia" (capas) necesaria para entender el concepto de "elevar al cuadrado".

Propuesta de mejora:

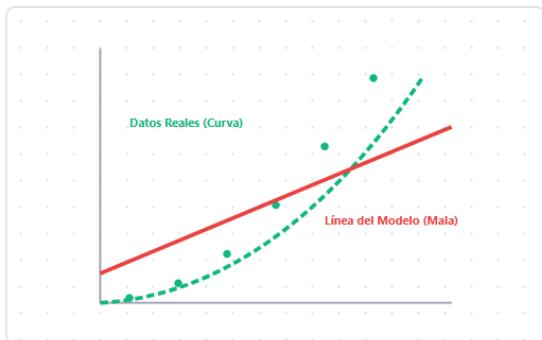
Para superar este límite, el siguiente paso técnico es evolucionar hacia un Perceptrón Multicapa (MLP) mediante:

- La adición de **Capas Ocultas** (Hidden Layers) con múltiples neuronas.
- El uso de la función de activación **ReLU**, que permite que la red neuronal aprenda a representar curvas mediante la combinación de múltiples segmentos lineales.

¿Por qué falla una sola neurona?

Visualización del límite matemático de la regresión lineal

Representación en el Plano Cartesiano



● Datos Reales

● Predicción Neurona

La Ecuación del Perceptrón

$$y = wx + b$$

Matemáticamente, esto es SIEMPRE una línea recta.

El Conflicto Lineal

Una sola neurona no tiene "bisagras". No puede doblarse para seguir la curva de los datos.

La Solución Oculta

Para arreglar esto, necesitamos añadir capas intermedias. Varias líneas rectas combinadas pueden imitar una curva.

Conclusión: En el gráfico, puedes ver que la línea roja intenta estar cerca de los puntos verdes, pero al ser una línea rígida, siempre habrá grandes espacios vacíos (**Error / Loss**). Por mucho que entrenes, el error nunca llegará a cero porque la arquitectura del modelo es físicamente incapaz de ser curva.

Resumen del Diagrama:

- Datos Reales (Puntos Verdes):** Representan la función $y = x^2$. Observa cómo se curvan hacia arriba exponencialmente.
- Línea del Modelo (Línea Roja):** Es el resultado de tu neurona única. Como solo conoce la fórmula $y = wx + b$, está obligada a ser recta.
- El "Gap" o Espacio:** La distancia vertical entre la línea roja y los puntos verdes es lo que llamamos **Error Cuadrático Medio (MSE)**. Debido a que la línea no puede doblarse, ese error siempre será muy grande.

Neural Networks for More Complex Non-Linear Data

Figure 4.5.2.1. depicts the model of a single neuron, which consists of a bunch of inputs that are multiplied by weights and summed together. A bias is then added to this sum, and the grand total is passed through an activation function to produce a usable output.

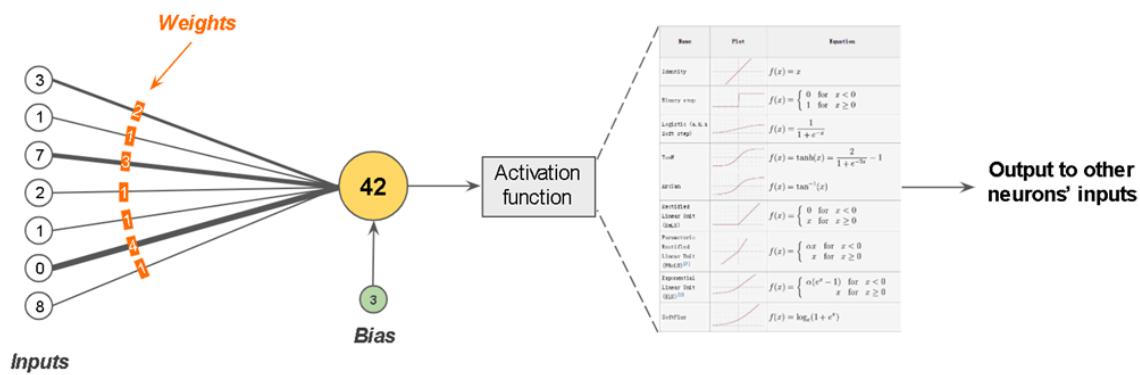


Figure 4.5.2.1. Complete Model of a Single Neuron

Observe the circle in Figure 4.5.2.2. Consider that this represents the complete neuron shown in Figure 4.5.2.1. A group of neurons arranged as shown in Figure 4.5.2.3. is called a layer of neurons.

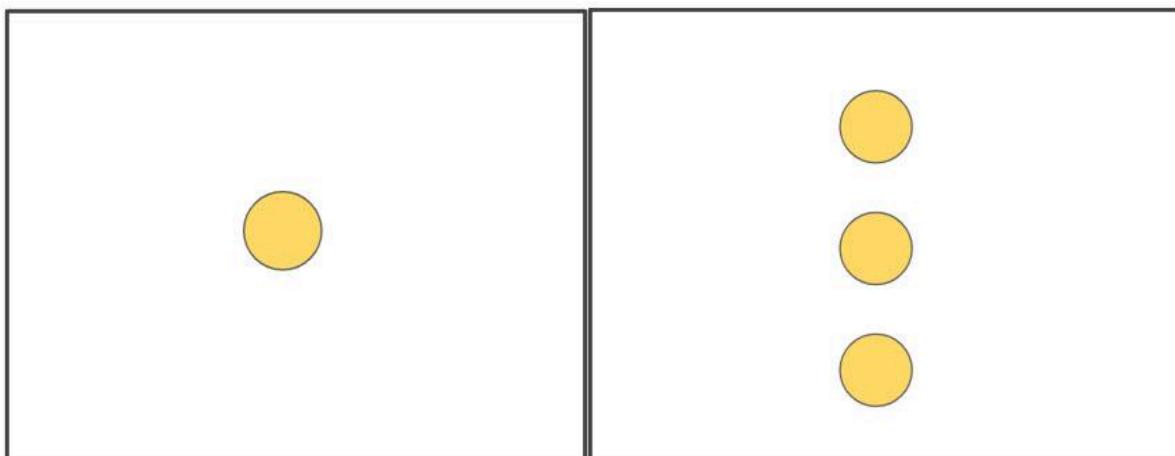
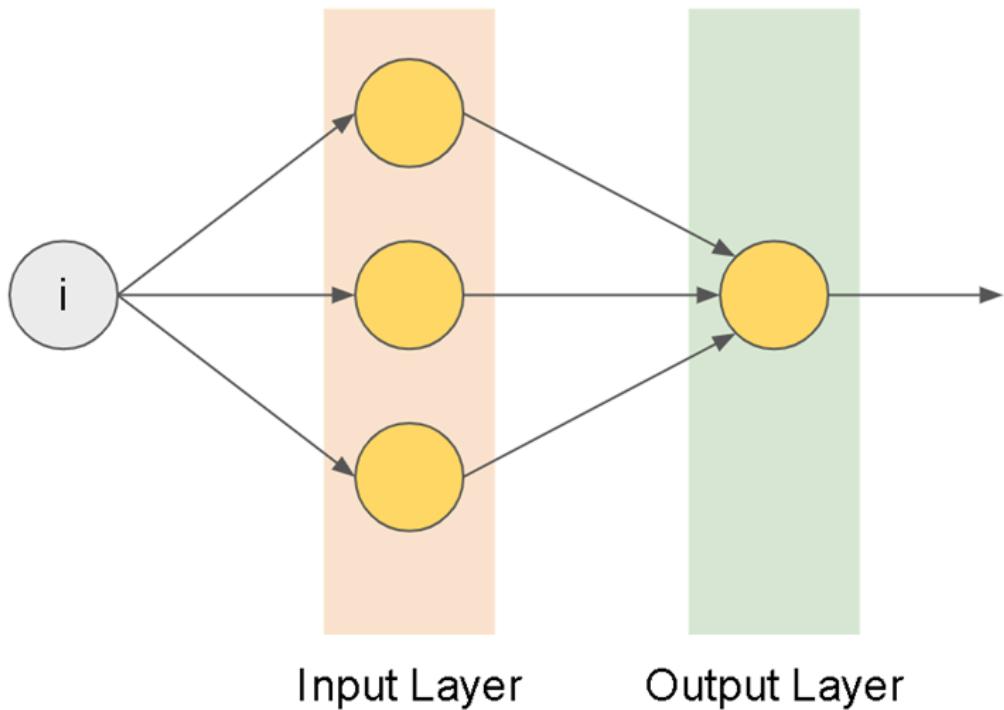


Figure 4.5.2.2.

Figure 4.5.2.3.

Figure 4.5.2.4. shows a neural network with a single input, an input layer with three neurons, each of which is connected to a single output neuron in a third layer. The output neuron then outputs a single number as needed.



4.5.2.4. Densely Connected Network with 2 Layers

In this session, you rebuild your model architecture using multiple layers like this and see how it performs.

Redes Neuronales para Datos No Lineales Complejos

Cuando nos enfrentamos a problemas del mundo real, como predecir el crecimiento exponencial o funciones cuadráticas ($y = x^2$), una sola neurona es insuficiente. Este documento explica cómo la arquitectura de

capas y las funciones de activación permiten a las máquinas modelar la complejidad.

1. El Límite de la Linealidad

Un solo perceptrón es una herramienta lineal. Por muchas épocas que entrene, siempre intentará ajustar una línea recta a los datos. Si los datos son curvos, el error (Loss) será permanentemente alto.

La Solución: Capas Ocultas (Hidden Layers)

Al agrupar neuronas en capas y conectar varias capas entre sí, creamos una **Red Neuronal Densamente Conectada**. Cada capa extrae características más abstractas y permite "doblar" la capacidad de representación del modelo.

2. Evolución de la Arquitectura

A. El modelo de 2 capas (Relación Lineal Persistente)

Iniciamos agregando una capa con 3 neuronas conectada a una salida.

```
const model = tf.sequential();

// Capa oculta con 3 neuronas

model.add(tf.layers.dense({inputShape: [1], units: 3}));

// Capa de salida

model.add(tf.layers.dense({units: 1}));
```

Problema: Aunque hay más neuronas, si no hay una función de activación, la combinación de dos funciones lineales sigue siendo **lineal**. El modelo sigue prediciendo una línea recta.

B. La Magia de la Función ReLU

Para romper la linealidad, introducimos **ReLU** (Rectified Linear Unit). Esta función permite que cada neurona se active solo en ciertos rangos, permitiendo que la red combine diferentes segmentos de líneas para imitar una curva.

```
model.add(tf.layers.dense({inputShape: [1], units: 20, activation: 'relu'}));
```

```
model.add(tf.layers.dense({units: 1}));
```

Visualización de ReLU combinando líneas:

(Cada neurona ReLU aporta una "pieza" de la curva final).

3. Parámetros Aprendibles: ¿Cómo "piensa" la red?

A medida que añadimos neuronas, los parámetros aumentan drásticamente. Tomemos el modelo de **dos capas ocultas de 100 neuronas**:

Capa	Conexiones (Weights)	Sesgos (Bias)	Total Parámetros
Entrada a Oculta 1	1 entrada * 100 neuronas = 100	100	200

Oculta 1 a Oculta 2	100 neuronas * 100 neuronas = 10,000	100	10,100
Oculta 2 a Salida	100 neuronas * 1 salida = 100	1	101
TOTAL			10,401

Este gran número de parámetros permite al modelo tener una "memoria" mucho más fina de la curva de los datos.

4. Ajustes Críticos en el Entrenamiento

El problema del NaN (Not a Number)

Al aumentar el número de neuronas, un **Learning Rate** de **0.01** suele ser demasiado alto. El modelo hace demasiados cambios pequeños a la vez, lo que causa que los pesos exploten matemáticamente.

- **Solución:** Reducir el Learning Rate a **0.001** o **0.0001** para estabilizar el aprendizaje.

Eficiencia Computacional

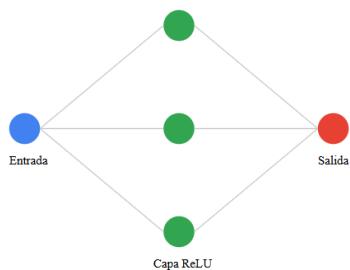
No siempre "más es mejor". Como muestra la tabla, una arquitectura de **31 neuronas (186 parámetros)** logra un error casi tan bajo como una de **10,401 parámetros**, pero siendo mucho más rápida y ligera para dispositivos móviles.

Capas	Neuronas Totales	Parámetros	Error (Loss)
1	1	2	30.4

2	21	61	4.7
3	31	186	2.18 (Óptimo)
3	201	10,401	1.4

5. Conclusión Visual de la Red Completa

El modelo final que hemos construido se visualiza como una red donde cada entrada se ramifica hacia múltiples neuronas ReLU, cuyas señales se filtran y combinan para entregar una predicción precisa de una función no lineal.



***Nota final:** El modelo multicapa es capaz de aprender Sx^2S dentro del rango de entrenamiento (1-20), pero tendrá dificultades para predecir números muy lejanos (ej. 100) debido a que nunca ha visto la escala de esos datos.

Comprendiendo la Complejidad: De la Línea Recta a la Curva Inteligente

Para entender por qué pasamos de una sola neurona a redes con cientos de parámetros, debemos mirar el proceso como la construcción de un cerebro artificial.

1. El Límite de la Neurona Única (La "Regla Rígida")

Imagina que intentas dibujar el contorno de una pelota usando solo una regla. No importa cuánto lo intentes, siempre obtendrás una línea recta.

- **Concepto:** Una sola neurona solo conoce la fórmula $y = wx + b$.
Es una herramienta lineal.
- **Falla:** Si los datos suben de forma curva (como el precio de una casa que se dispara exponencialmente), la línea recta de la neurona quedará muy lejos de la realidad. Esto produce un **Error (Loss)** permanentemente alto.

2. La Magia de las Capas Ocultas (La "Colaboración")

Para resolver el problema de la regla rígida, añadimos más neuronas en niveles llamados **Capas Ocultas**.

- **¿Qué hacen?:** En lugar de que un solo "juez" decida todo, creamos un comité.
- **El Problema Matemático:** Si solo sumamos neuronas sin nada más, el resultado sigue siendo una línea recta (una suma de líneas es otra línea). Aquí es donde entra el ingrediente secreto.

3. ReLU: El "Doblador" de Líneas

La función de activación **ReLU** es lo que permite que la red neuronal "se doble".

- **Cómo funciona:** ReLU dice: "Si el valor es negativo, ignóralo (0). Si es positivo, déjalo pasar".
- **El Resultado:** Al combinar muchas neuronas con ReLU, la red une pequeños segmentos de líneas rectas en diferentes ángulos. Vistas

desde lejos, estas pequeñas líneas unidas forman una **curva perfecta**.

4. La Anatomía del Aprendizaje: Pesos y Sesgos

Cuando el texto menciona los **10,401 parámetros**, se refiere a la capacidad de "detalle" del modelo.

- **Weights (Pesos):** Son las conexiones. En una red densa, cada neurona de una capa habla con todas las de la siguiente. 100 neuronas hablando con otras 100 generan 10,000 conexiones.
- **Bias (Sesgos):** Es la inclinación individual de cada neurona.
- **Conclusión:** A más parámetros, el modelo tiene más "pinceles" para pintar la curva, pero también es más pesado y lento.

5. El Problema del NaN y el Learning Rate

Cuando tienes miles de neuronas (miles de parámetros), el modelo se vuelve muy sensible.

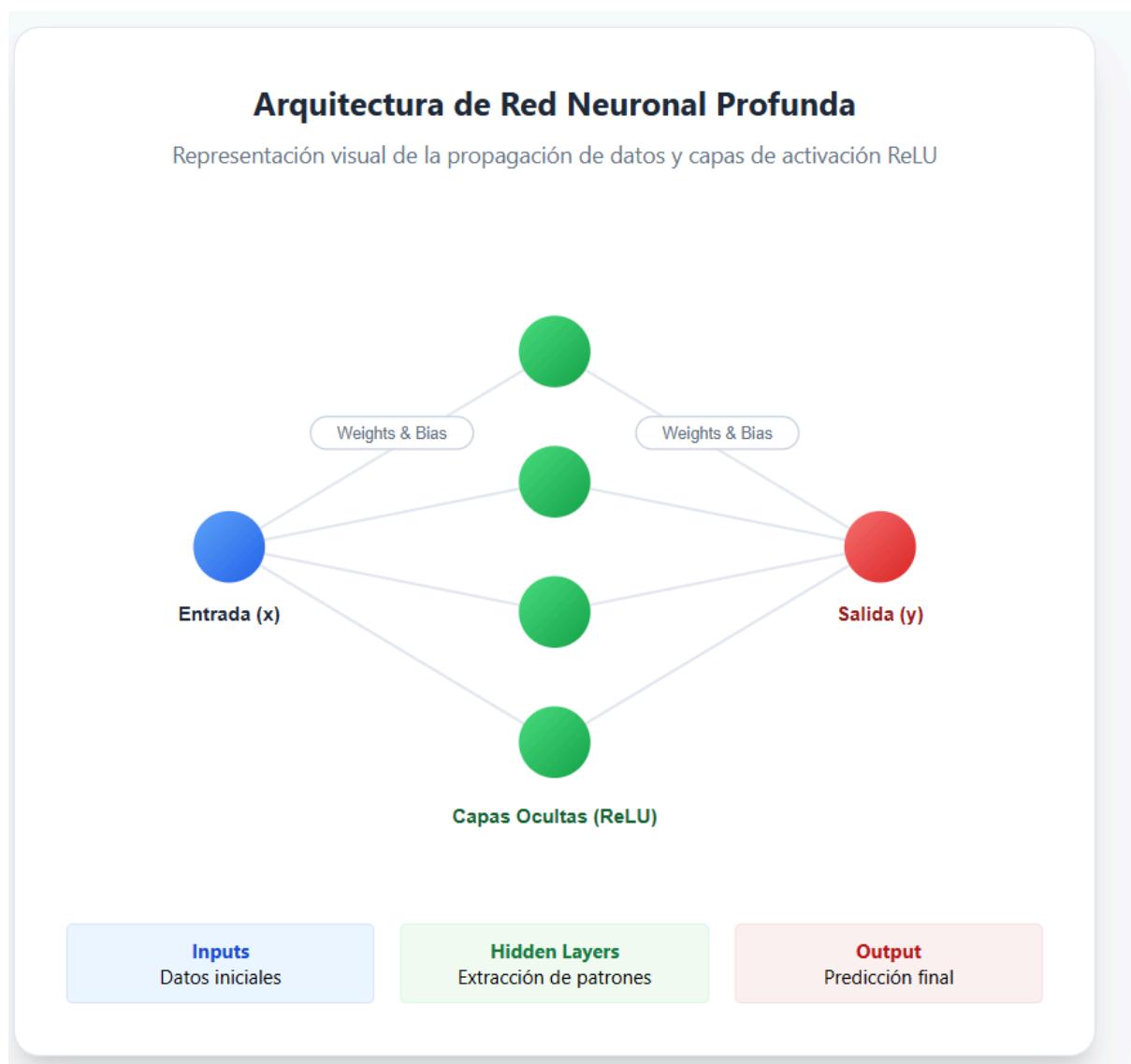
- **El Riesgo:** Si el **Learning Rate** es muy alto (0.01), los ajustes son tan bruscos que los números crecen hasta el infinito, resultando en **NaN** (Not a Number / Error matemático).
- **La Solución:** Pasos más pequeños. Un Learning Rate de **0.0001** permite que el "comité" de 10,000 parámetros se ponga de acuerdo sin causar un caos matemático.

Resumen del Éxito: El Modelo Óptimo

Según tus datos, el mejor modelo no es el más grande, sino el más equilibrado:

- **Arquitectura:** 3 capas (Entrada, Oculta 1, Oculta 2, Salida).
- **Neuronas:** 25 y 5.
- **Parámetros:** 186.
- **Resultado:** Logra un error de **2.18**, casi tan bueno como el modelo gigante de 10,000 parámetros, pero es mucho más eficiente para dispositivos móviles.

Visualización de la Estructura (Diagrama SVG)



Multi-Layer Perceptrons for Classification

In previous lessons, you performed linear regression with a single neuron to predict an output number given some inputs. You also used multiple layers of neurons with activation functions to represent non-linear data.

In this lesson, you will explore classification, which involves predicting what ‘class’ an input feature represents instead of some continuous numerical output value.

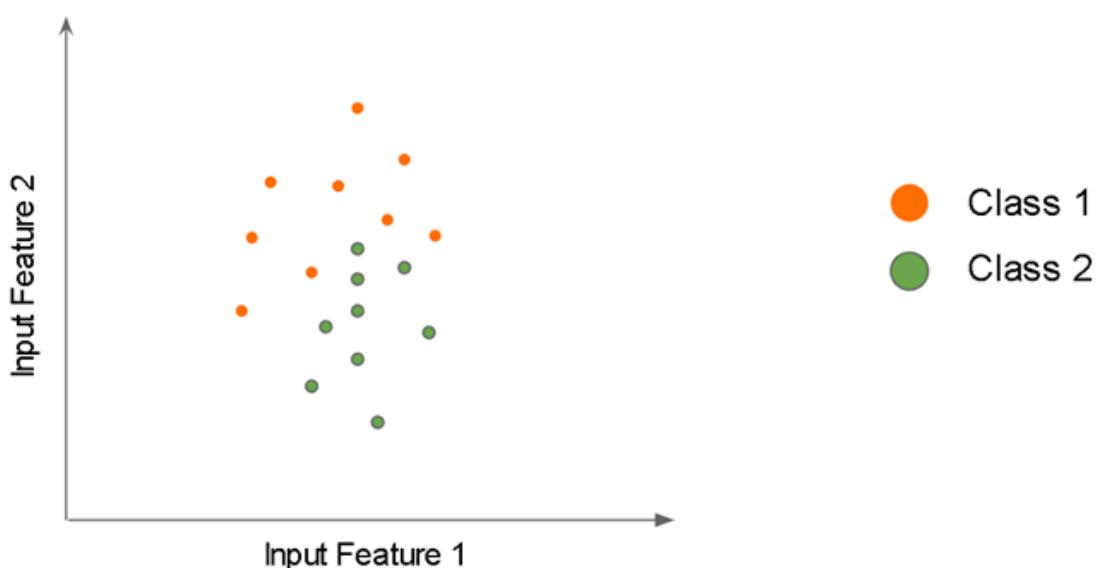


Figure 4.6.1.1. Classification Scatter Plot

In Figure 4.6.1.1., the orange and green data points are classes plotted using Input Feature 1 and Input Feature 2. These classes could be two different types of flowers and the input features could be the number of petals and stem length as an example.

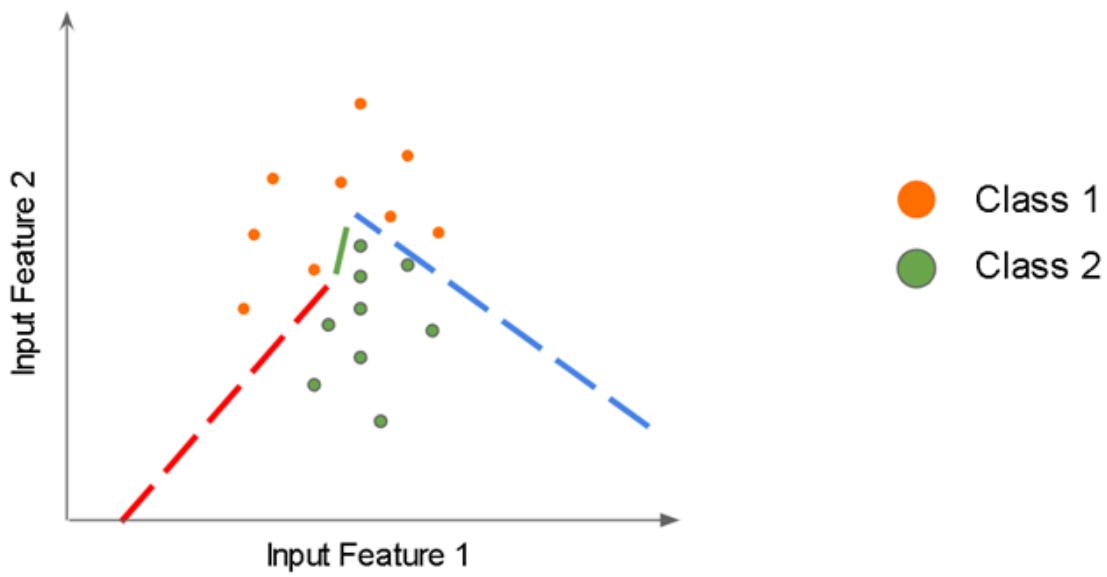


Figure 4.6.1.2. Classification Scatter Plot 2

For regression, you combined layers of neurons to generate multiple lines of best fit. These lines were used to predict output numbers based on input features. In classification, the line is used to separate one class from another instead as shown.

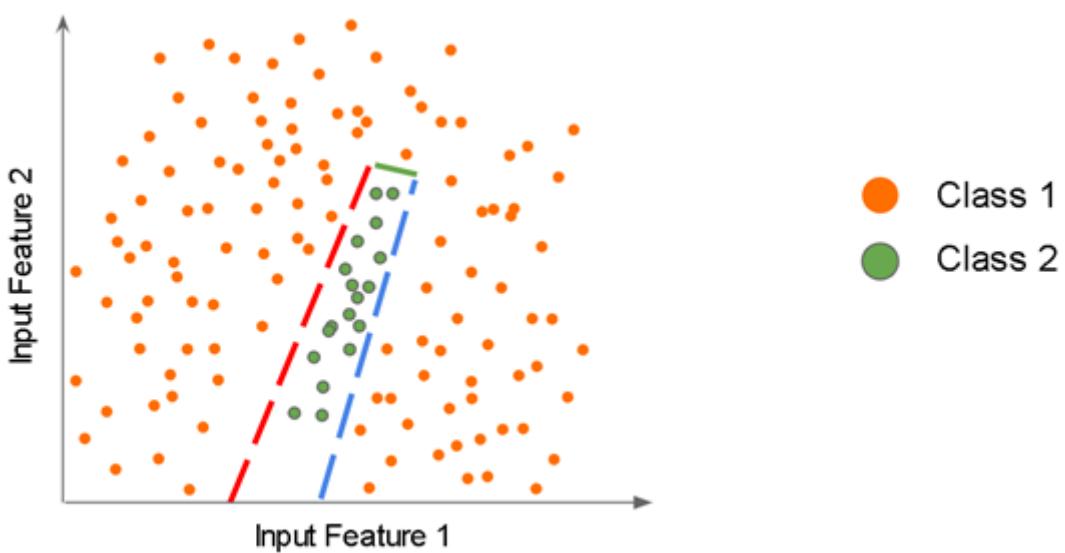


Figure 4.6.1.3. Classification Scatter Plot 3

With enough neurons and layers, you can potentially train a system to learn complex and intricate classifications.

In Figure 4.6.1.3., recordings that are contained within the lines belong to Class 2, while recordings outside the lines belong to Class 1.

But what if your input data consists of images? You can visualize an image as a graph with an example point for every x and y value within some given range as shown in figure 4.6.1.4 below.

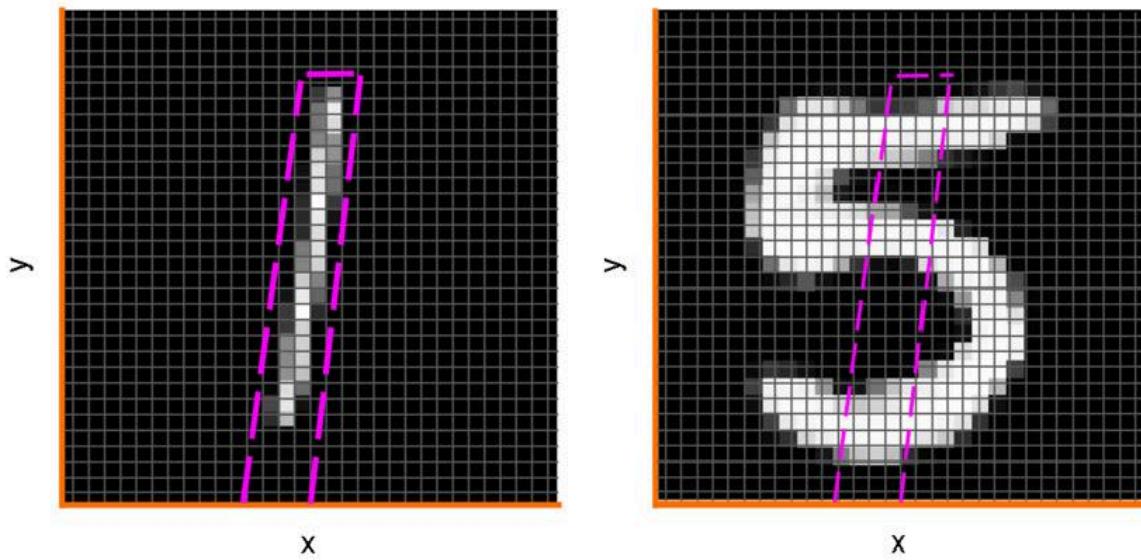


Figure 4.6.1.4. Grayscale Images of Handwritten Digits

The first image in Figure 4.6.1.4 is a magnified grayscale image with the number 1 drawn on it in white. Each pixel is represented by a colored square on the graph. Observe the three bounding lines that contain all the non-black pixels in this image. If you classify a new image and find that all

non-black pixels are contained within the bounds of the three lines, you may conclude that the image represents a drawing of the number 1.

However, if the new image has the number 5, as seen in the second image in Figure 4.6.1.4, you will find that only 30% of non-black pixels are contained within the bounds of the three lines, and you can conclude that this image does not represent a drawing of the number 1. Using neurons to look for features in different parts of images, you can train a system to recognize handwritten digits from 0 to 9.

Developing a model to recognize images of handwritten digits

Steps:

1. Transforming Images into Tensors

Figure 4.6.1.5. is a 28x28 grayscale image. It is essentially a matrix of numbers ranging from 0 to 255, with 0 representing pure black and 255 representing pure white.

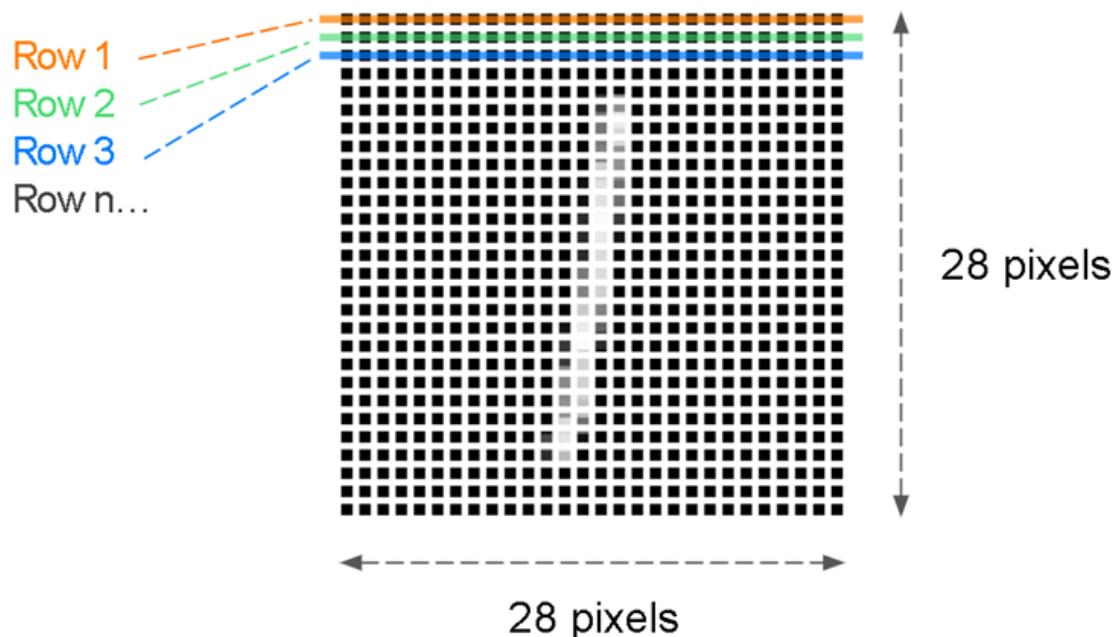


Figure 4.6.1.5. Grayscale Image of Handwritten Digit 1

All the rows of data in the image can be transformed into a 1-dimensional array of 784 values. This array will form the input for your machine learning model!

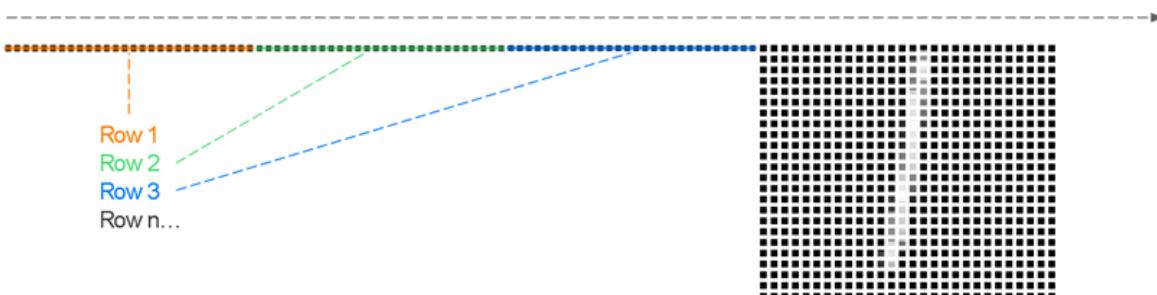


Figure 4.6.1.6. Rows being transformed into Array of 784 Values

2. Building a Neural Network: Multi-layered Perceptrons

- Consider that you fed in a 28 by 28px image of the number nine to your model. Your input data is a 1-dimensional array with 784 values (See Figure 4.6.1.7.) as shown below.

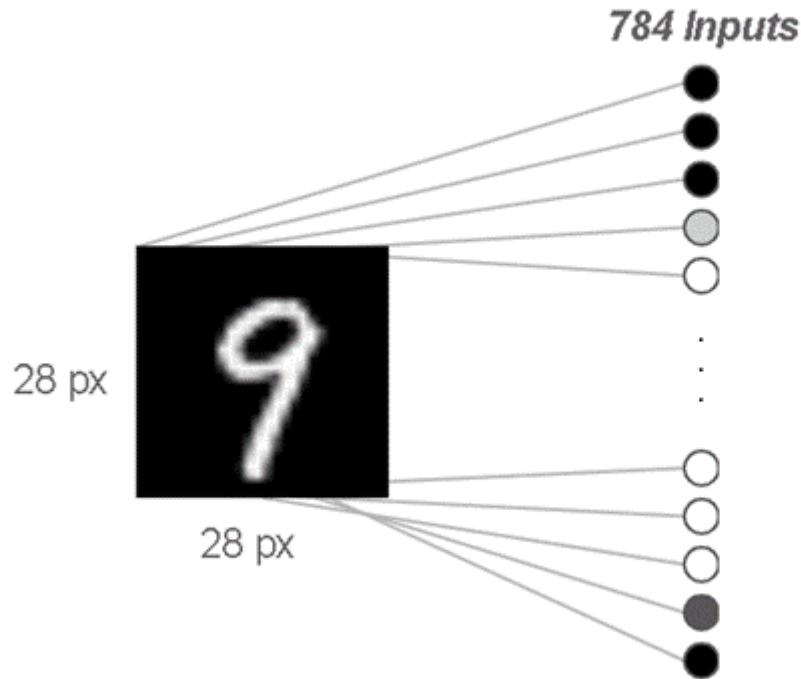


Figure 4.6.1.7. Input Array

- b. You can then add a fully connected dense layer with seven neurons (see Figure 4.6.1.8). Since this is a dense layer, each neuron receives input from each of the 784 input values, thus creating a total of 5,488 connections. This is the input layer that samples the image pixel values.

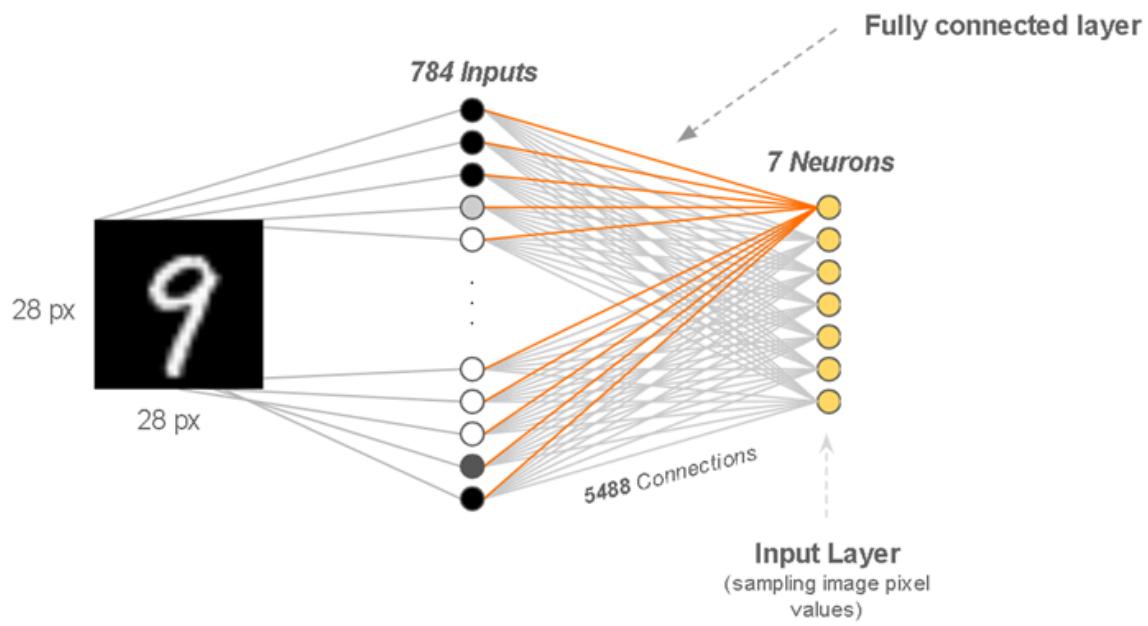


Figure 4.6.1.8. Input Layer with Seven Neurons

- c. You can then add another dense layer with five neurons (see Figure 4.6.1.9.). Each neuron in this layer is connected to every neuron in the input layer, thus creating a total of 35 more connections. This forms the hidden layer in your neural network.

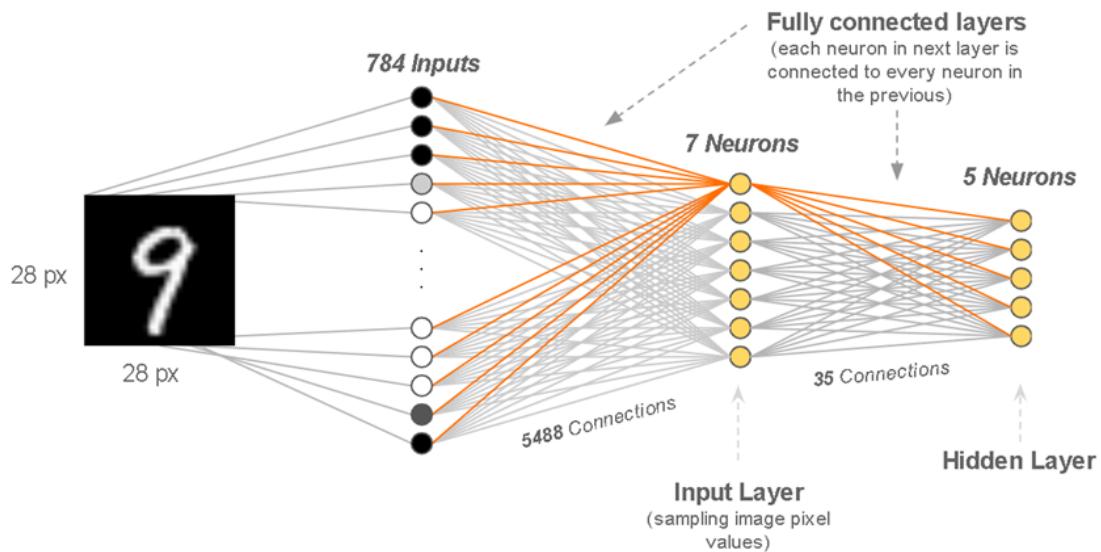


Figure 4.6.1.9 -Hidden Layer with Five Neurons

- d. Finally, you can add an output layer with 10 neurons. This layer samples the outputs of the hidden layer, creating 50 additional connections. Each neuron's output number in the output layer

represents a vote for each of the 10 possible digits (0 to 9) which is why the output layer must contain 10 neurons.

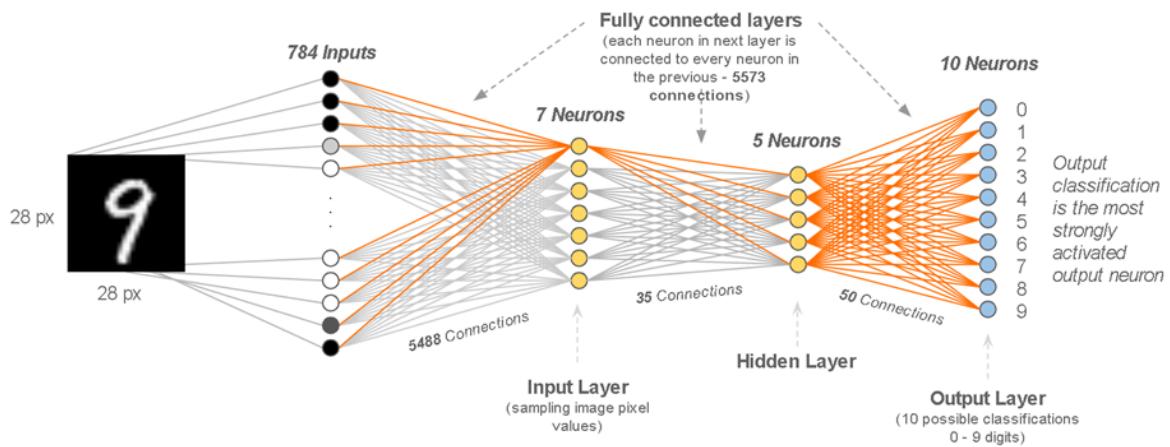


Figure 4.6.1.10. Output Layer with 10 Neurons

Note: Observe that your model has 5,573 connections. The model adjusts the weights for these connections (and the additional biases) thousands of times during the training process to figure out which pixels need higher weights and which pixels do not. Since the weights are initially random, the model will not perform well at the beginning but will adjust its predictions based on how wrong its guesses were.

In the next unit, you will explore how the model maps each output to a digit.

-Hot Encoding:

1-Hot Encoding is a technique used when dealing with representation of categorical data. It is essentially a 1-dimensional array of numbers where all numbers are 0, apart from the position that represents the class of interest, which is a 1.

For instance, the number 4 can be represented through 1-Hot Encoding as follows:

$$4 = [0, 0, 0, 0, 1, 0, 0, 0, 0]$$

Observe that the '1' is the fifth element since the first element represents 0 and the last element represents 9.

1-Hot Encoding for the digits 0 to 9

0 == [1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
1 == [0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
2 == [0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
3 == [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
4 == [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
5 == [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
6 == [0, 0, 0, 0, 0, 0, 1, 0, 0, 0]
7 == [0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
8 == [0, 0, 0, 0, 0, 0, 0, 0, 1, 0]
9 == [0, 0, 0, 0, 0, 0, 0, 0, 0, 1]

Figure 4.6.1.11. Hot Encoding for Handwritten Digits

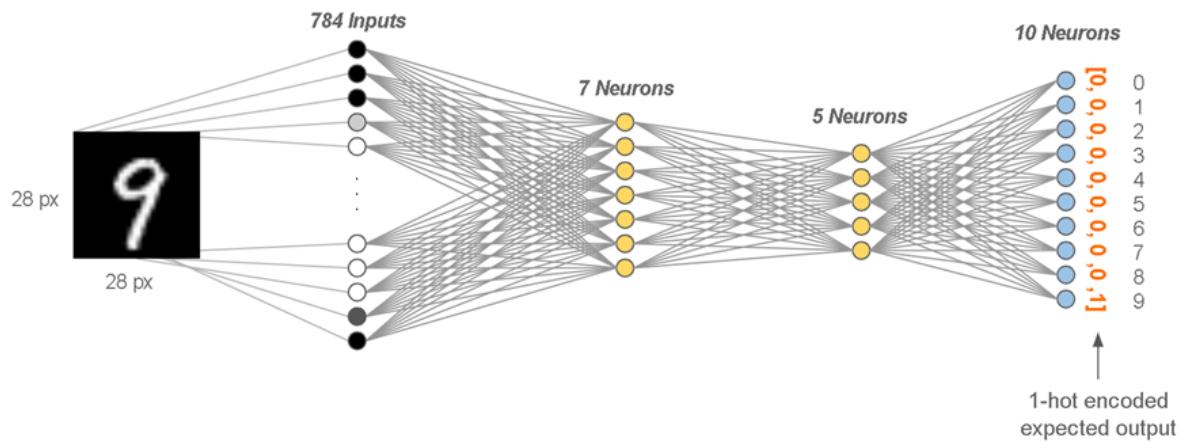


Figure 4.6.1.12. Hot Encoded Expected Output

Once your model is trained, when you feed in the number nine as the input, the model should now produce an output where the last element in the output tensor is close to one and the others, close to zero. This indicates that the network has recognized the digit 9!

Softmax Activation:

- Softmax is a special activation function used to perform classification.
- This function produces output numbers that add up to one. Note that this holds true regardless of the number of outputs used.
- The function works well only if the outputs you are trying to predict are mutually exclusive.

Categorical Crossentropy

- Categorical Crossentropy is a new loss function that is used for multi-class classification.
- This loss function is suited to classification tasks involving three or more classes.

```
model.compile({  
    optimizer: 'sgd',  
  
    loss: 'categoricalCrossentropy',  
  
});
```

- If your model needs to predict one of only two classes, the 'binaryCrossentropy' loss function can be used instead.

Training Data:

The MNIST training dataset contains hundreds of handwritten digits that can be used to train a deep neural network to learn to recognize handwritten digits.

In the next lesson, you will use the MNIST data (that has been normalized and transformed into a JavaScript friendly array form) to create an image classifier.

Entendiendo la Clasificación con MLP 🧠

Hasta ahora, tus redes neuronales predecían un número continuo (como el precio de una casa). Ahora, aprenderás a **clasificar**, que es como enseñarle a la IA a separar objetos en diferentes "cajas" o etiquetas.

1. Regresión vs. Clasificación: La Gran Diferencia

- **Regresión:** "¿A qué temperatura estamos?". La respuesta es un número infinito de posibilidades (22.5, 23, 24.1...).
- **Clasificación:** "¿Hace frío o calor?". La respuesta es una etiqueta de una lista cerrada (Caja A o Caja B).

2. El "Truco" de la Imagen (Flattening)

Una imagen de un número (como las de MNIST) es un cuadrado de 28 times 28 píxeles. Para que la red neuronal pueda leerla, hacemos un proceso llamado **aplanado (flattening)**:

1. Tomamos las filas de píxeles.
2. Las ponemos una al lado de la otra en una sola fila larga.

3. Resultado: Una lista de **784 números** ($28 \times 28 = 784$). Cada número representa qué tan blanco o negro es ese punto.

3. La Arquitectura: De Píxeles a Votos

Imagina que la red es un comité de expertos:

- **Capa de Entrada (784 neuronas):** Cada neurona mira un solo píxel de la imagen.
- **Capas Ocultas:** Buscan patrones. Una neurona puede especializarse en detectar "líneas verticales", otra en "curvas cerradas".
- **Capa de Salida (10 neuronas):** Aquí es donde ocurre la magia. Como queremos reconocer dígitos del 0 al 9, ponemos **10 neuronas**. Cada una representa un número.

4. Las Herramientas del Clasificador

A. One-Hot Encoding (El idioma de las etiquetas)

La IA no entiende qué es el concepto "Cuatro". Ella entiende listas de números. Por eso usamos **One-Hot Encoding**:

- En lugar de decirle **4**, le damos una lista: [0, 0, 0, 0, 1, 0, 0, 0, 0].
- Solo hay un **1** (caliente/hot) en la posición del número correcto. El resto son **0** (fríos).

B. Activación Softmax (La voz de la probabilidad)

Al final de la red, las 10 neuronas de salida darán números raros (ej: 5.2, 0.1, -1.2). La función **Softmax** transforma esos números en **probabilidades** que suman **100%**:

- Neurona del "4": 92% de probabilidad.
- Neurona del "9": 5% de probabilidad.
- Otras: 3% restante.
- **Resultado:** La IA dice "Estoy un 92% segura de que esto es un 4".

C. Categorical Crossentropy (El Juez)

Es la nueva función de pérdida (Loss). A diferencia del MSE, esta función es experta en comparar distribuciones de probabilidad. Castiga muy fuerte a la red si está "muy segura" de una respuesta que es incorrecta.

Resumen del Proceso

1. **Entrada:** Entran 784 píxeles.
2. **Proceso:** Las capas ocultas detectan formas.
3. **Salida:** 10 neuronas "votan" por un número.
4. **Decisión:** La función **Softmax** elige el ganador con el porcentaje más alto.
5. **Ajuste:** La **Categorical Crossentropy** le dice a la red cuánto se equivocó para que ajuste sus pesos con el optimizador (Adam o SGD).

Convolutional Neural Networks

The deep neural network you built in the previous session can classify images that:

- Contained a square shape (28 x 28 pixels)
- Were centered
- Contained pixels in grayscale

Using these images, you could classify handwritten digits.



Figure 4.7.1.1. Centered Digit



Figure 4.7.1.2. Off-Centered Digit

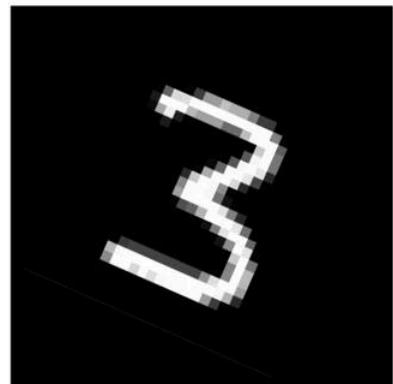


Figure 4.7.1.3. Crooked Digit

Your model used a multi-layered perceptron that sampled all the pixels in the input image and adjusted the weights during training, based on specific pixel values at different positions. As a result, it associated certain features with specific locations in the image. This approach will work with images that are resized, cropped, and centered, but may not succeed with real-life data. For instance, a model that classifies Figure 4.7.1.1. as the number three may fail to classify Figures 4.7.1.2. and 4.7.1.3. as three since the digits are either off-center or rotated slightly. This is especially true if the model has seen only centered images during training.

Convolutional Neural Networks (CNNs)

CNNs are networks that can work with larger images and can recognize objects that need not be in the same location, or may be even scaled or rotated! CNNs use special layers that are fed into a multi-layer perceptron at the end to perform final classification so builds upon your existing knowledge.

The initial layers essentially look for simple features, such as lines and curves, while deeper layers use these features to detect shapes. Further layers use these shapes to recognize objects and so on. The deeper you go the more complex things it can recognize.

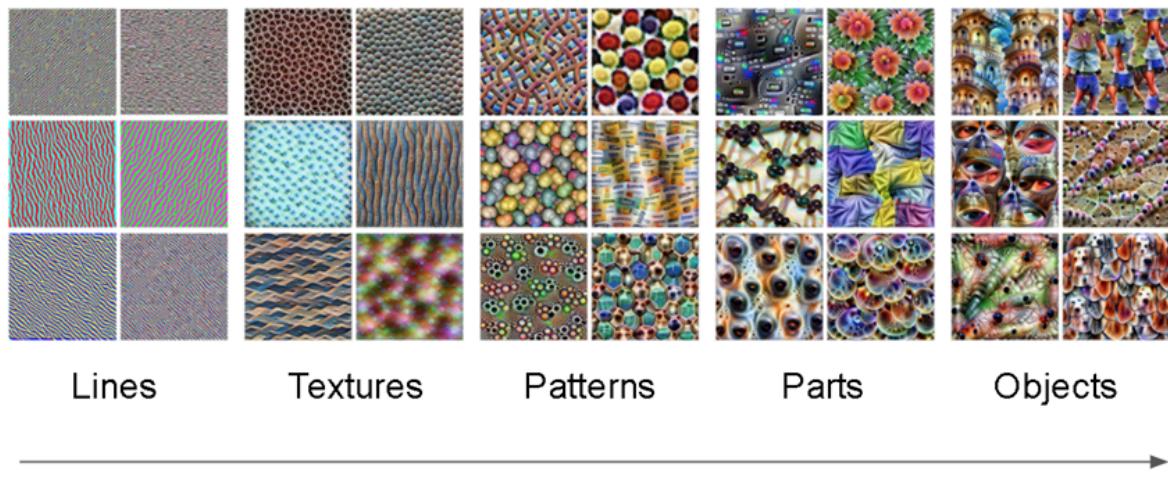


Figure 4.7.1.4. Visualizing Layers of a Complex CNN

CNNs use a specialized form of weights called ‘filters’ or ‘kernels’ to classify images. Let’s see what these are and how they are used.

Filters/Kernels

Consider Figures 4.7.1.5. and 4.7.1.6. These are 7×7 pixels grayscale images showing the number seven at different locations.

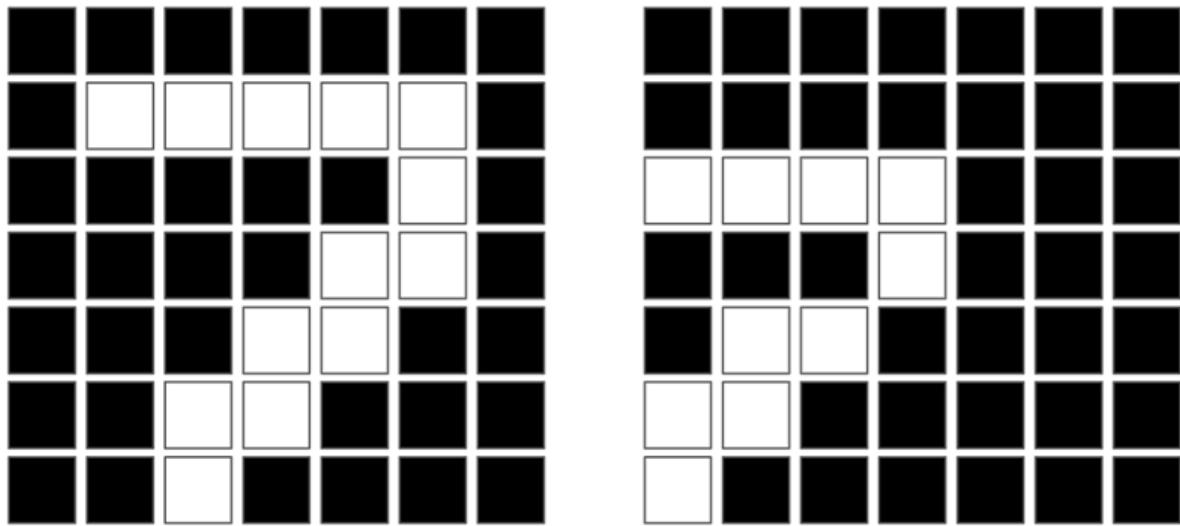


Figure 4.7.1.5. 7×7 Pixels Images of Seven

Assuming that black is represented by '0' and white is represented by '1', you can also visualize the two images as follows:

0	0	0	0	0	0	0
0	1	1	1	1	1	0
0	0	0	0	0	1	0
0	0	0	0	1	1	0
0	0	0	1	1	0	0
0	0	1	1	0	0	0
0	0	1	0	0	0	0

0	0	0	0	0	0	0
0	0	0	0	0	0	0
1	1	1	1	1	0	0
0	0	0	1	0	0	0
0	1	1	0	0	0	0
1	1	0	0	0	0	0
1	0	0	0	0	0	0

Figure 4.7.1.6. 7 by 7 Pixel Images of with Pixel Values

Note that if you overlay the images (see Figure 4.7.1.7.), there is no overlap of white pixels. For a computer, these are two very different images.

0	0	0	0	0	0	0
0	1	1	1	1	1	0
1	1	1	1	0	1	0
0	0	0	1	1	1	0
0	1	1	1	1	0	0
1	1	1	1	0	0	0
1	0	1	0	0	0	0

Figure 4.7.1.7 - 7x7 Pixels Image with Two Overlaid Sevens

Instead of sampling the whole image, CNNs attempt to match pieces of the image. For instance, in Figure 4.7.1.8., the patterns of pixels surrounded by orange squares in both images match. Likewise, the pixel patterns surrounded by the green squares in each image are identical, as are those

surrounded by the pink squares . CNNs find these important features using ‘filters’ or ‘kernels’.

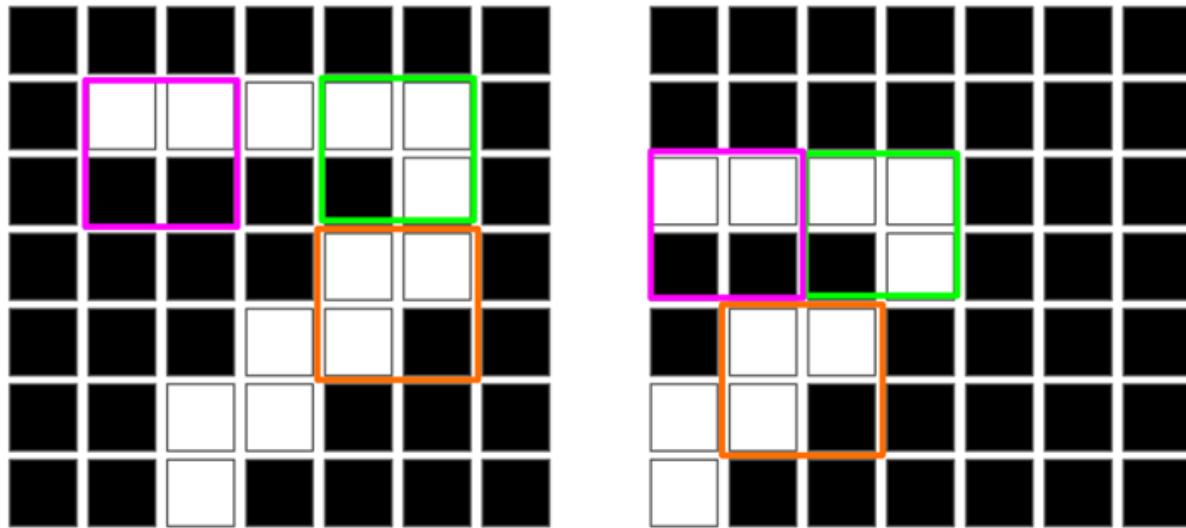


Figure 4.7.1.8 - Identifying Feature Using Filters

Filters or kernels are multi-dimensional arrays of numbers (2x2, 3x3, etc.). They may seem similar to tiny images, but with each pixel having a carefully calculated value to help it find useful things in the image. As the numbers can contain numbers like “-1” they are not images per se so you should think of them as a matrix of numbers instead. In Figure 4.7.1.9., filter 1 will detect diagonal lines, whereas filter 2 will detect horizontal lines. Observe that the pattern in the filters, when placed on specific parts of the image, can correlate more or less depending which part of the image it is sampling.

The values in these filters are initially randomly chosen, like the weights in the neural networks you covered previously. The values will be adjusted over time during training as the model figures out the best features with which to describe objects in the image.

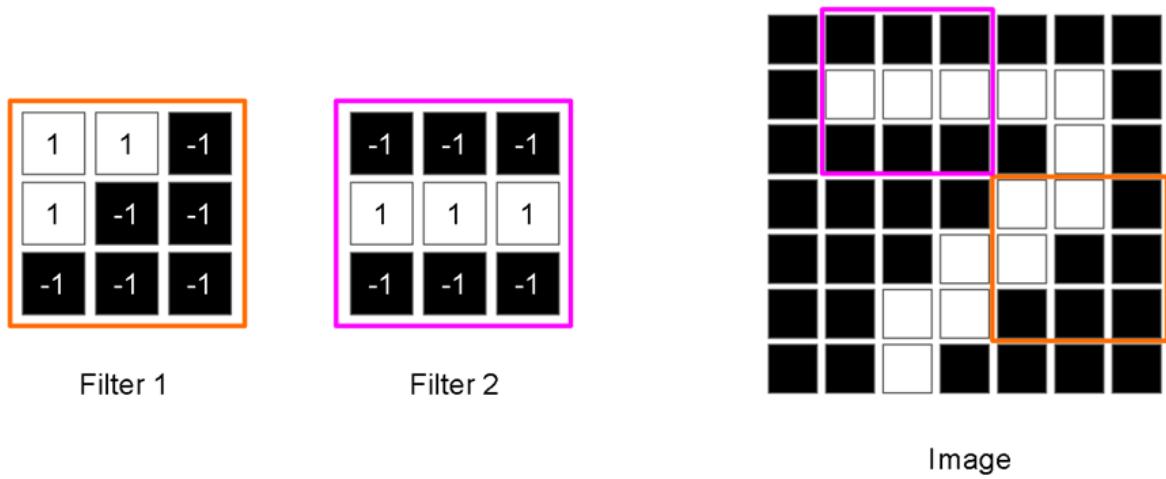


Figure 4.7.1.9. Diagonal and Horizontal Filters overlapping with images that are exactly the same

Using Filters to Find Features

Once a filter is defined, it is overlaid over each pixel in the image such that the image pixel (along with its surrounding pixels) exactly matches the shape of the filter. Observe the filter in Figure 4.7.1.10 which is overlaid on the pixel highlighted in green. You now multiply each of the values in the 9 pixels with the corresponding value of the filter at the same position. You finally sum up the numbers to get a new value for the pixel you inspected which shows how correlated the filter was with the image pixels around that pixel. In the image below you can see the sum comes to -2 which shows this pixel (and its surrounding pixels) is not very correlated with the features the filter is looking for.

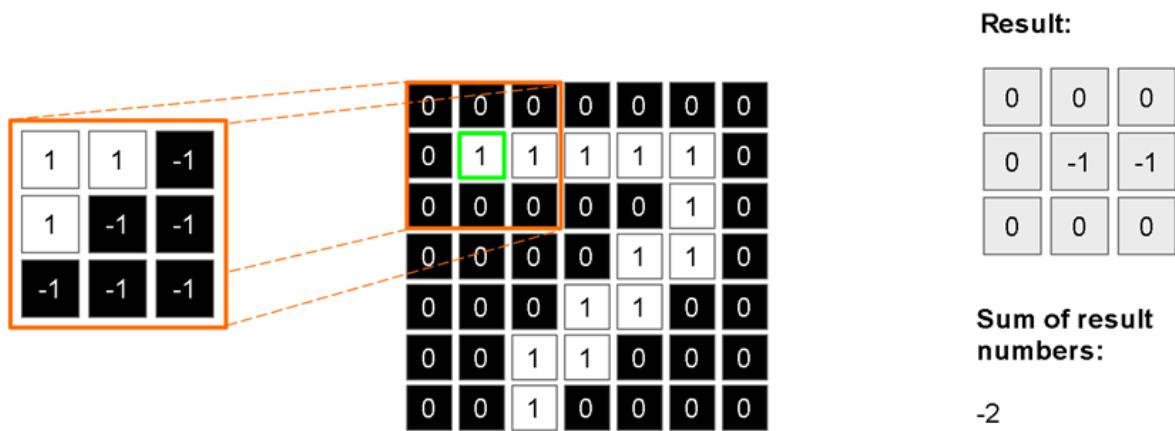


Figure 4.7.1.10. Calculating Convolutions 1

If you follow the same process with the same filter, but this time overlay it on the pixel highlighted in green in Figure 4.7.1.11, you obtain a final sum of 3. This indicates that the part of the image identified in Figure 4.7.1.11. is a better match for the filter than the part of the image identified in Figure 4.7.1.10. This makes intuitive sense when you see the pattern of values in the filter and the corresponding shape in the matching part of the image.

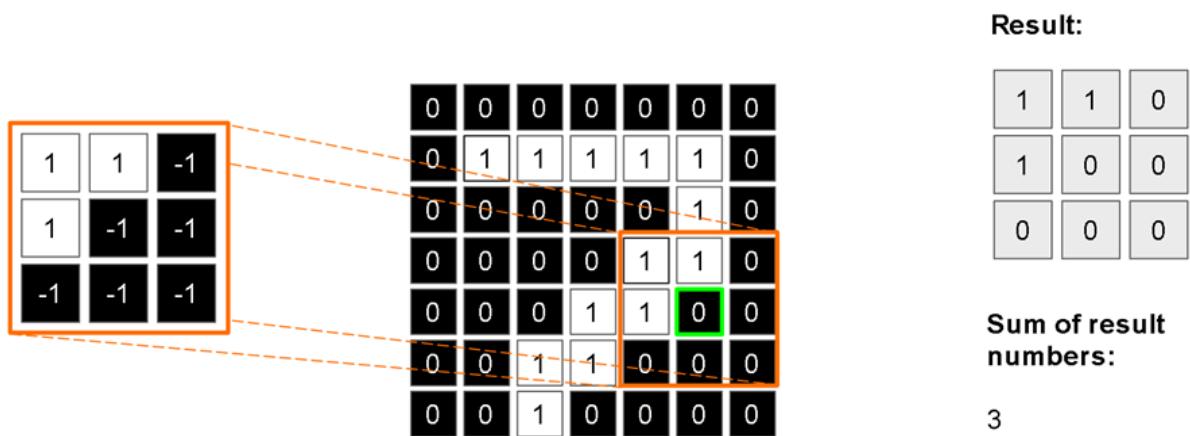


Figure 4.7.1.11. Calculating Convolutions 2

Padding

It might be difficult to overlay filters on every pixel in the image, especially when you reach pixels at the very edge of the image. To deal with this, CNNs use a concept called ‘padding’ that allows you to specify the numbers you can use to fill these out-of-bounds imaginary pixels for the purpose of convolution. See the gray padding pixels in Figure 4.7.1.12. You may also choose to only sample pixels inside the image that have enough data around them but you may then miss data near the edge of the image. This is a developer choice.

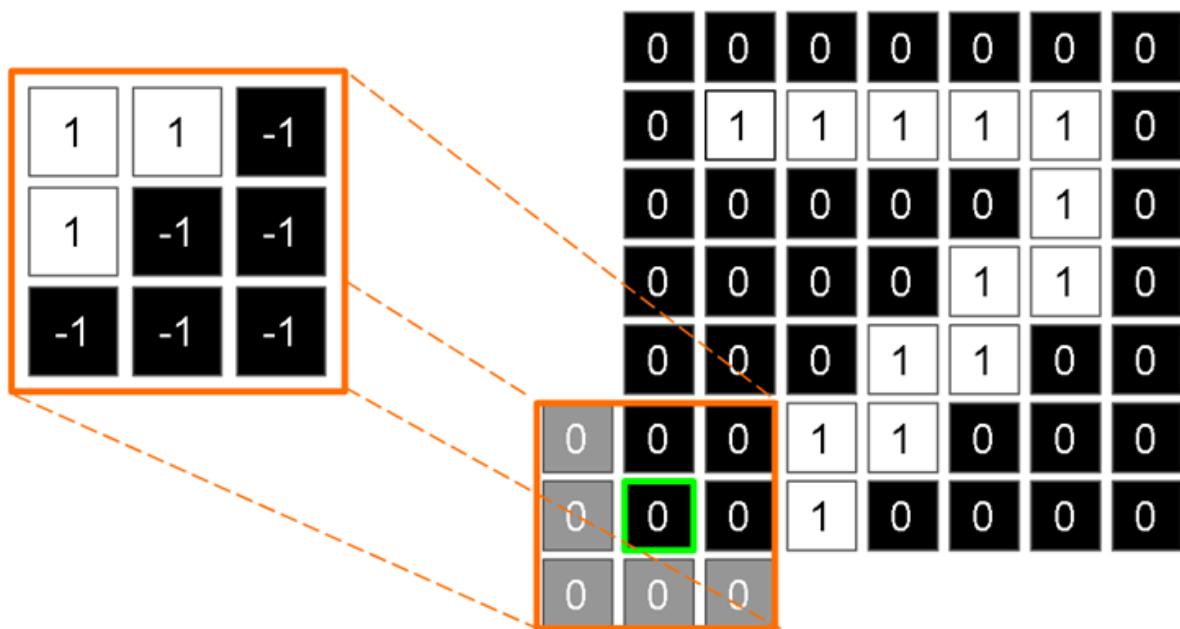


Figure 4.7.1.12. Padding

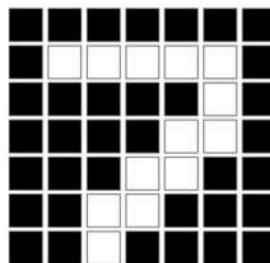
In Figures 4.7.1.13 and 4.7.1.14, the numbers in the colored grid are the output values obtained once the diagonal and horizontal filters are applied to each pixel of the image containing the larger “7”. Note that the numbers highlighted in green have higher values whereas the numbers highlighted

in red have lower values. Observe that the filters can highlight the features they are looking for in green in the image pretty well.

Similarly, in Figures 4.7.1.15 and 4.7.1.16, we apply the same filters to the smaller “7” input image. Even though this is an entirely different “7” it can still pick up on the horizontal and diagonal edges pretty well in each case.

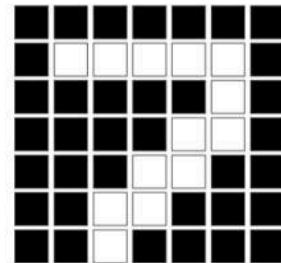
-1	-2	-3	-3	-3	-2	-1
-1	-2	-1	-1	-2	-1	0
-1	0	1	0	-2	-1	1
0	0	-1	-3	-5	0	2
0	-1	-3	-5	-1	3	1
0	-2	-4	-1	3	1	0
0	-2	-1	3	1	0	0

Figure 4.7.1.13. Diagonal Filter 1



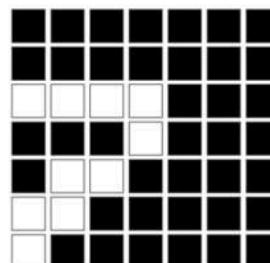
-1	-2	-3	-3	-3	-2	-1
1	2	3	3	2	1	0
-1	-2	-3	-4	-3	-2	0
0	0	-1	-1	-2	-1	-1
0	0	0	0	0	0	0
0	-1	-2	-1	-1	-1	0
0	1	0	0	-1	0	0

Figure 4.7.1.14. Horizontal Filter 1



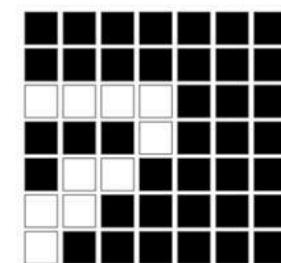
0	0	0	0	0	0	0
-2	-3	-3	-2	-1	0	0
-2	-1	-2	-1	0	0	0
-1	-1	-2	0	2	0	0
-3	-4	-2	2	1	0	0
-4	-1	3	1	0	0	0
-1	3	1	0	0	0	0

Figure 4.7.1.15. Diagonal Filter 2



0	0	0	0	0	0	0
-2	-3	-3	-2	-1	0	0
2	3	2	1	0	0	0
-3	-5	-4	-2	0	0	0
-1	0	0	0	-1	0	0
0	-1	-1	-1	0	0	0
-1	-1	-1	0	0	0	0

Figure 4.7.1.16. Horizontal Filter 2



To conclude, a convolutional layer in TensorFlow.js is a stack of trainable filters that you are applying to data such as images. Convolutions help classify things where the relative positions of data actually matter. Remember, you can also have 3D convolutions that sample data from more dimensions too.

Stride

This refers to the number of pixels you will move the filter by for each convolution to go through the image. With a stride of one, you would move one pixel to the right after each convolution, adding padding when needed

(see Figure 4.7.1.17). When you reach the right-most pixel of a row, you move to the next row (one pixel down).

With a stride of two, you would move the filter two pixels at a time instead.

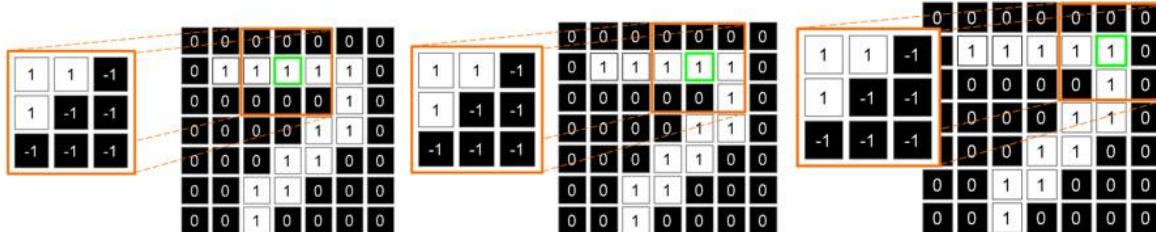


Figure 4.7.1.17 - Convolution Stride with stride of 1

Max Pooling

When you add multiple convolutional layers, each layer learns more complex features from the ones found in the layer before it. This involves a huge amount of processing between layers. To reduce the size of images after convolutions, a special max pooling layer is often used.

Figure 4.7.1.18. shows a 7x7 input image that already had a convolutional layer applied, and is now undergoing a max pooling layer being applied to the convolutional result. Here the max pooling has a stride of 2 and a size of 2 by 2. Using max pooling allows you to sample each position, find the maximum value from those numbers, and record only the highest number. This allows you to keep the most important information and disregard the rest, reducing the size of the layer considerably.

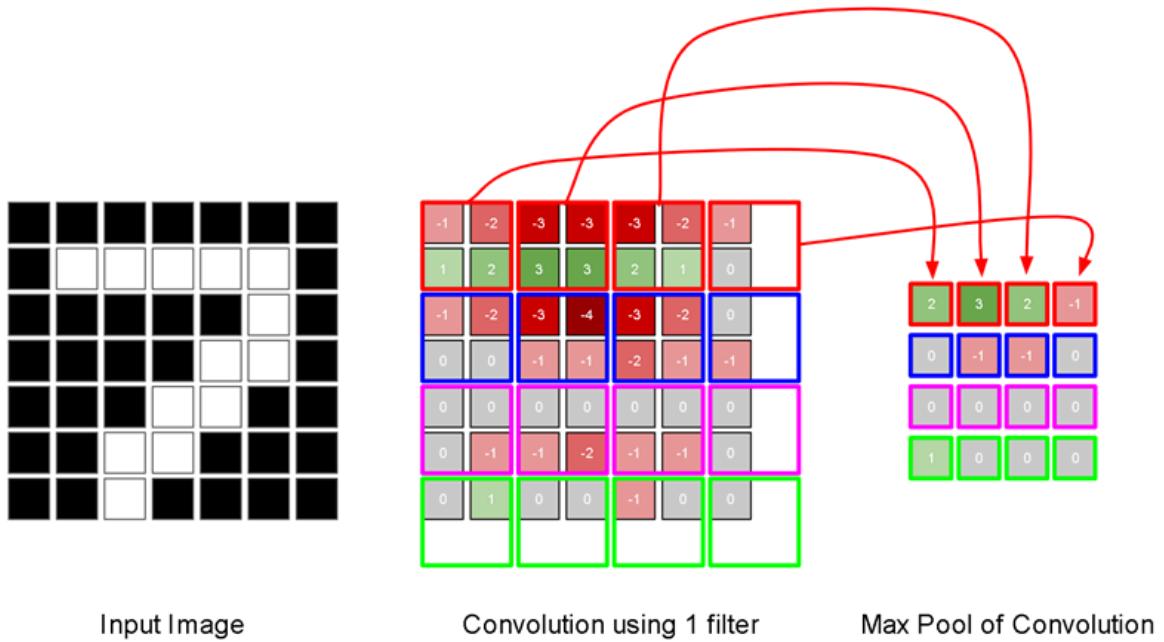


Figure 4.7.1.18 - Convolution and Max Pool of Convolution 1

Observe (see Figure 4.7.1.19) that the output of a max pooling layer would be smaller than the original input while still retaining crucial information necessary for classification. Max pooling thus greatly reduces the complexity of a model while still having a chance of recognizing objects that are at positions different from where they were in the training data.

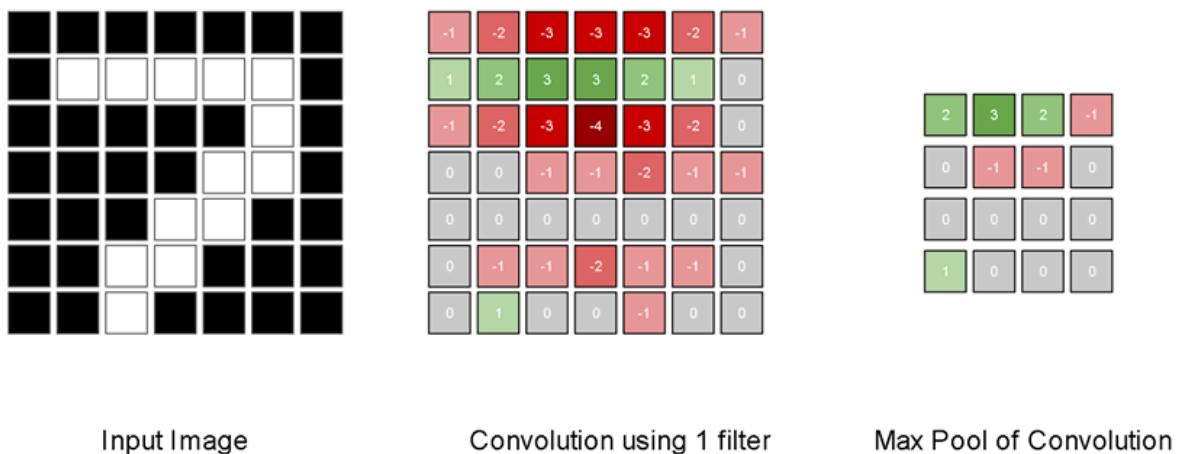


Figure 4.7.1.19: Convolution and Max Pool of Convolution 2

If you have multiple filters (see Figure 4.7.1.20), max pooling would be applied to the output of each convolutional filter's result. You would end up with a stack of max pooling results, each one recognizing a distinct feature.

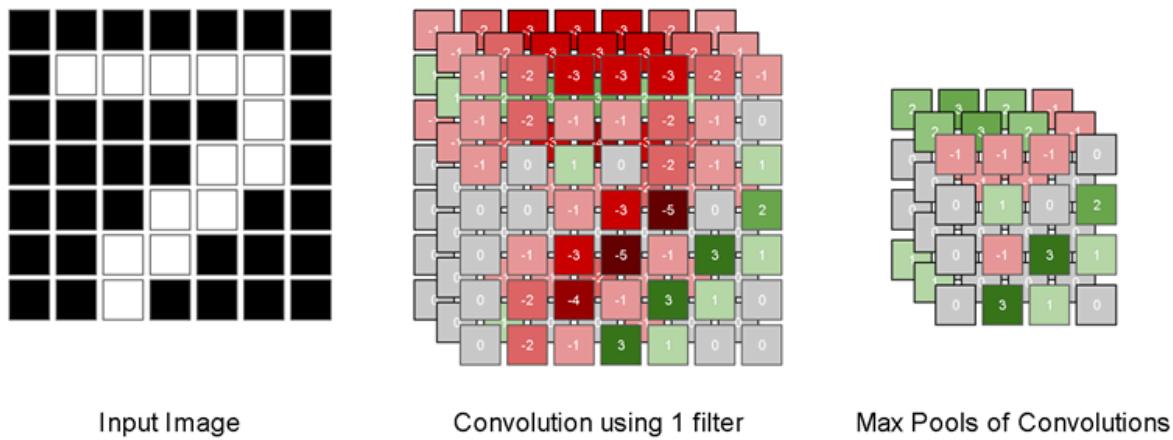


Figure 4.7.1.20: Multiple Convolutional Filters

Building a Realistic Convolutional Network

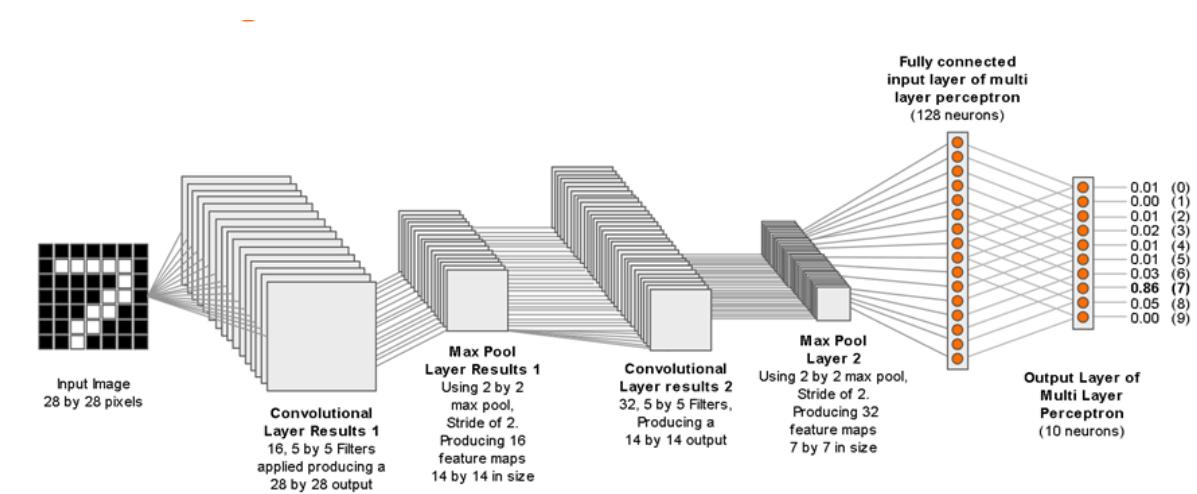


Figure 4.7.1.21. Sample Convolutional Neural Network

Steps:

1. The input is a 28x28 grayscale image.
2. The input image is followed by a convolutional layer containing 16 convolutions. The convolutions use a 5x5 filter and use padding to produce a 28x28 output shape.
3. This layer is now fed into a max pooling layer that uses a 2x2 max pooling size, with a stride of two. This results in 16 feature maps, each 14x14 in size. Note that max pooling has reduced the convolutional results to a quarter of the original size without losing any key information.
4. The results of the first max pool are then fed into a second convolutional layer containing 32 convolutions to try and find some more advanced filters at this deeper level. The convolutions use a 5x5 filter and use padding to produce a 14x14 output shape.
5. The results of the second convolutional layer are fed into a second max pool layer that uses a 2x2 max pooling size, with a stride of two. This results in 32 feature maps each 7x7 in size.
6. The 32 feature maps from the second max pool layer are flattened into an array of numbers. This array is then fed into a regular multi-layered perceptron. The perceptron has 128 neurons in the first layer and 10 neurons in the output layer. The neurons in the output layer represent each of the 10 digits the model can classify. Since the input image is a seven, if the model classifies accurately, the output value of the eighth neuron should be the highest as the first output neuron represents 0.

Note:

CNNs can be used for any data that looks like an image. This includes data where the spatial position of objects of information is important to understand how to classify or represent it. Sound is a good example of other data that can be represented as an image. Spectrograms, which are visual representations of sound depict the frequencies heard over time, and can also be classified by CNNs allowing you to do short form audio recognition too using this sort of network.

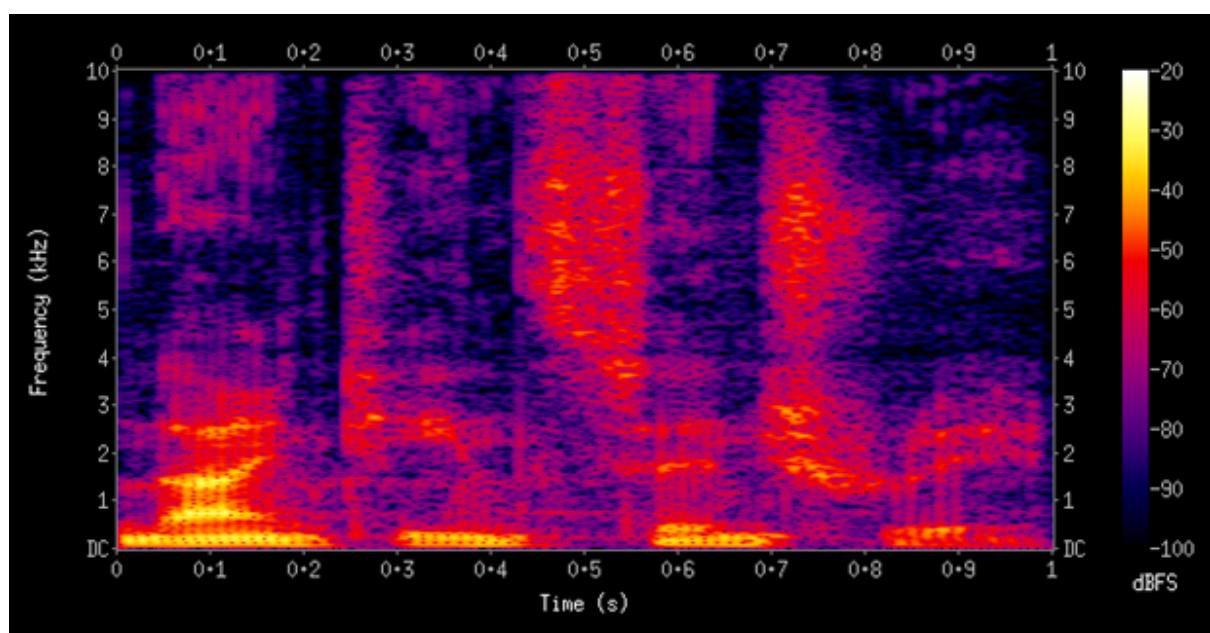


Figure 4.7.1.22. A spectrogram showing frequencies heard over time. You can get a CNN to recognize images like this too to detect sound.

Guía: Redes Neuronales Convolucionales (CNN)

Las CNN son una evolución de los Perceptrones Multicapa (MLP). Mientras que un MLP puede fallar si un número está un poco movido o rotado, la CNN es capaz de detectar características (líneas, curvas) en cualquier lugar de la imagen.

1. El Problema de las Redes "Simples" (MLP)

Un MLP mira cada píxel en una posición específica. Si aprendió que un "3" tiene píxeles blancos en el centro y tú le pasas un "3" movido a la derecha, la red dirá: *"No sé qué es esto, porque en el centro ahora hay píxeles negros"*.

Las CNN solucionan esto mediante **filtros**.

2. Componentes Clave de una CNN

A. Filtros o Kernels

Imagina que tienes una lupa pequeña (de 3x3 píxeles) que solo busca "líneas diagonales". Pasas esa lupa por toda la imagen.

- **Correlación alta:** Si la lupa encuentra una diagonal, genera un número alto.
- **Correlación baja:** Si no encuentra nada, genera un número bajo o negativo.

Estos filtros son matrices de números que la red aprende a ajustar durante el entrenamiento.

B. Convolución (El escaneo)

Es el proceso de deslizar el filtro sobre la imagen píxel por píxel. Al multiplicar los valores del filtro por los de la imagen y sumarlos, obtenemos un nuevo mapa de características que resalta dónde están las líneas, bordes o curvas.

C. Padding (Acolchado)

Cuando pasas un filtro, los bordes de la imagen se procesan menos que el centro. El **Padding** añade un marco de píxeles extra (usualmente ceros) alrededor de la imagen para que el filtro pueda pasar por los bordes sin perder información y mantener el tamaño original.

D. Stride (Zancada)

Es el "salto" que da el filtro.

- **Stride 1:** El filtro se mueve de 1 en 1 píxel.
- **Stride 2:** El filtro salta 2 píxeles, lo que reduce el tamaño del resultado a la mitad.

E. Max Pooling (Resumen)

Después de encontrar las características, la imagen sigue teniendo mucha información. El **Max Pooling** toma ventanas (ej. de 2x2) y se queda solo con el valor más alto (el más importante).

- **Beneficio:** Reduce drásticamente el tamaño de los datos y hace que la red sea resistente a pequeños movimientos del objeto.

3. Anatomía de una CNN Real (Ejemplo MNIST)

Un modelo realista para reconocer números sigue este flujo:

1. **Entrada:** Imagen de 28x28 píxeles.
2. **Capa Convolucional 1:** Se aplican 16 filtros (5x5). Detectan bordes simples.

3. **Max Pooling 1:** Reduce la imagen de 28x28 a 14x14.
4. **Capa Convolucional 2:** Se aplican 32 filtros más complejos. Detectan formas (círculos, cruces).
5. **Max Pooling 2:** Reduce la imagen a 7x7.
6. **Flatten (Aplanado):** Convertimos esos mapas de 7x7 en una lista larga de números (un vector).
7. **Capa Densa (MLP):** Esa lista entra a una red normal con neuronas finales para decidir: "*Es un 7*".

4. Más allá de las Imágenes: Sonido

¡Las CNN no solo sirven para fotos! El sonido se puede convertir en una imagen llamada **Espectrograma** (que muestra frecuencias a lo largo del tiempo). Una CNN puede "mirar" ese espectrograma y reconocer palabras, comandos de voz o ruidos ambientales.

Resumen técnico para recordar:

- **Filtros:** Buscan patrones.
- **Convolución:** Aplica el filtro.
- **Max Pooling:** Simplifica y reduce el tamaño.
- **Arquitectura:** Primero extraemos características (Conv + Pool) y al final clasificamos (Capa Densa).

Implementing a Convolutional Neural Network (CNN)

In this lesson, you will see how to train a CNN to classify data from the Fashion MNIST dataset. The Fashion MNIST dataset is similar to the original MNIST dataset but consists of clothing items instead of handwritten digits (see Figure 4.7.2.1).

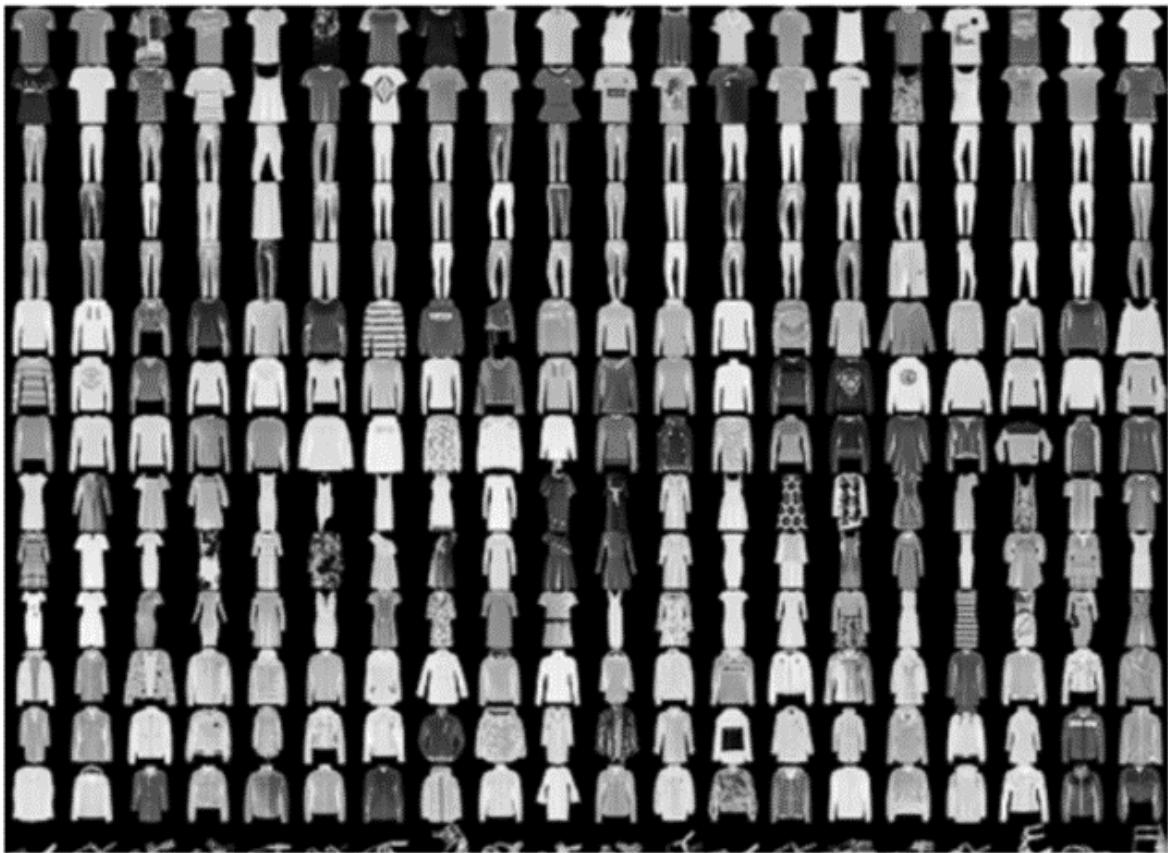


Figure 4.7.2.1 - Fashion MNIST Dataset Examples

Steps:

1. Navigate to a [**completed version**](#) of a Fashion MNIST demo that uses a multi-layered perceptron to classify the data. Fork a new copy.
2. The Fashion MNIST dataset provided differs from the MNIST handwritten digits dataset in the following aspects:
 - a. The Fashion MNIST dataset uses fashion images instead of numbers.
 - b. The data for this demo is not normalized, unlike the data you used for handwritten digits.
3. The images in this dataset are 28 x 28 grayscale images (similar to the images in the MNIST dataset), and the number of output classes is 10.
4. Note the differences in the code found in this template when compared to the MNIST number classification code.

- a. **Training Data:** The [URL](#) from which the training data is imported is different.

```
import {TRAINING_DATA} from  
'https://storage.googleapis.com/jmstore/TensorFlowJS/EdX/TrainingData/fashion-mnist.js';
```

- b. **Normalization function:**

- i. For normalizing the MNIST handwritten digit data, the ‘tf.min’ and the ‘tf.max’ functions were used to calculate the minimum and maximum values in a tensor. Since the pixels of images in the Fashion MNIST dataset are always in the range of 0 to 255, the minimum and maximum values (0 and 255) can directly be passed as numbers instead of tensors as I have updated the normalize functions code to account for this to make it a little cleaner to read in this use case.

```
function normalize(tensor, min, max) {  
  
  const result = tf.tidy(function() {  
  
    const MIN_VALUES = tf.scalar(min);  
  
    const MAX_VALUES = tf.scalar(max);  
  
    const TENSOR_SUBTRACT_MIN_VALUE = tf.sub(tensor, MIN_VALUES);  
  
    const RANGE_SIZE = tf.sub(MAX_VALUES, MIN_VALUES);  
  
    const NORMALIZED_VALUES = tf.div(TENSOR_SUBTRACT_MIN_VALUE, RANGE_SIZE);  
  
    return NORMALIZED_VALUES;  
  })  
  .node();  
  
  return result;};
```

```

        return NORMALIZED_VALUES;

    });

    return result;
}

// Input feature Array is 2 dimensional.

const INPUTS_TENSOR = normalize(tf.tensor2d(INPUTS), 0, 255);

```

- c. **Converting output to a human-readable form:** The Fashion MNIST dataset has images of 10 different clothing items. The model, however, produces a numerical output in the form of an array, with the position of the highest output number representing the predicted class. In order to convert this position to a human-readable form, an array of strings that corresponds to the order of 1-hot encodings is used. So if the model has the highest number output at position 0 on the output layer, the element 0 in the string array needs to be looked up to find the human-readable name which would be “t-shirt” as shown below:

```

// Map output index to label.

const LOOKUP = ['T-shirt', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal',
' Shirt', 'Sneaker', 'Bag', 'Ankle boot'];

```

- d. **Classification interval:** A ‘div’ element containing a paragraph and an input range slider is added in the HTML document. The slider is used to change the rate at which random images from the dataset are classified. An event listener is added to the JavaScript code to

listen for the change in the value of the input. When a change is detected, the event listener updates an ‘interval’ variable that is used by the ‘setTimeOut’ in the classification loop.

index.html

```
<div class="blocky">

    <p id="domSpeed">Change interval between classifications! Currently 1000ms</p>

    <input id="ranger" type="range" min="15" max="5000" value="2000"
    class="slider">

</div>
```

script.js

```
var interval = 2000;

const RANGER = document.getElementById('ranger');

const DOM_SPEED = document.getElementById('domSpeed');

// When user drags slider update interval.

RANGER.addEventListener('input', function(e) {

    interval = this.value;

    DOM_SPEED.innerText = 'Change speed of classification! Currently: ' + interval
    + 'ms';

});
```

5. HTML Code: Update the header and paragraph tag text and replace the second section tag with the content shown.

```
<h1>TensorFlow.js Fashion MNIST classifier</h1>

<p>This experiment shows how you can use the power of Machine Learning directly in your browser to train and use a more advanced Convolutional Neural Network to classify clothing items from the Fashion MNIST dataset. Warning training is slow in browser, ideally you would do this in Node.js with CUDA installed for faster training. In the browser without CUDA it will take about 5 mins to train, and then inference will be super fast :-)</p>

<section class="box">

  <h2>Prediction</h2>

  <p>Below you see what item the trained TensorFlow.js model has predicted from the input image.</p>

  <p>It should get it right most of the time, but of course it's not perfect, so there will be times when it doesn't. Red is a wrong prediction, Green is a correct one.</p>

  <p id="prediction">Training model. Please wait. This can take ~5 minutes. Check console for progress (30 epochs total)</p>

</section>
```

6. JavaScript Code in Script.js:

- a. Comment out the call to the ‘train’ function. This prevents Glitch from running your code automatically and starting the training process before everything is defined correctly.
- b. Then remove the current model definitions leaving you with the following code:

```
// Now actually create and define model architecture.

const model = tf.sequential();

model.summary();

// train();
```

- c. Add a convolutional layer with the following hyper-parameters.
 - i. The input shape should be in the form of width, height, and the number of color channels. The input images are 28x28 pixels in size and grayscale, so the expected input shape is [28, 28, 1].
Note: For RGB images, the last value would be three.
 - ii. Set the number of filters to 16.
 - iii. Set the size of each filter (or kernel) to three. This will generate sixteen 3x3 square filters.
Note: To generate rectangular filters, you could specify an array containing dimensions for width and height as needed.
 - iv. Set stride value to one. This means a filter value is calculated for every pixel in the input image.
 - v. Set the padding property to ‘same’. This ensures missing values are filled with zeroes.
 - vi. Add a ‘relu’ activation function.

d. Then add a max pooling layer with the following hyper-parameters.

Note that this results in an output that is 14x14 pixels in size.

- i. Set the pool size to two.
- ii. Set the stride value to two.

```
// Now actually create and define model architecture.

const model = tf.sequential();

model.add(tf.layers.conv2d({

    inputShape: [28, 28, 1], 

    filters: 16, 

    kernelSize: 3, // Square Filter of 3 by 3. Could also specify rectangle eg [2, 
3]. 

    strides: 1, 

    padding: 'same', 

    activation: 'relu'

})); 

model.add(tf.layers.maxPooling2d({poolSize: 2, strides: 2}));
```

- e. Add a second convolutional layer by replicating the code for the first one, but double the number of filters to 32.
- f. Add a second max pooling layer by replicating the code for the first one.

```
model.add(tf.layers.conv2d({  
  
    filters: 32,  
  
    kernelSize: 3,  
  
    strides: 1,  
  
    padding: 'same',  
  
    activation: 'relu'  
}));  
  
model.add(tf.layers.maxPooling2d({poolSize: 2, strides: 2}));
```

- g. Add code to run the outputs through a regular multi-layer perceptron.
- i. You need to first flatten the outputs from the final max pooling layer. The max pool currently has thirty-two 7x7 outputs. Use ‘tf.layers.flatten’ to convert the output into a list of numbers.
 - ii. Add a dense layer with 128 neurons and a ‘relu’ activation.
 - iii. Add another dense layer with 10 neurons. Since this is the output layer, add a ‘softmax’ activation function.

```
model.add(tf.layers.flatten());  
  
model.add(tf.layers.dense({units: 128, activation: 'relu'}));
```

```

model.add(tf.layers.dense({units: 10, activation: 'softmax'}));

model.summary();

// train();

```

You should see a model summary similar to the following on the console.

Layer (type)	Output shape	Param #
conv2d_Conv2D1 (Conv2D)	[null, 28, 28, 16]	160
max_pooling2d_MaxPooling2D1	[null, 14, 14, 16]	0
conv2d_Conv2D2 (Conv2D)	[null, 14, 14, 32]	4640
max_pooling2d_MaxPooling2D2	[null, 7, 7, 32]	0
flatten_Flatten1 (Flatten)	[null, 1568]	0
dense_Dense1 (Dense)	[null, 128]	200832
dense_Dense2 (Dense)	[null, 10]	1290
<hr/>		
Total params: 206922		
Trainable params: 206922		
Non-trainable params: 0		

Figure 4.7.2.2 - CNN Model Summary

With over 200 thousand trainable parameters, this will take a while to train so it is good to keep the call to the train function commented out until

everything is ready. In the next unit, you complete coding for the ‘train’ and ‘evaluate’ functions and see how your model performs.

JavaScript Code (contd.)

a. The train function: Adjust the following parameters in the train function:

- i. Reshape the current training data into a form that the CNN input layer can digest. Currently the training data is an array of arrays containing a list of 784 numbers. Use the ‘.reshape’ function on the input tensor with the following parameters:
 1. Size: Use ‘inputs.length’ to specify training data size.
 2. Height and Width: This should be 28x28 according to the size of the images.
 3. Channel Count: This should be one since the images are in grayscale.
- ii. Set the validation split to 0.15 in order to set aside 15% of the training data for validation.
- iii. Set the number of epochs to 30. Note that this value is arrived at after experimentation.
- iv. Set batch size to 256. Note that this value is arrived at after experimentation. Feel free to try other values to see if you get better results.

```
async function train() {

  model.compile({
    optimizer: 'adam', // Adam changes the learning rate over time which is
                      useful.

    loss: 'categoricalCrossentropy', // As this is a classification problem, dont
                                   use MSE.

    metrics: ['accuracy']
  });

  const RESHAPED_INPUTS = INPUTS_TENSOR.reshape([INPUTS.length, 28, 28, 1]);

  let results = await model.fit(RESHAPED_INPUTS, OUTPUTS_TENSOR, {
    shuffle: true,           // Ensure data is shuffled again before using each
                           time.

    validationSplit: 0.15,

    epochs: 30,              // Go over the data 30 times!

    batchSize: 256,

    callbacks: {onEpochEnd: logProgress}
  });

  RESHAPED_INPUTS.dispose();

  OUTPUTS_TENSOR.dispose();
}
```

```
INPUTS_TENSOR.dispose();

evaluate();

}
```

b. The evaluate function: Update the evaluate function as follows:

- i. Normalize the input by passing the known minimum and maximum values of 0 and 255.
- ii. Call the model.predict function with this normalized input after reshaping it into a form the CNN needs.

```
function evaluate() {

const OFFSET = Math.floor((Math.random() * INPUTS.length));

let answer = tf.tidy(function() {

    let newInput = normalize(tf.tensor1d(INPUTS[OFFSET]), 0, 255);

    let output = model.predict(newInput.reshape([1, 28, 28, 1]));

    output.print();

    return output.squeeze().argMax();

});

answer.array().then(function(index) {

    PREDICTION_ELEMENT.innerText = LOOKUP[index];

})};
```

```

PREDICTION_ELEMENT.setAttribute('class', (index === OUTPUTS[OFFSET]) ?
'correct' : 'wrong');

answer.dispose();

drawImage(INPUTS[OFFSET]);

});

}

```

Finally uncomment the call to the train function in your previously written code, open your console window in the browser, and wait for it to finish training. This will take some time so be patient. Each epoch for me took around 10 - 15 seconds so do not be surprised if it this takes over 5 minutes on your machine to train.

Data for epoch 13	script.js:125
{ val_loss: 0.4574549198150635, val_acc: 0.84133 ► 33296775818, loss: 0.4321088194847107, acc: 0.8 384705781936646}	
Data for epoch 14	script.js:125
{ val_loss: 0.4654689133167267, val_acc: 0.82933 ► 33053588867, loss: 0.410268634557724, acc: 0.84 78823304176331}	
Data for epoch 15	script.js:125
{ val_loss: 0.4727128744125366, val_acc: 0.82266 ► 66450500488, loss: 0.408008873462677, acc: 0.85 1411759853363}	

Figure 4.7.2.3 - Epoch Callbacks

Note: When you are training a model in the web browser, you **must** keep your browser window open and visible in order to access full system resources.

Results: You should obtain results comparable to the ones shown in Figure 4.7.2.4.

```
acc: 0.9210587739944458  
loss: 0.22044922411441803  
val_acc: 0.871999979019165  
val_loss: 0.351838618516922
```

Figure 4.7.2.4 - Model Results

Overfitting:

- Observe that the training accuracy is greater than the validation accuracy. This indicates a degree of overfitting. There are special layers called dropout layers on the TensorFlow.js API that can be added to a model's architecture in order to decrease overfitting.
- Essentially, these layers deactivate some fraction of the outputs of the prior layer randomly at training time by setting those selected values to zero. It should be noted this only happens during training. When using the trained model, these layers do not do anything.
- Now this added “noise” at training time can help, as only the features that really matter all the time will keep showing up enough, such that the network will learn those key patterns instead of ones that just show up occasionally.
- Common values for drop out are 0.5 or 0.25 representing 50% and 25% drop out respectively.

Your web page should now display the input images and predictions.

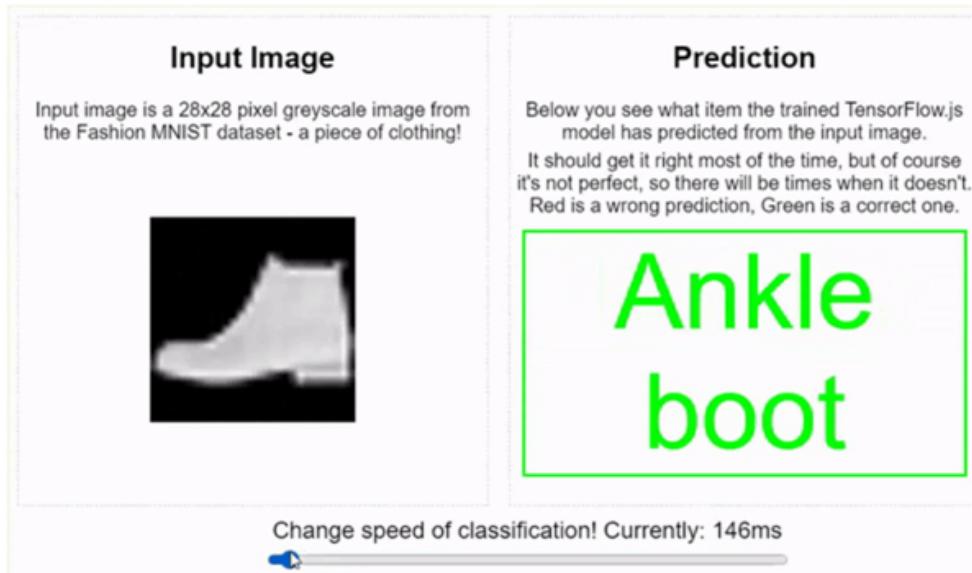


Figure 4.7.2.5: Fashion MNIST Classifier - Prediction

While the current model is accurate, in order to obtain better performance, you can train a model with more parameters within Node.js on a server with a good graphics card. This leverages the CUDA acceleration available on the server-side to make training go faster. The code for Node.js is essentially the same as the one used here, with minor differences. For instance, instead of using JS module imports for training data, you will need to load it from the local file system on the server which is great when dealing with gigabytes of data.

Transfer Learning

What You Will Learn

By the end of the chapter, you will be able to:

- Explain transfer learning in TensorFlow.js.
- Identify models that can be used for transfer learning.
- Demonstrate how to make custom image recognition models using transfer learning on MobileNet base models.

In addition to using pre-made models and creating new models from scratch, TensorFlow.js allows you to create models using a method called transfer learning. This involves reusing models trained for a particular task, and repurposing them for a similar task, using custom data, which can be much faster than starting with a fully untrained model.

You start by exploring use-cases, where it is advantageous to use transfer learning instead of choosing pre-made models or creating custom models from a blank canvas. You will then understand how to identify models that can be used for transfer learning and explore TensorFlow Hub to access existing models that can be used for retraining in this manner.

You will dive deeper by exploring how lower layers in the model may have learned to recognize common reusable features that can be repurposed for similar tasks.

Then using the MobileNet model architecture, you will implement transfer learning yourself and create your own version of Teachable Machine from a blank canvas that can be trained to recognize almost any custom object you show it. You will also understand the difference between using graph vs layers models in the context of transfer learning. You will learn why layers models can be more useful for transfer learning with the ability to freeze individual layers when training and also understand how they enable you to combine multiple models into one that you can then download and share with ease.

In the previous chapter, you saw how to create models from scratch. A disadvantage of this approach is that it can take a considerable amount of time to train models that use advanced model architectures.

State-of-the-art models may be much larger and can take hours, days, or even weeks to train.

In this chapter, you explore a new approach where you reuse a model that has already been trained on a particular task and you repurpose it for a similar new task using custom data. You are essentially reusing the knowledge the model has already learned! For instance, if you have a model that is already trained on some domain such as recognizing images of say, cats, you can reuse it to perform a different, but related task, like recognizing dogs.

Transfer Learning with the MobileNet Model

- MobileNet is a very popular research model that is able to perform image recognition on 1,000 different object types. It was trained on the ImageNet dataset, which consists of millions of labeled images.
- During training, the model learned how to extract common features for 1,000 objects. Many of the lower level features (e.g. lines, shapes, textures) that are used to classify these objects can be used to detect new objects too.

MobileNet is an example of a Convolutional Neural Network (CNN). As you saw in the previous chapter, a CNN begins with a convolutional/max-pooling layers and ends with a multi-layer perceptron (see Figure 5.1.1.). The convolutional layers are used to identify features that matter, whereas the perceptron layers are used to perform the classification.

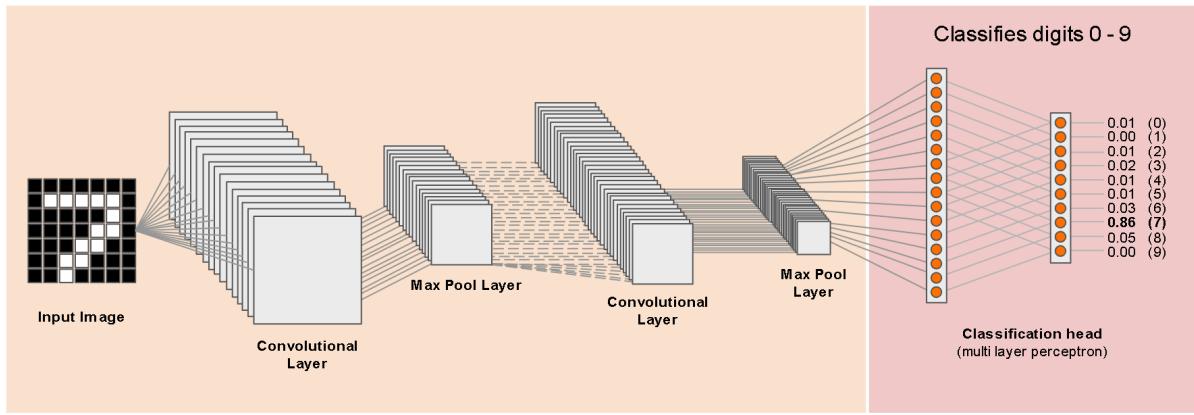


Figure 5.1.1. Convolutional Neural Network (CNN)

How Transfer Learning Occurs:

1. The pre-trained convolutional layers of an existing trained model are separated from the classification layers near the end of the model.
2. The convolutional layers are then used to produce output features for any image based on the original data it was trained on.

Note: The classification layers are also referred to as the ‘classification head’ of the model.

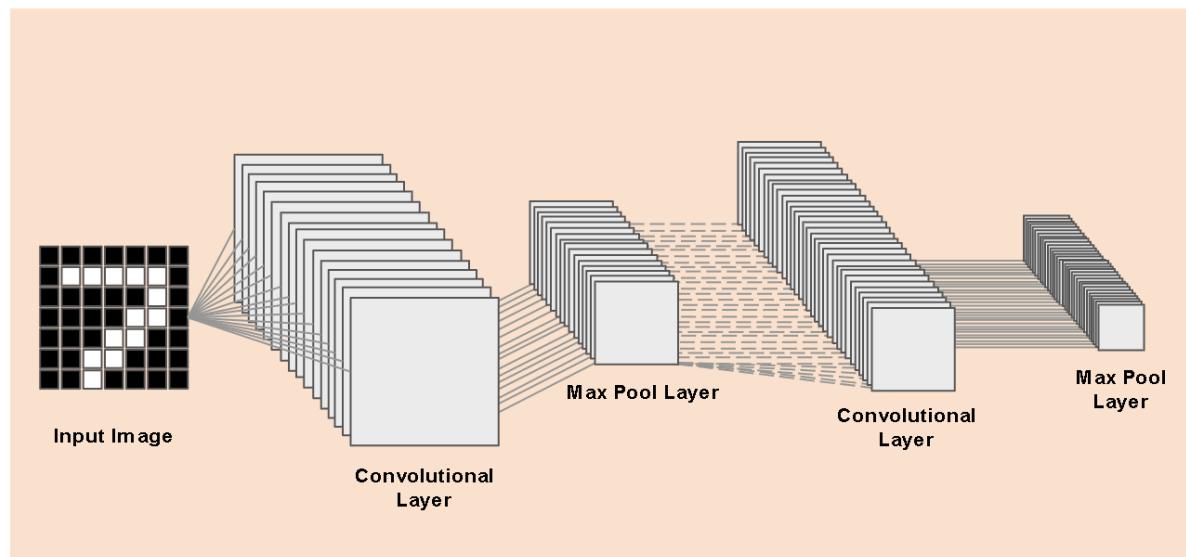


Figure 5.1.2. Convolutional Layers of a CNN

3. Transfer learning is based on the assumption that you have new images that can also make use of the output features that have been learned when the model was originally trained. For instance, the convolutional layers shown in Figure 5.1.2. were trained on digits, but the features it has learned can also be applied to recognize letters such as 'a', 'b', 'c', etc (see Figure 5.1.3.).

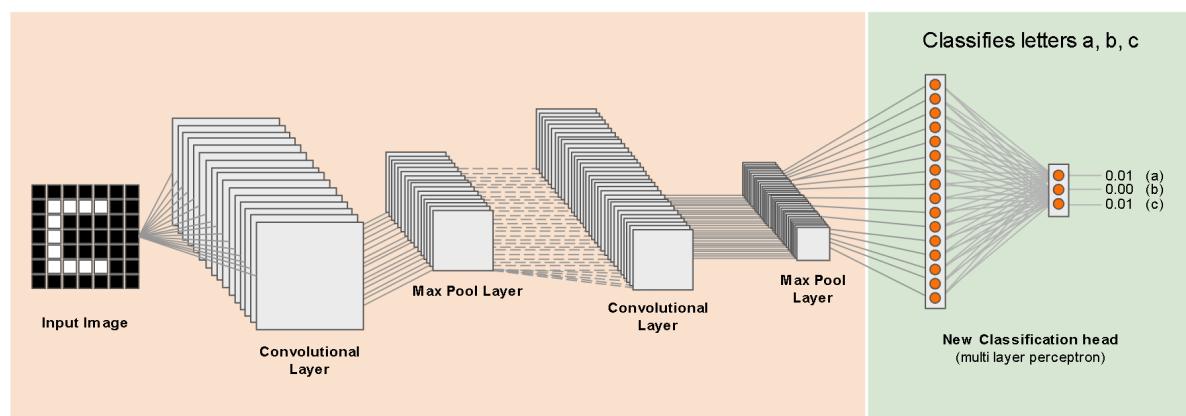


Figure 5.1.3. Repurposed Classification Head of a CNN

4. You can now add and train a new classification head that predicts letters instead of numbers while freezing the convolutional layers so that they do not change their pre-learned weights. Only having to train the classification head significantly shortens the network training period because you do not have to train the whole network from scratch.

Note: The Teachable Machine website that you worked with in earlier chapters uses transfer learning to classify images, poses, and sounds using this technique.

In the next unit, you will explore how to access parts of a model that can be used for transfer learning.

Accessing the MobileNet v3 Graph Model for Transfer Learning

Steps

1. Go to the TensorFlow Hub and filter for TensorFlow.js models that use the MobileNet v3 architecture. You will find results such as the ones shown in Figure 5.1.4.

MobileNet v3 graph model

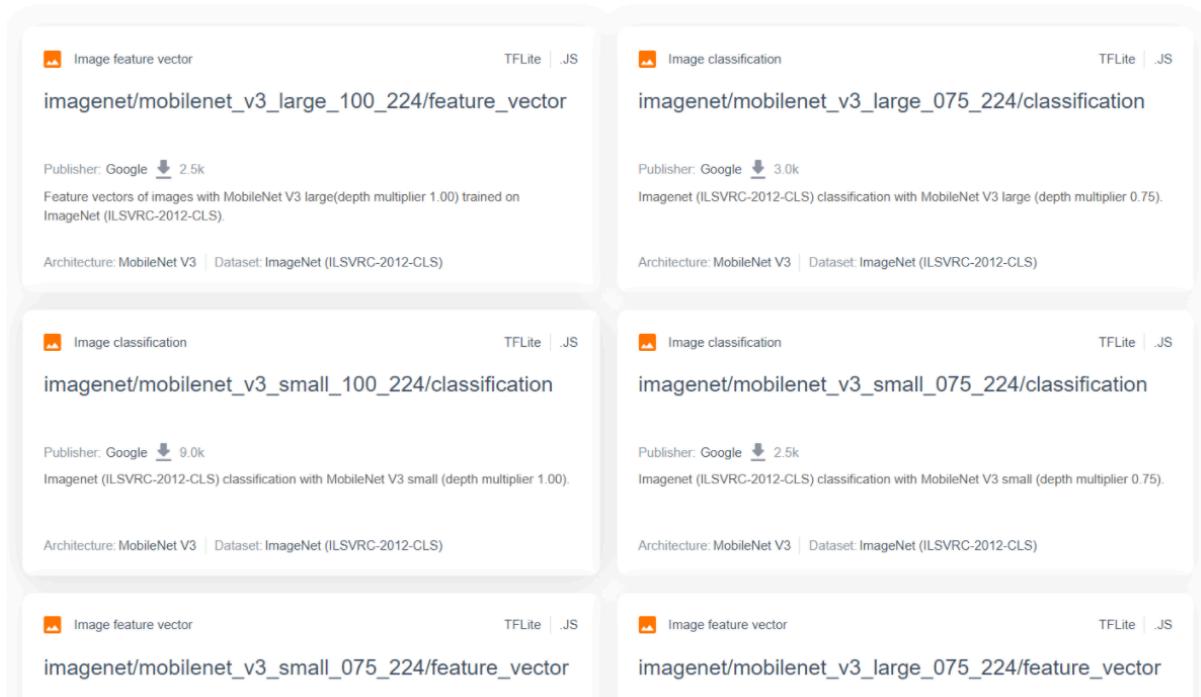


Figure 5.1.4. MobileNet v3 Models in TensorFlow Hub

2. Observe that some of the models are ‘image classification’ models, whereas others are ‘feature vector’ models. While the ‘image classification’ models are used to classify images, the ‘feature vector’ results are

pre-chopped versions of MobileNet that are used to obtain the image feature vectors only. These models are sometimes called ‘base models’ and can be used to perform transfer learning.

The screenshot shows two entries from the TensorFlow Model Zoo:

- Image classification**:
 - Publisher: Google ([Download](#)) 3.0k
 - Description: Imagenet (ILSVRC-2012-CLS) classification with MobileNet V3 large (depth multiplier 0.75).
 - Architecture: MobileNet V3 | Dataset: ImageNet (ILSVRC-2012-CLS)
- Image feature vector**:
 - Publisher: Google ([Download](#)) 2.5k
 - Description: Feature vectors of images with MobileNet V3 large(depth multiplier 1.00) trained on ImageNet (ILSVRC-2012-CLS).
 - Architecture: MobileNet V3 | Dataset: ImageNet (ILSVRC-2012-CLS)

5.1.5. Classification Models vs. Feature Vector Models

3. Navigate to the ‘feature vector’ MobileNet v3 model and read the JS documentation to find out if the models are in the form of a ‘graph’ model (see Figure 5.1.6.) or ‘layers’ model. Layers models are more useful for transfer learning because you can then combine the new classification head with the layers of the base MobileNet model once training is complete to combine the resulting trained model into a single output instead of 2

different models. Also, when using ‘layers models’, you have the option of choosing which layers to freeze and which to unfreeze for training which can allow fine-tuning the model further too.

Inputs

The input `images` are expected to have color values in the range [0,1], following the [common image input](#) conventions.
`height x width = 224 x 224 pixels.`

Example use

This model can be used with [TensorFlow.js](#) as:

```
// Be sure to load TensorFlow.js on your page. See
// https://github.com/tensorflow/tfjs#getting-started.

const model = await tf.loadGraphModel(
  'https://tfhub.dev/google/tfjs-model/imagenet/mobilenet_v3_small_100_224/feature_vector/5/d'
  { fromTFHub: true });
```

Figure 5.1.6. MobileNet v3 Graph Model

In this instance, you will be using a graph model from TFHub, but if you are fortunate enough to find a layers model for transfer learning, once training is complete, the combined model can be saved using the ‘model.save’ function. This will provide you with one model that you can use in the future, instead of needing to load two different models to perform inference.

Note: Transfer learning sometimes also involves retraining the ‘base model’ for fine-tuning. However, in this chapter, you will focus on retraining the classification head while freezing the ‘base model’ layers.

Advantages of Transfer Learning:

1. Can lead to a shorter training period
2. Reuses knowledge already learned by the base model
3. Requires less example data
4. Suitable for a web browser environment where resources may vary based on device executed on.

Transfer Learning: "No reinventes la rueda"

El **Transfer Learning** consiste en tomar un modelo que ya es un "experto" en una tarea general y aprovechar ese conocimiento para que aprenda una tarea nueva y específica en tiempo récord.

1. La Analogía del Chef

Imagina que quieres contratar a alguien para que aprenda a cocinar **comida tailandesa**:

- **Opción A (Desde cero):** Contratas a alguien que nunca ha entrado en una cocina. Tienes que enseñarle a cortar, a encender el fuego y qué es una sartén. (Esto es como entrenar una red desde cero: tardas semanas).
- **Opción B (Transfer Learning):** Contratas a un chef profesional de comida italiana. Él ya sabe usar cuchillos, conoce los tiempos de cocción y sabe cómo funcionan los sabores. Solo tiene que aprender los ingredientes tailandeses. (Esto es Transfer Learning: en 2 días ya es un experto).

2. ¿Cómo funciona técnicamente? (El modelo en dos partes)

Cualquier Red Neuronal Convolutinal (como **MobileNet**) se divide en dos secciones:

A. El "Cuerpo" o Extractor de Características (Capas Convolucionales)

Son las capas iniciales. Han visto millones de imágenes y han aprendido a reconocer:

- Líneas y bordes.
- Texturas y sombras.
- Formas básicas (círculos, cuadrados). **Este conocimiento es universal.** Una línea en una foto de un gato es igual a una línea en una foto de una pieza de motor.

B. La "Cabeza" de Clasificación (Capas Densas/Perceptrón)

Es la parte final del modelo. Es la que toma las formas detectadas por el "cuerpo" y decide: "*Esto es un gato*".

El Truco del Transfer Learning: Cortamos la "cabeza" original (la que sabía reconocer 1,000 objetos de ImageNet) y le ponemos una **cabeza nueva** diseñada por nosotros para reconocer nuestros propios objetos (ej. "Taza", "Gafas", "Llaves").

3. Conceptos clave que menciona tu texto

Ventajas Principales

1. **Velocidad:** Solo entrenas la "cabeza" nueva. El "cuerpo" está **congelado** (sus pesos no cambian). Esto hace que el entrenamiento dure segundos o minutos en lugar de días.
2. **Menos datos:** Como el modelo ya sabe "ver", solo necesitas unos pocos ejemplos (quizás 30 fotos) de tu nuevo objeto para que aprenda.
3. **Ideal para Navegadores:** Al requerir menos potencia, es perfecto para ejecutarlo en Chrome o en un móvil.

Modelos de "Feature Vector" vs. "Classification"

En **TensorFlow Hub** encontrarás dos versiones de MobileNet:

- **Classification Model:** El modelo completo (Cuerpo + Cabeza original). Te sirve si solo quieres reconocer lo que ya sabe (perros, barcos, etc.).
- **Feature Vector Model:** El modelo "decapitado". Solo tiene las capas convolucionales. Es el que **debes usar** para Transfer Learning porque te entrega los datos listos para conectarlos a tu propia red.

Graph vs. Layers Models

- **Graph Model:** Es muy rápido y eficiente, pero es más difícil de modificar una vez cargado.
- **Layers Model:** Es más flexible. Te permite "congelar" capas específicas y, al final, puedes guardar todo (el cuerpo de MobileNet + tu nueva cabeza) en un **único archivo** con `model.save()`.

4. El Proceso en 3 Pasos

1. **Cargar MobileNet** (sin la cabeza final).
2. **Pasar tus imágenes** por ese modelo para obtener los "Feature Vectors" (las características esenciales de tus fotos).
3. **Entrenar una pequeña red densa** (MLP) que aprenda a asociar esos vectores con tus etiquetas personalizadas.

Make Teachable Machine Using Transfer Learning

2. Navigate to the **index.html** page. Update the content within the body tags with the following changes that are explained below.

```
<body>

  <h1>Make your own "Teachable Machine" using Transfer Learning with MobileNet
v3 in TensorFlow.js using saved graph model from TFHub.</h1>

  <p id="status">Awaiting TF.js load</p>

  <video id="webcam" autoplay></video>

  <button id="enableCam">Enable Webcam</button>

  <button class="dataCollector" data-1hot="0" data-name="Class 1">Gather Class
1 Data</button>

  <button class="dataCollector" data-1hot="1" data-name="Class 2">Gather Class
2 Data</button>

  <button id="train">Train & Predict!</button>

  <button id="reset">Reset</button>

  <!-- Import TensorFlow.js library -->

  <script
src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@3.11.0/dist/tf.min.js"
type="text/javascript"></script>

  <!-- Import the page's JavaScript to do some stuff -->

  <script type="module" src="/script.js"></script>
```

```
</body>
```

- a. First, add a heading and paragraph tags with the id ‘status’.
This is where you will print information as you use different parts of the system to view outputs.
- b. Define a video element with the id ‘webcam’. Add the ‘autoplay’ attribute. Your webcam is rendered to this element later on.
- c. Define the following:
 - A button with the id ‘enableCam’ - This is to enable the webcam.
 - Two buttons with the class ‘dataCollector’ - These allow you to gather example images for the classes you define.
 - You will write code later on that is designed such that you can add any number of these buttons, and they will work as intended automatically.
 - These buttons also have special user-defined attributes. One is called ‘data 1-hot’ that takes integer values starting from 0 for the first class. integer values will be used in your 1-hot encoding later on.
 - These buttons also have a ‘data-name’ attribute. These represent the human-readable names that you wish to use for your classes.
 - A button with the id ‘train’ - This is used to start training your model.
 - A button with the id **reset** – This is used to clear all data if you want to start over.
- d. The code ends with two script imports, one for TensorFlow.js and one for the script.js code that you will add code to shortly.

The complete code for the **index.html** file is available at this [URL](#), which you can save, copy, and use on Glitch to save time writing it out.

3. Next, navigate to the **style.css** document and update it with the code provided below to position and size the document elements correctly.

```
width: 640px;  
  
height: 480px;  
  
}
```

The complete code for the **style.css** file is available at this [URL](#) to copy if you need.

See Figure 5.2.2. for how your live preview should look like at this point:

Make your own "Teachable Machine" using Transfer Learning with MobileNet v3 in TensorFlow.js using saved graph model from TFHub.

Awaiting TF.js load

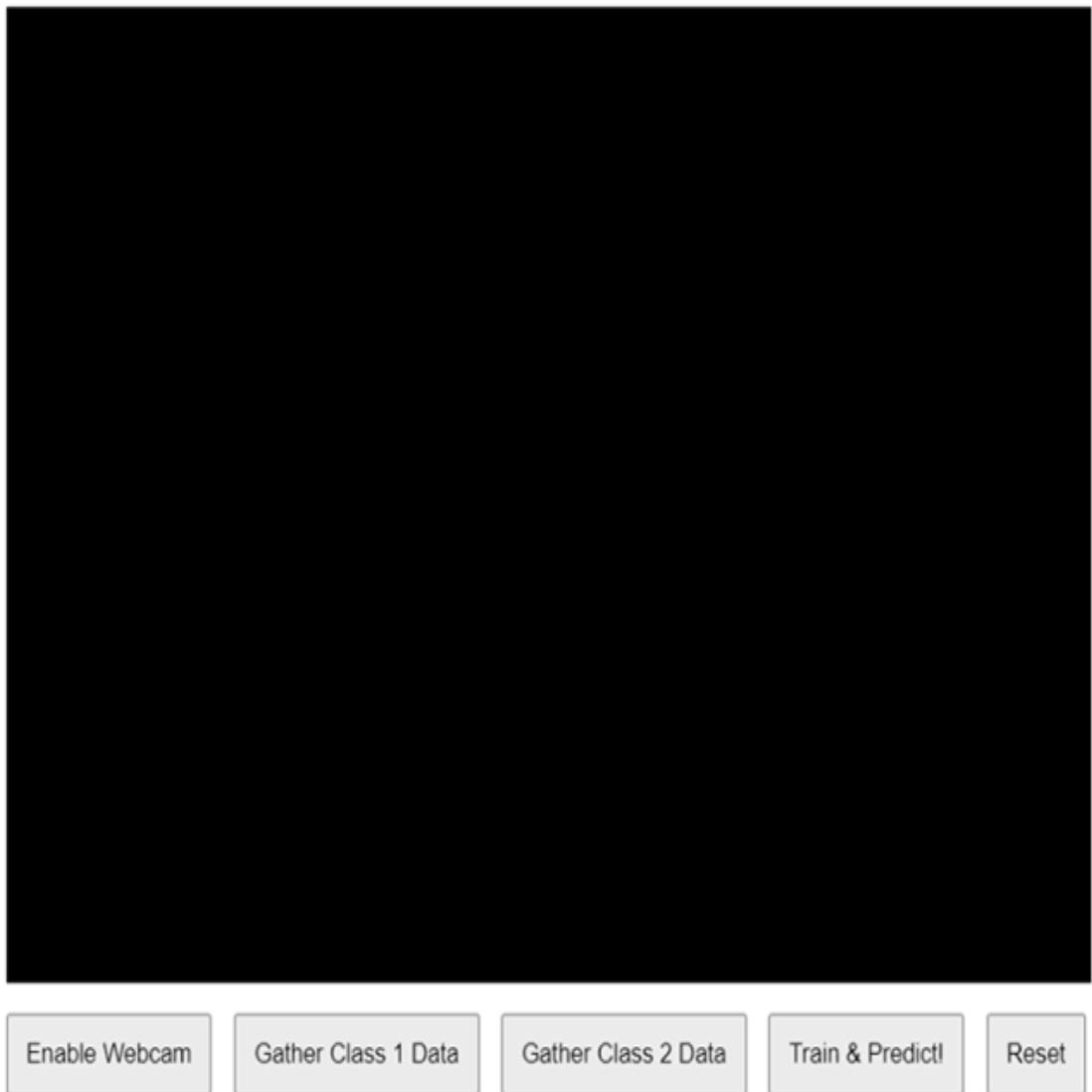


Figure 5.2.2. Make Your Own Teachable Machine - Live Preview

In the next unit, you start adding code to the **script.js** document.

1. Navigate to the **script.js** file. Start by adding key constants.

```
const STATUS = document.getElementById('status');

const VIDEO = document.getElementById('webcam');

const ENABLE_CAM_BUTTON = document.getElementById('enableCam');

const RESET_BUTTON = document.getElementById('reset');

const TRAIN_BUTTON = document.getElementById('train');

const MOBILE_NET_INPUT_WIDTH = 224;

const MOBILE_NET_INPUT_HEIGHT = 224;

const STOP_DATA_GATHER = -1;

const CLASS_NAMES = [];
```

- STATUS - This holds a reference to the paragraph tag which will contain status updates.
- VIDEO - This holds a reference to the ‘HTML’ video element that renders the webcam feed.
- ENABLE_CAM - This holds a reference to the button that enables the camera.
- RESET_BUTTON - This holds a reference to the button that resets the app to start over.
- TRAIN_BUTTON - This holds a reference to the button that begins training after data is gathered for each class.
- MOBILE_NET_INPUT_WIDTH and MOBILE_NET_OUTPUT_WIDTH - These store the expected input width and height for the MobileNet Model.

- The model you will use expects an input image of size 224x224 pixels.
 - Storing these values in a constant at the beginning allows you to update the values only once later on (if you are using a different version) instead of having to update them in many different places.
- STOP_DATA_GATHER - This constant is set to the value of -1.
-
- This stores a state value that tells you when a user has stopped clicking the button used to gather data.
 - CLASS_NAME - This is a simple array lookup that holds the human-readable names for the possible class predictions.
-
- This is populated later on in the app.
2. You next associate key event listeners to the elements that you created references to.

```
ENABLE_CAM_BUTTON.addEventListener('click', enableCam);

TRAIN_BUTTON.addEventListener('click', trainAndPredict);

RESET_BUTTON.addEventListener('click', reset);
```

```

let dataCollectorButtons =
document.querySelectorAll('button.dataCollector');

for (let i = 0; i < dataCollectorButtons.length; i++) {

    dataCollectorButtons[i].addEventListener('mousedown',
gatherDataForClass);

    dataCollectorButtons[i].addEventListener('mouseup',
gatherDataForClass);

    // Populate the human readable names for classes.

CLASS_NAMES.push(dataCollectorButtons[i].getAttribute('data-name'));

}

```

1. ENABLE_CAM_BUTTON - This will call the 'enableCam' function when clicked.
2. TRAIN_BUTTON - This will call the 'trainAndPredict' function when clicked.
3. RESET_BUTTON - This will call the 'reset' function when clicked.
4. Buttons with the class 'dataCollector':

- Use 'document.querySelectorAll' to find all buttons that have a particular class (in this case, the 'dataCollector' class). This returns an array of matched elements from the document that match this selector.
- You can then iterate through the found 'dataCollector' button array and associate two event listeners to each, one for the 'mousedown' event and one for the 'mouseup' event. Note that both events call a

‘gethetDataForClass’ function that you define later on in this lesson.

- At this point, you also populate the CLASS_NAMES array by pushing the human-readable class names from the HTML button attribute called ‘data-name’.

Note: By using a class here to find buttons instead of hardcoding a fixed number of ‘dataCollector’ buttons, you can change your HTML later when you need to add more buttons and the code will still function correctly for any number of buttons. You just need to update the data-1hot and data-name attributes for any new buttons you add.

3. In the next step, you add variables to store key elements that are used later.

```
let mobilenet = undefined;

let gatherDataState = STOP_DATA_GATHER;

let videoPlaying = false;

let trainingDataInputs = [];

let trainingDataOutputs = [];

let examplesCount = [];

let predict = false;
```

1. A variable called ‘mobilenet’ to store the loaded MobileNet model - this is initially set to ‘undefined’.
2. A ‘gatherDataState’ variable - when a ‘dataCollector’ button is pressed, this will change to be the 1-hot id of the button instead as defined in the HTML code.

- This is done so you know what class of data you are collecting when clicking a given button.
 - The value is initially set to 'STOP_DATA_GATHER' so that your data gather loop (that you write later on) will not gather any data when no button is being pressed.
3. A 'videoPlaying' variable that allows you to keep track if the webcam stream is successfully loaded and playing such that you can use it - The value is initially set to 'false' as the webcam is not on until you press the 'ENABLE_CAM_BUTTON'.
 4. Two arrays, with the names 'trainingDataInputs' and 'trainingDataOutputs' - These store the gathered training data values as you click the 'dataCollector' buttons.
 5. An array with the name 'examplesCount' - This will keep track of the number of examples sampled for each class.
 6. A variable called 'predict' that controls your prediction loop - This is set to 'false' initially. No predictions take place until the value of this variable is set to 'true'.

In the next unit, you load and compile the MobileNet v3 pre-chopped model. Remember, this provides an image feature vector as output instead of the final object classification.

Loading and Compiling the MobileNet v4 'feature vector' model

Steps

Add the following code to load and compile the model.

```
/**  
  
 * Loads the MobileNet model and warms it up so ready for use.  
  
 */  
  
async function loadMobileNetFeatureModel() {  
  
  const URL =  
  
    'https://tfhub.dev/google/tfjs-model/imagenet/mobilenet_v3_small_100_  
224/feature_vector/5/default/1';  
  
  mobilenet = await tf.loadGraphModel(URL, {fromTFHub: true});  
  
  STATUS.innerText = 'MobileNet v3 loaded successfully!';  
  
  // Warm up the model by passing zeros through it once.  
  
  tf.tidy(function () {  
  
    let answer = mobilenet.predict(tf.zeros([1,  
MOBILE_NET_INPUT_HEIGHT, MOBILE_NET_INPUT_WIDTH, 3]));  
  
    console.log(answer.shape);  
  
  });  
  
}  
}
```

```
loadMobileNetFeatureModel();
```

1. Start by defining a new function called ‘loadMobileNetFeatureModel’. This must be an asynchronous function.
2. From the TFHub documentation, define the URL at which the model you want to load is located.
3. Load the model using ‘await tf.loadGraphModel’. Remember to set the special property from ‘TFHub’ to ‘true’ since you are loading a model from this site.
4. Then set the value of the inner text in the STATUS paragraph element with a message to inform the user that the model has loaded correctly.

Note: Since this is a large model, it takes some time to set everything up. Therefore it’s prudent to pass zeroes as inputs through the model once before you start using it to avoid waiting for the initial setup to take place in the future where timing may be more critical.

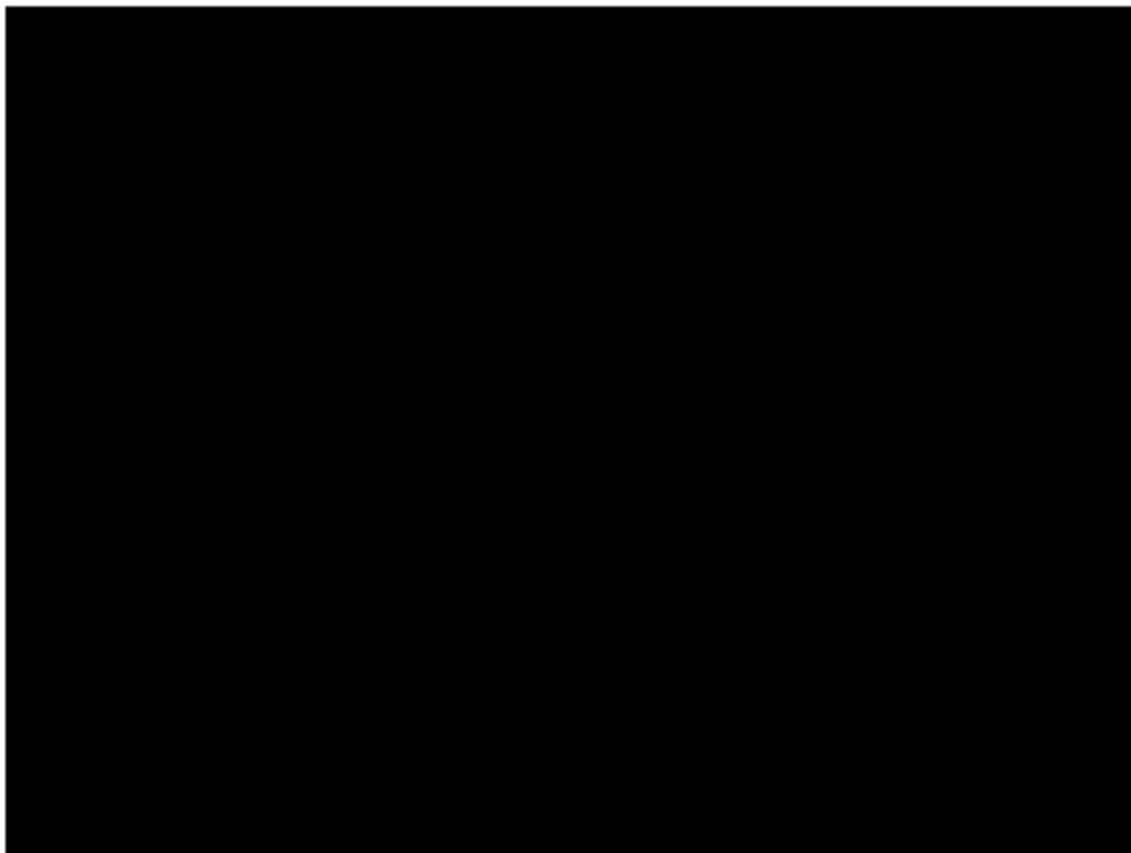
Use the ‘tf.zeros’ method with a batch size of 1 and the appropriate height and width and color channels (3, since the model expects RGB images). You may then log the resulting shape of the tensor (using ‘answer.shape’) to help you understand the size of the image feature vector this model produces as an output.

5. Finally, call the function you just created to start loading the model.

If you view the ‘live preview’, you should see the status change from ‘Awaiting TF.js load’ to ‘MobileNet v3 loaded successfully’ after a few moments. You can also check the console output that will have printed the size of the output features that this model produces. After running zeroes through the MobileNet model, you will see a shape of [1, 1024] printed.

This indicates that the model has 1024 features that can help you classify new objects.

MobileNet v3 loaded successfully!



Enable Webcam

Gather Class 1 Data

Gather Class 2 Data

Train & Predict!

Reset

Figure 5.2.3. Make Your Own Teachable Machine - Live Preview After Loading the Model

6. Next add code to define your model's new classification head (a multi-layer perceptron):

```
let model = tf.sequential();
```

```
model.add(tf.layers.dense({inputShape: [1024], units: 128,
activation: 'relu'}));

model.add(tf.layers.dense({units: CLASS_NAMES.length, activation:
'softmax'}));

model.summary();
```

1. Define a ‘tf.Sequential’ model.
2. Add a dense layer with the following parameters:
 - An input shape of 1024 (which you found was the size of the output of mobilenet in the previous step)
 - 128 neurons (units)
 - That use the ‘ReLU’ activation function
3. Add a final output layer with the following parameters:
 - The number of neurons (units) should equal the number of classes you are planning to classify. Set this to ‘CLASS_NAMES.length’, which will return the length of the array that contains your class names.
 - Use the ‘softmax’ activation function (since this is performing classification)
4. Print the model summary to the console to make sure everything is as expected.
7. In the next step, you compile your model and get it ready for training.

```
// Compile the model with the defined optimizer and specify a loss
function to use.

model.compile({

  // Adam changes the learning rate over time which is useful.

  optimizer: 'adam',
```

```
// Use the correct loss function. If 2 classes of data, must use  
binaryCrossentropy.  
  
// Else categoricalCrossentropy is used if more than 2 classes.  
  
loss: (CLASS_NAMES.length === 2 ? 'binaryCrossentropy':  
'categoricalCrossentropy',  
  
// As this is a classification problem you can record accuracy in  
the logs too!  
  
metrics: ['accuracy']  
  
));
```

1. Use the ‘model.compile’ function with the following settings:
2. Set the optimizer to ‘adam’. Remember, this is a highly useful optimizer since it changes the learning rate over time.
3. Set the ‘loss’ function to to be the correct type depending on the number of classes you have. In this case, you can use the ternary operator in JS to set it to be ‘binaryCrossEntropy’ if you have only two classes. If you have more than two classes, it will set the loss function to be ‘categoricalCrossEntropy’.
4. Add ‘accuracy’ metrics so that they can be monitored in the logs later.

At this point, if you check your console output, you will see that the model has over 130,000 trainable parameters. Since this is a dense layer of regular neurons, it will undergo training fast, unlike a full CNN model.

Layer (type)	Output shape	Param #
<hr/>		
dense_Dense1 (Dense)	[null,128]	131200
<hr/>		
dense_Dense2 (Dense)	[null,2]	258
<hr/>		
Total params: 131458		
Trainable params: 131458		
Non-trainable params: 0		

Figure 5.2.4. Make Your Own Teachable Machine - Model Summary After Compilation

In the next unit, you add code that deals with the webcam that is used to collect data. You also add code that allows the user to collect data when users press the ‘dataCollector’ buttons.

Steps


```
videoPlaying = true;

ENABLE_CAM_BUTTON.classList.add('removed');

});

});

} else {

console.warn('getUserMedia() is not supported by your browser');

}

}
```

1. Create a function to check if the browser supports ‘getUserMedia’ and allows access to the webcam.
2. Next, define the ‘enableCam’ function:
 1. First, check if getUserMedia is supported using the function you just defined above. If it isn’t, you can print a warning to the console.
 2. If it is supported, you can now define a few constraints for your getUserMedia call.
 - Specify the width of the video to be 640 and the height to be 480 pixels.
 - The video is going to be resized to 224 by 224 anyhow to be fed into the MobileNet model, so you may as well save some computing resources by requesting a smaller resolution - 640 by 480 is supported by most cameras.

3. Call 'navigator.mediaDevices.getUserMedia' with the constraints defined in the previous step and wait for the stream to be returned.
4. Set the video element to play the returned stream.
5. Add an event listener on the video element for the stream 'loadeddata' event. Remember to set the 'videoPlaying' variable value to 'true' once the stream loads.
6. Once the video is playing successfully, remove the ENABLE_CAM_BUTTON from the view to prevent it from being clicked again. This is done by setting its class to 'removed'.

Run your code at this point and click the enable camera button. If prompted, allow access to the webcam and you should then see yourself rendered to the video element on the page as shown:

MobileNet v3 loaded successfully!



Gather Class 1 Data

Gather Class 2 Data

Train & Predict!

Reset

Figure 5.2.5. Make Your Own Teachable Machine - Live Preview After Webcam is Enabled

7. Next, you add a function that deals with the ‘dataCollector’ button clicks:

```
/**  
 *  
 */  
  
* Handle Data Gather for button mouseup/mousedown.  
*/  
  
function gatherDataForClass() {  
  
    let classNumber = parseInt(this.getAttribute('data-1hot'));  
  
    gatherDataState = (gatherDataState === STOP_DATA_GATHER) ?  
        classNumber : STOP_DATA_GATHER;  
  
    dataGatherLoop();  
  
}  
}
```

1. Define a new function called ‘gatherDataForClass’. Remember, this is what you assigned as your event handler function for the ‘dataCollectorButtons’.
2. Use ‘this.getAttribute’ with ‘data-1hot’ as a parameter to grab the value of the ‘data-1hot’ attribute of the currently clicked button. As a string is returned, use ‘parseInt’ to cast the value to an integer and assign the result to a variable called ‘classNumber’.
3. Now you can set the value of the ‘gatherDataState’ variable.

- If the current ‘gatherDataState’ is equal to STOP_DATA_GATHER, which you set to be -1, it means you are not currently gathering any data, and this was a mousedown event that fired. Therefore you want to set the ‘gatherDataState’ to be the ‘classNumber’ you just found.
 - If the current ‘gatherDataState’ is NOT equal to STOP_DATA_GATHER, it means you are currently gathering data, and the event that fired must have been the mouseup event. In this case, you want to stop gathering data for that class, so you can set the value back to the STOP_DATA_GATHER state to reset it. This will end the data gathering loop (which you define in the next step).
4. Call the dataGatherLoop that will actually perform the recording of class data that is defined in the next step.

The ‘dataGatherLoop’ function - Code

This function is responsible for sampling images from the webcam video, passing them through the mobilenet model,

and capturing the outputs of the model - the 1024 feature vectors. It will then store them along with the ‘gatherDataState’ id of the button that is currently being pressed, which indicates the particular class this data represents. The code for this function is provided below followed by its explanation.

```
function dataGatherLoop() {  
  
  if (videoPlaying && gatherDataState !== STOP_DATA_GATHER) {  
  
    let imageFeatures = tf.tidy(function() {  
  
      let videoFrameAsTensor = tf.browser.fromPixels(VIDEO);  
  
      let resizedTensorFrame =  
        tf.image.resizeBilinear(videoFrameAsTensor, [MOBILE_NET_INPUT_HEIGHT,  
          MOBILE_NET_INPUT_WIDTH], true);  
  
      let normalizedTensorFrame = resizedTensorFrame.div(255);  
  
      return  
        mobilenet.predict(normalizedTensorFrame.expandDims()).squeeze();  
    });  
  
    trainingDataInputs.push(imageFeatures);  
  
    trainingDataOutputs.push(gatherDataState);  
  
    // Intialize array index element if currently undefined.  
  
    if (examplesCount[gatherDataState] === undefined) {  
  
      examplesCount[gatherDataState] = 0;  
    }  
  }  
}
```

```
examplesCount[gatherDataState]++;

STATUS.innerText = '';

for (let n = 0; n < CLASS_NAMES.length; n++) {

    STATUS.innerText += CLASS_NAMES[n] + ' data count: ' +
examplesCount[n] + '\n';

}

window.requestAnimationFrame(dataGatherLoop);

}

}
```

Steps

1. First check if the program is in a state where data should be gathered. The program should continue only if the ‘videoPlaying’ variable value is ‘true’ (webcam is active) and the ‘gatherDataState’ value is not equal to ‘STOP_DATA_GATHER’ (a button for gathering class data is being pressed).
2. Next, wrap your code in a ‘tf.tidy’ function to dispose of any created tensors. Store the result of the ‘tf.tidy’ code in a variable called ‘imageFeatures’.
3. Use ‘tf.browser.fromPixels’ to grab a frame of the webcam video. Store the resulting tensor in a variable called ‘videoFrameAsTensor’.
4. You now need to resize the ‘videoFrameAsTensor’ variable to be of the required shape for MobileNet.

- Use a ‘tf.image.resizeBilinear’ with the tensor you want to reshape as the first parameter and the new size you desire (defined by the MOBILE_NET_INPUT_HEIGHT and MOBILE_NET_INPUT_WIDTH constants) as the second parameter. Also set ‘align corners’ to ‘true’ as the third parameter as you have learnt before.
 - Store the result in a variable called ‘resizedTensorFrame’. Note that this will stretch the image as your webcam image is 640x480 pixels in size, where as ImageNet needs a square-shaped image that is 224x224 pixels in size. You may choose to try and crop a square from this image for even better results for your production systems but this trivial resize will be fine for now.
5. The next step is to normalize the image data. Since you have used ‘tf.browser.fromPixels’ the image data is always between the range 0 and 255. To normalize the data, you can simply divide the ‘resizedTensorFrame’ by 255 to ensure all resulting values lie between 0 and 1.
 6. You now call ‘mobilenet.predict’ and pass the expanded version of the ‘normalizedTensorFrame’ (use expandDims to convert it to a tensor2d to account for the batch of 1). You can then immediately squeeze the result to squash it back to a tensor1d which is returned and assigned to the ‘imageFeatures’ variable (which captures the results from the ‘tf.tidy’ function).
 7. You now have the image features from MobileNet. You can record them by pushing the resulting tensor onto the ‘trainingDataInputs’ array that you defined previously. Similarly, record what this input represents by pushing the current ‘gatherDataState’ values to the ‘trainingDataOutputs’ array. Remember, your ‘gatherDataState’ variable is set to the current class’s numerical ID you are recording data for when the button was clicked (in the ‘gatherDataForClass’ function defined previously).

8. At this point, you can also increment the number of examples you have for a given class. To do this, first check if the index within the 'examplesCount' array has been initialized. If it is undefined, you can set it to 0 to initialize the counter for a given class's numerical ID. You now increment the 'examplesCount' for the current 'gatherDataState'.
9. Now update the STATUS text on the webpage to show the current counts for each class as they're captured. This is done by looping through the CLASS NAMES array, and then printing the human-readable name combined with the data count at the same index in the 'examplesCount' array.
10. Finally, call 'window.requestAnimationFrame' with 'dataGatherLoop' passed as a parameter, to recursively call this function all over again. This will continue to sample frames from the video until the button mouseup is detected, and gatherDataState becomes STOP_DATA_GATHER at which point this data gather loop will end.

If you run your code at this point, you should be able to click the Enable Camera button, wait for the webcam to load, and then click and hold each of the data gather buttons to gather examples for each class. You should see the status text updated accordingly as it stores all the tensors in memory.

In the next unit, you will implement the 'train' and 'predict' function, which is where transfer learning takes place!

The 'trainAndPredict' function - Code

```
async function trainAndPredict() {  
  
  predict = false;  
  
  tf.util.shuffleCombo(trainingDataInputs, trainingDataOutputs);  
  
  let outputsAsTensor = tf.tensor1d(trainingDataOutputs, 'int32');  
  
  let oneHotOutputs = tf.oneHot(outputsAsTensor, CLASS_NAMES.length);  
  
  let inputsAsTensor = tf.stack(trainingDataInputs);  
  
  let results = await model.fit(inputsAsTensor, oneHotOutputs,  
    {shuffle: true, batchSize: 5, epochs: 10,  
     callbacks: {onEpochEnd: logProgress} }));  
  
  outputsAsTensor.dispose();  
  
  oneHotOutputs.dispose();  
  
  inputsAsTensor.dispose();  
  
  predict = true;  
  
  predictLoop();  
  
}  
}
```

```
function logProgress(epoch, logs) {  
  
    console.log('Data for epoch ' + epoch, logs);  
  
}
```

Steps

1. Stop any current predictions from taking place by setting ‘predict’ to ‘false’.
2. Shuffle your input and output arrays.
3. Convert your output array to be a tensor1d with type ‘int32’ so it is ready to be used in a 1-hot encoding. This is stored in a variable named ‘outputsAsTensor’.
4. Use the ‘tf.oneHot function’ with this ‘outputsAsTensor’ variable and the max number of classes to encode, which is just the length of the CLASS_NAMES array. Your 1-hot encoded outputs are now stored in a new tensor called ‘oneHotOutputs’.
5. ‘trainingDataInputs’ is currently an array of tensors. In order to use these for training, you will need to convert this into a regular tensor2d. To do this, use a function from the TensorFlow.js library called ‘tf.stack’, which takes an array of tensors and stacks them together to produce a single tensor as output. In this case, a tensor2d is returned, which is what you need for training - a batch of 1-dimensional inputs that are each 1,024 in length and contain the features recorded.
6. Now you can use ‘await model.fit’ as follows:

- Pass your ‘inputsAsTensor’ variable along with the ‘oneHotOutputs’ as parameters.
 - Set ‘shuffle’ to true
 - Use a batch size of 5 and set the number of epochs to 10.
 - Specify the ‘onEpochEnd’ callback to log progress, using the logProgress function.
7. Dispose of the created tensors as the model is now trained.
 8. Set ‘predict’ back to true to allow predictions to take place again.
 9. Call the ‘predictLoop’ function to start predicting live webcam images.

 10. Define the logProcess function to log the state of training.

In the next step, you complete coding for the ‘predictAndLoop’ function.

The ‘predictAndLoop’ function - Code

```
function predictLoop() {

  if (predict) {

    tf.tidy(function() {

      let videoFrameAsTensor = tf.browser.fromPixels(VIDEO).div(255);

      let resizedTensorFrame =
        tf.image.resizeBilinear(videoFrameAsTensor, [MOBILE_NET_INPUT_HEIGHT,
          MOBILE_NET_INPUT_WIDTH], true);

      let imageFeatures =
        mobilenet.predict(resizedTensorFrame.expandDims());
    });
  }
}
```

```
let prediction = model.predict(imageFeatures).squeeze();

let highestIndex = prediction.argmax().arraySync();

let predictionArray = prediction.arraySync();

STATUS.innerText = 'Prediction: ' + CLASS_NAMES[highestIndex] +
' with ' + Math.floor(predictionArray[highestIndex] * 100) + '%
confidence';

});

window.requestAnimationFrame(predictLoop);

}

}
```

Steps

1. Ensure that ‘predict’ is true, so that predictions are only made after a model is trained and is available to use.
2. Get the image features for the current image by grabbing a frame from the webcam using ‘tf.browser.fromPixels’, then normalize it by dividing its values by 255. This result will be stored in a variable called ‘videoFrameAsTensor’
3. Now resize the ‘videoFrameAsTensor’ variable to be 224 by 224 pixels in size and store the result in a variable named ‘resizedTensorFrame’.

4. At this point, you can now pass the ‘resizedTensorFrame’ variable through the MobileNet model to get the image feature vector which you will store in a variable named ‘imageFeatures’.
5. Use your newly trained model head to actually perform a prediction by passing ‘imageFeatures’ as an input to the model using the ‘model.predict’ function.
6. Use squeeze on the resulting output tensor to make it 1-dimensional again, and assign it to a variable called ‘prediction’.
7. With this ‘prediction’, you can find the index that has the highest value using the ‘argMax’ function and then convert this resulting tensor to an array sequentially using ‘arraySync’. This value is stored in a variable called ‘highestIndex’.
8. You can also get the actual prediction confidence scores in the same manner by calling ‘arraySync’ on the ‘prediction’ tensor directly and store the result in a variable named ‘predictionArray’.
9. You now have everything you need to update the status text with the predictions. To get the human-readable string for the class:
 1. Look up the ‘highestIndex’ in the CLASS_NAMES array.
 2. Grab the confidence value from the ‘predictionArray’ in the same manner using ‘highestIndex’ as the array index. To make it more readable as a percentage, multiply the resulting array value by 100 and use the ‘math.floor’ function on the result to round it down to the nearest whole number.
10. Once ready, use the ‘window.requestAnimationFrame’ to call the ‘predictionLoop’ all over again. This allows you to get real time classification on your video stream. It will continue until ‘predict’ is set

to false (which will be set to false if you choose to train a new model with new data).

Finally, in the next step, you will implement the code for the ‘reset’ button.

The ‘reset’ button - Code

```
/**  
  
 * Purge data and start over. Note this does not dispose of the  
loaded  
  
 * MobileNet model and MLP head tensors as you will need to reuse  
  
 * them to train a new model.  
  
 */  
  
function reset() {  
  
    predict = false;  
  
    examplesCount.splice(0);  
  
    for (let i = 0; i < trainingDataInputs.length; i++) {  
  
        trainingDataInputs[i].dispose();  
  
    }  
  
    trainingDataInputs.splice(0);  
}
```

```
    trainingDataOutputs.splice(0);

    STATUS.innerText = 'No data collected';

    console.log('Tensors in memory: ' + tf.memory().numTensors);

}
```

Steps

1. Stop any currently running prediction loops by setting ‘predict’ to false.
2. Delete all content in the ‘examplesCount’ array by using the ‘splice(0)’ method you learned in prior chapters.
3. Go through all the currently recorded ‘trainingDataInputs’ and ensure that you dispose of each tensor contained within it in order to free up memory again.
4. Next Call ‘splice(0)’ on both the ‘trainingDataInputs’ and ‘trainingDataOutputs’ arrays to clear those too.

Note: If you had called ‘splice(0)’ on the ‘trainingDataInputs’ array before disposing of the tensors contained within, the tensors would be unreachable but still in memory and not disposed of, which could cause a memory leak.

5. Set the status text to something appropriate and print out the tensors left in memory as a sanity check.

- Remember that since the MobileNet model and the multi-layer perceptron are not yet disposed of, a few hundred tensors will still be left in the device memory which is expected. This allows you to reuse them with new training data if you decide to train the model again after a reset.

Testing the complete demo:

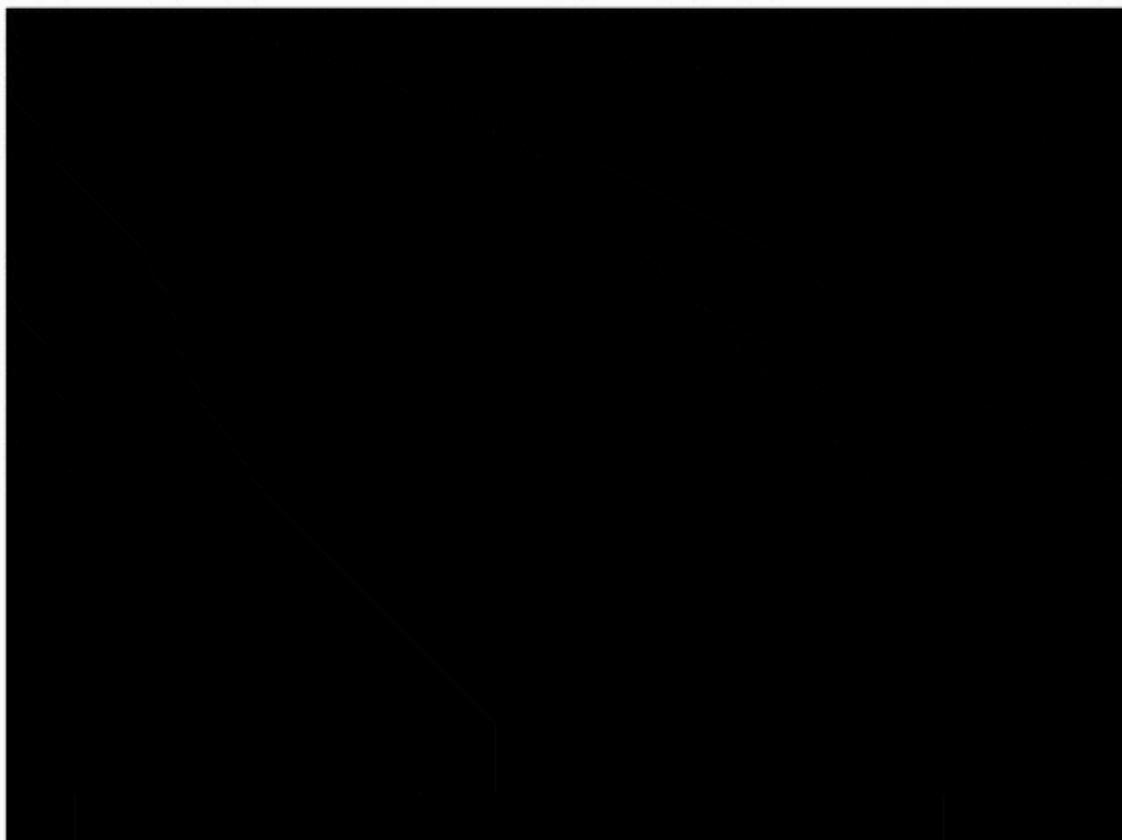
Enable the webcam in the live preview, gather at least 30 samples for class 1 for some object in your room and then do the same for class 2 for a different object.

Press the train button and check the console log to see progress.

Once the model is trained, show the objects to the camera to get live predictions that will be printed to the status text area on the web page. If you have trouble, check the completed working code on this [link](#) to see if you have missed anything.

Make your own "Teachable Machine" using Transfer Learning with MobileNet v3 in TensorFlow.js using saved graph model from TFHub.

MobileNet v3 loaded successfully!



Enable Webcam

Gather Class 1 Data

Gather Class 2 Data

Train & Predict!

Reset

Currently, if you want to use the resulting trained model, you will need to load two models:

1. The mobilenet base model used to generate the feature vectors
2. The trained multilayer perceptron head you just trained

If the base model was in fact a layers model, you could combine them after training is complete to save just one model. In the next section, you will see how to use ‘layers’ based models to do this!

Transfer Learning with Layers Models

In the previous lesson, a pre-trained ‘feature vector’ graph model was used to generate features that you used for transfer learning with your own custom classification head. In this lesson, you learn how to use a ‘layers’ model as the base model for transfer learning instead.

Benefits of using a ‘Layers’ model for Transfer Learning

1. Using a layers model allows you to fine-tune the base model if needed.
2. You can choose for yourself what layers to freeze/unfreeze, thus providing you with greater flexibility to make more advanced models.
3. Layers models enable you to deploy a single combined resulting model to production instead of two separate models that depend on each other.

script.js Code: Navigate to the script.js file and edit code as shown that is explained below.

Steps:

1. Add a new variable called ‘mobileNetBase’ at the end of your variables definitions near the top of the file. Set it to ‘undefined’. This is what you will use to store the chopped-up MobileNet model later on.
2. Define a new function called ‘customPrint’ that takes one parameter - a line of text that you wish to print.
 - This function takes the string that is passed, creates a new paragraph element, sets the text to be the string, and then adds it to the end of the document body.
 - Sometimes it can be useful to define your own printing functions instead of using the console and you will see this function in use later.

```
let mobilenet = undefined;

let gatherDataState = STOP_DATA_GATHER;

let videoPlaying = false;

let trainingDataInputs = [];

let trainingDataOutputs = [];

let examplesCount = [];

let predict = false;

let mobileNetBase = undefined;
```

```
function customPrint(line) {  
  
    let p = document.createElement('p');  
  
    p.innerText = line;  
  
    document.body.appendChild(p);  
  
}  

```

3. Update the 'loadMobileNetFeatureModel ()' as explained below.

```
async function loadMobileNetFeatureModel() {  
  
    const URL =  
        'https://storage.googleapis.com/jmstore/TensorFlowJS/EdX/SavedModels/  
        mobilenet-v2/model.json';  
  
    mobilenet = await tf.loadLayersModel(URL);  
  
    STATUS.innerText = 'MobileNet v2 loaded successfully!';  
  
    mobilenet.summary(null, null, customPrint);  
  
}  

```

- a. First, change the URL to load a MobileNet v2 model that is in the 'layers' model format.
- b. Use the 'await tf.loadLayersModel' instead of the 'loadGraphModel' to load this new model.

- c. Assign this model to a variable called ‘mobilenet’.
- d. Update the status text with the updated string of “mobilenet v2 loaded successfully”.
- e. Call ‘mobilenet.summary’, but this time with three parameters. Set the first two to null, and pass the ‘customPrint’ function that you defined as the third parameter. The model summary will now use your custom function to print itself out instead of going to the console.

If you run your code, you will see the model layers printed at the end of the body in the HTML as shown.

Layers model transfer Learning with MobileNet v2 in TensorFlow.js

MobileNet v2 loaded successfully!



1. First enable webcam and allow access when asked.
2. Now find an object, point cam at it, click and hold gather class 1 data button to gather at least 30 samples.
3. Repeat for class 2 with a different object of interest. Get similar number of samples.
4. Click train and predict and wait while the model is trained live in your browser. No data is sent to server.
5. Once trained you will see live predictions appear above the video for what it thinks it sees.

```
Layer (type) Output shape Param # Receives inputs
=====
input_2 (InputLayer) [null,224,224,3] 0

Conv1 (Conv2D) [null,112,112,32] 864 input_2[0][0]

bn_Conv1 (BatchNormalization) [null,112,112,32] 128 Conv1[0][0]
```

Figure 5.3.2. Console Output after Loading MobileNet v2 Layers Model

4. Scroll down to the end of the output to find the name of the penultimate layer before the classification layers begin. Observe that the name of this layer is 'global_average_pooling2d_1'. You will understand how this name is used in the subsequent code and description.

```
block_16_project (Conv2D) [null,7,7,320] 307200 block_16_depthwise_relu[0][0]
```

```
block_16_project_BN (BatchNorma [null,7,7,320] 1280 block_16_project[0][0]
```

```
Conv_1 (Conv2D) [null,7,7,1280] 409600 block_16_project_BN[0][0]
```

```
Conv_1_bn (BatchNormalization) [null,7,7,1280] 5120 Conv_1[0][0]
```

```
out_relu (ReLU) [null,7,7,1280] 0 Conv_1_bn[0][0]
```

```
global_average_pooling2d_1 (Glo [null,1280] 0 out_relu[0][0]
```

```
predictions (Dense) [null,1000] 1281000 global_average_pooling2d_1[0][0]
```

```
=====
```

```
Total params: 3538984
```

```
Trainable params: 3504872
```

```
Non-trainable params: 34112
```

Figure 5.3.3. MobileNet v2 Layers Model - Penultimate Layer

```
async function loadMobileNetFeatureModel() {  
  
  const URL =  
    'https://storage.googleapis.com/jmstore/TensorFlowJS/EdX/SavedModels/  
    mobilenet-v2/model.json';  
  
  mobilenet = await tf.loadLayersModel(URL);  
  
  STATUS.innerText = 'MobileNet v2 loaded successfully!';  
  
  mobilenet.summary(null, null, customPrint);  
  
  const layer = mobilenet.getLayer('global_average_pooling2d_1');  
  
  mobileNetBase = tf.model({inputs: mobilenet.inputs, outputs:  
    layer.output});  
  
  mobileNetBase.summary();  
  
  // Warm up the model by passing zeros through it once.  
  
  tf.tidy(function () {  
  
    let answer = mobileNetBase.predict(tf.zeros([1,  
      MOBILE_NET_INPUT_HEIGHT, MOBILE_NET_INPUT_WIDTH, 3]));  
  
    console.log(answer.shape);
```

```
});  
}  
}
```

5. Below the ‘mobilenet.summary’ call, grab a reference to the penultimate layer. Do this by calling ‘mobilenet.getLayer’ and then pass the string of the layer name that you want, in this case ‘global_average_pooling2d_1’. Store the result in a constant called ‘lastLayer’.
6. Create a truncated version of the model by calling ‘tf.model’ and pass an object to this function that specifies the inputs’ starting point and the outputs’ ending point. Note that the inputs are the same as the original inputs, so use ‘mobilenet.inputs’ to refer to those and for the outputs you can use ‘lastLayer.outputs’. Store the result of this chopped up model in a ‘mobileNetBase’ variable.
7. Print the mobileNetBase’s summary to the console or use the print function you created earlier if you prefer.
8. Also further down, change the name of the model you send zeros through to warm up the correct model from ‘mobilenet’ to ‘mobileNetBase’. Since this v2 model uses the same input shape as the MobileNet v3 model used in the previous session, the remaining code does not change.
9. Run the code at this point and switch to the console. You can see the output once the mobilenet model is loaded and chopped up. Note that the last layer in this truncated version is the ‘global_average_pooling2d_1’ layer and the dense layer no longer exists. You can also see that the output

shape of this model is [1, 1280] meaning a batch of 1 in this case with 1280 values.

You learn how to update the code in the next unit.

Steps (contd.)

10. Update your multi-layer perceptron to take an inputShape of 1280 given that this is the size of the new output from the chopped up ‘mobileNetBase’ model you just created. The rest of the code in this section stays the same as before.

```
let model = tf.sequential();

model.add(tf.layers.dense({inputShape: [1280], units: 64, activation:
'relu'}));

model.add(tf.layers.dense({units: CLASS_NAMES.length, activation:
'softmax'}));
```

11. Next head to the ‘calculateFeaturesOnCurrentFrame’ function that performs the data gathering when you click the buttons. Ensure you update the return line to use ‘mobileNetBase’, instead of the original ‘mobilenet’ model.

```
function calculateFeaturesOnCurrentFrame() {  
  
  return tf.tidy(function() {  
  
    // Grab pixels from current VIDEO frame.  
  
    let videoFrameAsTensor = tf.browser.fromPixels(VIDEO);  
  
    // Resize video frame tensor to be 224 x 224 pixels which is  
    // needed by MobileNet for input.  
  
    let resizedTensorFrame = tf.image.resizeBilinear(  
  
      videoFrameAsTensor,  
  
      [MOBILE_NET_INPUT_HEIGHT, MOBILE_NET_INPUT_WIDTH],  
  
      true  
    );  
  
    let normalizedTensorFrame = resizedTensorFrame.div(255);  
  
    return  
      mobileNetBase.predict(normalizedTensorFrame.expandDims()).squeeze();  
  });  
}
```

```
}
```

12. Now you can update the code in the ‘trainAndPredict’ function. You may decrease the number of epochs to just five since the model converges rather quickly.

```
async function trainAndPredict() {  
  
  predict = false;  
  
  tf.util.shuffleCombo(trainingDataInputs, trainingDataOutputs);  
  
  let outputsAsTensor = tf.tensor1d(trainingDataOutputs, 'int32');  
  
  let oneHotOutputs = tf.oneHot(outputsAsTensor, CLASS_NAMES.length);  
  
  let inputsAsTensor = tf.stack(trainingDataInputs);  
  
  let results = await model.fit(inputsAsTensor, oneHotOutputs, {  
    shuffle: true,  
    batchSize: 5,  
    epochs: 5,  
    callbacks: {onEpochEnd: logProgress}  
  });  
}
```

```
});  
  
outputsAsTensor.dispose();  
  
oneHotOutputs.dispose();  
  
inputsAsTensor.dispose();
```

13. Further down the function's code, add extra logic to the 'trainAndPredict' function to now combine the two models that you have in memory and offer the combined model for download once complete.

```
outputsAsTensor.dispose();  
  
oneHotOutputs.dispose();  
  
inputsAsTensor.dispose();  
  
predict = true;  
  
  
  
// Make combined model for download.  
  
let combinedModel = tf.sequential();  
  
combinedModel.add(mobileNetBase);  
  
combinedModel.add(model);
```

```
combinedModel.compile({  
  
    optimizer: 'adam',  
  
    loss: (CLASS_NAMES.length === 2) ? 'binaryCrossentropy' :  
        'categoricalCrossentropy'  
  
});  
  
combinedModel.summary();  
  
await combinedModel.save('downloads://my-model');  
  
predictLoop();  
  
}  
}
```

- a. Create a new 'tf.sequential' model and assign it to a variable called 'combinedModel'
- b. Use the 'add' function to add the 'mobileNetBase' first, followed by adding the prediction head model you just trained.
- c. Initialize this model with the same optimizer and loss as you did before.
- d. Print a summary of this new combined trained model. At this point, you can also save it to the user's hard drive using the save function.

Running the code at this point will allow you to use transfer learning to train a new prediction head for any object you want to recognize. Once trained, a new combined model will be created and offered for download along with the weights files, that you can save to your hard drive. You can then load that saved model on any other website or send it to someone else to use.

After the download is complete, the web app will then start performing live predictions using the newly trained model as before so you can test that it works well. See the completed code if you have any issues.

Fine-Tuning the Base Model

In this project, you trained the prediction head from the image features generated by the truncated MobileNet model. This keeps the truncated MobileNet model separate when training, so its weights are not changed when you train the new prediction head.

However, sometimes it is desirable to carefully calibrate the values of the last few layers of the base model after a prediction head has been trained. This is known as fine-tuning.

In this case, another approach is to force all layers in the base model to not be trainable by setting their trainable property to false as shown below. At the same time, you can grab a reference to the layers that you want to fine-tune later on by checking their names and pushing the ones that match onto an array to use later.

```
let fineTuningLayers = [];

for (const layer of mobileNetBase.layers) {
    layer.trainable = false;

    if (layer.name.indexOf('layer_name_of_interest') === 0) {
        fineTuningLayers.push(layer);
    }
}

}
```

You can now combine the new prediction head with the truncated base model as you did before, and train like normal, as only the prediction head will be trainable at this point.

Once trained you can then go back to one of the fine-tuning layers and set its trainable property to true and then perform further training such that it also updates in the training process too, allowing you to fine-tune the model to potentially get slightly better results.

```
fineTuningLayers[fineTuneIndex].trainable = true;

trainAgain();
```

In the next chapter, you will learn about model conversion, where you use models from Python and convert them into TensorFlow.js layers models, which you can use in the browser.

Teachable Machine: Transfer Learning Avanzado (Graph vs Layers Models)

Este proyecto explora cómo crear modelos de visión personalizados en el navegador. La magia reside en el **Transfer Learning**, pero dependiendo del tipo de modelo base (**Graph** o **Layers**), nuestras capacidades cambian drásticamente.

¿Qué es el Transfer Learning?

Es la técnica de reutilizar un modelo experto (entrenado en millones de imágenes) para una tarea nueva. Aprovechamos que modelos como **MobileNet** ya saben extraer características universales (bordes, texturas) y solo entrenamos una "cabeza" final para nuestras clases personalizadas.

Comparativa: Graph Models vs. Layers Models

En este proyecto trabajamos con dos enfoques:

1. Graph Models (MobileNet v3)

- **Origen:** TensorFlow Hub.
- **Estado:** Es un modelo "cerrado" y optimizado para velocidad.
- **Uso:** Lo usamos como una caja negra que recibe píxeles y entrega un vector de **1,024 características**.
- **Limitación:** No podemos modificar sus capas internas fácilmente ni guardarla junto a nuestra cabeza de clasificación en un solo archivo.

2. Layers Models (MobileNet v2) - ¡Nuevo!

- **Origen:** Archivo `model.json`.
- **Ventaja:** Permite acceso total a la arquitectura. Podemos elegir exactamente dónde "cortar" el modelo.
- **Flexibilidad:** Podemos **congelar** o **descongelar** capas específicas para realizar **Fine-tuning** (ajuste fino).
- **Portabilidad:** Permite crear un `combinedModel` que fusiona la base y la cabeza en un único archivo descargable.

Métodos Clave y Proceso Técnico

A. Carga y Truncado (Layers Model)

Para usar MobileNet v2 como base, primero debemos eliminar sus capas de clasificación originales:

1. **Carga:** Usamos `tf.loadLayersModel(URL)`.
2. **Identificación:** Buscamos la capa penúltima (ej:
`global_average_pooling2d_1`).

Truncado: Creamos un nuevo modelo que use la entrada original pero termine en esa capa específica:

```
const layer = mobilenet.getLayer('global_average_pooling2d_1');
```

```
mobileNetBase = tf.model({inputs: mobilenet.inputs, outputs: layer.output});
```

3.

B. Creación de la "Cabeza" Personalizada

Entrenamos un Perceptrón Multicapa (MLP) que recibe las características (1,024 para v3 o 1,280 para v2):

- **Entrada:** Vector de características.
- **Capas:** Densa (ReLU) -> Salida (Softmax).

C. El Modelo Combinado y Exportación

Una de las mayores ventajas de los **Layers Models** es la capacidad de unir las piezas:

```
let combinedModel = tf.sequential();

combinedModel.add(mobileNetBase); // El cuerpo de MobileNet

combinedModel.add(model);      // Nuestra cabeza entrenada

await combinedModel.save('downloads://my-model');
```

Esto genera un archivo que puedes compartir y usar en cualquier otra web sin necesidad de volver a entrenar.

Conceptos Avanzados: Fine-Tuning

A diferencia del entrenamiento básico donde la base está congelada, el **Fine-tuning** permite:

1. Entrenar primero la cabeza de clasificación.
2. "Descongelar" las últimas capas del modelo base
`(layer.trainable = true)`.

3. Entrenar de nuevo con una tasa de aprendizaje muy baja para que la base se adapte sutilmente a tus imágenes específicas.

Características de la Interfaz (Tailwind CSS)

- **Mirror Effect:** Video invertido para una experiencia de usuario natural.
- **Data Gathering:** Recolección de frames en tiempo real manteniendo pulsados los botones.
- **Monitor de Tensores:** Seguimiento estricto de la memoria con `tf.memory()` para evitar fugas de datos.

Este proyecto demuestra que el Transfer Learning no es solo usar modelos ajenos, sino saber cómo diseccionarlos y adaptarlos para crear soluciones portátiles y ultra-precisas.

QUIZ TRANSFER LEARNING:

Question 1

1 / 1 punto (calificado)

Which of the following statements apply to Transfer learning?

It involves retraining the lower base layers while keeping the classification head unchanged.

It involves training a new classification head (often a multi-layer perceptron) while freezing the lower level base layers.

It can sometimes involve unfreezing some base layers for fine-tuning.

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 2

1 / 1 punto (calificado)

Identify the TRUE statement about the MobileNet ‘feature vector’ model.?

On its own it can be used to classify images containing certain objects in imagesrect

Requires a classification head to be added to it and trained to identify new objects in images.

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 3

1 / 1 punto (calificado)

Identify the TRUE statements.

Graph models are highly optimized so can not be used in Transfer Learning.

Graph models are highly optimized but can still be used in Transfer Learning.

Transfer learning with TensorFlow.js graph models can result in creation of a single model that can be saved for use.

Transfer learning with TensorFlow.js layers models can result in creation of a single model that can be saved for use.

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 4

1 / 1 punto (calificado)

Which of the following is not an advantage of transfer learning?

It is faster to train a model using transfer learning than training a mode from scratch.

All the knowledge, features, and patterns learned by the model during training can now be applied to a similar task.

Since most of the features are already learned, you only need a few new example data points to train the classification head. This saves you time and resources.

Transfer learning allows reuse of any model for any other problem you may want to solve.

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 5

1 / 1 punto (calificado)

What is the correct way to use a graph model for transfer learning?

Gather new images, load the graph model, connect it to the new prediction head, and train both models with new image data gathered.

Load the graph model, gather new images, run images through the graph model to generate feature vectors, train prediction head on generated feature vector data.

Load the graph model, gather new images, train new prediction head on gathered images, and then combine with the graph model at the end.

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 6

1 / 1 punto (calificado)

If you want to fetch a reference to a specific layer of a layers model, which of the following is the correct way to do that?

model.fetchLayer(layerName);

model.obtainLayer(layerName);

model.getLayer(layerName);

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 7

1 / 1 punto (calificado)

Why is it advised to run data through larger models after they have been loaded even before you need to use the model?

To clean the model from previous data that was there

To run data through the model once to initialize the model so subsequent calls that matter are then faster.

To check that it works correctly

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 8

1 / 1 punto (calificado)

If you are working with an unknown model with poor documentation that outputs a feature vector you need to use but you are unsure of its size, what is the easiest way to check the output size?

Email the developer and hope they reply with the details that are missing

Pick a different model that has better documentation

Load the model, run some data through it, and print the output tensor shape to console

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 9

1 / 1 punto (calificado)

If you have 2 models, modelA and modelB, that you want to combine into one model to then save, which of the following code snippets would do that?

let newModel = tf.sequential(); newModel.add(modelA);
newModel.add(modelB);

let newModel = modelA.add(modelB);

let newModel = tf.merge(modelA, modelB);

correcto

Guardar

Guardar respuesta

Mostrar respuesta

Enviar

Has utilizado 1 de 2 intentos

Algunos problemas tienen opciones como guardar, restablecer, sugerencias o mostrar respuesta. Estas opciones aparecen después de oprimir el botón Enviar.

Question 10

1 / 1 punto (calificado)

If you have a reference to a layer in a model and want to freeze it so its weights are not updated when training, which of the following code snippets will do that?

layer.frozen = true;

layer.trainable = false

layer.freeze = true

correcto
Guardar
Guardar respuesta
Mostrar respuesta
Enviar
Has utilizado 1 de 2 intentos

Reusing existing models from Python

What You Will Learn

By the end of the chapter, you will be able to:

1. Demonstrate and describe how to convert and use pre-made models from Python.
2. Explain and implement how to retrain a Natural Language Processing (NLP) model to solve the task of spam detection.
3. Demonstrate how to convert saved models to the TensorFlow.js format.
4. Demonstrate how to retrain an NLP model to deal with edge cases and use that in a functional website.

What the Chapter Is About

Academic machine learning research at the time of creating this course is often written in TensorFlow Python due to folk at Universities having familiarity with that language, yet for production use cases in the real world, it can be useful to take such resulting models and use them in the browser or the wider TensorFlow.js ecosystem to get greater visibility or enable re-use across a wider range of industries.

In this chapter you will learn how to take pre-trained Python saved models and convert them to the TensorFlow.js model format for use in the web browser. In addition, you will also learn how to retrain and modify an existing Python model using a “Colab” (not “codelab”) to then save a new Python model that you can then convert to the TensorFlow.js format too - allowing you to make use of existing Python pipelines to produce modified versions of existing models that better suit your needs.

This allows you to leverage the reach and scale of the web since there are many more web developers than machine learning practitioners. Converting Python saved models into TensorFlow.js models can enable more eyes on the latest research, better feedback, which in turn can drive the creation of better models with fewer bugs and biases.

You will start by learning how to use the TFJS command-line converter on an exported saved Python model such that it can be deployed in a browser.

You are then introduced to the common problem of spam classification using Natural Language Processing (NLP) where you will learn how to select an appropriate model for the task required. Here you will learn how to convert written sentences to numerical forms so you can use them as inputs to a machine learning model at the tensor level.

Then you will move on to getting hands-on experience in creating a real-world application by implementing a pre-trained comment spam detection model and using it in the browser to block spam comments on a fictitious but functional video blog website.

In this process, you are also exposed to the limitations of pre-made models and will attempt to overcome these by retraining the model and using the new model to deal with edge cases you will identify. You then will be able to demonstrate how to use the retained saved model and show the improvement in performance on the live website once deployed.

Converting Models From Python

In this course, you have used pre-made TensorFlow.js models and have also created custom models from scratch on a blank canvas that run on a browser. While these are powerful skills to have, you might find that many existing models in the current ML ecosystem may not be in the programming language of your choice. Many ML engineers and academics create models that are stored in the TensorFlow Python model format. These models do not run on the client-side in the browser, and if you want to make use of these models, you will need to know how to convert them to the TensorFlow.js format.

Note: If you want to use models saved in the TensorFlow Python format within Node.js (server-side), you **do not** need to convert these models, since Python and Node are both just wrappers around a C++ core that are interoperable.

Advantages of Converting Models From Python to the TensorFlow.js Format:

At the beginning of the course, you explored the benefits of executing ML models in the browser. These include increased inference speed, reduced costs, greater privacy, wide access to a great number of devices or users,

and direct access to sensor data. In addition, by converting, you get access to:

1. A larger range of models and cutting-edge research in a form that is more suitable for production.
2. A chance for a wider range of feedback on research models to help drive improvements to such models.

This is significant because there are far more people with web-browsers out there than there are people who know how to set up and code on a Linux server.

- a. To build models on the servers side, you would require an approach similar to the one shown in Figure 6.1.1. Compared to the vast number of users who have access to web browsers, the number of programmers who have the knowledge or time to set up the workflow for Python based TensorFlow is relatively small.

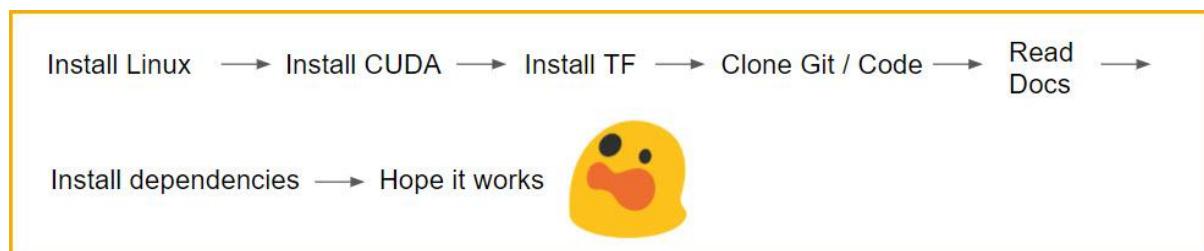


Figure 6.1.1 TensorFlow for Server Side Environments

- b. Using TensorFlow.js on the web, however, means anyone with a web browser can access these converted models and try them. Engineers and creatives alike can explore cutting-edge research in a very short period of time, before needing to dive deeper to set up more complex pipelines, thus enabling rapid ideation and prototyping of their ideas.

Visit a website → It works



Figure 6.1.2. TensorFlow.js on the Browser

- c. This also gets around the issue of many users having a programming language barrier to entry if they are not familiar with Python but are comfortable using JavaScript (of which 70% of professional developers do), who can then try out web-based models in their respective industries, which in turn drives innovation.
- d. The increased usage of these models results in ideas and use-cases that lead to valuable feedback to model creators since the models are exposed to a diverse spectrum of industries beyond the fields the researchers may have tested in. This feedback could lead to the discovery of potential biases in models that can then be addressed. Better models are then created down the line, industrial innovation happens at scale, and innovations are discovered for edge cases. This is in fact how the TensorFlow.js team works at Google whereby improvements to Pose Estimation models like MoveNet have been made after collaborating with real users in the health and fitness industries, which in turn resulted in even better models published for all to enjoy.

In this lesson, you use the TensorFlow.js command-line converter to convert models made from Python (with the extensions .pb or .h5) to the TensorFlow.js JSON format, thus allowing you to leverage the reach and scale of the web for your ML applications.

Introduction to Google Colab Notebooks

Python typically runs on the server-side environment via the command line terminal, often on the Linux operating system.

Google Colab (short for ‘Colaboratory’) was developed as an easy-to-use web interface where programmers could write and execute Python in the browser. The Colab environment allows you to combine executable code along with text and images so that you can take notes and share your findings and comments with others.

Each notebook contains snippets of code in cells that can be executed one after the other, with the notebook providing outputs after each step. The commands you type on the web page are sent to a real Linux server in the cloud to be executed. These outputs are then printed on the screen.

```
[1] import os
import tensorflow as tf

model = tf.keras.applications.mobilenet_v2.MobileNetV2(
    input_shape=(224, 224, 3), alpha=1.0, weights='imagenet',
    input_tensor=None, pooling=None, classes=1000,
    classifier_activation='softmax'
)

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet_v2_weights_tf_dim_ordering_tf_kernels_1.0_224.h5
14540800/14536128 [=====] - 0s 0us/step
14548992/14536128 [=====] - 0s 0us/step

[2] from tensorflow.python.saved_model import save
save_dir = os.path.join('/tmp', 'mobilenetv2/saved_model.h5')
model.save(save_dir)

WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. 'model.compile_metrics' will be empty until you train or evaluate the model.
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWarning: Custom mask layers require a config and must override get_config. When loading, the custom mask layer_config = serialize_layer_fn(layer)

```

```
!pip3 install tensorflowjs
Collecting tensorflowjs
  Downloading tensorflowjs-3.13.0-py3-none-any.whl (77 kB)
    77 kB 5.9 MB/s
Requirement already satisfied: tensorflow<1.12.0 in /usr/local/lib/python3.7/dist-packages (from tensorflowjs) (2.7.0)
Requirement already satisfied: six<1.12.0 in /usr/local/lib/python3.7/dist-packages (from tensorflowjs) (1.15.0)
Requirement already satisfied: tensorflow-hub<0.13.0 in /usr/local/lib/python3.7/dist-packages (from tensorflowjs) (0.12.0)
Requirement already satisfied: tensorflow<2.8.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>2.1.0>tensorflowjs) (2.7.0)
Requirement already satisfied: tensorflow-estimator<2.8.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>2.1.0>tensorflowjs) (2.7.0)
Requirement already satisfied: termcolor<1.1.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>2.1.0>tensorflowjs) (1.1.0)
Requirement already satisfied: gast<0.5.0,>=0.2.1 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>2.1.0>tensorflowjs) (2.7.0)
Requirement already satisfied: h5py<2.9.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>2.1.0>tensorflowjs) (0.4.0)
Requirement already satisfied: astunparse<1.6.0 in /usr/local/lib/python3.7/dist-packages (from tensorflow<3,>2.1.0>tensorflowjs) (3.1.0)
Requirement already satisfied: tensorflow<3,>2.1.0>tensorflowjs (1.6.3)
```

Figure 6.2.1. Sample Colab Notebook

Note on Linux:

Linux is an operating system that is a popular choice for server-side environments. Everything that can be executed in a Colab notebook can also be executed on a Linux server. A distribution of Linux called ‘Ubuntu’ is often used to power web servers running Node.js for websites. You may choose to use Linux, along with all the dependencies, to run the code for this chapter if you are familiar with working in this environment, either on the cloud or on your local device. If you are using Windows 10 or a later version, you can even install Ubuntu from the Microsoft Store and run it within the Windows environment with ease for free.

However, one major benefit of using a Colab is that it allows you to connect to a backend server that has already been set up with Python and TensorFlow configured on it. This saves you a lot of time, which you would have otherwise spent configuring if you were new to Linux and the

command line. In this chapter, you are going to work exclusively with Colab so everyone has a chance to follow along, although advanced users who are familiar with command line and Linux should be able to re-use these code snippets to run directly on the command line of a Linux server if desired.

Steps

1. Log in to your Google account and navigate colab.research.google.com.
2. Find the button to create a new notebook in the pop-up window that appears (see Figure 6.2.2.) and create a new project.

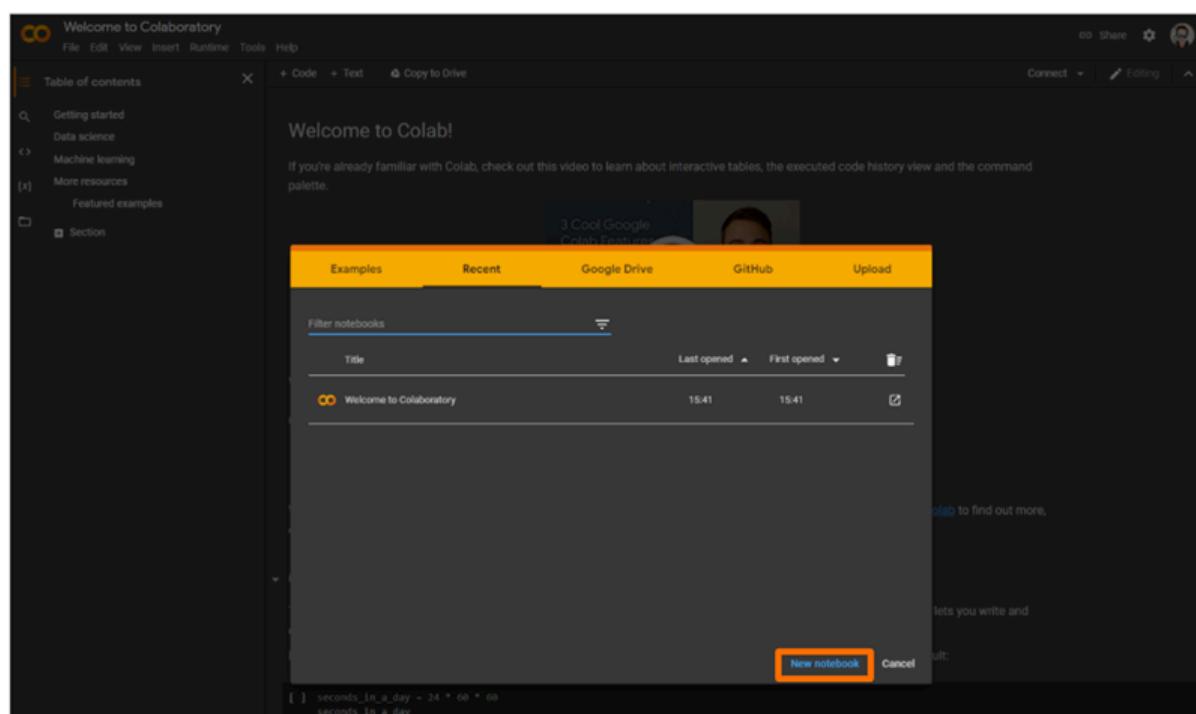


Figure 6.2.2 - Opening a New Colab Notebook

3. The new notebook should display an empty project page (See Figure 6.2.3.). Connect your session to a server-side backend that is used to execute any code you write. To do this, select the button named 'Connect' at the top right of the screen. The

notebook is now connected to a hosted runtime in the cloud.

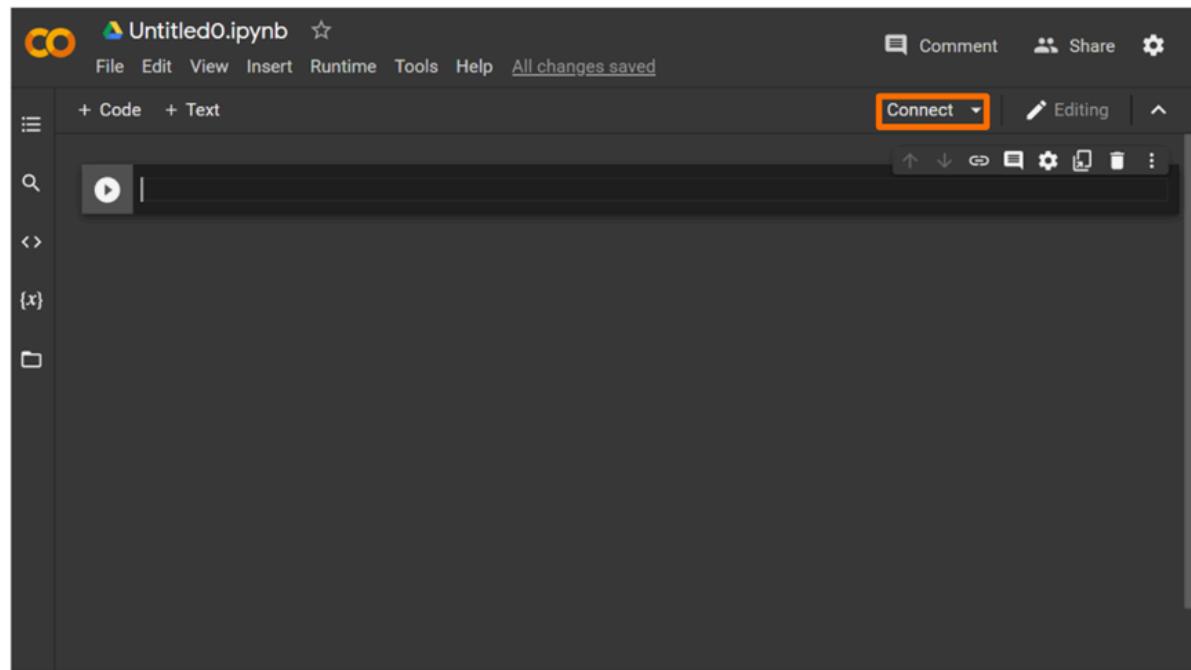


Figure 6.2.3 - Connecting Your Colab Session to a Backend

4. Once the notebook has successfully been allocated a back-end server, you will see the RAM and Disk usage (see Figure 6.2.4.). Your notebook can now execute Python and terminal commands.

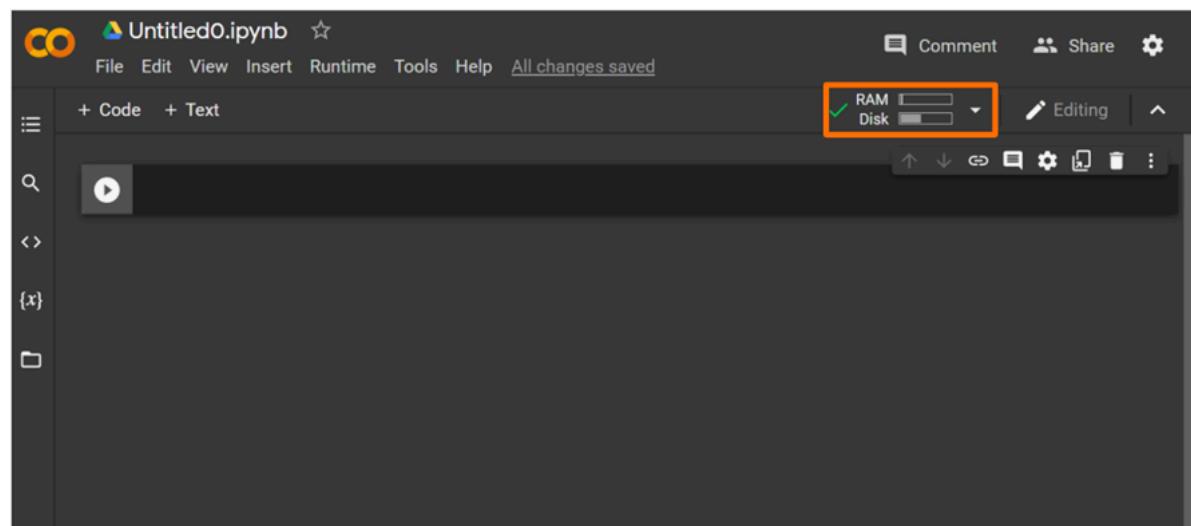


Figure 6.2.4 - Colab Notebook with Backend Connected

5. The main body of the notebook consists of a box with a play icon (see Figure 6.2.5.). This is the area where you write

Python code or regular terminal commands. You may also create multiple steps of execution or blocks of code. Usually, developers write a complete section of code to do a particular thing so you can step through the blocks one at a time and inspect various outputs at different stages of execution.

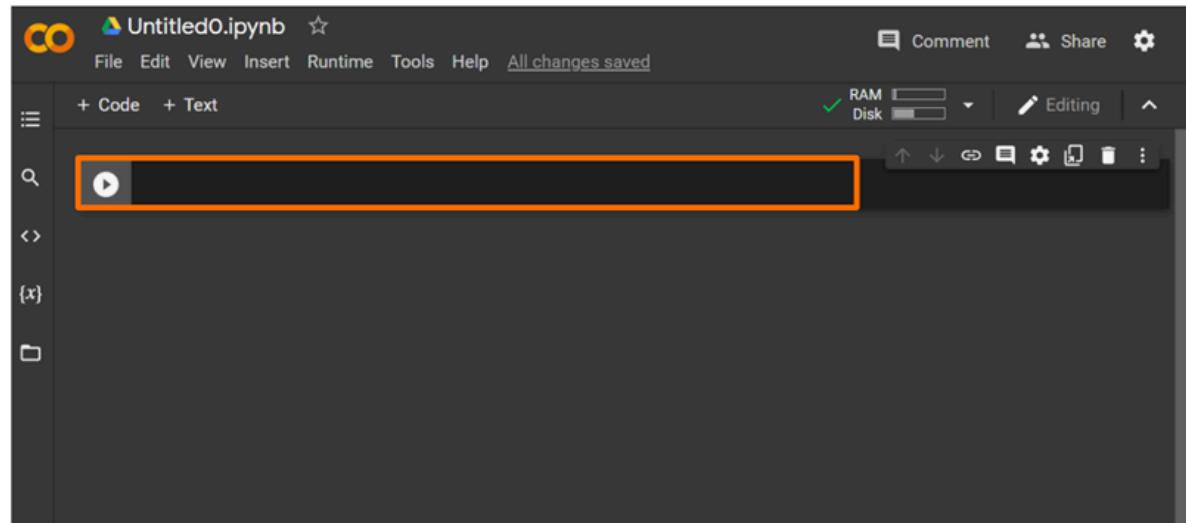


Figure 6.2.5. - Code Cells in Colab

6. Copy the code provided below into the Colab cell. It should look like Figure 6.2.6 once copied into the first cell block.

7.

```
import os  
  
import TensorFlow as tf
```

```
model = tf.keras.applications.mobilenet_v2.MobileNetV2 (
```



```
    input_shape=(224, 224, 3), alpha=1.0, weights='imagenet',
```



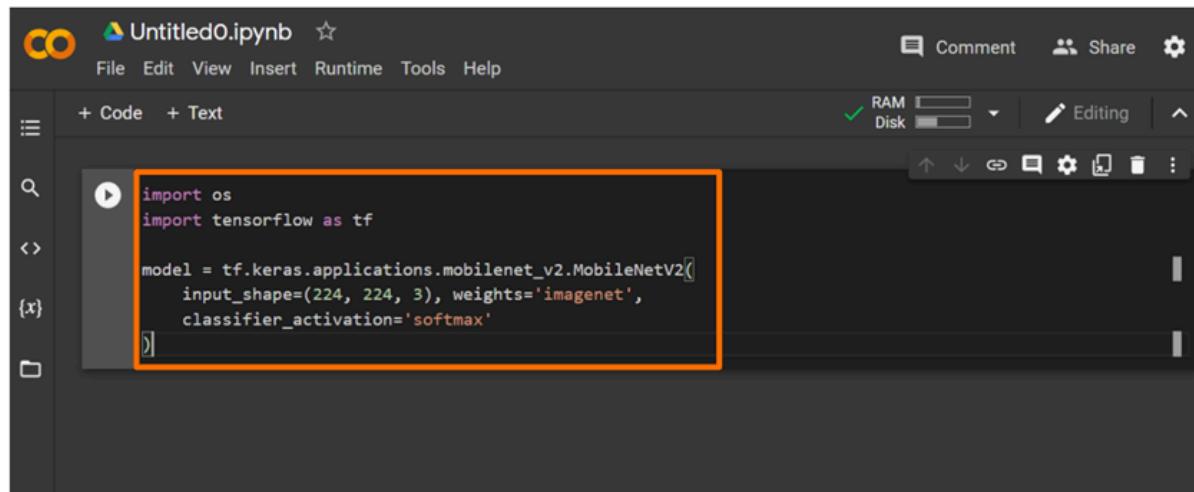
```
    input_tensor=None, pooling=None, classes=1000,
```



```
    classifier_activation='softmax'
```


)

8.



The screenshot shows the Google Colab interface with a dark theme. A code cell is selected and highlighted with an orange border. The code in the cell is:

```
import os
import tensorflow as tf

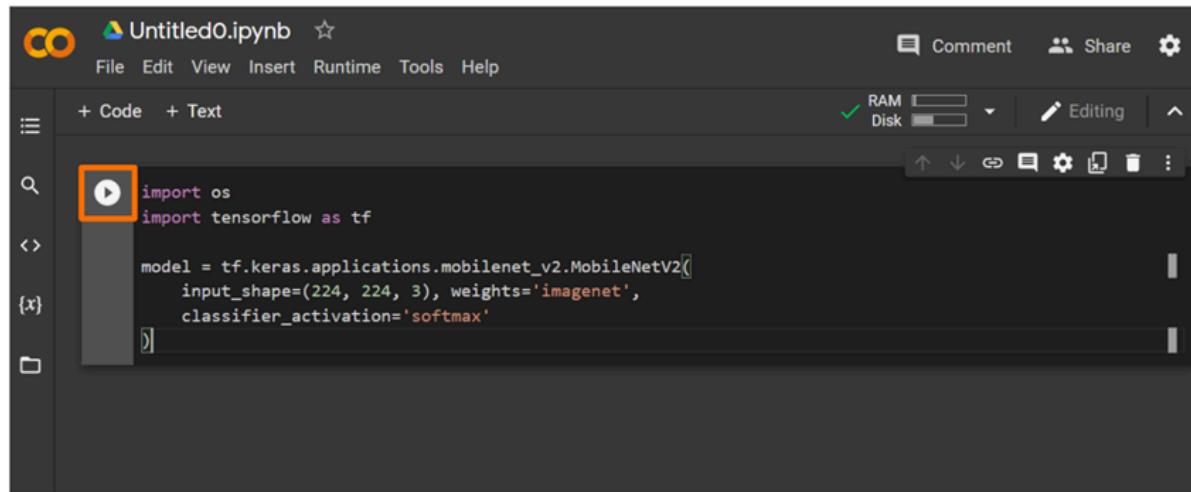
model = tf.keras.applications.mobilenet_v2.MobileNetV2([
    input_shape=(224, 224, 3), weights='imagenet',
    classifier_activation='softmax'
])
```

Figure 6.2.6 - Python code to copy into first Colab cell

The code essentially imports a few libraries and then goes on to access the MobileNet v2 model. The model is obtained

along with setting hyper-parameters such as the expected input shape, loading the default pre-trained weights for the model, and the classifier_activation function type.

9. Click the Play button on the left to execute the code. This imports the libraries that future code blocks will use, downloads the MobileNet v2 model, and stores it in a variable called 'model'.



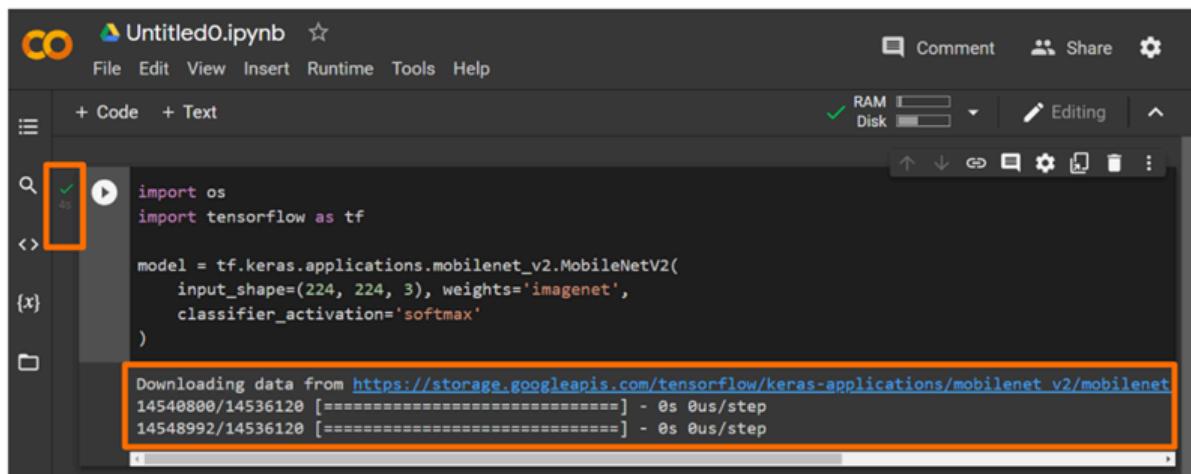
The screenshot shows the Google Colab interface with a code cell containing Python code. The code imports TensorFlow and defines a MobileNetV2 model with specific parameters. A red box highlights the play button icon in the toolbar above the code cell.

```
import os
import tensorflow as tf

model = tf.keras.applications.mobilenet_v2.MobileNetV2(
    input_shape=(224, 224, 3), weights='imagenet',
    classifier_activation='softmax')
```

Figure 6.2.7 - Executing Code in Colab

10. After a few seconds, the output appears below the cell. This is accompanied by a small green tick on the left hand of the code, indicating it has successfully executed. If you copied something incorrectly, you may see an error message printed instead.



The screenshot shows the Google Colab interface after the code has been executed. A green checkmark icon is visible next to the play button, indicating success. The output window at the bottom displays the progress of a data download from a Google Cloud storage URL, showing two steps completed with 0s execution time.

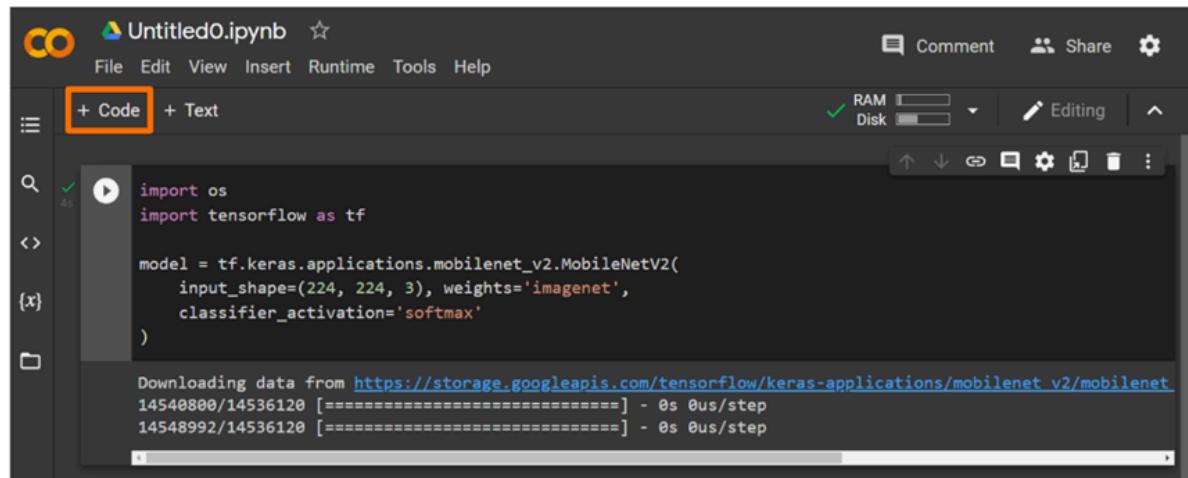
```
import os
import tensorflow as tf

model = tf.keras.applications.mobilenet_v2.MobileNetV2(
    input_shape=(224, 224, 3), weights='imagenet',
    classifier_activation='softmax')

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet
14540800/14536120 [=====] - 0s 0us/step
14548992/14536120 [=====] - 0s 0us/step
```

Figure 6.2.8 - Code Output in Colab

11. Next click on the ‘+ Code’ button (see Figure 6.2.9.) near the top left of the screen to add a new block. An empty code box will appear below the output of the first box see (Figure 6.2.10).



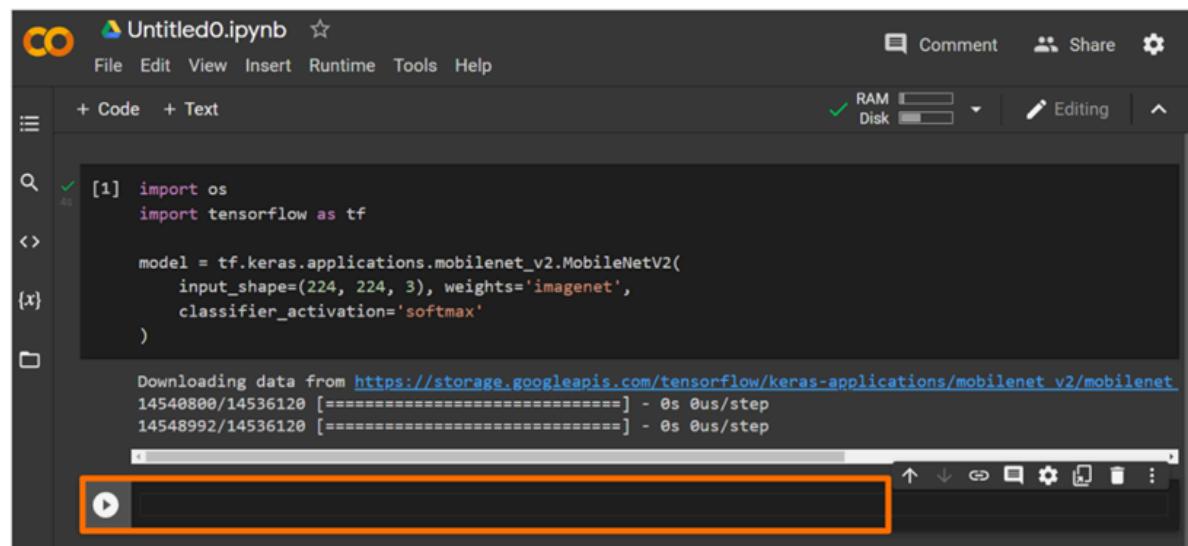
A screenshot of a Jupyter Notebook interface titled 'Untitled0.ipynb'. The menu bar includes File, Edit, View, Insert, Runtime, Tools, and Help. On the right, there are buttons for Comment, Share, and Settings, along with RAM and Disk status indicators. The toolbar has '+ Code' and '+ Text' buttons, with '+ Code' being highlighted with a yellow box. The code cell contains Python code to import TensorFlow and download a MobileNetV2 model. The output shows the download progress from Google Cloud Storage.

```
import os
import tensorflow as tf

model = tf.keras.applications.mobilenet_v2.MobileNetV2(
    input_shape=(224, 224, 3), weights='imagenet',
    classifier_activation='softmax'
)

Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet
14540800/14536120 [=====] - 0s 0us/step
14548992/14536120 [=====] - 0s 0us/step
```

Figure 6.2.9 - Adding More Code Blocks



A screenshot of a Jupyter Notebook interface titled 'Untitled0.ipynb'. The menu bar includes File, Edit, View, Insert, Runtime, Tools, and Help. On the right, there are buttons for Comment, Share, and Settings, along with RAM and Disk status indicators. The toolbar has '+ Code' and '+ Text' buttons. Below the toolbar, a new code cell has been added, indicated by a play button icon in a box, followed by an empty code block area.

Figure 6.2.10 - New Code Cell

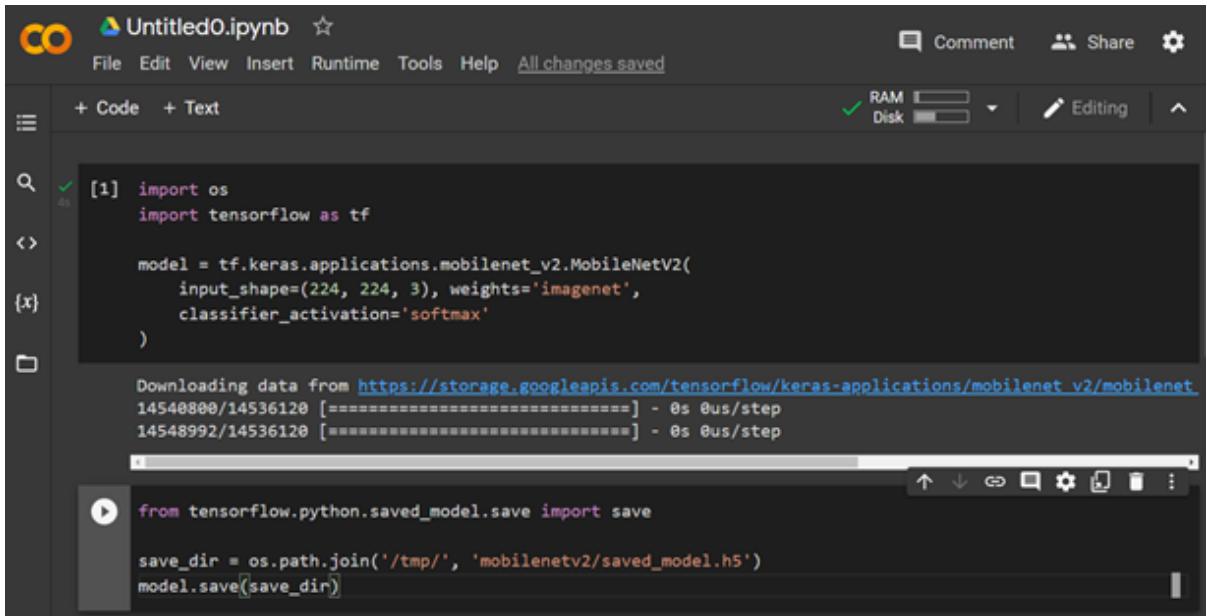
- Enter the Python code provided below into the new code block you added. It should look like Figure 6.2.1.1 once entered.

```
from tensorflow.python.saved_model import save

save_dir = os.path.join('/tmp/',
'mobilenetv2/saved_model.h5')

model.save(save_dir)
```

b.



The screenshot shows a Jupyter Notebook interface with the following details:

- Title:** Untitled0.ipynb
- Toolbar:** File, Edit, View, Insert, Runtime, Tools, Help, All changes saved
- Code Cell 1:** Contains code to import TensorFlow and define a MobileNetV2 model. It also includes a download progress bar for the model weights from Google Cloud Storage.
- Code Cell 2:** Contains code to save the model to disk at '/tmp/mobilenetv2/saved_model.h5'.

Figure 6.2.11 - Save Python Model to Disk

- c. This code will import the functions that allow you to save the model, and then create a folder called mobilenetv2 inside of the operating system's '/tmp' folder to which it will save the model.
- d. Your saved model file is named 'saved_model.h5'. This is a Python Keras saved model that uses the 'h5' format.

Note: Python models may be saved in either '.h5' or '.pb' formats. By exporting the current model to the '.h5' format, the TensorFlow.js converter is able to convert the model to a 'layers' model. On the other hand, exporting a model to the '.pb' format will result in a 'graph' model after conversion. You may choose the format that fits what you need - 'layers' for transfer learning ability, or 'graph' for speed of execution but less flexibility for modification later on.

10. Execute the second code cell which will save the model to disk. After a few seconds, you will see some output and the green tick when execution is complete. A warning message shows up at this point, but it can be safely ignored.
11. Next check the model was successfully saved. Select the file icon on the left bar (see Figure 6.2.12).

The screenshot shows a Jupyter Notebook interface titled 'Untitled0.ipynb'. The code cell [1] contains the following Python code:[1]:
model = tf.keras.applications.mobilenet_v2.MobileNetV2(
 input_shape=(224, 224, 3), weights='imagenet',
 classifier_activation='softmax'
)

Cell [x] shows the output of the code execution:Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/mobilenet_v2/mobilenet
14548800/14536120 [=====] - 0s 0us/step
14548992/14536120 [=====] - 0s 0us/step

Cell {x} contains the following code:{x}:
from tensorflow.python.saved_model import save

save_dir = os.path.join('/tmp', 'mobilenetv2/saved_model.h5')
model.save(save_dir)

Output for cell {x} shows a warning:WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have yet to be built. `model.compile` /usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1410: CustomMaskWarning: Custom mask layer_config = serialize_layer_fn(layer)

Figure 6.2.12 - Browse Remote Server Hard Drive

12. The file system explorer opens (see Figure 6.2.13), allowing you to browse the contents of the hard drive of the remote server to which you are connected.

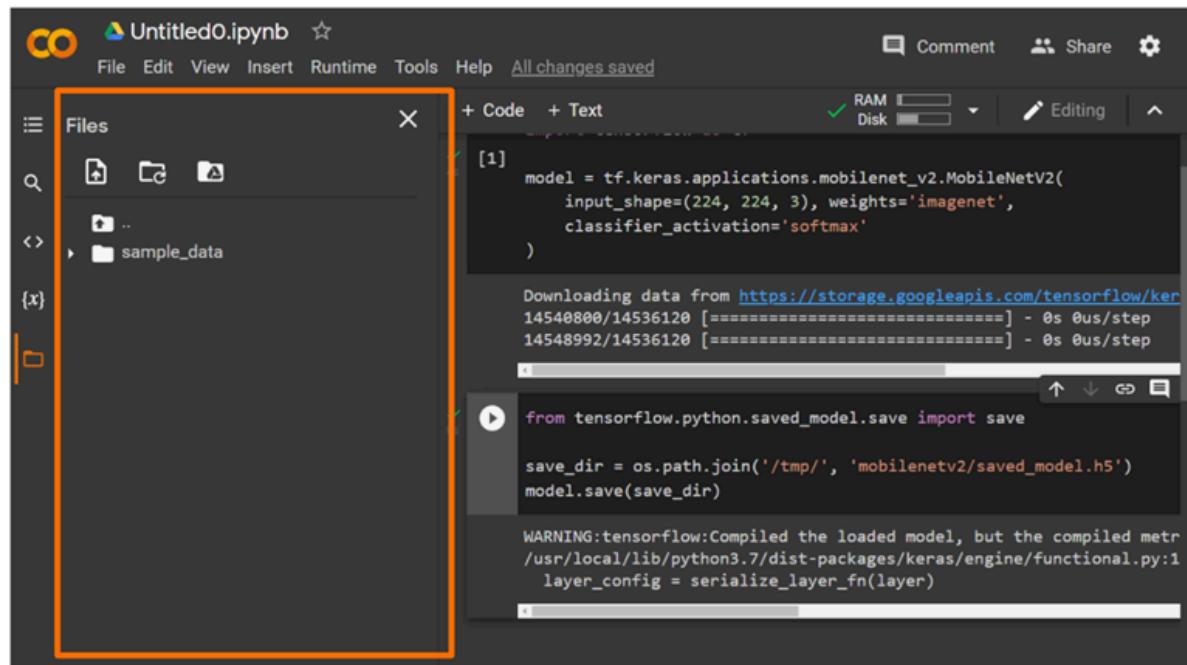


Figure 6.2.13 - Remote Hard Drive - File System Explorer

13. Navigate to the '/tmp' directory by selecting the folder icon that takes you to the parent folder one level higher (see Figure 6.2.14).

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Untitled0.ipynb
- File Menu:** File, Edit, View, Insert, Runtime, Tools, Help, All changes saved
- Toolbar:** Comment, Share, Settings
- Code Cell:** [1] contains Python code to load a MobileNetV2 model from TensorFlow's applications module. The code includes importing tensorflow, defining the model, and saving it to a temporary directory. A warning message is displayed about compiling the loaded model.
- Output Cell:** Shows the download of data from Google Cloud Storage and the successful saving of the model to '/tmp/mobilenetv2/saved_model.h5'.
- File Explorer:** On the left, it shows a folder structure with 'sample_data' selected. A specific file icon in the list is highlighted with an orange rectangle.

Figure 6.2.14 Remote Hard Drive - Temp Directory

14. You will find a lengthy list of folders on the Linux file system (see Figure 6.2.15). Expand the 'tmp' folder located near the bottom of the list (see Figure 6.2.16). Note that you may need to scroll down to view it depending on how tall your window is.

The screenshot shows a Jupyter Notebook interface with the following components:

- Title Bar:** Untitled0.ipynb
- File Menu:** File, Edit, View, Insert, Runtime, Tools, Help, All changes saved
- Comment and Share Buttons:** Comment, Share, Settings
- File Browser:** A sidebar titled "Files" containing a list of directory names. An orange rectangle highlights the entire list of files.
- Code Cell:** [1] contains Python code for loading a MobileNetV2 model from TensorFlow and saving it to a local directory. It also includes a warning message about compiled layers.
- Output Cell:** Shows the execution of the code, including the download of data from Google Cloud Storage and the creation of a saved model file.

Figure 6.2.15 - Remote Hard Drive - Linux File System

The screenshot shows a Jupyter Notebook interface with the title "Untitled0.ipynb". The left pane is a file browser titled "Files" showing the Linux file system structure. The "tmp" folder under "/tmp" is highlighted with an orange border. The right pane contains a code editor with a cell numbered [1] containing Python code to download and save a pre-trained MobileNetV2 model. The output area shows the download progress and a warning message about compiled metrics.

```
[1]
model = tf.keras.applications.mobilenet_v2.MobileNetV2(
    input_shape=(224, 224, 3), weights='imagenet',
    classifier_activation='softmax'
)

Downloading data from https://storage.googleapis.com/tensorflow/ker...
14540800/14536120 [=====] - 0s 0us/step
14548992/14536120 [=====] - 0s 0us/step

[2]
from tensorflow.python.saved_model import save

save_dir = os.path.join('/tmp', 'mobilenetv2/saved_model.h5')
model.save(save_dir)

WARNING:tensorflow:Compiled the loaded model, but the compiled metr...
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1
layer_config = serialize_layer_fn(layer)
```

Figure 6.2.16 - 'tmp' Folder in the Linux File System

2. Select the 'tmp' folder and find the 'mobilenetv2' folder that you created after executing the code (see Figure 6.2.17).

The screenshot shows a Jupyter Notebook interface. On the left, the 'Files' sidebar displays a directory tree. A folder named 'mobilenetv2' is selected and highlighted with an orange border. In the main notebook area, a code cell [1] contains Python code to download a pre-trained MobileNetV2 model and save it. The output of the cell shows the download progress and the successful creation of a 'saved_model.h5' file in the 'mobilenetv2' directory.

```
[1]
model = tf.keras.applications.mobilenet_v2.MobileNetV2(
    input_shape=(224, 224, 3), weights='imagenet',
    classifier_activation='softmax'
)

Downloading data from https://storage.googleapis.com/tensorflow/ker
14540800/14536120 [=====] - 0s 0us/step
14548992/14536120 [=====] - 0s 0us/step

from tensorflow.python.saved_model import save

save_dir = os.path.join('/tmp', 'mobilenetv2/saved_model.h5')
model.save(save_dir)

WARNING:tensorflow:Compiled the loaded model, but the compiled metr
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1
layer_config = serialize_layer_fn(layer)
```

Figure 6.2.17 - Finding the Python Saved mobilenetv2 Folder

Select the 'mobilenetv2' folder to see the newly created 'saved_model.h5' file (see Figure 6.2.18).

This screenshot is identical to Figure 6.2.17, showing the Jupyter Notebook interface. The 'mobilenetv2' folder is still selected in the 'Files' sidebar. The code cell [1] has run, and its output shows the 'saved_model.h5' file has been successfully created within the 'mobilenetv2' folder.

```
[1]
model = tf.keras.applications.mobilenet_v2.MobileNetV2(
    input_shape=(224, 224, 3), weights='imagenet',
    classifier_activation='softmax'
)

Downloading data from https://storage.googleapis.com/tensorflow/ker
14540800/14536120 [=====] - 0s 0us/step
14548992/14536120 [=====] - 0s 0us/step

from tensorflow.python.saved_model import save

save_dir = os.path.join('/tmp', 'mobilenetv2/saved_model.h5')
model.save(save_dir)

WARNING:tensorflow:Compiled the loaded model, but the compiled metr
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1
layer_config = serialize_layer_fn(layer)
```

Figure 6.2.18 - Finding the Saved h5 Model

If you see the saved file in this location you are ready to continue. In the next step, you will install TensorFlow.js Python utilities and

convert your saved h5 model to the TensorFlow.js format you are familiar with.

Installing TensorFlow.js Python Utilities

Steps:

1. Add another code block to your project and paste the code shown in Figure 6.2.19.

Note: This code block starts with an exclamation point. This indicates that it executes a terminal command instead of Python code. This command uses the ‘pip3’ package manager to install ‘tensorflowjs utilities’ for Python. Pip is similar to NPM but for the Python ecosystem instead of Node.js.

```
!pip3 install tensorflowjs
```

The screenshot shows a Jupyter Notebook interface with a file named 'Untitled0.ipynb'. On the left, there's a file tree showing directories like 'run', 'sbin', 'srv', 'sys', 'tensorflow-1.15.2', and 'tmp'. In the main area, there are two code cells. Cell [1] contains the command 'classifier_activation='softmax''. Cell [2] contains the command '!pip3 install tensorflowjs'. The output of cell [2] shows the process of downloading data from Google Cloud Storage and the successful installation of the tensorflowjs package.

```
classifier_activation='softmax'  
[1] )  
Downloading data from https://storage.googleapis.com/tensorflow/ker  
14540800/14536120 [=====] - 0s 0us/step  
14548992/14536120 [=====] - 0s 0us/step  
[2] !pip3 install tensorflowjs  
WARNING:tensorflow:Compiled the loaded model, but the compiled metr  
/usr/local/lib/python3.7/dist-packages/keras/engine/functional.py:1  
layer_config = serialize_layer_fn(layer)  
!pip3 install tensorflowjs
```

Figure 6.2.19 - Install TensorFlow.js Utilities for Python

2. Select the Play icon for this new line of code and wait for it to successfully install TensorFlow.js (see Figure 6.2.20.)

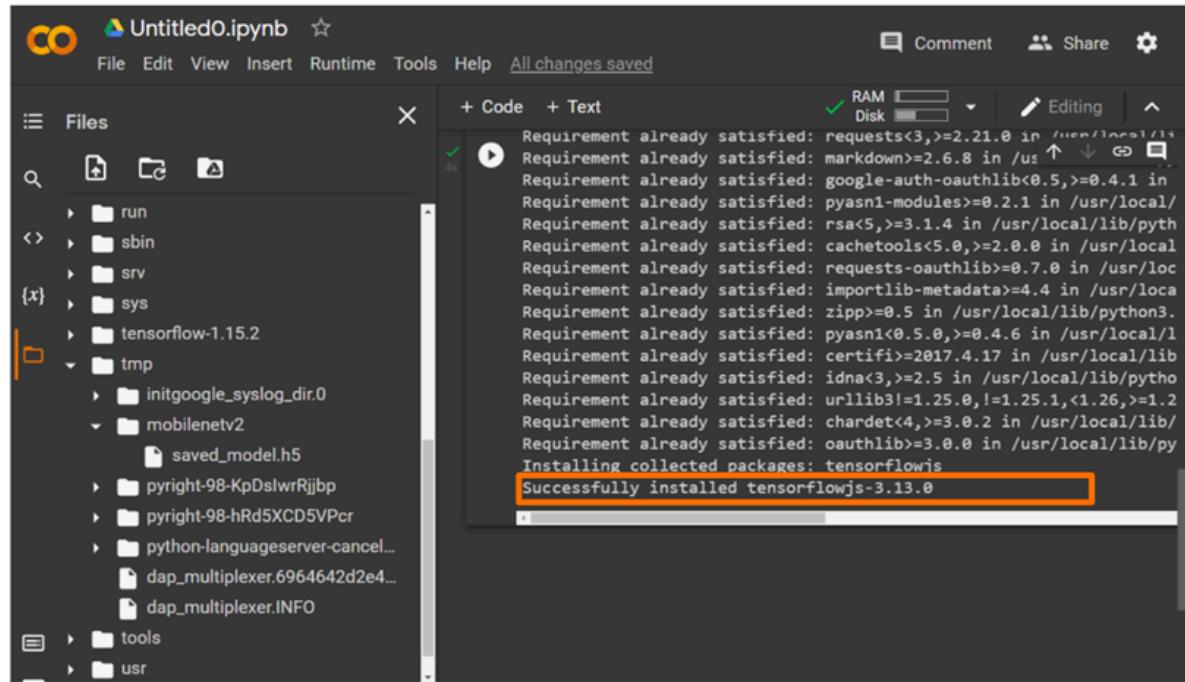


Figure 6.2.20 Successfully Installing TensorFlow.js

3. Add another new code block, and add the two lines of code shown in Figure 6.2.21.

```
!cd /tmp/mobilenetv2/
!tensorflowjs_converter --input_format=keras
--output_format=tfjs_layers_model
/tmp/mobilenetv2/saved_model.h5 /tmp/tfjs_mobilenetv2
```

Figure 6.2.21- Converting the Saved Model to the TensorFlow.js Format

The first line of code is a terminal command to change the directory to the ‘mobilenetv2’ folder created. The second line of code calls the ‘tensorflowjs_converter’. This takes the following parameters:

- a. The first parameter is the input format of the file that you want to convert, which is Keras (see Figure 6.2.22).

```

MobileNet V2 to TFJS Layers model ☆
File Edit View Insert Runtime Tools Help All changes saved
RAM Disk
+ Code + Text
[ ] Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=4.4->markdown>=3.0.0)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/dist-packages (from pyasn1-modules>=0.2.1->tensorboard>2.1.0)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard>2.1.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard>2.1.0)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard>2.1.0)
Requirement already satisfied: urllib3!=1.25.0,!!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard>2.1.0)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from requests-oauthlib>=0.7.0->google-auth-oauthlib>4.0.0)
Installing collected packages: tensorflowjs
Successfully installed tensorflowjs-3.13.0

```

The terminal command shown is:

```

!cd /tmp/mobilenetv2/
!tensorflowjs_converter --input_format=keras --output_format=tfjs_layers_model /tmp/mobilenetv2/saved_model.h5 /tmp/tfjs_model

```

Figure 6.2.22 - Converting the Saved Model to the TensorFlow.js - Specify Input Format

- The second parameter is the output format to which you want to convert. In this case, you need the ‘tfjs_layers_model’ format to create a layers model suitable for transfer learning (see Figure 6.2.23).

```

MobileNet V2 to TFJS Layers model ☆
File Edit View Insert Runtime Tools Help All changes saved
RAM Disk
+ Code + Text
[ ] Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=4.4->markdown>=3.0.0)
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/dist-packages (from pyasn1-modules>=0.2.1->tensorboard>2.1.0)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard>2.1.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard>2.1.0)
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard>2.1.0)
Requirement already satisfied: urllib3!=1.25.0,!!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests<3,>=2.21.0->tensorboard>2.1.0)
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from requests-oauthlib>=0.7.0->google-auth-oauthlib>4.0.0)
Installing collected packages: tensorflowjs
Successfully installed tensorflowjs-3.13.0

```

The terminal command shown is:

```

!cd /tmp/mobilenetv2/
!tensorflowjs_converter --input_format=keras --output_format=tfjs_layers_model /tmp/mobilenetv2/saved_model.h5 /tmp/tfjs_model

```

Figure 6.2.23 - Converting the Saved Model to the TensorFlow.js - Specify Output Format

- Next is the location of the input file. In this case, it is the ‘/tmp/mobilenetv2/saved_model.h5’ which is the location of the h5 file you just saved to disk (see Figure 6.2.24).

```

MobileNet V2 to TFJS Layers model ☆
File Edit View Insert Runtime Tools Help All changes saved
RAM Disk Editing
+ Code + Text
[ ] Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=4.4->markdown>=2.6.8->ten
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/dist-packages (from pyasn1-modules>=0.2.1->google-auth
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests<3,>>2.21.0->tensorboard~>2.6
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests<3,>>2.21.0->tensorboard~>2.6
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests<3,>>2.21.0->tensorboard~>2.6
Requirement already satisfied: urllib3!=1.25.0,!!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests<3,>>2.
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from requests-oauthlib>=0.7.0->google-auth-o
Installing collected packages: tensorflowjs
Successfully installed tensorflowjs-3.13.0

```

```

!cd /tmp/mobilenetv2/
!tensorflowjs_converter --input_format=keras --output_format=tfjs_layers_model /tmp/mobilenetv2/saved_model.h5 /tmp/tfjs_mobilenetv2

```

Figure 6.2.24 - Converting the Saved Model to the TensorFlow.js - Specify Input File Location

- Finally, you specify the output folder to place the generated tensorflow.js model files. Here you ask to create a new folder in the system tmp folder called ‘tfjs_mobilenetv2’.

```

MobileNet V2 to TFJS Layers model ☆
File Edit View Insert Runtime Tools Help All changes saved
RAM Disk Editing
+ Code + Text
[ ] Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packages (from importlib-metadata>=4.4->markdown>=2.6.8->ten
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/dist-packages (from pyasn1-modules>=0.2.1->google-auth
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-packages (from requests<3,>>2.21.0->tensorboard~>2.6
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/dist-packages (from requests<3,>>2.21.0->tensorboard~>2.6
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dist-packages (from requests<3,>>2.21.0->tensorboard~>2.6
Requirement already satisfied: urllib3!=1.25.0,!!=1.25.1,<1.26,>=1.21.1 in /usr/local/lib/python3.7/dist-packages (from requests<3,>>2.
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-packages (from requests-oauthlib>=0.7.0->google-auth-o
Installing collected packages: tensorflowjs
Successfully installed tensorflowjs-3.13.0

```

```

!cd /tmp/mobilenetv2/
!tensorflowjs_converter --input_format=keras --output_format=tfjs_layers_model /tmp/mobilenetv2/saved_model.h5 /tmp/tfjs_mobilenetv2

```

Figure 6.2.25 - Converting the Saved Model to the TensorFlow.js - Specify Output Folder

- Select Play (see Figure 6.2.26). Once the cell completes executing the code, navigate to the ‘tmp’ folder. You should see a new tfjs_mobilenetv2 folder with the model’s ‘json’ and ‘bin’ files inside (see Figure 6.2.27).

A screenshot of a terminal window titled "MobileNet V2 to TFJS Layers model". The window shows command-line output for installing packages and running the TensorFlow.js converter. The command run is:

```
!cd /tmp/mobilenetv2/  
!tensorflowjs_converter --input_format=keras --output_format=tfjs_layers_model /tmp/mobilenetv2/saved_model.h5 /tmp/tfjs_mobilenetv2
```

Figure 6.2.26 - Run the TensorFlow.js Conversion Code

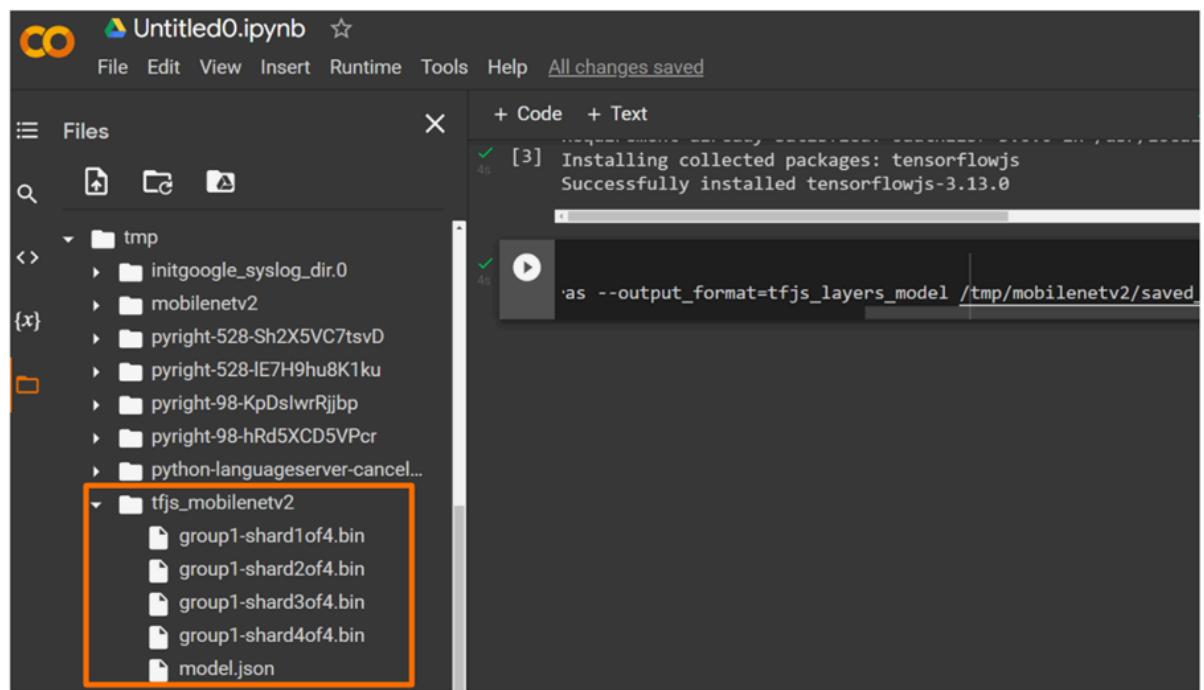


Figure 6.2.27 - View the Converted Files of the Python Model

7. To download each of the resulting files to your own hard drive, hover over the file, select the three dots that appear to the right of the file name, select download, and save the resulting file to your hard drive (see Figure 6.2.28).

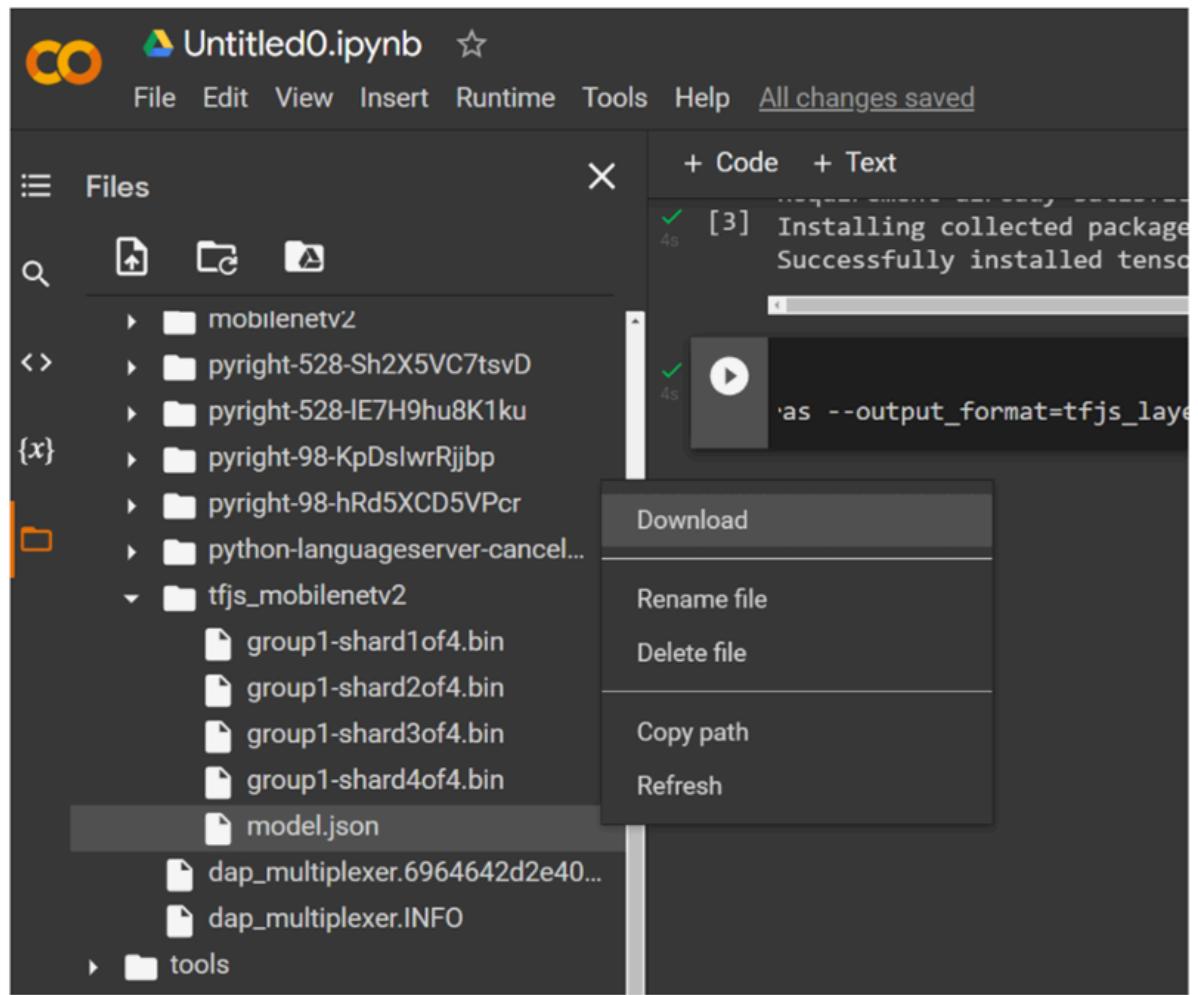


Figure 6.2.28 - Downloading Files

Note: If you end up converting larger models in the future with many binary output files to download, you may want to zip all the files together into one file to save time when downloading.

8. Figure 6.2.29. Shows how to call the zip program on the command line to create a new zip file called 'modedata.zip' in the 'tfjs_mobilenetv2' folder which contains all the files and folders that are currently contained within the 'tfjs_mobilenetv2' folder. Note the minus r at the start simply means to recursively do this for all subfolders too if there are any.

```
!zip -r /tmp/tfjs_mobilenetv2/modedata.zip  
/tmp/tfjs_mobilenetv2/
```

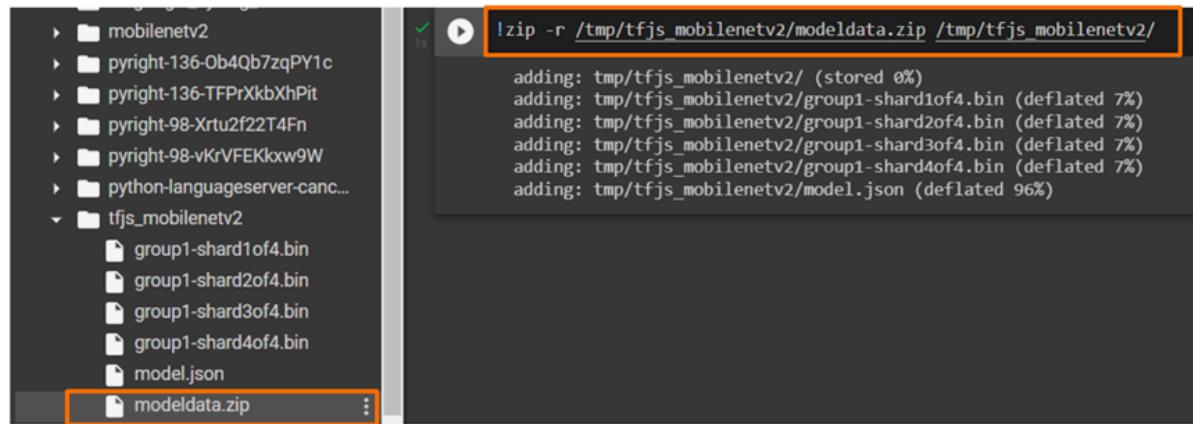


Figure 6.2.29 - Zipping the Downloaded Files

9. You can then see the new modedata.zip file has been created on the left which you can download in 1 click and then unzip on your own computer to view the contents in a much faster manner than downloading all the files individually, which is a lot more clicking.

Note: There will be times when more complex models that compile down to use less common operations will not be supported for conversion (see Figure 6.2.30.). The browser-based version of TensorFlow.js is a complete rewrite of TensorFlow and as such does not currently support all of the low-level operations that the TensorFlow C++ API has.

```
0.014ms.
Traceback (most recent call last):
  File "/home/jasonmayes/.local/bin/tensorflowjs_converter", line 8, in
    sys.exit(pip_main())
  File "/home/jasonmayes/.local/lib/python3.6/site-packages/tensorflowj
    main([' '.join(sys.argv[1:])])
  File "/home/jasonmayes/.local/lib/python3.6/site-packages/tensorflowj
    convert(argv[0].split(' '))
  File "/home/jasonmayes/.local/lib/python3.6/site-packages/tensorflowj
    experiments=args.experiments)
  File "/home/jasonmayes/.local/lib/python3.6/site-packages/tensorflowj
del
    initializer_graph=frozen_initializer_graph)
  File "/home/jasonmayes/.local/lib/python3.6/site-packages/tensorflowj
    ', '.join(unsupported))
ValueError: Unsupported Ops in the model before optimization
MatrixDiagV3
jasonmayes@tfjs-converter-codelab-test:~/tmp/saved_model$
```

Figure 6.2.30 - Python Models that Don't Convert

10. If you are unable to convert a Python saved model, you have two options:
 - a. Ask the original creator of the model not to use that op in favor of more standard operations
 - b. Contribute the missing op to the TensorFlow.js project - it is open source after all and welcomes contributions by the community.
11. **Note:** If a particular op keeps coming up as an issue for you (and others) for a certain type of model, you may want to consider submitting a feature request to the TensorFlow.js Github page detailing the use case, model details, and the impact it would have if it were supported.
12. If you are curious to learn more about ops and contributing missing ones, which is an advanced topic, check out some of the links shown below. For those of you who may be from a less mathematical background, the link to create a feature request is also shown at the bottom of this slide. By submitting a feature request someone from the community or the TensorFlow.js team may see the request and be able to implement it for

you if there is enough demand.

Current list of supported ops for client side TensorFlow.js:

github.com/tensorflow/tfjs/blob/master/tfjs-converter/docs/support_ed_ops.md

Guide on contributing a missing op:

github.com/tensorflow/tfjs/blob/master/CONTRIBUTING_MISSING_OP.md

Create a feature request:

github.com/tensorflow/tfjs/issues/new/choose

13. You can also navigate to the TensorFlow.js converter [documentation](#) (see Figure 6.2.31) on Github to find the full suite of options you can use when converting from Python to JS, along with a variety of useful options.

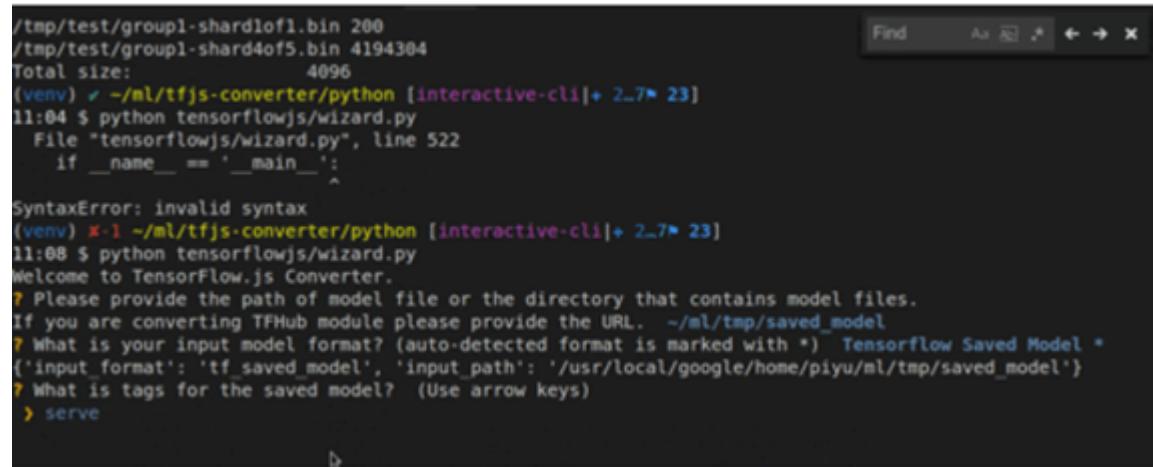
Some of these advanced options are beyond the scope of this introductory course but are pretty well documented so it is worth a read to be aware that they exist. A good example of a more advanced option is when you may be trying to optimize models to take up less space. Here the convert supports options for quantization that can come in handy. Quantizing a model essentially reduces the memory footprint it uses at the sacrifice of model accuracy. However, that may be an acceptable trade-off in some cases if you need to run on a mobile device for example.

Python-to-JavaScript		
--input_format	--output_format	Description
keras	tfjs_layers_model	Convert a keras or tf.keras HDF5 model file to TensorFlow.js Layers model format. Use <code>tf.loadLayersModel()</code> to load the model in JavaScript. The loaded model supports the full inference and training (e.g., transfer learning) features of the original keras or tf.keras model.
keras	tfjs_graph_model	Convert a keras or tf.keras HDF5 model file to TensorFlow.js Graph model format. Use <code>tf.loadGraphModel()</code> to load the converted model in JavaScript. The loaded model supports only inference, but the speed of inference is generally faster than that of a tfjs.layers_model (see above row) thanks to the graph optimization performed by TensorFlow. Another limitation of this conversion route is that it does not support some layer types (e.g., recurrent layers such as LSTM) yet.
keras_saved_model	tfjs_layers_model	Convert a tf.keras SavedModel model file (from <code>tf.contrib.saved_model.save_keras_model</code>) to TensorFlow.js Layers model format. Use <code>tf.loadLayersModel()</code> to load the model in JavaScript.
tf_hub	tfjs_graph_model	Convert a TF-Hub model file to TensorFlow.js graph model format. Use <code>tf.loadGraphModel()</code> to load the converted model in JavaScript.
tf_saved_model	tfjs_graph_model	Convert a TensorFlow SavedModel to TensorFlow.js graph model format. Use <code>tf.loadGraphModel()</code> to load the converted model in JavaScript.
tf_frozen_model	tfjs_graph_model	Convert a Frozen Model to TensorFlow.js graph model format. Use <code>tf.loadGraphModel()</code> to load the converted model in JavaScript.

JavaScript-to-Python		
--input_format	--output_format	Description
tfjs_layers_model	keras	Convert a TensorFlow.js Layers model (JSON + binary weight file(s)) to a Keras HDF5 model file. Use <code>keras.model.load_model()</code> or <code>tf.keras.models.load_model()</code> to load the converted model in Python.
tfjs_layers_model	keras_saved_model	Convert a TensorFlow.js Layers model (JSON + binary weight file(s)) to the tf.keras SavedModel format. This format is useful for subsequent uses such as TensorFlow Serving and conversion to TFLite .

Figure 6.2.31 - TensorFlow.js Converter Documentation

14. Finally, if you happen to be running on Linux directly and not in a Colab notebook, you can also run the slightly friendlier version of the converter known as the ‘TensorFlow js wizard’ from the command line terminal window (see Figure 6.2.32). To do that, use the ‘`pip3 install tensorflowjs[wizard]`’ command to install it, and then call ‘`tensorflowjs_wizard`’, instead of the ‘`tensorflowjs_converter`’ command, and follow the on-screen prompts. For more details about doing all of this in Linux, check the linked codelab that shows you how to use it on Ubuntu.



```
/tmp/test/group1-shard0of1.bin 200
/tmp/test/group1-shard4of5.bin 4194304
Total size: 4096
(venv) ✘ ~/ml/tfjs-converter/python [interactive-clι]+ 2.7* 23]
11:04 $ python tensorflowjs/wizard.py
  File "tensorflowjs/wizard.py", line 522
    if __name__ == '__main__':
      ^
SyntaxError: invalid syntax
(venv) ✘ ~/ml/tfjs-converter/python [interactive-clι]+ 2.7* 23]
11:08 $ python tensorflowjs/wizard.py
Welcome to TensorFlow.js Converter.
? Please provide the path of model file or the directory that contains model files.
If you are converting TFHub module please provide the URL. ~/ml/tmp/saved_model
? What is your input model format? (auto-detected format is marked with *) Tensorflow Saved Model *
{'input_format': 'tf_saved_model', 'input_path': '/usr/local/google/home/piyu/ml/tmp/saved_model'}
? What is tags for the saved model? (Use arrow keys)
> serve

```

Figure 6.2.32 - TensorFlow.js Wizard

Natural Language Processing

In subsequent lessons, you will create a website that is capable of detecting comment spam live in the web browser by using a pre-made TensorFlow.js model. In order to implement this model, you first need to understand the basics of how to interact with natural language models.

First though, navigate to the following [link](#) and explore the TensorFlow Lite Model Maker library which is where the pre-made model you will initially use actually comes from. You will use an exported saved model from this system in TensorFlow.js and you will write the code to interface directly with the model at the tensor level.

See Figure 6.3.1. for a list of available models for text classification.

Overview

The TensorFlow Lite Model Maker library simplifies the process of training a TensorFlow Lite model using custom dataset. It uses transfer learning to reduce the amount of training data required and shorten the training time.

Supported Tasks

The Model Maker library currently supports the following ML tasks. Click the links below for guides on how to train the model.

Supported Tasks	Task Utility
Image Classification: tutorial , api	Classify images into predefined categories.
Object Detection: tutorial , api	Detect objects in real time.
Text Classification: tutorial , api	Classify text into predefined categories.
BERT Question Answer: tutorial , api	Find the answer in a certain context for a given question with BERT.
Audio Classification: tutorial , api	Classify audio into predefined categories.
Recommendation: demo , api	Recommend items based on the context information for on-device scenario.

Figure 6.3.1 - TensorFlow Lite Model Maker Library

Selecting the right model:

- You will find three text-based models (see Figure 6.3.2.) in the model maker documentation. Note that all three can be used for comment spam detection, although some are more accurate or versatile than others. Since you are going to implement your model in the browser, size is a significant factor in choosing a model.

Model Name	Description	Size
Average Word Embedding	Averaging text word embeddings with RELU activation.	< 1MB
MobileBERT	4.3x smaller and 5.5x faster than BERT-Base while achieving competitive results, suitable for on-device applications.	25 MB with quantization, 100 MB without
BERT-Base	Standard BERT model that is widely used in NLP tasks.	300 MB

Figure 6.3.2 - TensorFlow Lite Model Maker Documentation

- The BERT_Base model is too large for you to deploy on the client-side, although it could be deployed on the server-side using Node.js.
- Of the remaining two models, note that the mobileBERT model's size ranges from 25MB to 100MB depending on a process called quantization (discussed in more detail below), and the 'Average Word Embedding' model is less than 1 MB in size.
- Since you want your web page to run fast and on as many devices as possible, the 1MB sized 'Average Word Embedding' model seems like a good choice. You may always switch to the larger quantized model if needed later on.



Key Concept

Quantization

Computers have different ways to represent numbers depending on their range. The more memory used to store a number, the larger the number that can be stored (see Figure 6.3.3).

Data Type	Memory used in bits to store 1 number of this data type	Range of numbers storable (assuming unsigned)
int-8 (char)	8	0 to 255
int-16 (short)	16	0 to 65,535
int-32 (long)	32	0 to 4,294,967,295

Figure 6.3.3 - Data Types, Memory Used, and Range of Storable Numbers

A model usually uses the 32-bit float data type to store model weights. However, if you reduce the precision to an 8-bit integer, you will find that the model does not lose too much accuracy yet results in a smaller-sized model.

Quantization is the process of reducing the precision of the numbers so that the model is smaller and can download faster. This provides a significant advantage when deploying models on the client-side. See Figure 6.3.4 to see how an unsigned 32-bit floating-point number is transformed to an 8bit integer.

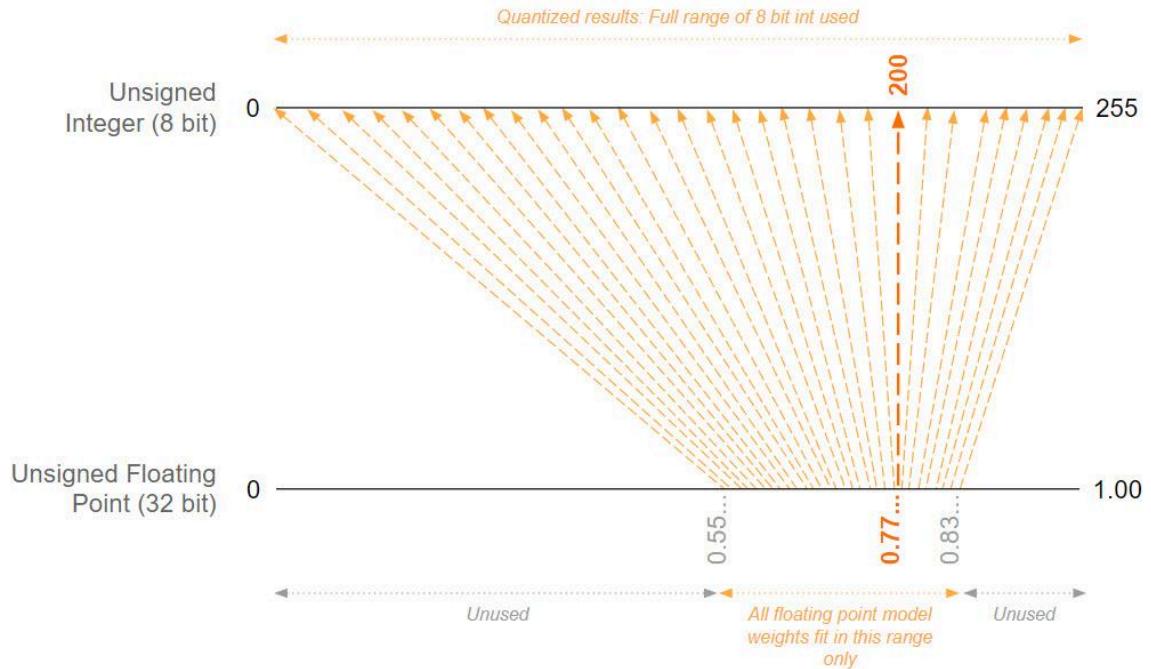


Figure 6.3.4 - Quantization

Since a very small part of the floating-point size is typically used, the bounds of that range are found and transformed to the full 8-bit range to retain as much detail as possible. As a result of this quantization, a number that previously took up 32 bits of memory now takes up just 8 bits of memory, using 4 times less storage space!

In the next unit, you will understand how to pass data into the ‘Average Word Embedding’ model that you have selected.

A comment spam detection model needs to learn information about words, which are strings of character (see Figure 6.3.5). However, ML models essentially work by processing and figuring out patterns in numbers. In this unit, you will understand how words can be represented by numbers such that similar words have a similar numerical representation that can be learned from.

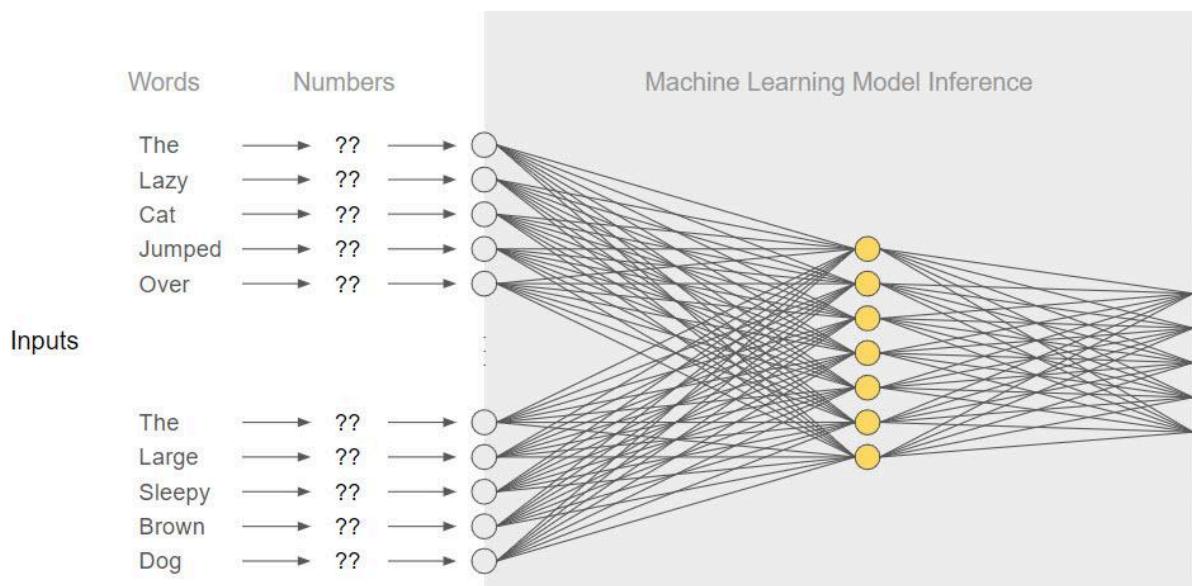


Figure 6.3.5 - Words must be converted to numbers to be used with an ML model somehow

Numerically representing words

- Consider the word 'car'. One way to represent this numerically is to assign numbers based on their literal order in the English dictionary (see Table 6.3.6) Observe that this system would provide a similar number for words such as 'car', or 'can' which, while they look similar by spelling, have very different meanings!

9838 cab
9839 cam
9840 can
9841 car
9842 cap
9843 cat

Table 6.3.6 - Literal Order of 3-Letter Words Starting with C

- A better option would be to create a dictionary of, say, 1,000 words, which can be compared and given a score for a ‘dimension’ of interest.
 - You may, for instance, define a dimension called ‘Medical’ to score words that are related to medicine, i.e. to find out how ‘medical’ a word is.
 - Figure 6.3.7 shows a group of words scored on the ‘medical dimension’. Observe that the word ‘doctor’ scores high (around 0.95) whereas the word ‘pilot’ scores low (around -0.4).

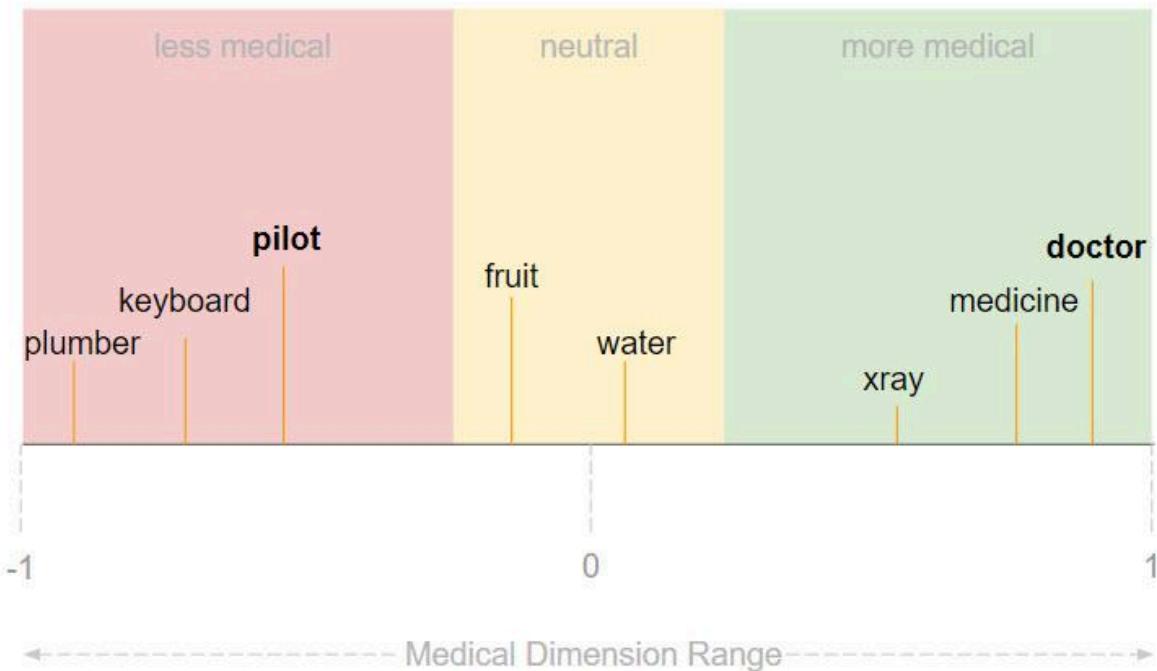


Figure 6.3.7 - Scoring Words on a ‘Medical Dimension’

- Considering the complexity of natural language, one dimension is not enough to separate words for accurate classification. You will usually need several dimensions to organize words in a way that allows for good separation to solve a particular problem. Figure 6.3.8 represents a 2-dimensional graph that can be used to score words across two dimensions instead - ‘biological and ‘medical’.

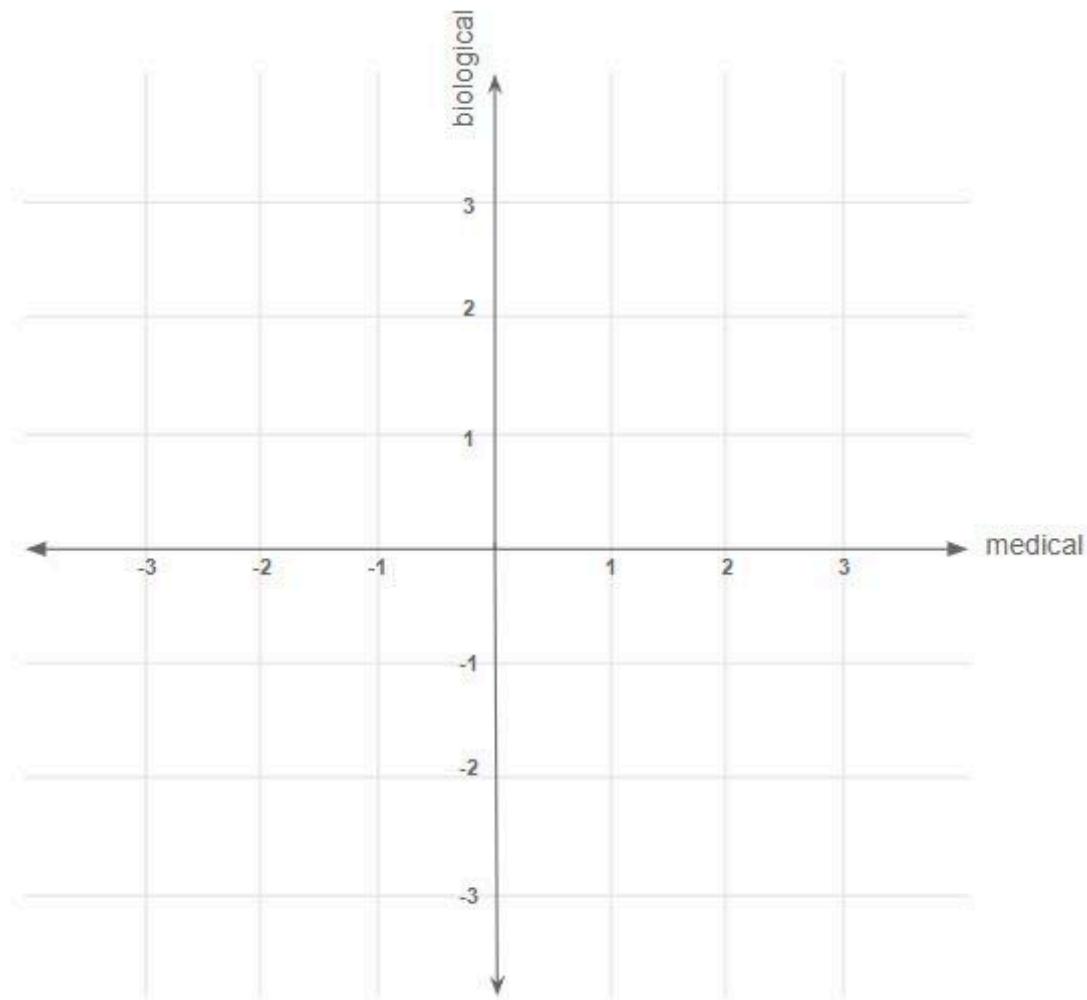


Figure 6.3.8 - 2D Graph to Represent Words in Two Dimensions

- If you use this 2-dimensional graph to score a word such as 'tumor', you will find where it lies on each of the dimensions (see Figure 6.3.9.). The word is related to 'biology' as well as 'medicine' and so scores high in both dimensions. The point at which the word lies [3,3] is a 2-dimensional vector that can be used to represent the word 'tumor' numerically. The same exercise when applied to the word 'x-ray' results in a vector of value [2, -3], since 'x-ray' is a medical term but not biological for example.

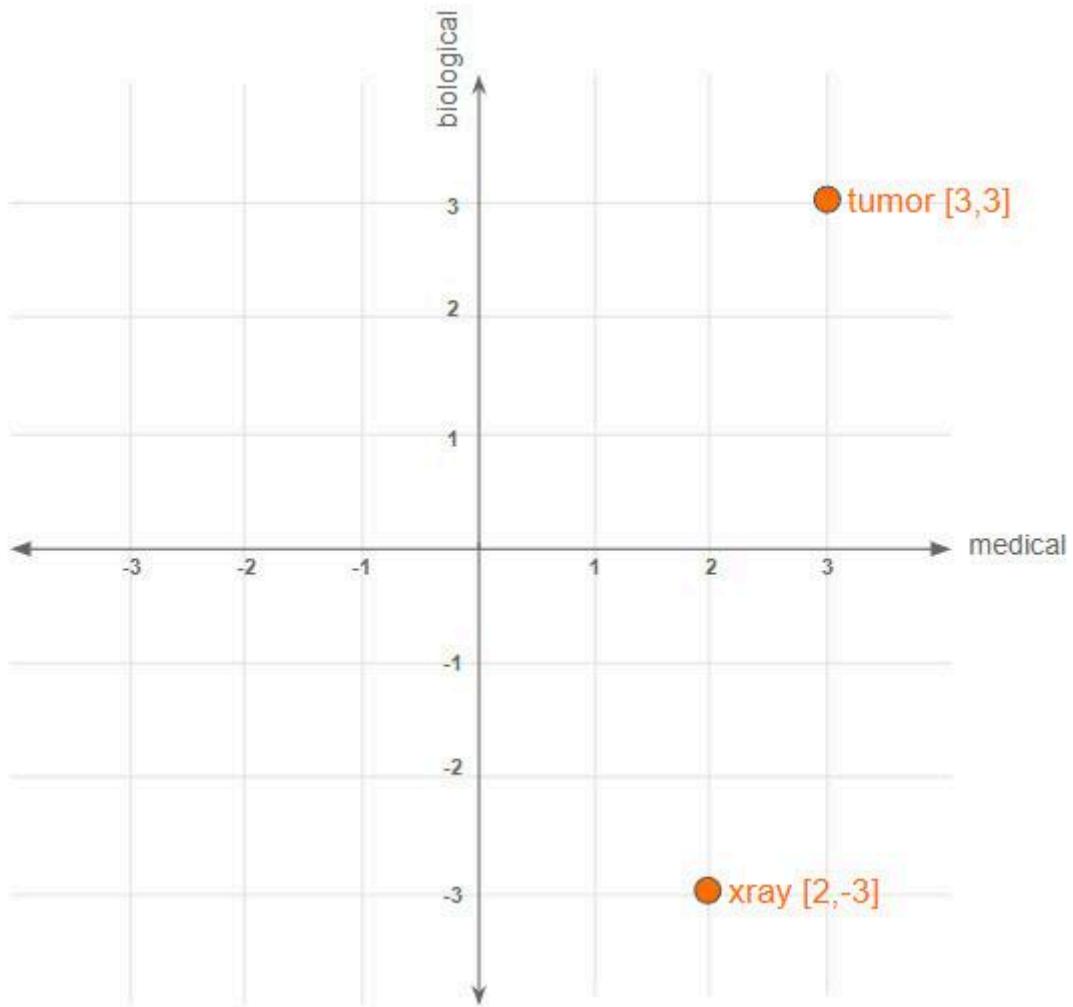


Figure 6.3.9 - Dimensional Vectors Representing Words

- When training an ML model for natural language processing, you can specify the number of dimensions you want the model to utilize for separating the words in your dictionary, and the model will learn the most meaningful ways to separate the training data to provide useful classifications.

Note: A rule of thumb determined from research is that the fourth root of the number of words you are working with works well in determining the number of dimensions required for good separation of data. So, if you are

using 2,000 words in a dictionary, 7 dimensions would be a good start to provide enough ways to separate the words for accurate classification.

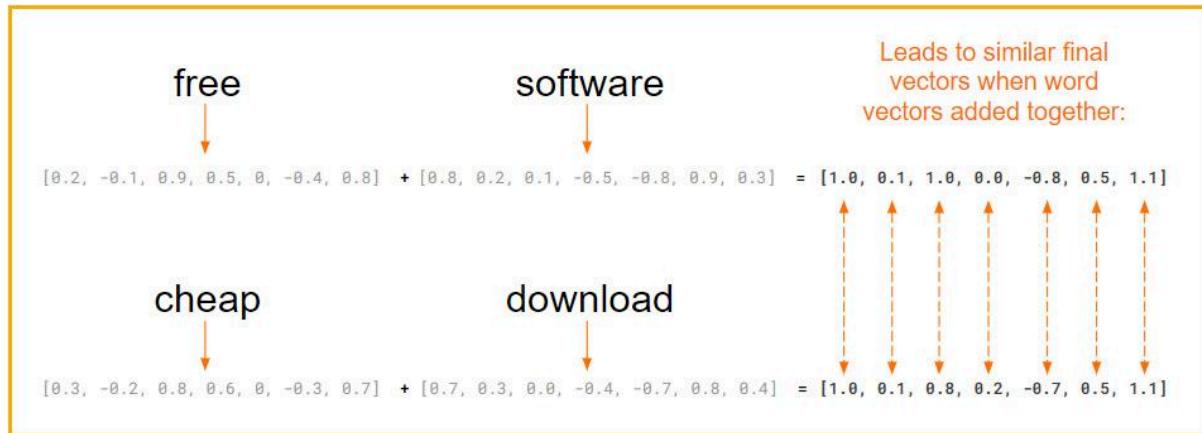


Figure 6.3.10 - Word Vectors

Figure 6.3.10 shows two sentences - ‘free software’ and ‘cheap download’. The words in each sentence are encoded using seven dimensions that a machine learning model has learnt.

Note that ‘free’ and ‘cheap’ have a similar vector encoding because they would score similarly in many dimensions. Similarly, ‘software’ and ‘download’ have similar vector encoding too.

If you add the values of both the words in each sentence, you would obtain a new vector with similar values, indicating that these sentences occupy the same space in the 7-dimensional graph showing they are highly related.

You now have a way to represent similar sentences (using vectors), and you can use these vector representations to train models to classify sentences as spam or not spam.

In the next lesson, you will create a website that can classify spam comments entirely in the browser.

Comment Spam Detection

In this session, you will start developing a website that can classify spam comments entirely in the web browser.

Introduction:

The exponential increase in social and interactive platforms has presented plenty of opportunities for spammers to abuse such systems. In order to gain more visibility for their content, spammers attempt to associate their content with the articles, videos, and posts others have written. Traditional methods of spam detection such as a list of blocked words are no match for today's advanced spam bots, which continuously evolve to bypass these simple methods.

Machine Learning (ML) for Spam Detection

Traditionally, ML models that pre-filtered comments for posting were deployed on the server-side. With TensorFlow.js, however, you can now execute ML models on the client-side in the browser. Note that executing the model at the browser level prevents the comment from ever reaching the back-end, thus potentially saving costly server-side resources.

For this application, you will use a premade spam detection model available from TensorFlow Hub. You can find the model at [this URL](#), or you may navigate to TensorFlow Hub and search for 'SPAM detection' and filter for the JS version (see Figure 6.4.1.1).

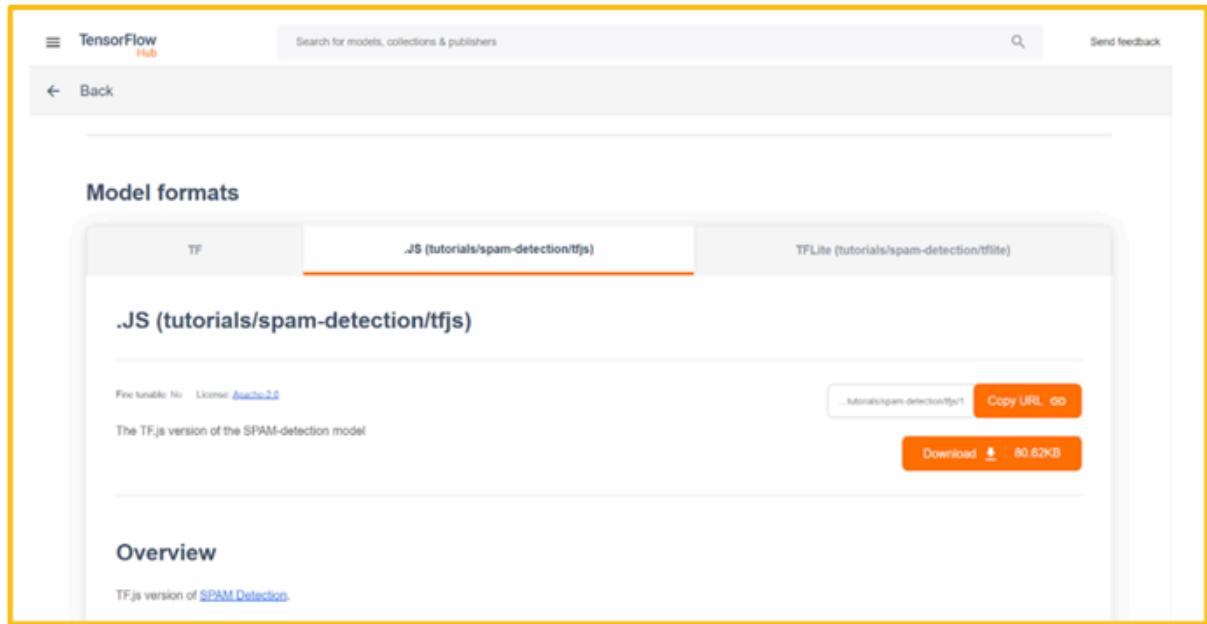


Figure 6.4.1.1. Pre-Trained Average Word Embedding Model

Note: You can also retrain this model for other purposes, such as detecting the language a comment was written in, or figuring out the content of a support request to route it to the correct team without having to read it yourself to decide. The knowledge you learn here can be applied in many ways beyond spam detection too.

Steps

1. First, navigate to the slightly more advanced TensorFlow.js Node Boilerplate Link on Glitch.com (see Figure 6.4.1.2). that has been set up for this project. This uses Node.js to create a simple back-end that is capable of using WebSockets to send data between connected front-end users. You will use this boilerplate to create a webpage wherein when you post a comment, you will see the comment appear everywhere the webpage is open if the comment is not found to be spam, replicating the functionality of a real video blog style website.



Figure 6.4.1.2. TensorFlow.js Boilerplate for Comment Spam Detection

2. Remix the project and explore the layout:

- WWW folder:** Note that all the static website files such as the ‘index.html’, ‘script.js’, and ‘style.css’ are now inside a folder named ‘www’ (see Figure 6.4.1.3).

```

grove-battle-pigeon
  index.html
  PRETTIER

  Settings
  Assets
  Files + 

  www/
    index.html
    script.js
    style.css

  .env
  README.md
  package.json
  server.js

```

```

1  <!DOCTYPE html>
2  <html lang="en">
3    <head>
4      <title>Node Express with TensorFlow.js</title>
5      <meta charset="utf-8">
6      <meta http-equiv="X-UA-Compatible" content="IE=edge">
7      <meta name="viewport" content="width=device-width, initial-scale=1">
8      <!-- Import the webpage's stylesheet -->
9      <link rel="stylesheet" href="/style.css">
10     </head>
11     <body>
12       <h1>TensorFlow.js Express via Node Hello World</h1>
13
14       <p id="status">Awaiting TF.js load</p>
15
16       <!-- Import TensorFlow.js library -->
17       <script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@3.11.0/dist/tf.min.js" type='
18
19       <!-- Import the page's JavaScript to do some stuff -->
20       <script src="/script.js" defer></script>
21     </body>
22   </html>
23

```

Figure 6.4.1.3. TensorFlow.js Boilerplate Files

- The ‘env’ file:** The ‘.env’ file is a special feature of Glitch to store passwords and sensitive data (see Figure 6.4.1.4). You will not need to use this but it is part of the project nevertheless.

The screenshot shows a code editor interface for a project named "grove-battle-pigeon". On the left, there's a sidebar with icons for Settings, Assets, and Files. Under "Files", there's a tree view with "www/" expanded, showing "index.html", "script.js", "style.css", and ".env". The ".env" file is selected and highlighted with an orange border. The main pane is titled "Environment Variables". It contains a note about the .env file being used for storing secrets like API keys. Below this, there's a code editor with the following content:

```

# Environment Config
# store your secrets and config variables in here
# only invited collaborators will be able to see your .env values
# Note: comments will be visible to remixes so don't put secrets in comments!
# reference these in your code with process.env.SECRET

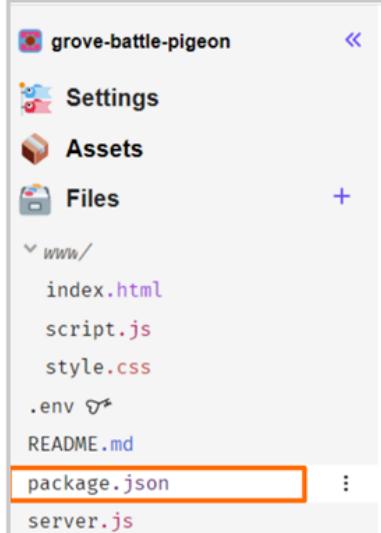
```

At the bottom, there are two rows of environment variable definitions:

SECRET	copy	Variable Value	X
MADE_WITH	copy	Variable Value	X

Figure 6.4.1.4. TensorFlow.js Boilerplate Files - Environment Variables

- c. The ‘**package.json**’ file: The file ‘package.json’ contains the Node.js configuration to specify metadata about the project along with any dependencies it needs to install in order to run the application correctly. The only dependency needed for this project at this stage is the ‘express’ framework that is already detailed under the dependencies property (see Figure 6.4.1.5). Note that the ‘server.js’ file is specified as the main script to run.



```

{
  "name": "tfjs-with-backend",
  "version": "0.0.1",
  "description": "A TFJS front end with thin Node.js backend",
  "main": "server.js",
  "scripts": {
    "start": "node server.js"
  },
  "dependencies": {
    "express": "^4.17.1"
  },
  "engines": {
    "node": "12.x"
  }
}

```

Figure 6.4.1.5. Package.json contents.

The ‘server.js’ file: Your server-side web app Node.js code will be written here. Node.js is just JavaScript for the server-side environment so has some extra functionality beyond the web browser environment.



```

/*
 * Copyright 2018 Google LLC. All Rights Reserved.
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
const express = require("express");
const app = express();
// Make all the files in 'www' available.
app.use(express.static("www"));

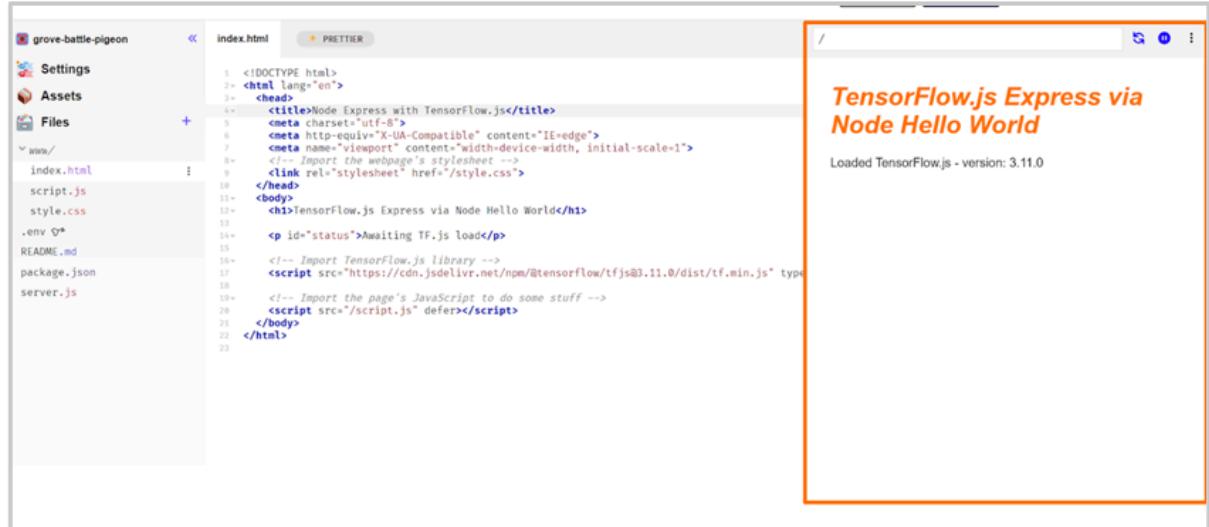
app.get("/", (request, response) => {
  response.sendFile(__dirname + "/www/index.html");
});

// Listen for requests.
const listener = app.listen(process.env.PORT, () => {
  console.log("Your app is listening on port " + listener.address().port);
});

```

Figure 6.4.1.6. TensorFlow.js BoilerPlate Files - server.js

- i. The ‘server.js’ file has some boilerplate code to set up an ‘express’ web server. This server serves the files inside the ‘www’ folder.
 - ii. It imports the ‘express’ library, creates an instance of an express server that is assigned to a constant called ‘app’, and then tells the ‘app’ to serve the contents of the ‘www’ folder.
 - iii. When a ‘get’ request is received from a client’s web browser with no file specified, the server returns the ‘index.html’ file by default.
 - iv. The app is also set up to listen on the default port that Glitch uses for serving Node web apps. This is done by using a special environment variable called ‘process.env.PORT()’.
- d. If you open the preview window at this point, it will show the placeholder ‘index.html’ page (see Figure 6.4.1.7)



The screenshot shows a Glitch project titled "grove-battle-pigeon". On the left, the file structure is visible:

```

  Settings
  Assets
  Files
  www/
    index.html
    script.js
    style.css
    .env
    README.md
    package.json
    server.js

```

The "index.html" file is selected and its content is displayed in the editor:

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Node Express with TensorFlow.js</title>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- Import the webpage's stylesheet -->
    <link rel="stylesheet" href="/style.css">
  </head>
  <body>
    <h1>TensorFlow.js Express via Node Hello World</h1>
    <p id="status">Awaiting TF.js load</p>
    <!-- Import TensorFlow.js library -->
    <script src="https://cdn.jsdelivr.net/npm@tensorflow/tfjs@3.11.0/dist/tf.min.js" type="text/javascript"></script>
    <!-- Import the page's JavaScript to do some stuff -->
    <script src="/script.js" defer></script>
  </body>
</html>

```

On the right, a preview window shows the rendered HTML with the heading "TensorFlow.js Express via Node Hello World" and the status message "Awaiting TF.js load". A red box highlights the title and status message in the preview window.

Figure 6.4.1.7. TensorFlow.js BoilerPlate Preview - Placeholder index.html Page

In this step, you update the HTML and CSS code.

Steps

1. Replace the content in the body tags of the HTML with the code provided below:

```
<body>

<h1>MooTube</h1>

<h2>Check out the TensorFlow.js rap for the show and tell!</h2>

<iframe width="100%" height="500"
src="https://www.youtube.com/embed/RhVs7ijB17c" frameborder="0" allow="autoplay;
encrypted-media;" allowfullscreen></iframe>

<section id="comments" class="comments">

<div id="comment" class="comment" contenteditable></div>

<button id="post" type="button">Comment</button>

<ul id="commentsList">

<li>

<span class="username">SomeUser</span>

<span class="timestamp">2/11/2021, 3:10:00 PM</span>

<p>Wow, I love this video, so many amazing demos!</p>

</li>

</ul>

</section>
```

```
<script  
src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@3.11.0/dist/tf.min.js"  
type="text/javascript"></script>  
  
<script type="module" src="/script.js"></script>  
  
</body>
```

- a. It adds a <h1> tag for the page title and a <h2> for the article title.
- b. An 'iframe' tag is added that embeds an arbitrary YouTube video. You can place any video here simply by changing the URL of the 'iframe'.
- c. Further down is a 'section' with an id and class of "comments". This section element contains:
 - i. A 'contenteditable' div to write new comments to
 - ii. A button to submit the new comment
 - iii. An unordered list of pre-existing comments that has the id named 'commentsList'. Each comment has a username and time of posting within a span tag inside each list item, and then finally the comment itself in a paragraph tag.
- d. Finally, the TensorFlow.js library and the 'script.js' are imported at the end of the body content.

You can find the complete code at this [URL](#) for convenience.

At this point, your live preview should look like what is shown in Figure 6.4.1.8.

MooTube

Check out the TensorFlow.js rap for the show and tell!



[Comment](#)

- SomeUser 2/11/2021, 3:10:00 PM

Wow, I love this video, so many amazing demos!

Figure 6.4.1.8 - Web Page Preview with Updated HTML. Play the TensorFlow.js rap for a fun 2-minute break!

2. In this step, you add CSS to the 'styles.css' document. Essentially these styles position and color the various HTML elements when they are in different states.

```
body {  
background: #212121;  
color: #ffffff;  
font-family: helvetica,  
sans-serif;  
}  
  
h1 {  
color: #f0821b;  
font-size: 24pt;  
padding: 10px;  
}  
  
section, iframe {  
background: #212121;  
padding: 10px;  
}  
  
h2 {  
font-size: 16pt;  
padding: 0 10px;  
}  
  
button:focus,  
button:hover,  
header a:hover {  
background: rgb(260,  
150, 50);  
}  
  
.comment {  
background: #212121;  
border: none;  
border-bottom: 1px solid  
#888888;  
color: #ffffff;  
min-height: 25px;  
display: block;  
padding: 5px;  
}  
  
.comments button {  
float: right;  
margin: 5px 0;  
}  
  
.comments ul  
li:nth-child(1) {  
background: #313131;  
}  
  
.comments ul li:hover {  
background: rgb(70, 60,  
10);  
}  
  
.username, .timestamp {  
font-size: 80%;  
margin-right: 5px;  
}  
  
.username {  
font-weight: bold;  
}  
  
.processing {  
opacity: 0.3;
```

```

}

.comments button,
.comments .comment {
    transition: opacity
    500ms ease-in-out;
}

iframe {
    display: block;
    padding: 15px 0;
}

button {
    color: #222222;
    padding: 7px;
    min-width: 100px;
    background: rgb(240,
130, 30);
    border-radius: 3px;
    border: 1px solid
#3d3d3d;
    text-transform:
uppercase;
    font-weight: bold;
    cursor: pointer;
}

.filter {
    filter: grayscale(1);
}

.comments ul li.spam {
    background-color:
#d32f2f;
}

.comments ul {
    clear: both;
    margin-top: 60px;
}

.comments ul li {
    margin-top: 5px;
    padding: 10px;
    list-style: none;
    transition: background
    500ms ease-in-out;
}

p {
    color: #cdcdcd;
}

```

You can find the completed code at this [URL](#) for convenience.

At this point, your live preview should now look like what is shown in Figure 6.4.1.9.

Note that when you click just above the comment button, the 'div' element should also allow you to write in it.

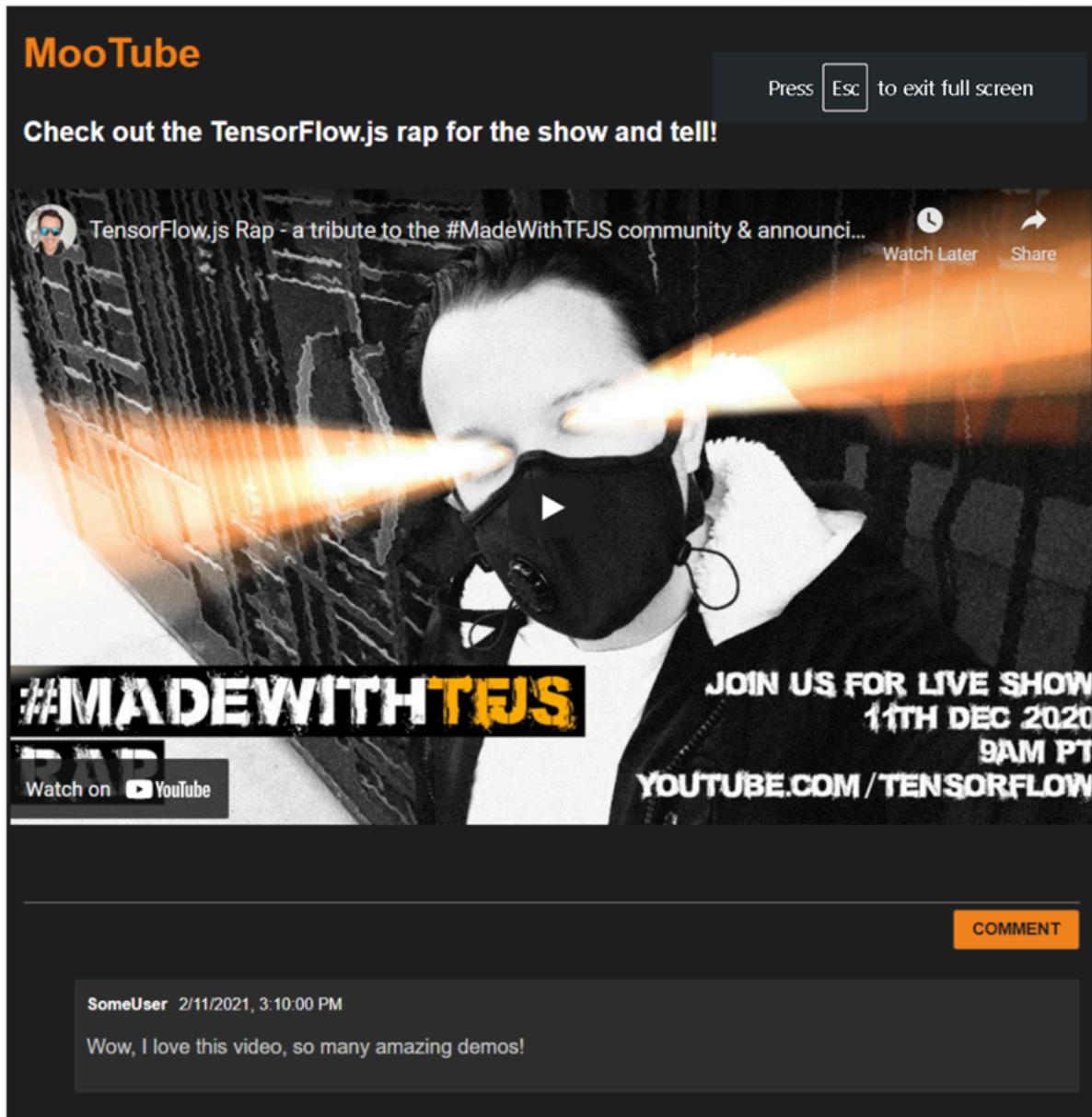


Figure 6.4.1.9 - Web Page Preview with Updated HTML and CSS

In this unit, you add interaction logic to the 'script.js' file.

Steps

1. Add constants to grab key references to the document's HTML elements.

```
const POST_COMMENT_BTN = document.getElementById('post');

const COMMENT_TEXT = document.getElementById('comment');

const COMMENTS_LIST = document.getElementById('commentsList');

// CSS styling class to indicate comment is being processed when

// posting to provide visual feedback to users.

const PROCESSING_CLASS = 'processing';
```

Then set the name of the CSS class to use in this case ‘processing’ such that when a comment is processed, as this class name is set, the interface will appear grayed out until the comment is approved and the class is removed.

2. Add a function named ‘handleCommentPost’ that deals with processing new comments to check if it’s spam. The function will only write the comment to the page if it is not considered spam once extra logic is added later.

```
/** 

 * Function to handle the processing of submitted comments.

 **/ 

function handleCommentPost() { 

    // Only continue if you are not already processing the comment. 

    if (! POST_COMMENT_BTN.classList.contains(PROCESSING_CLASS)) { 

        POST_COMMENT_BTN.classList.add(PROCESSING_CLASS); 

        COMMENT_TEXT.classList.add(PROCESSING_CLASS);
```

```

let currentComment = COMMENT_TEXT.innerText;

console.log(currentComment);

// TODO: Fill out the rest of this function later.

}

}

POST_COMMENT_BTN.addEventListener('click', handleCommentPost);

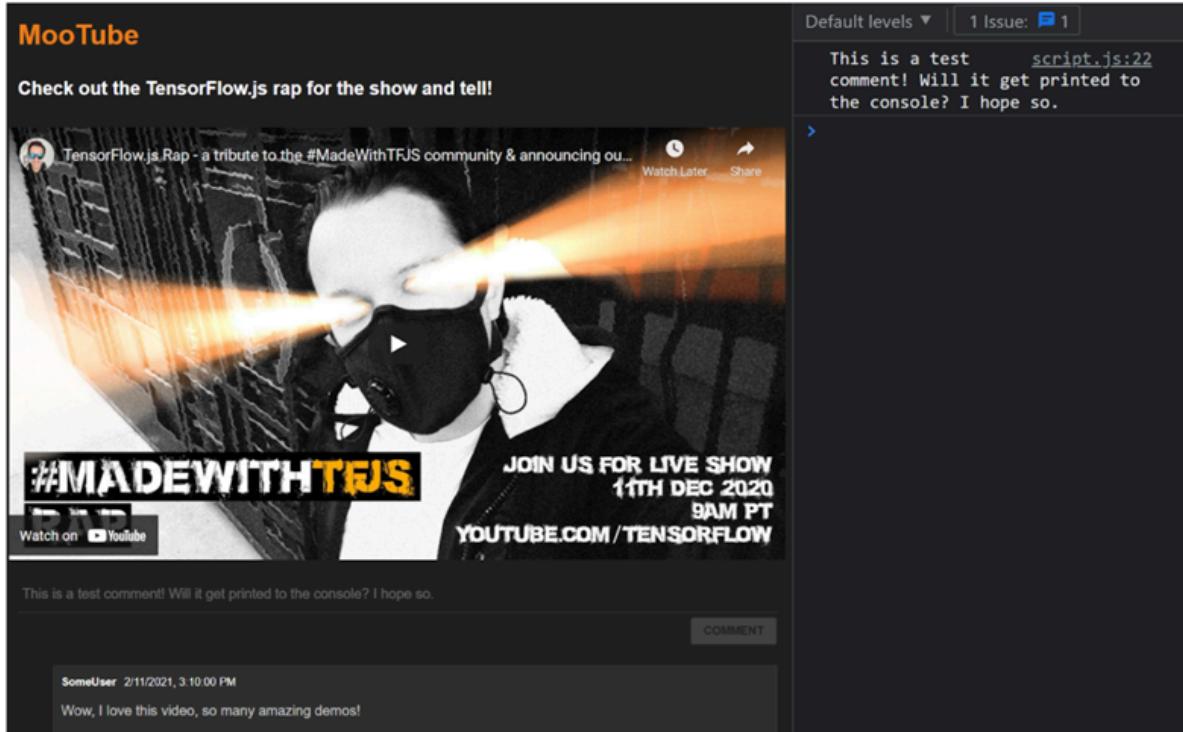
```

- a. Start by checking if you are already in the process of classifying a comment.
- b. If a comment is not being processed, the code sets the class of the 'POST_COMMENT_BTN' and the 'COMMENT_TEXT' to be the 'PROCESSING_CLASS' so that they appear to be grayed out.
- c. Grab a reference to the current 'COMMENT_TEXT's data and log it to the console to check if this function is working as expected.

Note: Remember the 'TODO' comment in this section since you add code here in later sessions to complete the code for this function.

- d. Now that the function is defined, add a click event listener to the 'POST_COMMENT_BTN' such that it calls this 'handleCommentPost' function when clicked.
- e. See Figure 6.4.1.10 to observe the output of your comment spam detection website set up as it currently stands when

running the above code. Note when you type in the text box and click the button, the text is logged to the console on the right.



6.4.1.10 - Web Page Preview: Processing Comments.

In the next section, you will actually use the pre-made natural language machine learning model for spam detection and integrate it into this skeleton web app.

Comment Spam Detection - Using a Pre-Trained NLP Model

In this lesson, you explore the files that are produced by Model Maker for the natural language model you will use. Unlike earlier models in this course, in addition to the 'model.json' and 'bin' files, you will find two extra ones that you need to understand.

1. **The vocab file:** 'vocab' is short for vocabulary. This is a file provided by Model Maker, sometimes with no file extension, that shows you how to encode words of a sentence into numbers such that the model can take a numerical representation of a sentence as an input. See Figure 6.4.2.1 to see the first 10 words in the vocab file. The list

has words separated by a space and followed by a number. This number will represent the word when using it with the model. You can view this raw file at this [URL](#).

```
vocab — commentsspamjs
1 <PAD> 0
2 <START> 1
3 <UNKNOWN> 2
4 i 3
5 check 4
6 video 5
7 song 6
8 com 7
9 please 8
10 like 9
```

Figure 6.4.2.1. vocab File

2. **The labels.txt file:** This file, generated by Model Maker, contains the resulting class names that the model will try and predict. In this case, the model needs to find out if a given comment is spam, so the file has the values ‘false’ and ‘true’ listed, in that particular order. Here false means not spam, true means spam. You could imagine that if you have a model that does something like predicting what languages it thinks a sentence was, then this file may contain a list of all the languages it is able to recognize instead, in the same order it would output such predictions allowing you to interpret the output of the model with a human-readable class name.

Writing the code to load and use the model.

Add the following code to the ‘script.js’ file. Note the model files have been hosted for you so that you can use the code here directly and it should work just fine.

Steps:

```
const MODEL_JSON_URL =
'https://storage.googleapis.com/jmstore/TensorFlowJS/EdX/SavedModels/spam/model.json';

const SPAM_THRESHOLD = 0.75;

var model = undefined;

async function loadAndPredict(inputTensor) {

  // Load the model.json and binary files you hosted. Note this is
  // an asynchronous operation so you use the await keyword

  if (model === undefined) {

    model = await tf.loadLayersModel(MODEL_JSON_URL);

  }

  // Once model has loaded you can call model.predict and pass to it
  // an input in the form of a Tensor. You can then store the result.

  var results = await model.predict(inputTensor);

  // Print the result to the console for us to inspect.

  results.print();

  // TODO: Add extra logic here later to do something useful
```

```
}
```

```
loadAndPredict(tf.tensor([[1,3,12,18,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]]));
```

1. Define two new constants and a variable.
 - a. The first constant MODEL_JSON_URL is the location of where the model is hosted.
 - b. The second constant SPAM_THRESHOLD is how confident you want to be before marking something as spam. Here it is set to 0.75 (representing 75% probability of spam).
 - c. The ‘model’ variable will store the model once loaded.
2. Create a new function named ‘loadAndPredict’. The input for this function is a single 2D tensor containing numerical values that represent words in a given sentence (each sentence is a tensor1d (an array of word numbers), but is passed as a batch so tensor2d is expected).
 - a. First, check if the model has been loaded. If it's not loaded, use ‘await tf.loadLayersModel’ and pass to it the MODEL_JSON_URL that was defined earlier to load the model.
 - b. You can then call ‘model.predict’ with ‘inputTensor’ and the input and await the output which will be stored in a variable named ‘results’.
 - c. Once an output is returned, print the ‘results’ to the console to inspect the output predictions.
3. Then call the function you just defined using the example data shown; in later units, you will understand how this data is created. Note the ‘TODO’ at the end of this function. You will add code in this section later on.

At this point, if you run the code, you will see the output tensor printed to the console (see Figure 6.4.2.2). The two numbers in the output tensor represent the probabilities of what the model thinks the classification is. Based on the labels.txt file, you can infer that this comment was not spam because:

- The first entry represents the label ‘false’. Here it shows a value of 0.98 or 98% confidence it is not spam.
- The second entry represents the label ‘true’. Here it predicts 0.016, which is 1.6% sure it thinks it may be spam.

Essentially, the labels.txt file allows you to identify the class (or label if you will) of each value of the output tensor.

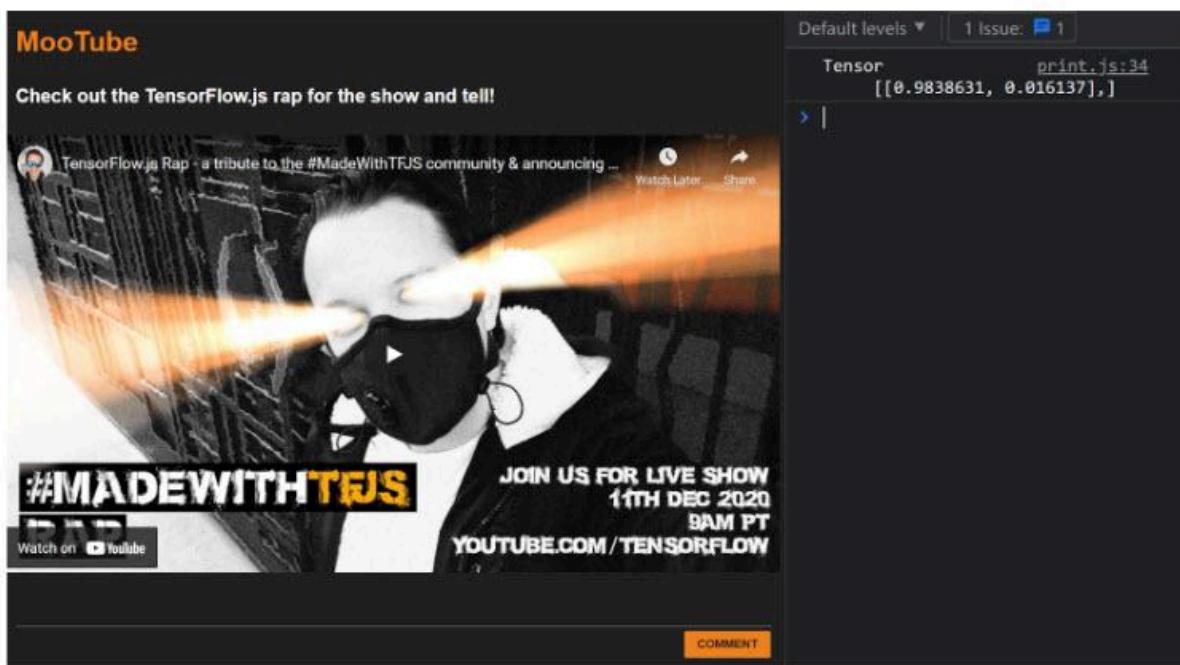


Figure 6.4.2.2. Web Page Preview with Prediction

Congratulations - you have sent data into the model, and you are getting valid outputs. In the next unit, you will understand how words are represented as numbers and sent as inputs to demystify the strange numbers you just sent into this model.

Comment Spam Detection - Tokenization

In this lesson, you learn how to convert sentences into numbers that can be sent as inputs to a model for classification.

Encoding Words to Numbers

Open the ‘vocab’ file in a text editor (see Figure 6.4.3.1). The file is a lookup table that assigns numerical IDs to meaningful words and tokens.

In the list of 10 items shown in the screenshot, the first three tokens are special cases detailed further below. The words from lines 4 to 10 are words that the model learned that have significance in detecting spam. This means each one of those words has a ‘word vector’ associated with it to represent it for the dimensions that matter for spam classification. As such it is associated with a numerical id to use to represent that word that will later be converted to the word vector that represents it when the model encounters one of those numbers.

1	<PAD>	0
2	<START>	1
3	<UNKNOWN>	2
4	i	3
5	check	4
6	video	5
7	song	6
8	com	7
9	please	8
10	like	9

Figure 6.4.3.1 - Vocab File

<PAD>: The PAD token is represented by the number 0. ML models expect a fixed number of inputs, no matter the length of the input sentence. The model you are using expects 20 numbers, so if the input sentence has fewer words, the remaining spaces in the array are filled with '0's (see Figure 6.4.3.2). If the input is greater than 20, you need to split it up and perform multiple classifications on smaller sentences instead.

<PAD> 0

Short for "padding", represented by number 0

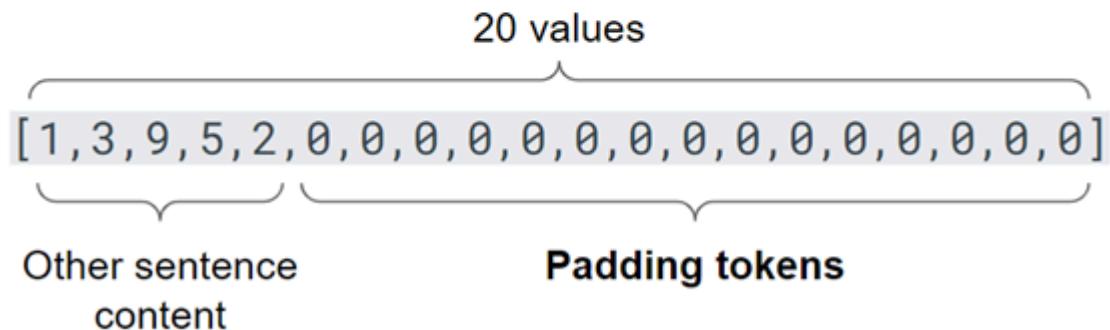


Figure 6.4.3.2 - Padding Tokens fill the unused parts of input if it is a short sentence.

<START>: Next, the START token is represented by the number 1. This is always the first token in the array. In the example shown in Figure 6.4.3.3., you can see that the array of numbers starts with a "1". This just defines the start of a sentence.

<START> 1

This is simply always the 1st token to indicate start of sentence.

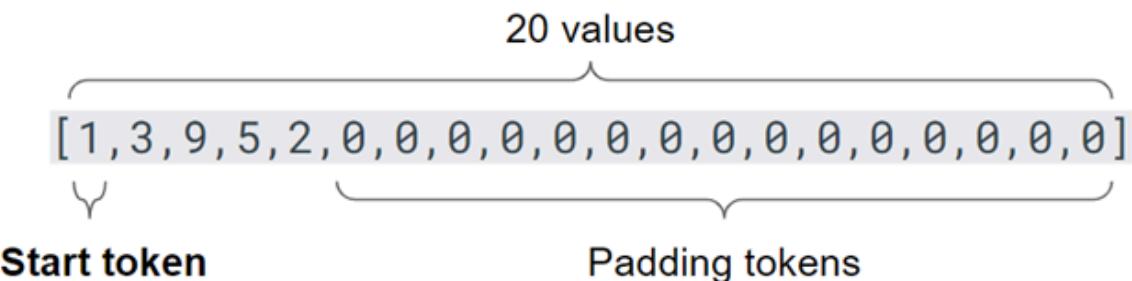


Figure 6.4.3.3 - Start Token

<UNKNOWN>: The UNKNOWN token is represented by the number 2. In the example shown in Figure 6.4.3.4, there is a single UNKNOWN token. This represents a word that does not exist in the ‘vocab’ file. When the model does not understand a particular word, it assigns the number 2 to it instead.

<UNKNOWN> 2

Used when a word is used that does not exist in this vocab file.

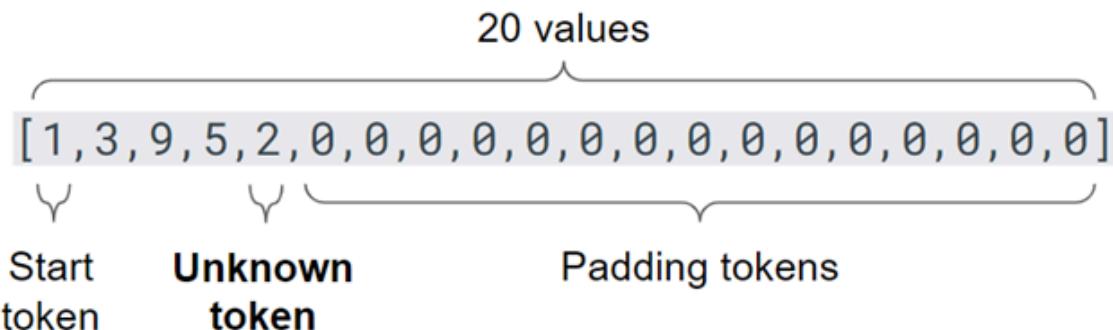
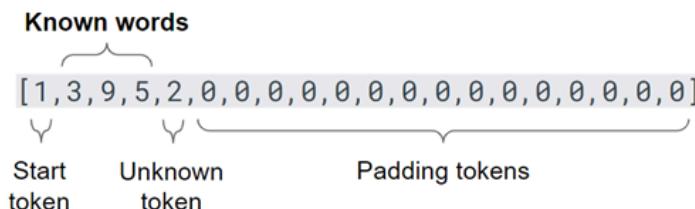


Figure 6.4.3.4 - Unknown Token

Adding Known Words

Observe the images in Figure 6.4.3.5. Note the numbers 3, 9, and 5 after the START token. These numbers represent the words ‘I’, ‘like’, and ‘video’ respectively (see Figure 6.3.4.5). This is how a sentence is encoded in a form the model can understand for words it knows about.

All other known words are represented by some integer.



A screenshot of a terminal window titled 'vocab — commentspamjs'. The window displays a list of words and their corresponding integer indices:

Index	Word	Index
1	<PAD>	0
2	<START>	1
3	<UNKNOWN>	2
4	i	3
5	check	4
6	video	5
7	song	6
8	com	7
9	please	8
10	like	9

Figure 6.4.3.5. Known Tokens

Note that since the START token takes up the first position in the array, any sentence input to this particular model can have up to 19 words before you would need to split it up.

When encoding longer sentences, you might end up with multiple short sentences that you need to classify individually, and then perform a final analysis based on how many of them were deemed spam or not spam manually.

Embeddings

In this step, you understand how the words in the ‘vocab’ file are chosen.

Figure 6.4.3.6. shows how words can be represented on a scale across several dimensions as you saw before. Each word is represented by a vector (7 dimensional in this case). These vectors are called embeddings. They can be used to establish a ‘direction’ for that word based on certain criteria to solve a problem.



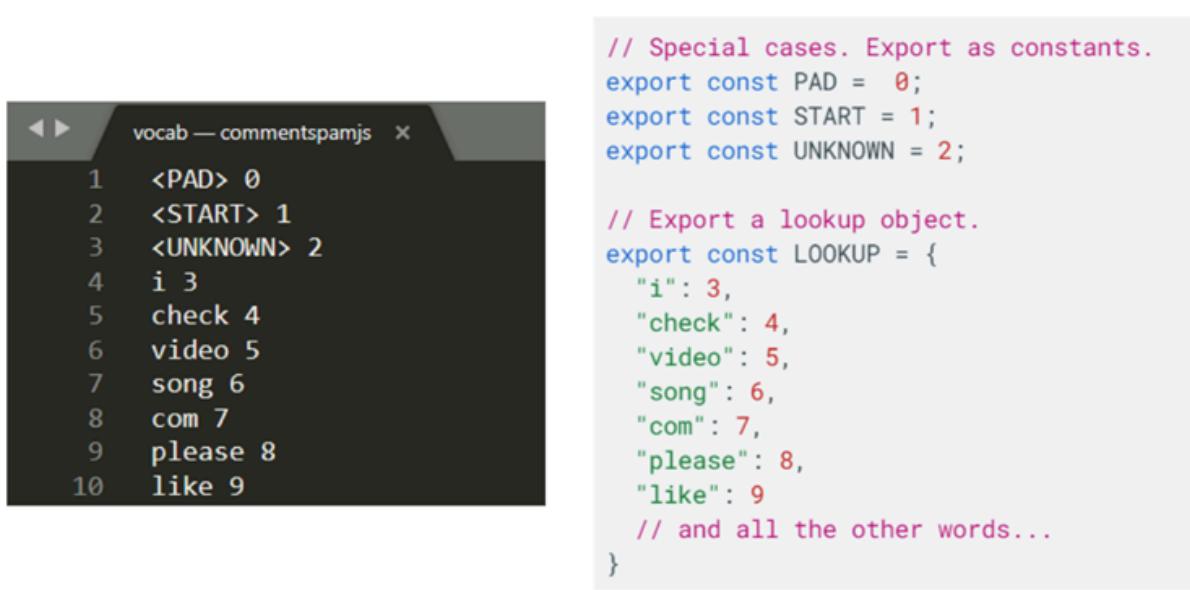
Figure 6.4.3.6. Word Embeddings

Words with similar meanings that are used frequently in spam messages have their vectors point in a similar direction, whereas vectors for words that may be associated with non spam may point in a different direction. The model learns which words that contribute the most in classifying words as spam and not spam. The ML engineer can then select the top few of these words (maybe the top 1000 for example) which are stored in the ‘vocab’ file and are then used to encode new sentences for the model to predict.

It should be noted that the numbers in the vocab file essentially map to known word vectors that are stored in a special “embeddings” layer in the pre-trained model itself. The model will convert these integer number lookups you have in the vocab file into the vectors you saw on the previous slide that it has learnt.

Converting the vocab file to be JS friendly

Currently, the vocab file is not in a very JS-friendly format to digest and use in your web app to help you convert words to numbers. Looking at Figure 6.4.3.7. The image on the right shows a format that would be much more suitable for use in a JS application.



The figure shows a terminal window with two tabs. The left tab is titled 'vocab — commentsspamjs' and contains a list of words with their corresponding integer encodings:

Index	Word	Encoding
1	<PAD>	0
2	<START>	1
3	<UNKNOWN>	2
4	i	3
5	check	4
6	video	5
7	song	6
8	com	7
9	please	8
10	like	9

The right tab contains the generated JavaScript code ('dictionary.js'):

```
// Special cases. Export as constants.  
export const PAD = 0;  
export const START = 1;  
export const UNKNOWN = 2;  
  
// Export a lookup object.  
export const LOOKUP = {  
    "i": 3,  
    "check": 4,  
    "video": 5,  
    "song": 6,  
    "com": 7,  
    "please": 8,  
    "like": 9  
    // and all the other words...  
}
```

Figure 6.4.3.7. JS - Friendly vocab File

For this exercise, we have already converted the vocab file to this form by writing a small web app that takes a raw vocab file as input and reformats it to be in the new form shown and saved as ‘dictionary.js’ instead.

Notes

- The special encodings for PAD, START, and UNKNOWN are represented by their own constants that can be used as needed. All the other words are stored in a LOOKUP object where the word is the object’s property name and the value of that property is the number encoding that represents the word.
- You can check this [link](#) to see how the ‘dictionary.js’ file is generated from a vocab file as an input

Why this works: JavaScript Object properties

An object property name can be any valid JavaScript string or anything that can be converted to a string, including the empty string. However, any property name that is not a valid JavaScript identifier, for example, a property name that has a space or a hyphen, or that starts with a number, can only be accessed using the square bracket notation.

As you know that the words stored are guaranteed to be in a valid form to use in this case, as long as you use square bracket notation, you can create an effective lookup mechanism through this simple transformation.

In this first step, you import the new dictionary.js file created and use it to tokenize a comment so it can be sent through the model.
Add the following code to the end of the **script.js** file.

```
import * as DICTIONARY from
'https://storage.googleapis.com/jmstore/TensorFlowJS/EdX/SavedModels/spam/diction
ary.js';

const ENCODING_LENGTH = 20;

function tokenize(wordArray) {

  let returnArray = [DICTIONARY.START];

  for (var i = 0; i < wordArray.length; i++) {

    if (wordArray[i] === '') {
      returnArray.push(DICTIONARY.EMPTY);
    } else {
      let word = wordArray[i];
      let index = DICTIONARY[word];
      if (index === undefined) {
        returnArray.push(DICTIONARY.UNKNOWN);
      } else {
        returnArray.push(index);
      }
    }
  }

  return returnArray;
}
```

```
let encoding = DICTIONARY_LOOKUP[wordArray[i]];

returnArray.push(encoding === undefined ? DICTIONARY.UNKNOWN : encoding);

}

while (returnArray.length < ENCODING_LENGTH) {

    returnArray.push(DICTIONARY.PAD);

}

console.log(returnArray);

}

return tf.tensor2d([returnArray]);
}
```

Steps

1. Import the dictionary.js file at the URL shown and assign it to a constant called ‘DICTIONARY’.
2. Define another constant called ‘ENCODING_LENGTH’ that is set to 20 (as this model currently supports 20 inputs only).
3. Define a ‘tokenize’ function that takes an array of strings as inputs. These are the words in a given sentence split by the spaces between them.
4. Define a variable called ‘returnArray’ that’s a 1D array that contains the ‘DICTIONARY.START’ token in the first position.
5. Next, loop through all the words in the ‘wordArray’. For each word in the array, check ‘DICTIONARY.LOOKUP’ using the square bracket notation to see if the word exists in the ‘LOOKUP’ object.

- a. If it does, a valid number will be returned and assigned to a variable called encoding.
 - b. If the word does not exist in the LOOKUP object then encoding will be ‘undefined’.
 - c. Using the ternary operator in JS you can then check if the encoding is undefined. If it is, push the ‘DICTIONARY.UNKNOWN’ token to the array, otherwise, push the valid number that was returned in the ‘encoding’ variable.
6. You can then add another loop to ensure padding is added in case the sentence was less than 19 words. Keep adding the padding token while the ‘returnArray’ length is less than the desired ‘ENCODING_LENGTH’.
7. Finally, log the returnArray to inspect the output, and then return a tensor2d containing these values.

In the next step, you complete the code for the ‘handleCommentPost function’ that was defined in Lesson 4.1.

Place the following code at the location of the TODO comment towards the end of the ‘handleCommentPost’ function.

```
let lowercaseSentenceArray = currentComment.toLowerCase().replace(/[^\w\s]/g,
' ').split(' ');

let li = document.createElement('li');

let p = document.createElement('p');

p.innerText = COMMENT_TEXT.innerText;

let spanName = document.createElement('span');

spanName.setAttribute('class', 'username');

spanName.innerText = currentUserName;
```

```
let spanDate = document.createElement('span');

spanDate.setAttribute('class', 'timestamp');

let curDate = new Date();

spanDate.innerText = curDate.toLocaleString();

li.appendChild(spanName);

li.appendChild(spanDate);

li.appendChild(p);

COMMENTS_LIST.prepend(li);

COMMENT_TEXT.innerText = '';

loadAndPredict(tokenize(lowercaseSentenceArray), li).then(function() {

    POST_COMMENT_BTN.classList.remove(PREPROCESSING_CLASS);

    COMMENT_TEXT.classList.remove(PREPROCESSING_CLASS);

});
```

Steps

1. Make the ‘currentComment’ text lowercase, and then use a regular expression to remove any non alphanumeric / whitespace characters found. This is to ensure no special characters remain in the sentence. Once the words do not contain any special characters, you can then split the words by spaces that separate them. This returns an array of words that are stored in the variable named ‘lowercaseSentenceArray’. [Regular expressions](#) are very powerful when you want to perform advanced searches and replacements like this.

2. Now create a list element in memory that you will add to the document later. This will contain new comments.
3. Create HTML elements to store the 'COMMENT_TEXT' in a paragraph element, the username of the person posting, along with the current date in span elements, along with the appropriate CSS classes set.
4. Append the name, date, and paragraph to the list item defined above. At this point, the list item has everything it needs to be rendered correctly, so add it to the 'COMMENTS_LIST' element using the prepend method so it appears at the top of the list.
5. Clean up the 'COMMENT_TEXT' by setting its text to nothing to reset it once the comment is posted.
6. Next, you can call the 'loadAndPredict' function, and pass to it the tokenized version of the comment by running the 'lowercaseSentenceArray' through the tokenize function you defined in the previous step.
7. At this point, you also pass the list item you just created so that the loadAndPredict function is able to set the correct class for spam or not on the list item based on the prediction's outcome to change its rendered style appropriately.
8. As 'loadAndPredict' is an async function, use the 'then' keyword to wait for results to come back. An anonymous function is defined here. This function just removes the 'PROCESSING_CLASS' from the 'POST_COMMENT_BTN' and the 'COMMENT_TEXT' element to make them visible again now that processing is complete.

Finally in the last step of this section, you add code to the previously defined 'loadAndPredict' function.

Head to this function and update it with the code shown below.

```
async function loadAndPredict(inputTensor, domComment) {  
  
  // Load the model.json and binary files you hosted. Note this is  
  
  // an asynchronous operation so you use the await keyword  
  
  if (model === undefined) {  
  
    model = await tf.loadLayersModel(MODEL_JSON_URL);  
  
  }  
  
  // Once model has loaded you can call model.predict and pass to it  
  
  // an input in the form of a Tensor. You can then store the result.  
  
  let results = model.predict(inputTensor);  
  
  // Print the result to the console for us to inspect.  
  
  results.print();  
  
  let dataArray = results.dataSync();  
  
  if (dataArray[1] > SPAM_THRESHOLD) {  
  
    domComment.classList.add('spam');  
  
  }  
}
```

Steps

1. Start by adding a second parameter called ‘domComment’. This is a reference to the comment list item that is about to be rendered to the

document. This is done so that you can style the element appropriately, based on whether or not the comment is considered by the model to be spam.

2. Next, you replaced the TODO in the previously written code with the following changes:
 - a. First, you call the ‘dataSync’ method on the results tensor to access its contents.
 - b. Then check if the second element that’s returned in the ‘dataArray’ is greater than the SPAM_THRESHOLD you defined at the start of this exercise.
 - i. Remember the second element in the output tensor represents how likely the comment is to be spam as you saw in the labels.txt file so you want to check that output’s score.
 - ii. If it is greater than the SPAM_THRESHOLD, you can add the spam class to ‘domComment’ list item element to style it appropriately when it is rendered to the document.

At this stage, if you run the web app in its current state and write a spam comment it will look like this (as seen in Figure 6.4.3.8):

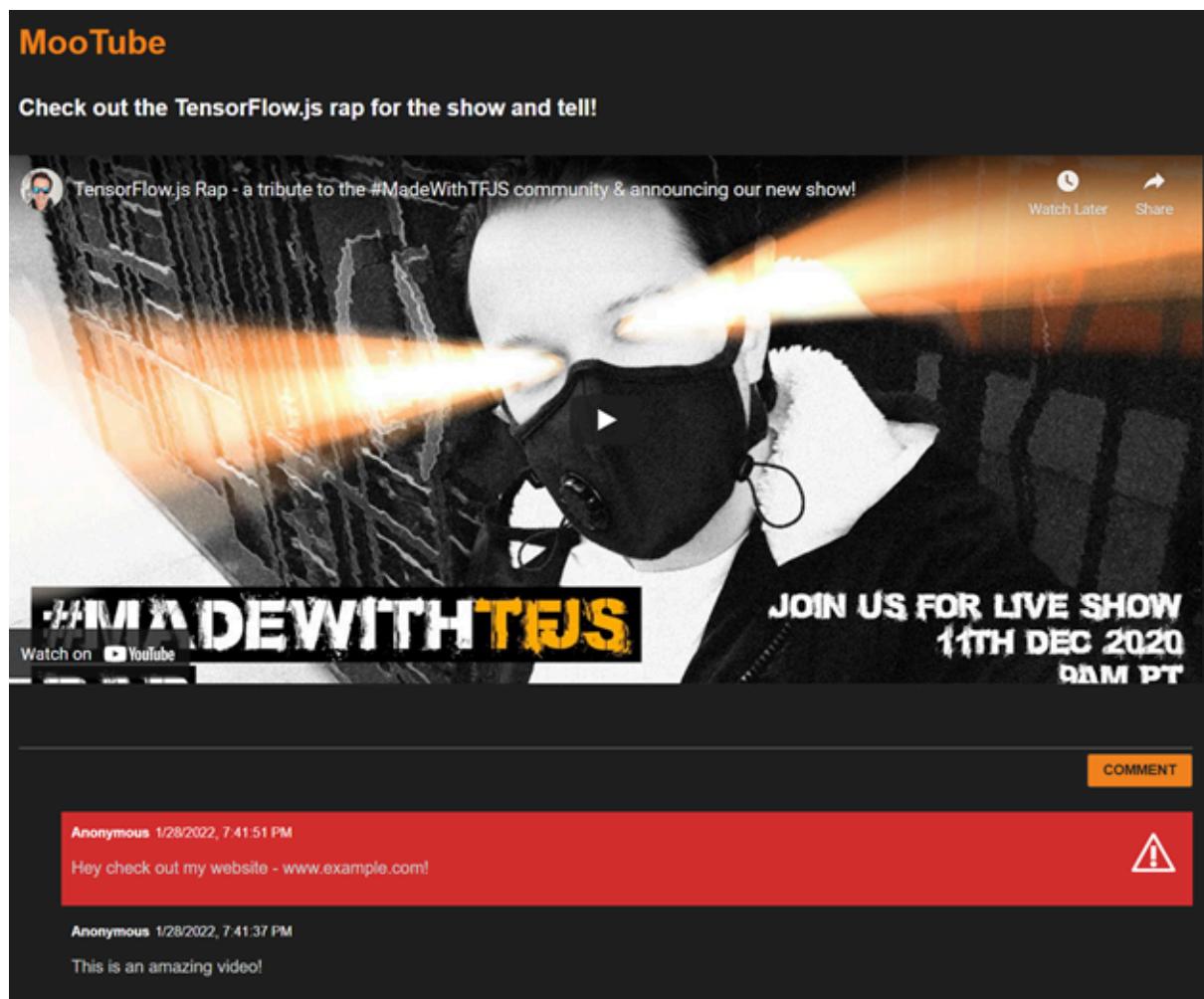


Figure 6.4.3.8 - Comment Spam Detection

The last piece of this interactive app is to use WebSockets via Node.js to relay the non-spam comments to other users who have the page open so only the spam-free comments get sent. In the next section, you will learn how to do that.

Comment Spam Detection - Web Sockets

In this lesson, you use WebSockets in Node.js to communicate comments that are deemed non-spam between web pages that are open.

WebSockets

- WebSockets enable two-way communication between multiple clients and a server in real-time. Unlike REST-based APIs, WebSockets are very efficient as they have persistent communication channels instead of having to recreate a connection every time.

- Using a WebSocket, a client can send a message to a server, and the server can relay the message rapidly to other connected clients (see Figure 6.4.4.1.). Since the connection remains open for the session duration, the data is both sent and received efficiently.

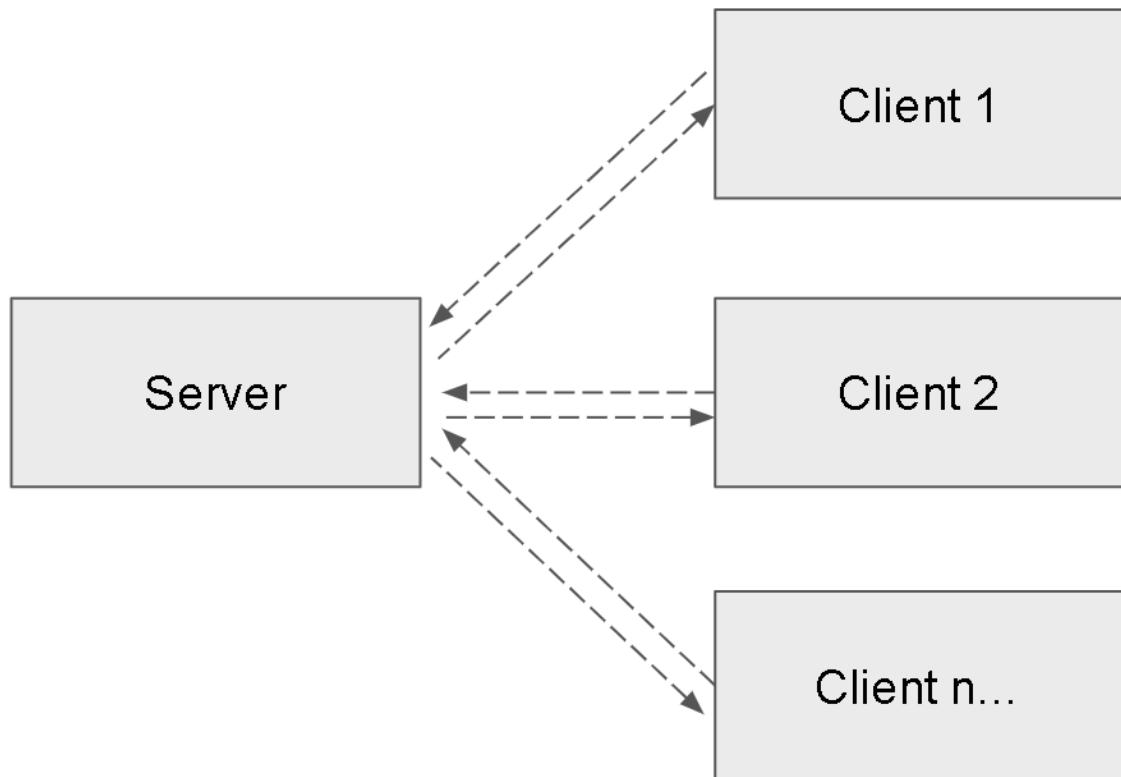


Figure 6.4.4.1 - WebSockets
Communication Through WebSockets

In Figure 6.4.4.2, client 1 sends a comment message to the server with some data contained within it after determining it was not a spam comment.

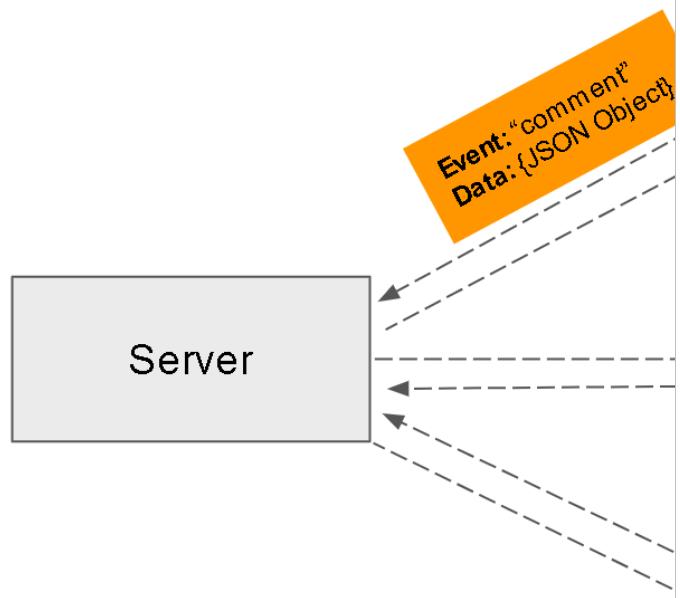


Figure 6.4.4.2 - WebSocket: Client to Server Communication

In Figure 6.4.4.3, the server receives the data and calls the event handler for this “comment” event type.

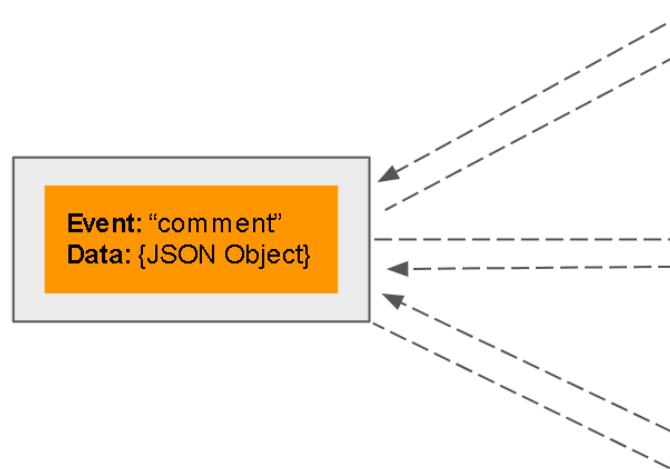


Figure 6.4.4.3 - WebSocket: Server Event Handling

In Figure 6.4.4.4, the server broadcasts the received comment message to all other connected clients so they can get a copy of the comment and render it to their user interfaces for the users to see.

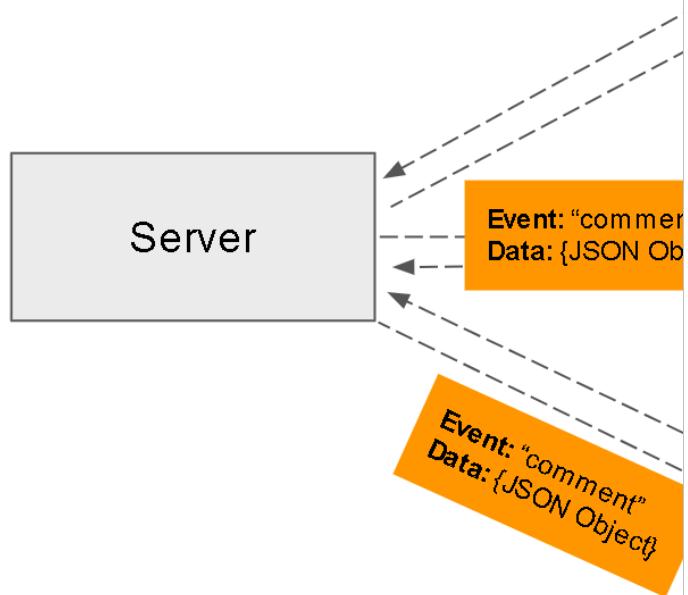


Figure 6.4.4.4 - WebSocket: Server toClients Communication

In order to communicate comment data between users who have your index.html page open in their browser, you will use ‘socket.io’, which is a very popular Node.js WebSocket library and can be set up in just a few lines of code.

Installing Socket.io on Node.js and Glitch.

Open the ‘package.json’ file in your Glitch project and add ‘socket.io’ to the dependencies (see Figure 6.4.4.5). Ensure that the ‘socket.io’ version is at least 4.0.1 using the special string annotation as shown. Remember that you need to add a comma on the line before as this is contained within a JSON object. Glitch will then rebuild the backend server to install socket.io, which you can then use immediately.

package.json

```
{  
  "name": "tfjs-with-backend",  
  "version": "0.0.1",  
  "description": "A TFJS front end with thin Node.js backend",  
  "main": "server.js",  
  "scripts": {  
    "start": "node server.js"  
  },  
  "dependencies": {  
    "express": "^4.17.1",  
    "socket.io": "^4.0.1"  
  },  
  "engines": {  
    "node": "12.x"  
  }  
}
```

Figure 6.4.4.5 - Installing socket.io for Node.js

In the next unit, you will update the Node.js server code in the ‘server.js’ file.

Updating server.js

Steps: Copy and update server.js with the new code shown which is explained below.

```
const http = require('http');  
const express = require("express");  
const app = express();  
const server = http.createServer(app);  
  
var io = require('socket.io')(server);  
app.use(express.static("www"));  
  
app.get("/", (request, response) => {
```

```

        response.sendFile(__dirname + "/www/index.html");
    });
}

io.on('connect', socket => {
    console.log('Client connected');
    socket.on('comment', (data) => {
        socket.broadcast.emit('remoteComment', data);
    });
});

const listener = server.listen(process.env.PORT, () => {
    console.log("Your app is listening on port " + listener.address().port);
}
)

```

1. First, you import the ‘http’ library using the standard Node.js’ require statement.
2. Then add a line to create a new ‘http’ server using ‘http.createServer’ and passing the existing express ‘app’ as the parameter.
3. Now, make sure ‘socket.io’ uses the ‘http’ server you created above by requiring it and then calling the returned class with the ‘server’ object you created on the previous line. This allows ‘socket.io’ to expose and serve the client-side ‘socket.io’ library files that will be used later in the ‘index.html’ client-side code.
4. Move past the next few lines of code and then add socket.io code to handle a client connecting to this server. Here you can add an event listener for a connection event using ‘io.on’ and listening for the ‘connect’ event. Once this event fires, it will call an anonymous

function that is passed the client socket connection as a parameter as shown:

```
io.on('connect', socket => {  
  ...  
});
```

5. Moving inside this anonymous function, the new client connection can be logged to the server logs using ‘console.log’. Note that this console.log occurs on the server via Node.js and is not printed to the browser’s console. It is printed on the server-side logs on the cloud machine Glitch is hosting for us. As such, Glitch has a special ‘logs’ button that can be used to see the server-side logs if needed.
6. The next line in the anonymous function is a custom event listener for events that come from the ‘socket’ that was returned that represents the connection to the client web page. Here you add a custom ‘comment’ event listener using ‘socket.on’ and passing a string as the first parameter representing the event type you want to listen for.

In ‘socket.io’, you can send custom message types like this that you can then listen for. Later, in your client-side code, you will send a message with the same event type so it gets picked up by this server-side listener which will broadcast the comment to other connected clients as you will see in the next step.

Note: Any custom event like this that’s detected will have some data associated with it. Typically, this is just a regular JSON object that ‘socket.io’ automatically parses and converts to a usable JavaScript object. So when a comment event is sent to this socket connection it will call an anonymous function with 1 parameter containing the data in the message that was sent as a parameter as shown:

```
socket.on('comment', (data) => {
  ...
});
```

7. Finally, once you have the received data for the comment event that fired, you can call ‘socket.broadcast.emit’ with the first parameter being a new custom event to emit to connected clients called “remoteContent” and the second parameter being the data object you just received. This informs ‘socket.io’ to relay the data using a custom event called ‘remoteComment’ to any other sockets that are connected to the server. The remaining code in the ‘server.js’ file is unchanged.

Now that ‘socket.io’ is set up on the server-side, you need to import the new ‘socket.io’ library to your front-end code.

Edit the code in the **index.html** file as follows:

In the body of your ‘index.html’ file, add a new ‘script’ import near the end of the file but just before the ‘script.js’ import as shown below. ‘socket.io’ will serve its own client-side library files on the ‘express’ server you just edited for convenience at the server URL shown.

```
<h1>MooTube</h1>
<h2>Check out the TensorFlow.js rap for the show and tell!</h2>
<iframe width="100%" height="500"
src="https://www.youtube.com/embed/RhVs7ijB17c" frameborder="0" allow="autoplay;
encrypted-media; allow-presentation" allowfullscreen></iframe>

<section id="comments" class="comments">
  <div id="comment" class="comment" contenteditable></div>
  <button id="post" type="button">Comment</button>
  <ul id="commentsList">
    <li>
      <span class="username">SomeUser</span>
      <span class="timestamp">2/11/2021, 3:10:00 PM</span>
      <p>Wow, I love this video, so many amazing demos!</p>
    </li>
  </ul>
```

```

        </section>

    <script
src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@3.11.0/dist/tf.min.js"
type="text/javascript"></script>
    <script src="/socket.io/socket.io.js"></script>
    <script type="module" src="/script.js"></script>
</body>

```

In the next unit, you will complete the JavaScript client-side code to actually send comments to the server to broadcast and receive comments from the server and render them to the page.

First, add the following code to the end of the ‘**script.js**’ file

Steps

```

let socket = io.connect();

function handleRemoteComments(data) {
    let li = document.createElement('li');
    let p = document.createElement('p');
    p.innerText = data.comment;

    let spanName = document.createElement('span');
    spanName.setAttribute('class', 'username');
    spanName.innerText = data.username;

    let spanDate = document.createElement('span');
    spanDate.setAttribute('class', 'timestamp');
    spanDate.innerText = data.timestamp;

    li.appendChild(spanName);
    li.appendChild(spanDate);
    li.appendChild(p);

    COMMENTS_LIST.prepend(li);
}

socket.on('remoteComment', handleRemoteComments);

```

- 1. Start with a request to connect to the WebSocket server by calling ‘`io.connect()`’. This returns a reference to the opened**

socket connection to the Node.js' server. Store this connection as a variable named 'socket'.

2. Create a function named 'handleRemoteComments' that takes one parameter called data. This function is called whenever you receive data from the 'Node.js' server. This data object will contain an object with all the data needed to recreate a comment that was posted that you can use to create the appropriate list item HTML elements to render a comment to the page as you have done before. As you know the comment will only be received if not deemed to be spam, there is no need to set any additional spam class styles in this case.
3. Finally here add a special event listener for 'socket' events that are of the name "remoteComment", which when received, will call the 'handleRemoteComments' function above with the data received from the server. Note that this event name is the same as what you defined in 'server.js' earlier, this is what the server broadcasts to connected clients when relaying received messages as its message event type.

In the final step, you send a comment to the server so that it may be broadcast to other users if it is deemed to be not spam.

Steps

Update your 'loadAndPredict' function with the following code (explained below):

```
async function loadAndPredict(inputTensor, domComment) {  
  if (model === undefined) {  
    model = await tf.loadLayersModel(MODEL_JSON_URL);  
  }  
  
  let results = model.predict(inputTensor);  
  results.print();
```

```

let dataArray = results.dataSync();
if (dataArray[1] > SPAM_THRESHOLD) {
  domComment.classList.add('spam');
} else {
  // Emit socket.io comment event for server to handle containing
  // all the comment data you would need to render the comment on
  // a remote client's front end.
  socket.emit('comment', {
    username: currentUserName,
    timestamp: domComment.querySelectorAll('span')[1].innerText,
    comment: domComment.querySelectorAll('p')[0].innerText
  });
}
}

```

1. Add an ‘else’ statement at the section where you were checking if a comment was greater than the defined `spam_threshold`. In the case the comment is not spam, it needs to be sent to the other users.
2. Call ‘socket.emit’ with the custom event name of “comment” as the first parameter. Remember that your Node.js server is listening for this event type from the client. The second parameter is just an object with data you want to send.
3. In this object that you will send to the server, specify the `username`, `timestamp`, and `comment` properties as shown. You can grab the values for the `timestamp` and `comment` from the ‘`domComment`’ element that was passed to this function. This object is then sent to the server which will then broadcast it to any other clients connected to the server who will then render the received comments using the ‘`handleRemoteComments`’ function you defined in the previous step.

Finally at this stage, if you open two live previews of your project in separate windows (or devices if you have multiple), and start typing comments, any comment that is not spam will appear on all the other windows and devices you have open.

If it is marked as spam, it will not be sent, so will not appear on the remote devices. Instead, it will show in red only on the window you typed it on, preventing potential spammers from spreading their messages.

At this point, it is worth testing the code on a variety of comments. You will notice that some things get still through or are incorrectly marked as spam. Also, note that the current code does not deal with sentences that are longer than 19 words - you may want to update the code to deal with that if you wish.

To refine the pre-made comment spam model to handle the edge cases you find, you will need to retrain it using your own custom data in addition to the original data it was trained on. You will explore how to do this in the next lesson.

Dealing with Edge Cases

When using pre-made models for comment spam detection, you will occasionally find comments that the model can not accurately classify. In this lesson, you learn how to retrain models to account for such edge cases.

Over the next four steps, you will explore how your pre-made model performed on a number of different comment types.

Remember, initially you set the threshold for spam to be 75% in your code in the prior section.

1. **True Negatives:** Comments that are not spam and are deemed not spam are known as true negatives (see Figure

6.5.1). These are legitimate comments and their percentages are all well below the spam threshold of 75%.

1. "Wow, I love that video, amazing work."
Probability Spam: **47.91%**
2. "Totally loved these demos! Got any more details?"
Probability Spam: **47.15%**
3. "What website can I go to learn more?"
Probability Spam: **15.32%**

2.

Figure 6.5.1 - True Negatives

3. **False Positives:** Comments that are not spam but are deemed spam are known as false positives. These comments are genuine questions that have been labeled spam. At this point, after checking the probabilities of these sentences, you could just increase the confidence threshold of the spam classification to be over 98.5%. In this case, all the sentences in Figure 6.5.2. would be classified as non-spam and all the examples from 6.5.1 would also still be valid.

1. "Can someone link the website for the mask he is wearing?"
Probability Spam: **98.46%**
2. "Can I buy this song on Spotify? Someone let me know!"
Probability Spam: **94.40%**
3. "Can someone contact me with details on how to download TensorFlow.js?"
Probability Spam: **83.20%**

Figure 6.5.2. False Positives

4. **True Positives:** Comments that are spam and are deemed spam are true positives. In Figure 6.5.3, observe that comment number 1 is a true positive since it is spam and has been classified as spam with this new increase in the spam threshold



to 98.5%. However, comments two and three are also spam but have not been classified as spam due to the change in the spam threshold. You could try to lower this to 96%, but if you did that, then one comment from the previous section in 6.5.2 would then be misclassified. Maybe that is acceptable though. So let's continue.

1. *"This is cool but check out the download links on my website that are better!"*
Probability Spam: **99.77%**
2. *"I know some people who can get you some medicines just see my pr0file for details"*
Probability Spam: **98.46%**
3. *"See my profile to download even more amazing video that are even better! <http://example.com>"*
Probability Spam: **96.26%**



Figure 6.5.3. True Positives

5. **False Negatives:** Comments that are spam, but are deemed not spam are false negatives (see Figure 6.5.4). Since the confidence percentage is too low, changing the threshold value will not lead to better results. In this scenario, the only option is to retrain the model to account for the edge cases you have discovered, so that it gets better at classifying such sentences.

1. "See my profile to download even more amazing video that are even better!"
Probability Spam: **7.54%**
2. "Get a discount on our gym training classes see pr0file!"
Probability Spam: **17.49%**
3. "omg GOOG stock just shot right up! Get before too late!"
Probability Spam: **20.42%**



Figure 6.5.4. False Negatives

The pre-made model you used in the last lesson was generated using TensorFlow's Model Maker, which was written in Python. In the next unit, you will start retraining the model using a Colab before converting and saving the model to the TensorFlow.js format to deploy and use in your web app.

Navigate to colab.research.google.com and create a new Colab notebook (see Figure 6.5.5).

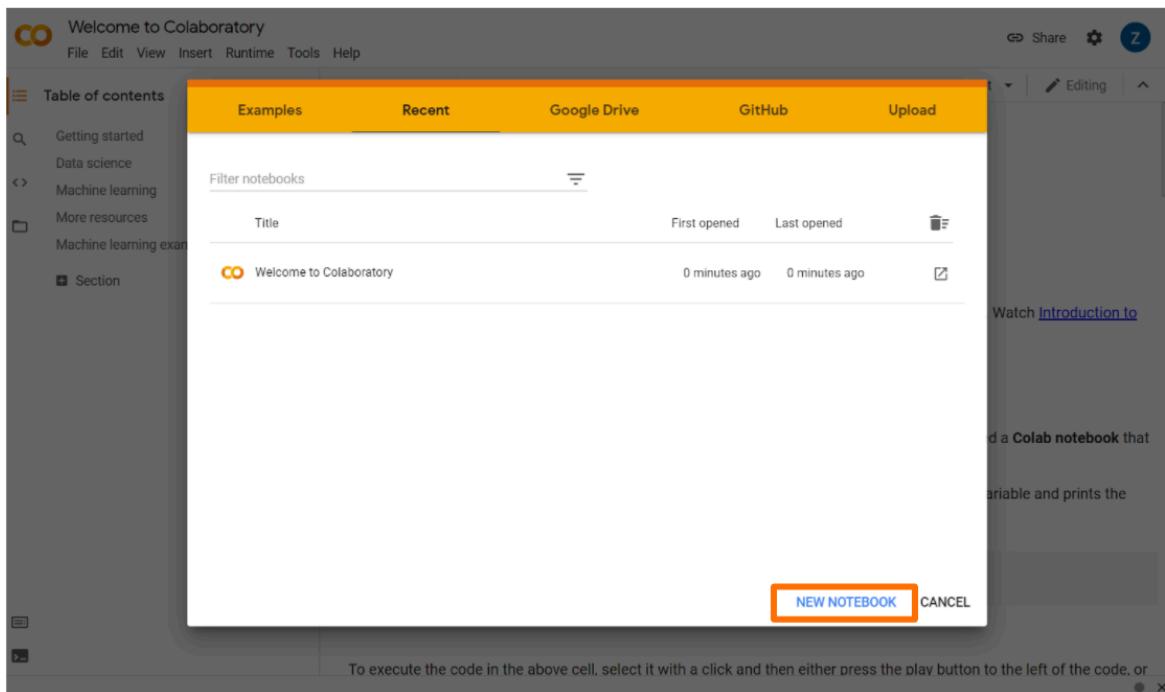


Figure 6.5.5 - Google Colab

Steps

1. You should now have a new Colab notebook to start editing as shown in figure 6.5.6.

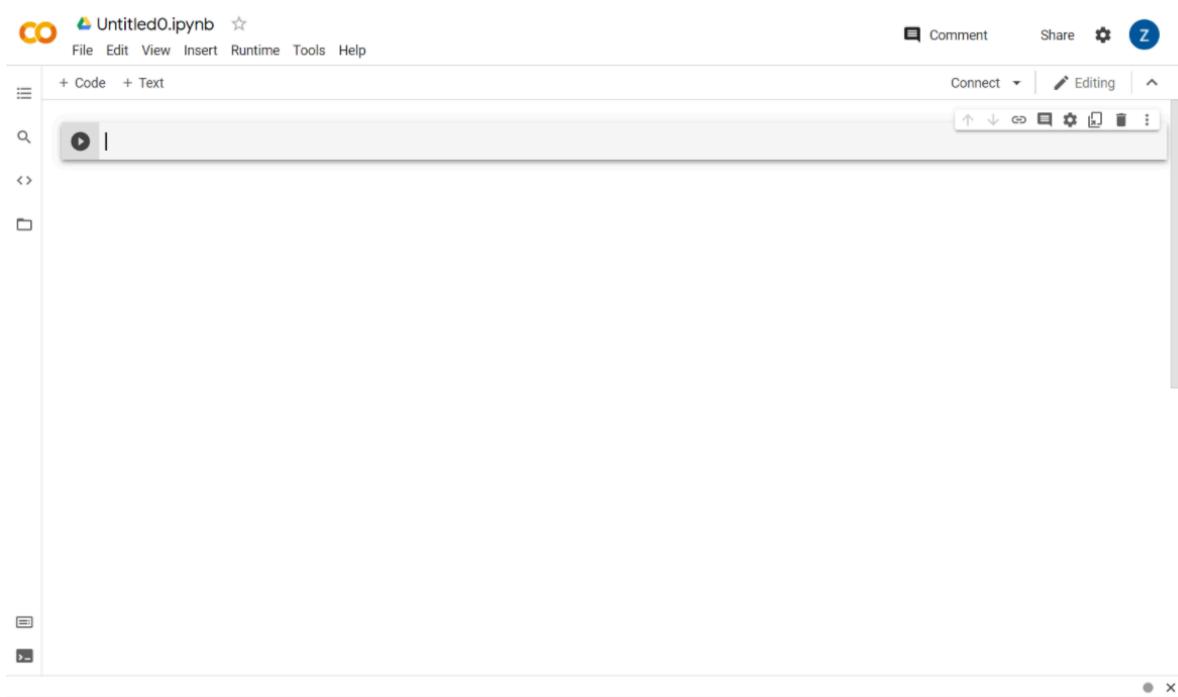


Figure 6.5.6 - New Notebook

2. Click on the Connect button at the top right to connect to a hosted runtime to start a server with TensorFlow and Python pre-installed as you have done before.

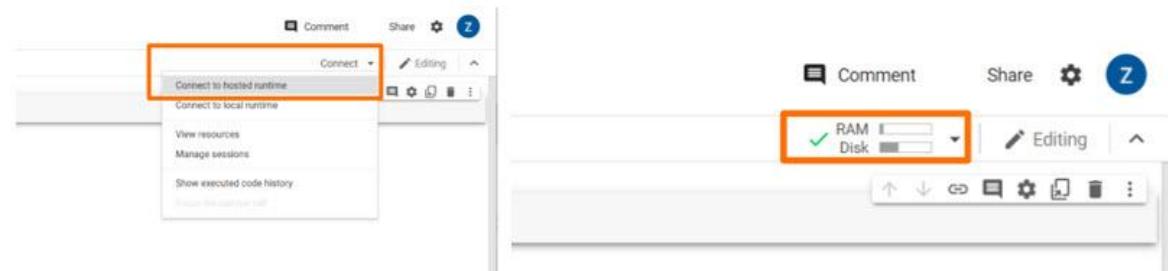


Figure 6.5.7 - Connect to Hosted Runtime

3. Type the code shown in Figure 6.5.8 into the first code cell to install Model Maker. Note that the line starts with an 'exclamation' mark to indicate that this is to be executed on the command line and is not a Python statement.

```
!pip install -q tflite-model-maker
```

Figure 6.5.8 - Code to install model maker.

4. Execute the cell by clicking the play button and waiting for the 'tflite' model maker to be installed (see Figure 6.5.9)

```
!pip install -q tflite-model-maker
```

501kB 8.5MB/s
174kB 18.2MB/s
5.5MB 11.9MB/s
1.2MB 44.4MB/s
112kB 48.7MB/s
849kB 39.7MB/s
92kB 10.1MB/s
133kB 46.4MB/s
71kB 7.9MB/s
706kB 44.9MB/s
645kB 45.7MB/s
1.0MB 43.3MB/s
122kB 51.2MB/s
102kB 10.8MB/s
37.6MB 116kB/s
358kB 56.9MB/s
194kB 56.3MB/s

Building wheel for fire (setup.py) ... done
Building wheel for py-cpuinfo (setup.py) ... done

Figure 6.5.9 -Installing Model Maker successfully

5. Add a new cell to continue coding (see Figure 6.5.10).

+ Code + Text

Insert code cell below
Ctrl+M B

[1] !pip install -q tflite-model-maker

Figure 6.5.10 - Insert Code Cell

6. Now add the code provided below in the new cell and execute this code.

```
import numpy as np

import os

from tflite_model_maker import configs

from tflite_model_maker import ExportFormat

from tflite_model_maker import model_spec

from tflite_model_maker import text_classifier

from tflite_model_maker.text_classifier import DataLoader

import tensorflow as tf

assert tf.__version__.startswith('2')
```

```
tf.get_logger().setLevel('ERROR')
```

7.

The code essentially imports useful utilities and Model Maker functions that are needed to set up the spam classifier model. The last section of the code ensures that the TensorFlow version is version 2 because this is required for the model maker to work correctly. In case any issues are found, an error message is logged.

8. You now need to add new training data to train your model. For this project, data has been hosted in a '.csv' file (where data is separated by commas) that contains the original training data used for training along with the extra examples to account for the edge cases you discovered earlier on.

See Figure 6.5.11 for a preview of the .csv file. Essentially the file contains lower case sentences with all punctuation removed, followed by a comma, and then followed by a classification of true or false, indicating whether the sentence was spam. The file contains 1,340 example sentences in total including the original training data the first pre-made model was trained with.

```

1316 make money with online trading,true
1317 the best way is to do online trading,true
1318 fly to the moon with online trading,true
1319 no need to work just do online trading,true
1320 can someone give me more details on the demos in the video,false
1321 can someone link me to the website for the mask he is wearing to
buy,false
1322 can i buy this song on spotify if so whats the link,false
1323 can someone contact me with details on how to download TensorFlow js
whats the link,false
1324 check out my website httpwwwexamplecomblah,true
1325 get a discount on our gym training classes httpwwwexamplecomgym,true
1326 omg goog stock just shot right up get before too late,true
1327 can you give me more details on the stuff in the video,false
1328 can you link to the website for the thing he is wearing to buy,false
1329 can i buy the song on spotify if so whats the link,false
1330 can you contact me with information on how to download TensorFlow js
whats the link,false
1331 check my website for free stuff httpwwwexamplecomblah,true
1332 get discount on our health training classes httpwwwexamplecomgym,true
1333 omg goog stock just shot right up get before too late,true
1334 what is the the object in the video,false
1335 what is the link to the website for the hat he is wearing to buy,false
1336 can i buy this song on google music if so whats the link,false
1337 please contact me with information on how to download TensorFlowjs
whats the link,false
1338 check my website for free stuff httpwwwexamplecomblah,true
1339 get discount on our health training classes httpwwwexamplecomgym,true
1340 omg goog stock just shot right up get before too late,true

```

Figure 6.5.11 - Training Data

- Add a new code block, then add the code provided below to download a .csv file with the training data. Note that this is all on one line of code.

```

data_file = tf.keras.utils.get_file(fname='comment-spam-extras.csv',
origin='https://storage.googleapis.com/jmstore/TensorFlowJS/EdX/code/
6.5/jm_blog_comments_extras.csv', extract=False)

```

The ‘tf.keras.utils.get_file’ function is used here to access a remote file and store it locally on the disk. In this case, it stores the file with the filename ‘comment-spam-extras.csv’ locally once downloaded. As this is not a zip file, ‘extract’ is set to false. Execute this cell block and then continue.

Note: Model Maker can train models from simple .csv files like the one you just saw. However, you need to specify the columns that hold the text, and columns that hold the labels which you will see further below how to do this. When using Model Maker, you usually don't build models from scratch, but use existing models to customize to your needs.

9. Add another code cell with the following code:

```
spec = model_spec.get('average_word_vec')

spec.num_words = 2000

spec.seq_len = 20

spec.wordvec_dim = 7
```

- a. Model Maker provides several pre-learned models that you can use, but the simplest and quickest, to begin with, is average_word_vec, which is stored in a variable called 'spec'.
- b. Specify the number of words you want the model to use. In this case use 2,000.

Note: It is important to arrive at an appropriate number of words that you want the model to store. If you use every word in the entire corpus, you could end up with the model trying to learn weights for words that are used only once. This will not help you classify future sentences accurately since these words are rarely used. Try to arrive at a reasonable number by analyzing how many words there are and their usage counts. Remember, using fewer words will lead to a smaller and faster

model, but may result in a less accurate model if too few are used. This takes experimentation to see what works well with your data.

- c. The next line of code represents the sequence length of the input sentence. This is the number of tokens the model can accept as input. Here, it is set to 20 just like the original pre-trained model.
- d. The last line involves a property called ‘wordvec_dim’, which stands for the number of word vector dimensions that are used to separate words by. These dimensions are the different characteristics, learned by the machine learning algorithm when training, by which any given word can be measured.

The model uses these dimensions to best associate words that are similar in a meaningful way. It uses these dimensions to then detect words that are more likely associated with spam. For instance, it may determine that spam emails are more likely to contain words that are both “medical” in nature and also related to “human body parts”, so a model may discover that using these two dimensions (medical and body parts) is useful to separate such data to classify well.

Note: A rule of thumb determined from research is that the fourth root of the number of words is appropriate for the number of dimensions. If you are using 2,000 words, a good starting point is 7 dimensions.

In this next step, you load the data from the .csv file you downloaded earlier and use it as training data for the model using the Python ‘DataLoader’ utilities class.

Steps

```
data = DataLoader.from_csv(  
  
    filename=data_file,  
  
    text_column='commenttext',  
  
    label_column='spam',  
  
    model_spec=spec,  
  
    delimiter=',',  
  
    shuffle=True,  
  
    is_training=True)  
  
train_data, test_data = data.split(0.9)
```

1. Start by specifying the following parameters:

- a. The filename which is simply equal to “data_file” variable you defined earlier.
- b. The column in the .csv file where sentences are found; in this case, the name of the column is ‘commenttext’.
- c. The column where the labels are found; in this case, the column name is ‘spam’.

Note: The names of the columns can be found on line 1 of the .csv file.

- d. Next Pass the ‘spec’ variable previously defined with all the configurations set
- e. Set the delimiter that the file uses. As this is a .csv file, a comma is used as the separator of columns.

- f. Set both ‘shuffle’ and ‘is_training’ values to ‘true’ so that the data gets shuffled and used for training.
2. Once the data is loaded, you can then split the data into training and testing data sets using ‘data.split’ and pass a value of 0.9. This ensures 90% of the data is reserved for training and 10% for testing.

Execute the code in this cell.

Build and train the new model

It’s time to build and train the model. Add a new code block with the following line of code:

```
model = text_classifier.create(train_data, model_spec=spec,  
epochs=50)
```

The ‘text_classifier.create’ method is called with three parameters. The first parameter passes the training data, the second parameter passes the model specification, and the third parameter is the number of epochs that the model needs to train for.

Execute the code in the current cell and wait for the model to train. You will see the epoch data printed as it completes each of the 50 epochs and if all goes to plan the loss should get lower and the accuracy higher over time as shown in figure 6.5.12.

```
Epoch 1/50  
28/28 [=====] - 4s 20ms/step - loss: 0.6793 - accuracy: 0.7130  
Epoch 2/50  
28/28 [=====] - 0s 4ms/step - loss: 0.6385 - accuracy: 0.8527  
Epoch 3/50  
28/28 [=====] - 0s 4ms/step - loss: 0.5983 - accuracy: 0.8874  
Epoch 4/50  
28/28 [=====] - 0s 4ms/step - loss: 0.5430 - accuracy: 0.9079  
Epoch 5/50  
28/28 [=====] - 0s 4ms/step - loss: 0.4931 - accuracy: 0.8955
```

Figure 6.5.12 - Epoch Data During Training

In the next step, you can convert and download the model to use in your JavaScript code.

Add a new code block and enter the code provided below:

```
model.export(export_dir="/tmp/js_export/",  
            export_format=[ExportFormat.TFJS, ExportFormat.LABEL,  
                          ExportFormat.VOCAB])  
  
!zip -r /tmp/js_export/ModelFiles.zip /tmp/js_export/
```

1. Call ‘model.export’ with 2 parameters. The first parameter defines the directory where the model should be exported to. Here a ‘js_export’ directory is created in the server’s ‘tmp’ folder. The second parameter specifies the export format. This function takes a further3 parameters. The first specifies it should be exported to the TensorFlow.js model format, the second requests an export of the labels file, and the third exports the vocab file.
2. Finally, call the zip utility on the command line to download all the resulting files in one go. Note that this line starts with an ‘exclamation’ point since it is not Python code. The resulting zip file is called ModelFiles.zip and will be stored in the ‘/tmp/js_export’ directory.

Once you have executed this code cell, navigate to the ‘tmp’ folder and find the ‘js_export’ subfolder where you can download the resulting zip file as shown below in figure 6.5.13.

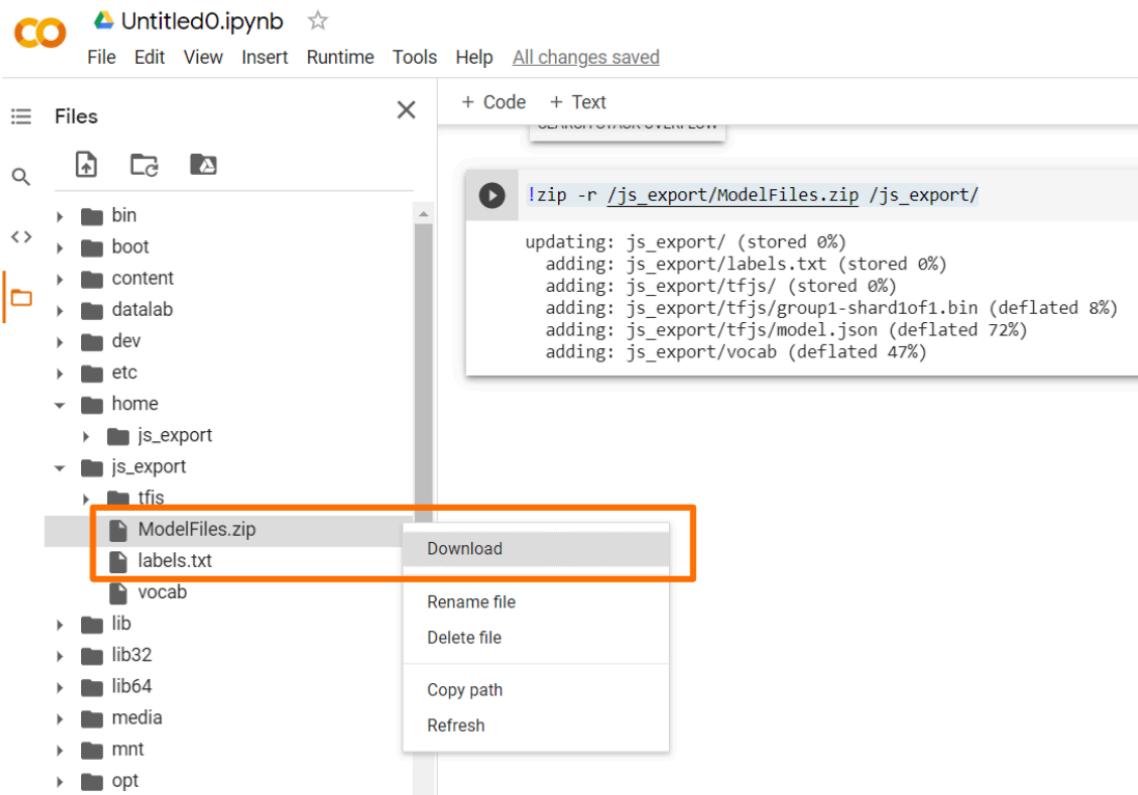


Figure 6.5.13 - Download Model Files

Note: You can use any .csv file with your own pairs of sentences and classification values if you wanted. That means you can even retrain this model to detect things beyond just spam if you have the necessary training data for it. For example, the same model could be trained to figure out the language in which a sentence has been written, in this case, you would have example sentences in different languages, and then a label for what it was eg “French, Spanish, German, English” etc. You could then use the model to predict what language a given input sentence was written in by updating the JS code logic you already wrote.

In the next session, you will see how to upload these files to ‘Glitch.com’ and use the newly trained model on your web app.

Using the Retrained Spam Detection Model

In this lesson, you learn how to take the new model files you trained and use them on Glitch.com within your existing web app.

Steps

1. Download and unzip the files for the model you just trained in the prior section and downloaded in the previous section.

Figure 6.6.1 shows the files contained within the files folders.

You can also [find the files I produced at this URL in case you were having issues generating them](#) in the Colab. Ensure that you have four files, the ‘vocab’ file, the ‘labels.txt’ file, the ‘model.json’ file, and the ‘.bin’ file.

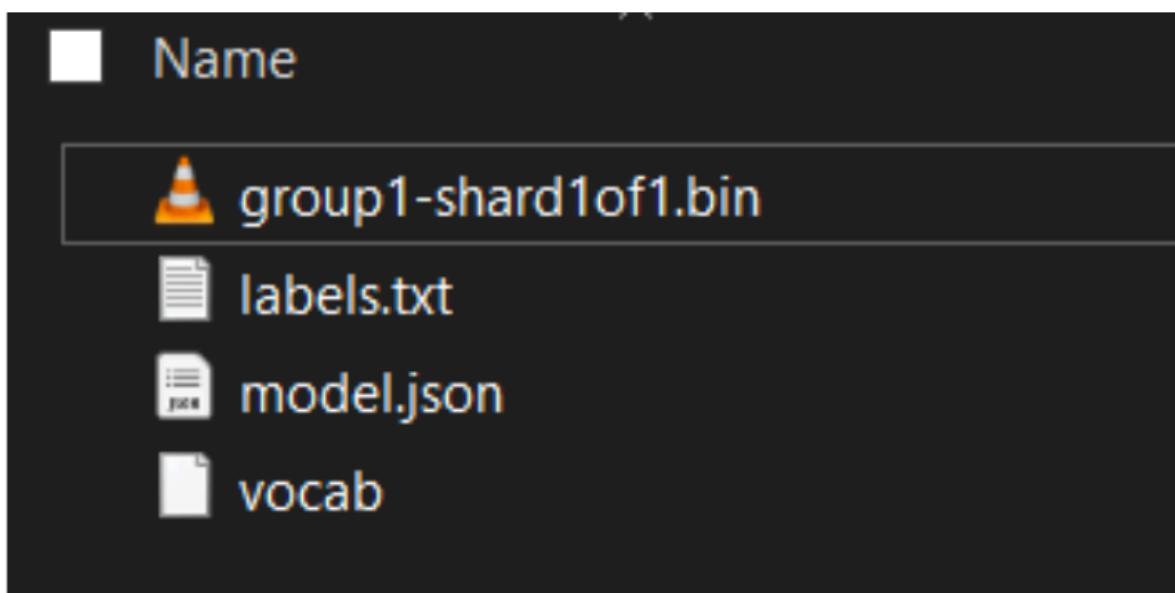


Figure 6.6.1 - Inspecting the Model Files

2. Place the ‘model.json’ and ‘.bin’ files on a web server so that they can be accessed from your web page. You can use any web server or a content delivery network (CDN) for this if you wish, but if you are using Glitch.com, you need to follow the instructions provided below.

3. First, remix a new copy of your completed working comment spam detection website that previously used the pre-made model. You can also [find a completed version of the project here](#) in case you need a working version to start from.

Fully working example of comment spam detection that uses socket.io on Node.js to send comments that are not spam only to other connected clients. Any spam comments do not get sent. Open on multiple devices or windows to try!

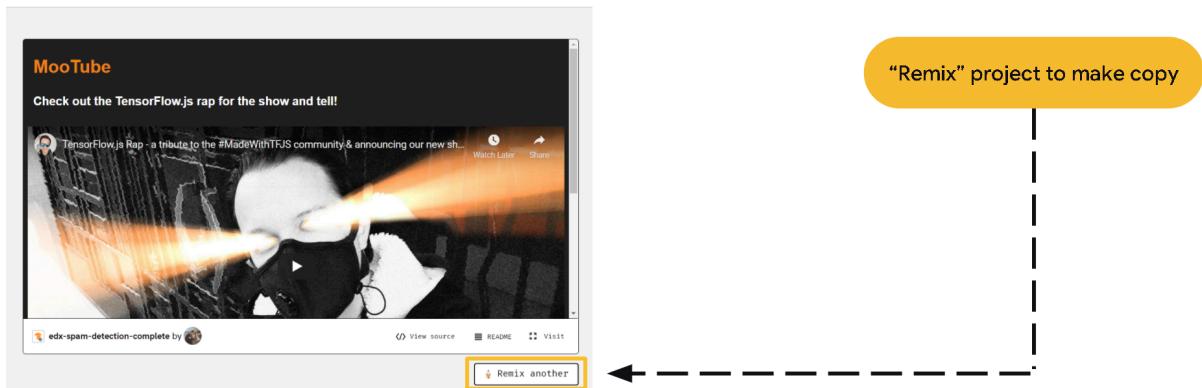


Figure 6.6.2 - Remix a copy of the completed spam detection web app on Glitch.com

4. Next, navigate to the special “assets” folder in the left panel of the Glitch project, and then select the button that says “upload an asset” (see Figure 6.6.3).

Note: Do not drag files to upload. Ensure you use the button since Glitch by default places files with certain extensions in different folders. Use the button to upload files to the same folder.

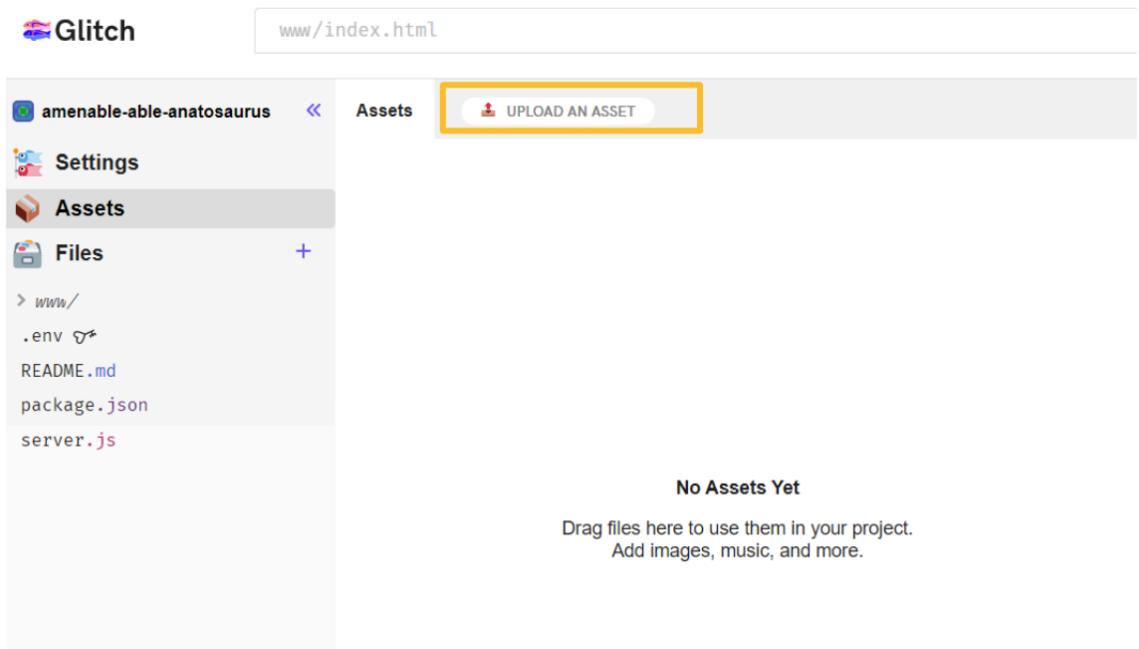


Figure 6.6.3 - Glitch.com - Assets Folder - use the upload an asset button as shown.

5. Upload the 2 model files, 'model.json' and 'group1-shard1of1.bin' to the assets folder (see Figure 6.6.4). You should then see the two files uploaded on the right of the screen as shown.

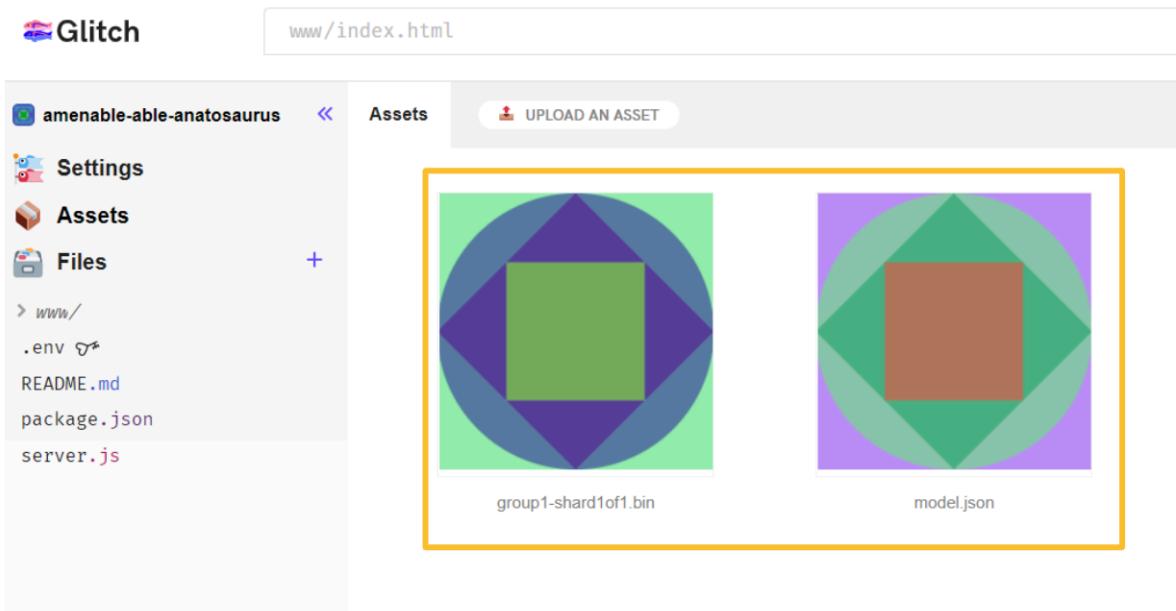


Figure 6.6.4 - Uploading Model Files

Note: If your operating system by default shows only ‘custom files’ for certain file extensions to choose from for upload, you may not be able to select your ‘.json’ and ‘.bin’ files. Be sure to select “all files” from the drop-down (see Figure 6.6.5) to make your local ‘binary’ and ‘json’ files visible to select for upload to Glitch.

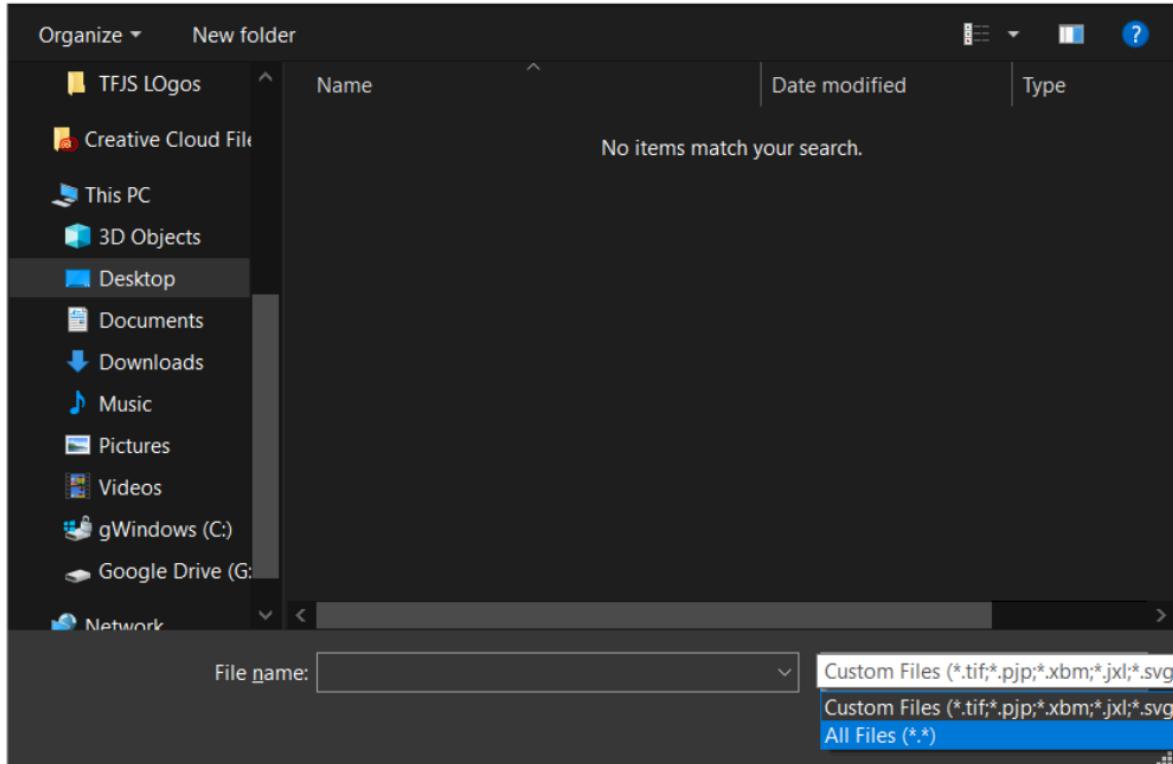


Figure 6.6.5 - Ensure visibility of json and bin files from your operating systems file selector to select them

6. Next select the ‘model.json’ file once it is uploaded and a modal window will open where you can copy the resulting URL that has been created by Glitch for this file. Copy this location by selecting the ‘Copy URL’ button (see Figure 6.6.6).

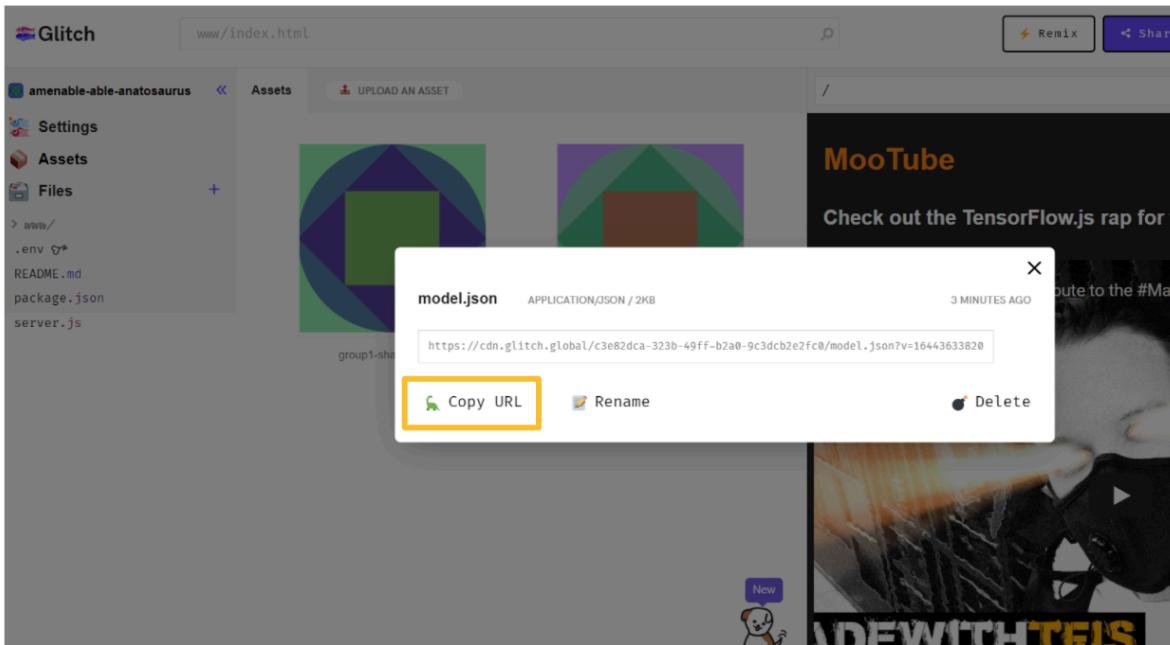


Figure 6.6.6 - model.json URL

7. Navigate to the ‘script.js’ file, and update the ‘MODEL_JSON_URL’ constant with the URL you just copied (see Figure 6.6.7). This will now load your newly trained model instead of the old one.

```

const MODEL_JSON_URL = 'https://cdn.glitch.global/c3e82dca-323b-49ff-b2a0-9c3dcb2e2fc0/model.json?v=1644363382026';
const SPAM_THRESHOLD = 0.75;
var model = undefined;

async function loadAndPredict(inputTensor, domComment) {
    // Load the model.json and binary files you hosted. Note this is
    // an asynchronous operation so you use the await keyword
    if (model === undefined) {
        model = await tf.loadLayersModel(MODEL_JSON_URL);
    }
}

```

Figure 6.6.7 - Update script.js File

8. Great! Next, it is time to generate a new ‘dictionary.js’ file so it has the new words to use. [Navigate to this URL](#) and use the web application hosted there to generate a new dictionary.js by uploading

your vocab.txt file to it (see Figure 6.6.8.). Save the resulting file when asked to download. You can also [view the source if you wish of this vocab.txt conversion web app](#) to see how the text file is converted to the required JavaScript friendly format.

Vocab to JS Converter for Model Maker

Use this page to take a vocab file produced by [Model Maker's average word embedding trained model](#) and reformat this to be more friendly for JS consumption.

You can [view the project source](#) for this page here if you are curious.

Select vocab file produced by model maker:

 vocab.txt

Figure 6.6.8 - Generate New dictionary.js using the linked website

9. Upload the new ‘dictionary.js’ file to the Glitch ‘assets’ folder (see Figure 6.6.9), and select it to copy the resulting URL.

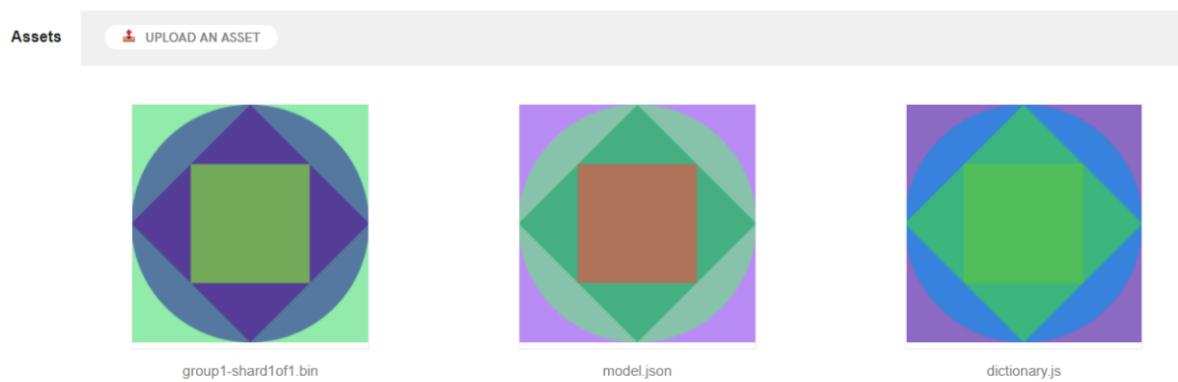


Figure 6.6.9 Upload dictionary.js File to the assets folder and copy its URL.

10. Navigate to the ‘script.js’ file, find the import for the ‘dictionary.js’ section (see Figure 6.6.10), and replace it with the URL you just copied for the new dictionary.js file that was uploaded.

```

import * as DICTIONARY from 'https://cdn.glitch.global/c3e82dca-323b-49ff-b2a0-9c3dcb2e2fc0/dictionary.js?v=1644364398225';

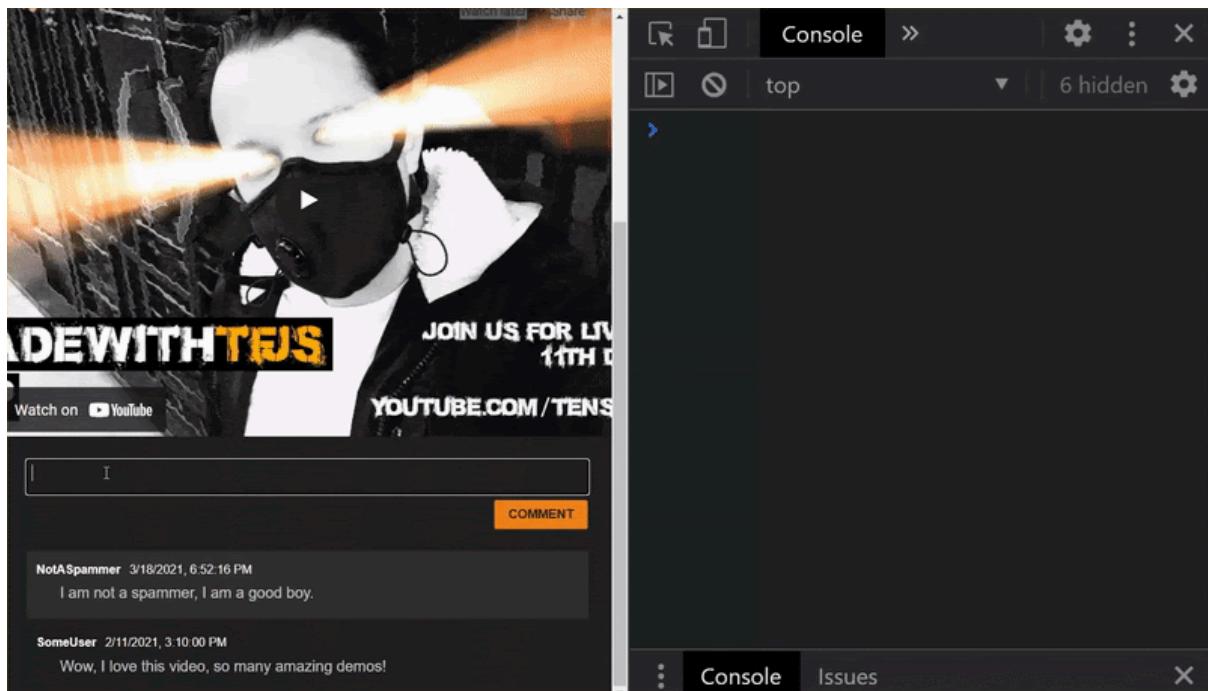
// The number of input elements the ML Model is expecting.
const ENCODING_LENGTH = 20;

/**
 * Function that takes an array of words, converts words to tokens,
 * and then returns a Tensor representation of the tokenization that
 * can be used as input to the machine learning model.
 */
function tokenize(wordArray) {
  // Always start with the START token.
  let returnArray = [DICTIONARY.START];
  // Loop through the words in the sentence you want to encode.
  // If word is found in dictionary, add that number else
}

```

Figure 6.6.10 - Import dictionary.js

Congratulations! You have now updated the demo to use your newly retrained spam detection model files. Try out the new model to see how it fares with some of the sentences that you had trouble with before and see if it has learned anything new. Hopefully, you will now see it is able to deal with these edge cases you were previously having trouble with as shown in the animation below:



In the past two lessons, you managed to retrain a spam detection model to update itself to work for the edge cases you found and deployed those changes to the browser with TensorFlow.js for a real-world web application.

In the next chapter, you will explore some of the more advanced ML models that exist and resources you can use to go even deeper into the world of Machine Learning.

To the Future and Beyond

What You Will Learn

In this chapter, you will:

- Explore and get inspired by a few exciting applications created by web developers using TensorFlow.js.
- Get introduced to advanced networks such as Autoencoders, Generative Adversarial Networks (GAN), Recurrent Neural Networks, and Transformer Networks.
- Learn how to work with the fast growing TensorFlow.js community, find further resources, and stay in touch after the course.
- In this lesson, you get a glimpse of how web engineers use ML in different fields using the same skills that you have learned in the course.

Domain

Overview

Human Body	<p>Human Library</p> <p>This library, available on GitHub, combines eight ML models that can understand the human body to gain extra insights into what may be going on in a given moment and provides detailed information to the end-user. Information includes 3D face detection & rotation tracking, face description & recognition, body pose tracking, 3D hand & finger tracking, iris analysis, age estimation, emotion prediction, gaze Tracking, gesture recognition, and body segmentation.</p>
Health and Fitness	<p>Welcome to YogAI</p> <p>This is a yoga teaching application that uses TensorFlow.js along with the older PoseNet pre-made model. The app correctly understands when you have held a certain pose for a certain number of seconds before prompting you to move on to the next one. We are seeing newer versions of this sort of application also appear that use our newer MoveNet and BlazePose 3D models which are faster and even more accurate.</p>
Gaming	<p>The Beat-Pose App</p> <p>An interactive game created by a member of the TensorFlow.js community where the user uses hand gestures to play the game to smash blocks that fly towards you in time with music.</p>

Arts	<p>Scroobly</p> <p>Allows you to draw any character you wish and then bring it to life using your own body. Essentially a novel form of motion capture powered by TensorFlow.js to drive the skeleton of any given character in real-time for 2D animation.</p> <p><u>Kalidoface: VTubers</u></p> <p>This demo allows you to control a full 3D character model instead of 2D in a realistic manner. VTubers are growing in popularity which allows people to join a meeting or live stream with a different digital avatar that they identify with.</p>
Emerging Web Technologies	<p>Web XR: Visualize Pose through time</p> <p>Web XR allows you to perform mixed reality right from your browser on a smartphone. In this example, the creator wanted to visualize pose over time in this very futuristic demo to see how the martial arts person changes his form, over time, in a novel way.</p>
Sports	<p>Pose Estimation for Sport Routines</p> <p>TeamSportz, a company based in the UK, has developed an application that enables users to define custom routines for any sport or activity to create interactive and gamified workouts. The app</p>

	<p>makes sports practice a lot more social and also allows for real-time feedback and tips for improvement.</p>
Custom Hardware	<ul style="list-style-type: none"> ● Automated Video Assistant: The first-ever Kickstarter powered by TensorFlow.js was a custom piece of hardware that can be used with a smartphone to capture sports action as they move around the field autonomously. ● Touchless Interfaces for Gaming and Kiosks: A number of creative agencies have created digital kiosks such as touchless photo booths and food ordering interfaces. Human-Computer Interaction has much potential to be explored further using TensorFlow.js. ● Car Gesture Recognition: A car gesture recognition system that runs in React Native and is powered by TensorFlow.js. Users can control their music and phone calls without looking at the screen using hand gestures. ● Home Automation: Home automation can be achieved by hooking TensorFlow.js up to a device such as Raspberry Pi. Here, NodeRed by IBM is used to deploy a TensorFlow.js model on the Raspberry Pi that can recognize the car and number plate and open the garage door automatically when the correct car arrives.

Miscellaneous

- **Enjoying the show:**
Presenters are using TensorFlow.js to understand their audiences better while preserving their privacy. In this example, as people react to the content of the presenter, their anonymous emotions are aggregated and displayed on the pie chart in real-time. This allows the presenter to see how engaged the audience might be in digital meetings, leading to an enhanced digital experience.
- **AR Sudoku Solver:**
Solves Sodoku puzzles in Augmented Reality, using computer vision and ML powered by TensorFlow.js.
- **Satellite Imagery Analysis:**
A satellite imaging company is using TensorFlow.js to recognize and even count relevant objects over large geographic regions.
- **Emulate a 3D Screen:**
An application that uses face tracking to create a motion-based parallax effect. This effect provides an illusion of having a 3D screen when used with 3D models displayed with Three.js in the browser.
- **Face Masks:**
The face mesh model can be used with Three.js to create a face mask from any image and the user can see it augmented in real-time live in the browser.
- **Tone Transfer:**
Magenta's Tone Transfer model turns your voice into an instrument of your choice. It runs entirely in the browser through TensorFlow.js and now anyone can play an

instrument by just using their voice and this generative model. Try it out!

Note: To further explore TensorFlow.js demos, search with the hashtag **#MadeWithTFJS** on Twitter or LinkedIn.

you have explored the following concepts:

- 1. What ML is and how it works**
- 2. What TensorFlow.js is and its advantages over server-side ML**
- 3. How and when to use pre-made TensorFlow.js models**
- 4. How to load and send data to raw pre-made models using Tensors**
- 5. When and how to write your own models:**
 - a. Single and multi-input linear regression**
 - b. Non-Linear regression**
 - c. Classification**
 - d. Multi-Layer Perceptrons (MLP) - Deep Neural Networks (DNNs)**
 - e. Convolutional Neural Networks (CNNs)**

6. How to implement Transfer Learning and make your very own teachable machine

7. How to use models from Python and convert to the TensorFlow.js format

8. How to retrain Python models via Colabs prior to conversion

9. How to use Natural Language models for comment spam detection in a real web application

10. How real businesses and people are using Web ML in their own products and services

In the next unit, you are introduced to a few new areas of ML that might be of interest for you to explore in future courses.

Autoencoders

Autoencoders are built upon multi-layer perceptrons. They can be used for many things such as transforming a grainy input image into one that's smoother as shown by the example images on Figure 7.2.1.

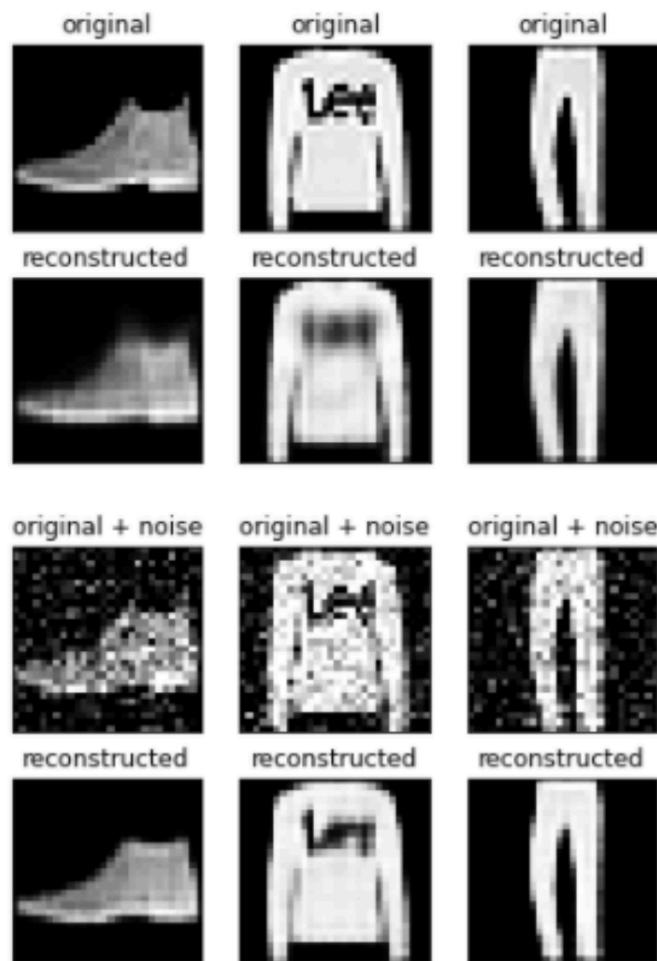


Figure 7.2.1 - Autoencoder Output

Autoencoders have two sub-networks. The first half of the network acts as an encoder and the second half of the network acts as a decoder. They are connected to each other through a middle layer in the network (see Figure 7.2.2).

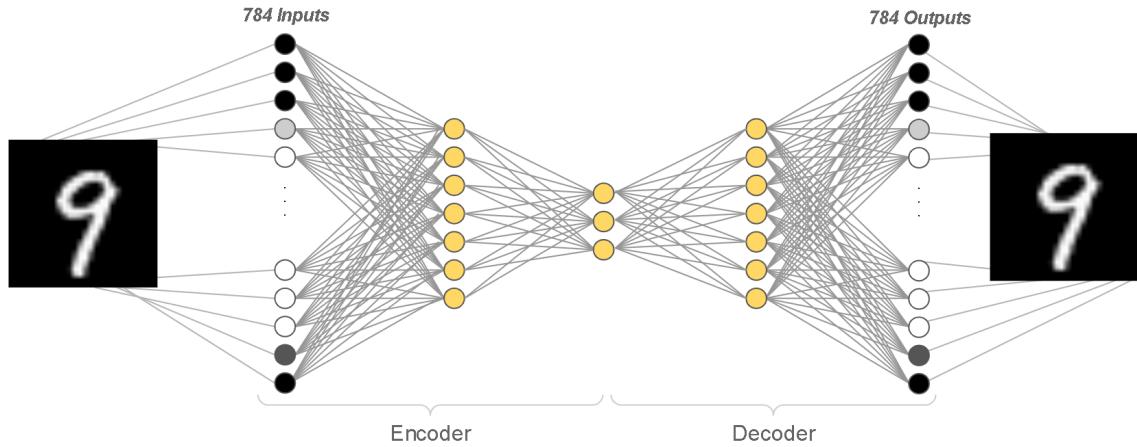


Figure 7.2.2 - Autoencoder Output

Observe that the inputs are forced to go through a middle layer that has far fewer neurons than the number of inputs. This forces the network to use the neurons to encode meaningful features from the input image. The network then decodes these features to re-create something meaningful as the output. Essentially, the network takes an input, performs a form of compression on it, and tries to recreate an output similar to the input.

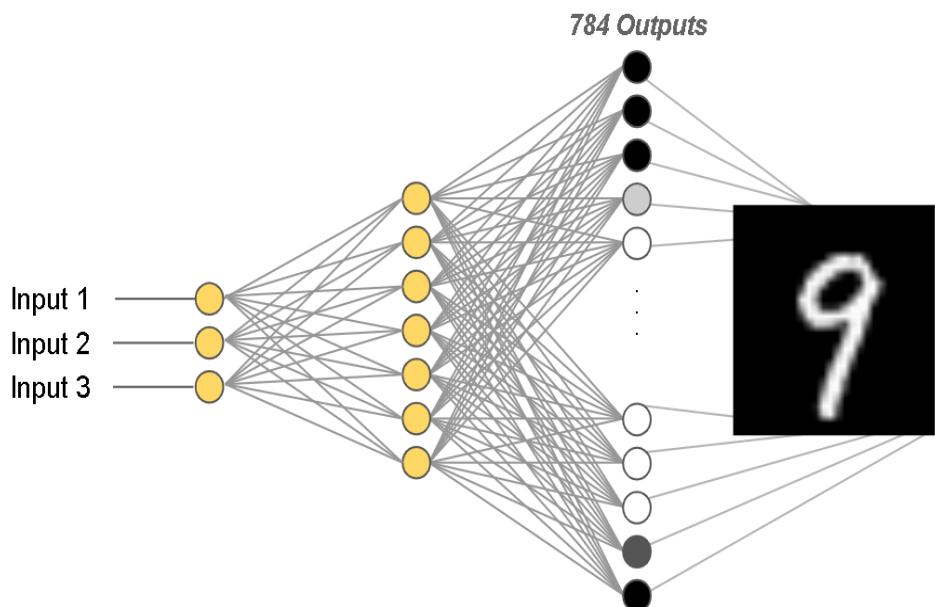


Figure 7.2.3 - Autoencoder Decoder

In fact, once the autoencoder is trained, you can then chop the network in the middle (see Figure 7.2.3) and generate new output images just by changing the inputs to the three neurons that were in the middle of the original full network. ML engineers refer to this compressed representation as the ‘learned latent space’, which you can then step through with different numbers and produce different outputs.



Here's an example of how this works: [Generating New Faces](#). This is a project that uses an autoencoder in TensorFlow.js that is able to generate new faces in real-time, live in the browser, just by tweaking two values. The act of changing these input numbers generates new outputs from the model that you can then visualize, and explore the latent space it had learned.

Generative Adversarial Networks (GANs)

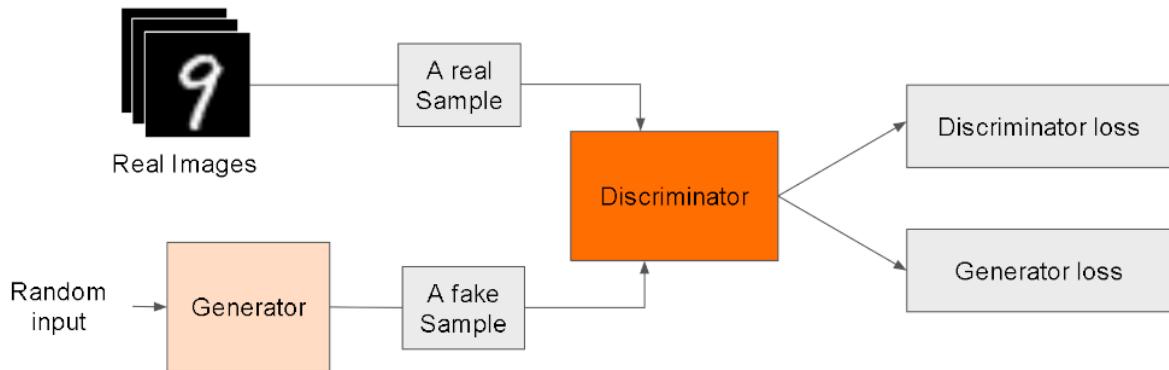


Figure 7.2.4 - Generative Adversarial Network

1. A Generative Adversarial Network consists of a generator and discriminator. The generator produces outputs such that the discriminator can not tell if they are the real thing or not.
2. Essentially, you train a regular classification model for the discriminator (such as a CNN), and then create a second network, called the generator, to generate outputs in order to try and fool the CNN classifier.
3. This generator network takes some random input values and uses deconvolution to create new images; this is the opposite of convolution. This technique allows image data to be produced from features instead of finding features from images.
4. Once the model is trained, the discriminator cannot reliably identify the differences between the generated inputs and real inputs. This feature can be used in many creative ways, to create sounds, videos, generate images from text descriptions, and much more.

Visit the following links to see some of the applications of GANs.

youtube.com/watch?v=4Up42g1q-bQ

developers.google.com/machine-learning/gan/applications

ecurrent Neural Networks (RNNs)

RNNs are similar to multi-layer perceptrons but with a time aspect.

Figure 7.2.5 shows a regular multi-layer perceptron

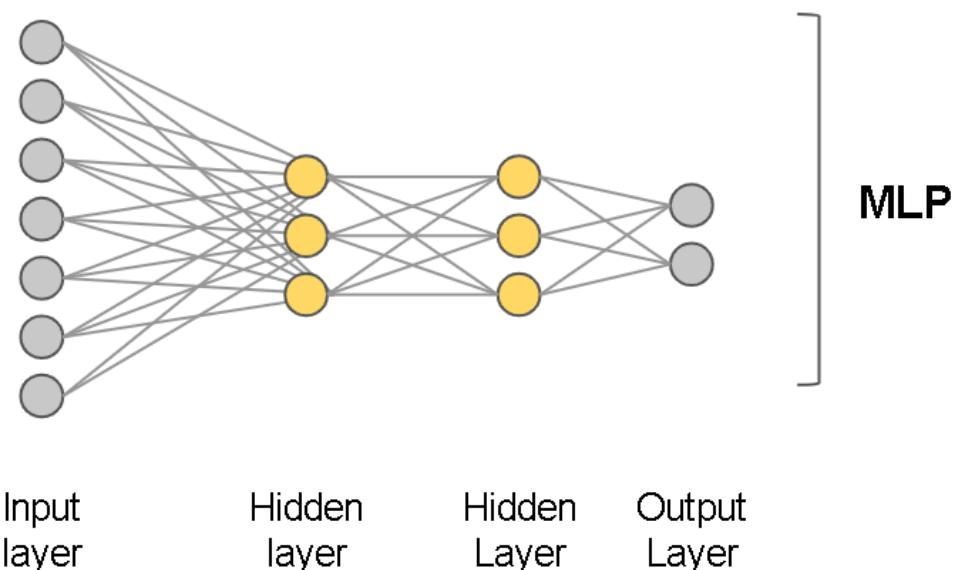


Figure 7.2.5. Recurrent Neural Network

Figure 7.2.6 shows a simplified version of Figure 7.2.5.

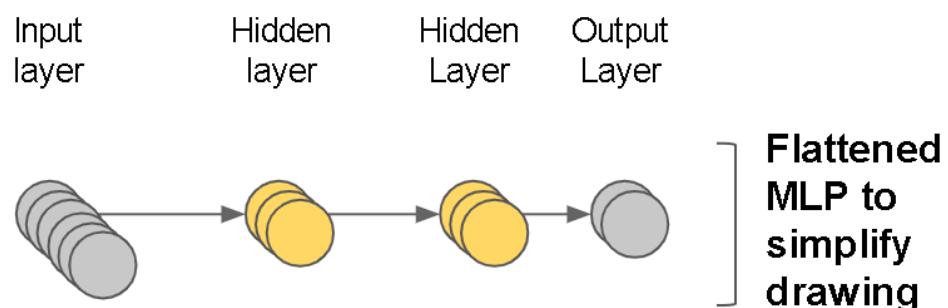


Figure 7.2.6. RNN - Simplified Image

Figure 7.2.7 shows the updated structure of an RNN where the hidden layers' outputs are allowed to loop back to become their inputs as well.

You may also hear of Long Short-Term Memory (LSTM) networks, which are a special kind of RNN.

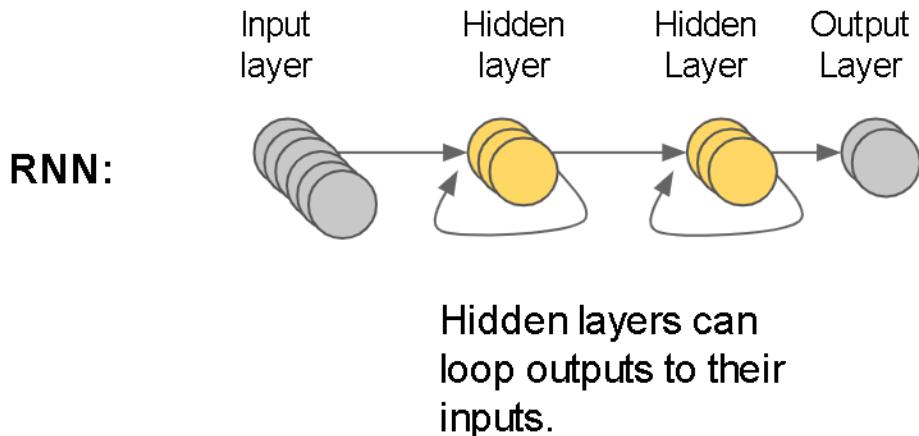


Figure 7.2.7. Hidden Layer Loops

In this case, connections between neurons can form a sequence over many time steps allowing them to capture temporal relationships better than other networks. Often you will see this sort of network be used for solving things like time series data, speech recognition, or even music generation which is time-based to name just a few examples.

Transformer Networks

Transformer networks can be faster than RNNs and can handle sequential tasks pretty well. These include tasks such as time series prediction, text summarization, and translation. In fact, one of their most well-known use cases is for Natural Language Processing. The key difference here is that it uses a breakthrough in research known as “attention” to understand at each given step through a given sequence, what other parts of the sequence are most important to focus on when learning about that data.

Note: This course focused on supervised learning, which currently is the most popular form of ML. However, techniques in unsupervised and reinforcement learning are rapidly evolving, providing you with a wide range of options to explore for further studies.

Other types of learning

This course focused on supervised learning, which currently is the most popular form of ML. However, techniques in unsupervised and reinforcement learning are rapidly advancing, providing you with an even wider range of options to explore for further studies should you wish to do so. Each one of those areas could have a course dedicated to the various popular techniques used to learn from data in those situations.

recurrent Neural Networks (RNNs)

RNNs are similar to multi-layer perceptrons but with a time aspect.

Figure 7.2.5 shows a regular multi-layer perceptron

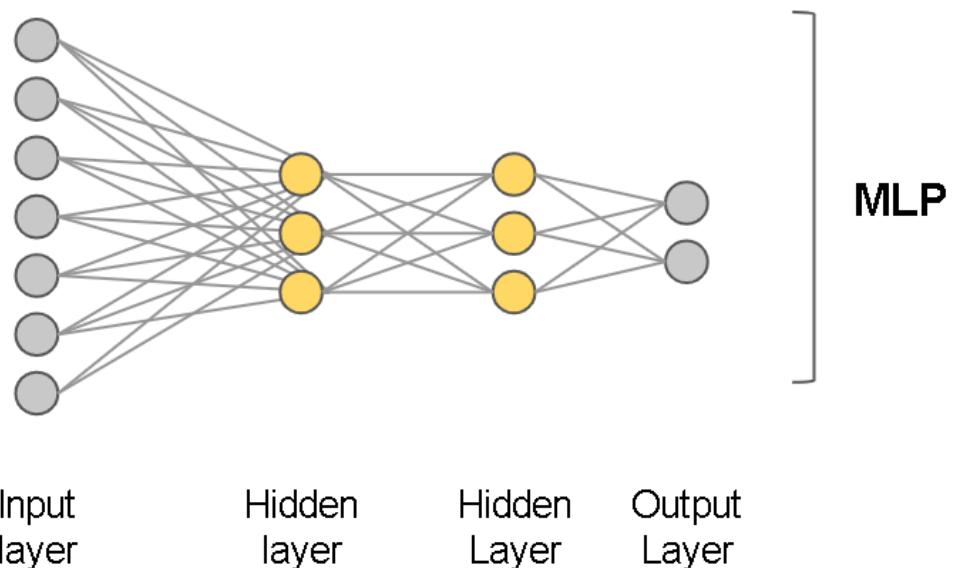


Figure 7.2.5. Recurrent Neural Network

Figure 7.2.6 shows a simplified version of Figure 7.2.5.

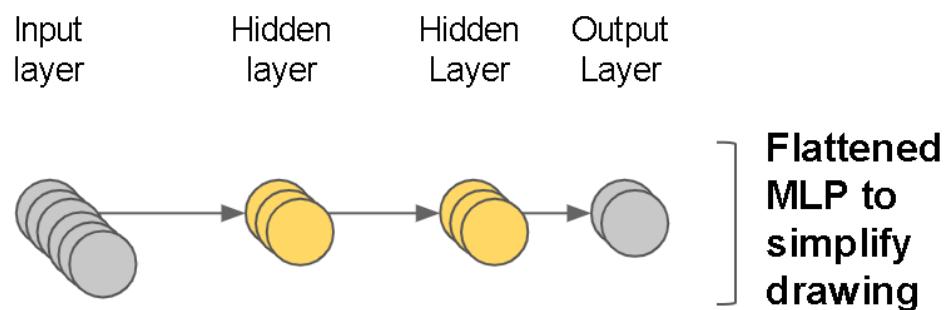


Figure 7.2.6. RNN - Simplified Image

Figure 7.2.7 shows the updated structure of an RNN where the hidden layers' outputs are allowed to loop back to become their inputs as well.

You may also hear of Long Short-Term Memory (LSTM) networks, which are a special kind of RNN.

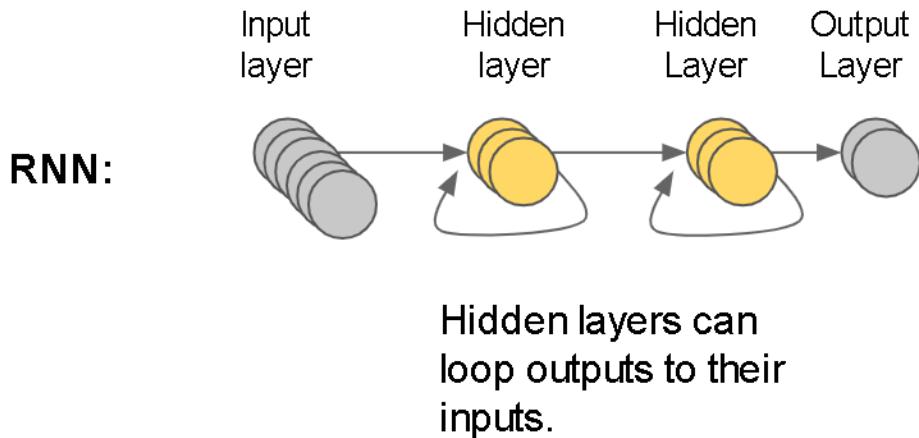


Figure 7.2.7. Hidden Layer Loops

In this case, connections between neurons can form a sequence over many time steps allowing them to capture temporal relationships better than other networks. Often you will see this sort of network be used for solving things like time series data, speech recognition, or even music generation which is time-based to name just a few examples.

Transformer Networks

Transformer networks can be faster than RNNs and can handle sequential tasks pretty well. These include tasks such as time series prediction, text summarization, and translation. In fact, one of their most well-known use cases is for Natural Language Processing. The key difference here is that it uses a breakthrough in research known as “attention” to understand at each given step through a given sequence, what other parts of the sequence are most important to focus on when learning about that

Note: This course focused on supervised learning, which currently is the most popular form of ML. However, techniques in unsupervised and reinforcement learning are rapidly evolving, providing you with a wide range of options to explore for further studies.

Other types of learning

This course focused on supervised learning, which currently is the most popular form of ML. However, techniques in unsupervised and reinforcement learning are rapidly advancing, providing you with an even wider range of options to explore for further studies should you wish to do so. Each one of those areas could have a course dedicated to the various popular techniques used to learn from data in those situations.