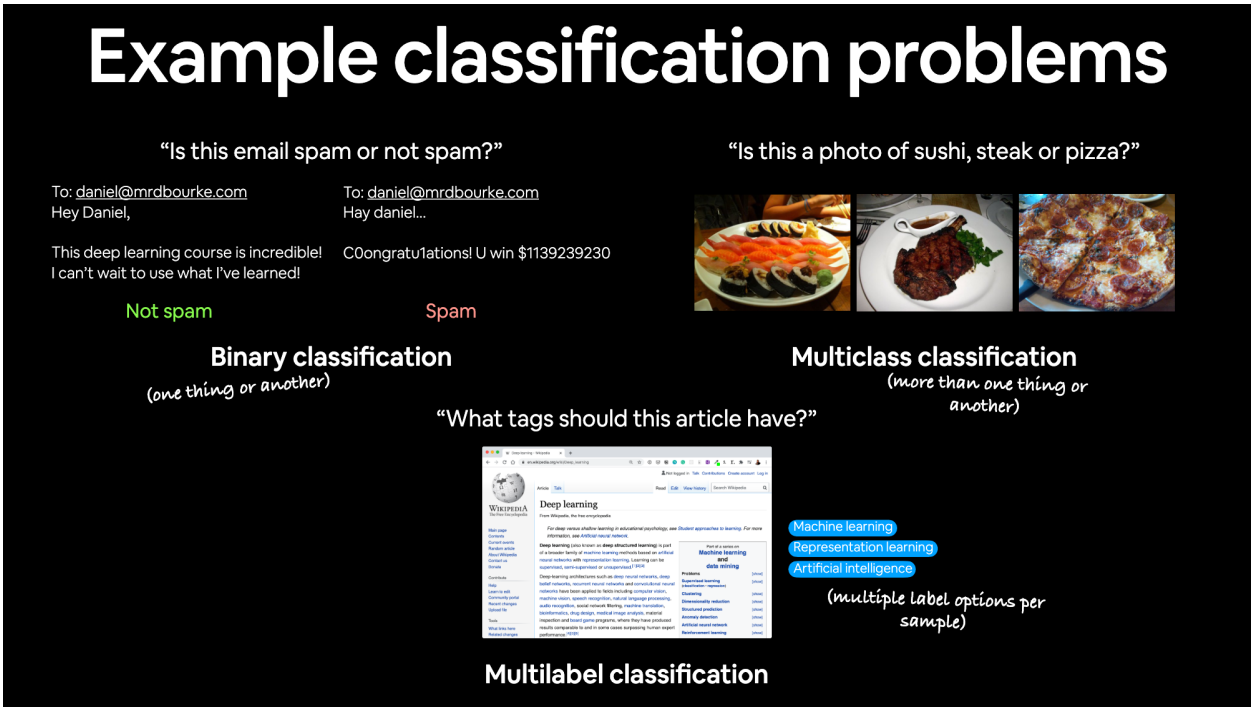
Neural Network Classification with TensorFlow

Table of contents:

Classification Overview

What is classification problem ?



Binary classification - one thing or another

Multi-class classification - more than one thing or another

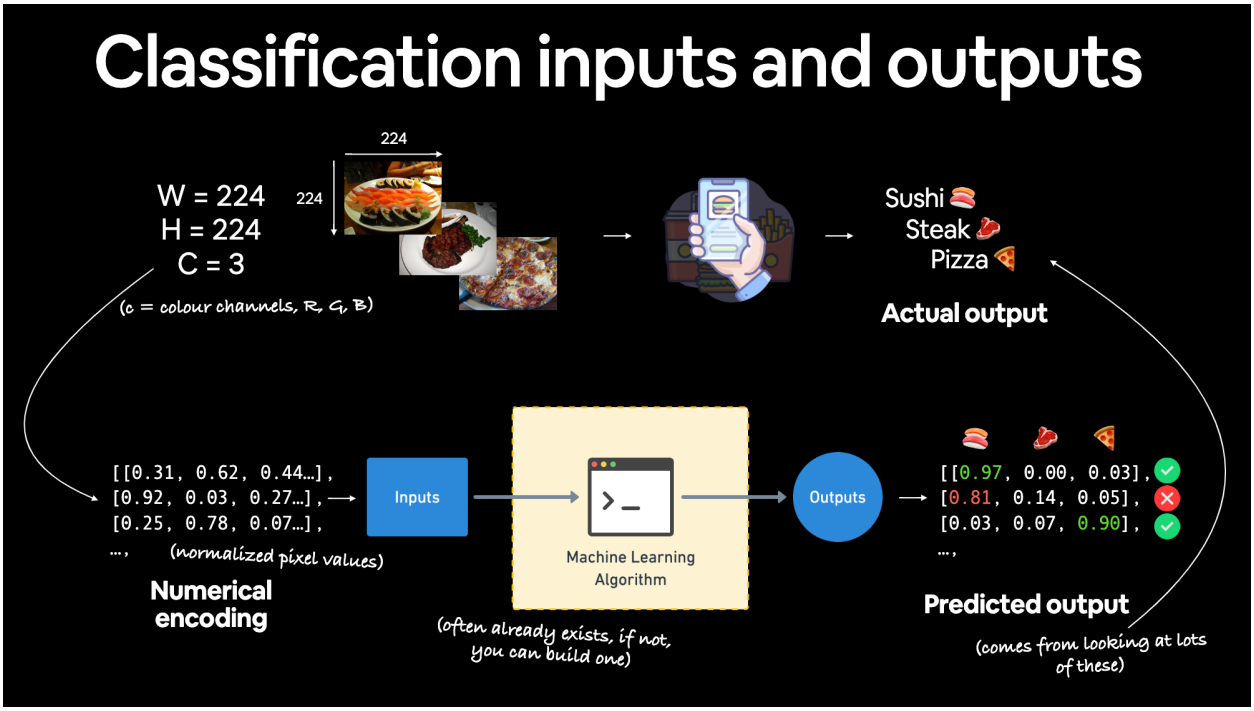
Multi-label classifcation - multiple label options per sample

> 💡 Wikipedia info:
>
> In statistics, **classification** is the problem of identifying which of a set of categories (sub-populations) an observation (or observations) belongs to.
>
> Examples are assigning a given email to the "spam" or "non-spam" class, and assigning a diagnosis to a given patient based on observed characteristics of the patient (sex, blood pressure, presence or absence of certain symptoms, etc.).

For example, you might want to:

- Predict whether or not someone has heart disease based on their health parameters. This is called **binary classification** since there are only two options.
- Decide whether a photo of is of food, a person or a dog. This is called **multi-class classification** since there are more than two options.
- Predict what categories should be assigned to a Wikipedia article. This is called **multi-label classification** since a single article could have more than one category assigned.

Classification inputs and outputs

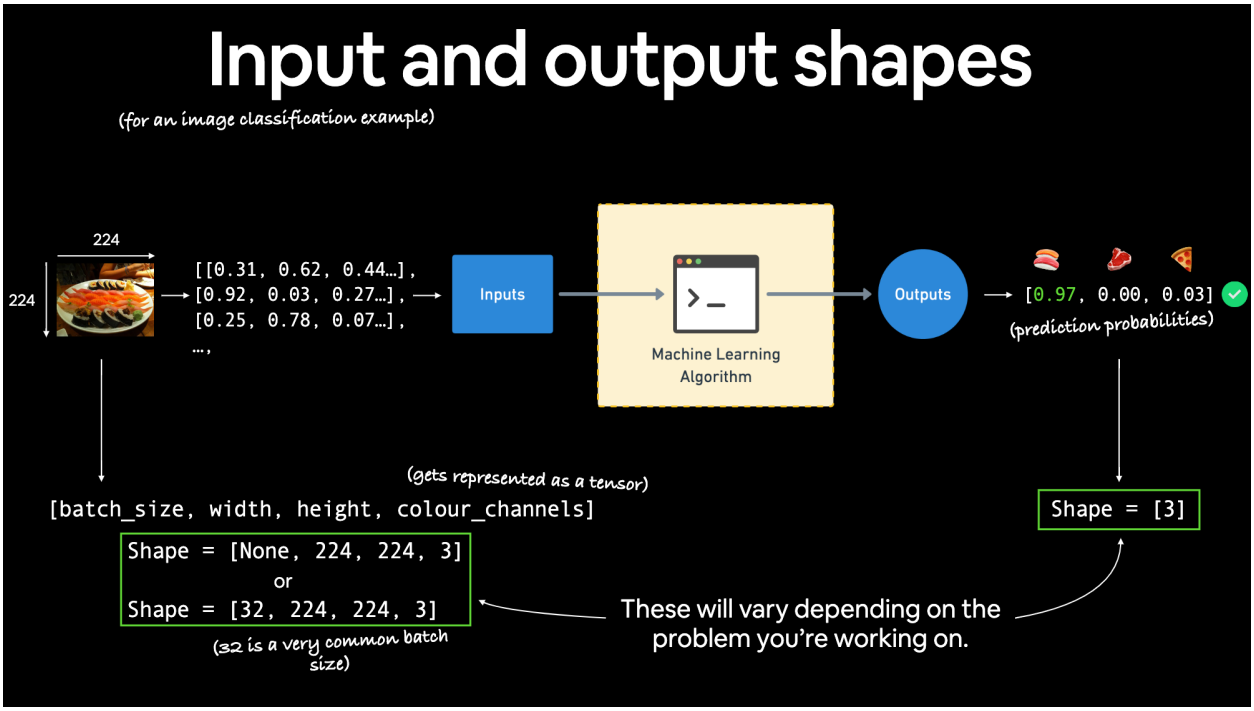


Example of multi-class classification problem: We take a photo of food and classify it as sushi, steak, or pizza.

Our inputs would be images of the food. Our ideal outputs would be a label. For example, if we took a photo of pizza, the output label would be pizza.

Our Machine Learning algorithm won't be able to understand the images itself. We need to encode them into a tensor. So, we have images, we should turn them into numbers before we can pass them to our Machine Learning algorithm. The Machine Learning algorithm will create prediction outputs.

Input and Output Shapes



🤔 One of the most common issues you'll run into when building neural networks is shape mismatches.

Input Shape:

[batch_size, width, height, colour_channels] - dimensions of a tensor.

Output Shape:

[3] - shape will be 3, since we have multi-class classification.

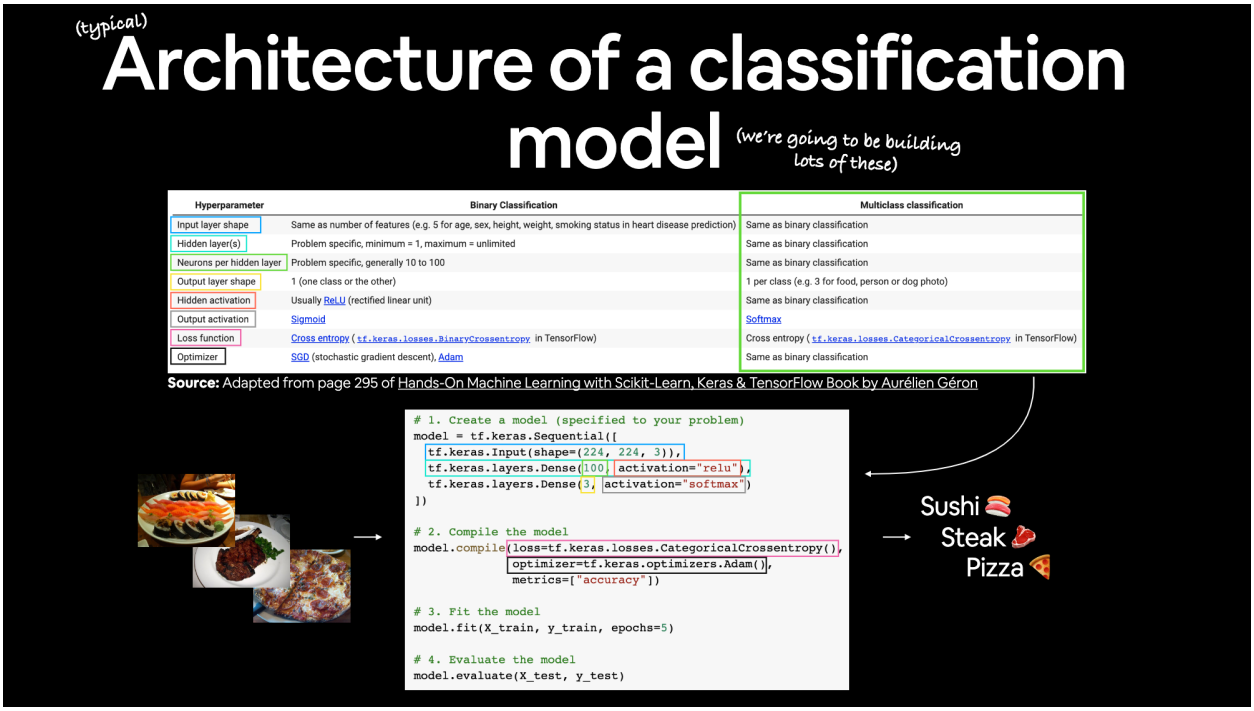
This may change on the problem we are working on.

Typical Architecture of a classification model

The architecture of a classification neural network can widely vary depending on the problem you're working on.

However, there are some fundamentals all deep neural networks contain:

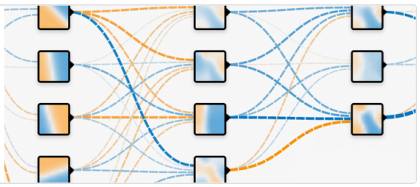
- An input layer.
- Some hidden layers.
- An output layer.



Hyperparameter	Binary Classification	Multi-class Classification
Input Layer Shape	Same as number of features (e.g. 5 for age, sex, height, weight, smoking status in heart disease prediction)	Same as binary classification
Hidden Layer(s)	Problem specific, minimum = 1, maximum = unlimited	Same as binary classification
Neurons per Hidden Layer	Problem specific, generally 10 to 100	Same as binary classification
Output Layer Shape	1 (one class or the other)	1 per class (e.g 3 for food, person, dog photo)
Hidden Activation	Usually ReLu (Rectified Linear Unit)	Same as binary classification
Output Activation	Sigmoid	Softmax
Loss Function	Cross Entropy	Cross Entropy
Optimizer	SGD, Adam	Same as binary classification

Resources:

- TensorFlow Playground



Tensorflow - Neural Network Playground

It's a technique for building a computer program that learns from data. It is based very loosely on how we think the human brain works. First, a collection of software "neurons" are created and connected together, allowing them to send messages to each other.

https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.61303&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false

- Andrej Karpathy's Neural Network Recipe

A Recipe for Training Neural Networks

Some few weeks ago I posted a tweet on "the most common neural net mistakes", listing a few common gotchas related to training neural nets. The tweet got quite a bit more engagement than I anticipated (including a webinar :)).

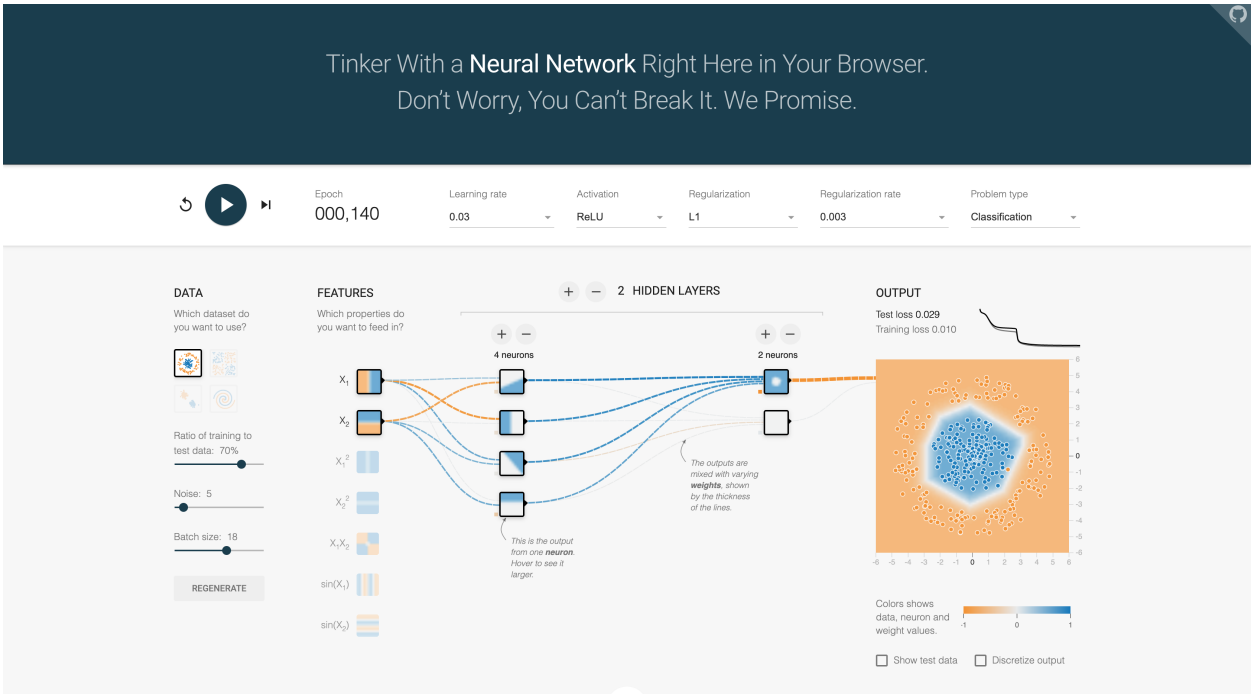
http://karpathy.github.io/2019/04/25/recipe/

- Evaluating Machine Learning Classification models

Evaluating a Classification Model

I am Ritchie Ng, a machine learning engineer specializing in deep learning and computer vision. Check out my code guides and keep ritching for the skies!

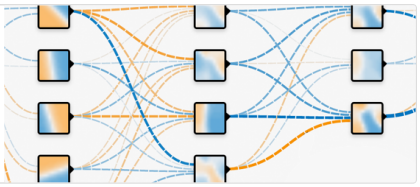
https://www.ritchieng.com/machine-learning-evaluate-classification-model/



Tensorflow - Neural Network Playground

It's a technique for building a computer program that learns from data. It is based very loosely on how we think the human brain works. First, a collection of software "neurons" are created and connected together, allowing them to send messages to each other.

https://playground.tensorflow.org/#activation=relu&regularization=L1&batchSize=18&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0.003&noise=5&networkShape=4,2&seed=0.48696&showTestData=false&discretize=false&percTrainData=70&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false



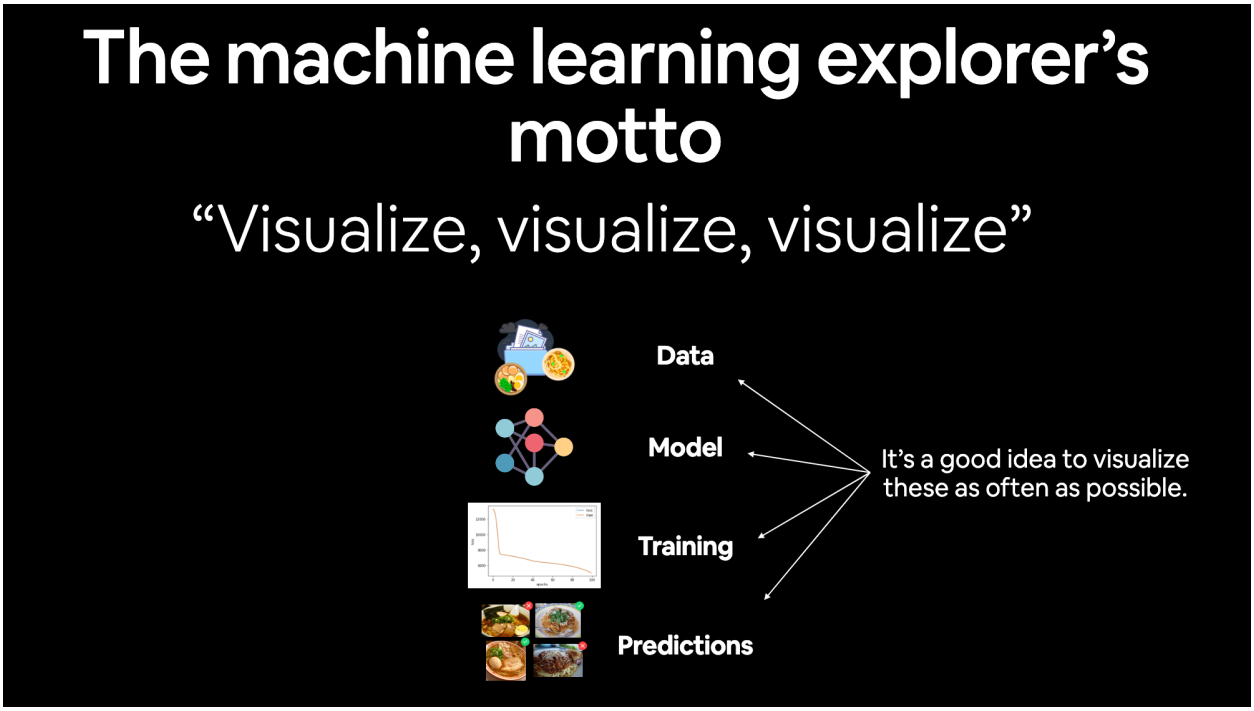
Binary Classification

Creating Classification Data

💡 It's a common practice to get you and model you build working on a toy (or simple) dataset before moving to your actual problem. Treat it as a rehersal experiment before the actual experiment(s).

Create the data

```
from sklearn.datasets import make_circles

# Make 1000 examples
n_samples = 1000

# Create circles
X, y = make_circles(n_samples, noise=0.03, random_state=42)

# Check out features and labels
X, y
```

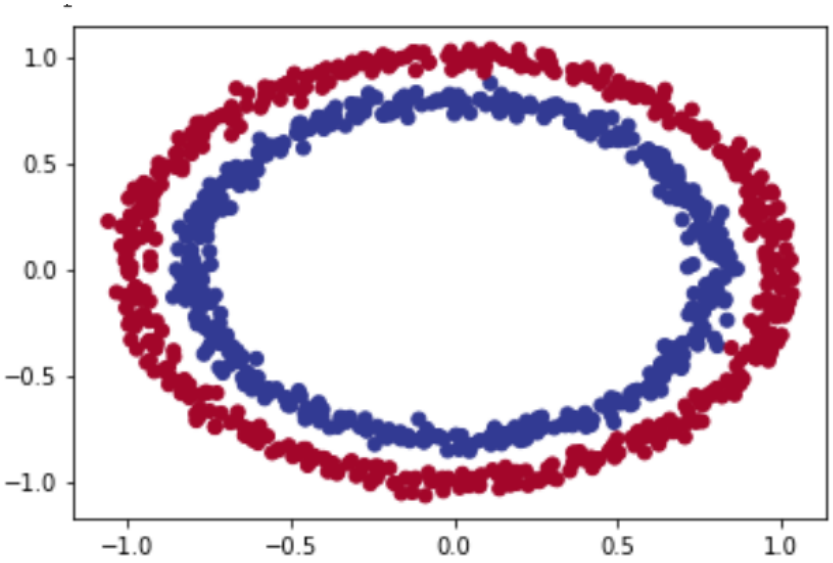


Visualize it

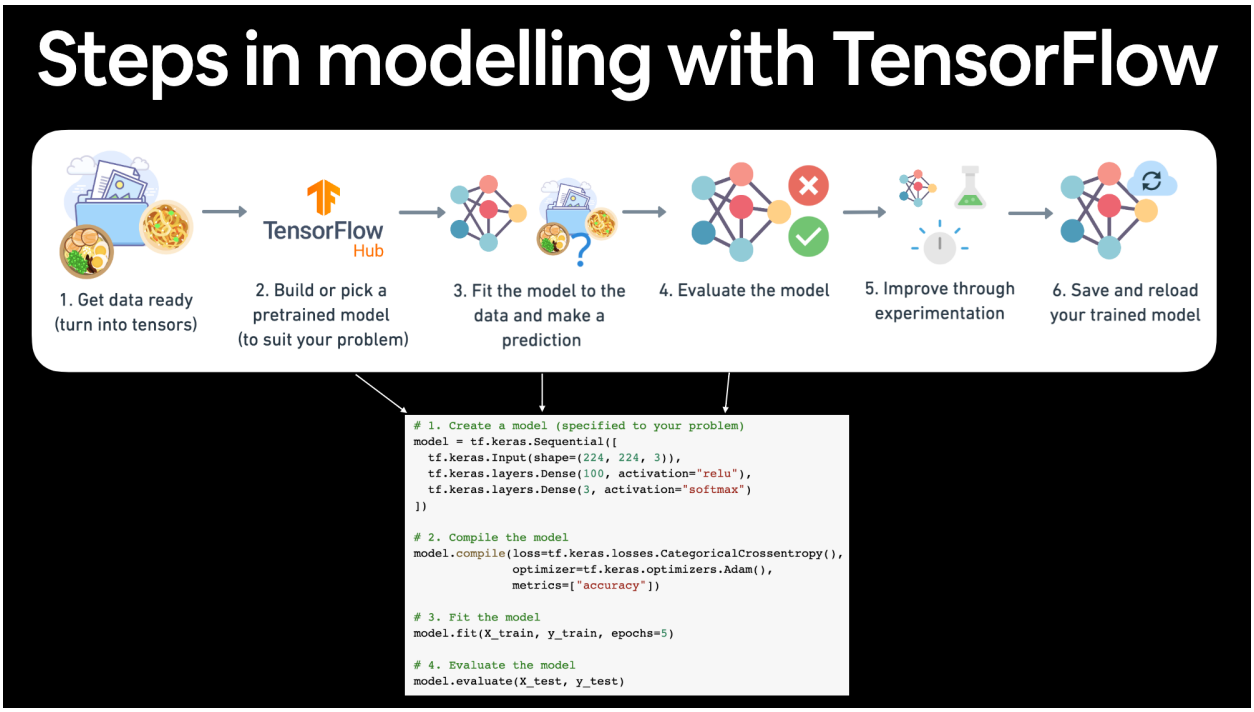
💡 One important step of starting any kind of machine learning project is to become one with the data. And one of the best ways to do this is to visualize the data you're working with as much as possible. The data explorer's motto is "visualize, visualize, visualize".

```
import pandas as pd

circles = pd.DataFrame({"X0":X[:,0], "X1":X[:,1], "label": y})
circles
```

```
# Visualize with a plot
import matplotlib.pyplot as plt

plt.scatter(X[:,0], X[:,1],c=y, cmap=plt.cm.RdYlBu)
```

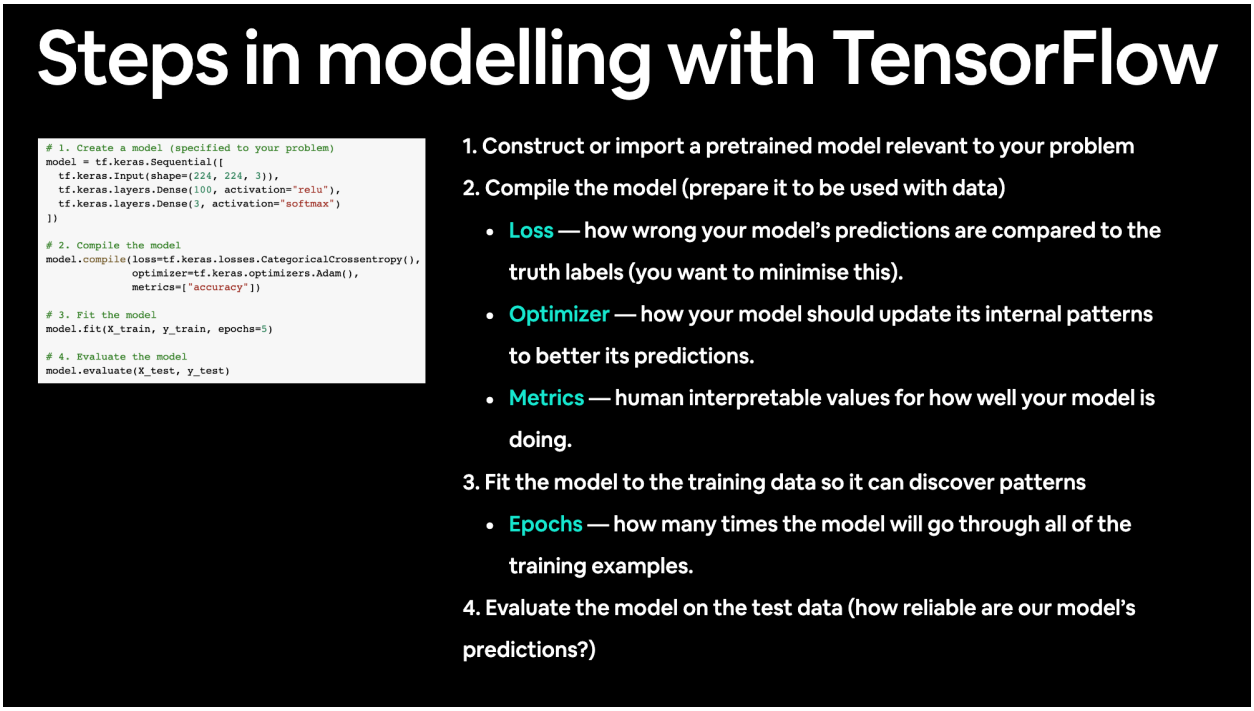


Steps in Modelling with TensorFlow

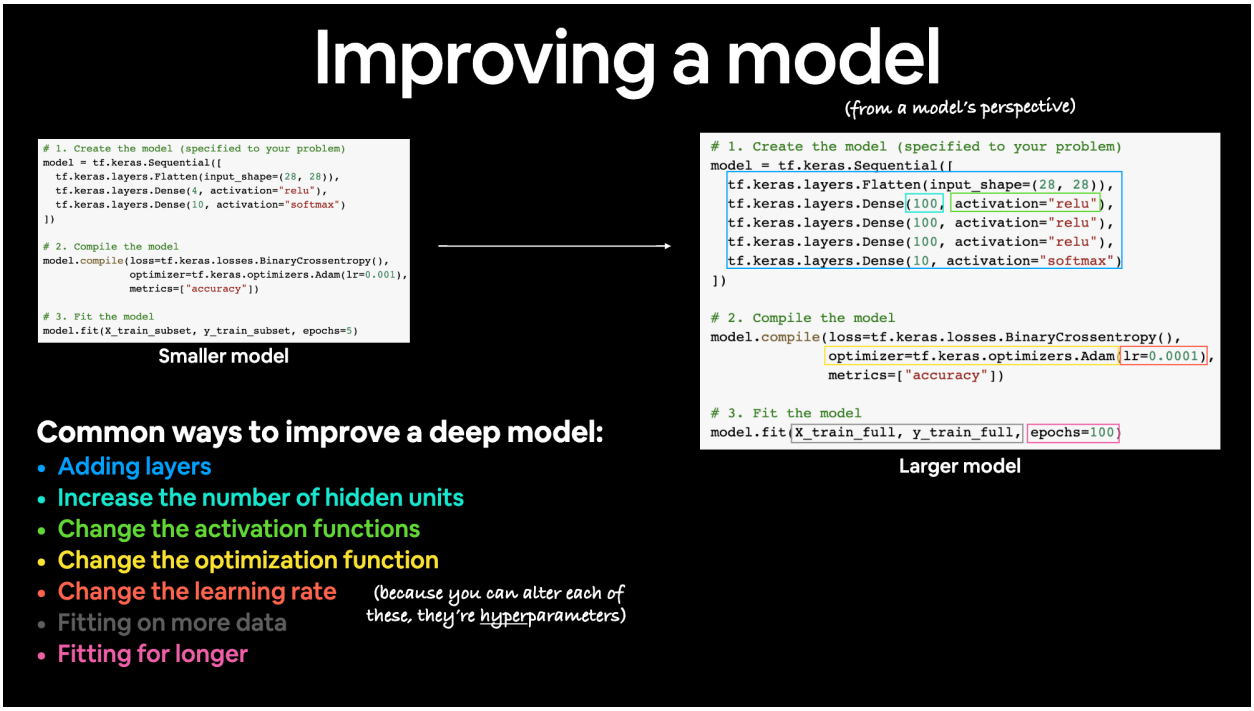


In TensorFlow, there are typically 3 fundamental steps to creating and training a model.

1. **Creating a model** - piece together the layers of a neural network yourself (using the functional or sequential API) or import a previously built model (known as transfer learning).
2. **Compiling a model** - defining how a model's performance should be measured (loss/metrics) as well as defining how it should improve (optimizer).
3. **Fitting a model** - letting the model try to find patterns in the data (how does `X` get to `y`).



Improving the Model



1. **Creating a model** - we might want to add more layers, increase the number of hidden units (also called neurons) within each layer, change the activation functions of each layer.
2. **Compiling a model** - we might want to choose a different optimization function (such as the Adam optimizer, which is usually pretty good for many problems) or perhaps change the learning rate of the optimization function.
3. **Fitting a model** - we could fit a model for more epochs (leave it training for longer).

Note:
There are many different ways to potentially improve a neural network.

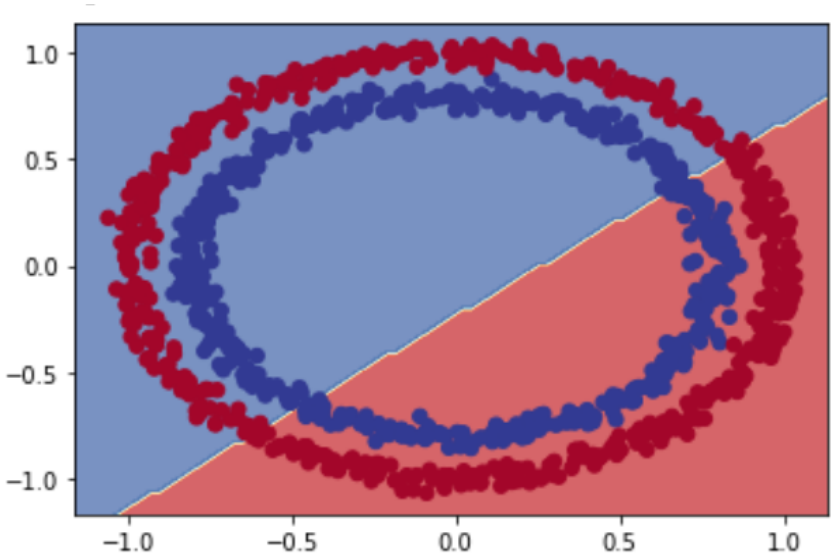
Some of the most common include: increasing the number of layers (making the network deeper), increasing the number of hidden units (making the network wider) and changing the learning rate.

Because these values are all human-changeable, they're referred to as hyperparameters) and the practice of trying to find the best hyperparameters is referred to as hyperparameter tuning.

Visualize Models Predictions

```
import numpy as np

def plot_decision_boundary(model, X, y):
  """
  Plots the decision boundary created by
  a model prediction on X.
  """
  # Define the axis boundaries of the plot and create a meshgrid
  x_min, x_max = X[:, 0].min() - 0.1, X[:, 0].max() + 0.1
```

```
y_min, y_max = X[:, 1].min() - 0.1, X[:, 1].max() + 0.1
# Create a meshgrid
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 100),
                     np.linspace(y_min, y_max, 100))
# Create X values (we're going to make predictions on these)
x_in = np.c_[xx.ravel(), yy.ravel()] # stack 2D arrays together

# Make predictions
y_pred = model.predict(x_in)

# Check for multi-class classification
if len(y_pred[0]) > 1:
  print("Multi-class classification")
  # We have to reshape our prediction to get them ready for plotting
  y_pred = np.argmax(y_pred, axis=1).reshape(xx.shape)
else:
  print("Binary classification")
  y_pred = np.round(y_pred).reshape(xx.shape)

# Plot the decision boundary
plt.contourf(xx, yy, y_pred, cmap=plt.cm.RdYlBu, alpha=0.7)
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.RdYlBu)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
```



Resources:

CS231n Convolutional Neural Networks for Visual Recognition

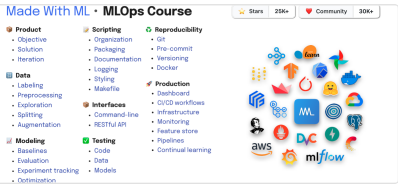
Table of Contents: In this section we'll walk through a complete implementation of a toy Neural Network in 2 dimensions. We'll first implement a simple linear classifier and then extend the code to a 2-layer Neural Network. As we'll see, this extension is surprisingly simple and very few changes are necessary.

https://cs231n.github.io/neural-networks-case-study/

MadeWithML/08_Neural_Networks.ipynb at main · GokuMohandas/MadeWithML

Learn how to responsibly deliver value with ML. Contribute to GokuMohandas/MadeWithML development by creating an account on GitHub.

https://github.com/GokuMohandas/MadeWithML/blob/main/notebooks/08_Neural_Networks.ipynb

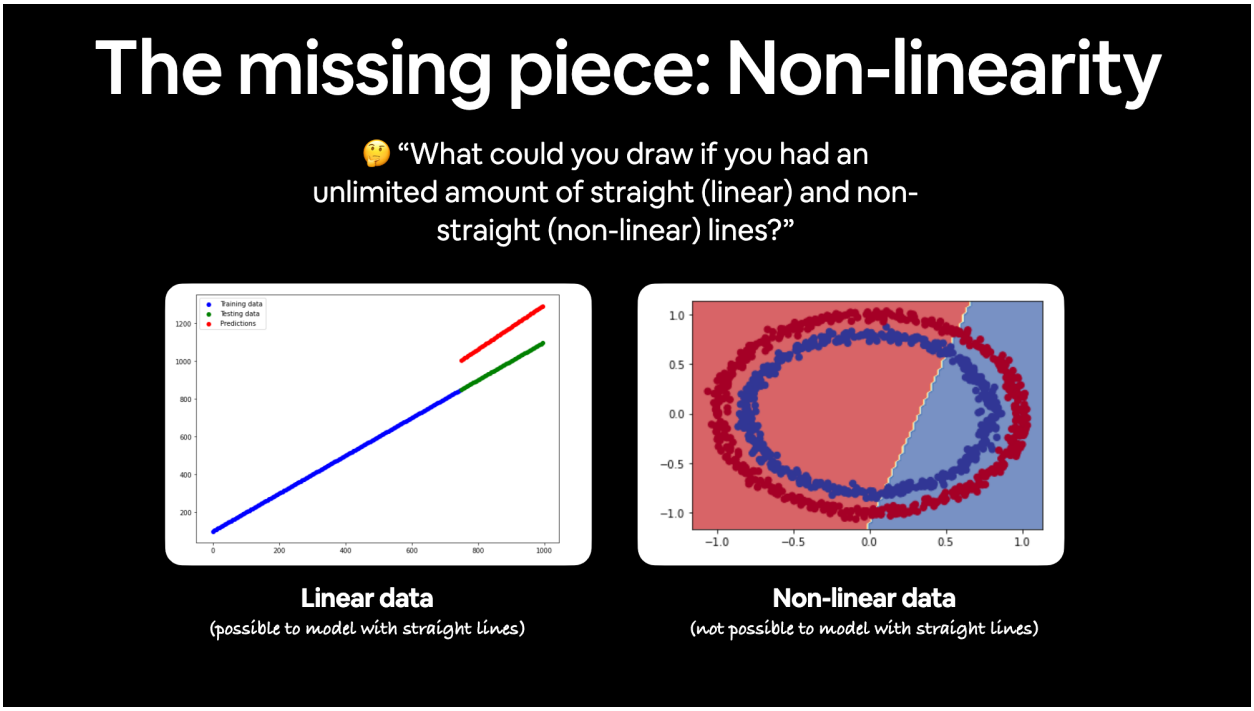


Non-linearity

We have seen that a neural network can model straight lines.

What about non-straight (non-linear) lines? 🤔

If we're going to model our classification data (the red and clue circles), we're going to need some non-linear lines.

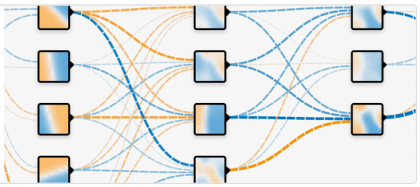


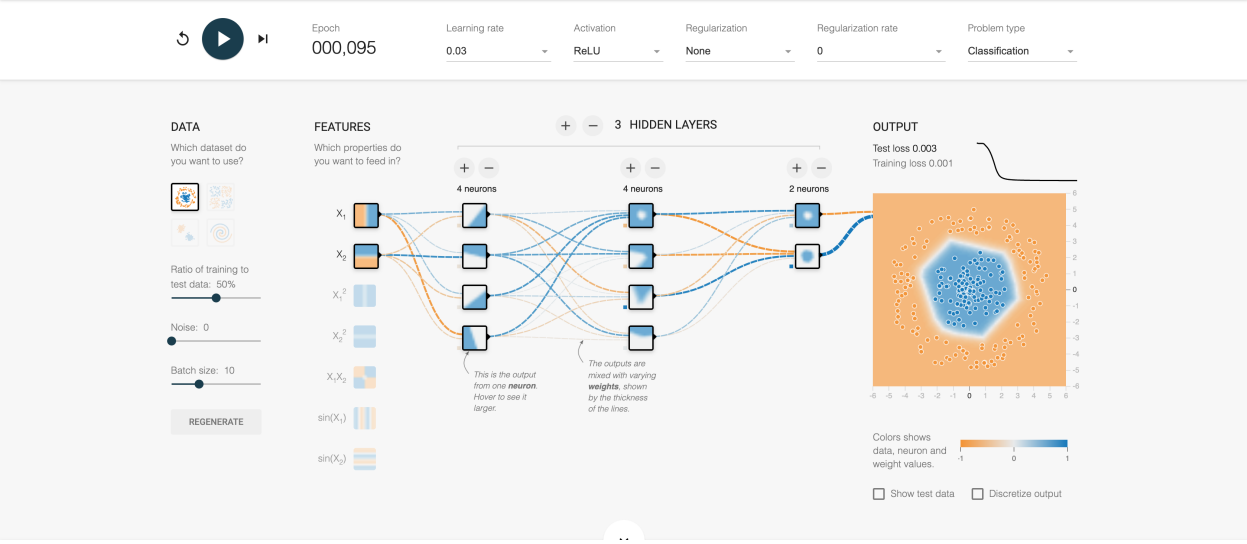
Resources:

Tensorflow - Neural Network Playground

It's a technique for building a computer program that learns from data. It is based very loosely on how we think the human brain works. First, a collection of software "neurons" are created and connected together, allowing them to send messages to each other.

https://playground.tensorflow.org/#activation=linear&batchSize=1&dataset=circle&regDataset=reg-plane&learningRate=0.01&regularizationRate=0&noise=0&networkShape=1&seed=0.09561&showTestData=false&discretize=false&percTrainData=70&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false&regularizationRate_hide=true&discretize_hide=true&regularization_hide=true&dataset_hide=true&noise_hide=true&batchSize_hide=true





We have created the above model

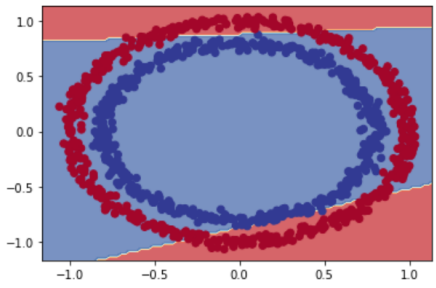
```
# Set random seed
tf.random.set_seed(42)

# 1. Create the model
model_6 = tf.keras.Sequential([
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(1)
])

# 2. Compile the model
model_6.compile(loss=tf.keras.losses.binary_crossentropy,
                optimizer=tf.keras.optimizers.Adam(lr=0.03),
                metrics=['accuracy'])

# 3. Fit the model
model_6.fit(X, y, epochs=100)
```

```
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 89/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 90/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 91/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 92/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 93/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 94/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 95/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 96/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 97/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 98/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 99/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
Epoch 100/100
32/32 [==============================] - 0s 2ms/step - loss: 7.7125 - accuracy: 0.5000
<keras.callbacks.History at 0x7f74455a39d0>
```

We have slightly increased the accuracy of our model. However, the value of an accuracy metrics is fixed at 50%. Something is wrong.

```
# Display the predictions
plot_decision_boundary(model_6, X,y)
```

```
Binary classification
```

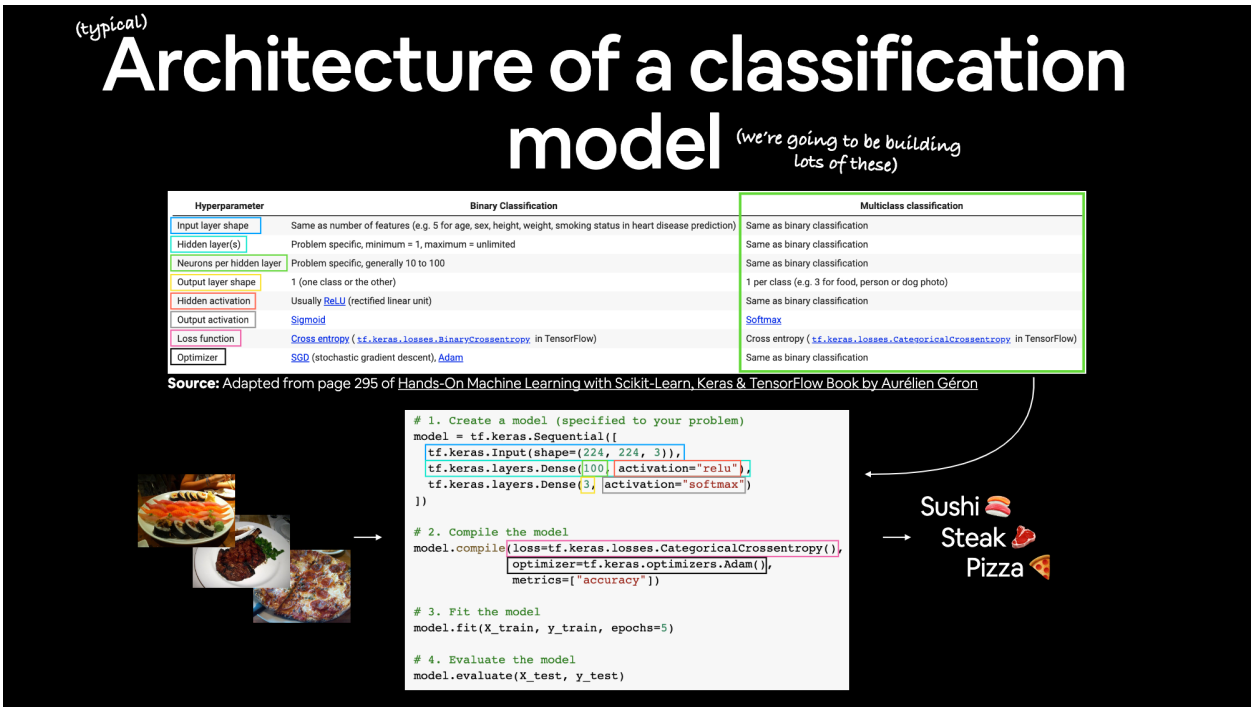


✓ 0 сек. выполнено в 03:0

It seems like our model is the same as the one in the TensorFlow Playground but model it's still drawing straight lines... Ideally, the yellow lines go on the inside of the red circle and the blue circle.

This time we'll change the activation function on our output layer too.

Refer to the architecture of a classification model:



For the binary classification model, we should use Sigmoid function as an Output Activation function.

Resource:

Activation Functions | Fundamentals Of Deep Learning

Activation function is one of the building blocks on Neural Network Learn about the different activation functions in deep learning Code activation functions in python and visualize results in live coding window This article was originally published in October 2017 and updated in January 2020 with three new

https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/#:~:text=Softmax%20function%20is%20often%20described,used%20for%20binary%20classification%20problems

```
# Set random seed
tf.random.set_seed(42)

# 1. Create the model
model_6 = tf.keras.Sequential([
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(1,activation='sigmoid')
])

# 2. Compile the model
model_6.compile(loss=tf.keras.losses.binary_crossentropy,
                optimizer=tf.keras.optimizers.Adam(lr=0.001),
                metrics=['accuracy'])

# 3. Fit the model
history = model_6.fit(X, y, epochs=100)
```

We have got 99% of accuracy!

Amazing!! 99% - accuracy

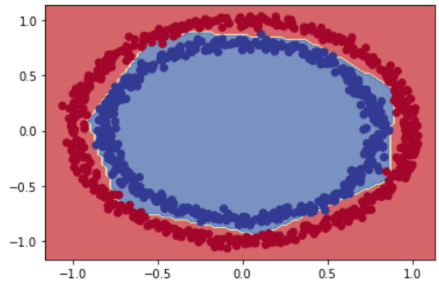
```
# 4. Evaluate our model
model_6.evaluate(X, y)
```

```
32/32 [==============================] - 0s 4ms/step - loss: 0.2948 - accuracy: 0.9910
[0.2948004901409149, 0.9909999966621399]
```

+ Код + Т

```
[103] # Visualize model predictions
plot_decision_boundary(model_6, X, y)
```

```
Binary classification
```



Note

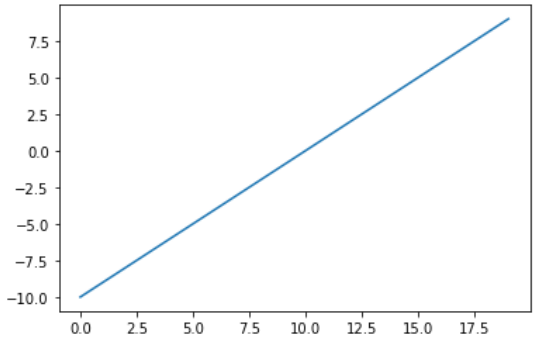
The combination of **linear (straight lines) and non-linear (non-straight lines) functions** is one of the key fundamentals of neural networks.

If we have an unlimited amount of linear and non-linear lines, we could draw any pattern that we wanted to. That is essential what our neural networks do.

Let's take a picture of food as an example. If we wanted to build a neural network to identify the patterns on the above picture. There are a lot of different patterns. We need a lot of straight and non-straight lines. That is the essence of what a neural network is doing when it looks at different examples of data. It is drawing patterns with straight and non-straight lines.

Replicating Non-linear Activation Functions from Scratch

```
# Create a toy tensor (similar to the data we pass into our models)

A = tf.cast(tf.range(-10, 10), tf.float32)
A

# Visualize our toy tensor
plt.plot(A) # Linear Line
```



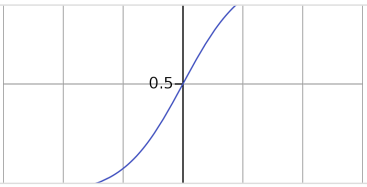
Sigmoid Function

Resource:

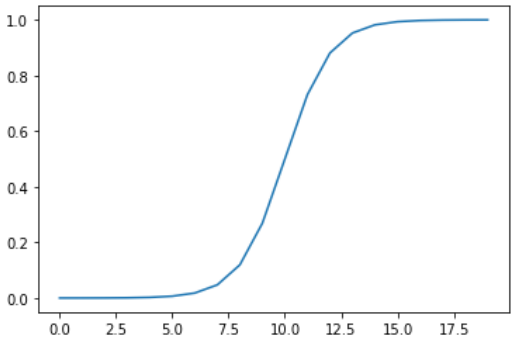
Sigmoid function - Wikipedia

A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. A common example of a sigmoid function is the logistic function shown in the first figure and defined by the formula: Other standard sigmoid functions are given in the Examples section.

https://en.wikipedia.org/wiki/Sigmoid_function




```
def sigmoid(x):
  return 1 / (1 + tf.exp(-x))

# Use the sigmoid funciton on our toy tensor
sigmoid(A)

# Visualize toy tensor transformed by sigmoid
plt.plot(sigmoid(A))
```



ReLU

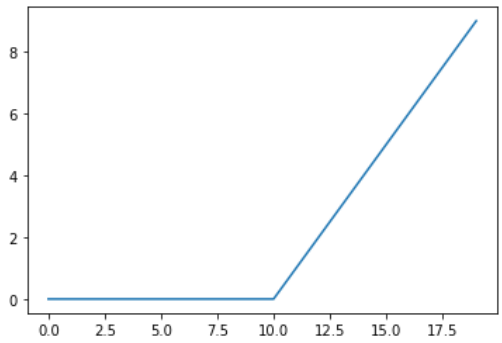
Resource:

A Gentle Introduction to the Rectified Linear Unit (ReLU) - Machine Learning Mastery

In a neural network, the activation function is responsible for transforming the summed weighted input from the node into the activation of the node or output for that input. The rectified linear activation function or ReLU for short is a piecewise linear function that will output the input directly if it is positive, otherwise, it

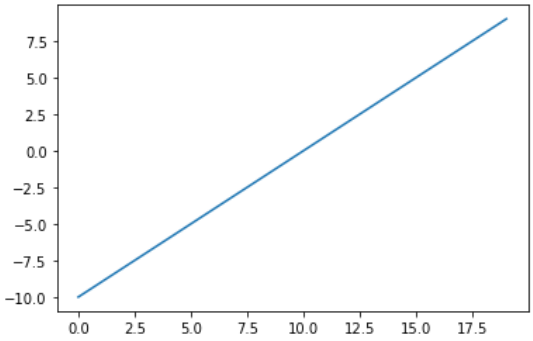
https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/#:~:text=The%20rectified%20linear%20activation%20function,otherwise%2C%20it%20will%20output%20zero


```
def relu(x):
  return tf.maximum(0, x)

# Use the relu function on our toy tensor
relu(A)

# Plot ReLU-modified tensor
plt.plot(relu(A))
```



Linear

```
# Linear Activation Function
tf.keras.activations.linear(A)

plt.plot(tf.keras.activations.linear(A))
```



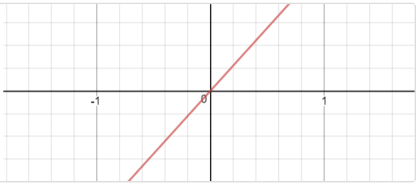
It makes sense now the model doesn't really learn anything when using only linear activation functions, because the linear activation function doesn't change our input data in anyway. Whereas, with our non-linear functions, our data gets manipulated. A neural network uses these kind of transformations at a large scale to figure draw patterns between its inputs and outputs.

Resource:

Activation Functions - ML Glossary documentation

A straight line function where activation is proportional to input (which is the weighted sum from neuron). Pros It gives a range of activations, so it is not binary activation. We can definitely connect a few neurons together and if more than 1 fires, we could take the max (or softmax) and decide based on that.

https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#



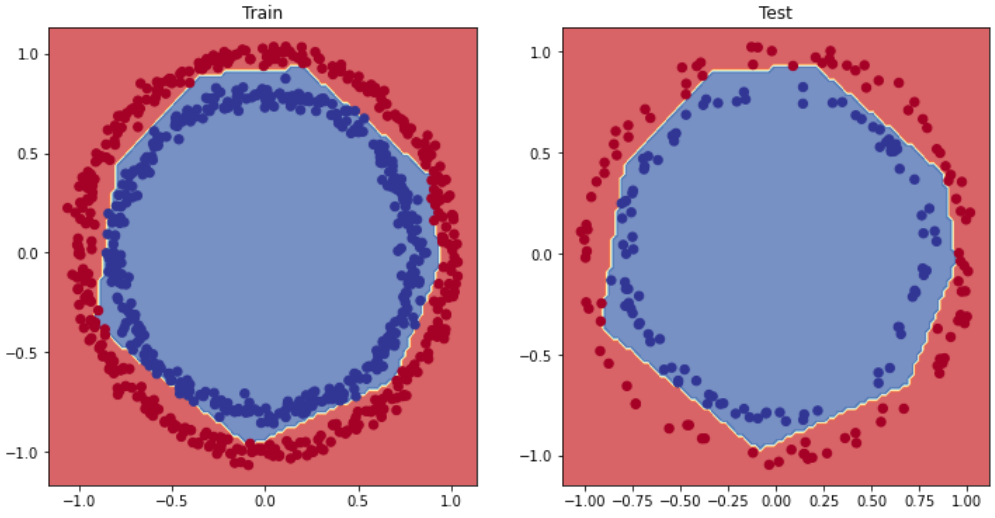
Evaluating and Improving Classification Model

Increasing Learning Rate

```
# Let's recreate a model to fit on the training data and test on the test data

# Set random seed
tf.random.set_seed(42)

# 1. Create the model
model_8 = tf.keras.Sequential([
        tf.keras.layers.Dense(4, activation='relu'),
        tf.keras.layers.Dense(4, activation='relu'),
        tf.keras.layers.Dense(1, activation='sigmoid')
])

# 2. Compile the model
model_8.compile(loss=tf.keras.losses.binary_crossentropy,
                optimizer=tf.keras.optimizers.Adam(lr=0.01), # by increasing the learning rate, our model might find patterns faster
                metrics=['accuracy'])

# 3. Fit the model
history = model_8.fit(X_train, y_train, epochs=25)
```

- Optimizer tells our model how it should improve. How it should update the internal patterns that it has learned.
- The loss function tells how wrong those patterns are
- Learning rate tells how much our model should improve those patterns

We have got 100% of accuracy on test set.



We have changed the following things:

- **The learning_rate (also lr) parameter** - We increased the learning rate parameter in the Adam optimizer to 0.01 instead of 0.001 (an increase of 10x).
    - You can think of the learning rate as how quickly a model learns. The higher the learning rate, the faster the model's capacity to learn, however, there's such a thing as a too high learning rate, where a model tries to learn too fast and

doesn't learn anything. We'll see a trick to find the ideal learning rate soon.

- **The number of epochs** - We lowered the number of epochs (using the epochs parameter) from 100 to 25 but our model still got an incredible result on both the training and test sets.
    - One of the reasons our model performed well in even less epochs (remember a single epoch is the model trying to learn patterns in the data by looking at it once, so 25 epochs means the model gets 25 chances) than before is because we increased the learning rate.

Plot the Loss (or Training) Curves

Plotting the loss (training) curves allow us to look at how did our model go whilst it was learning. As in, how did the performance change everytime the model had a chance to look at the data (once every epoch)?

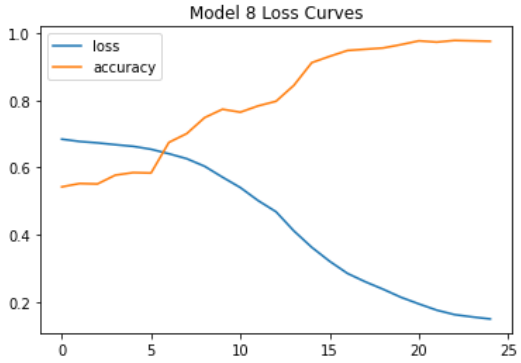
Method `Model.fit()` returns History object. A History object. Its History.history attribute is a record of training loss values and metrics values at successive epochs, as well as validation loss values and validation metrics values (if applicable).

tf.keras.Model | TensorFlow Core v2.8.0

Model groups layers into an object with training and inference features.

https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit

```
# Convert the History object into a DataFrame
pd.DataFrame(history.history)


# Plot the loss curves
pd.DataFrame(history.history).plot()
plt.title("Model 8 Loss Curves");
```

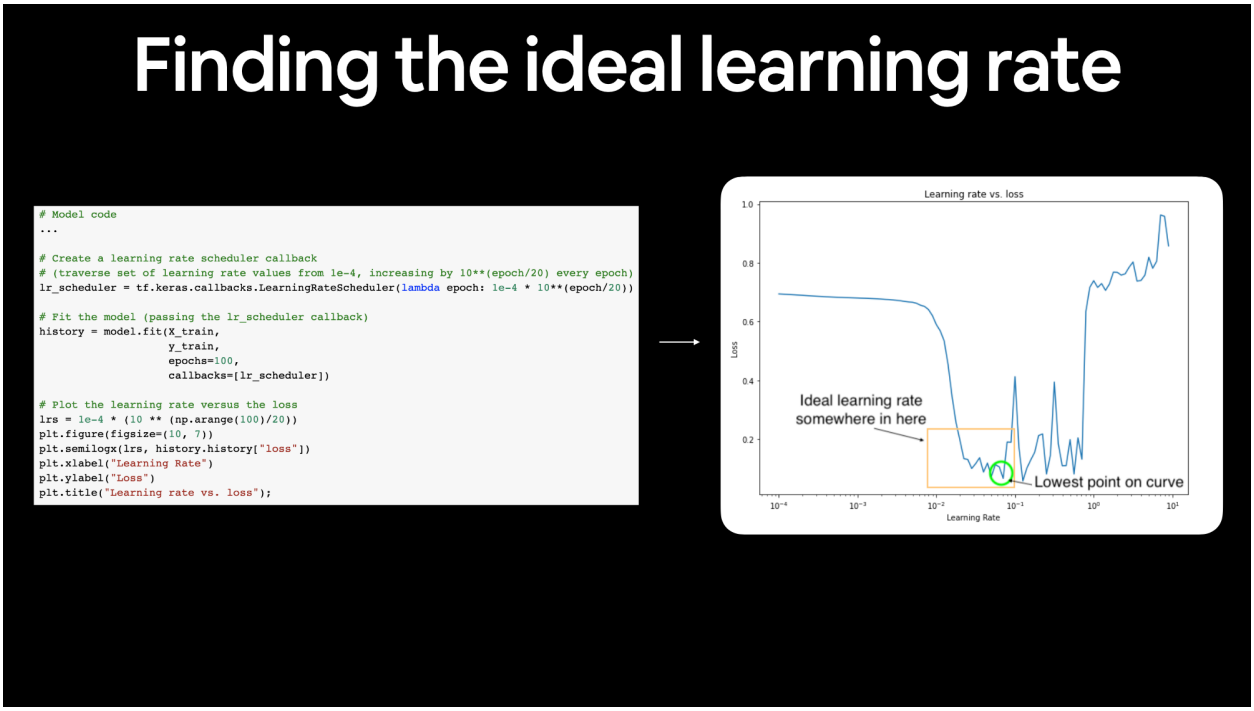


Note:
For many problems, the loss function going down means that the model is improving (the predictions it is making are getting close to the ground truth labels).

Finding the best Learning Rate

To find the ideal learning rate (the learning rate where the loss decreases the most during training). We're going to use the following steps:

- A learning rate **callback** - we can think of a callback as an extra piece of functionality, we can add to our model while it is training.
- Another model
- A modified loss curves plot



> **Note**:
> The default hyperparameters of many neural network building blocks in TensorFlow are setup in a way which usually work right out of the box (e.g. the Adam optimizer's
> default settings can usually get good results on many datasets). So it's a good idea to try the defaults first, then adjust as needed.

Setting up a callback function:

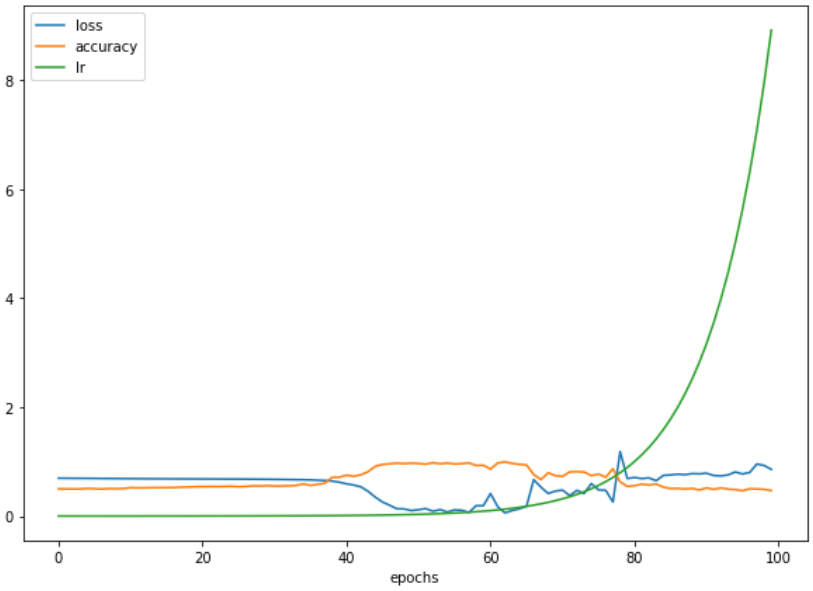
tf.keras.callbacks.LearningRateScheduler | TensorFlow Core v2.8.0

Learning rate scheduler.

https://www.tensorflow.org/api_docs/python/tf/keras/callbacks/LearningRateScheduler

```
# Set random seed
tf.random.set_seed(42)

# 1. Create a model (same as model_8)
model_9 = tf.keras.Sequential([
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(4, activation='relu'),
```

```
      tf.keras.layers.Dense(1, activation='sigmoid')
])

# 2. Compile the model
model_9.compile(loss='binary_crossentropy',
                optimizer="Adam",
                metrics=["accuracy"])

# A callback works during model training
# 3. Create a learning rate callback
# Callback is going to give our optimize the updated learning rate
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-4 * 10**(epoch/20))

# 4. Fit the model (passing lr_scheduler callback)
history_9 = model_9.fit(X_train, y_train, epochs=100,callbacks=[lr_scheduler])

# Checkout the history
pd.DataFrame(history_9.history).plot(figsize=(10,7), xlabel='epochs');
```



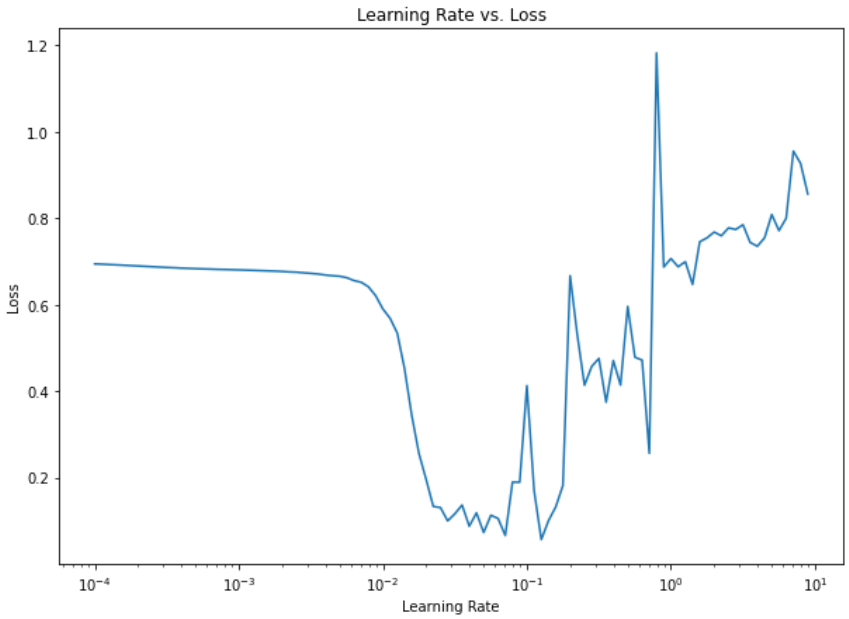
As we can see the learning rate exponentially increases as the number of epochs increases, whereas, the model's accuracy goes up (and loss goes down) at a specific point when the learning rate slowly increases.

matplotlib.pyplot.semilogx - Matplotlib 3.5.1 documentation

Animated image using a precomputed list of images

https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.semilogx.html

```
# Plot the learning rate vs. the loss
lrs = 1e-4 * (10 ** (tf.range(100)/20))
lrs

plt.figure(figsize=(10,7))
plt.semilogx(lrs, history_9.history["loss"])
plt.xlabel("Learning Rate")
plt.ylabel("Loss")
plt.title("Learning Rate vs. Loss");
```



To figure out the ideal value of the learning rate (at least the ideal value to begin training our model), the rule of thumb is to take the learning rate value where the loss is still decreasing but not quite flattened out (usually about 10x smaller than the bottom of the curve).

The ideal learning rate at the start of model training is somewhere just before the loss curve bottoms out (a value where the loss is still decreasing).

```
# Example of other typical learning rate values
10**0, 10**-1, 10**-2, 10**-3, 1e-4
```

The ideal learning rate is going to be between the lowest point and 10x smaller than that point.

tf.keras.optimizers.Adam | TensorFlow Core v2.8.0

Optimizer that implements the Adam algorithm.

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/Adam

Adam optimizer uses 0,001 lr.

Classification Evaluation Methods

(some common)

Classification evaluation methods

Key: **tp** = True Positive, **tn** = True Negative, **fp** = False Positive, **fn** = False Negative

Metric Name	Metric Forumla	Code	When to use
Accuracy	**Accuracy** $= \frac{tp + tn}{tp + tn + fp + fn}$	`tf.keras.metrics.Accuracy()` or `sklearn.metrics.accuracy_score()`	Default metric for classification problems. Not the best for imbalanced classes.
Precision	**Precision** $= \frac{tp}{tp + fp}$	`tf.keras.metrics.Precision()` or `sklearn.metrics.precision_score()`	Higher precision leads to less false positives.
Recall	**Recall** $= \frac{tp}{tp + fn}$	`tf.keras.metrics.Recall()` or `sklearn.metrics.recall_score()`	Higher recall leads to less false negatives.
F1-score	**F1-score** $= 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$	`sklearn.metrics.f1_score()`	Combination of precision and recall, usually a good overall metric for a classification model.
Confusion matrix	NA	`Custom function` or `sklearn.metrics.confusion_matrix()`	When comparing predictions to truth labels to see where model gets confused. Can be hard to use with large numbers of classes.

(some common)

Classification evaluation methods

Key: **tp** = True Positive, **tn** = True Negative, **fp** = False Positive, **fn** = False Negative

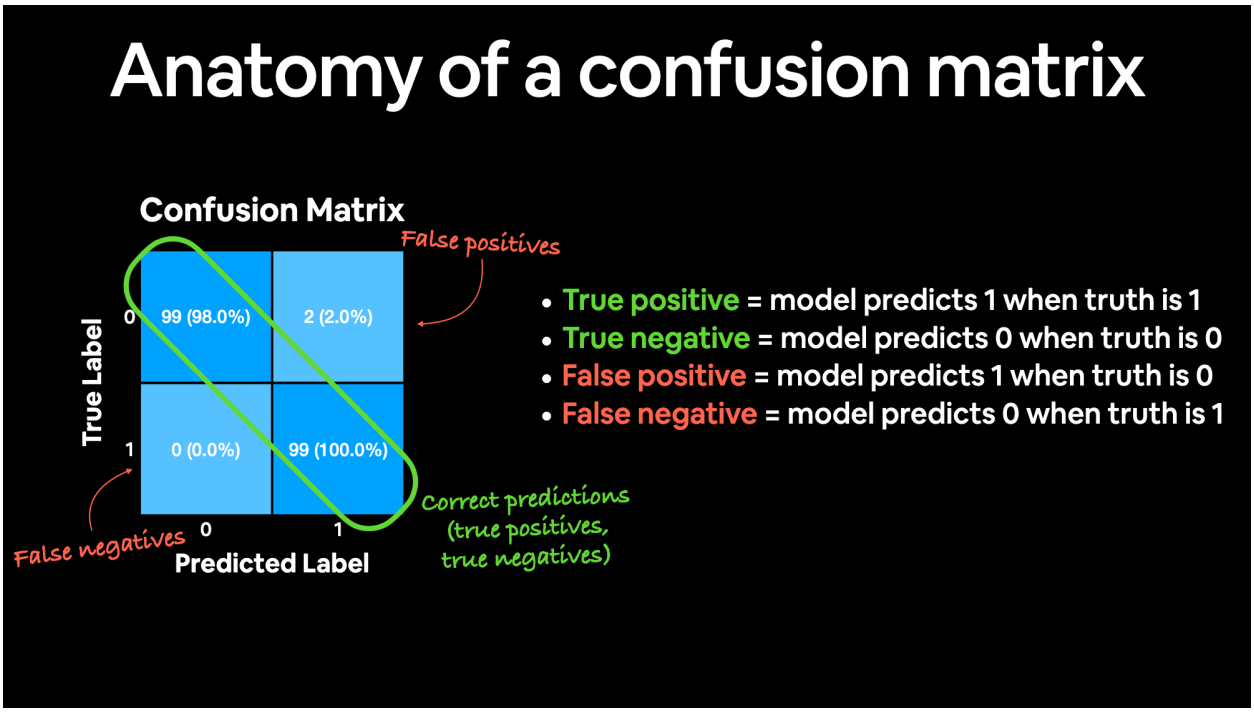
Metric Name	Metric Forumla	Code	When to use
Accuracy	**Accuracy** $= \frac{tp + tn}{tp + tn + fp + fn}$	`tf.keras.metrics.Accuracy()` or `sklearn.metrics.accuracy_score()`	Default metric for classification problems. Not the best for imbalanced classes.
Precision	**Precision** $= \frac{tp}{tp + fp}$	`tf.keras.metrics.Precision()` or `sklearn.metrics.precision_score()`	Higher precision leads to less false positives.
Recall	**Recall** $= \frac{tp}{tp + fn}$	`tf.keras.metrics.Recall()` or `sklearn.metrics.recall_score()`	Higher recall leads to less false negatives.
F1-score	**F1-score** $= 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$	`sklearn.metrics.f1_score()`	Combination of precision and recall, usually a good overall metric for a classification model.
Confusion matrix	NA	`Custom function` or `sklearn.metrics.confusion_matrix()`	When comparing predictions to truth labels to see where model gets confused. Can be hard to use with large numbers of classes.

- Accuracy
- Precision
- Recall
- F1-score
- Confusion Matrix

- Classification report (from scikit-learn)

```
# Check the accuracy of our model
loss, accuracy = model_10.evaluate(X_test, y_test)
print(f"Model loss on the test set: {loss}")
print(f"Model accuracy on the test set: {(accuracy*100):.2f}%")
```

Confusion Matrix



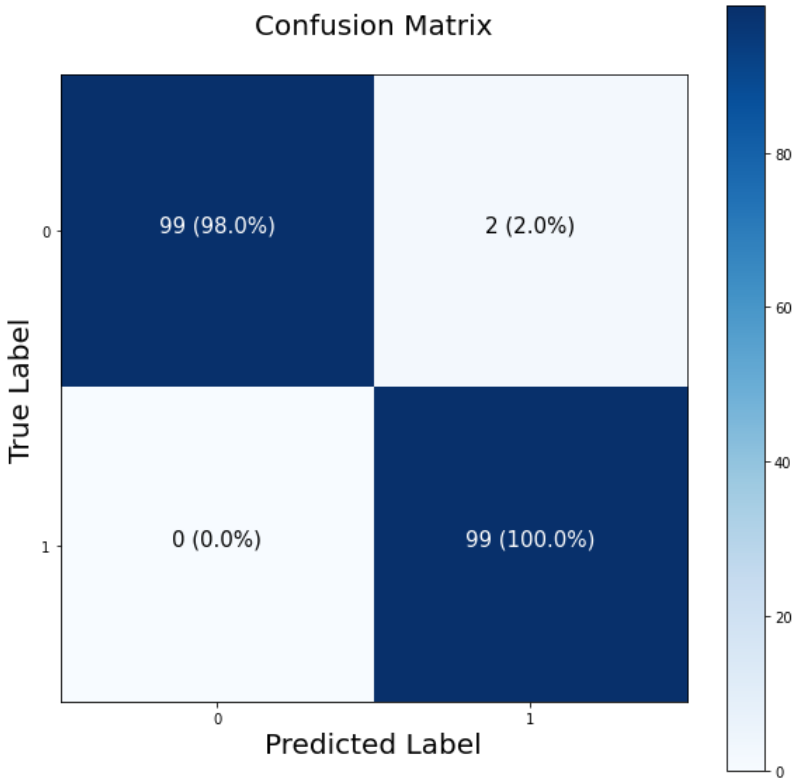

```
# Import a confusion matrix
from sklearn.metrics import confusion_matrix

# Make predictions
y_preds = model_10.predict(X_test)

# Creare confusion matrix
confusion_matrix(y_test, y_preds) # Will result to an error, since y_preds are prediction probabilities
```

Our predictions array has come out in **prediction probability** form. The standard output from the sigmoid (or softmax) activation functions.

```
9.852e-01 # Close to 1

# Everything high 0.5 goes to 1
# Everything low 0.5 goes to 0
# Convert prediction probabilities to binary format and view the first 10
tf.round(y_preds)[:10]

# Create a confusion matrix
confusion_matrix(y_test, tf.round(y_preds))
```

Create Custom Confusion Matrix

```
import itertools

figsize=(10,10)

def plot_confusion_matrix(y_test, y_preds):
  # Create the confusion matrix
  cm = confusion_matrix(y_test, tf.round(y_preds))
  # Normalize our confusion matrix
  cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis]
  # Define number of classes
  n_classes = cm.shape[0]

  # Let's prettify it
  fig, ax = plt.subplots(figsize=figsize)
  # Create a matrix plot - https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.matshow.html
  cax = ax.matshow(cm, cmap=plt.cm.Blues)
  fig.colorbar(cax)

  # Create classes
  classes = False

  if classes:
    labels = classes
  else:
    labels = np.arange(cm.shape[0])

  # Label the axes
  ax.set(title="Confusion Matrix",
        xlabel="Predicted Label",
        ylabel="True Label",
        xticks=np.arange(n_classes),
        yticks=np.arange(n_classes),
        xticklabels=labels,
        yticklabels=labels)

  # Set x-axis labels to bottom
  ax.xaxis.set_label_position("bottom")
  ax.xaxis.tick_bottom()

  # Adjust label size
  ax.yaxis.label.set_size(20)
  ax.xaxis.label.set_size(20)
  ax.title.set_size(20)

  # Set threshold for different colors
  threshold = (cm.max() + cm.min()) / 2.

  # plot the text on each cell
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, f"{cm[i,j]} ({cm_norm[i, j] * 100:.1f}%)",
             horizontalalignment="center",
             color="white" if cm[i, j] > threshold else "black",
             size=15)


# Call the function
plot_confusion_matrix(y_test, y_preds)
```



Multi-class Classification

When we have more than two classes as an option, it's known as **multi-class classification.**

- This means if we have 3 and more different classes, it's multi-class classification.

Getting the Data

```
import tensorflow as tf
from tensorflow.keras.datasets import fashion_mnist

(train_data, train_labels), (test_data, test_labels) = fashion_mnist.load_data()
```

fashion_mnist | TensorFlow Datasets

Fashion-MNIST is a dataset of Zalando's article images consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes.

https://www.tensorflow.org/datasets/catalog/fashion_mnist

TensorFlow

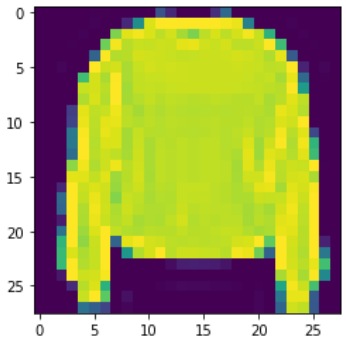
Become One with the Data

Check the dataset

```
# Show the first training examples
print(f"Training sample:\n{train_data[0]}\n")
print(f"Training label:\n{train_labels[0]}\n")

# Check the shape of a single example
train_data[0].shape, train_labels[0].shape
```

Check the images

```
# Plot a single sample
import matplotlib.pyplot as plt
plt.imshow(train_data[7])
```

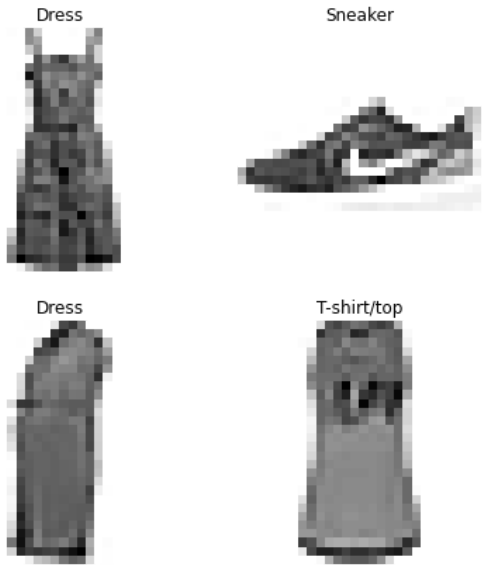


Check the labels

```
# Check out the label
train_labels[7]
```

```
# Create a small list so we can index onto our training lables so that they are human-readable
class_names = ["T-shirt/top","Trouser","Pullover", "Dress","Coat","Sandal","Shirt","Sneaker","Bag","Ankle Boot"]

len(class_names)
```

```
# Plot multiple random images of fashion MNIST
import random
plt.figure(figsize=(7,7))
for i in range(4):
  ax = plt.subplot(2,2, i+1)
  rand_index = random.choice(range(len(train_data)))
  plt.imshow(train_data[rand_index], cmap=plt.cm.binary)
  plt.title(class_names[train_labels[rand_index]])
  plt.axis(False);
```



Building a Multi-class Classification Model

For our multi-class classification model, we can use a similar architecture to our binary classifiers, however, we're going to have to tweak a few things:

- Input shape = 28 X 28 (the shape of our image)
- Output shape = 10 (one per class of clothing)
- Loss function = tf.keras.losses.CategoricalCrossentropy()
- Output layer activation = Softmax

```
# Out data needs to be flattened from 28*28 to (None, 784)
flatten_model = tf.keras.Sequential([tf.keras.layers.Flatten(input_shape=(28,28))])
flatten_model.output_shape
```

tf.keras.layers.Flatten | TensorFlow Core v2.8.0

Flattens the input. Does not affect the batch size.

👇 https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten

```
# Set random seed
tf.random.set_seed(42)

# 1. Create a model
model_11 = tf.keras.Sequential([
      tf.keras.layers.Flatten(input_shape=(28,28)), # Flat the input shape into long vector and pass it to the layers
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(10, activation='softmax')
])

# 2. Compile the model
model_11.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(), # If the labels are not hot-encoded
```

```
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])

# 3. Fit the model
non_norm_history = model_11.fit(train_data,
                                train_labels,
                                epochs=10,
                                validation_data=(test_data, test_labels))

# Check the model summary
model_11.summary()
```

If the labels are not hot-encoded, use SparseCategoricalCrossentropy()

tf.keras.losses.SparseCategoricalCrossentropy | TensorFlow Core v2.8.0

Computes the crossentropy loss between the labels and predictions.

https://www.tensorflow.org/api_docs/python/tf/keras/losses/SparseCategoricalCrossentropy

Normalizing the Data

```
# Check the min and max values of the training data
train_data.min(), train_data.max()
```

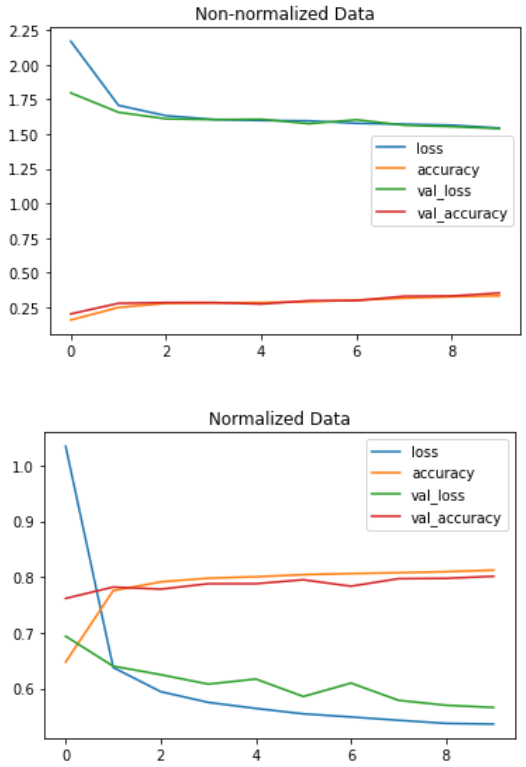
Neural networks prefer data to be scaled (or normalized), this means that they like to have the numbers in tensors they try to find patterns between 0 and 1.

```
# We can get our training and testing data between 0 and 1 by dividing the maximum()
# This refers to scaling or normalization - getting the dataset between 0 and 1

train_data_norm = train_data / 255.0
test_data_norm = test_data / 255

# Check the min and max values of scaled training data
train_data_norm.min(), train_data_norm.max()
```

```
# Now our data is normalized, lets build a model to find patterns in it

# Set random seed
tf.random.set_seed(42)

# 1. Create a model
model_12 = tf.keras.Sequential([
      tf.keras.layers.Flatten(input_shape=(28,28)),
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(10, activation='softmax')
])

# 2. Compile the model
model_12.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["accuracy"])

# 3. Fit the model
norm_history = model_12.fit(train_data_norm,
                            train_labels,
                            epochs=10,
                            validation_data=(test_data_norm, test_labels))
```

Note:
Neural Networks tend to prefer data in numerical form as well as scaled/normalized (numbers between 0 and 1).

```
import pandas as pd
# Plot non-normalized data loss curves
pd.DataFrame(non_norm_history.history).plot(title="Non-normalized Data")
# Plot normalized data loss curves
pd.DataFrame(norm_history.history).plot(title="Normalized Data");
```



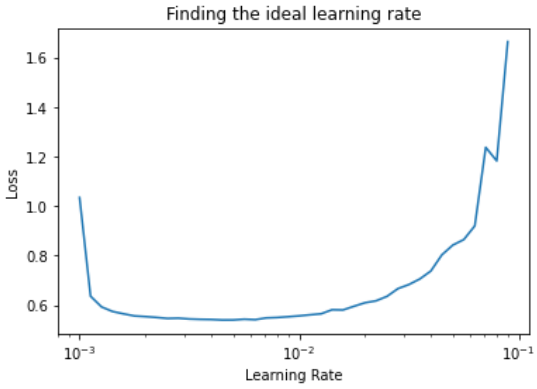
Note:
The same model with even *slightly* different data can produce *dramatically* different results.
When comparing models, it is important to make sure that we're comparing them on the same criteria (e.g same architecture but different data or same data but different architecture).

Finding Ideal Learning Rate

```
from sklearn.utils import validation
# Set random seed
tf.random.set_seed(42)

# 1. Create a model
```

```
model_13 = tf.keras.Sequential([
      tf.keras.layers.Flatten(input_shape=(28,28)),
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(4, activation='relu'),
      tf.keras.layers.Dense(10, activation='softmax')
])

# 2. Compile the model
model_13.compile(loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                 optimizer=tf.keras.optimizers.Adam(),
                 metrics=["accuracy"])

# 3. Create the learning rate callback
lr_scheduler = tf.keras.callbacks.LearningRateScheduler(lambda epoch: 1e-3 * 10**(epoch/20))

# 4. Fit the model
find_lr_history = model_13.fit(train_data_norm,
                               train_labels,
                               epochs=40,
                               validation_data=(test_data_norm, test_labels),
                               callbacks=(lr_scheduler))
```

```
# Plot the learning rate curve
import numpy as np
import matplotlib.pyplot as plt

lrs= 1e-3 * (10**(tf.range(40)/20))
plt.semilogx(lrs, find_lr_history.history["loss"])
plt.xlabel("Learning Rate")
plt.ylabel("Loss")
plt.title("Finding the ideal learning rate");
```



Evaluating Multi-class Classification Model

To evaluate our multi-class classification model we could:

- Evaluate its performance using other classification metrics (such as confusion matrix)
- Assess some of its predictions (through visualizations)
- Improve its results (by training it for longer or changing the architecture)
- Save and export it for use in an application

```
# Make some predictions with our model
y_probs = model_14.predict(test_data_norm) # probs is short for prediction probabilities

# View the first 5 predictions
y_probs[:5]
```

Note:
Remember to make predictions on the same kind of data your model was trained on (e.g if your model was trained on normalized data, you'll want to make predictions on normalized data)

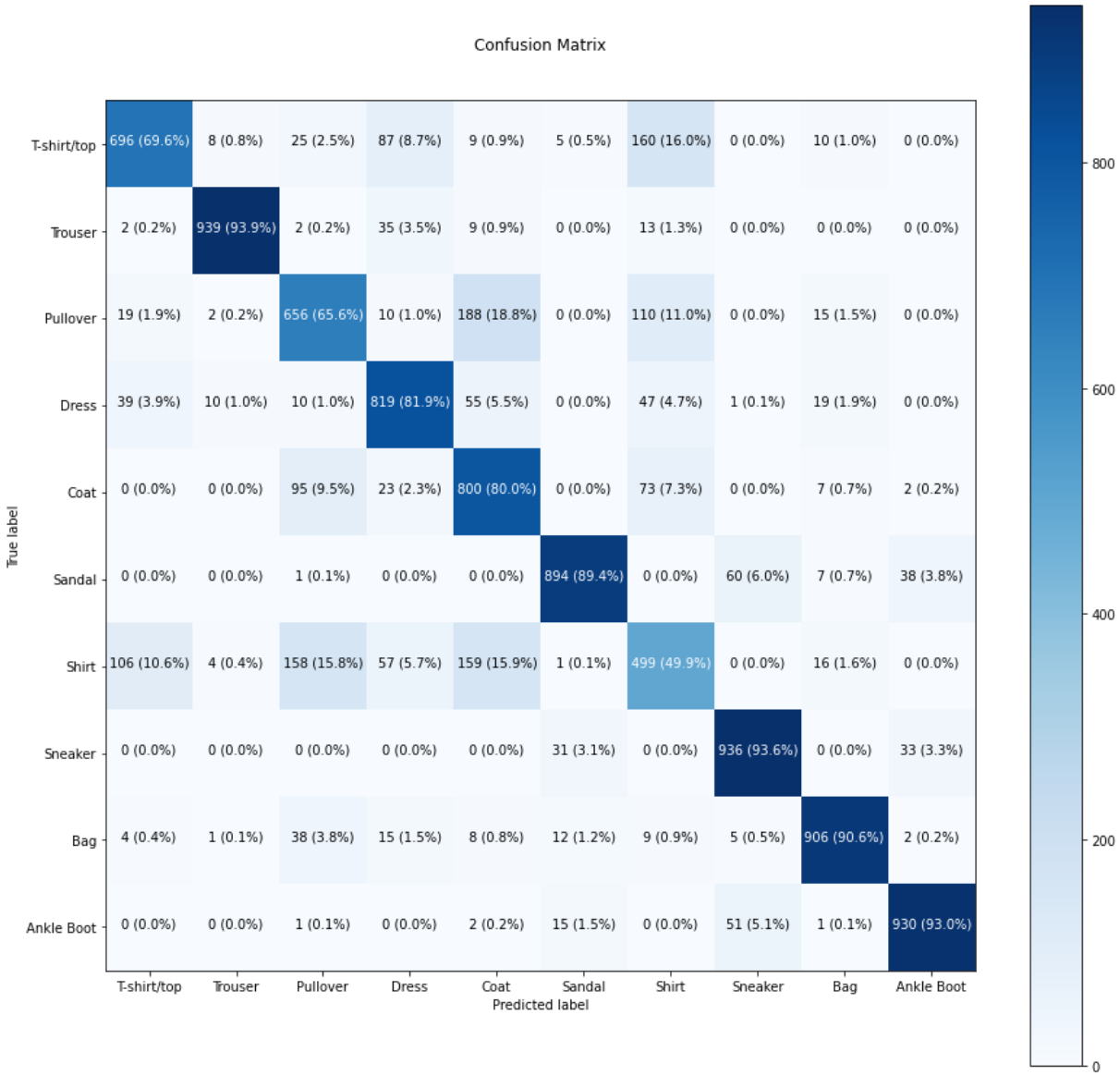
```
y_probs[0], tf.argmax(y_probs[0]), class_names[tf.argmax(y_probs[0])]

# Convert all of the predictions probabilities into integers
y_preds = y_probs.argmax(axis=1)

# View the first 10 predictions labels
y_preds[:10], test_labels[:10]

from sklearn.metrics import confusion_matrix
confusion_matrix(y_true=test_labels, y_pred=y_preds)
```

Custom Confusion Matrix

```
# Custom Multi-class confusion matrix
def make_confusion_matrix(y_true, y_pred, classes=None, figsize=(10, 10), text_size=15):
  """Makes a labelled confusion matrix comparing predictions and ground truth labels.

  If classes is passed, confusion matrix will be labelled, if not, integer class values
  will be used.

  Args:
    y_true: Array of truth labels (must be same shape as y_pred).
    y_pred: Array of predicted labels (must be same shape as y_true).
    classes: Array of class labels (e.g. string form). If `None`, integer labels are used.
    figsize: Size of output figure (default=(10, 10)).
    text_size: Size of output figure text (default=15).

  Returns:
    A labelled confusion matrix plot comparing y_true and y_pred.

  Example usage:
    make_confusion_matrix(y_true=test_labels, # ground truth test labels
                          y_pred=y_preds, # predicted labels
                          classes=class_names, # array of class label names
                          figsize=(15, 15),
                          text_size=10)
  """
  # Create the confustion matrix
  cm = confusion_matrix(y_true, y_pred)
  cm_norm = cm.astype("float") / cm.sum(axis=1)[:, np.newaxis] # normalize it
  n_classes = cm.shape[0] # find the number of classes we're dealing with

  # Plot the figure and make it pretty
  fig, ax = plt.subplots(figsize=figsize)
  cax = ax.matshow(cm, cmap=plt.cm.Blues) # colors will represent how 'correct' a class is, darker == better
  fig.colorbar(cax)

  # Are there a list of classes?
  if classes:
    labels = classes
  else:
```

```
    labels = np.arange(cm.shape[0])

  # Label the axes
  ax.set(title="Confusion Matrix",
         xlabel="Predicted label",
         ylabel="True label",
         xticks=np.arange(n_classes), # create enough axis slots for each class
         yticks=np.arange(n_classes),
         xticklabels=labels, # axes will labeled with class names (if they exist) or ints
         yticklabels=labels)

  # Make x-axis labels appear on bottom
  ax.xaxis.set_label_position("bottom")
  ax.xaxis.tick_bottom()

  # Set the threshold for different colors
  threshold = (cm.max() + cm.min()) / 2.

  # Plot the text on each cell
  for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
    plt.text(j, i, f"{cm[i, j]} ({cm_norm[i, j]*100:.1f}%)",
             horizontalalignment="center",
             color="white" if cm[i, j] > threshold else "black",
             size=text_size)
```

```
# Make a prettier confusion matrix
make_confusion_matrix(y_true=test_labels, y_pred=y_preds, classes=class_names, figsize=(15,15), text_size=10)
```

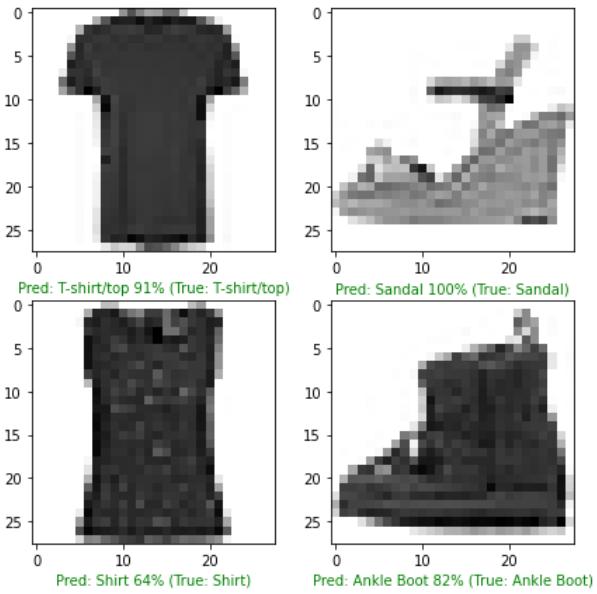


Note:
Often when working with images and other forms of visual data, its a good idea to visualize as much as possible to develop a further understanding of the data and the inputs and outputs of your models.

```
import random

def plot_random_image(model, images, true_labels, classes):
  """
  Picks a random iamge, plots it and labels it with a prediction
  and truth label
  """
  plt.figure(figsize=(7,7))
  for i in range(4):
    ax = plt.subplot(2,2,i+1)
    # Set up a random integer
```

```
    i = random.randint(0, len(images))

    # Create predictions and targets
    target_image = images[i]
    pred_probs = model.predict(target_image.reshape(1, 28, 28))
    pred_label = classes[pred_probs.argmax()]
    true_label = classes[true_labels[i]]

    # Plot the image
    plt.imshow(target_image, cmap=plt.cm.binary)

    # Change the color of the titles depending on if the prediction is right or wrong
    if pred_label == true_label:
      color = "green"
    else:
      color = "red"

    # Add xlabel information (prediction/true label)
    plt.xlabel("Pred: {} {:2.0f}% (True: {})".format(pred_label,
                                                     100*tf.reduce_max(pred_probs),
                                                     true_label),
               color=color) # set the color to green or red based on if the prediction is right or wrong
```

```
# Check out a random image as well as its prediction
plot_random_image(model=model_14,
                  images=test_data_norm,
                  true_labels=test_labels,
                  classes=class_names)
```



Patterns that our model has learned

We can list the layers of the model by accessing attribute `layers`

```
# Find the layers of our most recent mdoel
model_14.layers
```

We can access a target layer through indexing

```
model_14.layers[1]
```

To get the patterns fo a layer in out network, we need to call method `get_weights()` The method returns weights and biases of the model

```
# Get the patterns of a layer in our network
weights, biases = model_14.layers[1].get_weights()

# Shapes
weights, weights.shape # Internal patterns of Input Layer


--- OUPUT ---
array([[ 0.7150263 , -0.06077093, -0.99763095, -1.0484313 ],
       [ 0.2773212 , -0.471554  , -0.52916455,  0.02329255],
       [ 0.7752433 ,  0.5402759 , -1.128857  , -0.7426156 ],
       ...,
       [-0.3945342 ,  0.47628632, -0.2264153 ,  0.2550593 ],
       [-0.40515798,  0.61810046,  0.23928414, -0.50387603],
       [ 0.23884593,  0.11606961, -0.12131374,  0.04352392]],
      dtype=float32), (784, 4))
```

For us, those tensors might look like a bunch of numbers. But for our neural network, it consider these numbers as patterns which contribute to the decision that it makes.

```
model_14.summary()
```

```
Model: "sequential_16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_4 (Flatten)         (None, 784)               0

 dense_38 (Dense)            (None, 4)                 3140

 dense_39 (Dense)            (None, 4)                 20

 dense_40 (Dense)            (None, 10)                50

=================================================================
Total params: 3,210
Trainable params: 3,210
Non-trainable params: 0
_________________________________________________________________
```

The weights matrix is the same shape as the input data, which in our case is 784 (28x28 pixels). And there's a copy of the weights matrix for each neuron the in the selected layer (our selected layer has 4 neurons). Each value in the weights matrix corresponds to how a particular value in the input data influences the network's decisions.

That means that for each datapoint in our input tensor, our weights matrix has four numbers that it starts to learn and adjust to find patterns in these 784 numbers.

```
# Shape = 1 bias per neuron (we use 4 neurons in the first layer)
biases, biases.shape
```

Every neuron has a bias vector. Each of these is paired with a weight matrix.

The bias values get initialized as zeroes by default (using the `bias_initializer` parameter).

The bias vector dictates how much the patterns within the corresponding weights matrix should influence the next layer.`