Table of Contents

Introduction

- intro
  - exploring how computers learn with machine learning through fitting internal parameters of a function, or a neuron, to match x and y values together, when there was a linear relationship between them.
  - This is often called regression, where you're using a neural network to predict a single value from one or more inputs.
- block
  - classification, where the network can tell you something about the data.
  - For example, if it's an image, it can tell you the contents of that image.

intro

How the training loop works. You make a guess. You measure how accurate that guess is by calculating its loss. And then you use that data to make another guess with the next guess usually being a little bit better than the current one. The idea is that the smaller the loss, the more accurate your guess is.

- patterns - Fitting lines
- loss play

How minimize your loss. look at how gradients and derivatives can help us understand how to minimize loss - how an optimizer function works.

- Gradient descent
- neuralNetworks
- layers

Fitting lines

Finding patterns

Bookmark this page

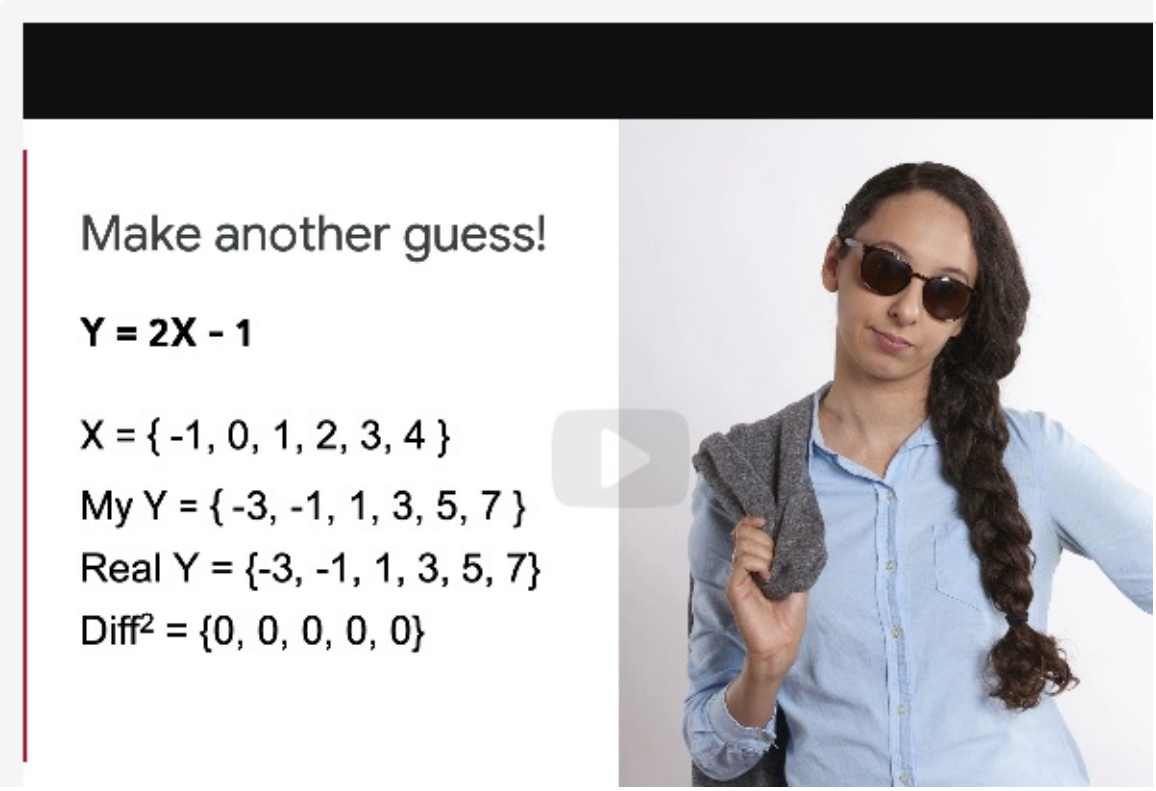
In the previous video you learned about how traditional programming is where you explicitly figure out the rules that act on some data to give an answer like this:

And then you saw that Machine Learning changes this, for scenarios where you may not be able to figure out the rules feasibly, and instead have a computer figure out what they are. That made the diagram look like this:

Then you read about the steps that a computer takes -- where it makes a guess, then looks at the data to figure out how accurate the guess was, and then makes another guess and so on.

see loss play

Thinking about loss...

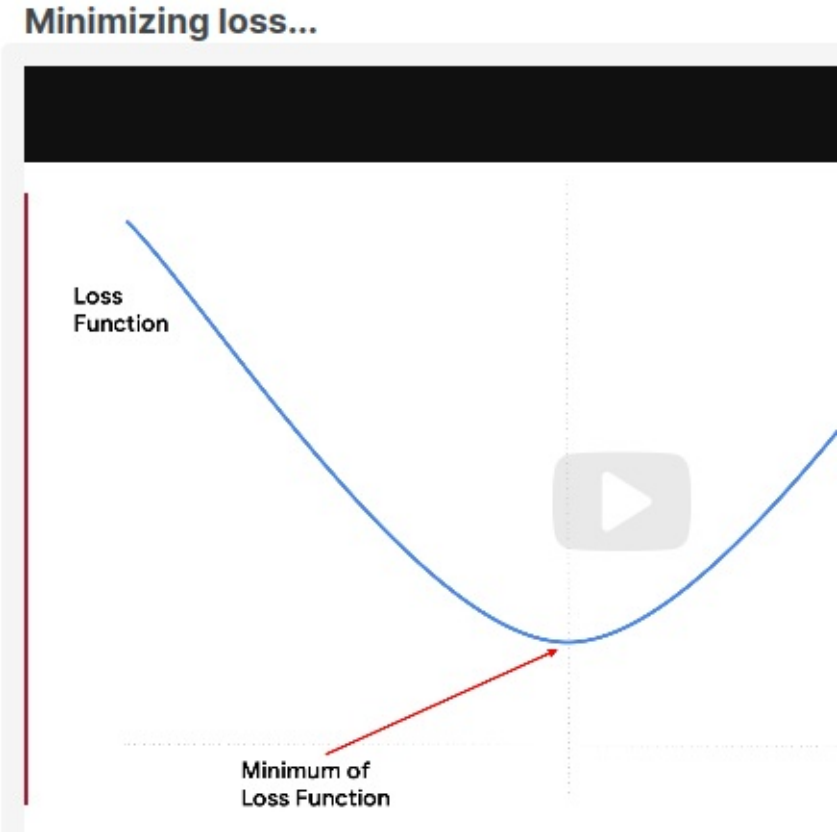


```
import math
# Edit these parameters to try different loss
# measurements. Rerun this cell when done
# Your Y will be calculated as Y=wX+b, so
# if w=3, and b=-1, then Y=3x-1

w = 3
b = -1

x = [-1, 0, 1, 2, 3, 4]
y = [-3, -1, 1, 3, 5, 7]
myY = []


for thisX in x:
  thisY = (w*thisX)+b
  myY.append(thisY)

print("Real Y is " + str(y))
print("My Y is   " + str(myY))

# let's calculate the loss
total_square_error = 0
for i in range(0, len(y)):
  square_error = (y[i] - myY[i]) ** 2
  total_square_error += square_error

print("My loss is: " + str(math.sqrt(total_square_error)))
```

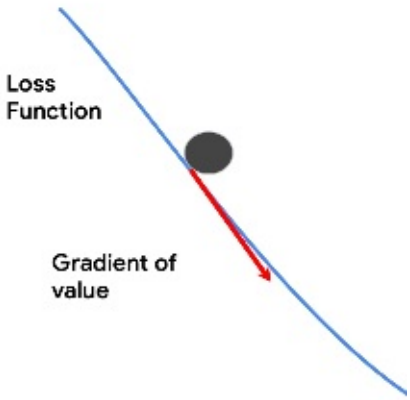
```
Real Y is [-3, -1, 1, 3, 5, 7]
My Y is   [-4, -1, 2, 5, 8, 11]
My loss is: 5.5677643628300215
```

Gradient descent

plot our loss function - squaring values to remove the negatives - so our function is a square of something. Square functions have a parabolic shape



If you differentiate the values with respect to the loss function, you'll get a gradient. use it to determine the direction towards the bottom. It's actually the negative of the gradient points that way. You have no idea how far it is to the bottom, but at least you know the correct direction.



You know what the direction is, and you can pick a step size. The step size is often called the learning rate.

an advanced technique is to adjust the learning rate on the fly.

- see lab2

lab2

- see: lab2_Mimimizing_Loss.ipynb

```
# First import the functions we will need
from __future__ import absolute_import, division, print_function, unicode_literals

try:
  # %tensorflow_version only exists in Colab.
  %tensorflow_version 2.x
except Exception:
  pass
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
```

```
# Define our initial guess
INITIAL_W = 10.0
INITIAL_B = 10.0

# Define our loss function
def loss(predicted_y, target_y):
  return tf.reduce_mean(tf.square(predicted_y - target_y))

# Define our training procedure
def train(model, inputs, outputs, learning_rate):
  with tf.GradientTape() as t:
    current_loss = loss(model(inputs), outputs)
    # Here is where you differentiate the model values with respect to the loss function
    dw, db = t.gradient(current_loss, [model.w, model.b])
    # And here is where you update the model values based on the learning rate chosen
    model.w.assign_sub(learning_rate * dw)
    model.b.assign_sub(learning_rate * db)
    return current_loss

# Define our simple linear regression model
class Model(object):
  def __init__(self):
    # Initialize the weights
    self.w = tf.Variable(INITIAL_W)
    self.b = tf.Variable(INITIAL_B)

  def __call__(self, x):
    return self.w * x + self.b
```

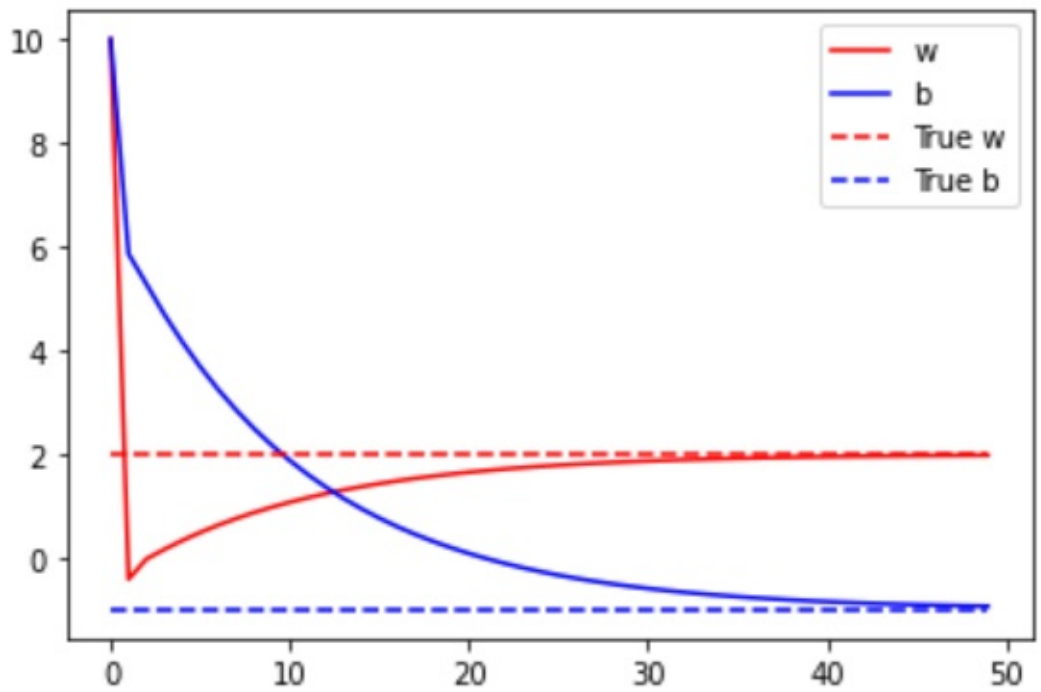
```


# Define our input data and learning rate
xs = [-1.0, 0.0, 1.0, 2.0, 3.0, 4.0]
ys = [-3.0, -1.0, 1.0, 3.0, 5.0, 7.0]
LEARNING_RATE=0.09

# Instantiate our model
model = Model()

# Collect the history of w-values and b-values to plot later
list_w, list_b = [], []
epochs = range(50)
losses = []
for epoch in epochs:
  list_w.append(model.w.numpy())
  list_b.append(model.b.numpy())
  current_loss = train(model, xs, ys, learning_rate=LEARNING_RATE)
```

```
  losses.append(current_loss)
  print('Epoch %2d: w=%1.2f b=%1.2f, loss=%2.5f' %
        (epoch, list_w[-1], list_b[-1], current_loss))
```

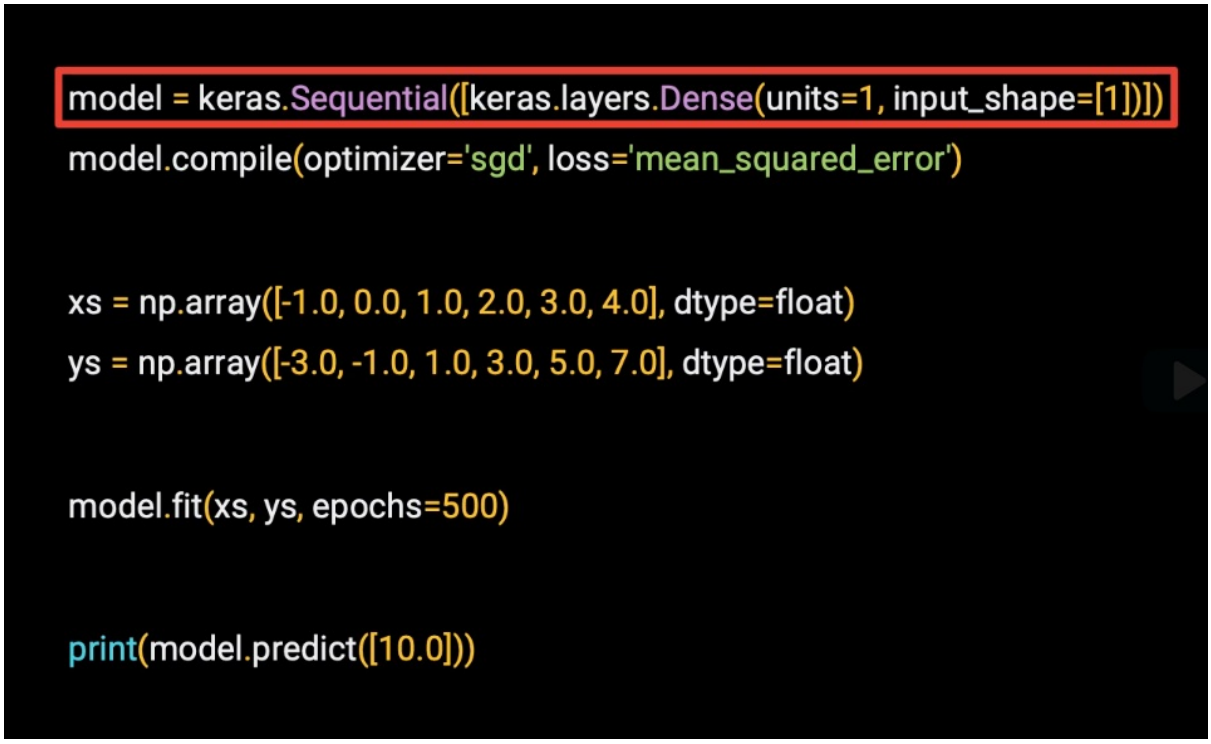
```
Mimimizing-Loss.ipynb_
GradientTape

The Calculus is managed by a TensorFlow Gradient Tape. You can learn more about the gradient tape at https://ww
w.tensorflow.org/api_docs/python/tf/GradientTape, and we will discuss it later in the course.
Train our model

Epoch  0: w=10.00 b=10.00, loss=715.66669
Epoch  1: w=-0.41 b=5.86, loss=27.47032
Epoch  2: w=-0.02 b=5.28, loss=22.43888
Epoch  3: w=0.16 b=4.69, loss=18.46284
Epoch  4: w=0.33 b=4.16, loss=15.19137
Epoch  5: w=0.49 b=3.68, loss=12.49958
Epoch  6: w=0.63 b=3.25, loss=10.28476
Epoch  7: w=0.76 b=2.85, loss=8.46238
Epoch  8: w=0.87 b=2.50, loss=6.96291
Epoch  9: w=0.98 b=2.17, loss=5.72914
Epoch 10: w=1.07 b=1.88, loss=4.71398
Epoch 11: w=1.16 b=1.61, loss=3.87870
Epoch 12: w=1.24 b=1.37, loss=3.19143
Epoch 13: w=1.31 b=1.15, loss=2.62593
Epoch 14: w=1.37 b=0.95, loss=2.16064
Epoch 15: w=1.43 b=0.77, loss=1.77779
Epoch 16: w=1.48 b=0.60, loss=1.46278
Epoch 17: w=1.53 b=0.45, loss=1.20359
Epoch 18: w=1.57 b=0.32, loss=0.99032
Epoch 19: w=1.61 b=0.20, loss=0.81484
Epoch 20: w=1.65 b=0.08, loss=0.67046
Epoch 21: w=1.68 b=-0.02, loss=0.55166
Epoch 22: w=1.71 b=-0.11, loss=0.45391
Epoch 23: w=1.74 b=-0.19, loss=0.37348
Epoch 24: w=1.76 b=-0.27, loss=0.30730
Epoch 25: w=1.79 b=-0.33, loss=0.25285
Epoch 26: w=1.81 b=-0.40, loss=0.20805
Epoch 27: w=1.82 b=-0.45, loss=0.17118
Epoch 28: w=1.84 b=-0.50, loss=0.14085
Epoch 29: w=1.85 b=-0.55, loss=0.11589
Epoch 30: w=1.87 b=-0.59, loss=0.09536
Epoch 31: w=1.88 b=-0.63, loss=0.07846
Epoch 32: w=1.89 b=-0.66, loss=0.06456
Epoch 33: w=1.90 b=-0.69, loss=0.05312
Epoch 34: w=1.91 b=-0.72, loss=0.04371
Epoch 35: w=1.92 b=-0.75, loss=0.03596
Epoch 36: w=1.93 b=-0.77, loss=0.02959
Epoch 37: w=1.93 b=-0.79, loss=0.02435
Epoch 38: w=1.94 b=-0.81, loss=0.02003
Epoch 39: w=1.95 b=-0.83, loss=0.01648
Epoch 40: w=1.95 b=-0.85, loss=0.01356
Epoch 41: w=1.95 b=-0.86, loss=0.01116
Epoch 42: w=1.96 b=-0.87, loss=0.00918
Epoch 43: w=1.96 b=-0.88, loss=0.00756
Epoch 44: w=1.97 b=-0.90, loss=0.00622
Epoch 45: w=1.97 b=-0.91, loss=0.00511
Epoch 46: w=1.97 b=-0.91, loss=0.00421
Epoch 47: w=1.97 b=-0.92, loss=0.00346
Epoch 48: w=1.98 b=-0.93, loss=0.00285
Epoch 49: w=1.98 b=-0.94, loss=0.00234

Plot our trained values over time
```

```
# Plot the w-values and b-values for each training Epoch against the true values
```

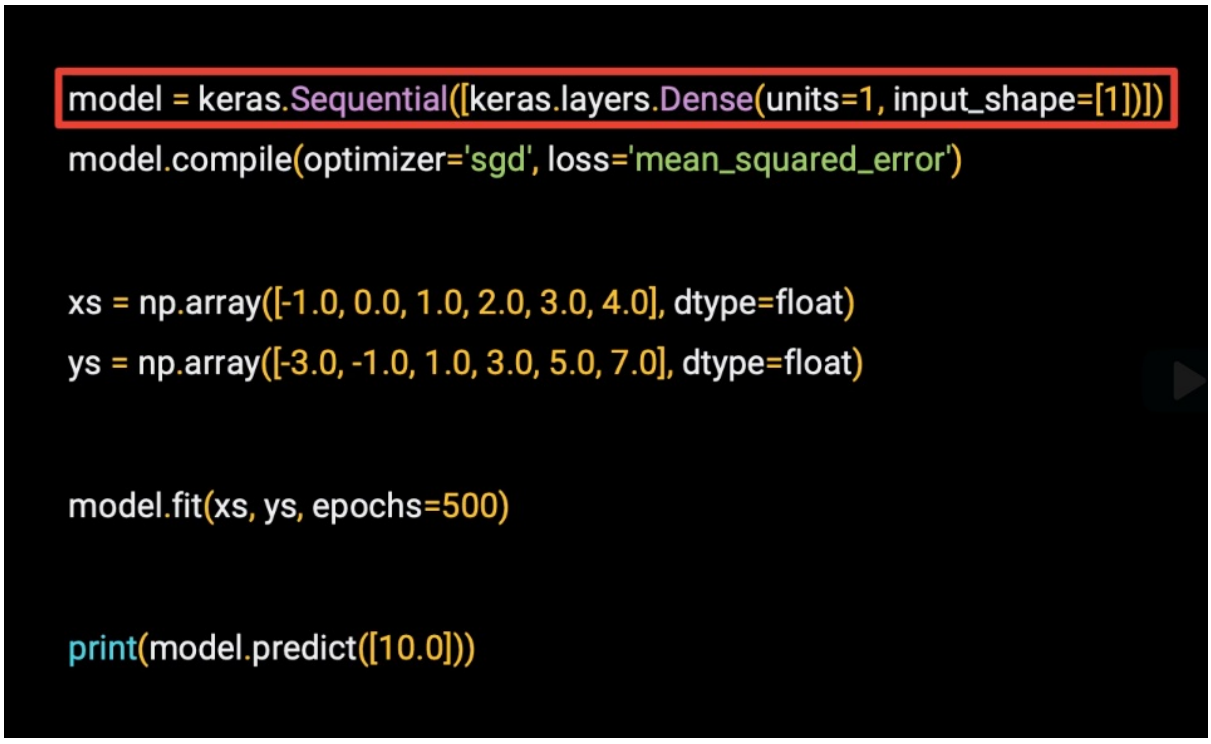
```
TRUE_w = 2.0
TRUE_b = -1.0
plt.plot(epochs, list_w, 'r', epochs, list_b, 'b')
plt.plot([TRUE_w] * len(epochs), 'r--', [TRUE_b] * len(epochs), 'b--')
plt.legend(['w', 'b', 'True w', 'True b'])
plt.show()
```



neural networks



sequential network



- we only have one entry, so we only have one layer.
- This layer is a dense, so that we know it's a densely connected network.
- The units parameter tells us how many neurons will be in the layer.
- tell at the input shape.*

- We're training a neural network on single x's to predict single y's.

compile

```
model = keras.Sequential([keras.layers.Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
```

- define the loss function and an optimizer.
- loss function is mean squared error, as we used previously.*
- The optimizer is SGD, which stands for Stochastic Gradient Descent.

fitting

```
model = keras.Sequential([keras.layers.Dense(units=1, input_shape=[1])])
model.compile(optimizer='sgd', loss='mean_squared_error')

xs = np.array([-1.0, 0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

model.fit(xs, ys, epochs=500)

print(model.predict([10.0]))
```

- We're fitting the x's to the y's.
- We'll do this for 500 epochs, where an epoch is each of the steps
  - make a guess,
  - measure the loss of the guess,
  - optimize, and continue.

predict

- use the model to predict the y for a given x.

test (lab3)

```
import tensorflow as tf
import numpy as np
from tensorflow import keras



# define a neural network with one neuron
# for more information on TF functions see: https://www.tensorflow.org/api_docs
model = tf.keras.Sequential([keras.layers.Dense(units=1, input_shape=[1])])

# use stochastic gradient descent for optimization and
# the mean squared error loss function
model.compile(optimizer='sgd', loss='mean_squared_error')

# define some training data (xs as inputs and ys as outputs)
xs = np.array([-1.0,  0.0, 1.0, 2.0, 3.0, 4.0], dtype=float)
ys = np.array([-3.0, -1.0, 1.0, 3.0, 5.0, 7.0], dtype=float)

# fit the model to the data (aka train the model)
model.fit(xs, ys, epochs=500)
```

```
print(model.predict([10.0]))

[[18.98506]]
```

neurons

In reality, what is referred to as a ‘neuron’ here is simply a function that has two learnable parameters, called a ‘weight’ and a ‘bias’, where, the output of the neuron will be:

Output = (Weight * Input) + Bias

When multiple neurons work together in layers, the learned weights and biases across these layers can then have the effect of letting the neural network learn more complex patterns.

layers type

- Neural Network that were densely connected to each other - Dense layer type
- Convolutional layers contain filters that can be used to transform data. The values of these filters will be learned in the same way as the parameters in the Dense neuron you saw here. Thus, a network containing them can learn how to transform data effectively. This is especially useful in Computer Vision.
- Recurrent layers learn about the relationships between pieces of data in a sequence. There are many types of recurrent layer, with a popular one called LSTM (Long, Short Term Memory), being particularly effective. Recurrent layers are useful for predicting sequence data (like the weather), or understanding text.

more ???

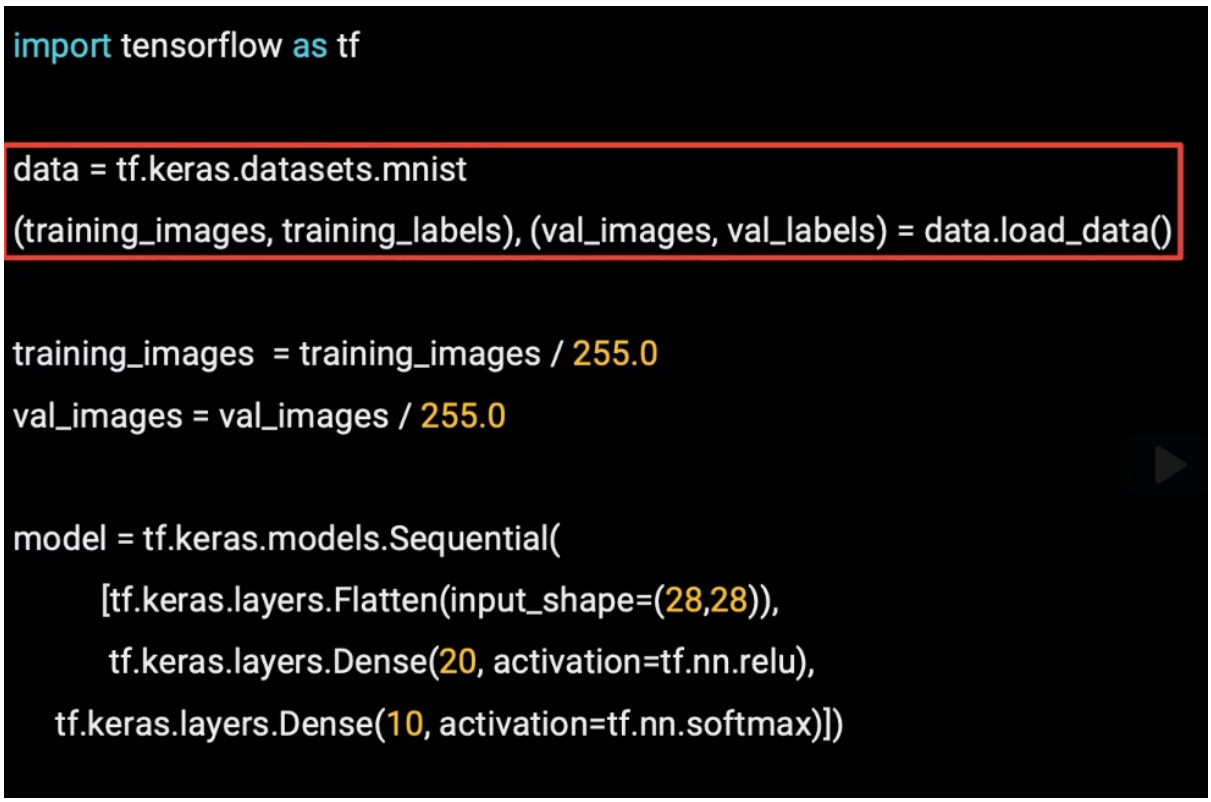
```
layer types that don’t learn parameters themselves, but which can affect the other layers. These include layers
 like dropouts, which are used to reduce the density of connection between dense layers to make them more effic
ient, pooling which can be used to reduce the amount of data flowing through the network to remove unnecessary
information, and lambda lambda layers that allow you to execute arbitrary code
```

building blocks

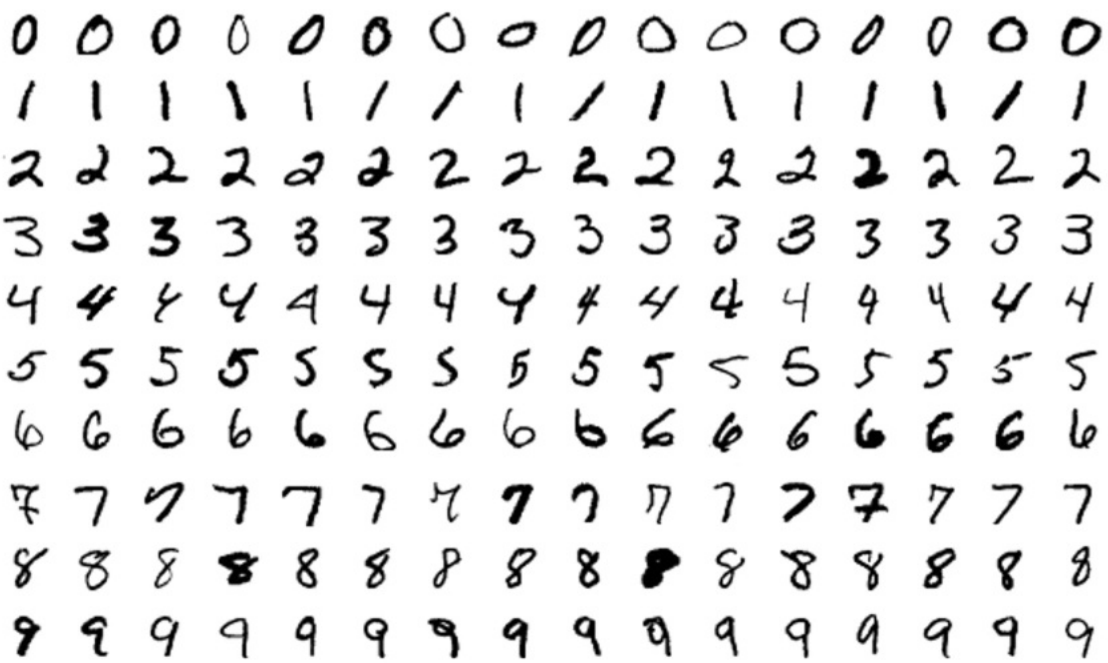
- classification
- validation

classification

example - figure classification

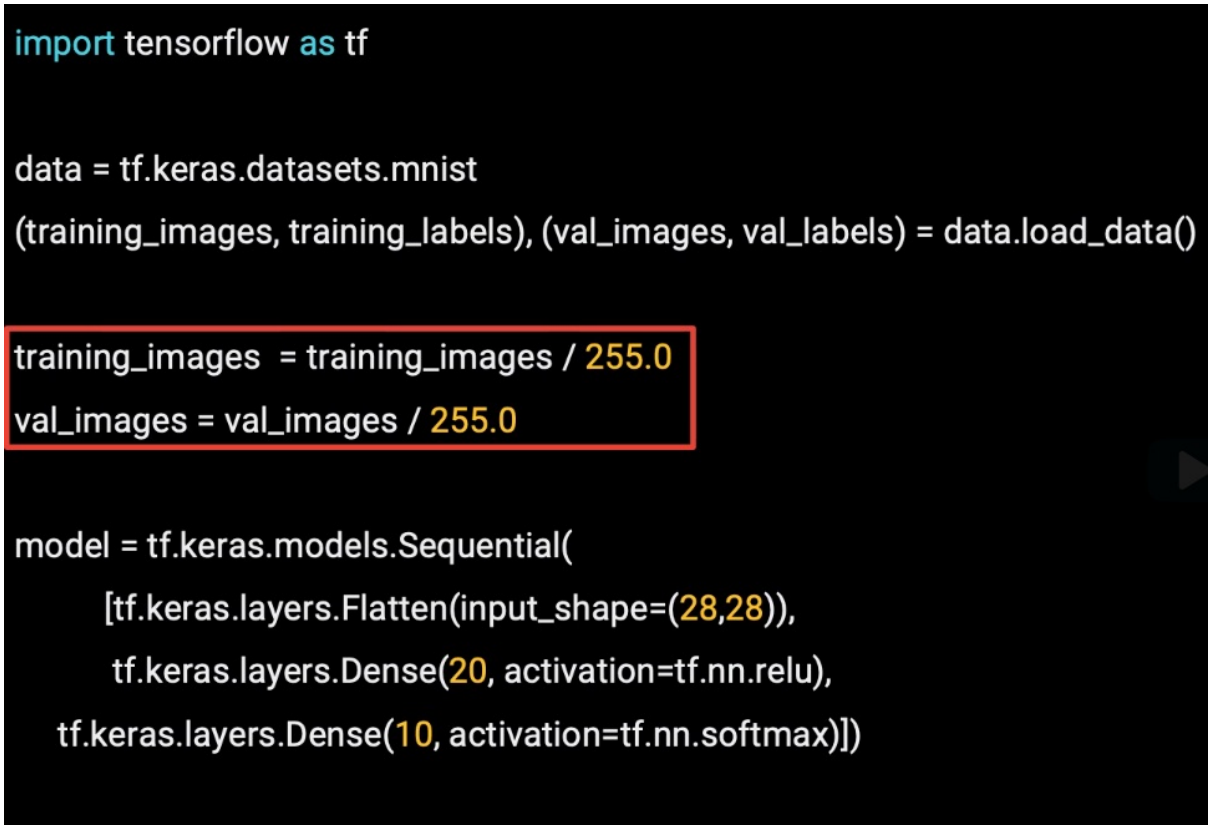


data set



60,000 Labelled Training Examples
10.000 Labelled Validation Examples

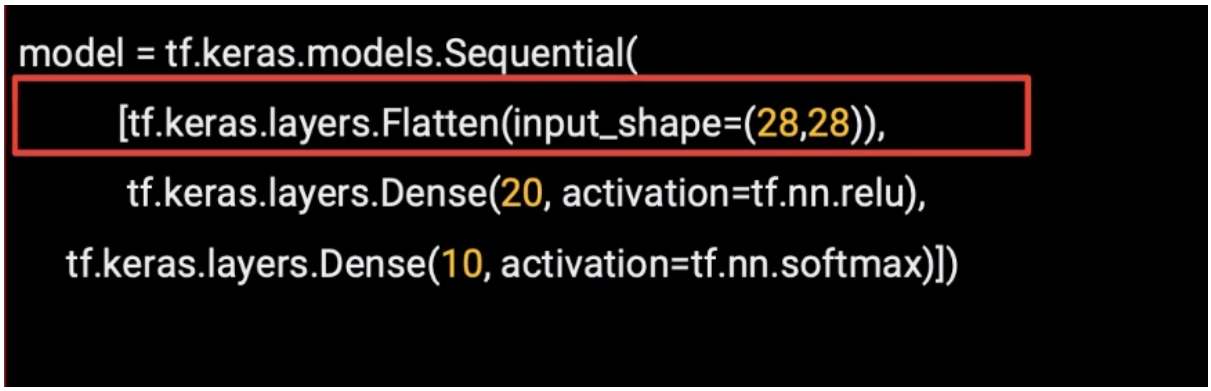
The handwriting digits are 28 by 28 monochrome, which means each pixel value is from 0 to 255. We want to normalize these to use them in a neural network. And it behaves much better if the values are between 0 and 1. So we can just divide the image values by 255.



network

- Sequential can be used to define a neural network by listing the layers of that neural network.

Our images are 28 by 28 in dimension. So the first thing that we can do in our sequential is to define an operation to flatten these out, so that we can feed them into the neural network. we'll define a layer of neurons And we have 20 neurons in this layer (This number is purely arbitrary for now)



The next layer has 20 neurons, so we need our images to be in the same dimension as that layer, considered to be 20 by 1, so our images will also need to be something by 1, in order to fit. The easiest way to do this is to flatten the 28 by 28, which is 784 pixels, into a 784 by 1. And that's what a flattened layer does.

```
model = tf.keras.models.Sequential(
        [tf.keras.layers.Flatten(input_shape=(28,28)),
         tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

activation function

```
model = tf.keras.models.Sequential(
        [tf.keras.layers.Flatten(input_shape=(28,28)),
         tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

It's called ReLU, which stands for rectified linear unit. every neuron will call its activation function when that layer is used. The ReLU activation function changes any output that is less than 0 to 0.

```
model = tf.keras.models.Sequential(
        [tf.keras.layers.Flatten(input_shape=(28,28)),
         tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

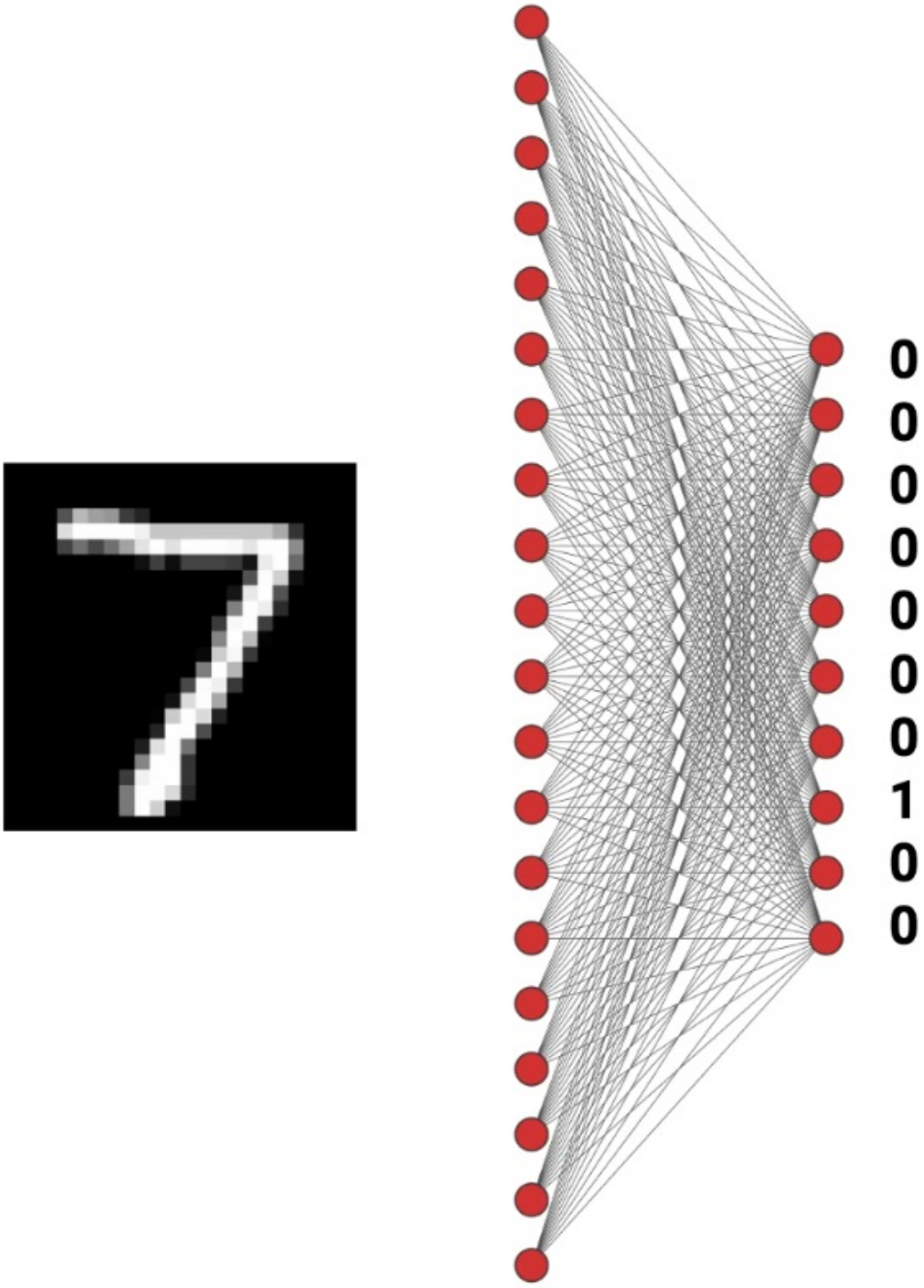
this will find the neuron from amongst the 10 that has the highest value.

final layer

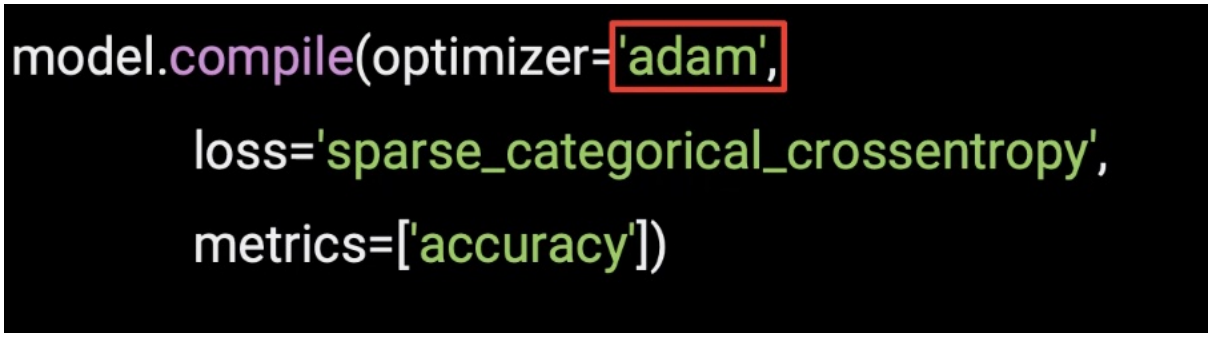
The final layer will have 10 neurons in it because we have 10 classes. And these are the digits 0 through 9.

```
model = tf.keras.models.Sequential(
        [tf.keras.layers.Flatten(input_shape=(28,28)),
         tf.keras.layers.Dense(20, activation=tf.nn.relu),
    tf.keras.layers.Dense(10, activation=tf.nn.softmax)])
```

network



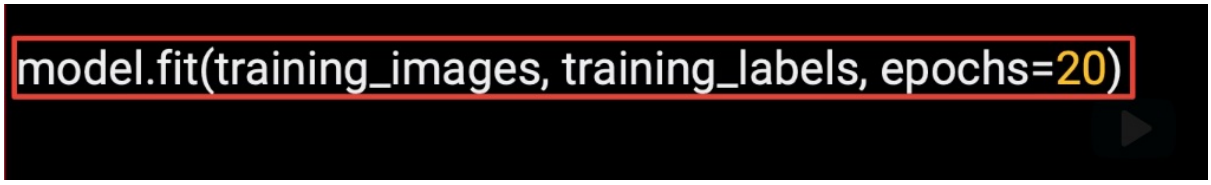
compile



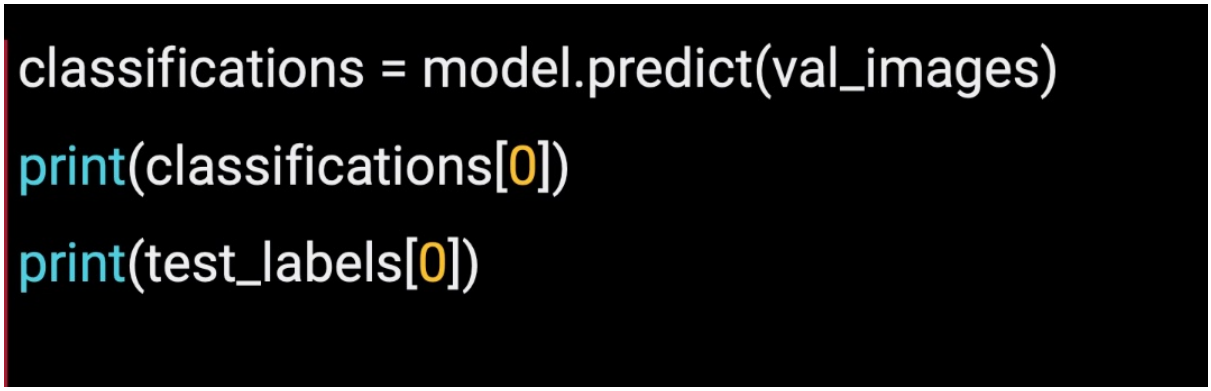
- optimizer - Adam can actually vary its learning rate helping us to converge more quickly.
- for classification, as we have a number of categories instead of a single value. And we need the loss across all the categories. This means that you'll generally use a categorical loss function.

train

we can train for just 20 epochs.



predict



running out on the entire validation set of 10,000, printing out the classification for the first one, and comparing it against the actual label.

validation

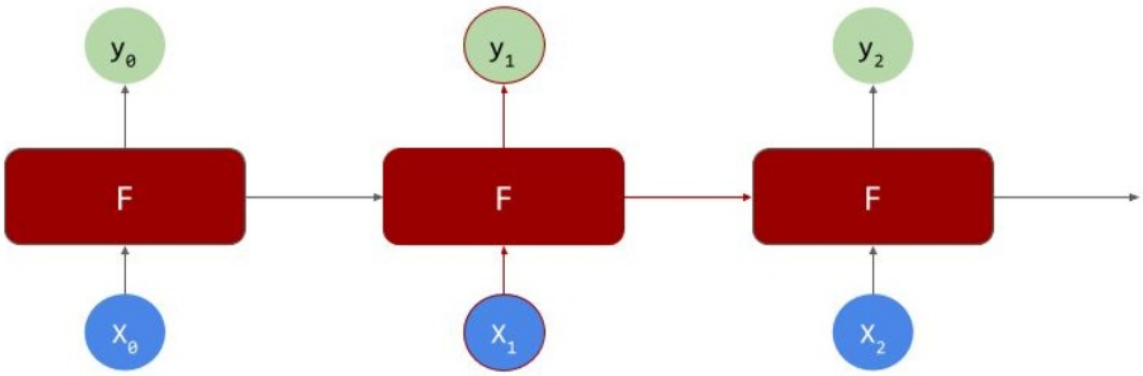

model.fit(training_images, training_labels,
validation_data=(val_images, val_labels),
epochs=20)
Epoch 14/20
1875/1875 [==========] - 4s 2ms/step - loss: 0.0960 - accuracy: 0.9713 - val_loss: 0.1410 - val_accuracy: 0.9588
Epoch 15/20
1875/1875 [==========] - 4s 2ms/step - loss: 0.0933 - accuracy: 0.9722 - val_loss: 0.1353 - val_accuracy: 0.9626
Epoch 16/20
1875/1875 [==========] - 4s 2ms/step - loss: 0.0910 - accuracy: 0.9729 - val_loss: 0.1356 - val_accuracy: 0.9622
Epoch 17/20
1875/1875 [==========] - 4s 2ms/step - loss: 0.0880 - accuracy: 0.9734 - val_loss: 0.1339 - val_accuracy: 0.9625
Epoch 18/20
1875/1875 [==========] - 4s 2ms/step - loss: 0.0855 - accuracy: 0.9740 - val_loss: 0.1349 - val_accuracy: 0.9619
Epoch 19/20
1875/1875 [==========] - 4s 2ms/step - loss: 0.0832 - accuracy: 0.9754 - val_loss: 0.1341 - val_accuracy: 0.9624
Epoch 20/20
1875/1875 [==========] - 4s 2ms/step - loss: 0.0808 - accuracy: 0.9759 - val_loss: 0.1340 - val_accuracy: 0.9626

about

- recurrentLayers
- Long Short Term Memory
- convolutions
- cnnPlay1
- cnnPlay2

Recurrent layers

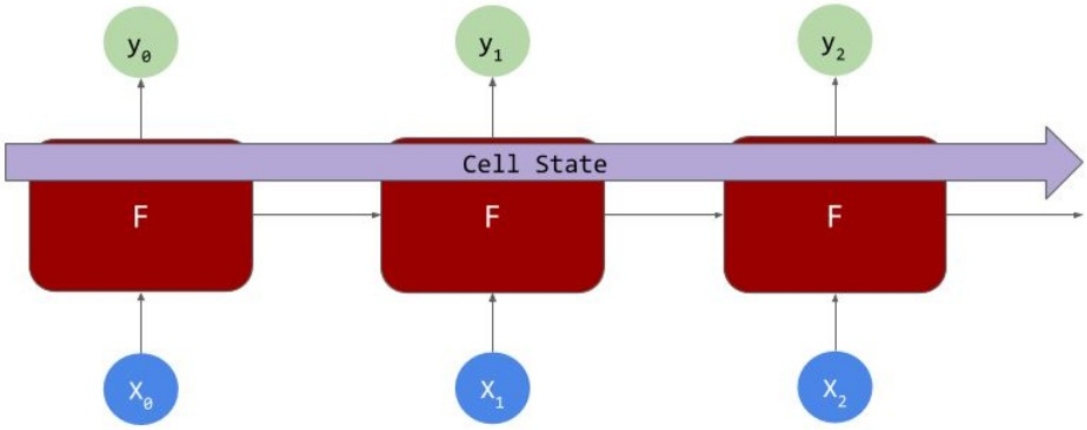
are able to learn values in a sequence where each neuron learns values and passes those values to another neuron in the same layer. The basic idea can be represented like this:



Where, if we have a sequence of values, X0, X1, X2 that we want to learn a corresponding sequence Y0, Y1, Y2 for, then the idea is the the neurons (F) can not only try to match X0->Y0, but also pass values along the sequence, indicated by the right pointing arrows, so the input to the second neuron is the output from the first and X1, from which it can learn the parameters for Y1 and so on.

Long Short Term Memory

not only are values passed from neuron to neuron, so that X0 can impact Y1, X1 can impact Y2, but values from further back can also have an impact -- so that X0 could impact Y99 for example. This is achieved using a data structure called a Cell State where context can be preserved across multiple neurons.

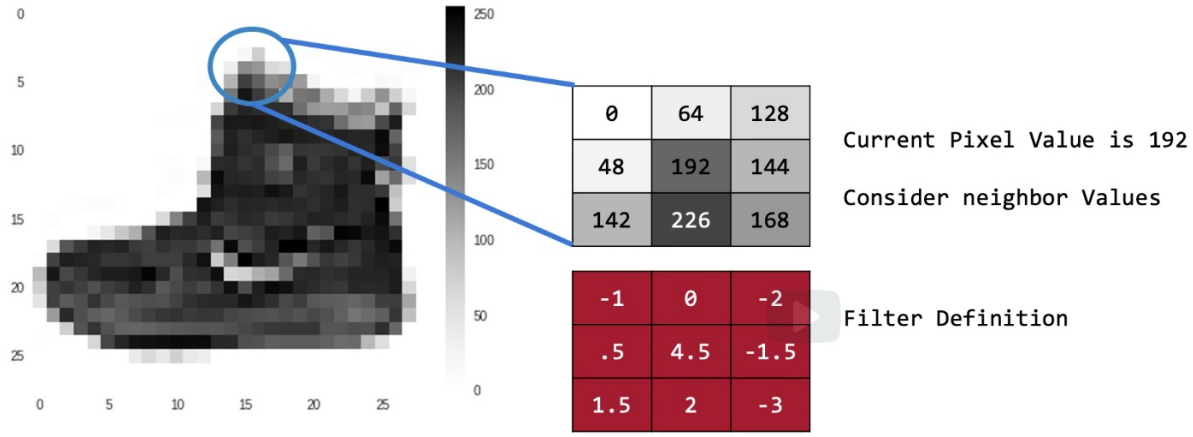


convolutions

A convolution is a filter that passes over an image, processing it, and extracting features that show a commonolatity in the image.

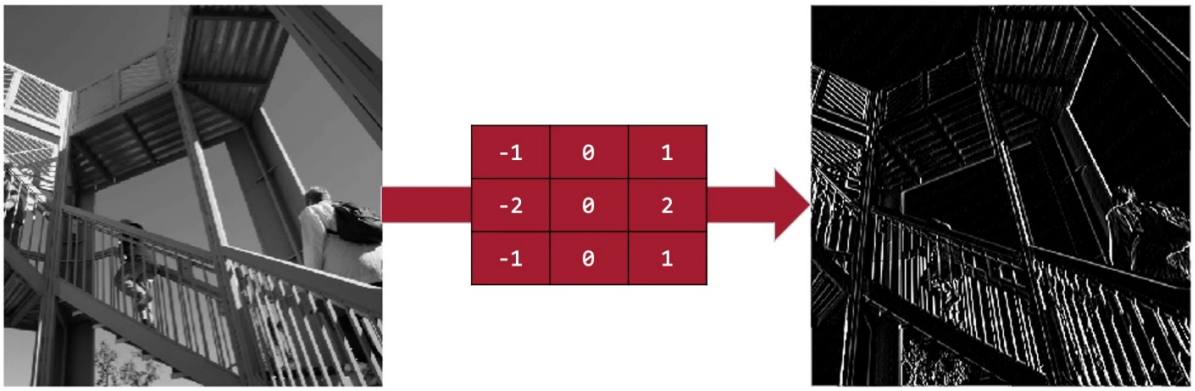
- see lab8: lab8_ExploringConvolutions.ipynb

filters




```
CURRENT_PIXEL_VALUE = 192
NEW_PIXEL_VALUE = (-1 * 0) + (0 * 64) + (-2 * 128) +
                  (.5 * 48) + (4.5 * 192) + (-1.5 * 144) +
                  (1.5 * 42) + (2 * 226) + (-3 * 168)
```

for example filter as horizontal line detector



By applying filters like these, you can remove almost everything but a distinguishable feature. And this process is called feature extraction.

poling

Pooling is simply the process of removing pixels while maintaining important information.

From each of these blocks, we pick the biggest value and we throw the rest away. We then reassemble these, and we have a new set of four pixels.

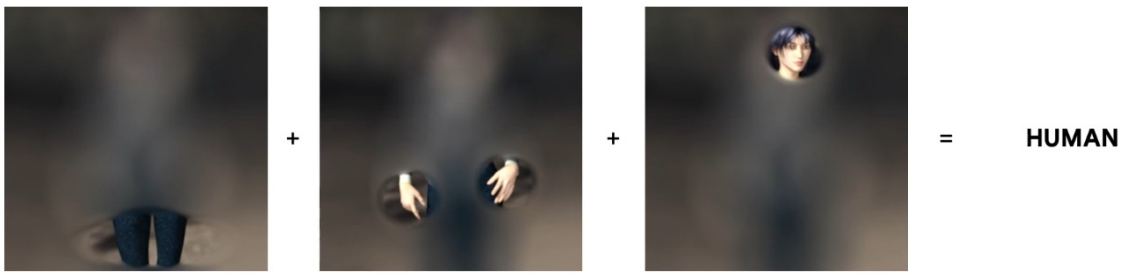
if we apply it to an image after filtering the image. It can have the effect of enhancing the features that we extracted.

compress

if you're applying many filters to your image in a layer, you're in effect making many copies of your image. So if it's a large image, you end up with a lot of data flowing through your network. So it's good to have a way to compress that data without losing the important features.

multi layer

When you apply multiple layers of filters, then really complex features, such as faces or hands instead of the vertical or horizontal lines I've shown here, could be spotted and extracted.



play1 cnn

Convolutional Neural Networks (CNNs) play test

```
import tensorflow as tf
mnist = tf.keras.datasets.fashion_mnist
(training_images, training_labels), (val_images, val_labels) = mnist.load_data()
training_images=training_images.reshape(60000, 28, 28, 1)
training_images=training_images / 255.0
val_images=val_images.reshape(10000, 28, 28, 1)
val_images=val_images/255.0
`
```

- (1) a convolution of layer expects the image dimensions to be fully defined. So a color image will have three channels for red, green, and blue. So it will be something by something by 3. But a monochrome image only has one channel.
- (2) output layer will be 10 neurons as before. It's because we have 10 classes.
- (3) flatten before we get into the DNN.This will occur after the filters have done their job.
- (4) define the convolutional neural network: 2D, because the images have a width and height. And we'll pass filters across them to change the image to extract features.These filter values will be learned over time!
- (5) Pooling that we described previously is implemented by specifying max pooling as a layer.
  - There are different types of Pooling.There's max, min and average.
  - we'll use max pooling, where we take the largest value from the pool.
- (6) this is the number of filters to apply to the image at this layer. the filter values are going to be randomly initialized as our Guess.And then, they will be learned over time.
- (7) the size of the filter. we choose filters that we had defined a 3 by 3 filter that multiplied across the current pixel and each of its neighbors.
- (8) the size of the Pool is specified as a parameter, which in this case is a 2 by 2 pool. So we'll be picking one pixel out of four.

by Pooling, we're also reducing the size of our image,helping us to emphasize and summarize the features that we extracted. The convolutional layer had 64 filters in it, which means 64 copies of the image are going to be made. So reducing the size of them will reduce the amount of data flowing through our network.

```
model = tf.keras.models.Sequential([
                  //4////6////7////                    ///// 1 ////////////////
  tf.keras.layers.Conv2D(64, (3,3), activation='relu', input_shape=(28, 28, 1)),

                  ////5/////////8////
  tf.keras.layers.MaxPooling2D(2, 2),

                  ///4///6////7///
  tf.keras.layers.Conv2D(64, (3,3), activation='relu'),

                  ///5/////////8////
  tf.keras.layers.MaxPooling2D(2,2),


  /////////////// 3 ////////////
  tf.keras.layers.Flatten(),
  tf.keras.layers.Dense(20, activation='relu'),

                    //// 2 ////////
  tf.keras.layers.Dense(10, activation='softmax')
])
```

```
`
```

- see lab 9 lab9_Fashion_MNIST_Convolutions.ipynb
- to see what the model sees : lab10_Fashion_MNIST_sees.ipynb

play2

- dataset: CIFAR-10 ; design your own model and then select your optimizer and loss function.
- the dataset: https://www.cs.toronto.edu/~kriz/cifar.html
- CIFAR are 32x32 color images (3 color channels)
- CIFAR ones can have the object with a background -- for example airplanes might have a cloudy sky behind them!

layers

- your model may want to learn some high level features and then classify them.

```
FIRST_LAYER = layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3))

HIDDEN_LAYER_TYPE_1 = layers.MaxPooling2D((2, 2))

HIDDEN_LAYER_TYPE_2 = layers.Conv2D(64, (3, 3), activation='relu')

HIDDEN_LAYER_TYPE_3 = layers.MaxPooling2D((2, 2))

HIDDEN_LAYER_TYPE_4 = layers.Conv2D(64, (3, 3), activation='relu')

HIDDEN_LAYER_TYPE_5 = layers.Dense(64, activation='relu')

LAST_LAYER = layers.Dense(10)
```

loss function

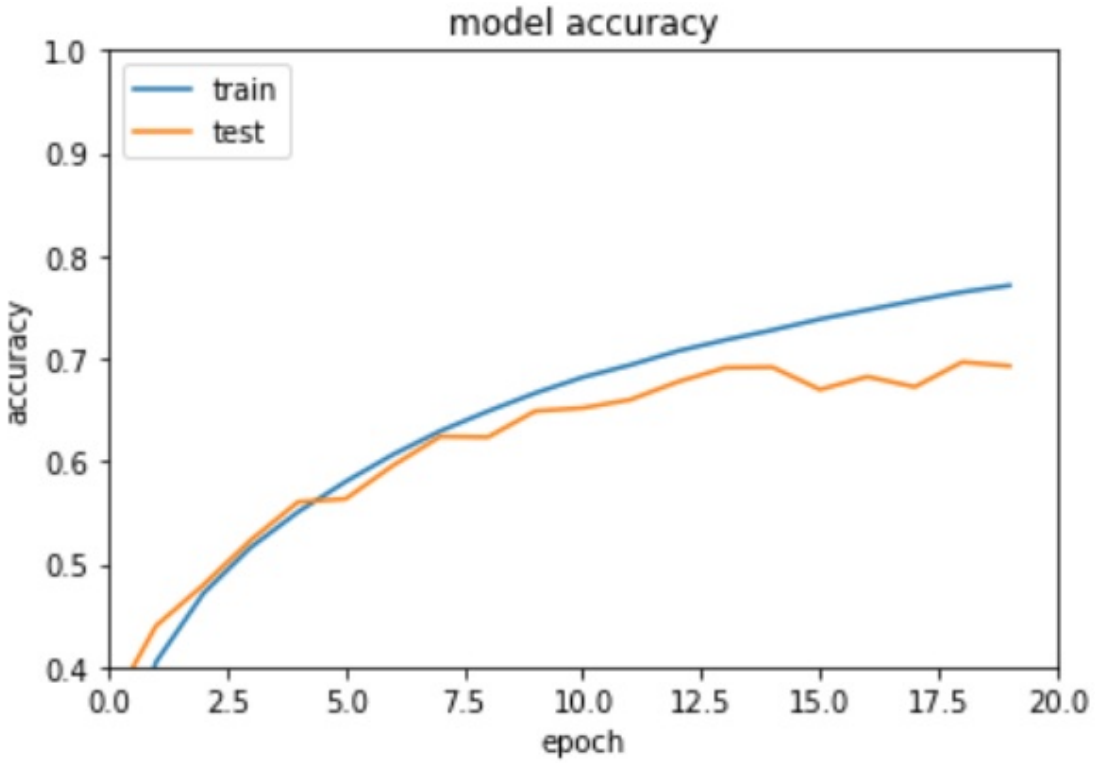
- for the loss we want something that can help optimize for differences across categories

```
LOSS = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)

# Compile the model
model.compile(optimizer='sgd',
              loss=LOSS,
              metrics=['accuracy'])
```

optimizer

for:

```
model.compile(optimizer='sgd',
==============================

1563/1563 [==============================] - 11s 7ms/step - loss: 0.6585 - accuracy: 0.7712 - val_loss: 0.9173
- val_accuracy: 0.6928
```



- pick a better optimizer : optimizer with an adaptive learning rate often performs quite well

```
OPTIMIZER = 'adam'
======================
1563/1563 [==============================] - 12s 8ms/step - loss: 0.1525 - accuracy: 0.9470 - val_loss: 1.7749
- val_accuracy: 0.6943
```