Deep Learning And Computer Vision with CNNs

By Dan Howarth & Ajit Jaokar

Published by Data Science Central

Deep Learning and Computer Vision with CNNs

By
Dan Howarth
and
Ajit Jaokar

Published by Data Science Central

https://www.datasciencecentral.com/

October 2019

Contents

Links to Notebooks

1. Notebook 1 – Deeplearning with TensorFlow

2. Notebook 2 – Computer Vision with CNNs

Part 1

TensorFlow 2.0:
Notebook 1: 'Hello World'

Deep Learning
with TensorFlow 2.0

1. Introduction to the Notebooks

1.1 What we will cover today

- We will: provide an introduction to the core Deep Learning Concepts; provide an introduction to TensorFlow 2.0; and, provide an introduction to Computer Vision.

How will we go about it?

- We will have four sessions today. The first will provide a basic introduction to Deep Learning and TensorFlow 2.0. The second will go in to more detail about Computer Vision.
- Session Three will cover Transfer Learning, an important technique for developing state of the art models. Session Four will be an opportunity to put together everything you have learned and develop your own models.

How are the sessions structured?

- We will provide some introductory concepts and instructions at the start of session. You will then work through a notebook that will cover the topics for that session. You can do this individually or in conjunction with others around you. We will answer questions and support throughout the session. At the end of the session, we will summarize what we have learnt.
- In addition, we are preparing some advanced topic notebooks that you can work through following today's session. They will build on the initial notebooks and provide an insight into the lower level TensorFlow API.

Can you set out all the notebooks?

- Session 1: 'Hello World' Deep Learning with TensorFlow 2.0 (this notebook)
- Session 2: Computer Vision with CNNs
- Session 3: Transfer Learning

And the Advanced Notebooks?

- Advanced 1: Model and Layers
- Advanced 2: Custom Training Loops
- Advanced 3: Data Pipelines and Augmentation
- Advanced 4: Tensors

What will you not cover?

- There are certain things that we can't cover today because of time. We won't cover the math behind Deep Learning (calculus and Linear Algebra). There won't be detailed coverage of the topics, although we will cover the main concepts and provide some follow-up reading.

- And, we won't have exciting datasets. We will use benchmark (but slightly dated) datasets that are available via the TensorFlow 2.0 API. Our focus is on concepts and code, and this means using datasets that are available to everyone and can be trained on easily.

One final thing...

- The Session 1-3 tutorials are based on tutorials published on the TensorFlow 2.0 website link. We have provided a lot more material than is in those tutorials, and the advanced tutorials are new.

2. Introduction to this Notebook

What will we cover in this notebook?

- This notebook will introduce the core concepts of Deep Learning. We will also start coding straightaway with TensorFlow 2.0.
- Let's start by loading the necessary libraries, and introducting the problem we are going to work on.

2.1 Loading the Libraries

```
[ ]  # we need to install tensorflow 2.0 on the
     google cloud notebook we have opened
     !pip install -q tensorflow==2.0.0-alpha0
```

```
     |████████████████████████████████| 79.9MB 1.2MB/s
     |████████████████████████████████| 419kB 51.8MB/s
     |████████████████████████████████| 3.0MB 42.3MB/s
```

```
[ ]  # imports future functionality that might
     modify modules otherwise used and make them
     incompatible in the future.
```

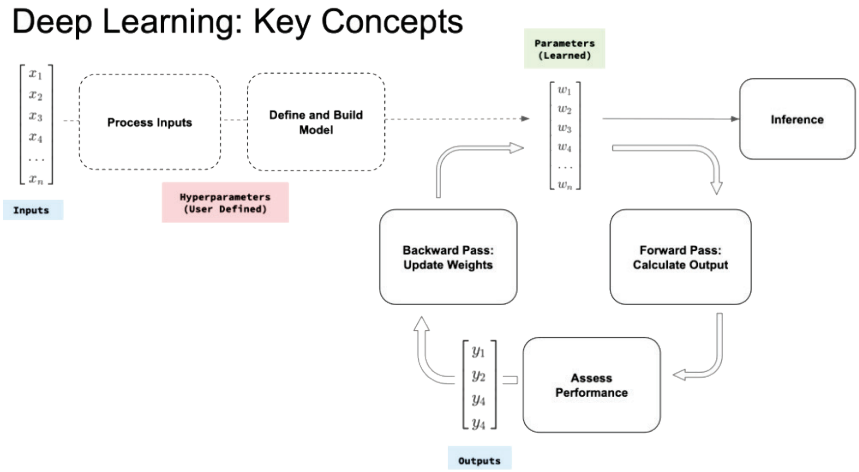
```
    # We are future proofing by importing modules
    that modify or replace existing modules that we
    may have used now
    from __future__ import absolute_import,
    division, print_function, unicode_literals
[ ] # import tensorflow and tf.keras
    import tensorflow as tf
    from tensorflow import keras
[ ] # import helper libraries
    import numpy as np
    import matplotlib.pyplot as plt
[ ] # let's print out the version we are using
    print(tf.__version__)
2.0.0-alpha
```

2.2 Introduction to our problem

What problem are we trying to solve?

- We will use the Fashion MNIST dataset. This is a dataset of images of clothes (you will see what the data looks like soon). The task is to train a model on this dataset so that when the model sees a new image of clothes, it classifies it correctly.
- The dataset is quite large – 60,000 images to train on and 10,000 to test on. However, the images are small and therefore its possible for us to train on easily.

3. Deep Learning Conceptual Introduction



What are the main concepts in Deep Learning?

- The diagram above shows what I think are the core concepts of Deep Learning (in a supervised learning context). We will assume we are using a training set where we know the matching input and output values.
- We have a dataset that is an input to the deep learning model. We need to define how we will process input data, if at all.
- We will define how we build our model. These user-defined concepts are our hyperparameters (things that the deep learning practitioner sets). They create the total number of parameters that will be trained to map our input data to output values. Parameters are things learned by the model – they represent the learning in deep learning. In the case of deep learning, the parameters and weights and biases of the model.
- We will then use our model to train on the data. We will pass our data through the model in a forward pass; this will pro-

vide an output value by performing mathematical operations at each of stage of our model.

- We will assess the performance of our model by comparing this output value with the actual value to generate a loss value. The loss will represent the difference between the model's performance and the actual dataset. We will also measure the performance with other metrics
- We will then perform a backward pass, where we use the loss value to update our parameters using an optimizer. This optimizer performs a version of backpropagation and the parameters, so that their contribution to the overall loss is identified and corrected to some extent during each training loop.
- This training loop is repeated until the parameters are sufficiently updated that they are able to accurately map the input data to the output values.

Is that it?

- There are other facets to deep learning and it's hard to keep things at a high level and not delve in to the details.
- Throughout the notebooks, we will update the chart above with more detail as we learn it. This should hopefully start to build knowledge around the key concepts, and let you see how new things fit in to an overall framework.

Does TensorFlow 2.0 cover all these areas?

- Yes. Once we have learnt a new piece of code, we will update the chart above to see where that bit of code fits. Again, hopefully this will help you learn how the code implements the concepts more easily.

TODO: Update chart. Add in additional ML words

4. Data

What do we need to think about with regard to our data?

- We need to understand what our input and output data is. Given the problem we are trying to solve, does the data help us?
- For this notebook, we are classifying images. This means that our input data are a series of images and our output data are the classes we want to use to classify images.
- We are using a well-known dataset so can expect the data to be complete and not corrupted, but you might need to check this when you are using different datasets.

What about processing inputs, the first conceptual block above?

- Our aim is to ensure the data is a suitable state to model, and that the data is loaded into the training loop in a way that maximises learning.
- There are a few things we might do to ensure that our data is processed correctly. For now, we will begin by understanding the size and shape of the data, and by rescaling it so that it can be modelled effectively. We will also split our data into training and test sets.
- Let's start by going through some code to load the data.

```
[ ]  # using a preloaded dataset
     fashion_mnist = keras.datasets.fashion_mnist
[ ]  # load_data() is used to load a keras_dataset
     # it returns two sets of tuples that provides
     data set arrays and labels, one for training
     and one for testing data
     (train_images, train_labels), (test_images,
     test_labels) = fashion_mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/train-labels-idx1-ubyte.gz
```

```
32768/29515 [================================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/train-images-idx3-ubyte.gz
26427392/26421880 [==============================] - 0s 0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/t10k-labels-idx1-ubyte.gz
8192/5148 [===============================================] - 0s
0us/step
Downloading data from https://storage.googleapis.com/tensorflow/tf-
keras-datasets/t10k-images-idx3-ubyte.gz
4423680/4422102 [==============================] - 0s 0us/step
```

What did we just do and why?

- We loaded and split our data into training and test sets. Each set has our images and a corresponding array of labels.
- In machine learning, it is good practice to have both a training and a testing set.
- We will train our model on the training set, meaning we will compare the predicts our model makes against known results to update the model parameters. This will help improve the model.
- At the end of training, we will use our model to predict results on our test data. This is input data that the model has not been trained on, and the corresponding output labels that the model doesn't see. We will then compare the results of our model with the actual labels.
- The ability of the model to predict accurately using unseen data is the benchmark that determines how effective the model.

What are our inputs and outputs?

- In this instance, we returned four numpy arrays:
  - train_images
  - train_labels
  - test_images
  - test_labels

- Now that we have loaded and split our data, we can explore the size and shape of the dataset, and preprocess it as required.

TODO: Diagram showing training and testing split

```
[ ]  # lets start by looking at the size of the
     train and test sets
     # lets get the shape
     train_images.shape
(60000, 28, 28)
[ ]  # get the same info for our test set
     test_images.shape
(10000, 28, 28)
```

What does this show?

- We have 60,000 images in our train set, and 10000 images in our test set. It is common practice to have significantly more training than testing images.
- We can also see that the shape of the data for each image is 28 x 28. This is 28 rows by 28 columns. Compared to other image data you might model in the future, this is small and is why this is a good dataset for a tutorial.
- If the training and testing image sizes were different, we would need to get them to the same size to pass in to the model.
- When we create our model we will need to pass the shape information to the model's first layer.

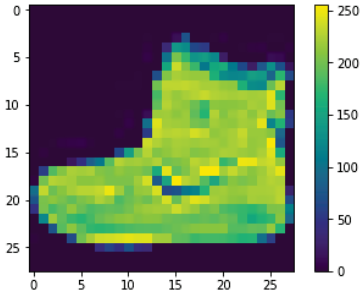
```
[ ]  ## let's look at our training labels
     # we can see that there are 10 labels
     np.unique(train_labels)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=uint8)
[ ]  est labels correspond to the size of the train
     and test sets
     len(train_labels)
```

```
    # and that the train and test labels correspond
    to the size of the train and test sets
    len(train_labels)
60000
[ ] test_labels.shape
(10000,)
[ ] ## we can see that our labels are just numbers.
    We need to match them to description of the
    image
    # create a list of the labels
    class_names = ['T-shirt/top', 'Trouser',
    'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt',
    'Sneaker', 'Bag', 'Ankle boot']
```

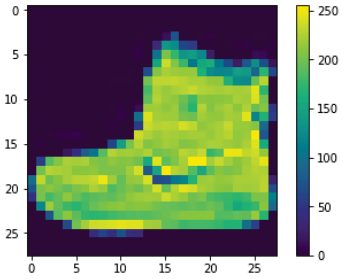
We also need to preprocess the images by rescaling them. Why?

- As you will see below, all of our image arrays are between the values of 0 and 255, with each value corresponding to a colour. We need to rescale them so that they are all between 0 and 1.
- The high level reason for this is that this helps with training the model. The model will update its parameters more effectively if all the input values are on the same scale, and in a defined range between 0 and 1.

```
[ ] ## lets look at an image and use matplotlib to
    plot the array values
    # converts the numpy array to an image and
    displays it
    plt.imshow(train_images[0])
    # displays the image values
    plt.colorbar()
    # displays the chart only - comment out to see
    what info it omits
    plt.show()
```




```
[ ]  # lets preprocess
     train_images = train_images / 255.0
     # we need to do the same to the test and
     training set
     test_images = test_images / 255.0
[ ]  # lets look again at the pixel range
     plt.imshow(train_images[0])
     plt.colorbar()
     plt.show()
```




```
[ ]  # now we have the class names, lets look at a
     selection of the images
     # sets the size of the overall display for our
     images
     plt.figure(figsize=(10,10))
     # loops through the first 25 images
     for i in range (25):
     # sets a location for each of the images
     plt.subplot(5,5,i+1)
     # removes the axis lables
     plt.xticks([])
```

```
plt.yticks([])
plt.grid(False)
# displays the image
plt.imshow(train_images[i], cmap=plt.cm.binary)
# plots the label, mapping the label to our
list of clothing items
plt.xlabel(class_names[train_labels[i]])
# displays the image and label
plt.show()
```

So, what did we cover in this section?

- We looked at understanding and processing our data.
- Specifically, we looked at data size and shape, rescaling our data, and splitting our data into training and test sets.

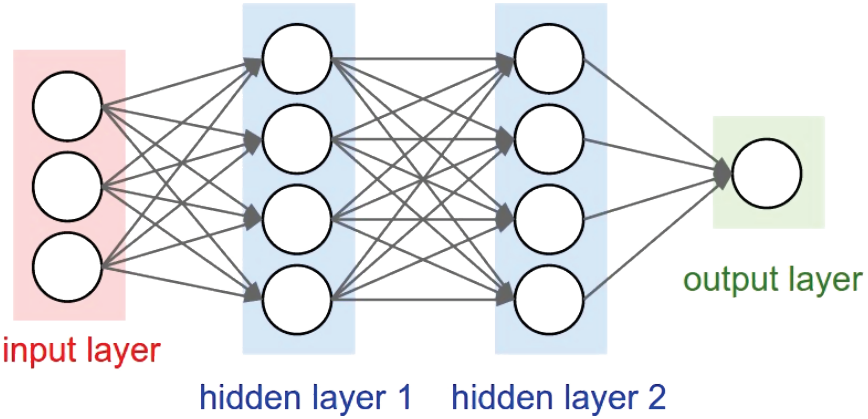
How does it add to our existing knowledge?

- This starts to add to the high level concepts we introduced in section 3.

What else can I learn to improve my knowledge?

- We touch briefly on how each array value represents a colour value. We will look at this in more detail in the next notebook.
- There is more to learn on splitting training and tests. In later notebooks, we will add a validation set.

5. Model



What is a deep learning model?

- The image above (credit) sets out the main components of a deep learning model:
  - an input layer that takes our data
  - an output layer that returns a value
  - hidden layers that generate the parameters that will learn the mapping between inputs and outputs. * The hidden layers provide the depth to the model.
  - the arrows, which represent the connections between the layers. Each arrow shows how a value from one layer will be passed to another. As the value is passed from one

layer to another, it is multiplied by a weight, which is a learned parameter, to return a different value.
    - nodes in each of the layers that represent the activation functions within the model. An activation function is where the product of all the inputs and weights into the node are summed, along with a bias term (biases are also a learned parameter) and where an output value is returned dependent on the sort of activation function we choose.
- We configure the model by setting the number of layers and defining what each layers does prior to training the model on our data.

How do we build Deep Learning Models?

- In order to build a deep learning model, we can define the following hyperparameters: the number of layers, the size of layer, the type of layer, and the type of activation function in each layer.

So layers are important?

- The building block of a neural network is the layer. Layers make up models as we can see from the diagram above.
- Layers extract representations from the data fed into them (taken from 'first CNN tutorial'). They contain the parameters (weights and biases) that our model learns to make predictions.
- Given that the layer parameters are learned during train, our task in building a deep learning model is to ensure there is sufficient capacity in the model to learn the representations required to map between the input and output data.
- This capacity is provided by the number of layers and the size of each layer (the number of parameters in each layer).

- The larger the capacity, the more representations we can have. But we need to trade off the number of parameters with the dangers of overfitting, or developing a model that is too specific to the training data and doesn't perform well on unseen data. (We will look at overfitting later)

What about layer type?

- Layers perform a set of mathematical operations on the data. The sort of operations change depending on what representations we want and define with the code. We will use dense, or fully connected, layers in this tutorial, and convolutional layers in the following tutorials.
- These different layers are best suited to different tasks within the mode, as we will see in these notebooks.

So it's just layers?

- No, we also define activation functions. These take input values (which will be the product of the output values of the previous layer and the weight parameter), sum them together and produce an output value. The value returned will depend on the type of activation function selected.
- The output layer will usually have a different activation function to the rest of the model, one that will be based on the required output of the model.

That's a lot to take in...

- It is. But these tutorials will set out some good guidelines. We will learn that:
  - Convolutional layers are good for extracting representations from image data, while dense layers are better suited for classifying those representation by mapping them to the outputs.

- An effective activation function for the dense hidden layers is Relu.
- An effective activation function for the output layer is Softmax.

How do we build a model in TensorFlow 2.0?

- There are a number of ways to build models in TensorFlow 2.0.
- Most straightforward is using the sequential API. We will use this to develop our models.
- We can also use the functional API and a technique known as sub-classing. We will explore these in the advanced notebooks as they offer benefits over the sequential API if you want to build certain sorts of model.
- We will develop out first model below. As you will see – and this holds true for a lot of deep learning – the code required to implement complex ideas that require a lot of explanation is pretty small.

```
[ ]  # here we instantiate a Sequential model
     # note that are passing in the layers as a list
     model = keras.Sequential([
        # the input / flatten layer changes the
        input shape to a 1D array of 28x28 size
        keras.layers.Flatten(input_shape=(28,28)),
        # here we define the hidden / Dense layer.
        We specify the number of nodes - 128 - and
        the activatin function - relu
        keras.layers.Dense(128, activation='relu'),
        # we define the output / Dense layer with 10
        nodes for 10 classes, and a softmax to
        return an array of 10 probability scores
        that sum to 1
        keras.layers.Dense(10, activation='softmax')
        ])
[ ]  # ability to print summary
```

```
model.summary()
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 128)               100480
_________________________________________________________________
dense_1 (Dense)              (None, 10)                1290
=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

How do we arrive at the number of parameters?

- Our first layer has 784 output parameters (28 x 28). Each of those outputs get passed to each of the nodes in the next layer (784 x 128). We also have a bias term for each node (128) giving us 100,480 learnable parameters.

It's worth reinforcing that, in a dense layer, every output value gets passed to every input node in the next layer. The thing that is different is the weight applied to that value, which will be different – or has the potential to be different given that this is a learned parameter. Therefore, in this instance, each node in the hidden layer will receive 784 values from the previous layer, which will have been modified by a learned weight parameter. To this, we will also add one bias term, making 785 learnable parameters.

- Take the time to be comfortable with howe arrived at 1290 parameters for the output layer.

So, what did we cover in this section?

- The building blocks of deep learning models: layers (size, shape, type) and activation functions.
- How to build a model using the keras sequential API, specifically using a list

How does it add to our existing knowledge?

- This starts to add to the high level concepts we introduced in section 3.

What else can I learn to improve my knowledge?

- Advanced Notebook 2: Models and Layers covers model building with the functional API and using subclassing. It also demonstrates the output of different layer types and activations functions.
- We will cover other ways of creating a sequential model in the following notebooks.

6. Training the Model

How do we train a model?

- As per the chart in section 3, we training by making a forward pass and a backward pass.
- A forward pass means passing the inputs through the model and performing mathematical operations on the data as we defined in the model section to make predictions.
- A backward pass means updating the weights and biases of our model based on the results of the predictions.
- To do this using TensorFlow 2.0, we will use the keras API. We first pass a .compile() method to our model, and then a .fit() method to our model.

What is compiling?

- Compiling means that we assign certain variables to our model that are important in the training process:
  - a loss function, which measures the distance between our predicted and actual classes.

  - an optimizer, which updates the weights based on the backpropagation algorithm
  - a metric to provide an appropriate additional measure of training performance. and an accuracy
- This is not an exhaustive list of what we variables we can pass to the .compile() method. We will pass additional variables in the following notebooks.

```
[ ]  # we can see that this is a relatively easy
     thing to code
     # we pass in the optimiser that we wish to use
     model.compile(optimizer='adam',
        # specify the loss function
        loss='sparse_categorical_crossentropy',
        # and we specify our metrics
        metrics=['accuracy'])
```

What is fitting?

- The .fit() method governs the training process. To fit our model to the data is to pass the data through the model and see how well it predicts what we want it to predict, and then to update the model based on these results – that is, .fit() implements the forward and backward pass based on the model and training variables we have defined.
- In the .fit() method, we get to define the number of epochs (the number of times we go through the data), and a number of other variables that we will cover in following notebooks.

```
[ ]  # we just pass the fit method to the model,
     along with training data and corresponding
     output data
     # We also specify the number of epochs
     history = model.fit(train_images,train_labels,
     epochs=5)
Epoch 1/5
60000/60000 [==============================] - 4s 74us/sample -
loss: 0.4958 - accuracy: 0.8264
```

```
Epoch 2/5
60000/60000 [==============================] - 4s 72us/sample -
loss: 0.3732 - accuracy: 0.8657
Epoch 3/5
60000/60000 [==============================] - 4s 72us/sample -
loss: 0.3328 - accuracy: 0.8777
Epoch 4/5
60000/60000 [==============================] - 4s 72us/sample -
loss: 0.3091 - accuracy: 0.8866
Epoch 5/5
60000/60000 [==============================] - 4s 73us/sample -
loss: 0.2920 - accuracy: 0.8924
```

What do the different parts of the training print out mean?

- The left hand side shows the number of epochs we have performed.
- For each epoch, we can see how much of the training data we have used (it's now 60000 / 60000 because training is over, but this figure changes as the data is loaded and passed through the model.)
- On the right hand side of the data, we can see: the time it has taken to complete the epoch; the loss for that epoch effect on the accuracy for that epoch.

How did we do?

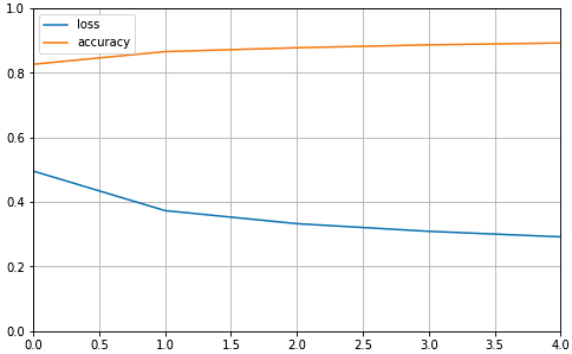
- On the face of it, 89% accuracy seems pretty good. Given this is a publicly available dataset, then we can search to see how others have done on it. We will also apply different models to this dataset to see if we can improve our accuracy.
- We can see that our loss steadily reduced as the epochs increased, and the accuracy improved. This is a good sign and we would probably see further improvement if we ran the training for more epochs.
- The keras .fit() method returns a history object, which records the values of the training. We will explore this now and use it to easily plot these values.

```
[ ] # model.fit returns a history object, which
    contains a history dictionary about everything
    that happened during training
    history_dict = history.history
    history_dict.keys()
dict_keys(['loss', 'accuracy'])

[ ] # for example we can access the loss like this
    history_dict['loss']
[0.49576409851312636,
0.37323990731636686,
0.33276325614054997,
0.3090944565196832,
0.2919588043630123]

[ ] ## we can pass this to a pandas dataframe
    # we need to import pandas
    import pandas as pd
    # pass history data to dataframe object
    history_df = pd.DataFrame(history_dict)
    # and display it
    history_df
loss  accuracy
0     0.495764    0.826450
1     0.373240    0.865667
2     0.332763    0.877717
3     0.309094    0.886600
4     0.291959    0.892450

[ ] # we can use plot functionality of pandas to
    quickly plot our results
    history_df.plot(figsize=(8,5))
    # tailor our plot. Show the grid
    plt.grid(True)
    # set the vertical range to [0 -1]
    plt.gca().set_ylim(0,1)
    # display plot
    plt.show()
```



What does this show?

- The chart shows the improvement in loss and accuracy values over the epochs.
- If we had used a validation set, we would be able to compare the values from the validation and training sets at each epoch, which would tell us a little more about how the model will be able to generalize to unseen data (we will look at this in next notebook).

So, what did we cover in this section?

- That we need to perform a forward and backward pass to do training in deep learning.
- That to do this in TensorFlow 2.0 is fairly easy using the keras API. We simply:
  - pass a .compile() method to our model to define our loss, optimizer and metric variables
  - and pass the .fit() method to our model in order to set the number of training loops and govern other related behavior of the training process.

How does it add to our existing knowledge?

- This starts to add to the high level concepts we introduced in section 3.

What else can I learn to improve my knowledge?

- We will provide more detail about optimizers, loss functions and metrics in the following notebooks.
- We will pass additional arguments to the .compile() and .fit() methods in the following notebooks. For example, we can add a validation set to our training loop, and we can record more information about our model performance.
- Advanced Notebook 3: Training covers training using custom training loops in TensorFlow 2.0.

7. Evaluation and Inference

How do we know how we well the model is performing?

- We have seen how the model performs on the training set, but we need to test it on unseen data to see how well the model really is peforming.
- To do this we can call the .evaluate() method on the model and pass in our test set. This returns the loss and scoring metric that we passed in to the .compile() method (in this case, accuracy).
- We can also use the .predict() method to make predictions on the test image. This method will return (in this instance) an array of 10 values, each representing a class label and the probability the model believes the passed in image is one of the 10 classes.

```
[ ]  # use the evaluate method
     # it returns two values, the loss and accuracy
     from our model
     test_loss, test_acc = model.evaluate(test_images,
     test_labels)
10000/10000 [==============================] - 0s
38us/sample - loss: 0.3425 - accuracy: 0.8765

[ ]  # here we can print out the accuracy and loss
     print('\nTest accuracy:', test_acc)
     print('\nTest loss:', test_loss)
Test accuracy: 0.8765
Test loss: 0.34245488489866255

[ ]  # use our model to make predictions
     predictions = model.predict(test_images)
```

What has been returned?

- a prediction for each image. The prediction actually is a value against each of the 10 labels for each image. This value is the probability the model has given that one of the 10 labels is correct for the unseen image.

```
[ ]  # we can view the predictions for just one of
     the predictions
     predictions[0]
array([1.2182815e-06, 9.0716959e-08, 3.3988119e-08, 1.8084372e-09,
       3.9402700e-08, 2.1158228e-02, 2.7726156e-07, 1.4800583e-02,
       2.4397614e-06, 9.6403706e-01], dtype=float32)
```

What is the predicted label?

- The one with the highest probability. We can retrieve this using the np.argmax function

```
[ ]  # get highest value of the predictions
     np.argmax(predictions[0])
9
```

How can we compare this with the actual label?

```
[ ]  # we can use class_names to see what was the
     9th label of the classes
     class_names[np.argmax(predictions[0])]
'Ankle boot'
[ ]  # and the get our test_label
     test_labels[0] == class_names[np.argmax
     (predictions[0])]
False
```

7.1 Plotting our results

How can we display our results?

- We can use matplotlib to display the results of our model.
- We will plot an image, what the predicted label is, and whether this was correct. We will also plot a bar chart showing the probability assigned by the model to the different classes. We start by defining some helper functions.

```
[ ]  # define a function that plots the predicted
     image
     def plot_image(i, predictions_array,
     true_label, img):
        # assign variable names to our parameters
        predictions_array, true_label, img =
        predictions_array[i], true_label[i], img[i]
        # remove grid and axis values
        plt.grid(False)
        plt.xticks([])
        plt.yticks([])
        # display images
        plt.imshow(img, cmap=plt.cm.binary)
        # return predicted label
        predicted_label = np.argmax(predictions_array)
        # and assign it a colour based on whether it
        was correct
        if predicted_label == true_label:
        color = 'blue'
```

```
        else:
        color = 'red'
        # define label format
        plt.xlabel("{}{:2.0f}%
        ({})".format(class_names[predicted_label],
        100*np.max(predictions_array),
        class_names[true_label],
        color=color))
[ ]  # plot a function to graph the probabilities
        def plot_value_array(i, predictions_array,
        true_label):
        # assign variable names to our parameters
        predictions_array, true_label =
        predictions_array[i], true_label[i]
        # remove grid and axis values
        plt.grid(False)
        plt.xticks([])
        plt.yticks([])
        # plot a bar chart
        thisplot = plt.bar(range(10),
        predictions_array, color='#777777')
        # reduce y axis to between 0,1 values
        plt.ylim([0,1])
        # create prediction
        predicted_label = np.argmax(predictions_array)
        # set plot colour
        thisplot[predicted_label].set_color('red')
        thisplot[true_label].set_color('blue')
```

Now let's use our functions to plot one image and a series of images together

```
[ ]  # look at the 0 image
     i = 0
     # set size of figure for the plot
     plt.figure(figsize=(6,3))
     # display image one side of the figure
     plt.subplot(1,2,1)
     plot_image(i, predictions, test_labels,
     test_images)
     # display chart on the other side of the figure
     plt.subplot(1,2,2)
     plot_value_array(i, predictions, test_labels)
```

```
    # show plot
    plt.show
<function matplotlib.pyplot.show>
[ ] # look at the 12 image
    i = 12
    # as above
    plt.figure(figsize=(6,3))
    # as above
    plt.subplot(1,2,1)
    plot_image(i, predictions, test_labels,
    test_images)
    # as above
    plt.subplot(1,2,2)
    plot_value_array(i, predictions, test_labels)
    # as above
    plt.show
<function matplotlib.pyplot.show>
```

We can use this functionality to plot more than one image

```
[ ] # Plot the first X test images, their predicted
    labels, and the true labels.
    # Color correct predictions in blue and
    incorrect predictions in red.
    num_rows = 5
    num_cols = 3
    num_images = num_rows*num_cols
    plt.figure(figsize=(2*2*num_cols, 2*num_rows))
    for i in range(num_images):
       plt.subplot(num_rows, 2*num_cols, 2*i+1)
       plot_image(i, predictions, test_labels,
       test_images)
       plt.subplot(num_rows, 2*num_cols, 2*i+2)
       plot_value_array(i, predictions, test_labels)
    plt.show()
```

7.2 Making a prediction on a single image

Didn't we do this earlier?

- No. We made predictions on the entire test set and then selected one of those predictions to view. Here we just want to make a prediction on one of the images.

```
[ ] # get an image from the test_dataset
    img = test_images[0]
```

We need to talk about batches...

- We haven't encountered batch sizes yet. This a value that represents the number of images (or other data type if we aren't doing image classification) that are passed to the model.
- In training, we pass a batch size and this represents how many images the model will make predictions on before recording the loss and updating the model. In the example above, this value defualted to the entire dataset (60000).
- When making predictions with test (or other) images, tf.keras needs to make predictions on a batch. This means, it needs to understand how many images are about to passed to it. In practical terms, this means we need to add a batch dimension to the shape of our data.
- Getting the data into the right shape for modelling and predicting can be a bit tricky sometimes. The Advanced Notebook: Tensors covers this in more detail.

```
[ ] # lets looks at the shape of image before we
    add a batch dimension to it
    print(img.shape)
[ ] # add batch dimension by adding a dimension to
    the image shape
    img = (np.expand_dims(img,0))
```

```
    print(img.shape)
[ ] # make prediction
    predictions_single = model.predict(img)
[ ] # show the array
    predictions_single
[ ] # again we can use argmax
    np.argmax(predictions_single[0])
[ ] # and we can plot the result
    plot_value_array(0, predictions_single,
    test_labels)
    #
    _ = plt.xticks(range(10), class_names,
    rotation=45)
```

So, what did we cover in this section?

- The .evaluate() and .predict() methods for understanding how our models perform on unseen data.
- Plotting our results so that we can see how the model did
- Predicting the class of a single image by passing in a batch dimension to the image shape.

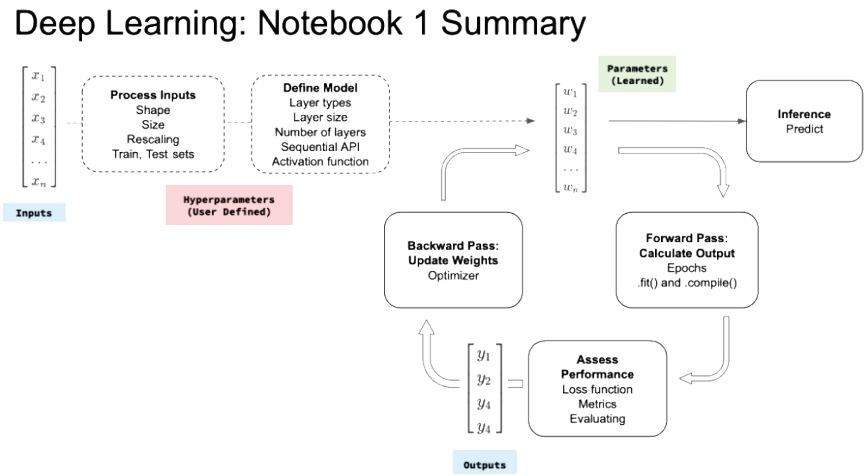
How does it add to our existing knowledge?

- This starts to add to the high level concepts we introduced in section 3.

What else can I learn to improve my knowledge?

- We will cover batches further in notebook 2.
- We will use the .evaluate() and .predict() methods in later notebooks too.

8. Summary



What have we learnt?

- We can see from the chart above, quite a lot.
- We have introduced core Deep Learning concepts, explained them in high level terms, and put them in to practice by writing code.
- If you feel like this was a lot, don't worry because it was! It is a lot to take in. Let's end with the opportunity to build and train your own model!

9. Exercise

- The best way to learn code is to write it out for yourself. Take the opportunity to reinforce what you have learnt by adding code cells below and doing the following:

Fit the model

- Change the number of epochs when you .fit() the model again and see how performance changes.

Build a new model

- Create a new model and:
  - add an additional layer; and/or:
  - add more nodes to the hidden layer(s)
- You will need to recompile the model, which means adding the .compile() method to the new model and passing in the arguments we set out above.

Good luck!

```
[ ]
```

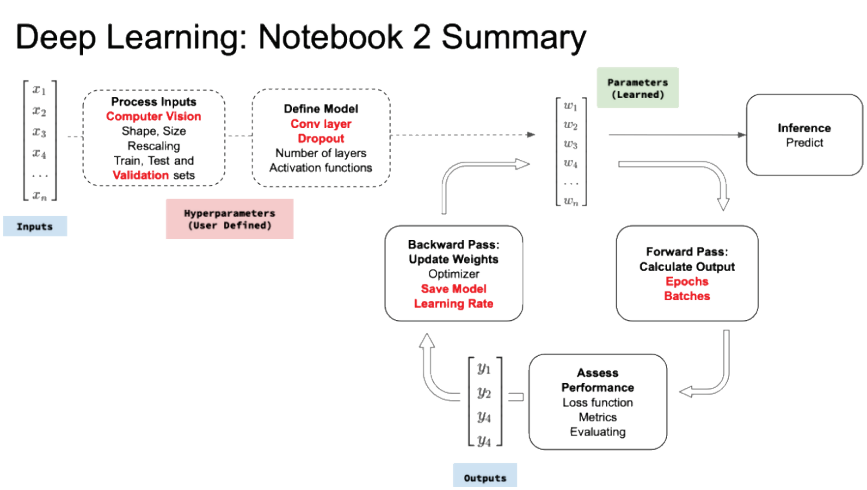
Part 2

TensorFlow 2.0: Notebook 2:

Computer Vision with CNNs

1. Introduction to this Notebook

- In the previous notebook (see here) we introduced deep learning concepts and the TensorFlow 2.0 code to implement these concepts.
- In this notebook, we will use another dataset – the mnist dataset – to build on our knowledge. In particular, we will:
  - introduce Computer Vision
  - introduce convolutional layers into our models
  - introduce the concept of regularisation
  - introduce the validation set in training our model
  - introduce how to save and reuse our model
- The image below sets out how this fits within our deep learning framework and exising knowledge



1.1 Load Libraries

```
[ ]  # we need to install tensorflow 2.0 on the
     google cloud notebook we have opened
     !pip install -q tensorflow==2.0.0-alpha0
[ ]  ## importing as per previous notebook
     # We are future proofing by importing modules
     that modify or replace exising modules that we
     may have used now
     from __future__ import absolute_import,
     division, print_function, unicode_literals

     # import tensorflow and tf.keras
     import tensorflow as tf
     from tensorflow import keras

     # import helper libraries
     import numpy as np
     import matplotlib.pyplot as plt

     # let's print out the version we are using
     print(tf.__version__)
[ ]  ## some additional imports for this notebook
     from tensorflow.keras import datasets, layers,
     models
```

1.2 Loading our Data

```
[ ]  # split our data
     (train_images, train_labels), (test_images,
     test_labels) = datasets.mnist.load_data()
[ ]  # lets have a quick look at our data
     plt.imshow(train_images[0])
[ ]  # and at our labels
     np.unique(train_labels)
```

What is the problem we are trying to solve?

- As we can see, we have images of digits from 0-9, and labels from 0-9. We are trying to build a model that correctly classifies the digits in the image.

2. Data: Introduction to Computer Vision

What is Computer Vision?

- Computer Vision is the field of how computers can gain understanding from images and videos. It includes tasks such as image recognition and object detection. Deep Learning is seen as the state of the art technology for solving computer vision problems.

Why is Deep Learning particularly good at it?

- The layers within a deep learning model are good for identifying and modelling the different aspects of an image (such as edges, parts of faces, and other important parts of an image). The meaning that each layer extracts can be built up to form representations for lots of different image types that can then be classified.
- In particular, convolutional layers are good at extracting representation from image data and they form the basis of deep

learning models for image recognition. The ability to build larger and larger models that consist of these convolutional layers, and to train them with more and more data (thanks to increasing compute power), led to a leap forward in state of the art for computer vision.

How does it work?

- Every image is represented by an array of numbers. You may have noticed this when we looked at the shape of the images we were processing. This shape represents the number of pixels in an image, and each pixel has a numerical value. This numerical value maps to a colour value that is displayed. It is also what we use as input values to our model.

```
[ ]  ## lets start by looking at the shape of an
     image

     ## we can see that it is 28 x 28 pixels
     train_images[0].shape
[ ]  ## we can also see that these pixels are
     represented in an array of numbers
     train_images[0]
[ ]  # we need plt.imshow() - or another library
     such as OpenCV or PIL - to output an image from
     this array
     plt.imshow(train_images[0])
```

What do the array values mean?

- Each value leads to a colour for the pixel that the array value represents. Actualy what colour is displayed depends somewhat on the number of colour channels the array has. We have only one channel present in this dataset. This is gray-scale channel. Typically, we will see three channels for colour images, with each channel representing one of Red, Green,

Blue. A value in one channel will display a different colour than a value in another channel.

- See the tutorials here for more detail on how the values within a channel map to a colour.
- Its worth noting here that there are typically 256 values (0-255) available in each channel, making a total combination of c. 16.8m colours available per a three channel image!
- As per the previous notebook, we will rescale the arrays to between 0 and 1. This needs to happen in order to maximise the success of the training.

What about images of different shapes?

- The size of an image can and does vary. In this case, we have small image of 28 x 28 pixels (or 28, 28, 1) given we have one channel. This was the same for the previous dataset and it makes it easy to train models.
- Outside of introductory tutorials, it is likely that you will see much larger images, meaning many more pixels and therefore larger arrays to train and learn representations on. This will make the models larger and training more involved.
- One final thing to note is that Deep Learning models always require an array of the same size to be passed to it. This means that images which differ in size need to be preprocessed so that they are the same size before being passed to the model.

```
[ ]  # we now need to reshape the data to add a
     colour channel
     train_images = train_images.reshape((60000, 28,
     28, 1))
     test_images = test_images.reshape((10000, 28,
     28, 1))
[ ]  # we can view the new shape
```

```
    train_images.shape
[ ] # and normalize the data
    train_images, test_images = train_images /
    255.0, test_images / 255.0
```

So, what did we cover in this section?

- An introduction to computer vision, including how images are represented by arrays.
- How the shape of an array matters for our model and the preprocessing required prior to feeding the arrays to our model.

How does it add to our existing knowledge?

- This builds on the deep learning concepts from notebook 2.

What else can I learn to improve my knowledge?

- Images have to be fed into the model in the same shape each time. This requires pre-processing.
- Prior to feed images into a model, we can also change the image in certain ways to add noise and variety to the training data. This should mean that the model is more robust and better at generalizing to unseen data. We will look at both of these in the Advanced: Data Augmentation notebook.

3. Model Building

3.1 Convolutional Models

What did we do in the previous notebook?

- In Notebook 1, we looked at the main elements of deep learning models: input and output layers, hidden layers –

which contain the learned parameters of the model, and activation functions.
- We also looked briefly at the different sort of hidden layers available to us, such as dense and convolutional layers.
- And, we built model that took an image as an input and flattened it to a 1D array.

How do we build a convolutional neural network?
- A convolutional neural network (CNN) contains both dense and convolutional layers. The convolutional layers form the base of the model and extracts representation from the image. The dense layers form the head of the model and takes this represetation and maps it to our output classes
- A convolutional layer takes our image as it (subject to any preprocessing to get it in a standard shape or augmented to add noise and variety to the dataset) – that is, we do not need to flatten the image into a 1D array. We flatten the array after our final convolutional layer and prior to passing our input to the dense layer.

Why use a convolutional layer?
- A convolution better encodes the key information in an image than other types of layers. Their application to computer vision resulted in a marked improvement in what was state of the art. That's why we use them.

What is a convolutional layer?
- Simply, a convolutional layer is a layer that performs a mathematical operations known as convolutional on the input data. In contrast, a dense layer performs matrix multiplication on its inputs.

- Each convolutional layer has a user-defined set of filters (or windows) that we pass over the image. We define the number and size of filters, although they are typically a 3 x 3 matrix.
- This filter contains a set of weights that will be learned by the model and which are used to multiply the input values and return a new value in the layer's output. It's these filters that contain the learning of the convolutional layers of the model, whose weights will be updated as we train so that they are more and more able to extract key information from the image.
- The filter is applied to all the image channels as it passes over each pixel location such that it will look at a specific row and column index position and all the array values available at that index:

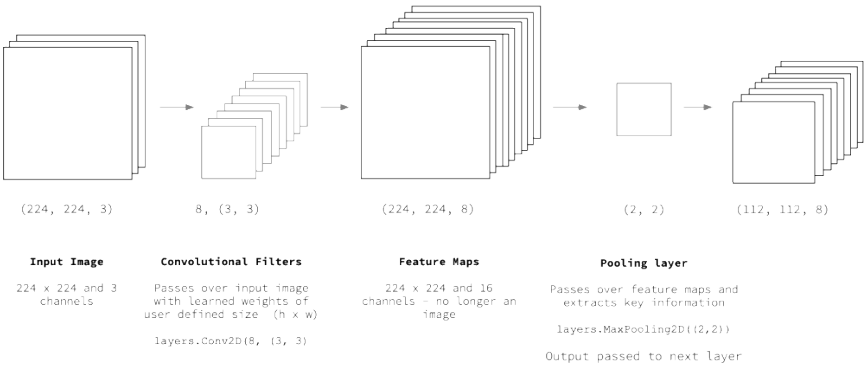
```
(row,column,:)
```

- We won't go in to how convolutional works here, but see the cell at the end of this section for links that do explain how it works.

So what does a convolutional layer return?

- A convolutional layer returns an output array of the same (row, column) shape as the input array, but with one channel only.
- It tends to be the case that convolutional layer is paired with a pooling layer. We won't cover these in any detail, but its sufficient to know that a pooling layer tries to extract the key information from the convolutional layer while typically halving its size.
- The diagram below set this out.

Deep Learning: Convolutional and Pooling Layers




```
[ ] ## lets build our convolutional base
    ## we use the Sequential API but use .add()
    rather than passing the layers in as a list

    # build model using sequential
    model = models.Sequential()
[ ] # start adding layers. input shape has been
    defined, including the channel value we added
    via reshape earlier
    model.add(layers.Conv2D(32, (3, 3),
    activation='relu', input_shape=(28, 28, 1)))
    # max pooling layers
    model.add(layers.MaxPooling2D((2,2)))
    # this is then repeated to build
    model.add(layers.Conv2D(64, (3, 3),
    activation='relu'))
    # max pooling layers
    model.add(layers.MaxPooling2D((2,2)))
    # additional convolutional layer
    model.add(layers.Conv2D(64, (3, 3),
    activation='relu'))
[ ] # print model
    model.summary()
```

How do we get to the parameter count?

- The parameters of a convolutional layer are defined by:
  ```
  ((filter height ×filter width) + bias term)
  ×number of filters
  ```
- The bias term is a value of 1 so the number of parameters for the first convolutional layer is:
  ```
  ((3×3)+1) ×32=320
  ```

What about the classification layer?

- As we said above, a convolutional based needs a classification layer to take the information extracted from an image and map it to output classes.
- We take the final output shape of the Convolutional layer and flatten it to a 1D array. We then define our output layer, which in this instance is a layer of 10 with a softmax activation function. We have also added an additional layer to provide additional parameters between the flattened and output layer.

```
[ ]  # flatten
     model.add(layers.Flatten())
     # add a fully connected layer
     model.add(layers.Dense(64, activation='relu'))
     # add the output layer, a fully connected layer
     with a softmax activation
     model.add(layers.Dense(10, activation='softmax'))
[ ]  # complete model summary
     model.summary()
```

So, what did we cover in this section?

- We looked at convolutional models and how they are constructed
- We looked at what convolutional layers are

- We built our model using convolutional and dense layers, using the Sequential API but by using model.add(layers) rather than passing the layers to the model as a list.

How does it add to our existing knowledge?

- We built on our understanding of how models are built in deep learning.
- We built on our understanding of the Sequential API.

What else can I learn to improve my knowledge?

- Understand more about convolution:
- We will cover this in more detail in the Advanced Notebook 3: Model and Layers.
- There are a lot of good articles out there too. For example, Cezanne Comacho, and Chris Olah all provide a good understanding of how convolution works.
- For the math of convolution, have a look at this paper.

4. Training

4.1 Validation Sets, Batch Sizes and Learning Rates

What did we do in the previous notebook?

- We covered forward and backward passes, compiling our model and fitting it.

What will we cover now?

- We will cover more concepts around training the model: the purpose of a validation set; overfitting; and regularization. We

will add in a validation and look for overfitting in our model performance. We won't add any regularization methods
- We will also cover learning rates and batch sizes.

What is validation set?
- A validation set is a dataset that is kept aside from the training dataset. Typically, at the end of each epoch, the trained model is passed the validation set in inference model, and the loss and other metrics recorded. The model is then trained again for another epoch, and at the end of this epoch is passed the validation set in inference mode.
- This allows us to see during training how the model performs on unseen data and also whether the model is under- or over-fitting.

How do we create a validation set?
- We can set aside some data from our training set prior to beginning training, store it as a variable (or a set of (data, label) variables) and pass this to the validation_data argument in model.fit()
- We can use the validation_split parameter in model.fit() to specify the fraction of the data to be used as training data.

What is under and overfitting?
- When we train our models, they can sometimes struggle to generalize well. This means that they do not perform well on unseen data.
- A model that overfits will get better and better results (loss and metrics) on the training data and decreasing results on the validation set. This is because it is fitting too much to the

specific characteristics of the training data, which may not be present in the unseen data.
- A model that underfits does not perform well on either the training or validation data.
- By including a validation set, we can monitor how well the model performs, and if the performance on the training and validation sets diverge too much then we can start to conclude that something need to be changed.

What can we do to address this?
- We can use regularization to help prevent overfitting.
- We can regularize our model and its training in a number of ways, and to some extent they all penalize actions that may cause the model to overfit to the training data:
- When building a model, we add a dropout layer. This turns off a user defined number of outputs from a layer r during training and helps prevents the model becoming reliant on certain paths through the model.
- During training, weight regularization. This penalizes large weights (our learned parameters) in the model. Larger weights will create a larger loss value that the model will then use to update the weights. A model with fewer larger weights, i.e. with the learning more evenly spread across the nodes, will have a better chance of generalizing well.

What is a batch_size?
- A batch size is the number of samples the model trains on before performing a backward pass (model update). The number of batches in an epoch is equal to:
- number of sample in a training setbatch size

- We can set the batch size and choose to see the model performance as trains per batch size (using a parameter in model.fit()).

What is the learning rate?

- The learning rate controls the size of the update to the learnable parameters (weights and biases) during the backward pass, based on the loss of the model.
- It is a parameter of the optimizer, set at the model.compile() stage, and can be set by the user. The trick is to set the learning to update the weights sufficiently to change performance, but not update them too much so that the model weights swing between higher and lower values without settling on a path to the best model.
- There are various strategies for mitigating this problem that are worth investigating.

```
[ ]  ## Lets compile the model again

     model.compile(optimizer='adam',
             loss='sparse_categorical_crossentropy',

             metrics=['accuracy'])
[ ]  ## and build a training loop

     history = model.fit(train_images, train_labels,
             # add batch size
             batch_size = 32,
             # epochs
             epochs = 10,
             # and add a validation set
             validation_split=0.2)
```

Did we do any good?

- 98% plus on the validation set seems pretty good!
- Let's plot the loss and accuracy and see what it shows us

```
[ ]  # if we print out our history object we can see
     that it now includes validation values
     history_dict = history.history
     history_dict.keys()
[ ]  ## we can pass this to a pandas dataframe

     # we need to import pandas
     import pandas as pd

     # pass history data to dataframe object
     history_df = pd.DataFrame(history_dict)

     # and display it
     history_df
[ ]  # we can use plot functionality of pandas to
     quickly plot our results
     history_df.plot(figsize=(8,5))
     # tailor our plot. Show the grid
     plt.grid(True)
     # set the vertical range to [0 -1]
     plt.gca().set_ylim(0, 1)
     # display plot
     plt.show()
```

4.2 Saving Models

What do we mean by saving a model?

- We can save our progress as we train our models. We can save our progress in two ways:
- during training, so that our models are saved after each epoch (or, after an epoch that shows model improvement).
- after training, so that our model has completed its training before we save it
- Of course, we can opt to save the model both during and after training.

- Using TensorFlow 2.0, we can opt to save the model either manually, i.e. after the model has trained, or by using callbacks – i.e. incorporating saving into the training process.

What are we saving?

- We can save the model weights only, the full model (including the weights and the architecture), and the optimizer state.
- It's useful to remember that when we are training a model, the parameters we are updating during the training process are the weights at each layer of the model. Our aim is to train on (data, labels) pairs that mean we can predict effectively on unseen data using the weights we have trained. So it is these weights that are saved, optionally along with the model architecture.
- Optimizer state. We haven't focused too much on optimizers, but remember that this is the way that the model weights are updated. The size of the update is set by the (user defined) learning rate. When we save a model we can therefore save the optimizer-state, meaning we can continue training a loaded model from the state it was in when the model was saved.

Why save a model?

- So that we can reuse it later. This could be to deploy it and use it in inference mode, or to continue training from the point at which we stopped.

Once we have saved a model, how do we use it again?

- Once we have saved our weights and/or model, we can restore the model in a couple of different ways. If we decide to save the weights only, we need to create an identical model to the one that was used to create our weights. If we saved both the model and weights, we can load this entire model.

What are the ways of doing it?

- In this tutorial (notebook 2), we will look at saving and loading model weights and model + weight manually, i.e. after training.
- In notebook 3, we will look at how to save during and after training using callbacks.
- We will use the Keras API. Note there are some other ways to save the model covered in the TensorFlow 2.0 tutorials provided by Google.

One more thing...

- If you are using Google Colab, then we need to save the model to the Google Drive.
- The code below doesn't do that, it assumes we are saving locally.
- So, for now, look at the code to get a feel for what to do but it won't work. I will update this code with a solution.
- The same goes for the checkpoint example in notebook 3.

4.2.1 Saving and Loading Weights Only

```
[ ] ## lets go through the steps of saving the
    model weights only

    # here we define a location for the weights to
    be saved
    model.save_weights('./checkpoints/my_checkpoint')
[ ] # to load the weights we need to create a new
    instance of the same model architecture
    new_instance = model
[ ] # then we can load the weights
    new_instance.load_weights('./checkpoints/
    my_checkpoint')
[ ] new_instance.weights[0][0][0][0]
```

4.2.2 Saving and Loading an entire model

```
[ ]  # this saves it to the HDF5 format
     model.save('my_model.h5')
[ ]  # recreate the saved model, including weights
     and optimizer
     new_model = keras.models.load_model('my_model.h5')
```

So, what did we cover in this section?

- Validation sets
- Overfitting
- Regularization
- Learning Rates, Batch Sizes
- Saving Models

How does it add to our existing knowledge?

- We have built on our understanding of the training loop by adding different aspects to it such as batch size and validation set, as well as getting a feel for what we need to guard against in training models (e.g. overfitting)

What else can I learn to improve my knowledge?

- In the next notebook, we will use a callback to save a model as it trains.
- Overfitting – much more to this topic. There is a good treatment of this subject in Introduction to Statistical Learning.

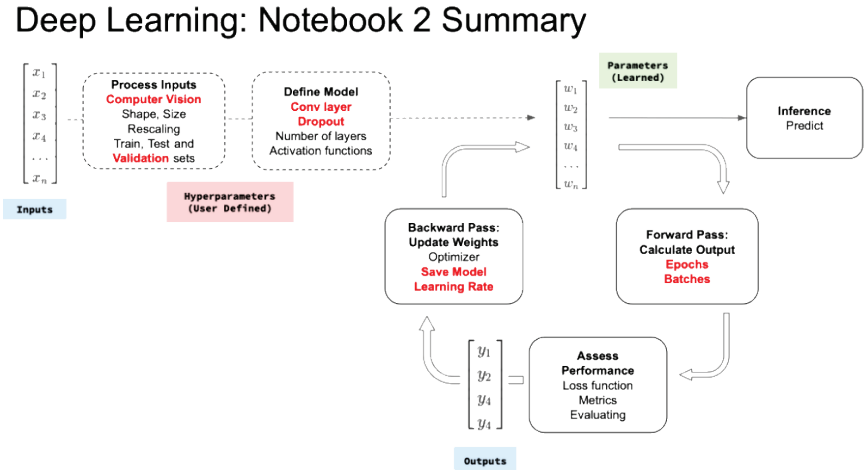
5. Evaluation and Inference

```
[ ]  ## using the model we have just saved, evaluate
     it on the test set
     ## if you are stuck, use some of the code from
     notebook 1
    #
```

```
    test_loss, test_acc =
     model.evaluate(test_images, test_labels)

    ## if we had saved our weights or model and
     loaded them up, we would use the model
    ## variable name we had saved here
[ ]  # show how the model performs in inference
     mode. Again, use some code from previous
     notebook
```

6. Summary



- The chart above shows what we covered in this notebook.
- We have covered a lot so don't worry if you need to revisit some of this again.

7. Exercises

```
[ ]  ## build another convolution model
     ## add more layers, vary their size and look up
     dropout and how to add that

[ ]  ## recompile the model and fit it

[ ]  ## evaluate the model and use it inference mode

[ ]  ## save the model
```