Outline

##Tensorflow Keras-1

- Business-case introduction - Healthyfi.me
- Basic EDA
- Tensorflow documentation overview
- tf.keras overview
- Keras Sequential API
- Dense Layer
    1. Input shape
    2. Activation function
        - Binary classification
    3. Custom layer name
        - Sequential classification model
        - Multi target output
- Methods for sequential model
    - model.add
    - model.weights
    - model.summary
        - number of parameters
- Parameter Initialisation
- Model Compilation
    1. Loss
        - Type of loss
    2. Optimiser
    3. Metrics
        - Logistic regression model
- Model Training
    1. X_train, y_train
    2. epochs
    3. batch_size

4. validation_split
5. validation_data

- history

1. verbose
2. history object attributes
3. Loss and Accuracy Curve

##Tensorflow and Keras-2

- Keras Functional API

1. Why Functional API
2. Keras Functional Model Class
    - Functional model
    - Model summary
    - Model Hyperparameter

- prediction and evaluation

1. model.evaluate
2. model.predict
    - Model prediction
    - Binary classifier
    - What's the price?

- Callbacks

1. Why do we need a callback?
2. Custom callback
3. Standard callbacks and their purpose
    - Adding callbacks
    - Callbacks in tensorflow
    - Customized loss function
    - Avoid overfitting

- Tensorboard

1. CallBack
2. Dashboard

Quizes of lec-3

Quiz-1 and 2

```
Suppose we have a MLP network that has 2 inputs going into 3 neurons in the first layer.
The results of these neurons are going to 5 neurons in the hidden layer.
The output layer consists of 2 neurons.

What will be the shape of $W^2$

a) 1 x 3
b) 2 x 3
c) 5 x 2
d) 3 x 5
```

Ans: d

W^2 refers to the weight matrix connecting the first layer to the second layer.

There are 3 neurons in L-1, and 5 in L-2. Therefore, there will be a total of 15 weight values, and the shape of W^2 will be 3×5

```
What will be the shape of $b^3$?

a) 1 x 2
b) 2 x 1
c) 5 x 2
d) 2 x 5
```

Ans: a

Since there are 2 neurons in layer 3, i.e. output layer, we will have 2 biases.

This will be represented as 1 x 2 matrix.

Quiz-3

```
Can we use a NN for a regression task?

a) No.
b) Yes, by adding more layers in order to increase computation.
c) Yes, by using linear functions for activation throughout the network.
d) Yes, by using linear function for activation for the neuron(s) in output layer.
```

Ans: d

If the last neuron in chain is:

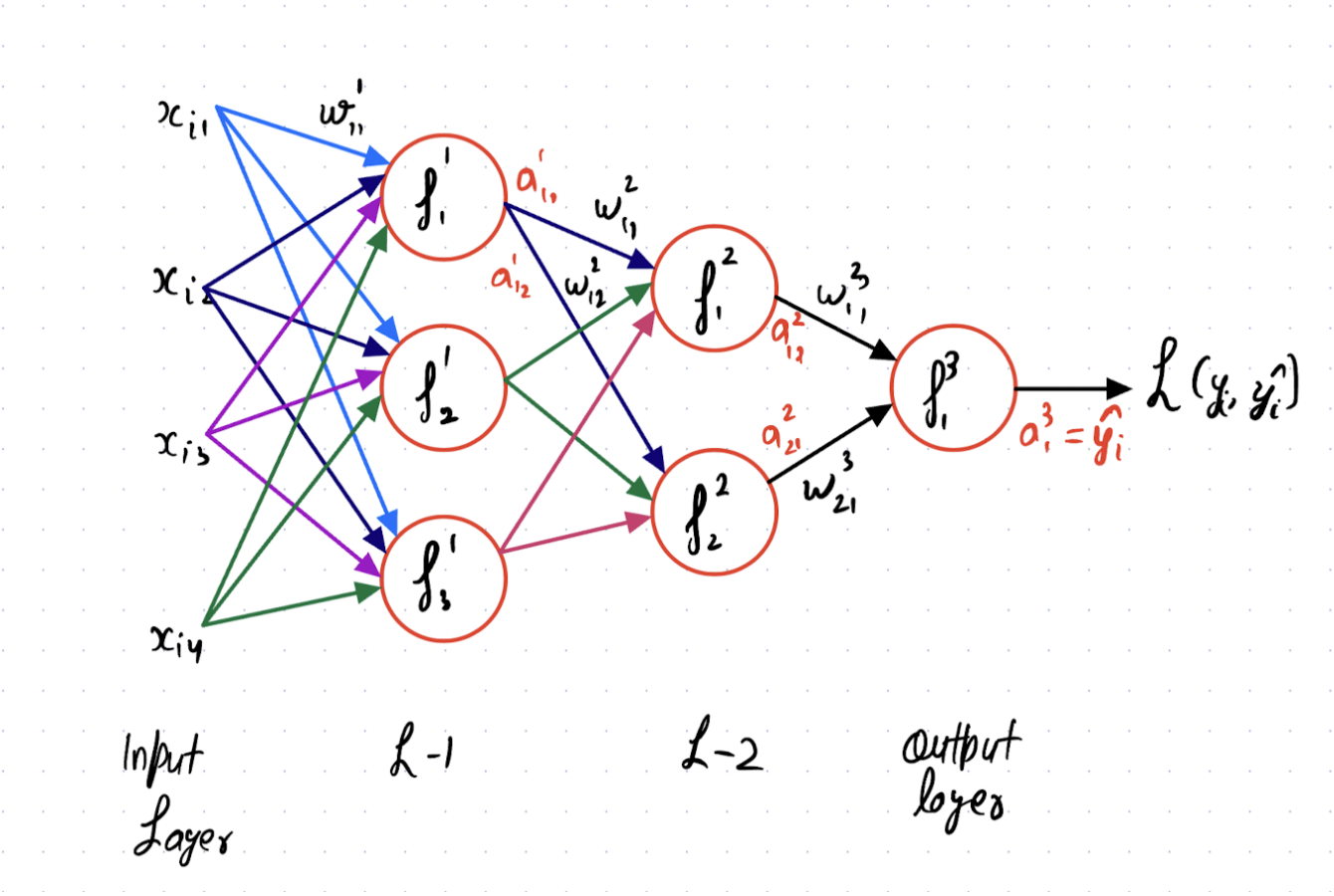
- Linear --> then NeuralNet will do regression. Ex: y = x
- Probabilistic (Sigmoid/softmax) --> NeuralNet will do classification. Ex: y = sig(x)

If the activations of intermediate layers are linear also, then, NN wouldn't be able to create high order complex features

The activations in intermediate layers should always be non-linear.

Quiz-4

Consider the following NN:-



How will we calculate the value of $\frac{\partial Loss}{\partial w_{11}^1}$?

a) $\frac{\partial L}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial a_{11}^2} \cdot \frac{\partial a_{11}^2}{\partial a_{11}^1} \cdot \frac{\partial a_{11}^1}{\partial w_{11}^1}$

b) $\frac{\partial L}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial a_{21}^2} \cdot \frac{\partial a_{21}^2}{\partial a_{12}^1} \cdot \frac{\partial a_{12}^1}{\partial w_{11}^1}$

c) $\frac{\partial L}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial a_{11}^2} \cdot \frac{\partial a_{11}^2}{\partial a_{11}^1} \cdot \frac{\partial a_{11}^1}{\partial w_{11}^1} + \frac{\partial L}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial a_{21}^2} \cdot \frac{\partial a_{21}^2}{\partial a_{12}^1} \cdot \frac{\partial a_{12}^1}{\partial w_{11}^1}$

d) $\frac{\partial L}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial a_{11}^2} \cdot \frac{\partial a_{11}^2}{\partial a_{11}^1} \cdot \frac{\partial a_{11}^1}{\partial w_{11}^1} \cdot \frac{\partial L}{\partial a_1^3} \cdot \frac{\partial a_1^3}{\partial a_{21}^2} \cdot \frac{\partial a_{21}^2}{\partial a_{12}^1} \cdot \frac{\partial a_{12}^1}{\partial w_{11}^1}$

Ans: c

As per the backpropagation, we employ chain rule to calculate the gradient of each parameter, in order to update them.

There can be multiple paths to reach a certain parameter.

In order to combine the derivate from path-1 and path-2, we take the **sum** of the derivatives in these paths.

Quiz-5

```
Which of the following is true regarding Vanishing gardients?

a) This is caused when we're using sigmoid or tanh as activation functions
b) This is caused when we're using ReLu or Leaky ReLu as activation functions
c) This happens because the gradients are ~0, for most of the values of z, causing the lea
d) This happens because the learning is too fast, causing it to overshoot, and not find op
```

Ans: a, c

For sigmoid and tanh, the derivative is very close to zero for most values.

This makes the gradients ~0, thereby causing the learning to be too slow.

- In the pre-read, you would have studied about using Google Colab, we will use it in today's lecture.
- But lets first understand the business use-case for today.

Business-use case

1 cell hidden

Data Acquisition

[ ] *8 cells hidden*

EDA

[ ] *22 cells hidden*

Tensorflow

- We will start-off directly with importing and checking it version.

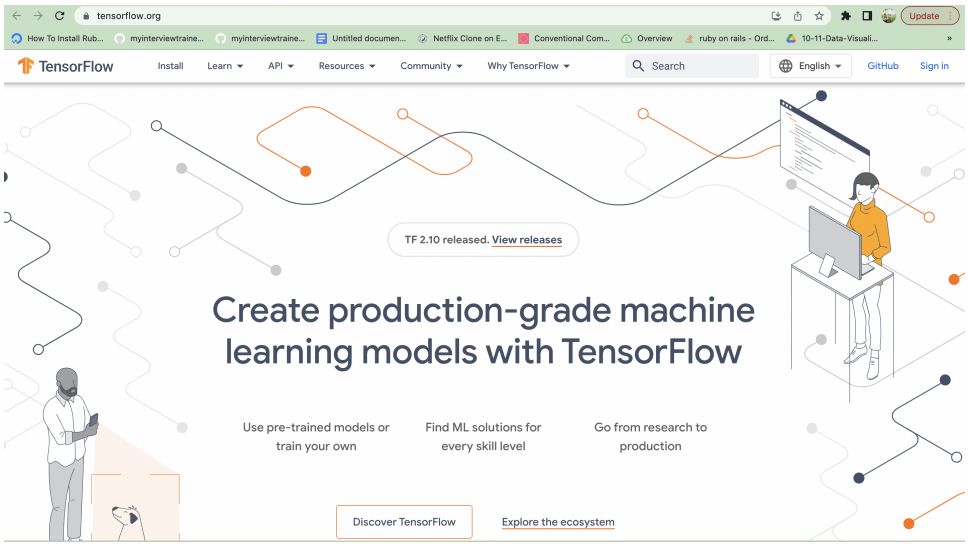
```
import tensorflow as tf
tf.__version__
```

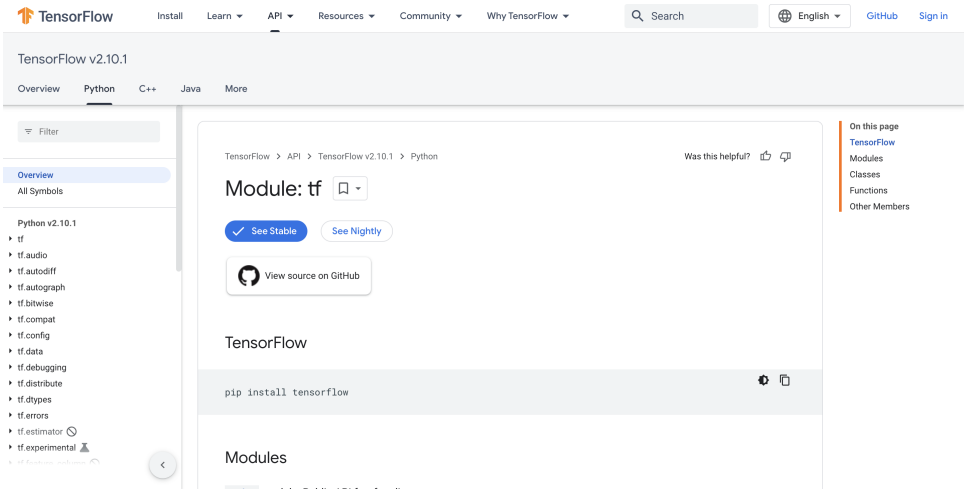
```
'2.9.2'
```

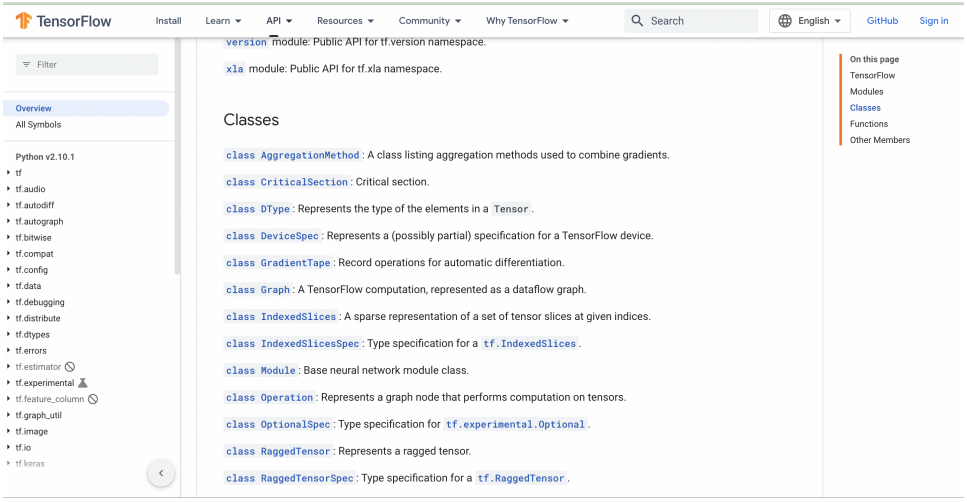
- In TensorFlow 2, Keras has become the default high-level API for - TensorFlow.
- Because of Keras'es ability to write very simple code
- The complete keras API is now wrapped up as part of the TensorFlow installation and has become seamlessly integrated with TensorFlow.
- **You don't need to seperately install Keras now**.
- Even though it's the high-level API for TensorFlow, we'll be able to do most, if not all, of your model development using keras.
- If you want to do more complex stuff like designing your own activation function or any research work, you can do it using Tensorflow-2

Tensorflow documentation

- To browse the TensorFlow documentation, just head to www.tensorflow.org and go to the API documentation at the top.
- Lets look at the latest stable release 2.7v.
- We can look through all of the modules that are within the library by going to the [API](API) section - you'll see there's quite a few here.
- On further scrolling, you will find Classes, functions and data types supported by tensorflow.
- In the left, you will find all these modules listed in alphabetical order.







There are two modules which we will start using right away.

tf.keras Module

The first one is `tf.keras`, click on the module to see whats inside, or you can use the `dir()` function to get the list of contents as well.

```
dir(tf.keras)
```

```
['Input',
 'Model',
 'Sequential',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__internal__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '__version__',
 '_sys',
 'activations',
 'applications',
```

```
 'backend',
 'callbacks',
 'constraints',
 'datasets',
 'dtensor',
 'estimator',
 'experimental',
 'initializers',
 'layers',
 'losses',
 'metrics',
 'mixed_precision',
 'models',
 'optimizers',
 'preprocessing',
 'regularizers',
 'utils',
 'wrappers']
```

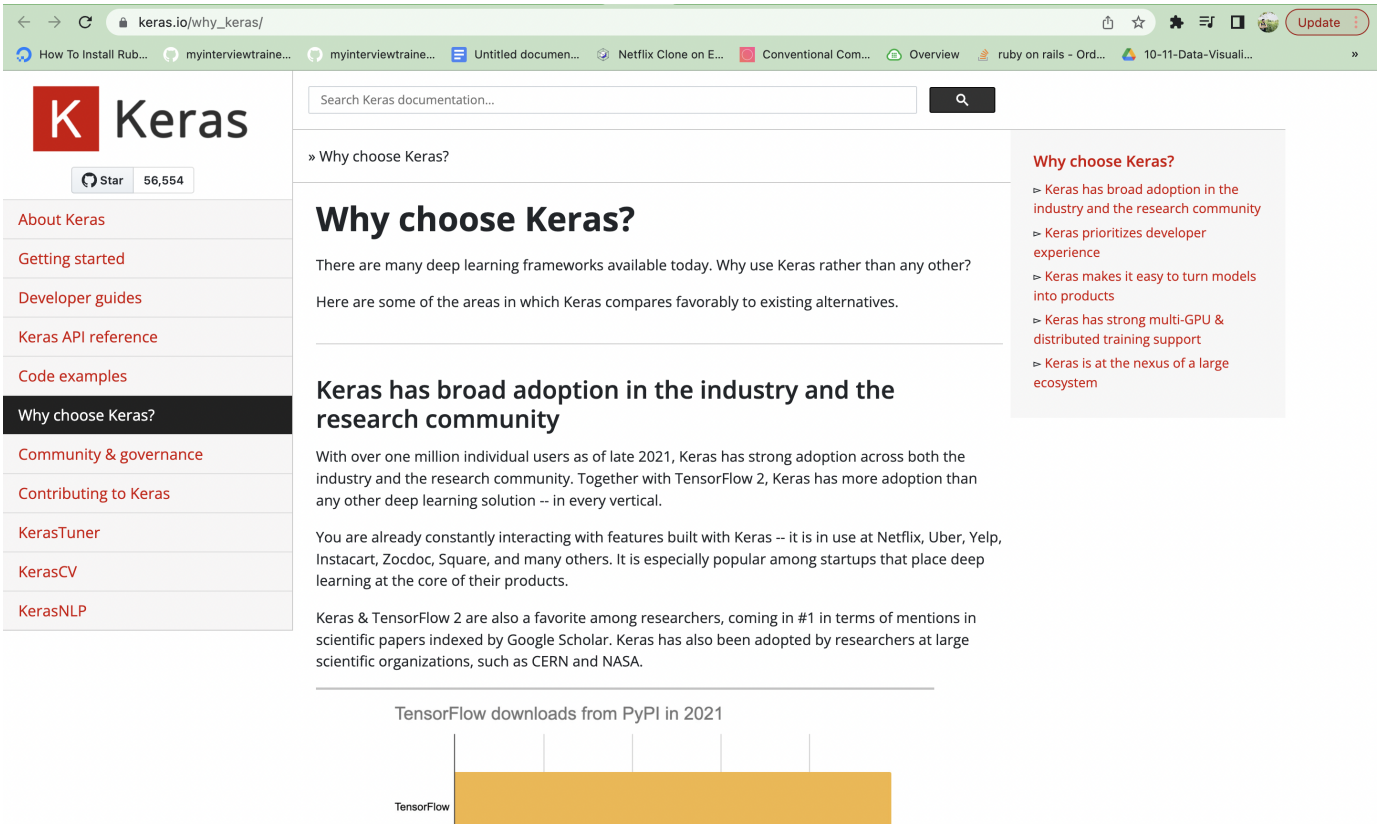
- As we can see, there are some classes like Input, Model and Sequential which we will dicuss in the class.
- There are submodules like layers, losses and activations.

But what is keras? and How is it different from tensorflow?

- Its a library providing high-level neural networks APIs developed with the specific goal of making it easy to build and use giant deep learning models.
- Note: Keras merely provides high-level APIs for deep learning, the underlying code responsible for actual execution of algorithms (mostly written in C/C++) is provided by a "backend".
- It can support other backends as well besides tensorflow, eg: Theano - more at www.keras.io
- But when you install tensorflow2, you get it packaged as the part of the library.
- Also, it is Open Source code – Large community support

Lets look at one of them, the better way of course is to browse through the documentation

Note: You may go through this article which explains why one should choose Keras



Let's look at the various activation functions available inside keras

```
dir(tf.keras.activations)
```

```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_sys',
 'deserialize',
 'elu',
 'exponential',
 'gelu',
 'get',
 'hard_sigmoid',
 'linear',
 'relu',
 'selu',
 'serialize',
 'sigmoid',
 'softmax',
 'softplus',
 'softsign',
 'swish',
 'tanh']
```

tf.data

- Other interesting module that we will use in further classes is `tf.data`
- We'll be looking at this further on in the course when we get to talking about **data pipelines**
- There, we will define our custom classes to load and pass different types of dataset like rows, images, text to NN as mini-batches

```
dir(tf.data)
```

```
['AUTOTUNE',
 'Dataset',
 'DatasetSpec',
 'FixedLengthRecordDataset',
 'INFINITE_CARDINALITY',
 'Iterator',
 'IteratorSpec',
 'Options',
 'TFRecordDataset',
 'TextLineDataset',
 'ThreadingOptions',
 'UNKNOWN_CARDINALITY',
 '__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__path__',
 '__spec__',
 '_sys',
 'experimental']
```

We can write code in keras using two ways:

- Sequential API
- Functional API

But for this lecture, we will talk about the most used keras Sequential API which will help us in most of the Deep NN that we will study in this course.

Keras Sequential API

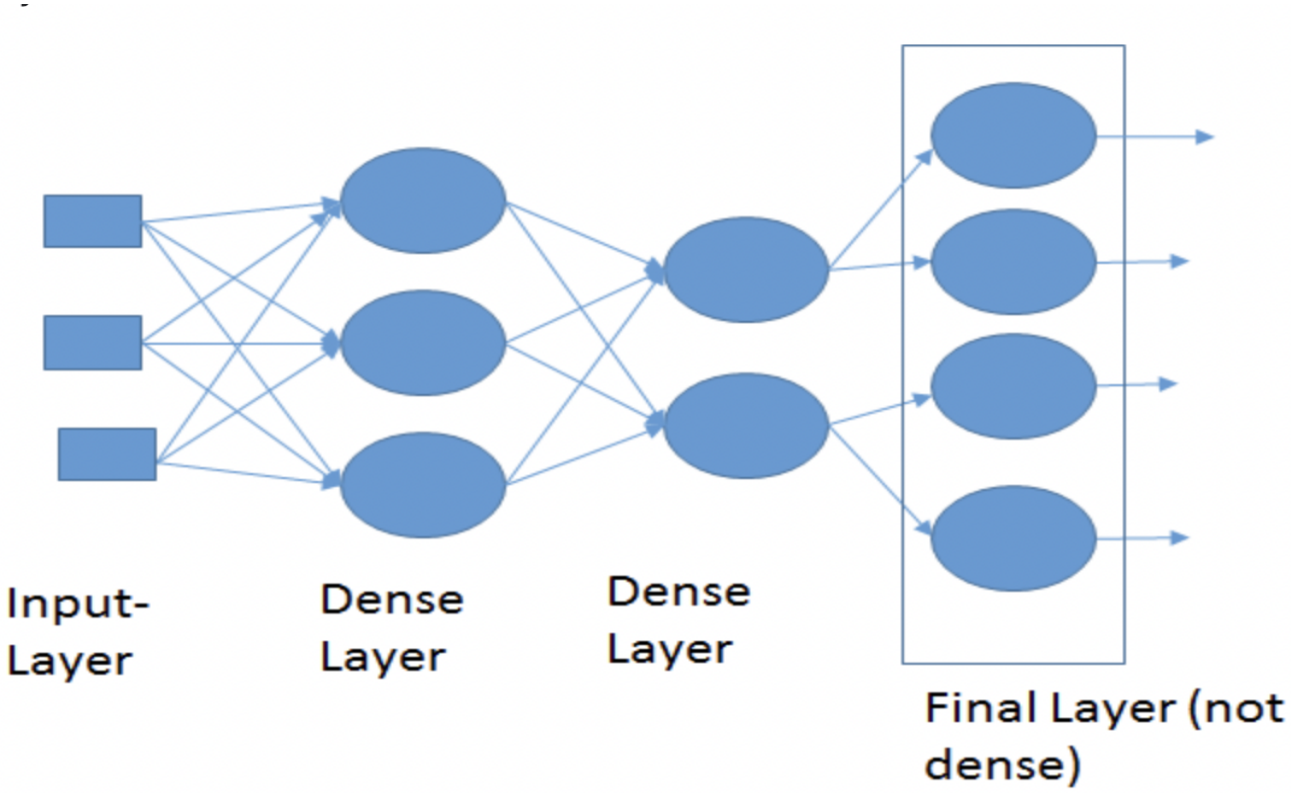
- We will use the keras's Sequential API to build our first model, train it as well as make predictions from it.
- Along the way you'll see how you can select different optimizers (like SGD) to train a neural network.
- And using different loss functions (MSE, CCE), depending on the task.

- We will also see and how to keep track of any number of metrics during the training.

By the end lecture, you'll learn everything to design, build, train and make predictions from NN models in using tf.keras module.

First look at the imports

- We import the Sequential class from `tensorflow.keras.models` in which we will "sequentially" position the layers of the NN.
- We will also import `Dense` layer from `tensorflow.keras.layers`
- A Dense layer helps us define one layer of a Feedforward NN.
- In a dense layer, each neuron in the layer is connected to all the neurons from the previous layer thus called dense




```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

- We will now create an instance (or object) of the Sequential class and call it `model`.
- Its same as creating an instance as we did using some sklearn's model class, just that, now we will now define the model as well.
- Sequential model will take a list of layers as an argument.

Lets define a feed forward network with a single hidden layer.

```
model = Sequential([
                    Dense(64, activation="relu"), #hidden dense layer with 64 neuro
                    Dense(4, activation="softmax") #output layer with 4 units and s
])
```

- The activation argument is optional, and if you don't pass it, it will be a linear (or no) activation

Question: Why do we need activation function?

- It provides non linearity to problems.

Question: Give an example of a case when we wouldn't like to pass an activation?

- Output layer of regression model

Question: If we want to find out if there is a method to check the weights of model, how will you find that method?

Use dir(model)

- From dir(model) we can find out about the weights() method for sequential models.

Lets check the model weights

```
#model.weights # should give some error
```

Looks like the model hasn't created the weights yet.

Passing input shape

- Notice that we haven't told the model about the input size yet, and thus, tensorflow doesn't have any information to create W and b yet.
- Optionally, we can pass the input size right away as well in the first layer.

```
model = Sequential([
                    Dense(64, activation="relu", input_shape=(11,)),
                    Dense(4, activation="softmax")
])
```

Question: Why didn't we define the input shape in the second dense layer?

The layers in the sequential model interact with each other therefore we don't need to define the input shape for all the layers.

- Here we are saying that one training example will be a 11-dimensional feature vector - (11,)

- For each of the feature vector from the input, there will be output of 64-dimensional feature vector - (64,)
- Incase of a dense layer Y = X.W + b, where X has all the features as columns.
- The output of first layer will have dimension (m, 64), where m is the number of input data samples.

```
type(model.weights)
```

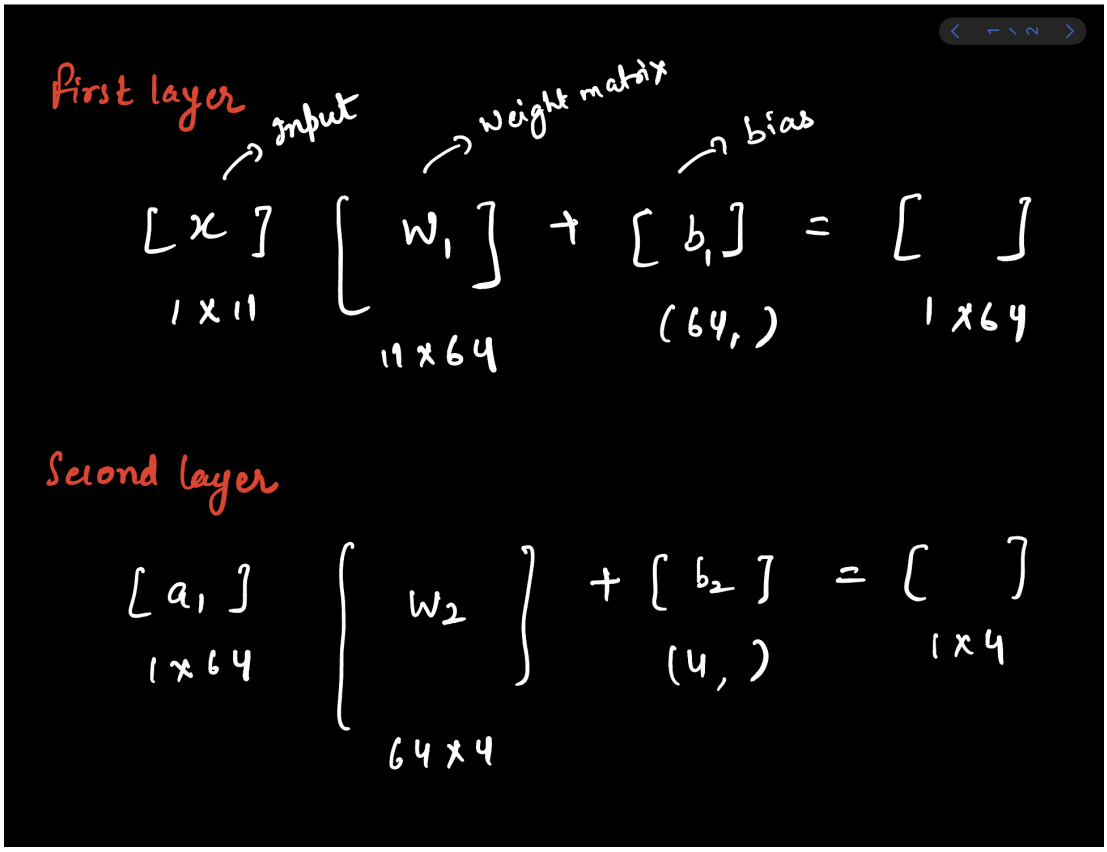
```
list
```

```
for param in model.weights:
  print(param.shape)
```

```
(11, 64)
(64,)
(64, 4)
(4,)
```

The diagram below shows how the model's outputs look like for each layer.

- For the first layer, if an observation of dimension 1x11 is passed then an output of dimension 1x64 will be generated. Similarly it will be done for m observations.
- For the second layer when an input of dimension (1,64) is passed then an output of dimension 1x4 will be generated. Similarly it will be done for m observations.



Lets look at another way to define the same model

model.add()

- Instead of passing the list of layers as an argument while creating a model instance, we can use the `add` method.

```
model = Sequential()
model.add(Dense(64, activation="relu", input_shape=(11,)))
model.add(Dense(4, activation="softmax"))
```

Question: Which method (list of layers or .add) to choose?

- It really depends on the user.
- If the model depends on a condition or may add some X number of similar layers in the loop, we might use `model.add`
- We can then check the condition using `if` and add the layer to the model later using `add`

Optional Exercise: What if the input had been a multi-dimensional data?

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten

model_2D = Sequential([
                    Flatten(input_shape=(28, 28)), # flatten the data to make it (7
                    Dense(64, activation="relu"),
                    Dense(4, activation="softmax")
])
```

Note: We can also directly pass multi-dimensional input directly for which we will study different types of networks like convolution neural network in later modules.

Quiz-3

```
Which of the following is valid model definition?

1. `Sequential([Dense(activation='relu'), Dense(activation='softmax')])`
2. `Sequential(Flatten(), Dense(16, activation='sigmoid'))`
3. `Sequential([Flatten(), Dense(32)])`
4. `Sequential([Flatten(64), Dense(32, activation='relu')])`
```

Correct answer (3)

Model summary

- There is another short way to check dimensions and parameters of each layer
- we can do it using the printing model summary

```
model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_4 (Dense)             (None, 64)                768

 dense_5 (Dense)             (None, 4)                 260

=================================================================
Total params: 1,028
Trainable params: 1,028
Non-trainable params: 0
_________________________________________________________________
```

Question: How are number of parameters computed in the above summary?

Answer:

- Input is of shape 11* (Batch Size) which is densely connected to 64 Neurons. So parameters are computed as $(11 * 64)\ weights + 64\ for\ biases = 768.$

Similarly,

- For second: $4 * 64 + 4 = 260$

Question: What does None represent in the above model's `Output Shape`?

None makes the model capable of handling the multiple points.

- In the image we saw above where we were calculating the dimensions, we have calculated the dimensions for a single example.
- But in an ideal scenario there will be multiple points. And we haven't defined the number of points while defining the model, therefore the model is keeping it 'None' for handling the multiple number of passed observations.

Giving custom names to the layers

- As you can see in the model summary, keras has provided the names by itself.
- At times, we might want to give custom names to the layer as well

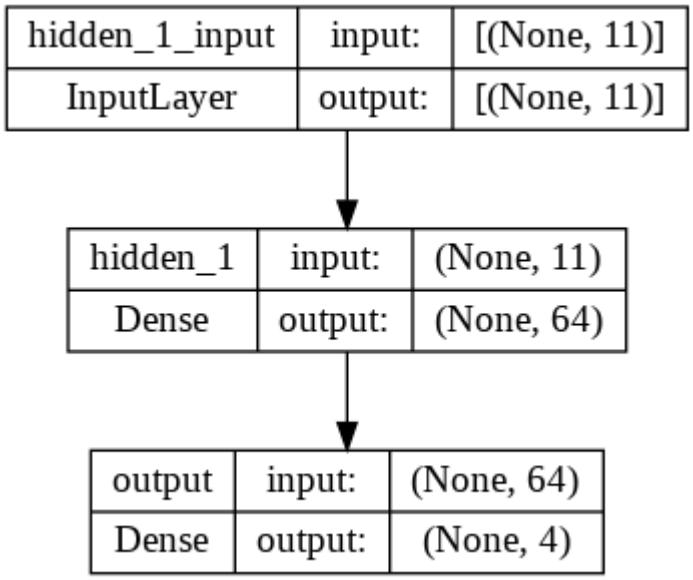
```
model = Sequential([
                    Dense(64, activation="relu", input_shape=(11,), name="hidden_1"
                    Dense(4, activation="softmax", name="output")
])
```

```
model.summary()
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_1 (Dense)            (None, 64)                768

 output (Dense)              (None, 4)                 260

=================================================================
Total params: 1,028
Trainable params: 1,028
Non-trainable params: 0
_________________________________________________________________
```

- We can also plot the model as a graph
- And save it as a png file

```
from tensorflow.keras.utils import plot_model

plot_model(model,
    to_file='model.png',
    show_shapes=True, show_layer_names=True)
```



From the plot we can see that the first layer is the input layer and then the input and output shapes of each of layers `hidden_1` and `output` are given.

Assessments Covered

- https://www.scaler.com/hire/test/problem/18024/
- https://www.scaler.com/hire/taest/problem/38183/

Weights and Bias Initializer

- When we worked on NN from scratch, we intialised W with `np.random.rand` and b with `np.zeros`.
- We also learnt about various weight initialization techniques
  - Glorot Normal

    $w_{ij}^k \sim N(0, \sigma_{ij})$, where $\sigma_{ij} = \frac{2}{fanin+fanout}$
  - Glorot Uniform

    $$w_{ij}^k \sim Uniform\left[\frac{-\sqrt{6}}{\sqrt{fanin+fanout}}, \frac{\sqrt{6}}{\sqrt{fanin+fanout}}\right]$$
  - He Normal

    $N(0, \sigma)$, where $\sigma = \frac{2}{fanin}$
  - He uniform

    $$Uniform\left[\frac{-\sqrt{6}}{\sqrt{fanin}}, \frac{\sqrt{6}}{\sqrt{fanin}}\right]$$
- Now let see how keras implements them.
- The end results of the classification and regression gets affected by the initialization of the weights and biases.
- For now, we will just look at some methods of weights and bias initialization.

In Keras, in `Dense` layer,

1. the biases are set to zero (`zeros`) by default
2. the weights are set according to `glorot_uniform`, the Glorot uniform initialiser.

For example:

- $$c = \frac{\sqrt{6}}{\sqrt{11+64}} = 0.28$$

for the first hidden layer of model as $fanin$ (input) is 11 and $fanout$(output) is 64

Note:

- There are several researches proposing different ways of randomly initialising the weights of the layers.
- But **`glorot_uniform` has been the most widely used** one in most of the Deep Learning frameworks today.

▾ What if I want to initialise my own weights and bias? May be for my own research?

- Each layer has optional arguments `kernel_initialiser` and `bias_initialiser` to set the weights and biases respectively.

```
model_X = Sequential([
                    Dense(64, activation="relu", input_shape=(11,), name="hidden_1"
                    Dense(4, activation="softmax", name="output", kernel_initialize
])
```

We can totally go from scratch as well (only if we need to) using keras.initialiser submodule

```
Dense(64, kernel_initializer=tf.keras.initializers.RandomNormal(mean=0.0, stddev=0.
          bias_initializer=tf.keras.initializers.Constant(value=0.4),
          activation='relu')
```

```
    <keras.layers.core.dense.Dense at 0x7f73df5999a0>
```

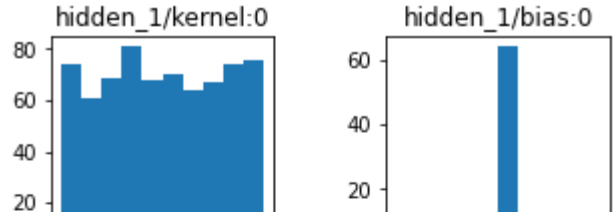
- Let's try to retrieve the layers of the model.

```
model.layers
```

```
    [<keras.layers.core.dense.Dense at 0x7f73e342ce50>,
     <keras.layers.core.dense.Dense at 0x7f73e342ce20>]
```

- Lets look at how are they initialised, we can check how the weights distribution (optional).

```
# Plot histograms of weight and bias values
import matplotlib.pyplot as plt
fig, axes = plt.subplots(2, 2, figsize=(5,5))
fig.subplots_adjust(hspace=0.5, wspace=0.5)

# get the weights from the layers
weight_layers = [layer for layer in model.layers]

for i, layer in enumerate(weight_layers):
    for j in [0, 1]:
        axes[i, j].hist(layer.weights[j].numpy().flatten(), align='left')
        axes[i, j].set_title(layer.weights[j].name)
```



- layers() method consist of a list of layers.
- In the code, we are firstly getting all the weights and biases of the layers using `layer.weights[0/1]`, then converting them to a numpy array and then flattening all the arrays to just get an array of initialized weights and biases.
- From the above plots it can be observed that all the biases are intialized to zero by default.

Compile - loss and optimizer

Until now, we have defined the model architecture.

After defining the model, we have to compile the model.

Question: What do you think, which specific information we should pass to model while compiling ?

1. Loss function - To measures the model performance as it trains
2. Optimizer (like Gradient Descent) - To performs the gradient update

We do this by using two arguments of the compile method `optimizer` and `loss`.

Lets take an example of a binary classification task here

```
model_2C = Sequential([
          Dense(64, activation="relu", input_shape=(11,)),
          Dense(1, activation="sigmoid")])

# new piece of code
model_2C.compile(
    optimizer = "adam", # stochastic gradient descent, adam, rmsprop, adadelts
    loss = "binary_crossentropy", # sigmoid loss, # mean_squared_error, categorical
    metrics = ["accuracy"]
)
```

- There are multiple options for optimisers, loss and metrics which we will discuss in detail later
- All these strings which we have passed as argument i.e. `sgd`, `binary_crossentropy` and `accuracy` are reference to some default objects defined in keras. They are called to string identifiers.

- We can also directly instantiate these custom objects here with the classes defined in keras submodules for example `opt = keras.optimizers.Adam(learning_rate=0.01)`
- We can even pass customized loss and optimizer functions in keras models.

Assessments Covered

- https://www.scaler.com/hire/test/problem/20402/

Lets change the learning rate by initialising a custom object

```
model.compile(
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.01),
    loss = tf.keras.losses.SparseCategoricalCrossentropy(),
)
```

Optionally, we can define a list of metrics which we might want to track during the training, like accuracy

```
model = Sequential([
                    Dense(16, activation="relu", input_shape=(11,), name="hidden_1"
                    Dense(8, activation="relu", name="hidden_2"),
                    Dense(4, activation="softmax", name="output")
])

model.compile(
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
    loss = tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=["accuracy"]
)
```

Other available string identifiers to pass in `loss` parameter.

How sparse categorical crossentropy different from categorical cross-entropy?

1. Use `categorical_crossentropy` if target vector is one hot-encoded - [1,0,0,0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]
2. Use `sparse_categorical_crossentropy` if target vector is ordinal integer values - 0, 1, 2, 3

- We can use these objects when we need to change their arguments like a using different learning rate.
- We can change the cut-off from 0.5-0.7 or anything using this.

- These metrics will be calculated and saved after each epoch (one pass of whole data to update the model).

But what exactly is an epoch?

- When data is too big, data is passed in small batches instead of one big batch due to memory constraints. Each pass of mini-batch is called an iteration.
- Each pass of whole datasets is called an Epoch.
- One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters.

Lets finally check if the loss and optimisers for the model are set. We can use the model attributes for getting these.

```
model.loss
```

```
    <keras.losses.SparseCategoricalCrossentropy at 0x7f73df4bdfd0>
```

```
model.optimizer
```

```
    <keras.optimizers.optimizer_v2.adam.Adam at 0x7f73df4bdd90>
```

Assessments Covered

- https://www.scaler.com/hire/test/problem/38153/
-

Training the model using `fit` method

Training the model would mean updating the weights using the optimizer and loss functions on the dataset.

```
model.fit(X_train, y_train)
```

X_train = (num_samples, num_features)

y_train = (num_samples, num_classes) or y_train = (num_samples,)

Lets look at some other arguments of model.fit method

Set the epochs

You might want to train the model for more than 1 epochs.

```
model.fit(X_train, y_train, epochs=500)
```

Set the batch size

- Another option is to set the batch size, the default is 16.
- Usually, the batch size used are of the form 2^x - 4, 8, 16, 32, 64
- And we try to take as big of a bach size as GPU memory can manage.
- Setting up batch_size=16 means that 16 training samples are passed in single iteration.
- Num_iterations in an epoch = num_samples/batch_size

There are some other sophisticated arguments of `model.fit` which we will dicuss later as we progress.

Lets train our model!

Here we have mentioned `validation_split = 0.1,`
which means 10% of our training data will be used for validation

- Seed in ML means intilization state of a pseudo random number generator.
- After setting the same seed whenever random() function is called we will get a specific way of generating numbers.
- If you use the same seed you will get exactly the same pattern of numbers during weights initializations, which will be helpful for us as if all of us run the code we will get the same output.

```
def seed_everything(seed = 42):
    np.random.seed(seed)
    tf.random.set_seed(seed)


seed_everything()
```

Let's train for few epochs first and see what happens

```
%%time
model.fit(X_train, y_train, epochs=10, batch_size=256, validation_split=0.1, verbos

## no of iterations: ( 10847 (training size) - 1084.7 (validation split) )/(256) ==

     Epoch 1/10
     39/39 [==============================] - 1s 8ms/step - loss: 1.3526 - accuracy
     Epoch 2/10
     39/39 [==============================] - 0s 3ms/step - loss: 1.2592 - accuracy
     Epoch 3/10
     39/39 [==============================] - 0s 3ms/step - loss: 1.1722 - accuracy
     Epoch 4/10
     39/39 [==============================] - 0s 3ms/step - loss: 1.0951 - accuracy
     Epoch 5/10
     39/39 [==============================] - 0s 3ms/step - loss: 1.0361 - accuracy
     Epoch 6/10
```

```
39/39 [==============================] - 0s 3ms/step - loss: 0.9892 - accuracy
Epoch 7/10
39/39 [==============================] - 0s 3ms/step - loss: 0.9516 - accuracy
Epoch 8/10
39/39 [==============================] - 0s 3ms/step - loss: 0.9219 - accuracy
Epoch 9/10
39/39 [==============================] - 0s 3ms/step - loss: 0.8989 - accuracy
Epoch 10/10
39/39 [==============================] - 0s 3ms/step - loss: 0.8815 - accuracy
CPU times: user 2.08 s, sys: 108 ms, total: 2.19 s
Wall time: 3.29 s
<keras.callbacks.History at 0x7f73df462bb0>
```

Observe

- Here we trained our model for 10 epochs
- `model.fit` is printing all the metrics like accuracy, loss, validation loss, validation accuracy etc.

Question: How can we use all this information and use it for analyising training process ?

- For this let's check what this `model.fit` is returning

History

- `model.fit` returns a history object which contains the record of progress NN training.
- History object contains records of loss and metrics values for each epoch.
- History object is an example of something called "callback" (will study it later).

Compiling and training the model

- Until now, we have seen how to build a NN model, compile it with relevant loss function, optimizer and metrics to track while training.
- We also saw how we can train the model using the fit method for certain number of epochs and with certain batch_size.

Let's train model for 500 epochs and store training process inside a variable called **history**

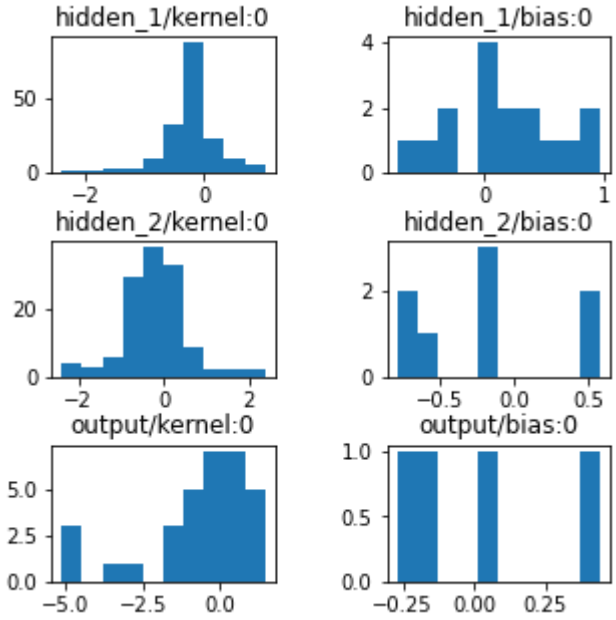
```
%%time
history = model.fit(X_train, y_train, epochs=500, batch_size=256, validation_split=

## no of iterations: ( 10847 (training size) - 1084.7 (validation split) )/(256) ==
```

```
39/39 [==============================] - 0s 3ms/step - loss: 0.7291 - accuracy
Epoch 27/500
39/39 [==============================] - 0s 3ms/step - loss: 0.7247 - accuracy
Epoch 28/500
39/39 [==============================] - 0s 3ms/step - loss: 0.7200 - accuracy
Epoch 29/500
39/39 [==============================] - 0s 3ms/step - loss: 0.7164 - accuracy
Epoch 30/500
39/39 [==============================] - 0s 3ms/step - loss: 0.7125 - accuracy
Epoch 31/500
39/39 [==============================] - 0s 3ms/step - loss: 0.7085 - accuracy
Epoch 32/500
39/39 [==============================] - 0s 3ms/step - loss: 0.7052 - accuracy
Epoch 33/500
39/39 [==============================] - 0s 3ms/step - loss: 0.7016 - accuracy
Epoch 34/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6981 - accuracy
Epoch 35/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6946 - accuracy
Epoch 36/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6918 - accuracy
Epoch 37/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6886 - accuracy
Epoch 38/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6862 - accuracy
Epoch 39/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6831 - accuracy
Epoch 40/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6810 - accuracy
Epoch 41/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6785 - accuracy
Epoch 42/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6760 - accuracy
Epoch 43/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6740 - accuracy
Epoch 44/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6717 - accuracy
Epoch 45/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6697 - accuracy
Epoch 46/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6676 - accuracy
Epoch 47/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6669 - accuracy
Epoch 48/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6647 - accuracy
Epoch 49/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6626 - accuracy
Epoch 50/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6608 - accuracy
Epoch 51/500
39/39 [==============================] - 0s 3ms/step - loss: 0.6597 - accuracy
Epoch 52/500
```

- Lets try to plot histograms of model weights and baises to see if there is any difference after training
- **[Q] What will be the distributions now? Still glorot_uniform ?**
    - Weights now follow normal distribution

- Biases are not Zero now

```
# Plot histograms of weight and bias values after training
import matplotlib.pyplot as plt
fig, axes = plt.subplots(3, 2, figsize=(5,5))
fig.subplots_adjust(hspace=0.5, wspace=0.5)

# get the weights from the layers
weight_layers = [layer for layer in model.layers]

for i, layer in enumerate(weight_layers):
    for j in [0, 1]:
        axes[i, j].hist(layer.weights[j].numpy().flatten(), align='left')
        axes[i, j].set_title(layer.weights[j].name)
```



Question: Is one iteration / step is one forward and then one back propagation?

Answer:

- Yes, that is correct.

validation_data

- We can also explicitly use a validation set (which was earlier created while splitting the data) using `validation_data` argument in `fit` method.
- But, we will have to re-initialise the model after defining with `validation_data`, else, the model will start getting trained from its current stage.
- Ideally, we should have written this code as a function, so that we don't have to write model definition again and again

```
def create_model():
    model = Sequential([
```

```
                    Dense(32, activation="relu", input_shape=(11,), name="hidden_1"
                    Dense(16, activation="relu", name="hidden_2"),
                    Dense(4, activation="softmax", name="output")])
    model.compile(
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001),
    loss = tf.keras.losses.SparseCategoricalCrossentropy(),
    metrics=["accuracy"])
    return model

model = create_model()
```

Also, this time, lets change the `verbose=0` to make the training process silent. This will prevent the printing of loss and other metrics for each epoch.

```
history = model.fit(X_train, y_train, validation_data = (X_val, y_val),  epochs=500
```

Lets look at the history object dictionary. It's an alternative to dir(). `__dict__` attribute can be used to retrieve all the keys associated with the object on which it is called.

```
history.__dict__.keys()
```

```
    dict_keys(['validation_data', 'model', '_chief_worker_only',
    '_supports_tf_logs', 'history', 'params', 'epoch'])
```

We can see that history object's dictionary has another dictionary with key "history" inside it

We can also call the `history` method of the above object as it is available in keys associated with the `model.fit()`.

```
history.history.keys()
```

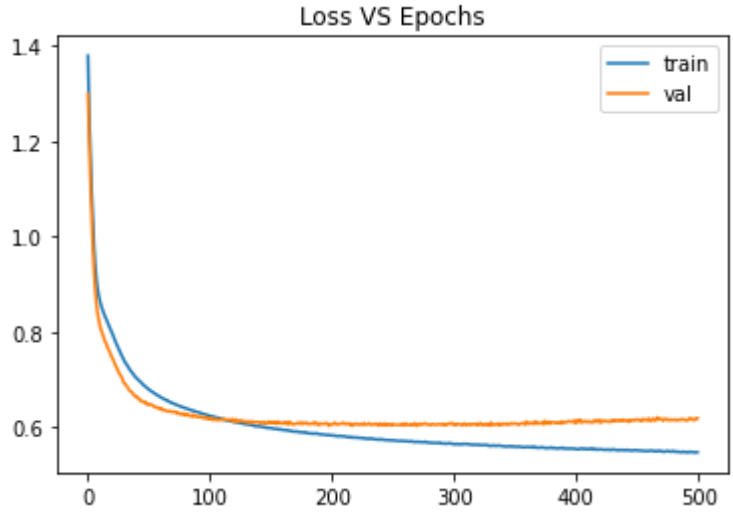
```
    dict_keys(['loss', 'accuracy', 'val_loss', 'val_accuracy'])
```

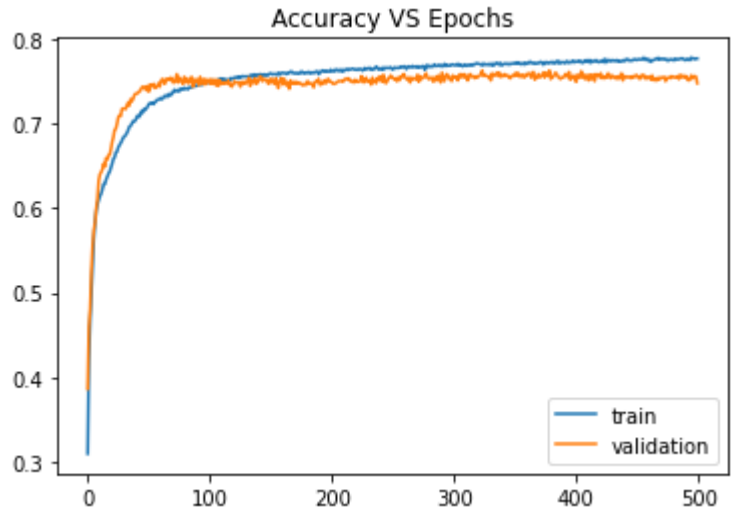
model.fit() has saved all the loss and metrics values for each epoch inside the `history` dictionary where all the values are stored in different lists.

```
epochs = history.epoch
loss = history.history["loss"]
accuracy = history.history["accuracy"]
val_loss = history.history["val_loss"]
val_accuracy = history.history["val_accuracy"]
```

Lets plot the loss and accuracy curves for both training and validation data

```
plt.figure()
plt.plot(epochs, loss, label="train")
plt.plot(epochs, val_loss, label="val")
plt.legend()
plt.title("Loss VS Epochs")
plt.show()

plt.figure()
plt.plot(epochs, accuracy, label="train")
plt.plot(epochs, val_accuracy, label="validation")
plt.legend()
plt.title("Accuracy VS Epochs")
plt.show()
```





- We can see that both training and validation loss decrease with epochs.
- After around 120 epochs, training loss still keeps on decreasing, but validation loss starts to increase.
- This means, **the model starts overfitting the training dataset after 120 epochs**.
- Params learnt after 120 epochs would be the one which are overfitting the training dataset.
- Later, **we will see ways to save intermediate model parameters** - **early stopping**.
- But for now, lets evaluate the model performance based on 500 epochs trained model only.

Assessments Covered

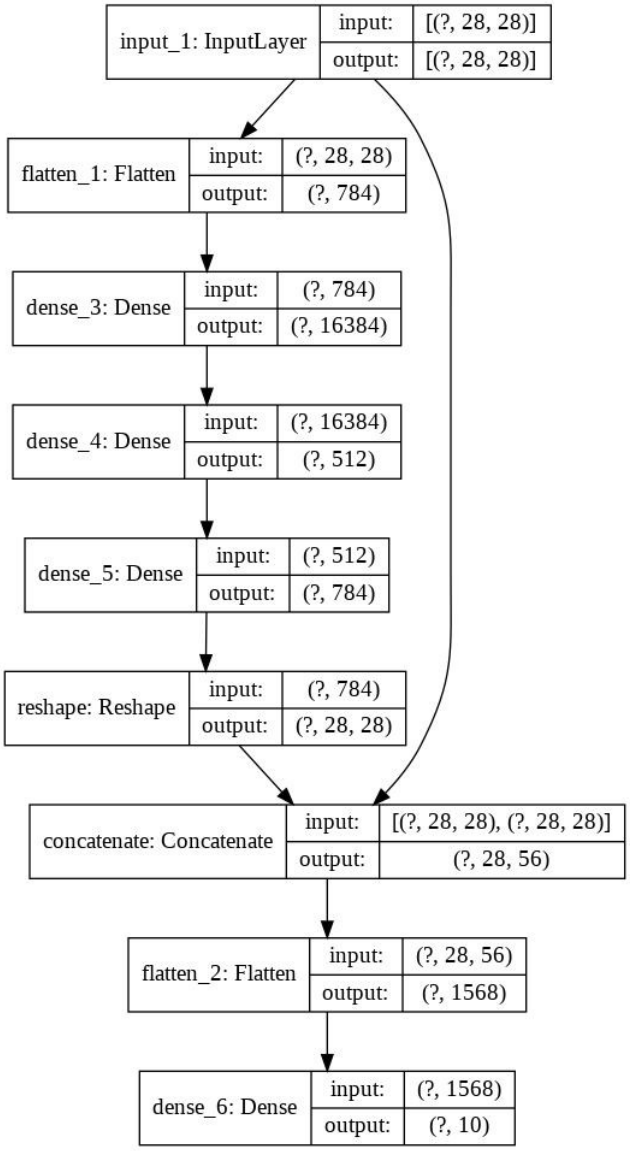
- https://www.scaler.com/hire/test/problem/20092/
- https://www.scaler.com/hire/test/problem/38182/
- https://www.scaler.com/hire/test/problem/20310/
- https://www.scaler.com/hire/test/problem/38313/
- https://www.scaler.com/hire/test/problem/38184/

Keras Functional API

Look at this complex model

Question: Can you design this model using Keras Sequential API ?

- No, there is no way to pass two inputs to one layer in Sequential API
- We use another API of Keras called as Functional API to design such complex models

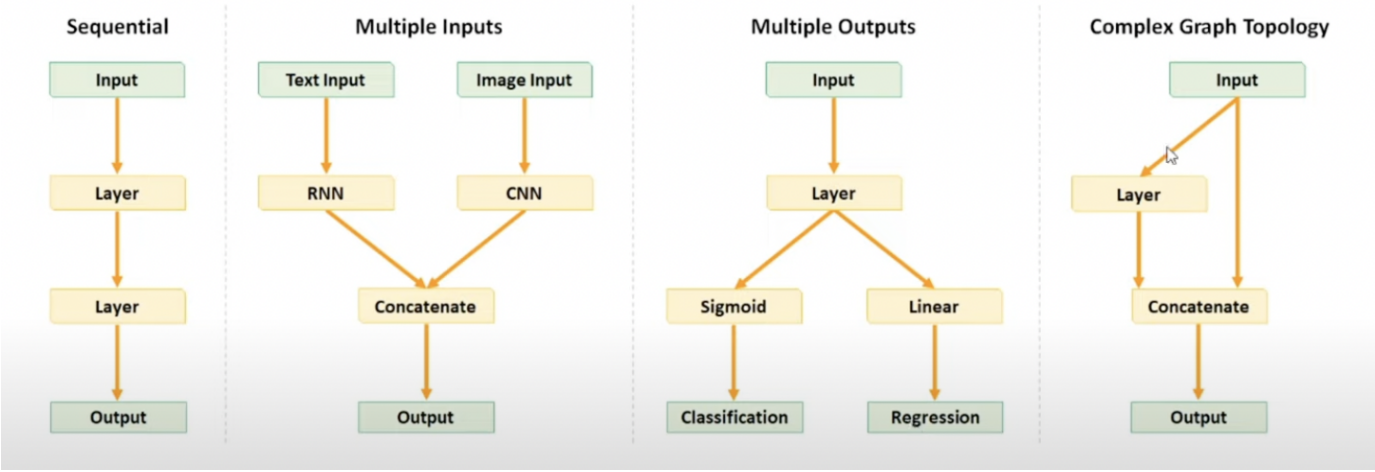


Why do we need Functional API instead of Sequential API?

- Functional API gives us **more flexibility**
- This API can handle **multiple inputs and outputs**
- lets say we have an image and a text description as our training data
- Or we want model to output two or more target variables
- Ex- A weather forecast model predicting Min & Max temp at the same time
- Sequential API wont be able to do this

Look at this complex model design, it can not be created using Sequential API but easily by Functional API

- Multiple inputs to one layer
- Multiple outputs of one layer



- So functional api gives us more **flexibility for network architectures**
- architectures are not always in sequential manner
- we can have two layers in parallel

- It is always recommend to **use the simplest method while building networks**
- Simple models can be easily built with Sequential API
- But sometimes we need the flexibility

> INSTRUCTOR NOTES
> Instructor may go to this link and further explain Functional API in detail

```
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import Sequential
```

```
model_sequential = Sequential([
                    Dense(16, activation="relu", input_shape=(11,), name="hidden_1"
                    Dense(8, activation="relu", name="hidden_2"),
                    Dense(4, activation="softmax", name="output")
])
```

- Earlier we have created this model using Sequential API
- Now lets create same model using Functional API

```
from tensorflow.keras.layers import Input
```

- In Sequential we passed input shape in the first layer
- But Here we will be using an additional layer : **Input layer**
- we're going to explicitly have a separate layer to represent the data input.

- Another main diference is `tf.keras.models.Model`
- Here instead of Sequential, here **we will be using Model Class**

Keras Functional Model Class

- First we create an input layer with the shape of the dataframe

```
inp = Input(shape=(11, 1))
```

- Next we will be creating our first two layer of the model
- Instead of creating a list (as in Sequential)
- We will also pass previous layer in the current layer
- In first dense `hidden_1` we will pass `inp`
- And in second `hidden_1` we will pass `hidden_2`

```
h1 = Dense(16, activation="relu", name="hidden_1")(inp)
h2 = Dense(4, activation="relu", name="hidden_2")(h1)
```

- Now we will create the final output layer

```
out = Dense(4, activation="softmax", name="output")(h2)
```

- We have defined the flow of the model
- Finally, to built a model using this directed graph
- We will use `tf.keras.models.Model`, and pass all the inputs and outputs

```
from tensorflow.keras.models import Model

model_functional = Model(inputs=inp, outputs=out, name="simple_nn")
```

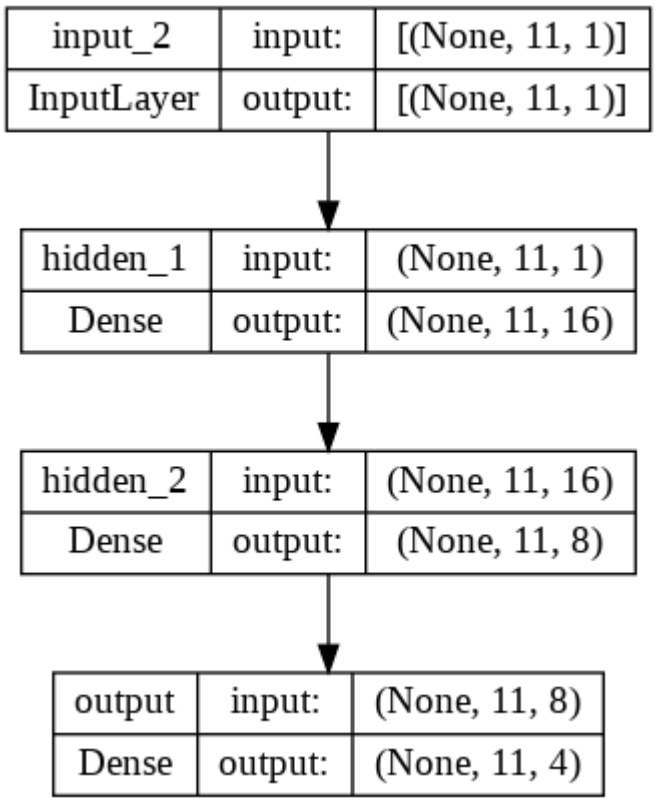
Let's generalize above code inside a function

```
def create_model_functional():

  inp = Input(shape=(11, 1))

  h1 = Dense(16, activation="relu", name="hidden_1")(inp)
  h2 = Dense(8 , activation="relu", name="hidden_2")(h1)

  out = Dense(4, activation="softmax", name="output")(h2)
```

```
  model = Model(inputs=inp, outputs=out, name="simple_nn")
  return model


model_functional = create_model_functional()


model_functional.summary()
```

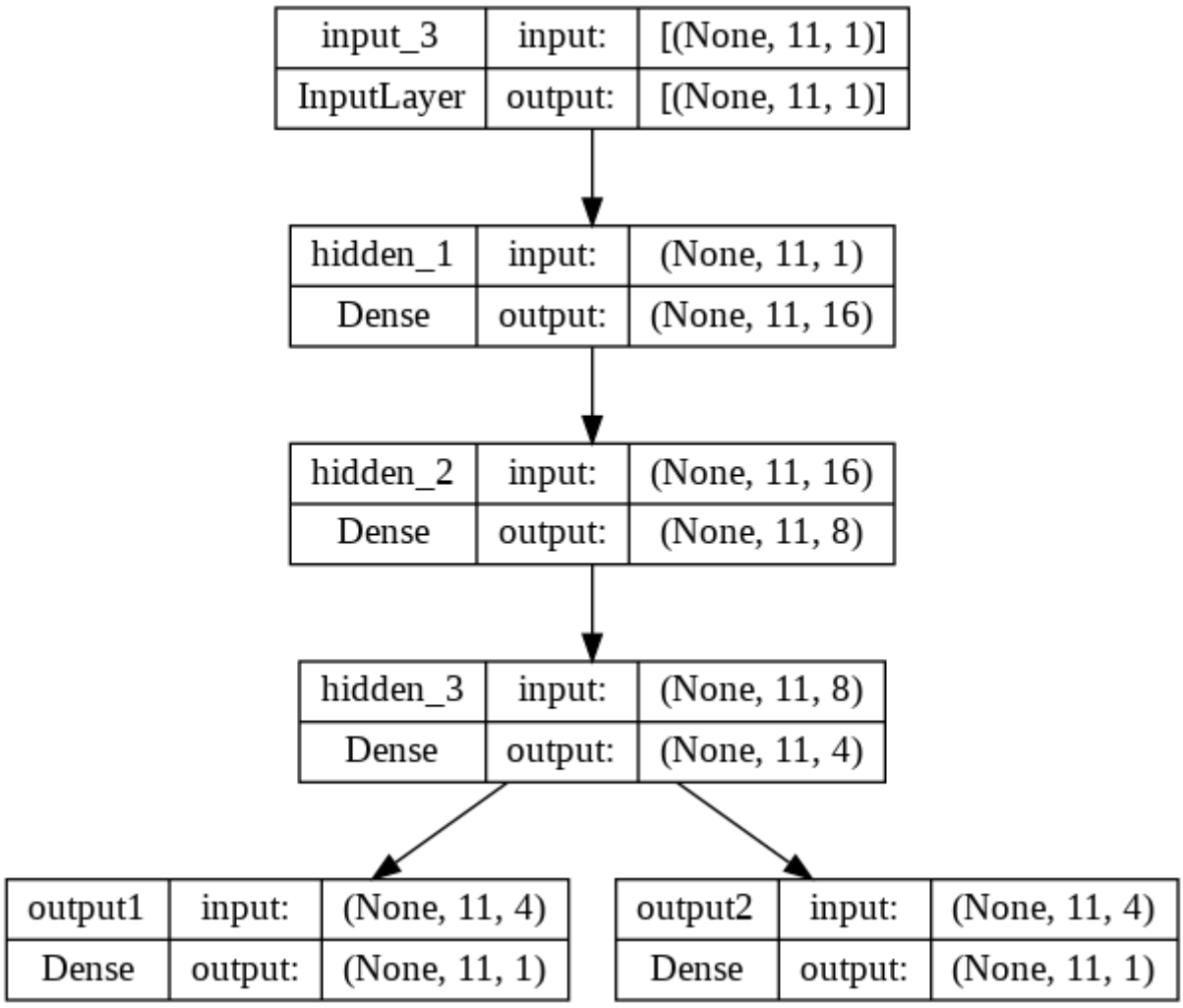
```
Model: "simple_nn"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 11, 1)]           0

 hidden_1 (Dense)            (None, 11, 16)            32

 hidden_2 (Dense)            (None, 11, 8)             136

 output (Dense)              (None, 11, 4)             36

=================================================================
Total params: 204
Trainable params: 204
Non-trainable params: 0
_________________________________________________________________
```

```
#And, optionally, display the input and output shapes of each layer in the plotted

tf.keras.utils.plot_model(model_functional,show_shapes=True)
```



> **Optional code**

```
# let's create a little complex model using functional API's with more than one out

def create_model_multiple_output():
  inp = Input(shape=(11, 1))

  h1 = Dense(16, activation="relu", name="hidden_1")(inp)
  h2 = Dense(8 , activation="relu", name="hidden_2")(h1)
  h3 = Dense(4 , activation="relu", name="hidden_3")(h2)

  out1 = Dense(1, activation="sigmoid", name="output1")(h3)
  out2 = Dense(1, activation="relu", name="output2")(h3)

  model = Model(inputs=inp, outputs=[out1,out2], name="simple_nn")
  return model

# creating model with multiple output
model_multiple_output = create_model_multiple_output()

# plotting model with multiple output
tf.keras.utils.plot_model(model_multiple_output,show_shapes=True)
```



Assessments Covered

- https://www.scaler.com/hire/test/problem/38193/

Prediction and Evaluation

Lets evaluate the model performance on different sets, we will carry out all predictions using the model created via sequential API

Evaluate the model

```
model.evaluate(X_test, y_test)
```

- model.evaluate **returns the loss value & metrics value** for the model.
- It is important to note that, **weights/parameters are not updated during evaluation** (and prediction)
- This also means - **only forward pass, no backward pass**

Now lets evalute our models on training, validation and testing datasets

```
loss, accuracy = model.evaluate(X_train, y_train)
print('Train Set')
print("Loss value : ", loss)
print("Accuracy   : ", accuracy)
```

```
339/339 [==============================] - 1s 2ms/step - loss: 0.5458 - accura
Train Set
Loss value :  0.5458030700683594
Accuracy   :  0.7781875133514404
```

```
loss, accuracy = model.evaluate(X_val, y_val)
print('Validation Set')
print("Loss value : ", loss)
print("Accuracy   : ", accuracy)
```

```
38/38 [==============================] - 0s 2ms/step - loss: 0.6194 - accuracy
Validation Set
Loss value :  0.6194478869438171
Accuracy   :  0.7479270100593567
```

```
loss, accuracy = model.evaluate(X_test, y_test)
print('Test Set')
print("Loss value : ", loss)
print("Accuracy   : ", accuracy)
```

```
42/42 [==============================] - 0s 2ms/step - loss: 0.5842 - accuracy
Test Set
Loss value :  0.5841851830482483
Accuracy   :  0.7634328603744507
```

Predictions

- If we need to get the prediction for an **unseen data**, we can use the `predict` method to get the predictions. `model.predict(X_test)`
- It **returns raw output** from the model (i.e. **probabilities** of an observation belong to each one of the 4 class)

```
pred = model.predict(X_test)
pred
```

```
42/42 [==============================] - 0s 1ms/step
array([[9.99999940e-01, 2.10884883e-17, 1.79274864e-33, 0.00000000e+00],
       [1.13163900e-03, 1.37660122e-02, 1.01408757e-01, 8.83693516e-01],
       [1.93726644e-02, 2.39155009e-01, 2.92279452e-01, 4.49192822e-01],
       ...,
       [1.64477840e-01, 8.35509479e-01, 1.26833165e-05, 9.64229950e-15],
       [6.74094200e-01, 3.25786144e-01, 1.19630786e-04, 1.66241088e-12],
       [1.01807564e-02, 2.50488132e-01, 7.39329159e-01, 2.02724095e-06]],
      dtype=float32)
```

What will be the sum of probabilities of an observation belong to each of the 4 classes?

- 1
- As the model outputs the probabilities for all the observations and each observation can only belong to one of these 4 classes, therefore the sum of probs will be 1.

```
np.sum(pred, axis=1)
```

```
array([0.99999994, 0.99999994, 0.99999994, ..., 1.        , 0.99999994,
       1.0000001 ], dtype=float32)
```

How can we know the class an observation belongs to, using these 4 probability values?

We can find the index having the largest probability and that will be the predicted class.

```
pred_class = np.argmax(pred, axis = 1)
pred_class
```

```
array([0, 3, 3, ..., 1, 0, 2])
```

Now, to cross-check, we will again check accuracy of the model using sklearn's `accuracy_score`.

```
from sklearn.metrics import accuracy_score

acc_score = accuracy_score(y_test, pred_class)
```

```
print("Test acc: ", acc_score)
```

```
Test acc:  0.7634328358208955
```

Assessments Covered

- https://www.scaler.com/hire/test/problem/38181/

Callbacks

Earlier, we saw

- with verbose=1, model training prints associated data after every epoch
- with verbose=0, model training prints nothing

What if we want to customise the printing behaviour?

Let's use something called **callbacks** to do this.

But, first what are callbacks?

A callback **defines a set of functions** which are **excuted at different stages of the training** procedure.
For example: A callback function may run

1. function `on_epoch_begin` before every epoch
2. function `on_epoch_end` after every epoch and such multiple functions.

How can these callbacks be useful?

They can be **used to view internal states of the model** during training.

- For example, we may **want to print loss, accuracy or lr every 2000th epoch**.
- For this, may be we can **add a condition like `if epoch%2000==0`** then only do certain task, example, printing loss, accuracy or lr

Lets create our own customized callback class for printing Loss and Acc for every 50th epoch

- The custom class will inherit from `tf.keras.callbacks.Callback`.
- Which means that all the attributes and methods available in the keras.callbacks.callback class will be available for our customized class, and we can also override them.

```
class VerboseCallback(tf.keras.callbacks.Callback):
  # runs only before the training starts
  def on_train_begin(self, logs=None):
        print("Starting training...")
```

```
  # runs after every epoch
  def on_epoch_end(self, epoch, logs = None):
    if epoch % 50 == 0:
       print(f'Epoch {str(epoch).zfill(3)}', '- loss : ', logs['loss'], '- Acc : ',

  # runs once training is finished
  def on_train_end(self, logs=None):
        print("...Finished training")
```

In the above code snippet we are printing a sentence at the start of training and also printing the loss and accuracy for each 50th epoch's end. We are also printing a sentence at the end of training.

- In the code snippet `logs` is the dictionary that callback methods take as an argument that will consist of the keys for quantities relevant to the current batch or epoch like loss, accuracy etc.

- We will have to **pass a list of callback objects** to `callbacks` argument of the fit method
- (Optional) We **can pass callback objects to evaluate and predict** method as well.

```
history = model.fit(X_train, y_train, epochs=500, batch_size=256, validation_split=
```

```
Starting training...
Epoch 000 - loss :  0.5516670346260071 - Acc :  0.7737144231796265
Epoch 050 - loss :  0.5456981658935547 - Acc :  0.7760704755783081
Epoch 100 - loss :  0.5408483743667603 - Acc :  0.776172935962677
Epoch 150 - loss :  0.5364638566970825 - Acc :  0.7808850407600403
Epoch 200 - loss :  0.5334064364433289 - Acc :  0.7805777788162231
Epoch 250 - loss :  0.530765175819397 - Acc :  0.7842655181884766
Epoch 300 - loss :  0.5280439853668213 - Acc :  0.7856996655464172
Epoch 350 - loss :  0.524202823638916 - Acc :  0.7886703610420227
Epoch 400 - loss :  0.5229038000106812 - Acc :  0.7881581783294678
Epoch 450 - loss :  0.5198482275009155 - Acc :  0.7865191698074341
...Finished training
```

The parent class `tf.keras.callbacks.Callback` supports various kinds of methods which we can override

- Global methods
  at the beginning/ending of training
- Batch-level method
  at the beginning/ending of a batch
- Epoch-level method
  at the beginning/ending of an epoch

Example -

```
class TrainingCallback(tf.keras.callbacks.Callback):

    def on_train_begin(self, logs=None):
        print("Starting training...")

    def on_epoch_begin(self, epoch, logs=None):
        print(f"Starting epoch {epoch}")

    def on_train_batch_begin(self, batch, logs=None):
        print(f"Training: Starting batch {batch}")

    def on_train_batch_end(self, batch, logs=None):
        print(f"Training: Finished batch {batch}, loss is {logs['loss']}")

    def on_epoch_end(self, epoch, logs=None):
        print(f"Finished epoch {epoch}, loss is {logs['loss']}, accuracy is {logs['accurac

    def on_train_end(self, logs=None):
        print("Finished training")
```

- There are some pre-defined callback classes such as CSVLogger, EarlyStopping, LearningRateScheduler.
- They can be useful for various tasks e.g..., EarlyStopping can be used to stop the training as soon as the validation loss starts to increase.

Other examples include:

1. CSVLogger - save history object in a csv file `csv_logger = keras.callbacks.CSVLogger("file_name.csv")`
2. EarlyStopping - stop the training when model starts to overfit
3. ModelCheckpoint - saves the intermediate model weights
4. LearningRateScheduler - control/change Learning Rate in between epochs

we will use them in the later lectures

So far we have learnt about the following steps for using a model:

1. Defining a model.
2. Compiling the model using `model.compile()` with `loss` and `optimizers`
3. Training the model with `model.fit()`.
4. Predicting using a model through `model.predict()`.
5. Evaluating the performance using `model.evalulate()`.
6. Using callbacks for performing certain tasks during different stages of training process.

Assessments Covered

- https://www.scaler.com/hire/test/problem/21573/
- https://www.scaler.com/hire/test/problem/38154/
- https://www.scaler.com/hire/test/problem/21017/
- https://www.scaler.com/hire/test/problem/20860/

Tensorboard

- It is always a **good practice to closely monitor the training process** - changes in the loss, performance, changes in the parameters.
- We do get the values of these attributes using `history` but we need to explicitly plot them to visualize.
- Lets learn about a **handy tool which we can use to visually track metrics, weights**

Tensorboard is a dashboard that allows us to visualise information regarding the training process like

- Metrics - loss, accuracy
- Visualise the model graphs
- Histograms of W, b, or other tensors as they change during training - distributions
- Displaying images, text, and audio data (in later lectures)

Start Tensorboard

- We **don't need to install** tensorboard in **colab**.
- Else, we can do that using one of the possibilities

```
pip install tensorboard
conda install -c conda-forge tensorboard
```

We will directly load the Tensorboard to Colab.

Load tensorboard in the notebook.

```
%load_ext tensorboard
```

- Now we have have to set a log directory, say `logs`

```
log_folder = 'logs'
```

- TensorBoard will **store all the logs in this log directory**.

- It will read from these logs in order to display the various visualizations.
- If we want to reload the TensorBoard extension, we can use the reload magic method

```
%reload_ext tensorboard
```

- Before saving the new logs to the folder, we might want to clear the current logs (if there are any).

```
!rm -rf logs
```

Inorder to use Tensorboard, we need to import `tf.keras.callbacks.TensorBoard`

```
from tensorflow.keras.callbacks import TensorBoard
```

Callback arguments:

- **log_dir** - (Path)
  - directory where logs will be saved
  - This directory should not be used by any other callbacks.
- **update_freq** - (int/str)
  - how frequently losses/metrics are updated.
  - when set to `batch`, lossess/ metrics will be updated after every batch/iteration
  - when set to an integer `N`, updation will be done after `N` batches
  - when set to 'epoch', updation will be done after every epoch
- **histogram_freq** - (int)
  - how frequently (in epochs) histograms(Distribution of W) will be updated.
  - Setting this to 0 means, histograms will not be computed.
- **write_graph** - (Bool), `True` if we want to visualize our training process
- **write_images** - (Bool), `True` if we want to visualize our model weights

```
tb_callback = TensorBoard(log_dir=log_folder, histogram_freq=1)
```

Now we will train our model again.
This time using Tensorboard callback

```
model = create_model()
history = model.fit(X_train, y_train, epochs=500, batch_size=512, validation_data =
```

Let us now look at the various tabs on the TensorBoard dashboard.

- **Scalars**: shows loss and metrics
- **Graphs** : shows model training structure
- **distributions** : distribution of W & b
- **histograms** : histogram of W & b over epoch

Launch tensorboard using the following command line.

```
%tensorboard --logdir={log_folder}
```

403. That’s an error.

That’s all we know.