# Tensorflow and Keras 1

## Tensorflow

TensorFlow is the premier open-source deep learning framework developed and maintained by Google

## Keras

- Using TensorFlow directly can be challenging,
- In TensorFlow 2, Keras has become the default high-level API
- No need to separately install keras
- The modern tf.keras API brings Keras's simplicity and ease of use to the TensorFlow project.

## Why keras?

- easy to build and use giant deep learning models
- **Light-weight and quick**
- can support other backends as well besides tensorflow, eg: Theano
- Open source

## Ways of writing code in Keras

- Sequential API
- Functional API

## Keras Sequential API

- The simplest and recommended API to start with
- Called as "Sequential"  because we add layers to the model one by one in a
- linear manner, from input to output.
- You can select optimizers, loss functions, and metrics while writing code

The dense layer helps us define one layer of a Feedforward NN.

Example model using Sequential API

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

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

We can check model weights using model.weights

## 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"))
```

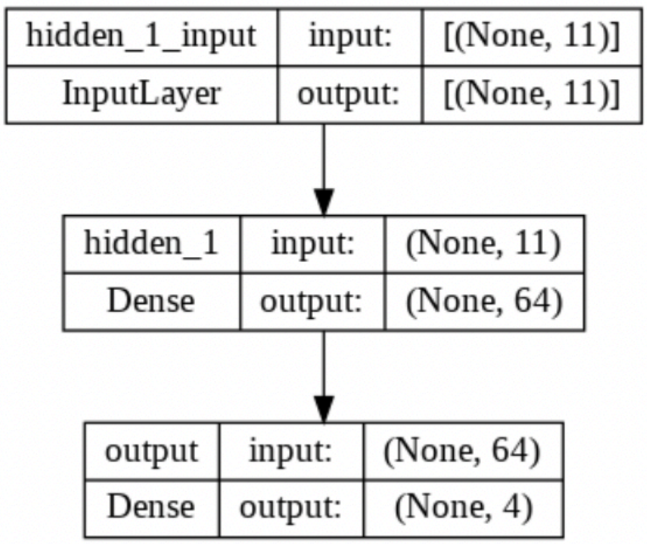
## model.summary()

To print the summary of the model we have created

```
[ ] 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
_________________________________________________________________
```

## Plotting model

```python
from tensorflow.keras.utils import plot_model

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




## Weights and Bias Initializer

in the Dense layer,

1) the biases are set to zero (zeros) by default
2) the weights are set according to glorot_uniform

For our own custom initializer, we can use **bias_initialiser** and **kernel_initialiser**

## Compiler - loss and optimizer

What things to decide while compiling the model?

- Loss
- Optimizer

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

```
[ ]  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_crossentropy,
                                       #sparse_categorical_crossentropy, binary_crossentropy
         metrics = ["accuracy"]
     )
```

- Another way to define **optimizer = keras.optimizers.Adam(learning_rate=0.01)**
- We can also pass customized loss and optimizer functions in keras models.
- These metrics will be calculated and saved after each epoch (one pass of whole data to update the model).

## Epoch

- To avoid memory issues, data is passed in small batches instead of whole
- Each pass of the 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.

## Training: model.fit ()

It means 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, )
```

Arguments we can pass to this method

- epochs ( number of epochs you want to train for )
- batch_size ( Batch size usually in form of $2^x$ like 4,8,16,32 )
- Validation_split ( size of validation data )
- verbose ( 0 for silent training, 1 to print each iteration )

```
%%time
model.fit(X_train, y_train, epochs=10, batch_size=256, validation_split=0.1, verbose=1)

## no of iterations: ( 10847 (training size) - 1084.7 (validation split) )/(256) == 39
```

```
Epoch 1/10
39/39 [==============================] - 1s 8ms/step - loss: 1.3526 - accuracy: 0.3514 - val_loss: 1.2917 - val_accuracy: 0.4184
Epoch 2/10
39/39 [==============================] - 0s 3ms/step - loss: 1.2592 - accuracy: 0.4327 - val_loss: 1.2054 - val_accuracy: 0.4525
Epoch 3/10
39/39 [==============================] - 0s 3ms/step - loss: 1.1722 - accuracy: 0.4666 - val_loss: 1.1224 - val_accuracy: 0.4876
Epoch 4/10
39/39 [==============================] - 0s 3ms/step - loss: 1.0951 - accuracy: 0.4942 - val_loss: 1.0554 - val_accuracy: 0.5051
Epoch 5/10
39/39 [==============================] - 0s 3ms/step - loss: 1.0361 - accuracy: 0.5233 - val_loss: 1.0032 - val_accuracy: 0.5318
```

**Note**: After training, weights now will follow normal distribution and biases will not be Zero now

**History object**

- 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.
- 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 = model.fit(X_train, y_train, validation_data = (X_val, y_val),  epochs=500, batch_size=512, verbose=0)
history.__dict__.keys()
```

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

The model 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"]
```

# Prediction and Evaluation

## Evaluate the model

loss, accuracy = model.evaluate(X_test, y_test)

- model.evaluate **returns the loss value & metrics value** for the model.

- **weights/parameters are not updated during evaluation** (and prediction) which means **only forward pass, no backward pass**

```
[ ] 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: 0.7634
    Test Set
    Loss value :  0.5841851830482483
    Accuracy   :  0.7634328603744507
```

## Predictions

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

```
[ ] 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)
```

- **To get predictions on unseen data, model.predict method is used**
- **It returns the raw output from the model (i.e. probabilities of an observation belonging to each one of the 4 classes**
- the sum of probabilities of an observation belonging to each of the 4 classes will be 1 i.e, **np.sum(pred, axis=1)**

To know the class an observation belongs to, using these 4 probability values

- Find the index having the largest probability and that will be the predicted class.
- **pred_class = np.argmax(pred, axis = 1)**

To check the accuracy of the model using sklearn's accuracy_score

**from sklearn.metrics import accuracy_score**
**acc_score = accuracy_score(y_test, pred_class)**