Implementing Neural Networks

by **Sanjeev Gupta**

1. Building a neural network from scratch with tensorflow operations
2. Keras Sequential API
3. Keras Functional API
4. Keras Model subclassing
5. Callbacks

Import libraries

```
import tensorflow as tf
from tensorflow import keras
import math
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from keras.layers import Dense, Flatten
from keras import Input
from tensorflow.keras.utils import plot_model

from tensorflow.keras.datasets import mnist, fashion_mnist
```

1. Basic Sequential Model

We want to build a sequential model. This means that the layers of our neural network are stacked sequentially. The approach is as follows:

1. First implement a class to build a dense layer. We call it "NaiveDense"
2. Implement a class ("NaiveSequential") to stack the layers sequentially and build or suquential model.

```
# Implemnting our dense layer class
class NaiveDense:
  def __init__(self, input_size, output_size, activation):  #input and output sizes for the layer
    self.activation = activation

    w_shape = (input_size, output_size)       # matrix of weights
    w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
    self.w = tf.Variable(w_initial_value)     # we can only update values of tf.Variables

    b_shape = (output_size,)                  # vector of biases
    b_initial_value = tf.zeros(b_shape)
    self.b = tf.Variable(b_initial_value)

  def __call__(self, inputs):                 # executed when the class object is used as a function
    return self.activation(tf.matmul(inputs, self.w)+self.b)      # self.b is broadcasted

  @property                                   # enables us to use the method as an attribute
  def weights(self):
    return (self.w, self.b)
```

Great! We implemented a dense layer. Now we will stack them together sequentially in our NaiveSequential class

```
class NaiveSequential:
  def __init__(self,layers):    # layers: list of layer objects
    self.layers = layers

  def __call__(self, inputs):
    x = inputs
    for layer in self.layers:     #ouptut of the prev layer is the input to the next layer
      x = layer(x)
    return x

  @property
  def weights(self):
    weights = []
    for layer in self.layers:     # save weights of each layer to a list
      weights += layer.weights    # Q: What does layer.weights return?
    return weights                # A: layer.weights calls the function layer.weights() since it decorated with @property. It returns (w
```

Great! Now we know how this sequential stacking of dense layer is implemented !

Next, let's instantiate our NaiveSequential class and make our first NN model

```
# define the model
model = NaiveSequential([
        NaiveDense(input_size=28*28,output_size=512,activation=tf.nn.relu),
        NaiveDense(input_size=512,output_size=10,activation=tf.nn.softmax)
])
#  Q: What input argument does NaiveSequential take? A:  list of layer objects
#  Q: What is the input and output dimension of the overall model? A: input dim = 784 , output dim = 10
#  Q: Can the output_size of 1 layer be different from the input_size of the next layer? A: No, they have to be the same.
```

```
28*28
```

```
784
```

Now we have our sequential model! But it is untrained and currently not useful.

We must train the model to make it learn useful representaions but first we need data !

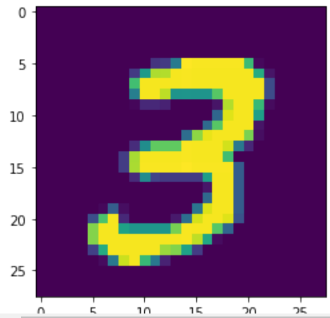
So let's solve a calssification problem by using above sequential model for [MNIST data set](#).

We must

1. Load the data,
2. Reshape the data according to the input shape of the model and
3. Normalize the data

```
# Load data
(train_images, train_labels),(test_images, test_labels) = mnist.load_data()
print(f"train_images.shape = {train_images.shape}")
print(f"train_labels.shape = {train_labels.shape}")
# Q: How many samples does the training set have? A: 60000

idx = 7
plt.imshow(train_images[idx])
print(f"label for image id {idx}: ",train_labels[idx])
```

```
train_images.shape = (60000, 28, 28)
train_labels.shape = (60000,)
label for image id 7:  3
```



```
# reshape and normalize data
train_images = train_images.reshape((len(train_images),
                                     28*28)).astype("float32")/255
test_images = test_images.reshape((len(test_images),
                                     28*28)).astype("float32")/255
```

```
print(train_labels)
```

```
[5 0 4 ... 5 6 8]
```

Next we divide the data into batches. For this operation let's implement a class for Batch Generation.

```
class BatchGenerator:

  def __init__(self, images, labels, batch_size=128):
    assert len(images) == len(labels)
    self.index = 0
```

```
    self.images = images
    self.labels = labels
    self.batch_size = batch_size
    self.num_batches = math.ceil(len(images)/batch_size)
    print(f"batch size = {batch_size}")
    print(f"num of batches = {self.num_batches}")

  def next(self):
    images = self.images[self.index:self.index + self.batch_size]
    labels = self.labels[self.index:self.index + self.batch_size]
    self.index += self.batch_size
    return images, labels
```

Finally it is time to train the model!

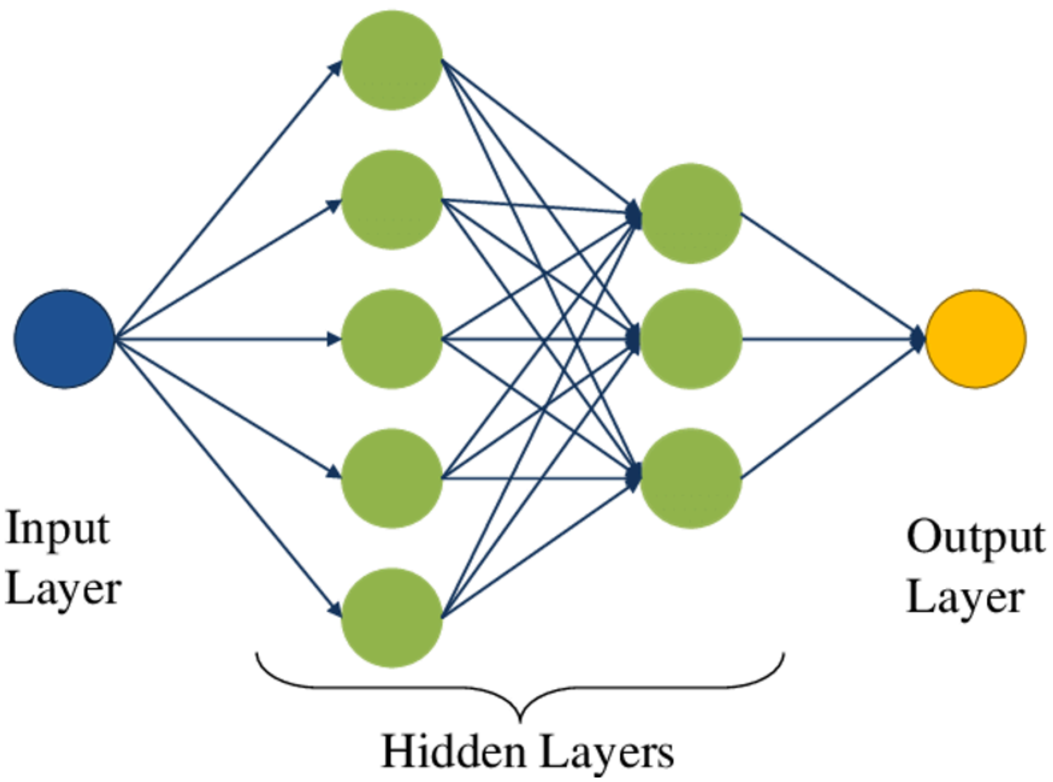
Keras has made life easy for us. Once we have defined the model all we have to do to train it is

1. model.compile()
2. model.fit()

But we should know what goes on behind the scenes. So let's see the steps involved in training a model\

Training steps:

1. Compute the predictions using current weights (Forward Pass).
2. Compute the loss value for these predictions.
3. Compute the gradient with respect to model weights.
4. update the weights.




```
# one_training_step function gives the idea of how loss is computed and layer \
# parameters (weights and biases) are updated
def one_training_step(model, images_batch, labels_batch):
  with tf.GradientTape() as tape:                    # GradientTape() is the computational graph
    predictions = model(images_batch)               # forward pass.
    per_sample_losses = keras.losses.sparse_categorical_crossentropy(  # define loss
        labels_batch, predictions
    )
    average_loss = tf.reduce_mean(per_sample_losses)
  gradients = tape.gradient(average_loss, model.weights)      # Compute gradients
  update_weights(gradients, model.weights)                    # Update the weights
  return average_loss

learning_rate = 1e-3
def update_weights(gradients, weights):
  for g,w in zip(gradients, weights):
    w.assign_sub(g*learning_rate)             # w -= g*lr
```

```
# Full training loop
def fit(model, images, labels, epochs, batch_size=128):
  for epoch in range(epochs):                       # repeat for epochs
    print(f"Epoch {epoch}")
    batch_generator = BatchGenerator(images, labels)
    for batch_counter in range(batch_generator.num_batches):      # go through all mini-batches in the data
      images_batch, labels_batch = batch_generator.next()
      loss = one_training_step(model, images_batch, labels_batch)
      if batch_counter%100 == 0:
        print(f"loss at batch {batch_counter}:{loss:.2f}")

# Q: Identify the 4 training steps in the above code
# A: They are present in the one_training_step function. See comments.
```

Now let's train the model on MNIST data set.

```
fit(model, train_images, train_labels, epochs=10, batch_size=128)

# Q: we didn't do a compile step.... or did we? (read again after seeing keras API)
# A: In model.compile(), we pass information about the loss function, optimizer, and evaluation metric. \
#     In our naive implementation, instead of defining a separate compile() function, we have defined the loss \
#     inside one_training_step; implemented the optimizer 'mini-batch gradient descent' in update_weights(); \
#     and we are doing the evaluation (accuracy) separately in a later cell.
```

```
Epoch 0
batch size = 128
num of batches = 469
loss at batch 0:5.57
loss at batch 100:2.24
loss at batch 200:2.21
loss at batch 300:2.11
loss at batch 400:2.22
Epoch 1
batch size = 128
num of batches = 469
loss at batch 0:1.91
loss at batch 100:1.88
loss at batch 200:1.84
loss at batch 300:1.74
loss at batch 400:1.84
Epoch 2
batch size = 128
num of batches = 469
loss at batch 0:1.59
loss at batch 100:1.58
loss at batch 200:1.52
loss at batch 300:1.45
loss at batch 400:1.52
Epoch 3
batch size = 128
num of batches = 469
loss at batch 0:1.33
loss at batch 100:1.34
loss at batch 200:1.25
loss at batch 300:1.23
loss at batch 400:1.29
Epoch 4
batch size = 128
num of batches = 469
loss at batch 0:1.13
loss at batch 100:1.15
loss at batch 200:1.05
loss at batch 300:1.06
loss at batch 400:1.12
Epoch 5
batch size = 128
num of batches = 469
loss at batch 0:0.98
loss at batch 100:1.01
loss at batch 200:0.91
loss at batch 300:0.94
loss at batch 400:1.00
Epoch 6
batch size = 128
num of batches = 469
loss at batch 0:0.87
loss at batch 100:0.91
loss at batch 200:0.80
loss at batch 300:0.85
loss at batch 400:0.91
Epoch 7
batch size = 128
```

After 10 epochs the loss has come down to 0.73!

Our model has definitely learned something. Lets evaluate how accurately it can predict labels for images **it has not seen before**. These are the **images in the test set**.

Lets use the "accuracy" metric. Here we simply find the fraction of times the model succeeded in predicting the correct label.

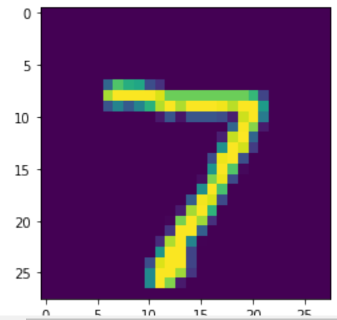
When using Keras, we would mention this metric in model.compile(). (More on Keras later)

```
#  Evaluation step
predictions = model(test_images).numpy()
predicted_labels = np.argmax(predictions,axis=1)
matches = predicted_labels == test_labels
print(f"accuracy:{matches.mean():.2f}")
print(predictions.shape)
print(predicted_labels)
print(matches)
```

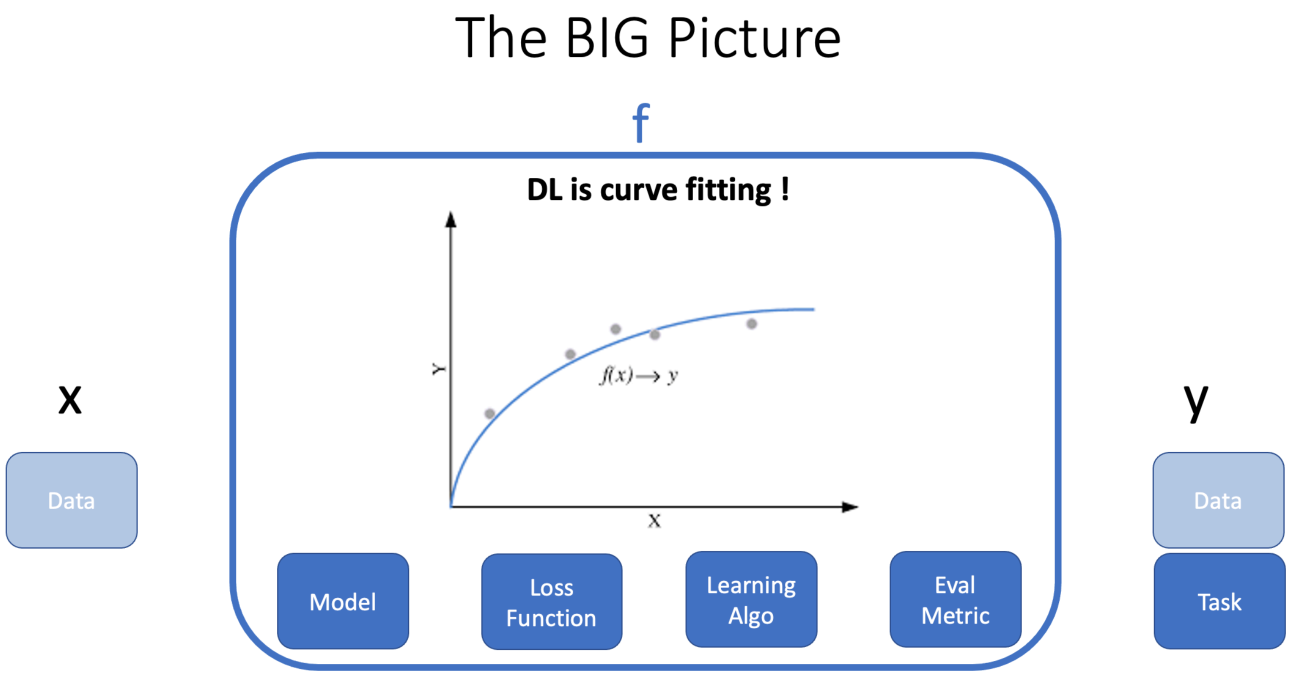
```
accuracy:0.82
(10000, 10)
[7 2 1 ... 4 8 6]
[ True  True  True ...  True False  True]
```

```
idx = 0
plt.imshow(test_images[idx].reshape(28,28))
print(predicted_labels[idx], test_labels[idx])
```

```
7 7
```



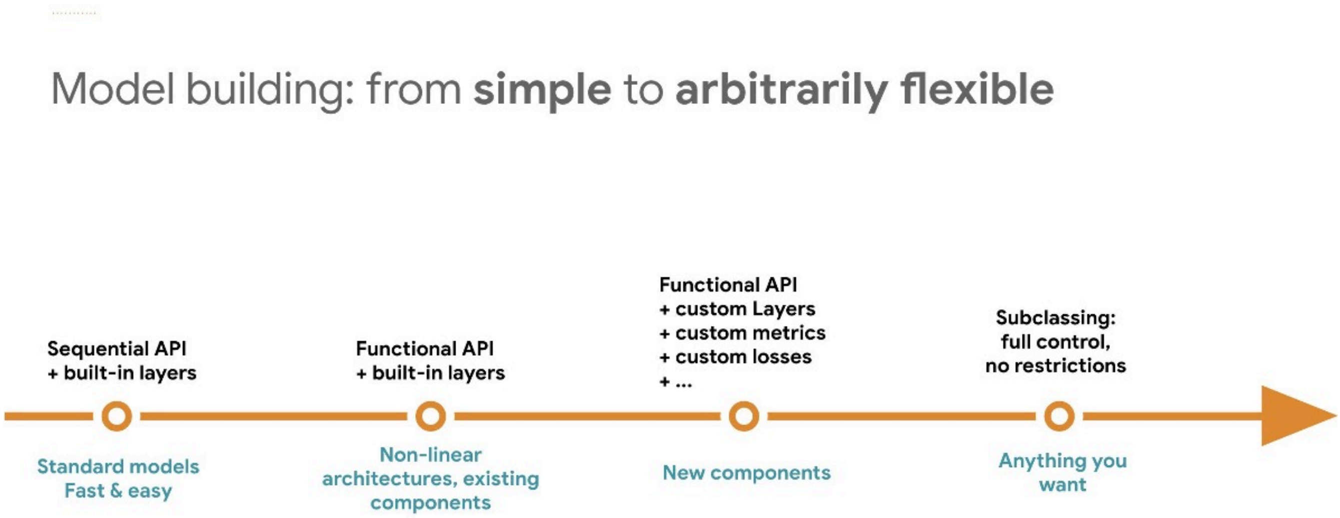
Summary



Different APIs

1. Sequential Model
2. Functional API

3. Model subclassing



2.Sequential API

Whatever we have implemented so far can be done alternatively using Sequential class in keras. In the following approach layer are passed as a list

```
# from keras.layers import Dense, Flatten
# from keras import Input

seq_model = keras.Sequential([
                         Dense(64, activation="relu"),
                         Dense(10, activation="softmax")
])
# Q: Do you notice a difference in arguments of the Dense layers, compared to our implementation?
# A: Yes. No need to mention input size for keras.Sequential. It is inferred by keras itself.
```

Alternatively, instead of passing layers as list, we can build a sequential model by adding layers incrementally to the model.

```
seq_model_inc = keras.Sequential()
seq_model_inc.add(Dense(64, activation="relu"))
seq_model_inc.add(Dense(10, activation="softmax"))
```

Notice that we have not yet provided information of input dimensions.

These layers are referred to as symbolic layers.

Unless until you build the model layer weights are not created.

```
try:
  seq_model_inc.weights
except:
  print("seq_model_inc.weights did not work because model was not built and weights were not initialized.")
```

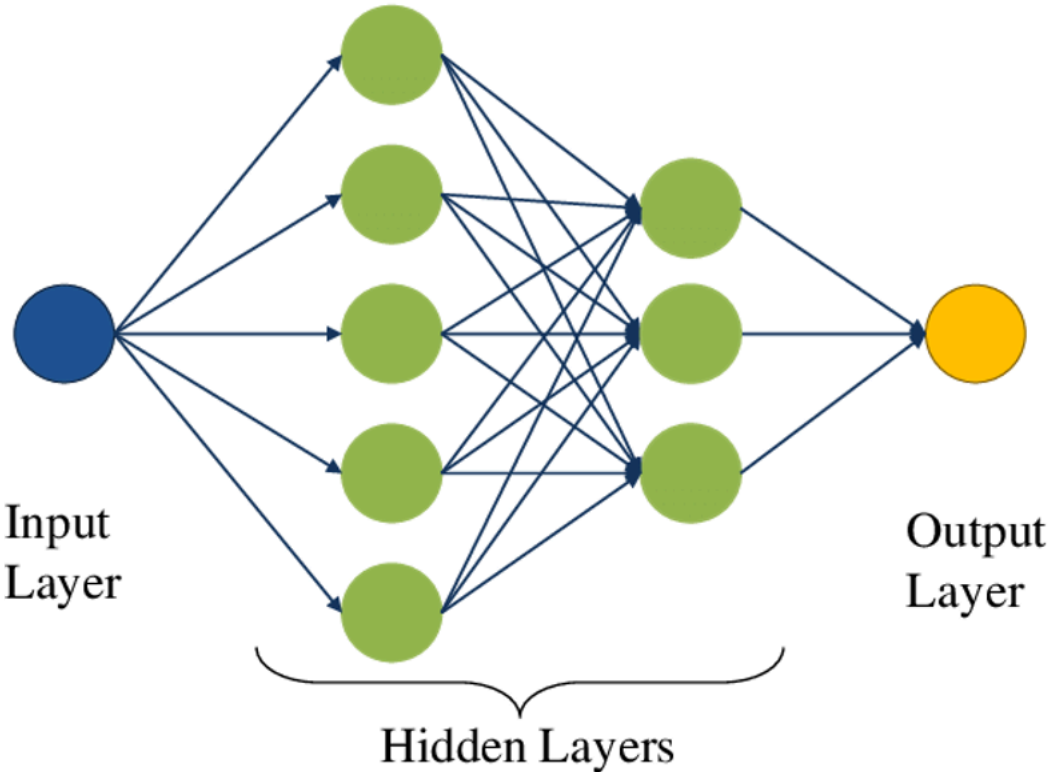
```
seq_model_inc.weights did not work because model was not built and weights were not initialized.
```

To create a weights you need to call on some data or call its build method with input shape

```
seq_model_inc.build(input_shape=(None, 3))  #None means it can take any batch size; 3 is the number of features in your input
seq_model.build(input_shape=(None, 3))

print(type(seq_model_inc.weights))
seq_model_inc.weights
```

```
<class 'list'>
```

```
[<tf.Variable 'dense_15/kernel:0' shape=(3, 64) dtype=float32, numpy=
 array([[ 1.44846976e-01, -2.18670100e-01,  2.29430258e-01,
         -1.12742782e-01,  1.36850089e-01,  1.01428717e-01,
         -1.97391719e-01, -2.33028114e-01, -1.45720556e-01,
          2.20789492e-01, -8.29304010e-02, -2.48935312e-01,
          6.46039546e-02, -1.67079866e-02,  1.26166314e-01,
          1.20301038e-01,  2.29161739e-01, -3.97650599e-02,
         -2.61659771e-01,  1.77647531e-01,  1.50566190e-01,
          1.51140183e-01,  1.02343738e-01,  9.82773006e-02,
         -2.41253480e-01,  2.93537140e-01,  2.20617294e-01,
         -1.70859471e-01, -2.71017343e-01,  7.28799105e-02,
         -2.69627869e-01,  1.87116683e-01,  2.83798873e-01,
         -5.05889654e-02,  2.56579220e-01, -8.36384445e-02,
          2.35531926e-01,  2.91702092e-01,  1.91416949e-01,
         -1.45027846e-01, -1.42255217e-01, -2.58929133e-02,
         -2.61746168e-01, -7.54176527e-02, -1.28255069e-01,
         -2.36875176e-01, -2.05620587e-01,  1.02031231e-02,
          8.78518820e-02, -7.40671903e-02,  1.67623580e-01,
         -5.38714528e-02, -1.76747844e-01, -2.11665481e-01,
          8.26619864e-02, -9.00956839e-02,  2.38787055e-01,
         -2.89154440e-01, -2.00049490e-01, -2.38261253e-01,
         -8.48178267e-02, -3.04153264e-02,  2.77565122e-02,
          9.41936672e-02],
        [ 3.28582525e-03, -1.53826490e-01, -2.07747310e-01,
         -2.85859972e-01, -1.96267068e-01,  1.04508013e-01,
         -2.46805385e-01,  2.88513303e-02, -2.62435108e-01,
          1.49608076e-01,  1.64378703e-01, -2.65429258e-01,
         -1.61449030e-01,  2.38224804e-01, -1.97246030e-01,
          1.37537569e-01,  7.95944035e-02, -1.44790545e-01,
         -2.10254103e-01, -4.22446728e-02, -1.66679859e-01,
          1.38105214e-01,  2.66237855e-02, -2.02402458e-01,
          2.35694528e-01,  1.61121696e-01,  1.64229870e-01,
          2.08356559e-01, -2.50580221e-01,  1.11113548e-01,
          1.86853856e-01, -8.17737132e-02,  1.69566095e-01,
         -1.15439638e-01,  2.46289253e-01, -1.89692974e-01,
         -2.54041076e-01,  7.88029432e-02, -1.33986980e-01,
         -1.26948759e-01,  6.54541254e-02,  8.13392699e-02,
          2.53158867e-01, -2.55190998e-01,  2.17753053e-01,
          1.03417307e-01, -2.14490712e-01,  2.62766302e-01,
          2.93517709e-01, -9.50333476e-03, -2.23486334e-01,
          2.00392246e-01,  1.04625732e-01, -1.44753873e-01,
          4.41914499e-02,  3.47997844e-02,  1.69060737e-01,
          2.31785297e-01, -3.53853405e-02, -1.66206911e-01,
         -1.33526579e-01, -2.68126845e-01, -2.46547043e-01,
          1.63272232e-01],
        [ 1.03941858e-02, -1.73210725e-01,  2.63023078e-01,
          1.94167882e-01,  1.41594261e-01, -1.69026345e-01,
          1.38517261e-01, -5.26740998e-02,  2.46134996e-02,
          4.81922626e-02,  2.74699211e-01, -2.86230117e-01,
         -1.43859819e-01,  2.02106476e-01, -8.32575560e-03,
          2.46754348e-01,  8.95386636e-02,  1.03722185e-01,
         -5.42291850e-02, -9.46594179e-02,  2.53973305e-01,
          2.88602710e-01, -1.62680984e-01, -9.43397880e-02,
          8.43972266e-02,  2.73092330e-01, -2.59637833e-04,
          1.68687165e-01,  9.12820399e-02,  1.61256403e-01,
          7.51611590e-02, -1.39940783e-01, -2.11833864e-01,
         -1.83983535e-01,  2.48822451e-01, -2.84342110e-01,
```




```
seq_model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_13 (Dense)            (None, 64)                256

 dense_14 (Dense)            (None, 10)                650

=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

Lets calculate the no. of params in each layer. For layer 1:

```
(64*3) + 64
```

```
256
```

Q: Can you verify the number of parameters by a quick caluclation?

A: 650 = 10*64 + 10

weight matrix has 64*10 weights and 10 biases for the 10 neurons

```
256 + 650
```

```
906
```

Specifying input shape in advance

```
model_seq = keras.Sequential(name="sequential_model")
model_seq.add(keras.Input(shape=(3, )))   #specifying the input here
model_seq.add(keras.layers.Dense(64, activation=tf.nn.relu, name="first_layer"))
model_seq.add(keras.layers.Dense(10, activation=tf.nn.softmax,
                                 name="second_layer"))
```

```
model_seq.summary()
```

```
Model: "sequential_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
```

```
=================================================================
 first_layer (Dense)         (None, 64)                256

 second_layer (Dense)        (None, 10)                650

=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

Double-click (or enter) to edit

3. Functional API

Next we will use Keras functional API to create the same model. Keras functional API can create more flexible models than Sequential API. It can handle models with non-linear topology, shared layers, and even multiple inputs or outputs.

Key Idea- Expresses each layer as a function of the previous layer.

(input: 3-dimensional vectors)
↧
[Dense (64 units, relu activation)]
↧
(output: 10 units, softmax activation)

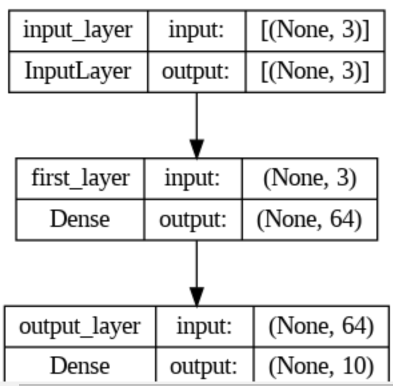
```
inputs = Input(shape=(3,), name="input_layer")
features = Dense(64, activation="relu",name="first_layer")(inputs) #f(inputs)
outputs = Dense(10, activation="softmax",
                             name="output_layer")(features) #f(features)
fun_model = keras.Model(inputs,outputs)
```

```
fun_model.summary()
```

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_layer (InputLayer)    [(None, 3)]               0

 first_layer (Dense)         (None, 64)                256

 output_layer (Dense)        (None, 10)                650

=================================================================
Total params: 906
Trainable params: 906
Non-trainable params: 0
_________________________________________________________________
```

We get effectively the same summary. Becuase we have implemented the same model using the functional API.

```
plot_model(fun_model, show_shapes=True)
```



Lets see a deeper network.

```
# from keras.layers import Dense
# import keras
#node = Layer(nodes, extra_params)(prev_node)
```

```
inputs = keras.Input(shape=(64,))
dense1 = Dense(32, activation='relu')(inputs)
dense2 = Dense(32, activation='relu')(dense1) #defining dense2 node whose parent is dense1
outputs = Dense(4, activation='softmax')(dense2) #defining output node where parent is dense2
model = keras.Model(inputs=inputs, outputs=outputs, name="linear_topology")
```

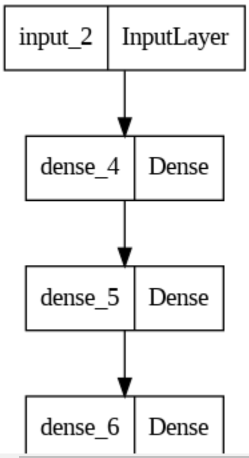
```
model.summary()
```

```
Model: "linear_topology"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 64)]              0

 dense_4 (Dense)             (None, 32)                2080

 dense_5 (Dense)             (None, 32)                1056

 dense_6 (Dense)             (None, 4)                 132

=================================================================
Total params: 3,268
Trainable params: 3,268
Non-trainable params: 0
_________________________________________________________________
```

```
32*64 +32
```

```
2080
```

```
plot_model(model)
```



Now let's see an example where the Sequential API would not be sufficeint.

Multi-Input and Multi-output: Consider an example of building a system to rank customer tickets by priority and route them to the appropriate departments.

Outputs: model need to give two outputs

1. First task of the model is to classify the tickets into priority and non priority (Binary classification)
2. Second task is to route the ticket to appropriate department (Multi-class classification based on the number of departments)

These two task are need to be done simultaneously

Inputs:

1. Title of the ticket (text input)
2. The text body of the ticket (text input)
3. Any tags added by the user

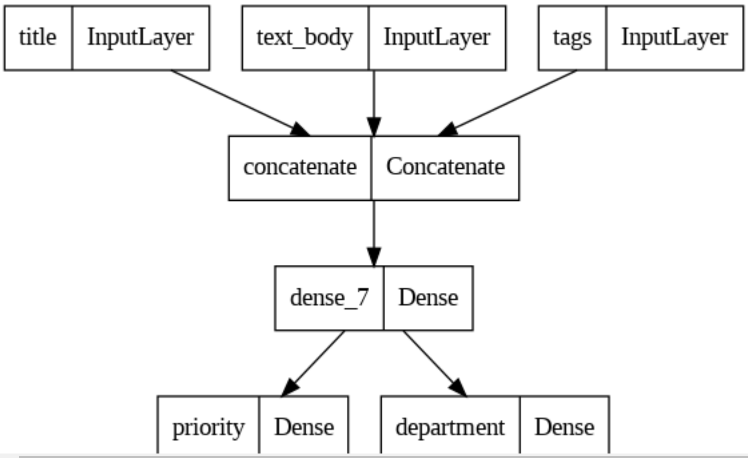
Q. Is it possible to build the model sequentially?

A: No, we cannot build a multi-input , multi-output model through the sequential API, because, by definition itself, the required model is not sequential.

```
vocabulary_size = 10000
num_tags = 100
num_departments = 4
```

```
# Inputs
title = keras.Input(shape=(vocabulary_size,), name="title")
text_body = keras.Input(shape = (vocabulary_size, ), name="text_body")
tags = keras.Input(shape=(num_tags, ), name="tags")

features = keras.layers.Concatenate()([title, text_body, tags])
features = keras.layers.Dense(64, activation = "relu")(features)

# Outputs
priority = keras.layers.Dense(1, activation="sigmoid",        # sigmoid for binary classification
                              name = "priority")(features)
department = keras.layers.Dense(num_departments, activation="softmax", # softmax for multi-class classification
                               name="department")(features)

model = keras.Model(inputs=[title,text_body,tags],
                    outputs=[priority, department])
```

```
plot_model(model)
```




```
model.summary()
```

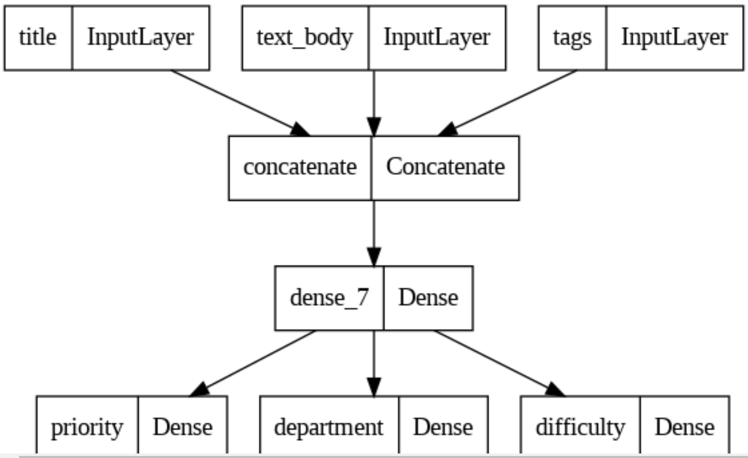
```
Model: "model_1"
__________________________________________________________________________________________________
 Layer (type)                Output Shape         Param #     Connected to
==================================================================================================
 title (InputLayer)          [(None, 10000)]      0           []

 text_body (InputLayer)      [(None, 10000)]      0           []

 tags (InputLayer)           [(None, 100)]        0           []

 concatenate (Concatenate)   (None, 20100)        0           ['title[0][0]',
                                                               'text_body[0][0]',
                                                               'tags[0][0]']

 dense_7 (Dense)             (None, 64)           1286464     ['concatenate[0][0]']

 priority (Dense)            (None, 1)            65          ['dense_7[0][0]']

 department (Dense)          (None, 4)            260         ['dense_7[0][0]']

==================================================================================================
Total params: 1,286,789
Trainable params: 1,286,789
Non-trainable params: 0
__________________________________________________________________________________________________
```

Reusing the model by training intermediate layer output

```
features = model.layers[4].output
difficulty = keras.layers.Dense(3, activation="softmax",
                                name="difficulty")(features)
```

```
new_model = keras.Model(inputs=[title, text_body,tags],
                        outputs=[priority, department,difficulty])
```

```
keras.utils.plot_model(new_model)
```



4. Subclassing the Model class

We saw how the functional API enabled us to make more complex models compared to the sequential API. We moved up the ladder of progressive disclosure of complexity.

Now let's move a step further.

But first, let us see how to define the same old model by subclassing the Model class.

```
class CustomerTicketModel(keras.Model):

# Define the layers in in the __init__ method
  def __init__(self, num_departments):
    super().__init__()
    self.concat_layer = keras.layers.Concatenate()
    self.mixing_layer = keras.layers.Dense(64, activation="relu")
    self.priority_scorer = keras.layers.Dense(1, activation="sigmoid")
    self.department_classifier = keras.layers.Dense(num_departments,
                                                    activation="softmax")


# Define the relationship between layers in in the call method
# See Section 7.2.3 in Francois chollet for more details
  def call(self,inputs):
    # input should be dictionary type
    title = inputs["title"]
    text_body = inputs["text_body"]
    tags = inputs["tags"]

    features = self.concat_layer([title, text_body, tags])
    features = self.mixing_layer(features)

    priority = self.priority_scorer(features)
    department = self.department_classifier(features)

    return priority, department


sub_class_model = CustomerTicketModel(num_departments=4)


try:
  sub_class_model.summary()
except:
  print("summary() did not work because we have not built the model")
```

```
summary() did not work because we have not built the model
```

```
# here model is built by calling the data since build() method is not
# defined in model subclass
# generate random data
title_data = np.random.randint(0, 2, size=(1000,vocabulary_size))
text_body_data = np.random.randint(0, 2, size=(1000,vocabulary_size))
tags_data = np.random.randint(0, 2, size=(1000,num_tags))

priority, department = sub_class_model({"title":title_data,
                                        "text_body":text_body_data,
                                        "tags":tags_data})
```

```
sub_class_model.summary()
```

```
Model: "customer_ticket_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 concatenate_1 (Concatenate)  multiple                 0         
                                                                 
 dense_8 (Dense)             multiple                  1286464   
                                                                 
 dense_9 (Dense)             multiple                  65        
                                                                 
 dense_10 (Dense)            multiple                  260       
                                                                 
=================================================================
Total params: 1,286,789
Trainable params: 1,286,789
Non-trainable params: 0
_________________________________________________________________
```

```
from keras.initializers import RandomNormal
```

```
# Creating a custom LAYER, by subclassing keras.layers.Layer
class Custom_Dense(keras.layers.Layer):
  def __init__(self, units, activation=None):
    super() .__init__()
    self.units = units
    self.activation = activation

  # Subclassing gives us the flexibility here to initialize weights on our own
  def build(self, input_shape):
    input_dim = input_shape[-1]
    std_dev = np.sqrt(2/(input_dim + self.units))
    self.W = self.add_weight(shape=(input_dim, self.units),
                            initializer=RandomNormal(stddev=std_dev))
    self.b = self.add_weight(shape=(self.units,),
                             initializer="zeros")

  def call(self, inputs):
    y = tf.matmul(inputs, self.W) + self.b
    if self.activation is not None:
      y = self.activation(y)
    return y
```

We can even define custom metrics and custom loss functions using the subclassing API. Refer to Section 7.3.1 of Chollet for details.

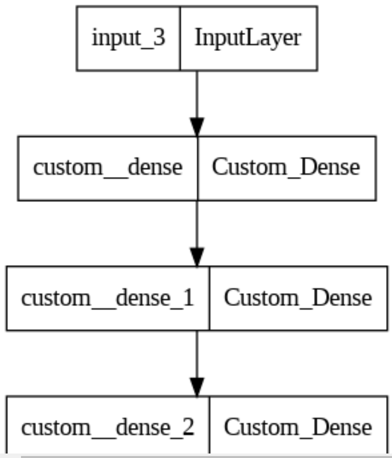
Building the model using custom dense layer and functional API

```
inputs = Input(shape=(28*28,))
features = Custom_Dense(512, activation=tf.nn.relu)(inputs)
features = Custom_Dense(128, activation=tf.nn.relu)(features)
outputs = Custom_Dense(10, activation=tf.nn.softmax)(features)
model = keras.Model(inputs,outputs)
```

```
model.summary()
```

```
Model: "model_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 input_3 (InputLayer)        [(None, 784)]             0         
                                                                 
 custom__dense (Custom_Dense  (None, 512)              401920    
 )                                                               
                                                                 
 custom__dense_1 (Custom_Den  (None, 128)              65664     
 se)                                                             
                                                                 
 custom__dense_2 (Custom_Den  (None, 10)               1290      
 se)                                                             
                                                                 
=================================================================
Total params: 468,874
Trainable params: 468,874
Non-trainable params: 0
_________________________________________________________________
```

```
plot_model(model)
```




```
model.compile(optimizer =keras.optimizers.RMSprop(),
              loss = keras.losses.SparseCategoricalCrossentropy(),
              metrics = ["accuracy"])
```

```
train_x = train_images[10000:]
train_y = train_labels[10000:]
val_x = train_images[:10000]
val_y = train_labels[:10000]
```

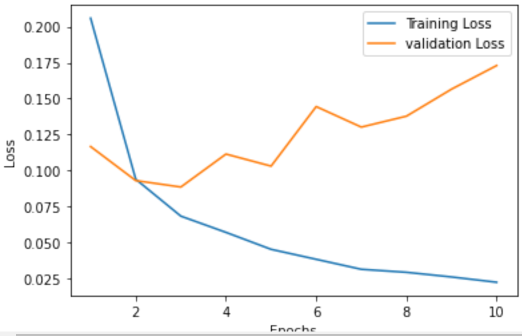
```
history = model.fit(x=train_x, y=train_y, epochs=10,
                    validation_data=(val_x, val_y))
```

```
Epoch 1/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.2059 - accuracy: 0.9377 - val_loss: 0.1166 - val_accuracy: 0.966
Epoch 2/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0939 - accuracy: 0.9733 - val_loss: 0.0929 - val_accuracy: 0.973
Epoch 3/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0682 - accuracy: 0.9809 - val_loss: 0.0884 - val_accuracy: 0.977
Epoch 4/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0569 - accuracy: 0.9854 - val_loss: 0.1114 - val_accuracy: 0.975
Epoch 5/10
1563/1563 [==============================] - 5s 3ms/step - loss: 0.0451 - accuracy: 0.9882 - val_loss: 0.1030 - val_accuracy: 0.978
Epoch 6/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0381 - accuracy: 0.9902 - val_loss: 0.1444 - val_accuracy: 0.975
Epoch 7/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0312 - accuracy: 0.9922 - val_loss: 0.1301 - val_accuracy: 0.976
Epoch 8/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0291 - accuracy: 0.9929 - val_loss: 0.1377 - val_accuracy: 0.979
Epoch 9/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0258 - accuracy: 0.9936 - val_loss: 0.1566 - val_accuracy: 0.979
Epoch 10/10
1563/1563 [==============================] - 5s 4ms/step - loss: 0.0222 - accuracy: 0.9949 - val_loss: 0.1730 - val_accuracy: 0.977
```

```
data = pd.DataFrame(history.history)
data.head()
```

	loss	accuracy	val_loss	val_accuracy
0	0.205926	0.93768	0.116598	0.9660
1	0.093874	0.97334	0.092927	0.9733
2	0.068189	0.98088	0.088432	0.9773
3	0.056886	0.98538	0.111375	0.9751
4	0.045090	0.98816	0.102987	0.9783

```
plt.plot(range(1,11),data['loss'], label="Training Loss")
plt.plot(range(1,11),data['val_loss'],label="validation Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Q. What is the difference between evaluate() and predict()

A: evelaute() returns the loss score and evaluation score. predict() runs a forward pass for the given input data.

```
model.evaluate(test_images,test_labels)
#HW: LOAD the best saved model by using tf.load_model and evaluate the loaded model
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1546 - accuracy: 0.9799
[0.15462972223758698, 0.9799000024795532]
```

```
class_predicted = np.argmax(model.predict(test_images),axis=1)
accuracy = np.sum(class_predicted == test_labels)/len(test_labels)
print(accuracy)
```

```
313/313 [==============================] - 1s 2ms/step
0.9799
```

```
from sklearn.metrics import confusion_matrix, classification_report
```

```
print(classification_report(test_labels, class_predicted))
```

```
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       980
           1       0.99      0.99      0.99      1135
           2       0.99      0.97      0.98      1032
           3       0.97      0.99      0.98      1010
           4       0.98      0.98      0.98       982
           5       0.99      0.97      0.98       892
           6       0.98      0.98      0.98       958
           7       0.98      0.97      0.98      1028
           8       0.97      0.97      0.97       974
           9       0.97      0.98      0.97      1009

    accuracy                           0.98     10000
   macro avg       0.98      0.98      0.98     10000
weighted avg       0.98      0.98      0.98     10000
```

```
print(confusion_matrix(test_labels, class_predicted))
```

```
[[ 973    1    0    1    1    0    2    1    1    0]
 [   1 1125    1    1    0    0    2    1    4    0]
 [   7    3  996    2    3    0    3    8   10    0]
 [   2    1    1  996    0    0    0    1    3    6]
 [   1    0    1    1  964    0    4    2    1    8]
 [   3    0    0   12    2  868    4    0    2    1]
 [   5    3    0    0    5    2  941    0    1    1]
 [   2    3    3    3    2    1    0 1001    3   10]
 [   6    1    2    5    1    6    0    1  949    3]
 [   2    4    0    4    8    2    0    2    1  986]]
```

```
#save model
# https://www.tensorflow.org/api_docs/python/tf/keras/models/save_model
```

Using Callbacks

A callback is a powerful tool to customize the behavior of a Keras model during training, evaluation, or inference.

```
# build model using functional API
inputs = Input(shape=(28*28,))
features = Dense(512,activation="relu")(inputs)
```

```
features = keras.layers.Dropout(0.5)(features)
outputs = Dense(10,activation="softmax")(features)

mnist_model = keras.Model(inputs, outputs)


from keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard


callbacks_list = [EarlyStopping(monitor="val_loss", patience=2),
                  ModelCheckpoint("mnist_model_checkpoint",save_best_only=True),
                  TensorBoard(log_dir="/tensorboard_files")]


mnist_model.compile(optimizer =keras.optimizers.Adam(),
              loss = keras.losses.SparseCategoricalCrossentropy(),
              metrics = ["accuracy"])


mnist_model.fit(x=train_x, y=train_y, epochs=10,
                    validation_data=(val_x, val_y),
                   callbacks=callbacks_list,)
```

```
Epoch 1/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.2884 - accuracy: 0.9147 - val_loss: 0.1436 - val_accuracy: 0.957
Epoch 2/10
1563/1563 [==============================] - 5s 4ms/step - loss: 0.1462 - accuracy: 0.9543 - val_loss: 0.1022 - val_accuracy: 0.970
Epoch 3/10
1563/1563 [==============================] - 5s 3ms/step - loss: 0.1126 - accuracy: 0.9644 - val_loss: 0.0913 - val_accuracy: 0.972
Epoch 4/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0962 - accuracy: 0.9697 - val_loss: 0.0865 - val_accuracy: 0.973
Epoch 5/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0839 - accuracy: 0.9737 - val_loss: 0.0801 - val_accuracy: 0.975
Epoch 6/10
1563/1563 [==============================] - 6s 4ms/step - loss: 0.0746 - accuracy: 0.9761 - val_loss: 0.0768 - val_accuracy: 0.977
Epoch 7/10
1563/1563 [==============================] - 8s 5ms/step - loss: 0.0678 - accuracy: 0.9783 - val_loss: 0.0770 - val_accuracy: 0.976
Epoch 8/10
1563/1563 [==============================] - 7s 4ms/step - loss: 0.0618 - accuracy: 0.9798 - val_loss: 0.0823 - val_accuracy: 0.977
<keras.callbacks.History at 0x7fe4e3101760>
```

```
%load_ext tensorboard
%tensorboard --logdir /tensorboard_files
```

TensorBoard SCALARS GRAPHS TIME SERIES INACTIVE

Reference

- Chollet, F. (2021). Deep learning with python. Manning Publications.

epoch_accuracy

```
mnist_model.fit(x=train_x, y=train_y, epochs=10,
                    # validation_data=(val_x, val_y),
                   callbacks=callbacks_list,)
```

```
Epoch 1/10
1554/1563 [============================>.] - ETA: 0s - loss: 0.0573 - accuracy: 0.9810WARNING:tensorflow:Early stopping conditioned
WARNING:tensorflow:Can save best model only with val_loss available, skipping.
1563/1563 [==============================] - 4s 3ms/step - loss: 0.0572 - accuracy: 0.9810
Epoch 2/10
1557/1563 [============================>.] - ETA: 0s - loss: 0.0545 - accuracy: 0.9821WARNING:tensorflow:Early stopping conditioned
WARNING:tensorflow:Can save best model only with val_loss available, skipping.
1563/1563 [==============================] - 4s 3ms/step - loss: 0.0544 - accuracy: 0.9821
Epoch 3/10
1559/1563 [============================>.] - ETA: 0s - loss: 0.0499 - accuracy: 0.9837WARNING:tensorflow:Early stopping conditioned
WARNING:tensorflow:Can save best model only with val_loss available, skipping.
1563/1563 [==============================] - 4s 3ms/step - loss: 0.0500 - accuracy: 0.9837
Epoch 4/10
1555/1563 [============================>.] - ETA: 0s - loss: 0.0500 - accuracy: 0.9832WARNING:tensorflow:Early stopping conditioned
WARNING:tensorflow:Can save best model only with val_loss available, skipping.
1563/1563 [==============================] - 4s 3ms/step - loss: 0.0499 - accuracy: 0.9832
Epoch 5/10
1562/1563 [============================>.] - ETA: 0s - loss: 0.0462 - accuracy: 0.9849WARNING:tensorflow:Early stopping conditioned
WARNING:tensorflow:Can save best model only with val_loss available, skipping.
1563/1563 [==============================] - 4s 3ms/step - loss: 0.0462 - accuracy: 0.9849
Epoch 6/10
 191/1563 [==>...........................] - ETA: 3s - loss: 0.0450 - accuracy: 0.9853
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call last)
<ipython-input-65-0c17a4dd562f> in <module>
----> 1 mnist_model.fit(x=train_x, y=train_y, epochs=10,
```