Course: Artificial Intelligence and Machine Learning Code: 20CS51I

WEEK - 10: KERAS – DENSE LAYER

- **Keras Dense Layer**
  - **Overview**
  - **Parameters**
  - **Operation**
- **Keras optimizers**
- **Keras Losses**

Session No. 6

Keras Dense Layer:

Overview

Dense layer is the regular deeply connected neural network layer. It is most common and frequently used layer. Dense layer does the below operation on the input and return the output.

```
output = activation(dot(input, kernel) + bias)
```

where,

- **input** represents the input data
- **kernel** represent the weight data
- **dot** represents numpy dot product of all input and its corresponding weights
- **bias** represent a biased value used in machine learning to optimize the model
- **activation** represent the activation function.

Let us consider sample input and weights as below and try to find the result −

- input as 2 x 2 matrix **[[1, 2], [3, 4]]**
- kernel as 2 x 2 matrix **[[0.5, 0.75], [0.25, 0.5]]**
- bias value as **0**
- activation as **linear**. As we learned earlier, linear activation does nothing.

```
>>> import numpy as np

>>> input = [ [1, 2], [3, 4] ]
>>> kernel = [ [0.5, 0.75], [0.25, 0.5] ]
>>> result = np.dot(input, kernel)
```

result is the output and it will be passed into the next layer.

The output shape of the Dense layer will be affected by the number of neuron / units specified in the Dense layer. For example, if the input shape is **(8,)** and number of unit is 16, then the output shape is **(16,)**. All layer will have batch size as the first dimension and so, input shape will be represented by **(None, 8)** and the output shape as **(None, 16)**. Currently, batch size is None as it is not set. Batch size is usually set during training phase.

```
>>> from keras.models import Sequential
>>> from keras.layers import Activation, Dense

>>> model = Sequential()
>>> layer_1 = Dense(16, input_shape = (8,))
>>> model.add(layer_1)
>>>  layer_1.input_shape
(None, 8)
```

where,

- **layer_1.input_shape** returns the input shape of the layer.
- **layer_1.output_shape** returns the output shape of the layer.

The argument supported by **Dense layer** is as follows −

- ***units*** represent the number of units and it affects the output layer.
- ***activation*** represents the activation function.
- ***use_bias*** represents whether the layer uses a bias vector.
- ***kernel_initializer*** represents the initializer to be used for kernel.
- ***bias_initializer*** represents the initializer to be used for the bias vector.
- ***kernel_regularizer*** represents the regularizer function to be applied to the kernel weights matrix.
- ***bias_regularizer*** represents the regularizer function to be applied to the bias vector.

- ***activity_regularizer*** represents the regularizer function tp be applied to the output of the layer.
- ***kernel_constraint*** represent constraint function to be applied to the kernel weights matrix.
- ***bias_constraint*** represent constraint function to be applied to the bias vector.

As you have seen, there is no argument available to specify the ***input_shape*** of the input data. ***input_shape*** is a special argument, which the layer will accept only if it is designed as first layer in the model.

❖ Parameters/Arguments:

The argument supported by the **Dense layer** is as follows −

- ***units*** represent the number of units and it affects the output layer.
- ***activation*** represents the activation function.
- ***use_bias*** represents whether the layer uses a bias vector.
- ***kernel_initializer*** represents the initializer to be used for kernel.
- ***bias_initializer*** represents the initializer to be used for the bias vector.
- ***kernel_regularizer*** represents the regularizer function to be applied to the kernel weights matrix.
- ***bias_regularizer*** represents the regularizer function to be applied to the bias vector.
- ***activity_regularizer*** represents the regularizer function tp be applied to the output of the layer.
- ***kernel_constraint*** represent constraint function to be applied to the kernel weights matrix.
- ***bias_constraint*** represent constraint function to be applied to the bias vector.

As you have seen, there is no argument available to specify the ***input_shape*** of the input data. ***input_shape*** is a special argument, which the layer will accept only if it is designed as first layer in the model.

❖ Keras Operations

Basic Operations with Dense Layer:

The parameters we have three main attributes: activation function, weight matrix, and bias vector. Using these attributes a dense layer operation can be represented as:

Output = activation(dot(input, kernel) + bias)

Where if the input matrix for the dense layer has a rank of more than 2, then dot product between the kernel and input along the last axis of the input and zeroth axis of the kernel using the tf.tensordot calculated by the dense layer if the use_bias is False.

How to Implement the Dense Layer?

In this section of the article, we will see how to implement a dense layer in a neural network with a single dense layer and a neural network with multiple dense layers.

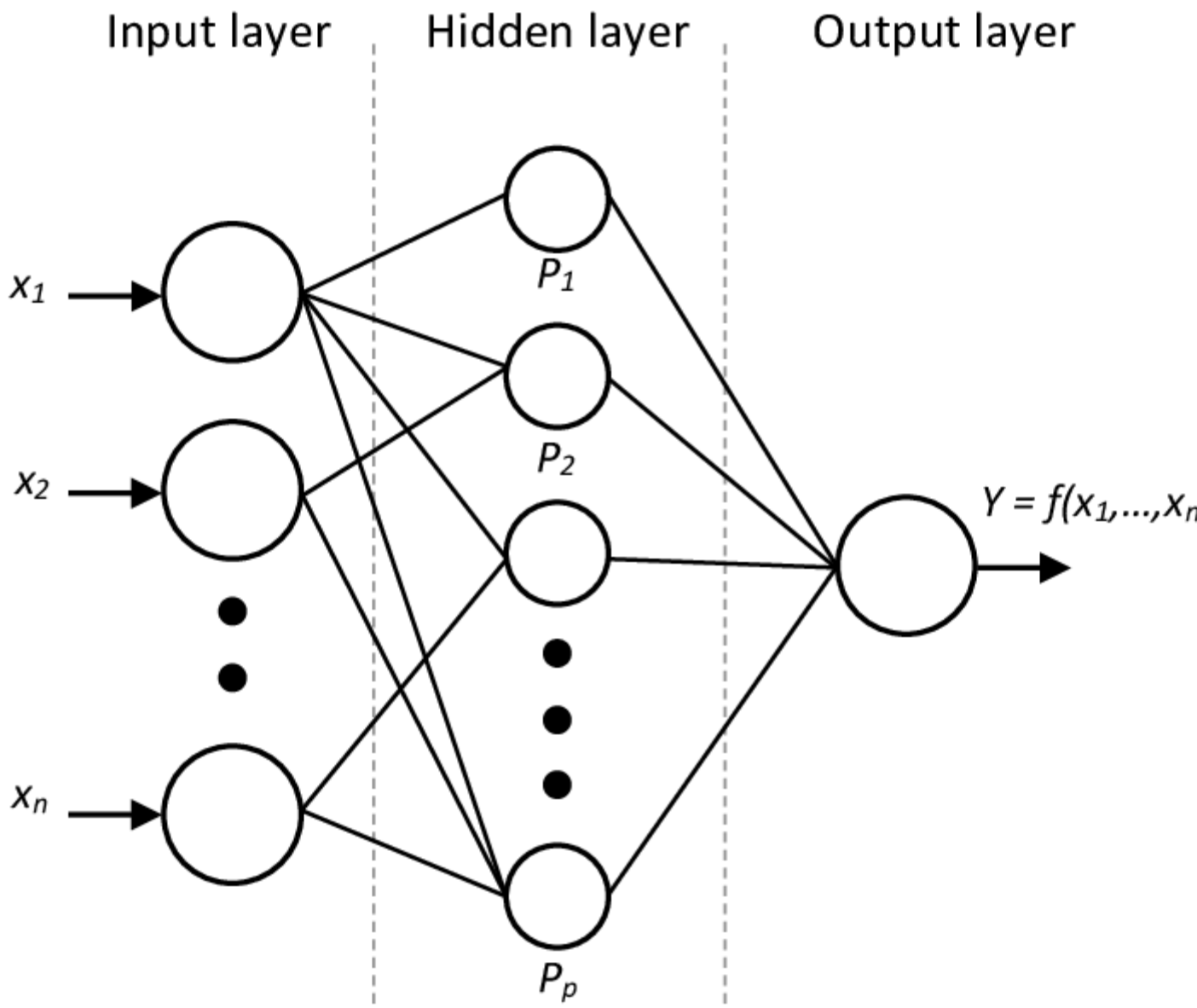
A sequential model with a single dense layer.

```
import tensorflow

model = tensorflow.keras.models.Sequential()

model.add(tensorflow.keras.Input(shape=(16,)))

model.add(tensorflow.keras.layers.Dense(32, activation='relu'))

print(model.output_shape)

print(model.compute_output_signature)
```

Output:

```
(None, 32)
[<keras.layers.core.Dense object at 0x7f5da8243c50>]
<bound method Layer.compute_output_signature of <keras.engine.sequential.Sequential object at
```

Here in the output, we can see that the output of the model is a size of (None,32) and we are using a single Keras layer and the signature of the output from the model is a sequential object.



The above image represents the neural network with one hidden layer. If we consider the hidden layer as the dense layer the image can represent the neural network with a single dense layer.

A sequential model with two dense layers:

```
model1 = tensorflow.keras.models.Sequential()

model1.add(tensorflow.keras.Input(shape=(16,)))

model1.add(tensorflow.keras.layers.Dense(32, activation='relu'))

model1.add(tensorflow.keras.layers.Dense(32))
```

print(model1.output_shape)

print(model1.layers)

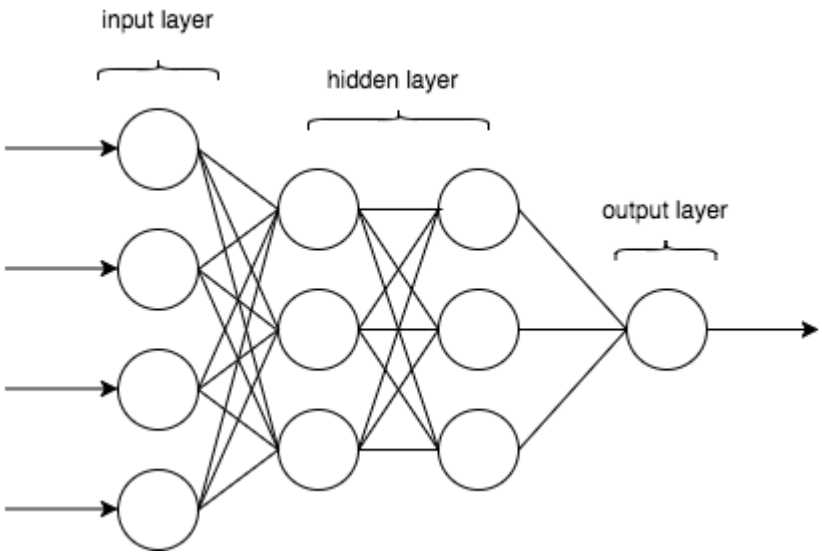
print(model1.compute_output_signature)

Output:

```
(None, 32)
[<keras.layers.core.Dense object at 0x7f5da8256410>, <keras.layers.core.Dense object at 0x7f5
<bound method Layer.compute_output_signature of <keras.engine.sequential.Sequential object at
```

Here in the output, we can see that the output shape of the model is (None,32) and that there are two dense layers and again the signature of the output from the model is a sequential object.

After defining the input layer once we don't need to define the input layer for every dense layer.



❖ Keras optimizers:

During the training of the model, we tune the parameters (also known as hyperparameter tuning) and weights to minimize the loss and try to make our prediction accuracy as correct as possible.

Now to change these parameters the optimizer's role came in, which ties the model parameters with the loss function by updating the model in response to the loss function output.

Simply optimizers shape the model into its most accurate form by playing with model weights. Theloss function just tells the optimizer when it's moving in the right or wrong direction.

Optimizers are Classes or methods used to change the attributes of your machine/deep learning modelsuch as weights and learning rate in order to reduce the losses. Optimizers help to get results faster.

Tensorflow Keras Optimizers Classes:

Gradient descent optimizers, the year in which the papers were published, and the components theyact upon

TensorFlow mainly supports 9 optimizer classes, consisting of algorithms like Adadelta, FTRL,NAdam, Adadelta, and many more.

- Adadelta: Optimizer that implements the Adadelta algorithm.
- Adagrad: Optimizer that implements the Adagrad algorithm.
- Adam: Optimizer that implements the Adam algorithm.
- Adamax: Optimizer that implements the Adamax algorithm.
- Ftrl: Optimizer that implements the FTRL algorithm.
- Nadam: Optimizer that implements the NAdam algorithm.
- Optimizer class: Base class for Keras optimizers.
- RMSprop: Optimizer that implements the RMSprop algorithm.
- SGD: Gradient descent (with momentum) optimizer.

❖ Keras Losses:

A loss is a callable with arguments loss_fn(y_true, y_pred, sample_weight=None):

- **y_true**: Ground truth values, of shape (batch_size, d0, ... dN). For sparse loss functions, suchas sparse categorical crossentropy, the shape should be (batch_size, d0, ... dN-1)
- **y_pred**: The predicted values, of shape (batch_size, d0, .. dN).
- **sample_weight**: Optional sample_weight acts as reduction weighting coefficient for the per-sample losses. If a scalar is provided, then the loss is simply scaled by the given value.

- If the sample weight is a tensor of size [batch size], then the total loss for each sample of thebatch is rescaled by the corresponding element in the sample_weight vector.
- If the shape of sample_weight is (batch_size, d0, ... dN-1) (or can be broadcasted to this shape), then eachloss element of y_pred is scaled by the corresponding value of sample weight. (Note ondN-1:all loss functions reduce by 1 dimension, usu ally axis=-1.)