**DSE 3159 DEEP LEARNING LABORATORY MANUAL**

**August 2023**

**WEEK 1- TENSORFLOW & KERAS TUTOTIAL**

## What is TensorFlow?

TensorFlow is an open-source deep learning framework developed by the Google Brain team. It allows users to create, train, and deploy machine learning models, especially deep neural networks. TensorFlow provides a flexible architecture to work with numerical data using multi-dimensional arrays called **tensors**. It supports both CPU and GPU computations, making it suitable for running on a variety of hardware.

## What are Tensors?

In TensorFlow, tensors are the fundamental data structures used for representing data. They are similar to multi-dimensional arrays and can hold data of any number of dimensions. Tensors are the building blocks of neural networks, as they store the input data, weights, biases, and intermediate outputs during the computation.

Examples of Tensors:

1. Scalar (0-D tensor): A single value is a 0-D tensor.

   Eg: `scalar_tensor = 5  #rank-0 tensor`

2. Vector (1-D tensor): A 1-D tensor contains a sequence of values.

   Eg: `vector_tensor = [1, 2, 3, 4, 5]  #rank-1 tensor`

3. Matrix (2-D tensor): A 2-D tensor is an array of arrays.

   Eg: `matrix_tensor = [[1, 2, 3], [4, 5, 6], [7, 8, 9]] #rank-2 tensor`

4. Higher-dimensional tensor (e.g., 3-D tensor):

   Eg: `tensor_3d = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]] #rank-3 tensor`

Note: For a detailed explanation, visit the TensorFlow | Tensor documentation: https://www.tensorflow.org/guide/tensor

## Graph Computation:

TensorFlow follows a symbolic approach for computation using graphs. A graph is a computational graph that represents the flow of data through a series of operations (nodes) to produce output (tensors). The nodes in the graph represent operations, and the edges represent tensors flowing between these operations.

Example of Graph Computation:

```
import tensorflow as tf

# Define input variables (placeholders)
x = tf.placeholder(tf.float32)
```

```
y = tf.placeholder(tf.float32)

# Define operations
x_squared = tf.square(x)    # Square operation
x_squared_times_y = tf.multiply(x_squared, y)    # Multiply operation
result = tf.add(x_squared_times_y, tf.add(y, 2))    # Add operation

# Create a session to run the computation graph
with tf.Session() as sess:
    # Provide input values and run the graph
    output = sess.run(result, feed_dict={x: 3.0, y: 4.0})
    print("Output:", output)
```

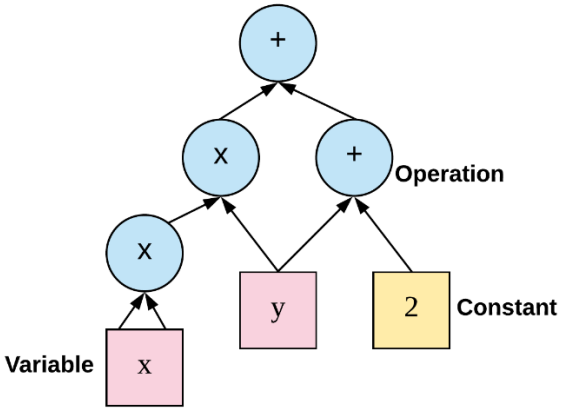


Fig1: Computation graph in tensorflow for **f(x, y) = $x^2y$ + y + 2**
[Image Source: https://iq.opengenus.org]

### What is Keras?

Keras is an open-source high-level neural networks API written in Python and capable of running on top of TensorFlow, among other backends. It was designed with a focus on enabling fast experimentation and easy-to-use syntax for building deep learning models. Keras provides a user-friendly interface for constructing complex neural networks, making it an ideal choice for beginners in deep learning.

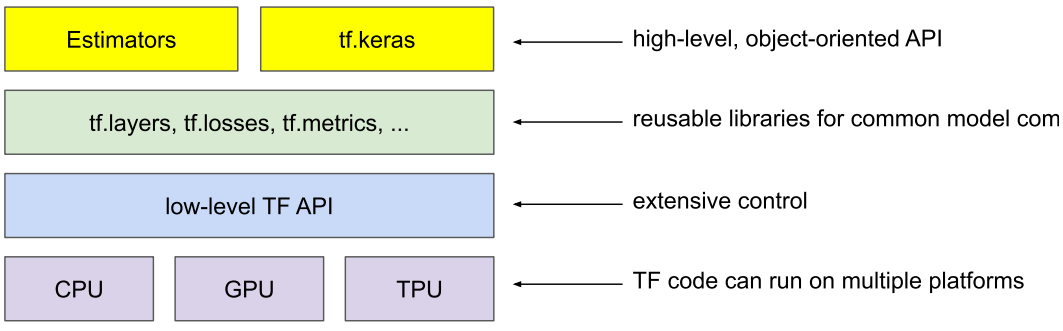


Fig 2. Tensorflow and Keras as API
Image Source: https://developers.google.com/

Note: For a detailed explanation, visit the TensorFlow | Keras documentation: https://www.tensorflow.org/guide/keras

In Keras, there are two primary ways to create deep learning models: the **Sequential API** and the **Functional API**. Each approach serves a different purpose and offers distinct advantages.

**Sequential API:**

The Sequential API is the simplest and most straightforward way to build deep learning models in Keras. It allows you to create a linear stack of layers, where each layer has exactly one input tensor and one output tensor. This means that the data flows sequentially through each layer in the order they are added to the model. The Sequential API is well-suited for simple feedforward neural networks and other models that have a clear linear flow of data.

Example of Sequential API:

```
from keras.models import Sequential
from keras.layers import Dense, Input

# Create a sequential model
model = Sequential()

# Add layers to the model
model.add(Input(shape=(input_dim,)))
model.add(Dense(64, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Print the model summary
model.summary()
```

**Functional API:**

The Functional API in Keras allows you to create more complex models with multiple input and output tensors, as well as models with shared layers. It provides greater flexibility and is particularly useful when building models with branching or merging architectures.

Example of Functional API:

```
from keras.models import Model
from keras.layers import Input, Dense


# Define input tensor
input_tensor = Input(shape=(input_dim,))
```

```
# Create layers and connect them
hidden_layer1 = Dense(64, activation='relu')(input_tensor)
hidden_layer2 = Dense(32, activation='relu')(hidden_layer1)
output_tensor = Dense(10, activation='softmax')(hidden_layer2)

# Create the model
model = Model(inputs=input_tensor, outputs=output_tensor)

# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])

# Print the model summary
model.summary()
```

**Deep Learning Model Life-Cycle**

The deep learning model life cycle typically involves the following steps: Define the model, Compile the model, Fit the model, Evaluate the model, and Make predictions.

**1. Define the Model:**

In this step, you specify the architecture of your deep learning model. You define the layers, their configurations, activation functions, and any other required settings. The architecture depends on the problem you are trying to solve, and it may include fully connected layers, convolutional layers, recurrent layers, etc.

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

# Define the model
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(input_dim,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

**2. Compile the Model:**

After defining the model, you need to compile it. During this step, you specify the loss function, optimizer, and evaluation metrics. The loss function is used to measure how well the model is performing on the training data. The optimizer determines how the model's weights are updated during training, and the evaluation metrics provide additional performance metrics during training.

```
# Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
```

**3. Fit the Model:**

In this step, you train the model on your training data. You provide the input features (X) and their corresponding target labels (y) to the model. The model then adjusts its internal parameters (weights) through an optimization process (usually gradient descent) to minimize the defined loss function.

```
# Fit the model
model.fit(X_train, y_train, epochs=10, batch_size=32,
validation_data=(X_val, y_val))
```

**4. Evaluate the Model:**

After the model is trained, you need to evaluate its performance on a separate set of data that it has never seen before (e.g., a validation set or a test set). This step gives you an indication of how well the model generalizes to unseen data.

```
# Evaluate the model
loss, accuracy = model.evaluate(X_test, y_test)
print(f"Test loss: {loss}, Test accuracy: {accuracy}")
```

**5. Make Predictions:**

Once the model is trained and evaluated, you can use it to make predictions on new, unseen data. You pass the new data to the model, and it will provide predictions based on what it has learned during training.

```
# Make predictions
predictions = model.predict(X_new_data)
```

Example: Building a Simple Neural Network with Keras

```
#1) Import the necessray libraries

import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Input

#2) For the tutorial, lets experiment with random data

# Generate random input data (features)
X = np.random.rand(num_samples, num_features)
```

```
# Generate random output labels (classes)
y = np.random.randint(0, num_classes, size=num_samples)

# Split the data into training and testing sets
split_ratio = 0.8
split_index = int(num_samples * split_ratio)

X_train, X_test = X[:split_index], X[split_index:]
y_train, y_test = y[:split_index], y[split_index:]

#3) Define the model

# Build the neural network model using Sequential API
model = Sequential([
    Input(shape=(num_features,)),
    Dense(6, activation='relu'),  # Hidden layer with 6 neurons
    Dense(num_classes, activation='softmax')  # Output layer with
num_classes neurons and softmax activation for classification
])

# Display a summary of the model architecture
model.summary()

#4) Compile the model

# Compile the model
model.compile(optimizer='adam',
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

#5) Fit/train the model

# Train the model using the training data
epochs = 50
batch_size = 32
model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_split=0.1)

6) Evaluate/test the model

# Evaluate the model on the testing data
loss, accuracy = model.evaluate(X_test, y_test,
batch_size=batch_size)
print("Test Loss:", loss)
print("Test Accuracy:", accuracy)
```

**Exercise 1:**

Accurate measurement of body fat is inconvenient/costly and it is desirable to have easy methods of predicting Body Fat. Using the Body Fat dataset, write a Neural Network to predict body fat:

a. Number of Hidden layers = 3 and number of units are 128,64,32
b. Use RELU activation function, let learning rate be 0.1

Spit the data into (80,20) split and tabulate the performance in terms of RMSE for100 epochs and comment on performance.

The attributes of the dataset are :

1. Density determined from underwater weighing
2. Percent body fat from Siri's (1956) equation
3. Age (years)
4. Weight (lbs)
5. Height (inches)
6. Neck circumference (cm)
7. Chest circumference (cm)
8. Abdomen 2 circumference (cm)
9. Hip circumference (cm)
10. Thigh circumference (cm)
11. Knee circumference (cm)
12. Ankle circumference (cm)
13. Biceps (extended) circumference (cm)
14. Forearm circumference (cm)
15. Wrist circumference (cm)

**WEEK 2 – EXPERIMENTING WITH DEEP NEURAL NETWORKS**

Consider the following dataset 'Churn_Modelling.csv'
https://www.kaggle.com/datasets/aakash50897/churn-modellingcsv

The data set has 14 features which are as follows:-

1. RowNumber:- Represents the number of rows
2. CustomerId:- Represents customerId
3. Surname:- Represents surname of the customer
4. CreditScore:- Represents credit score of the customer
5. Geography:- Represents the city to which customers belongs to
6. Gender:- Represents Gender of the customer
7. Age:- Represents age of the customer
8. Tenure:- Represents tenure of the customer with a bank
9. Balance:- Represents balance hold by the customer
10. NumOfProducts:- Represents the number of bank services used by the customer
11. HasCrCard:- Represents if a customer has a credit card or not
12. IsActiveMember:- Represents if a customer is an active member or not
13. EstimatedSalary:- Represents estimated salary of the customer
14. Exited:- Represents if a customer is going to exit the bank or not.

1. Perform the required pre-processing and write comment lines to explain the pre-processing steps.

2. Perform experiments using (70,15,15) split and tabulate the performance in terms of Accuracy, Precision & Recall for the following experimental setup :
   1. Number of Hidden Layers and Number of Units per Layer

| Number of Hidden Layers | Number of Units |
|---|---|
| 1 | 128, 0 ,0 |
| 2 | 128, 64, 0 |
| 3 | 128, 64, 32 |

   2. Epochs (10,20,30)
   3. Activation function (Sigmoid )
   4. Without Regularization, with Regularization (L1/L2)
   5. Learning rate ( 0.1, 0.01,0.001)
   6. Visualize the training and validation loss against the epochs and comment on optimal hyperparameters.

**WEEK 3- CONVOLUTIONAL NUERAL NETWORKS VS FULLY CONNECTED NEURAL NETWORKS**

Consider the Fashion MNIST dataset [Fashion MNIST dataset, an alternative to MNIST (keras.io)] and do the following:

Q1) Understanding the Dataset and Pre-processing: Implement the following:

a. Compute and display the number of classes.
b. Compute and display the dimensions of each image.
c. Display one image from each class.
d. Perform normalization.

Q2) Performing experiments on Fully Connected Neural Networks (FCNN):

a. Design a FCNN which is most suitable for the given dataset: Experimentally choose the best network (the intuitions and learnings from the experiments you have performed in Week-1 and Week-2 will help you choose the hyperparameters for the network).
b. Train and test the network (choose the best epoch size so that there is no overfitting).
c. Plot the performance curves.

Q3) Performing experiments on a Convolutional Neural Networks (CNNs):

a. Design **CNN-1** which contains:
   - One Convolution layer which uses 32 kernels each of size 5x5, stride = 1 and, padding =0.
   - One Pooling layer which uses MAXPOOLING with stride =2.
   - One hidden layer having number of neurons = 100
b. Design **CNN-2** which contains:

- Two back-to-back Convolution layers which uses 32 kernels each of size 3x3, stride = 1, and padding =0.
- One Pooling layer which uses MAXPOOLING with stride =2.
- One hidden layer having number of neurons = 100

Note: use ReLU activation function after each convolution layer.

c. Train and test the networks (choose the best epoch size so that there is no overfitting).
d. Plot the performance curves for CNN-1 and CNN-2.
e. Compare the performances of CNN-1 and CNN-2.

Q4) Compare the performances of FCNN and CNN.

Q5) Compare the number of parameters in the FCNN and the CNN.

Q6) Discuss the computational efficiency of both networks. Which one took longer to train and why?