Assignment No.2

Aim: Implementing Feed-forward neural networks with Keras and TensorFlow:

a. Import the necessary packages
b. Load the training and testing data (MNIST/CIFAR10)
c. Define the network architecture using Keras
d. Train the model using SGD
e. Evaluate the network
f. Plot the training loss and accuracy

Objective: Student will learn:

1. Understand how to use Tensorflow Eager and Keras Layers to build neural network architecture.
2. Understand how a model is trained and evaluated.
3. Identify digits from images.
4. Our main goal is to train a neural network (using Keras) to obtain > 90% accuracy on MNIST dataset.
5. Research at least 1 technique that can be used to improve model generalization.

Methodology to be used -

1. Deep Learning
2. Feed Forward Neural Network

Prerequisites –

- Basic understanding of Python programming
- Familiarity with the concepts of Neural Networks.

Theory:

Deep learning has revolutionized the world of machine learning as more and more ML practitioners have adopted deep learning networks to solve real-world problems. Compared to the more traditional ML models, deep learning networks have been shown superior performance for many applications.

The first step toward using deep learning networks is to understand the working of a simple feedforward neural network. We get started with how we can build our first neural network model using Keras running on top of the Tensorflow library.

TensorFlow is an open-source platform for machine learning. Keras is the high-level application programming interface (API) of TensorFlow. Using Keras, we can rapidly develop a prototype system and test it out. This is the first in a three-part series on using TensorFlow for supervised classification tasks.

A Conceptual Diagram of the Neural Network:

We'll build a supervised classification model that learns to identify digits from images. We’ll use the well known MNIST dataset to train and test our model. The MNIST dataset consists of 28-by-28 images of handwritten digits along with their corresponding labels.
We’ll construct a neural network with one hidden layer to classify each digit image into its corresponding label. The figure below shows a conceptual diagram of the neural network we are about to build. The output layer consists of 10 units, where each unit corresponds to a different digit. Each unit computes a value that can be interpreted as the confidence value of its respective digit. The final classification is the digit with the maximum confidence value.

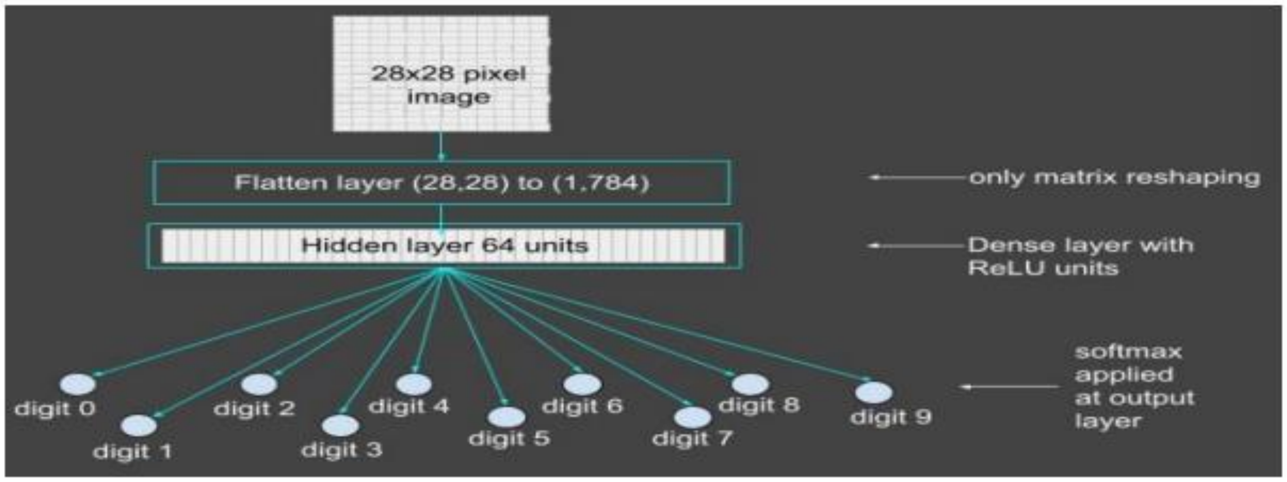


Figure 1: Conceptual diagram of the neural network. Each output unit corresponds to a digit and has its confidence value. The final classification is the digit with the maximum confidence value.

TensorFlow and Keras Libraries:

If Keras and TensorFlow are not installed on your system, you can easily do so using pip or conda depending upon your Python environment.
pip install tensorflow
In the context of ML, a tensor is a multidimensional array, which in its simplest form is a scalar. Vectors and matrices are special cases of tensors. In TensorFlow, a tensor is a data structure. It is a multidimensional array composed of elements of the same type. Tensors are used to encapsulate all inputs and outputs to a deep learning network. The training dataset and each test example has to be cast as a tensor. All operations within the layers of the network are also performed on tensors

Layers in TensorFlow:

You can build a fully connected feedforward neural network by stacking layers sequentially so that the output of one layer becomes the input to the next. In TensorFlow, layers are callable objects, which take tensors as input and generate outputs that are also tensors. Layers can contain weights and biases, which are both tuned during the training phase.

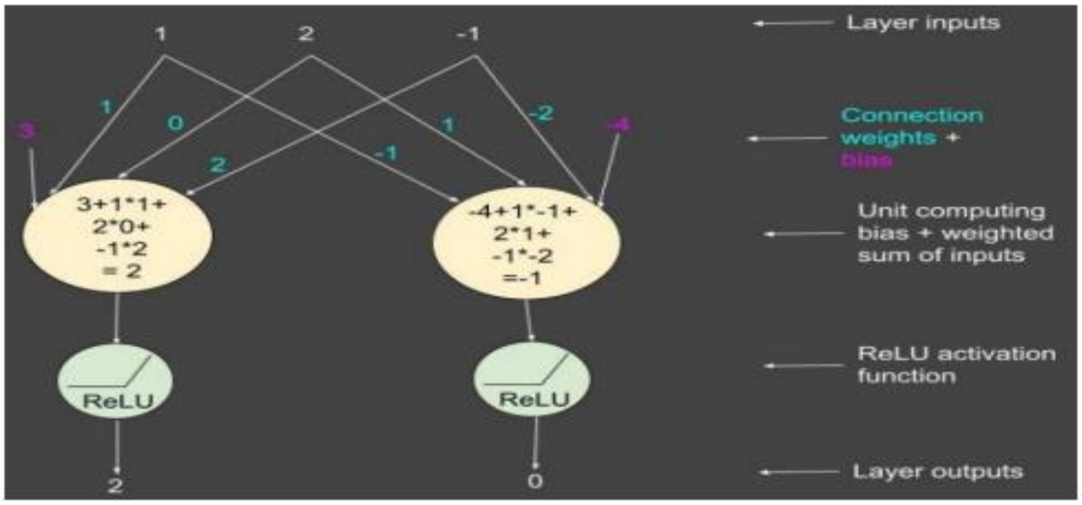
We'll create a simple neural network from two layers:

1. Flatten layer
2. Dense layer

The Flatten Layer:

This layer flattens an input tensor without changing its values. Given a tensor of rank n, the Flatten layer reshapes it to a tensor of rank 2. The number of elements on the first axis remains unchanged. The elements of the input tensor's remaining axes are stacked together to form a single axis. We need this layer to create a vectorized version of each image for further input to the next layer.

The Dense Layer :The dense layer is the fully connected, feedforward layer of a neural network. It computes the weighted sum of the inputs, adds a bias, and passes the output through an activation function. We are using the ReLU activation function for this example. This function does not change any value greater than 0. The rest of the values are all set to 0. The computations of this layer for the parameters shown in the code above are all illustrated in the figure below.



import the necessary libraries:

import numpy as np

import tensorflow as tf

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

Load the Dataset: The following code loads the training set and the test set of the MNIST data. It also prints the statistics of both sets. Because our model has 10 outputs, one for each digit, we need to convert the absolute image labels to categorical ones. The utils module in the Keras library provides the method to_categorical() for this conversion.

(trainX, trainY), (testX, testY) = mnist.load_data()

Creating a Model in TensorFlow:

We are now ready to create a model of a simple neural network with one hidden layer. The simplest method is to use Sequential() with a list of all the layers you want to stack together. The code below creates a model and prints its summary. Note the use of relu as the activation function in the first dense layer and a softmax in the output layer. The softmax function normalizes all outputs to sum to 1 and ensures that they are in the range [0, 1].

model = tf.keras.Sequential([

tf.keras.layers.Flatten(input_shape=(28, 28)),

tf.keras.layers.Dense(64, activation='relu'),

tf.keras.layers.Dense(10, activation='softmax')])

Compile the Model:

Next we compile the model. Here, we specify the optimizer and loss function of our model. The optimization algorithm determines how the connection weights are updated at each training step with respect to the given loss function. Because we have a multiclass classification problem, the loss function we'll use is categorical_crossentropy, coupled with the adam optimizer. You can experiment with other optimizers too. The value of the metrics argument sets the parameter to monitor and record during the training phase.

model.compile(loss="categorical_crossentropy", optimizer=SGD(0.01), metrics=["accuracy"])

Train the Model: The fit() method of the model object trains the neural network. If you want to use a validation set during training, all you have to do is define the percentage of validation examples to be taken from the training set. The splitting of the training set into a train and validation set is automatically taken care of by the fit() method. In the code below, the fit() method is called in 10 epochs.

history = model.fit(trainX, trainY, validation_data=(testX, testY), epochs=20, batch_size=128)

View the Training Process: The fit() method returns a history object with detailed information regarding model training. The history attribute is a dictionary object. To view the learning process, we can plot the accuracy and loss corresponding to different epochs for both the training and validation sets. The following code creates two graphs for these two metrics

The predict() method: If you want to see the output of the network for one or more train/test examples, use the predict() method. The following example code prints the values of the output layer when the first test image is used as an input. It is a 10-dimensional vector of confidence values corresponding to each digit. The final classification of the image is the argmax() of this vector.

predictions = model.predict(testX, batch_size=128)

print(classification_report(testY.argmax(axis=1), predictions.argmax(axis=1)))

The evaluate() method: The evaluate() method computes the overall accuracy of the model on the dataset passed as an argument. The code snippet below prints the classification accuracy on both the training set and the test set.

loss, accuracy = model.evaluate(testX, testY)

print(f"Test Loss: {loss:.4f}")

print(f"Test Accuracy: {accuracy:.4f}")

Conclusion:

With the above code we can see that, throughout the epochs, our model accuracy increases and loss decreases that is good since our model gains confidence with our prediction

This indicates the model is trained in a good way:

1. The two losses(loss and val_loss) are decreasing and the accuracy (accuracy and val_accuracy) increasing.

2. The val_accuracy is the measure of how good the model is predicting so, it is observed that the model is well trained after 10 epochs

Assignment No.3