Lab 1

Deep Learning – B.Sc Data Science and Analytics (Hons.)

Fifth Semester 2023

Lab Date/Time: 05/07/2023

Theory and Lab Faculty: Mr. Kunal Dey

Lab Description

In this lab you will learn about multi-layer perceptron for classification and its implementation using scikit-learn and tensorflow.

Objectives

Lab Tool(s)

https://www.kaggle.com/
or
https://colab.research.google.com/

Lab Deliverables

At the end of the semester you are required to submit the lab record, before the completion of final CA.

References:

https://github.com/Drakunal/Deep-Learning

Exploratory Data Analysis

1- Classification with a Fully Connected Neural Network using the Iris dataset

Scikit-learn: Scikit-learn is a comprehensive machine learning library that provides a wide range of algorithms for tasks such as classification, regression, clustering, and dimensionality reduction. It is well-suited for traditional machine learning tasks and is relatively easy to use, making it a good choice for beginners. However, when it comes to deep learning specifically, scikit-learn has limited capabilities compared to Keras and TensorFlow.
Keras: Keras is a high-level deep learning library that is built on top of TensorFlow. It provides a user-friendly and intuitive API, making it easy to create and train deep neural networks. Keras allows you to quickly prototype and experiment with different network architectures and supports both convolutional networks and recurrent networks. It also includes various pre-trained models and utilities for tasks such as image classification, object detection, and natural language processing. Keras is particularly suitable for beginners and rapid prototyping.
TensorFlow: TensorFlow is a powerful open-source library for numerical computation and deep learning. It provides a flexible and efficient framework for building and training all types of deep learning models. TensorFlow allows for low-level control and customization, which makes it suitable for advanced users and researchers. It also supports distributed computing, enabling the training of large-scale models. TensorFlow has a wide range of pre-built models and extensive community support. In addition to Python, TensorFlow also supports other languages such as C++, Java, and Swift.

1.1 Using Scikit-learn

Importing the Necessary Libraries

```
import numpy as np
import plotly.graph_objects as go
from sklearn.datasets import load_iris
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
```

These lines import the necessary libraries and modules for the experiment. numpy is imported for numerical operations, train_test_split is imported to split the dataset into training and testing sets, MinMaxScaler and OneHotEncoder are imported for data preprocessing, MLPClassifier is imported for building the fully connected neural network model, and accuracy_score, confusion_matrix, and pyplot from matplotlib are imported for evaluation and visualization purposes.

Dataset Loading

Here, we load the Iris dataset using load_iris() from scikit-learn. iris.data contains the feature vectors, and iris.target contains the corresponding target labels:

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target
```

Dataset Preprocessing

An instance of MinMaxScaler is created to scale the input features. The feature vectors are then scaled using fit_transform().

An instance of OneHotEncoder is created to perform one-hot encoding on the target labels. The labels are reshaped to a column vector using reshape(-1, 1) and then one-hot encoded using fit_transform().

```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y.reshape(-1, 1))
```

Model Architecture

Here, we create an instance of MLPClassifier, which represents a multi-layer perceptron (fully connected neural network) model. The model is specified with a single hidden layer of 32 neurons, using the ReLU activation function. random_state is set to ensure reproducibility of results:

```
model = MLPClassifier(hidden_layer_sizes=(32,), activation='relu',
random_state=42)
```

Model Training

The dataset is split into training and testing sets using train_test_split(). 70% of the data is used for training (X_train and y_train), and 30% is used for testing (X_test and y_test). The model is then trained on the training data using fit():

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded,
test_size=0.3, random_state=42)
model.fit(X_train, y_train)
```

Model Evaluation

Predictions are made on the testing data using predict(), resulting in predicted probabilities for each class. np.argmax() is used to obtain the predicted class labels by selecting the index with the highest probability. The same process is applied to the ground truth labels (y_test) to obtain the true class labels.

The accuracy score and confusion matrix are calculated using accuracy_score() and confusion_matrix() respectively, to evaluate the performance of the model. The accuracy and confusion matrix are then printed:

```
y_pred = model.predict(X_test)
y_pred_labels = np.argmax(y_pred, axis=1)
```

```
y_test_labels = np.argmax(y_test, axis=1)

accuracy = accuracy_score(y_test_labels, y_pred_labels)
confusion_mat = confusion_matrix(y_test_labels, y_pred_labels)

print("Accuracy:", accuracy)
print("Confusion Matrix:")
print(confusion_mat)
```

1.2 Using Tensorflow

Importing the Necessary Libraries

```
import numpy as np
import plotly.graph_objects as go
from sklearn.datasets import load_iris
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
from sklearn.model_selection import train_test_split
```

We import the necessary libraries and modules, including numpy, load_iris from sklearn.datasets to load the Iris dataset, MinMaxScaler and OneHotEncoder from sklearn.preprocessing for data preprocessing, train_test_split from sklearn.model_selection for splitting the data, tensorflow for building the neural network model, Sequential and Dense from tensorflow.keras.layers for defining the model architecture, to_categorical from tensorflow.keras.utils for one-hot encoding the target variable, and matplotlib.pyplot for plotting the convergence.

Dataset Loading

Here, we load the Iris dataset using load_iris() from scikit-learn. iris.data contains the feature vectors, and iris.target contains the corresponding target labels:

```
from sklearn.datasets import load_iris

iris = load_iris()
X = iris.data
y = iris.target
```

Dataset Preprocessing

An instance of MinMaxScaler is created to scale the input features. The feature vectors are then scaled using fit_transform().
An instance of OneHotEncoder is created to perform one-hot encoding on the target labels. The labels are reshaped to a column vector using reshape(-1, 1) and then one-hot encoded using fit_transform().

```
scaler = MinMaxScaler()
X_scaled = scaler.fit_transform(X)

encoder = OneHotEncoder(sparse_output=False)
```

```
y_encoded = encoder.fit_transform(y.reshape(-1, 1))
```

Model Architecture

We create a sequential model using Sequential() from tensorflow.keras.models. This model will contain a series of layers.
We add a dense layer to the model using model.add(Dense(...)). The first dense layer has 32 units and uses the ReLU activation function. It expects input data of shape (4,), which corresponds to the four input features.
We add another dense layer with 3 units and the softmax activation function. This layer represents the output layer, which will produce probabilities for the three classes in the Iris dataset.

```
model = Sequential()
model.add(Dense(32, activation='relu', input_shape=(4,)))
model.add(Dense(3, activation='softmax'))
```

Model Training

We split the scaled input features X_scaled and the encoded target variable y_encoded into training and testing sets using train_test_split() from sklearn.model_selection. The testing set will be 30% of the data, and we set the random seed to 42 for reproducibility. The split data is stored in X_train, X_test, y_train, and y_test.
We compile the model using model.compile() with the categorical cross-entropy loss function, the Adam optimizer, and the accuracy metric.
We train the model on the training data using model.fit(). We specify the number of epochs as 100, the batch size as 16, and set verbose=0 to suppress the training progress output.
The training history is stored in the history variable by calling model.fit() again.

```
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded,
test_size=0.3, random_state=42)

model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
model.fit(X_train, y_train, epochs=100, batch_size=16, verbose=0)
history = model.fit(X_train, y_train, epochs=100, batch_size=16,
verbose=0)
```

Model Evaluation

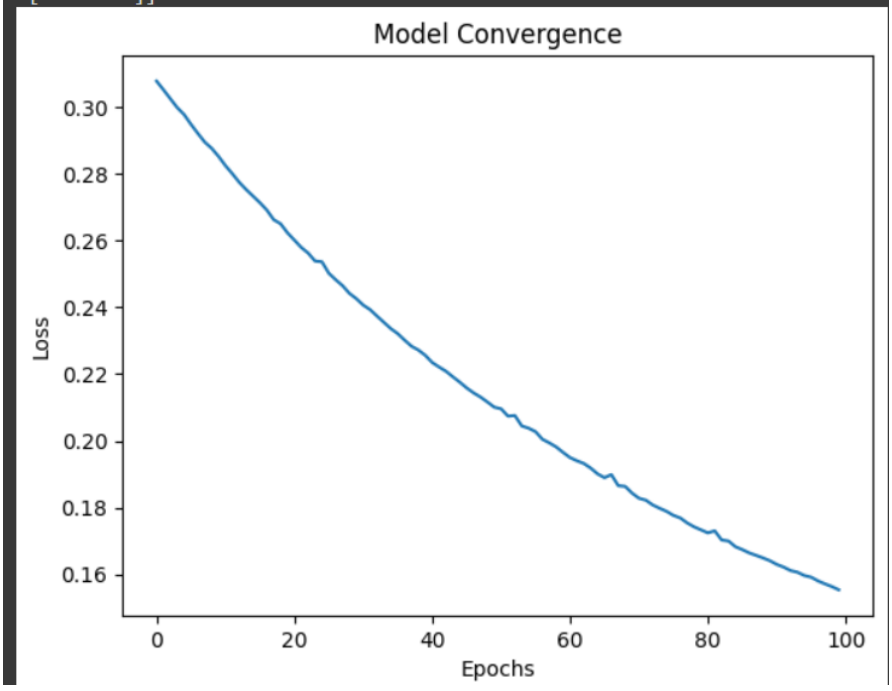
- We use the trained model to predict the labels for the test data X_test using model.predict(). The predicted probabilities for each class are stored in y_pred.
- We use np.argmax() to get the index of the class with the highest probability for each sample in y_pred, resulting in y_pred_labels.
- We also get the true labels for the test data y_test by finding the index of the class with the highest value, stored in y_test_labels.
- We calculate the accuracy by comparing the predicted labels y_pred_labels with the true labels y_test_labels using np.mean().
- We calculate the confusion matrix using tf.math.confusion_matrix() from TensorFlow, providing the true labels and predicted labels as arguments. The confusion matrix is stored in confusion_mat.
- We print the accuracy and the confusion matrix.

- We plot the training loss over the epochs using plt.plot(), providing history.history['loss'] as the data to plot. This data represents the loss values recorded during training.
- We set the title of the plot to 'Model Convergence' using plt.title().
- We label the x-axis as 'Epochs' using plt.xlabel().
- We label the y-axis as 'Loss' using plt.ylabel().
- Finally, we display the plot using plt.show(), which will show the convergence of the model's loss over the training epochs.

```
y_pred = model.predict(X_test)
y_pred_labels = np.argmax(y_pred, axis=1)
y_test_labels = np.argmax(y_test, axis=1)

accuracy = np.mean(y_pred_labels == y_test_labels)
confusion_mat = tf.math.confusion_matrix(y_test_labels, y_pred_labels)

print("Accuracy:", accuracy)
print("Confusion Matrix:")
print(confusion_mat.numpy())

plt.plot(history.history['loss'])
plt.title('Model Convergence')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.show()
```

Output –

```
2/2 [==============================] - 0s 5ms/step
Accuracy: 0.9777777777777777
Confusion Matrix:
[[19  0  0]
 [ 0 12  1]
 [ 0  0 13]]
```



Loss

In the context of neural networks, "loss" refers to the value that measures how well the model is performing on a given task. It quantifies the dissimilarity or error between the predicted output of the model and the true output.

During training, the model aims to minimize this loss by adjusting its parameters (weights and biases) through the process of optimization. The choice of loss function depends on the specific task and the type of data being used.

In the provided code, the loss function used is categorical cross-entropy, which is commonly used for multi-class classification problems. This loss function measures the dissimilarity between the predicted probability distribution and the true one-hot encoded labels. The goal is to minimize this loss, which indicates that the model's predicted probabilities are closer to the true labels.

By monitoring the loss value over the training epochs, you can observe how the model's performance improves. A decreasing loss indicates that the model is learning and converging towards a better solution.