Assignment 1

import numpy as np
import tensorflow as tf
print(tf.__version__)

- Import the NumPy library as `np` and TensorFlow library as `tf`.
- Print the TensorFlow version.

from keras import datasets
(train_images, train_labels), (test_images, test_labels) =
datasets.mnist.load_data()

- Import the `datasets` module from Keras.
- Load the MNIST dataset into four variables: `train_images`, `train_labels`, `test_images`, and `test_labels`.

Train_images.shape
Test_images.shape

- These lines are attempting to access the shape attribute of `Train_images` and `Test_images`, but there are typos in the variable names. It should be `train_images.shape` and `test_images.shape`.

!pip install Theano
- This line uses a Jupyter Notebook magic command (`!pip`) to install the Theano library.

import theano.tensor as T
from theano import function

x = T.dscalar('x')
y = T.dscalar('y')
z = x + y
f = function([x, y], z)
f(5, 7)

- Import the `theano.tensor` module as `T`.
- Import the `function` class from the `theano` package.
- Declare two scalar symbolic variables `x` and `y` using Theano.
- Define a computation graph where `z` is the sum of `x` and `y`.
- Create a callable function `f` that takes `x` and `y` as inputs and returns the sum.
- Call the function `f(5, 7)` which computes the sum of 5 and 7 using Theano and prints the result (12).

!pip3 install torch torchvision torchaudio --extra-index-url https://download.pytorch.org/whl/cu115

- Use `!pip3` to install PyTorch, torchvision, and torchaudio. The `--extra-index-url` flag specifies an additional index URL for downloading packages.

import torch
import torch.nn as nn
print(torch.__version__)
torch.cuda.is_available()

- Import the PyTorch library and the `nn` module.
- Print the PyTorch version.
- Check if CUDA (GPU support) is available by calling `torch.cuda.is_available()`. It returns `True` if a compatible GPU is available; otherwise, it returns `False`.

Assignment 2

```
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import classification_report
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.datasets import mnist
from tensorflow.keras import backend as K
import matplotlib.pyplot as plt
import numpy as np
```

- Import necessary libraries:
  - `LabelBinarizer` for converting class labels to binary representation.
  - `classification_report` for generating a classification report.
  - `Sequential` for creating a sequential neural network model.
  - `Dense` for specifying densely-connected neural network layers.
  - `SGD` for Stochastic Gradient Descent optimizer.
  - `mnist` for loading the MNIST dataset.
  - `K` from Keras backend for specific backend operations.
  - `matplotlib.pyplot` for plotting.
  - `numpy` for numerical operations.

```
((X_train, Y_train), (X_test, Y_test)) = mnist.load_data()
```
- Load the MNIST dataset into `X_train`, `Y_train` (training data and labels) and `X_test`, `Y_test` (test data and labels).

```
X_train = X_train.reshape((X_train.shape[0], 28 * 28 * 1))
X_test = X_test.reshape((X_test.shape[0], 28 * 28 * 1))
```

- Reshape the data from 3D arrays (images) to 2D arrays, where each image is flattened into a 1D array of size 784 (28 * 28).

X_train = X_train.astype("float32") / 255.0
X_test = X_test.astype("float32") / 255.0

- Normalize the pixel values of the images to be between 0 and 1 by dividing by 255. This step is important for neural networks to learn effectively.

lb = LabelBinarizer()
Y_train = lb.fit_transform(Y_train)
Y_test = lb.transform(Y_test)

- Create a `LabelBinarizer` object `lb` to convert class labels to binary representation.
- Transform `Y_train` (training labels) and `Y_test` (testing labels) using the `fit_transform` and `transform` methods respectively.

model = Sequential()
model.add(Dense(128, input_shape=(784,), activation="sigmoid"))
model.add(Dense(64, activation="sigmoid"))
model.add(Dense(10, activation="softmax"))

- Create a sequential neural network model.
- Add layers to the model:
  - First layer: Dense layer with 128 units, input shape of 784 (flattened image size), and sigmoid activation function.
  - Second layer: Dense layer with 64 units and sigmoid activation function.
  - Output layer: Dense layer with 10 units (for 10 classes in MNIST), using softmax activation function for multi-class classification.

sgd = SGD(0.01)
epochs = 10
model.compile(loss="categorical_crossentropy", optimizer=sgd, metrics=["accuracy"])

- Create a Stochastic Gradient Descent (SGD) optimizer with a learning rate of 0.01.
- Compile the model, specifying:
  - Loss function: `"categorical_crossentropy"` for multi-class classification.
  - Optimizer: `sgd` (Stochastic Gradient Descent) with a learning rate of 0.01.
  - Metrics to monitor during training: `"accuracy"`.

H = model.fit(X_train, Y_train, validation_data=(X_test, Y_test), epochs=epochs, batch_size=128)

- Train the model using the training data `X_train` and `Y_train`.
- Validate the model using the validation data `X_test` and `Y_test`.
- Train for `epochs` number of epochs with a batch size of 128.

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

- Use the trained model to make predictions on the test data `X_test`.

print(classification_report(Y_test.argmax(axis=1), predictions.argmax(axis=1), target_names=[str(x) for x in lb.classes_]))

- Generate a classification report comparing the true labels (`Y_test.argmax(axis=1)`) and the predicted labels (`predictions.argmax(axis=1)`) after converting them back from binary representation to class labels.

```
plt.style.use("ggplot")
plt.figure()
plt.plot(np.arange(0, epochs), H.history["loss"], label="train_loss")
plt.plot(np.arange(0, epochs), H.history["val_loss"], label="val_loss")
plt.plot(np.arange(0, epochs), H.history["accuracy"], label="train_acc")
plt.plot(np.arange(0, epochs), H.history["val_accuracy"], label="val_acc")
plt.title("Training Loss and Accuracy")
plt.xlabel("Epoch #")
plt.ylabel("Loss/Accuracy")
plt.legend()
```

- Plot the training loss and accuracy over epochs using `matplotlib.pyplot`.
- The x-axis represents the number of epochs, and the y-axis represents the loss and accuracy values.
- The plot is styled using the "ggplot" style.
- Finally, labels and legend are added to the plot for clarity.

Assignment 3

#importing the libraries
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras import datasets, layers, models

- Import necessary libraries:
  - `matplotlib.pyplot` for plotting.
  - `tensorflow` for creating and training neural networks.
  - `datasets`, `layers`, and `models` modules from `tensorflow.keras` for handling datasets, building neural network layers, and creating models respectively.

#grabbing CIFAR10 dataset
(train_images, train_labels), (test_images, test_labels) =
datasets.cifar10.load_data()

- Load the CIFAR-10 dataset into `train_images`, `train_labels` (training data and labels) and `test_images`, `test_labels` (test data and labels).

train_images, test_images = train_images / 255.0, test_images / 255.0

- Normalize the pixel values of the images to be between 0 and 1 by dividing by 255. This step is important for neural networks to learn effectively.

```
#showing images of mentioned categories
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer','dog', 'frog', 'horse',
'ship', 'truck']
plt.figure(figsize=(10,10))
for i in range(10):
   plt.subplot(5,5,i+1)
   plt.xticks([])
   plt.yticks([])
   plt.grid(False)
   plt.imshow(train_images[i])
   plt.xlabel(class_names[train_labels[i][0]])
plt.show()
```

- This code block visualizes 10 sample images from the CIFAR-10 dataset along with their corresponding labels.
- `plt.figure(figsize=(10,10))` creates a figure with a size of 10x10 inches.
- A loop iterates over the first 10 images in the dataset and displays them with labels using `plt.imshow()` and `plt.xlabel()` functions.
- The resulting images are displayed in a 5x5 grid.

```
#building CNN model
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(64, activation='relu'))
model.add(layers.Dense(10))
```

- Build a convolutional neural network (CNN) model using Keras sequential API.
- `Conv2D` layers create convolutional layers with specified filters and kernel size, using ReLU activation function.
- `MaxPooling2D` layers add max-pooling operations to reduce the spatial dimensions of the output volume.
- `Flatten` layer flattens the 3D output to 1D before passing it to the fully connected layers.
- `Dense` layers are fully connected layers with specified number of units and activation functions.

model.summary()

- Print a summary of the model architecture, displaying the layers, output shapes, and number of parameters.

#model compilation
model.compile(optimizer='adam',
loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
metrics=['accuracy'])

- Compile the model:
  - Optimizer: 'adam', an advanced variant of gradient descent that adapts learning rates for each parameter.
  - Loss function: `SparseCategoricalCrossentropy`, used for multi-class classification problems with integer labels.
  - Metrics: Monitor accuracy during training.

epochs = 1
h = model.fit(train_images, train_labels, epochs=epochs,
validation_data=(test_images, test_labels))

- Train the model using the training data `train_images` and `train_labels`.
- Validate the model using the test data `test_images` and `test_labels`.
- Train for `epochs` number of epochs.
- The training history, including loss and accuracy, is stored in the `h` variable.

Assignment 4

```
#importing libraries and dataset
import numpy as np
import pandas as pd
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from tensorflow.keras.optimizers import Adam
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras import Model, Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from tensorflow.keras.losses import MeanSquaredLogarithmicError
```

- Import necessary libraries: NumPy for numerical operations, Pandas for data manipulation, TensorFlow for building and training neural networks, Matplotlib for plotting, and scikit-learn functions for accuracy calculation and train-test split.

```
PATH_TO_DATA =
'http://storage.googleapis.com/download.tensorflow.org/data/ecg.csv'
data = pd.read_csv(PATH_TO_DATA, header=None)
data.head()
```

- Specify the URL of the dataset and load it into a Pandas DataFrame.
- Display the first few rows of the dataset to understand its structure using `data.head()`.

#finding shape of the dataset
Data.shape

- Print the shape of the dataset, indicating the number of rows and columns.

#splitting training and testing dataset
features = data.drop(140, axis=1)
target = data[140]
x_train, x_test, y_train, y_test = train_test_split(
features, target, test_size=0.2, stratify=target
)

- Split the dataset into features (`features`) and target labels (`target`).
- Split the data into training and testing sets using `train_test_split()`.
- The `stratify=target` argument ensures that the class distribution is preserved in the splits.

#scaling the data using MinMaxScaler
min_max_scaler = MinMaxScaler(feature_range=(0, 1))
x_train_scaled = min_max_scaler.fit_transform(train_data.copy())
x_test_scaled = min_max_scaler.transform(x_test.copy())

- Scale the features using `MinMaxScaler` to normalize the data within the range of 0 to 1.
- `fit_transform()` is used on the training data, and `transform()` is used on the test data after fitting on the training data.

```
#creating autoencoder subclass by extending Model class from keras
class AutoEncoder(Model):
  def __init__(self, output_units, ldim=8):
    super().__init__()
    # Encoder layers
    self.encoder = Sequential([
      Dense(64, activation='relu'),
      Dropout(0.1),
      Dense(32, activation='relu'),
      Dropout(0.1),
      Dense(16, activation='relu'),
      Dropout(0.1),
      Dense(ldim, activation='relu')
    ])
    # Decoder layers
    self.decoder = Sequential([
      Dense(16, activation='relu'),
      Dropout(0.1),
      Dense(32, activation='relu'),
      Dropout(0.1),
      Dense(64, activation='relu'),
      Dropout(0.1),
      Dense(output_units, activation='sigmoid')
    ])

  def call(self, inputs):
    encoded = self.encoder(inputs)
    decoded = self.decoder(encoded)
    return decoded
```

- Define an autoencoder model as a subclass of `Model` from Keras.
- The `AutoEncoder` class consists of an encoder and a decoder.
- The encoder and decoder are defined using the `Sequential` API, specifying the layers and activation functions.

```
#model configuration
model = AutoEncoder(output_units=x_train_scaled.shape[1])
model.compile(loss='msle', metrics=['mse'], optimizer='adam')
epochs = 20

history = model.fit(
    x_train_scaled,
    x_train_scaled,
    epochs=epochs,
    batch_size=512,
    validation_data=(x_test_scaled, x_test_scaled)
)
```

- Create an instance of the `AutoEncoder` class.
- Configure the model: specify the loss function ('msle' - Mean Squared Logarithmic Error), metrics (mean squared error), and optimizer ('adam').
- Train the model using the training data (`x_train_scaled`) and target data (`x_train_scaled`) for a specified number of epochs and batch size.
- Validation data is provided to monitor the model's performance on the test data during training.

```
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.xlabel('Epochs')
plt.ylabel('MSLE Loss')
plt.legend(['loss', 'val_loss'])
plt.show()
```

- Plot the training and validation loss over epochs using Matplotlib.

```
#finding threshold for anomaly and doing predictions
def find_threshold(model, x_train_scaled):
  reconstructions = model.predict(x_train_scaled)
  reconstruction_errors = tf.keras.losses.msle(reconstructions, x_train_scaled)
  threshold = np.mean(reconstruction_errors.numpy()) +
np.std(reconstruction_errors.numpy())
  return threshold

def get_predictions(model, x_test_scaled, threshold):
  predictions = model.predict(x_test_scaled)
  errors = tf.keras.losses.msle(predictions, x_test_scaled)
  anomaly_mask = pd.Series(errors) > threshold
  preds = anomaly_mask.map(lambda x: 0.0 if x == True else 1.0)
  return preds

threshold = find_threshold(model, x_train_scaled)
print(f"Threshold: {threshold}")

#getting accuracy score
predictions = get_predictions(model, x_test_scaled, threshold)
accuracy_score(predictions, y_test)
```

- Define a function `find_threshold()` to calculate the threshold for anomaly detection based on the mean and standard deviation of reconstruction errors.
- Define a function `get_predictions()` to classify anomalies based on the calculated threshold.
- Calculate the threshold using the `find_threshold()` function.
- Get predictions using the `get_predictions()` function.
- Calculate and print the accuracy score of anomaly detection using `accuracy_score()` from scikit-learn.

Assignment 5

```
#importing libraries
from keras.preprocessing import text
from keras.utils import np_utils
from keras.preprocessing import sequence
from keras.utils import pad_sequences
import numpy as np
import pandas as pd
```

- Import necessary libraries: `text` and `sequence` from Keras for text processing, `np_utils` for utilities related to NumPy arrays, `pad_sequences` for padding sequences, `numpy` for numerical operations, and `pandas` for data manipulation.

```
#taking random sentences as data
data = """Deep learning (also known as deep structured learning) is part of a broader family of machine learning methods based on artificial neural networks with representation learning. Learning can be supervised, semi-supervised or unsupervised. Deep-learning architectures such as deep neural networks, deep belief networks, deep reinforcement learning, recurrent neural networks, convolutional neural networks and Transformers have been applied to fields including computer vision, speech recognition, natural language processing, machine translation, bioinformatics, drug design, medical image analysis, climate science, material inspection and board game programs, where they have produced results comparable to and in some cases surpassing human expert performance.
"""
dl_data = data.split()
```

- Define a string `data` containing random sentences.
- Split the string into individual words and store them in `dl_data`.

#tokenization
tokenizer = text.Tokenizer()
tokenizer.fit_on_texts(dl_data)
word2id = tokenizer.word_index

word2id['PAD'] = 0
id2word = {v: k for k, v in word2id.items()}
wids = [[word2id[w] for w in text.text_to_word_sequence(doc)] for doc in dl_data]

- Tokenize the words using the `Tokenizer` class from Keras.
- Create a dictionary `word2id` mapping words to unique integer IDs and add a special token `'PAD'` with ID 0.
- Create a reverse dictionary `id2word`.
- Convert the words into sequences of IDs and store them in `wids`.

vocab_size = len(word2id)
embed_size = 100
window_size = 2

- Calculate the vocabulary size.
- Set the embedding size to 100 and the window size for context words to 2.

print('Vocabulary Size:', vocab_size)
print('Vocabulary Sample:', list(word2id.items())[:10])

- Print the vocabulary size and a sample of word-to-ID mappings for verification.

```
#generating (context word, target/label word) pairs
def generate_context_word_pairs(corpus, window_size, vocab_size):
    context_length = window_size*2
    for words in corpus:
        sentence_length = len(words)
        for index, word in enumerate(words):
            context_words = []
            label_word   = []
            start = index - window_size
            end = index + window_size + 1

            context_words.append([words[i]
                                  for i in range(start, end)
                                  if 0 <= i < sentence_length
                                  and i != index])
            label_word.append(word)

            x = pad_sequences(context_words, maxlen=context_length)
            y = np_utils.to_categorical(label_word, vocab_size)
            yield (x, y)
```

- Define a function `generate_context_word_pairs()` to generate (context word, target word) pairs for the given corpus.
- Iterate through the words in the corpus, and for each word, create context words and their corresponding target words.
- Use `pad_sequences()` to ensure context words have consistent lengths.
- Yield the pairs as `(x, y)` where `x` is the input context and `y` is the target word.

```
#model building
import keras.backend as K
from keras.models import Sequential
from keras.layers import Dense, Embedding, Lambda
```

- Import necessary modules for building the neural network model.

```
cbow = Sequential()
cbow.add(Embedding(input_dim=vocab_size, output_dim=embed_size,
input_length=window_size*2))
cbow.add(Lambda(lambda x: K.mean(x, axis=1), output_shape=(embed_size,)))
cbow.add(Dense(vocab_size, activation='softmax'))
cbow.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

- Build the Continuous Bag of Words (CBOW) model using Keras' sequential API.
- Add an embedding layer, averaging layer, and a dense softmax output layer.
- Compile the model with categorical cross-entropy loss and RMSprop optimizer.

```
print(cbow.summary())
```

- Print the summary of the CBOW model architecture.

```
for epoch in range(1, 6):
    loss = 0.
    i = 0
    for x, y in generate_context_word_pairs(corpus=wids,
window_size=window_size, vocab_size=vocab_size):
        i += 1
        loss += cbow.train_on_batch(x, y)
        if i % 100000 == 0:
            print('Processed {} (context, word) pairs'.format(i))

    print('Epoch:', epoch, '\tLoss:', loss)
    print()
```

- Train the CBOW model for 5 epochs using the generated context-word pairs.
- Print the training loss for each epoch.

```
weights = cbow.get_weights()[0]
weights = weights[1:]
print(weights.shape)
```

- Get the learned weights from the embedding layer of the CBOW model, excluding the padding token.
- Print the shape of the learned weights.

pd.DataFrame(weights, index=list(id2word.values())[1:]).head()

- Create a DataFrame containing the learned word embeddings, excluding the padding token, and display the first few rows.

from sklearn.metrics.pairwise import euclidean_distances

distance_matrix = euclidean_distances(weights)
print(distance_matrix.shape)

- Compute the Euclidean distance matrix between word embeddings.
- Print the shape of the resulting distance matrix.

similar_words = {search_term: [id2word[idx] for idx in
distance_matrix[word2id[search_term]-1].argsort()[1:6]+1]
for search_term in ['deep']}

- Find the 5 most similar words for the search term `'deep'` based on the computed distance matrix.

similar_words

- Print the dictionary containing similar words for the search term `'deep'`.