Assignment 2 – Implementing Feedforward neural networks with Keras and TensorFlow

This code is for creating a simple neural network using TensorFlow and Keras to recognize handwritten digits from the MNIST dataset, which contains 28x28 pixel images of handwritten digits from 0 to 9. Here's a breakdown of what the code does:

1. Import necessary libraries: The code starts by importing TensorFlow, Keras, Pandas, NumPy, and Matplotlib for working with data and creating and visualizing the neural network.

2. Load the MNIST dataset: The code loads the MNIST dataset, which consists of a training set (x_train and y_train) and a test set (x_test and y_test).

3. Data exploration: It prints the length and shape of the training and testing datasets to give you an idea of the dataset's size.

4. Normalize the images: The pixel values of the images are scaled between 0 and 1 by dividing each pixel value by 255. This is a common preprocessing step for neural networks.

5. Create a simple neural network model: The code defines a sequential model with three layers. The first layer is a flattening layer that converts the 28x28 images into a 1D array. The second layer is a dense layer with 128 neurons and a ReLU activation function. The third layer is another dense layer with 10 neurons and a softmax activation function, which is used for multiclass classification.

6. Compile the model: The model is compiled with stochastic gradient descent (SGD) as the optimizer, sparse categorical cross-entropy as the loss function, and accuracy as the evaluation metric.

7. Train the model: The model is trained on the training data for 10 epochs, and the training history is stored in the 'history' variable.

8. Evaluate the model: The code evaluates the model's performance on the test data and prints the test loss and accuracy.

9. Display a random test image: It displays a randomly selected image from the test dataset.

10. Make predictions: The code uses the trained model to make predictions on the test data and prints the predicted digit for the randomly selected test image.

11. Plot accuracy and loss graphs: The code uses Matplotlib to create two graphs. One graph shows the training and validation accuracy over epochs, and the other shows the training and validation loss over epochs.

Overall, this code is a simple example of how to build, train, and evaluate a neural network for image classification using TensorFlow and Keras. It's a good starting point for understanding the basic steps involved in creating a machine learning model.

Assignment 3 - Build the Image classification model

This code is for training and using a convolutional neural network (CNN) to classify images of cats and dogs. Here's a simplified explanation of what each part does:

1. Import necessary libraries:

- Imports libraries like NumPy for numerical operations, random for generating random numbers, Matplotlib for visualization, and TensorFlow/Keras for building and training the neural network.

2. Load the training and testing data:

- Loads image data from CSV files ('input.csv' and 'labels.csv') for training and ('input_test.csv' and 'labels_test.csv') for testing.

- Reshapes the data to have the right dimensions.

- Normalizes the image data to values between 0 and 1.

3. Create and display a random training image:

- Picks a random image from the training data and displays it using Matplotlib.

4. Build a Convolutional Neural Network (CNN) model:

- Creates a sequential model, which is a linear stack of layers.

- Adds layers to the model:

  - Two convolutional layers with 32 filters and a 3x3 filter size, using the ReLU activation function.

  - Two max-pooling layers to reduce the spatial dimensions.

  - Flattens the output to prepare for the fully connected layers.

  - Two fully connected dense layers with 64 and 1 neuron(s) respectively, using ReLU and sigmoid activation functions.

5. Compile the model:

- Configures the model for training by specifying the loss function (binary cross-entropy), optimizer (Adam), and performance metric (accuracy).

6. Train the model:

- Fits the model to the training data for a specified number of epochs (5) and a batch size of 64.

7. Evaluate the model on the testing data:

- Calculates the model's performance on the testing dataset and prints the results (e.g., loss and accuracy).

8. Make predictions on a random testing image:

- Selects a random image from the testing dataset and displays it.

- Uses the trained model to predict whether the image is a cat or a dog.

- Compares the prediction to a threshold (0.5) to determine if it's a cat or a dog and prints the result.

In summary, this code loads image data, builds and trains a CNN model to classify cats and dogs, and then uses the trained model to predict the animal in a random test image. The model is a binary classifier, which means it predicts either "cat" or "dog" for each image.

Assignment 4 – Anomaly detection

This code is an example of an anomaly detection system using a neural network (autoencoder) in TensorFlow. Let's break down the key components:

1. Data Loading and Preprocessing:

- The code starts by downloading an ECG dataset and loading it into a Pandas DataFrame.

- It separates the last column as "labels" and the rest of the data as "data."

- The data is then split into training and testing sets, with 80% for training and 20% for testing.

- Data normalization is performed by scaling it between 0 and 1.

2. Data Visualization:

- Two ECG plots are displayed, one normal and one anomalous, to give an idea of what the data looks like.

3. Model Definition:

- An autoencoder neural network is defined. It consists of an encoder and a decoder.

- The encoder reduces the dimensionality of the input data.

- The decoder attempts to reconstruct the original data from the encoded representation.

4. Model Training:

- The autoencoder is trained using the "normal" ECG data, with the goal of making it learn to reconstruct normal ECG signals well.

- The model is optimized using the Adam optimizer and Mean Absolute Error (MAE) loss.

- Training history (losses) is recorded and plotted.

5. Data Reconstruction and Visualization:

- The code reconstructs ECG data and plots the original data, the reconstructed data, and the error (difference) between them.

- This helps visualize how well the model is capturing normal and anomalous patterns.

6. Anomaly Detection:

- The code calculates the Mean Absolute Error (MAE) loss between the original and reconstructed data for both normal and anomalous ECG data.

- It creates a "threshold" for anomaly detection, based on the mean loss plus one standard deviation of the training loss.

7. Predictions and Evaluation:

- It uses the threshold to predict anomalies in the test data (both normal and anomalous) and stores the results.

- The code calculates and prints accuracy, precision, and recall as evaluation metrics for anomaly detection.

In summary, this code trains an autoencoder to reconstruct normal ECG data and then uses the reconstruction error to detect anomalies in ECG signals. Anomalies are identified based on how much the reconstruction differs from the original data. The code ultimately evaluates the model's performance using accuracy, precision, and recall.

Assignment 5 - Simple word embedding and language model (CBOW Model).

This code is an implementation of a simple word embedding and language model using a neural network. It tries to learn word representations from a given text and use them to predict the most likely word to come next in a sentence. Here's a simplified explanation of the code:

1. Import necessary libraries: The code starts by importing the required libraries, such as `re` for regular expressions, `numpy` for numerical operations, and `matplotlib` for plotting.

2. Preprocess the text:

- Special characters are removed from the text, leaving only letters and numbers.

- Single-letter words are also removed.

- All characters are converted to lowercase.

3. Tokenize the text:

- The text is split into individual words, and a set of unique words (vocabulary) is created.

4. Define model parameters:

- The code defines parameters like vocabulary size, embedding dimensions, and context size.

5. Create data for training:

- The code constructs training data by creating context-target pairs. For each word in the text, a context of two words before and two words after is selected, along with the target word.

6. Initialize word embeddings:

- Random embeddings are generated for each word in the vocabulary.

7. Define some helper functions:

- `linear`: Computes a linear transformation.

- `log_softmax`: Calculates the log of the softmax function.

- `NLLLoss`: Computes the negative log-likelihood loss.

- `log_softmax_crossentropy_with_logits`: Computes the loss with respect to the target.

- `forward`: Performs the forward pass through the network to get predictions.

- `backward`: Calculates gradients during the backpropagation process.

- `optimize`: Updates model parameters using gradient descent.

8. Initialize model parameters (`theta`) with random values.

9. Training loop:

- The code enters a loop that goes through a specified number of epochs.

- Within each epoch, it iterates through the training data.

- For each context-target pair, it computes predictions, calculates the loss, and updates model parameters using gradient descent.

10. Collect and plot loss values: Loss values at each epoch are collected and plotted to visualize the training progress.

11. Define a `predict` function:

- This function takes a list of words as input and predicts the most likely word to come next.

12. Calculate accuracy:

- An accuracy function is defined to evaluate the model's performance. It checks how often the model correctly predicts the target word for the given context.

Overall, this code trains a simple neural network to understand the relationships between words in a text and predict the next word in a sequence. It uses word embeddings and gradient descent to optimize its performance.

Assignment 6-Transfer Learning

This code is written in Python and uses the TensorFlow and Keras libraries for deep learning to train a model for classifying images of flowers. Let me explain it step by step in a simple way:

1. Import necessary libraries:

- `matplotlib.pyplot` for creating plots and graphs.

- `numpy` for numerical operations.

- `os` for interacting with the operating system.

- `PIL` (Python Image Library) for working with images.

- `tensorflow` and its submodules for deep learning.

- `pathlib` for working with file paths.

2. Download flower images dataset:

- The code defines the URL of a flower images dataset and downloads it using `tf.keras.utils.get_file`.

- The dataset is stored in a directory, and its path is printed.

3. Load flower images:

- The code lists all the images of sunflowers and roses in the dataset using the `glob` function and stores their file paths in `all_sunflowers` and `all_roses` lists.

- It then opens and displays the first image from each category.

4. Set image dimensions and batch size:

- The height and width for the images are set to 180x180 pixels.

- The training batch size is set to 32.

5. Create training and validation datasets:

- It uses `tf.keras.preprocessing.image_dataset_from_directory` to create training and validation datasets from the flower dataset.

- The dataset is split into 80% training and 20% validation.

- Images are resized to 180x180 pixels, and the batch size is set to 32.

- The class names are printed.

6. Define the deep neural network (DNN) model:

- A new DNN model is created using `Sequential`.

- A pre-trained ResNet50 model is imported using `tf.keras.applications.ResNet50`.

- The ResNet50 layers are added to the new model, with their weights frozen (not trainable).

- A `Flatten` layer is added to convert the 2D features into a 1D vector.

- Two fully connected layers are added with 512 and 5 units, respectively, using ReLU activation for the first layer and softmax activation for the final output layer.

7. Model summary:

- The model's summary, including layer details and the number of trainable parameters, is printed.

8. Compile the model:

- The model is compiled with the Adam optimizer (learning rate of 0.001), sparse categorical cross-entropy loss, and accuracy as the metric.

9. Train the model:

- The model is trained using the training dataset (`train_set`) with validation on the validation dataset (`validation_set`) for 10 epochs.

- Training history, including loss and accuracy, is stored in the `history` variable.

This code essentially downloads a flower image dataset, sets up training and validation data, builds a deep learning model based on the ResNet50 architecture, compiles it, and trains it to classify flower images into five categories. The model's performance is monitored using the training history.