Final Project

~

# EEG Dataset Classifier

Arnav Goyal - 251244778

Graeme Watt - 251201502

*April 8, 2024*

## Overview

In this report, we will talk about our Final Project, where we used a pytorch classifier to predict classes of the dataset in Task 3, more specifically the EEG Brain DE Features Dataset. This dataset will be referred to as *the dataset* from here on out. Furthermore our entire codebase can be found at my (arnavs) github repository. I already had a repository for everything in this year, and i was too lazy to add Graeme to the repository, so all the commits are from me (arnav), but we both worked on the code present there equally.

## The Code Base

Before we start talking about the individual methods and processes present within our codebase, we would like to present a brief overview of our entire codebase and explain the purpose of each file. The files/directories present in our code base are listed below.

- `root.py`
- `models.py`
- `dataset.py`
- `load_data.py`
- `hyperparams.json`
- `data/`
- `checkpoints/`

`root.py` is the backbone of our codebase and holds the methods used to test, train, load from and save checkpoints of various models, it also contains the entry point (main method) of the codebase. It works with two other key scripts `models.py` and `dataset.py`. `models.py` is a script that contains pytorch classes with differing network architectures, at the moment it only contains the finalized network we used, but it was useful in holding many different classes during the training stages of the project. `dataset.py` is a script that casts the data from the dataset into pytorch DataSet classes that work with pytorch DataLoaders. `load_data.py` is a script that comes directly from the github of the dataset and we invoke it within the `dataset.py` script. These 4 files are the only script files in the entire codebase, the other files are just for storage/modification purposes.

The `hyperparams.json` file holds the training parameters the script uses for training. Its data can be changed easily to alter the training hyperparameters. The `data/` directory must be present within the codebase as this is where the dataset is downloaded to. The `checkpoints/` directory must also be present in the codebase. In the file we provide as a submission, this directory will contain a state dictionary with our most accurate model, which you can easily load by running `root.py`, this will be explained later in the report.

## Method Descriptions

Now we will describe the methods present within each file in our codebase.

`root.py`

- `train()`: trains a model according to some architecture which can be defined in this script, using some defined hyperparameters in `hyperparameters.json`, on the training part of the EEG dataset. It has a nice progress bar and shows the training curve after the training loop is done.
- `test()`: tests a model according to some architecture which can be defined in this script, on the testing portion of the EEG dataset. It also has a nice progress bar and prints the testing accuracy after completion of the testing loop
- `printInfo()`: displays the class name of the model being trained, the device on which it is being trained on, the optimizer it is using, and the training hyperparameters used, these hyperparameters are located in `hyperparameters.json`.
- `plot()`: assumes the list passed into the function is a list of losses per epoch, it plots them against epoch, and displays them with labeled axes.
- `save_weights()`: prompts the user of the program for input, and either saves weights to the `checkpoints/` directory as a `.pth` file or discards them.
- `load_weights()`: prompts the user of the program for input, and either loads the model architecture from a `.pth` file located in the `checkpoints/` directory that is compatible with the current model architecture chosen, or it loads the current model architecture with random parameters initially
- `main()`: holds the main method of the codebase. It tells the user what model architecture they are using, asks if they would like to use any of the saved weights if they exist for that architecture, and then asks the user if they would like to further train the model, or just test the model. Based on the users input the program then trains, displays the training curve, tests and prints test accuracy, or it tests and prints test accuracy.

`dataset.py`

- `eegpca()`: performs principal component analysis (with components that make up 99% of the variance in the original data) on the training and testing EEG dataset classes.
- This file also contains the classes `EEGTrainingData` and `EEGTestingData` which were created in order to cast the EEG dataset from github to pytorch DataSet classes so that we can use them in pytorch DataLoaders

`load_data.py`

- `maybe_download()`: If there is no directory of the predefined DATA_DIRECTORY name, the function makes a directory with that name. If there is no file within DATA_DIRECTORY, the function downloads the dataset file from the SOURCE_URL.
- `dense_to_one_hot()`: Takes the labeled class data, and using one-hot encoding, translates it to have three features per datapoint, one for each of the three classes being predicted.

- `load_data()`: opens the passed filename, and using pickle, encodes the file data into a dictionary. After splitting the dictionary into data and labels, uses `dense_to_one_hot()` if it is not already one-hot encoded, and returns the data and labels
- `DataSet()`: a class containing an `__init__()` function, 4 property functions, and one other function, this was not used in our code. However we are still going to document it.
  - `__init__()`: Asserts the length of the data and labels being passed are equal. Sets the _num_examples variable as the length of the data, _data as the passed data, _labels as the passed labels, _epochs_completed, and _index_in_epoch equal to zero.
  - `@property data()`: Returns the objects _data variable
  - `@property labels()`: Returns the objects _labels variable
  - `@property num_examples()`: Returns the objects _num_examples variable
  - `@property epochs_completed()`: Returns the objects _epochs_completed variable
  - `next_batch()`: Increases the _index_in_epoch by the passed batch_size variable. If _index_in_epoch is greater than _num_examples, _epochs_completed is incremented by 1, the data and corresponding labels are randomly shuffled, _index_in_epoch is set to the batch_size, and batch_size is asserted to be less than or equal to _num_examples. Finally, a slice of _data and _labels are returned, from the start position (_index_in_epoch at the start of the function call), to the end position (_index_in_epoch at the end of the function)

- `read_data_sets()`: Makes a DataSets object, taking an object type, and not specifying a constructor. Makes a train_filename and test_filename by calling maybe_download with train and test respectively. One-hot encodes the data and labels by calling load_data with both filenames. Creates a DataSets object, data_sets, then adding a train and test to data_sets, with data and labels from the previous step, before returning data_sets.

## Dataset Extraction

Since we are using the EEG dataset, we implemented the methods defined in the GitHub repository. In addition to performing the standard loading that was predefined. This included using the load_data.py file provided. Within this file, are five library imports, three standalone methods, one class with 6 built in methods, and one method which implements the class

We also performed Principal Component Analysis on the data, to reduce the input featureset from 310 features. After performing three tests of PCA, we found that 16 features were needed to explain 90% of the variance in the training data, 26 features were needed to explain 95% of the variance, and 66 features were needed to explain 99% of the variance. Since even having 66 features was a drastic reduction, nearly five times, compared to the original features, we decided on the 66 feature reduced set, as it would hopefully still give us a high accuracy.

To implement this PCA, all that was needed was a short method called eegpca, in the dataset.py file, and calling eegpca on the trainset and testset within root.py, before making the dataloaders. The code required for eegpca, along with the required import statement has been placed below.

*Note:* This is the only code snippet in this report, the source code is provided with the submission, which means it would be redundant to place it here as well.

```
from sklearn.decomposition import PCA


def eegpca(trainset:EEGTrainingData, testset:EEGTestingData):
pca = PCA(n_components=0.9)
pca.fit(trainset.eeg_train_data)
test_data = pca.transform(testset.eeg_test_data)
testset.eeg_test_data = torch.from_numpy(test_data).float()
train_data = pca.transform(trainset.eeg_train_data)
trainset.eeg_train_data = torch.from_numpy(train_data).float()
```

**Code Snippet 1:** Code required to perform PCA on the EEG dataset

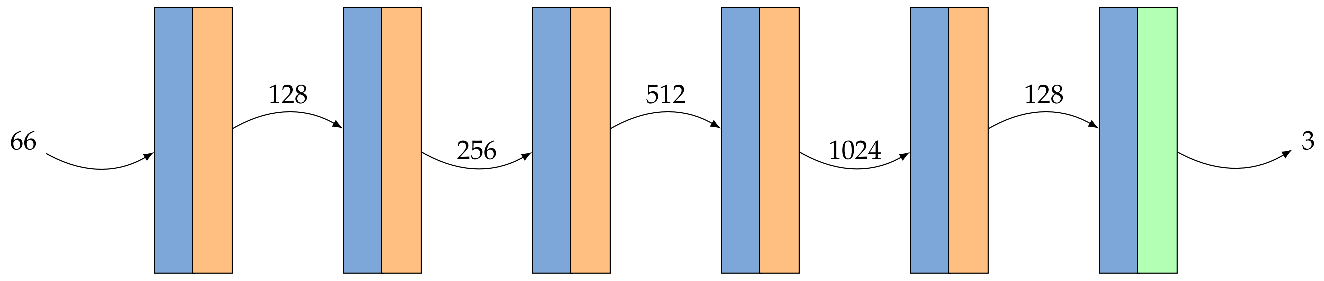
## Network Architecture

The network architecture we used is defined in the models.py file. The most accurate network architecture we were able to find was a network of 6 fully connected layers, with a normalization layer between each pair of layers. For our activation function, we used the ReLu function on the first through fifth layer, and softmax on the final layer, since this is a classification problem. We also chose Cross Entropy Loss as our loss function, Stochastic Gradient Descent with Momentum, SGDM, as our optimizer.

For the individual layers, we began with a fully connected layer with 66 input features, which is the feature size of our reduced dataset, and 128 output features. For each of the subsequent layers, we used Lazy Linear layers, which automatically infer the number of input features from the previous layer. This made the rest of the layers have the following number of input and output features:

- 66 input, 128 output
- 128 input, 256 output
- 256 input, 512 output
- 512 input, 1024 output
- 1024 input, 128 output
- 128 input, 3 output

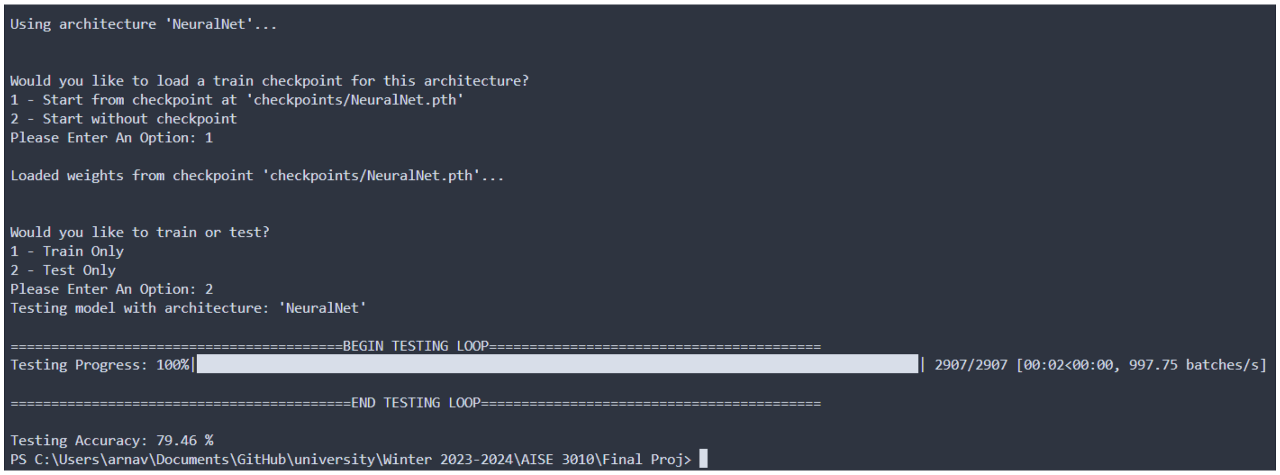
The final layer, with 3 output features represents the three classes to predict. This architecture is depicted via a really nicely drawn LaTeXdiagram below.



**Architecture 1:** The architecture we used for the classifier.
*The blue blocks represent linear pytorch layers, the orange blocks represent the relu activation, the green block represents a concluding softmax activation. The number of features through the model is shown on the arrows.*

## Results & Verification

After training our model with a batch size of 32, for 20 epochs at a learning rate of 0.001 and momentum of 0.85 we produced a model that can obtain a testing accuracy of 79.46%, We were supposed to average it over two test runs, but our model attains this accuracy every time during the testing loop.



```
Using architecture 'NeuralNet'...


Would you like to load a train checkpoint for this architecture?
1 - Start from checkpoint at 'checkpoints/NeuralNet.pth'
2 - Start without checkpoint
Please Enter An Option: 1

Loaded weights from checkpoint 'checkpoints/NeuralNet.pth'...


Would you like to train or test?
1 - Train Only
2 - Test Only
Please Enter An Option: 2
Testing model with architecture: 'NeuralNet'

========================================BEGIN TESTING LOOP========================================
Testing Progress: 100%|████████████████████████████████████████████████████████████| 2907/2907 [00:02<00:00, 997.75 batches/s]

=========================================END TESTING LOOP=========================================

Testing Accuracy: 79.46 %
PS C:\Users\arnav\Documents\GitHub\university\Winter 2023-2024\AISE 3010\Final Proj> █
```

**Image 1:** The verified 79.46% accuracy on our VSCode environment

In order to verify this result yourself, extract our entire submission into a folder and run root.py to download the dataset, once the dataset is downloaded you should see a prompt asking you if you would like to load a train checkpoint. Enter a 1 into the console to load our pretrained model located at checkpoints/NeuralNet.pth. After that the program will ask if you want to train further, or test the model. In this scenario enter a 2 into the console and the program will begin the testing loop, and will print testing accuracy after it is completed. If you have any questions related to program execution please email: agoyal57@uwo.ca and gwatt23@uwo.ca