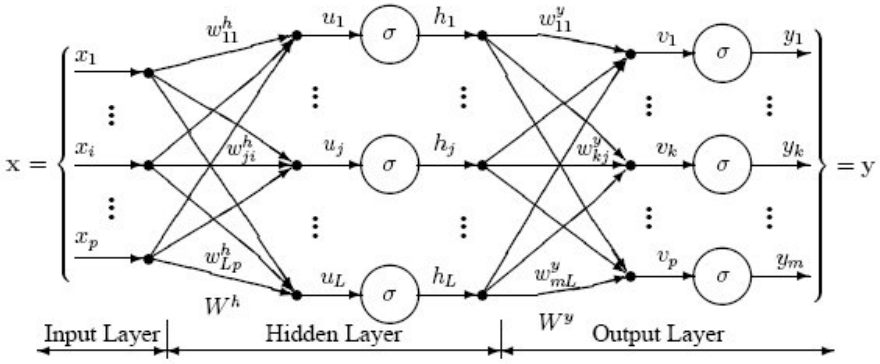
Different Neural Network types and architectures for multiclass image classification

Vasileios Papastergios (Academic ID: 3651)

November 26, 2023



Aristotle University of Thessaloniki
School of Informatics

SCHOOL OF INFORMATICS

Course name:	Neural Networks - Deep Learning
Course ID:	NDM-07-05
Semester:	7th
Professor:	Anastasios Tefas

Contents

List of Figures

1 Introduction

The present document serves as technical report for the programming assignments in the course of Neural Networks - Deep Learning. The author attended the course during their 7^{th} semester of BSc. studies at the School of Informatics, AUTh. The topic of the assignments is implementing various Neural Network types and architectures to solve a multi-class image classification problem. The present document is gradually authored (on the fly) throughout the semester and, in its final form, it will contain all three solutions on the assignment series, as separate chapters.

The first programming assignment emphasizes on implementing a *Multilayer Perceptron (MLP)* Neural Network for a multiclass image classification problem (dataset) of our own choice. In the following sections we thoroughly describe the selected dataset, the process of implementing the MLP, as well as comparing its performance over "classic" classification techniques, such as *K-Nearest Neighbors (KNN)* and *Nearest Centroid (NC)* classifiers on the same dataset.

2 The dataset

The dataset we are going to use throughout this assignment series is named **Intel Image Classification** dataset and can be found on https://www.kaggle.com/datasets/puneet6060/intel-image-classification. The latter is a *multiclass* image classification dataset. The dataset consists of 14K images for training and 3K images for testing. Each image belongs to one of the 6 (mutually exclusive) following classes:

1. buildings
2. forest
3. glacier
4. mountain
5. sea
6. street

For enhanced understanding of the classification task, we opt for taking a random sample of the dataset images. Figure 1 shows a representative for each one of the six classes, as they emerged from the random sampling.

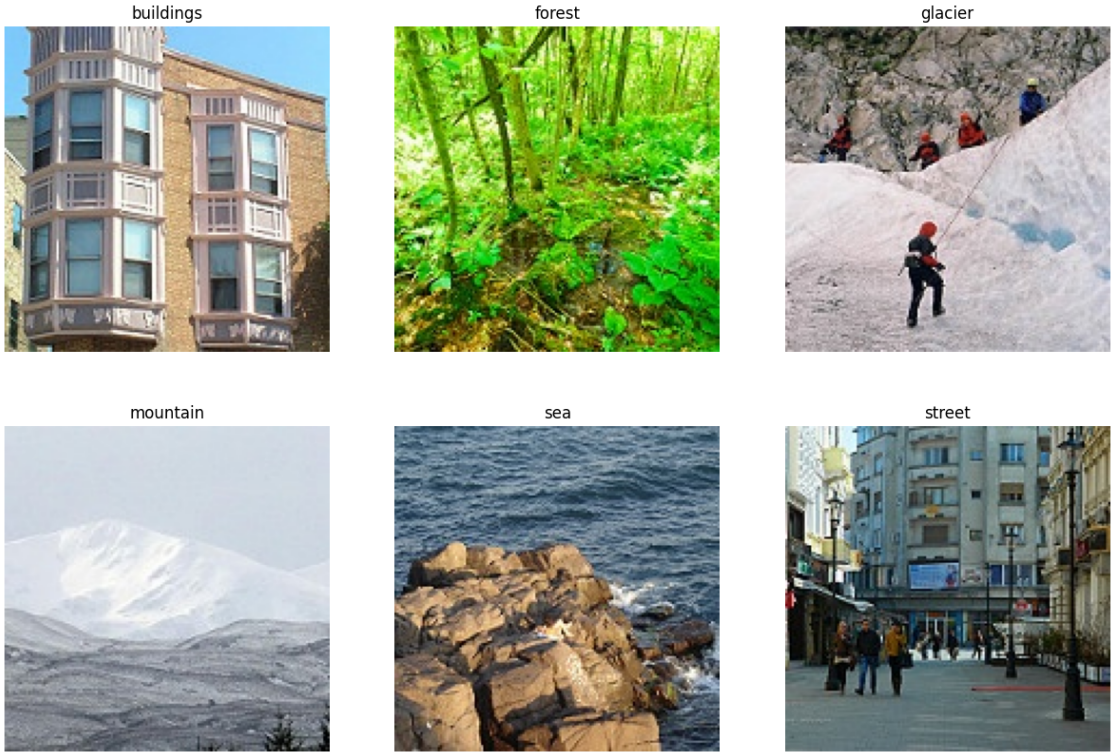


Figure 1: A random sample of the dataset images

2.1 The train set

2.1.1 Train data exploration

The train set consists of 14′034 images in total. We observe that the images are relatively well-balanced among the six classes. Figure 2 is a bar plot that depicts the appearance frequencies of the six classes in the train set. The frequencies distribution has mean $\mu = 2'339$ images per class and standard deviation $\sigma \approx 105.5$.

Another important and useful observation we would like to mention is the shape of the train set images. The shape of the train images is crucially important during the training procedure, since the images will have to be flattened for several Neural Network architectures we are going to implement. As a result, the image shapes need to be the same, so that the (flattened) input vector $\vec{\mathbf{X}}$ has the same dimensions in all training examples.

In practice, all train images are RGB; thus they have a red, green and blue channel. The vast majority of the train images have 150×150 pixels in each channel, which results in a `(150, 150, 3)` shape per image. However, as shown in figure 3, there are also a few images in the train set that have different shapes. In particular, the 13′986 out of the 14′034 ($\approx 99.66\%$) images have shape `(150, 150, 3)`, which is the vast majority of the train images. We will come back to this observation when reaching the step of train data pre-processing.

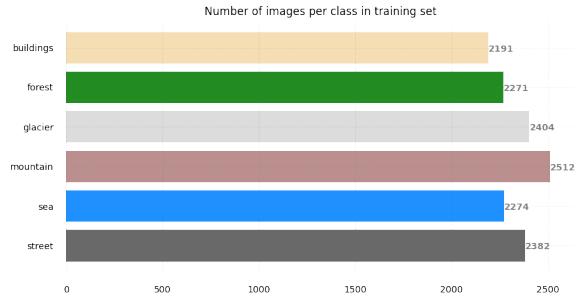


Figure 2: Appearance frequencies of the classes in train set

Figure 3: Appearance frequencies of image shapes in train set

2.1.2 Constructing a `train.csv` file

At this point, we consider useful to present the exact form the train set directories are structured in the dataset, as well as the way we construct a convenient `train.csv` file for more readable and clean code writing later on. More specifically, we download the dataset into our Google Drive storage directly using the following Kaggle *API command*:

```
kaggle datasets download -d puneet6060/intel-image-classification
```

The above command downloads the directories of the dataset. The train set images are located inside the `\seg _train\seg_train` folder. Inside that folder, one can find 6 sub-directories, one for each class with respective names. Taking this into account, we can simply construct a convenient `train.csv` file, that has the following three columns: `filepath`, `label_verbose` and `label`, where `label_verbose` is the name (word) of the class (e.g. "buildings"), while the `label` is the numeric representation (class ID) of it. Code Listing 1 contains the code snippet for creating the `train.csv` file, while Figure 4 shows the first rows of its contents.

```python
def get_file_list(directory):
  '''
  a utility function that gets a directory and recursively explores it
  returns a 2D array, containing records in the form [path, parent_directory_name]
  '''
  file_list = os.listdir(directory)
  all_files = []
  for entry in file_list:
    path = os.path.join(directory, entry)
    parent_directory = path.split('/')[-2]
    if os.path.isdir(path):
      all_files = all_files + get_file_list(path)
    else:
      all_files.append([path, parent_directory])
  return all_files

path = '/gdrive/My Drive/Kaggle/seg_train/seg_train/'
all_files_list = get_file_list(path)
train_data = pd.DataFrame(data = all_files_list, columns=['filepath', 'label_verbose'])
train_data['label'] = train_data['label_verbose'].map(labels_dict)
```

Listing 1: Python code to construct a convenient `train.csv` file

	filepath	label_verbose	label
6277	/gdrive/My Drive/Kaggle/seg_train/seg_train/gl...	glacier	2
12623	/gdrive/My Drive/Kaggle/seg_train/seg_train/st...	street	5
4314	/gdrive/My Drive/Kaggle/seg_train/seg_train/fo...	forest	1
4711	/gdrive/My Drive/Kaggle/seg_train/seg_train/gl...	glacier	2
10549	/gdrive/My Drive/Kaggle/seg_train/seg_train/se...	sea	4
5169	/gdrive/My Drive/Kaggle/seg_train/seg_train/gl...	glacier	2
5595	/gdrive/My Drive/Kaggle/seg_train/seg_train/gl...	glacier	2
3054	/gdrive/My Drive/Kaggle/seg_train/seg_train/fo...	forest	1
3468	/gdrive/My Drive/Kaggle/seg_train/seg_train/fo...	forest	1
2246	/gdrive/My Drive/Kaggle/seg_train/seg_train/fo...	forest	1

Figure 4: First 10 rows of the `train.csv` file

2.2 The test set

2.2.1 Test data exploration

The test set consists of 3′000 images in total. The images are, again, relatively well-balanced among the six classes. Figure 5 is a bar plot that depicts the appearance frequencies of the six classes in the test set. The frequencies distribution has mean $\mu = 500$ images per class and standard deviation $\sigma \approx 36.9$.

Exactly as we did for the train set, we note the shape of the test set images. Similarly, the train set images need to have the same shape, so that they match the expected input, when reaching the inference step, later on our analysis. All test images are RGB; thus they have a red, green and blue channel. The vast majority of the test images have 150×150 pixels in each channel, which results in a `(150, 150, 3)` shape per image. However, as shown in figure 6, there are also a few images in the test set that have different shapes. In particular, the 2′993 out of the 3′000 ($\approx 99.77\%$) images have shape `(150, 150, 3)`, which is the vast majority of the test images. We will come back to this observation when reaching the step of test data pre-processing.

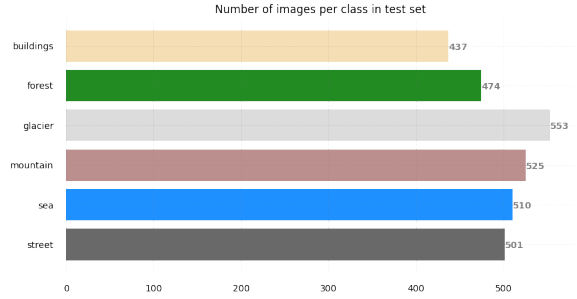


Figure 5: Appearance frequencies of the classes in test set

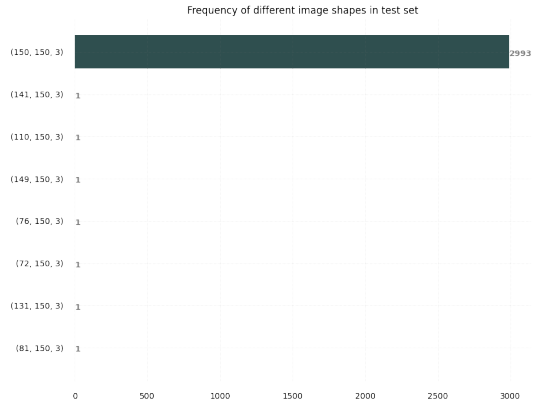


Figure 6: Appearance frequencies of image shapes in test set

2.2.2 Constructing a `test.csv` file

Completely similarly with the train data, we can construct a `test.csv` file for our own convenience in, later, in the inference step. We can use the same code snippet presented in Listing 1, altering the path to point at the test set folder `\seg _test\seg_test`. Figure 4 shows the first rows of its contents.

	filepath	label_verbose	label
1198	/gdrive/My Drive/Kaggle/seg_test/seg_test/glac...	glacier	2
1800	/gdrive/My Drive/Kaggle/seg_test/seg_test/moun...	mountain	3
2196	/gdrive/My Drive/Kaggle/seg_test/seg_test/sea/...	sea	4
2462	/gdrive/My Drive/Kaggle/seg_test/seg_test/sea/...	sea	4
1699	/gdrive/My Drive/Kaggle/seg_test/seg_test/moun...	mountain	3
2143	/gdrive/My Drive/Kaggle/seg_test/seg_test/sea/...	sea	4
570	/gdrive/My Drive/Kaggle/seg_test/seg_test/fore...	forest	1
1400	/gdrive/My Drive/Kaggle/seg_test/seg_test/glac...	glacier	2
1230	/gdrive/My Drive/Kaggle/seg_test/seg_test/glac...	glacier	2
246	/gdrive/My Drive/Kaggle/seg_test/seg_test/buil...	buildings	0

Figure 7: First 10 rows of the `test.csv` file

3 Traditional classification techniques

Before diving into Neural Network architectures to address the classification problem, we try to set a baseline for our work. In this section we try to approach the multiclass classification problem using "classic" (non Neural Network) classification techniques. In particular, we investigate the performance of different variations of a K-Nearest Neighbors (KNN) classifier (for $k = 1$ and $k = 3$ neighbors), as well as a Nearest Centroid classifier.

3.1 Baseline solution 1: The K-Nearest Neighbors (KNN) classifier

We implement two variations of the KNN classifier: one for $k = 1$ and one for $k = 3$ neighbors. We utilize the `scikitlearn` Python library, and, more specifically, the `KNeighborsClassifier` module. Code listing 2 can provide the reader with the simple `Python3` code needed to create a KNN classifier and fit it to the train data. As shown in the code listing, we save (`dump`) the classifier to a `.joblib` file, to be able to make multiple inferences (loading it from disk) whenever needed, instead of fitting the train data from start again and again.

```
from sklearn.neighbors import KNeighborsClassifier
from joblib import dump

knn_classifier1 = KNeighborsClassifier(n_neighbors=1)
knn_classifier1.fit(X_train, y_train)
dump(knn_classifier1, '/gdrive/My Drive/Kaggle/classifiers/knn1.joblib')
y_pred_knn1 = knn_classifier1.predict(X_test)
print('KNN 1')
print(classification_report(y_test, y_pred_knn1))
```

Listing 2: Python code to construct and train a KNN classifier for $k = 1$ neighbor (KNN-1)

Having created and trained the classifier, the next step is to use it, in order to make inferences on the test data and assess its performance. Figure 8 shows several evaluation metrics for the KNN-1 model. The best accuracy we can achieve with the model is 35%. We consider this accuracy as a baseline for the classification problem we are examining.

KNN 1

	precision	recall	f1-score	support
0	0.48	0.03	0.06	437
1	0.62	0.30	0.40	473
2	0.41	0.36	0.38	549
3	0.34	0.72	0.46	523
4	0.24	0.49	0.33	510
5	0.68	0.16	0.25	501
accuracy			0.35	2993
macro avg	0.46	0.34	0.31	2993
weighted avg	0.46	0.35	0.32	2993

Figure 8: Model evaluation metrics for KNN-1 model

Similarly, we can easily construct a second variation of the same traditional classification technique; a KNN classifier for $k = 3$ neighbors this time. Code listing 3 demonstrates the trivial way of constructing and training such a classifier, using again the `scikit-learn` Python library.

```
from sklearn.neighbors import KNeighborsClassifier
from joblib import dump

knn_classifier3 = KNeighborsClassifier(n_neighbors=3)
knn_classifier3.fit(X_train, y_train)
```

```
dump(knn_classifier3, '/gdrive/My Drive/Kaggle/classifiers/knn3.joblib')
y_pred_knn3 = knn_classifier3.predict(X_test)
print('KNN 3')
print(classification_report(y_test, y_pred_knn1))
```

Listing 3: Python code to construct and train a KNN classifier for $k = 3$ neighbor (KNN-3)

The consfusion matrix as well as the total accuracy of the classifier on the test set are shown in figure 9. The accuracy of the KNN-3 classifier is 35% again.

KNN 3

	precision	recall	f1-score	support
0	0.48	0.03	0.06	437
1	0.62	0.30	0.40	473
2	0.41	0.36	0.38	549
3	0.34	0.72	0.46	523
4	0.24	0.49	0.33	510
5	0.68	0.16	0.25	501
accuracy			0.35	2993
macro avg	0.46	0.34	0.31	2993
weighted avg	0.46	0.35	0.32	2993

Figure 9: Model evaluation metrics for KNN-3 model

3.2 Baseline solution 2: The Nearest Centroid classifier

Besides the KNN approach, we can also use another traditional classification technique, the Nearest Centroid one. According to this technique, each class is represented by a representative (centroid) and inference is made based on the closest distance between the test example and some class centroid. The prediction of the model is the class, whose centroid is closest to the vector representation of the test example. Code listing **??** shows the way one can initialize and train a Nearest Centroid classifier, using, again, the `scikit-learn` Python library.

```
from sklearn.neighbors import NearestCentroid
from joblib import dump
nc_classifier = NearestCentroid()
nc_classifier.fit(X_train, y_train)
dump(nc_classifier, '/gdrive/My Drive/Kaggle/classifiers/nc.joblib')
y_pred_nc = nc_classifier.predict(X_test)
print('Nearest Centroid')
print(classification_report(y_test, y_pred_nc))
```

Listing 4: Python code to construct and train a Nearest Centroid (NC) classifier

Before presenting the model evaluation, we consider it quite interesting to enlighten a not so profound aspect of the Nearest Centroid classifier, the centroids themselves. Figures 10 - 15 depict the class centroids, as they are formed on the total train set. It is quite interesting, yet expected, to see that the class centroids seem to be a blurred image of the class they represent. For instance, the forest class centroid (figure 11) has dominant green colors, while the mountain class centroid forms an abstract line between the mountains and the sky on bottom and top of figure 13 respectively.

That exact observation (the intuitively-interpreted centroid images) could explain the significantly higher (yet still absolutely low) accuracy of the Nearest Centroid classifier, as compared to the baseline we found with the KNN classifiers. Figure 16 shows several evaluation metrics for the Nearest Centroid classifier, which reaches a total accuracy of 44%.

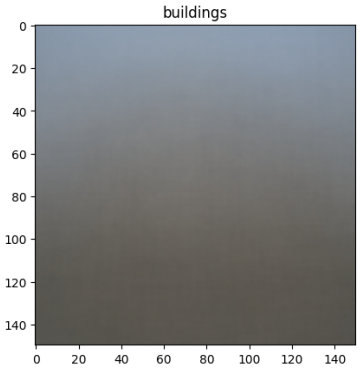


Figure 10: Buildings centroid

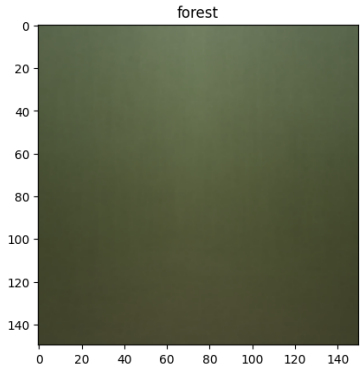


Figure 11: Forest centroid

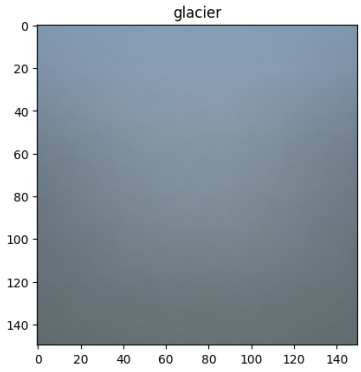


Figure 12: Glacier centroid

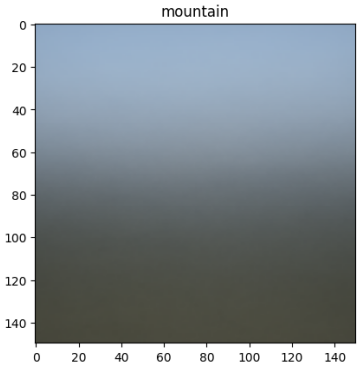


Figure 13: Mountain centroid

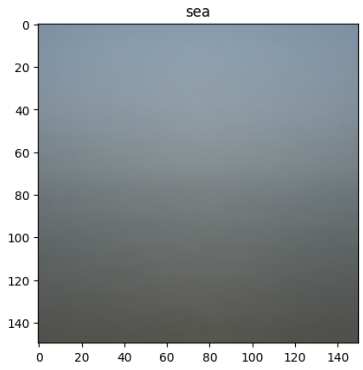


Figure 14: Sea centroid

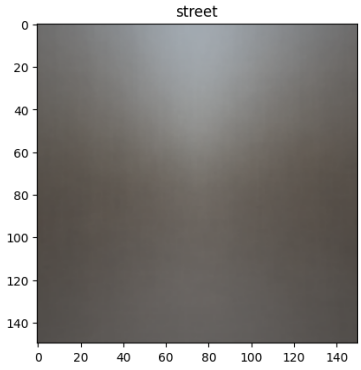


Figure 15: Street centroid

Nearest Centroid

	precision	recall	f1-score	support
0	0.33	0.22	0.27	437
1	0.53	0.75	0.62	473
2	0.44	0.57	0.50	549
3	0.43	0.60	0.50	523
4	0.34	0.12	0.18	510
5	0.45	0.37	0.41	501
accuracy			0.44	2993
macro avg	0.42	0.44	0.41	2993
weighted avg	0.42	0.44	0.42	2993

Figure 16: Model evaluation metrics for Nearest Centroid model

To that end, we have thoroughly covered the selected classification problem (dataset), as well as the traditional, non Neural Network approaches we can resort to, in order to solve it. At this point, we would like to clearly state that the only reason we implemented the KNN and NC classifiers is to use them as a reference point throughout the current report. In other words, we use them (their performance) as a baseline hook to assess the models we are going to train later in this assignment series.

4 Multilayer Perceptron

In this section we present a *Multilayer Perceptron* approach to solve the classification problem. In the lines that follow we are going to get a deeper insight into the workflow we adopted, analyze the outcomes of our work and compare them with the baseline classification models.

4.1 Implementation overview

In this approach we try to specify an appropriate Multilayer Perceptron architecture that solves the demonstrated classification problem. We use the Categorical Cross Entropy loss function to train multiple models and evaluate their performance on test set, based on the accuracy metric. In particular, we opt for executing a grid search in the (infinite) architecture search space. In order to explain our workflow, we need to introduce the following sets:

- $\mathcal{H}$, the set that contains the different numbers (multitudes) of hidden layers we want to examine in our architecture. Withing the context of the current report we define $\mathcal{H} = \{3, 7, 10\}$.
- $\mathcal{O}$, the set that contains the different optimizers we investigate for our MLPs. Withing the context of the current report we define $\mathcal{O} = \{\text{"sgd"}, \text{"adam"}\}$, where "sgd" stands for Stochastic Gradient Descent.
- $\mathcal{B}$, the set containing the different batch sizes we examine. We define $\mathcal{B} = \{32, 64, 128, 256\}$.
- $\mathcal{F}$, the set of the different activation functions we want to investigate. We define $\mathcal{S} = \{\text{"sigmoid"}, \text{"relu"}\}$.

Having decided on the candidate parameters of the Multilayer Perceptron, we can perform a **grid search** in the parameter space we created; thus $\mathcal{H} \times \mathcal{O} \times \mathcal{B} \times \mathcal{F}$ is the finite space of models we are searching in, where $\times$ stands for the cartesian product of the defined sets. The number of different models that can emerge from this grid search is $|\mathcal{H}| \cdot |\mathcal{O}| \cdot |\mathcal{B}| \cdot |\mathcal{F}| = 48$ different models. We train all these models for 10 epochs and keep track of their accuracy and loss per epoch. Figures 18 and 17 visualize the accuracy curves on train and test data of a trained model over 10 epochs.

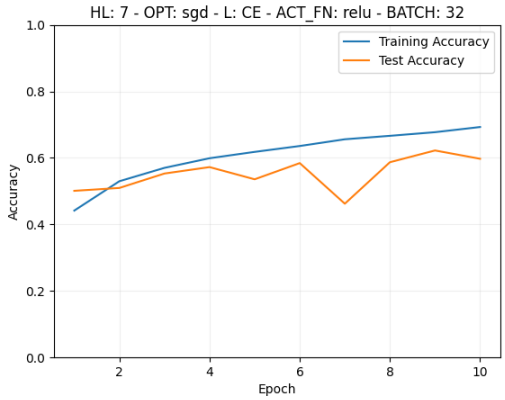


Figure 17: Training and test accuracy curves of a MLP with 7 hidden layers, compiled with Stochastic Gradient Descent and trained with RELU activation function and batch size of 32 images

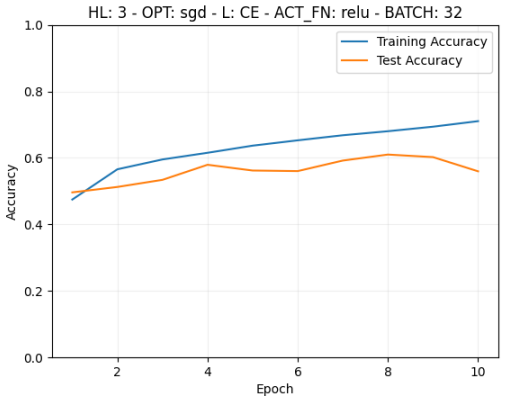


Figure 18: Training and test accuracy curves of a MLP with 3 hidden layers, compiled with Stochastic Gradient Descent and trained with RELU activation function and batch size of 32 images

In an overview, we train 48 different models, all having a different combination of the parameters we demonstrated. We use Cross Entropy as the loss function and we evaluate the models on their accuracy in the test set. The interested reader can examine the grid search code in listing 5.

```python
# GRID SEARCH
import pickle
import tensorflow as tf
import kerasm keras.layers

number_of_hidden_layers = [3, 7, 10]
hidden_layer_units = [3750, 1875, 935, 512, 256, 128, 64, 32, 16, 8]
optimizers = ['sgd', 'adam']
losses = [keras.losses.SparseCategoricalCrossentropy()]
activation_functions = ['sigmoid', 'relu']
batch_sizes = [32, 64, 128, 256]

for hidden_layers in number_of_hidden_layers:
  for optimizer in optimizers:
    for loss in losses:
      for batch_size in batch_sizes:
        for activation_function in activation_functions:

            model_name = f"Model_{str(hidden_layers)}_{str(optimizer)}_SCCE_{activation_function}_{batch_size}"
            file_name = f'/gdrive/My Drive/Kaggle/NN Models Reports/{model_name}.txt'

            print(f'Started Training model: {model_name}')
            model = tf.keras.models.Sequential(name=model_name)
            model.add(tf.keras.Input(shape=(7500,)))

            for hidden_layer in range(hidden_layers):
              model.add(tf.keras.layers.Dense(
                  units=hidden_layer_units[hidden_layer],
                  activation=activation_function,
                  name=f"hidden_layer_{hidden_layer}",
                  kernel_initializer='glorot_uniform',
                  bias_initializer='zeros'
              ))
            model.add(tf.keras.layers.Dense(units=6, activation='softmax', name="output"))

            model.compile(
              optimizer= optimizer,
              loss=loss,
              metrics=['accuracy']
            )
            print('Model successfully compiled. Starting training...')

            history = model.fit(
              X_train,
              y_train,
              batch_size=batch_size,
              epochs=10,
              verbose='auto',
              validation_data=(X_test, y_test)
            )

            print('Starting writing history to file...')
            writefile = open('/gdrive/My Drive/Kaggle/NN Models Reports/' + model_name + "_history.pkl", "wb")
            pickle.dump(history, writefile)
            writefile.close()
            print('Finished writing history to file ...')
```

Listing 5: Python code to a grid search

Based on this grid search, we can take our reasoning one step further by defining some criteria for selecting the "best" models and examine them more. Within the context of this report, we have chosen **maximum accuracy**, **mean accuracy** and **mean loss**, all measured on the test set. Based on these criteria, the "best"

three models in our investigation were:

- **Model 1**: 3 hidden layers, Stochastic Gradient Descent optimzer, RELU activation function and batch size 32 images
- **Model 2**: 7 hidden layers, Adam optimzer, RELU activation function and batch size 64 images
- **Model 3**: 3 hidden layers, Stochastic Gradient Descent optimzer, RELU activation function and batch size 32 images

The next step in our workflow was to re-train all these three models, for 40 epochs this time, measuring again their performance.

4.2 Model 1 analysis

The first model we examined in detail was the one with 3 hidden layers, Stochastic Gradient Descent optimzer, RELU activation function and batch size 32 images. The model is a fully connected MLP with the following architecture. The average time for each epoch is 152ms and the model was trained for 40 epochs.

```
Model: "FinalModel_3_sgd_SCCE_relu_32"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_layer_0 (Dense)      (None, 3750)              28128750

 hidden_layer_1 (Dense)      (None, 1875)              7033125

 hidden_layer_2 (Dense)      (None, 935)               1754060

 output (Dense)              (None, 6)                 5616

=================================================================
Total params: 36921551 (140.84 MB)
Trainable params: 36921551 (140.84 MB)
Non-trainable params: 0 (0.00 Byte)
```

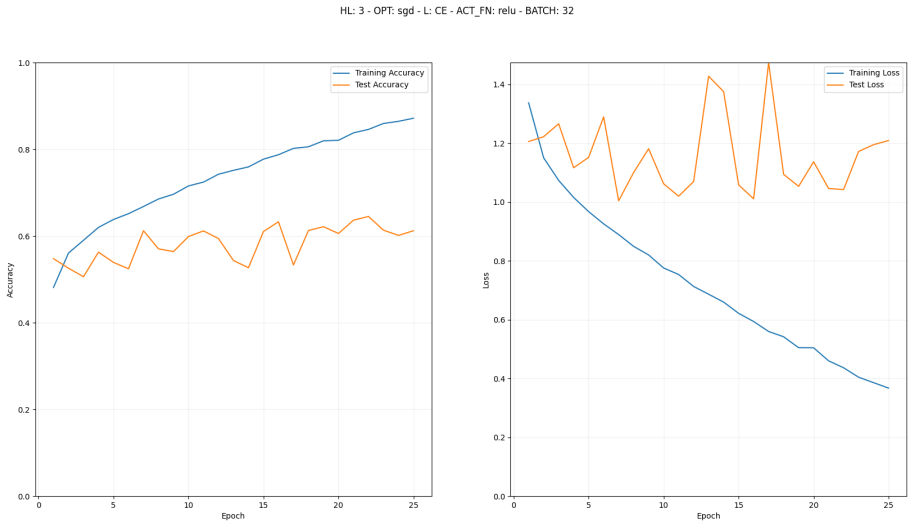


Figure 19: Accuracy and loss curves for Model 1

4.3 Model 2 analysis

The first model we examined in detail was the one with 7 hidden layers, Stochastic Gradient Descent optimzer, RELU activation function and batch size 32 images. The model is a fully connected MLP with the following architecture. The average time for each epoch is 158ms and the model was trained for 40 epochs.

```
Model: "FinalModel_7_sgd_SCCE_relu_32"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_layer_0 (Dense)      (None, 3750)              28128750

 hidden_layer_1 (Dense)      (None, 1875)              7033125

 hidden_layer_2 (Dense)      (None, 935)               1754060

 hidden_layer_3 (Dense)      (None, 512)               479232

 hidden_layer_4 (Dense)      (None, 256)               131328

 hidden_layer_5 (Dense)      (None, 128)               32896

 hidden_layer_6 (Dense)      (None, 64)                8256

 output (Dense)              (None, 6)                 390

=================================================================
Total params: 37568037 (143.31 MB)
Trainable params: 37568037 (143.31 MB)
Non-trainable params: 0 (0.00 Byte)
```

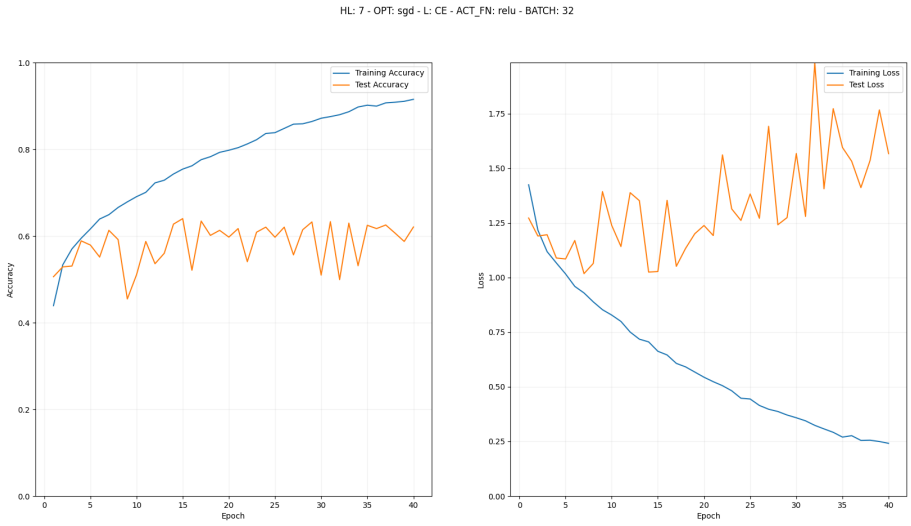


Figure 20: Accuracy and loss curves for Model 2

4.4 Model 3 analysis

The first model we examined in detail was the one with 7 hidden layers, Stochastic Gradient Descent optimzer, RELU activation function and batch size 32 images. The model is a fully connected MLP with the following architecture. The average time for each epoch is 198ms and the model was trained for 40 epochs.

```
Started Training model: FinalModel_7_adam_SCCE_relu_64

Model: "FinalModel_7_adam_SCCE_relu_64"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 hidden_layer_0 (Dense)      (None, 3750)              28128750

 hidden_layer_1 (Dense)      (None, 1875)              7033125

 hidden_layer_2 (Dense)      (None, 935)               1754060

 hidden_layer_3 (Dense)      (None, 512)               479232

 hidden_layer_4 (Dense)      (None, 256)               131328

 hidden_layer_5 (Dense)      (None, 128)               32896

 hidden_layer_6 (Dense)      (None, 64)                8256

 output (Dense)              (None, 6)                 390

=================================================================
Total params: 37568037 (143.31 MB)
Trainable params: 37568037 (143.31 MB)
Non-trainable params: 0 (0.00 Byte)
```

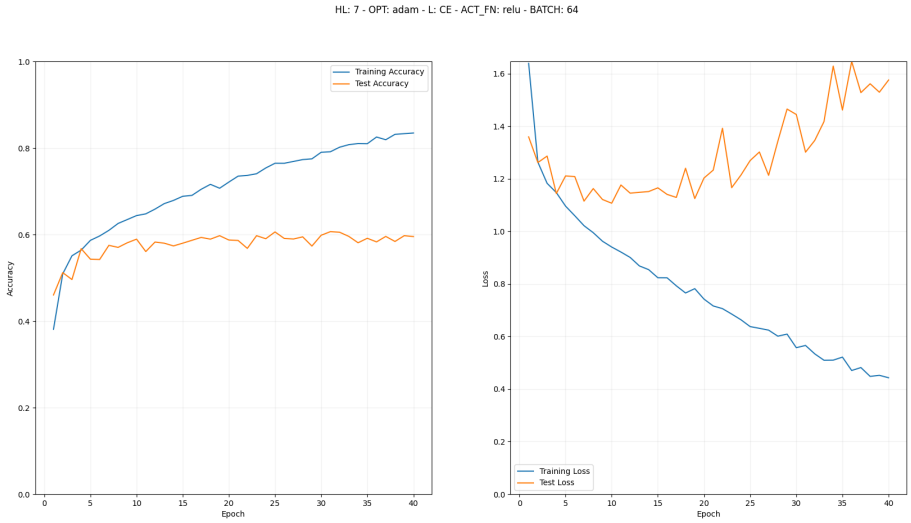


Figure 21: Accuracy and loss curves for Model 3