INF264
Project 2 Report

Kim Austgulen
Fillip Lampe

October 2024

Contents

Introduction

This project focuses on making a model to classify handwritten digits from (0-9) and (A-F) from gray scale images. We used methods such as Support Vector Machines (SVM), Convolutional Neural Networks (CNN), and Random Forest using libraries such as SKLearn, PyTorch, and Keras. This also focuses on how PCA affects these models, and finding methods to detect OOD samples in a dataset.

Division of Labor

We mostly shared the workload, discussing and contributing when we had time. We split up a few tasks, as Fillip took care of testing out the PyTorch library, whilst Kim took more care of testing out the Keras library and its functions. Even though we split this part up we discussed, and went through these parts together to learn from each other.

During the project we met up multiple times and discussed problems, implementations, and ways to structure this project.

1 Technical Report

1.1 Data Exploring

The provided datasets consists of gray scale images of size 20x20 pixels, with each image representing a gift marked with a hexadecimal digit or a noisy, missing image. Each of these images then have 400 unique features, one per pixel, which can in a range from 0 to 255. The corrupt images had some extra, but that we will come back to.

1.1.1 Clean Dataset

All of this data was split 0.7 for training, and 0.15 for val and test, on seed 42. Upon initial inspection of the dataset, we observed the following:

- **Class Distribution**: This dataset appears imbalanced, with some classes significantly underrepresented (see Figure 1). We anticipated that this might affect the model's ability to generalize.
- **Image Quality**: The images are relatively low-resolution, which could impact feature extraction. The pixel intensities (Figure 3) are most dominant at towards the the black and towards the white, which suggest that using a threshold-based function during the preprocessing might be beneficial.

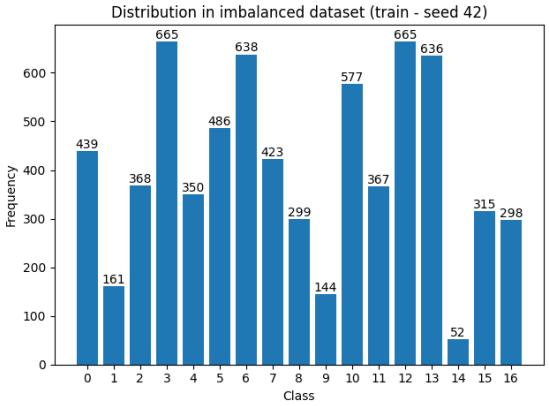

Figure 1: Clean dataset distribution imbalance

1.1.2 Corrupt Dataset

In addition to the clean dataset, we received a corrupt dataset containing 85 out-of-distribution (OOD) samples, which were unlabeled. These samples appears to be from the Fashion-MNIST dataset. Visualizing these samples (Figure 4), we that they differ significantly from the clean dataset, suggesting that both distance-based and feature-based methods could help detecting OOD samples.

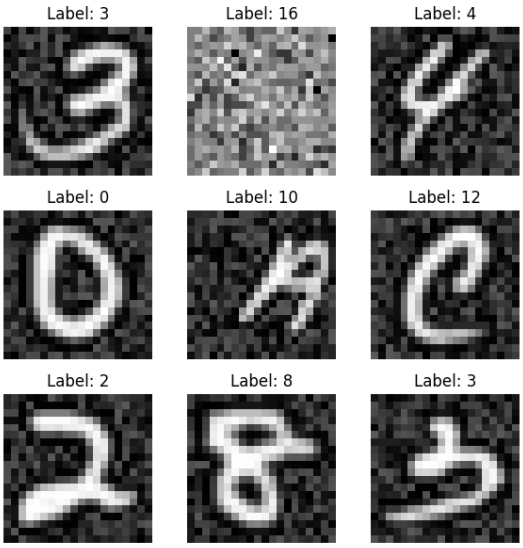

Figure 2: Samples from the training dataset

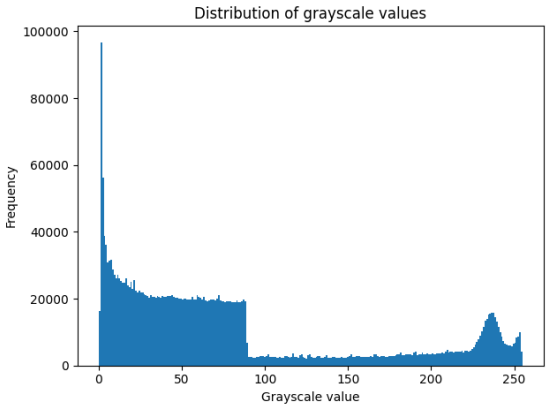

Figure 3: Gray scale Distribution over the training dataset

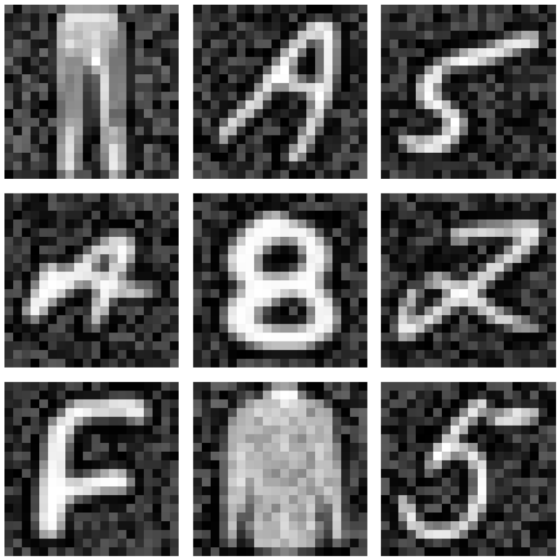

Figure 4: Samples from corrupt dataset

1.2 Preprocessing

Before training the models, we implemented several preprocessing steps to reduce redundancy, address data imbalance, improve model accuracy and runtime. Some of the methods we experimented with were:

- **Data Augmentation**: To address the class imbalance, we applied data augmentation techniques to generate additional samples. This was done using `ImageDataGenerator` from TensorFlow. This allowed us to generate additional samples for the underrepresented classes by applying transformations such as rotation, translation, shear, and zoom. This method helped improve the models ability to generalize across classes while keeping the same pattern (see Figure 5).

  SMOTE: We considered using SMOTE (Synthetic Minority Over-sampling Technique) to deal with the imbalance, but decided against it. [SMOTE is primarily made designed for continuous features,mhm], and generating synthetic image through interpolation could lead to unrealistic patterns, and in some cases not deal with the imbalance at all! In contrast (Figure 6) we can see that augmentation preserves real world patterns, and builds upon it. When looking at the differences (Figure 6) in the generated images, one can clearly see what is better, if not an attachment is added in the image folder to more clearly prove this. Something worth mentioning is that smote worked when the data was scaled down fist.

- **HOG Features**: We extracted the Histogram of Oriented Gradients (HOG) features to capture the structural information within each image. This improved the model accuracy, this could be due to the edge detection and gradients in the extracted data. While a threshold-based method yielded similar results (found in `preprocessing.py`), HOG features performed better for OOD detection, which is why we ended up using it. Worth mentioning, but the threshold floored values below 90 and ceiled values over 235 to 255, as this will make the important pixels (Figure 5) more intense.

- **Scaling**: We scaled the data to a [0,1] range, ensuring numerical stability, and improving performance of SMOTE when it was used. This was either done by using `MinMaxScaler` or just dividing on 255[1].

[1]Something which could considered data leakage, but in this case we know the structure of the data, as it is

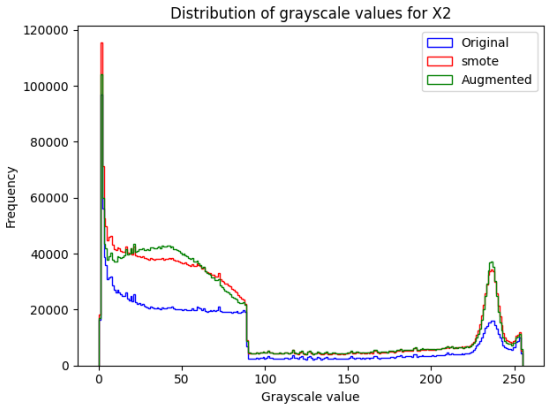

Figure 5: Showing how smote, and augmentation differ

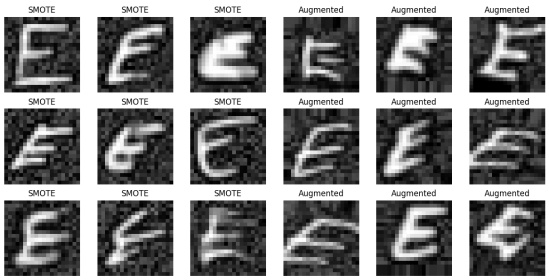

Figure 6: SMOTE vs. Augmentation

- **PCA**: Principal Component Analysis (PCA) was used to reduce the dimensionality of the data, particularly for task 2 and onward. We will get a little back to the effects of PCA on model performance later.

1.3 Model Selection - Candidates

We experimented with several classifiers to solve this task, focusing on SVM, CNN[2], and Random Forest. CNNs are known for their success in image classification, we chose this immediately. SVM, as they work well with high dimensional data, such as our dataset. Lastly Random Forest to address the minority in non-neural models, and because this is a model we known well from earlier.

1.3.1 Convolutional Neural Network

The first classifier we wanted to implement was CNN. We both wanted to learn more about these models and libaries, and it seemed fitting for this project, as CNNs are proven to be strong within image classification task. We ended up implemented two CNN architectures - AlexNet and LeNet. LeNet is simple and well-known, while AlexNet is more complex and capable of detecting more complex patterns. We ended up implementing both architectures in two different libraries (Tensorflow Keras and PyTorch).

The assignment requiers reproducability, and must be run on the CPU. To meet these criteria, we make sure the models run on a single CPU core. Due to these computational constrains we had

given by the task.

[2]Although CNNs are not one of the models we have used thus far in the course, we both wanted to learn more about how they work, and how to implement them.

to make some compromises for training and hyperparameter tuning. We had to limit the search space, as to make the runtime barable.

The architecture of our LeNet-esque model:[3]

```
Convelutional (input) Layer
Max Pooling Layer
Convelutional Layer
Max Pooling Layer
Flatten (turn original 2D input to 1D)
Dropout Layer (randomly deactivate some nodes)
Dense Layer (fully connected)
Dense (output) Layer
```

We included dropout to reduce overfitting. This will deactivate a percentile of the nodes each pass.
The architecture of our AlexNet-esque model:

```
Convelutional (input) Layer
Max Pooling Layer
Convelutional Layer
Max Pooling Layer
Convelutional Layer
Convelutional Layer
Convelutional Layer
Max Pooling Layer
Flatten (turn original 2D input to 1D)
Dense Layer (fully connected)
Dropout Layer (randomly deactivate some nodes)
Dense Layer
Dropout Layer
Dense (output) Layer
```

We use **categorical cross-entropy loss** (softmax) as our loss function, which penalizes the model for making low-probability guesses on a correct class, encouraging more confident guesses. For optimization we chose **Stochastic Gradient Decent** (SGD) and **Adam**, both of which gave good results without increasing runtime significantly .

The **hyperparameter tuning** varied somewhat between our implementations. The technicalities are discussed in their respective sections. We first used accuracy as out metric for training, as we augment the training dataset to account for class imbalance. However, the validation set and trainingset are *not* augmented this way. We therefor thought it best to implement balanced F1 score everywhere.

Despite the limitations on the model complexity and search space, both CNN's preformed very well. The specific result will be presented in their respective sections.

Our main struggle with these models were reproducability. Even with seeded random variables, and enforced deterministic behavior enabled, training results varied across different hardware due to floating-point differences. As per now, we have a semi-deterministic implementation of all models. By this we mean that training the model will be locally reproducible, but results may differ across computer hardware. One solution we implemented to circumvent this issue, was saving the locally trained optimal model to file. This allows the user to directly extract the fully trained model, without the need for re-training, giving the same results as we present here in our report. Of course, we also allow to re-train the models from scratch, to ensure or claim of (local) reproducability can be tested.

[3] to see how each layer scaled the tensor go to **nets.py**

1.3.1.1 PyTorch Implementation Overview

Over-processing with HOG and PCA: Initially, we used HOG and PCA in the preprocessing pipeline, but this led to worse results. We concluded that this had to be over-processing of the data in some way. Simply fixing the class imbalance and scaling worked better, giving improved results.

Grid Search: For hyperparameter tuning, we searched through different **learning rates** and **momentum** for the optimizer, to find the best configuration. An cross validation was used here to find out what set of parameters worked best, without using the validation data.

Model Training And Validating: The training loop specifies the number of epochs, architecture, optimizer, criterion, and scheduler. We used **SGD** as the optimizer for better runtime, and generalization, **CrossEntropyLoss** as the criterion to penalize incorrect predictions. The **ReduceLROnPlateau** scheduler was used to adapt the learning rate when validation loss plateaued. This would technically reduce the the need for a search on the learning rate, but due to the task we still kept it.

Reproducibility: As mentioned earlier this model is not (strictly) reproducible training wise. We tried to train with double floating points as well, but that was futile. Due to these constrictions the model will run slower. The results here will be reproducible as PyTorch has a way to save its models in a `.pth` file.

Performance: While running the AlexNet-esque architecture on Fillip's i7-13705H this took about 13min and 3s to train. And on validation reached an accuracy of 95.86% using only 10 epochs, and the found best learning rate of 0.01. As this generalizes pretty good, it will give better result over more epochs, but we choose not to, so the running time, while doing a grid search stays somewhat short. We have checked, and this accuracy can go up to roughly 97.8%. This will then also generalize well on to the test data.

For the LeNet architecture running 10 epochs with early stop, which is a way of generalizing in itself, and it only ran for 1 min and 8 seconds, and this also did a grid search. For the performance accuracy on the validation we got 95.93%, which is one of the lower scores for this model, but still competitive. This will most likely overfit if we run too many epochs on this model.

1.3.1.2 Tensorflow Keras

The second library we tried, was Tensorflow with the Keras API. We use Tensorflow and Keras version `2.14.0`, as newer versions s.a `2.17.0` can cause problem with loading/writing the `.keras` files where out trained models are stores. We here also experimented with implementing both the LeNet-esque and AlexNet-esque architecture such as we did in PyTorch.

Hyperparameter Tuning

To tune the hyperparameters, we used another library called `keras-tuner`. This allowed us to define the hyperparameters we wanted to tune, as well as their possible values. The tuner also saves the progress made during training. This means that we only needed to train once, and any subsequent calls on the tuner would return the already calculated results. There are multiple different searches one can use with keras-tuner. We started with `Hyperband`, which was really fast on GPU with good results. This runs almost like a tournament, starting out with a few epochs, only letting the top performers through to the next round, until only the winning combination remain. However, with the computational limitations, the run time of the complete search was over an hour. We therefore ended up using randomsearch, limited to a few epochs with early-stopping.

We decided here to use leave one out cross validation (LOOCV), as opposed to KFold. This again is purely in the interest of time. As our Keras model tune more hyperparameters, and uses the Adam optimizer, adding KFold was too costly time-wise.

The hyperparameters we tuned were number of units in dense layer, number of filters, learning rate and activation function. We chose to include two different params for number of filters. This meant that the convolutional layers could have a different amout of layers, which they did for the optimal combination. As can be confirmed by the implementation in `nets.py`, we started out by also treating kernel size and pool size as hyperparameters. We had to cut these down, and keep the more impactful parameters. The search-space is also reduced in the interest of time.

We ended up training 6 epochs for 6 iterations. This is by no means optimal, but done in the interest of time. It is also implemented in such a way that extending it is easy. When testing with GPU we did multiple complete searches, and the model can easily be changed to do so. Only one parameter needs to change. The best model is selected automatically, using weighted F1 score on the (fold) validation data.

After the best model is selected, we write it to a `.keras` file, named appropriately. This can be read directly with no need to run the training. This somewhat circumvents the issue of reproducibility, by adding the option to reload our results from training, but we will go more into these issues in its respective section.

Preformance
Even with the limited scope of epochs and parameter searching, the Keras LeNet-esque model needs about 15 minutes to train. This could be due to a different choice in optimizer, as we here use the Adam optimizer instead of the standard SDG. However, the results on the validation data was good, achieving an accuracy score of 96.54%. The addition of the dropout layer here helps with reducing overfitting.

The AlexNet-esque model ran slower. To fully train from scratch, it needs around 20-30 minutes. This is not optimal, but to get a representative result to present, any further limitation on the model seemed unfair. We include this more as a illustration of our thinking along the process, and our implementation. When the same limitations as for LeNet-esque were imposed, it reached an F1 score of 96.61% on the validation data.

Reproducability

Again, as for PyTorch, this was hard to make reproducible across machines. The implementation as per now, gives a semi-reproducible result, meaning training stays deterministic when run on the same hardware. As mentioned, we also chose to include the trained models as files. By default, the notebook will make use of this pre-trained version. Instructions on how to train from scratch are included in the notebook and `README.txt`.

1.3.2 Support Vector Machine

In addition to CNNs, we implemented an SVM classifier for comparison, as they are robust and well-established. Although SVMs need a little more feature manipulation, it performed well for this classification task. This model will probably not find complex patterns the same way some of the CNNs will do, but given the small image sizes, this will be solid choice.

For hyperparameter tuning, we used `RandomSearchCV` with a wide range of parameters, where we adjusted the different kernels, kernel coefficient (gamma), regularization (C), the independent term (coef0), degrees for the polynomial kernel. We also tested if the SVM should use shrinking or not, which is just an optimization setting.

We experimented with Sigmoid, Polynomial (degrees 2-4), and RBF (radial basis function) kernel to explore a range of potential representations of the data.

Overall, the SVM classifier performed well, especially when combined with **HOG**, and **PCA** (discussed later). It generally is more slower to train compared to the CNN in large hyperparameter search on big data, but it proved to do well, even with its simplicity.

1.3.3 Random Forest

4

Lastly we also decided to add the Random Forest, which will also serve as one of the non-neural ones. Random Forest also proved quite well and robust. This ensemble based method will use multiple different decision trees to improve classification performance by reducing overfitting and increasing generalization.

We used an `RandomSerachCV` for hyperparameter tuning, optimizing parameters as the number of trees (n_estimator, the max features for a given node (max_features), the maximal depth to prevent overfitting (max_depth), if we are going to bootstrap, and the minimum samples which can stay in a split. This is to try to find the most optimal random forest for this dataset.

Here we also used HOG features as mentioned earlier, to enhance the image data representation before sending it through `MinMaxScaler` for feature scaling. This achieved the worst scores respectively compared to the others model, but still a competitive score.

1.4 Model Selection

1.4.1 Process

As we have touched upon until in earlier sections, our performance measure used throughout was **weighted F1 score**. We chose this metric, as he weighted F1 score takes into account the number of true occurrences of a class when averaging the F1 scores, and should be a balance between treating the classes with the same importance, while addressing the class imbalance. This is important, since we regard each label as of the same importance, and treat wrong prediction as equally wrong. In other words, predicting the input is a 7, when the true value was 4, is not more or less wrong than predicting an A.

Thus far the training has only used data from the training set, so the original validation set is still unseen for each of them. To determine the final classifier, we score each of them using the above described weighted F1 score on this validation data. This will give us a decent indicator on how it will do on the final test data.

1.4.2 Final Classifier

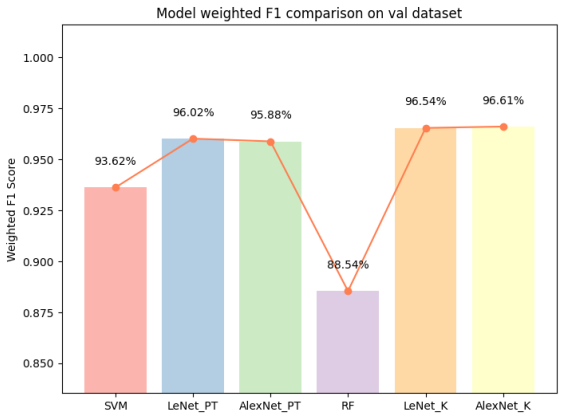

Figure 7: Trained Candidates F1 score on Original Validation Data

[4]The results might differ from different hardware. When run on same computer, it is deterministic and should meet the criteria set by the exercise. We found out right before the deadline, and did not dig too deep into why this was happening. (Checked with TA)

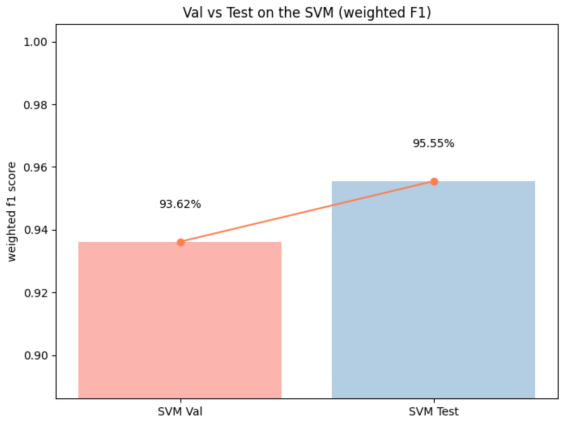

Figure 9: Test vs Val on weighted F1 score

Even tough we got good results from both Keras and PyTorch CNNs, they are not strictly reproducible. For this reason, we decided to use SVM, as it is reproducible on any machine, and therefore answers the criteria set by the exercise better.

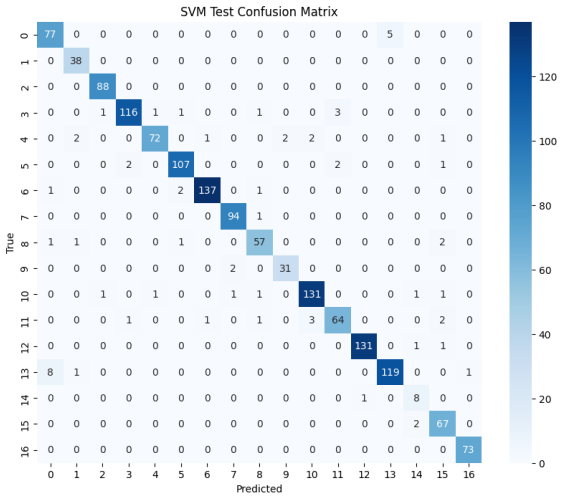

Figure 8: SVM test confusion matrix

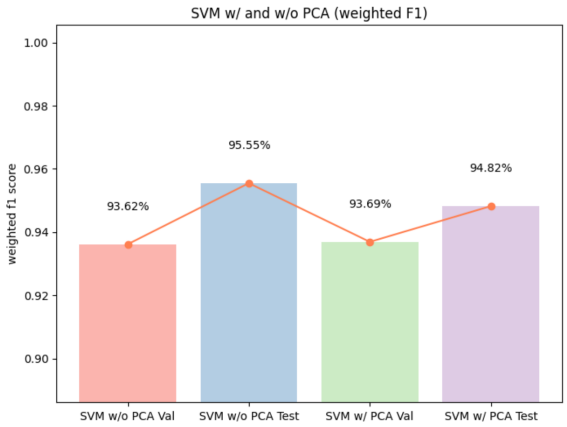

Figure 10: SVM with and without PCA with weighted F1

As part of this exercise, we compare the different results of our model both with and without using PCA. The first notable effect when implementing this, was a reduction on overall training time, being around 2 to 3 times faster on our dataset.

We also saw an increase in overall accuracy, both on validation and test data. The input given is high dimensional (400), and PCA can help by removing redundancies, and thus preventing over-fitting to the training data.

To find the best number of components, we include `n_components` as a hyperparameter in our search. To be fair, we need to tune the SVM as we change the PCA, and not just extract the previously found best parameters.

1.5 OOD images

To find the **Out of Distribution (OOD)** images, we tried a few different approaches. We began by adapting our original model and then explored more specialized methods, including **Gaussian Mixture Models (GMM)** and Mahalanobis distance-based detection. Both were implemented in the `ood_detection.py` file, and the final result will be found in the main notebook as they are a little big.

1.5.1 Gaussian Clustering

The first approach we went about was **Gaussian Mixture Model (GMM)** from Sklearn. This probabilistic model assumes the data is generated from a mixture of Gaussian distributions, and then will provide you with how likely a given sample belong to one of the learned distributions. This way we can detect samples not belonging to these learned distributions, and marking them as OOD.

We applied this method to our clean dataset first, then sent in the corrupt dataset, and knowing how many OOD there we chose an 5.5 percentile as a threshold for detecting the outliers.This allowed us to guess **52 OOD** samples, where 7 guesses was wrong and 45 was correct.

We tried to experiment with how using the SVMs predictive abilities to use a clustering on top of that. This didn't work that well, so we chose not to continue this.

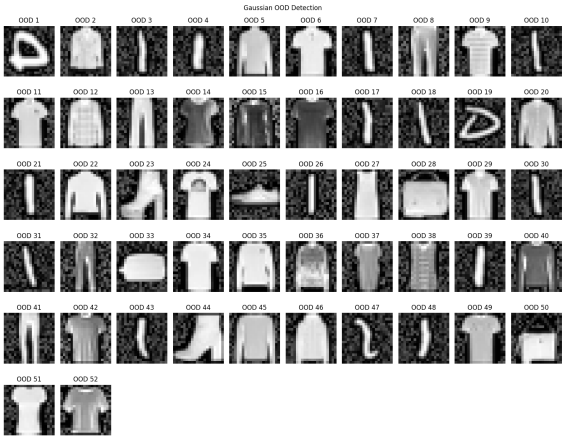

Figure 11: Gaussian OOD

1.5.2 Mahalanobis Distance

Our second approach, and the most effective, used the **Mahalanobis distance** to detect OOD samples. This Mahalanobis distance measures how far a sample is from the mean of the distribution, making it effective for detecting outliers and OOD samples.

For each of the classes in the training data, we calculated the mean and covariance matrix of the processed training data (after applying HOG, scaling, PCA). Then we computed the inverse covariance matrix to get the mahalanobis distance between each sample and the class means. We then used the minimum distance across the classes to label its mahalanobis distance.

We used a percentile-based threshold (93rd percentile) to label the data as outliers and OOD samples. This allowed us to guess **66 OOD** samples, where 16 was wrong and 50 was correct when looking through it.

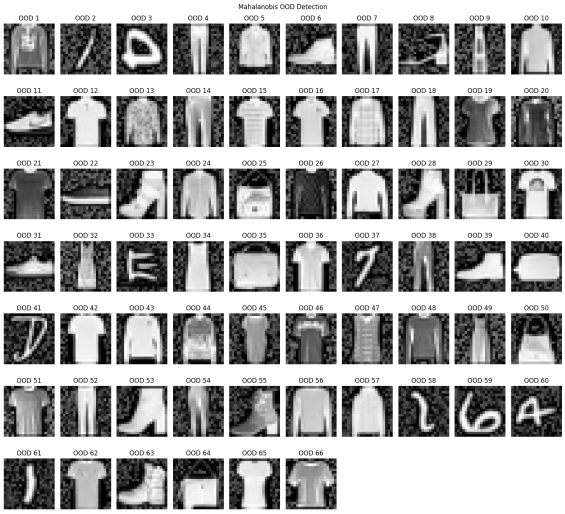

Figure 12: Mahalanobis OOD

Conclusion

Learning Outcome

This project was definitely more challenging when compared to the first one, but this also means that we learned a lot more as well. Firstly, we got to experiment with CNNs using two different libraries. We learned a lot about different ways to preprocess data, and what methods to using and trying techniques as data augmentation, SMOTE, HOG, scaling, threshold cutoff, and PCA.

We also learned a lot about the preprocessing affects the SVM, and how to improve non-neural models via feature extraction using techniques mentioned above.

Finally, exploring the different ways of doing OOD detection, we learned lots about using GMM and Mahalanobis distance for tasks such as this.

Improvements Given More Resources

As we have mentioned several times throughout our report, computing time was the most limited resource. If we could let our models train arbitrarily long, a deeper search would definitely be implemented.

After some consideration, we found out that we are over-complicating the model selection process somewhat. If we were to redo this project, we would split the data into 80% training, 20% testing. Then, we could use cross validation on the training data to assess model strength, and pick our final model among the candidates based on this score. Although our implementation as per now is not *wrong*, it might be more difficult to read than it needs to be.

Self Assessment

The project overall went quite well. We are happy with the results of our models, and the way in which they are presented. Since we both wanted to learn more about neural networks, we ended up spending a lot of time on these models, trying to optimize runtime and make them reproducible. Although this was great for learning, we could have prioritized differently to make it easier on ourselves. We would maybe approach this a little different in the future, but this was good learning.

2 Summary

In this project, we aimed to classify handwritten digits from (0-9) and letters (A-F), from gray scale images. We ended up using methods as CNNs, SVMs, and Random Forests to solve this task.

We preprocessed the given dataset, augmenting it to account for class imbalance, and also using others methods such as HOG, and PCA.

Our results showed that CNNs, particularly LeNet and AlexNet preformed very good on this task. SVM also gave back really good results when combined with feature extraction, proving to be a strong contender. We ended up choosing SVM as it is by all counts reproducible, and preformed very well as seen in the final test.

When comparing the how the SVM preformed with and without the PCA we could see that in our case it preformed better, and gave a lot better runtime.

The last task we focused on OOD detection, where we used Gaussian Mixture Models and Mahalanobis distance, which successfully identified outliers in the data.

Overall, we think we answered the task, and showed the effectiveness of different approaches, highlighting trade-offs between complexity, accuracy, and computational constrains.