Introduction to Machine Learning - Report 2

Alexandre Raybaut , Yassine El Graoui , Radu Tipurita

CS 233, EPFL

1. Introduction

This is a report stating the work we have done for the course project milestone 2. The results for the best model (CNN in our case) were uploaded to Ai Crowd under the team name OverfitAnalytics (4th place). We have implemented three main neural networks: MLP, CNN and ViT. MLP only uses the basic functionalties of a neural networks, meaning that it only has activation functions and linear functions (with a variable number of hidden layers). CNN and ViT both sophisticated approaches to learn the correct feature expansions before projecting it to the 10 dimensional space (with an MLP or linear layer). In order to run the models, the program uses the following additional arguments:

1. Parameter `--data` to load the path to the dataset.
2. Parameter `--nn_type` to choose the network architecture. The options are: 'mlp', 'cnn', 'transformer'.
3. Parameter `--nn_batch_size` the batch size for the NN training.
4. Parameter `--use_pca` chooses whether the model reduces the number of features or not.
5. Parameter `--pca_d` specifies the number of principal components.
6. Parameter `--test` chooses if the model will train on the entire training data or use a validation set method.
7. Parameter `--k_fold` specifies the number of folds to be exectued on the data.
8. Parameter `--load_model` allows the user to load an already trained model from the disk.

Below there is a sample command one could use to run an MLP model:

```
python main.py --data dataset  --nn_type mlp --lr 1e-3 --
    max_iters 50 --nn_batch_size 128 --use_pca
```

As a sidenote, the training is passed to CUDA if the hardware permits to do so.

2. Data Engineering

2.1. Data Pre-Processing

We begin by normalising the data such that each pixel has a value between 0 and 1 (indeed, originally, a pixel can have a value in the range [0, 255]). Then, if the user chose to apply PCA, we reduce the dimentionality of our inputs in the traning data and the testing data. PCA can only be done for MLP as it does not make sense to pass it to the contents of the image itself. Other approaches of data prepossessing that we have tested were: histogram equalization, sharpening or blurring the images by applying a specific convolution kernel, deskewing, centering. None of those proved to substantially contribute to the accuracy and f1 scores of our models so we have decided to not apply them on the final version of our project, although you can find the numpy methods in main.py.

2.2. Data Augmentation

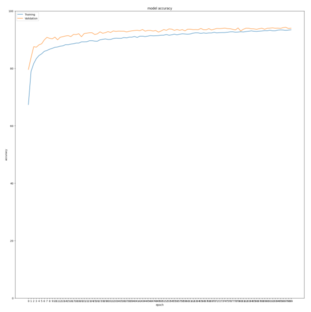
Data augmentation is a critical technique in the toolkit of anyone trying to train neural network. This is why we decided to test two different approaches:

- **Augmenting the number of data samples**

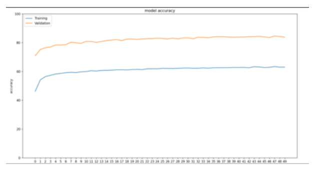
We have first decided to test the approach of increasing the number of samples in our dataset by applying random horizontal flips and random cropping on the images in numpy format. This technique did not substantially improve our accuracies (only by a fraction), hence we have postulated that the problem with this technique is that the model overfits on the data, whether we increase the number of data samples or not.

- **Inplace augmentation using pytorch**

Instead of artificially increasing the number of data samples, we apply in place transformations by doing data augmentation at each epoch. One issue was that the pytorch Tensor Dataset and Transform classes are limited. Tensor Dataset does not allow transforms and some transforms like Random Gaussian Blur apply the transformation to each image with probability 1. We have thus override the custom dataset and transform pytorch classes, allowing us to apply random transformations to data samples before passing them to the model that is being trained. This is also where we doing appropriate resizing for the MLP and PCA (because MLP requires vectors and not images as input). This allows us to simulate having a very large dataset by picking samples from the same distribution, without augmenting the dataset on disk and more



(a) Training vs Validation Accuracy of our best performing model over 100 epochs with Data Augmentation



(b) Training vs Validation Accuracy of our best performing MLP model over 50 epochs with Data Augmentation

Figure 1. Impact of Data Augmentation on different models

importantly without letting the model to overfit. Overfitting does not happen in this case, because the model sees a different set of samples each epoch. The set of random pytorch transformations that we apply consists of: random horizontal flips, random Gaussian blurs, random affine transforms (rotation + shifts) and random crops. Finally, we needed to distinguish between MLP and the two others architectures (CNN and Tranformer) because in CNN and transformer the images are passed as tensors of shape (28, 28), whereas for MLP, the vectors of shape (784, 1) are passed. So, we needed to do a reshape before applying the transforms, then go back to the original shape of the vector.

3. Models

3.1. MLP

Instead of tweaking the hyper parameters, we tried to perform a brute force neural architecture search. We tried to find the best combinations of activation functions possible, from the following set of functions : RELU, ELU, Leaky RELU, tanh and sigmoid, with both a variable number of hidden layers and a fixed number of hidden neurons per layer (which was 512 each time). Then we selected the three best models and trained them to find the best layer architecture: each time we could have 32, 64, 124, 256, 512, 784 or 1024 hidden neurons per layer. This experiment yielded us the three following best models :

1. activation functions : relu, tanh, elu, with the hidden neurons being 512 then 64
2. activation functions : sigmoid, relu, elu, with the hidden neurons being 512 then 1024
3. activation functions : leaky relu, relu, tanh, with the hidden neurons being 512 then 512

Also, we added dropout (with a probability of 0.25) to the model to avoid overfitting and to reduce the influence of "useless" neurons. Finally, we ran two experiments, one with pca and without data augmentation and 120 epochs and one with data augmentation and without pca and 120 epochs. This resulted in the model activation functions : relu, tanh, elu, with the hidden neurons being 512 then 64 being the best in both case with the following results :

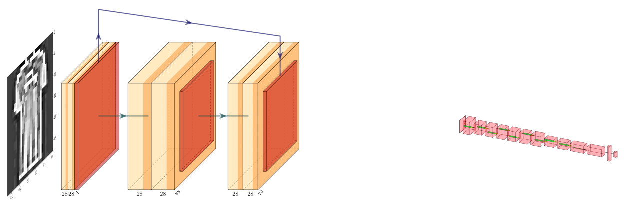
1. PCA / No Data Augmentation : testing accuracy : 86.7 and testing F1-score : 0.866
2. No PCA / Data Augmentation : testing accuracy : 86.5 and testing F1-score : 0.865

3.2. CNN

The clear winner of the three models in terms of performance on the validation set. We have tested three different approaches:

3.2.1. First steps

The dataset consists of grayscale (28, 28) images with relatively simple contours. Thus there is no apparent need to add more layers to the cnn architecture to get better performance. However one can overcome the overfitting due to a deeper network by adding more intermediate dropout during training. This is exactly the approach we test in our code and it proved to increase performance (although only by a fractional amount).



(a) Inverted Residual Block Visualization **(b)** Mini MobileNet Visualization

Figure 2. Visualisation of the MobileNet CNN

3.2.2. Inverted Residual Blocks

While standard convolution applies the operation directly to the input features, it can be computationally expensive and lead to inefficient resource utilisation. Increasing the feature count of the input makes the filters deeper and increasing the feature count of the ouput increases the number of filters. Inverted Residual Blocks overcomes the issue by applying depthwise and pointwise convolution. Our blocks consist of an expansion layer, that uses a (1, 1) convolution to expand the input channels and applying a non-linearity of choice. The non-linearity we use mostly is the SiLU activation function, $swish(x) = x \cdot \sigma(x) = \frac{x}{1+e^{-x}}$, although the first layer does classical ReLU. The next layer is depthwise convolution, which performs spatial convolution independently over each expanded channels (c.f groups parameter in Conv2d) separately, allowing massive parallelism. The last layer is a projection layer consisting of pointwise convolution which reduces the number of expanded channels back to the desired output dimension. Pointwise convolution combines the outputs of the expantion layer by applying a (1, 1) convolution. The last detail is adding a residual connection which allows for better gradient flow during back propagation, not over complicating the neural net architecture. Unlike traditional residual blocks, instead of narrowing the input, the input is expanded using more channels at each block. A visualisation of this block is illustrated bellow.

3.2.3. Mini MobileNet

The architecture, similar to Google's MobileNet v2, consists of four main blocks with inverted residual layers, followed by dropout in between (p=0.35, p=0.55, p=0.6). We have decided to add aggressive dropout in between layers, as empirically it has proven to substantially reduce over fitting, To get more details, please refer to the code in `deep_networks.py`. We have decided not to rely on squeeze and excitation and/or inception modules, even though our experiments show that replacing the last blocks by inception modules does not worsen our results.

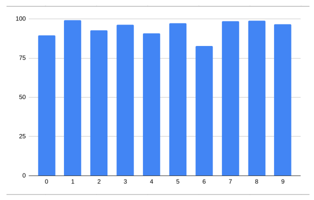
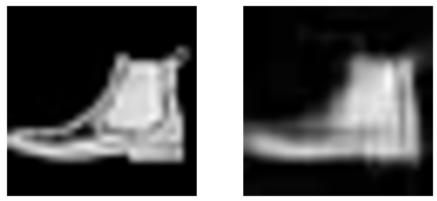


Figure 3. Histogram showing our per label accuracy for our best performing model, Mini Mobilenet, trained for 100 epochs, which achieves: a validation set accuracy of 96.750% and an F1-score of 0.966964 on our validation set, as well an average F1-score 0.941 and an ajusted accuracy score of 93.5% on AIcrowd (4th team in ratings)

3.3. Transformer

The transformer model was the most difficult to train. This is mainly due to the fact that in order to obtain good results one needs to fine tune a considerable amount of parameters. As a matter of fact, the main application for which the Transformer model was intended was in the NLP field and consequently, as we have seen during our project, for the Image Classification task this model was clearly outperformed by the others, notably the CNN, in both training computational efficiency and testing accuracy. Concerning our implementation, we proceeded the following way: each image of the batch was divided into patches and passed through a linear embedding. The optimal number of patches per image we found was 4. In order not to lose track of our patches during the later parts of the transformer we opted for a classical positional encoding based on sinusoids instead of treating them as learnable parameters. The next part of our architecture is the encoder. After running multiple experiments we concluded that our best hyper-parameters for maximizing our computational power are the following: 8 blocks, a hidden dimension of size 44 and an attention mechanism which is repeated 4 times. Thus, the number of heads is 4. After running the model for 18 epochs we obtain 82.35 as Testing accuracy and 0.81580611 for the F1 score. In the same spirit of maximizing efficiency, a potential improvement to our final model would have been to apply linear projection of both keys and values to a lower dimension before computing the similarity matrices. This can be done because these matrices are low-rank, or differently put, the information captured in higher and higher dimensions follows a rather logarithmic curve. Therefore, one could compress the information in such a way so as to reduce the computational overhead of the algorithm. As a side note, this is the main reasoning behind the Linformer model.



(a) Image before applying PCA **(b)** Image after applying PCA (conserving 89% of variance)

Figure 4. PCA conserves the shape

4. Discussion

4.1. Overcoming difficulties

When doing the project, our biggest problem was the time necessary to train our models. Indeed, models like CNN or transformer took a lot of time just to do one epoch. To improve that we knew we had to seek paralysing and vectoring our code. Firstly, we have used depthwise convolution + pointwise convolution in the CNN building blocks, which allows computational efficiency by leveraging massive parallelism. Secondly, we have fully vectorised the two main bottleneck operations performed in the vision transformer (our slowest model yet) : the positional embeddings and patchify methods. For the first method, we effectively leverage numpy's broadcasting capabilities to reduce the for loop to simple broadcasting operations. When it comes to the second method, the expensive python for loops we reduced to two simple tensor unfolds along the height and width dimensions of the images and a reshape:

```

    patches = images.unfold(2, h // n_patches, h // n_patches).
  unfold(3, w // n_patches, w // n_patches)
          patches = patches.reshape(n, n_patches ** 2, h * w *
    c // n_patches ** 2)


```

4.2. PCA

To implement PCA, we decided to modify the original code to make it faster and more numerically stable. Based on this post https://stats.stackexchange.com/questions/134282/relationship-between-svd-and-pca-how-to-use-svd-to-perform-pca, we can leverage the SVD decomposition of a matrix to get to the same results. The usual way of doing PCA by computing first the covariance matrix and applying eigendecomposition to it, which effectively squares the condition number: $Cond(X^TX) \approx Cond(X)^2$, leads to numerically unstable results. Instead, we leverage the spectral decomposition of the matrix X, namely $X = USV^T \Rightarrow C = \frac{\sum_{i=1}^{n} x_i x_i^T}{n-1} = \frac{X^TX}{n-1} = \frac{VSU^TUSV^T}{n-1} = \frac{VSSV^T}{n-1} = \frac{VS^2V^T}{n-1}$ Hence, since C is a symmetric square matrix, we get that its eigen-decomposition coincides with its spectral decomposition modulo the diagonal matrix S, the latter being trivially related to the eigenvalues. This way only uses transformations, which are isometries (in this case rotations given by the orthogonal matrices U and V). The matrix V contains the vectors spanning the vector space in which the column of X are living. This is because : $XV = USV^TV = US$
Since V is an orthogonal matrix: $V^TV = I$, also true for a subset of the vectors in V, since they form an orthonormal basis. By taking $W = V_d^T$, we find that indeed, $Y = X_d \cdot W^T = U_d \cdot S_d \cdot V_d^T \cdot V_d = U_d \cdot S_d$
In figure 4 there are two pictures illustrating the results we get with PCA.