1 Data overview

The available dataset consists of 30134 examples of handwritten letters and digits. The input images are 56x56 arrays, with binary pixels, and are stored as 3136-dimensional rows in the input matrix train_x. The labels are coded with the integers ranging from 0 to 35 and stored in the input array train_y. The train.pkl file contains the pickled tuple (train_x, train_y).

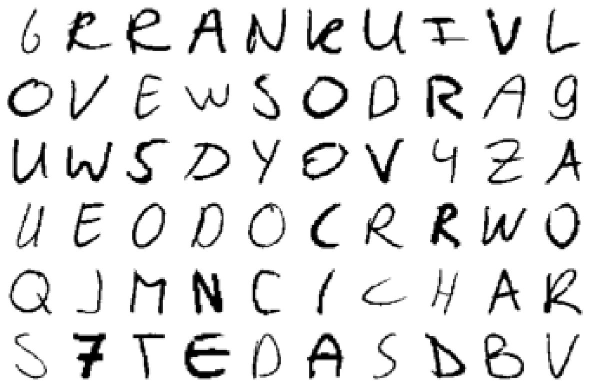


Fig. 1: A few sample images from the data.

The first step is to look at the distribution of each class in the dataset. As we can see in Figure 2, our data is heavily unbalanced. The class sizes range from about a hundred (although there is the special case of class 30, which contains only one image) to more than 3000.

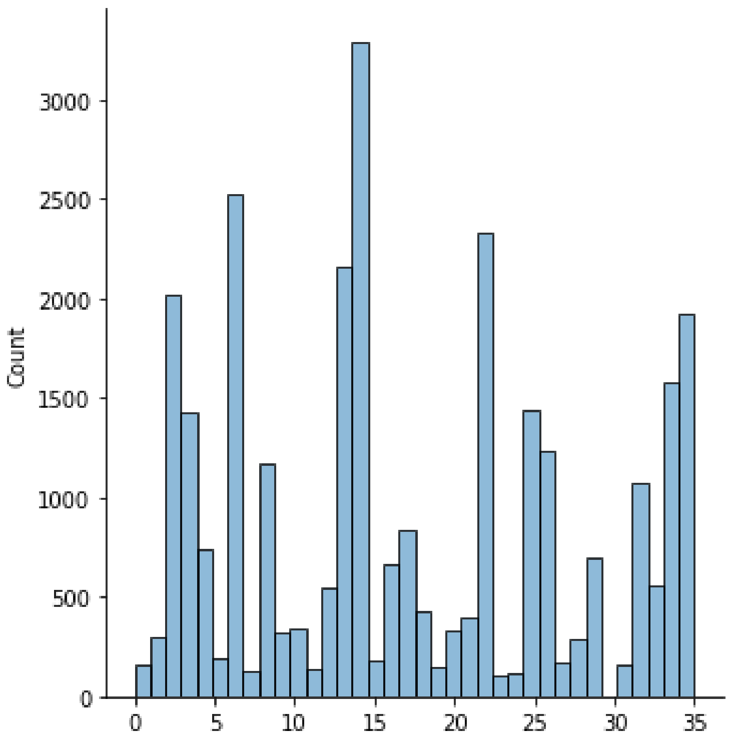


Fig. 2: Class distribution in the data.

Dealing with imbalanced dataset

One of the ways to deal with unbalanced data is oversampling minority classes – increasing the frequency that images from these classes are seen by the model during training. For this purpose, I used weighted random sampling (WRS), in which elements are weighted and the probability of selecting each element is determined by its relative weight (WeightedRandomSampler from the pytorch.utils.data module).

Splitting the dataset

Once the data has been properly reshaped, we can proceed to divide the dataset into a training and validation set. Since our dataset is not very large, I decided to allocate only 10% of the data to the validation set, which gives us 27120 training images and 3014 validation images. The data divided in this way was then packaged into DataLoaders (training and validation) to enable efficient iteration through the datasets in possession. A batch size of 32 was chosen for both DataLoaders, in addition, in the training DataLoader we used the aforementioned RandomWeightSampler to even out inequalities in the classes.

To illustrate how RandomWeightSampler works, let's look at how the classes would break down if we treated the entire training set as one big batch of data (Figure 3). The same is the way it works with smaller, 32-element batches; the sampler tries to pass a similar number of all classes to the model each time.

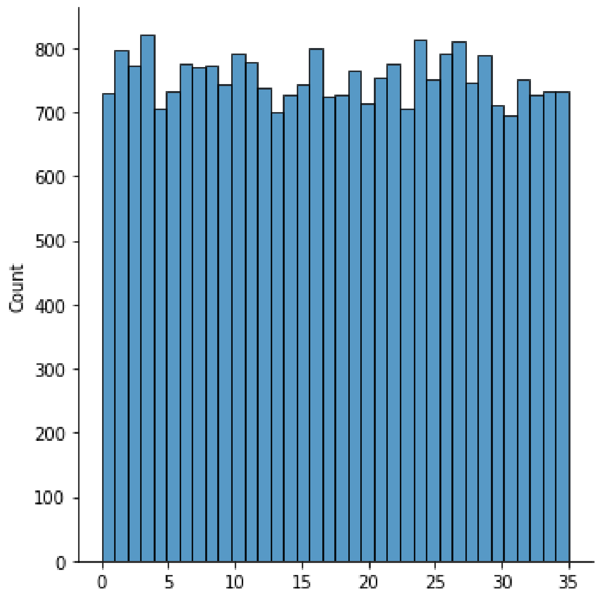


Fig. 3: Class distribution in the data after random weighed sampling.

Since our data consists of binary pixels, it doesn't make much sense to apply data normalization, so we can consider the data preprocessing stage complete.

2 CNNs

Although convolutional neural networks were developed many years ago, having gained their popularity in the early 2010s with the development of GPUs, they still form the basis of many top computer vision algorithms. Since sometimes the simplest solutions are the best ones, I decided to start my work with data modeling by building simple convolutional networks. The architecture I went for was 3 convolution layers with ReLU activation, two fully connected layers and softmax at the end. Between the convolution layers I used max pooling. In addition, when learning the network, the model uses three dropout layers to regularize the model. The probability of connection being zeroed out is 0.25 (I also tested a higher probability of 0.5 however this proved to be too much of a burden on our network and negatively affected the model results).

Selection of model hyperparameters and training parameters

Through a trial-and-error method, and observation of the model's behavior after each change, the following parameters were selected:

- Kernel size equal to 5 – 3x3 kernels were also tested, but the results were worse;
- Number of output channels respectively: 32, 32, 64;
- Size of the pooling region equal to 2;
- Number of epochs to train 50 – in this case, the selection of epochs was based on visual assessment of learning curves, while in the future it is better to use one of the early stopping methods;
- Cross entropy as a loss function – this function was used in a training process for all models described in this paper;
- SGD optimizer – also used in training for all models, to be further examined. Momentum = 0.9 and learning rate = 0.001.

Training the CNN model

As we can see in Figure 4, the loss function of the training set during training at 50 epochs steadily decreased, while for the validation set the function began to stagnate, so it makes no sense to continue training - it risks overfitting. Having a slightly larger validation set, we could look for the stopping point of training a little more precisely, in this case it is difficult to know exactly when the error of the validation set would start to increase (100 epochs were tested and the behavior of the validation set was still similar, the loss did not increase).

Let's take a look at how the accuracy and F1-score were presented during data training (Figure 5). The F1-score is calculated using a function from the sklearn library, and the type of averaging performed on the multiclass data is "weighted" which means the average for each label is weighed by support (the number of true instances for each label). Such definition takes label imbalance into account.

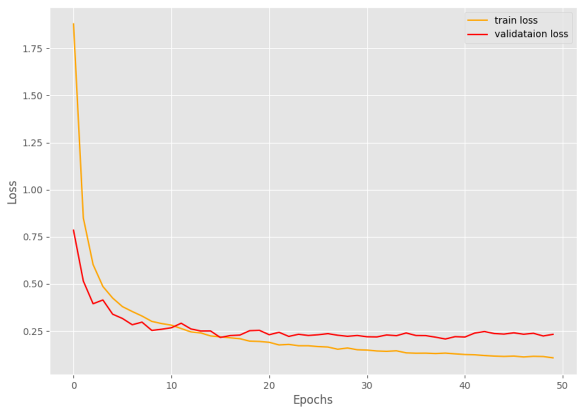
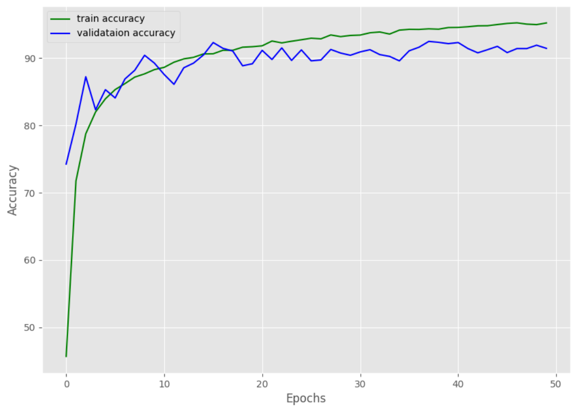
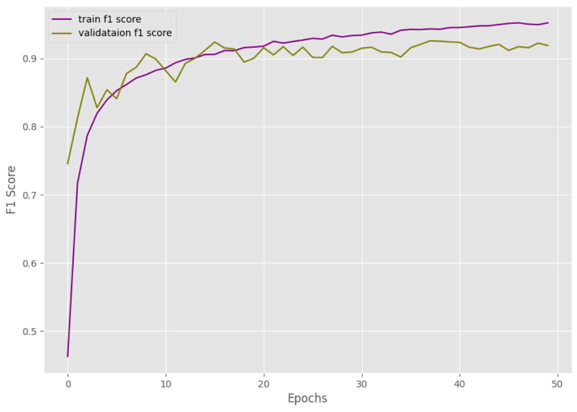


Fig. 4: Loss function results when training the CNN model.



(a) Accuracy



(b) F1-score

Fig. 5: Changes in accuracy and F1-score on training and validation sets during CNN model training.

After training, the accuracy of our CNN model on the training set is 95.428% while the F1-score is 95.2%. The loss function score on the training set was 0.105. This means that our categorizer has learned the relationships in the training data quite well, let's see how it performed on the validation set.

CNNs validation

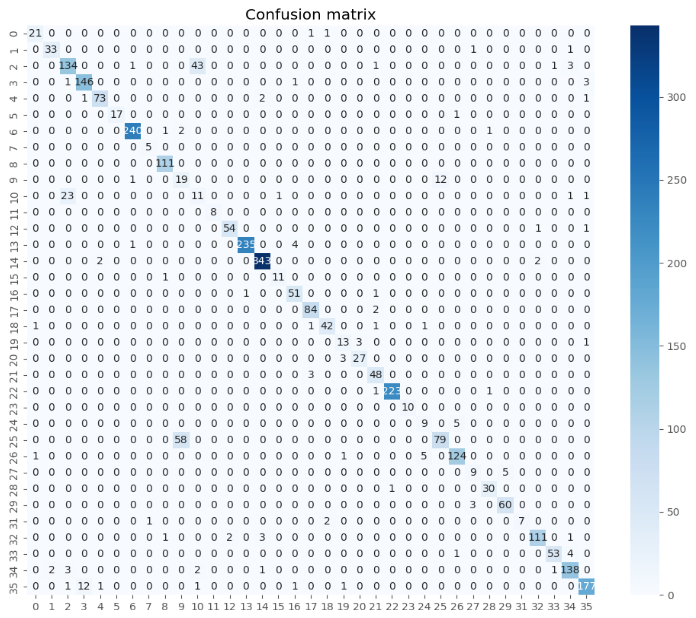


Fig. 6: Confusion matrix built for validation set (CNN model).

The trained CNN model achieves the following results on the validation set: loss function 0.217, accuracy 92% and F1-score 95.2%. A confusion matrix of the validation set was also produced (Figure 6), on which we can see that the vast majority of the data was classified correctly, while there are some classes with which the model still has problems, for example, 43 instances of class 2 were classified as 10 and as many as 58 instances of class 25 were marked as 9.

To summarize the work with the CNN model, the results obtained are good, but not great, given that there are advanced algorithms that achieve almost 100% efficiency in classifying handwriting. The unquestionable advantages of the classical CNN are its simplicity, ease of interpretation (leaving aside the operation of the channels), as well as full control over the architecture and the ability to test any combination of hyperparameters. The difficulty in training convolutional networks is the need for quite a large dataset; in our case, it is very possible that with more data, including balanced classes, we could get better results. It would also be worthwhile to work on incorporating assistive algorithms into the learning process, such as early stopping or a learning scheduler to better control the learning rate.

3 ResNet

As we can see, the results obtained with a simple convolutional network are good, but not sensational. So, let's use another approach, also using convolutions, while this time we will add layers of skip connections, thus residual neural networks (ResNet). ResNet networks having their beginnings in 2015 quickly beat competing models in many computer vision tasks, including image classification, they are still being developed today and are considered one of the state of the art algorithms.

In addition, given our relatively small dataset, we will use a technique called transfer learning, so we will import a model from PyTorch's pre-trained model library. In this way we will get a model with weights pre-trained for ImageNet competition on thousands of data. We will thus save the time and computational effort of building and training the ResNet model from scratch, we will only change the first and last layer of the model so that the dimensions of the network match our problem, and then fine-tune the model to work with our data.

In the torchvision.mdoels.resnet library, there are 5 pre-trained models with different network depths: 18, 34, 50, 101 and 152. For this task, I decided to test two ResNet variants: 18 and 50. I retrained both models using the same optimizer and loss function as before (again, there is some space here for more research and even better algorithm adjustment). I trained both models at 10 epochs, since the tendency of both models to over-fit could be clearly observed at higher numbers.

ResNet18 results

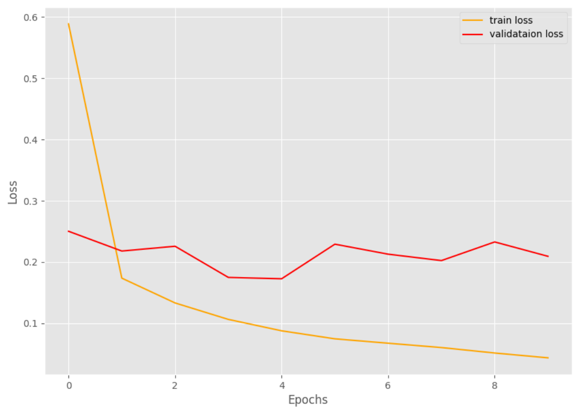
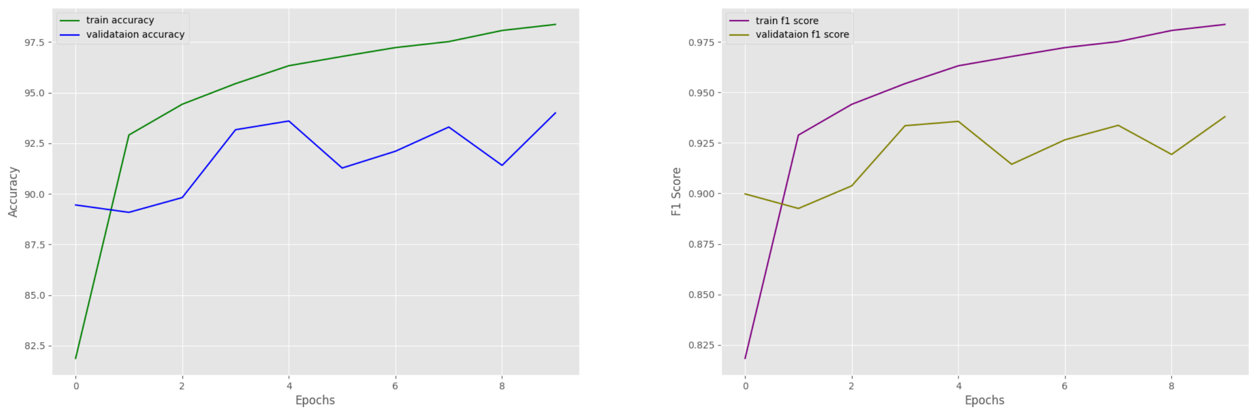


Fig. 7: Loss function results when training the ResNet18 model.



(a) Accuracy

(b) F1-score

Fig. 8: Changes in accuracy and F1-score on training and validation sets during ResNet18 model training.

After fine-tuning the ResNet18 model, we obtained the following results: loss function on the training set 0.043, accuracy 98.367% and F1-score 98.4%. On the validation set, the values were as follows: loss function 0.209, accuracy 93.995% and F1-score 98.4%.

ResNet50 results

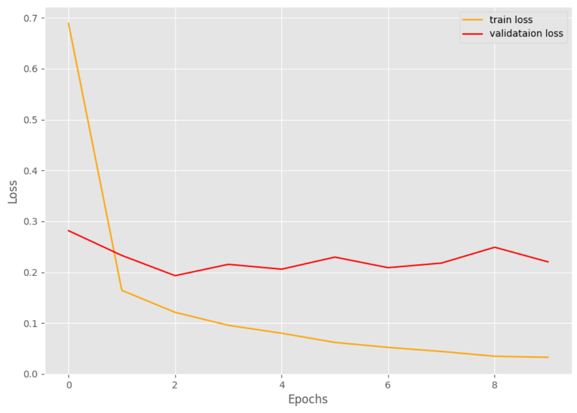
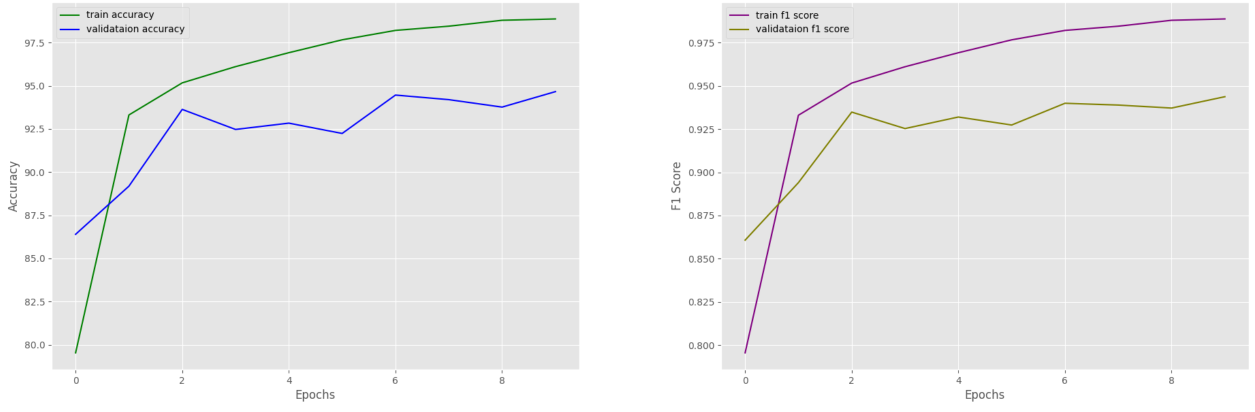


Fig. 9: Loss function results when training the ResNet50 model.



(a) Accuracy

(b) F1-score

Fig. 10: Changes in accuracy and F1-score on training and validation sets during ResNet50 model training.

After fine-tuning the ResNet50 model, we obtained the following results: loss function on the training set 0.033, accuracy 98.868% and F1-score 98.9%. On the validation set, the values were as follows: loss function 0.220, accuracy 94.658% and F1-score 98.9%.

This means that the deeper ResNet50 model performed better. Let's look at the confusion matrix obtained on the validation set using this model.

ResNet50 validation

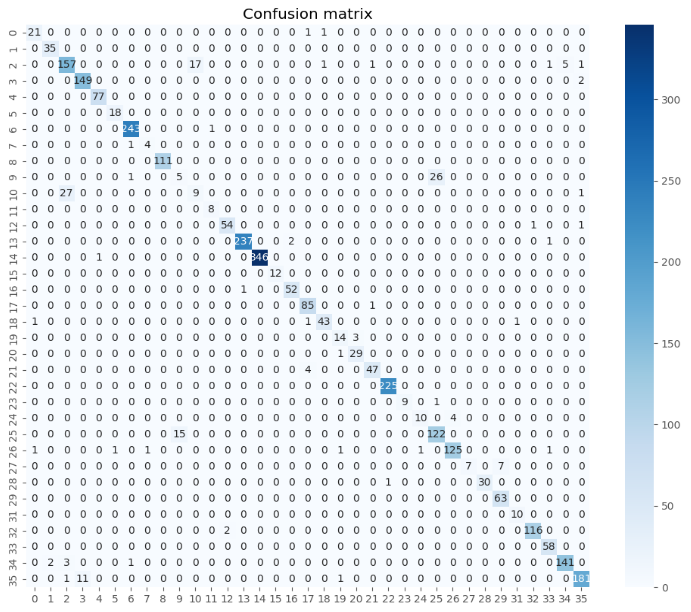


Fig. 11: Confusion matrix built for validation set (ResNet50 model).

Just as the metrics obtained indicate, we see significantly less mistakes in the confusion matrix (Figure 11) for ResNet50 than for a simple CNN. The most significant confusions are the classification of 27 instances of class 10 as 2 and also 26 instances of class 9 as 25.

Summing up the approach to the problem from the side of transfer learning and ResNet models, we can say that it was a good move. We got much better results, the time to train the models was much shorter, as we only matched the existing model weights to our problem, and thus it is a cheaper approach. In addition, there are many pre-trained models for image classification, there are dozens of them in the PyTorch library alone, so it is worth exploring the topic and thinking about choosing yet another model. Of the downsides of this solution, I would mention the lack of 100% control over the model, its architecture and training from scratch, as well as the ease of over-fitting, which we need to watch out for.

4 Vision Transformer

The previous two approaches are based on convolutions, which are a basic tool in working with images. The last approach presented in this report is vision transformer. Having its origins in 2017, transformer models are currently the basis of most top-notch NLP models, while in 2020 the creators of the article "An image is worth 16x16 words" adapted the transformer concept to the tasks of computer vision, and thus the vision transformer (ViT) was created. Just as in NLP the attention mechanism aims to capture the relationships between different words, in ViT the attention mechanism focuses on the relationships between different portions of an image.

While pre-trained vision transformer models are also available in the PyTorch vision model database, in order to get a better idea of the model implementation, as well as to observe the process of learning the model from scratch, I decided to use the open-sourced implementation available in the popular repository: https://github.com/lucidrains/vit-pytorch/tree/c0eb4c01509659e9d1b5c3a96e99946a758854cf.

Selection of ViT hyperparameters

The following hyperparameters were given to the ViT model. These parameters were chosen as a "reasonable starting point," while we would need to think more deeply about fitting the parameters to our problem and test different options.

- Patch size 7 – for 56x56 image it means we have $8 \cdot 8 = 64$ patches per image;
- Embedding dimension of 64 units;
- Depth of 6 transformer blocks;
- 8 transformers head (multihead attention mechanism);
- 128 units in the hidden layer of the output MLP head.

Training the ViT model

The results of training the model at 30 epochs will be presented, while, as we will note in the graphs presented, there is some potential for further training of the model, as the error on both sets kept decreasing and the accuracy increased. Due to time constraints, the potential for further training of the model will not be described in this report, but this is undoubtedly one possible direction for model development.

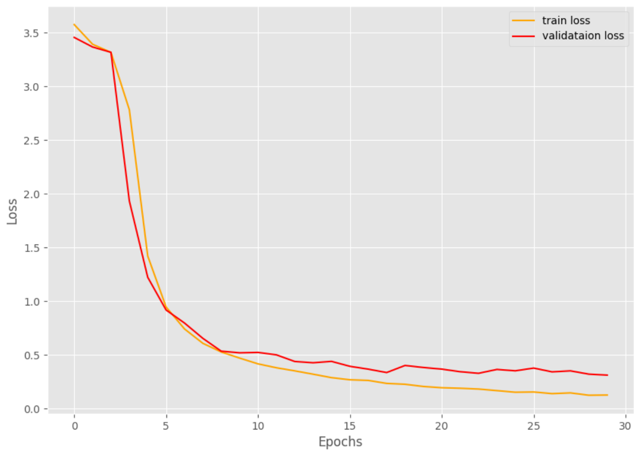
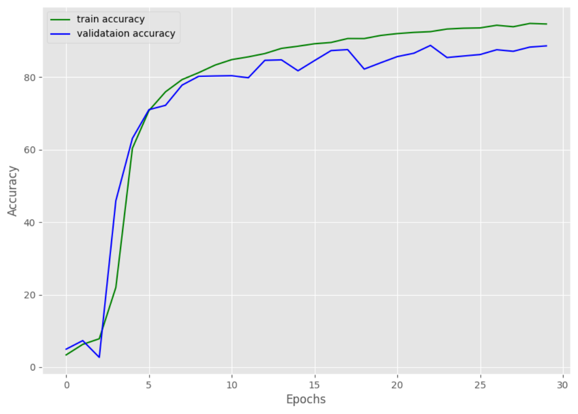
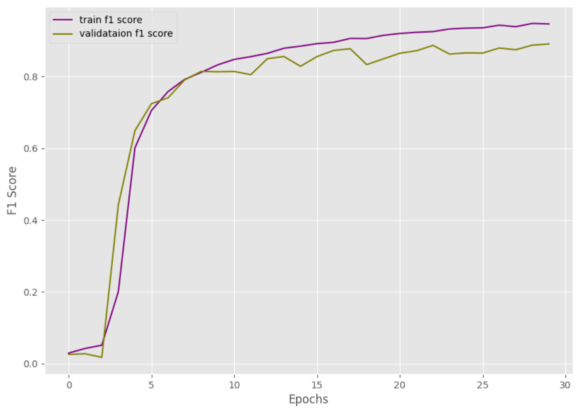


Fig. 12: Loss function results when training the ViT model.



(a) Accuracy



(b) F1-score

Fig. 13: Changes in accuracy and F1-score on training and validation sets during ViT model training.

After training, the accuracy of our ViT model on the training set is 94.664% while the F1-score is 94.6%. The loss function score on the training set was 0.127. As we can see in the figures 12 and 13, the loss function graph shows a gradual decrease in values for both sets and the accuracy and F1-score graphs show increases, indicating the potential for further learning and improvement of the ViT model.

ViT validation

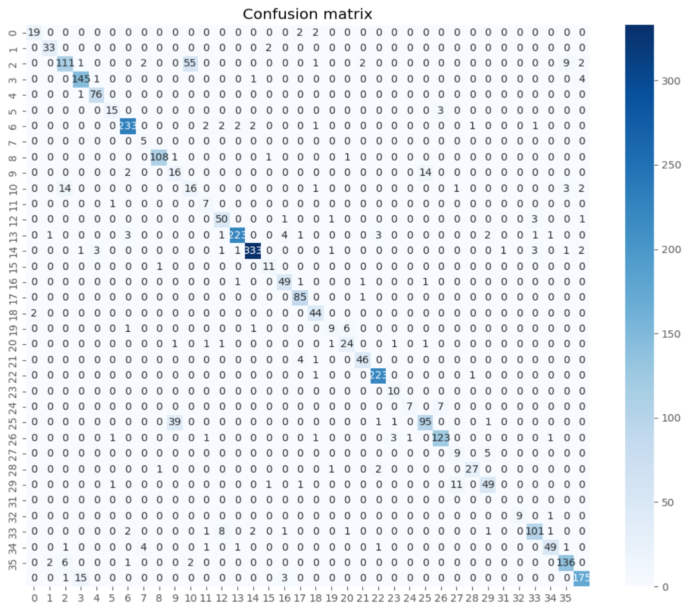


Fig. 14: Confusion matrix built for validation set (ViT model).

ViT model trained for 30 epochs obtained the following scores on the validation set: loss 0.312, accuracy 88.62% and F1-score 94.6%. Weaker performance on the validation set is also evident in the confusion matrix (Figure 14), where we can see some significant model errors, such as 55 instances of class 2 classified as class 10, or 39 instances of class 25 classified as 9.

5 Conclusions

Let's try to summarize the work done on the task, we tested three different models, one of which (ResNet50) was pre-trained. Table 1 shows the collected results of the various models on the validation set, as can be easily seen, the ResNet model was the best, the simple CNN model performed only slightly worse, and unfortunately the ViT model performed the worst, but this is most likely due to the insufficient length of the model training process.

Table 1: Comparison of model results on the validation set

Model	Cross Entropy loss	Accuracy	F1-score
CNN	0.217	92.0	95.2
ResNet50	0.220	94.658	98.9
ViT	0.312	88.62	94.6

It is also worth noting the most frequently confused classes, and these were class 2 and 10, and 25 and 9, respectively. These are images of handwritten ones and the letter I, as well as zero and the letter O (Figure 15).

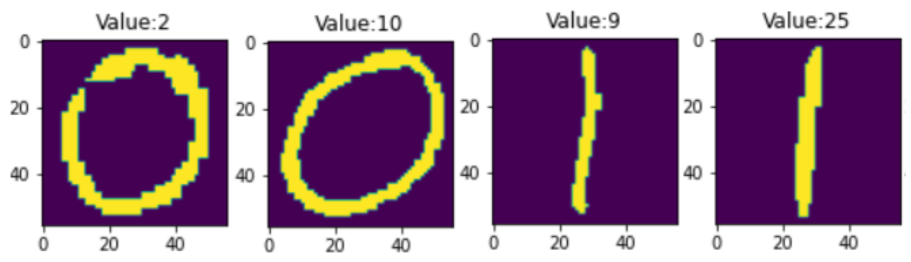


Fig. 15: Sample images from the most commonly confused classes.

Each of the models tested was wrong most often in classifying these images, in the following amounts:

- predicted 10 – actual 2 $\longrightarrow$ CNN: 43, ViT: 55
- predicted 2 – actual 10 $\longrightarrow$ ResNet: 27
- predicted 9 – actual 25 $\longrightarrow$ CNN: 58, ViT: 39
- predicted 25 – actual 9 $\longrightarrow$ ResNet: 26.

Interestingly, the CNN and ViT models were wrong in the same way, and the ResNet model was symmetrical to them. However, if we look at the examples of the aforementioned classes, we can conclude that we ourselves have difficulty distinguishing between them having at our disposal the most powerful "computer" and therefore our human brain - so it is no wonder that our models have fallen into a trap.

6 Recommendations of the next steps

In the last section, I will present my thoughts on further steps we could take to improve the classification of handwritten numbers and letters. I have organized them into the following points, of which the first three are, in my opinion, the most relevant and the others I would treat more or less equally.

1. Get a larger dataset, and more importantly a dataset with balanced classes – despite the weighted random sampling procedure, deficiencies in the diversity of individual classes still negatively affect models.

2. Check out the pre-trained vision transformer model – although in our case ViT did not achieve spectacular results (most likely because it was too briefly trained), the algorithm itself is considered a state-of-the-art-model with great potential, it is worth checking out. Transformer-based models need large amounts of data, so it is possible that fine-tuning a previously trained model would be a better solution than training the model from scratch.

3. Do a deeper study of ResNet models and their learning process – the results obtained are very good, but it's worth seeing if, with small changes in the choice of the underlying pre-trained model or changes in the way of fine-tuning (optimizer, loss function, learning rate etc.), our model wouldn't do even better.

4. Continue training the ViT model – the learning curves clearly show the potential for further learning. It would be interesting to see how the model will behave after training for an appropriate number of epochs determined, for example, by the early stopping method.

5. A step closely related to the previous recommendation, that is, to explore the topic of ViT hyperparameters – the parameters used are quite universal, while other combinations should be tested, the depth of transformer blocks, the number of attention heads, etc. should be changed.

6. Dig deeper into the selection of hyperparameters for the CNN model – use early stopping, learning scheduler, see how the results will change after using average pooling instead of max, and check more of the various parameters of the model itself such as kernel size, network depth, etc.

7. Test yet another type of model from among the available pre-trained models – given more time, it is worth delving into the subject literature and finding out which models perform best in similar tasks, maybe some of them are already pre-trained, which would make the task much easier.

8. Work more on matching the optimizer and loss function to specific models – currently all models were trained using the same loss function and optimizer with the same learning rate, we would probably get slightly different results using, for example, optimizer Adam instead of SGD.