Project-3 Report

Daniel Murray David Budaghyan Artiom Matvei

I. Introduction and Task description

In this project, we will process images, build neural networks and explore different architectural, hyper-parameter and optimizer choices for our image classification model. Our training dataset comprises of 56000 images containing 1 to 5 MNIST digits in the range [0, 9] each. The test dataset contains 14000 such images. Our task is to classify each image by outputting the digits present and to indicate the number of digits (n) in the image by adding the number ”10” $5 - n$ times at the end of our label. To do so, we will use a convolutional neural network, as this is the current go-to standard for classifying images. In order to simplify the learning task, we will pre-process the images by extracting the individual digits from each image. The neural network will thus be trained and tested on individual digits.

II. Data Pre-Processing

In order to extract the digits from the original images, we used the OpenCv library. The main elements used were the contours and boundingbox functions. We manually tweaked the different thresholds for those functions as well as custom thresholds in order to get an extraction algorithm that we deemed acceptable. Our final extraction algorithm collected the correct number of digits from all but three images in the training set. In fact, the algorithm only failed on a single digit (the digit 3) in the entire training dataset. The figure below demonstrates the three images containing this digit 3 as well as the extracted digits of our algorithm. Once extracted, each digit was adjusted to 10x10 pixels. This gave us 167874 individual 10x10 digits to train our model.

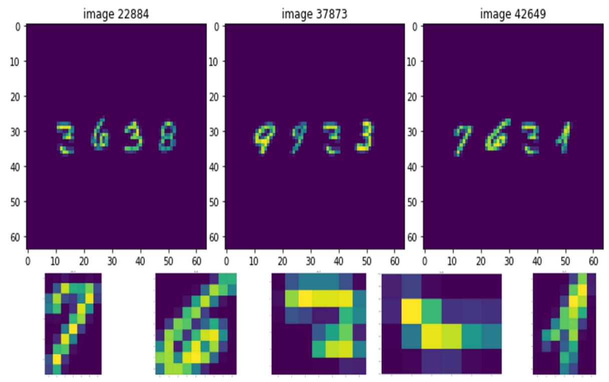


Fig. 1. The three images from which we extracted the wrong number of digits and the extracted digits from image 42649. The exact same digit 3, horizontally split in half, is the cause.

III. Architecture selection

Once we started working on implementing a learning algorithm, we quickly realised that due to the much longer training times for CNN’s, it will not be possible to do a comprehensive grid-search with cross-validation on different hyper-parameter tuples as we did in project 2. Moreover, the nature of neural networks is such that there are many choices one must make without rigorous testing (e.g. architecture). Because of these restrictions, we began our search for the best architecture by simple trial and error. Validation was always done with a randomly sampled subset, usually 10% or 2% of the training data.

We initially used the SGD optimizer, since this is what we’ve been working with this semester and it’s most familiar to us. We set generic values for it’s hyper-parameters (lr=0.01, mom=.95, bs=32) in order to concentrate on the model’s architecture, but as we discuss later in this paper, we soon realised that this batch size was too small and learning rate was to big.

All of our attempts followed the general structure of Alex-net. Namely, the idea is to take in a one-channel image, pass it through convolutional layers in order to decrease the image’s size but increase the number of channels, then pass each individual pixel through linear layers in order to get the final output. We naturally used softmax with negative log-likelihood for our loss function, implemented as ”cross-entropy” in pytorch.

A. Architecture 0

Our first architecture that passed the .99 mark on the test dataset is called architecture 0 (appendix fig. 4). Since we are working with smaller original images and a much smaller classification problem than Alex-net’s, we decided to use 3 convolutional layers that progressively increase the number of channels from 1 to 56, followed by a max-pooling that transforms the 10x10’s into 5x5’s, followed by 5 linear layers each decreasing the number of features. As suggested in class, we used the Relu activation function on all layers except the last one, as we don’t want to cut out information from the output. For regularization, we used a dropout percentage of 25% on all linear layers. We plot the accuracies and losses of this model trained with 2 different learning rates and batch sizes. The diamond indicates the epoch which achieved the lowest validation loss. We used a patience of 20 epochs.

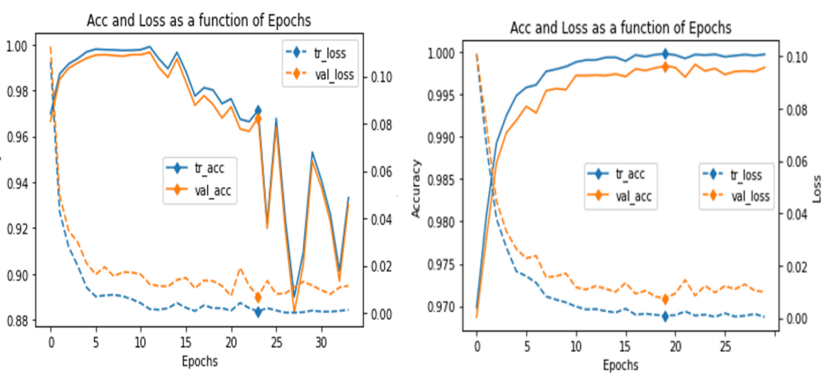


Fig. 2. Left:(lr=0.01, mom=0.97, bs=64)
Right:(lr=0.001, mom=0.97, bs=256)

As mentioned earlier, we noticed that a learning rate of 0.01 is actually too big for this model. Similarly, a batch size of 64 is too small. We came to these conclusions because architectural changes were not able to rectify the degradation that happens on the left plot, yet decreasing learning rate and increasing batch size did the job (right plot). The qualitative explanation for this is that lr=0.01 is too big for the minute adjustments required once the model learns the basics of the digits. Add the small batch size and the gradients you get once you're model is close to the optimum are wildly inaccurate. The model in the right plot at epoch 19 (diamond) achieved .995 on the test dataset.

B. *Other Architectures*

We figured we could improve this result by adding more convolutional and linear layers to our architecture. Architecture 1 (appendix fig. 5) is a bigger version of architecture 0. It has 4 convolutional layers which give 75 channels in the end, followed by 6 linear layers. We also sub-sample the images differently. Instead of going from 10x10 to 5x5 immediately, we first use max-pool after the first layer to go from 10x10 to 10x5 and we use strided convolution for the 3rd layer to get to 5x5. Since this is a bigger model, we increase our dropout rate to 50%. It surprised us to find that this architecture did not perform better than the previous one containing fewer layers. The only difference was that it required a bigger batch size to not start degrading (bs=1024) and that it took longer to reach the patience threshold of 20 (42 epochs). In fact, no other modifications we made to the architecture demonstrated any significant improvement over architecture 0. Having a better idea of which optimizer hyper-parameters perform better, we decided to go back to architecture 0 and improve it using non-architectural changes.

IV. Hyper-parameter Grid search

Due to how long it takes to train, we conducted a small grid search only over the hyper-parameters (lr, mom, bs) that we believed would do well (appendix fig. 6). We used a max limit of 50 epochs and 15 patience. Since we already had prior knowledge of which hyper-parameters were appropriate, all combinations in our search showed potential. We must also note that our plots suggest that convergence speed (shown by the slope of the plots and the x-coordinate of the diamond) decreases as batch size increases. But plotting with respect to epochs is not a fair comparison when it comes to comparing different batch sizes. Greater batch sizes will result in models that take less optimizer steps per epoch. Thus the apparent difference is misleading. The better metric for this particular comparison would be to track optimizer steps or just plain time.

V. Final model

Taking the above observations into consideration, we decided to train our final model with a very big batch size in order to get the most representative gradients in the last stages of the learning. This involved using a very large 500 epoch limit with 30 patience. The optimizer hyper-parameters were (lr=0.001, mom=0.95, bs=1000). Due to the large number of instances and our desire to maximize learning, validation was done on a random 2% subset of the training data.

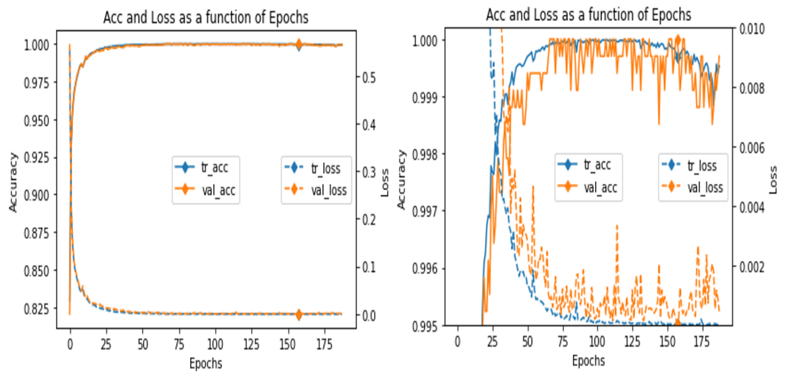


Fig. 3. Our final model's performance (left) and a zoomed-in view (right)

The training was stopped early at epoch 187. The model at epoch 157 (diamond) achieved the smallest validation loss of $4e^{-5}$ and a validation and training accuracy of (1, 0.9996) respectively. Since we used only 2% for validation, it is not surprising that validation accuracy was so high for many epochs. This model achieved a score of 0.99702 on kaggle. We note that other models at prior epochs had higher validation loss but also higher training accuracy. We also wanted to submit their predictions but google colab decided to restart the run-time before we could save the model states :(.

A. *Adam optimizer*

We also trained architecture 0 with the Adam optimizer. Since this uses different hyper-parameters for different weights, we had to use a much bigger batch size of 10000. We use lr=0.001, eps=$1e^{-8}$ and betas=(0.9, 0.999). We ran with a patience of 30. The lowest validation loss was achieved at epoch 24 (appendix fig. 7). This model achieved .99559 on kaggle.

VI. Conclusion

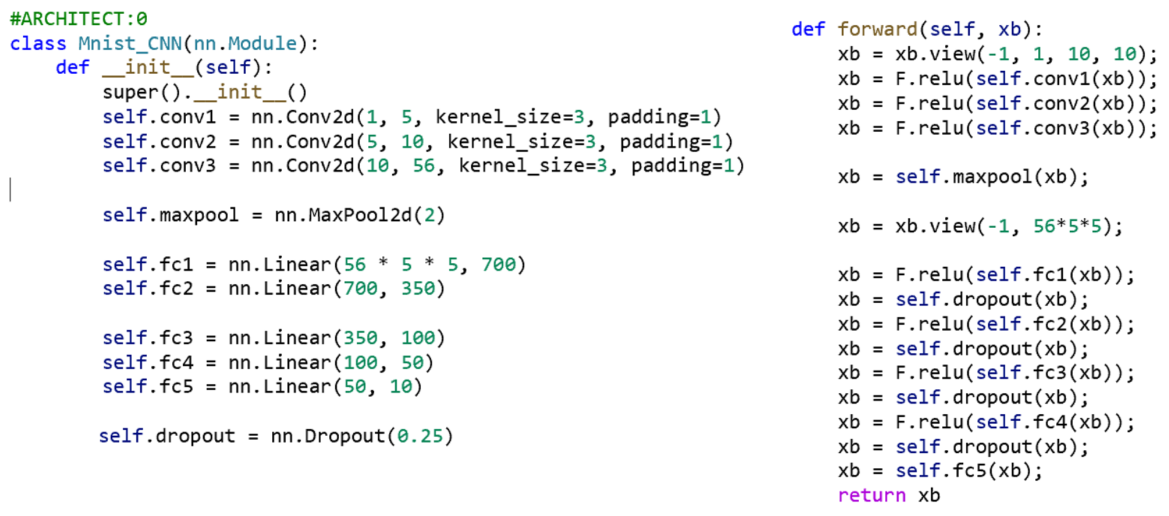
In conclusion, we successfully built a CNN for classifying MNIST digits and combined it with data processing in order to classify images which contain multiple digits. We also gained experience designing neural network architectures, including sub-sampling and regularization layers. We experimented with different gradient descent optimizers and tweaked their hyper-parameters. We also implemented a scheduled decreasing learning rate (using optim.reduceRLonplateau). Namely, we decrease lr by 10% whenever validation loss hasn't decreased for 10 epochs. This model will end training by the deadline at best, hence we do not have a plot to show it's results.

We think that improving our data-processing could improve our accuracy, particularly detecting that digit 3 which is cut in half. Given more time we would try to augment our data (by rotations, translations, etc) as we think this may be one way to reach the 0.998 mark.

VII. Statement of Contributions

All team members worked, contributed and collaborated on all stages of the project.

Appendix



```
#ARCHITECT:0
class Mnist_CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 5, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(5, 10, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(10, 56, kernel_size=3, padding=1)

        self.maxpool = nn.MaxPool2d(2)

        self.fc1 = nn.Linear(56 * 5 * 5, 700)
        self.fc2 = nn.Linear(700, 350)

        self.fc3 = nn.Linear(350, 100)
        self.fc4 = nn.Linear(100, 50)
        self.fc5 = nn.Linear(50, 10)

       self.dropout = nn.Dropout(0.25)
```

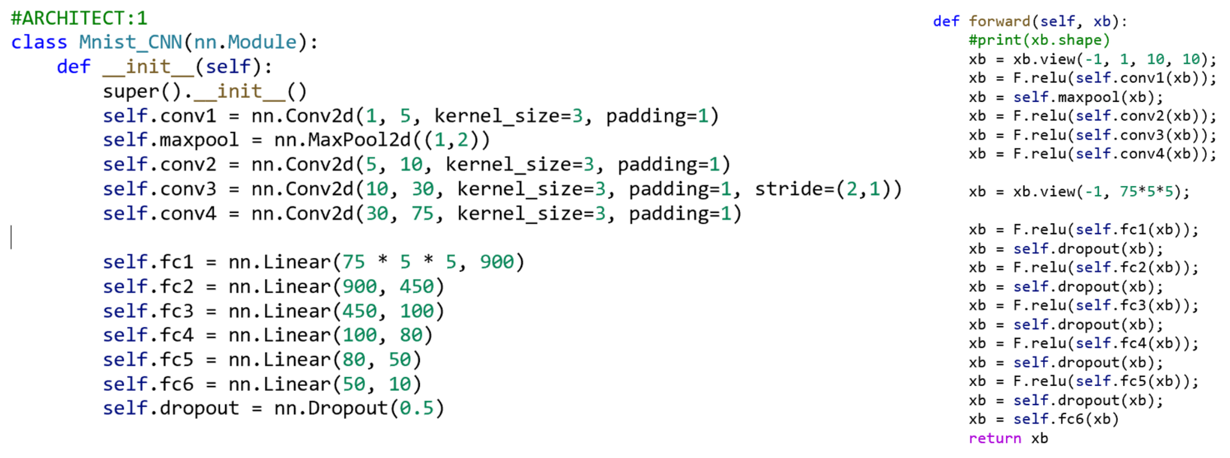
```
def forward(self, xb):
    xb = xb.view(-1, 1, 10, 10);
    xb = F.relu(self.conv1(xb));
    xb = F.relu(self.conv2(xb));
    xb = F.relu(self.conv3(xb));

    xb = self.maxpool(xb);

    xb = xb.view(-1, 56*5*5);

    xb = F.relu(self.fc1(xb));
    xb = self.dropout(xb);
    xb = F.relu(self.fc2(xb));
    xb = self.dropout(xb);
    xb = F.relu(self.fc3(xb));
    xb = self.dropout(xb);
    xb = F.relu(self.fc4(xb));
    xb = self.dropout(xb);
    xb = self.fc5(xb);
    return xb
```

Fig. 4. Architecture 0.



```
#ARCHITECT:1
class Mnist_CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 5, kernel_size=3, padding=1)
        self.maxpool = nn.MaxPool2d((1,2))
        self.conv2 = nn.Conv2d(5, 10, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(10, 30, kernel_size=3, padding=1, stride=(2,1))
        self.conv4 = nn.Conv2d(30, 75, kernel_size=3, padding=1)

        self.fc1 = nn.Linear(75 * 5 * 5, 900)
        self.fc2 = nn.Linear(900, 450)
        self.fc3 = nn.Linear(450, 100)
        self.fc4 = nn.Linear(100, 80)
        self.fc5 = nn.Linear(80, 50)
        self.fc6 = nn.Linear(50, 10)
        self.dropout = nn.Dropout(0.5)
```

```
def forward(self, xb):
    #print(xb.shape)
    xb = xb.view(-1, 1, 10, 10);
    xb = F.relu(self.conv1(xb));
    xb = self.maxpool(xb);
    xb = F.relu(self.conv2(xb));
    xb = F.relu(self.conv3(xb));
    xb = F.relu(self.conv4(xb));

    xb = xb.view(-1, 75*5*5);

    xb = F.relu(self.fc1(xb));
    xb = self.dropout(xb);
    xb = F.relu(self.fc2(xb));
    xb = self.dropout(xb);
    xb = F.relu(self.fc3(xb));
    xb = self.dropout(xb);
    xb = F.relu(self.fc4(xb));
    xb = self.dropout(xb);
    xb = F.relu(self.fc5(xb));
    xb = self.dropout(xb);
    xb = self.fc6(xb)
    return xb
```

Fig. 5. Architecture 1.

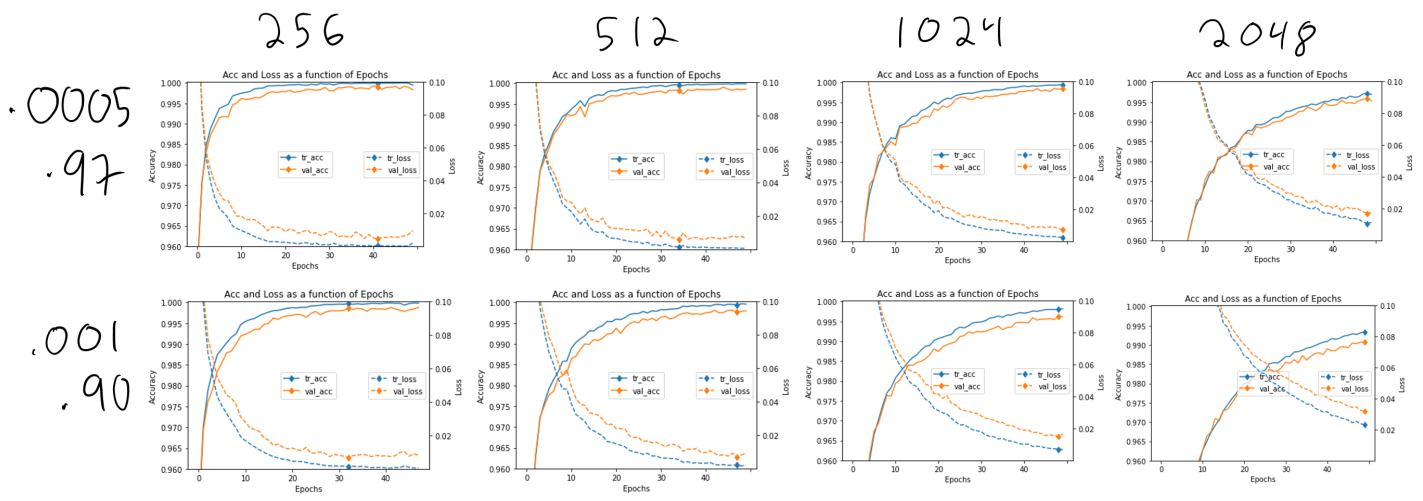


Fig. 6. Hyper-parameter grid. As batch size increases, convergence in terms of epochs is delayed, independent of learning rate and momentum.

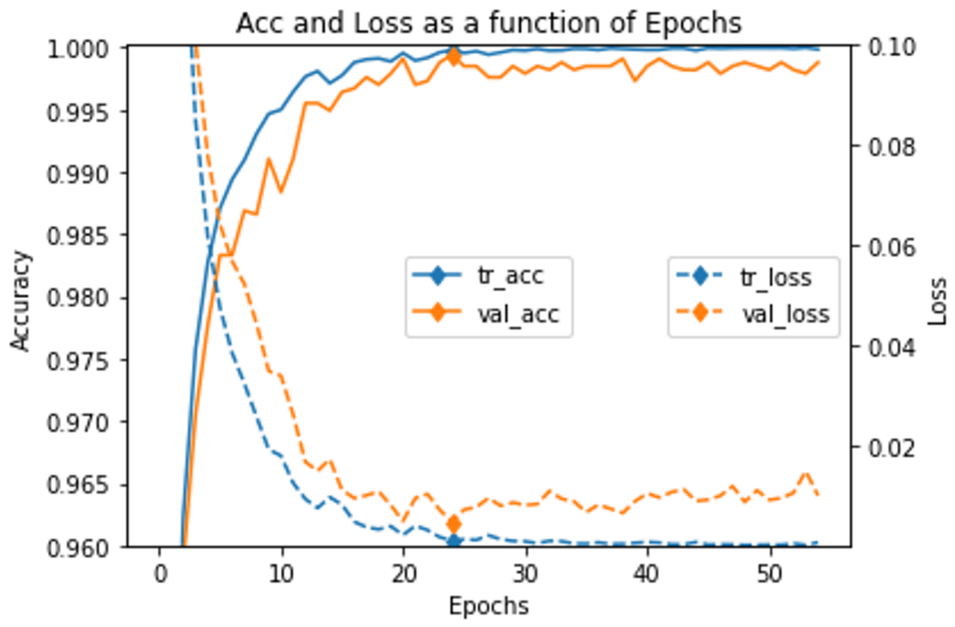


Fig. 7. Using the Adam optimizer did not improve our model's performance (.995 on test data)