Advanced Deep Learning
Assignment 1

pwn274

May 10, 2022

1 Convolutional neural networks

1.1 Basic CNN definition

1.1.1 Give a description of each part of the neural network in words.

The net is a fully convolutional neural network. This means it consists of multiple convolutional layers with pooling layers in-between followed by a ”conventional” fully-connected network. More specifically, as we are doing image-classification the implemented layers are PyTorch `Conv2d`. The first input layer has 3 channels (one for each of the RGB-pixel values) and has a kernel size of 5x5. which is about a fifth of our original image size. This layer outputs with a width of 64 which can be interpreted as the main size of the network. Following this is a convolutional layer with the same hyperparameters (kernel size and stride) but it scales down a size of 32 channels. Hereafter we have the first the first max-pooling layer where we have chosen a kernel size and stride of 2. This means that we take each 2x2 group of ”pixels” and combine them into a single one by choosing the maximal. This of course halves the image-size. These two are then repeated at a size of 3x3 (a convolutional layer outputting a width of 32 and a 2-dimentional max-pool) before moving into the linear part of the model. This last layer works as an output layer, giving us a prediction-node for each of our 53 classes.

1.1.2 Calculate and explain why the number of input features to the last layer (the linear layer) should be 512

We start out by a (28x28) image. As we do not use any padding on the 5x5 convolutional layer it ’looses’ two pixels in each end. Two of these in a row gives us a (20x20) image. Hereafter, the max-pooling layer with kernel size 2 and stride 2 halves the image size to (10x10). Now, another convolution with kernel size (3x3) and pooling takes us to image sizes of (8x8) and (4x4) respectively. Now multiplying this with the width/number of channels in the last convolution layer gives:

$$4 \cdot 4 \cdot 32 = 512$$

Exactly as predicted!

1.1.3 Implement the network in the notebook.

The network class was written as follows:

```
class Net(nn.Module):
    def __init__(self, img_size=28,batch_norm = True):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=(5,5), stride=(1,1))
        self.conv2 = nn.Conv2d(64, 32, kernel_size=(5,5), stride=(1,1))
        self.pool1 = nn.MaxPool2d(2,stride = 2, ceil_mode = False)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=(1,1))
        self.pool2 = nn.MaxPool2d(2, stride = 2, ceil_mode = False)
        self.fc3   = nn.Linear(512, 53, bias = True)

    def forward(self, x):
        x = F.elu(self.conv1(x))
        x = F.elu(self.conv2(x))
        x = self.pool1(x)
        x = F.elu(self.conv3(x))
        x = self.pool2(x)
        x = x.view(-1,512)
        x = self.fc3(x)

        return x
```

Some hyper-parameters from are left out, as, for example, `padding = 0` and `stride = (1,1)` are the default arguments for the `Conv2d`-function in PyTorch.

1.2 Batch normalization

There were some discussion during the TA-session about where the batch-normalization should go. I ended up following this quote from the paper: ”We add the BN transform immediately before the nonlinearity” and added them after the convolutions, but before any activation-functions.

The resulting made the model from the last section look like:

```
class Net(nn.Module):
    def __init__(self, img_size=28,batch_norm = True):
        super(Net, self).__init__()

        self.conv1 = nn.Conv2d(3, 64, kernel_size=(5,5), stride=(1,1))
        self.conv2 = nn.Conv2d(64, 32, kernel_size=(5,5), stride=(1,1))
        if batch_norm:
          self.batchnorm2 = nn.BatchNorm2d(32, affine=False)
        else:
          self.batchnorm2 = lambda x: x
        self.pool1 = nn.MaxPool2d(2,stride = 2, ceil_mode = False)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=(3,3), stride=(1,1))
        if batch_norm:
          self.batchnorm3 = nn.BatchNorm2d(32, affine=False)
        else:
          self.batchnorm3 = lambda x: x

        self.pool2 = nn.MaxPool2d(2, stride = 2, ceil_mode = False)
```

```
        self.fc3   = nn.Linear(512, 53, bias = True)

    def forward(self, x):
            x = F.elu(self.conv1(x))
            x = F.elu(self.batchnorm2(self.conv2(x)))
            x = self.pool1(x)
            x = F.elu(self.batchnorm3(self.conv3(x)))
            x = self.pool2(x)
            x = x.view(-1,512)
            x = self.fc3(x)

            return x
```

I do not remember the reasoning, but I did not add the batch normalization for the input- and output nodes. I am unsure why I did this, but it is too late to run my experiments again at this point - keep this in mind for any results I get. As I wanted to be able to run both the batch-normalized and the original model from a loop, I gave the model's `__init__`-function an additional boolean parameter which activated the corresponding layers.

1.3 Optimizers

1.3.1 Switch to a different optimizer and show the changes to the code in the report.

As I know Adam is a sophisticated optimizer, I wanted to see what the more where I understand the theory. I implemented Stochastic Gradient Descend using the following line:

```
optimizer = optim.SGD(net.parameters(), lr=LR, momentum=0.9)
```

1.3.2 Describe the main features of the optimizer, and argue for your choice of optimizer and hyperparameters.

After doing some research, I understood that Adam partly can be seen as doing the same as the simpler Stochastic Gradient Descend with momentum, so I chose to also add momentum. The value of 0.9 seemed generally to be the canonical starting point. I chose to keep the learning rate the same as with Adam to make the comparison somewhat more valid.

1.4 Experimental architecture comparison

In the report, you are supposed to describe what you did, present the results, discuss the results, and draw very careful preliminary conclusions. The report should contain a plot (remember proper axes labels etc.) showing the training progress vs. epochs.

I wrote a loop which trained, tested and plotted the model for different choices of optimizers and choice of batch-normalization. All are trained for 200 epochs.[1] This gave the following results:

[1] I just realized that the x-axis is wrong on my plot, as this counts the individual batches and is therefore off by a factor of 3 - just know that I am aware of the dissonance.

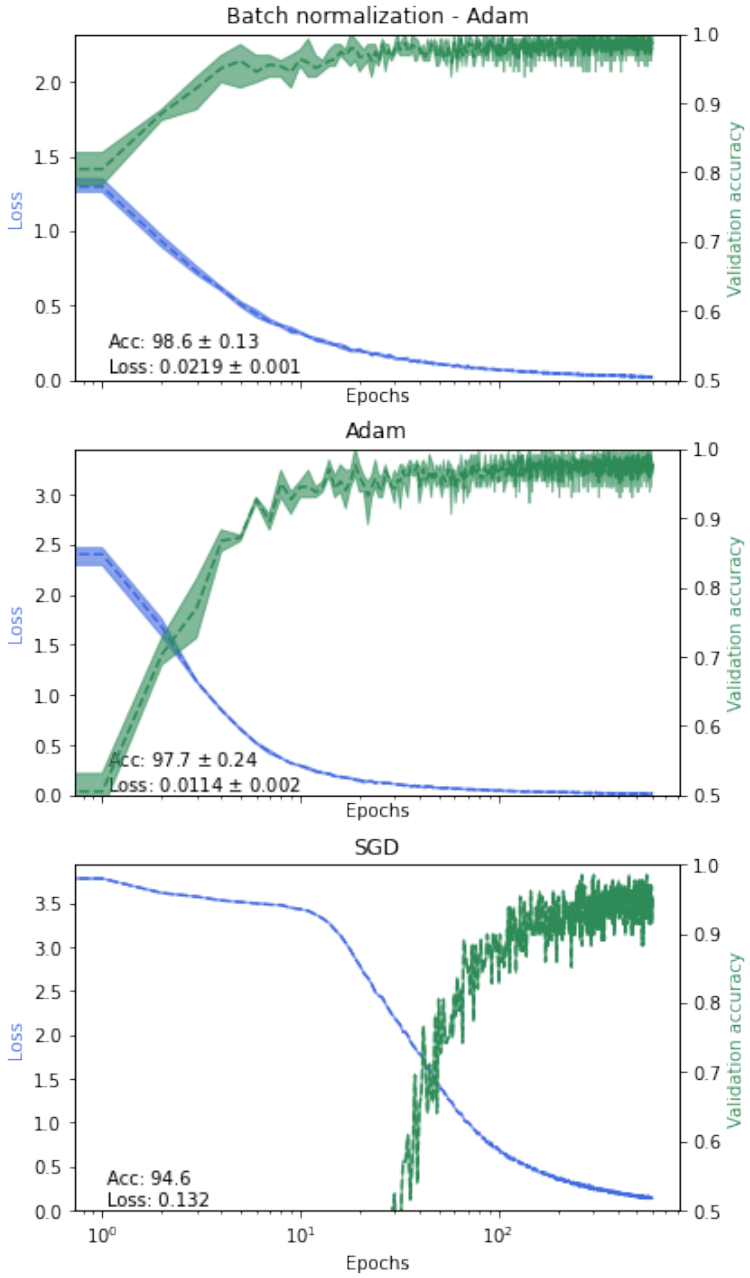

Figure 1: On top we have the standard model with batch-normalization added as described in the above section. In the middle (for easy comparison), the model as initially described in the assignment trained with Adam using a learning rate of 0.01. The last is the same model, but this time trained using Stochastic Gradient Descend. Note that this model was only trained once.

In the plot, the coloured regions encapsulate the min/max of the values from each of the three runs. The dotted lines are the average. Blue represents training loss and green validation accuracy. Please note that the x-axis is shared for each column and is on a log scale. The accuracies and losses written in the plots are the average final results on the test-set.

It seems that the model with batch normalization never reaches the same loss as the one without. Within the significance of my limited experiment, I would feel quite secure in saying that batch-normalization helps in achieving a higher accuracy given the same amount of training on this setup of data and model. It can also be seen why Adam is the de-facto industry standard as the simple Stochastic Gradient Descend seems quite a lot less accurate. But as it graphically does not seem to have converged as much as the other two and it was only run a single time, I do not feel able to conclude anything.

1.5 Challenge (optional)

Colab, Kaggle and my power bill are already overused, so I did not get around to doing the optional part of this assignment.

2 U-Nets

To implement this experiment, an improved training loop was made that trained each version of the model for 3x200, 3x400 and 3x1000 epochs before testing it on the test-set. All running losses, the f1-validation-accuracies and final test-accuracies were then saved. The resulting plots can be seen below:

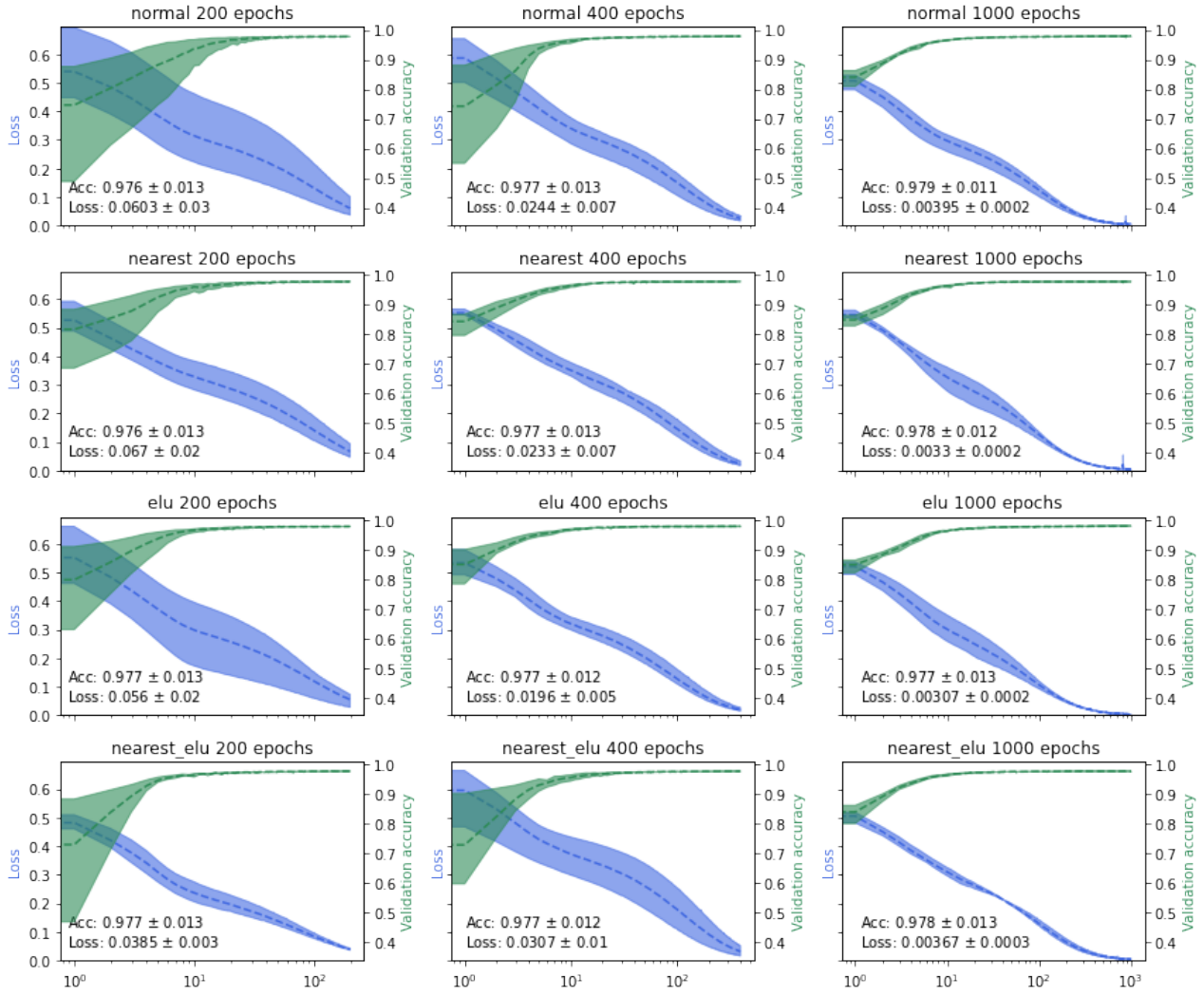

Figure 2: The summary plots for my training runs for each of the different models for each number of epochs. *Normal* is the U-net as handed out, *nearest* uses the Nearest Neighbohr method for upsampling, *elu* uses the ELU activation function and described in the assignment and *nearest_elu* uses both changes to the original structure.

On the plot, the final test accuracies and losses are written. If the final test-accuracies are hard to read I have also arranged them in the following table. The certainties are all in close to each other in the $0.011 - 0.013$-range so I have left these out and kept three significant digits:

Type/Epochs	200	400	1000
Normal	0.976	0.977	0.979
Nearest Neighbor	0.976	0.977	0.978
Using ELU	0.977	0.977	0.977
Nearest and ELU	0.977	0.977	0.978

It seems that all the final models were just about equally good, with the exception being, that the 'Nearest Neighbor' model (no ELU-activation nearest-neighbor upsampling) are

generally lower, albeit not significantly so. The

Assuming any noise in training is Gaussian (which my experience tells me is a quite good assumption), I use the Standard score/z-value formula

$$\left(\frac{a-b}{\sqrt{\sigma_a^2 + \sigma_b^2}} \right)$$

to figure out how many standard deviations any of the two means are from each other. Doing this for the 'normal' model after 200 and 1000 epochs (biggest difference in results I could find), I can see that it only grows by 0.17-sigma which in no way is statistically significant. Running the same test more times would most likely improve the uncertainties and would allow for better conclusions to be made.

It can be seen, that when running fewer epochs the ELU activation function gives a significantly lower final test-loss across the board.

Please note: Looking at figure 2, I do not know why accuracy and loss of the model is less dependent on initial seed when I was going for a higher total of epochs and I unfortunately do not have the time to run my experiments any additional times. I might have made a mistake in my code that I am unable to find, but as the results are already uncertain, I think this only further cements the idea that more than 3 runs are required for proper statistical significance.