Computational Intelligence 2020 **(Due: May 14th)**

Submission Assignment #4

Instructor: Jakub Tomczak *Name:* Ravi Bossema, *VUnetID:* 2640682

1 Introduction

The goal of this assignment is to get familiar with neural networks (**NN**s) by implementing two **NN**s: A fully-connected neural network (**FCN**), and a convolutional neural network (**CNN**). A neural network is an algorithm that takes inspiration from the neural networks in our brain. A neural network is made up of layers of neurons that are connected to each other. Usually there is an input layer, a number of hidden layers and an output layer as shown below.

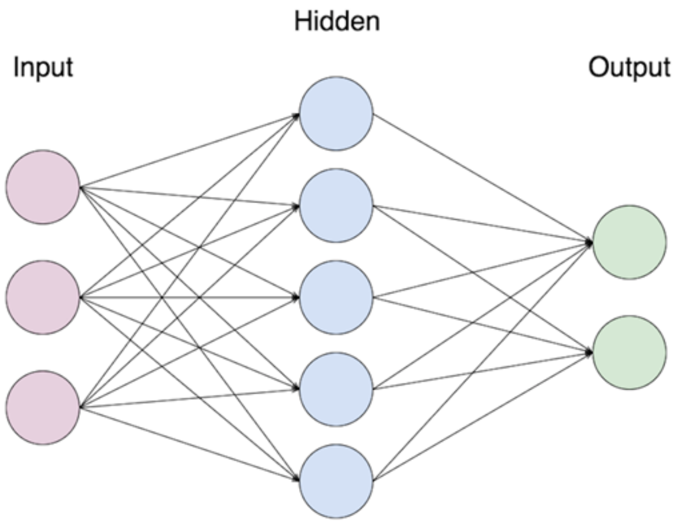


Figure 1: Example of a fully-connected neural network

These neurons are essentially containers that hold a value. The values of the input neurons are determined by the input and the values of the other neurons, in the hidden layer(s) and the output layer, are determined by the connections and their weights. In the example in figure 1, the values of the input neurons are sent to all the neurons in the hidden layer with a certain weight. The weight affects the outcome, thus the value of the neurons in the hidden layers, by multiplying with the value of the input neuron. Each connection has its own weight and (hopefully) after training, these weights will be optimized so that the desired values appear in the output layer.

To train a **NN** we first do a forward pass, filling in the input values and calculating the hidden layer(s) and the output layer according to the weights and connections. We then compare the output values to the actual/target output and compute the loss. We then do a backward pass, adjusting all the weights in accordance with the gradient, this is called back propagation. This way, the network learns the optimal weights for the connections between all the layers, thus learning how to predict the right output for a given input.

In figure 1 we can see an example of a fully-connected network, in which all neurons of one layer are connected to all neurons on the next layer. Thus the name 'fully-connected'. In this assignment however, I also test a convolutional neural network. **CNN**s take a different approach from **FCN**s in the sense that not all the neurons are connected directly. Different kernels are made that essentially create groups of neurons that are connected to the next layer. This approach is usually used for image recognition since it is useful to be able to look at different sections of an image and not just the entire image. This should make it easier for the **NN** to recognize subsections (like shapes) of the image and piece them together to figure out what it's looking at.

2 Problem statement

The problem of this assignment is image classification using neural network training. For the assignment I was assigned a dataset containing a total of 13394 images. This dataset was seperated into a training set: 8049 images, a validation set: 1420 images, and a test set: 3925 images. Each image is part of one of ten classes: English springer, French horn, cassette player, chain saw, church, garbage truck, gas pump, golf ball, parachute, tench. These are some examples:

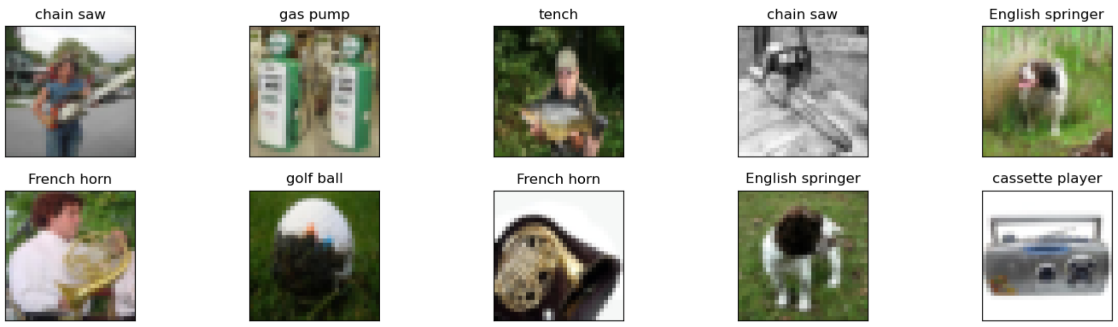


Figure 2: Sample of images from the dataset with their respective classes

As you can see there can be a large difference between two images of the same class which would make it quite difficult for a neural network to properly classify them. In the example in figure 2 it is specifically noticeable that the first image of the French horn looks much more similar to the images of the English springer. And considering that the dataset that was assigned is significantly smaller than other similar datasets such as CIFAR-10 Krizhevsky (2017), which has a training set of 50000 images, it is very possible that the **NN** will not be able to overcome such difficulties due to a lack of training data. One way to combat this is to increase the number of **epochs** the **NN** will be trained over but since the training data stays the same, too many **epochs** can cause the network to over-fit on the training data. This is something to look out for in the experiment.

3 Methodology

The implementation of the **NN**s was largly done using the Python library PyTorch. PyTorch contains functions and modules that allow you to more easily implement, or even skip, certain steps in the process. Using these functions and modules I created two classes:

- **FCN**

  This is the class used for the fully-connected network (**FCN**). An instance of this class has two linear layers with the following structure: $x \rightarrow h \in \mathbb{R}^{300} \rightarrow y$. As instructed in the assignment I used ReLU for the first non-linearity and softmax for the output. ReLU (Rectified Linear Unit) is a function that behaves like a linear function for positive inputs but always returns 0 for negative values: $relu(x) = max\{0, x\}$. This has the property of sparsifying the gradient, since it alsways either 0 or 1. Softmax is a function that normalizes an input vector into a probability distribution: $softmax(x) = exp(x_i)/\sum exp(x_j)$.

  Thus, to summarize, I take an input vector x and I transform it using the ReLU function before it goes into the hidden layer h (which has 300 neurons). Then I use the softmax function to create a probability distribution which will be represented in the output layer y. y will then be returned after a forward pass.

- **CNN**

  This is the class used for the convolutional neural network (**CNN**). An instance of this class also has two linear layers but the structure is slightly different:

$$input \rightarrow Conv2d + ReLU \rightarrow Pooling \rightarrow Linear + ReLU \rightarrow Linear + Softmax \rightarrow output$$

  Conv2d applies a convolution over an input of a number two-dimensional planes. The output value of the layer with input size (N, C_{in}, H, W) can be described as:

$$out(N_i, C_{out}) = bias(C_{out_j}) + \sum_{k=0}^{C_{in}-1} weight(C_{out}, k) * input(N_i, k)$$

For the pooling I used the PyTorch function nn.MaxPool2d which applies a max pooling over an input of a number of two-dimensional planes. The output value of the layer with input size (N, C, H, W), output (N, C, H_{out}, W_{out}) and kernel size (kH, kW) can be described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\dots,kH-1} \max_{n=0,\dots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

For every **epoch** I train the network over the entire training set and then I validate it over the entire validation set. The training of the network is done with the same steps regardless of whether it is an **FCN** or a **CNN**. I get a batch of images and their classes from the dataset and I put them into the network to get a batch of predictions. These predictions and the actual/target classes are put into the objective function to calculate the loss of that batch. Then I set all the weights to zero so that I can do a backward pass and update the weights according to the optimizer function. After training all the batches I validate the network on the validation set. This process is the same as the training only I don't do a backward pass so I don't update the weight such that the network doesn't learn from the validation set. The goal of validating is to find out how well the network performs with data it hasn't trained on, partially to check that the network isn't over-fitting on the training set. It is essentially a smaller version of testing that you can do during the experiment.

For this assignment I was instructed to use cross entropy loss as the objective function and the stochastic gradient descent as the optimizer. Cross entropy loss is the combination of log softmax and the negative log likelihood loss. The loss can be described as:

$$loss(x, class) = -log(\frac{exp(x[class])}{\sum_j exp(x[j])}) = -x[class] + log(\sum_j exp(x[j]))$$

I implemented the stochastic gradient descent (SGD) with the Nesterov momentum which is based on the formula from Sutskever et al. (2013). SGD continuously updates cerain parameters, in this case the weights. With p, g, v, μ being the parameters, gradient, velocity, and momentum respectively, the update can be written as:

$$v_{t+1} = \mu * v_t + g_{t+1}$$
$$p_{t+1} = p_t - learning\_rate * v_{t+1}$$

After training and validating I plotted the mean individual training- and validation-losses per **epoch** and I tested the network on the test set. Testing works in approximately the same way as validating in the sense that the model doesn't learn during testing, however, for testing I was not only interested in the mean loss but I also wanted to know the accuracy over the entire test set and more specifically the accuracy per class. Thus while testing I also kept track of the proportion of correct and incorrect predictions and to which classes these predictions belong.

4 Experiments

In the experiment I want to compare the two approaches **FCN** and **CNN** by training a number of models with different amounts of **epochs** and different **learning rates**. I will compare the different networks based on the average loss of the test set, the accuracy on the test set and the accuracies per class. Furthermore I also keep track of the average training duration of each **epoch**.

Since I was instructed to use a lot of specific values, only the **learning rate** is a real hyperparameter. I could have tested different **momentum** values as well but since I am unfamiliar with this approach I decided to keep it at 0.9 as was advised in a number of examples and tutorials. Based on the dataset, the size of the input layer is 3072 (3*32*32) and the size of the output layer is 10, because there are 10 classes. The hidden layer contains 300 neurons, as per the instructions from the assignment. In the **CNN** the first linear layer has an *input_size* of 8192 (32*16*16). I am not entirely sure how this is calculated but after some trial and error I found that this was the size that remained after applying Conv2d and pooling. Each image has 3 *in_channels* (RGB) and 32 *out_channels* or *filters* as per the instructions from the assignment. Furthermore I was instructed to use a **stride** of 1, a *kernel_size* of 3 by 2, a *padding* of 1 and a *pooling_size* of 2 by 2. Also, since the training set and the validation set are gathered from one larger training set, since the original dataset only contains a training set and a test set, I couldnt easily calculate the size of these sets with a simple function like I could do with the test set. So instead I calculated them beforehand and I put these values in my script. The training set length is 8049 and the validation set length is 1420.

I quickly realised that the *batch_size* had little to no impact on the performance of the model, apart from the duration, with lower smaller *batch_sizes* leading to longer training durations. Therefore I decided to stick with the default *batch_size* of 32. The main hyperparameter that I want to test is the **learning rate**. For the initial test I ran both networks (**FCN** and **CNN**) for 100 **epochs** with a *batch_size* of 32, and a **learning rate** of 1e-4, 5e-4, and 1e-3. I chose to run this initial test for 100 **epochs** to find out what the optimal number of **epochs** is. Luckily, this is not a blind search since the plot of the training loss and validation loss can indicate when a model stops improving and starts to over-fit on the training data. Thus I will use this initial test to approximate the optimal number of **epochs** for each **learning rate**.

5 Results and discussion

After the initial tests the results are as follows:

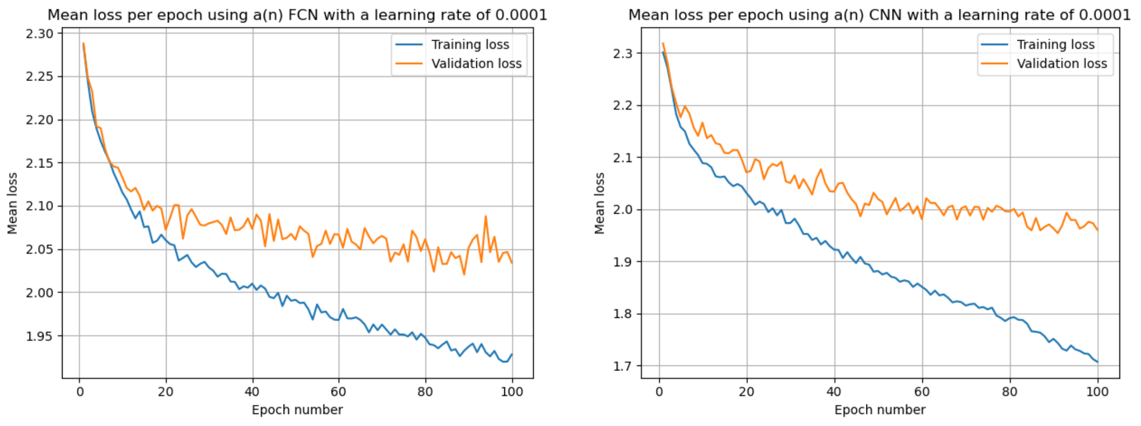


Figure 3: Plots of training and validation loss

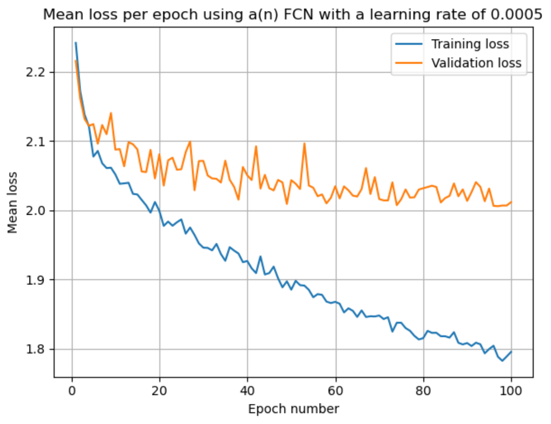
Duration per **epoch**	5.43 s
Average loss	2.0401
Average accuracy	42%
Accuracy of English springer	28%
Accuracy of French horn	33%
Accuracy of cassette player	70%
Accuracy of chain saw	0%
Accuracy of church	55%
Accuracy of garbage truck	50%
Accuracy of gas pump	0%
Accuracy of golf ball	48%
Accuracy of parachute	56%
Accuracy of tench	60%

Table 1: FCN

Duration per **epoch**	5.84 s
Average loss	1.9111
Average accuracy	55%
Accuracy of English springer	55%
Accuracy of French horn	56%
Accuracy of cassette player	70%
Accuracy of chain saw	4%
Accuracy of church	67%
Accuracy of garbage truck	68%
Accuracy of gas pump	51%
Accuracy of golf ball	57%
Accuracy of parachute	62%
Accuracy of tench	66%

Table 2: CNN

Table 3: **learning rate** = 1e-4, **epochs** = 100



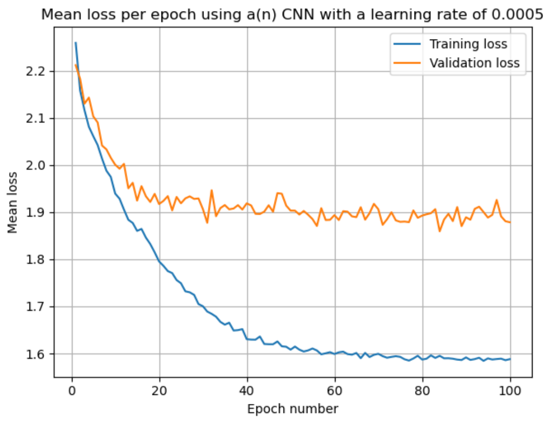


Figure 4: Plots of training and validation loss

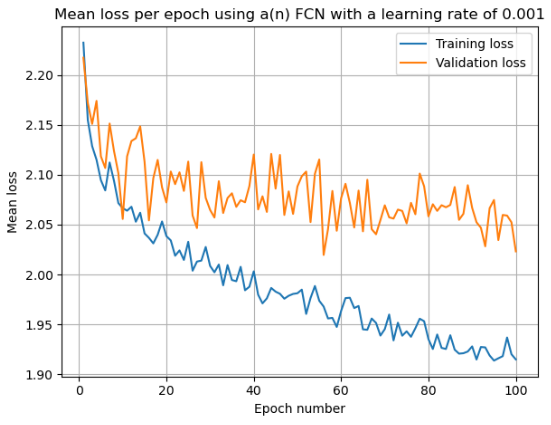
Duration per **epoch**	5.49 s
Average loss	2.0096
Average accuracy	45%
Accuracy of English springer	46%
Accuracy of French horn	25%
Accuracy of cassette player	70%
Accuracy of chain saw	0%
Accuracy of church	59%
Accuracy of garbage truck	56%
Accuracy of gas pump	34%
Accuracy of golf ball	57%
Accuracy of parachute	47%
Accuracy of tench	68%

Table 4: FCN

Duration per **epoch**	5.90 s
Average loss	1.8930
Average accuracy	57%
Accuracy of English springer	53%
Accuracy of French horn	47%
Accuracy of cassette player	65%
Accuracy of chain saw	31%
Accuracy of church	61%
Accuracy of garbage truck	60%
Accuracy of gas pump	40%
Accuracy of golf ball	46%
Accuracy of parachute	66%
Accuracy of tench	70%

Table 5: CNN

Table 6: **learning rate** = 5e-4, **epochs** = 100



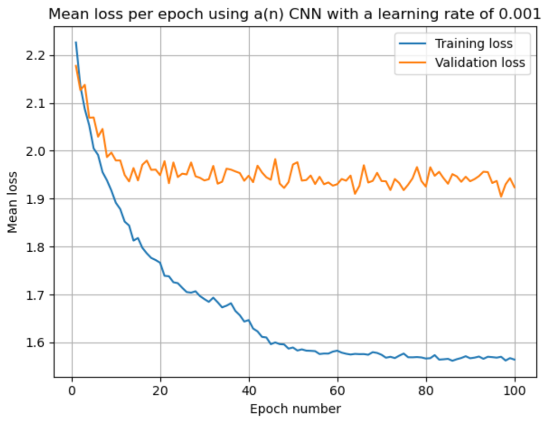


Figure 5: Plots of training and validation loss

Duration per **epoch**	5.62 s
Average loss	2.0198
Average accuracy	44%
Accuracy of English springer	32%
Accuracy of French horn	37%
Accuracy of cassette player	63%
Accuracy of chain saw	16%
Accuracy of church	48%
Accuracy of garbage truck	37%
Accuracy of gas pump	26%
Accuracy of golf ball	48%
Accuracy of parachute	43%
Accuracy of tench	68%

Table 7: FCN

Duration per **epoch**	5.97s
Average loss	1.8925
Average accuracy	57%
Accuracy of English springer	51%
Accuracy of French horn	54%
Accuracy of cassette player	72%
Accuracy of chain saw	37%
Accuracy of church	73%
Accuracy of garbage truck	64%
Accuracy of gas pump	38%
Accuracy of golf ball	61%
Accuracy of parachute	70%
Accuracy of tench	64%

Table 8: CNN

Table 9: **learning rate** = 1e-3, **epochs** = 100

From these results it becomes clear that the **CNN**s outperform the **FCN**s by a significant margin, 12-13% in fact. It makes sense that this would be the case since **CNN**s are specifically designed for image classification. What is notable, however, is that both **NN**s seem to have a lot of trouble with the chain saw class. Especially the **FCN**s since they scored a 0% accuracy for this class in two of the three tests. But the **CNN** also struggled quite a bit with this class performing well under the average accuracy of the network for all of the tests. The gas pump also seems to have been a point of trouble for both **NN**s but not quite to the same extent as the chain saw. Out of curiosity I calculated the average accuracy of the tests if the chain saw was not part of the classes and I found that the accuracy would go up by approximately 2-6%. This would, however, not be a fair adjustment since I could keep following that logic until I only have one class and all the guesses are correct since there is only one possibility. Nonetheless, it is an interesting result.

Another interesting result is the fact that the **FCN**s seem to run slightly faster than the **CNN**s. This was pretty surprising since one of the TA's (I won't name names) suggested that the **CNN**s should run faster than the **FCN**s. It does however seem reasonable since the **FCN**s perform fewer tasks and the difference in duration is quite small (approximately 0.4 seconds per batch). Furthermore, in figure 3 you can see that the networks (especially the **CNN**) seem to still be learning after 100 **epochs**, it seems as if the **learning rate** is low enough such that they can keep learning at that point. That might also explain the difference in performance (although small) between this network and the **CNN**s with a higher **learning rate**. To find out if this is the case I decided to run another test with a **learning rate** of 1e-4 but this time with 150 **epochs**.

In figure 5 it becomes quite clear that the network doesn't improve much after 50 **epochs**. Interestingly, the validation loss doesn't go up after the training loss seems to stagnate. One would expect that to happen since this stagnation is an indication that the network is starting to over-fit on the training data. Therefore I would like to see if the network would perform better if we stop the training after 50 **epochs**. I would also like to see if we can increase the learning rate even more to 5e-3.

With these changes I ran three more tests and the results are as follows:

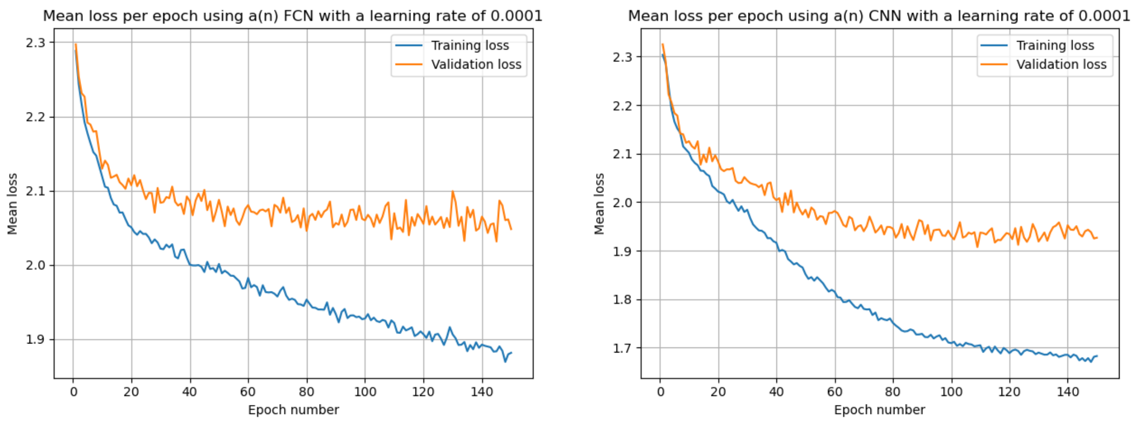


Figure 6: Plots of training and validation loss

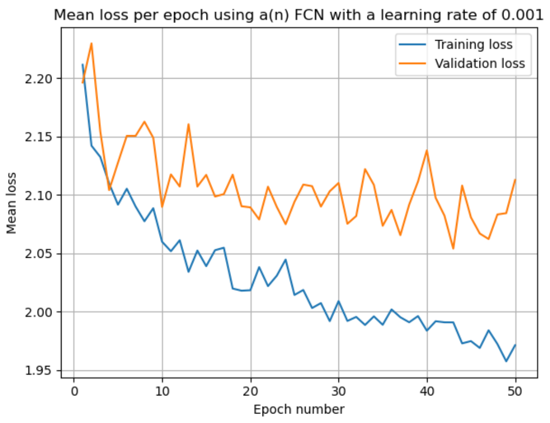
Duration per **epoch**	5.62 s
Average loss	2.0348
Average accuracy	42%
Accuracy of English springer	36%
Accuracy of French horn	47%
Accuracy of cassette player	61%
Accuracy of chain saw	0%
Accuracy of church	53%
Accuracy of garbage truck	43%
Accuracy of gas pump	0%
Accuracy of golf ball	51%
Accuracy of parachute	60%
Accuracy of tench	72%

Table 10: FCN

Duration per **epoch**	6.35s
Average loss	1.9116
Average accuracy	55%
Accuracy of English springer	57%
Accuracy of French horn	47%
Accuracy of cassette player	75%
Accuracy of chain saw	0%
Accuracy of church	67%
Accuracy of garbage truck	58%
Accuracy of gas pump	53%
Accuracy of golf ball	61%
Accuracy of parachute	68%
Accuracy of tench	70%

Table 11: CNN

Table 12: **learning rate** = 1e-4, **epochs** = 150



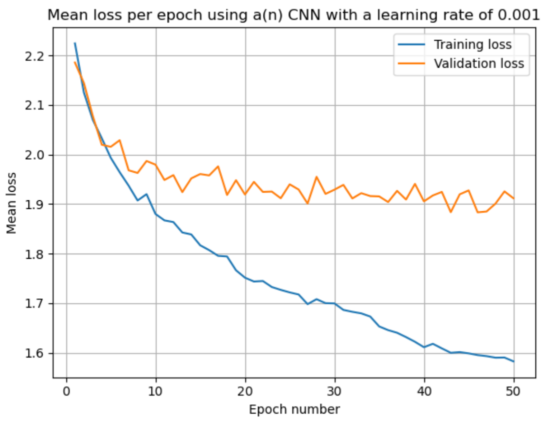


Figure 7: Plots of training and validation loss

Duration per **epoch**	5.57 s
Average loss	2.0613
Average accuracy	40%
Accuracy of English springer	24%
Accuracy of French horn	42%
Accuracy of cassette player	68%
Accuracy of chain saw	0%
Accuracy of church	57%
Accuracy of garbage truck	56%
Accuracy of gas pump	9%
Accuracy of golf ball	44%
Accuracy of parachute	50%
Accuracy of tench	47%

Table 13: FCN

Duration per **epoch**	6.66 s
Average loss	1.8971
Average accuracy	56%
Accuracy of English springer	51%
Accuracy of French horn	33%
Accuracy of cassette player	75%
Accuracy of chain saw	45%
Accuracy of church	73%
Accuracy of garbage truck	54%
Accuracy of gas pump	44%
Accuracy of golf ball	57%
Accuracy of parachute	66%
Accuracy of tench	70%

Table 14: CNN

Table 15: **learning rate** = 1e-3, **epochs** = 50

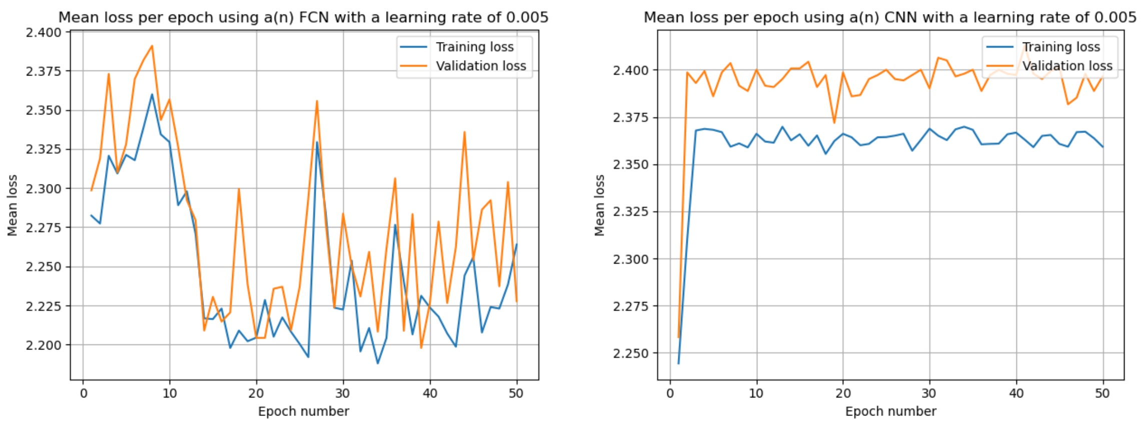


Figure 8: Plots of training and validation loss

Duration per **epoch**	5.82 s
Average loss	2.2170
Average accuracy	25%
Accuracy of English springer	2%
Accuracy of French horn	79%
Accuracy of cassette player	40%
Accuracy of chain saw	0%
Accuracy of church	0%
Accuracy of garbage truck	16%
Accuracy of gas pump	0%
Accuracy of golf ball	50%
Accuracy of parachute	60%
Accuracy of tench	1%

Table 16: FCN

Duration per **epoch**	6.05 s
Average loss	2.3605
Average accuracy	10%
Accuracy of English springer	100%
Accuracy of French horn	0%
Accuracy of cassette player	0%
Accuracy of chain saw	0%
Accuracy of church	0%
Accuracy of garbage truck	0%
Accuracy of gas pump	0%
Accuracy of golf ball	0%
Accuracy of parachute	0%
Accuracy of tench	0%

Table 17: CNN

Table 18: **learning rate** = 5e-3, **epochs** = 50

From these results we can gather that increasing the number of **epochs** did not significantly improve the performance of either network with a **learning rate** of 1e-4. In fact, the accuracies stayed the exact same and even the individual accuracies of each class closely resemble those of the same networks trained for 100 **epochs**.

Decreasing the amount of **epochs** for the networks with a **learning rate** of 1e-3 also had little to no effect on their performance. The accuracy even went down for both networks, especially for the **FCN**.

Most interesting though, are the results from the test with a **learning rate** of 5e-3. The **FCN** performs incredibly poorly with an accuracy of only 25%. It seems as if the network decided to mainly focus on learning a couple of the classes, in this case the French horn, cassette player, golf ball, and parachute. The accuracies for these classes are still quite high but the rest seem to be largely disregarded by the network. The **CNN** takes this to the extreme, seemingly deciding that every image is an English springer. You can also see in figure 8 that both the training- and validation-losses immediately shoot up and stay around that level for the rest of the training.

It is clear that **CNN**s are better suited for this kind of problem and while the right combination of hyperparameters still gave quite an impressive accuracy, I think that the size of the dataset severely limited the potential of the networks. Furthermore, it could be interesting to try out different values for the hyperparameters that I was not allowed to change, such as the *kernel_size* and the *stride*. However, these boundaries were quite useful for saving time since trying out different values for each of these hyperparameters would have created a humongous set of combinations. But if I have the time, I would like to explore these hyperparameters more in future experiments.

References

Krizhevsky, A. (2017). The cifar-10 dataset.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning.