Exercise 1 – Report

Antigen Discovery for SARS-CoV-2 (“Corona”) Virus Vaccine

[Data Handling 1](#_Toc86434733)

[Models Architectures 1](#_Toc86434734)

[First Training 2](#_Toc86434735)

[Hyper-Parameters Tuning 2](#_Toc86434736)

["Basic" Hyper-Parameters Tuning 3](#_Toc86434737)

[Batch size 3](#_Toc86434738)

[Learning rate 3](#_Toc86434739)

[Epochs number 5](#_Toc86434740)

[Activation function 5](#_Toc86434741)

[Architectures Comparison 6](#_Toc86434742)

# Data Handling

Each data item is a 9-lengthed sequence of characters, and all of the characters are in a vocabulary of 20 characters. We decided to map each sequence to 9\*20 vector – concatenation of nine (one for character) one-hot encoding vectors, each one is of size 20. This seems like a natural choice because we do not have prior assumptions about the data, we want to keep the order and we do not know if there is any connection between some of the characters.

# Models Architectures

We will try multi-layered perceptron networks with different parameters. The parameters are numbers of layers, number of neurons in each hidden layer and activation function. In all cases the first layer will get 9\*20-sized input vector and the last layer will output 1-sized output vector after going through sigmoid activation. In inference, we round the output, so effectively the threshold is 0.5.

# Experiments

We actually wrote this report twice. In the first time we trained our network and saw good results – accuracy, recall and precision were all above 0.9. We tried different hyper-parameters and optimized the results. We knew we would have to deal with the imbalanced data, but were mislead by the high scores and thought this specific issue is a matter of some precents improvement. However, when we measured a relevant metric directly (false negative ratio) we noticed that we trained models that only output “True”. We easily fixed it with oversampling the negative samples and repeated all our experimetns. We left here (in the end under “Problematic Experiments”) also the first set of experiments to show the full research process.

First Training

For the first training, we tried feed forward network with one hidden layer with 256 neurons with relu activation. We trained the network for 5 epochs. We used batch size of 64.

We used binary cross entropy loss and learning rate of 0.001.

Results on the (oversampled) test set:

(tn:2170, fp:171, fn:381, tp:2120)

true positive rate (AKA recall): 0.848, true negative rate: 0.923

accuracy: 0.886

recall: 0.848

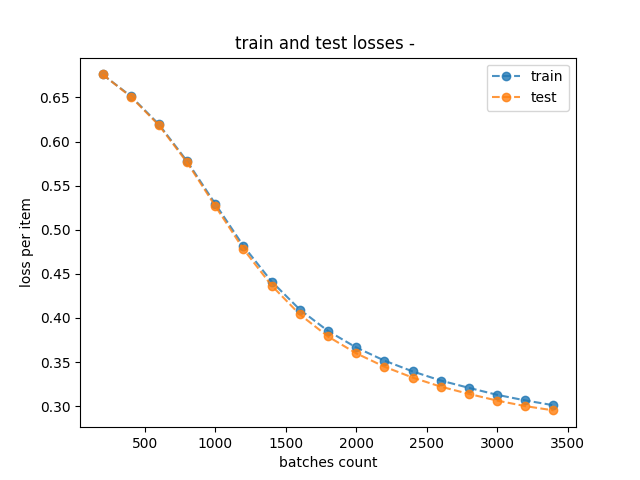
precision: 0.925

f1: 0.885

The results were measured on the oversampled data. It is not the best practice and we will improve it in the section that deals directly with imbalanced data, but it is fine for now.

We can see that the results are fine – all of the metrics are above 84%, and the model succeeds both on positive and negative results.

Those are the learning curves:



This was a sanity check and now we can go forward to hyper-parameters tuning.

Hyper-Parameters Tuning

There are many architectural choices and parameters to tune when training a neural net. We decided to split them to two groups: "basic" hyper-parameters and architectural parameters.

Basic hyper-parameters

* Learning rate
* Batch size
* Epochs number
* Activation function

Architectural parameters

* Number of layers
* Number of neurons in each layer

We will start by choosing the basic hyper-parameters with an arbitrary architecture, and once find a good configuration test the different architectural parameters with this configuration.

"Basic" Hyper-Parameters Tuning

The basic architecture I will try the parameters with is multi-layered perceptron with one hidden layer of 256 neurons. The loss function is binary cross-entropy.

Batch size

Usually choosing batch size has less to do with model quality and more with performance issues so we just tried some batch sizes and chose 64, which gave fine results in short training time. There is a connection between batch size and learning rate but once we set a fixed batch size, we can just choose the proper learning rate.

**Batch size=64**

Learning rate

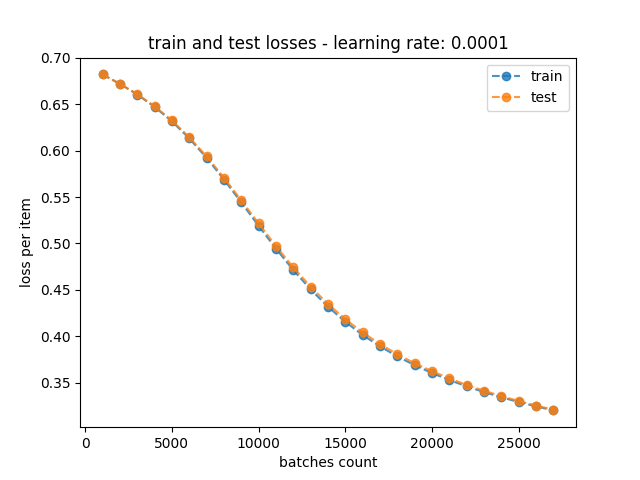
We trained the network with different learning rates, from 0.0001 to 0.1, for 40 epochs.

The full list:

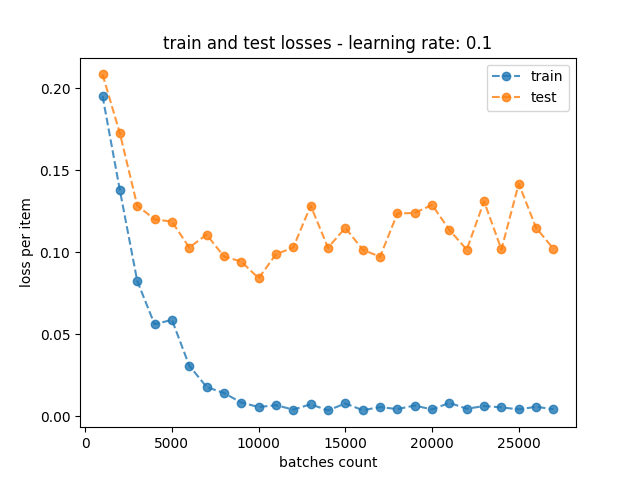
learning\_rates = [10 \*\* (-4), 5 \* 10 \*\* (-4), 10 \*\* (-3), 5 \* 10 \*\* (-3), 10 \*\* (-2), 5 \* 10 \*\* (-2), 10 \*\* (-1)]

Let's take a look in some learning curves to choose a good value:

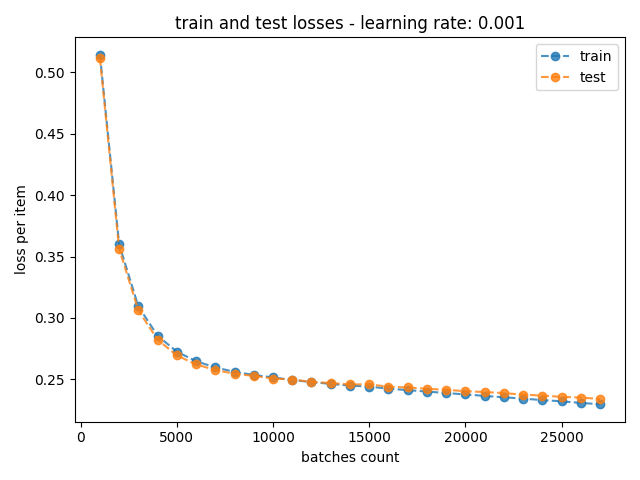
When we have a very small learning rate (0.0001), we have stable but slow learning:



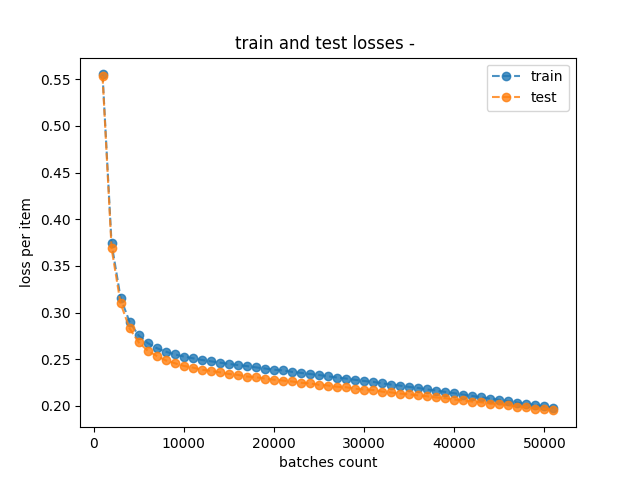
When we have a very big learning rate, (0.1) the train loss goes down but the test loss does not – overfit.



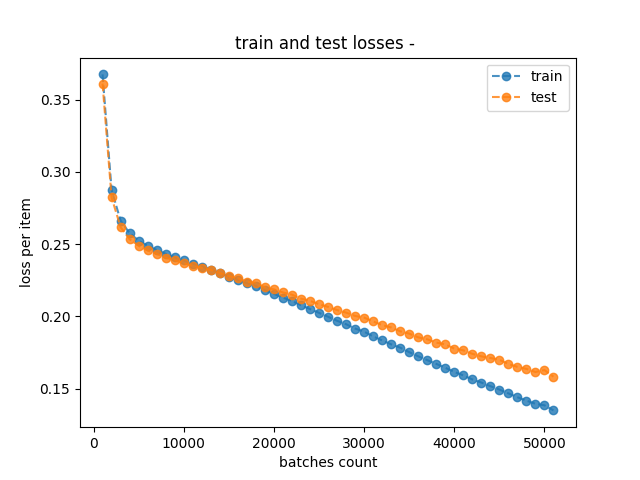
With a medium learning rate (0.001), we get fast convergence as wanted:



We can see that after 40 epochs there is still a downward trend of the test loss, so we will try to train a little more – for 75 epochs.

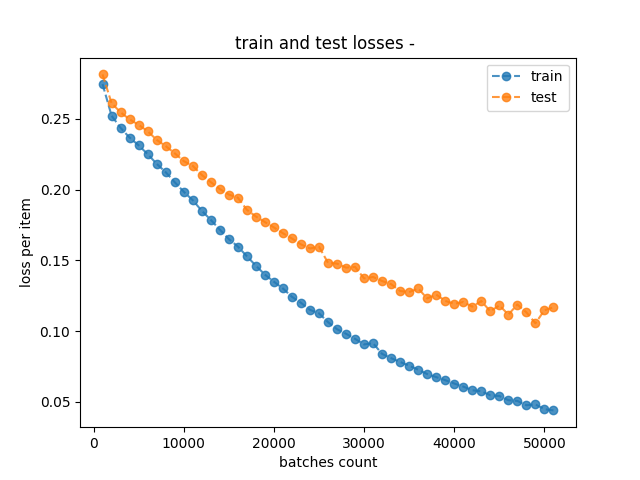


We also tried a slightly higher learning rate – 0.002:



Here there is a stronger fit – the train loss is better than the test loss, but also the test loss and the other metrics on the test set are better so we will go with this learning rate.

That’s how the learning looks with learning rate of 0.005:



Here the test loss converges faster, so we will go with this learning rate.

**Learning rate = 0.005**

Epochs number

According to last learning curve 70 epochs seems like a good choice, about where the test loss converges.

**Epochs number = 70**

Activation function

We will try four different activation functions for the hidden layer:

* Sigmoid
* Relu
* LeakyRelu
* Tanh

We will compare them by test accuracy as instructed in the exercise definition.

The results:

|  |  |
| --- | --- |
| **activation function** | **accuracy** |
| Sigmoid | 0.903 |
| Relu | 0.967 |
| LeakyReLU | **0.970** |
| Tanh | 0.915 |

We can see that LeakyRelu gives us the best accuracy so we chose it.

**Activation function =** **LeakyReLU**

Architectures Comparison

After choosing the basic hyper-parameters, we searched the best network architecture. The parameters are number of layers and number of neurons in each layer, and of course, they are dependent so we searched for both of them together.

We tried:

Linear layer with sigmoid activation (kind of logistic regression) – **Zero** hidden layers

Shallow network – **One** hidden layer of sizes: 32, 64, 128, 256, 512, 1024, 2048

Deep network – **Two** hidden layers from same size each: 16, 32, 64, 128, 256, 512, 1024

Very deep network – **Five** hidden layers with 128 neurons each.

For now, I follow the instructions and look at the accuracy (and not recall or precision).

We looked at the learning curves as sanity check and they were just fine, similar to first part the curve I showed before, just as expected.

The results are:

|  |  |  |
| --- | --- | --- |
| **# hidden layers** | **# neurons in each hidden layer** | **accuracy** |
| 0 | - | TODO |
| 1 | 32 |  |
| 1 | 64 |  |
| 1 | 128 |  |
| 1 | 256 |  |
| 1 | 512 |  |
| 1 | 1024 |  |
| 1 | 2048 |  |
| 2 | 16 |  |
| 2 | 32 |  |
| 2 | 64 |  |
| 2 | 128 |  |
| 2 | 256 |  |
| 2 | 512 |  |
| 2 | 1024 |  |
| 5 | 128 |  |

[TODO]

For each positive number of hidden layers, I plotted the results as function of the number of neurons in each layer.

[TODO]

[TODO]

Therefore, the chosen architecture is:

**Number of layers = [TODO]**

**Neurons in each layer = [TODO]**

# Problematic Experiments

# First Training

For the first training, we tried feed forward network with one hidden layer with 128 neurons with relu activation. We trained the network for 5 epochs. We used batch size of one for simplicity.

I used binary cross entropy loss and learning rate of 0.001.

Results on the test set:

Accuracy: 0.917

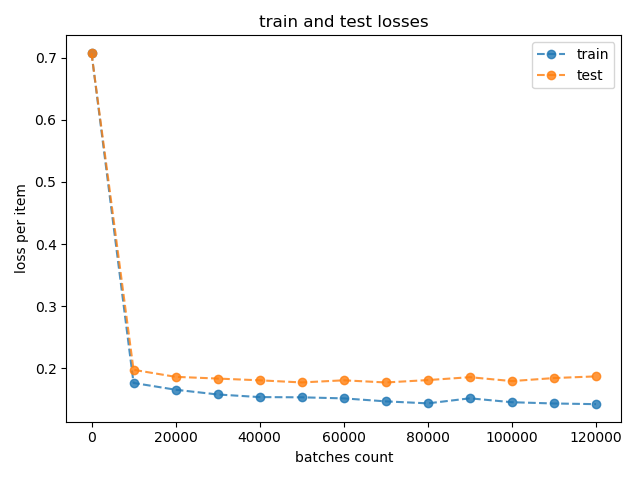
Recall: 0.968

Precision: 0.941

f1: 0.954

We can see that the results are fine – all of the metrics I checked are above 90%. Especially it is good to see that both the recall and the precision are pretty high even though the data set is imbalanced.

Those are the learning curves:



This was sanity check and now we can go forward to hyper-parameters tuning.

# Hyper-Parameters Tuning

There are many architectural choices and parameters to tune when training a neural net. We decided to split them to two groups: "basic" hyper-parameters and architectural parameters.

Basic hyper-parameters

* Learning rate
* Batch size
* Epochs number
* Activation function

Architectural parameters

* Number of layers
* Number of neurons in each layer

We will start by choosing the basic hyper-parameters with an arbitrary architecture, and once find a good configuration test the different architectural parameters with this configuration.

## "Basic" Hyper-Parameters Tuning

The basic architecture I will try the parameters with is multi-layered perceptron with one hidden layer of 256 neurons. The loss function is binary cross-entropy.

Important note: The results showed here are not always statistically significant, but because this is an exercise, we focused on exploration and trying a lot of parameters and not on being sure our parameters are the best.

### Batch size

Usually choosing batch size has less to do with model quality and more with performance issues so we just tried some batch sizes and chose 64, which gave fine results in short training time. There is a connection between batch size and learning rate but once we set a fixed batch size, we can just choose the proper learning rate.

**Batch size=64**

### Learning rate

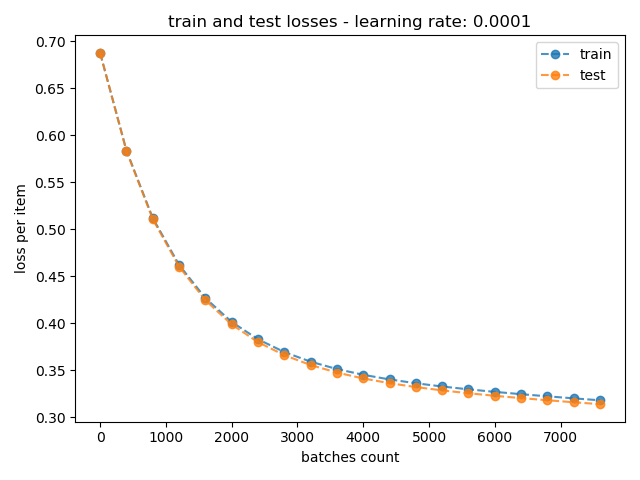
We trained the network with different learning rates, from 0.0001 to 0.1.

The full list:

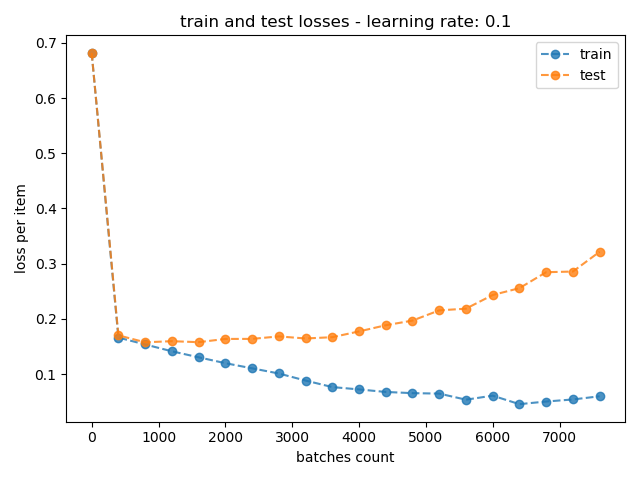
learning\_rates = [10 \*\* (-4), 5 \* 10 \*\* (-4), 10 \*\* (-3), 5 \* 10 \*\* (-3), 10 \*\* (-2), 5 \* 10 \*\* (-2), 10 \*\* (-1)]

Let's take a look in some learning curves to choose a good value:

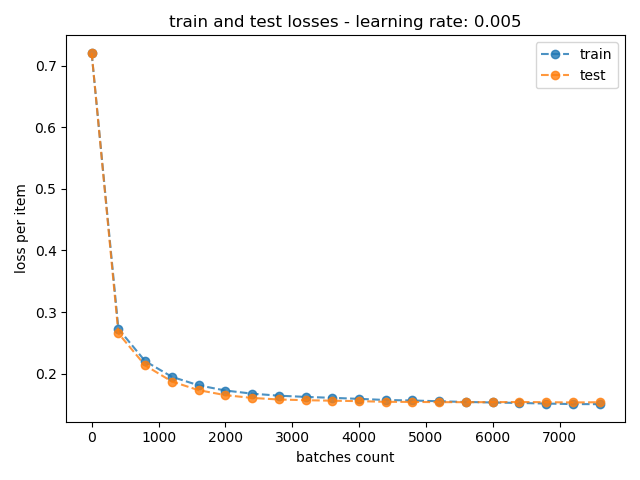
When we have a very small learning rate (0.0001), we have stable but slow learning:



When we have a very big learning rate, (0.1) the train loss goes down but the test loss goes up – overfit.



With a medium learning rate (0.005), we get fast convergence as wanted:

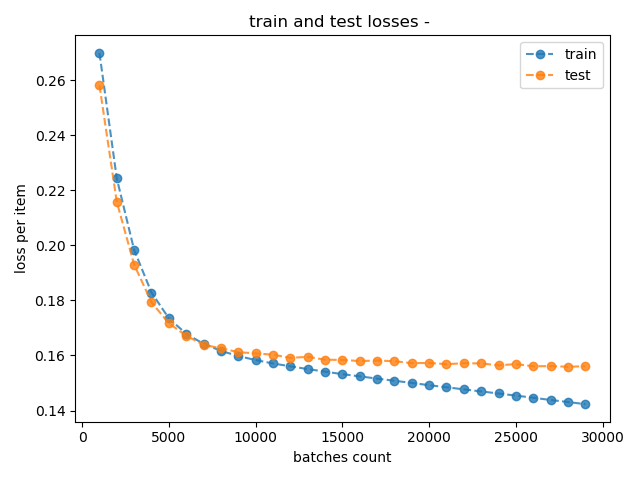


We will go on the safe side and choose a slightly smaller learning rate of 0.002 to avoid problems later.

**Learning rate = 0.002**

### Epochs number

We trained the network for 75 epochs with the chosen params and got this learning curve:



The test loss does not get much better from the 10,000 batch, which corresponds to about 25 epochs. From this point we get to a kind of overfit – the test loss does not get worse but the train loss goes to zero without real improvement of the model.

**Epochs number = 25**

### Activation function

We will try four different activation functions for the hidden layer:

* Sigmoid
* Relu
* LeakyRelu
* Tanh

We will compare them by test accuracy as instructed in the exercise definition.

The results:

|  |  |
| --- | --- |
| **activation function** | **accuracy** |
| Sigmoid | 0.907 |
| **Relu** | **0.930** |
| LeakyRelu | 0.884 |
| Tanh | 0.907 |

We can see that Relu gives us the best accuracy so we chose it.

**Activation function = Relu**

## Architectures Comparison

After choosing the basic hyper-parameters, we searched the best network architecture. The parameters are number of layers and number of neurons in each layer, and of course, they are dependent so we searched for both of them together.

We tried:

Linear layer with sigmoid activation (kind of logistic regression) – **Zero** hidden layers

Shallow network – **One** hidden layer of sizes: 32, 64, 128, 256, 512, 1024, 2048

Deep network – **Two** hidden layers from same size each: 16, 32, 64, 128, 256, 512, 1024

Very deep network – **Five** hidden layers with 128 neurons each.

For now, I follow the instructions and look at the accuracy (and not recall or precision).

We looked at the learning curves as sanity check and they were just fine, similar to first part the curve I showed before, just as expected.

The results are:

|  |  |  |
| --- | --- | --- |
| **# hidden layers** | **# neurons in each hidden layer** | **accuracy** |
| 0 | - | 93.0% |
| 1 | 32 | 83.7% |
| 1 | 64 | 95.3% |
| 1 | 128 | 83.7% |
| 1 | 256 | 95.3% |
| 1 | 512 | 93.0% |
| 1 | 1024 | 97.7% |
| 1 | 2048 | 90.7% |
| 2 | 16 | 95.3% |
| 2 | 32 | 97.7% |
| 2 | 64 | 90.7% |
| 2 | 128 | 93.0% |
| 2 | 256 | 95.3% |
| 2 | 512 | 93.0% |
| 2 | 1024 | 95.3% |
| 5 | 128 | 97.7% |

We can see big differences – the accuracies goes from ~84% to ~98% (8 times less errors). In general we see that the deeper the net – the better the accuracy.

For each positive number of hidden layers I plotted the results as function of the number of neurons in each layer.

We did not connect the dots with a line because we think that what we see here is mainly statistical noise. We think that because there is no clear trend and because when we notices that when we run the same training several times we sometimes get different results. However, when we look at the average accuracy for each network depth we can see a trend – the deeper, the better.

Therefore, we decided to check this trend. We trained 3 different models – three MLPs with 256 neurons in each layer, but with different depths: 2, 5, 10. Each model was trained three times to make the results more reliable.

The results:

|  |  |
| --- | --- |
| Network depth | Mean accuarcy |
| 2 | 93.8% |
| **5** | **96.9%** |
| 10 | 85.3% |

Therefore the chosen architecture is:

**Number of layers = 5**

**Neurons in each layer = 256**