DL – Exercise 1 – Report

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

Eldan Chodorov
ID: 201335965

Amit Keinan
ID: 208296426

November 2021

Table of Contents

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 misled 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 experiments. We left here (in the end under "Appendix – 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

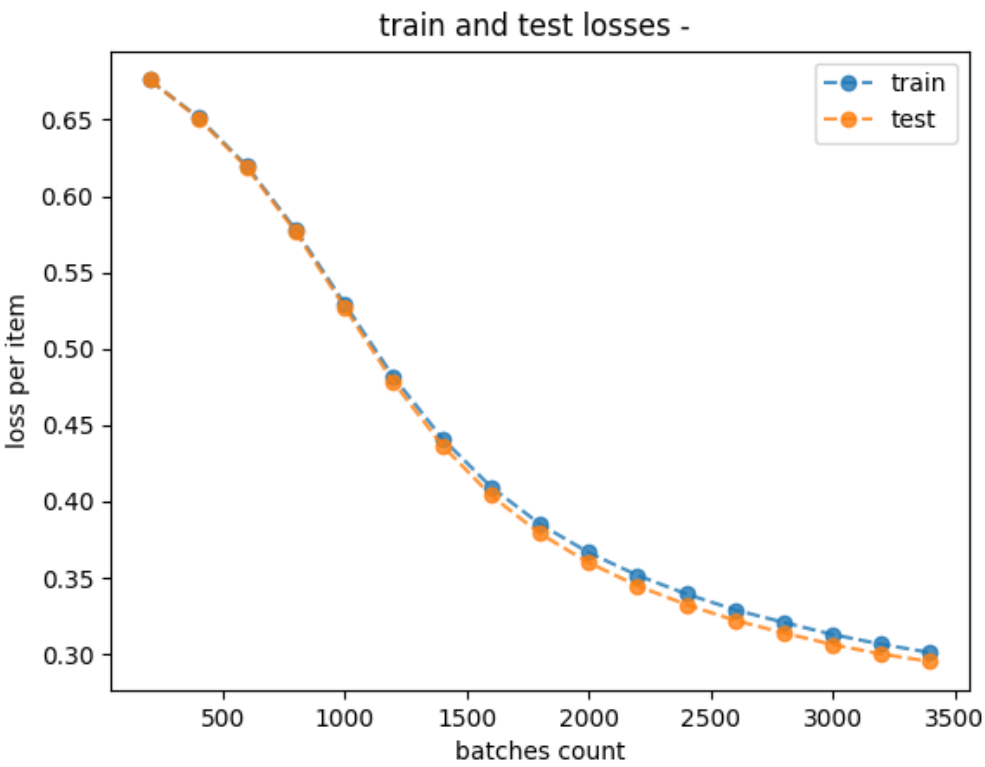
precision: 0.925

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.

<u>Basic hyper-parameters</u>

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

<u>Architectural parameters</u>

- 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

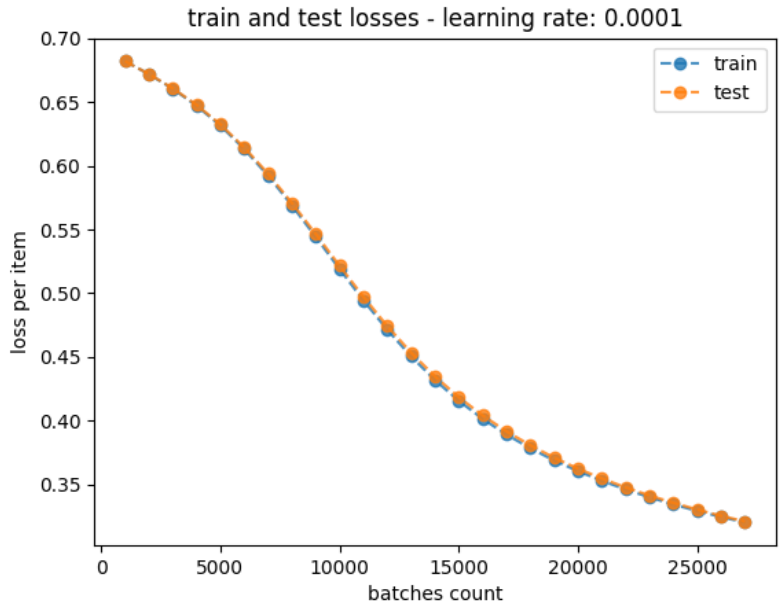
Usually choosing batch size has less to do with model quality and more with performance issues so we just tried some batch sizes and chose 64, which gave fine results in short training time. There is a connection between batch size and learning rate but once we set a fixed batch size, we can just choose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

Batch size=64

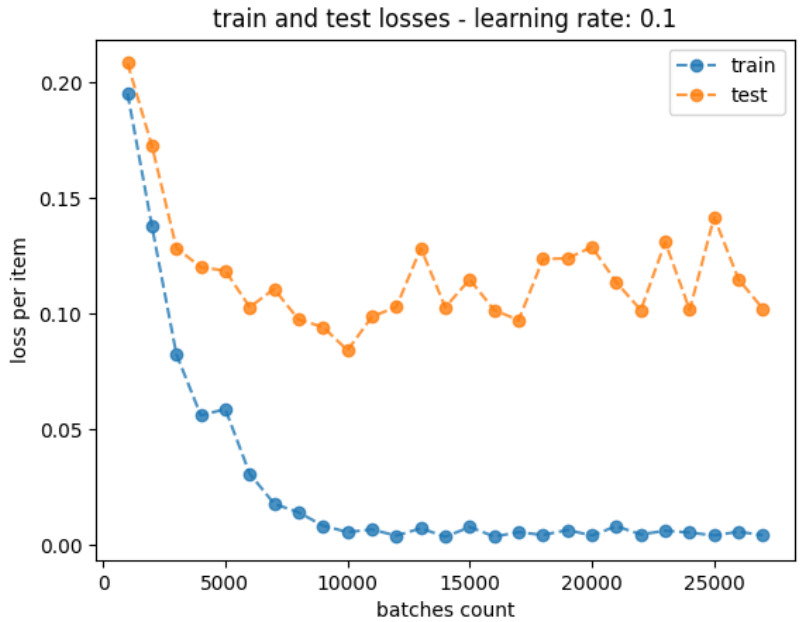
Learning rate

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:

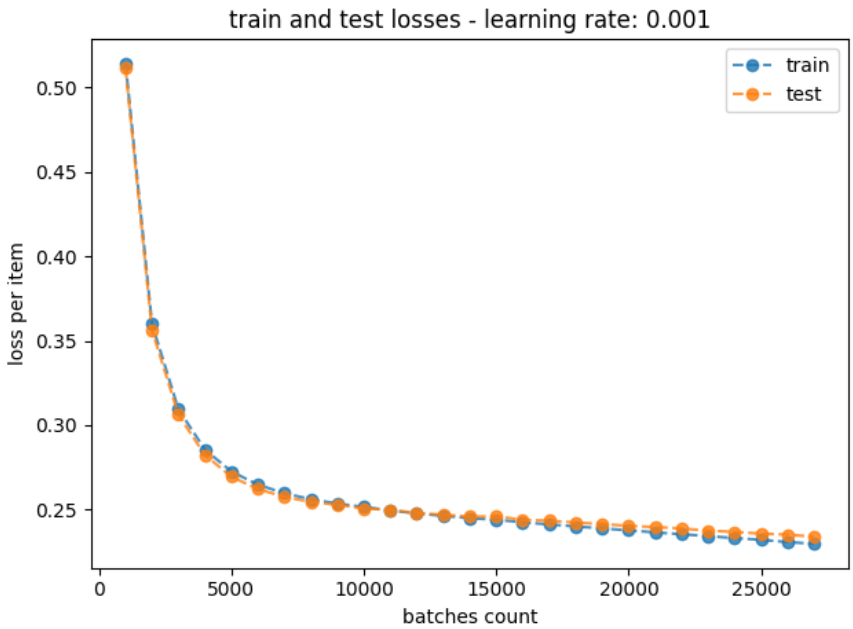
When we have a very small learning rate (0.0001), we have stable but slow learning:



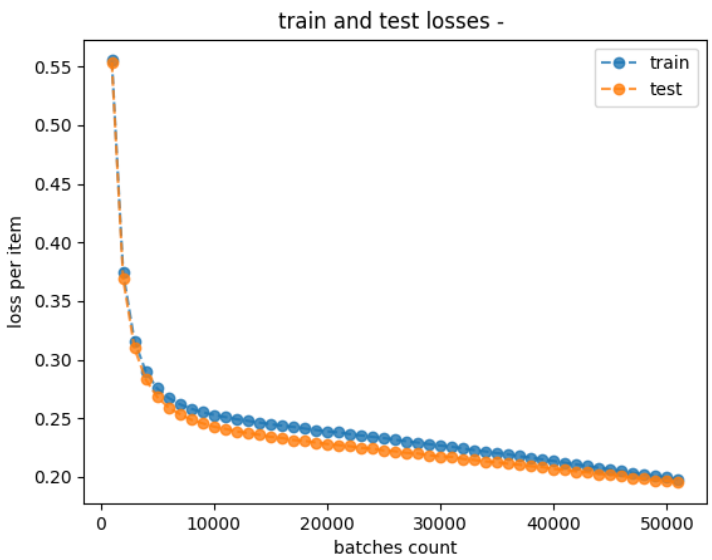
When we have a very big learning rate, (0.1) the train loss goes down but the test loss does not – overfit.



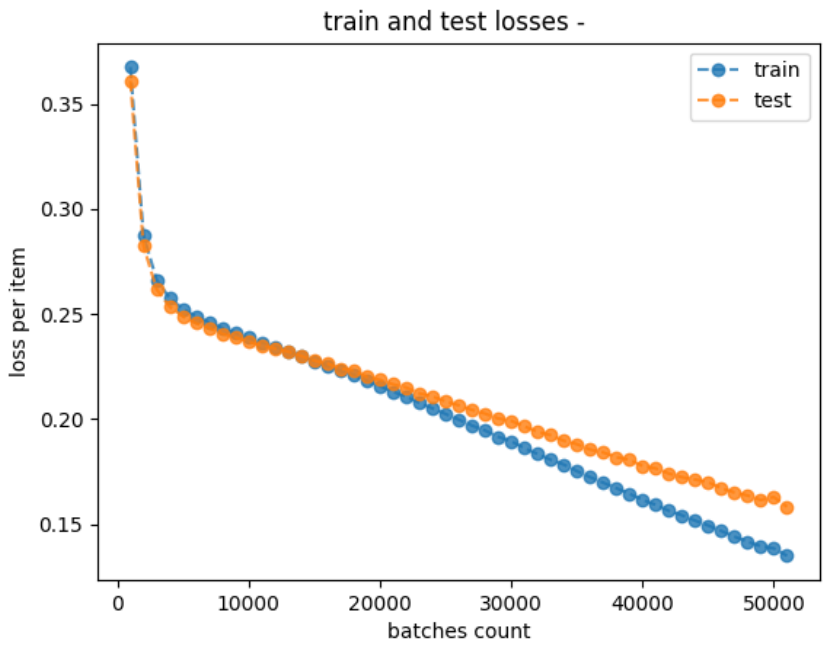
With a medium learning rate (0.001), we get fast convergence as wanted:



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.

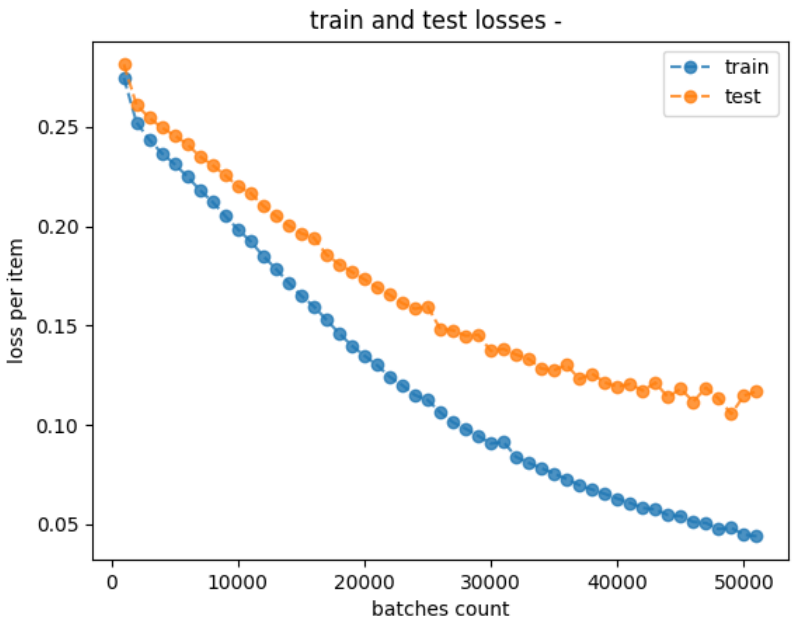


We also tried a slightly higher learning rate – 0.002:



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	-	90.0%
1	32	92.4%
1	64	93.2%
1	128	92.4%
1	256	91.6%
1	512	91.4%
1	1024	91.1%
1	2048	90.6%
2	16	95.8%
2	32	96.4%
2	64	97.2%
2	128	97.8%
2	256	97.4%
2	512	97.0%
2	1024	97.6%
5	128	97.4%

We can see that two hidden layers models are significantly better than one-layer models which are better than zero hidden layers models. We can also see that the number of neurons in each layer (width) does not make a dramatic difference in the accuracy.

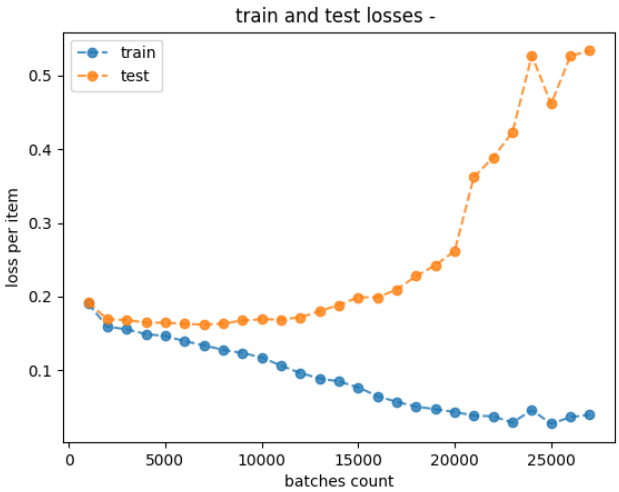
Therefore, the chosen architecture is:

Number of hidden layers = 2

Neurons in each layer = 128

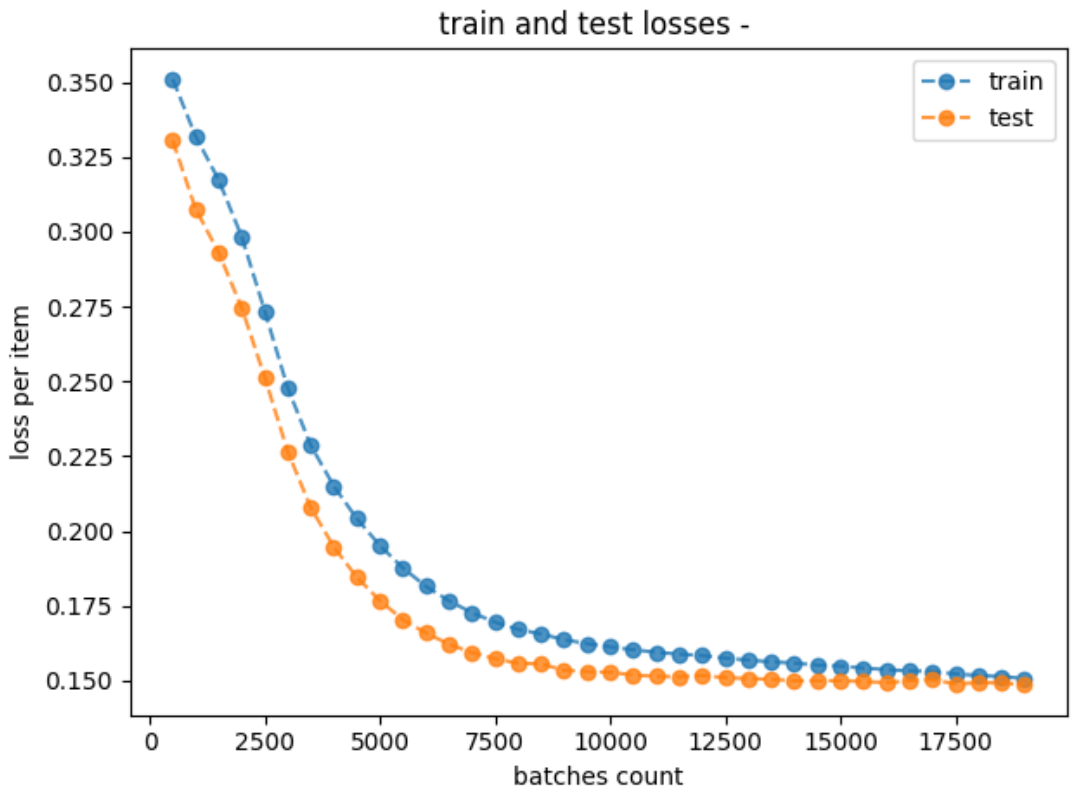
Adjusting learning rate and epochs number

When we look in the learning curves, we see that we get overfit:



That's because we tuned each parameter as it is independent of the others, and this assumption is not true. The best way is to try some kind of grid search, but this is really expensive in computation time. For now, we will just re-tune the learning rate and epochs number to prevent overfit.

We found out that training the network with smaller learning (0.001) rate and less epochs (50) gives good learning:



We even can train for less epochs and get close results so we will go with 40 epochs.

To sum up, the chosen params are:

- Batch size=64
- Learning rate = 0.001
- Epochs number = 40
- Activation function = LeakyReLU
- Number of hidden layers = 2
- Neurons in each layer = 128

Dealing With Imbalanced Data

In the previous sections we dealt with the imbalanced data with oversampling. In this section we will improve our pipeline and measurement and focus specifically on this problem.

Evaluation

The first step in dealing with imbalanced data situation, just like most machine learning situations, is defining a proper metric. At first, we did not do it (you can see it in the appendix) and did not understand out model is doing a very bad job.

First of all, we will oversample our data better – we will oversample the negative samples only in the train set and leave the test set as is. Next, we will separate the single accuracy metric to two – one only for the positive samples, and one only for the negative ones.

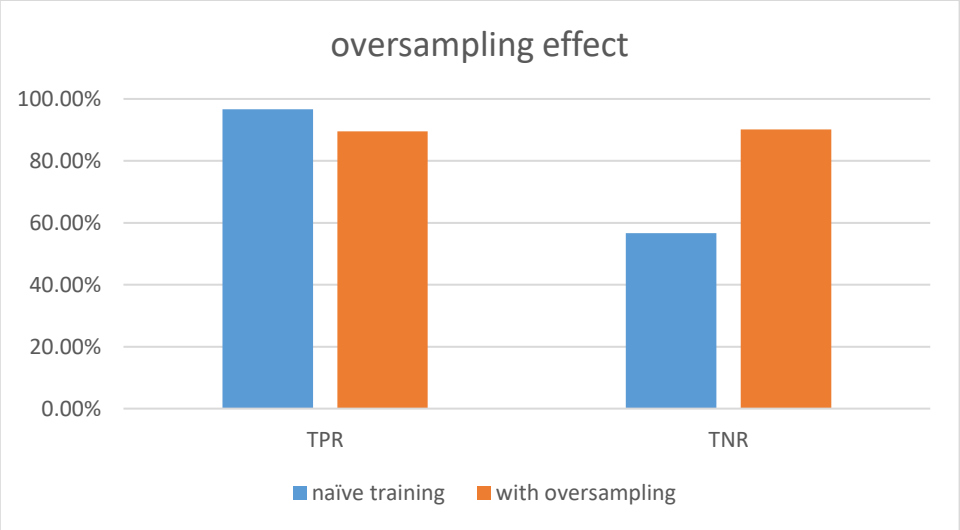
$$True\ Positive\ Ratio = Recall = \frac{TP}{P}$$

$$True\ Negative\ Ratio = \frac{TN}{N}$$

Those metrics are agnostic to the positive-negative ratio in the data.

Oversampling

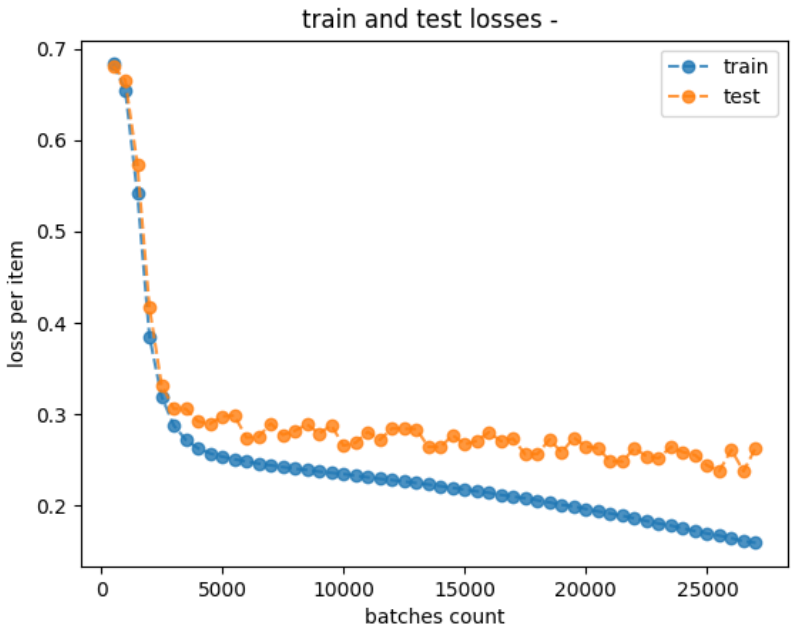
We used a simple oversample technique. We calculate the ratio between the positive samples and the negative samples in the train set (it is about 8.23, the train-test split is random so it changes a bit each run), and then duplicate each negative sample for int(ratio) times (usually 8). That way the loss function will be affected also by the negative samples, and the positive samples won't take over so easily.



In the plot above we can see comparison of two trainings – one with oversampling and one without. We can see that the oversampling improves the prediction of the negative samples dramatically (about 4 times less mistakes), and hurt the TPR a little bit. Overall, it's worth it.

Final training

We chose all of the hyper-params and decided how to deal with imbalanced data. Now it is time to show the final learning curve and metrics.



The naïve reader may think that the test loss does not go down fast enough, but the wise reader will remember that the test loss does not say a lot because it is not oversampled, so it gives far much more weight to performance on positive samples.

	test	Train – **oversampled**
TPR	0.89	0.90
TNR	0.86	0.99
accuracy	0.89	0.94

The metrics are fine – pretty high results on the test set. An interesting observation is that we got overfit on the negative samples because they were aggressively over sampled. Maybe it was better to try a lower over sampling factor or try other methods (under sampling, oversampling + dropout, ect.).

Inference on Spike Protein

We used our trained model to identify sequences in the Spike protein. The protein is from length 1273, and therefore we have 1265 peptides (1273 – 9 + 1).

Most of the predictions were 1171 positive predictions out of 1265 predictions, which is about 92.6%.

The most detectable peptides are:

peptide	score
VYYPDKVFR	1.0
VYADSFVIR	1.0
RVDFCGKGY	1.0
ERDISTEIY	~1.0 (9.999…)
KTSVDCTMY	~1.0 (9.999…)

Of course, we first notified the CDC 😊

Optimize Input

In the last section we optimized the input with respect to the pretrained model. It turns out that our model gives high scores to random values, so the loss was usually pretty low from the beginning.

To convert the arbitrary tensor to peptide we took the argmax of each 20-sized segment of the tensor and took the matching letter of the peptide. (kind of inverse one-hot encoding)

We got different results each run. For example:

ADEVEQSWV

Because the model tends to give high scores to random values, we tried to change the mean and the std of the normal distribution, but it didn't help much. So, in order to make sure the process works properly we tried to do the opposite – find the least detectable peptide. That way we started with high loss and watched it goes down while the input changes. A very undetectable peptide is:

NVEAIHYSA

Appendix – Problematic Experiments

In this appendix we show the first time we wrote this report – with bad models that only output 1…

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

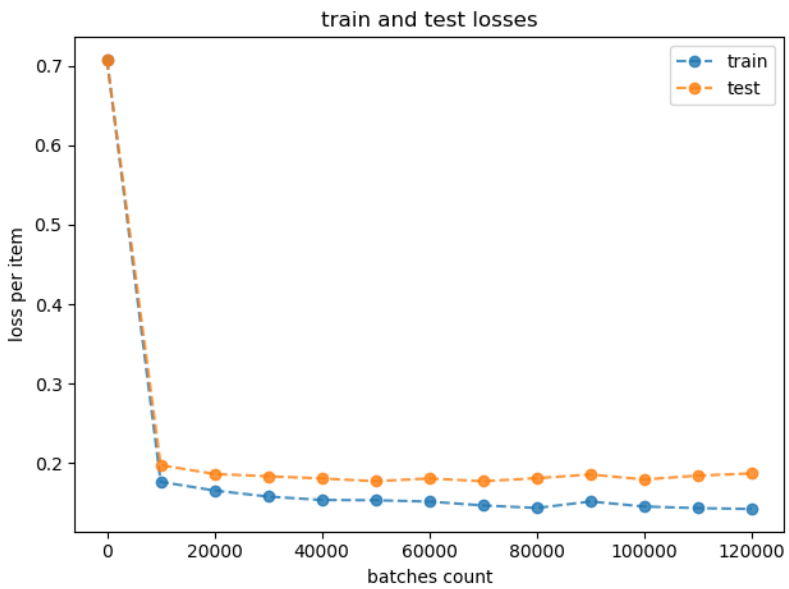
Recall: 0.968

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.

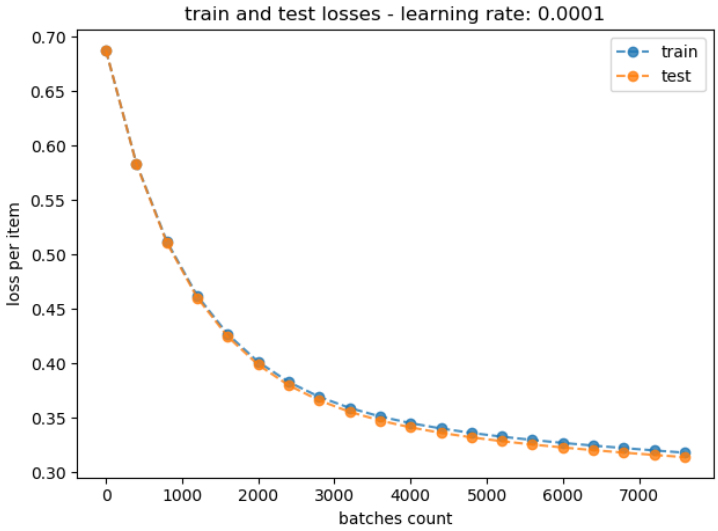
Batch size

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.

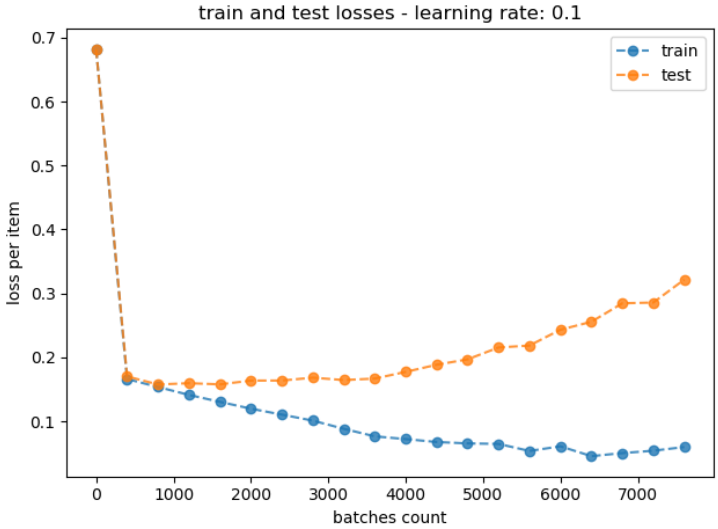
Batch size=64

Learning rate

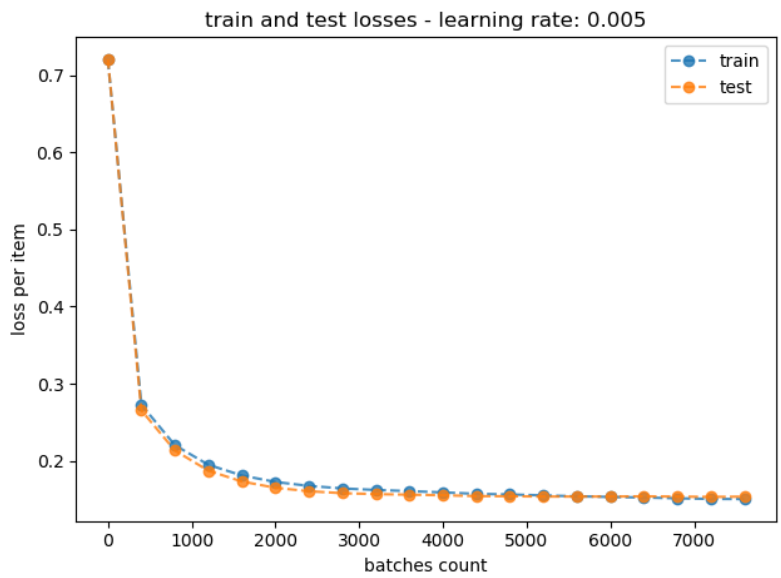
We trained the network with different learning rates, from 0.0001 to 0.1.
The full list:
learning_rates = [10 ** (-4), 5 * 10 ** (-4), 10 ** (-3), 5 * 10 ** (-3), 10 ** (-2), 5 * 10 ** (-2), 10 ** (-1)]
Let's take a look in some learning curves to choose a good value:
When we have a very small learning 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:

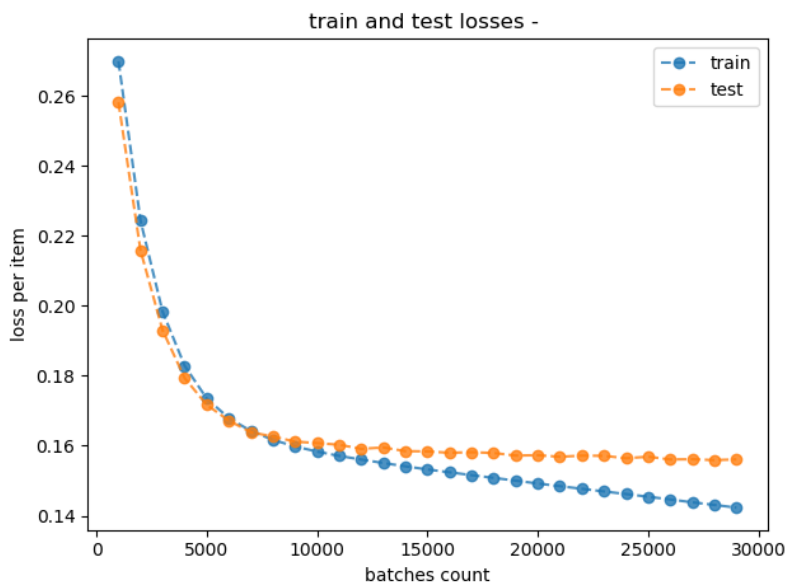


We will go on the safe side and choose a slightly smaller learning rate of 0.002 to avoid problems later.

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.

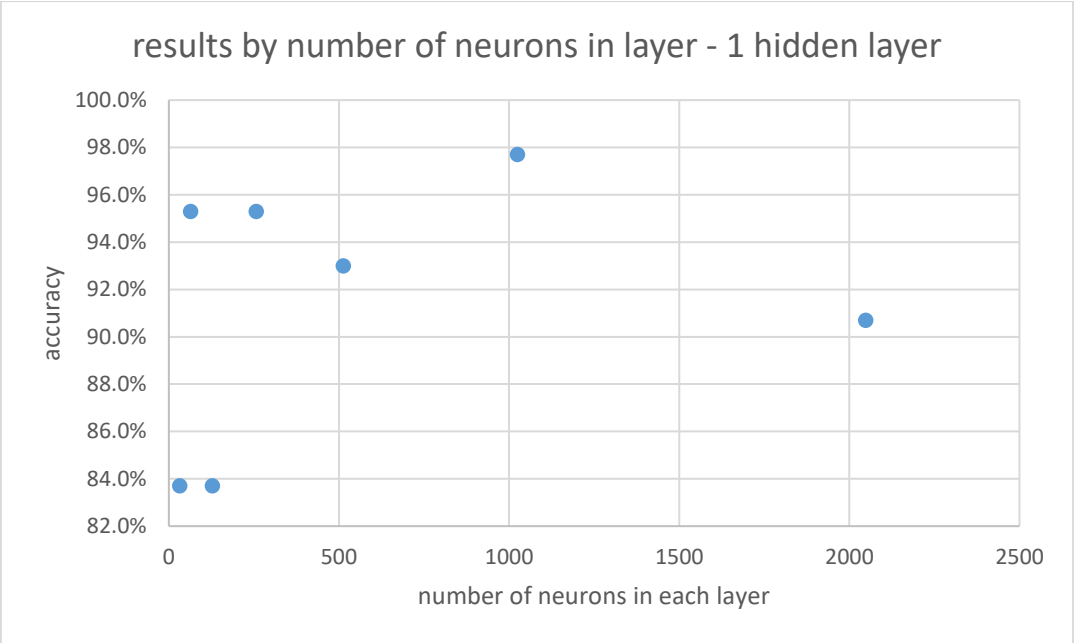
The results are:

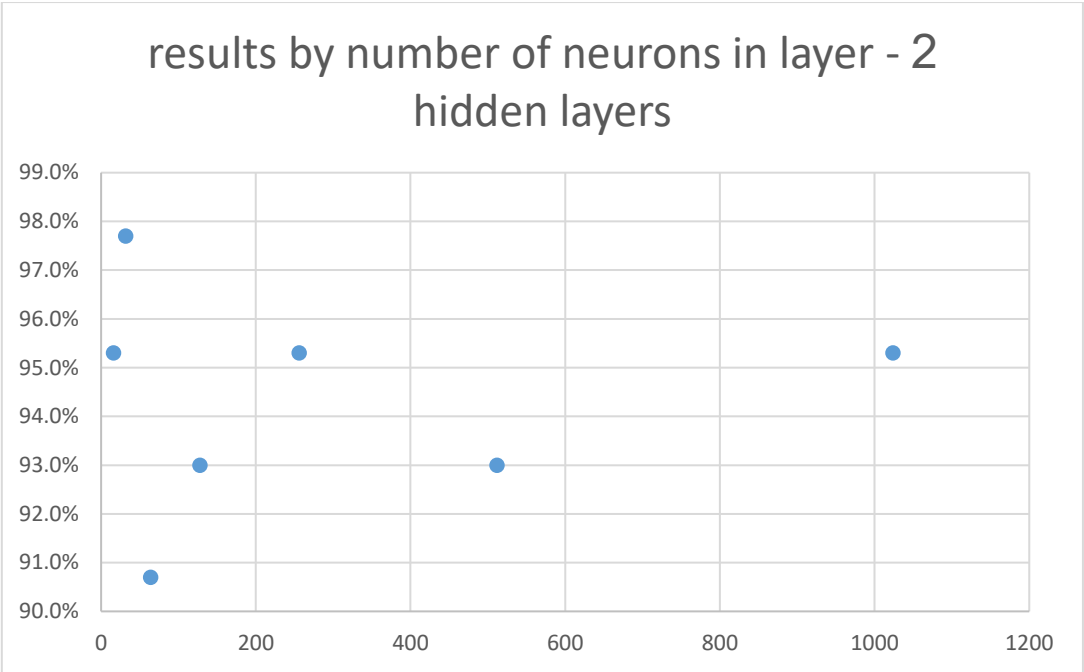
# hidden layers	# neurons in each hidden layer	accuracy

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