UNIVERSITÉ CÔTE D'AZUR

MSc. Data Science & Artificial Intelligence

Advanced Deep Learning

Dr Alessandro BETTI

Assignment 1 & 2

Author: Joris LIMONIER

joris.limonier@gmail.com

Due: November 4, 2022

Contents

List of Figures

List of Tables

1 Assignment 1

1.1 Exercise 1

The code was completed in the Python file.

1.2 Exercise 2

1.2.1 Question 1

I was not familiar with the PyTorch library, so I had to perform some research in order to know how its methods/classes work. I consulted several sources, including the following, which were useful in order to answer this question:

1. https://discuss.pytorch.org/t/how-sgd-works-in-pytorch/8060/2
2. https://discuss.pytorch.org/t/performing-mini-batch-gradient-descent-or-stochastic-gradient-descent-on-a-mini-batch/21235
3. https://pytorch.org/docs/stable/generated/torch.optim.SGD.html

According to those sources, PyTorch's SGD actually computes a full-batch (vanilla) Gradient Descent, based on the data that is passed to it. It is my understanding that in order to perform actual mini-batch (*i.e.* where 1 < batch size < number of observations), one simply needs to give subsets of the data a each iteration.
In our case, we use the full dataset in `outputs = net(inputs)`, which is why we perform full-batch GD, although we call the `optim.SGD` class.

1.2.2 Question 2

Figures 1, 2 and 3 show a comparison between train and test cross entropy loss. The leftmost diagrams have a linear scale for the vertical axis, while the rightmost diagrams have a logarithmic scale for the vertical axis. Beware that the number of epochs varies significantly between figures. Indeed, more pararmeters in the network implies longer training times per epoch, which in turn means that running for the same time both experiments results in different number of epochs.

We see that in both experiments the train error keeps decreasing, while the test error

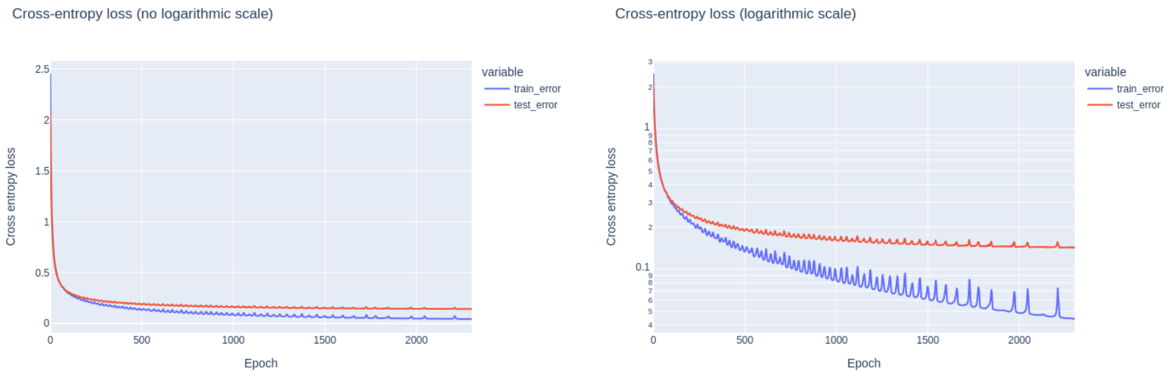

Figure 1: Cross entropy loss, train vs test (300 fully-connected)

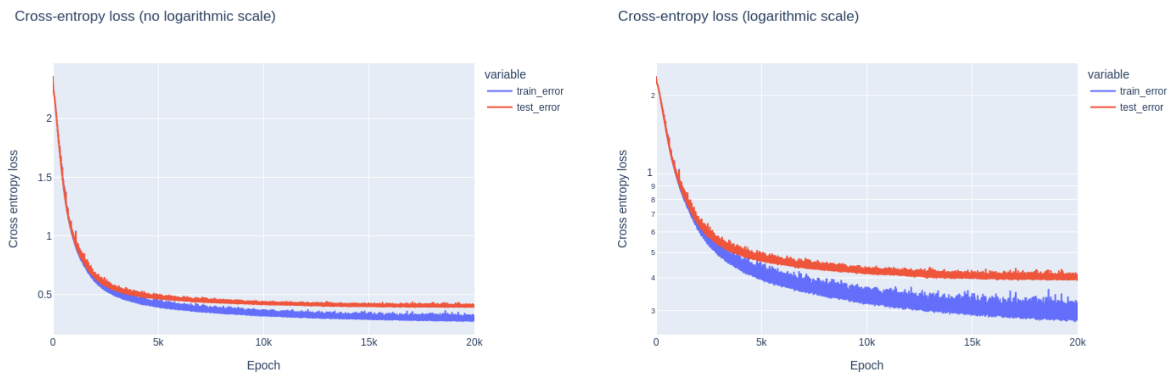

Figure 2: Cross entropy loss, train vs test ($10-10$ fully-connected)

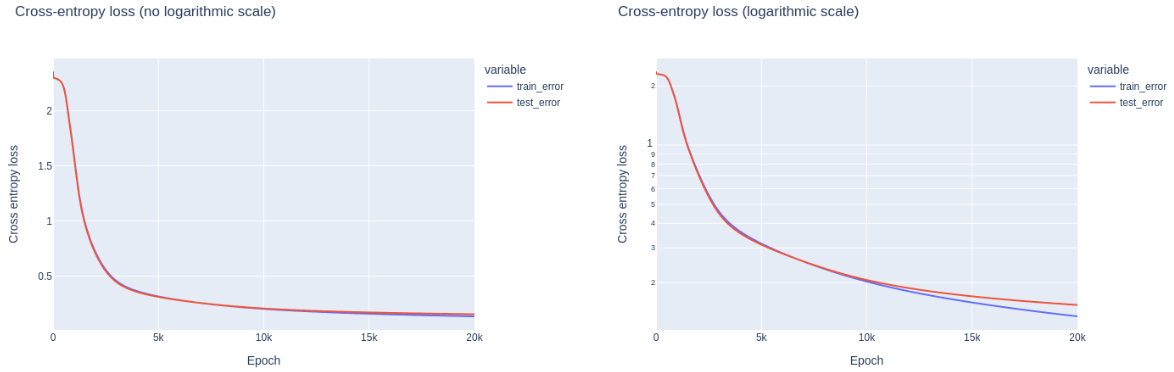

Figure 3: Cross entropy loss, train vs test ($20-20$ fully-connected)

stays constant. This means that our network is overfitting and there is no need to continue the experiment further. One way to prevent wasting time training when the network isn't making actual progress is to print the loss, or even to plot the loss curves, every couple iterations. We estimate however that the over-fitting phenomenon isn't too extreme in our case.

1.2.3 Question 3

Let $\beta : \mathbb{N}^{l+1} \to \mathbb{N}$ denote the number of parameters in a fully connected neural network. Then β is given by:

$$\beta(h_0, \ldots, h_l) := \sum_{i=1}^{l} (h_{i-1} * h_i) + h_i \tag{1}$$

Indeed, the multiplication term "$h_{i-1} * h_i$" accounts for the weights by counting the connections from a layer to the next. The last term "h_i" is responsible for counting the biases on each of the hidden layers, as well as the output layer.
We compute β for a few network architectures we used, with $h_0 = 28 * 28 = 784$ in all

cases, since this is the size of the input:

$$\begin{aligned}\beta(784,10,10,10) &= (784*10+10)+(10*10+10)+(10*10+10)\\ &= 7850+110+110\\ &= 8070\end{aligned}$$

$$\begin{aligned}\beta(784,20,20,10) &= (784*20+20)+(20*20+20)+(20*10+10)\\ &= 15700+420+210\\ &= 16330\end{aligned}$$

$$\begin{aligned}\beta(784,300,10) &= (784*300+300)+(300*10+10)\\ &= 235500+3010\\ &= 238510\end{aligned}$$

However, we know that not all parameters are created equal. Indeed, adding more layers includes non-linearities in the network. Increasing the number of layers is useful because it "bends" the space in which the data lives, which allows to find more patterns. On the other hand, adding layers causes other issues, namely the vanishing gradient problem: as the gradient goes through multiple activation functions, it shrinks until making close to no update in the weight. The argument for less layers is also strengthened by Cybenko's *Universal approximation theorem* (1989), which proves that a unique hidden layer is sufficient to approximate any *reasonable* function with arbitrary precision. However, it may require the hidden layer to be of exponential size with respect to the desired precision.
The bottom line to this reflexion is as follows: increasing the size of the layers is good because the network can learn more, however there are risks of overfitting. Increasing the number of layers is also an option, but it can cause overfitting and gradients may vanish. Most certainly some trial and error is required, with close baby-sitting of training and validation errors in order to adapt the architecture.

1.2.4 Question 4

In PyTorch, with NLLLoss meaning Negative Log-Likelihood Loss:

$$CrossEntropyLoss = SoftMax \circ NLLLoss \tag{2}$$

The $NLLLoss$ part isn't too much of interest as it simply outputs the most likely class. The SoftMax function however is more interesting. Indeed, it acts as a normalized exponential and therefore outputs probabilities. This is not true with quadratic loss that can output any non-negative number.

1.2.5 Question 5

Using a single output means that the network give us one definitive answer and we have no idea of how certain it is. One hot encoding on the other hand provides probabilities of how sure it is when it makes predictions. This is useful because a network should be penalized equally for different degrees of certainty. Ideally, we want the network to be certain of its prediction and the prediction to be right. If the network is certain of a wrong prediction, we should penalize it a lot. We would prefer a network that makes a wrong prediction but is unsure. This latter network would take less weights adjustments in order to make the prediction right.

1.2.6 Question 6

For this question, we will keep the architecture of a single 300-neurons hidden layer and we will monitor the loss with various learning rates.

Let us call the learning rate η. We already tested the case $\eta = 0.1$ in figure 1. Table 1 gives a summary of associated figures for various values of η. The figures in table 1 are located in appendix A.

Learning rate (η)	Figure number
0.001	5
0.1	1
0.3	6
0.5	7
0.7	8
0.9	9
1.2	10
2	11
5	12
8	13

Table 1: Figure number per learning rate

Result analysis It is my understanding that learning rates are often of order $\eta = 10^{-k}$ for $k = 1, \dots, 4$. In this case however, this is not what I found at all. Indeed, we see that learning rates $\eta \in \{0.001, 0.1, 0.3, 0.5, 0.7, 0.9, 1.2, 2\}$ converge nicely even though converging rates vary. For instance, $\eta = 2$ converges faster than lower values. One may be tempted to increase η even further. On the other hand, we see that larger learning rates such as $\eta \in \{5, 8\}$ result in instabilities, but $\eta = 5$ still reaches good (low) values of cross entropy loss.

We didn't try every possible value of η but it seems that somewhere between $\eta = 2$ and $\eta = 5$ could be a good candidate. Smaller values do indeed converge, but take many epochs (*i.e.* long times), while larger values tend to "overshoot" over minima from times to times, resulting in irregularities in their gradient descent procedure.

One thing worth noting is that we tested all learning rates over 2000 epochs. Such a number of epochs wasn't too long on my machine but also wasn't ridiculously low, which allows to reach decent loss values. We must note that we are probably putting low learning rates at a disadvantage because they most certainly converge as expected, but also take longer to do so (steps in the right direction, but small steps).

1.2.7 Question 7

The layer described in question 3 is a 300 neurons (single) hidden layer. As a result, only 2 gradients are computed (input to hidden and hidden to output). The vanishing gradient problem tends to occur on networks with more hidden layers. In such cases, gradients have to go through activation functions several times. Doing so squishes them between 0 and 1 (for many activation functions, *e.g.* Sigmoidal, SoftMax) multiple times. As a result, gradients end up going to 0 and eventually make no update to the weights anymore.
Several solutions can be implemented in order to solve the vanishing gradient problem. Here are some of them:

- The simplest, most radical way to solve the vanishing gradient problem is to remove hidden layers. This removes some of the activation functions that are applied to the gradient.

- On complex problems however, removing hidden layers may not be an option because there is may be more information to discover in the data. In such cases, one may use ReLU as an activation function, where ReLU is defined as follows:

  $$ReLU(x) := \max(0, x)$$

  ReLU is faster than many other activation functions because it has less computations to make (computing exponentials, summing, dividing, ...). ReLU leaves positive gradients unchanged and sets negative ones to 0, in particular it doesn't squish gradients. This solves the vanishing gradient problem in most cases.
  Although ReLU often solves the vanishing gradient problem, it also comes with its downsides, namely making gradients explode.

- Another way to solve the vanishing gradient problem is to use residuals. Indeed, by randomly setting some weights in the network to 0, the network "learns to use alternative routes" (grossely speaking). This also tends to fix the vanishing gradient problem.

2 Assignment 2

I will start fresh for assignment 2 and reimplement some of the functions.

2.1 Exercise 1

Please allow us to give an overall feeling from the experiments I have performed, rather than reporting for each learning rate, for each batch size the result after a given number of epochs.
From the experiments conducted, we notice a few things. We will explain each of them.

Epochs take much longer as the batch size decreases This makes sense. As the batch size decreases, we have to go over more batches to complete one epoch. For each of these batches, we must compute the loss, compute the gradient and update the weights. As a result, there are more updates per epoch, which makes each epoch longer. This doesn't really matter though because instead of looking at the time per epoch, a more interesting metric would be the time per update. Although the time per epoch increases, there are actually $len(X\_train)/batch\_size$ updates per epoch, and each of these updates is faster than if the network had had to compute the gradients over the whole dataset.

High learning rates are not acceptable anymore This also makes sense. Consider the full-batch case (previously). Since the gradient was computed over the whole dataset, this was a (very long, but) very carefully chosen descent direction. As a result, it was acceptable to make a big step. In the minibatch setting however, the steps are way faster, but a bit more imprecise. For this reason, it is not wise anymore to take large steps anymore. Nevertheless, this option is still very interesting because we can take "mostly correct" steps very fast and make fast progress towards the minimum.

Minibatch is still the best method overall As mentioned previously, epochs do take longer and steps sizes must be reduced. We showed however that it doesn't matter because we make much more frequent updates, which are "mostly in the right direction". For this reason and according to our experiments, minibatch converges much faster than full batch. After only a couple epochs, we are already in a better place than after hundreds of epochs in the full batch case, which shows very interesting results in favor of mini batch.

2.2 Exercise 2

In the case where all parameters are set to 0, the network doesn't learn anything. After performing some searches, this is called the symmetry break. I will try to display my understanding without copy-pasting my sources (1, 2).

When going from one layer to the next, what is applied is basically a given number of dot products. Let us consider the case of going from the input layer I (size 784) to the first hidden layer H_1 (size 300). Let us disregard the bias here because the explanation is the same, but it brings confusion into the dimensions. Then this transition is done by multiplying the input matrix of size (n_samples × 784) by a weight matrix W_1 of size (784 × 300). This gives a hidden layer of size (n_samples × 300). The matrix that has all weights 0 in this case is W_1. If $W_1 = 0 \in \mathcal{M}_{(784\times300)}$, then each input vector is put through a dot product with a vector of 0's to give the corresponding entry of the hidden layer. As a result, all values of the hidden layer are equal (in this case, equal to 0). Then they all go through the activation function and are propagated through the rest of the network. For gradient descent to "work" however, the hidden layer should not contain all same entries. If each unit in the hidden layer gets the same signal, it will not propagate varied, useful information through the network, but only limited, redundant information.

2.3 Exercise 3

We sort digits and verify that they are indeed ordered by computing labels for each training example. Then we plot the image and obtain figure 4. We see that as expected, the first layer (top of the image) contains all 0's, then come the 1's, then all the way until the 9's.

We understand that the sequence $e_n := z_{n \mod 60000}$ is not unique because for each given label, the samples can be shuffled to produce a new sequence. This new sequence still meets the property that any two labels 60000 positions apart still have the same labels. Let us called the dataset with labels in sorted order the "stratified" data set.

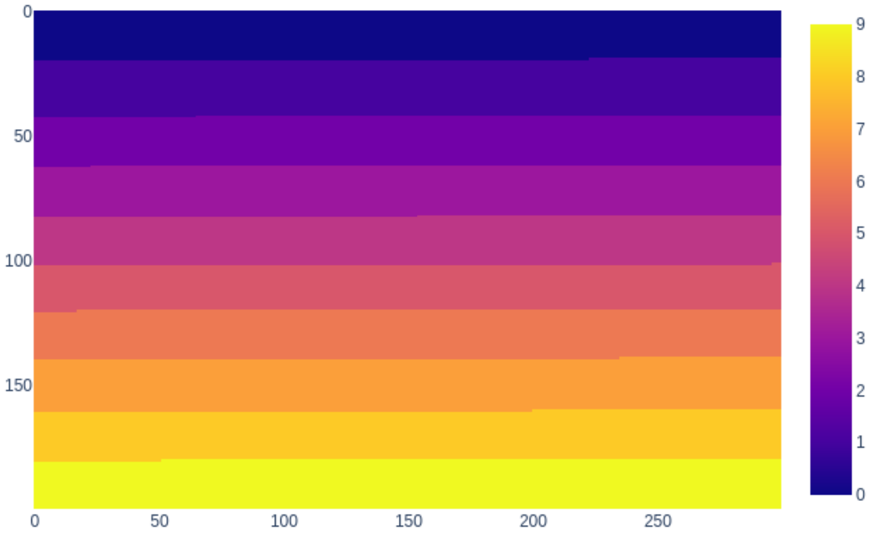

Figure 4: Image of the labels in the train dataset. One pixel represents one label.

2.3.1 Question 1

When training the network on the stratified dataset, the network learns on all zeros first, then all ones, ...etc. Compare with SGD with batch size 1 where the network also learns on one sample at a time. In the first case, the network has very similar samples by period. In the second case, the network sees a variety of examples, one at a time. In both cases, we feed examples in a somewhat randomly fashion, with the only guarantee that the expectation of this randomness is "close enough" to the true gradient. In the stratified case however, by feeding only identical digits for long periods of time, it is almost like we feed the same example of a zero many times, the the same example of a one many times, and so on. As a result, the steps made by the gradient are similar so it is almost as if we had a very stubborn network that took many steps "in the direction of a zero", then many steps in "the direction of a one", and so on.
We find that the network converges either very slowly, or not at all if the learning rate is not small enough. This is explained by the fact that the network make several steps almost in the same direction we presented with the same label multiple times. The stratified case shows all of the inconvenients of SGD with batch size 1, but pushed to the extreme. Very high learning rates like we had in the GD setting are not reasonable anymore. Learning rates on the order of 10^{-2} or even 10^{-3} are more appropriate for the stratified case.

2.3.2 Question 2

Because the diversity of examples shown to the network in SGD is much better than the one in the stratified case, the network is able to learn much faster. It also reaches better accuracies in a reasonable time. Maybe, if we let both the stratified and the simple SGD networks run for a long time (think "in the limit"), both networks would reach similar accuracies. Indeed, in the end we feed both networks the same information and in both cases we hope that the expectation value will bring us (although with some zig-zags due to randomness) to a minimum; ideally "global", most realistically "local". Putting it crudlely, one could say that the central limit theorem "takes longer" to converge because locally the stratified case is less representative of the actual distribution we are trying to approximate.

2.3.3 Question 3

The convergence analysis of SGD in the lecture notes cannot be adapted to online gradients. Indeed, this analysis requires a few conditions. From slide 7 and using lecture 3's notation, these conditions are:

1. The loss on the i-th batch (f_i) is continuous on the relevant spaces
2. The gradient of f_i is smooth enough
3. The full dataset loss is not infinite on the negative side
4. The gradients on each given batch are not too far from the gradients on the whole dataset (their variance is bounded)

Condition 1, 2 and 3 don't change in the case of online gradient. Condition 4 however is not met. Indeed, condition 4 ensures that the data on any given batch is not completely absurd when looking at the whole dataset. In the stratified case however, most batches contain a completely screwed distribution of data, representing only one digit (the only ones containing multiple digits are when passing from one digit to another, but these are very few compared to the whole dataset). As a result, the distance between the batch gradients and the gradient of the full dataset is non negligeable and may break the convergence result because of condition 4.

2.4 Exercise 4

According to my experiments, the difference between Adam and SGD is not huge in this problem. We notice however a few differences.

Sensibility to the learning rate We notice that SGD performs globally well over a wide range of learning rates. Adam on the other hand performs average-to-poorly with learning rates on the order of 10^0 and 10^{-1}. It improves greatly for learning rates on the order of 10^{-2} to 10^{-3} though. Quoting (the half-jokingly) Andrej Karpathy: *3e-4 is the best learning rate for Adam, hands down.*. Although this is quite a strong generality, Andrej Karpathy is not completely wrong there because we notice that Adam performs much better with small learning rates.

Speed of convergence This is the biggest difference we have noticed between Adam and SGD. Adam converges in fewer epochs that SGD. The first few epochs usually have an order of magnitude difference in favor of Adam, however after a couple more epochs, the difference usually lessens.

Final results on test set According to the experiments I ran, Adam tends to overperform SGD with the right choice of the learning rate. Several sources online however mentioned that Adam sometimes strugle with generalization issues. For this reason, I suspect that either Adam behaves well on this dataset, or my few initializations were somewhat luckier with Adam than with SGD. Note that luck takes a non-zero place in this problem since the initialization of the weights does indeed matter for both methods, as well as the samples picked in the batch.

A Varying learning rates

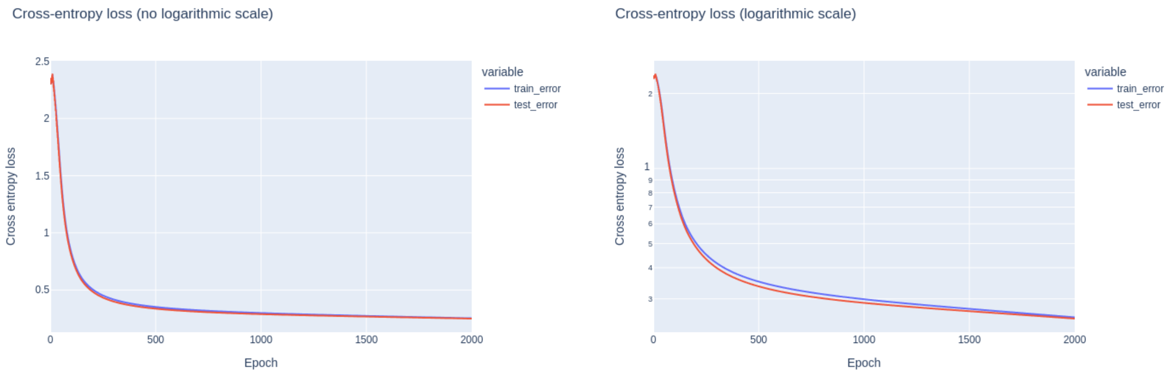

Figure 5: Cross entropy loss, train vs test (300 fully-connected), $\eta = 0.001$

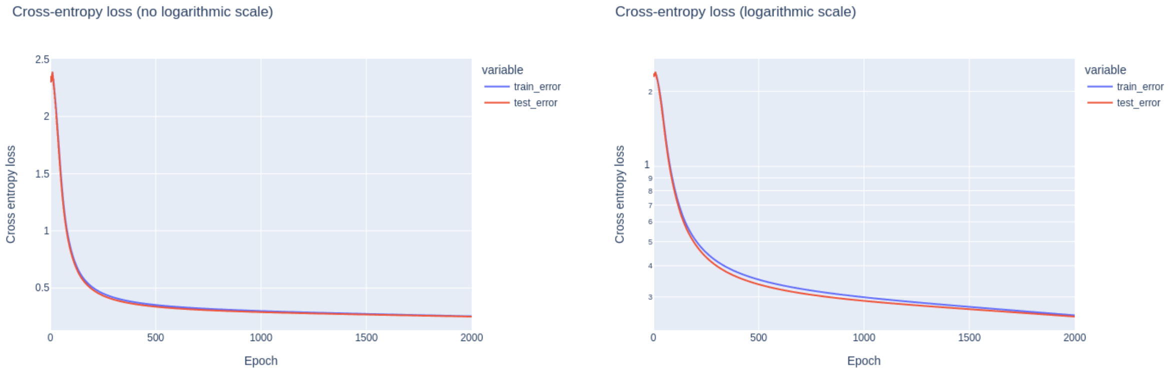

Figure 6: Cross entropy loss, train vs test (300 fully-connected), $\eta = 0.3$

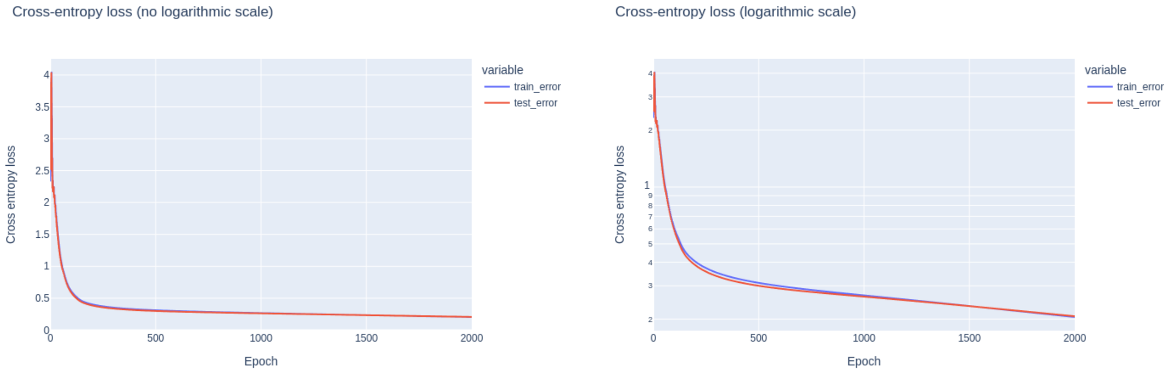

Figure 7: Cross entropy loss, train vs test (300 fully-connected), $\eta = 0.5$

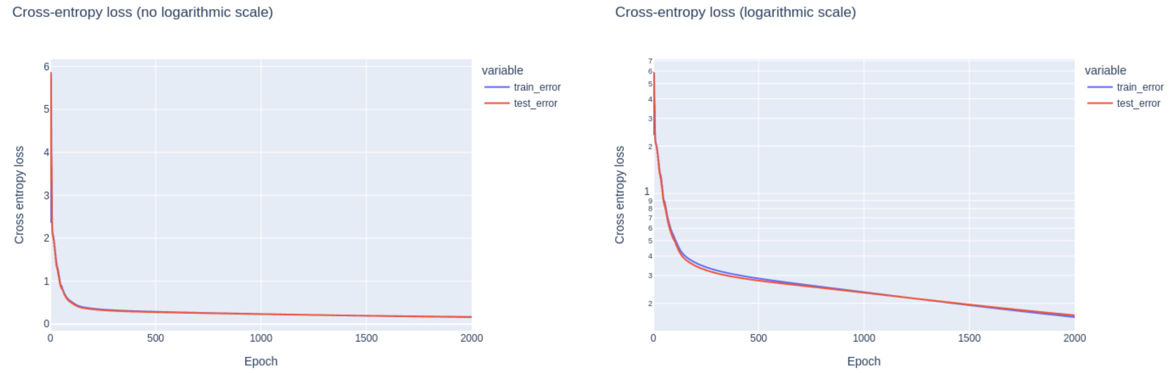

Figure 8: Cross entropy loss, train vs test (300 fully-connected), $\eta = 0.7$

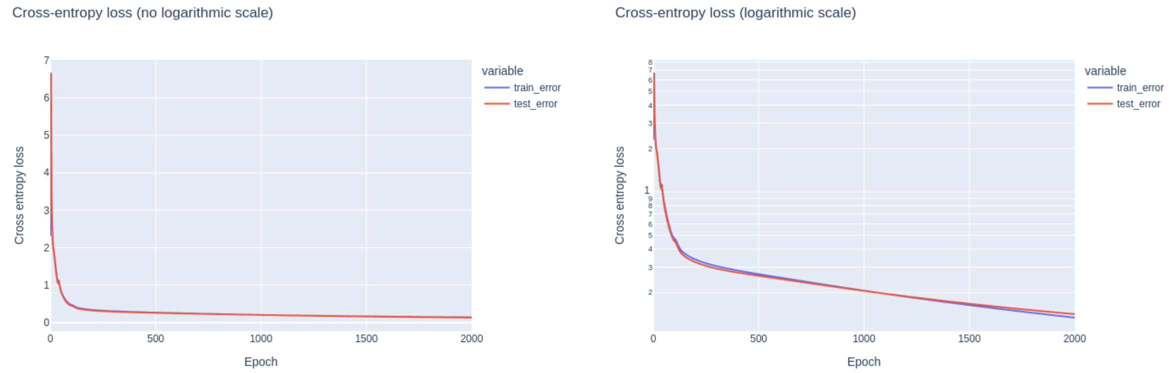

Figure 9: Cross entropy loss, train vs test (300 fully-connected), $\eta = 0.9$

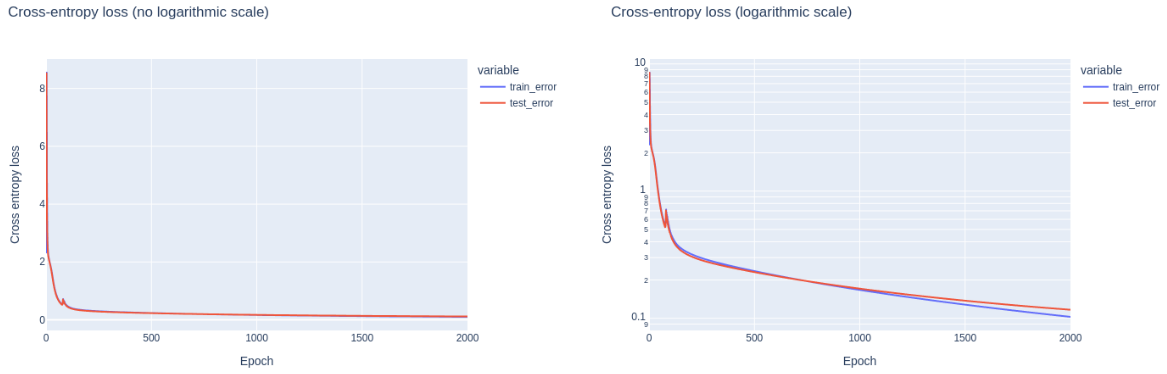

Figure 10: Cross entropy loss, train vs test (300 fully-connected), $\eta = 1.2$

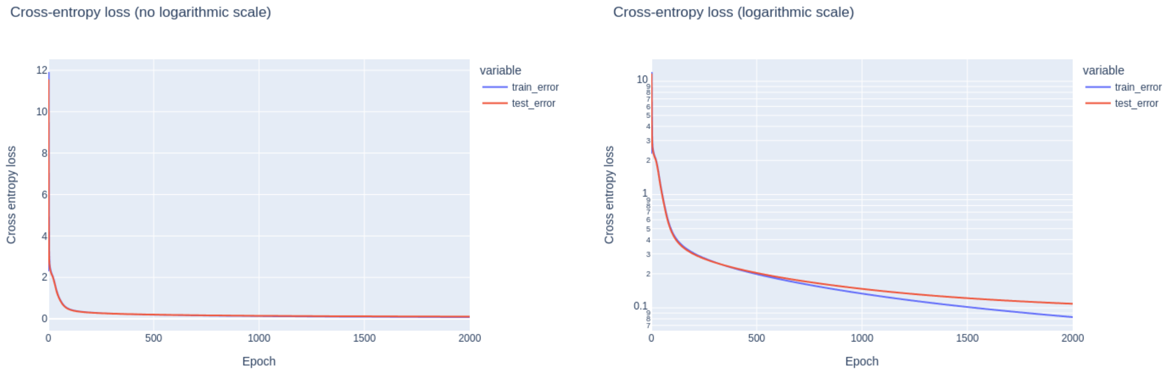

Figure 11: Cross entropy loss, train vs test (300 fully-connected), $\eta = 2$

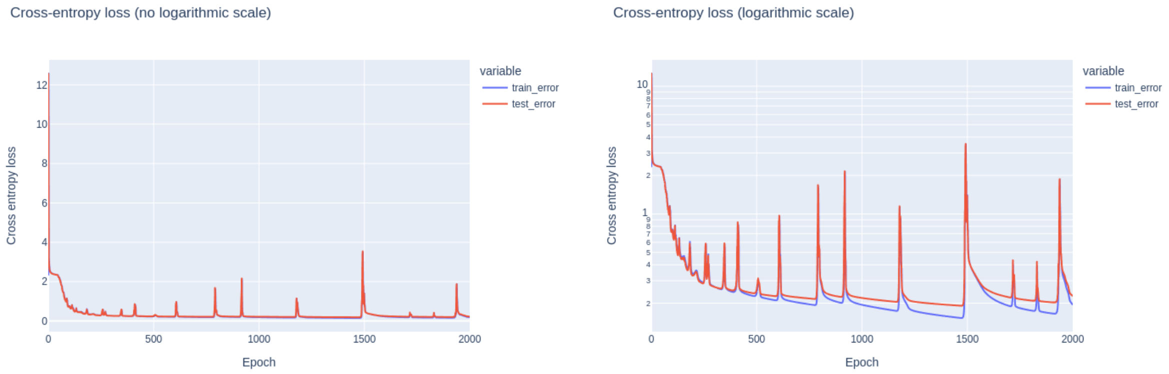

Figure 12: Cross entropy loss, train vs test (300 fully-connected), $\eta = 5$

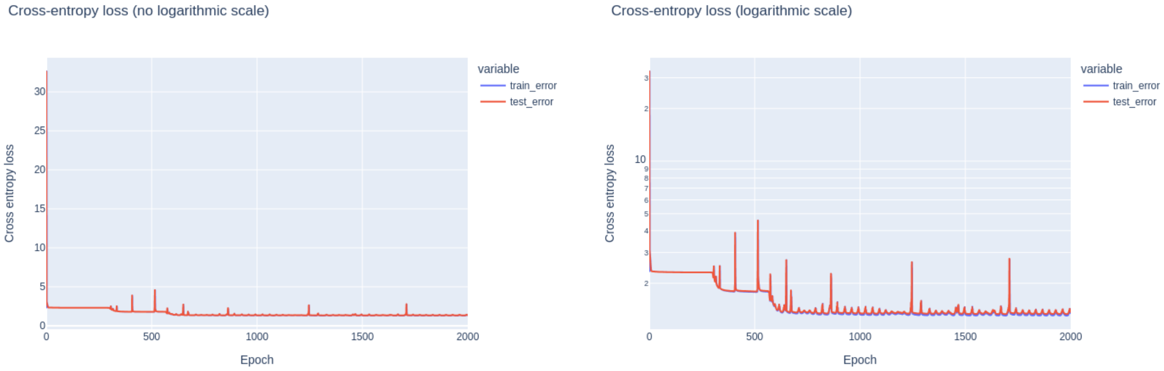

Figure 13: Cross entropy loss, train vs test (300 fully-connected), $\eta = 8$