Exercise 1
ConvNets
Advanced Practical Course In Machine Learning

Alon Netser
31160253-6

November 19, 2019

1 Theoretical Questions

1.1 Parameterized ReLU

We define the following function

$$f_i(o;t) = \max\{t, o_i\}$$

and claim that incorporation of this function into a network **does not** define a larger hypothesis class.

If we take a specific architecture of a neural-network, we can think of the class of all function that can be represented using this architecture (by setting the weights of the computational graph differently). This is the hypothesis-class of this network. If we claim that replacing ReLU with the above function f_i in the graph does not define a larger hypothesis-class, it means that for both architectures, every function that can be expressed using the first one can be expressed using the second one, vice versa.

Well, any function that can be expressed using the architecture with the regular ReLU obviously can be expressed using the new architecture by setting the constant t to be 0, thus making it a regular ReLU.

Now, let's look at a function that is being represented by the new architecture. We want to show that we can express the same function with an architecture where the parameterized ReLU is replaced with a regular ReLU. We go through the layers of the neural-network one after another. This process can be done repeatedly for all layers, therefore proving our point.

In a layer ℓ, if there is a parameterized ReLU with t as a parameter, we replace this activation with a regular ReLU and adjust the bias vector of that layer by subtracting t from each of its coordinates. Then the output of the layer (after the activation function) is the same as the original neural-network (with the parameterized ReLU), but shifted by t. Indeed, if previously the output of the neuron was $\max\{t, x\}$ (where x is the input of the neuron and t is the parameter of the parameterized ReLU) now it is $\max\{0, x - t\}$ so if x

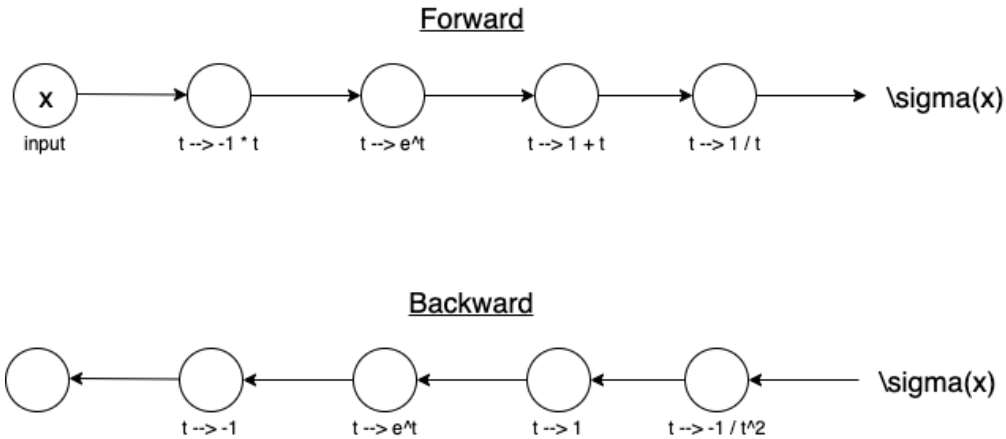


Figure 1: The computational graph of the sigmoid function

was larger than t the output will be $x - t$, and if x was smaller than t the output will be 0 (both are the same outputs shifted by t).

Now we need to handle the next layer $\ell+1$, to fix the shift by t issue. Denote the neurons of the previous layer as $x_1, \ldots, x_m$, the number of neurons in the $(\ell+1)$-th layer as k, and the weights and biases as $b \in \mathbb{R}^k$ and $w \in \mathbb{R}^{k \times m}$. The input of the i-th neuron in the $(\ell+1)$-th layer is $b_i + \sum_{j=1}^{m} w_{ij} \cdot x_j$. This is in the original network (with the parameterized ReLU). Now, each x_i is shifted by t, so we get $b_i + \sum_{j=1}^{m} w_{ij} \cdot (x_j - t)$. Well, now in order to make this expression similar to the original one, what we need to do is adjust the bias, so the new b_i will be modified as in $b_i \leftarrow b_i + \sum_{j=1}^{m} w_{ij} \cdot t$.

We showed how we can take a layer and replace the parameterized ReLU with a regular ReLU and adjust the biases (and only the biases) so that the output of the network will be the same. After this will be done iteratively for all layers, we'll get and identical function. This means that the hypothesis class did not grow by adding the parameterized ReLU, as claimed.

1.2 Sigmoid Derivative

The computational-graph of the sigmoid function $x \mapsto \sigma(x)$ (both forward and backward) is shown in Figure 1.

In order to take derivative of σ with respect to x we use the chain-rule:

$$\begin{aligned}
(\frac{1}{1+e^{-x}})' &= \frac{-1}{(1+e^{-x})^2} \cdot e^{-x} \cdot (-1) \\
&= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{1+e^{-x}-1}{(1+e^{-x})^2} \\
&= \frac{1}{1+e^{-x}} - \frac{1}{(1+e^{-x})^2} \\
&= \sigma(x) - \sigma(x)^2
\end{aligned}$$

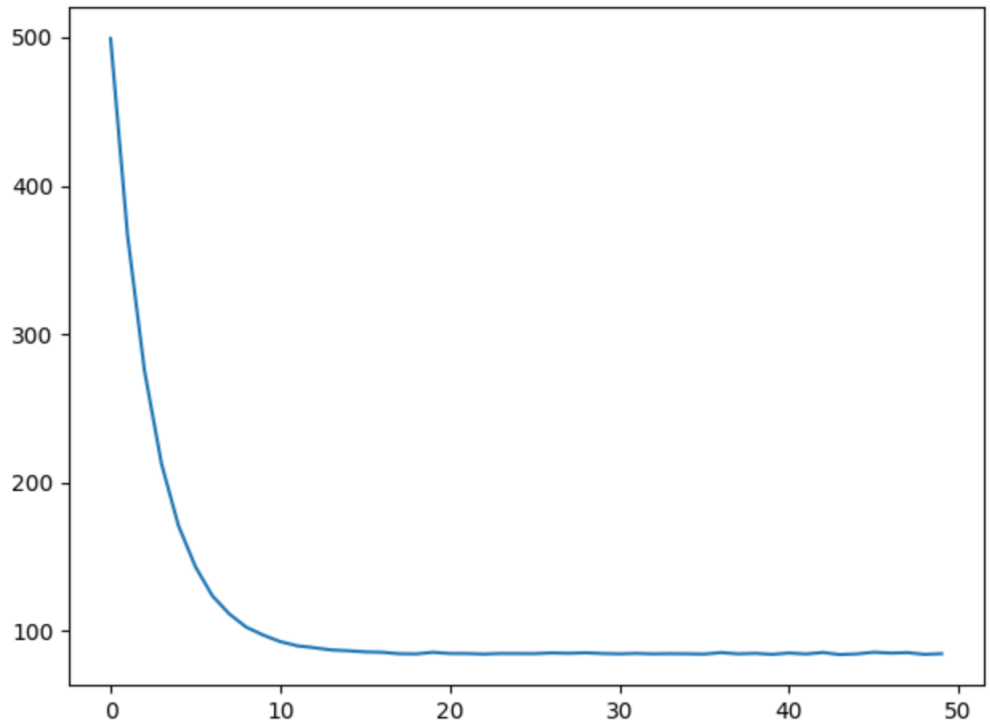


Figure 2: Linear regression loss

So we see that in order to calculate the derivative of σ with respect to x, we can just "remember" the value of $\sigma(x)$ from the forward pass (which we needed to compute anyway), and calculate the value minus the value-squared.

On the other hand, if we had to use the normal back-propagation, we had to divide minus-one by some squared value, then exponentiate and multiply and then multiplying by minus 1. These are a lot of operations that can be avoided by using the value from the forward pass and subtracting the squared value.

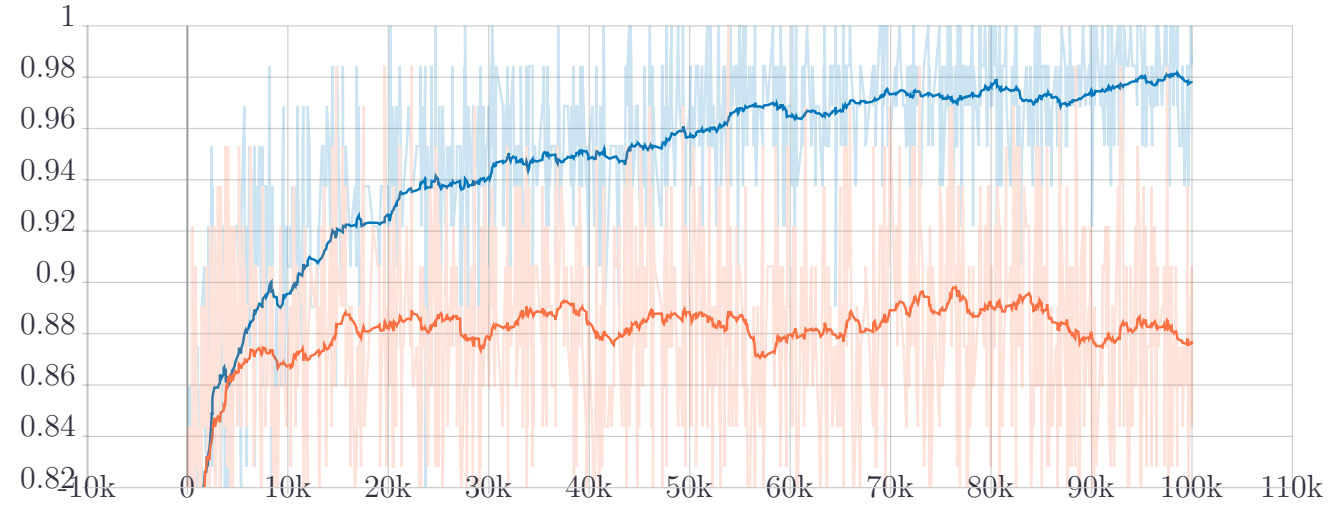
2 Practical Exercise

2.1 Linear Regression

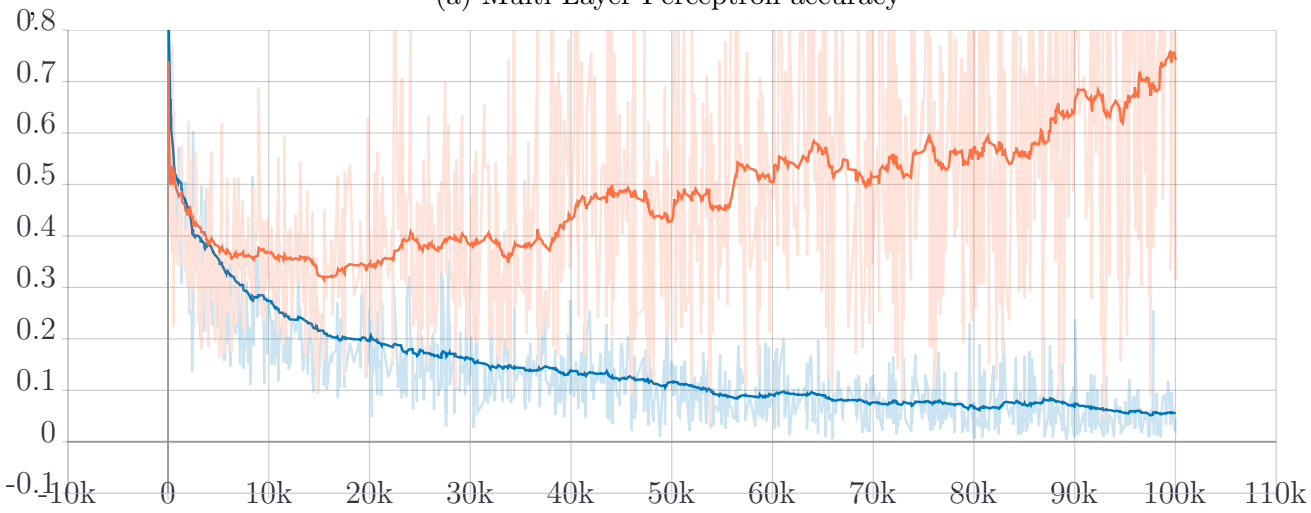
Figure 2 shows the mean loss (over all mini-batches in this epoch) as a function of the epoch number. The learning-rate that was used in 0.0001 (bigger learning-rates results in less "smoothed" graph).

2.2 Multi-Layer-Perceptron

Figure 3 shows the accuracy and loss as a function of the iteration number. The graphs are smoothed (using the TesnorBoard smoothing mechanism), because the values themselves (of each iteration) are quite noisy. One can see that the training process over-fit, as the training loss decreases and the testing loss increases.

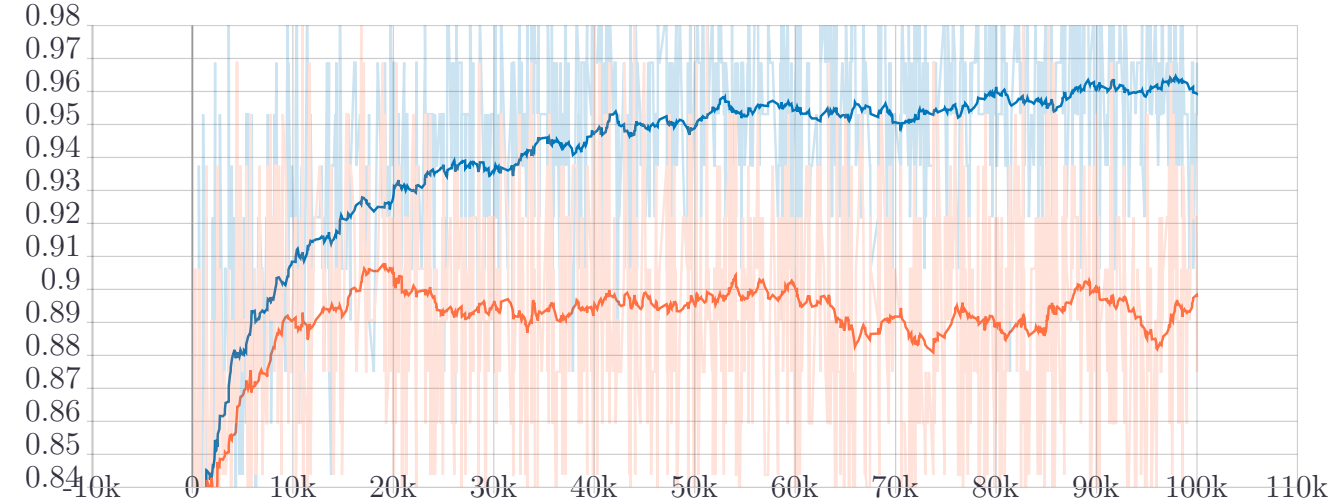


(a) Multi-Layer-Perceptron accuracy

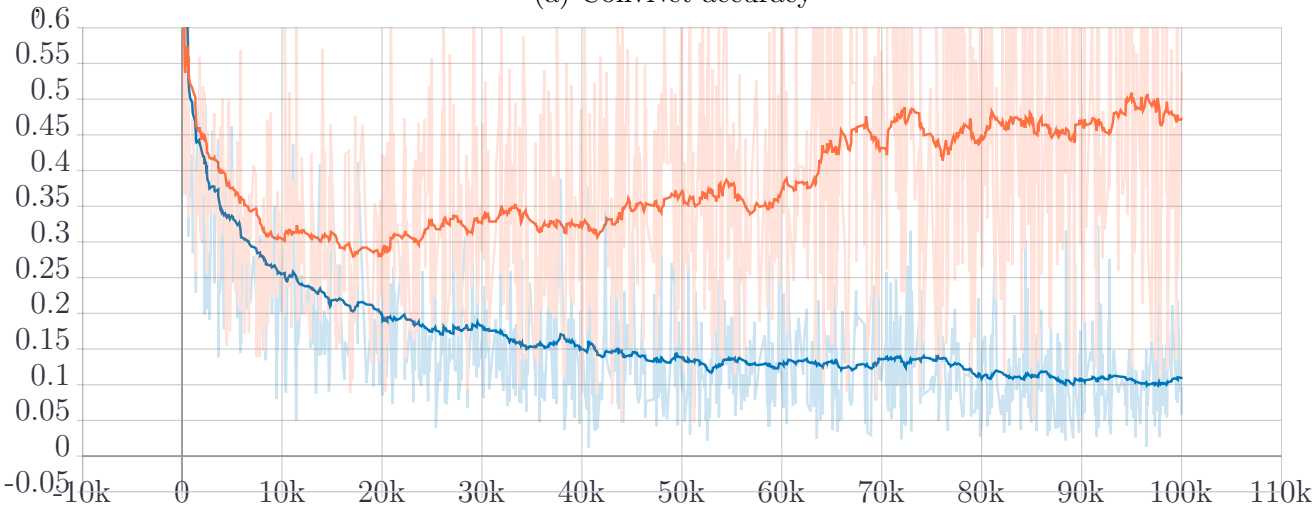


(b) Multi-Layer-Perceptron loss

Figure 3: The accuracy and loss of the Multi-Layer-Perceptron, as a function of the iteration number. The blue graph refers to the training performance, whereas the orange one refers to the testing performance.



(a) ConvNet accuracy



(b) ConvNet loss

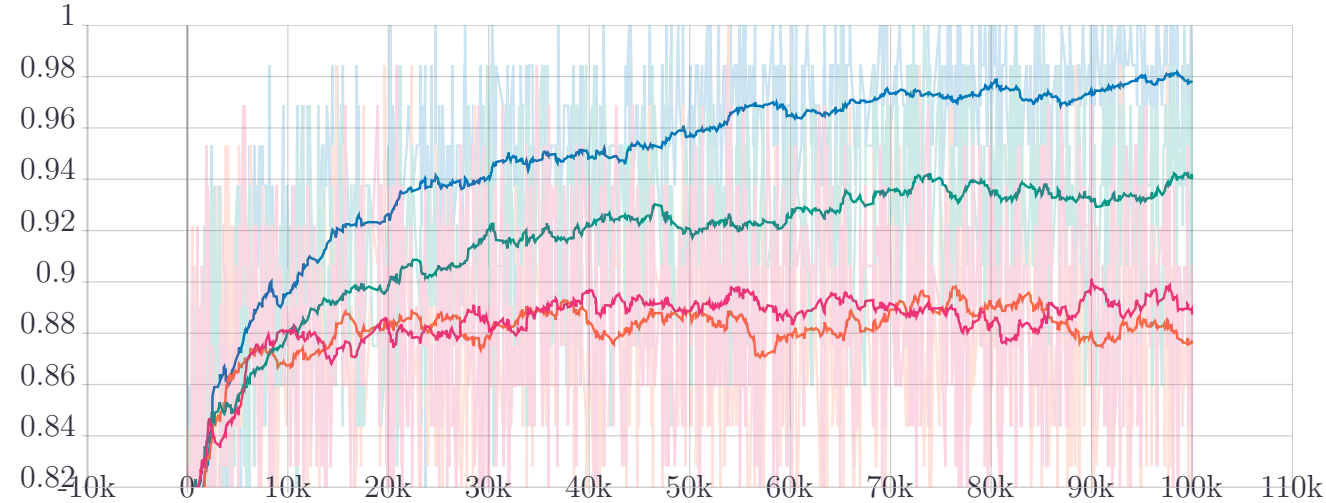
Figure 4: The accuracy and loss of the ConvNet, as a function of the iteration number. The blue graph refers to the training performance, whereas the orange one refers to the testing performance.

2.3 ConvNet

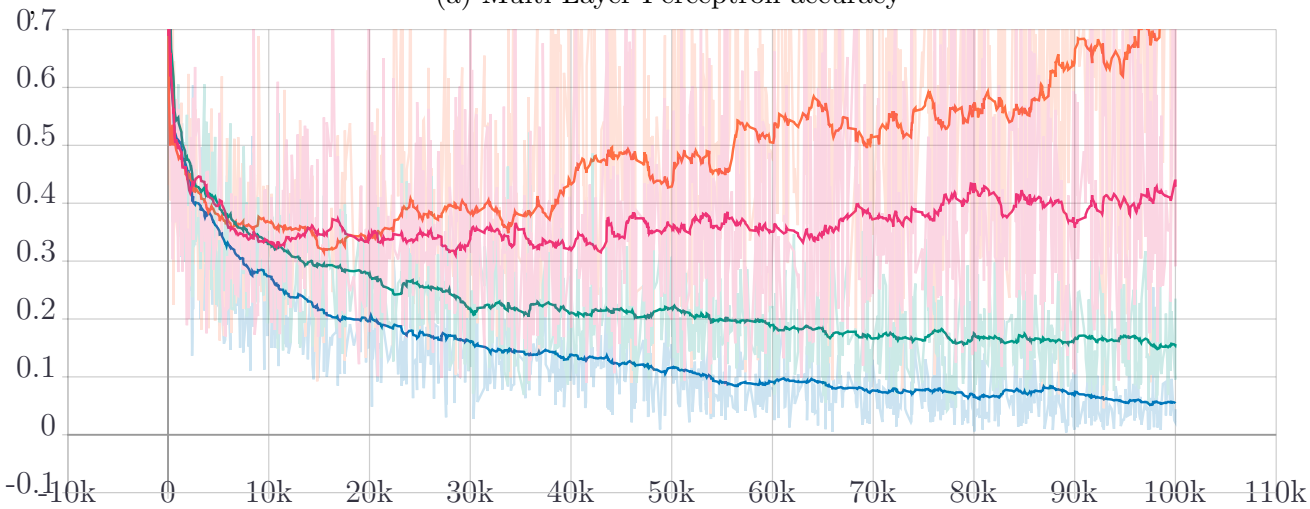
Figure 4 shows the accuracy and loss as a function of the iteration number. The graphs are smoothed (using the TesnorBoard smoothing mechanism), because the values themselves (of each iteration) are quite noisy. One can see that after about 10,000 iterations the training process begins to over-fit, as the training loss decreases and the testing loss increases.

2.4 Regularization - dropout

I tried to add regularization, in the form of dropout layers. In the Multi-Layer-Perceptron a dropout on the hidden layer was added, with a rate of 0.25. In the ConvNet two layers of dropout were added (after each conv-pool block), each with a rate of 0.25. Note that in both



(a) Multi-Layer-Perceptron accuracy



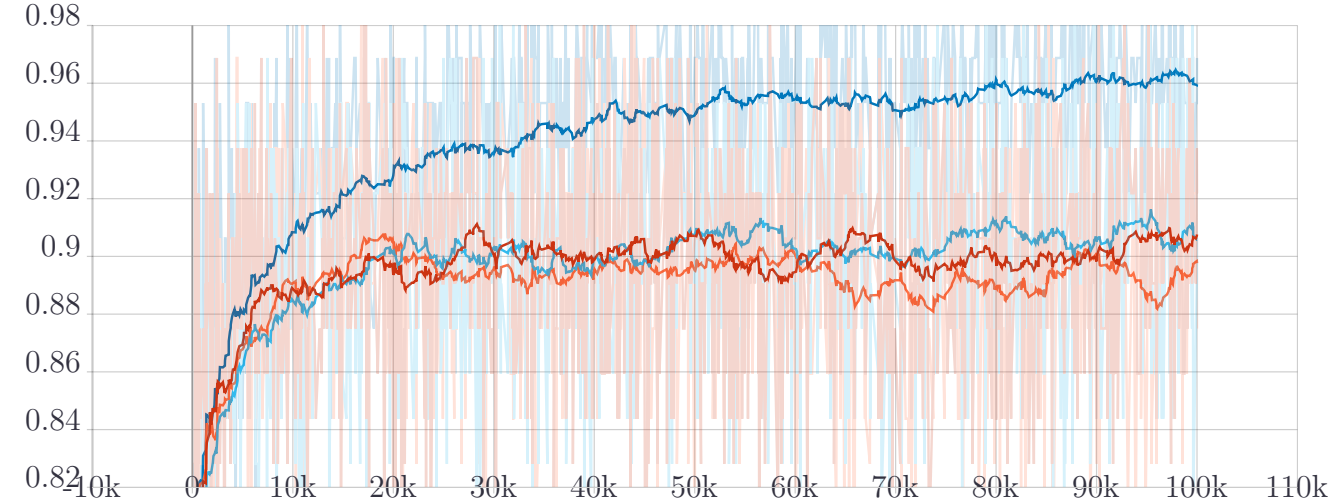
(b) Multi-Layer-Perceptron loss

Figure 5: The accuracy and loss of the Multi-Layer-Perceptron, as a function of the iteration number. Comparison between a version without dropout to a version with a dropout layer with rate 0.25, after each hidden layer. As before, the blue/orange graphs refer to the training/testing performance of the no-dropout version. The green/pink graphs refer to the training/testing performance of the dropout version.

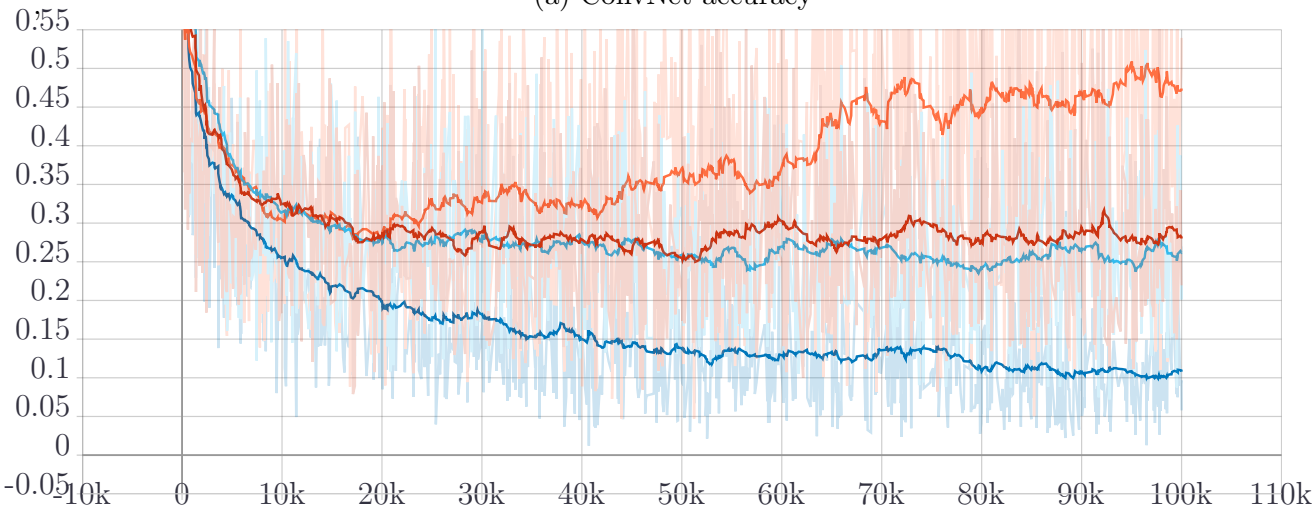
models different probabilities were taken, but 0.25 achieved the best performance (although the differences were quite small). The results show clearly that adding dropout helped avoid over-fitting the training-data, however the performance did not increases a lot.

2.4.1 Multi-Layer-Perceptron

Figure 5 shows the accuracy and loss as a function of the iteration number. The graphs are smoothed (using the TesnorBoard smoothing mechanism), because the values themselves (of each iteration) are quite noisy. Clearly, the version with the 0.25 dropout layers helped avoid



(a) ConvNet accuracy



(b) ConvNet loss

Figure 6: The accuracy and loss of the ConvNet, as a function of the iteration number. Comparison between a version without dropout to a version with two layers of 0.25 dropout, after each pooling layer. As before, the blue/orange graphs refer to the training/testing performance of the no-dropout version. The light-blue/red graphs refer to the training/testing performance of the dropout version.

over-fitting, despite the fact that the train/test performance are still not the same. Note that the dropout helped decreasing the test-loss, comparing to the regular version without dropout (although the test-accuracy remains about the same).

2.4.2 ConvNet

Figure 6 shows the accuracy and loss as a function of the iteration number. The graphs are smoothed (using the TesnorBoard smoothing mechanism), because the values themselves (of each iteration) are quite noisy. Clearly, the version with the 0.25 dropout layers avoids over-

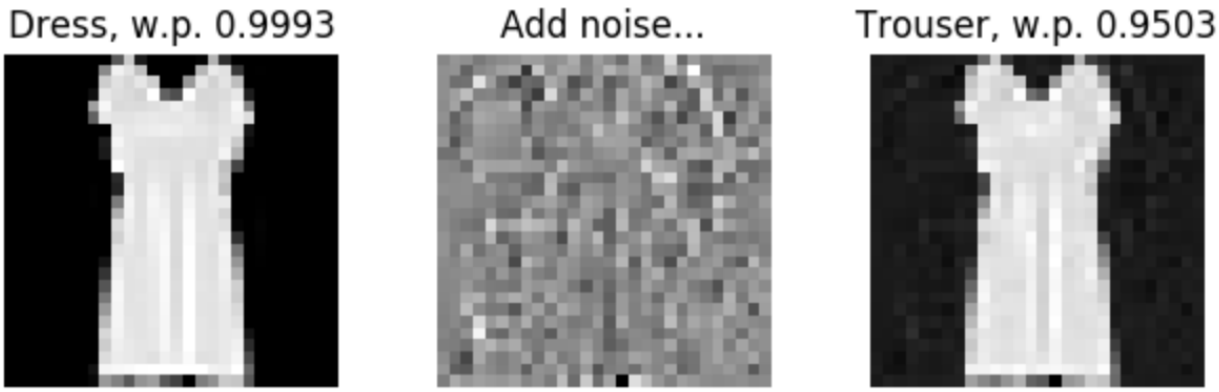
fitting, and the train/test performance are quite the same. Note that the dropout helped decreasing the loss (and also slightly increasing the accuracy), comparing to the regular version without dropout.

2.5 Adversarial sample

I found an adversarial using the following technique. I loaded a model (the trained ConvNet with dropout from the previous section). I took some random image from the training-set, and sampled a random label (i.e. target-label) that is different from its correct label. Then, I ran the network with that image as a input.

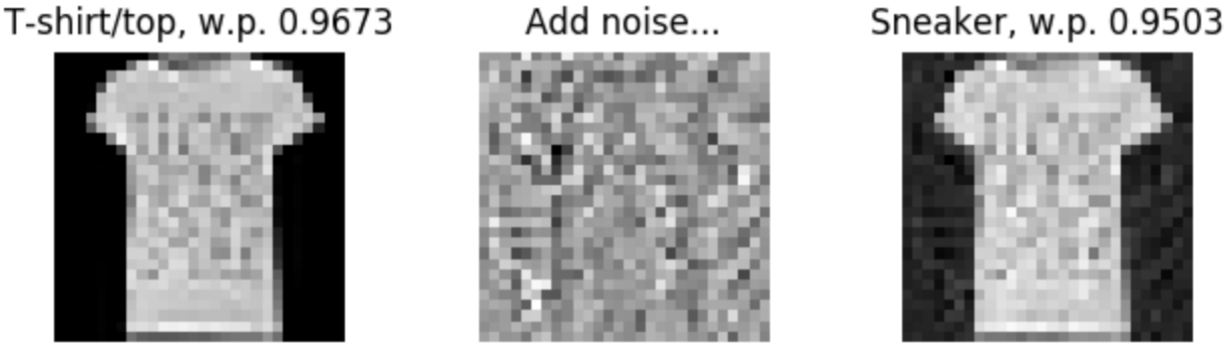
The new loss function was an addition of two terms - the first was that the prediction should be the same as the target-label, and the second is that the image should not be far from the original image (using ℓ_1-norm). The second term was in the new loss function was multiplied with a constant of 0.05 Then, I calculated the gradient of this loss function, with respect to the input image, and updated the image according to this gradient (using a learning-rate of 0.001).

In Figure 7 some nice adversarial images are shown, together with the original images and the added "noise" that resulted in the adversarial images (which is simply the difference between the new image and the original one).

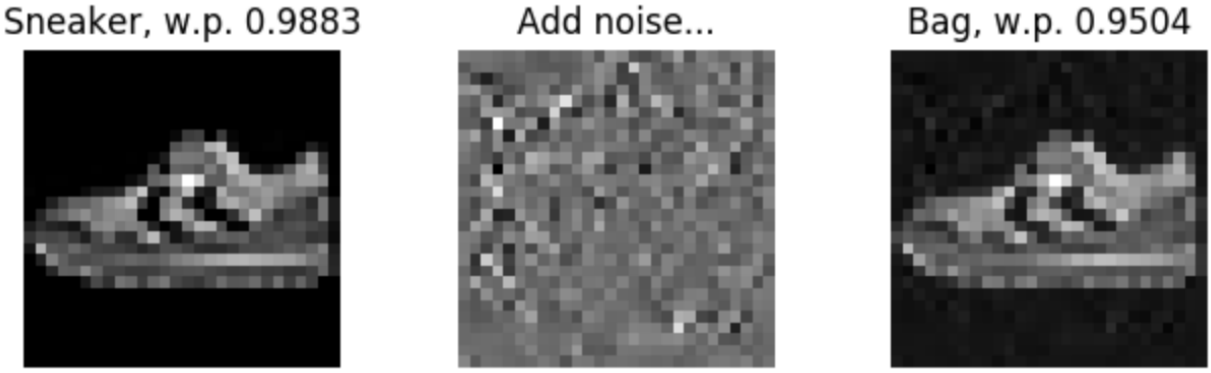


(a) A dress becomes a trouser

,



(b) A shirt becomes a sneaker



(c) A sneaker becomes a bag

Figure 7: These are all adversarial samples created using the method discussed.