Deep learning Lab
Uni Freiburg

Assignment 1
Maryna Kapitonova

Classifying MNIST with MLP

Introduction
The task was to implement feedforward neural network (multilayer perceptron) for MNIST digits classification using Python and evaluate its performance.

Implementation
Requirements:

- *possibility to use 2-4 layers*
- *sigmoid/tanh and ReLU for the hidden layer*
- *softmax output layer*
- *optimization via gradient descent (gd)*
- *optimization via stochastic gradient descent (sgd)*
- *gradient checking code*
- *weight initialization with random noise (!!!) (use normal distribution with changing std.)*

MLP is feedforward NN, each layer is using nonlinear activation function, except of the topmost layer, for which we use softmax f-n and 10 units (amount of classes).
Learning technique (supervised) implemented in model is backpropagation.
Learning occurs by changing weights in each iteration based on the error in the output.

Training procedures
GD
Gradient gives us linear first-order approximation to our function at the current point.
Step size (learning rate) is a hyperparameter.
SGD
Rather than computing the loss of gradient over whole sample space, instead on each iteration we sample some small set of training set minibatch. We use these batches to compute an estimate of a full sum and an estimate of true gradient.

The most tricky part for me during implementation was gradient checking procedure, which is actually a great debugging tool and behaves like a unit test for checking if implemented analytic gradient is correct. In the end, I've implemented this part using following formula:

$$g_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2 \times \text{EPSILON}}.$$

source

Results

For gradient checking procedure:

checking gradient for layer 1
diff 4.60e-12
diff 3.58e-12
checking gradient for layer 2
diff 1.07e-11
diff 9.59e-11
checking gradient for layer 3
diff 4.09e-12
diff 1.14e-10

For predefined network sample with 3 layers (100 units in first two layers, 10 in the output) relu activation functions and shg training procedure the following configurations were tested:

#	Input	Epochs	Batch Size	Learning rate	Training error	Validation error	Test error
1	10000	20	16	0.1	0	0.0383	0.0414
2		20	32	0.1	0.0016	0.0473	0.0507
3		20	64	0.1	0.015	0.0526	0.055
4		20	64	0.2	0.0004	0.0449	0.0489
5		20	32	0.2	0	0.0389	0.0419
6		20	16	0.2	0	0.0364	0.0385
7		20	16	0.3	0.0032	0.0398	0.0434
8		40	16	0.2	0	0.0351	0.0377
9		40	32	0.2	0	0.039	0.0484
10	40000	40	32	0.2	0	0.0211	**0.0206**
11		20	32	0.2	0.0014	0.0228	0.0226
12		20	64	0.2	0.0018	0.0213	**0.0209**
13	100/100/10	40	100	0.15	0.0001	0.0221	0.0235
14	512/256/10	20	100	0.2	0.0003	0.0199	**0.0202**
15	1028/512/10	40	100	0.2			

Optimal learning rate was found to be 0.2, less values converges very slow, bigger doesn't improve performance.

We can see that increase input set help us to significantly improve error, although computations become more time consuming in this case.

Small batch size induces noise, we can see that from illustration:

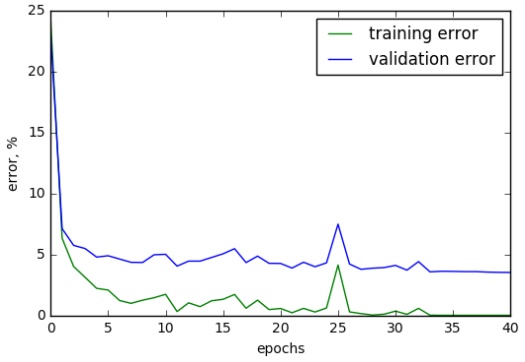


Fig.1. case#9 errors

The smallest test error 0.0202 (~2%) was achieved increasing the number of units per each layer. However, further increase of this parameter was a bit redundant and doesn't help to

reduce errors. In this particular implementation one thing that it'll be good to do - increasing epochs number. Graph for validation and test error evolution is following

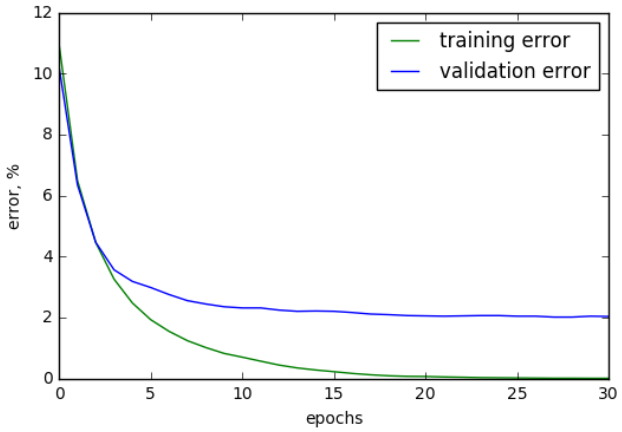


Fig. 2. Case #14 errors

We can see problems here, cause validation error doesn't improve over time, and the training error is close to zero (signs of overfitting).

After I've investigated different combinations of activation functions for 3-layer network with units number 512/256/10. Learning rate, batch size was the same in all cases. Results are summarized in the table 2:

3 layers	epoch	batch	learning rate	training error	validation error	test error
relu/tanh/softmax	30	128	0.2	0	0.205	0.0203
tanh/tanh/softmax	30	128	0.2	0.0023	0.024	0.0235
relu/relu/softmax	30	128	0.2	0.0001	0.021	0.0205
sigmoid/relu/softmax	20	128	0.2	0.0238	0.0329	0.0353
sigmoid/tahn/softmax	40	64	0.2	0.0022	0.0239	0.025

We can see that we have fairly good test error almost for all cases, although testing showed me that sigmoid function in combination with tanh function performs good in this case, cause in case of sigmoid function in 1st layer validation error is almost close to training error, and that's the sign of good fit.

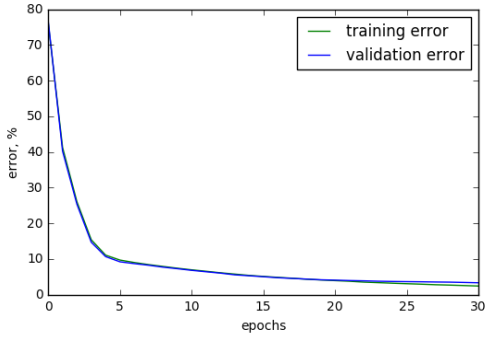


Fig.3. Case sigmoid/relu/softmax

Afterwards I've tested 4-layers models, choosing sigmoid/relu/relu/softmax activation functions, units number was 1028/512/256/10, learning rate 0.2, batch size 64, epochs 60.
I've got following errors:

Training error	Validation error	Test error
0.0036	0.0257	0.027

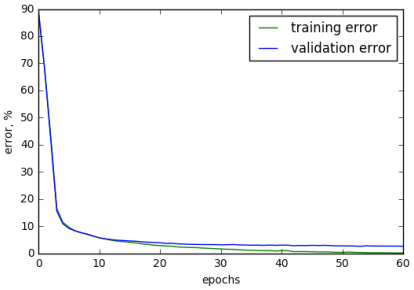
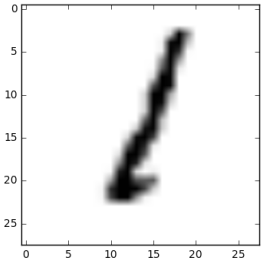
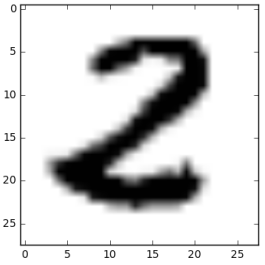
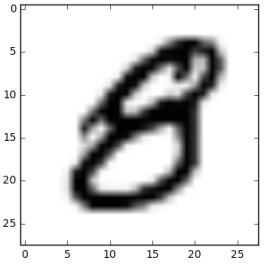
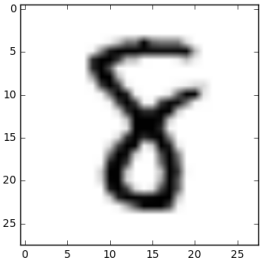


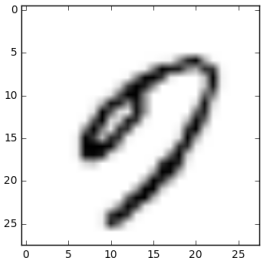
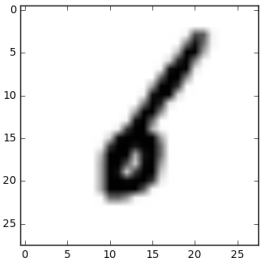
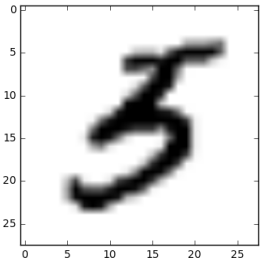
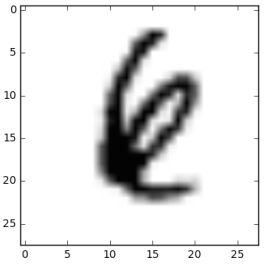
Fig.4. Error for 4-layer case

Interestingly, that increasing epoch number to 100 didn't help me to improve performance. As we can see, adding layer didn't improve our results, but it's much more time consuming.

Plotting

Finally, i've plotted several misclassified digits and correctly classified ones. I must admit here, that model is much better in recognizing digits than my visual system.

Randomly chosen correctly classified digits			
			
Predicted 1 Correct 1	Predicted 2 Correct 2	Predicted 8 Correct 8	Predicted 8 Correct 8

Randomly chosen misclassified digits			
			
Predicted 0 Correct 9	Predicted 1 Correct 6	Predicted 3 Correct 5	Predicted 8 Correct 6

Discussion

Training procedures differences
Gd vs SGD
SGD performance is better than GD because of frequent updates of W and b.

Activation functions
Sigmoid, tahn, ReLU, Softmax
ReLU doesn't have gradient vanishing problems as sigmoid and tanh functions.
Sigmoid seems to be a good choice for input layer

Parameters impacting model performance

Input size
Increasing input size significantly improves model performance, although this takes longer.

Number of units per layer
Number of hidden units controls the number of weights which are updated. Non sufficient amount of neurons can lead to underfitting. Consequently, increasing the number of units per layer increases “learning space”, resulting in better performance.
Although this parameter increases complexity and duration of training.

Number of layers
Choosing the number of layers depends on features that we care about. A multi layer perceptron with one hidden layer(N=1) is capable to draw a >=2nd order decision boundary.
However, increasing the number of layers not necessarily lead to model improvement. This parameter is task-dependent and for lots of application one hidden layer is sufficient.
To check this i’ve performed experiment with 3 and 4-layered perceptron.

Batch size
I’ve choosed a batch size in powers of two.
Small batch size can introduce noise to gradient estimation.
Bigger batches reduces training time.

Learning rate (step size)
Small learning rates takes long to converge, but if the rate is too high it can overshoot.
So the aim is to find optimal learning rate.
During testing i’ve found out that learning rate correlates to activation function.

Epochs number
Since epoch is one forward and backward path through all training samples, it’s intuitive that increasing epochs number leads to decrease of error.
However, we must chose optimal value, because at some point network isn’t learning anymore, and further increase of epoch size is unreasonable. As well, we should always keep in mind such parameter as training time.

General observations:

- Training error underestimates validation error, although we want this error to be close to each other in order to achieve a “good fit”.
- To decrease error we can increase learning rate, number of epochs, input size.
- To speed up computations batch size can be increased, in contrast, small batch size results in “noisy” gradient.
- Using sophisticated models(redundant layers, big number of units) can lead to overfitting(high validation error, low training)

Conclusions

Network structure optimization is sophisticated task.

This assignment revealed for me the unpredictability of neural network and importance of fine tuning of network parameters. As well I’ve figured out the necessity for developing training methodology.

Adjusting different parameters (batch size, learning rate, epochs number, size of the input set) along with choice of appropriate training procedure and network architecture can significantly improve performance of model.

I’ve enjoyed the experience a lot, although unfortunately i didn’t have enough time to implement bonus tasks (dropout implementation, l2 regularization and different optimization schemes) and improve network performance.