TDA 231 Machine Learning: Homework 3
Goal: Feed-forward neural networks

Mikael Kågebäck
kageback@chalmers.se

Due Date: February 15, 2016

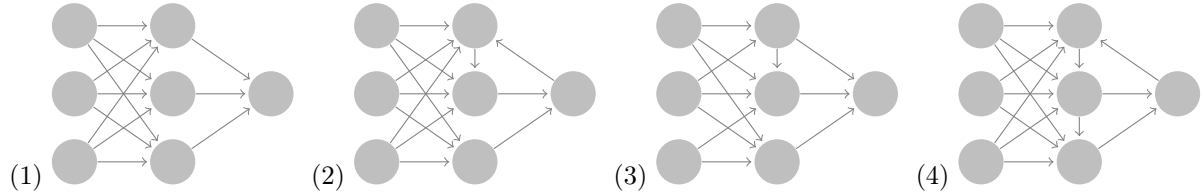
General guidelines:

1. All solutions to theoretical problems, and discussion regarding practical problems, should be submitted in a single file named *report.pdf*
2. All matlab files have to be submitted as a single zip file named *code.zip*.
3. The report should clearly indicate your name, personal number and email address
4. All datasets can be downloaded from the course website.
5. All plots, tables and additional information should be included in *report.pdf*
6. For questions regarding the homework contact Mikael Kågebäck (kageback@chalmers.se)

1 Theoretical problems

Problem 1.1 [Topological properties, 2 points]

(a) Which of the following neural networks are examples of feed-forward neural networks?



(b) In general: Which topological properties must the network fulfill to be a feed-forward neural network?

Problem 1.2 [Committee, 2 points]

Brian wants to make his feed-forward network (with no hidden units) using a linear output neuron more powerful. He decides to combine the predictions of two networks by averaging them. The first network has

weights w_1 and the second network has weights w_2. The prediction of this committee for an example x is therefore:

$$y = \frac{1}{2}\mathbf{w}_1^T\mathbf{x} + \frac{1}{2}\mathbf{w}_2^T\mathbf{x}$$

Can we get the exact same predictions as this combination of networks by using a single feed-forward network (again with no hidden units) using a linear output neuron and weights $\mathbf{w}_3 = f(\mathbf{w}_1, \mathbf{w}_2)$, where $f(\mathbf{w}_1, \mathbf{w}_2)$ is some linear function? Prove your claim!

Problem 1.3 [Backpropagation - shallow network, 2 points]

Consider a neural network with only one training case with input $x = (x_1, x_2, \ldots, x_n)^T$ and correct output $t \in \{0, 1\}$. There is only one output neuron, which is linear, i.e. $y = w^T x$ (notice that there are no biases). The cost function is a simple squared error ($E = \frac{1}{2}(t-y)^2$). The network has no hidden units, so the inputs are directly connected to the output neuron with weights $w = (w_1, w_2, \ldots, w_n)^T$. We're in the process of training the neural network with the backpropagation algorithm. What will the algorithm add to w_i for the next iteration if we use a step size (also known as a learning rate) of λ?

Problem 1.4 [Backpropagation, 4 points]

Derive general expressions for the partial derivatives of an error function E, surrounding a neuron j, in the feed-forward neural network depicted in Figure 1.

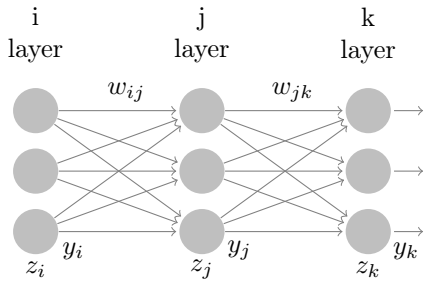


Figure 1: Three layer feed-forward neural network. Each layer labeled by its receptive index variable.

For convenience, we may consider only one training example and ignore the bias term. Forward propagation of the input z_i is done as follows.

$$y_i = g(z_i)$$
$$z_j = \sum_i w_{ij} y_i$$
$$y_j = g(z_j)$$
$$z_k = \sum_j w_{jk} y_j$$
$$y_k = g(z_k)$$

Where $g(z)$ is some differentiable function (e.g. the logistic function).

Now it is your job to do the back propagation. The incoming error derivatives $\frac{\partial E}{\partial y_k}$ are here given and can be used to compute downstream derivatives using the chain-rule. More precisely, give expressions where each factor is a computable derivative (or already computed as in the case of $\frac{\partial E}{\partial y_k}$). Tips: It is ok to reuse computed factors in subsequent subproblems.

(a) $\frac{\partial E}{\partial z_k} =$

(b) $\frac{\partial E}{\partial z_j} =$

(c) $\frac{\partial E}{\partial w_{jk}} =$

(d) $\frac{\partial E}{\partial w_{ij}} =$

2 Practical problems

In this assignment, you are going to train a feed-forward neural network to recognizing handwritten digits. You will be implementing back propagation and experimenting with efficient optimization and regularization.

The input to the neural network is a 16 by 16 image of greyscale pixels, showing an image of a handwritten digit. The output, that the model is supposed to generate, is which of the 10 different digits that is currently presented to the model.

The model needs an input layer consisting of 256 units, i.e. one for each pixel, one hidden layer of logistic units, and it uses a 10-way softmax as the output layer. To keep things as simple as possible, we're not including biases in our model.

The dataset for this assignment is the USPS collection of handwritten digits. It consists of scans (images) of digits that people wrote.

For training, we are mostly interested in the cross-entropy error of the model. As opposed to the classification error rate, which is more useful for the evaluation of the model. The reason is that the cross-entropy error is continuous and behaves better than the classification error rate. Only at the very end will we look at the classification error rate.

To investigate generalization, we need a training set, a validation set, and a test set. Hence, the dataset has been split in 3 groups. We train our networks on the training set, we use the validation set to find out what parameter settings generalize well, and we use the test set to evaluate the models performance. Those three subsets have already been made for you, and you're not expected to change them (you're not even allowed to change them). The full USPS dataset has 11,000 images. We're using 1,000 of them as training data, another 1,000 as validation data, and the remaining 9,000 as test data. Normally, one would use most of the data as training data, but for this assignment we'll use less, so that our programs run more quickly.

Before we get to the issue of generalization, we need a good optimization strategy. The optimizer that we will be using is stochastic gradient descent with momentum. The code is given but you will need to find good values for the learning rate and the momentum multiplier.

2.1 Setting up

Download the code and the data from the course homepage. Make sure that the code file is called "net.m" and the data file is called "data.mat". Place both of them in the same directory, start Matlab, cd to that directory, and run a test run without any training: net(0, 0, 0, 0, 0, false, 0). You should see messages that tell you the cost and classification error rate without any training. The cost on the training data for that test run should be 2.302585.

2.2 Programming

Most of the code has already been written for you. The script in net.m loads the data (training, validation, and test), performs the optimization, and reports the results, including some numbers and a plot of the training data and validation data cost as training progresses. For the optimization it needs to be able to compute the gradient of the cost function, and that part is up to you to implement, in function grad. You're not allowed to change any other part of the code. However, you should take a quick look at it, and in particular you should make sure that you understand the meaning of the various parameters that the main script takes (see line 1 of net.m).

The program checks your gradient computation for you, using a finite difference approximation to the gradient. If that finite difference approximation results in an approximate gradient that's very different from what your gradient computation procedure produced, then the program prints an error message. This is hugely helpful debugging information. Imagine that you have the gradient computation done wrong, but you don't have such a sanity check: your optimization would probably fail in many weird and wonderful ways, and you'd be worrying that perhaps you picked a bad learning rate or so. With a finite difference gradient checker, at least you'll know that you probably got the gradient right. It's all approximate, so the checker can never know for sure that you did it right, but if your gradient computation is seriously wrong, the checker will probably notice.

Take a good look at the cost computation, and make sure that you understand it. Notice that there's classification cost (cross-entropy error) and weight decay cost, which are added together to make the cost. Also notice that the cost function is an average over training cases, as opposed to a sum. Of course, that affects the gradient as well.

Problem 2.1 [Backpropagation on paper, 3 points]

Compute expressions for the gradients of the cost function, corresponding to the above described network, using back propagation. Include your derivations in the report!

Problem 2.2 [Backpropagation, 2 points]

Use the expressions you derived in Problem 2.1 to implement Backpropagation for the model by following the steps below.

1. Run net(1e7, 7, 10, 0, 0, false, 4), i.e. a run with a huge weight decay so that the weight decay cost overshadows the classification cost. You should see a gradient error message from gradient checker.
2. Implement the weight decay part of the cost gradient and run net(1e7, 7, 10, 0, 0, false, 4) again. If the gradient check passes, then you probably did this right. If it doesn't, take a close look at the error message, and try to figure out where you may have made a mistake.
3. Run net(0, 7, 10, 0, 0, false, 4), i.e. turn weight decay off, and you will see the gradient error message coming back.
4. Implement the classification cost gradient, and if you get any error message from the gradient checker, look closely at the numbers in that error message.

Report the training data cost when running net(0.1, 7, 10, 0, 0, false, 4). Use at least 5 digits after the decimal point. Please, include all the code **you write** in the the report.

2.3 Experimentation

We will start with a small version of the task, to best see the effect of the optimization parameters. The small version do not use weight decay or early stopping, only 10 hidden units, 70 optimization iterations, and mini-batches of size 4 (usually, mini-batch size is more like 100, but for now we use 4).

While investigating how the optimization works best, concentrate on the cost on training data. That's what we're directly optimizing, so if that gets low, then the optimizer did a good job, regardless of whether the solution generalizes well to the validation data.

Do an initial run with learning rate 0.005 and no momentum: run net(0, 10, 70, 0.005, 0, false, 4). What is the training data cost? Write it down for reference, but you do not need to report it.

In the plot you'll see that training data cost and validation data cost are both decreasing, but they're still going down steadily after those 70 optimization iterations. We could run it longer, but for now we won't. We'll see what we can do with 70 iterations.

Try a bigger learning rate: LR=0.5, and still no momentum. You'll see that this works better.

Finding a good learning rate is important, but using momentum well can also make things work better. Without momentum, we simply add $\lambda(-\frac{\partial E}{\partial w})$ to w at every iteration, but with momentum, we use a more sophisticated strategy: we keep track of the momentum speed, using $v_{t+1} = v_t\beta - \frac{\partial E}{\partial w}$, and then we add $v\lambda$ to w. The momentum parameter β can be anything between 0 and 1, but usually 0.9 works well.

Problem 2.3 [Optimization, 2 points]

Experiment with a variety of learning rates to find out which works best. Try 0.002, 0.01, 0.05, 0.2, 1.0, 5.0, and 20.0. All of those both without momentum (i.e. momentum=0.0 in the program) and with momentum (i.e. momentum=0.9 in the program). Hence, a total of 7 x 2 = 14 experiments to run. Remember, what we are interested in right now is the cost on the training data, because that shows how well the optimization works. Which of those 14 worked best?

(a) Was the best run a run with momentum or without momentum?

(b) What was the learning rate for the best of those 14 runs?

Now that we found good optimization settings, we are switching to a somewhat bigger task, in order to investigate generalization. Now we are mostly interested in the classification cost on the validation data: if that's good, then we have good generalization, regardless whether the cost on the training data is small or large.

Notice that we're measuring only the classification cost. We are not interested in the weight decay cost. The classification cost is what shows how well we generalize. When we don't use weight decay, the classification cost and the final cost are the same, because the weight decay cost is zero.

Problem 2.4 [Generalization, 3 points]

(a) We'll start with zero weight decay, 200 hidden units, 1000 optimization iterations, a learning rate of 0.35, momentum of 0.9, no early stopping, and mini-batch size 100, i.e. run net(0, 200, 1000, 0.35, 0.9, false, 100). This run will take more time. What is the validation data classification cost now? Write your answer with at least 5 digits after the decimal point.

(b) The simplest form of regularization is early stopping, i.e. use the weights as they were when validation

data cost was as lowest. As can be seen in the plot, this is not at the end of the 1000 optimization iterations, but quite a bit earlier. The script has an option for early stopping. Run the experiment with the early stopping parameter set to true. Now the generalization should be better. What is the validation data classification cost now, i.e. with early stopping?

(c) Another regularization method is weight decay. Let's turn off early stopping, and instead investigate weight decay. The script has an option for L2 weight decay. As long as the coefficient is 0, in effect there is no weight decay, but let's try some different coefficients. We've already run the experiment with $WD = 0$. Run additional experiments for $WD \in \{10, 1, 0.0001, 0.001, 5\}$, and indicate which of them gave the best generalization. Be careful to focus on the classification cost (i.e. without the weight decay cost), as opposed to the final cost (which does include the weight decay cost).

(d) Yet another regularization strategy is reducing the number of model parameters, so that the model simply doesn't have the brain capacity to overfit (learning fine grained details of the training set that does not generalize). In our case, we can vary the number of hidden units. Since it's clear that our model is overfitting, we'll look into reducing the number of hidden units.

Turn off the weight decay, and instead try the following hidden layer sizes $10, 30, 100, 130, 200$. Indicate which one worked best.

(e) Most regularization methods can be combined quite well. Let's combine early stopping with a carefully chosen hidden layer size. Which number of hidden units works best that way, i.e. with early stopping? Remember, best, here, is based on only the validation data cost. Try the following hidden layer sizes $18, 37, 83, 113, 236$.

(f) Of course, we could explore a lot more, such as maybe combining all 3 regularization methods, and that might work a little better. If you want to, you can play with the code all you want. You could even try to modify it to have 2 hidden layers, to add dropout, or anything else. The code is a reasonably well-written starting point for Neural Network experimentation. All of that, however, is beyond the scope of this assignment; here, we have only one question left.

Now that we have carefully established a good optimization strategy, as well as a good regularization strategy, it's time to see how well our model does on the task that we really cared about. Reading handwritten digits. For the settings that you chose on the previous question, what is the test data classification error rate? *For a gold star: Can you beat it?*