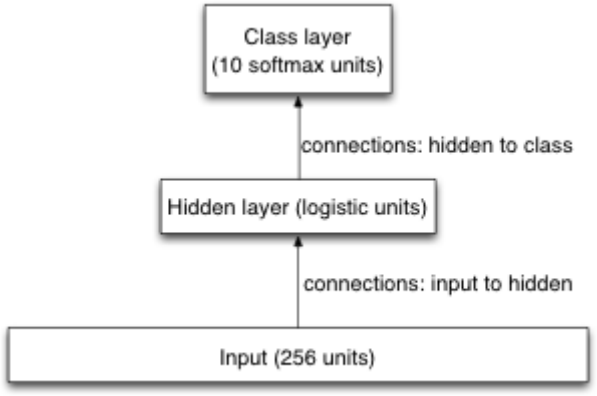
CSC 321 assignment 2: Optimization and Generalization

Due on Tuesday March 4 at 1:10pm, on paper (i.e. hard copy). For more details, see the Assignments page on the course website.

By Hannes Bretschneider & Chris Maddison (csc321a@cs.toronto.edu)

In this assignment, you're going to train a simple neural network, for recognizing handwritten digits. You'll be programming, looking into efficient optimization, and looking into effective regularization to achieve good generalization. Generalization and regularization will be discussed in class on February 27, so you're advised to postpone the last part of this assignment until that class.

The dataset for this assignment is the USPS collection of handwritten digits. It consists of scans (images) of digits that people wrote. The input is a 16 by 16 image of greyscale pixels, showing an image of a handwritten digit. The output is simply which of the 10 different digits it is, so we're using a 10-way softmax as the output layer of our neural network. The input layer is simply 256 units, i.e. one for each pixel. We use one hidden layer of logistic units. One of the issues we'll be investigating is what number of hidden units works best for generalization. To keep things as simple as possible, we're not including biases in our model. In the diagram you can see that this model is significantly simpler than the model that we used in programming assignment 1.



In this assignment, we're mostly interested in the cross-entropy error, as opposed to the classification error rate. The reason for that is that the cross-entropy error is continuous and behaves better than the classification error rate. That makes it easier to analyze how our optimization and regularization efforts are going. 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, so 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's generalizing well and what isn't; and after we've made our choice of regularization strategy, we'll see how well our model performs on the test set. 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're using is gradient descent with momentum, but we'll need to find good values for the learning rate and the momentum multiplier.

Part 1: Setting up

We're using Matlab. Download the code from here, and the data from here. Make sure that the code file is called "a2.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: **a2(0, 0, 0, 0, 0, false, 0)**. You should see messages that tell you the loss and classification error rate without any training. The loss on the training data for that test run should be 2.302585.

To make the code behave as predictable as possible, we don't use truly random initialization of the weights. However, normally you would use randomness there.

Part 2: Programming

Most of the code has already been written for you. The script in a2.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 loss as training progresses. For the optimization it needs to be able to compute the gradient of the loss function, and that part is up to you to implement, in function d_loss_by_d_model. 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 a2.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. The gradient checker computes a linear approximation to the correct gradient, so even if the gradient checker passes, this is not a guarentee that your gradients are correct. However, if your gradient computation is seriously wrong, the checker will probably notice.

Take a good look at the loss computation, and make sure that you understand it.

- Notice that there's classification loss and weight decay loss, which are added together to make the total loss. Before we start using weight decay for regularization, the weight decay part of the loss is simply zero, so you can ignore it.
- Also notice that the loss function is an average over training cases, as opposed to a sum. Of course, that affects the gradient as well.

Now take a pen & paper, figure out how to compute the gradient, and implement it in Matlab. Make sure that your computation is reasonably efficient, i.e. make sure your code is vectorized and don't write nested loops over matrix dimensions. (Of course, you might find that it helps your understanding to write non-vectorized code at first, but make sure that you optimize your code by vectorizing it before you turn it in.)

Here are some step-by-step suggestions, but you don't need to use them, as long as you get that gradient computation right.

- Start by running *a2(0, 7, 10, 0, 0, false, 4)*, and you'll see the gradient checker complaining.
- Then implement the classification loss gradient, and if you get any error message from the gradient checker, look closely at the numbers in that error message. When you have a correct implementation, proceed to the next part. Don't worry about the weight decay loss gradient for now: it's zero anyway.
- If you're completely at a loss about how to implement the error backpropagation algorithm in Matlab, here are two suggestions. Suggestion 1: think about it some more, on paper, without thinking about computer programming (this may involve reviewing lecture videos). Suggestion 2: if you're not getting

anywhere, review this example that we studied in class, review the code of the January 23 tutorial, and come to office hours.

Now run *a2(0, 10, 30, 0.01, 0, false, 10)* and report the resulting training data classification loss.

Part 3: Optimization

We'll start with a small version of the task, to best see the effect of the optimization parameters. The small version is that we don't use regularization (i.e. weight decay or early stopping), we use 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 we're investigating how the optimization works best, we'll only look at the loss on training data. That's what we're directly optimizing, so if that gets low, then the optimizer did a good job, regardless whether the solution generalizes well to the validation data.

Let's do an initial run with with learning rate 0.005 and no momentum: run *a2(0, 10, 70, 0.005, 0, false, 4)* The training data loss that that run reports at the end should be 2.301771.

In the plot you'll see that training data loss and validation data loss 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. If we would run our optimization for an hour (i.e. many iterations), then when we get a bigger task and bigger network, the program might take a lot longer than an hour to do the same number of optimization iterations.

Let's 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 the learning rate times negative the gradient to the model parameters at every iteration, but with momentum, we use a more sophisticated strategy: we keep track of the momentum **speed**, using *new speed = old speed* * lambda - *the gradient*, and then we add the speed times the learning rate to the model parameters. That lambda (a.k.a. the momentum multiplier, a.k.a. the viscosity constant) can be anything between 0 and 1, but usually 0.9 works well. Setting it to zero means that we're not using momentum (check that for yourself, mathematically).

Let's try 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. We'll try 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), so we have a total of 7 x 2 = 14 experiments to run. Remember, what we're interested in right now is the loss on the training data, because that shows how well the optimization works. Which of those 14 worked best? See if you can fine-tune the learning rate a little better still, to get even better performance.

Part 4: Generalization

This part builds on material in videos 9a and 9b, which we'll discuss in class on February 27. We recommend that you postpone this last part until after that class. However, do look at the end of the assignment now, for a description of what you're expected to write. That way, you can finish most of the assignment before February 27.

First, you'll have to finish the gradient computation, to also work when the parameter wd_coefficient is not zero. Again, the gradient checker should come in handy for this.

Now that we found good optimization settings, we're switching to a somewhat bigger task, and there we'll investigate generalization. Now we're interested mostly in the classification loss on the validation data: if that's good, then we have good generalization, regardless whether the loss on the training data is small or large. Notice that we're measuring only the classification loss: we're not interested in the weight decay loss. The classification loss is what shows how well we generalize. When we don't use weight decay, the

classification loss and the final loss are the same, because the weight decay loss is zero.

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 *a2(0, 200, 1000, 0.35, 0.9, false, 100)*. This run will take more time. The validation data classification loss should now be 0.413518.

The simplest form of regularization is early stopping: we use the weights as they were when validation data loss was lowest. You'll see in the plot that that 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. Report what the validation data classification loss is now, i.e. **with** early stopping.

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. Try 0.01, and see if it's better than 0. Explore with different values to find an even better setting. Report the best value that you found, and report the classification loss for that value. Be careful to focus on the **classification loss** (i.e. without the weight decay loss), as opposed to the total loss (which does include the weight decay loss).

Yet another regularization strategy is reducing the number of model parameters, so that the model simply doesn't have the brain capacity to overfit a lot by learning too many details of the training set. 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 a variety of hidden layer sizes. Explore, again based on the classification loss. Indicate which one worked best, and what the classification loss is for that hidden layer size.

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? Is it very different from the best value without early stopping? Remember, **best**, here, is based on only the validation data loss.

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've quite 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, report the test data classification error rate.

What to hand in

- Hand in your implementation of the function d_loss_by_d_model. Don't hand in any other code.
- Do write a short description of how you implemented the backpropagation algorithm. Part 2 of the assignment is worth 50% of the mark of this assignment (half of that is for correctness; the other half is for writing something about it). Your description doesn't need to be long, and it doesn't need to be a complete recipe. We just want some of your observations about implementing backpropagation.
- Also write a bit about your experiences in part 3. For example, how do momentum and learning rate seem to interact, if at all? Part 3 is worth 25%.
- Part 4 is worth the remaining 25%. Answer the specific questions in the text above, and write about the interaction, and anything else that you may happen to notice.
- When you're asked to find a good setting for e.g. weight decay or learning rate, you're not expected to try a million different values and find the absolute best one. Try something like 10 different values (or 20 if you're feeling ambitious), but choose them wisely: if you find that small values typically work better than large values, focus on exploring small values.

- If you're feeling extra ambitious, you can try to improve the optimizer with other methods, or add other types of regularization. The optimizer would probably work better with rmsprop, and regularization would work better with dropout. By trying something like this, and properly reporting on what you tried and what happened, you won't be able to get a mark of more than 100% for the assignment, but we will grade that extra work up to a maximum of an extra 10%, which could make a difference if you lose marks in other parts. However, these suggestions are more intended as an invitation to creativity than as part of the graded coursework.
- Where you're asked to report some output from the program, provide all digits that Matlab gives you: don't round.
- The written part of what you hand in should be between one and two pages. More than two pages will not be graded. However, we don't count figures and program code as part of those two pages.