CS 189:289A Introduction to Machine Learning
Spring 2017

Homework 6: Neural Nets
Due Fri., April 14

Instructions

- We prefer that you typeset your answers using LaTeX or other word processing software. Neatly handwritten and scanned solutions will also be accepted.
- Please make sure to start **each question on a new page**, as grading (with Gradescope) is much easier that way!
- Deliverables. Submit a **PDF of your writeup** to the Homework 6 assignment on Gradescope. Include your code in your writeup in the appropriate sections. Submit your **code zip** and a README to the Homework 6 Code assignment on Gradescope. Finally, submit **your predictions** for the test sets to Kaggle. Be sure to include your Kaggle display name and score in your writeup.
- Due **Friday, April 14, 2017 at 11:59 PM.**

Contents

1 Assignment Overview

In this assignment, you will implement a two-layer fully-connected neural network from scratch and train it on a dataset of handwritten letters. We highly recommend that you start early, as there is no skeleton code provided and neural networks can take hours to train if your code is not optimized or your hyperparameters are not well-tuned. (On the other hand, if your code is well optimized and you've chosen good hyperparameters, this particular dataset and net configuration will take a couple of seconds to train.)

2 Dataset

You will be training on a dataset of uppercase and lowercase handwritten letters. Each sample is a 28×28 grayscale image which has been flattened to be a 784-length vector. There are 26 classes (a–z). The training split contains 124,800 images, and the test split contains 20,800 images.

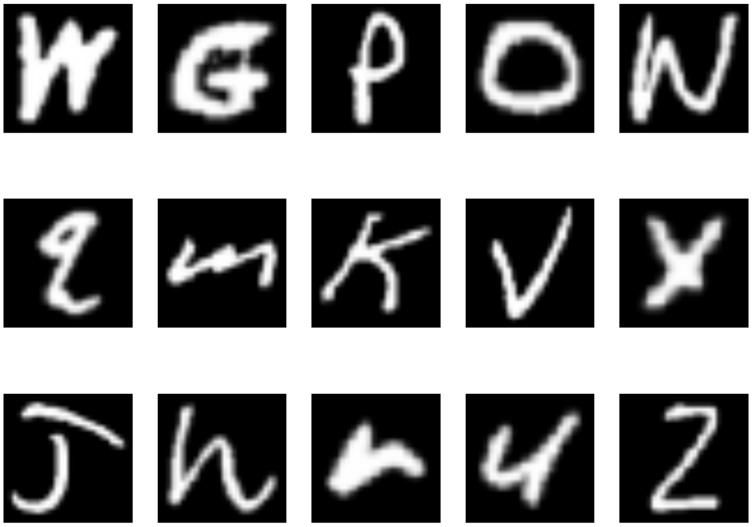

Figure 1: Examples from the training split

3 Neural Network Architecture

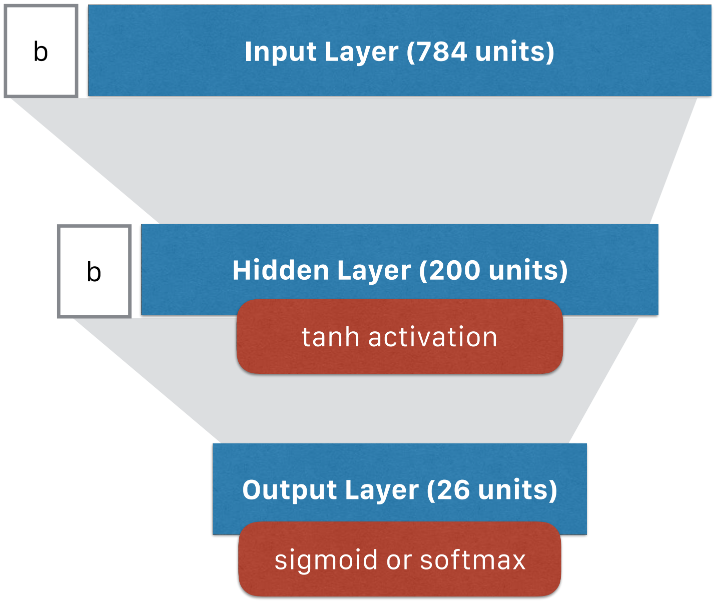

Figure 2: Architecture of the neural net you will be building

You will be implementing what's called a *two-layer fully-connected neural network*, which means there are two layers of edges/weights and three layers of units/neurons: the input layer (layer 0), the hidden layer (layer 1), and the output layer (layer 2).

3.1 Layers

1. Your input layer will have $d = 784$ ordinary input units, plus one fictitious unit always set to "1" for the bias terms.
2. Your hidden layer will have 200 units, plus one fictitious unit always set to "1" for the bias terms. (You can tweak the number of hidden units later.)
3. The output layer has 26 units, which is the number of classes. Each label will have to be 1-hot encoded (transformed to a vector of length 26 which has a single 1 in the position of the true class and 0 everywhere else).

3.2 Parameters

You will train two weight matrices.

1. V is a 200×785 matrix that connects the input units to the hidden units. The last column of V is made up of bias terms, which are multiplied by 1 instead of an input value.

2. W is a 26×201 matrix that connects the hidden units to the output units. The last column is bias terms.

3.3 Activation functions

Without activation functions, a neural network is just a linear model.

1. Use the tanh (hyperbolic tangent) activation function on the hidden units.
2. For the output units, you have a choice. The easier choice is the sigmoid activation function

$$s(\gamma) = \frac{1}{1 + e^{-\gamma}}.$$

   If you wish, you may use softmax output units. If you're not very secure in both your math and programming skills, we recommend you get your neural network completely working and debugged with sigmoid units before you try softmax units.

3.4 Loss function

To penalize outputs, use the cross-entropy loss function.

If you use the sigmoid activation function for the output units, the loss function is

$$L(z, y) = -\sum_{j=1}^{26} y_j \ln z_j + (1 - y_j) \ln(1 - z_j),$$

where y_j is the correct label of unit j (for one specified sample) and z_j is the output predicted by output unit j. (We view the problem as 26 binary classification tasks.)

If you use the softmax activation function for the output units, the loss function is

$$L(z, y) = -\sum_{j=1}^{26} y_j \ln z_j$$

4 Writeup Requirements

4.1 Problem 1: Derivations

Derive the stochastic gradient descent for V and W. To do this, you must derive the partial derivatives of L with respect to V and W in the form they are needed for the backpropagation algorithm, with particular attention to the tanh units in the hidden layer. Write the results in as simple a form as you reasonably can. Please define all the notation you use in your derivation and show your work!

4.2 Problem 2: Implementation

Set aside 80% of the labelled data as your training split and 20% as your validation split.

Implement and train your two-layer fully-connected neural network. You must implement this neural network from scratch, using only basic linear algebra libraries like Numpy and Scipy. You are **not allowed** to use any framework that performs backpropagation, including but not limited to: Tensorflow, Caffe, Theano, Torch, Keras, etc. That would defeat the point of this assignment.

Although you will have the opportunity to add tweaks and optimizations to your neural network (see Problem 4 below), we **very strongly recommend** that you start with the simplest implementation possible within our specifications above and fully debug it before you get ambitious.

Predict the labels on the test data and submit your results to Kaggle. Include the following in your writeup.

1. Any hyperparameters that you tuned.
2. Your training accuracy.
3. Your validation accuracy.
4. A plot of the loss value versus the number of iterations. You may sample (i.e., compute the loss every x iterations).
5. Your Kaggle score and display name.
6. Your code (as an appendix at the end).

Expect validation accuracies of 85% or higher. Your TAs are able to achieve 87.3% validation accuracy after 1 minute of training on a MacBook Pro. With some tweaks (800 hidden units, ReLU hidden layer activation, softmax output layer activation), your TAs are able to get 90.17% validation accuracy after 3 minutes of training.

4.3 Problem 3: Visualization

Visualize 5 digits (and their labels) from the validation set that your neural network correctly classifies and 5 digits (and their labels) from the validation set that your neural network does not correctly classify.

4.4 Problem 4: Bells and Whistles

In lecture, we covered many variations and ways to improve the learning speed and accuracy of neural networks. After you have implemented the basic neural network as we have prescribed, you may go above and beyond to improve your network for your Kaggle submissions. As with Problem 2, you must implement any additional functionality yourself. (E.g., you **cannot** train a neural network with Tensorflow and use that.) In this section of your writeup, report any extra functionality or tricks you implemented. Some ideas:

- Use different learning rates for different layers, and have those rates decay over time. (Highly recommended. See Section 5.4.)
- Change the number of hidden layer units, or even add one or more additional hidden layers.
- Use different activation functions. ReLUs for the hidden layer and softmax output units can speed up training and improve accuracy.
- Implement mini-batch gradient descent. This will dramatically speed up your training time.
- Regularization: both L_2 regularization and dropout are surprisingly easy to implement, and tend to reduce overfitting.
- Momentum, different initialization schemes, an ensemble of neural networks, etc. See the "Implementation Tips" section for ideas.

5 Implementation Tips

5.1 Suggested Structure

This is the basic gist of how to structure your code.

```
def trainNeuralNetwork(images, labels, params*)
    images: training images ($X$)
    labels: training labels ($y$)
    params: hyperparameters, e.g., learning rate $\epsilon$, weight decay rate $\lambda$ for $L_2$ regularization, etc.

Initialize the weights $V$ and $W$ somehow (not zero!)
while (some stopping criterion)
    pick one image/label pair $(X_i, y_i)$ at random from the training set
    perform forward pass (compute hidden & output values and predicted labels)
    perform backward pass (compute partial derivatives needed for gradient descent)
    perform stochastic gradient descent update
store $V, W$
```

```
def predictNeuralNetwork(images, $V$, $W$)
    images: test images
    $V$, $W$: network weights (previously trained)

for each test image $x$
    with $V$ & $W$, perform forward pass (compute hidden & output values and predicted labels)
return all the predicted labels
```

5.2 Stopping Criteria

There are several options for the stopping criterion.

1. Set a maximum number of iterations.

2. Stop when the training loss stops falling, or the gradients are always very small. Because you are doing stochastic gradient descent, the loss will be quite stochastic from iteration to iteration, so be careful how you set up this criterion.

3. Stop when the validation error stops falling. (This has the same issue with stochasticity as criterion 2).

4. Stop when you feel like it and save the neural net model so you can resume training (see checkpointing). This is the most popular in practice!

5.3 Preprocessing Data

You should center and normalize all your features. You are encouraged to try other preprocessing methods. Remember to **shuffle** the data! If data is not shuffled and your neural network sees all the A's first, the network will not learn anything because at first it will simply learn to always output A; then it will forget that and learn to always output B, and so on.

The mean vector that you subtract from the training data must be the same as the vector you subtract from the test data! This also applies to the values you divide by to normalize. Failure to do this means the network will see test data as slightly shifted from training data.

5.4 Step Size and Convergence

The *single most important parameter to tune* is the learning *schedule*. This is a combination of learning rate (step size) and how your learning rate decays over time. The learning rate cannot be too small or too large. With a learning rate that is too small, training will be slow and stochastic gradient descent can easily get trapped in bad local minima. With a learning rate that is too large, training will fail to converge. You should reduce your learning rate every so often. It's typical to scale the learning rate by some constant hyperparameter, typically between 0.5 and 0.9, every constant number of epochs. (An *epoch* is one complete pass over the shuffled data, constituting n iterations.)

As discussed in lecture, it is wise to use different learning rates for different layers of units, because for sigmoids and tanh units, the partial derivatives we compute tend to be smaller in the early layers than in the late layers. (If you switch to ReLUs, you will probably need to retune the learning rates.)

5.5 Initialization of Weights

Make sure you initialize your weights with random values. This breaks the symmetry that occurs when all weights are initialized to zero. The initialization method has a strong effect on the number of iterations it takes to reach convergence. The most basic method is to initialize from a Gaussian or uniform distribution with mean 0 and variance $\sigma^2 \propto 1/\eta$, where η is the fan-in of the neuron the weight is an input to.

More complex neural networks use other methods of initialization which you are free to look up. These might not be useful for the shallow fully-connected neural network you will be implementing, so more advanced initializations schemes might have little effect.

5.6 Vectorization

Vectorize your code when possible. (That means using Numpy library functions rather than relying on "for" loops.) This can save you hours of training time. Make sure you your backpropagation updates are matrix and vector computations such as matrix-matrix multiplication or element-wise operations. Neural networks will take much longer than other assignments to train if your operations are not optimized. It is ideal to implement the *entire* assignment with only a single "for" loop that loops over gradient descent iterations.

5.7 Gradient Checking

Backpropagation is tricky to implement correctly. If your network is failing to train and you have looked carefully at your code and tried tuning the learning schedule, we suggest verifying the gradients your code computes. You can do this by comparing its output to gradients estimated by finite differences. Let $f : \mathbb{R}^d \to \mathbb{R}$ be an arbitrary function (for example, the loss function of a neural network with d weights and biases), and let a be a d-dimensional vector. You can approximate each partial derivative of f as

$$\frac{\partial f}{\partial x_k}(a) \approx \frac{f(\dots, a_k + \epsilon, \dots) - f(\dots, a_k - \epsilon, \dots)}{2\epsilon}$$

where ϵ is a small constant. As ϵ approaches 0, this expression theoretically approaches the true derivative, but on a computer it becomes prone to numerical precision issues. The choice $\epsilon = 10^{-5}$ works well. This technique is extremely slow, so don't forget to remove it from your code after you've used it.

5.8 Checkpointing

As training neural networks takes lots of time, we encourage you to write your code so that it saves its progress in a file when you terminate the program (see the Python `signal` package for how to capture an interrupt signal) and/or after every fixed number of iterations, and to write code that loads this file and allows you to resume training. That way, you can terminate your program safely, save multiple models, etc.

You might find it useful to look into the Python `pickle` module, or `numpy.save`. This allows you to save and load arbitrary Python or numpy objects (such as your neural network weights) as files.

5.9 Data Augmentation

A way to improve your Kaggle score is to implement data augmentation, where you artificially generate new data points by slightly jittering, stretching, and/or skewing your training data.

5.10 Ensembles

As we found with decision trees and random forests, an ensemble of models can reduce variance and improve accuracy. You can train multiple neural networks and *ensemble* their predictions. This means that given a test data point, the predicted label is a majority vote of multiple neural networks.

5.11 Mini-Batch Gradient Descent

The single best way to speed up your code is to use mini-batch gradient descent instead of stochastic gradient descent. In mini-batch gradient descent, you sample k data points instead of one data point and average the gradient update over those data points. If $k = n$ this becomes batch gradient descent. Typically k is some number between 16 and 256 (50 is a good starting point). Note that larger batches use more memory.

5.12 Momentum

Another useful trick is momentum, where you add a fraction of the previous gradient update to the current gradient update, as described in lecture. If you're adventurous, you might attempt to implement batch normalization.

Remember that if you implement momentum, you will need to adjust your learning rate to compensate. As momentum increases the magnitude of your step size, you have to decrease the learning rate.

5.13 Regularization

To reduce overfitting, you might try L_2 regularization (weight decay) or dropout (`https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf`). However, these tend to be more useful for nets with more than one hidden layer.

The easiest way to implement dropout is to simply zero out the value of each dropped unit after it's been computed, on both the forward and backward passes. (It may seem wasteful to compute a value you're just going to zero out immediately, but it means fewer changes to your code and doesn't disturb the vectorization.)

5.14 Convolutional Layers

The truly adventurous may attempt to implement convolutional layers as early hidden layers. (The last hidden layer, possibly the last two, should be ordinary, fully-connected hidden layers.) The backpropagation update is a bit tricky to get right. Here's a resource we recommend: `http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/`