CS 5600/6600: Intelligent Systems
Assignment 4

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 20, 2019

Learning Objectives

1. Training and Testing ANN Architecture
2. Learning Slow Down
3. Overfitting
4. Regularization

Introduction

This assignment will give you a flavor of training larger ANNs systematically and dealing with overfitting, learning slowdown, through cross-entropy, normalization and systematic evaluation of different ANN architectures. I hope you can use the tools and insights you'll develop in this assignment when you work on Project 1.

Problem 1 (5 pts)

The class `Network` defined in `ai_f18_hw04.py` is another implementation of ANN that allows us to experiment with two different cost functions: MSE and cross-entropy. The MSE is implemented in the class `QuadraticCost`. The cross-entropy function is defined in `CrossEntropyCost`. The method `__init__()` has a keyword parameter that allows you to set a cost function to be used in training and testing a specific ANN.

To begin with, let's load the MNIST training, validation, and testing data using the function `load_data_wrapper` from `mnist_loader.py`, as we did in the previous assignment.

```
train_d, valid_d, test_d = load_data_wrapper()
>>> len(train_d)
50000
>>> len(valid_d)
10000
>>> len(test_d)
10000
```

The member function `SGD` of the `Network` class trains its network object using stochastic gradient descent. Unlike the `SGD` function in the previous version of the network class you worked with in the previous assignment, this method allows you to pass the value of the λ argument used in *L2 regularization*. Regularizatsion is a way to reduce overfitting. A *regularization term* is added to the cost function as follows.

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2,$$

where $\lambda > 0$ is the regularization parameter and n is the size of the training data set. Note that the regularization term does not include the biases. When λ is small, preference is given to minimizing the cost function, when λ is larger, preference is given to finding smaller weights.

The `SGD` function takes the training data given in `training_data`, the number of epochs specified by `epochs`, the size of the mini-batch specified by `mini_batch_size`, and the learning rate given by `eta`. If the evaluation data is given by the keyword argument `evaluation_data`, then these data are used estimate the network's accuracy after each epoch.

The function returns a 4-tuple of lists. The first list is a list of evaluation cost values: one value per epoch. The second list is a list of evaluation accuracy values: one value per epoch. The third list is a list of training cost values: one value per epoch. The fourth list is a list of training accuracy values: one value per epoch.

Let's create a 784x30x10 network with the cross-entropy cost function and train it on `train_d` for just 2 epochs with a mini-batch size of 10, a η of 0.5, a λ of 5.0. We will use `valid_d` as our evaluation data. In other words, these data are used to evaluate the evaluation cost and accuracy after each epoch.

```
>>> import network2
>>> net = network2.Network([784, 30, 10], cost=network2.CrossEntropyCost)
>>> net_stats = net.SGD(train_d, 2, 10, 0.5, lmbda=5.0,
                        evaluation_data=valid_d,
                        monitor_evaluation_cost=True,
                        monitor_evaluation_accuracy=True,
                        monitor_training_cost=True,
                        monitor_training_accuracy=True)
Epoch 0 training complete
Cost on training data: 1.32348339529
Accuracy on training data: 45866 / 50000
Cost on evaluation data: 4.33465283079
Accuracy on evaluation data: 9190 / 10000

Epoch 1 training complete
Cost on training data: 0.920509073127
Accuracy on training data: 46807 / 50000
Cost on evaluation data: 2.90920686448
Accuracy on evaluation data: 9358 / 10000
```

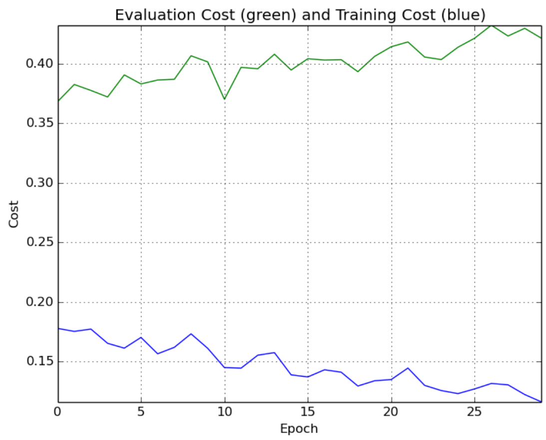
Let's inspect the contents of `net_stats`.

```
>>> net_stats
([4.3346528307886603, 2.9092068644832847],
 [0.919, 0.9358],
```

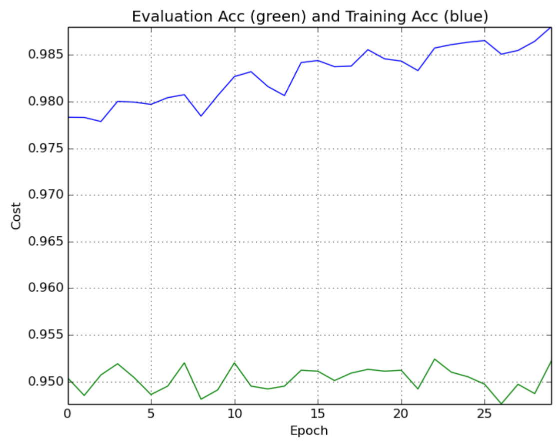
```
[1.3234833952859093, 0.92050907312657826],
[0.91732, 0.93614])
```

The first list contains 2 evaluation cost values (i.e., 1 value for each of the two epochs). The second list has 2 evaluation accuracy values (i.e., 1 value for each of the two epochs). The third list has 2 training cost values (i.e., 1 value for each of the two epochs). The fourth list has 2 training accuracy values (i.e., 1 value for each of the two epochs).

Let's develop two graphing tools you can use to estimate how well your net is training. The first tool is the function `plot_costs(eval_costs, train_costs, num_epochs)` that takes the first and third elements of the 4-tuple returned by `net.SGD()` and the number of epochs over which the ANN was trained with `SGD()` and returns a graph where the evaluation and training costs are plotted as the green and blue lines, respectively. Here is a graph generated by this function for an ANN trained over 30 epochs.



The second tool is a function `plot_accuracies(eval_accs, train_accs, num_epochs)` that takes the second and fourth elements of the 4-tuple returned by `net.SGD()` and the number of epochs over which the ANN was trained with `SGD()` and returns a graph where the evaluation and training accuracies are plotted as the green and blue lines, respectively. Here is a graph generated by this function for the same ANN trained over 30 epochs.



Let's now develop a few functions that will allow us to do systematic training of ANNs on MNIST. Write the function

```
collect_1_hidden_layer_net_stats(lower_num_hidden_nodes,
                                 upper_num_hidden_nodes,
                                 cost_function,
                                 num_epochs,
                                 mbs,
                                 eta,
                                 lmbda,
                                 train_data,
                                 eval_data)
```

Let the first two parameters have the values of l and u, respectively. The first two parameters specify the lower and upper bounds of the parameter n in a $784 \times n \times 10$ ANN for MNIST, where $l \leq n \leq u$. The values specified by the other parameters are evident from their names. In particular, `mbs` specifies the minimum batch size, `eta` is η, and `lmbda` is λ. We cannot use `lambda` as a parameter, because it's a Py keyword. This function returns a dictionary where the keys are the values of n and the values are the 4-tuples returned after training a created $784 \times n \times 10$ ANN with the values specified by parameters 3 through 9.

Let's go through an example. Let's train two networks ($784 \times 10 \times 10$ and $784 \times 11 \times 10$). The training stats for the first network are filed away under key `10` in the returned dictionary. The training stats for the second network are saved under key `11`.

```
>>> d = collect_1_hidden_layer_net_stats(10, 11,
                                         network2.CrossEntropyCost,
                                         2, 10, 0.1, 0.0,
                                         train_d, test_d)
>>> d
{10: ([1.359573119521807, 0.99817263661946443],
      [0.15252, 0.16918],
      [1.3932450994786085, 1.0170595802578979],
      [0.75024, 0.83708]),
 11: ([1.370898305189044, 0.95761536737711583],
      [0.14944, 0.16774],
      [1.3981990807392013, 0.96290041988665143],
      [0.73996, 0.84036])}
```

After you implement this function, use it to find and pickle the best $784 \times n \times 10$ ANN for MNIST, where $30 \leq n \leq 100$ and the training is done over 30 epochs. Pickle this ANN as `net1.pck`. Use your graphing functions to detect overfitting. Experiment with various values of λ and η. You may want to be systematic with λ and η values and write an auxiliary function that loops over λ and η ranges and calls `collect_1_hidden_layer_net_stats` for each possible combination of λ and η from the specified ranges.

Let's now write the same tool for 2-layer ANNs. Write the function

```
collect_2_hidden_layer_net_stats(lower_num_hidden_nodes,
                                 upper_num_hidden_nodes,
                                 cost_function,
                                 num_epochs,
                                 mbs,
                                 eta,
                                 lmbda,
                                 train_data,
                                 eval_data)
```

This function returns the same data structure as `collect_1_hidden_layer_net_stats` by creating and testing all $784 \times n_1 \times n_2 \times 10$ MNIST ANNs where $l \leq n_1, n_2 \leq u$. The first two parameters specify the values of l and u, respectively. This function constructs a dictionary where the keys are strings of the form `'x_y'`, where x and y are specific values of n_1 and n_2. Here is a sample test.

```
>>> d = collect_2_hidden_layer_net_stats(2, 3,
                                          network2.CrossEntropyCost,
                                          2, 10, 0.1, 0.0, train_d, valid_d)
>>> d['2_2']
([2.8617112909670452, 2.761559872222534],
 [0.2331, 0.2189], [2.860738768965704, 2.7646677631818055],
 [0.24584, 0.22082])
>>> d['2_3']
([2.5370767499523663, 2.3592176429391434],
 [0.3564, 0.3937],
 [2.5490976212297496, 2.372676815181264],
 [0.35792, 0.39044])
>>> d['3_2']
([2.819447702054708, 2.7555993049376193],
 [0.1979, 0.2588],
 [2.833450300330617, 2.7643062314776534],
 [0.20744, 0.2695])
>>> d['3_3']
([2.4301053165922357, 2.031793281662343],
 [0.3853, 0.5305],
 [2.429567226198757, 2.0365565490753634],
 [0.38722, 0.53174])
```

Use this function to find and pickle the best $784 \times n_1 \times n_2 \times 10$ ANN for MNIST, where $30 \leq n_1, n_2 \leq 100$ and the training is done over 30 epochs. Pickle this ANN as `net2.pck`. Use your graphing functions to detect overfitting. Experiment with various values of λ and η. Again, you may want to be systematic with λ and η values and write an auxiliary function that loops over λ and η ranges and calls `collect_1_hidden_layer_net_stats` for each possible combination of λ and η from the specified ranges.

Finally, let's systematically tackle 3-layer ANNs. Write the function

```
collect_3_hidden_layer_net_stats(lower_num_hidden_nodes,
                                 upper_num_hidden_nodes,
                                 cost_function,
                                 num_epochs,
                                 mbs,
                                 eta,
                                 lmbda,
                                 train_data,
                                 eval_data)
```

This function returns the same data structure as its two above counterparts. This function creates and tests all $784 \times n_1 \times n_2 \times n_3 \times 10$ MNIST ANNs where $l \leq n_1, n_2, n_3 \leq u$. The first two parameters have the values of l and u, respectively. This function constructs a dictionary where the keys are strings of the form `'x_y_z'`, where x, y, and z are legitimate values of n_1, n_2, and n_3, respectively. Here is a sample test.

```
>>> d = collect_3_hidden_layer_net_stats(2, 3,
                                        network2.CrossEntropyCost,
                                        2, 10, 0.1, 0.0, train_d, valid_d)
>>> d['2_2_2']
([2.616874650046163, 2.317136038701688],
 [0.3807, 0.4025],
 [2.633321254651256, 2.324513198743874],
 [0.3772, 0.40258])
>>> d['2_3_3']
([2.441494271658661, 2.2311953726499225],
 [0.3835, 0.3855],
 [2.438624182307029, 2.2245470998628067],
 [0.38602, 0.39346])
>>> d['3_3_3']
([2.1731375457234785, 1.7794279308657845],
 [0.5595, 0.5871],
 [2.201013781357794, 1.8243996186268596],
 [0.5458, 0.57372])
```

Use this function to find and pickle the best $784 \times n_1 \times n_2 \times n_3 \times 10$ ANN for MNIST, where $30 \leq n_1, n_2, n_3 \leq 100$ and the training is done over 30 epochs. Pickle this ANN as `net3.pck`. Use your graphing functions to detect overfitting. Experiment with various values of λ and η. Again, you may want to be systematic with λ and η values and write an auxiliary function that loops over λ and η ranges and calls `collect_1_hidden_layer_net_stats` for each possible combination of λ and η from the specified ranges.

What to Submit

1. `cs5600_6600_f19_hw04.py` with your code at the end: there are five function stubs at the end of this file that you need to complete. Write in your comments the best performance of your `net1.pck`, `net2.pck`, and `net3.pck`.

2. zipped directory `pck_nets` with `net1.pck`, `net2.pck`, and `net3.pck`.

Happy Hacking!