CS 5600/6600: F20: Intelligent Systems
Assignment 4

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 26, 2020

Learning Objectives

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

Introduction

This assignment will give you an opportunity to train larger ANNs systematically and deal with overfitting and learning slowdown through cross-entropy, normalization and systematic evaluation of different ANN architectures. My objective is to give you some conceptual and programmatic tools and insights to work on Project 1. You may want to review lectures 6 and 7, especially the sections on learning slowdown, overfitting, and regularization, before working on this assignment.

Paper Analysis (2 points)

Read and write a one-page analysis on "ImageNet Classification with Deep Convolutional Neural Networks" by Krizhevsky, Sutskever, and Hinton. This is a relatively recent paper. It was published in 2012 and revived image classification research on large datasets. It was one of the papers that rejuvinated the research on deep learning in image classification. The paper discusses many practical issues that are applicable both to ANNs and ConvNets: overfitting, regularization, dropout,and GPU training. Pay attention to the results section and ask yourself impartially how much trust you, as a researcher, place in them. Save your writeup in `cs5600_6600_F20_hw04_paper.pdf` and submit it in Canvas.

Problem 0 (0 pts)

Let's start with a coding lab and load the MNIST training, validation, and testing data, as we did in Assignment 3.

```
>>> from mnist_loader import *
train_d, valid_d, test_d = load_data_wrapper()
>>> len(train_d)
50000
>>> len(valid_d)
```

```
10000
>>> len(test_d)
10000
```

Take a look at the class `ann` in `ann.py`. This class is an augmented 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 `ann.__init__()` has a keyword parameter for setting a cost function to be used in training, testing, and validating a given ANN. The member function `ann.mini_batch_sgd()` trains its network using stochastic gradient descent. However, unlike the `ann.mini_batch_sgd()` function we worked with in Assignment 3, this function allows us to pass the value of the λ argument used in *L2 regularization*. Recall from lecture 7 that 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 doesn't 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 function `ann.mini_batch_sgd()` 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 to estimate the ann's accuracy after each epoch.

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

Let's create a 784x20x10 ann with the cross-entropy cost function and train it on `train_d` for 5 epochs with a mini-batch size of 10, a η of 0.5, a λ of 5.0. We'll use `valid_d` as our evaluation data. In other words, this dataset will be used to evaluate the cost and accuracy of the trained network after each epoch. This test is implemented in `test_ut01()` in `cs5600_6600_f20_hw04_unit_tests.py` as follows. Note that the keyword `lmbda` is not a misspelling, because `lambda` is a Python keyword.

```
def test_ut01(self):
    global train_d
    net = ann([784, 20, 10], cost=CrossEntropyCost)
    net_stats = net.mini_batch_sgd(train_d,
                                   5, 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)
    plot_costs(net_stats[0], net_stats[2], 5)
```

When you run this unit test, you should see the output similar to my output below. The exact numbers will (most likely) be different.

```
Epoch 0 training complete
Cost on training data: 1.137904453160612
Accuracy on training data: 45259 / 50000
Cost on evaluation data: 3.2380033703254165
Accuracy on evaluation data: 9110 / 10000

...

Epoch 4 training complete
Cost on training data: 0.5813772094957639
Accuracy on training data: 46952 / 50000
Cost on evaluation data: 1.2649980670376755
Accuracy on evaluation data: 9387 / 10000
.
----------------------------------------------------------------------
Ran 1 test in 145.857s
OK
```

In the end, you should also see a plot similar to the plot in Fig. 1.

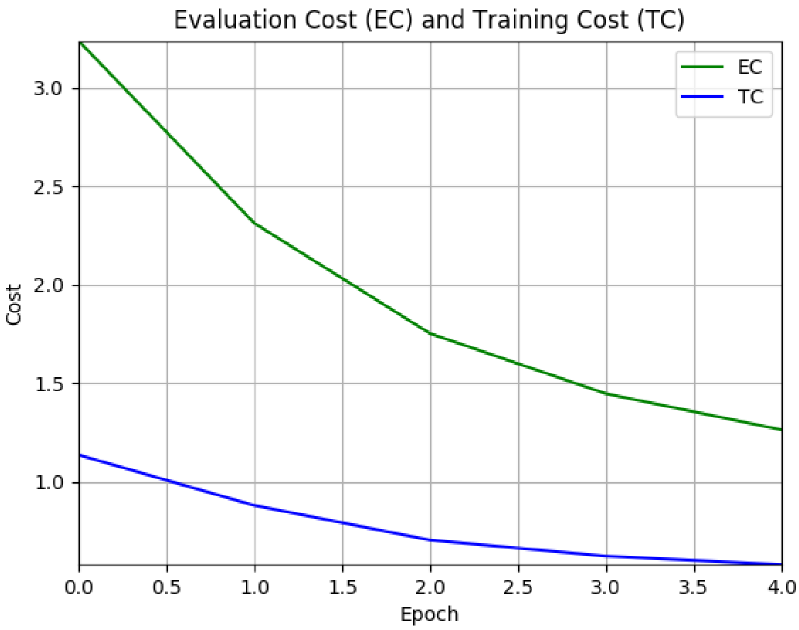

Figure 1: Evaluation and training costs from unit test 1.

Let's first inspect the contents of `net_stats` returned by `ann.mini_batch_sgd()`.

```
>>> 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).

There are two visalization functions I defined for you in `cs5600_6600_f20_hw04.py`: `plot_costs()` and `plot_accuracies()` to estimate how well your net is training. The function `plot_costs()` takes the first and third elements of the 4-tuple returned by `ann.mini_batch_sgd()` and the number of epochs over which the ANN was trained and returns a graph where the evaluation and training costs are plotted as the green and blue lines, respectively, as shown in Fig. 1. What's important about the plots in Fig. 1 is that the evaluation cost is going down, which means that the net is training. The evaluation cost should be approximating the training cost.

The function `plot_accuracies()` that takes the second and fourth elements of the 4-tuple returned by `mini_batch_sgd()` and the number of epochs over which the ANN was trained and returns a plot where the evaluation and training accuracies are plotted as the green and blue lines, respectively.

Let's create another 784x20x10 ann with the cross-entropy cost function and train it on `train_d` for 10 epochs with a mini-batch size of 10, a η of 0.5, a λ of 5.0. We'll use `valid_d` as our evaluation data. In other words, this dataset will be used to evaluate the cost and accuracy of the trained network after each epoch. This time we'll display the training and testing accuracies. This test is implemented in `test_ut02()` in `cs5600_6600_f20_hw04_unit_tests.py` as follows.

```
def test_ut02(self):
    global train_d
    net = ann([784, 20, 10],
              cost=CrossEntropyCost)
    net_stats = net.mini_batch_sgd(train_d,
                                   10, 10, 0.5, lmbda=0.5,
                                   evaluation_data=valid_d,
                                   monitor_evaluation_cost=True,
                                   monitor_evaluation_accuracy=True,
                                   monitor_training_cost=True,
                                   monitor_training_accuracy=True)
    plot_accuracies(net_stats[1], net_stats[3], 10)
```

My accuracy plot is shown in Fig. 2. Note that the evaluation accuracy curve is below the training accuracy curve, which is to be expected after 10 epochs. A more important thing is that the accuracy curve is edging upward. In general, we want the cost curve to edge downward and the accuracy curve to edge upward.

Problem 1 (3 points)

Let's now develop several that will allow us to collect some vitals of ANN training stats 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)
```

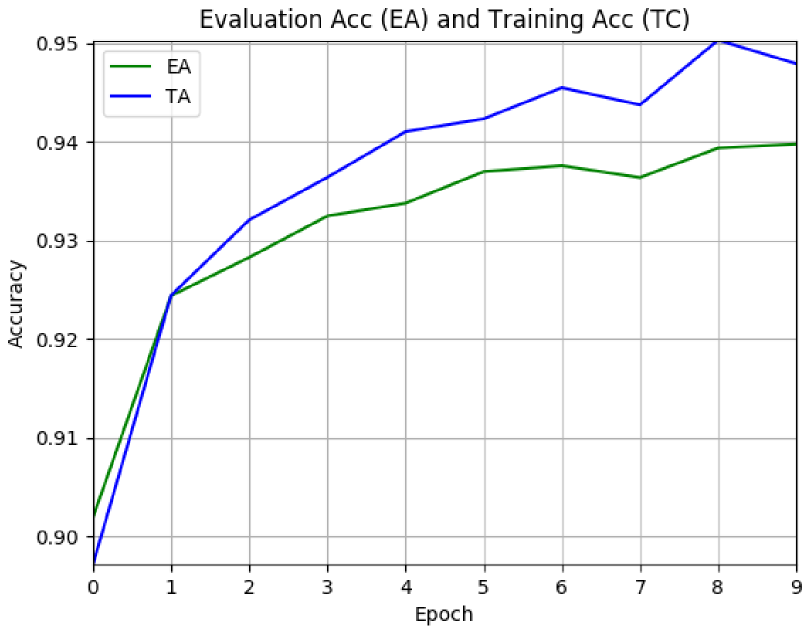

Figure 2: Evaluation and training accuracies from unit test 2.

Let's agree to refer to the values of the first two parameters as l and u, respectively. These 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. The parameter `mbs` specifies the minimum batch size, `eta` is η, and `lmbda` is λ. This function returns a dictionary where the keys are the values of n and the values are the 4-tuples returned after training and evaluating a created $784 \times n \times 10$ ANN on `train_data` and `eval_data` with the values specified by the parameters `cost_function`, `num_epochs`, `mbs`, `eta`, and `lmbda`.

Let's go through an example and train 2 nets ($784 \times 10 \times 10$ 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,
                                          CrossEntropyCost,
                                          2, 10, 0.1, 0.0,
                                          train_d, valid_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 save the best of your $784 \times n \times 10$ ANNs for MNIST, where $30 \leq n \leq 100$ and the training is done over 30 epochs. Save this ANN as `net1.json`. You can use `ann.save()` to persist your ANN into a JSON file. Use the plotting functions to visualize costs and accuracies. 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. The values of these ranges are up to you. I encourage you to experiment as hard as your hardware allows.

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's a sample test.

```
>>> d = collect_2_hidden_layer_net_stats(2, 3,
                                        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 persist the best of your $784 \times n_1 \times n_2 \times 10$ ANNs for MNIST, where $30 \leq n_1, n_2 \leq 100$ and the training is done over 30 epochs. Save this ANN as `net2.json`. Use plotting functions to monitor costs and accuracies. 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_2_hidden_layer_net_stats()` for each possible combination of λ and η from the specified ranges.

Finally, let's 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 counterparts above. The 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,
                                         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 save the best of your $784 \times n_1 \times n_2 \times n_3 \times 10$ ANNs for MNIST, where $30 \leq n_1, n_2, n_3 \leq 100$ and the training is done over 30 epochs. Save this ANN as `net3.json`. Use the plotting functions to visualize training and evaluation costs and accuracies. 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. Your one page paper analysis saved as `cs5600_6600_F20_hw04_paper.pdf`
2. `cs5600_6600_f20_hw04.py` with your code: there are three function stubs in this file that you need to complete. Describe in your comments the architectures, the costs and the accuracies of your best performing `net1.pck`, `net2.pck`, and `net3.pck`.
3. zipped directory `pck_nets` with `net1.pck`, `net2.pck`, and `net3.pck`.
4. Zip components 1, 2, 3 into `hw04.zip` ans submit it in Canvas.

Happy Reading, Writing, and Hacking!