CS 5600/6600/7890: Intelligent Systems
Assignment 3

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 14, 2019

Learning Objectives

1. Backpropagation
2. MNIST
3. Training and Testing ANNs
4. ANN Ensembles

Problem 1 (2 pts)

Forty three years after the seminal paper "A Logical Calculus of the Ideas Immanent in Nervous Activity" by McCulloch and Pitts, three researchers (David Rumelhart, Geoffrey Hinton, and Ronald Williams) published another seminal paper "Learning representations by back-propagating errors," where they formalized some of the ideas original articulated by McCulloch and Pitts. This paper was the first one to describe the modern version of backpropagation. Their paper is available here:

`https://www.academia.edu/2520405/Learning_representations_by_back-propagating_errors`.

Write a one page summary of this paper. Your summary should include four components: 1) a brief statement of the problem addressed in the paper; 2) what you liked in the paper and why; 3) what you did not like in the paper and why; 4) any inspirations you found in the paper. Each of these points should be addressed in a separate paragraph. Save your summary in `rumelhart_et_al.pdf`.

Problem 2 (4 pts)

Let's continue to play with MNIST. As we discussed in class, the file `mnist_loader.py` contains the function `load_data_wrapper` that loads the processed training, validation, and testing data for the MNIST dataset. This function unpacks the MNIST images and their targets from `data/mnist.pkl.gz`. Let's load the MNIST data and play with it. As I said in class, this code is Py2-compatible. If you don't have it installed on your machine, go to `www.python.org` and install it.

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

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

We have 50000 training images with targets, 10000 validation images with targets, and 10000 test images with targets. The `network.py` file has the class `Network`. We can use it to construct ANNs with arbitrarily many layers. Here is how we can construct an 784 x 30 x 60 x 120 x 10 ANN.

```
>>> net = Network([784, 30, 60, 120, 10])
```

Let's construct a 784 x 30 x 10 ANN and train it with Stochastic Gradient Descent (SGD) on the MNIST training data with the learning rate $\eta = 3.0$ and a mini-batch of 10.

```
>>> from mnist_loader import load_data_wrapper
>>> train_d, valid_d, test_d = load_data_wrapper()
>>> from network import Network
>>> net = Network([784, 30, 10])
>>> net.SGD(train_d, 20, 10, 3.0, test_data=test_d)
Epoch 0: 9112 / 10000
Epoch 1: 9164 / 10000
Epoch 2: 9310 / 10000
Epoch 3: 9332 / 10000
Epoch 4: 9347 / 10000
Epoch 5: 9385 / 10000
Epoch 6: 9354 / 10000
Epoch 7: 9409 / 10000
Epoch 8: 9426 / 10000
Epoch 9: 9421 / 10000
Epoch 10: 9459 / 10000
Epoch 11: 9475 / 10000
Epoch 12: 9473 / 10000
Epoch 13: 9476 / 10000
Epoch 14: 9484 / 10000
Epoch 15: 9472 / 10000
Epoch 16: 9477 / 10000
Epoch 17: 9483 / 10000
Epoch 18: 9488 / 10000
Epoch 19: 9489 / 10000
```

As you can see, I got 94.89% accuracy at the end of epoch 20, which is not bad. Your results, of course, may be different. But they should be in the same ballpark.

Define five ANNs that you will train on the MNIST data. At least one of the ANNs should have 6 layers, one – 5 layers, and one 3 – layers. The architectures of the remaining two ANNs are up to you. Of course, the input layer of each ANN must have 784 nodes and the output layer – 10 nodes.

Let's assume that you have defined your five networks and saved them in the variables `net1`, `net2`, `net3`, `net4`, and `net5`. Let's put them into a 5-tuple.

```
networks = (net1, net2, net3, net4, net5)
```

Let's define several η and mini-batch values that we'll use in training.

```
eta_vals = (0.1, 0.25, 0.3, 0.4, 0.5)
mini_batch_sizes = (5, 10, 15, 20)
```

Now define the function

`train_nets(networks, eta_vals, mini_batch_sizes, num_epochs, path),`

where `networks` is a tuple of ANNs, `eta_vals` is a tuple of η values, `mini_batch_sizes` is a tuple of minibatch sizes, `num_epochs` is the number of epochs over which each network in `networks` is trained with the SGD (you can use `Network.SGD()`, as shown above), and `path` specifies a path where the tuple of the trained networks should be persisted after training. Each network should be trained on `train_d` and tested after each epoch on `test_d`.

For each network `n` in `networks`, the function `train_nets` randomly chooses an η value from `eta_vals` and a mini-batch size from `mini_batch_sizes` and then trains the network `n` over the number of epochs specified by `num_epochs`. Here is a sample call.

```
>>> train_nets(networks, eta_vals, mini_batch_sizes, 30, '/home/vladimir/pck_nets/')
```

This sample call trains each network in `networks` over 30 epochs and persists each network in the directory `/home/vladimir/pck_nets/`. Each network should be saved in a pickle file whose name specifies its architecture, its η value, and the minibatch size with which it was trained. The η value should be multipled by 100 to avoid dots in file names. For example, uppose we have defined a 784 x 15 x 30 x 60 x 10 ANN and trained it with an η value of 0.1 and a mini-batch size of 10. Then, when trained, this ANN should be persisted in the file `net_784_15_30_60_10_10_10.pck`.

Define the function `load_nets(path)` that takes a path to the directory where the trained networks are persisted and returns a list of 2-tuples where each 2-tuple contains the string name of the file with the name of the network (e.g., `net_784_15_30_60_10_10_10.pck`) and the second element is the trained network loaded from that file. We'll use this function to load your nets from the pickle files and run unit tests on them.

Some AI/ML researchers and practioners argue that ensemble learning is more robust than individual learning. The idea is that instead of training a single model for a specific dataset, we train a bunch of different models (i.e., an ensemble) and use all of them to classify test data. The theory says that each model in the ensemble may be better than the rest on some data features. Therefore, when put together, an ensemble of models is more robust to noise and more accurate.

Define the function `evaluate_net_ensemble(net_ensemble, test_data)` that takes an ensemble of ANNs trained on MNIST images and a test data set of MNIST images (e.g., `valid_d`). This function applies each trained net to each sample and takes the majority vote of all the classifications. For example, if we have an ensemble of 5 networks, of which 3 networks classify a given image as 1, 1 network classifies it as 5 and 1 – as 0, then the whole ensemble classifies the image as 1. This function returns a 2-tuple whose first element is the number of accurately classified examples and the second element is the total number of examples. Here is a sample call that shows that an ensemble of networks defined by `nets` classified 9486 images correctly out of 10000 validation images.

```
>>> evaluate_net_ensemble(nets, valid_d)
(9486, 10000)
```

Problem 3 (3 pts)

Write a one page report that compares the performance of your ensemble of 5 ANNs on the validation data with each individual ANN on the same validation data. Some ANN researchers think that ANN ensembles perform better than individual ANNs. Their rationale is that even though all

ANNs are trained on the same data, their different learning rates and mini-batch sizes as well as the randomization of mini-batches make ANN ensembles more robust to data idiosyncrasies than any individual ANN. Those of you who are familiar with random forests may recall that it was the same argument that convinced researchers to move from individual decision trees to collections of randomly initialized and trained decision (aka random forests).

Your report should include either a table and/or a graph to compare the classification performance of your individual networks with the classification performance of your ensemble on the validation dataset of MNIST. Your report should also describe the architecture of each individual network in your ensemble, its learning rate, its mini-batch size, and the number of epochs over which you trained them.

What to Submit

1. `rumelhart_et_al.pdf` with your summary of the paper "Learning representations by back-propagating errors";
2. `cs5600_6600_f19_hw03.py` with your code;
3. `cs5600_6600_f19_hw03.py` with your code;
4. `cs5600_6600_f19_hw03.pdf` with your report;
5. zipped directory `pck_nets` with your five trained persisted networks.

Happy Hacking, Reading, and Writing!