# CS 5600/6600: F20: Intelligent Systems
# Assignment 3

Vladimir Kulyukin
Department of Computer Science
Utah State University

September 19, 2020

## Learning Objectives

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

## Problem 1 (1 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 a modern version of backpropagation. While many backprop derivatives have been proposed since then, the basic ideas are as applicable today as when they were originally proposed by Rumelhart, Hinton, and Williams. Their paper is available here:

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

Write a one page analysis of this paper. Your analysis 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. There's no problem if you go over 1 page, but don't write an essay. Save your analysis in `rumelhart_et_al.pdf`.

## Problem 2 (0 pts)

The purpose of this problem (it's really a lab!) is to prepare you for Problems 3 and 4 below. Let's start playing with the MNIST dataset that we discussed in Lecture 4. The file `mnist_loader.py` (included in the homework zip) 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` (also included in the zip). Let's load the MNIST data and play with it.

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

```
train_d, valid_d, test_d = load_data_wrapper()
>>> len(train_d)
50000 ### there are 50,000 training images
>>> len(valid_d)
10000 ### there are 10,000 validation images
>>> len(test_d)
10000 ### there are 10,000 testinging images
```

As the above interaction shows, we have 50,000 training images with targets, 10,000 validation images with targets, and 10,000 test images with targets. The `ann.py` file has the class `ann`. We can use it to construct ANNs with arbitrarily many layers. Here's how we can construct an `784x30x60x120x10` ANN.

```
>>> from ann import *
>>> net = ann([784, 30, 60, 120, 10])
>>> net
<ann.ann object at 0x7f57ca22afd0>
```

Let's construct a `784x30x10` ANN and train it with Stochastic Gradient Descent (SGD) on the MNIST training data with the learning rate $\eta = 3.0$, 20 epochs (0 – 19), and a mini-batch of 10.

```
>>> from mnist_loader import load_data_wrapper
>>> train_d, valid_d, test_d = load_data_wrapper()
>>> from ann import ann
>>> net = ann([784, 30, 10])
>>> net.mini_batch_sgd(train_d, 20, 10, 3.0, test_data=test_d)
Epoch 0: 8262 / 10000
...
Epoch 19: 9475 / 10000
```

As you can see, I got 94.75% accuracy at the end of epoch 20, which is not bad. Your results, of course, may be different but, I hope, in the same ballpark or better. This training took $\approx$3 minutes on my laptop.

Let's construct another network by reducing the hidden layer size (HLS) to 5 nodes, setting the learning rate $\eta$ to 0.5, and keeping the mini-batch size the same.

```
>>> net2 = ann([784, 5, 10])
>>> net2.mini_batch_sgd(train_d, 20, 10, 0.5, test_data=test_d)
Epoch 0: 6085 / 10000
...
Epoch 19: 8079 / 10000
```

This ANN hasn't done as well as the previous one. The recognition accuracy after 20 epochs is 80.79%. Let's train it for another 50 epochs.

```
>>> net2.mini_batch_sgd(train_d, 50, 10, 0.5, test_data=test_d)
Epoch 0: 8102 / 10000
...
Epoch 49: 8164 / 10000
```

I've managed to increase a recognition accuracy from 80.79% to 81.64%. Nothing to write home about, but, hey, it's a gain.

## Problem 3 (2 pts)

So, let's get more systematic about ANN design, training and testing. Problems 3 and 4 will give you a flavor of what a neural network designer/data scientist is doing with large datasets. To keep things simple (KIS), we'll confine ourselves to ANNs with 1 and 2 hidden layers.

Write the function `train_1_hidden_layer_anns(lwr=10, upr=50, eta=0.25, mbs=10, ne=10)` that takes the lower (`lwr`) and upper (`upr`) bounds of the number of nodes in the hidden layer (the only hidden layer), the learning rate (`eta`), the mini batch size (`mbs`), and the number of epochs (`ne`) to train and test, and calls the method `ann.mini_batch_sgd()` to train/test each network whose size of the hidden layer is in `[lwr, upr]` on the MNIST training and testing data. Use the increment of 10 to go to the next HLS in the range. Here's a test run with the default parameters and the output from the first two networks. In general, if the increment of 10 is used on the HLS range, this function will train and test all 784 x $i$ x 10 ANNs, where $i \in \{10, 20, 30, 40, 50\}$.

```
>>> from cs5600_6600_f20_hw03 import *
>>> train_1_hidden_layer_anns()
==== Training 784x10x10 ANN ======
Epoch 0: 6633 / 10000
Epoch 1: 7670 / 10000
Epoch 2: 8273 / 10000
Epoch 3: 8559 / 10000
Epoch 4: 8687 / 10000
Epoch 5: 8774 / 10000
Epoch 6: 8835 / 10000
Epoch 7: 8891 / 10000
Epoch 8: 8910 / 10000
Epoch 9: 8951 / 10000
==== Training 784x10x10 ANN DONE... ======
==== Training 784x20x10 ANN ======
Epoch 0: 6819 / 10000
Epoch 1: 8256 / 10000
Epoch 2: 8623 / 10000
Epoch 3: 8792 / 10000
Epoch 4: 8898 / 10000
Epoch 5: 8959 / 10000
Epoch 6: 9016 / 10000
Epoch 7: 9058 / 10000
Epoch 8: 9095 / 10000
Epoch 9: 9107 / 10000
==== Training 784x20x10 ANN DONE... ======
```

Save your implementation in `cs5600_6600_f20_hw3.py` and use it to fill in Table 1, where the cell ($\eta$, HLS) is the accuracy of the ANN with the HLS nodes in the hidden layer trained with the given value of $\eta$. Keep the mini batch size at 10 and the number of epochs at 50. Save this table with your stats in `ann_report.pdf` under the heading **Problem 3**.

Generalize `train_1_hidden_layer_anns()` to implement the function `train_2_hidden_layer_anns(lwr=10, upr=50, eta=0.25, mbs=10, ne=10)`. This function behaves like `train_1_hidden_layer_anns()` but trains and tests ANNs with 2 hidden layers whose sizes range from `lwr` to `upr` in increments of 10. In other words, it will train and test on MNIST all ANNs with the architecture 780 x $i$ x $j$ x 10, where $i, j \in \{10, 20, 30, 40, 50\}$. Here's a test run with the output where $i = 10$ and $j$ ranges from 10 to 50 in increments of 10.

Table 1: Training ANNs with 1 Hidden Layer on MNIST; mini batch = 10, num epochs=50

| Eta/HLS | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $\eta = 2$ | | | | | |
| $\eta = 1.5$ | | | | | |
| $\eta = 1.0$ | | | | | |
| $\eta = 0.5$ | | | | | |
| $\eta = 0.25$ | | | | | |

```
>>> train_2_hidden_layer_anns()
==== Training 784x10x10x10 ANN ======
Epoch 0: 3914 / 10000
Epoch 1: 6133 / 10000
Epoch 2: 7360 / 10000
Epoch 3: 8064 / 10000
Epoch 4: 8385 / 10000
Epoch 5: 8575 / 10000
Epoch 6: 8660 / 10000
Epoch 7: 8709 / 10000
Epoch 8: 8771 / 10000
Epoch 9: 8819 / 10000
==== Training 784x10x10x10 ANN DONE... ======
==== Training 784x10x20x10 ANN ======
Epoch 0: 6848 / 10000
Epoch 1: 7815 / 10000
Epoch 2: 8204 / 10000
Epoch 3: 8424 / 10000
Epoch 4: 8557 / 10000
Epoch 5: 8633 / 10000
Epoch 6: 8692 / 10000
Epoch 7: 8732 / 10000
Epoch 8: 8788 / 10000
Epoch 9: 8817 / 10000
==== Training 784x10x20x10 ANN DONE... ======
==== Training 784x10x30x10 ANN ======
Epoch 0: 6953 / 10000
Epoch 1: 7991 / 10000
Epoch 2: 8352 / 10000
Epoch 3: 8565 / 10000
Epoch 4: 8651 / 10000
Epoch 5: 8753 / 10000
Epoch 6: 8815 / 10000
Epoch 7: 8849 / 10000
Epoch 8: 8889 / 10000
Epoch 9: 8916 / 10000
==== Training 784x10x30x10 ANN DONE... ======
==== Training 784x10x40x10 ANN ======
Epoch 0: 6640 / 10000
Epoch 1: 7814 / 10000
Epoch 2: 8265 / 10000
Epoch 3: 8524 / 10000
Epoch 4: 8668 / 10000
Epoch 5: 8706 / 10000
Epoch 6: 8848 / 10000
Epoch 7: 8890 / 10000
Epoch 8: 8926 / 10000
Epoch 9: 8941 / 10000
==== Training 784x10x40x10 ANN DONE... ======
==== Training 784x10x50x10 ANN ======
Epoch 0: 7368 / 10000
Epoch 1: 8060 / 10000
Epoch 2: 8397 / 10000
Epoch 3: 8581 / 10000
Epoch 4: 8683 / 10000
Epoch 5: 8778 / 10000
Epoch 6: 8858 / 10000
Epoch 7: 8901 / 10000
Epoch 8: 8931 / 10000
Epoch 9: 8964 / 10000
==== Training 784x10x50x10 ANN DONE... ======
```

Use your implementation of `train_2_hidden_layer_anns()` to fill in Tables 2, 3, 4, 5, and 6. Save these tables with your stats in `ann_report.pdf` under the heading **Problem 3**. Briefly (no more than a paragraph) state your observations on the impact of the architecture and $\eta$ on recognition accuracy.

Table 2: Training ANNs with 2 Hidden Layers on MNIST; 1st HLS=10; mini batch = 10, num epochs=50

| Eta/2nd HLS | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $\eta = 2$ | | | | | |
| $\eta = 1.5$ | | | | | |
| $\eta = 1.0$ | | | | | |
| $\eta = 0.5$ | | | | | |
| $\eta = 0.25$ | | | | | |

Table 3: Training ANNs with 2 Hidden Layers on MNIST; 1st HLS=20; mini batch = 10, num epochs=50

| Eta/2nd HLS | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $\eta = 2$ | | | | | |
| $\eta = 1.5$ | | | | | |
| $\eta = 1.0$ | | | | | |
| $\eta = 0.5$ | | | | | |
| $\eta = 0.25$ | | | | | |

Table 4: Training ANNs with 2 Hidden Layers on MNIST; 1st HLS=30; mini batch = 10, num epochs=50

| Eta/2nd HLS | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $\eta = 2$ | | | | | |
| $\eta = 1.5$ | | | | | |
| $\eta = 1.0$ | | | | | |
| $\eta = 0.5$ | | | | | |
| $\eta = 0.25$ | | | | | |

Table 5: Training ANNs with 2 Hidden Layers on MNIST; 1st HLS=40; mini batch = 10, num epochs=50

| Eta/2nd HLS | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $\eta = 2$ | | | | | |
| $\eta = 1.5$ | | | | | |
| $\eta = 1.0$ | | | | | |
| $\eta = 0.5$ | | | | | |
| $\eta = 0.25$ | | | | | |

Table 6: Training ANNs with 2 Hidden Layers on MNIST; 1st HLS=50; mini batch = 10, num epochs=50

| Eta/2nd HLS | 10 | 20 | 30 | 40 | 50 |
|---|---|---|---|---|---|
| $\eta = 2$ | | | | | |
| $\eta = 1.5$ | | | | | |
| $\eta = 1.0$ | | | | | |
| $\eta = 0.5$ | | | | | |
| $\eta = 0.25$ | | | | | |

## Problem 4 (2 pts)

Define five ANNs that you will train on the MNIST data in `cs5600_6600_f20_hw03.py`. This file has some starter code. 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.

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 $\eta$ and mini-batch values that we'll use in training and testing.

```
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 $\eta$ 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 `andd.mini_batch_sgd()` 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 $\eta$ 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 $\eta$ value, and the minibatch size with which it was trained. The $\eta$ 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 $\eta$ 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 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 should 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)
```

Write 2-3 paragraphs to compare the performance of your ensemble of 5 ANNs on the validation data with each individual ANN on the same validation data. It would be best if you description included a table or a graph that compares the classification performance of your individual networks with the classification performance of your ensemble on the validation dataset of MNIST. Make sure that you 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 it Save your description with your stats in `ann_report.pdf` under the heading **Problem 4**.

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).

## What to Submit

1. rumelhart_et_al.pdf with your summary of the paper "Learning representations by back-propagating errors";
2. ann_report.pdf with your answers to Problems 3 and 4;
3. cs5600_6600_f20_hw03.py with your code;
4. zipped directory pck_nets with your five trained persisted networks.

Happy Hacking, Reading, and Writing!