Sai Aman Gottumukkala
CSCE 420 – Spring 2019
426008787

Character Recognition ANN Report

Statement of the problem

The problem being tackled in this project is creating a system that can identify capital letters in a 9x14 BDF format to a high degree of accuracy. Motivation for completing this project aside from it being an assignment, is that neural nets have the capability of modeling functions that can solve problems which "weak" linear methods cannot. Applying this to identify BDF fonts is a great first step to learning about how neural nets are stronger tools than standard linear functions.

Another motivation of creating a neural net is that they can identify less obvious patterns that people might not see in a problem. If one were to create a linear classifier for this assignment, they would model it based on visual patterns they physically see, such as the size or dimensions of each letter. A neural net however, can identify deep patterns resulting in more accurate results.

Restrictions and limitations

The current project specification requires us to train a network to identify all 26 letters with high degree of accuracy, but a huge restriction is that we only have one training case for each letter. The following restrictions and limitations will assume that an ideal neural network would be able to identify more than the training input.

The model is largely overfit to the specific format given. An example is that translating letters to any position other than the given input would yield our model to return less accurate results, because it's only trained with one specific position. The given input is restricted to a very specific format which is 9x14 BDF font format. Applying the trained neural net to other fonts would give largely inaccurate results assuming it's possible to normalize the input data to a 9x14 or 126 binary vector format. Even if the font was BDF but a different dimension size, the input would have to scale to different dimensions to meet our input specifications, which could lead to inaccurate results since we would either have to assume or remove information to make a binary input vector from the original input.

Explanation of Approach

Network Structure

I only experimented with layers of 127 x 126 x 26, because of the initial strong positive results using this structure while also regarding the project guidelines which required this as a minimum. It's a strongly connected layered network where all neighboring layers are fully connected. The first 126 bits of the input layer are the input vector for either training or evaluation. The 127th bit is a biased node whose activation value is always 1. There is then a hidden layer of 126 neurons and an output layer of size 26 that is used to determine the final evaluation / error of the neural net. This can be used as an evaluation given the input or used to back propagate and adjust weights due to the errors associated with the forward propagation.

Activation Functions

As the project guidelines specified, the logistic sigmoid function and it's derivative was used for the output layer and the hyperbolic logistic sigmoid function and it's derivative was used for every other layer.

Learning Rate/Schedule

Different constant and dynamic learning rates were experimented with during training.
The following are the tested learning rates:

.01 * epoch #
.001 * epoch #
.1
.1 / epoch #

The learning rates were evaluated based on the average number of flipped bits allowed before resulting in a wrong evaluation along with training time. The raw number of flipped bits were written to a file to determine averages for different learning rates. This same information along with the wrong evaluation with the expected value was also written to another file. The latter information will be used later on in the report along with the results of the different learning rates.

Back-propagation iterations

These 4 different stopping criterion for training were experimented with:

1. Stopped all training and back-propagation cycles once the average correct evaluations was over 98%
2. Stopped all training and back-propagation cycles once the average correct evaluations was over 98% and the value in the max output node in the output layer was over .95.
3. Stopped all training and back-propagation cycles once the average correct evaluations was over 98%
4. Stopped all training and back-propagation cycles once the average correct evaluations was over 98% and the value in the max output node in the output layer was over .95.

They both resulted in different # of total epochs for training and also different thresholds of allowed noise which will be explained later in the report. Both of these stopping criterion were tested with learning rates of .1.

Sample Run

The following specifications are used for the sample run:

Learning rate: .1
Stopping Criteria: Avg. correct evaluation >= 98% and max output node has to be >= .95

The following are all of the different functions shown:

Backpropagation
Printing weights of a neural network to file
Read weights into a neural network.
Evaluation function
Noise evaluation

Backpropagation – writes to a file called “trainingdata.txt”

Image for replication:

```
int main(){
    srand(NULL);
    vector< vector<double> > charToInputMap(26, vector<double>(126, 0.0)); // Map capital letters to 126 size one-dimensional input buffer
    vector< vector<int> > dottedIndexes; // Maps all the dotted indexes for a character


    bool parsed = parseBDF(charToInputMap, dottedIndexes); // Parse the BDF File

    if(!parsed){
        cout << "Error parsing" << endl;
        return 0;
    }

    network neuralnet; //Untrained neural net with random weights between -.1 and .1

    // Train an untrained neural net given BDF Format data
    network adjustedNet = backPropogation(charToInputMap, neuralnet);

    //Uses inputted weights from a text file
    // readWeights(neuralnet, "weightVals.txt");

    // Print weights of a neural net to a text file after training
    // printWeights(adjustedNet.weights, "stopweights.txt");

    // Evaluate how many times noise is introduced to get a wrong value
    // evaluateNoise(neuralnet, dottedIndexes, charToInputMap, "stopnoise");
    return 0;
}
```

The following screenshots are information for every 50 epoch until training is over. Total 443 epochs.
In order for a letter to be counted as a success, it needs to evaluate correctly and also have an output node value over .95.

The format is (Expected : Predicted : Max Output Node Value)

neuralnet.cpp | trainingfile.txt

```
A : L : 0.580292
B : A : 0.549555
C : B : 0.535521
D : C : 0.457318
E : C : 0.447879
F : E : 0.425533
G : C : 0.343269
H : E : 0.300994
I : G : 0.316711
J : F : 0.320661
K : I : 0.23062
L : K : 0.237118
M : L : 0.24043
N : L : 0.175898
O : K : 0.164343
P : O : 0.147001
Q : O : 0.142454
R : L : 0.116536
S : L : 0.135984
T : M : 0.245647
U : T : 0.123433
V : T : 0.186699
W : T : 0.119859
X : S : 0.137091
Y : V : 0.171589
Z : X : 0.132518
Current Epoch: 1
Average Correct: 0
<----------------------------->
```

neuralnet.cpp | trainingfile.txt

```
A : A : 0.843754
B : P : 0.147844
C : C : 0.705373
D : D : 0.351162
E : F : 0.166313
F : F : 0.42592
G : G : 0.575116
H : H : 0.591377
I : I : 0.65585
J : J : 0.861472
K : K : 0.627425
L : L : 0.616069
M : M : 0.836238
N : N : 0.528153
O : O : 0.622339
P : P : 0.295373
Q : Q : 0.786826
R : K : 0.18573
S : S : 0.698597
T : T : 0.835425
U : U : 0.36083
V : V : 0.81998
W : W : 0.703076
X : X : 0.825313
Y : Y : 0.849921
Z : Z : 0.84177
Current Epoch: 50
Average Correct: 0
<-------------------------------->
```

neuralnet.cpp | trainingfile.txt

```
A : A : 0.912296
B : B : 0.635989
C : C : 0.884384
D : D : 0.829707
E : E : 0.785524
F : F : 0.833061
G : G : 0.872086
H : H : 0.847496
I : I : 0.852105
J : J : 0.916982
K : K : 0.873695
L : L : 0.870788
M : M : 0.911059
N : N : 0.863561
O : O : 0.85594
P : P : 0.830299
Q : Q : 0.90866
R : R : 0.779513
S : S : 0.895751
T : T : 0.912558
U : U : 0.842067
V : V : 0.908649
W : W : 0.904658
X : X : 0.908192
Y : Y : 0.918693
Z : Z : 0.922831
Current Epoch: 100
Average Correct: 0
<-------------------------------->
```

neuralnet.cpp | trainingfile.txt

```
A : A : 0.937969
B : B : 0.851279
C : C : 0.921746
D : D : 0.894895
E : E : 0.881233
F : F : 0.896359
G : G : 0.915082
H : H : 0.903929
I : I : 0.903063
J : J : 0.939451
K : K : 0.915617
L : L : 0.911919
M : M : 0.93706
N : N : 0.910741
O : O : 0.904687
P : P : 0.895125
Q : Q : 0.936306
R : R : 0.879089
S : S : 0.927441
T : T : 0.938117
U : U : 0.903537
V : V : 0.935598
W : W : 0.935103
X : X : 0.936374
Y : Y : 0.940208
Z : Z : 0.942973
Current Epoch: 150
Average Correct: 0
<-------------------------------->
```

neuralnet.cpp | trainingfile.txt

```
A : A : 0.94995
B : B : 0.897482
C : C : 0.938504
D : D : 0.92002
E : E : 0.912227
F : F : 0.920814
G : G : 0.933634
H : H : 0.926804
I : I : 0.924154
J : J : 0.950724
K : K : 0.934343
L : L : 0.930081
M : M : 0.949467
N : N : 0.930729
O : O : 0.925724
P : P : 0.9201
Q : Q : 0.948981
R : R : 0.911935
S : S : 0.942379
T : T : 0.950423
U : U : 0.925884
V : V : 0.948028
W : W : 0.948303
X : X : 0.94923
Y : Y : 0.951011
Z : Z : 0.953335
Current Epoch: 200
Average Correct: 0.153846
<-------------------------------->
```

neuralnet.cpp | trainingfile.txt

```
A : A : 0.957061
B : B : 0.918714
C : C : 0.948141
D : D : 0.933647
E : E : 0.928243
F : F : 0.934209
G : G : 0.944258
H : H : 0.939231
I : I : 0.936165
J : J : 0.957556
K : K : 0.945037
L : L : 0.940695
M : M : 0.956858
N : N : 0.942042
O : O : 0.937735
P : P : 0.933723
Q : Q : 0.956476
R : R : 0.928537
S : S : 0.951203
T : T : 0.957731
U : U : 0.938208
V : V : 0.955486
W : W : 0.955971
X : X : 0.956795
Y : Y : 0.957722
Z : Z : 0.959742
Current Epoch: 250
Average Correct: 0.423077
<-------------------------------->
```

neuralnet.cpp trainingfile.txt ×

```
A : A : 0.961883
B : B : 0.931284
C : C : 0.95452
D : D : 0.94242
E : E : 0.938275
F : F : 0.942852
G : G : 0.95127
H : H : 0.94722
I : I : 0.944101
J : J : 0.962233
K : K : 0.95207
L : L : 0.947807
M : M : 0.961863
N : N : 0.949456
O : O : 0.945646
P : P : 0.942497
Q : Q : 0.961534
R : R : 0.938791
S : S : 0.957118
T : T : 0.962664
U : U : 0.946213
V : V : 0.960574
W : W : 0.961105
X : X : 0.961881
Y : Y : 0.962376
Z : Z : 0.964158
Current Epoch: 300
Average Correct: 0.538462
<--------------------------------->
```

neuralnet.cpp trainingfile.txt ×

```
A : A : 0.965423
B : B : 0.939742
C : C : 0.959116
D : D : 0.948634
E : E : 0.945253
F : F : 0.948979
G : G : 0.956309
H : H : 0.952873
I : I : 0.949815
J : J : 0.965685
K : K : 0.957109
L : L : 0.952975
M : M : 0.965525
N : N : 0.954757
O : O : 0.951321
P : P : 0.948706
Q : Q : 0.965229
R : R : 0.945861
S : S : 0.96141
T : T : 0.966265
U : U : 0.951913
V : V : 0.964319
W : W : 0.96484
X : X : 0.965586
Y : Y : 0.965833
Z : Z : 0.967426
Current Epoch: 350
Average Correct: 0.730769
<--------------------------------->
```

neuralnet.cpp trainingfile.txt ×

```
A : A : 0.96816
B : B : 0.945891
C : C : 0.962617
D : D : 0.953313
E : E : 0.950442
F : F : 0.953594
G : G : 0.960139
H : H : 0.957129
I : I : 0.954167
J : J : 0.968363
K : K : 0.96093
L : L : 0.956937
M : M : 0.968346
N : N : 0.958772
O : O : 0.955629
P : P : 0.953378
Q : Q : 0.968073
R : R : 0.951086
S : S : 0.964694
T : T : 0.969036
U : U : 0.95622
V : V : 0.967216
W : W : 0.96771
X : X : 0.968431
Y : Y : 0.968526
Z : Z : 0.969966
Current Epoch: 400
Average Correct: 0.961538
<--------------------------------->
```

neuralnet.cpp trainingfile.txt ×

```
Average Correct: 0.961538
<--------------------------------->

A : A : 0.970074
B : B : 0.950009
C : C : 0.965039
D : D : 0.956522
E : E : 0.953971
F : F : 0.95676
G : G : 0.962783
H : H : 0.960048
I : I : 0.957177
J : J : 0.970241
K : K : 0.963563
L : L : 0.959689
M : M : 0.970313
N : N : 0.961537
O : O : 0.958601
P : P : 0.956579
Q : Q : 0.970055
R : R : 0.954623
S : S : 0.966972
T : T : 0.970966
U : U : 0.959182
V : V : 0.969243
W : W : 0.969708
X : X : 0.970412
Y : Y : 0.970418
Z : Z : 0.971748
Current Epoch: 443
Average Correct: 1
<--------------------------------->
```

Printing Weights of Neural Net to File

Format
Weight Layers
Layer 1 Size | Layer 2 Size
Layer 1 Node | Layer 2 Node | Weight value
...... (All connections between Layer 1 & 2)
Layer 2 Size | Layer 3 Size
Layer 2 Node | Layer 3 Node | Weight value
..... (All connections between Layer 2 & 3)

Image for Replication (Printing to adjustedweights.txt)

```cpp
int main(){
    srand(NULL);
    vector< vector<double> > charToInputMap(26, vector<double>(126, 0.0)); // Map capital letters to 126 size one-dimensional input buffer
    vector< vector<int> > dottedIndexes; // Maps all the dotted indexes for a character


    bool parsed = parseBDF(charToInputMap, dottedIndexes); // Parse the BDF File

    if(!parsed){
        cout << "Error parsing" << endl;
        return 0;
    }

    network neuralnet; //Untrained neural net with random weights between -.1 and .1

    // Train an untrained neural net given BDF Format data
    network adjustedNet = backPropogation(charToInputMap, neuralnet);

    //Uses inputted weights from a text file
    // readWeights(neuralnet, "weightVals.txt");

    // Print weights of a neural net to a text file after training
    printWeights(adjustedNet.weights, "adjustedweights.txt");

    // Evaluate how many times noise is introduced to get a wrong value
    // evaluateNoise(neuralnet, dottedIndexes, charToInputMap, "stopnoise");
    return 0;
}
```

neuralnet.cpp | adjustedweights.txt

```
2
127 127
0 0 -0.0855197
0 1 -0.161866
0 2 -0.0127828
0 3 0.0161745
0 4 -0.0810357
0 5 0.13988
0 6 -0.00883121
0 7 0.00384177
0 8 -0.2259
0 9 -0.15689
0 10 0.152034
0 11 0.0867592
0 12 0.070513
0 13 0.026048
0 14 0.0923206
0 15 -0.105323
0 16 0.0440378
0 17 -0.101091
0 18 0.152778
0 19 0.0367377
0 20 0.151499
0 21 0.0541439
0 22 -0.0646871
```

Reading Weights Into a Neural Net From File

Usage for reading weights:

Reading weights from a file will result in the same as training a neural net, printing to a file, and reading in that file to a new neural net. Therefore, think of everything after this step as possible from either reading in weights or training a neural net.

Image for Replication (Reading from “adjustedweights.txt” from previous example)

neuralnet.cpp | adjustedweights.txt | trainingfile.txt

```cpp

int main(){
    srand(NULL);
    vector< vector<double> > charToInputMap(26, vector<double>(126, 0.0)); // Map capita
    vector< vector<int> > dottedIndexes; // Maps all the dotted indexes for a character


    bool parsed = parseBDF(charToInputMap, dottedIndexes); // Parse the BDF File

    if(!parsed){
        cout << "Error parsing" << endl;
        return 0;
    }

    network neuralnet; //Untrained neural net with random weights between -.1 and .1

    // Train an untrained neural net given BDF Format data
    // network adjustedNet = backPropogation(charToInputMap, neuralnet);

    //Uses inputted weights from a text file
    readWeights(neuralnet, "adjustedweights.txt");

    // Print weights of a neural net to a text file after training
    // printWeights(adjustedNet.weights, "adjustedweights.txt");

    // Evaluate how many times noise is introduced to get a wrong value
    // evaluateNoise(neuralnet, dottedIndexes, charToInputMap, "stopnoise");
    return 0;
}
```

Evaluation Function

Given a binary input vector of size 126 and a neural network, it will return the predicted value from the neural net. This function is not used independently, but is used in conjunction with the noise evaluation function.

Code Snippet:

neuralnet.cpp × | adjustedweights.txt | trainingfile.txt

```cpp
    return nn;
}

char evaluate(vector<double> binaryInput, network &nn){
    nn.resetNodeVals();
    int numLayers = nn.networkNodes.size();

    /* Prep Input Layer */
    for(int i = 0; i<binaryInput.size(); i++){
        nn.networkNodes[0].at(i) = binaryInput[i];
    }

    for(int layer = 1; layer < numLayers; layer++){
        for(int currLayerNode = 0; currLayerNode < nn.networkNodes[layer].size(); currLayerNode++){
            double inVal = 0;
            for(int prevLayerNode = 0; prevLayerNode < nn.networkNodes[layer-1].size(); prevLayerNode++){
                inVal += nn.weights[layer-1][prevLayerNode][currLayerNode] * nn.networkNodes[layer-1][prevLayerNode];
            }

            double activationVal;
            if(layer == numLayers-1){
                activationVal = logisticSigmoid(inVal);
            }else{
                activationVal = hyperbolicTangentSigmoid(inVal);
            }

            nn.networkNodes[layer][currLayerNode] = activationVal;
        }
    }

    double maxOutputVal = 0;
    char actual = '0';

    for(int outputNode = 0; outputNode<nn.networkNodes[numLayers-1].size(); outputNode++){
        if(nn.networkNodes[numLayers-1][outputNode] > maxOutputVal){
            maxOutputVal = nn.networkNodes[numLayers-1][outputNode];
            actual = 'A' + outputNode;
        }
    }

    //cout << "Evaluated to : " << actual << " : " << maxOutputVal << endl;
    return actual;
}
```

Noise Evaluation

Given a neural net, it will determine the # of flipped bits (black -> white) for each letter until the predicted value is incorrect. This is printed to a file of your choice in the following format:
Expected | # failed bits | Failed evaluation

Image for Replication (Printing to noisedata.txt)
I ran this using a neural net which read weights from a file. This is not necessary and can be trained first and then used to evaluate noise.

neuralnet.cpp × | adjustedweights.txt | trainingfile.txt

```cpp
        evaluate() - evaluates a given binary input vector with a given neural network
        checkNoise() - checks how many "noise iterations" or flippedBits it takes until expected value != returned value
*/

int main(){
    srand(NULL);
    vector< vector<double> > charToInputMap(26, vector<double>(126, 0.0)); // Map capital letters to 126 size one-dimensional input buffer
    vector< vector<int> > dottedIndexes; // Maps all the dotted indexes for a character

    bool parsed = parseBDF(charToInputMap, dottedIndexes); // Parse the BDF File

    if(!parsed){
        cout << "Error parsing" << endl;
        return 0;
    }

    network neuralnet; //Untrained neural net with random weights between -.1 and .1

    // Train an untrained neural net given BDF Format data
    // network adjustedNet = backPropogation(charToInputMap, neuralnet);

    //Uses inputted weights from a text file
    readWeights(neuralnet, "adjustedweights.txt");

    // Print weights of a neural net to a text file after training
    // printWeights(adjustedNet.weights, "adjustedweights.txt");

    // Evaluate how many times noise is introduced to get a wrong value
    evaluateNoise(neuralnet, dottedIndexes, charToInputMap, "noisedata.txt");
    return 0;
```

Sample of noisedata.txt

neuralnet.cpp | noisedata.txt | adjustedweights.txt | trainingfile.txt

```
A failed with 33 flipped bits. Evaluated to: I
B failed with 34 flipped bits. Evaluated to: J
C failed with 26 flipped bits. Evaluated to: L
D failed with 27 flipped bits. Evaluated to: U
E failed with 24 flipped bits. Evaluated to: L
F failed with 24 flipped bits. Evaluated to: P
G failed with 23 flipped bits. Evaluated to: L
H failed with 30 flipped bits. Evaluated to: P
I was evaluated correctly for all 22/22 flipped bits
J failed with 23 flipped bits. Evaluated to: K
K failed with 30 flipped bits. Evaluated to: E
L failed with 13 flipped bits. Evaluated to: I
M failed with 42 flipped bits. Evaluated to: Z
N failed with 36 flipped bits. Evaluated to: P
O failed with 20 flipped bits. Evaluated to: D
P failed with 27 flipped bits. Evaluated to: I
Q failed with 32 flipped bits. Evaluated to: O
R failed with 16 flipped bits. Evaluated to: H
S failed with 24 flipped bits. Evaluated to: I
T failed with 27 flipped bits. Evaluated to: F
U failed with 34 flipped bits. Evaluated to: A
V failed with 31 flipped bits. Evaluated to: F
W failed with 42 flipped bits. Evaluated to: K
X failed with 32 flipped bits. Evaluated to: I
Y failed with 28 flipped bits. Evaluated to: T
Z failed with 28 flipped bits. Evaluated to: C
```

Results and Analysis

Learning Rate results

Four different learning rates were tested and the following are the results:

.01 * passIteration - Average 26.8 flipped bits (noise)
.001 * passIteration - Average of 26.38 bits (noise)
.1 - Average of 28 flipped bits (noise)
.1 / passIteration – Unknown (Training made no progress after significant amount of time)

Back-propagation iterations

Four different stopping criterion were tested and the following are the results:

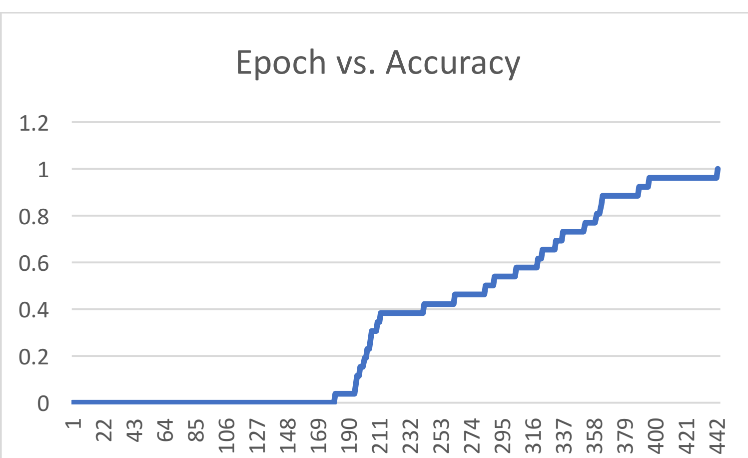
Stopped all training and back-propagation cycles after the average of correct evaluations was over 90%.
Number of epochs: 53
Average flipped bits (noise): 24.8

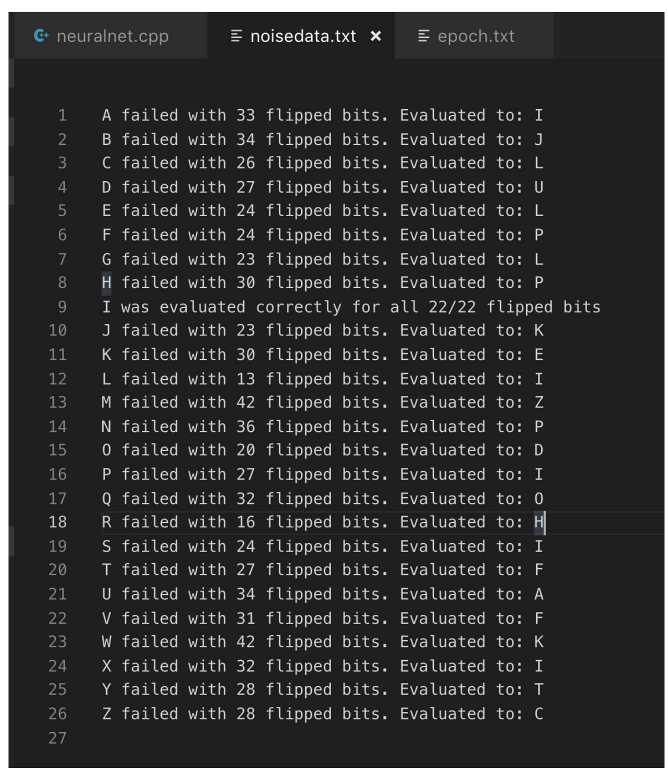
Stopped all training and back-propagation cycles once the average of correct evaluations was over 90% and the value in the max output node in the output layer was over .95.
Number of epochs: 389
Average flipped bits (noise): 27.4

Stopped all training and back-propagation cycles once the average of correct evaluations was over 98%
Number of epochs: 71
Average flipped bits (noise): 26.3

Stopped all training and back-propagation cycles once the average of correct evaluations was over 98% and the value in the max output node in the output layer was over .95.
Number of epochs: 443
Average flipped bits (noise): 28

The results from the last back-propagation test yielded the best results. This is solely based off of the allowed noise. Although the number of epochs is the greatest, the difference in training time is minimal. The following is a graph and noise chart from that training.




neuralnet.cpp   noisedata.txt   epoch.txt

```
A failed with 33 flipped bits. Evaluated to: I
B failed with 34 flipped bits. Evaluated to: J
C failed with 26 flipped bits. Evaluated to: L
D failed with 27 flipped bits. Evaluated to: U
E failed with 24 flipped bits. Evaluated to: L
F failed with 24 flipped bits. Evaluated to: P
G failed with 23 flipped bits. Evaluated to: L
H failed with 30 flipped bits. Evaluated to: P
I was evaluated correctly for all 22/22 flipped bits
J failed with 23 flipped bits. Evaluated to: K
K failed with 30 flipped bits. Evaluated to: E
L failed with 13 flipped bits. Evaluated to: I
M failed with 42 flipped bits. Evaluated to: Z
N failed with 36 flipped bits. Evaluated to: P
O failed with 20 flipped bits. Evaluated to: D
P failed with 27 flipped bits. Evaluated to: I
Q failed with 32 flipped bits. Evaluated to: O
R failed with 16 flipped bits. Evaluated to: H
S failed with 24 flipped bits. Evaluated to: I
T failed with 27 flipped bits. Evaluated to: F
U failed with 34 flipped bits. Evaluated to: A
V failed with 31 flipped bits. Evaluated to: F
W failed with 42 flipped bits. Evaluated to: K
X failed with 32 flipped bits. Evaluated to: I
Y failed with 28 flipped bits. Evaluated to: T
Z failed with 28 flipped bits. Evaluated to: C
```

The image on the right shows the noise allowed for each letter until the neural net predicted a wrong classification.

Conclusions

The results show that having a constant learning rate yields the best results compared to smaller or increasing/decreasing learning rates. While using this constant learning rate, it's possible for the neural net to yield better accuracy with noise if the stopping criterion expectations are increased. This is important, because even though the stopping criterion is increased, the increased time needed to train is minimal and has little to no effect towards the usage of the neural net, since the main purpose would be to run evaluations not train. The activation functions specified in the project guidelines worked well and no issues were encountered while training and evaluating.

Future research

If the structure of the neural network were to stay the same, a massive improvement would be to use more testing data for each letter. Assuming the testing data would have a variety of changes to the initial letter test data such as translations, flips, rotations, and scaling, this would allow the neural net to identify letters that present lots of noise compared to the single letter input vector that the neural net was originally tested on. Aside from using the same BDF font, multiple fonts could be used to make a more general letter classifier. Another obvious addition is to use more than just capital letters for training.

Since the premise of this assignment is to predict a visual font accurately, another improvement would be to restructure the artificial neural network to a convolutional neural network. CNN's use feature detectors and feature maps which use locality to find patterns in training data. This allows a convolutional neural network to identify unique visual features which an artificial neural network may or may not come across.

Instructions on how to run your program

Precursor:
The BDF file used is manipulated for easy parsing during every program. Therefore, it is required to use the im9x14u.bdf file in the given directory. The default downloaded im9x14u.bdf file will not work.

In order to test different features, it's required to change which methods are called inside the main function.
The main function specifies which commands are required and which ones you can test yourself. The following are the functions you can manipulate:

- backPropogation() – writes test data to "trainingfile.txt"
- readWeights() – reads weights from a file of your choice and inserted into a network
- printWeights() – prints weights from any network to a file of your choice
- evaluateNoise() – prints noise evaluation to a file of your choice

All functions are currently in the main file with an example of how to run the function.

Compilation: g++ neuralnet.cpp
Run: ./a.out

```
/* COMMENTED CODE */

#include <iostream>
#include <unordered_map>
#include <fstream>
#include <string>
#include <vector>
#include <math.h>
#include <stdlib.h>
#include <time.h>
#include <stdio.h>
#include <random>
using namespace std;


int colSize = 9;
int rowSize = 14;

string getBinary(char c){
    switch(c){
        case '0':
            return "0000";
        case '1':
            return "0001";
        case '2':
            return "0010";
        case '3':
            return "0011";
        case '4':
            return "0100";
        case '5':
            return "0101";
        case '6':
            return "0110";
        case '7':
            return "0111";
        case '8':
            return "1000";
        case '9':
            return "1001";
```

```
        case 'A':
            return "1010";
        case 'B':
            return "1011";
        case 'C':
            return "1100";
        case 'D':
            return "1101";
        case 'E':
            return "1110";
        case 'F':
            return "1111";
        default:
            return "";
    }
}

string hexToBinary(string hex){
    string binaryStr = "";

    char first = hex[0];
    char second = hex[1];

    return getBinary(first) + getBinary(second);
}

double logisticSigmoid(double input){
    double eVal = exp(-1 * input);
    return 1.0 / (1.0 + eVal);
}

double hyperbolicTangentSigmoid(double input){
    return tanh(input);
}

double randomVal(double minVal, double maxVal){
    double f = (double)rand() / RAND_MAX;
    double ans = minVal + f * (maxVal - minVal);

    return ans;
}
```

```
class network{
    public:
        int inputLayerSize = 127;
        int layerTwoSize = 127;
        int layerThreeSize = 26;

        vector< vector<double> > networkNodes;
        vector< vector< vector<double> > > weights;

        vector<double> inputLayerVals;
        vector<double> layerTwoVals;
        vector<double> layerThreeVals;

        void resetNodeVals(){
            networkNodes = vector< vector<double> >(3); // 3 Layers

            vector<double> layerOne(127, 0);
            layerOne[126] = 1;
            networkNodes[0] = layerOne;

            vector<double> layerTwo(127, 0);
            layerTwo[126] = 1;
            networkNodes[1] = layerTwo;

            vector<double> outputLayer(26, 0);
            networkNodes[2] = outputLayer;
        }

        static vector< vector<double> > errorVector(){
            vector< vector<double> > deltas;

            vector<double> layerOne(127, 0);
            deltas.push_back(layerOne);

            vector<double> layerTwo(127, 0);
            deltas.push_back(layerTwo);

            vector<double> outputLayer(26, 0);
            deltas.push_back(outputLayer);
```

```
        deltas.push_back(outputLayer);

        return deltas;
    }

    network(){
        // Initialize networkNodes
        resetNodeVals();

        double minInitWeight = -.1;
        double maxInitWeight = .1;

        vector< vector<double> > inputLayerWeights; // 127 x 127
        vector< vector<double> > layerTwoWeights; // 127 x 26

        // Initialize inputLayerWeights
        for(int i = 0; i<inputLayerSize; i++){
            vector<double> temp;
            for(int j = 0; j<layerTwoSize; j++){

                // Get random value between -.1 and .1
                double rVal = randomVal(minInitWeight, maxInitWeight);
                while(rVal == 0){
                    rVal = randomVal(minInitWeight, maxInitWeight);
                }
                temp.push_back(rVal);
            }
            inputLayerWeights.push_back(temp);
        }

        // Initialize layerTwoWeights
        for(int i = 0; i<layerTwoSize; i++){
            vector<double> temp;
            for(int j = 0; j<layerThreeSize; j++){

                // Get random value between -.1 and .1
                double rVal =
```

```
                    double rVal = randomVal(minInitWeight, maxInitWeight);
                    while(rVal == 0){
                        rVal = randomVal(minInitWeight, maxInitWeight);
                    }
                    temp.push_back(rVal);
                }
                layerTwoWeights.push_back(temp);
            }

            weights.push_back(inputLayerWeights);
            weights.push_back(layerTwoWeights);
        }
};

void printWeights(vector< vector< vector<double> > > &weights, string fileName){
    ofstream weightFile;
    weightFile.open(fileName);

    weightFile << weights.size() << endl;

    for(int layer = 0; layer < weights.size(); layer++){
        weightFile << weights[layer].size() << " " <<weights[layer][0].size() << endl;
        for(int node = 0; node < weights[layer].size(); node++){
            for(int nextNode = 0; nextNode < weights[layer][node].size(); nextNode++){
                weightFile << node << " " << nextNode << " " << weights[layer][node][nextNode] << endl;
            }
        }
    }
}

void readWeights(network &nn, string readFile){
    ifstream weightFile;
    weightFile.open(readFile);
```

```
    weightFile.open(readFile);

    int numLayers;
    weightFile >> numLayers;

    vector< vector< vector<double> > > weights;

    for(int i = 0; i<numLayers; i++){
        int layerOneSize, layerTwoSize;
        weightFile >> layerOneSize >> layerTwoSize;
        vector< vector< double> >
        layerWeights(layerOneSize,
        vector<double>(layerTwoSize));

        for(int iter = 0; iter < layerOneSize *
        layerTwoSize; iter++){
            int prev, next;
            double weight;
            weightFile >> prev >> next >> weight;
            layerWeights.at(prev).at(next) = weight;
        }

        weights.push_back(layerWeights);
    }

    nn.weights = weights;
}

network backPropogation(vector< vector<double> >
charToInputMap, network &nn){
    ofstream tFile;
    tFile.open("trainingfile.txt");

    vector< vector<double> > errorVector =
    nn.errorVector();
    int numLayers = nn.networkNodes.size();

    double a = 0.1;
    double epoch = 0;

    nn.resetNodeVals();
```

```
        while(true){ //Figure out stopping criterion
            epoch++;
            double learningRate = a;
            int numCorrect = 0;

            for(int c = 0; c<charToInputMap.size(); c++){
                vector<double> binaryVector = charToInputMap[c]; //input

                char character = 'A' + c;
                vector<double> outputVector(26, 0); //output
                outputVector[c] = 1;

                /*               FORWARD PROPOGATION                */
                for(int i = 0; i<binaryVector.size(); i++){ // Initialize input layer with binary vector
                    nn.networkNodes[0][i] = binaryVector[i];
                }

                // layer == numLayers - 1 ? logistic sigmoid function : hyperbolic tangent function
                for(int layer = 1; layer < numLayers; layer++){
                    for(int currLayerNode = 0; currLayerNode < nn.networkNodes[layer].size(); currLayerNode++){
                        double inVal = 0;
                        for(int prevLayerNode = 0; prevLayerNode < nn.networkNodes[layer-1].size(); prevLayerNode++){
                            inVal += nn.weights[layer-1][prevLayerNode][currLayerNode] * nn.networkNodes[layer-1][prevLayerNode];
                        }

                        double activationVal, logisticVal, hyperbolicVal;
                        logisticVal = logisticSigmoid(inVal);
                        hyperbolicVal =
```

```
                    hyperbolicTangentSigmoid(inVal);

                    if(layer == numLayers-1){
                        activationVal = logisticVal;
                    }else{
                        activationVal = hyperbolicVal;
                    }

                    nn.networkNodes[layer][currLayerNode] = activationVal;
                }
            }

            /*                BACKWARD PROPOGATION                */
            for(int outputNode = 0; outputNode < nn.networkNodes[numLayers-1].size(); outputNode++){
                // logistic sigmoid function derivative is g(x) * (1 - g(x))
                double derivVal = nn.networkNodes[numLayers-1][outputNode] * (1 - nn.networkNodes[numLayers-1][outputNode]);
                double diffVal = outputVector[outputNode] - nn.networkNodes[numLayers-1][outputNode];
                errorVector[numLayers-1][outputNode] = derivVal * diffVal;
            }

            for(int layer = numLayers-2; layer >= 1; layer--){
                for(int node = 0; node < nn.networkNodes[layer].size(); node++){
                    // hyperbolic tangent sigmoid function derivative is 1 - ( g(x) ^ 2)
                    double derivVal = 1 - (nn.networkNodes[layer][node] * nn.networkNodes[layer][node]);

```

```
                double summationVal = 0;
                for(int nextLayerNode = 0; nextLayerNode < nn.weights[layer][node].size(); nextLayerNode++){
                    summationVal += nn.weights[layer][node][nextLayerNode] * errorVector[layer+1][nextLayerNode];
                }

                errorVector[layer][node] = summationVal * derivVal;
            }
        }

        double maxOutput = 0;
        char testOut = '0';

        for(int outputNode = 0; outputNode < nn.networkNodes[numLayers-1].size(); outputNode++){
            if(nn.networkNodes[numLayers-1][outputNode] >= maxOutput){
                maxOutput = nn.networkNodes[numLayers-1][outputNode];
                testOut = 'A' + outputNode;
            }
        }

        for(int layer = 0; layer < numLayers-1; layer++){
            for(int currLayerNode = 0; currLayerNode < nn.weights[layer].size(); currLayerNode++){
                for(int nextLayerNode = 0; nextLayerNode < nn.weights[layer][currLayerNode].size(); nextLayerNode++){
                    double currWeight =
```

```cpp
                    nn.weights[layer][currLayerNode][nextLayerNode];
                    double newWeight = currWeight + (learningRate * nn.networkNodes[layer][currLayerNode] * errorVector[layer+1][nextLayerNode]);

                    nn.weights[layer][currLayerNode][nextLayerNode] = newWeight;
                }
            }
        }

        if(testOut == character && maxOutput >= .95){
            numCorrect++;
        }

        tFile << character << " : " << testOut << " : " << maxOutput << endl;
    }

    double avgCorrect= numCorrect / 26.0;
    tFile << "Current Epoch: " << epoch << endl;
    tFile << "Average Correct: " << avgCorrect << endl;
    tFile << "<--------------------------------->" << endl << endl;

    cout << "Current Epoch: " << epoch << endl;
    cout << "Average Correct: " << avgCorrect << endl;
    cout << "<--------------------------------->" << endl << endl;

    nn.resetNodeVals();
    if(avgCorrect >= .98){
        break;
    }
}

```

```
    return nn;
}

char evaluate(vector<double> binaryInput, network &nn){
    nn.resetNodeVals();
    int numLayers = nn.networkNodes.size();

    /* Prep Input Layer */
    for(int i = 0; i<binaryInput.size(); i++){
        nn.networkNodes[0].at(i) = binaryInput[i];
    }

    for(int layer = 1; layer < numLayers; layer++){
        for(int currLayerNode = 0; currLayerNode < nn.networkNodes[layer].size(); currLayerNode++){
            double inVal = 0;
            for(int prevLayerNode = 0; prevLayerNode < nn.networkNodes[layer-1].size(); prevLayerNode++){
                inVal += nn.weights[layer-1][prevLayerNode][currLayerNode] * nn.networkNodes[layer-1][prevLayerNode];
            }

            double activationVal;
            if(layer == numLayers-1){ // Output Layer Activation Function
                activationVal = logisticSigmoid(inVal);
            }else{
                activationVal = hyperbolicTangentSigmoid(inVal);
            }

            nn.networkNodes[layer][currLayerNode] = activationVal;
        }
    }

    double maxOutputVal = 0;
    char actual = '0';
```

```
    // Evaluating output layer
    for(int outputNode = 0; outputNode<nn.networkNodes[numLayers-1].size(); outputNode++){
        if(nn.networkNodes[numLayers-1][outputNode] > maxOutputVal){
            maxOutputVal = nn.networkNodes[numLayers-1][outputNode];
            actual = 'A' + outputNode;
        }
    }

    return actual;
}

void evaluateNoise(network &neuralnet, vector< vector<int> > dottedIndexes, vector< vector<double> > charToInputMap, string outFile){
    ofstream noiseOut, rawNoiseOut;
    noiseOut.open(outFile);

    for(int i = 0; i<26; i++){
        char expected = 'A' + i;
        int flippedBits = 0;

        vector<int> currIndexes = dottedIndexes[i];
        int totalFilledBits = currIndexes.size();
        vector<double> binaryVector = charToInputMap[i];

        char actualVal = evaluate(binaryVector, neuralnet);

        while(actualVal == expected && currIndexes.size() > 0){
            int removeVal = rand() % currIndexes.size();
            binaryVector[currIndexes[removeVal]] = 0;
            currIndexes.erase(currIndexes.begin() + removeVal);

            actualVal = evaluate(binaryVector, neuralnet);
            flippedBits++;
```

```
            flippedBits++;
        }

        if(flippedBits == totalFilledBits){
            noiseOut << expected << " was evaluated correctly for all " << flippedBits << "/" << flippedBits << " flipped bits" << endl;
        }else{
            noiseOut << expected << " failed with " << flippedBits << " flipped bits. Evaluated to: " << actualVal << endl;
        }
    }
}

bool parseBDF(vector< vector<double> > &charToInputMap, vector< vector<int> > &dottedIndexes){
    //Open the BDF file
    ifstream bdfFile;
    bdfFile.open("im9x14u.bdf");

    for(int i = 0; i<26; i++){
        char character = 'A' + i;
        vector<double> charInput(126, 0.0);
        vector<int> charIndexes;

        //Receive 1-d input buffer based on the BDF file format
        int bbx, bby, bbxOff, bbyOff;
        bdfFile >> bbx >> bby >> bbxOff >> bbyOff;

        for(int j = 0; j<bby; j++){
            int currRow = j;
            string hex; bdfFile >> hex;
            string binary = hexToBinary(hex);

            for(int k = 0; k<binary.length(); k++){
                if(binary[k] == '1'){
                    int index = j * colSize + k;
                    charIndexes.push_back(index);
                    charInput[index] = 1.0;
                }
```

```
                }
            }
        }

        string fin; bdfFile >> fin;
        if(fin != "ENDCHAR"){ //Make sure parsing is correct
            cout << "ERROR" << endl;
            return false;
        }
        charToInputMap[i] = charInput;
        dottedIndexes.push_back(charIndexes);
    }

    return true;
}

/*
    Available Functions:
        parseBDF() - parses a bdf file
        readWeights() - read weights from a text file and inputs it to a neural network
        printWeights() - prints out the weights of a neural network to a given text file
        backPropogation() - trains a neural net
        (not really back propagation but also includes forward propagation, named only because we were meant to follow pseudo-code in book)

        evaluate() - evaluates a given binary input vector with a given neural network
        checkNoise() - checks how many "noise iterations" or flippedBits it takes until expected value != returned value
*/

int main(){
    srand(NULL);

    /*
    REQUIRED                        */
```

```
    vector< vector<double> > charToInputMap(26, vector<double>(126, 0.0)); // Map capital letters to 126 size one-dimensional input buffer
    vector< vector<int> > dottedIndexes; // Maps all the dotted indexes for a character
    bool parsed = parseBDF(charToInputMap, dottedIndexes); // Parse the BDF File
    if(!parsed){
        cout << "Error parsing" << endl;
        return 0;
    }
    network neuralnet; //Untrained neural net with random weights between -.1 and .1


    /*                      TEST IT YOURSELF!                      */

    // Trains an untrained neural net given BDF Format data
    network adjustedNet = backPropogation(charToInputMap, neuralnet);

    //Uses inputted weights from a text file
    readWeights(neuralnet, "adjustedweights.txt");

    // Print weights of a neural net to a text file after training
    printWeights(adjustedNet.weights, "adjustedweights.txt");

    // Evaluate how many noise is introduced to get a wrong value
    evaluateNoise(neuralnet, dottedIndexes, charToInputMap, "noisedata.txt");

    return 0;
}
```

Bibliography

3Blue1Brown, director. *YouTube*. *YouTube*, YouTube, 3 Nov. 2017, www.youtube.com/watch?v=Ilg3gGewQ5U&t=265s.

"IBM Fonts." *IBM Fonts by Farsil*, farsil.github.io/ibmfonts/.

Russell, Stuart J., and Peter Norvig. *Artificial Intelligence a Modern Approach*. Pearson, 2018.

"Stack Overflow." *Stack Overflow*, rep_movsd, stackoverflow.com/a/2704552.