Project Report: UI Data Science Competition using Neural networks

Umair Mohammad
Research Assistant at the Department of Electrical and Computer Engineering

Contents

University of Idaho

1. Task Description

This report describes the project submission for the UI Data Science Competition. The goal of the project was to correctly grade mathematical expressions consisting of arithmetic sums and subtractions. To elaborate, we were given a dataset with digits and arithmetic operators. The objective was to train a machine learning model to classify the digits and operators correctly and then use the trained model to grade mathematical expressions. The data was provided by the competition organizers and was available on the competition web-page (https://dscomp.ibest.uidaho.edu/).

1.1. Dataset description

The data consisted of 80,000 training samples and 20,000 test images. The training data comprised 24x24 pixel grayscale images of written digits from 0-9 and the arithmetic operators '+', '-' and '='. In addition to that, the data labels were provided such that the digits 0-9 were assigned the labels 0-9 and the operators '+', '-' and '=' were assigned the labels 10, 11 and 12, respectively. Therefore, there were 13 classes in total. Since it was a classification problem, the labels were converted into one-hot matrices such that for example, for 13 classes each image had a label vector **P** of length 13 with all values set to zero except the element corresponding to the class to which the sample belonged. For example, only element zero was set to '1' for an image belonging to class zero and so on. An example of the first 12 training images is attached below:

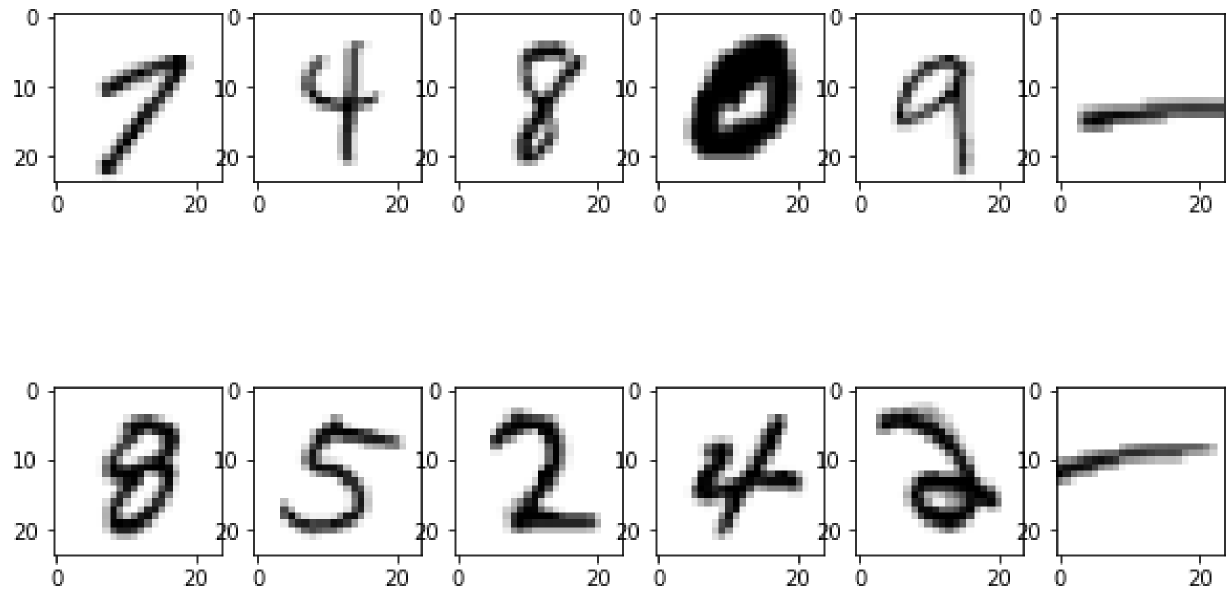
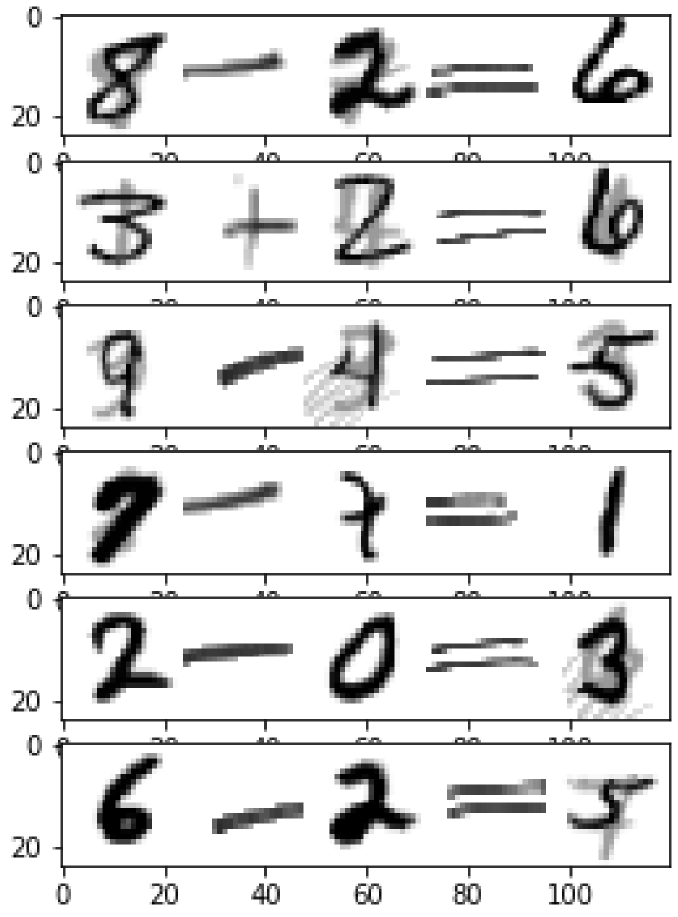


Figure 1: The first 12 train images of the UI Data Science Competition

The test data actually consisted of 20000 24x120 pixel grayscale images. Each image represents a complete mathematical expression. The expression consists of 5 elements in one of the following forms:

(i) a + b = c
(ii) a – b = c
(iii) a = b + c
(iv) a = b – c

The expressions a, b and c represent digits from 0-9 with the other two fields containing the operators. A few examples of the test images are shown below:



2. METHOD DESCRIPTION

The method used for classification is the artificial neural network. Two types of models have been used: Fully connected Neural Networks and Convolutional Neural Networks (CNN's). Convolutional neural networks are just a series of feature extraction stages followed by the neural network (NN) based classifier. Although we discuss all models and optimizers used in general, the major part of neural networks is based on unconstrained gradient descent optimization. So, this is how we organized our discussion: we give a brief description about fully connected neural networks, then discus the output calculation (forward pass) for the NN, then we go in depth of the major part which is the gradient descent (GD) approach for optimization of the NN parameters (backward pass). We discuss the one-layer case and the k-layer cases separately. Lastly, we talk about the modified optimizers based on the GD as well as briefly discuss the CNN.

2.1. **Forward Pass**

As described earlier, we have 80,000 grayscale images of size 24x24. For a fully connected NN, we transform the image into a vector. In our case, the inputs of each image or sample will be a 576 (24x24) grayscale values in vector form. In general, we have an input layer, K hidden layers, and an output layer. As mentioned, for our specific problem we have 576 inputs and 13 outputs in the training phase. We will describe the testing phase in more details in the experimental results section.

The optimization method used in modern NN's is not actually the original GD, but actually a modified version called the mini-batch stochastic gradient descent (SGD). The SGD is actually based on the GD. The only difference is that the data is shuffled, then split into mini-batches, and the training is done on one batch at a time rather than a single sample. In other words, the complete training dataset of 80,000 images (called an epoch) will be randomly shuffled, and then split into smaller sets (called batches). Based on this concept, let us see how we can calculate the output of a single-layer NN.

Let us denote one batch of data as X where X contains N images. Each image has D inputs (D = 576 in our example). Let us consider an NN that has one hidden layer. The input layer size is D, the hidden layer has L neurons and the output layer has M classes. The weight matrix for the first layer $W^{(1)}$ will be D*L and for the output layer $W^{(2)}$ it will be L*M. The biases are $B^{(1)}$ of size L and $B^{(2)}$ of length M. The input the first layer will be

$$o_{ij}^{(1)} = \sum_{k=1}^{L} x_{ik}\, w_{kj}^{(1)} + b_k^{(1)}, \qquad i = 1 \dots N; j = 1, \dots L$$

$$O^{(1)} = XW^{(1)} + B^{(1)}$$

$$u_{ij}^{(1)} = f_1\left(o_{ij}^{(1)}\right), \qquad i = 1 \dots N; j = 1, \dots L$$

$$U^{(1)} = f_1(O^{(1)})$$

$$o_{ij}^{(2)} = \sum_{k=1}^{M} u_{ik}^{(2)}\, w_{kj}^{(2)} + b_k^{(2)}, \;\; i = 1 \dots N; j = 1, \dots M$$

$$O^{(2)} = U^{(1)}W^{(2)} + B^{(2)}$$

$$u_{ij}^{(2)} = f_2\left(o_{ij}^{(2)}\right), \qquad i = 1 \dots N; j = 1, \dots M$$

$$U^{(2)} = f_2\left(O^{(2)}\right)$$

1.2.1 ***Extension to K-layers***

If we follow the same matrix notation, then the forward pass for K-layers can be written as:

$$O^{(1)} = XW^{(1)} + B^{(1)}$$
$$O^{(l)} = f_{l-1}\left(O^{(l-1)}\right)W^{(l)} + B^{(l)}, l = 1, \dots, K$$

Let us denote the number of inputs a D_0. Each hidden layer *l* that follows consists of D_l neurons. The weight matrix at each hidden layer $W^{(l)}$ is of size D_{l-1} x D_l and the bias vector is of size D_l. Once the output is calculated, the objective is to optimize the weights and biases based on a certain cost function. One issue that arises is how to initialize these weights and biases. It has been shown that generating initial weights using the normal distribution with zero-mean and standard deviation related to matrix size works very well. The biases can be initialized to zero.

2.2. Backward Pass (Backpropagation with SGD)

In the case of classification problems, we make a use certain fixed functions and models. For example, the activation function for all layers except the output layer (in our case the only hidden layer) is typically the relu function which can be defined as relu(a) = max(0, a). If the input is greater than 0, the output is the input itself, otherwise its zero. The derivative of this function will be only one where the input is greater than 1, (so it is a unit-step function). We will leave the activation to be general in our analysis. However, for the last layer, we will use the softmax an activation function, where

$$softmax\left(x_n | x_n \in x = [\, x_1, \dots, x_n, \dots, x_{N]}\right) = \frac{e^{-x_n}}{e^{-x_1} + e^{-x_2} + \cdots + e^{-x_n} + \cdots e^{-x_N}}$$

The softmax function will ensure that the output at each neuron in the last layer is the probability of the image being in the corresponding class. The best loss function (*$f_0(x)$*) to use for this type of output is called the cross-entropy loss and is given by,

$$f_0(y) = cross - entropy(y|x) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{M} x_{ij} \ln y_{ij}$$

where N represents the number of sample being tested and M is the number of possibilities. For our K-layer neural network described earlier, the output probabilities matrix $\mathbf{P}^O$ for each batch can be given by:

$$p_{nm}^{O} = \frac{e^{-o_{nm}^{K}}}{e^{-o_{n1}^{K}} + e^{-o_{n2}^{K}} + \cdots + e^{-o_{nm}^{K}} + \cdots e^{-o_{nM}^{K}}}, \qquad for\ n = 1, \dots, N\ and\ m = 1, \dots, M$$

The ground truth is X and Y is the estimated probability of a sample being in a given class. In a fully-connected NN (FCNN), we know the inputs and the output probabilities in the training phase. Therefore, the optimization problem is an unconstrained minimization (min *$f_0[x]$*) with respect to all weights and biases.

Let us apply the gradient descent algorithm to the backpropagation for the single hidden-layer NN. We want to optimize the weights and biases for each layer. Therefore, the weights and biases update rules based on the GD are as follows:

$$W_{t+1}^{1} = W_t^1 - \delta * \nabla W_t^1$$
$$B_{t+1}^{1} = B_t^1 - \delta * \nabla B_t^1$$
$$W_{t+1}^{2} = W_t^2 - \delta * \nabla W_t^2$$
$$B_{t+1}^{2} = B_t^2 - \delta * \nabla B_t^2$$

$$W_{t+1}^{l} = W_{t}^{l} - \delta * \nabla W_{t}^{l}\ , l = 1, \dots, L$$
$$B_{t+1}^{l} = B_{t}^{l} - \delta * \nabla B_{t}^{l}\ , l = 1, \dots, L$$

The update rule is for iteration *t+1* and δ is the learning rate. Unlike the traditional convex optimization techniques, the learning rate is constant or at least backtracking line search is not used. Instead, a simple decay may be introduced each epoch.

The main part of this algorithm is to now calculate the gradients of each weight and bias matrix. The easier way is to look at each individual element and then come up with matrix/vector notations. Before we do that let us define the cost function in terms of our classification problem. Please note that our output is the softmax values or probabilities for each class and the output is available as a one-hot matrix. In other words, it is a vector that is only 1 at the element that represents the class of the image.

Number (Class)	Image Representation	Outputs												
		P_0	P_1	d_2	d_3	d_4	d_5	d_6	d_7	d_8	d_9	d_{10}	d_{11}	d_{12}
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
2	2	0	0	1	0	0	0	0	0	0	0	0	0	0
3	3	0	0	0	1	0	0	0	0	0	0	0	0	0
4	4	0	0	0	0	1	0	0	0	0	0	0	0	0
5	5	0	0	0	0	0	1	0	0	0	0	0	0	0
6	6	0	0	0	0	0	0	1	0	0	0	0	0	0
7	7	0	0	0	0	0	0	0	1	0	0	0	0	0
8	8	0	0	0	0	0	0	0	0	1	0	0	0	0
9	9	0	0	0	0	0	0	0	0	0	1	0	0	0
10	+	0	0	0	0	0	0	0	0	0	0	1	0	0
11	-	0	0	0	0	0	0	0	0	0	0	0	1	0
12	=	0	0	0	0	0	0	0	0	0	0	0	0	1

As mentioned before, for our case, let us denote the output of the last layer as $P^{(O)}$ and the actual one-hat vector as $P^{(R)}$. We can now represent the loss function as:

$$\tilde{L} = -\frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} p_{nm}^{R} \ln p_{nm}^{O}$$

The total loss function is given below where θ is the regularization parameter and the additional term is called the regularization loss.

$$L = \tilde{L} + \vartheta \sum_{i=1}^{D}\sum_{j=1}^{L}\left[w_{ij}^{(1)}\right]^2 + \vartheta \sum_{i=1}^{L}\sum_{j=1}^{M}\left[w_{ij}^{(2)}\right]^2$$

The objective of the problem is to minimize this loss over all weights and biases.

$$\min_{W^1, B^1, W^2, B^2} L(\boldsymbol{P}^{O})$$

Extending to K hidden layers, we have the following total loss and objective.

$$L = \tilde{L} + \vartheta \sum_{l=1}^{K} \left\| W^{(l)} \right\|^2$$

$$\min_{W^l,\, B^l} L(\boldsymbol{P}^O)\,, l = 1, \dots K$$

As it can be clearly observed, the function is **convex**. It is a sum of 2 convex functions, the first part is negative log and the last term a quadratic program.

We have to calculate four different gradient matrices for the single hidden-layer NN: $\nabla W^{(1)}, \nabla W^{(2)}, \nabla B^{(1)}, \nabla B^{(2)}$. The reason this method is called the backpropagation is because we will calculate the gradients backwards.

$$\nabla B^{(2)} = \frac{\partial L}{\partial b_j^{(2)}} = \frac{\partial \tilde{L}}{\partial b_j^{(2)}}, j = 1, \dots, M$$

As we move backwards, we can see that the loss function is a function of the output probabilities which is actually a function of the weights and biases of the current layer and outputs of the previous layer. Before we start with the last layer, let us revisit the final (second/output) layer outputs. We noticed before that the final layer output is:

$$U^{(2)} = f_2\left(O^{(2)}\right)$$

Let us replace U^2 by P^O and set f_2 to be the softmax function as described earlier. Then

$$p_{nm}^O = \frac{e^{-o_{nm}^{(2)}}}{e^{-o_{n1}^{(2)}} + e^{-o_{n2}^{(2)}} + \cdots + e^{-o_{nm}^{(2)}} + \cdots e^{-o_{nM}^{(2)}}}, \qquad for\ n = 1, \dots, N\ and\ m = 1, \dots, M$$

The way to calculate the gradients is to apply **the chain rule**.

$$\frac{\partial \tilde{L}}{\partial b_j^{(2)}} = \sum_{n=1}^{N} \sum_{m=1}^{M} \frac{\partial \tilde{L}}{\partial o_{nm}^{(2)}} \frac{\partial o_{nm}^{(2)}}{\partial b_j^{(2)}}, j = 1, \dots, M$$

Let us tackle both gradients $\frac{\partial \tilde{L}}{\partial o_{nm}^{(2)}}$ and $\frac{\partial o_{nm}^{(2)}}{\partial b_j^{(2)}}$ individually.

$$\frac{\partial \tilde{L}}{\partial p_{nm}^{o}} = \frac{\partial \left[-\frac{1}{N} \sum_{n=1}^{N} \sum_{m=1}^{M} p_{nm}^R \ln p_{nm}^O\right]}{\partial o_{nm}^{(2)}}$$

$$= -\frac{\partial \left[\frac{1}{N} \sum_{n=1}^{N} \sum_{m=1}^{M} p_{nm}^R \ln \left(\frac{e^{-o_{nm}^{(2)}}}{e^{-o_{n1}^{(2)}} + e^{-o_{n2}^{(2)}} + \cdots + e^{-o_{nm}^{(2)}} + \cdots e^{-o_{nM}^{(2)}}}\right)\right]}{\partial o_{nm}^{2}}$$

$$= -\frac{\partial \left[\frac{1}{N} \sum_{n=1}^{N} \sum_{m=1}^{M} p_{nm}^R \left\{\ln\left(e^{-o_{nm}^{(2)}}\right) - \ln\left(e^{-o_{n1}^{(2)}} + e^{-o_{n2}^{(2)}} + \cdots + e^{-o_{nm}^{(2)}} + \cdots e^{-o_{nM}^{(2)}}\right)\right\}\right]}{\partial o_{nm}^{2}}$$

$$= \frac{\partial \left[\frac{1}{N} \sum_{n=1}^{N} \sum_{m=1}^{M} p_{nm}^R \left\{\ln\left(e^{-o_{n1}^{(2)}} + e^{-o_{n2}^{(2)}} + \cdots + e^{-o_{nm}^{(2)}} + \cdots e^{-o_{nM}^{(2)}}\right) - o_{nm}^{(2)}\right\}\right]}{\partial o_{nm}^{2}}$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} p_{nm}^{R}\left\{e^{-o_{nm}^{(2)}}\left(\frac{1}{e^{-o_{n1}^{(2)}} + e^{-o_{n2}^{(2)}} + \cdots + e^{-o_{nm}^{(2)}} + \cdots e^{-o_{nM}^{(2)}}}\right) - 1\right\}$$

$$= \frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} p_{nm}^{R}\{p_{nm}^{O} - 1\}$$

$$\frac{\partial \tilde{L}}{\partial o_{nm}^{(2)}} = \frac{1}{N}\sum_{n=1}^{N} p_{nm}^{O} - p_{nm}^{R}$$

$$\frac{\partial o_{nm}^{(2)}}{\partial b_j^{(2)}} = \frac{\partial \sum_{k=1}^{M} u_{ik}^{(2)}\, w_{kj}^{(2)} + b_k^{(2)}}{\partial b_j^{(2)}}, \quad i = 1 \dots N; j = 1, \dots M$$

$$\frac{\partial o_{nm}^{(2)}}{\partial b_j^{(2)}} = \begin{cases} 1, k = m \\ 0, k \neq m \end{cases}$$

$$\frac{\partial L}{\partial b_j^{(2)}} = \frac{1}{N}\sum_{n=1}^{N} p_{nj}^{o} - p_{nj}^{R}$$

The process for $\frac{\partial \tilde{L}}{\partial w_{ij}^{(2)}}$ is similar because we already have calculated $\frac{\partial \tilde{L}}{\partial o_{nm}^{(2)}}$. The remaining part is $\frac{\partial o_{nm}^{(2)}}{\partial w_{ij}^{(2)}}$.

$$\frac{\partial L}{\partial w_{ij}^{(2)}} = \sum_{n=1}^{N} \frac{\partial \tilde{L}}{\partial w_{ij}^{(2)}} + 2\vartheta w_{ij}^{(2)}$$

$$\frac{\partial \tilde{L}}{\partial w_{ij}^{(2)}} = \sum_{n=1}^{N}\sum_{m=1}^{M} \frac{\partial \tilde{L}}{\partial o_{nm}^{(2)}} \frac{\partial o_{nm}^{(2)}}{\partial w_{ij}^{(2)}}, \qquad j = 1, \dots, M$$

$$\frac{\partial \tilde{L}}{\partial o_{nm}^{(2)}} = \frac{1}{N}\sum_{n=1}^{N} p_{nj}^{o} - p_{nj}^{R}$$

$$\frac{\partial o_{nm}^{(2)}}{\partial w_{ij}^{(2)}} = \frac{1}{N}\sum_{n=1}^{N} u_{ni}^{(2)}$$

$$\frac{\partial \tilde{L}}{\partial w_{ij}^{(2)}} = \frac{1}{N}\sum_{n=1}^{N} (p_{nj}^{o} - p_{nj}^{R}) u_{ni}^{(2)}$$

$$\frac{\partial L}{\partial w_{ij}^{(2)}} = \frac{1}{N}\sum_{n=1}^{N} (p_{nj}^{o} - p_{nj}^{R}) u_{ni}^{(2)} + 2\vartheta w_{ij}^{(2)}$$

A similar process can be followed to derive the first-layer gradients. Please note that only the chain rule expressions and then the final expressions are given here.

$$\nabla B^{(1)} = \frac{\partial L}{\partial b_j^{(1)}} = \frac{\partial \tilde{L}}{\partial b_j^{(1)}}, j = 1, \dots, L$$

$$\frac{\partial \tilde{L}}{\partial b_j^{(1)}} = \sum_{n=1}^{N}\sum_{m=1}^{M} \frac{\partial \tilde{L}}{\partial o_{nm}^{(2)}} \sum_{a=1}^{N}\sum_{b=1}^{L} \frac{\partial o_{nm}^{(2)}}{\partial u_{ab}^{(2)}} \frac{\partial u_{ab}^{(2)}}{\partial o_{ab}^{(1)}} \frac{\partial o_{ab}^{(1)}}{\partial b_j^{(1)}}, \qquad j = 1, \dots,$$

$$\frac{\partial \tilde{L}}{\partial b_j^{(1)}} = \frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} (p_{nj}^{o} - p_{nj}^{R}) w_{jm}^{(2)} \nabla f_1\left(o_{nm}^{(1)}\right)$$

For the case of the Relu function, the derivative can be simplified to:

$$\frac{\partial \tilde{L}}{\partial b_j^{(1)}} = \frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} (p_{nj}^{o} - p_{nj}^{R}) w_{jm}^{(2)} \widetilde{o_{nm}^{(1)}}, \qquad \widetilde{o_{nm}^{(1)}} = \begin{cases} 1, & o_{nm}^{(1)} > 0 \\ 0, & else \end{cases}$$

$$\frac{\partial \tilde{L}}{\partial w_{ij}^{(1)}} = \sum_{n=1}^{N}\sum_{m=1}^{M} \frac{\partial \tilde{L}}{\partial o_{nm}^{(2)}} \sum_{a=1}^{N}\sum_{b=1}^{L} \frac{\partial o_{nm}^{(2)}}{\partial u_{ab}^{(2)}} \frac{\partial u_{ab}^{(2)}}{\partial o_{ab}^{(1)}} \frac{\partial o_{ab}^{(1)}}{\partial w_{ij}^{(1)}}, \qquad j = 1, \dots, L$$

$$\frac{\partial \tilde{L}}{\partial w_{ij}^{(1)}} = \frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} (p_{nj}^{o} - p_{nj}^{R}) w_{jm}^{(2)} x_{ni}^{(1)} \nabla f_1\left(o_{nm}^{(1)}\right)$$

Once again, for the Relu function we have:

$$\frac{\partial \tilde{L}}{\partial w_{ij}^{(1)}} = \frac{1}{N}\sum_{n=1}^{N}\sum_{m=1}^{M} (p_{nj}^{o} - p_{nj}^{R}) w_{jm}^{(2)} x_{ni}^{(1)} \widetilde{o_{nm}^{(1)}}, \qquad \widetilde{o_{nm}^{(1)}} = \begin{cases} 1, & o_{nm}^{(1)} > 0 \\ 0, & else \end{cases}$$

2.2.2 ***Extension to K-layers***

It is apparent that each gradient can be written as a function of the gradients of the previous layer. Please note that the differentiation operation is for the loss function L by default over the given variable adjacent to the del operator. The final expressions of the gradients are given in matrix form:

$$\nabla O^{K} = \frac{1}{N}(\boldsymbol{P}^{O} - \boldsymbol{P}^{R})$$

$$\nabla O^{(k)} = \left[\nabla O^{(k+1)}\right] W^{(k+1)} * f_k'\left(O^{(k)}\right), \qquad k \geq K-1, K-2, \dots, 3, 2$$

For Relu case:

$$\nabla O^{(k)} = \left[\nabla O^{(k+1)}\right]\left[W^{(k+1)}\right]^{T} * \tilde{O}^{(k)}, \qquad k \geq K-1, K-2, \dots, 3, 2$$

where $\widetilde{o_{nm}^{(k)}} = \begin{cases} 1, & \tilde{o}_{nm}^{(k)} > 0 \\ 0, & else \end{cases}, n = 1, \dots N \text{ and } m = 1, \dots, D_k$

$$\nabla W^{(1)} = X^{T}\left[\nabla O^{(1)}\right]$$

$$\nabla W^{(k)} = f(O^{(k-1)})^{T}\left[\nabla O^{(k)}\right], \qquad k \geq 2, \dots, K$$

$$\nabla B^{(k)} = \sum_{n=1}^{N} [\nabla o_{nm}^{k}], \qquad k \geq 2, \dots, K, \qquad m = 1, \dots, D_k$$

2.3. **Momentum and ADAM optimizers**

The other optimizers use the same approach as the gradient descent except for the update equations which are slightly changed. For example, in the momentum optimizer another update parameter called the momentum is introduced. For each Weight/ Bias Matrix for each layer, we calculate the momentum matrix based on the given rule, and then use it to update the weight/bias for each layer.

$$M_{t+1}^{l} = \beta M_{t}^{l} - \delta * \nabla W_{t}^{l}\,, l = 1, \dots, K$$

$$W_{t+1}^{l} = W_{t}^{l} + M_{t+1}^{l}\,, l = 1, \dots, K$$

A similar strategy is applied to the ADAM optimizer but we have an additional parameter.

$$M_{t+1}^{l} = \beta_1 M_{t}^{l} - (1 - \beta_1) * \nabla W_{t}^{l}\,, l = 1, \dots, K$$

$$S_{t+1}^{l} = \beta_2 S_{t}^{l} - (1 - \beta_2) * \nabla W_{t}^{l} * \nabla W_{t}^{l}\,, l = 1, \dots, L, \qquad * \sim element\ multiplication$$

$$M_{t+1}^{l} = \frac{M_{t+1}^{l}}{(1 - \beta_1)^{t}}, l = 1, \dots, K$$

$$S_{t+1}^{l} = \frac{S_{t+1}^{l}}{(1 - \beta_2)^{t}}, l = 1, \dots, K$$

$$W_{t+1}^{l} = \frac{M_{t+1}^{l}}{\sqrt{S_{t+1}^{l}} + \varepsilon}\,, l = 1, \dots, K$$

2.4. **Summary of optimization algorithms**

A summary of the algorithms is provided below

Summary of the stochastic mini-batch gradient descent algorithm

- Initialize weights and biases
- For epoch = 0: number of epochs or while accuracy < threshold
    - Shuffle training data randomly
    - Split into TrainSize/N batches of N images per batch
    - For batch_number = 0: TrainSize/N
        - Calculate the forward pass as described
        - Use the guessed output to calculate the loss
        - Calculate the gradients using the GD approach
        - Update the weights and biases matrix using the GD update equations
    - Calculate the validation accuracy using validation
    - Decrease the learning rate by the decay factor
- Predict the correct grades using the testing methodology

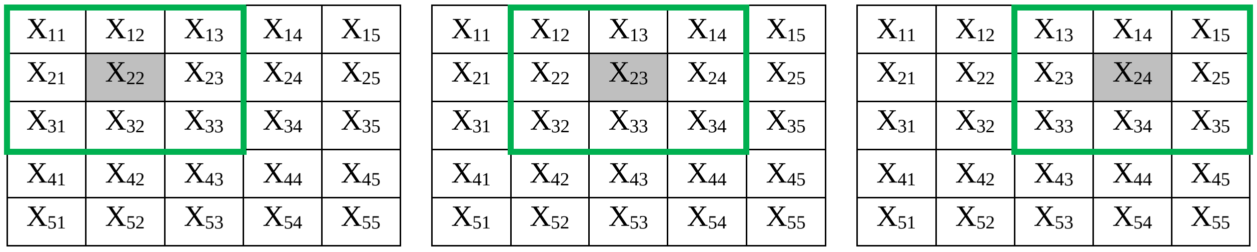
Summary of the ADAM or Momentum optimizers with SGD

- For epoch = 0: number of epochs or while accuracy < threshold
    - Shuffle training data randomly
    - Split into TrainSize/N batches of N images per batch
    - For batch_number = 0: TrainSize/N

        - Calculate the forward pass as described
        - Use the guessed output to calculate the loss
        - Calculate the gradients using the GD approach
        - ***If (optimizer == Momentum)***
            - ***Update the weights and biases matrix using the Momentum update equations***
        - ***Else***
            - ***Update the weights and biases matrix using the ADAM update equations***
    - Calculate the validation accuracy using validation
    - Decrease the learning rate by the decay factor
- Predict the correct grades using the testing methodology

2.5. **Convolutional Neural Network (CNN) and Augmentation**

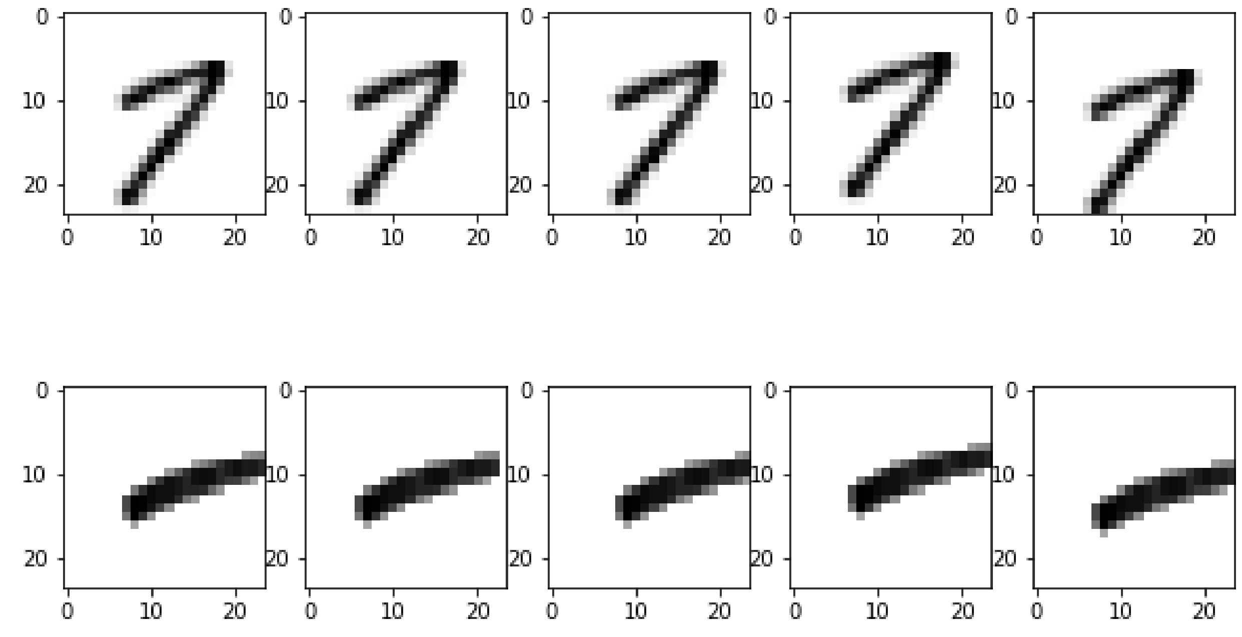
It turned out that using simple K-layer FCNN was not enough to achieve a high accuracy. Convolutional Neural Networks or (CNN's) were employed to what would essentially be feature extracting stages. The convolution operation is simply when one matrix or vector is moved across another larger vector and the output is the sum of the multiples of the elements of each. Typically, the filtering vector is the one being moved with a certain 'window' size and a stride. This operation can be carried out in 1D or 2D. The figure below shows a filter vector of window 3x3 moving across a larger 5x5 matrix with stride 1, i.e. the window is moved only by one element at each iteration.



(a) (b) (c)

The key parameters for the CNN method to extract features is the window size (typically 3x3) and the stride. Usually, the stride is kept as 1 and a pooling layer (typically maximum of 2x2 windows with stride 2) is implemented. Please note that each stride length will reduce the image size by the corresponding factor. If we start with 24 x 24, within 3 strides of 2 we will be down to a 3x3 image. To combat that, several convolutional layers are followed by pooling in small image datasets. It is also important to note that the input to a CNN is 2D and the outputs are channels representing the number of convolutions carried out at each layer.

It turned out the training data alone was still not enough to get the desired accuracy with CNN's. Therefore, a simple augmentation technique was used to improve the results. In our case, we generated a dataset five times larger than the original by translating each image left, right, up and down by 1 pixel. An example of the generated values for each image is shown below.



2.6. Testing Strategy

This problem was unique in the sense that the test set was not actually a version of the training images or an extension of the train set. Rather, actual expressions to be graded were given. The error metric or in other words, the testing accuracy were the percentage of the correctly graded assignments. Let us say that the true labels are stored in a 20000x1 vector Q^R and our predicted outputs are Q^O. Then the error can be given by:

$$Error = \frac{1}{20000} * \sum_{i=1}^{20000} |q_i^R - q_i^O|$$

The accuracy is just the complement of the error (1-*Error*).
For the actual testing function, it was noted that images 1, 3 and 5 are digits and the operators are in position 2 and 4.

$$Digit\ n = \arg\max P_{nm}^O, m = 0, \dots, 9$$
$$Operator\ n = \arg\max P_{nm}^O, m = 10, 11, 12$$

There are some cases where two operators may be reported, for example the confusion in the '-' and '=' signs and so on. Based on these rules, we can re-build the actual mathematical expression and use it to grade whether it is arithmetically correct.

3. Experimental Results

The best accuracy achieved was **98.86%** by using the augmented dataset with convolutional neural networks. The model that achieved this accuracy used one convolutional layer with a 3x3 kernel, stride one and 36 outputs. This layer was followed by a 2x2 max pooling layer with stride 2. This was followed by another convolutional layer with a 3x3 kernel, stride one and

48 outputs. This layer was followed by a 2x2 max pooling layer with stride 2. This layer was followed by a single-layer FCNN classifier with 288 neurons in the hidden layer and the 6 outputs of the second max-pool layer as the inputs. The last stage was of course the classification stage with 1 outputs. Batch normalization was used only in the FCNN stage. The initialization method for the variables was a Gaussian distribution with standard deviation $1/\sqrt(N_i)$, the ADAM optimizer with leaning rate 0.1, momentums 0.9 and a decay factor of 0.9 was used. This choice was made after several experiments.

The highest score achieved was actually 1.0. However, the score achieved while utilizing only 20,000 test equations was just **99.41%** so the next goal is to achieve this level of accuracy at least. The top 20 scores are displayed on the public leaderboard on the competition website. My score achieved the 10th position. Since the website is still active and accepting submissions, the next goal is to train more models and see if we can improve the accuracy. The next paragraph presents a summary of the results with different optimizers. Actually, more than 50 experiments were carried out. We will present a general summary and leave the reader to go over the tables of results with different configurations.

We started by using the (stochastic mini-batch training with) GD, Momentum and ADAM optimizers with just FCNN's. The best batch size was found to be 50, with a learning rate of 0.1 for the GD, 0.001 for Momentum and ADAM optimizers. It was discovered that a range of 0.001-0.003 gave the best results for both. With the GD and ADAM without any convolutional layers, we achieved a best-case test accuracy of around 94%. Momentum actually gave an accuracy of 95% without the CNN. However, when we applied the convolutional layers, the GD approach gave poor results and the momentum only improved to 96%. With the ADAM optimizer in collaboration with neural nets, an accuracy of up to 98% could be achieved. However, for the highest accuracy, augmentation was needed to get the performance to around 99%.

Simple Fully Connected Deep Neural Nets

Configuration	Optimizer	Learning Rate	Decay	Std. Dev. For W/B intital	Epochs	Validation Accuracy	Testing Accuracy
[576 288 124 60 13]	GD	0.1	0.956	$2/\sqrt(N_i+N_o)$	50	98.100%?	94.28%
[576 600 300 150 13]	ADAM	0.001	0.9	$1/\sqrt(N_i)$	100	98.794%	93.92%
[576 600 300 150 13]	ADAM	0.001	0.9	$2/\sqrt(N_i+N_o)$	20	98.800%	94.42%
[576 600 300 150 13]	ADAM	0.001	0.9	$2/\sqrt(N_i+N_o)$	30	98.831%	94.39%
[576 600 300 150 13]	ADAM	0.001	0.9	$1/\sqrt{N_i}$	30	98.862%	93.98%

Momentum Optimizer (Batch Size = 50) with and without CNN

Configuration	Learning Rate	Decay	Std. Dev. For W/B intital	Batch Normalization	Dropout	Epochs	Validation Accuracy	Testing Accuracy
[24x24 4 8 12 288 13]	0.001	0.9	$2/\sqrt(N_i+N_o)$	No	0.0	50	98.9%	95.90%

[24x24 120 60 36 13]	0.001	0.9	$2/\sqrt{(N_i+N_o)}$	No	0.0	50	99.26%	94.90%

CNN with Augmentation using ADAM (Batch Size = 50)

Attempt #	Configuration	Optimizer	Kernel	Learning Rate	Learning Rate Decay	Std. Dev. For W/B intital	Batch Normalization	Dropout	Epochs	Validation Accuracy	Testing Accuracy
1	[36 (max) 48 (max) 12 288]	ADAM	3x3	0.001	0.9	$1/\sqrt{(N_i)}$	No	0.0	25	99.559%	98.86%
2	[36 48 72 avg 84 96 120 max 288 120 36	ADAM	5x5 +3x3	0.001	0.9	$2/\sqrt{(N_i +N_0)}$	Yes (FC only)	0.0	30	99.654%	97.95
3	[36 48 72 avg 84 96 120 max 288 120 36	ADAM	5x5 +3x3	0.001	0.9	$2/\sqrt{(N_i +N_0)}$	Yes (FC only)	0.0	60	99.661	97.45

4. Conclusions, Challenges and Future Work

To sum up, in this course project we participated in the UI Data Science competition which was a classification problem to grade arithmetic expressions. We used convolutional nets and fully connected neural nets (deep learning) to achieve our task. Our best model achieved an accuracy of 98.86% which was 10th on the public leaderboard. Since this was an optimization class, we also focused on multiple optimizers and parameter tuning. We did a number of experiments and we present a partial list of results that achieved a good accuracy.

The challenges in this project included the large computational power required for training and the memory requirements, first time use of python and associated libraries for programming as well as being new to machine learning/deep learning in general. The idea for the future is to of course see how to achieve an accuracy of 1.0! Practically speaking, we want to run the network with more layers and larger number of output channels in the convolutional layer. Furthermore, more sophisticated configurations can be employed such as a series of convolutional layers followed by pooling. We can experiment with batch normalization and dropout with much larger output channels. Lastly, we may also try augmentation with more sophisticated operation such as distortions, sheer/elastic stretching and other transformations.