1 Technical Description

1.1 Notational Note

When denoting vector or matrix variables I use a capital letter to indicate the entire structure and an indexed lowercase letter to indicate a specific component value. As illustration $X = \begin{bmatrix} x^{(0)}_{11} & \dots & x^{(0)}_{1n} & \dots & x^{(0)}_{1N} \\ \dots & \dots & \dots & \dots & \dots \\ x^{(0)}_{m1} & \dots & x^{(0)}_{mn} & \dots & x^{(0)}_{mN} \\ \dots & \dots & \dots & \dots & \dots \\ x^{(0)}_{M1} & \dots & x^{(0)}_{Mn} & \dots & x^{(0)}_{MN} \end{bmatrix}$

When denoting indices I use a Mathcal formatted letter to represent the set of indices, the lowercase letter to indicate an element from the set of indices and the capital letter to indicate the largest value in the set of indices. As Illustrstion
$\mathcal{J} = \{1, ...j, ...J\}$

1.2 The Network Architecture

Experiments part 2 and part 3 both take the form of a neural network with the generic architecture of an input layer, a convolutional network with a specified quantity of convolutional filters, a flattening of the convolutional network outputs, a multilayer perceptron and then an output. TODO show picture here. Although these two parts are listed as seperate for the project description it is easiest to view them as the same structure of network with different hyperparameters. Consequently in the mathematical derivations and code implementation I describe the network generically using variables for all hyperparameters distinguishing these two project components. In the experimetn section I describe the specific values taken on by certain variables.

1.3 The Math of training our Network

1.3.1 forward

I first mathematically characterize our traing data set as the following

$$D = \{X^{(0)}(t), P(t)\}, X^{(0)} = \begin{bmatrix} x^{(0)}_{11} & \dots & x^{(0)}_{1c} & \dots & x^{(0)}_{1D} \\ \dots & \dots & \dots & \dots & \dots \\ x^{(0)}_{c1} & \dots & x^{(0)}_{cd} & \dots & x^{(0)}_{cD} \\ \dots & \dots & \dots & \dots & \dots \\ x^{(0)}_{C1} & \dots & x^{(0)}_{Cd} & \dots & x^{(0)}_{CD} \end{bmatrix}, P(t) \in \mathbb{Z}_R \qquad (1)$$

Here t is used to denote the index of our training sample in the dataset. The variable $X(t)^{(0)}$ refers to the tth sample of our data set and $P(t)$ refers to the corresponding label for that training sample. Our $P(t) \in \mathbb{R}$, where R is the number of possible labels and the space $\mathbb{R}$ is one-hot encoded so that it can be expressed by the output layer of our perceptron as well as for other computational purposes that are outlined in **??**. Each training sample is a matrix of dimension CxD.

The first hidden layer applied to the input sample of our network is the convolutional layer. The convolutional layer is defined by a kernel

$$A_l^{(0)} = \begin{bmatrix} a_{11}^{(0)} & \dots & a_{1n}^{(0)} & \dots & a_{1J}^{(0)} \\ \dots & \dots & \dots & \dots & \dots \\ a_{i1}^{(0)} & \dots & a_{ij}^{(0)} & \dots & a_{iJ}^{(0)} \\ \dots & \dots & \dots & \dots & \dots \\ a_{I1}^{(0)} & \dots & a_{Ij}^{(0)} & \dots & a_{IJ}^{(0)} \end{bmatrix} \tag{2}$$

. I combine all filters into one tensor A where l denotes the index of the filter and m and n are the indices of the receptive field. I know through convolutional arithmetic that

$$M = C - I + 1, N = D - J + 1. \tag{3}$$

This is, ofcourse, assuming that the stride is 1 and that the convolution operation conatins no padding. Each filter $A^(0)_l$ will be convoled across the input layer show in the following equation.

$$Z_{l,m,n}^{(0)} = \sum_{j\in\mathcal{J}} \sum_{i\in\mathcal{I}} a_{l,i,j} x_{m+i,n+j}^{(0)} + b_l^{(0)} \tag{4}$$

. In this case b_l denotes the bias of filter l. It is worth noting that in this notation the convolutional layer is subject to the constraint that all filters in at a given convolutional layer are the same size, which is not always the case, but is satisfied in all stages of this project.

After the inputs have passed through the convolution layer I apply a non-linear activation function

$$X_{l,m,n}^{(1)} = \sigma^{(1)}(Z_{l,m,n}) \tag{5}$$

I denote the generic non-linear activation as σ and describe the different non-linearities considered in the section **??**

After our training data has passed through the non-linearity it must traverse through the perceptron Layer. However, because of the current formulation of our convolutional layer our data is a tensor of shape (L,M-I, N-J). A fully conencted perceptron expects a 1-dimensional input so it is necessary to flatten the tensor out. This simply involves unrolling along the dimensions of our tensor via the equation

$$V^{(1)} = flatten(X_{l,m,n}^{(1)}) = \begin{bmatrix} v_1^{(1)} & \dots & v_k^{(1)} \end{bmatrix} \tag{6}$$

and re-indexing the flattened layer with the variable

$$k = l * NM + m * N + n. \tag{7}$$

We note that $K = L * (M - I) * (N - J)$. Now that we have a 1-dimensional input we pass it to the perceptron layer defined by the weights and bias.

$$W^{(0)} = \begin{bmatrix} w_{11}^{(0)} & \dots & w_{1k}^{(0)} & \dots & w_{1K}^{(0)} \\ \dots & \dots & \dots & \dots & \dots \\ w_{r1}^{(0)} & \dots & w_{rk}^{(0)} & \dots & w_{rK}^{(0)} \\ \dots & \dots & \dots & \dots & \dots \\ w_{R1}^{(0)} & \dots & w_{Rk}^{(0)} & \dots & w_{RK}^{(0)} \end{bmatrix}, b^{(1)} \tag{8}$$

Where R is the desired number of outputs. Note that this R is the same R as is seen in equation 1, because the output of our perceptron will be compared against the label of the datapoint we are currently observing. We pass the data through this layer via the qeuation

$$z_r^{(1)} = \sum_{k \in \mathcal{K}} w_{r,k} v_k + b^{(1)} \tag{9}$$

.

We then apply a second non-linear activation function.

$$x_1^{(2)} = \sigma^{(2)}(z_r^{(1)}). \tag{10}$$

The output of this non-linear activation will now be treated as a prediction for the label $P(t)$. We evaluate the effectiveness of this forward pass in predicting the label by using the vector $X^{(2)}$ and $P(t)$ as inputs to an error function which I denote generically as

$$E(X^{(2)}(t), P(t)). \tag{11}$$

We describe the different erorr functions considered in 1.4.1

1.4 BackPropagation

My network is structured so that there are 4 different classes of parameters to tune. The weights and bias of the perceptron and the weights and bias of the convolutional kernels. We denote them with the following partials:

$$\frac{\partial E}{\partial w_{r,k}}, \frac{\partial E}{\partial b^{(1)}}, \frac{\partial E}{\partial a_{l,i,j}}, \frac{\partial E}{\partial b_l^{(0)}} \tag{12}$$

and evaluate them via the chain rule

$$\frac{\partial E}{\partial w_{r,k}} = \frac{\partial E}{\partial x_r^{(2)}} \frac{\partial x_r^{(2)}}{\partial z_r^{(1)}} \frac{\partial z_r^{(1)}}{\partial w_{r,k}} \tag{13}$$

$$\frac{\partial E}{\partial b^{(1)}} = \frac{\partial E}{\partial x_r^{(2)}} \frac{\partial x_r^{(2)}}{\partial z_r^{(1)}} \tag{14}$$

$$\frac{\partial E}{\partial a_{l,i,j}} = \sum_{r \in \mathcal{R}} \frac{\partial E}{\partial x_r^{(2)}} \frac{\partial x_r^{(2)}}{\partial z_r^{(1)}} \sum_{n \in \mathcal{N}} \sum_{m \in \mathcal{M}} \frac{\partial z_r}{\partial x_{l,m,n}^{(1)}} \frac{\partial x_{l,m,n}^{(1)}}{\partial z_{l,m,n}^{(0)}} \frac{\partial z_{l,m,n}^{(0)}}{\partial a_{l,i,j}} \tag{15}$$

$$\frac{\partial E}{\partial b_l} = \sum_{r\in\mathcal{R}} \frac{\partial E}{\partial x_r^{(2)}} \frac{\partial x_r^{(2)}}{\partial z_r^{(1)}} \sum_{n\in\mathcal{N}} \sum_{m\in\mathcal{M}} \frac{\partial z_r}{\partial x_{l,m,n}^{(1)}} \frac{\partial x_{l,m,n}^{(1)}}{\partial z_{l,m,n}^{(0)}} \tag{16}$$

After these partials are determined we can use them to update the weights and biases of the convolutional kernels and perceptrion.

$$w_{r,k} := w_{r,k} - \alpha * \frac{\partial E}{\partial w_{r,k}} \tag{17}$$

$$b^{(1)} := b^{(1)} - \alpha * \frac{\partial E}{\partial b^{(1)}} \tag{18}$$

$$a_{l,i,j} := a_{l,i,j} - \alpha * \frac{\partial E}{\partial a_{l,i,j}} \tag{19}$$

$$b^{(0)} := b^{()_l} - \alpha * \frac{\partial E}{\partial b_l^{(0)}} \tag{20}$$

Here, α represents the learning rate of our network.

The next subsections discuss a few of the posisble values for the component partials which make up these equations

1.4.1 Error Functions

Most of our experiments are run using the squared error function

$$E_{squared-error}(X^{(2)}(t), P(t)) = \frac{1}{2} \sum_{r\in\mathcal{R}} (x_r^{(2)}(t) - P(t))^2. \tag{21}$$

Taking the partial as will be used for backpropagation we see

$$\frac{\partial E}{\partial x_r^{(2)}} = x_r^{(2)} - P(t) \tag{22}$$

1.5 Activation Functions

We consider multiple different activation functions. The first activation function considered is the hyperbolic tangent denoted

$$x = \sigma_{th}(z) = \frac{e^{(2z)} - 1}{e^{(2z)} + 1} \tag{23}$$

. The tangent

$$\frac{\partial \sigma_{th}}{\partial z} = 1 - \sigma_t(z) \tag{24}$$

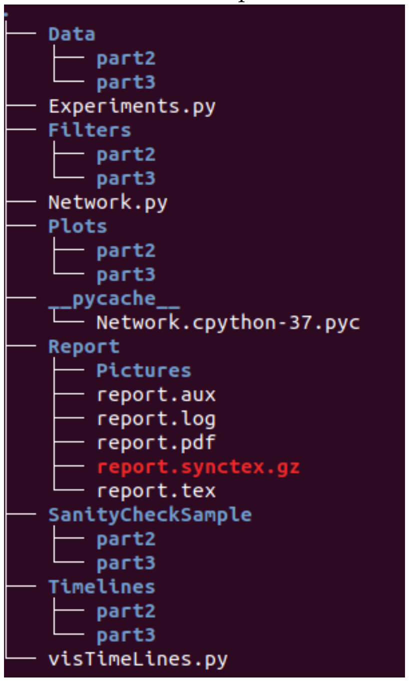
$$x = \sigma_s(z) = \frac{1}{1 + e^{-z}} \tag{25}$$

$$\frac{\partial \sigma_s}{\partial z} = \sigma_s(z)(1 - \sigma_s(z)) \tag{26}$$

Both of these activation functions have an advantage of their partials being functions of the activation function. This is useful in computation of gradients during backpropagation

2 Code Description

The files used in this report are stored the directory tree shown below.



```
.
├── Data
│   ├── part2
│   └── part3
├── Experiments.py
├── Filters
│   ├── part2
│   └── part3
├── Network.py
├── Plots
│   ├── part2
│   └── part3
├── __pycache__
│   └── Network.cpython-37.pyc
├── Report
│   ├── Pictures
│   ├── report.aux
│   ├── report.log
│   ├── report.pdf
│   ├── report.synctex.gz
│   └── report.tex
├── SanityCheckSample
│   ├── part2
│   └── part3
├── Timelines
│   ├── part2
│   └── part3
└── visTimeLines.py
```

The python module *Network.py* is where the Network is defined. Module contains the following Methods:
# Loss Functions:
SquaredError
# Activation Functions:
tanhAct, sigmoidAct, reluACt, noAct
# Auxillary Functions:
SaveImagesAndTimelines getTrainingData, DecisionLayer
# runNetwork

The bulk of the computations are performed in the *runNetwork* method. The method takes one dictionary parameter which contains all configuration settings that define an individual experiment including auxilarry information and hyper-parameters. With the variables I tried to make the notation as consistent as possible with the derivations outlined in the paper.

The *runNetwork()* method is called with various iterations of the potential parameters within the *Experiments.py* module. The possible configurations settings are as follows:

Epochs	number of Epochs	
ExNum	Part Num	2,3
T	Num Train Sample	32,100
TT	Num Test Sample	98,100
L	num Conv Layers	Int
I	conv Kernel Height	Int
J	conv Kernel Width	Int
C	Image Height	Int
D	Image Width	Int
R	Perceptron Outputs	2,20
lrW	Learning Rate	Float
lrA	Learning Rate Conv	Float
ACT	Activation Function	tanhAct, sigmoidAct, reluAct, noAct
LOSS	Loss Function	SquaredError
ORDER	Order of training Data	Fixed, Shuffle, Random
vis	Visualize Data	True,False

3 Experiments

3.1 Order

I run the network for experiment 2 in three different orders: Fixed, Shuffled, and Random. With the sequential ordering I train over all samples with label 1 followed by all samples with label 3. In Shuffled I mix the training data so that it alternates between samples of 3 and 1 every iteration. With random I present the samples to the network in a random ordering. In all three orderings the Network is presented each sample once per epic

3.2 Learning Rate

I run the network with across learning rates increasing by powers of 10. $\alpha \in \{0.1, 0.01, 0.001, 0.0001\}$

3.2.1 Varied Learning Rates

In noticed that when looking at the partials $\frac{\partial E}{\partial W}$ and $\frac{\partial E}{\partial K}$ that the partials were significiantly larger in $\frac{\partial E}{\partial W}$. In retrospect this makes sense as the convolutional layer occurs later in the backpropogation step. I know that this phenomenon is referred to the vanishing gradient problem and often occur in deeper networks, but did not realize it would happen in a network as shallow as this one. I adjusted the network so that rather than applying a universal learning rate to all gradient descent steps I used a larger learning rate for the convolutional kernels. $\alpha_a = 0.01$ $\alpha_w = 0.0001$

3.3 Freezing Perceptron Weights

3.4 Debuging error

I am fairly certain there is a single, minor error somewhere in my code that is negatively impacting the results of these experiments. Because of the way this was coded up the parameters for number of filters and filter size do not alter implementation and the only thing differentiating part 3 and part 2 is some configuration settings.