Eli Pinkus
CS 155
Set 4

1 Deep Learning Principles

Problem A

Part i

We begin by noting that the first of the above networks has weights initialized seemingly at random to some small value ~<0.5. As this network trains, the performance increases until it reaches 0 error.

We also note that the second network has weights initialized to 0. Nothing happens as this network trains. All the weights start at 0 and remain zero.

In order to see why this happens, we need only look at the ReLU function and the signal at a given neuron. The signal is always $w^T x$, where w is some vector of weights that is in the overall matrix of weights W, and x is an equal dimensioned vector that represents the input to some layer. The ReLU function is defined:

$$\mathrm{ReLU}(s) = \begin{cases} s, & s > 0 \\ 0, & s \leq 0 \end{cases}$$

So if we consider the iterative optimization using gradient descent that occurs for a given neuron during backprop we see that:

$$\nabla_s \mathrm{Re}LU(S) = \begin{cases} 1, & s > 0 \\ 0, & s \leq 0 \end{cases}$$

Since the calculation of the overall gradient of the error consists of summations of gradients of signals of the form $w^T x$

and since $w = 0 \; \forall w \Rightarrow w^T x = s = 0 \;\; \forall w, x$

We know that all components of the gradient will be 0

We can see that the update rule becomes:

$$W^{(t+1)} = W^{(t)} - 0$$
$$= W^{(t)} - 0$$
$$= W^{(t)} = 0$$

So, the model never changes at any layer or node since weights for all nodes and layers are initially 0.

Part ii

The first network starts with low random weights and they slowly begin to increase and the model collectively comes to achieve a very low loss similar to before. This model takes significantly longer to achieve good training relative to the model in part i. Additionally, the boundary that is learned is smoother whereas the model in part I learned a boundary that looks like a hexagon with sharp edges.

The speed of training is due to the fact that the sigmoid has a derivative that approaches zero as the signal gets large or small, so the weights change slower via gradient descent compared to ReLU units that have constant gradients until a threshold is passed (i.e. signal becomes ≤0) and then the gradients are 0.

The second network begins by make barely noticeable changes until around 3500 epochs at which point the network updates more drastically althought the resulting fit behavior is pretty poor. Because the weights are initialized to 0, the result of each layer $\sigma(w^T x)$ will start out very small which means that the updating rule with is dependent on that quantity from previous layers, will change at a very slow rate until it reaches a threshold were it can change more drastically. The loss function reaches a local minimum with a poorly fit strait line through the

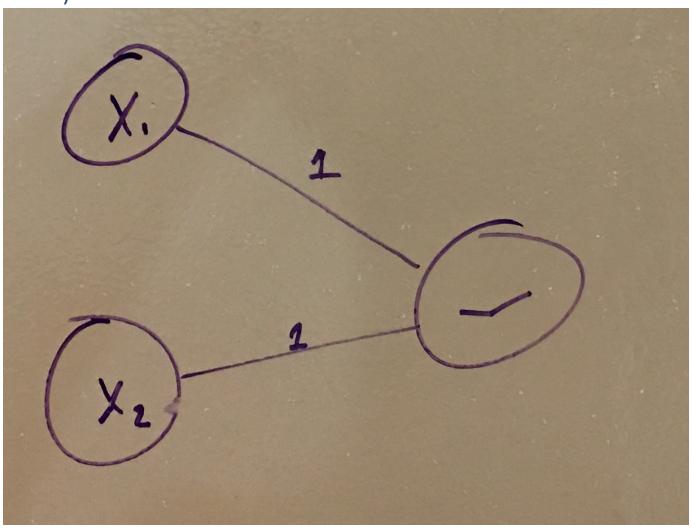
data and the nature of the sigmoid makes it difficult to escape this local min because the gradient of the sigmoid approaches zero as the signal becomes arbitrarily large or small.

Problem B

If we loop through all the negative exmples first, the weights for features that are indicative of negative outputs will become very large while the weights that are indicators for positive outputs will go to zero and will die irrecoverably when the positive points are read at which point the opposite would happen and we would end up with a dead network.

Problem C

Part i)

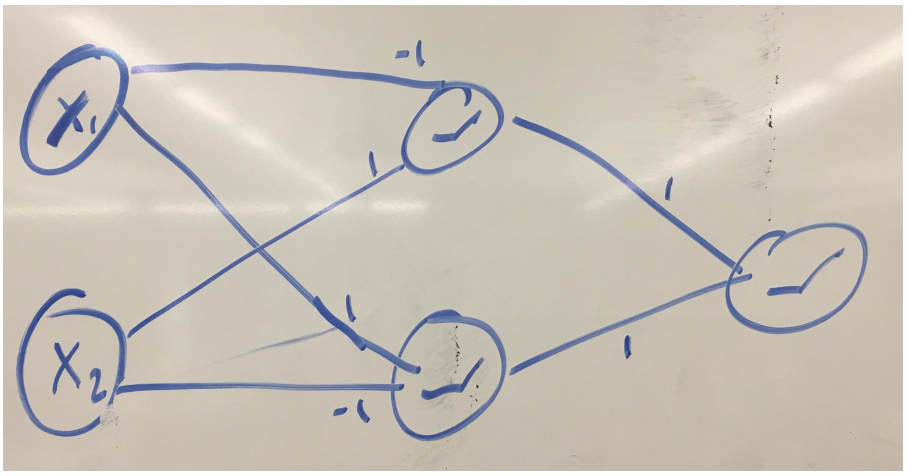


Part ii)

The minimum number of hidden layers we would need to implement the XOR function would be one. A network with fewer than 1 hidden layer cannot compute XOR because XOR is an inherently non-linear function and therefore cannot be modeled by a linear method and a 0 hidden layer neural net is a linear model.

$$\mathrm{XOR}(x_1, x_2) = \mathrm{OR}(x_1, x_2) \text{ and } \left(\text{not } \mathrm{AND}(x_1, x_2)\right)$$

So the first layer can have units for $\mathrm{OR}(x_1, x_2)$, $\left(\text{not } \mathrm{AND}(x_1, x_2)\right)$ and the next layer operate an and unit with the results



2 Depth vs Width on the MNIST Dataset

Problem A

```
tensorflow==1.5.0
Keras==2.1.3
```

Problem B

Part i)

Each input consists of a 28 by 28 matrix of values (values from 0 to 255), that indicate the intensity of each pixel in a 28 by 28 picture of a hand drawn digit.

Part ii)

The new shape of each training point is a $28 \times 28 = 784$ length vector that represents the same data as the matrix but with each row in the matrix sequentially located in the vector.

Problem C

```
Test score: 0.12535707663121862
Test accuracy: 0.9785
```

Problem D

```
Test score: 0.12291225457715546
Test accuracy: 0.9808
```

Problem E

```
Test score: 0.0615268822472106
Test accuracy: 0.9832
```

3 Convolutional Neural Networks

Problem A

Paddings can be helpful because it prevents downsizing of the image. If we didn't zero pad, it would only take a couple such convolutions to reduce to an image that was too small to be useful.

On the other hand, the information computed by the convolution on patches with zero paddings isn't quite as relevant as the data from non padded patches. The addition of zeros introduces an arbitrary source of input for the convolution computation that can make the resulting data less useful although the size of the output will be preserved.

Problem B

Part i)

Input size 5x5x3
8 filters
Stride 1

Parameters with biases = 8x5x5x3+8 biases = 608 parameters

Part ii)

Each dimension of the image will be reduced to $32 - (5 - 1) = 28$
There will still be 3 channels:

Tensor shape: $28 \times 28 \times 3$

Problem C

Part i

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1/2 \\ 1/2 & 1/4 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1/2 & 1 \\ 1/4 & 1/2 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1/4 & 1/2 \\ 1/2 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1/2 & 1/4 \\ 1 & 1/2 \end{bmatrix}$$

Part ii)

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

Part iii)

Pooling can help filter out noise via averaging in order to reduce the impact that a 'missing' pixel will have on the resulting values if we wish to do further computation. We also consider that in images for example, relevant patterns tend not to occur on the individual pixel leve. Thus, in these cases some levels of pooling allow us to preserve useful data while reducing dimensionality which is good for generalization.

Problem D

Final model:

```
Test score: 0.03665039017227246
Test accuracy: 0.9873
```

Dropout = 0.1:

```
Test score: 0.03374705762484664
Test accuracy: 0.9887
```

Dropout = 0.2
```
Test score: 0.03534207882474002
Test accuracy: 0.9869
```

Dropout = 0.3
```
Test score: 0.044779233676521105
Test accuracy: 0.9852
```

Dropout = 0.4
```
Test score: 0.05504944748673588
Test accuracy: 0.9833
```

Dropout = 0.6
```
Test score: 0.08381787260100246
Test accuracy: 0.9794
```

Dropout = 0.7
```
Test score: 0.10473713158369065
Test accuracy: 0.9722
```

Dropout = 0.8
```
Test score: 0.2807534393787384
Test accuracy: 0.9248
```

Dropout = 0.9
```
Test score: 0.5690752512454986
Test accuracy: 0.8706
```

Dropout = 1
```
Test score: 0.0723223465348247
Test accuracy: 0.985
```

What I found to be an effective strategy to modify the sample net was have increasingly strong dropout layers in the network feeding forward. This seems to have helped a lot in preventing overfitting and it makes sense considering we have a higher chance of overfitting as we get deeper into the network because the representations of the data are increasingly abstract so we need to make sure we maintain a level of stochasticity to prevent overfitting.

This trial and error approach seems not so exhaustive and it seems like we could be leaving a lot of available performance on the table relative to a more principled inspection of various hyperparameters. It is also difficult to accurately predict the performance of a model in only and handful of epochs. With the complexity of the networks, the loss surface is highly complex and as such it might have unexpected changes in concavity and other properties that may lead to drastically better performance long term, in a model that doesn't seem to be training too well short term.