Intro to AI

Joshua Redona

December 10, 2023

Task 1

Input Space

For the first task, our input space is constructed by reading each pixel and encoding their color to one of the following:

$$\text{R} : [1, 0, 0, 0]$$
$$\text{Y} : [0, 1, 0, 0]$$
$$\text{B} : [0, 0, 1, 0]$$
$$\text{G} : [0, 0, 0, 1]$$
$$\text{E} : [0, 0, 0, 0]$$

where E represents an empty, non colored pixel. The other colors should be quite straightforward. The resulting 2D matrix is flattened to a (1 x 1600) sized array. Since we are both training and testing in batches of n diagrams, our final input space is a matrix of size (n x 1600)

Output Space

Since this is a binary classification, our one hot encoded output would be of the following form:

$$\text{Dangerous} : [1, 0]$$
$$\text{Not Dangerous} : [0, 1]$$

When actually implementing this however, it was easier to isolate just one column and use this for calculating our loss function, so our actual output space, A, is a matrix of size (n x 1), where each row contains a float from 0 to 1. Our prediction function takes in A and encodes

$$A[i] > 0.5 : [1]$$
$$\text{else} : [0]$$

to get our one hot encoded prediction.

Model Space

We have 2 matrices used for decision making that are trained, which is 1 weight matrix and 1 bias matrix, denoted W and b respectively. W is of size (1 x n) and b is of size (1 x 1).

Cost Function

Copied exactly from the training function, our cost function is as follows:

$$J(Y, A) = -\frac{1}{m}\sum_{i=1}^{m}[Y_i \log(A_i) + (1 - Y_i)\log(1 - A_i)]$$

Training Algorithm

For task 1, we decided to go with a one layer neural network. Our forward propagation only calculates one preactivation layer, Z, using

$$Z = W * A_0 + b$$

and since we are training for binary classification, we use the sigmoid activation function to get our output layer,

$$A_1 = \text{sigmoid}(Z)$$

For backward propagation, we calculate

$$dz = A_1 - Y$$

where Y is our one hot encoded labels. Then we simply calculate dw and db, which are proportional to the amount we want to change W and b to reduce the cost function. We get these values from directly using dz as follows:

$$dw = \frac{1}{n}\sum(dz * A_0)$$
$$db = \frac{1}{n}\sum dz$$

where n is the number of diagrams being trained. Finally, we adjust W and b to give us new predictions that should line up better with the labels. To do this, we choose a learning rate α, and implement the dw and db values we just found:

$$W = W - \alpha * \text{dw} \quad \text{and} \quad b = b - \alpha * \text{db}$$

Prevent Overfitting

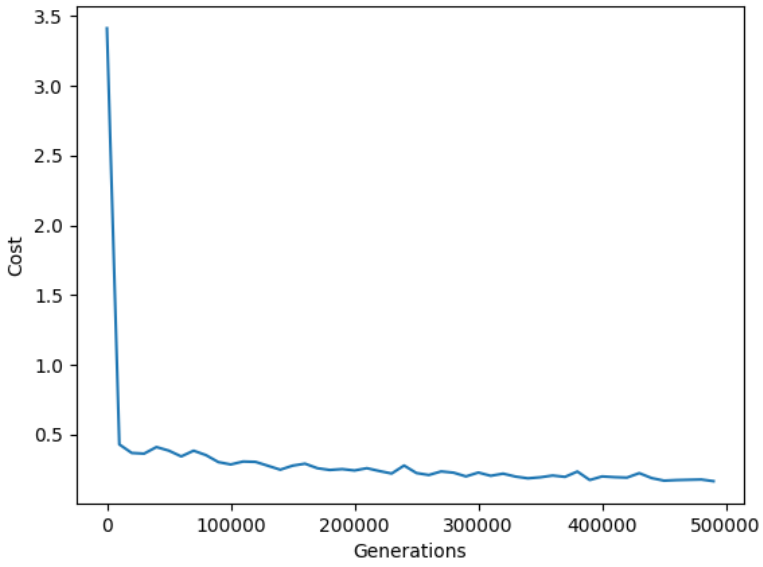
One of our biggest problems was overfitting when we first implemented the neural network. To mitigate these effects, we have tried a number of solutions.

First, we reduced our learning rate, nearly to 0, and increased the number of generations we ran the model for. Before this, we had $\alpha = 0.6$ as the learning rate, and our models would quickly (in about 3 generations) converge to a 1.0 accuracy. However, we noticed that when we switched to our test data, the bot would go back to around 0.5 accuracy, with the exact same guesses as the previous input. After the switch, we used $\alpha = 0.03$ over 5000 generations, and while convergence was no longer returned 1.0 accuracy, the bot no longer kept its previous guesses when switching datasets.

More importantly, we started switching the dataset every 5000 generations (after it converges for that set). That is, we generate an entirely new set of diagrams and labels for the network to train on and use for adjustments. While we can achieve similar results to shuffling the dataset just as often, we found it easier to implement generating a new dataset instead, and the result is the networt handles new data much better.

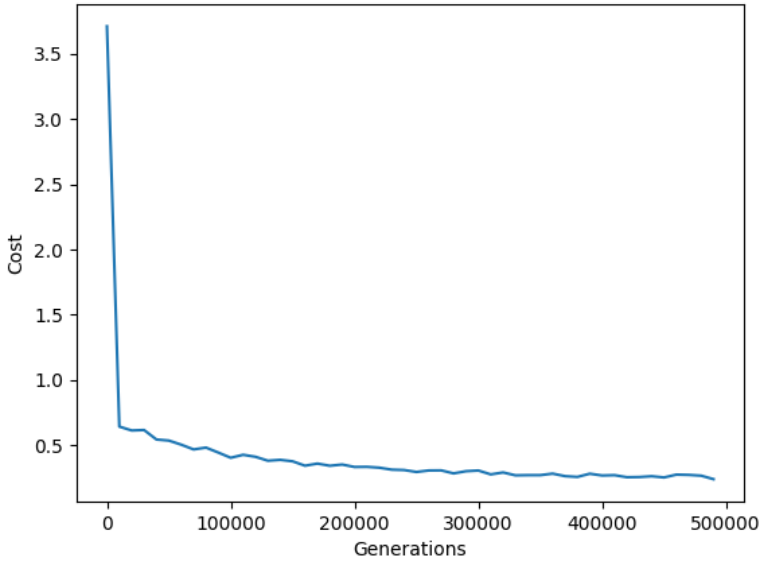
Performance

500 images



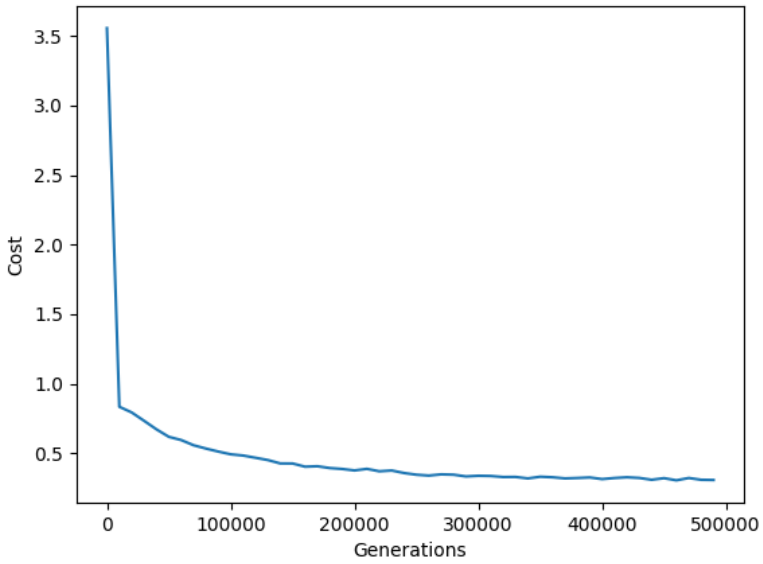
The accuracy for 500 test images was 0.712, or 71.2% accurate.

1000 images



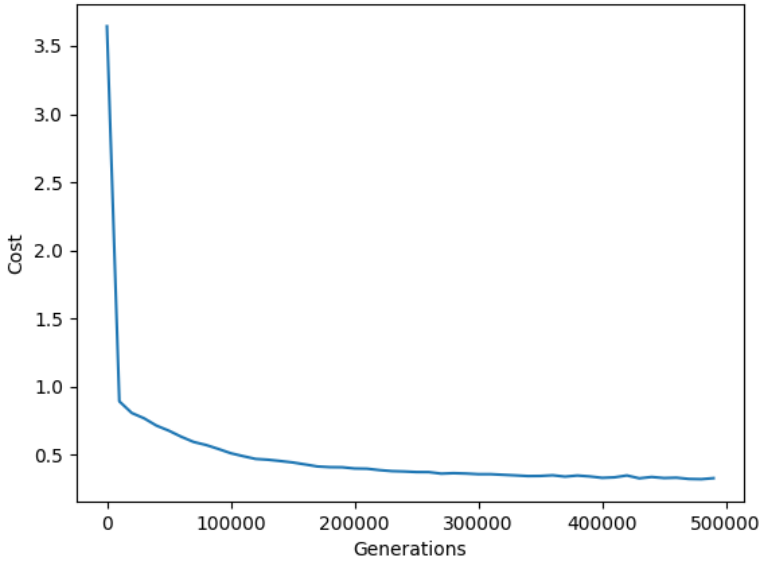
The accuracy for 1000 images was 0.766 or 76.6% accurate.

2500 images



The accuracy for 2500 images was 0.828 or 82.8% accurate.

5000 images



The accuracy for 5000 images was 0.825 or 82.5% accurate.

Assessment

Our model for task 1 is fairly good. We see the trend that using more data during training results in a higher average accuracy for the model. It is apparent from each of the graphs that the loss function drops out fairly quickly and the model quickly converges. In all likelyhood, running this bot for more than 300k generations was unecessary since there doesn't appear to be too much change after this point in each graph.

We can see that the bot generalizes well since it's guesses for the test data, a data set that it has not seen before, is significantly more accurate than the 50% we expect from a bot that is randomly guessing, or one that has overfit the data its training on.

Task 2

Input Space

For the second task, we used the exact same input space. To recap, we use

$$\begin{aligned} \text{R} &: [1, 0, 0, 0] \\ \text{Y} &: [0, 1, 0, 0] \\ \text{B} &: [0, 0, 1, 0] \\ \text{G} &: [0, 0, 0, 1] \\ \text{E} &: [0, 0, 0, 0] \end{aligned}$$

and flatten each image to a (1 x 1600) matrix, to be used as the input, and for n diagrams used, we have a final input of size (n x 1600).

Output Space

Since we now have 4 classes for the bot to predict, our output space must change to reflect that. Luckily, since each class represents a color, we can use the same encoding for the input to represent a one hot encoded label. That is,

$$\begin{aligned} \text{R} &: [1, 0, 0, 0] \\ \text{Y} &: [0, 1, 0, 0] \\ \text{B} &: [0, 0, 1, 0] \\ \text{G} &: [0, 0, 0, 1] \end{aligned}$$

for each color being the third color placed in the image. The result is that when we input a matrix representing n diagrams, we output a matrix of size (n x 4).

Model Space

Since we need to account for 4 different classes to predict, our weights matrix becomes (4 x n) and our bias matrix becomes (4 x 1). We can no longer use just one of the columns to represent the bot's full prediction (since our choice is no longer a binary one), so we must output all 4 nodes of the final layer. The result is that our weights and bias need to account for the shape of our output.

Cost Function

The cost is as follows:

$$\text{cost} = -\frac{1}{n}\sum_{i=1}^{n} Y_i \log(A_i + \epsilon)$$

Note that ϵ is here to prevent errors for when $A_i = 0$, and is exactly set to the value $\epsilon = 1e - 15$.

Training Algorithm

We're using a very similar neural network to the one we used for task 1. The math is exactly the same, with the only exception being our choice of activation function. Because we now have multiple classes, we chose to use $g(z)$ =ReLU(z).

Another slight difference is that we have chosen to use more generations to train this task, since the problem is considerably more complex than the last. To be more specific, we are running the network over 220k generations to return a trained weights and bias matrix. Of course, many of these generations are in place for the 5000 images case we have to go over, and we decided to run each case with the same number of generations to keep the graphs and data consistent.

Preventing Overfitting

Most notably, we were able to successsfully implement shuffling the data rather than generating a new dataset. This is particularly useful for this task since we are generating dangerous images by generating until we find a dangerous image.

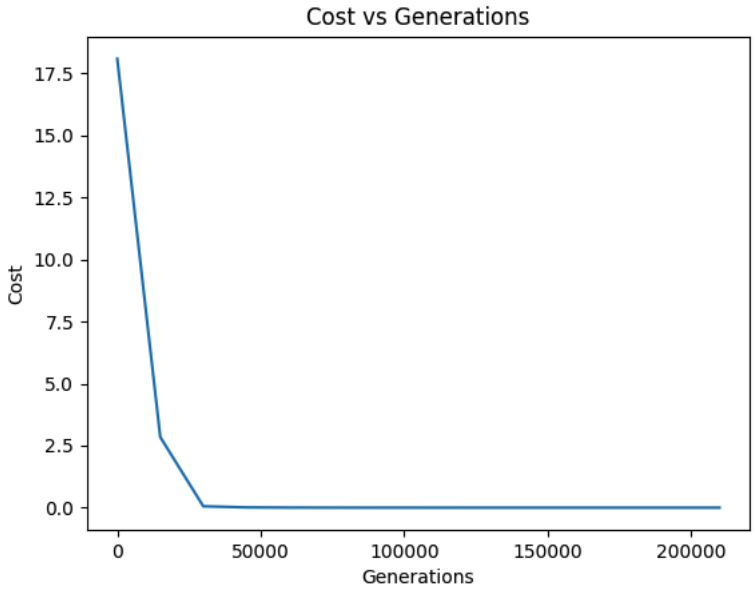
Since we have more generations to run the network for, we shuffle the dataset every 15000 generations in order to prevent overfitting. With a bit of trail and error, we have set the learning rate for this network to $\alpha = 0.02$, which seems to maximize the accuracy of the final prediction.

Performance

Let us go over each graph of the loss function, as well as the accuracy of the model after being trained. Please note that because of our specific implementation, every accuracy we will mention is evaluated by testing out the same number of images as we trained, albeit being generated after all the training has been done.

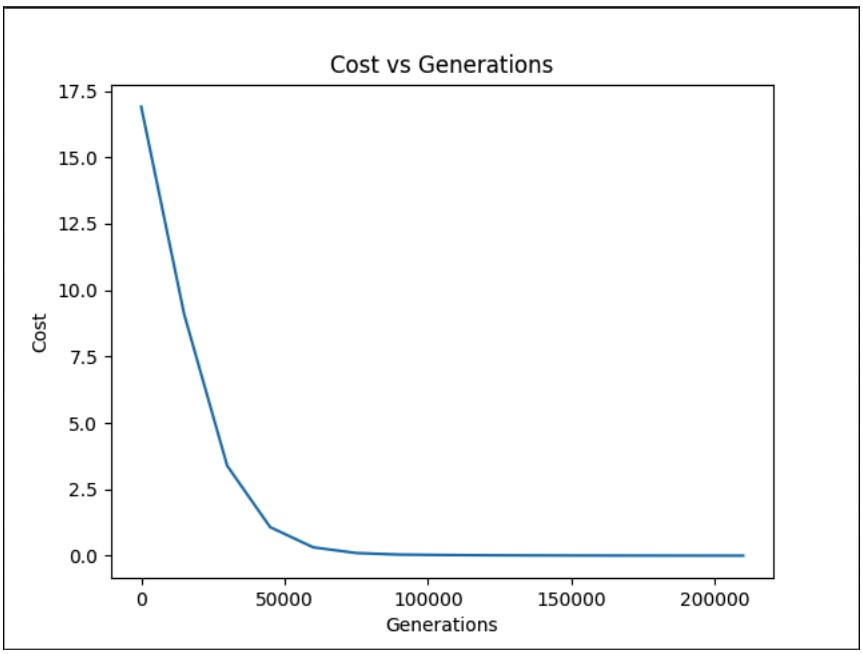
It is worth noting that each point in the graph is generated after 15000 generations, at the same time that the dataset is shuffled as mentioned in the overfitting section.

500 images



We can see that for 500 images, the graph quickly drops off to a value of $2.35e-7$, which in comparison to the starting magnitude of 18.0 is essentially 0. While the cost function has dropped nearly all the way, the bot yields an accuracy of 0.778, or 77.8% when given 500 test images.

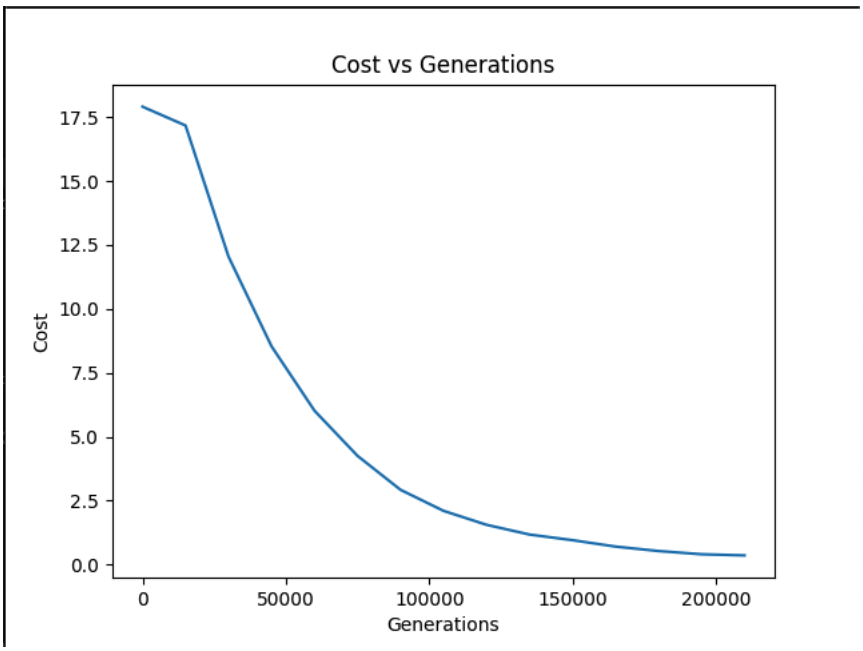
1000 images



The curve is considerably smoother than the 500 image case, but still drops off to a final value of 0.003, which again, is relatively close to 0 in comparison to the

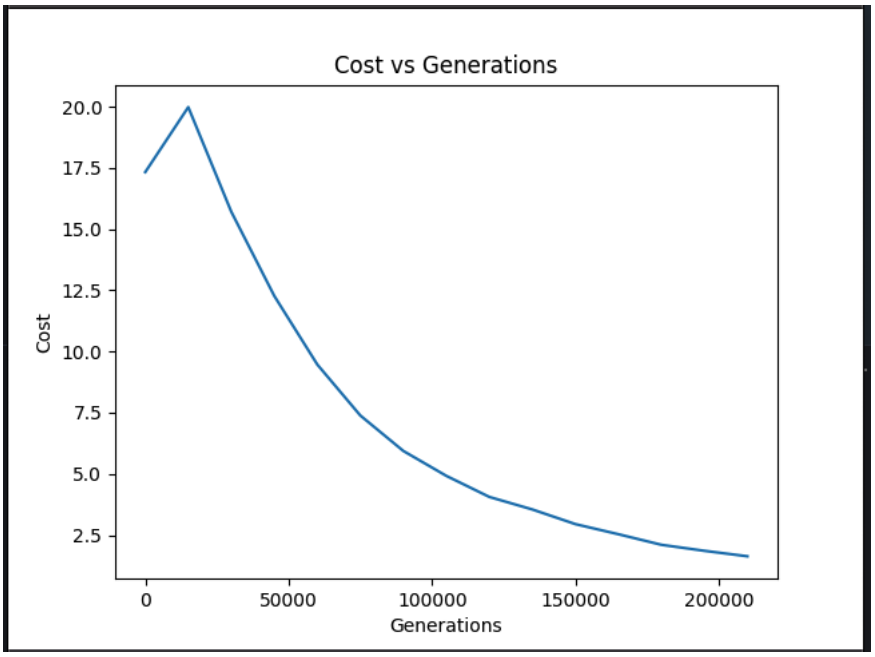
starting value of 16.9. The model still predicts with decent accuracy of 0.746, or 74.6% correct guesses out of 1000 test images.

2500 images



With 2500 images, it takes a considerable amount of time for the cost function to drop off, and it ends with a final value of 0.359. Despite a cost that is somewhat higher than the previous two examples, the model still yields an accuracy of 0.758, or 75.8% correct guesses of the 2500 test images.

5000 images



While we initially have a spike in the cost function, it proceeds to drop off just as the other examples have. The final value we get to is 1.63, which is much higher than any other case, but it still yields an accuracy of 0.761, or 76.1% correct guesses out of 5000 test images.

Assessment

Our neural network for task 2 does well for our implementation. Our loss function consistently goes down and we have our model consistently predicting with just about 75% accuracy.

The fact that the model is able to get a significantly higher accuracy than 25% (which would be the average for random guessing), on a data set it has never seen before shows that the model is not overfit to the training data, and is able to generalize the task well.