Project 3 – 440 Fall 2023

Professor – Dr. Cowan

Students – Kush Patel (kp1085), Pavitra Patel (php51), Huzaif Mansuri (htm23)

The First Task:

Q. How are you defining your input space?

Our input space represents a 20*20 pixel wiring diagram. Each cell or pixel can be one of four colors - Red, Blue, Yellow, Green. To feed this data into the machine learning model, we convert this representation in a numerical format. The numerical format that we have chosen is one-hot encoding where each color is represented as a unique binary vector.

'⬜' → [0, 0, 0, 0]

'🟥' → [0, 0, 0, 1]

'🟨' → [0, 0, 1, 0]

'🟦' → [0, 1, 0, 0]

'🟩' → [1, 0, 0, 0]

As indicated from above, each pixel or color represents 4 features. Our input space consists of 1601 linear features and 200 non-linear features.

Linear features:

Our image is represented as 20*20 pixel image which implies we have 400 pixels in total. Each pixel has 4 features so total features would be 400*4 = 1600. We will add one more feature which is x_0 that corresponds to bias weight w_0. So, the total number of linear features will be 1601.

Mathematical representation of linear features:

- Let P_{ij} be a pixel at row i and column j in the grid, where $1 \leq i, j \leq 20$
- The one-hot encoding for each pixel P_{ij} is a vector $V_{ij} = [v_1, v_2, v_3, v_4]$, where each V_{ij} corresponds to a hot-encoding of the color in Pixel P_{ij}
- The entire grid can be represented as a flattened vector F of length 1600, constructed by concatenating the one-hot encoded vectors of all pixels in row-major order: $F=[V_{(1,1)}, V_{(1,2)}, \dots, V_{(i,j)}, \dots, V_{(20,20)}]$
- Including the '1' for bias term w_0, the final linear feature vector $F' = [1, V_{(1,1)}, V_{(1,2)}, \dots, V_{(i,j)}, \dots, V_{(20,20)}]$, of length 1601.

Non-Linear features:

As we know each row has 20 pixels, we will now group these 20 pixels in groups of 4. So, we will have 5 groups in a row. As we have 20 rows, we will have 20*5 = 100 groups.

Now, we perform dot product of each vector contained within a group. We will get a scalar value i.e., 0 or 1. We count this as 1 feature. So, total number of features considering rows will be 100.

Similarly, we will get 100 features considering columns, hence totalling to 200 non-linear features.

Mathematical representation of non-linear features:

- Let G_{ij} be a group of 4 pixels starting from P_{ij} either horizontally or vertically, where i represents ith row, and j represents jth column
- The non-linear features for each group G_{ij}, is the dot product of its pixel vectors:

  $A_{ix} = V_{iy} * V_{i(y+1)} * \ldots * V_{i(y+3)}$ for horizontal groups, where $x = 0,1,2,3$; $y = 1 + 4*x$

  $B_{xj} = V_{yj} * V_{(y+1)j} * \ldots * V_{(y+3)j}$ for vertical groups, where $x = 0,1,2,3$; $y = 1 + 4*x$
- For 100 such groups horizontally and 100 vertically, we get 200 non-linear features $N = [A_{(1,0)}, A_{(1,1)}, \ldots, A_{(20,2)}, A_{(20,3)}, B_{(0,1)}, B_{(1,1)}, \ldots, B_{(2,20)}, B_{(3,20)}]$

So, total features (linear + non-linear) = 1600 + 1 + 100 + 100 = 1801.

Mathematical representation for linear and non-linear features combined:

- The final input vector $X = [F' + N]$, whose length is 1801

Intuition behind the idea of non-linear features:

- 🟥🟥🟨🟥 (This represents a possible 1 non-linear feature group as explained)

- 🟥 (This represents a possible 1 non-linear feature group as explained)
  🟥
  🟥
  🟩

Why will non-linear features help us?

- Linear features only provide information about a single pixel whereas non-linear features provide information about neighboring pixels.
- Non-linear features provide information about how wires are laid on the grid and where they are located.
  - For instance, if a group contains pixels RRRR, the dot product would be 1 which indicates group contains same cells are together.
  - For instance, if a group contains pixels RRYR, the dot product would be 0 which indicates group dosn't contain same color cells.
- If a certain group of pixel imply that it is safe or dangerous, the model would learn and perform better on the test data set.

Q. How are you defining your output space?

We want to know whether a given wiring image is safe or dangerous. So, this problem can be classified to have a binary output.

- Returns 0 when safe
- Returns 1 when not-safe/dangerous

Q. What model space are you considering, and what parameters does it have? Be sure to specify any design choices you make here.

In logistic regression, our model space and parameters are defined as follows:

Model Space:

Linear combination:

We need to compute a linear combination of the input features before applying the sigmoid function. For 1801 features, the linear combination is:

$w * X = w_0x_0 + w_1x_1 + w_2x_2 + \ldots + w_{1801}x_{1801}$, where $x_0 = 1$

Logistic/Sigmoid Function:

For logistic regression, we take f to be a non-linear function of a linear combination of the inputs. If X is our input vector and w is the weight vector, the logistic regression model predicts the probability P(Y = 1/X) as:

$$f_w(X) = \frac{1}{1 + e^{-(w \cdot X)}}$$

Parameters:

Weights (w): These are co-efficeints for each feature in the input space. In our case, we have 1801 weights, 1 for each feature, which the model learns during training to minimize prediction error. w_0 is the bias term and so x_0 is set to 1.

The range of weight is calculated as $1/(n)^{0.5} = 1/(1600)^{0.5} = 1/40 = 0.025$

The range is -0.025 to +0.025.

Initially, the weights are assigned at random to each feature in the input space. If the train data set contain Q images, then we shuffle those Q images, and pass them to train at once. After, each image is trained weights will be updated and that's when 1 epoch will be completed. The images are shuffled again and the same process repeats again.

Learning rate (α):

Alpha controls how much weights are adjusted during training. A small learning rate makes the training slower but precise. Our model learns best between alpha values 0.01 to 0.25 for total datasets 2000, 2500, 3000, and 5000.

Q. How are you measuring the loss or error of a given model?

We are using the binary cross-entropy or log loss as our loss function. The loss increases as the predicted probability diverges from the actual label (y).

For a given data point x, with a true label y (which can be 0 or 1) and a predicted probability $f_w(x)$ from the model (where $f_w(x)$ is the output of the logistic/sigmoid function given x), the Loss L is defined as:

$$Loss(f_w(x), y) = -y\ln(f_w(x)) - (1-y)\ln(1-f_w(x))$$

Q. What training algorithm are you using to find the best model you can? Include any necessary math to specify your algorithm.

We use the Stochastic Gradient Descent (SGD) training algorithm.

Given the loss in the step above, we calculate the gradient of the loss function with respect to the weight vector w.

First, applying derivative to sigmoid function:

$$\frac{\partial}{\partial w} f_w(x) = f_w(x)(1-f_w(x))x$$

Now, calculating gradient of the loss function:

$$\nabla_w Loss = \frac{\partial}{\partial w}\left(-y\ln(f_w(x)) - (1-y)\ln(1-f_w(x))\right)$$

$$= -y\frac{1}{f_w(x)}f_w(x)(1-f_w(x))x + (1-y)\frac{1}{1-f_w(x)}f_w(x)(1-f_w(x))x$$

$$= -y(1-f_w(x))x + (1-y)f_w(x)x$$

$$= (f_w(x)-y)x$$

Now, the update rule for SGD on the weight vector w is:

$$w := w - \alpha\nabla_w Loss$$

Substituting, the gradient ∇_wLoss:

$$w := w - \alpha(f_w(x) - y)x$$

This is the update rule for the weights using GSD for logistic regression. It means that for each training instance, we calculate the prediction $f_w(x)$, determine the error $f_w(x)-y$, scale this error by the input vector x, and adjust the weights in the opposite direction of this scaled error, scaled by the learning rate alpha.

Algorithm:

1. We first choose a positive learning rate that determines the size of the steps taken during optimization.
2. Initialize weights with a random value within range of -0.025 to 0.025.
3. Given a dataset we first shuffle it and then for each data point (x, y):
    a. Compute the prediction using the current weights: $y' = f_w(x)$.
    b. Calculate the loss based on the prediction and the value of y.
    c. Calculate SGD or new weight using the current weight, learning rate, and gradient of the loss function.
4. Repeat step 3 until loss stops decreasing. We monitor a certain number of loss values depending on the data set and then decide whether improvement has stopped or not.

Q. How are you preventing overfitting?

Two steps we implemented to prevent overfitting:

1. **Shuffling data set**: Randomly shuffle the dataset before each epoch. This is an important step in SGD to prevent any order effects that might influence the gradient updates and helps to ensure that the algorithm does not get biased by the order of the data.
2. **Early stopping**: We use a "patience" variable, which defines how many epochs to wait before loss has stopped decreasing. If the model's performance ceases to improve or starts to worsen, this is a clear indication that the model has begun to overfit the training data.

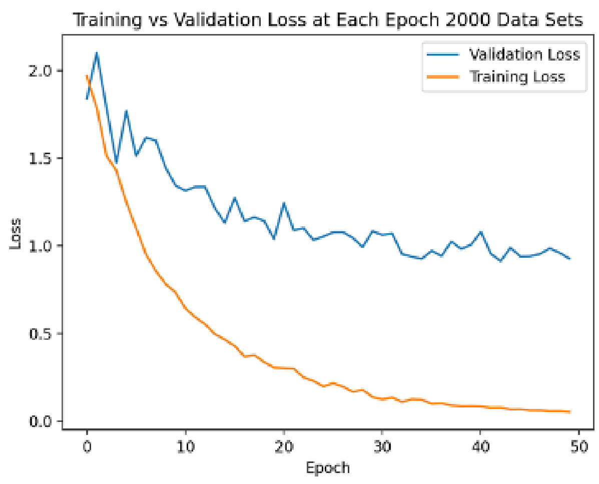
Q. Performance of your model on a training set of 2k examples, 2.5k examples, 3k examples, and 5k examples.

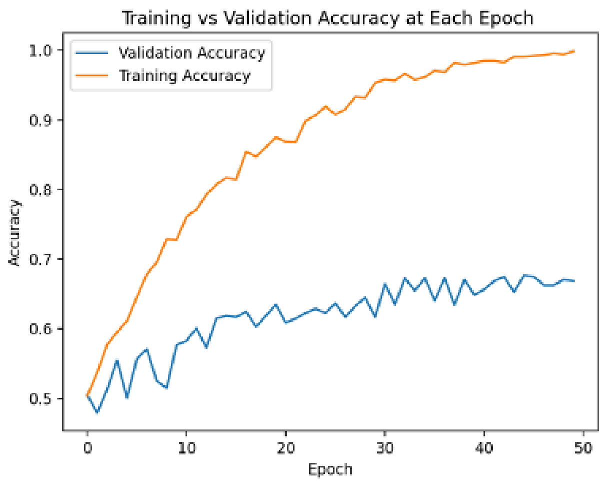
Q. A graph of your model's loss over time, demonstrating learning.

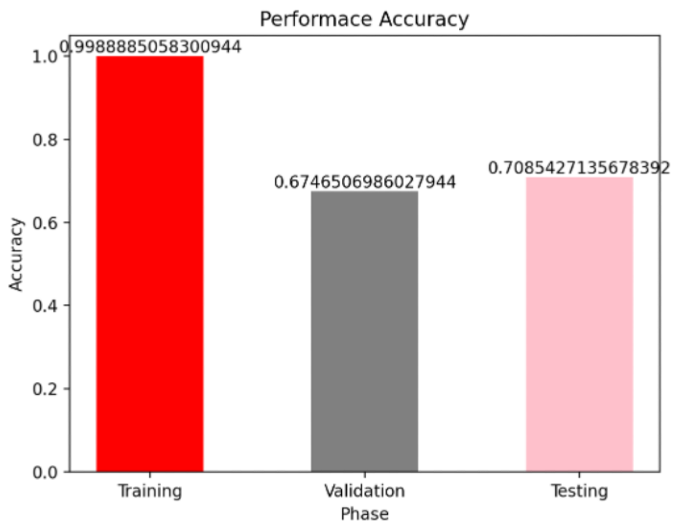
Q. Assessment of your trained model, and evidence that it is not overfit to the data, and instead generalizes well.

(All three questions answered together)

Image Dataset of 2000, the learning rate of 0.1, and Data split of 90/10:



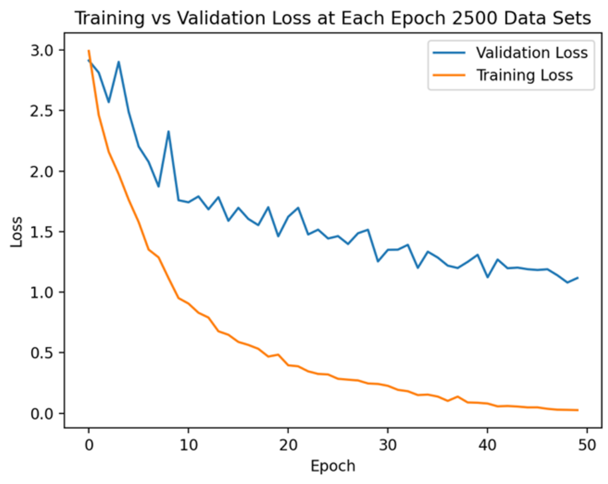


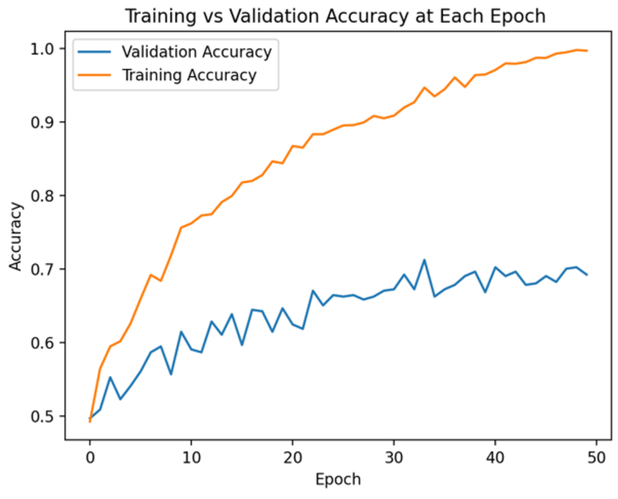


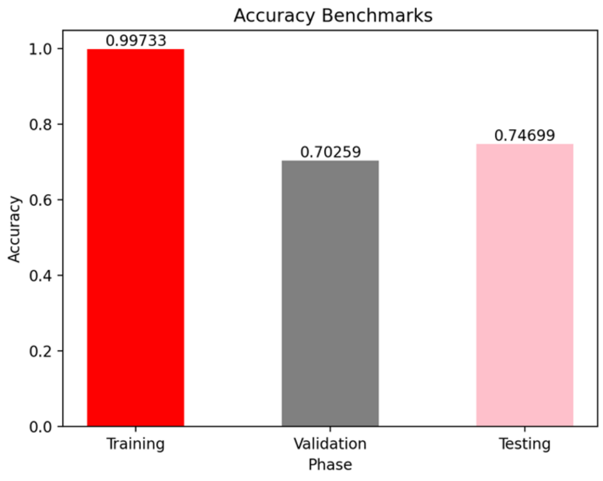
Graph Analysis:

- The first graph displays training vs validation loss at each epoch for a total data set of 2000 and learning rate of 0.1. The orange line represents the amount of loss during the training phase as a result of prediction as per prior weights. The blue line represents the amount of loss during the validation phase as a result of updated/new weights. We can observe that loss decreased over time and the model kept leaning until 50 epochs.
- The second graph displays training vs validation accuracy at each epoch. The orange line represents how accurately the model predicts the true label y in the training phase as per the prior weights. The blue line represents how accurately the model predicts the true label y in the validation phase as a result of updated/new weights.
- The third graph displays the performance accuracy of each phase at the last epoch. For this scenario, our training accuracy is 99.8%, validation accuracy is 67.4%, and testing accuracy is 70.8%. Accuracies for validation and testing are almost similar and this implies our model generalizes well without overfitting.
- For validation loss we observe that we don't have high fluctuating spikes and so this is one more evidence that confirms that our model doesn't overfit.

Image Dataset of 2500, learning rate of 0.13, and Data Split of 90/10:



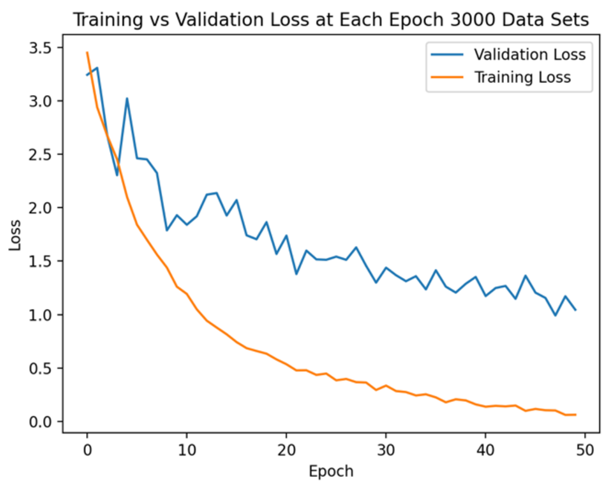


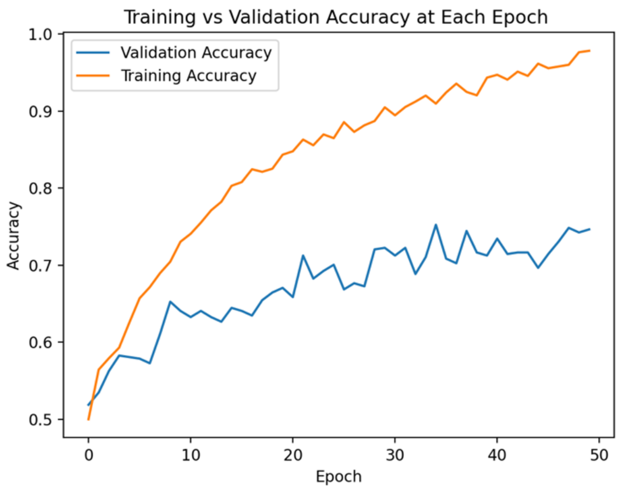


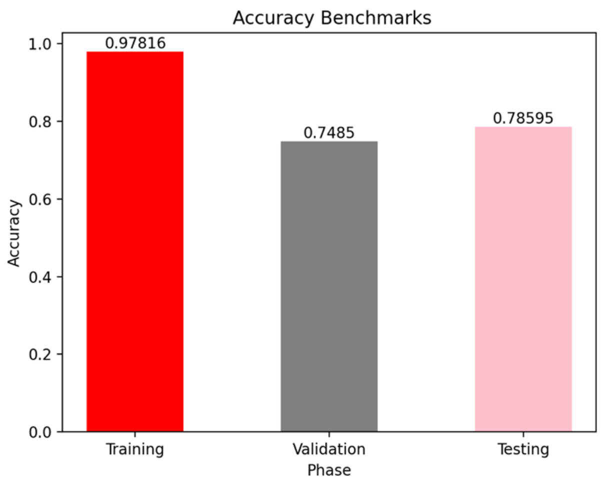
Graph Analysis:

- The first graph displays training vs validation loss at each epoch for a total data set of 2500 and learning rate of 0.13. The orange line represents the amount of loss during the training phase as a result of prediction as per prior weights. The blue line represents the amount of loss during the validation phase as a result of updated/new weights. We can observe that loss decreased over time and the model kept leaning until 50 epochs.
- The second graph displays training vs validation accuracy at each epoch. The orange line represents how accurately the model predicts the true label y in the training phase as per the prior weights. The blue line represents how accurately the model predicts the true label y in the validation phase as a result of updated/new weights.
- The third graph displays the performance accuracy of each phase at the last epoch. For this scenario, our training accuracy is 99.7%, validation accuracy is 70.2%, and testing accuracy is 74.6%. Accuracies for validation and testing are almost similar and this implies our model generalizes well without overfitting.
- For validation loss we observe that we don’t have high fluctuating spikes and so this is one more evidence that confirms that our model doesn’t overfit.

Image Dataset of 3000, learning rate of 0.18, and Data Split of 80/20:



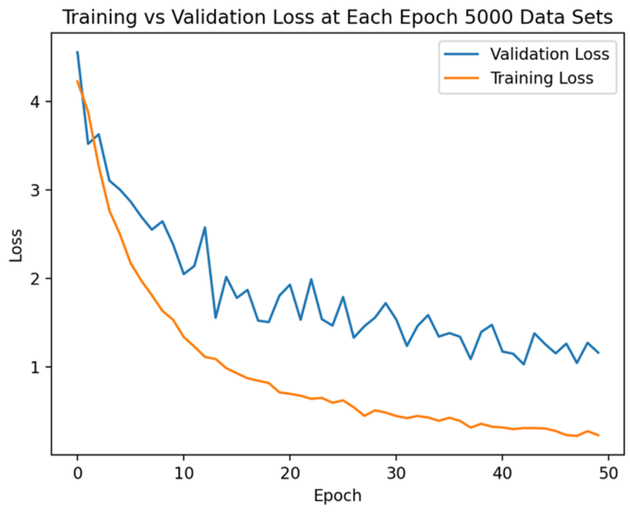


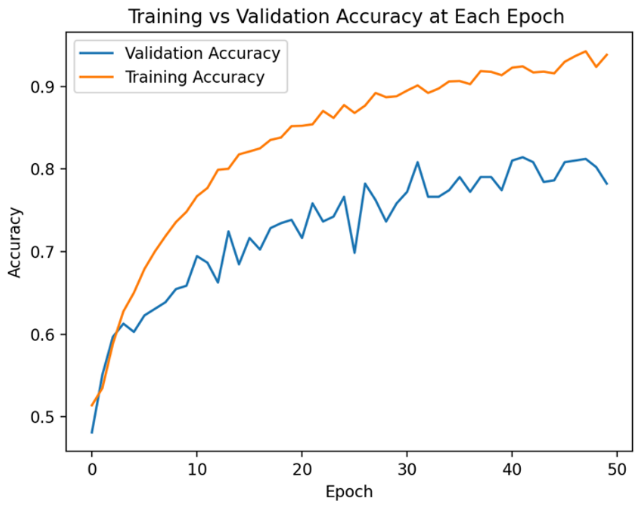


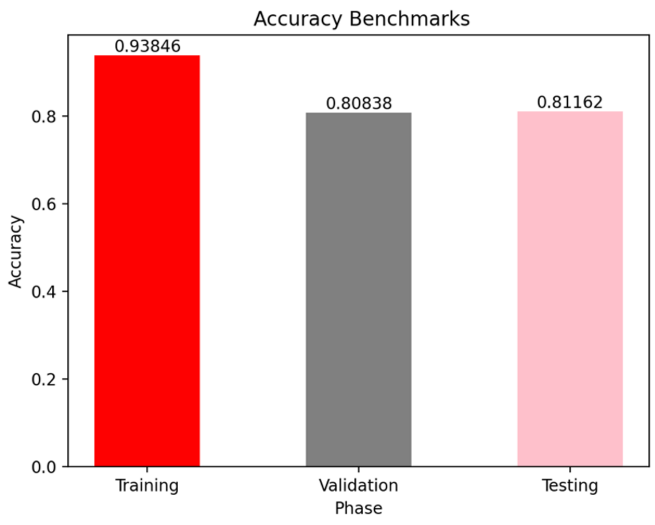
Graph Analysis:

- The first graph displays training vs validation loss at each epoch for a total data set of 3000 and learning rate of 0.18. The orange line represents the amount of loss during the training phase as a result of prediction as per prior weights. The blue line represents the amount of loss during the validation phase as a result of updated/new weights. We can observe that loss decreased over time and the model kept leaning until 50 epochs.
- The second graph displays training vs validation accuracy at each epoch. The orange line represents how accurately the model predicts the true label y in the training phase as per the prior weights. The blue line represents how accurately the model predicts the true label y in the validation phase as a result of updated/new weights.
- The third graph displays the performance accuracy of each phase at the last epoch. For this scenario, our training accuracy is 97.8%, validation accuracy is 74.8%, and testing accuracy is 78.5%. Accuracies for validation and testing are almost similar and this implies our model generalizes well without overfitting.
- For validation loss we observe that we don’t have high fluctuating spikes and so this is one more evidence that confirms that our model doesn’t overfit.

Image Dataset of 5000, learning rate of 0.23, and Data split of 80/20:







Graph Analysis:

- The first graph displays training vs validation loss at each epoch for a total data set of 5000 and learning rate of 0.23. The orange line represents the amount of loss during the training phase as a result of prediction as per prior weights. The blue line represents the amount of loss during the validation phase as a result of updated/new weights. We can observe that loss decreased over time and the model kept leaning until 50 epochs.
- The second graph displays training vs validation accuracy at each epoch. The orange line represents how accurately the model predicts the true label y in the training phase as per the prior weights. The blue line represents how accurately the model predicts the true label y in the validation phase as a result of updated/new weights.
- The third graph displays the performance accuracy of each phase at the last epoch. For this scenario, our training accuracy is 93.8%, validation accuracy is 80.8%, and testing accuracy is 81.1%. Accuracies for validation and testing are almost similar and this implies our model generalizes well without overfitting.
- For validation loss we observe that we don't have high fluctuating spikes and so this is one more evidence that confirms that our model doesn't overfit.

The Second Task:

Q. How are you defining your input space?

Similar to task-1, Our input space represents a 20*20 pixel wiring diagram where each cell or pixel can be one of four colors - Red, Blue, Yellow, Green. To feed this data into the machine learning model, we take the rows and columns of the grid with colors other than white, and convert its representation into numerical format. We also concatenate one extra bit at the start of each row/column vector to indicate if it is a row or a column by setting the bit value = 0 (if row) or 1 (if column). The numerical format that we have chosen to represent each pixel of color is one-hot encoding, where each color is represented as a unique binary vector.

'⬜' → [0, 0, 0, 0]

'🟥' → [0, 0, 0, 1]

'🟨' → [0, 0, 1, 0]

'🟦' → [0, 1, 0, 0]

'🟩' → [1, 0, 0, 0]

As indicated from above, each pixel or color represents 4 features. Hence, The feature size of one row/column vector equals **1 (row/column binary bit) + (4 features per color * 20 pixels per row/column) = 81 features per row/column vector.** Since, we have 4 colors other than white in our grid, hence we take the numerical format representation for each of these colors' row/column. Thus, Our input space consists of **1 (bias term x_0) + 4*81 = 325 linear features** and **486 non-linear features (detailed below)**.

Linear features:

Our image is represented as 20*20 pixel image, where we have 2 rows, and 2 columns of color other than white, which implies we have (4*20) - 4 = 76 non-white color pixels in total, so the feature size of one row/column vector becomes 81. Each pixel has 4 features so total features for all 4 color strands would be 4*80 = 320, plus we add one more feature bit to represent row/column per color strand, and we also add one bias term feature corresponding to bias weight w_0. So, the total number of linear features will be 4*80 + 4*1 + 1 = 325.

Mathematical representation of linear features:

- Let P_{r1}, P_{r2} be two rows of pixels with color other than white. Similarly, let P_{c1}, P_{c2} be two columns of pixels with color other than white.
- The one-hot encoding for each pixel P_{ij} in the grid is a vector $V_{ij} = [v_1, v_2, v_3, v_4]$, where $1 \leq i, j \leq 20$
- We represent the image as a flattened vector F of length 324, constructed by concatenating the one-hot encoded vectors of all pixels in rows P_{r1}, P_{r2}, and columns P_{c1}, P_{c2} in order, along with the row/column bit (RC_{bit}) => 0 for row, 1 for column:

$$F = [0, V_{(r1, 1)}, V_{(r1, 2)}, \ldots, 0, V_{(r2, 1)}, V_{(r2, 2)}, \ldots, 1, V_{(1, c1)}, V_{(2, c1)}, \ldots, 1, V_{(1, c2)}, V_{(2, c2)}, \ldots, V_{(20, c2)}]$$

- Including the '1' for bias term w_0, the final linear feature vector would be:

$F' = [1, 0, V_{(r1, 1)}, V_{(r1, 2)}, \ldots, 0, V_{(r2, 1)}, V_{(r2, 2)}, \ldots, 1, V_{(1, c1)}, V_{(2, c1)}, \ldots, 1, V_{(1, c2)}, V_{(2, c2)}, \ldots, V_{(20, c2)}]$, with length 325.

Non-Linear features:

As we know we have a total of 4 row+column vectors from our linear features, each with feature size 81. We would now like to generate some non-linear features with the idea of providing the intuition of which colors intersect, and where they intersect to our model with the help of these features. Since, we also know that each of our row or column vectors represent a color strand of some color other than white, we can do some non-linear operation on these vectors in order to generate the new non-linear features of our interest.

And so, we perform the XOR ($\oplus$) operation on each pair of row+column vectors that we generated above. Specifically, this XOR ($\oplus$) operation will perform a bitwise comparison operation on the input vector pair, such that it returns 0 if both bits are same, and returns 1 otherwise.

Since we have 4 row+column vectors, we get C(4, 2) = 6 pairs. Also, as the XOR ($\oplus$) is a bitwise operation, it would output the feature vector of size equal to the row/column vector. Hence, each feature vector outputted after the XOR ($\oplus$) is of size 81. Finally, as we have 6 pairs, we would get total of 6*81 = 486 non-linear features.

Mathematical representation of non-linear features:

- Let G_{r1}, G_{r2} be two row vectors of rows (r1, r2) with color other than white. Similarly, let G_{c1}, G_{c2} be two column vectors of columns (c1, c2) with color other than white.
- Let (G_x, G_y) be some pair from the 4 row/column vectors, where $(x, y) \in \{(r1, r2), (r1, c1), (r1, c2), (r2, c1), (r2, c2), (c1, c2)\}$
- The non-linear features for each pair (G_x, G_y) is the XOR ($\oplus$) operation on vectors G_x and G_y:

  $A_{(x, y)} = G_x \oplus G_y$

- For 6 such pairs, we get 486 non-linear features:

  $N = [A_{(r1, r2)}, A_{(r1, c1)}, A_{(r1, c2)}, A_{(r2, c1)}, A_{(r2, c2)}, A_{(c1, c2)}]$

So, total features (linear + non-linear) = 324+ 1 + 486 = 811.

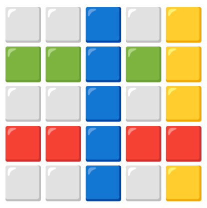
Mathematical representation for linear and non-linear features combined:

- The final input vector X = [F' + N], whose length is 811

Intuition behind the idea of non-linear features:

Since It might be tricky to understand the intuition behind the choice of XOR ($\oplus$) function over other non-linear functions. Let's try to understand it using an example:

For the sake of simplicity, suppose we have a 5*5 grid as represented below:



- We would now like to first generate our linear features. To do so, we iterate through the rows, and columns of colors other than white, and use the below hot-encoding representation mapping to get the correct numerical representation of these rows, and columns.

'⬜' → [0, 0, 0, 0]

'🟥' → [0, 0, 0, 1]

'🟨' → [0, 0, 1, 0]

'🟦' → [0, 1, 0, 0]

'🟩' → [1, 0, 0, 0]

- Hence, for Linear Features, we would get:
    - B = [1 0100 0100 0100 0100 0100]
    - Y = [1 0010 0010 0010 0001 0010]
    - G = [0 1000 1000 0100 1000 0010]
    - R = [0 0001 0001 0100 0001 0001]
- To Compute our non-linear features, we take XOR (⊕) on each of the pair possible from the above 4 vectors, doing so, we get:
    - BY = [**0** 0110 0110 0110 0101 0110]
    - BG = [1 1100 1100 **0000** 1100 0110]
    - BR = [1 0101 0101 **0000** 0101 0101]
    - YG = [1 1010 1010 0110 1001 **0000**]
    - YR = [1 0011 0011 0110 **0000** 0011]
    - GR = [**0** 1001 1001 **0000** 1001 0011]
- Now that we have both our linear and non-linear features ready, we can dive into understanding the details that could be potentially observed from these features:
    - Using the linear features, we get to know the number of times a color occurred in a grid, and if its strand was in form of a row or a column. However, we can't clearly understand the intersection scenario here.
    - Looking at the non-linear features, we get to know two things: a) If the RC_{bit} (the first bit) is set to 1, and we have a feature of type 0000, we can get the intuition about the intersecting colors, and its position. b) If the RC_{bit} (the first bit) is set to 0, and we have a feature of type 0000, we can get the intuition about the last color placed on the grid.

Hence, the choice of XOR (⊕) provided our model with strong non-linear features to derive the intuition about the grid.

Why will non-linear features help us?

- Linear features only provide information about the total count of pixels for each color in the grid, whereas non-linear features provide information about intersections with intuition about who intersects, and where does the intersection occur.
- Mathematically, Non-linear features would generate a pattern which could allow the model to learn the location of intersection:
    - For instance, if we look at the non-linear feature generated using a row vector and a column vector, we would get the first bit (RC_{bit}) set to 1 in the XOR (⊕) output vector, since the values at the RC_{bit} in our row vector, and column vector would be different. Also, if there exists a color: say R [0, 0, 0, 1] in both the vectors, we would get a sequence of 4 zeroes [0, 0, 0, 0] in the XOR (⊕) output vector at the position of intersection, which indicates that these row, and column share a common color pixel at that particular position.
    - Now, suppose we look at the non-linear feature generated using two row vectors or two column vectors, we would get the first bit (RC_{bit}) set to 0 in the XOR (⊕) output vector, since the values at the RC_{bit} in our input vectors would be the same. Also, in this case, if we encounter 4 consecutive zeroes feature [0, 0, 0, 0], then that's only possible due to the case that these two rows or columns is being overlapped by the last color placed on grid. Hence, this can give our model the clear intuition to spot the last color placed based on comparison of this intersection position in other vectors.
- In conclusion, if the model can get the intuition of who intersects, and where the intersections take place, the model would be able to learn how to determine the third wire based on these features, and so would perform better on the test data set.

Q. How are you defining your output space?

We know the order of wire placed on the grid, so we take the third wire and give it to the output space.

To represent the output space into numerical form, we used one hot encoding.

'🟥' → [0, 0, 0, 1]

'🟨' → [0, 0, 1, 0]

'🟦' → [0, 1, 0, 0]

'🟩' → [1, 0, 0, 0]

Based on the above encoding, if the third wire is red colored then, output space is [0,0,0,1].

Q. What model space are you considering, and what parameters does it have? Be sure to specify any design choices you make here.

Model Space:

SoftMax Activation Function:

For softmax regression, we take the Z which is the dot product of each class (red, green, yellow, blue) wires, its respective weights and X vector. Since, this predictions are not between 0 and 1, so to get the probability between 0 and 1, normalize it by the summing all class probabilities.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

Parameters:

Weights (w):

These are coefficients for each feature in the input space. In our case, we have 4 classes, so initializing weights for each class (W_R, W_G , W_Y , W_B). Moreover, we have 811 weights, 1 for each feature, which the model learns during training to minimize prediction error. w_0 is the bias term and so x_0 is set to 1.

The range of weight is calculated as $1/(n)^{0.5} = 1/(811)^{0.5} = 0.035$

The range is -0.035 to +0.035. [for each class weight]

Initially, each class weights are assigned at random to each feature in the input space. If the train data set contains Q images, then we shuffle those Q images, and pass them to train at once. After each image is trained, all weights for each class will be updated and that's when 1 epoch will be completed. The images are shuffled again and the same process repeats again.

Learning rate (α):

Alpha controls how much weights are adjusted during training. A small learning rate makes the training slower but precise. Our model learns best between alpha values 0.01 to 0.25 for total datasets 2000, 2500, 3000, and 5000.

Q. How are you measuring the loss or error of a given model?

We know that our Softmax Activation Function calculated the prediction for each class (color wires):

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{C} e^{z_j}}$$

Finding the loss using given data point, activation function, and output space:

$\text{Max}\,(W_R, W_G , W_Y , W_B) = P(\text{data})$: represents the probability of each wire

$P(\text{data}) = F_{y1}(X_1) \,.\, F_{y2}(X_2) \,.\, \ldots\, F_{yn}(X_n)$: dot product of output space and prediction

$$\text{Max}_k = \prod_{i=1}^{n} F_{yi}(X_i)$$

$$\text{Max}_{(Wk)} = \prod_{i=1}^{n} F_{yi}(X_i)$$, simplifying the representation $F(X_i)$ prediction, and y_i output space

$$\text{Min}_{(Wk)} = \sum_{i=i}^{n} - y_i \ln (F(X_i))$$ **- Loss Cross Entropy**

To calculate the loss based on the prediction $F(X_i)$ and actual output (Y_i), we can observe that where $y_i = 1$ for the correct classification and $y_i = 0$ for all others.

Q. What training algorithm are you using to find the best model you can? Include any necessary math to specify your algorithm.

We use the stochastic Gradient Descent (SGD) training algorithm

Given the loss in the step above, we calculate the gradient of the loss function with respect to the weight vector w

First, applying Partial derivative with respect to a weight:

- We have output layer W, and input layer X. Consider a W_{ij} which connects to (j) in the input neuron and i-th is the output neuron.

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial z_i} \cdot \frac{\partial z_i}{\partial w_{ij}} \qquad - (1)$$

From the Softmax function, we know that

$$\hat{y}_i = \frac{e^{z_i}}{\sum_{k=1}^{C} e^{z_k}}$$

Therefore, from the derivation of loss by derivation of Z:

$$\frac{\partial L}{\partial z_i} = \hat{y}_i - y_i \qquad - (2)$$

So, now we know that the derivation Z with respect to weights is:

$$z_i = w_{ij} x_j$$

$$\frac{\partial z_i}{\partial w_{ij}} = x_j \qquad - (3)$$

We now plug the value of equation (3) and (2) into equation (1). We obtain final equation, derivation of loss with respect to weights.

$$\frac{\partial L}{\partial w_{ij}} = (\hat{y}_i - y_i) \cdot x_j \qquad - (4)$$

Updating the parameters of equation of 1 using the equation (4)

$$w_{ij}^{\text{new}} = w_{ij}^{\text{old}} - \eta \cdot (\hat{y}_i - y_i) \cdot x_j$$

In our derivation, we have considered the bias term as the first term of Z as bias, where W_oX_0 and $X_0 = 0$

This is the update rule for the weights using SGD for softmax regression. It mens for each training instance, we calculate the prediction for each class and determine the error on the prediction with f(X_i)= Y_i and scale this error by the input vector X, and adjust the weights in the opposite direction of this scaled error, scaled by the learning rate (alpha).

Algorithm:

1. We first choose a positive learning rate that determines the size of the steps taken during optimization.
2. Initializing weights for each class:
   a. Initializing the weights for red color wire with a random value within range -0.035 to 0.035
   b. Initializing the weights for Green color wire with a random value within range -0.035 to 0.035
   c. Initializing the weights for yellow color wire with a random value within range -0.035 to 0.035
   d. Initializing the weights for Blue color wire with a random value within range -0.035 to 0.035
3. At every epoch, we update: Given a dataset we first shuffle it and then for each data point (X,Y):
   a. Compute the prediction using current weights for each class: y' = f_w(x).
   b. Calculate the the loss for each class based on the prediction f(X_i) and the value of Y_i
   c. Using SGD, calculating the new weights for each class using the current weights, learning rate, and gradient of the loss function.
4. Repeat step 3 until loss stops decreasing, we monitor a certain number of loss values depending on the data set and then decide whether improvement has stopped or not.

Q. How are you preventing overfitting?

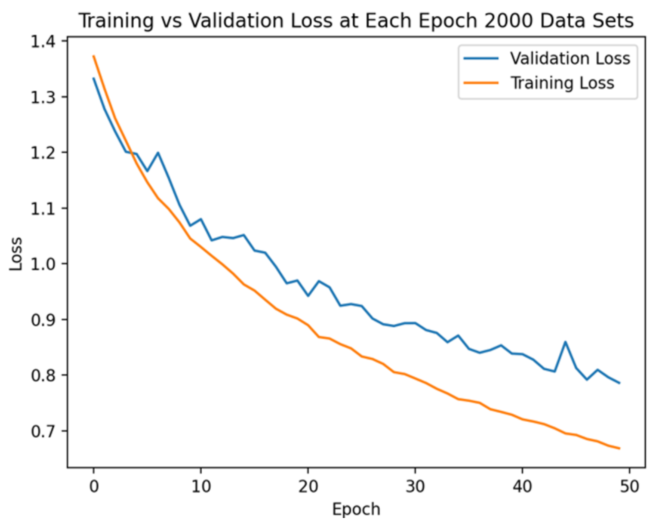
Two steps we implemented to prevent overfitting:

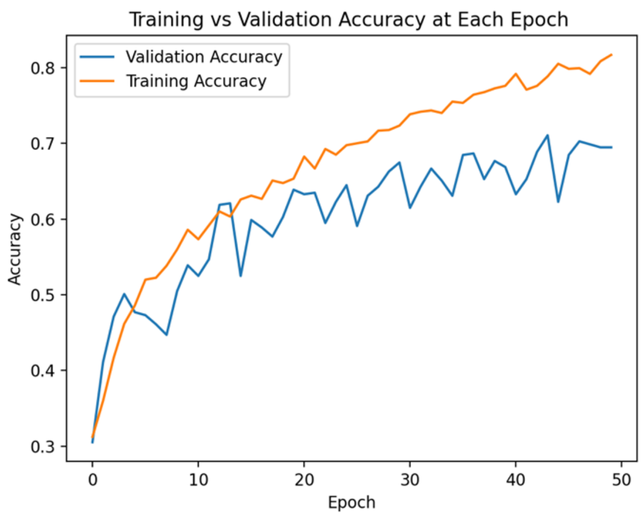
1. **Shuffling data set**: Randomly shuffle the dataset before each epoch. This is an important step in SGD to prevent any order effects that might influence the gradient updates and helps to ensure that the algorithm does not get biased by the order of the data.
2. **Early stopping**: We use a "patience" variable, which defines how many epochs to wait before loss has stopped decreasing. If the model's performance ceases to improve or starts to worsen, this is a clear indication that the model has begun to overfit the training data.

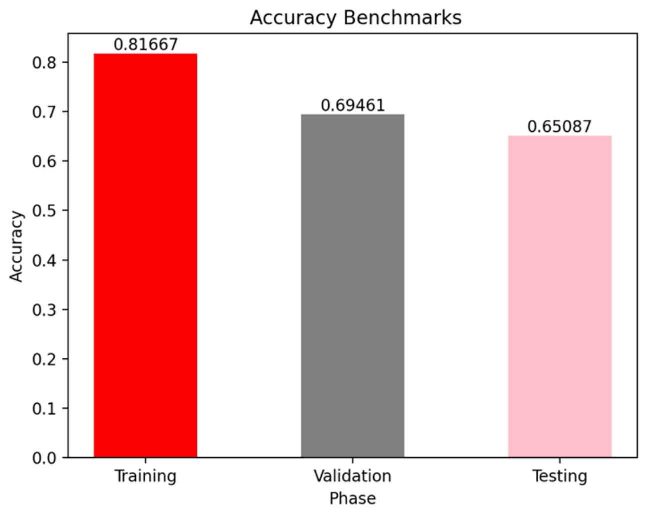
Q. Performance of your model on a training set of 500 examples, 1k examples, 2.5k examples, and 5k examples. **Q.** A graph of your model's loss over time, demonstrating learning. **Q.** Assessment of your trained model, and evidence that it is not overfit to the data, and instead generalizes well.

(All three questions answered together)

Image Dataset of 2000, learning rate of 0.0005, and Data split of 80/20:



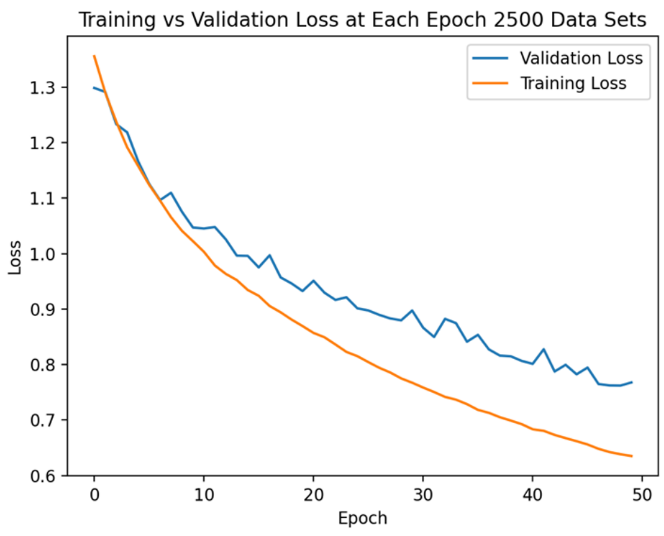


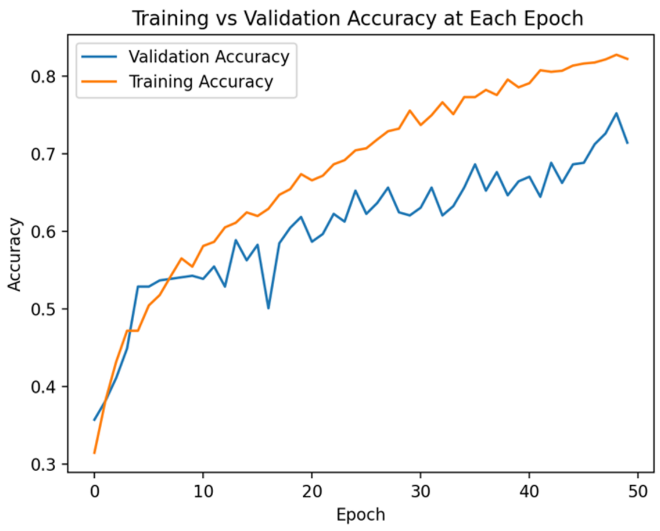


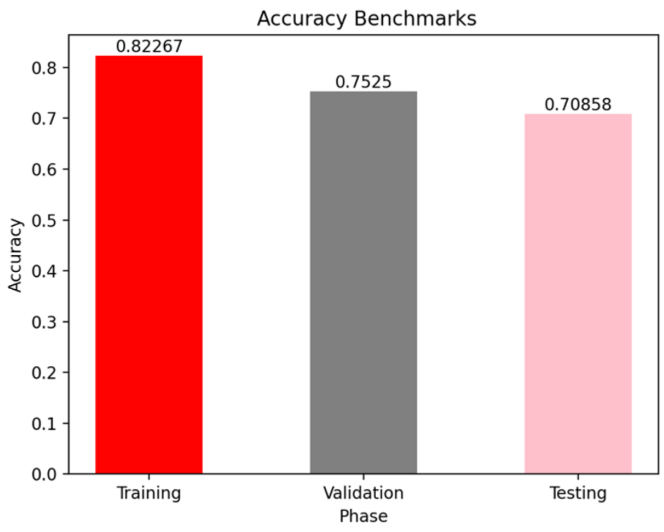
Graph Analysis:

- The first graph displays training vs validation loss at each epoch for a total data set of 2000 and learning rate of 0.0005. The orange line represents the amount of loss during the training phase as a result of prediction as per prior weights. The blue line represents the amount of loss during the validation phase as a result of updated/new weights. We can observe that loss decreased over time and the model kept leaning until 50 epochs.
- The second graph displays training vs validation accuracy at each epoch. The orange line represents how accurately the model predicts the true label y in the training phase as per the prior weights. The blue line represents how accurately the model predicts the true label y in the validation phase as a result of updated/new weights.
- The third graph displays the performance accuracy of each phase at the last epoch. For this scenario, our training accuracy is 81.6%, validation accuracy is 69.4%, and testing accuracy is 65.08%. Accuracies for validation and testing are almost similar and this implies our model generalizes well without overfitting.
- For validation loss we observe that we don't have high fluctuating spikes and so this is one more evidence that confirms that our model doesn't overfit.

Image Dataset of 2500, learning rate of 0.0005, and Data split of 80/20:



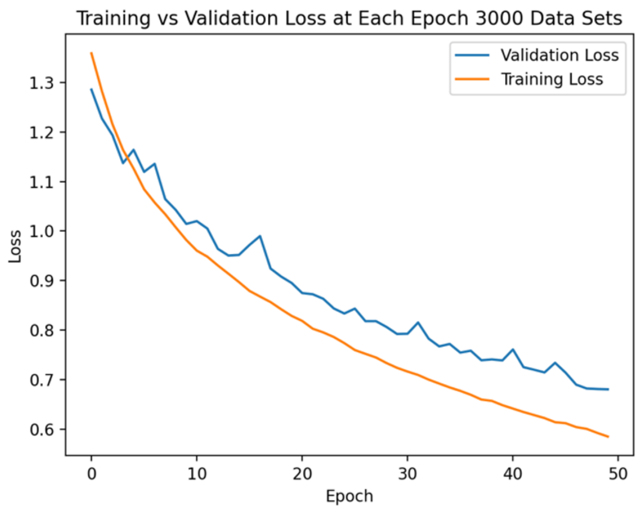


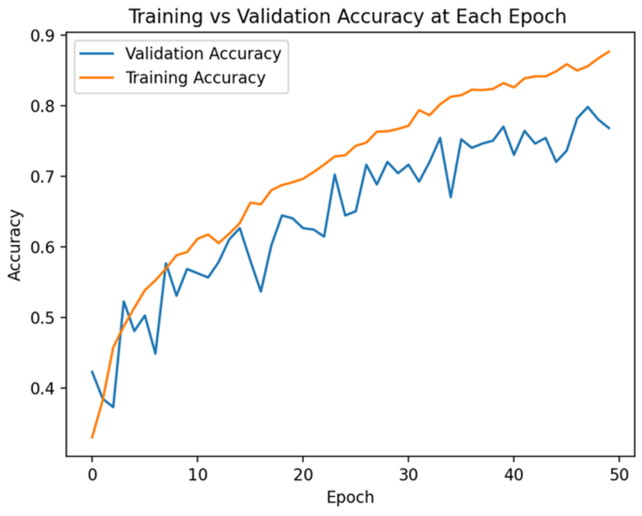


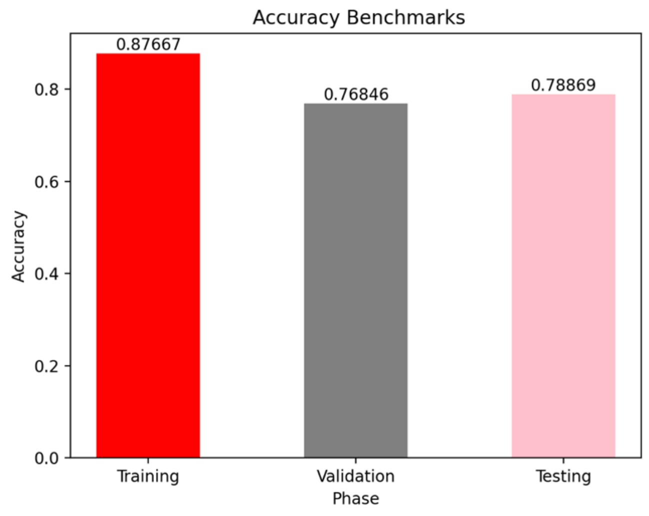
Graph Analysis:

- The first graph displays training vs validation loss at each epoch for a total data set of 2500 and learning rate of 0.0005. The orange line represents the amount of loss during the training phase as a result of prediction as per prior weights. The blue line represents the amount of loss during the validation phase as a result of updated/new weights. We can observe that loss decreased over time and the model kept leaning until 50 epochs.
- The second graph displays training vs validation accuracy at each epoch. The orange line represents how accurately the model predicts the true label y in the training phase as per the prior weights. The blue line represents how accurately the model predicts the true label y in the validation phase as a result of updated/new weights.
- The third graph displays the performance accuracy of each phase at the last epoch. For this scenario, our training accuracy is 82.2%, validation accuracy is 75.2%, and testing accuracy is 70.8%. Accuracies for validation and testing are almost similar and this implies our model generalizes well without overfitting.
- For validation loss we observe that we don’t have high fluctuating spikes and so this is one more evidence that confirms that our model doesn’t overfit.

Image Dataset of 3000, learning rate of 0.0005, and Data split of 80/20:



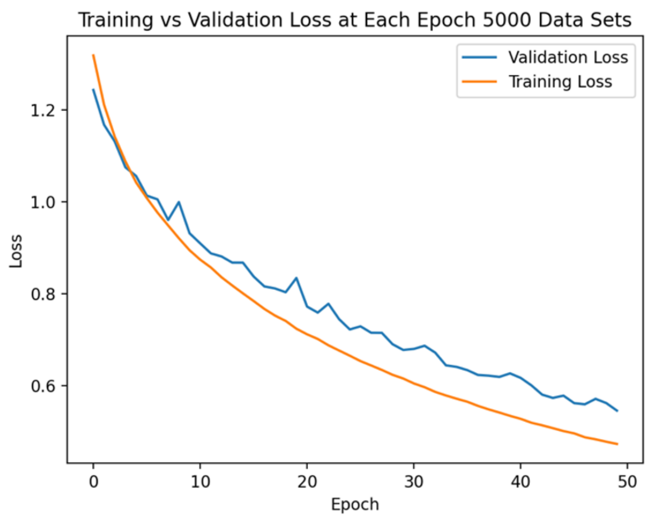


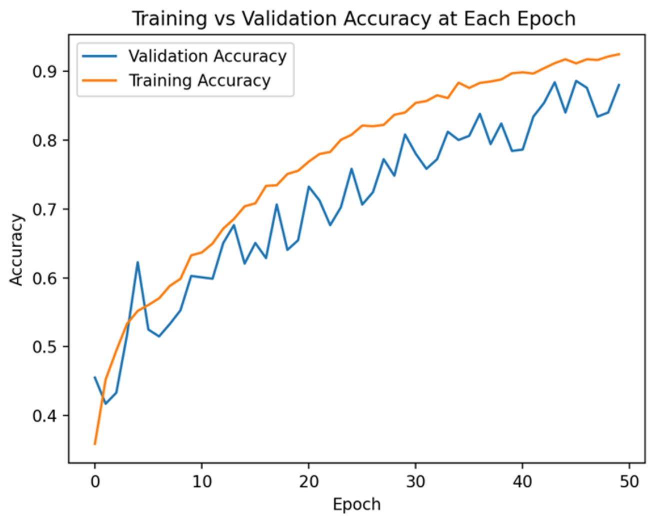


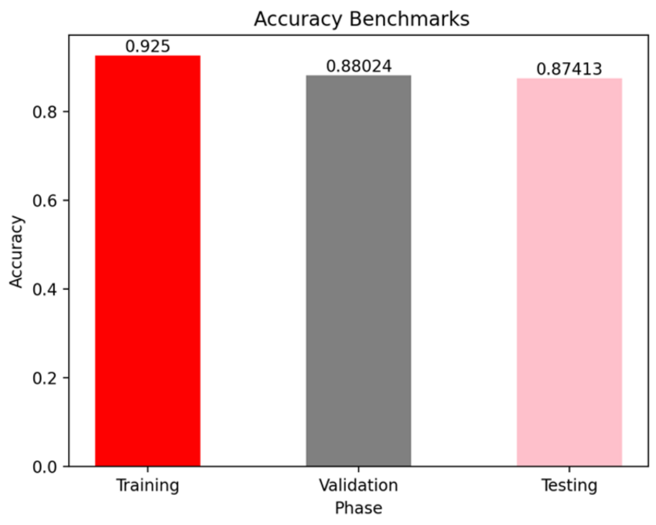
Graph Analysis:

- The first graph displays training vs validation loss at each epoch for a total data set of 3000 and learning rate of 0.0005. The orange line represents the amount of loss during the training phase as a result of prediction as per prior weights. The blue line represents the amount of loss during the validation phase as a result of updated/new weights. We can observe that loss decreased over time and the model kept leaning until 50 epochs.
- The second graph displays training vs validation accuracy at each epoch. The orange line represents how accurately the model predicts the true label y in the training phase as per the prior weights. The blue line represents how accurately the model predicts the true label y in the validation phase as a result of updated/new weights.
- The third graph displays the performance accuracy of each phase at the last epoch. For this scenario, our training accuracy is 87.6%, validation accuracy is 76.8%, and testing accuracy is 78.8%. Accuracies for validation and testing are almost similar and this implies our model generalizes well without overfitting.
- For validation loss we observe that we don't have high fluctuating spikes and so this is one more evidence that confirms that our model doesn't overfit.

Image Dataset of 5000, learning rate of 0.0005, and Data split of 80/20:







Graph Analysis:

- The first graph displays training vs validation loss at each epoch for a total data set of 5000 and learning rate of 0.0005. The orange line represents the amount of loss during the training phase as a result of prediction as per prior weights. The blue line represents the amount of loss during the validation phase as a result of updated/new weights. We can observe that loss decreased over time and the model kept leaning until 50 epochs.
- The second graph displays training vs validation accuracy at each epoch. The orange line represents how accurately the model predicts the true label y in the training phase as per the prior weights. The blue line represents how accurately the model predicts the true label y in the validation phase as a result of updated/new weights.
- The third graph displays the performance accuracy of each phase at the last epoch. For this scenario, our training accuracy is 92.5%, validation accuracy is 88.02%, and testing accuracy is 87.4%. Accuracies for validation and testing are almost similar and this implies our model generalizes well without overfitting.
- For validation loss we observe that we don't have high fluctuating spikes and so this is one more evidence that confirms that our model doesn't overfit.

Libraries Imported:

import random

import numpy as np

import pandas as pd

import ast

import matplotlib.pyplot as plt

import math

import openpyxl

Group contributions:

Kush Patel:

- Model 1 implemented
- Wrote report for Model 2

Pavitra Patel:

- Image and graph display and generating data set
- Presented group members with several ideas of input spaces for both models (Provided them with what linear + non linear features can be provided as input space to the model)
- Helped implement Model 2
- Wrote report for Model 1

Huzaif Mansuri:

- Model 2 implemented
- Wrote report for Model 2