

## PART Q1

**What was the main problem/task/tool implemented? Give brief theoretical information about it (using visuals and formulas/equations is encouraged).**

The main problem was the implementation of Support Vector Machine (SVM) and Softmax Classifiers.

SVM classifier is a discriminative classifier that works by finding a hyperplane in the input space that separates the data points into two classes. The hyperplane is chosen such that the margin between the two classes is maximized. This is achieved by finding the support vectors, which are the data points that lie closest to the hyperplane. SVM classifiers are known for their good performance on high-dimensional data and their ability to handle complex decision boundaries. However, they can be computationally expensive to train, especially for large datasets.

Softmax classifier is a generative classifier that learns a probability distribution over the classes for each input data point. The probability distribution is computed using the softmax function, which is a nonlinear function that normalizes the output of a linear model. Softmax classifiers are known for their good performance on multi-class classification problems. They are also relatively efficient to train, even for large datasets.

### 1. SVM Classifier

Firstly, the naïve implementation (with loops) was executed. Loss part of the function was already given in the notebook. The first part of the implementation was storing loss function gradient as  $dW$ . The derivative was computed at the same time with the loss function.

```
dW[:, j] += X[i]
dW[:, y[i]] -= X[i]
```

Gradient of the loss function( $dW$ ) was computed. Loss was computed as close to 9.

```
dW = dW/num_train + 2 * reg * W
```

After naïve implementation, vectorized implementation was executed. The vectorized version of the SVM was **89.03** faster when for loss calculation and **113.75** times faster for gradient loss and gradient calculation.

The vectorized version of loss was calculated.

```
(N, D), C = X.shape, W.shape[1]
scores = torch.mm(X, W)
temp_v = (scores + 1 - scores[range(0, N), y].view(-1, 1))
```

```
loss += temp_v[temp_v > 0].sum() / N - 1
loss += reg * (W * W).sum()
```

The vectorized version of dW was calculated

```
temp_w = (temp_v > 0).to(X)
temp_w[range(0, N), y] -= temp_w.sum(dim=1)
dW += torch.mm(X.t(), temp_w)
dW /= N
dW += reg * 2 * W
```

Sample batch was gathered from training data.

```
indices = torch.randint(num_train, (batch_size,))
X_batch = X[indices]
y_batch = y[indices]
```

The weights were updated according to gradient and learning rate.

```
W -= learning_rate * grad
```

Labels were predicted using weights then, stored as y\_pred.

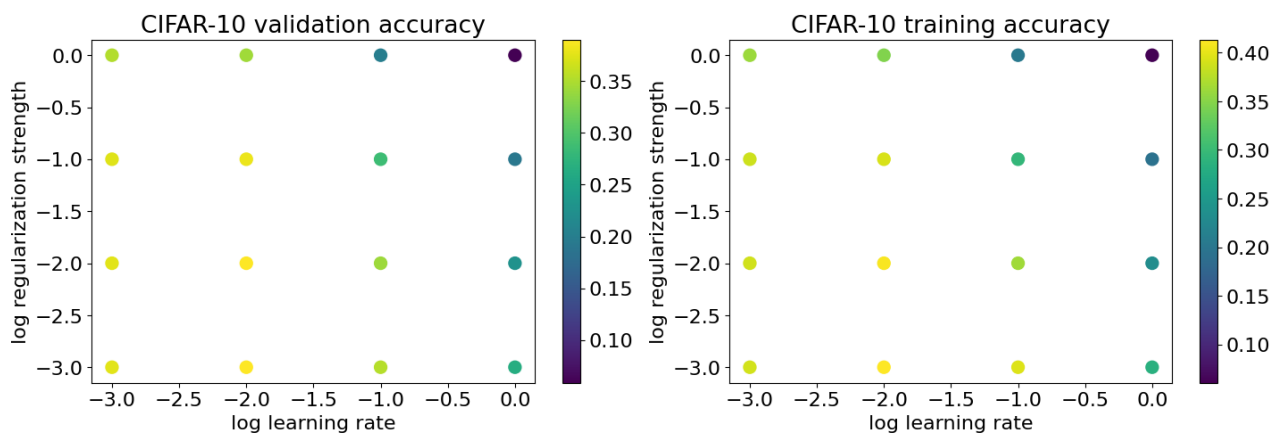
```
y_pred = torch.argmax(X.mm(W), dim=1)
```

Learning rates were determined to test how different values work and the aim is to find the best model for the classification to work.

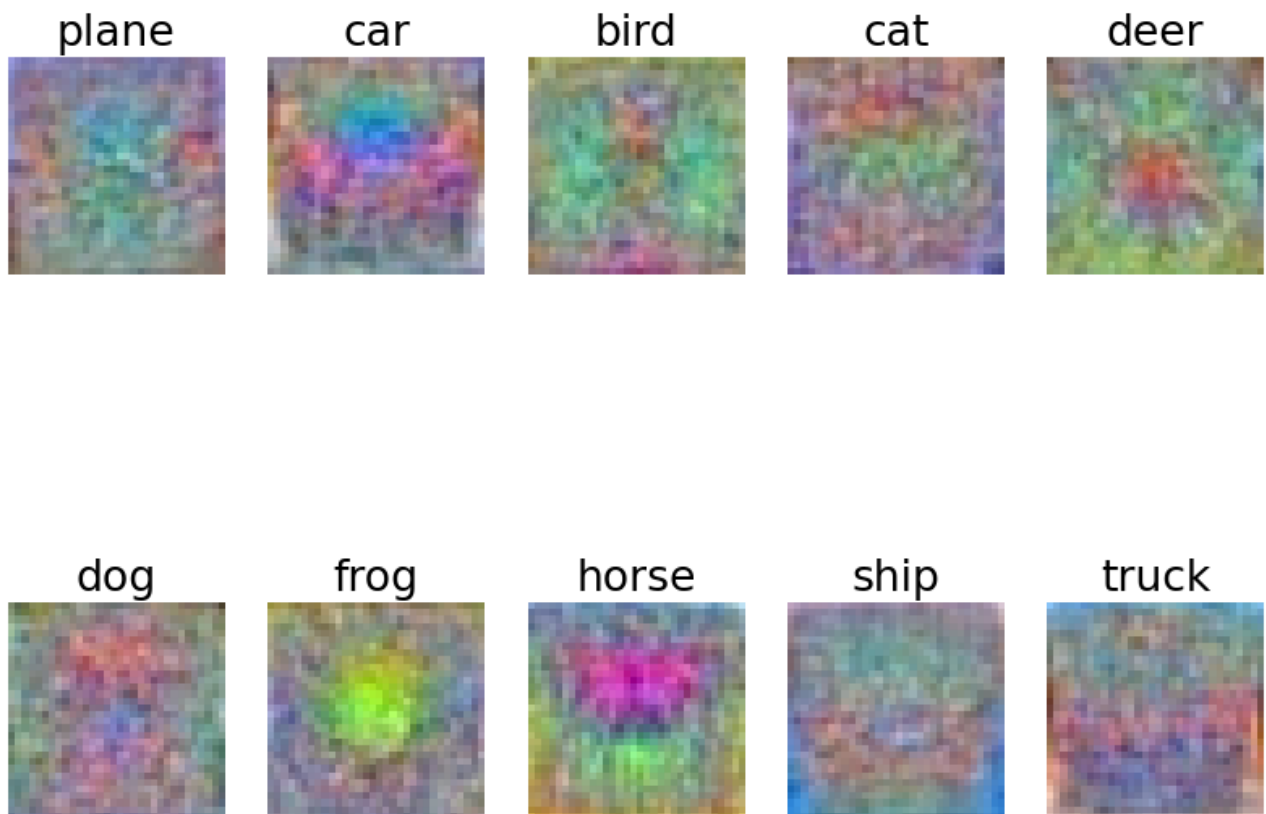
```
learning_rates = [1e-3, 1e-2, 1e-1, 1e0]
regularization_strengths = [1e-3, 1e-2, 1e-1, 1e0]
```

The models were trained and tested. Then the best model was selected. The best model was found with %39.10 cross- validation accuracy. It uses  $10^{-2}$  for both learning rate and regularization strength.

```
X_train, y_train = data_dict['X_train'], data_dict['y_train']
W = cls.train(X_train, y_train, lr, reg, num_iters)
y_train_pred = cls.predict(X_train)
train_acc = (y_train == y_train_pred).float().mean().item()
X_val, y_val = data_dict['X_val'], data_dict['y_val']
y_val_pred = cls.predict(X_val)
val_acc = (y_val == y_val_pred).float().mean().item()
```



Graphs of training accuracy and validation accuracy parameters.



The test of best model was then tested on the final raw pixel test with 38.89% accuracy.

## 2. Softmax Classifier

### a. The Naive Softmax Loss Function

In the first part of the softmax classifier, the naive softmax loss function with nested loops is implemented. It works by iterating over each training example and computing the loss and gradient for that example. The loss and gradient are then accumulated over all training examples and averaged.

$$L(W, X, y) = -\log(y \mid X, W) + \text{reg} * ||W||^2$$

```
# Get the number of classes and the number of training examples
num_class, num_train = W.shape[1], X.shape[0]

# Iterate over the training examples
for i in range(num_train):
    # Compute the scores
    scores = W.t().mv(X[i])
    scores -= scores.max()
    # Compute the loss
    loss -= torch.log(torch.exp(scores[y[i]]) / torch.sum(torch.exp(scores)))

    # Compute the gradient
    for j in range(num_class):
        # dL/dW = dL/ds * ds/dW
        dW[:, j] += torch.exp(scores[j]) / torch.sum(torch.exp(scores)) * X[i]

    # Subtract the gradient for the current training example's correct class
    dW[:, y[i]] -= X[i]

# Add the regularization
loss = loss / num_train + reg * torch.sum(W * W)

# Average the gradient and add regularization
dW = dW / num_train + 2 * reg * W
```

### b. Vectorized Softmax Loss Function

This is a vectorized implementation of the softmax loss function, which means that it does not use any explicit loops. This makes it much more efficient than the naive implementation, especially for large datasets.

```
num_train = X.shape[0]
scores = torch.mm(X, W) # Compute the scores
```

```

# Shift the scores so that the maximum score is 0
scores -= torch.max(scores, dim=1, keepdim=True).values

# Compute the probabilities
probs = torch.exp(scores) / torch.sum(torch.exp(scores), dim=1,
keepdim=True)

# Compute the loss
loss = torch.sum(-torch.log(probs[torch.arange(num_train), y]))
loss /= num_train
loss += reg * torch.sum(W * W) # Add the regularization

# Compute the gradient
probs[torch.arange(num_train), y] -= 1
dW = torch.mm(X.t(), probs)
dW = dW / num_train + 2 * reg * W # Add the regularization
naive loss: 2.302841e+00 computed in 137.929201s
vectorized loss: 2.302841e+00 computed in 2.840042s
Loss difference: 0.00e+00
Gradient difference: 7.40e-16
Speedup: 48.57X

```

As expected, the results from vectorized version is much more faster than naïve version.

### c. Hyperparameter Experiments

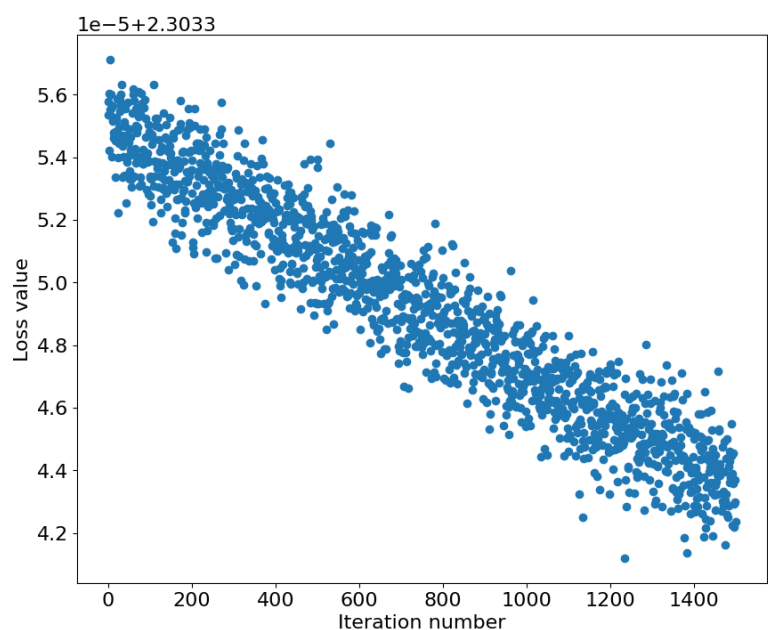
With below default hyperparameters, training accuracy is 8.90% and validation accuracy is 8.54%.

```

learning_rate=1e-10
reg=2.5e4
num_iters=1500

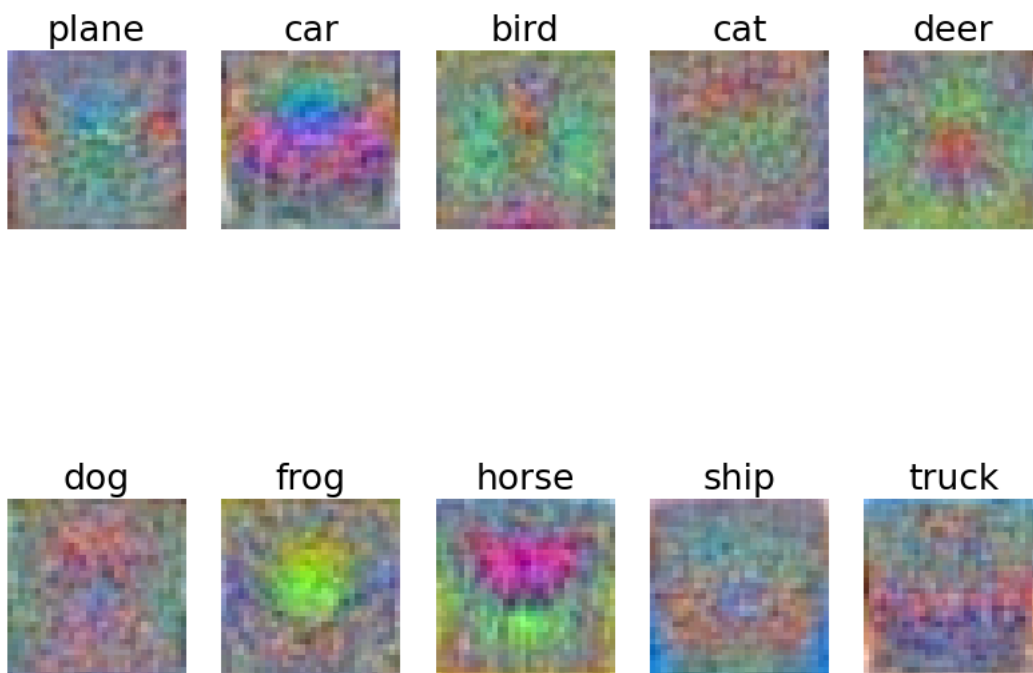
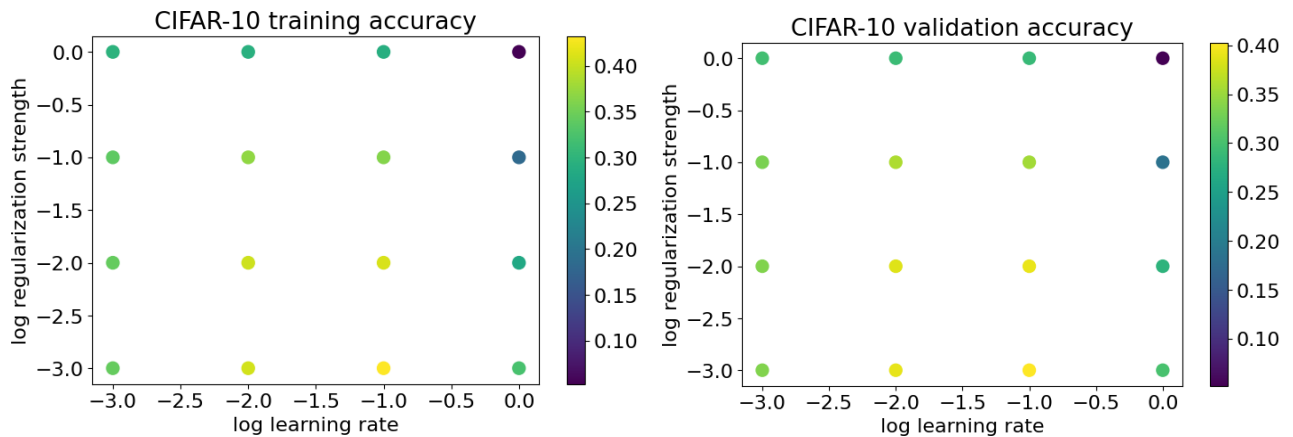
```

The loss is decreasing over time linearly, which indicates that the network is learning and improving. However, the training loss is not yet zero. Also, the shape of the loss function is very flat. The fact that it is in the form of a downward curve indicates a better learning rate.



For parameter generation,  $4 \times 4 = 8$  combinations were tested

```
learning_rates = [1e-3, 1e-2, 1e-1, 1e0]
regularization_strengths = [1e-3, 1e-2, 1e-1, 1e0]
```

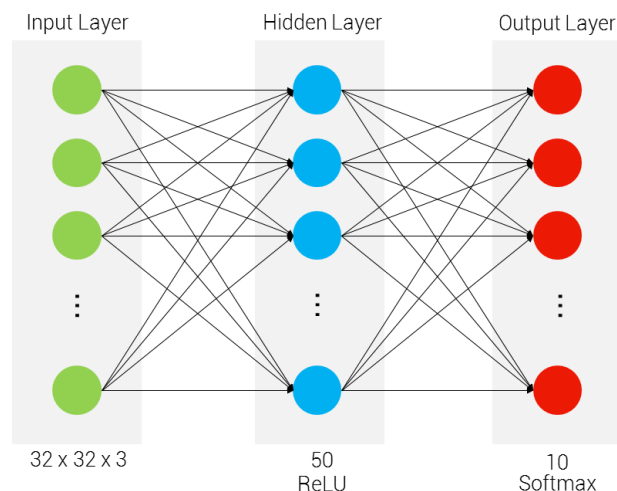


A classification accuracy 40.28% was achieved on the validation set.  
A classification accuracy 40.14% was achieved on the test set.

## PART Q2

**What was the main problem/task/tool implemented? Give brief theoretical information about it (using visuals and formulas/equations is encouraged).**

The main problem was the implementation of two-layer neural network. A two-layer neural network is a type of artificial neural network that consists of two layers of neurons: a hidden layer and an output layer. The hidden layer neurons are connected to the pixels of the input image, and the output layer neurons represent the different classes that the network can predict.



**How did you fulfill the given steps? For the main functions/classes, explain your implementation in general (focus on elementary components).**

The following steps are the implementation of the KNN algorithm in Python:

### 3. Implementing of Model

#### a. Forward pass: compute scores

The forward pass of a neural network is the process of calculating the output of the network, given the input data. This was done by traversing through all of the neurons in the network, from the first layer to the last layer (second layer for this model).

```
hidden = X.mm(W1) + b1
hidden[hidden < 0] = 0 #ReLU
scores = hidden.mm(W2) + b2
```

First hidden layer is computed then ReLU activation function is applied to the hidden layer representation and lastly clasification scores are computed.

## b. Forward pass: compute loss

A loss function is used to measure how well the network performed on the input data. The loss function is calculated by comparing the network's output to the desired output. For the data loss, the softmax loss is computed. For the regularization loss L2 regularization is used on the weight matrices.

```
# Softmax classifier loss
scores = scores - torch.max(scores, 1)[0].reshape(-1, 1)
exp_scores = torch.exp(scores)
probability = exp_scores / torch.sum(exp_scores, dim=1, keepdim=True)
correct_scores = probability[(range(N), y)]
loss = (1/N) * -torch.sum(torch.log(correct_scores))

# L2 regularization
loss += reg * torch.sum(W1**2) + reg * torch.sum(W2**2)
```

## Backward pass

The backward pass of a neural network is the process of updating the weights of the network, based on the loss function. This is done using a gradient descent algorithm with respect to the weights and biases.

```
# Compute gradient on scores
der_scores = probability.clone()
der_scores[(range(N), y)] -= 1
der_scores /= N

# Compute gradients on W and b
grads['b2'] = der_scores.sum(dim = 0)
grads['W2'] = h1.t().mm(der_scores) + 2 * reg * W2

der_h1 = der_scores.mm(W2.T)
der_h1[h1 == 0] = 0

grads['b1'] = der_h1.sum(dim = 0)
grads['W1'] = X.t().mm(der_h1) + 2 * reg * W1
```



## c. Train the network

### Train

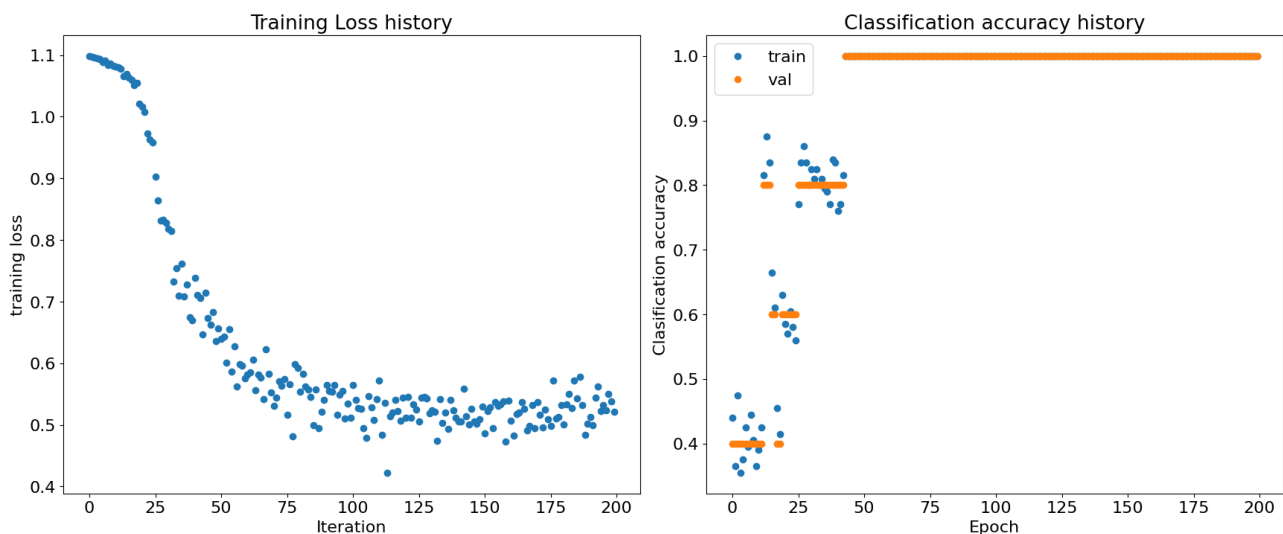
The `nn_train()` function trains a neural network using stochastic gradient descent (SGD). In the `nn_train()` function, just the loss and gradients on the minibatch are computed.

```
params['W2'] -= learning_rate * grads['W2']
params['b2'] -= learning_rate * grads['b2']
params['W1'] -= learning_rate * grads['W1']
params['b1'] -= learning_rate * grads['b1']
```

### Predict

The `nn_predict()` function takes as input the parameters of a neural network, a loss function, and a batch of input data. The scores for each class for each input data point are computed then the label for each input data point is predicted by taking the argmax of the scores.

```
scores, _ = nn_forward_pass(params, X)
y_pred = torch.argmax(scores, dim = 1)
```



The training loss is a measure of how well the network is performing on the training data. The training loss is decreasing over time, which indicates that the network is learning and improving. However, the training loss is not yet zero. the training loss history for this two-layer neural network is promising.

In the classification accuracy history, the validation accuracy is increasing over time, which indicates that the network is learning and improving well. Then, the accuracy is starting to plateau, which means that the network is nearing its maximum performance. This suggests that the network is learning and generalizing well to new data.

## 4. Testing our NN on a real dataset: CIFAR-10

### a. Train a network

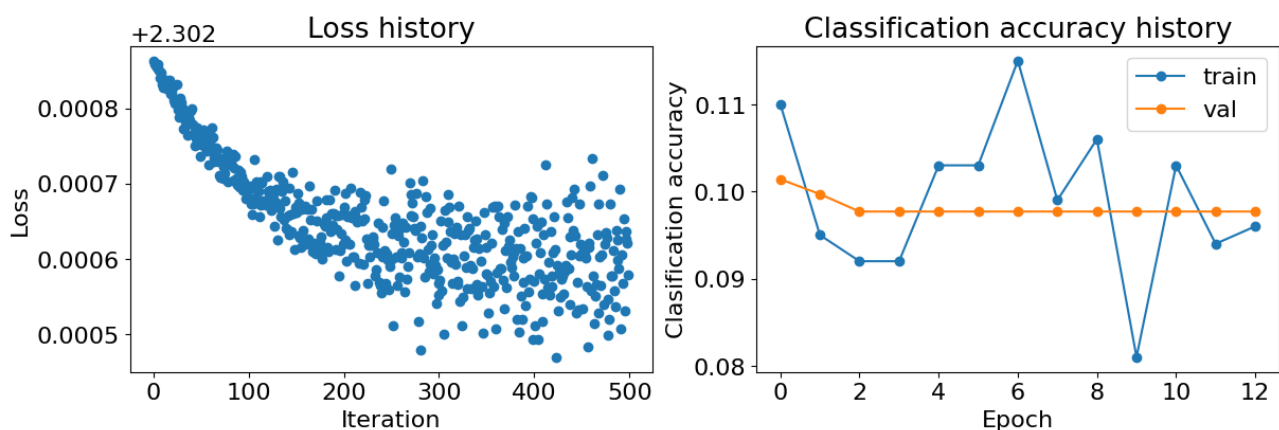
In this section, the implemented model is trained on CIFAR-10 dataset with below default hyper parameters.

```
hidden_size = 36
num_iters=500
batch_size=1000
learning_rate=1e-2
learning_rate_decay=0.95
reg=0.25
```

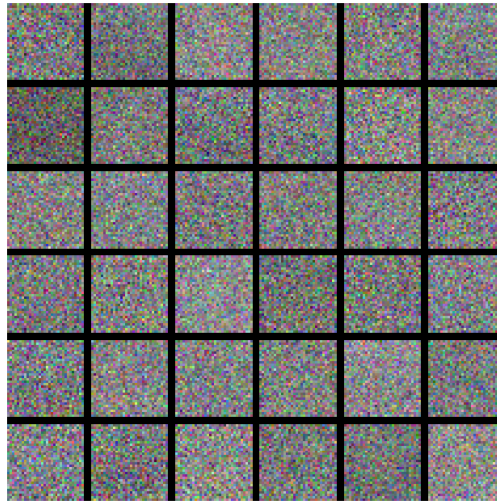
Also the below results were achieved. It is seen that even if the iteration increased the loss didn't change notably.

```
iteration 0 / 500: loss 2.302864
iteration 100 / 500: loss 2.302695
iteration 200 / 500: loss 2.302669
iteration 300 / 500: loss 2.302552
iteration 400 / 500: loss 2.302571
Validation accuracy: 9.77%
```

### b. Debug the training



The loss is seen to decrease linearly, indicating the possibility that the learning rate may be too low. Moreover, the fact that there is no gap between training and validation accuracy indicates that the capacity of the model we use is low and we need to increase its size. Which normally, in a very large model, we would expect to see more fit, which manifests itself as a very large gap between training and validation accuracy.



As you can see above, the image is too noisy due to poor classification and the model is underfitting.

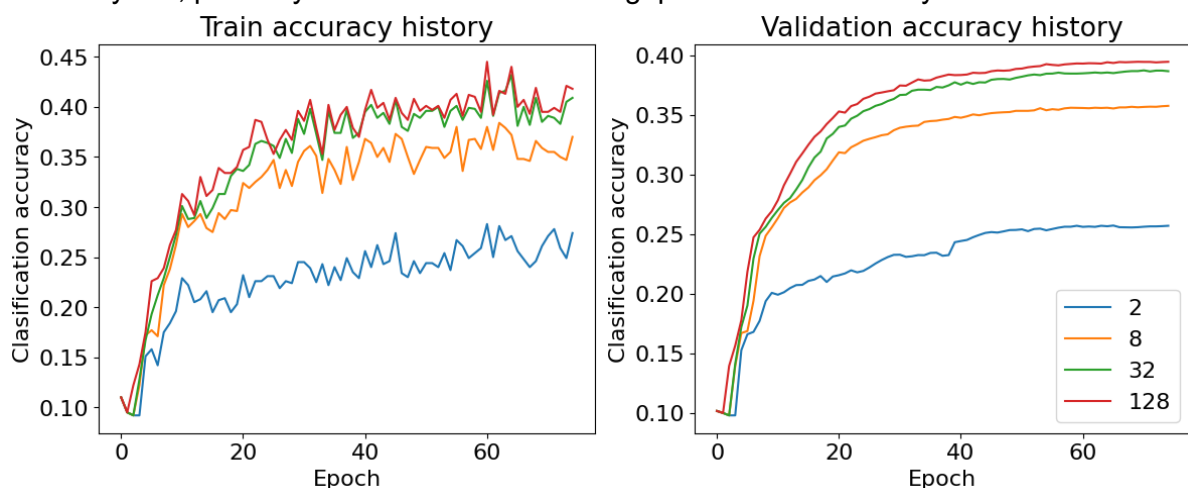
### c. Experiments

#### Capacity

First experiment is held on capacity with below hyper parameters.

```
hidden_sizes = [2, 8, 32, 128]
num_iters=3000
batch_size=1000
learning_rate=0.1
learning_rate_decay=0.95
reg=0.001
```

Considering the significant increase in the number of iterations, the image shows that the network with the largest hidden layer size (128) has the highest training accuracy and validation accuracy. This suggests that the network with the largest hidden layer size can learn the training data better and generalize well to new data. Also, as the layer size increases, the gap decreases exponentially. So, probably there will be no notable gap between hidden layer size 1024 and 2048.

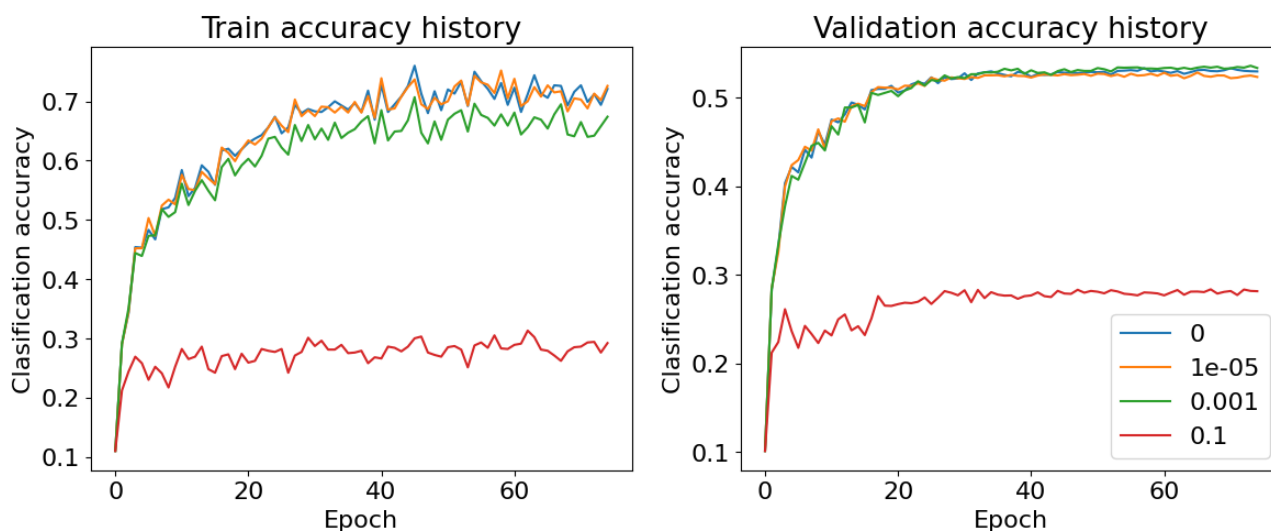


## Regularization

The second experiment is held on regularization with below hyper parameters.

```
hidden_sizes = 128
num_iters=3000
batch_size=1000
learning_rate=1.0
learning_rate_decay=0.95
reg=[0, 1e-5, 1e-3, 1e-1]
```

Although regularization parameter is intended to be tested, learning rate has also made a huge change and it should be taken into consideration.



A too high regularization strength will prevent the network from learning the training data well, while a too low regularization strength will allow the network to overfit the training data. The image shows that the network with the highest regularization strength (0.001) has the highest validation accuracy. This suggests that regularization can help the network generalize better to new data. On the other hand, 0.1 regularization is too low to be usage and it is underfitting.

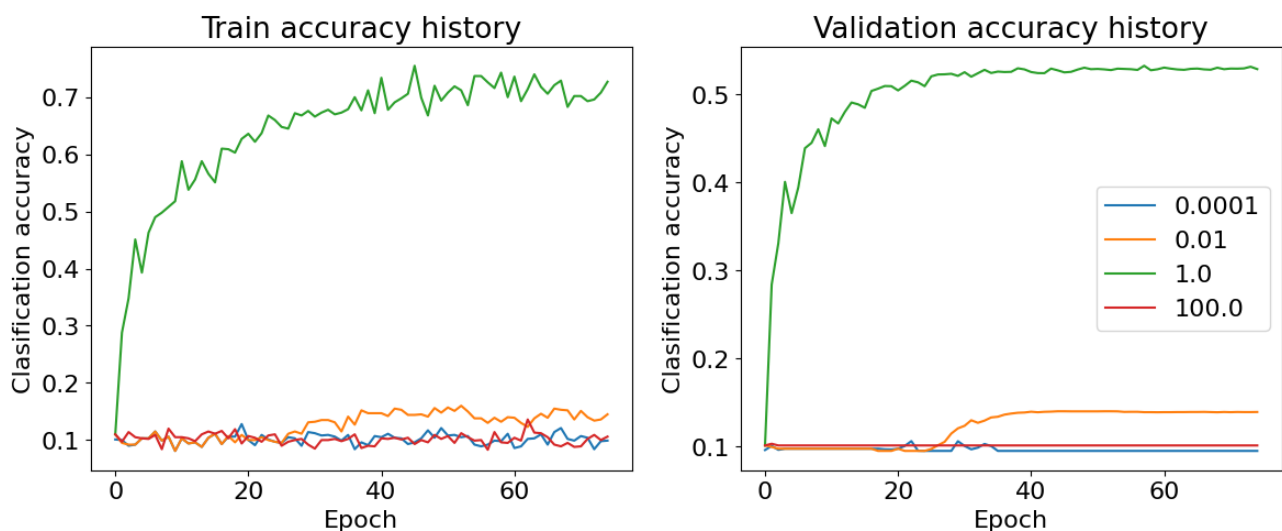
## Learning Rate

The last experiment is held on capacity with below hyper parameters.

```
hidden_sizes = 128
num_iters=3000
batch_size=1000
learning_rate= [1e-4, 1e-2, 1e0, 1e2]
learning_rate_decay=0.95
reg= 1e-4
```

The learning rate is a hyperparameter that controls how much the network updates its weights in response to the error on the training data. A higher learning rate will allow the network to learn more quickly, but it is also more likely to cause the network to overshoot the optimal solution. A lower learning rate will allow the network to learn more slowly, but it is less likely to cause the network to overshoot the optimal solution.

The image shows that the network with the highest learning rate (1.0) has the highest validation accuracy. This suggests that the network with the highest learning rate can find a better set of weights and generalize better to new data. However, a too high learning rate may cause the network to become unstable and not converge to a good set of weights.



#### d. Search Parameters

`nn_get_search_params()` is a grid search algorithm and it is used to search parameters for implemented neural network. For parameter generation,  $4 \times 3 \times 3 = 144$  combinations were tested.

```
learning_rates = [1e-3, 1e-2, 1e-1, 1.0]
hidden_sizes = [256, 512, 1024]
regularization_strengths = [1e-5, 1e-4, 1e-3]
learning_rate_decays = [1.0, 0.97, 0.95]
```

The second function, `find_best_net()`, tunes the hyperparameters using the validation set. The function iterates over all possible combinations of hyperparameters and trains a `TwoLayerNet` model with each combination. The function then evaluates the trained model on the validation set and selects the model with the highest validation accuracy. The function returns the best model, the training statistics for the best model, and the validation accuracy of the best model.

```

# Get the search parameters
learning_rates, hidden_sizes, regularization_strengths, learning_rate_decays
= nn_get_search_params()

# Iterate over all possible combinations of hyperparameters
for lr in learning_rates:
    for hs in hidden_sizes:
        for rs in regularization_strengths:
            for lr_decay in learning_rate_decays:

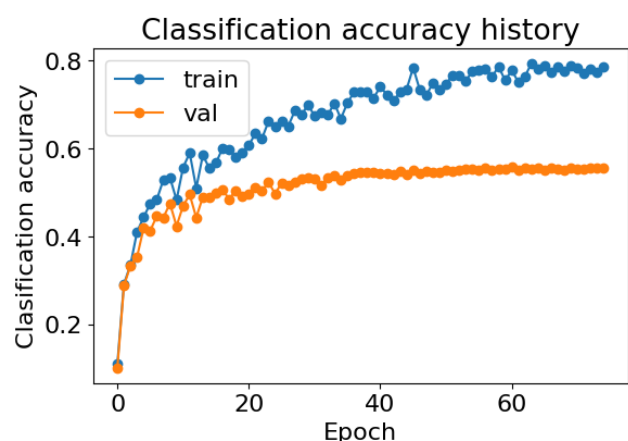
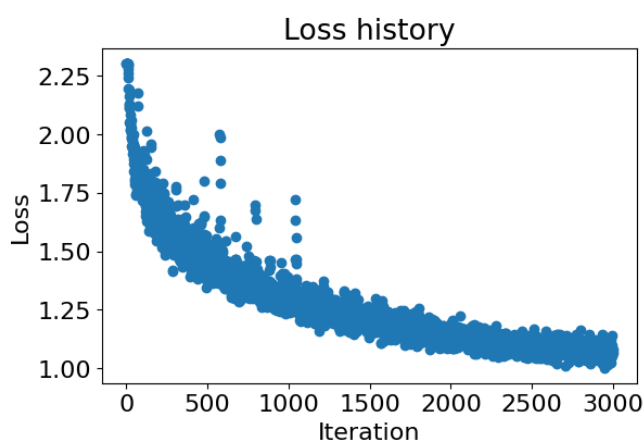
                # Create a new neural network with the given hyperparameters
                net = TwoLayerNet(3 * 32 * 32, hs, 10)

                # Train the neural network
                stats = net.train(data_dict['X_train'], data_dict['y_train'],
                                data_dict['X_val'], data_dict['y_val'],
                                num_iters=3000, batch_size=1000, \
                                learning_rate=lr, learning_rate_decay=lr_decay, reg=rs,
                                verbose=False)

                # Get the validation accuracy
                val_acc = stats['val_acc_history'][-1]

                # If the validation accuracy is better than the current best, update
the best parameters
                if val_acc > best_val_acc:
                    best_net = net
                    best_stat = stats
                    best_val_acc = val_acc

```








A classification accuracy 56.02% was achieved on the validation set.  
A classification accuracy 55.90% was achieved on the test set.

Even if the accuracy is above 50%, the above image is too meaningless for human eyes.  
There is any differentiable object in the image.

**How did it perform? Were you satisfied with your results? Write down your observations and maybe criticize yourself.**

Two-layer neural network with softmax loss function achieved a validation accuracy of 56.02% and a test accuracy of 55.90% on the CIFAR-10 dataset. This is a good performance, but it is not state-of-the-art.



I am satisfied with the results because I am simply trying to learn how to implement a two-layer neural network and train it on a real-world dataset.

**Observations and criticisms:**

The validation accuracy is slightly higher than test accuracy. This suggests that your model is overfitting the training data to some extent. Maybe more regularization or increasing the size of the training dataset can be used try to reduce overfitting.