Image Classification

Members :

- Amr Mohamed Nasr (47)
- Michael Raafat (57)

Screenshot of some results & code:

-**KNN:**

```
def compute_distances_no_loops(self, X):
  """
  Compute the distance between each test point in X and each training point
  in self.X_train using no explicit loops.

  Input / Output: Same as compute_distances_two_loops
  """
  num_test = X.shape[0]
  num_train = self.X_train.shape[0]
  dists = np.zeros((num_test, num_train))
  #########################################################################
  # TODO:                                                                 #
  # Compute the l2 distance between all test points and all training      #
  # points without using any explicit loops, and store the result in      #
  # dists.                                                                #
  #                                                                       #
  # You should implement this function using only basic array operations; #
  # in particular you should not use functions from scipy.                #
  #                                                                       #
  # HINT: Try to formulate the l2 distance using matrix multiplication    #
  #       and two broadcast sums.
  # by michael : root(sum(X^2) - 2 X train^T + sum(train^2)).             #
  # with two sum all matrix operation                                     #
  #########################################################################
  dists = np.sqrt(np.sum(np.power(X, 2),axis= 1, keepdims=1) + np.sum(np.power(self.X_train, 2), axis=1) - 2 * X.dot(self.X_train.T) )
  #########################################################################
  #                       END OF YOUR CODE                                #
  #########################################################################
  return dists
```



```
  num_test = dists.shape[0]
  y_pred = np.zeros(num_test)
  for i in range(num_test):
    # A list of length k storing the labels of the k nearest neighbors to
    # the ith test point.
    closest_y = []
    #########################################################################
    # TODO:                                                                 #
    # Use the distance matrix to find the k nearest neighbors of the ith    #
    # testing point, and use self.y_train to find the labels of these       #
    # neighbors. Store these labels in closest_y.                           #
    # Hint: Look up the function numpy.argsort.                             #
    #########################################################################
    near_i_ks = np.argsort(dists[i,:])[0:k]
    closest_y = self.y_train[near_i_ks]
    #########################################################################
    # TODO:                                                                 #
    # Now that you have found the labels of the k nearest neighbors, you    #
    # need to find the most common label in the list closest_y of labels.   #
    # Store this label in y_pred[i]. Break ties by choosing the smaller     #
    # label.                                                                #
    #########################################################################
    lbls, counts = np.unique(closest_y, return_counts= True)
    y_pred[i] = lbls[np.argmax(counts)]
    #########################################################################
    #                           END OF YOUR CODE                            #
    #########################################################################

  return y_pred
```

```
def compute_distances_two_loops(self, X):
  """
  Compute the distance between each test point in X and each training point
  in self.X_train using a nested loop over both the training data and the
  test data.

  Inputs:
  - X: A numpy array of shape (num_test, D) containing test data.

  Returns:
  - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
    is the Euclidean distance between the ith test point and the jth training
    point.
  """
  num_test = X.shape[0]
  num_train = self.X_train.shape[0]
  dists = np.zeros((num_test, num_train))
  for i in range(num_test):
    for j in range(num_train):
      #####################################################################
      # TODO:                                                             #
      # Compute the l2 distance between the ith test point and the jth    #
      # training point, and store the result in dists[i, j]. You should   #
      # not use a loop over dimension.                                    #
      #####################################################################
      dists[i,j] = np.sqrt(np.sum(np.power(X[i]-self.X_train[j], 2)))
      #####################################################################
      #                       END OF YOUR CODE                            #
      #####################################################################
```

```
def compute_distances_two_loops(self, X):
  """
  Compute the distance between each test point in X and each training point
  in self.X_train using a nested loop over both the training data and the
  test data.

  Inputs:
  - X: A numpy array of shape (num_test, D) containing test data.

  Returns:
  - dists: A numpy array of shape (num_test, num_train) where dists[i, j]
    is the Euclidean distance between the ith test point and the jth training
    point.
  """
  num_test = X.shape[0]
  num_train = self.X_train.shape[0]
  dists = np.zeros((num_test, num_train))
  for i in range(num_test):
    for j in range(num_train):
      #####################################################################
      # TODO:                                                             #
      # Compute the l2 distance between the ith test point and the jth    #
      # training point, and store the result in dists[i, j]. You should   #
      # not use a loop over dimension.                                    #
      #####################################################################
      dists[i,j] = np.sqrt(np.sum(np.power(X[i]-self.X_train[j], 2)))
      #####################################################################
      #                       END OF YOUR CODE                            #
      #####################################################################
```

- Linear Classifier:

```
def predict(self, X):
  """
  Use the trained weights of this linear classifier to predict labels for
  data points.

  Inputs:
  - X: A numpy array of shape (N, D) containing training data; there are N
    training samples each of dimension D.

  Returns:
  - y_pred: Predicted labels for the data in X. y_pred is a 1-dimensional
    array of length N, and each element is an integer giving the predicted
    class.
  """
  y_pred = np.zeros(X.shape[0])
  ###########################################################################
  # TODO:                                                                   #
  # Implement this method. Store the predicted labels in y_pred.            #
  ###########################################################################
  y_pred = X.dot(self.W).argmax(axis=1)
  ###########################################################################
  #                           END OF YOUR CODE                              #
  ###########################################################################
  return y_pred
```

```
# y_batch; after sampling X_batch should have shape (dim, batch_size)   #
# and y_batch should have shape (batch_size,)                           #
#                                                                       #
# Hint: Use np.random.choice to generate indices. Sampling with         #
# replacement is faster than sampling without replacement.              #
#########################################################################
random_is = np.random.choice(num_train, size= batch_size)
X_batch = X[random_is]
y_batch = y[random_is]
#########################################################################
#                       END OF YOUR CODE                                #
#########################################################################

# evaluate loss and gradient
loss, grad = self.loss(X_batch, y_batch, reg)
loss_history.append(loss)

# perform parameter update
#########################################################################
# TODO:                                                                 #
# Update the weights using the gradient and the learning rate.          #
#########################################################################
self.W = self.W - learning_rate * grad
#########################################################################
#                       END OF YOUR CODE                                #
#########################################################################
```

-Linear SVM:

```
def svm_loss_vectorized(W, X, y, reg):
  """
  Structured SVM loss function, vectorized implementation.

  Inputs and outputs are the same as svm_loss_naive.
  """
  num_train = X.shape[0]
  loss = 0.0
  dW = np.zeros(W.shape) # initialize the gradient as zero

  #############################################################################
  # TODO:                                                                     #
  # Implement a vectorized version of the structured SVM loss, storing the    #
  # result in loss.                                                           #
  #############################################################################
  scores = X.dot(W)
  correct_class_scores = scores[np.arange(num_train),y].reshape(num_train,1)
  deltas = np.ones(scores.shape)


  margins = scores - correct_class_scores + deltas
  margins[np.arange(num_train),y] = 0
  margins[margins < 0] = 0

  loss = np.sum(margins)
  # Right now the loss is a sum over all training examples, but we want it
  # to be an average instead so we divide by num_train.
  loss /= num_train
```

```
loss /= num_train


# Add regularization to the loss.
loss += reg * np.sum(W * W)

#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################


#############################################################################
# TODO:                                                                     #
# Implement a vectorized version of the gradient for the structured SVM     #
# loss, storing the result in dW.                                           #
#                                                                           #
# Hint: Instead of computing the gradient from scratch, it may be easier    #
# to reuse some of the intermediate values that you used to compute the     #
# loss.                                                                     #
#############################################################################
margins[margins>0] = 1
margins[np.arange(num_train),y] = -np.sum(margins,axis=1)
dW = X.T.dot(margins)

dW /= num_train
dW += reg * 2 * W
#############################################################################
#                              END OF YOUR CODE                             #
#############################################################################
```

```
        loss /= num_train


        # Add regularization to the loss.
        loss += reg * np.sum(W * W)

        #############################################################################
        #                              END OF YOUR CODE                             #
        #############################################################################


        #############################################################################
        # TODO:                                                                     #
        # Implement a vectorized version of the gradient for the structured SVM     #
        # loss, storing the result in dW.                                           #
        #                                                                           #
        # Hint: Instead of computing the gradient from scratch, it may be easier    #
        # to reuse some of the intermediate values that you used to compute the     #
        # loss.                                                                     #
        #############################################################################
        margins[margins>0] = 1
        margins[np.arange(num_train),y] = -np.sum(margins,axis=1)
        dW = X.T.dot(margins)

        dW /= num_train
        dW += reg * 2 * W
        #############################################################################
        #                              END OF YOUR CODE                             #
        #############################################################################
```

-Neural Net:

```
        # Compute the loss
        loss = None
        #############################################################################
        # TODO: Finish the forward pass, and compute the loss. This should include  #
        # both the data loss and L2 regularization for W1 and W2. Store the result  #
        # in the variable loss, which should be a scalar. Use the Softmax           #
        # classifier loss.                                                          #
        #############################################################################
        inter_z2 = z2 - np.max(z2, axis = 1, keepdims = 1);
        bef_a2 = np.exp(inter_z2);
        a2 = bef_a2 / np.sum(bef_a2, axis = 1, keepdims = 1);
        examples = range(N);
        loss = -np.sum(np.log(a2[examples,y]));
        loss /= N;
        loss += reg * np.sum(W1 * W1);
        loss += reg * np.sum(W2 * W2);
        #############################################################################
        #                              END OF YOUR CODE                             #
        #############################################################################

        # Backward pass: compute gradients
        grads = {}
```

```
    # grads['W1'] should store the gradient on W1, and be a matrix of same size #
    ############################################################################
    dW1 = np.zeros_like(W1);
    dW2 = np.zeros_like(W2);
    db1 = np.zeros_like(b1);
    db2 = np.zeros_like(b2);
    # derivative of loss over z2.
    dz2 = a2.copy();
    dz2[examples, y] -= 1;
    # Weight 2
    dW2 = np.matmul(a1.T, dz2) / N + reg * 2 * W2;
    # Bias 2
    db2 = np.sum(dz2, axis=0, keepdims = 1);
    db2 /= N;
    # Layer 1.
    da1 = np.matmul(dz2, W2.T);
    dz1 = da1 * (z1 > 0).astype(int);
    db1 = np.sum(dz1, axis = 0, keepdims = 1);
    db1 /= N;
    dW1 = np.matmul(X.T, dz1) / N + reg * 2 * W1;

    grads['W1'] = dW1;
    grads['W2'] = dW2;
    grads['b1'] = db1;
    grads['b2'] = db2;
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################

for it in range(num_iters):
  X_batch = None
  y_batch = None

  ###########################################################################
  # TODO: Create a random minibatch of training data and labels, storing  #
  # them in X_batch and y_batch respectively.                             #
  ###########################################################################
  random_is = np.random.choice(num_train, size= batch_size)
  X_batch = X[random_is]
  y_batch = y[random_is]
  ###########################################################################
  #                             END OF YOUR CODE                            #
  ###########################################################################

  # Compute loss and gradients using the current minibatch
  loss, grads = self.loss(X_batch, y=y_batch, reg=reg)
  loss_history.append(loss)

  ###########################################################################
  # TODO: Use the gradients in the grads dictionary to update the         #
  # parameters of the network (stored in the dictionary self.params)      #
  # using stochastic gradient descent. You'll need to use the gradients   #
  # stored in the grads dictionary defined above.                         #
  ###########################################################################
  self.params['W1'] = self.params['W1'] - learning_rate * grads['W1'];
  self.params['W2'] = self.params['W2'] - learning_rate * grads['W2'];
  self.params['b1'] = self.params['b1'] - learning_rate * grads['b1'];
  self.params['b2'] = self.params['b2'] - learning_rate * grads['b2'];
  ###########################################################################
```

```
def predict(self, X):
  """
  Use the trained weights of this two-layer network to predict labels for
  data points. For each data point we predict scores for each of the C
  classes, and assign each data point to the class with the highest score.

  Inputs:
  - X: A numpy array of shape (N, D) giving N D-dimensional data points to
    classify.

  Returns:
  - y_pred: A numpy array of shape (N,) giving predicted labels for each of
    the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
    to have class c, where 0 <= c < C.
  """
  y_pred = None

  ###########################################################################
  # TODO: Implement this function; it should be VERY simple!                #
  ###########################################################################
  z2 = self.loss(X);
  inter_z2 = z2 - np.max(z2, axis = 1, keepdims = 1);
  bef_a2 = np.exp(inter_z2);
  a2 = bef_a2 / np.sum(bef_a2, axis = 1, keepdims = 1);
  y_pred = a2.argmax(axis = 1);
  ###########################################################################
  #                             END OF YOUR CODE                            #
  ###########################################################################

  return y_pred
```

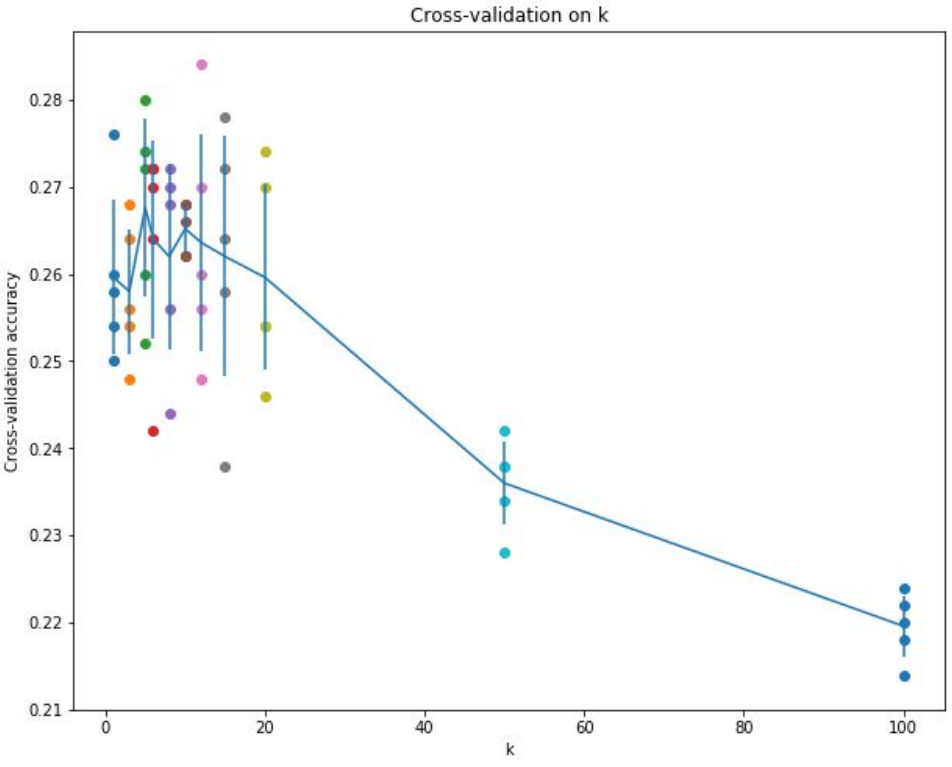
```
def predict(self, X):
  """
  Use the trained weights of this two-layer network to predict labels for
  data points. For each data point we predict scores for each of the C
  classes, and assign each data point to the class with the highest score.

  Inputs:
  - X: A numpy array of shape (N, D) giving N D-dimensional data points to
    classify.

  Returns:
  - y_pred: A numpy array of shape (N,) giving predicted labels for each of
    the elements of X. For all i, y_pred[i] = c means that X[i] is predicted
    to have class c, where 0 <= c < C.
  """
  y_pred = None

  ###########################################################################
  # TODO: Implement this function; it should be VERY simple!                #
  ###########################################################################
  z2 = self.loss(X);
  inter_z2 = z2 - np.max(z2, axis = 1, keepdims = 1);
  bef_a2 = np.exp(inter_z2);
  a2 = bef_a2 / np.sum(bef_a2, axis = 1, keepdims = 1);
  y_pred = a2.argmax(axis = 1);
  ###########################################################################
  #                              END OF YOUR CODE                           #
  ###########################################################################

  return y_pred
```

-Softmax:

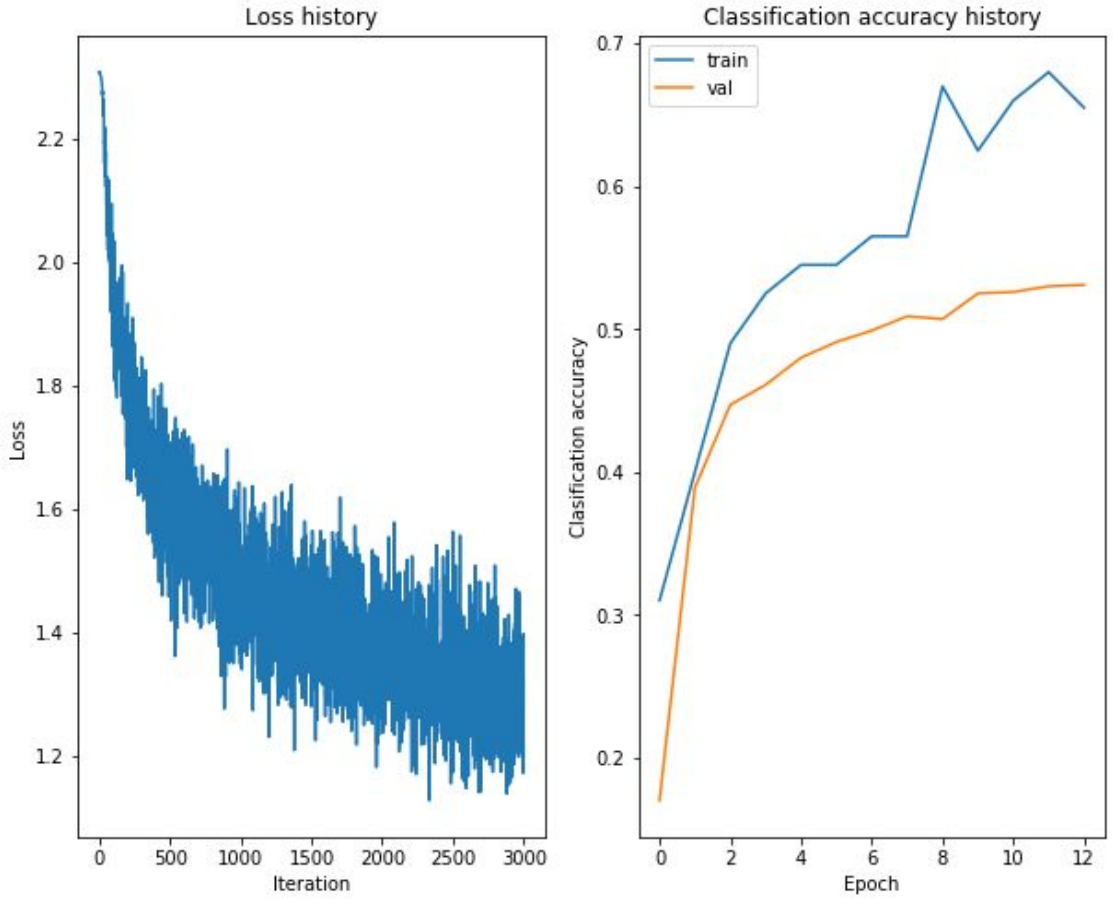
```
        # Initialize the loss and gradient to zero.
        loss = 0.0
        dW = np.zeros_like(W)

        ###########################################################################
        # TODO: Compute the softmax loss and its gradient using no explicit loops.  #
        # Store the loss in loss and the gradient in dW. If you are not careful     #
        # here, it is easy to run into numeric instability. Don't forget the        #
        # regularization!                                                           #
        ###########################################################################
        # y_hat (N,C).
        N = X.shape[0];
        C = W.shape[1];
        examples = range(N);
        y_hat = np.matmul(X,W);
        y_hat -= np.max(y_hat, axis = 1, keepdims = 1);
        y_hat_prob = np.exp(y_hat) / np.sum(np.exp(y_hat), axis = 1, keepdims = 1);
        loss = -np.sum(np.log(y_hat_prob[examples,y]));
        loss /= N;
        loss += reg * np.sum(W * W);
        y_hat_prob[examples, y] -= 1;
        dW = np.matmul(X.T, y_hat_prob);
        dW /= N;
        dW += reg * 2 * W;
        ###########################################################################
        #                             END OF YOUR CODE                            #
        ###########################################################################

        return loss, dW
```

```
#############################################################################
# TODO: Compute the softmax loss and its gradient using explicit loops.     #
# Store the loss in loss and the gradient in dW. If you are not careful     #
# here, it is easy to run into numeric instability. Don't forget the        #
# regularization!                                                           #
#############################################################################
N = y.shape[0];
C = W.shape[1];
for i in range(N):
    # Calculate estimations (1,C).
    y_hat = np.matmul(X[i,:],W);
    #prevent numeric instability.
    y_hat -= np.max(y_hat);
    y_hat_exp = np.exp(y_hat);
    exp_sum = np.sum(y_hat_exp);
    y_hat_prob = y_hat_exp / exp_sum;
    for c in range(C):
        act = 0;
        if (c == y[i]):
            loss += -np.log(y_hat_exp[c]/exp_sum);
            act = 1;
        dW[:,c] += X[i,:].T * (y_hat_prob[c] - act);


loss /= N;
dW /= N;
dW += reg * 2 * W;
loss += reg * np.sum(W * W);
#############################################################################
#                             END OF YOUR CODE                              #
#############################################################################
```

Some Screenshots from notebooks



```
Net specification :  1000 0.0007 0.2 3000
```

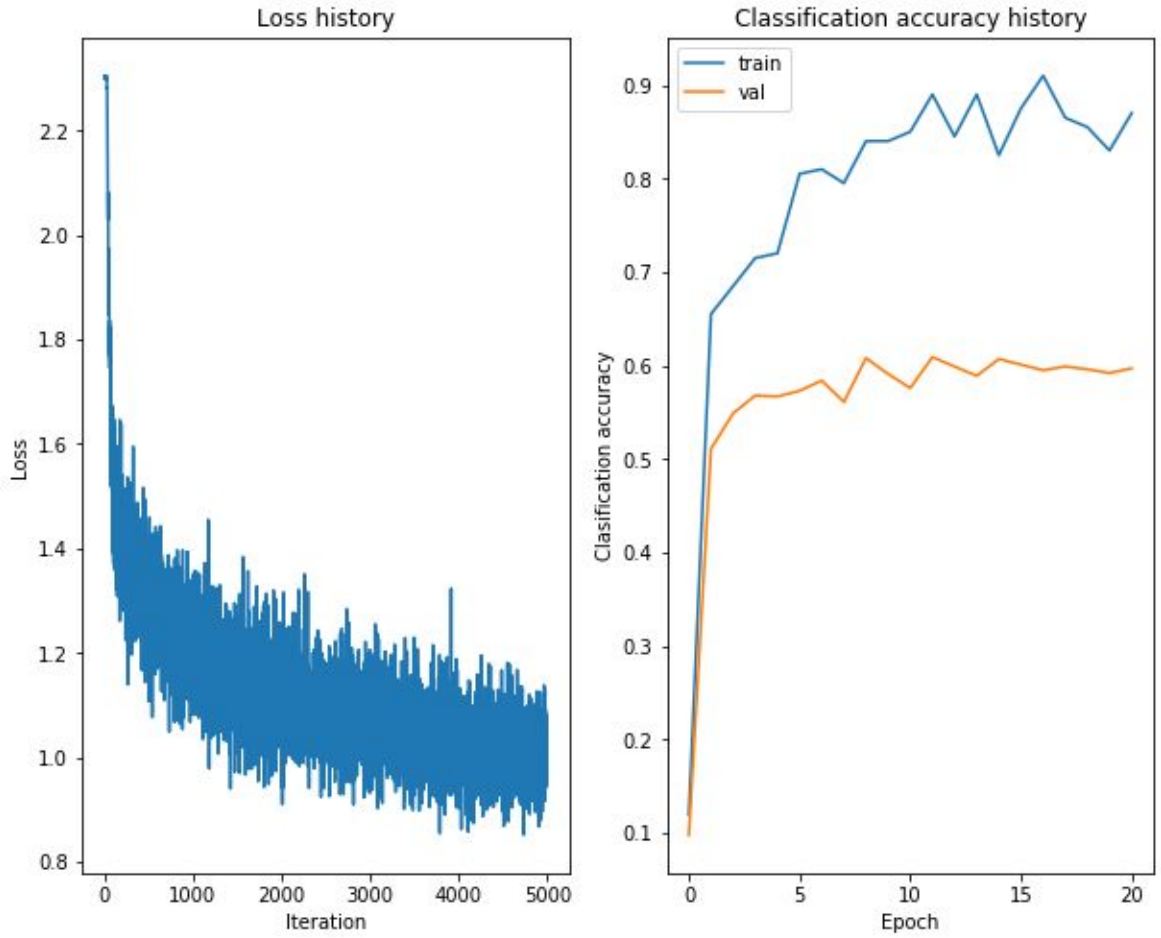



```
Train accuracy :  0.5975510204081632  Validation accuracy:  0.542
```

```
# evaluate on test set
# Evaluate the best softmax on test set
y_test_pred = best_softmax.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print('softmax on raw pixels final test set accuracy: %f' % (test_accuracy, ))
```

```
softmax on raw pixels final test set accuracy: 0.360000
```

```
Net specification :  1000 0.55 0.001 5000
```




```
Train accuracy :  0.7668979591836734  Validation accuracy:  0.607
```

Presentation of data to each classifier:

KNN/Softmax/Two_layer_network : raw pixel data(rgb flattened) entered for each sample.
SVM : raw pixel data(rgb flattened) entered for each sample with mean of each pixel reduced to 0.
SVM & two layer network were then reused but on the extracted features from the raw pixels using hog(something that works like an edge detector), and color histogram.

Models Accuracies

Model	Validation Accuracy	Test Accuracy	Hyperparameters
KNN	26.4%	28.2%	K = 6
SVM	36.6%	35.9%	LR = 2.5e-07 Reg = 7e+04
Softmax	37.5%	36%	LR = 3e-06 Reg = 3e3
Two layer network	54.2%	54.5%	Hidden unit = 1000 LR = 0.0007 Reg = 0.2 Iterations = 3000
SVM(feature data)	44.1%	41.9%	LR = 1e-07 Reg = 1e06
Two layer network(feature data).	60.7%	59.8%	Hidden unit = 1000 LR = 0.55 Reg = 0.001 Iterations = 5000