Assignment 4 - Convolutional Neural Network

In this assignment we will develop a neural network with fully-connected layers to perform classification

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import sys
from sklearn import datasets
if sys.version_info >= (3, 0):
    def xrange(*args, **kwargs):
        return iter(range(*args, **kwargs))
```

In [2]:
```
#load dataset
def load_dataset():
    iris = datasets.load_iris()
    X = iris.data
    y = iris.target
    return X, y

def train_test_split(X, y):
    idx = np.arange(len(X))
    train_size = int(len(X) * 2/3)
    val_size = int(len(X) * 1/6)
    np.random.shuffle(idx)
    X = X[idx]
    y = y[idx]
    X_train, X_val, X_test = X[:train_size], X[train_size:train_size+val_size], X[train_size+val_size:]
    y_train, y_val, y_test = y[:train_size], y[train_size:train_size+val_size], y[train_size+val_size:]

    return X_train, y_train, X_val, y_val, X_test, y_test
```

We will use the following class `TwoLayerCNN` to represent instances of our network. The network parameters are stored in the instance variable `self.params` where keys are string parameter names and values are numpy arrays. Below, we initialize toy data and a toy model that we will use to develop your implementation. You need to complete the functions.

In [11]:
```
class TwoLayerCNN(object):
  """
  A two-layer fully-connected neural network. The net has an input dimension of
  N, a hidden layer dimension of H, and performs classification over C classes.
  We train the network with a softmax loss function and L2 regularization on the
  weight matrices. The network uses a ReLU nonlinearity after the first fully
  connected layer.

  In other words, the network has the following architecture:

  input - fully connected layer - ReLU - fully connected layer - softmax

  The outputs of the second fully-connected layer are the scores for each class.
  """

  def __init__(self, input_size, hidden_size, output_size, std=1e-4):
    """
    Initialize the model. Weights are initialized to small random values and
    biases are initialized to zero. Weights and biases are stored in the
    variable self.params, which is a dictionary with the following keys:

    W1: First layer weights; has shape (D, H)
    b1: First layer biases; has shape (H,)
    W2: Second layer weights; has shape (H, C)
    b2: Second layer biases; has shape (C,)

    Inputs:
    - input_size: The dimension D of the input data.
    - hidden_size: The number of neurons H in the hidden layer.
    - output_size: The number of classes C.
    """
    self.params = {}
    self.params['W1'] = std * np.random.randn(input_size, hidden_size)
    self.params['b1'] = np.zeros(hidden_size)
    self.params['W2'] = std * np.random.randn(hidden_size, output_size)
    self.params['b2'] = np.zeros(output_size)

  def loss(self, X, y=None, reg=0.0):
    """
    Compute the loss and gradients for a two layer fully connected neural
    network.

    Inputs:
    - X: Input data of shape (N, D). Each X[i] is a training sample.
    - y: Vector of training labels. y[i] is the label for X[i], and each y[i] is
      an integer in the range 0 <= y[i] < C. This parameter is optional; if it
      is not passed then we only return scores, and if it is passed then we
      instead return the loss and gradients.
    - reg: Regularization strength.
```

```python
    Returns:
    If y is None, return a matrix scores of shape (N, C) where scores[i, c] is
    the score for class c on input X[i].

    If y is not None, instead return a tuple of:
    - loss: Loss (data loss and regularization loss) for this batch of training
      samples.
    - grads: Dictionary mapping parameter names to gradients of those parameters
      with respect to the loss function; has the same keys as self.params.
    """
    # Unpack variables from the params dictionary
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    N, D = X.shape

    # Compute the forward pass
    scores = None
    #############################################################################
    # Full Mark: 1                                                              #
    # TODO: Perform the forward pass, computing the class scores for the input. #
    # Store the result in the scores variable, which should be an array of      #
    # shape (N, C).                                                             #
    #############################################################################

    # Using ReLUs as the Activation Function
    raw_input = np.dot(X,W1) + b1
    relu_activation = np.array(np.maximum(0, (raw_input)))
    scores = np.array(np.dot(relu_activation, W2) + b2)

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

    # If the targets are not given then jump out, we're done
    if y is None:
      return scores

    # Compute the loss
    loss = None

    scores -= np.max(scores, axis=1, keepdims=True) # avoid numeric instability

    #############################################################################
    # Full Mark: 2                                                              #
    # 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.                                                          #
    #############################################################################

    # softmax
    exp_scores = np.exp(scores)
    scores = exp_scores / exp_scores.sum()

    # Loss
    y_matrix = y.reshape(-1)
    y_matrix = np.eye(3)[y_matrix]
    reg_term = (reg / (2 * y.size)) * (np.sum(np.square(W1)) + np.sum(np.square(W2)))
    loss = (-1 / y.size) * np.sum((np.log(scores) * y_matrix) +
                          np.log(1 - scores) * (1 - y_matrix)) + reg_term

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

    # Backward pass: compute gradients
    grads = {}
    #############################################################################
    # Full Mark: 2                                                              #
    # TODO: Compute the backward pass, computing the derivatives of the weights #
    # and biases. Store the results in the grads dictionary. For example,       #
    # grads['W1'] should store the gradient on W1, and be a matrix of same size #
    #############################################################################


    out_a_diff = scores
    out_a_diff[np.arange(N), y] -= 1

     # W2 gradient
    dW2 = np.dot(relu_activation.T, out_a_diff) / N + reg * W2

    # b2 gradient
    db2 = np.sum(out_a_diff, axis=0) / N

    delta = np.dot(out_a_diff, W2.T)
    delta_z = np.dot(out_a_diff, W2.T) * (raw_input > 0)

    # W1 gradient
    dW1 = np.dot(X.T, delta_z) / N + reg * W1
```

```python
    # b1 gradient
    db1 = np.sum(delta_z , axis=0) / N

    # store the results in the grads dictionary
    grads = {'W1':dW1, 'b1':db1, 'W2':dW2, 'b2':db2}

    # store the results in the grads dictionary
    grads = {'W1':dW1, 'b1':db1, 'W2':dW2, 'b2':db2}

    # store the results in the grads dictionary
    grads = {'W1':dW1, 'b1':db1, 'W2':dW2, 'b2':db2}

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

    return loss, grads

  def train(self, X, y, X_val, y_val,
            learning_rate=1e-3, learning_rate_decay=0.95,
            reg=5e-6, num_iters=100,
            batch_size=200, verbose=False):
    """
    Train this neural network using stochastic gradient descent.

    Inputs:
    - X: A numpy array of shape (N, D) giving training data.
    - y: A numpy array f shape (N,) giving training labels; y[i] = c means that
      X[i] has label c, where 0 <= c < C.
    - X_val: A numpy array of shape (N_val, D) giving validation data.
    - y_val: A numpy array of shape (N_val,) giving validation labels.
    - learning_rate: Scalar giving learning rate for optimization.
    - learning_rate_decay: Scalar giving factor used to decay the learning rate
      after each epoch.
    - reg: Scalar giving regularization strength.
    - num_iters: Number of steps to take when optimizing.
    - batch_size: Number of training examples to use per step.
    - verbose: boolean; if true print progress during optimization.
    """
    num_train = X.shape[0]
    iterations_per_epoch = max(num_train / batch_size, 1)

    # Use SGD to optimize the parameters in self.model
    loss_history = []
    train_acc_history = []
    val_acc_history = []

    for it in xrange(num_iters):

      #########################################################################
      # Full Mark: 0.5                                                        #
      # TODO: Create a random minibatch of training data and labels using     #
      # given num_train and batch_size, storing them in X_batch and y_batch   #
      # respectively.                                                         #
      #########################################################################

      rand_id = np.random.choice(num_train, batch_size)
      X_batch = X[rand_id]
      y_batch = y[rand_id]

      #########################################################################
      #                             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)

      #########################################################################
      # Full Mark: 0.5                                                        #
      # 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'] -= learning_rate * grads['W1']
      self.params['W2'] -= learning_rate * grads['W2']
      self.params['b1'] -= learning_rate * grads['b1']
      self.params['b2'] -= learning_rate * grads['b2']

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

      if verbose and it % 10 == 0:
        print('iteration %d / %d: loss %f' % (it, num_iters, loss))
```

```
        # Every epoch, check train and val accuracy and decay learning rate.
        if it % iterations_per_epoch == 0:
          # Check accuracy
          train_acc = (self.predict(X_batch) == y_batch).mean()
          val_acc = (self.predict(X_val) == y_val).mean()
          train_acc_history.append(train_acc)
          val_acc_history.append(val_acc)

          # Decay learning rate
          learning_rate *= learning_rate_decay

      return {
        'loss_history': loss_history,
        'train_acc_history': train_acc_history,
        'val_acc_history': val_acc_history,
      }

    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.
      """

      ###########################################################################
      # Full Mark: 1                                                            #
      # TODO: Implement this function                                           #
      ###########################################################################

      h_z = np.dot(X, self.params['W1']) + self.params['b1']
      h_a = np.maximum(0, h_z)
      o_z = np.dot(h_a, self.params['W2']) + self.params['b2']
      o_a = np.exp(o_z) / np.exp(o_z).sum()
      y_pred = np.argmax(o_a, axis=1)

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

      return y_pred
```

In [81]:

```
# To check your implementations.
X,y =load_dataset()
X_train, y_train, X_val, y_val, X_test, y_test=train_test_split(X, y)

###########################################################################
# Full Mark: 1                                                            #
# TODO: 1. Using TwoLayerCNN to train on given datasets                   #
#       2. Print out the final loss                                       #
#       3. Print out the test accuracy                                    #
###########################################################################

input_size = 4
hidden_size = 12
num_classes = 3
net = TwoLayerCNN(input_size, hidden_size, num_classes)
# TODO

NN_model = net.train(X_train,y_train,X_val,y_val)
print ('Final training loss: ', NN_model['loss_history'][-1])
val_acc = (net.predict(X_val) == y_val).mean()
print ('Validation accuracy: ', val_acc)

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

```
Final training loss:  6.400271245193619
Validation accuracy:  0.44
```

The loss function and the accuracies on the training and validation sets would give more insight views.

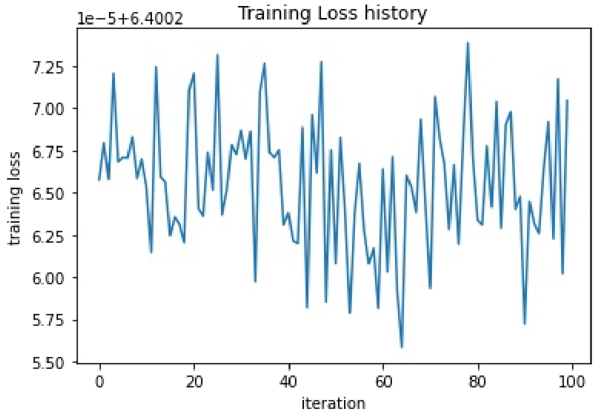
In [82]:

```
###########################################################################
# Full Mark: 0.5                                                          #
# TODO: Plot training loss history                                        #
###########################################################################

plt.plot(stats['loss_history'])
plt.title('Loss history')
```

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

plt.xlabel('iteration')
plt.ylabel('training loss')
plt.title('Training Loss history')
plt.show()
```



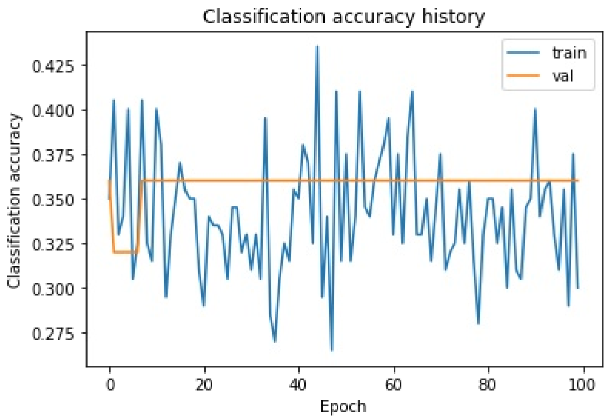
In [83]:

```
###########################################################################
# Full Mark: 0.5                                                          #
# TODO: Plot Classification accuracy history, compare train/val accuracy  #
###########################################################################

plt.plot(stats['train_acc_history'], label='train')
plt.plot(stats['val_acc_history'], label='val')

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

plt.xlabel('Epoch')
plt.ylabel('Classification accuracy')
plt.title('Classification accuracy history')
plt.legend()
plt.show()
```



In [89]:

```
###############################################################################
# Full Mark: 1                                                                #
# TODO: Describe or using codes to show how you tune your hyperparameters     #
# (hidden layer size, learning rate, numer of training epochs, regularization #
# strength and so on). Is your result good? Does it look underfiting?         #
# Overfiting?                                                                 #
###############################################################################

# Type 1
input_size = 4
hidden_size = 15
num_classes = 3
net = TwoLayerCNN(input_size, hidden_size, num_classes)
NN_model = net.train(X_train, y_train, X_val, y_val,learning_rate=1e-2, learning_rate_decay=0.95,reg=1e-3,num_ite
            batch_size=100,verbose=False)

print ('Final training loss: ', NN_model['loss_history'][-1])
val_acc = (net.predict(X_val) == y_val).mean()
print ('Validation accuracy: ', val_acc)

# Type 2
input_size = 4
hidden_size = 9
num_classes = 3
net = TwoLayerCNN(input_size, hidden_size, num_classes)
```

```
NN_model = net.train(X_train, y_train, X_val, y_val,learning_rate=1e-1, learning_rate_decay=0.95,reg=1e-4,num_ite
            batch_size=100,verbose=False)

print ('Final training loss: ', NN_model['loss_history'][-1])
val_acc = (net.predict(X_val) == y_val).mean()
print ('Validation accuracy: ', val_acc)


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

```
Final training loss:  5.710700459295474
Validation accuracy:  0.44
Final training loss:  5.698047987642522
Validation accuracy:  0.32
```

Explain your hyperparameter tuning process below.

$\color{blue}{\textit Your Answer:}$ As the number of iterations increase, the accuracy increases. The model with smaller batch size lead to convolving quicker but the model can overfit due to frequent updation of weights with smaller batches. A smaller learning late with a high iteration helps us reach a better accuracy rate.