1. Two Layer NN
    a. Forward Pass:
        i. Forward pass:

```python
def relu(a):
  result = a * (a > 0)
  return result

#First-layer output
l1_output = np.dot(X, W1.T) + b1
l1_relu = relu(l1_output)

#Second-layer output
l2_output = np.dot(l1_relu, W2.T) + b2
#Using softmax as the output score
scores = l2_output

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #


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

        ii.
        iii. Loss Calculation:

```python
# scores is num_examples by num_classes
def l2_norm(a):
  result = np.sqrt(np.sum(a**2))
  return result
def softmax(y_hat):
  exps = np.exp(y_hat - np.max(y_hat, axis= 1, keepdims= True))
  result = exps / np.sum(exps)
  return result
def softmax_loss(sc, y_actual):
  sc = softmax(sc)
  yhat = sc[np.arange(sc.shape[0]), y_actual]
  result = -np.sum(np.log(yhat)) / sc.shape[0]
  return result

# Calculate the l2 norm for each weight
W1_l2norm = l2_norm(W1)
W2_l2norm = l2_norm(W2)
reg_loss = reg * (W1_l2norm + W2_l2norm)

loss = softmax_loss(scores, y) + reg_loss
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

        iv.
    b. Gradient:

```python
def relu_backward(dout, cache):
  x = cache
  dx = dout * (x > 0)
  return dx
# Calculate dW2
dW2 = (relu(np.dot(X, W1.T) + b1).T @ np.ones(scores.shape)).T
# Calcualte db2 (derivative of softmax)
# Back pass value of b2 is only one
db2 = 1 #np.ones(b2.shape)
# Calcualte dW1
dW1 = X.T @ relu_backward(np.ones(scores.shape) @ W2, np.dot(X, W1.T) + b1)
dW1 = dW1.T
# Calculate db1
db1 = np.mean(relu_backward(np.ones(scores.shape) @ W2, np.dot(X, W1.T) + b1).T, axis= 1)

#print("Finish Calculation")

grads['W1'] = dW1
grads['b1'] = db1
grads['W2'] = dW2
grads['b2'] = db2
```

        i.

c. Minibatch and learning rate:

i.

```
for it in np.arange(num_iters):
  X_batch = None
  y_batch = None

  # ================================================================ #
  # YOUR CODE HERE:
  #   Create a minibatch by sampling batch_size samples randomly.
  # ================================================================ #
  random_data_index = np.random.randint(num_train, size= batch_size)
  X_batch = X[random_data_index, :]
  y_batch = y[random_data_index]
  # ================================================================ #
  # END YOUR CODE HERE
  # ================================================================ #

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

  # ================================================================ #
  # YOUR CODE HERE:
  #   Perform a gradient descent step using the minibatch to update
  #   all parameters (i.e., W1, W2, b1, and b2).
  # ================================================================ #

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

d. Prediction:

i.

```
"""
y_pred = None

# ================================================================ #
# YOUR CODE HERE:
#   Predict the class given the input data.
# ================================================================ #
def relu(a):
  result = np.maximum(np.zeros(a.shape), a)
  return result
l1_output = X @ self.params['W1'].T + self.params['b1']
l1_relu = relu(l1_output)
l2_output = l1_relu @ self.params['W2'].T + self.params['b2']

y_pred = l2_output
```

ii.

2. FC

a. Affine Forward:

i.

```
# ================================================================ #
# YOUR CODE HERE:
#   Calculate the output of the forward pass.  Notice the dimensions
#   of w are D x M, which is the transpose of what we did in earlier
#   assignments.
# ================================================================ #
N, D = x.shape[0], w.shape[0]
x_reshape = x.reshape(N, D)
out = x_reshape @ w + b

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

cache = (x, w, b)
return out, cache
```

b. Affine Backward:

```
N, D = x.shape[0], w.shape[0]
x_reshape = x.reshape(N, D)
dx = (dout @ w.T).reshape(x.shape)
dw = x_reshape.T @ dout
db = np.mean(np.ones((b.shape[0], dout.shape[0])) @ dout, axis= 0)

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

i.

```
return dx, dw, db
```

c. RELU Forward:

```
"""
# ================================================================ #
# YOUR CODE HERE:
#   Implement the ReLU forward pass.
# ================================================================ #

def relu(a):
  result = a * (a > 0)
  return result

out = relu(x)
# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #

cache = x
return out, cache
```

i.

d. RELU Backward:

```
"""
x = cache

# ================================================================ #
# YOUR CODE HERE:
#   Implement the ReLU backward pass
# ================================================================ #

# ReLU directs linearly to those > 0


dx = dout * (x > 0)

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
```

i.

```
return dx
```