CS 189 Homework6

Xu Zhihao

August 7, 2019

I certify that all solutions are entirely in my own words and that I have not looked at another student's solutions. I have given credit to all external sources I consulted.

Signature: Zhihao Xu

Contents

1 Modular Fully-Connected Neural Networks

1.1 Layer Implementations

The affine transformation of the input here is $fc(x) = Wx + b$, Write the Loss function as L, $\frac{\partial L}{\partial f} = dout$, the derivative in the backward pass is

$$dw = \frac{\partial L}{\partial W} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial A} \cdot \frac{\partial A}{\partial W} = x^T \cdot \frac{\partial L}{\partial A}$$
$$dx = \frac{\partial L}{\partial x} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial A} \cdot \frac{\partial A}{\partial x} = \frac{\partial L}{\partial A} \cdot W^T$$
$$db = \frac{\partial L}{\partial b} = \frac{\partial L}{\partial f} \cdot \frac{\partial L}{\partial b} = \frac{\partial L}{\partial f} \cdot \mathbf{1}$$

where $\mathbf{1}$ is the all-one vector.

Kaggle Username: Jack_xzh

kaggle Score: 0.96580

1.1.1 Fully-Connected (fc) Layer

```
def affine_forward(x, w, b):
    out = None
    ###########################################################################
    # TODO: Implement the affine forward pass. Store the result in out. You #
    # will need to reshape the input into rows.                               #
    ###########################################################################
    x_ = x.reshape(x.shape[0], -1)
    out = x_.dot(w) + b

    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = (x, w, b)
    return out, cache


def affine_backward(dout, cache):
    x, w, b = cache
    dx, dw, db = None, None, None
    ###########################################################################
    # TODO: Implement the affine backward pass.                               #
    ###########################################################################
    # w
    # ----+
    # dw  \
    #      (*)----+
    # x    /      \
    # ----+        \   out
    # dx            (+)------
    #               /   dout
    # b            /
    # ------------+
    # db

    reshaped_x = np.reshape(x, (x.shape[0], -1))
    dx = np.reshape(dout.dot(w.T), x.shape)
    dw = (reshaped_x.T).dot(dout)
    db = np.sum(dout, axis=0)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx, dw, db
```

The output of running numerical gradient checking:

```
In [2]: # gradient checking: compare the analytical gradient with the numerical gradi
        # taking the affine layer as an example
        from gradient_check import eval_numerical_gradient_array
        import numpy as np
```

```
from tqdm import tqdm_notebook
from layers import *
N = 2
D = 3
M = 4
x = np.random.normal(size=(N, D))
w = np.random.normal(size=(D, M))
b = np.random.normal(size=(M, ))
dout = np.random.normal(size=(N, M))

out, cache = affine_forward(x, w, b)
f=lambda w: affine_forward(x, w, b)[0]
grad = affine_backward(dout, cache)[1]
ngrad = eval_numerical_gradient_array(f, w, dout)
print(grad)
print(ngrad)
# they should be similar enough within some small error tolerance
```

```
[[ 2.64528394 -2.96322162 -0.76223478  1.62866452]
 [ 1.15500095 -1.53884306 -0.24904242  1.28048743]
 [-4.53968039  4.51152205  1.50427081 -1.46168856]]
[[ 2.64528394 -2.96322162 -0.76223478  1.62866452]
 [ 1.15500095 -1.53884306 -0.24904242  1.28048743]
 [-4.53968039  4.51152205  1.50427081 -1.46168856]]
```

1.1.2 Activation Functions

Implement the Activation Functions ReLU following the instruction and two convenience layer functions:

```
def relu_forward(x):
    out = None
    ###########################################################################
    # TODO: Implement the ReLU forward pass.                                  #
    ###########################################################################
    out = np.maximum(0, x)
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    cache = x
    return out, cache


def relu_backward(dout, cache):
    dx, x = None, cache
    ###########################################################################
    # TODO: Implement the ReLU backward pass.                                 #
    ###########################################################################
    dx = np.array(dout, copy=True)
    dx[x <= 0] = 0
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return dx

# performs an affine transform followed by a ReLU
def affine_relu_forward(x, w, b):
    a, fc_cache = affine_forward(x, w, b)
    out, relu_cache = relu_forward(a)
    cache = (fc_cache, relu_cache)
    return out, cache

# performs a ReLU followed by an affine transform
def affine_relu_backward(dout, cache):
    fc_cache, relu_cache = cache
    da = relu_backward(dout, relu_cache)
    dx, dw, db = affine_backward(da, fc_cache)
    return dx, dw, db
```

Activation functions ReLU commonly have vanishing gradients. Since when $\sigma < 0$, the derivative of ReLu activation function equals 0. If the one-dimensional inputs are all smaller than 0, it would lead to this behavior.

The output of running numerical gradient checking:

```
In [19]: N = 2
         D = 3
         M = 4
```

```
        x = np.random.normal(size=(N, D))
        w = np.random.normal(size=(D, M))
        b = np.random.normal(size=(M, ))
        dout = np.random.normal(size=(N, D))

        out, cache = relu_forward(x)
        grad = relu_backward(dout, cache)
        ngrad = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)
        print(grad)
        print(ngrad)
```

```
[[ 0.6075281   0.         -0.0567495 ]
 [ 0.00940507 -0.20690946  0.        ]]
[[ 0.6075281   0.         -0.0567495 ]
 [ 0.00940507 -0.20690946  0.        ]]
```

1.1.3 Softmax Loss

Implement the softmax loss function following the instruction:

```
def softmax_loss(x, y):
    """
    Computes the loss and gradient for softmax classification.

    Inputs:
    - x: Input data, of shape (N, C) where x[i, j] is the score for the jth
      class for the ith input.
    - y: Vector of labels, of shape (N,) where y[i] is the label for x[i] and
      0 <= y[i] < C

    Returns a tuple of:
    - loss: Scalar giving the loss
    - dx: Gradient of the loss with respect to x
    """

    loss = 0.0
    dx = None
    ###########################################################################
    # TODO: Implement the softmax loss                                        #
    ###########################################################################
    logits = x - np.max(x, axis=1, keepdims=True)
    probs = np.exp(logits)
    probs /= np.sum(probs, axis=1, keepdims=True)
    N = x.shape[0]
    log_likelihood = -np.log(probs[np.arange(N), y])
    loss = np.sum(log_likelihood) / N

    dx = probs.copy()
    dx[np.arange(N), y] -= 1
    dx /= N
    ###########################################################################
    #                             END OF YOUR CODE                            #
    ###########################################################################
    return loss, dx
```

The output of running numerical gradient checking:

```
In [21]: N = 2
         D = 3
         M = 4
         x = np.random.normal(size=(N, D))
         w = np.random.normal(size=(D, M))
         b = np.random.normal(size=(M, ))
         dout = np.random.normal(size=(N, D))

         num_classes, num_inputs = 2,4
         x = 0.001 * np.random.normal(size=(num_inputs, num_classes))
         y = np.random.randint(num_classes, size=num_inputs)
```

```
        ngrad = eval_numerical_gradient_array(lambda x: softmax_loss(x, y)[0], x, 1)
        loss, grad = softmax_loss(x, y)
        print(grad)
        print(ngrad)
```

```
[[-0.12494204  0.12494204]
 [-0.12512408  0.12512408]
 [ 0.12497339 -0.12497339]
 [-0.12487521  0.12487521]]
[[-0.12494204  0.12494204]
 [-0.12512408  0.12512408]
 [ 0.12497339 -0.12497339]
 [-0.12487521  0.12487521]]
```

1.2 Two-layer Network

The implementation of **FullyConnectedNet** class

```
class FullyConnectedNet(object):
    def __init__(self, input_dim, hidden_dim=[10, 5], num_classes=10,
                 weight_scale=0.1):
        self.params = {}
        self.hidden_dim = hidden_dim
        self.num_layers = 1 + len(hidden_dim)
        ############################################################################
        # TODO: Initialize the weights and biases of the net. Weights              #
        # should be initialized from a Gaussian centered at 0.0 with               #
        # standard deviation equal to weight_scale, and biases should be           #
        # initialized to zero. All weights and biases should be stored in the      #
        # dictionary self.params, with first layer weights                         #
        # and biases using the keys 'W1' and 'b1' and second layer                 #
        # weights and biases using the keys 'W2' and 'b2'.                         #
        ############################################################################

        all_dims = [input_dim] + self.hidden_dim + [num_classes]

        L = self.num_layers+1
        for l in range(1, L):
            self.params['W' + str(l)] =
                np.random.normal(loc=0,scale=weight_scale,
                                                    size=all_dims[l-1]*all_dims[l]).\
                                    reshape(all_dims[l-1],all_dims[l])
            self.params['b' + str(l)] = np.zeros(all_dims[l])
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################


    def loss(self, X, y=None):
        scores = None
        ############################################################################
        # TODO: Implement the forward pass for the net, computing the              #
        # class scores for X and storing them in the scores variable.              #
        ############################################################################
        activation = X
        caches = []
        L = self.num_layers
        for l in range(1, L):
            activation, cache = affine_relu_forward(
                activation, self.params['W'+str(l)], self.params['b'+str(l)])
            caches.append(cache)

        scores, cache = affine_forward(
            activation, self.params['W'+str(L)],
                self.params['b'+str(self.num_layers)])
        caches.append(cache)
```

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

        # If y is None then we are in test mode so just return scores
        if y is None:
            return scores

        loss, grads = 0, {}
        ############################################################################
        # TODO: Implement the backward pass for the net. Store the loss    #
        # in the loss variable and gradients in the grads dictionary. Compute
            data #
        # loss using softmax, and make sure that grads[k] holds the gradients
            for #
        # self.params[k].                                                  #
        ############################################################################
        L = len(caches)
        loss, final_deriv = softmax_loss(scores, y)

        current_cache = caches[L-1]
        current_deriv = final_deriv
        current_deriv, dw, grads["b" +
                              str(L)] = affine_backward(current_deriv,
                                  current_cache)
        grads["W"+str(L)] = dw + self.params["W"+str(L)]

        for l in reversed(range(L-1)):
            current_cache = caches[l]
            dw, db = 0, 0
            current_deriv, dw, db = affine_relu_backward(
                current_deriv, current_cache)
            grads["W"+str(l+1)] = dw + self.params["W"+str(l+1)]
            grads["b"+str(l+1)] = db
        ############################################################################
        #                             END OF YOUR CODE                             #
        ############################################################################

        return loss, grads
```

Here is the implementation of my model, load my training and validation data, and use a Solver instance to train my model. I tried all the mapping of the hyperparameters listed in the code. The best model I get is **lr_decay** $= 0.95$, **num_epochs** $=$ 80, **batch_size** $= 8$, **learning_rate** $= 0.0001$, **weight_scale** $= 0.1$, **hidden_dims** $=$ $[200]$. The best validation accuracy I get is 0.9643.

```
In [3]: # Load the dataset
        import scipy.io
        import numpy as np
        data =   scipy.io.loadmat("mnist_data.mat")
        X = data['training_data']
        y = data['training_labels'].ravel()
        X_test = data['test_data']

        # Split the data into a training set and validation set.
        num_train = X.shape[0]
        indices = np.array(range(num_train))
        np.random.shuffle(indices)
        train_indices, val_indices = indices[0:50000], indices[50000:]
        X_train, X_val = X[train_indices], X[val_indices]
        y_train, y_val = y[train_indices], y[val_indices]

        from solver import Solver
        from classifiers.fc_net import FullyConnectedNet

In [4]: data = {
              'X_train': X_train,
              'y_train': y_train,
              'X_val': X_val,
              'y_val': y_val}

In [30]: # TODO: fill out the hyperparamets
         hyperparams = {'lr_decay': 0.95,
                        'num_epochs': 20,
                        'batch_size': 4,
                        'learning_rate': 0.0001,
                        'weight_scale': 0.1
                       }

         # TODO: fill out the number of units in your hidden layers
         hidden_dim = [250]  # this should be a list of units for each hiddent layer

         model = FullyConnectedNet(input_dim=784,
                                   hidden_dim=hidden_dim,
                                   weight_scale=hyperparams['weight_scale'])

         solver = Solver(model, data,
                         update_rule='sgd',
                         optim_config={
```

```
                        'learning_rate': hyperparams['learning_rate'],
                    },
                    lr_decay=hyperparams['lr_decay'],
                    num_epochs=hyperparams['num_epochs'],
                    batch_size=hyperparams['batch_size'],
                    print_every=10000)
        solver.train()
        solver.best_val_acc
```

```
(Iteration 1 / 250000) loss: 351.635005
(Epoch 0 / 20) train acc: 0.076000; val_acc: 0.072200
(Iteration 10001 / 250000) loss: 0.584828
(Epoch 1 / 20) train acc: 0.940000; val_acc: 0.926900
(Iteration 20001 / 250000) loss: 0.264482
(Epoch 2 / 20) train acc: 0.957000; val_acc: 0.947200
(Iteration 30001 / 250000) loss: 0.984741
(Epoch 3 / 20) train acc: 0.961000; val_acc: 0.949200
(Iteration 40001 / 250000) loss: 0.026470
(Epoch 4 / 20) train acc: 0.957000; val_acc: 0.947000
(Iteration 50001 / 250000) loss: 0.626866
(Iteration 60001 / 250000) loss: 0.122101
(Epoch 5 / 20) train acc: 0.963000; val_acc: 0.952200
(Iteration 70001 / 250000) loss: 0.167379
(Epoch 6 / 20) train acc: 0.959000; val_acc: 0.952200
(Iteration 80001 / 250000) loss: 0.136405
(Epoch 7 / 20) train acc: 0.959000; val_acc: 0.955400
(Iteration 90001 / 250000) loss: 0.889769
(Epoch 8 / 20) train acc: 0.969000; val_acc: 0.957600
(Iteration 100001 / 250000) loss: 0.379156
(Iteration 110001 / 250000) loss: 0.029785
(Epoch 9 / 20) train acc: 0.970000; val_acc: 0.957700
(Iteration 120001 / 250000) loss: 0.103715
(Epoch 10 / 20) train acc: 0.968000; val_acc: 0.958600
(Iteration 130001 / 250000) loss: 0.043163
(Epoch 11 / 20) train acc: 0.962000; val_acc: 0.957700
(Iteration 140001 / 250000) loss: 0.010857
(Epoch 12 / 20) train acc: 0.967000; val_acc: 0.960000
(Iteration 150001 / 250000) loss: 0.123178
(Iteration 160001 / 250000) loss: 0.583210
(Epoch 13 / 20) train acc: 0.971000; val_acc: 0.958000
(Iteration 170001 / 250000) loss: 0.024317
(Epoch 14 / 20) train acc: 0.963000; val_acc: 0.958500
(Iteration 180001 / 250000) loss: 0.111074
(Epoch 15 / 20) train acc: 0.965000; val_acc: 0.959300
(Iteration 190001 / 250000) loss: 0.157925
(Epoch 16 / 20) train acc: 0.965000; val_acc: 0.960300
(Iteration 200001 / 250000) loss: 0.125781
(Iteration 210001 / 250000) loss: 0.073907
(Epoch 17 / 20) train acc: 0.966000; val_acc: 0.958100
```

```
(Iteration 220001 / 250000) loss: 0.046422
(Epoch 18 / 20) train acc: 0.964000; val_acc: 0.957300
(Iteration 230001 / 250000) loss: 0.333958
(Epoch 19 / 20) train acc: 0.954000; val_acc: 0.958600
(Iteration 240001 / 250000) loss: 0.585628
(Epoch 20 / 20) train acc: 0.955000; val_acc: 0.960500
```

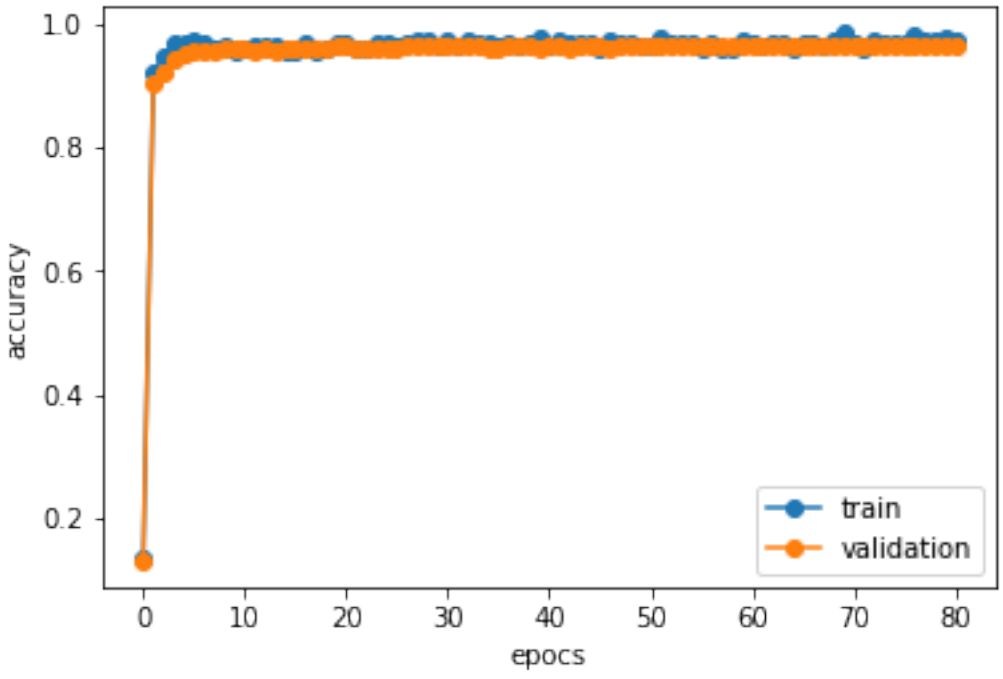
Out[30]: 0.9605

```
In [6]: def train_model(paras):
            hyperparams = {'lr_decay': paras[1],
                           'num_epochs': paras[2],
                           'batch_size': paras[3],
                           'learning_rate': paras[4],
                           'weight_scale':paras[5]
                          }
            hidden_dim = paras[0]

            model = FullyConnectedNet(input_dim=784,
                                      hidden_dim=hidden_dim,
                                      weight_scale=hyperparams['weight_scale'])
            solver = Solver(model, data,
                            update_rule='sgd',
                            optim_config={
                              'learning_rate': hyperparams['learning_rate'],
                            },
                            lr_decay=hyperparams['lr_decay'],
                            num_epochs=hyperparams['num_epochs'],
                            batch_size=hyperparams['batch_size'],
                            verbose=False
                            #print_every=100
                           )

            solver.train()
            return solver,model

In [15]: import itertools
         best_val = 0.0
         best_model = None
         best_para = None
         best_solver = None

         lr_decay = [0.95]
         num_epochs = [30,50,80]
         batch_size = [2,4,8]
         learning_rate = [0.0001]
         weight_scale = [0.1]
         hidden_dims = [[h] for h in [100,150,200]]
```

```
        paras = [i for i in itertools.product\
        (hidden_dims,lr_decay,num_epochs,batch_size,learning_rate,weight_scale)]
        total = str(len(paras))

        for para in tqdm_notebook(paras):
            solver,model = train_model(para)
            if solver.best_val_acc > best_val:
                best_val = solver.best_val_acc
                best_model = model
                best_solver = solver
                best_para = para
```

```
HBox(children=(IntProgress(value=0, max=27), HTML(value='')))
```

```
In [16]: from matplotlib import pyplot as plt
         plt.plot(best_solver.train_acc_history, '-o', label="train")
         plt.plot(best_solver.val_acc_history, '-o', label="validation")
         plt.xlabel("epocs")
         plt.ylabel("accuracy")
         plt.legend()
         plt.show()
         print("hyperparameters of the best model: "+str(best_para));
         print("Validation accuracy of the best model:",\
               best_solver.best_val_acc)
```



```
hyperparameters of the best model: ([200], 0.95, 80, 8, 0.0001, 0.1)
Validation accuracy of the best model: 0.9643
```

```
In [17]: y_test_pred = np.argmax(best_model.loss(X_test), axis=1)
```

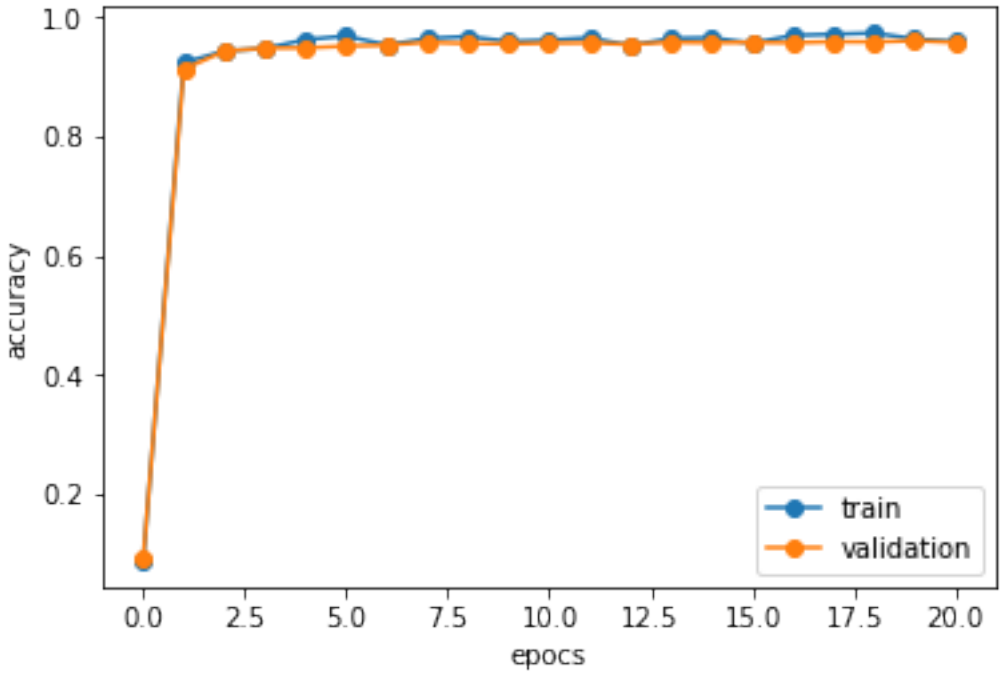
```
In [18]: results_to_csv(y_test_pred,"MNIST")
```

1.3 Multi-layer Network

In [22]:
```
best_val = 0.0
best_model = None
best_para = None
best_solver = None

lr_decay = [0.95]
num_epochs = [20]
batch_size = [4,8]
learning_rate = [0.0001]
weight_scale = [0.1]
hidden_dims = [[200]*i for i in range(1,5)]

prog = 1
paras = [i for i in itertools.product\
(hidden_dims,lr_decay,num_epochs,batch_size,learning_rate,weight_scale)]
total = str(len(paras))

for para in tqdm_notebook(paras):
    solver,model = train_model(para)
    avg_val = np.mean(solver.val_acc_history)
    if avg_val > best_val:
        best_val = avg_val
        best_model = model
        best_solver = solver
        best_para = para
    # print("\rfinished training model # "+str(prog)+"/"+total,end="")
    prog += 1
```

```
HBox(children=(IntProgress(value=0, max=8), HTML(value='')))
```

In [24]:
```
from matplotlib import pyplot as plt
plt.plot(best_solver.train_acc_history, '-o', label="train")
plt.plot(best_solver.val_acc_history, '-o', label="validation")
plt.xlabel("epocs")
plt.ylabel("accuracy")
plt.legend()
plt.show()
print("hyperparameters of the best model: "+str(best_para));
print("Validation accuracy of the best model:",\
      best_solver.best_val_acc)
```




```
hyperparameters of the best model: ([200], 0.95, 20, 4, 0.0001, 0.1)
Validation accuracy of the best model: 0.961
```

```
In [ ]: y_test_pred = np.argmax(best_model.loss(X_test), axis=1)
        results_to_csv(y_test_pred,"MNIST")
```

I tried the 1 hidden layer to 4 hidden layers, other hyperparameters keep the same as previous best model except **num_epochs** decreases to 20 to decrease the running time. However the best model I get is still the two layers network.

2 Convolution and Backprop Revisited

(a) Choose mask G[] is

$$G[t] = \begin{cases} \frac{1}{2}, & t = -1 \\ -\frac{1}{2}, & t = 1 \\ 0, & \text{Otherwise} \end{cases}$$

We can see that

$$\begin{aligned} (I * G)[t] &= \sum_{k=-\infty}^{\infty} I[k]G[t-k] \\ &= I[t-1]G[t-(t-1)] + I[t+1]G[t-(t+1)] \\ &= I[t-1]G[1] + I[t+1]G[-1] \\ &= \frac{I[t+1] - I[t-1]}{2} \\ &\approx I'[t] \end{aligned}$$

(b) The pseudocode of computing the convolution of an image I with a set of masks G and a stride of s.

Algorithm 1 Compute I

```
for j from 0 to H-1 by step s do
  for i from 0 to W-1 by step s do
    I' ← I[i, i+1, ..., i+w-1; j, j+1, ..., j+h-1]
    R[i, j] ← sum(I'*G)
  end for
end for
return R
```

Notation:

(1) * represent the element-wise multiplication of two matrices (which is not the same as matrix multiplication)

(2) sum() represent summing all the elements in the matrix.

(3) The submatrix of $A \in \mathbb{R}^{m \times n}$ is denoted as follows.
Let:

$\{a_1, a_2, \ldots, a_r\}$ be the indices of the r selected rows
$\{b_1, b_2, \ldots, b_s\}$ be the indices of the s selected columns

where all of $a_1, a_2, \ldots, a_r$ are between 1 and m, and all of $b_1, b_2, \ldots, b_s$ are between 1 and n. Then the submatrix formed from rows $\{a_1, a_2, \ldots, a_r\}$ and columns $\{b_1, b_2, \ldots, b_s\}$ is denoted as:

$$A[a_1, a_2, \ldots, a_r; b_1, b_2, \ldots, b_s]$$

(c) Using Prewitt operator,

$$G = G_x = \begin{bmatrix} +1 & 0 & -1 \\ +1 & 0 & -1 \\ +1 & 0 & -1 \end{bmatrix}$$

(d)

$$
\begin{aligned}
\frac{\partial L}{\partial G_c[x,y]} &= \frac{\partial L}{\partial R[x,y]} \frac{\partial R[x,y]}{\partial G_c[x,y]} \\
&= \frac{\partial L}{\partial R[x,y]} \frac{\partial (I * G)[x,y]}{\partial G_c[x,y]} \\
&= \frac{\partial L}{\partial R[x,y]} \frac{\partial}{\partial G_c[x,y]} \left\{ \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} \sum_{d \in \{r,g,b\}} I_d[x+i, y+j] G_d[i,j] \right\} \\
&= \frac{\partial L}{\partial R[x,y]} \frac{\partial}{\partial G_c[x,y]} \left\{ \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} I_c[x+i, y+j] G_c[i,j] \right\} \\
&= \frac{\partial L}{\partial R[x,y]} \frac{\partial}{\partial G_c[x,y]} \{ I_c[2x, 2y] G_c[x,y] \} \\
&= I_c[2x, 2y] \frac{\partial L}{\partial R[x,y]} \\
\frac{\partial L}{\partial I_c[x,y]} &= \frac{\partial L}{\partial R[x,y]} \frac{\partial R[x,y]}{\partial I_c[x,y]} \\
&= \frac{\partial L}{\partial R[x,y]} \frac{\partial (I * G)[x,y]}{\partial I_c[x,y]} \\
&= \frac{\partial L}{\partial R[x,y]} \frac{\partial}{\partial I_c[x,y]} \left\{ \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} \sum_{d \in \{r,g,b\}} I_d[x+i, y+j] G_d[i,j] \right\} \\
&= \frac{\partial L}{\partial R[x,y]} \frac{\partial}{\partial I_c[x,y]} \left\{ \sum_{i=0}^{w-1} \sum_{j=0}^{h-1} I_c[x+i, y+j] G_c[i,j] \right\} \\
&= \frac{\partial L}{\partial R[x,y]} \frac{\partial}{\partial I_c[x,y]} \{ I_c[x,y] G_c[0,0] \} \\
&= G_c[0,0] \frac{\partial L}{\partial R[x,y]}
\end{aligned}
$$

(e) First compute the top left entry $[i_l, j_l]$,

$$i_l = (i-1)s + 1$$
$$j_l = (j-1)s + 1$$

Then compute the right button entry $[i_r, j_r]$,

$$i_r = i_l + w - 1 = (i-1)s + w$$
$$j_r = j_l + h - 1 = (j-1)s + h$$

Hence, by the max pooling method,

R[i,j] =

$$\begin{bmatrix} \max I[(i-1)s+1, (j-1)s+1] & \max I[(i-1)s+1, (j-1)s+2] & \cdots & \max I[(i-1)s+1, (j-1)s+h] \\ \max I[(i-1)s+2, (j-1)s+1] & \max I[(i-1)s+2, (j-1)s+2] & \cdots & \max I[(i-1)s+2, (j-1)s+h] \\ \ddots & \ddots & \vdots & \ddots \\ \max I[(i-1)s+w, (j-1)s+1] & \max I[(i-1)s+w, (j-1)s+2] & \cdots & \max I[(i-1)s+w, (j-1)s+h] \end{bmatrix}$$

(f) The gradient w.r.t non-maximum values is 0. Since if we change these values slightly, it will not affect the maximum value. The gradient w.r.t maximum values is 1. Since the relation between gradient and coefficient is linear with coefficient 1. In the backprop algorithm, the derivative only passes to the maximum value neuron and others get 0.