Ryan Shaw
ryshaw

1.

```
def f3(s1, s2, y):

    assert y == 1 or y == 2
    # Forward pass: Compute loss
    L = None

    e1 = math.exp(s1)
    e2 = math.exp(s2)
    d = e1 + e2
    p1 = e1 / d
    p2 = e2 / d
    if y == 1:
        L = -math.log(p1)
    else:
        L = -math.log(p2)

    # Backward pass: Compute gradients
    grad_s1, grad_s2, grad_L = None, None, 1.0
    grad_p1, grad_p2 = None, None
    grad_e1, grad_e2 = None, None

    # Compute dL/dp1 and dL/dp2
    if y == 1:
        dp1 = -1 / p1
        dp2 = 0
    else:
        dp1 = 0
        dp2 = -1 / p2

    # Compute dp1/ds1, dp1/ds2, dp2/ds1, and dp2/ds2
    e1d = e1 / d
    e2d = e2 / d
    dp1ds1 = e1d * (1 - e1d)
    dp1ds2 = -e1d * e2d
    dp2ds1 = -e2d * e1d
    dp2ds2 = e2d * (1 - e2d)
```

```python
# Compute dL/ds1 and dL/ds2 using the chain rule
grad_s1 = dp1 * dp1ds1 + dp2 * dp2ds1
grad_s2 = dp1 * dp1ds2 + dp2 * dp2ds2

grads = (grad_s1, grad_s2)
return L, grads
```

2.

(Fc-forward)

```python
"""
out = None
###########################################################################
# TODO: Implement the forward pass. Store the result in out.              #
###########################################################################
out = np.dot(x, w) + b
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
cache = (x, w, b)
return out, cache
```

(fc-backward)

```python
x, w, b = cache
grad_x, grad_w, grad_b = None, None, None
###########################################################################
# TODO: Implement the backward pass for the fully-connected layer         #
###########################################################################
grad_x = np.dot(grad_out, w.T)
grad_w = np.dot(x.T, grad_out)
grad_b = np.sum(grad_out, axis=0)
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
return grad_x, grad_w, grad_b
```

(relu-forward)

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

(relu-backward)

```python
grad_x, x = None, cache
###########################################################################
# TODO: Implement the ReLU backward pass.                                 #
###########################################################################
grad_x = grad_out
grad_x[x <= 0] = 0
###########################################################################
#                              END OF YOUR CODE                           #
###########################################################################
return grad_x
```

(softmax loss)

```python
loss, grad_x = None, None
###########################################################################
# TODO: Implement softmax loss                                            #
###########################################################################
# Shift x by max value to avoid numerical instability
x_max = np.max(x, axis=1, keepdims=True)
x_exp = np.exp(x - x_max)
x_sum = np.sum(x_exp, axis=1, keepdims=True)

# Compute softmax probabilities
p = x_exp / x_sum

# Compute loss
n = x.shape[0]
loss = -np.sum(np.log(p[np.arange(n), y])) / n

# Compute gradient of loss with respect to scores
grad_x = p.copy()
grad_x[np.arange(n), y] -= 1
grad_x /= n

###########################################################################
#                              END OF YOUR CODE                           #
###########################################################################
return loss, grad_x
```

(l2-regularization)

```python
loss, grad_w = None, None
###########################################################################
# TODO: Implement L2 regularization.                                      #
###########################################################################
loss = 0.5 * reg * np.sum(w * w)
grad_w = reg * w
###########################################################################
#                              END OF YOUR CODE                           #
###########################################################################
return loss, grad_w
```

3. (two-layer net)

(init)

```
"""
############################################################################
# TODO: Initialize the weights and biases of a two-layer network.      #
############################################################################
self.params = {}
self.params['W1'] = weight_scale * np.random.randn(input_dim, hidden_dim)
self.params['b1'] = np.zeros(hidden_dim)
self.params['W2'] = weight_scale * np.random.randn(hidden_dim, num_classes)
self.params['b2'] = np.zeros(num_classes)
############################################################################
#                             END OF YOUR CODE                             #
############################################################################
```

(params)

```
def parameters(self):
    params = None
    ############################################################################
    # TODO: Build a dict of all learnable parameters of this model.        #
    ############################################################################
    params = self.params
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################
    return params
```

(forward)

```
def forward(self, X):
    scores, cache = None, {}
    ############################################################################
    # TODO: Implement the forward pass to compute classification scores    #
    # for the input data X. Store into cache any data that will be needed #
    # during the backward pass.                                            #
    ############################################################################
    W1, b1 = self.params['W1'], self.params['b1']
    W2, b2 = self.params['W2'], self.params['b2']
    out1, cache1 = fc_forward(X, W1, b1)
    out2, cache2 = relu_forward(out1)
    out3, cache3 = fc_forward(out2, W2, b2)
    scores = out3
    cache = (cache1, cache2, cache3)
    ############################################################################
    #                             END OF YOUR CODE                             #
    ############################################################################
    return scores, cache
```

(backward)

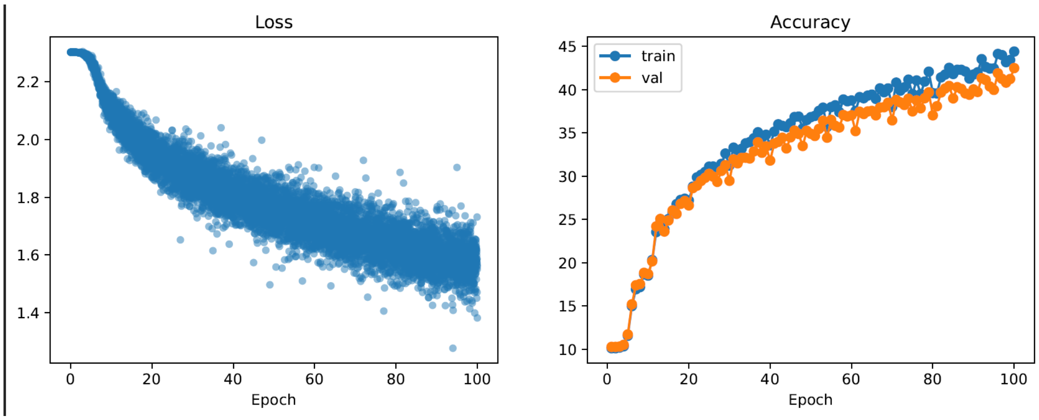
```python
grads = {}
############################################################################
# TODO: Implement the backward pass to compute gradients for all          #
# learnable parameters of the model, storing them in the grads dict       #
# above. The grads dict should give gradients for all parameters in       #
# the dict returned by model.parameters().                                 #
############################################################################
cache1, cache2, cache3 = cache
dX3, dW2, db2 = fc_backward(grad_scores, cache3)
dX2 = relu_backward(dX3, cache2)
dX1, dW1, db1 = fc_backward(dX2, cache1)
grads = {'W1': dW1, 'b1': db1, 'W2': dW2, 'b2': db2}
############################################################################
#                             END OF YOUR CODE                             #
############################################################################
return grads
```

4.

a. (training-step)

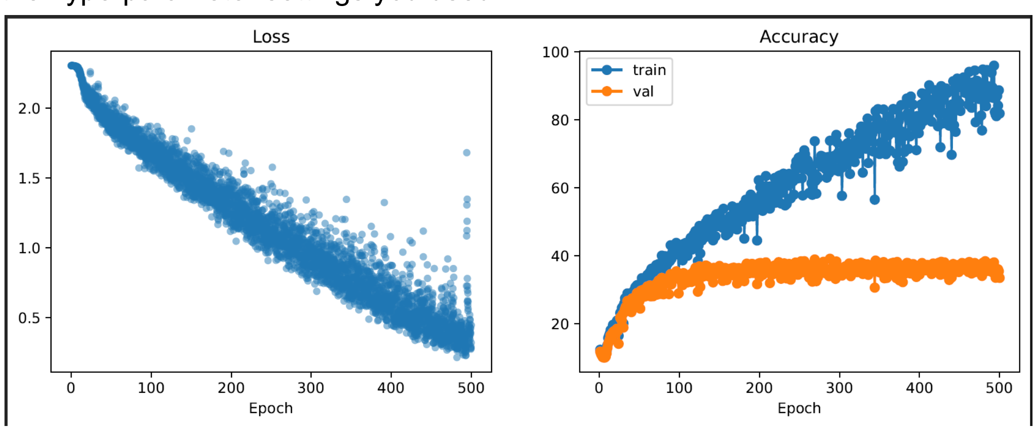
```python
loss, grads = None, None
###########################################################################
# TODO: Compute the loss and gradient for one training iteration.         #
###########################################################################
#forward pass
scores, cache = model.forward(X_batch)

#calculate loss
loss, dloss = softmax_loss(scores, y_batch)
loss1, dloss1 = l2_regularization(model.parameters()['W1'], reg)
loss2, dloss2 = l2_regularization(model.parameters()['W2'], reg)
loss += loss1 + loss2

#backward pass
grads = model.backward(dloss, cache)
grads['W1'] += dloss1
grads['W2'] += dloss2
###########################################################################
#                             END OF YOUR CODE                            #
###########################################################################
return loss, grads
```

b. In your report, include the loss / accuracy plot for your best model, describe the hyperparameter settings you used, and give the final test-set performance of your model.




```python
# How much data to use for training
num_train = 20000

# Model architecture hyperparameters.
hidden_dim = 300

# Optimization hyperparameters.
batch_size = 256
num_epochs = 100#500
learning_rate = 1e-2#6e-3
reg = 1e-6
```

c. In your report, include the loss / accuracy plot for your overfit model and describe the hyperparameter settings you used.




```python
# How much data to use for training
num_train = 2024

# Model architecture hyperparameters
hidden_dim = 256

# Optimization hyperparameters.
batch_size = 256
num_epochs = 500#500
learning_rate = 5e-2#6e-3
reg = 0
```

5.

(architecture)

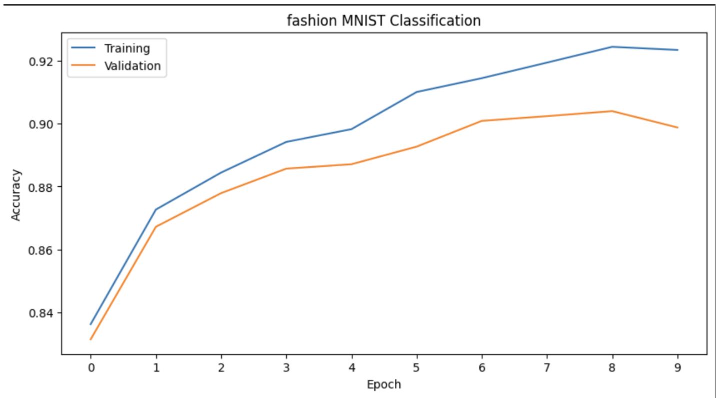
```
#downsampling
self.conv1 = nn.Conv2d(1,64,kernel_size=3, stride=2, padding=1) #14x14x64
self.pool = nn.AvgPool2d(kernel_size=2, stride=2) #7x7x64

#classifier
self.conv2 = nn.Conv2d(64,64,kernel_size=3, stride=1, padding=1) #7x7x64
self.conv3 = nn.Conv2d(64,16,kernel_size=3, stride=2, padding=1) #4x4x16 (stride=2 make 7->4, 64->16)
self.lin = nn.Linear(256,10)

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

f forward(self, x):
###############################################################################
# TODO: Design your own network, implement forward pass here                  #
###############################################################################
x = x.to(device)
relu = nn.LeakyReLU()

# Flatten each image in the batch
x = self.conv1(x)
x = relu(x)
x = self.pool(x)

x = self.conv2(x)
x = relu(x)
# # No need to define self.relu because it contains no parameters

x = self.conv3(x)
x = relu(x)

x = x.view(x.size(0), -1)
x = self.lin(x)
# The loss layer will be applied outside Network class
return x
```

(hyperparameters)

```
####################
batch_size=256
####################
```

```
# Set up optimization hyperparameters
learning_rate = 3e-3
weight_decay = 1e-5
num_epoch = 10  # TODO: Choose an appropriat
optimizer = optim.Adam(model.parameters(), l
                       weight_decay=weight_d
```

(plot)




(test set)

```
 Evaluate on test set
100%|██████████| 40/40 [00:02<00:00, 16.35it/s]
 Evaluation accuracy: 0.8961
CPU times: user 4min 56s, sys: 2.22 s, total: 4min 58s
Wall time: 5min 2s

0.8961
```

6.

(top image: good prediction; bottom: bad)

Predicted Class: Chihuahua

Predicted Class: sorrel