Problem 1: Basics of Neural Networks

- **Learning Objective:** In this problem, you are asked to implement a basic multi-layer fully connected neural network from scratch, including forward and backward passes of certain essential layers, to perform an image classification task on the SVHN dataset. You need to implement essential functions in different indicated python files under directory `lib` .
- **Provided Code:** We provide the skeletons of classes you need to complete. Forward checking and gradient checkings are provided for verifying your implementation as well.
- **TODOs:** You are asked to implement the forward passes and backward passes for standard layers and loss functions, various widely-used optimizers, and part of the training procedure. And finally we want you to train a network from scratch on your own. Also, there are inline questions you need to answer. See `README.md` to set up your environment.

In [ ]:
```python
from lib.mlp.fully_conn import *
from lib.mlp.layer_utils import *
from lib.mlp.datasets import *
from lib.mlp.train import *
from lib.grad_check import *
from lib.optim import *
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of
plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/autoreload-of-mod
ules-in-ipython
%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

Loading the data (SVHN)

Run the following code block to download SVHN dataset and load in the properly splitted SVHN data. The script `get_datasets.sh` use `wget` to download the SVHN dataset. If you have a trouble with executing `get_datasets.sh` , you can manually download the dataset and extract files.

In [ ]:
```
!./get_datasets.sh
# !get_datasets.sh for windows users
```

```
--2022-02-19 15:32:53--  http://ufldl.stanford.edu/housenumbers/tr
ain_32x32.mat
Resolving ufldl.stanford.edu (ufldl.stanford.edu)... 171.64.68.10
Connecting to ufldl.stanford.edu (ufldl.stanford.edu)|171.64.68.1
0|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 182040794 (174M) [text/plain]
Saving to: ‘data/train_32x32.mat.5’

train_32x32.mat.5   100%[===================>] 173.61M   720KB/s
in 4m 9s

2022-02-19 15:37:02 (713 KB/s) - ‘data/train_32x32.mat.5’ saved [1
82040794/182040794]

--2022-02-19 15:37:02--  http://ufldl.stanford.edu/housenumbers/te
st_32x32.mat
Resolving ufldl.stanford.edu (ufldl.stanford.edu)... 171.64.68.10
Connecting to ufldl.stanford.edu (ufldl.stanford.edu)|171.64.68.1
0|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 64275384 (61M) [text/plain]
Saving to: ‘data/test_32x32.mat.5’

test_32x32.mat.5    100%[===================>]  61.30M  1.08MB/s
in 81s

2022-02-19 15:38:23 (775 KB/s) - ‘data/test_32x32.mat.5’ saved [64
275384/64275384]
```

Load the dataset.

In [ ]:
```
data = SVHN_data()
for k, v in data.items():
    print ("Name: {} Shape: {}".format(k, v.shape))
```

```
Name: data_train Shape: (70000, 32, 32, 3)
Name: labels_train Shape: (70000,)
Name: data_val Shape: (3257, 32, 32, 3)
Name: labels_val Shape: (3257,)
Name: data_test Shape: (26032, 32, 32, 3)
Name: labels_test Shape: (26032,)
```

Implement Standard Layers

You will now implement all the following standard layers commonly seen in a fully connected neural network (aka multi-layer perceptron, MLP). Please refer to the file `lib/mlp/layer_utils.py` . Take a look at each class skeleton, and we will walk you through the network layer by layer. We provide results of some examples we pre-computed for you for checking the forward pass, and also the gradient checking for the backward pass.

FC Forward [2pt]

In the class skeleton `flatten` and `fc` in `lib/mlp/layer_utils.py` , please complete the forward pass in function `forward` . The input to the `fc` layer may not be of dimension (batch size, features size), it could be an image or any higher dimensional data. We want to convert the input to have a shape of (batch size, features size). Make sure that you handle this dimensionality issue.

In [ ]:
```
%reload_ext autoreload

# Test the fc forward function
input_bz = 3 # batch size
input_dim = (7, 6, 4)
output_dim = 4

input_size = input_bz * np.prod(input_dim)
weight_size = output_dim * np.prod(input_dim)

flatten_layer = flatten(name="flatten_test")
single_fc = fc(np.prod(input_dim), output_dim, init_scale=0.02, na
me="fc_test")

x = np.linspace(-0.1, 0.4, num=input_size).reshape(input_bz, *inpu
t_dim)
w = np.linspace(-0.2, 0.2, num=weight_size).reshape(np.prod(input_
dim), output_dim)
b = np.linspace(-0.3, 0.3, num=output_dim)

single_fc.params[single_fc.w_name] = w
single_fc.params[single_fc.b_name] = b

out = single_fc.forward(flatten_layer.forward(x))

correct_out = np.array([[0.63910291, 0.83740057, 1.03569824, 1.233
99591],
                        [0.61401587, 0.82903823, 1.04406058, 1.259
08294],
                        [0.58892884, 0.82067589, 1.05242293, 1.284
16997]])

# Compare your output with the above pre-computed ones.
# The difference should not be larger than 1e-8
print ("Difference: ", rel_error(out, correct_out))
```

```
Difference:  4.02601593296122e-09
```

FC Backward [2pt]

Please complete the function `backward` as the backward pass of the `flatten` and `fc` layers. Follow the instructions in the comments to store gradients into the predefined dictionaries in the attributes of the class. Parameters of the layer are also stored in the predefined dictionary.

```
In [ ]: %reload_ext autoreload

        # Test the fc backward function
        inp = np.random.randn(15, 2, 2, 3)
        w = np.random.randn(12, 15)
        b = np.random.randn(15)
        dout = np.random.randn(15, 15)

        flatten_layer = flatten(name="flatten_test")
        x = flatten_layer.forward(inp)

        single_fc = fc(np.prod(x.shape[1:]), 15, init_scale=5e-2, name="fc
        _test")
        single_fc.params[single_fc.w_name] = w
        single_fc.params[single_fc.b_name] = b

        dx_num = eval_numerical_gradient_array(lambda x: single_fc.forward
        (x), x, dout)
        dw_num = eval_numerical_gradient_array(lambda w: single_fc.forward
        (x), w, dout)
        db_num = eval_numerical_gradient_array(lambda b: single_fc.forward
        (x), b, dout)

        out = single_fc.forward(x)
        dx = single_fc.backward(dout)
        dw = single_fc.grads[single_fc.w_name]
        db = single_fc.grads[single_fc.b_name]
        dinp = flatten_layer.backward(dx)

        # The error should be around 1e-9
        print("dx Error: ", rel_error(dx_num, dx))
        # The errors should be around 1e-10
        print("dw Error: ", rel_error(dw_num, dw))
        print("db Error: ", rel_error(db_num, db))
        # The shapes should be same
        print("dinp Shape: ", dinp.shape, inp.shape)
```

```
dx Error:  1.2994445976166083e-09
dw Error:  3.2910185983553206e-09
db Error:  1.961733773414855e-10
dinp Shape:  (15, 2, 2, 3) (15, 2, 2, 3)
```

Leaky ReLU Forward [2pt]

In the class skeleton `leaky_relu` in `lib/mlp/layer_utils.py` , please complete the `forward` pass.

A "leaky" ReLU is similar to a ReLU, but rather than zero-ing out features valued less than 0, they are multiplied by a constant value less than 1.

$$LeakyReLU(x) = \begin{cases} x & x \geq 0 \\ c * x & x < 0 \end{cases}, \text{ where } 0 \leq c < 1$$

When $c = 0$, a Leaky ReLU is equivalent to a standard ReLU.

```
In [ ]: %reload_ext autoreload

        # Test the leaky_relu forward function
        x = np.linspace(-1.5, 1.5, num=12).reshape(3, 4)
        lrelu_f = leaky_relu(negative_slope=0.01, name="leaky_relu_f")

        out = lrelu_f.forward(x)

        correct_out = np.array([[-0.015,        -0.0122727273, -0.00954545
        45, -0.0068181818],
                                [-0.0040909091, -0.0013636364,  0.13636363
        64,  0.4090909091],
                                [ 0.6818181818,  0.9545454545,  1.22727272
        73,  1.5         ]])

        # Compare your output with the above pre-computed ones.
        # The difference should not be larger than 1e-7
        print ("Difference: ", rel_error(out, correct_out))
```

```
Difference:  1.3333332805929594e-08
```

Leaky ReLU Backward [2pt]

Please complete the `backward` pass of the class `leaky_relu` .

```
In [ ]: %reload_ext autoreload

        # Test the relu backward function
        x = np.random.randn(15, 15)
        dout = np.random.randn(*x.shape)
        lrelu_b = leaky_relu(negative_slope=0.01, name="leaky_relu_b")

        dx_num = eval_numerical_gradient_array(lambda x: lrelu_b.forward
        (x), x, dout)

        out = lrelu_b.forward(x)
        dx = lrelu_b.backward(dout)

        # The error should not be larger than 1e-10
        print ("dx Error: ", rel_error(dx_num, dx))
```

```
dx Error:  5.704260604188748e-12
```

Dropout Forward [2pt]

In the class `dropout` in `lib/mlp/layer_utils.py` , please complete the `forward` pass. Remember that the dropout is **only applied during training phase**, you should pay attention to this while implementing the function.

Important Note1: The probability argument input to the function is the "keep probability": probability that each activation is kept.

Important Note2: If the keep_prob is set to 0, make it as no dropout.

In [ ]:
```
%reload_ext autoreload

# x = np.random.randn(100, 100) + 5.0
x = np.random.randn(10, 10) + 5.0

print
("-------------------------------------------------------------------
")
for p in [0, 0.25, 0.50, 0.75, 1]:
    dropout_f = dropout(keep_prob=p)
    out = dropout_f.forward(x, True)
    out_test = dropout_f.forward(x, False)

    # Mean of output should be similar to mean of input
    # Means of output during training time and testing time should
be similar
    print ("Dropout Keep Prob = ", p)
    print ("Mean of input: ", x.mean())
    print ("Mean of output during training time: ", out.mean())
    print ("Mean of output during testing time: ", out_test.mean
())
    print ("Fraction of output set to zero during training time:
", (out == 0).mean())
    print ("Fraction of output set to zero during testing time: ",
(out_test == 0).mean())
    print
("-------------------------------------------------------------------
")
```

```
----------------------------------------------------------------
Dropout Keep Prob =  0
Mean of input:  5.038117277530198
Mean of output during training time:  5.038117277530198
Mean of output during testing time:  5.038117277530198
Fraction of output set to zero during training time:  0.0
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
Dropout Keep Prob =  0.25
Mean of input:  5.038117277530198
Mean of output during training time:  5.519216184043529
Mean of output during testing time:  5.038117277530198
Fraction of output set to zero during training time:  0.74
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
Dropout Keep Prob =  0.5
Mean of input:  5.038117277530198
Mean of output during training time:  5.507070169769443
Mean of output during testing time:  5.038117277530198
Fraction of output set to zero during training time:  0.46
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
Dropout Keep Prob =  0.75
Mean of input:  5.038117277530198
Mean of output during training time:  4.917805210236621
Mean of output during testing time:  5.038117277530198
Fraction of output set to zero during training time:  0.26
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
Dropout Keep Prob =  1
Mean of input:  5.038117277530198
Mean of output during training time:  5.038117277530198
Mean of output during testing time:  5.038117277530198
Fraction of output set to zero during training time:  0.0
Fraction of output set to zero during testing time:  0.0
----------------------------------------------------------------
```

Dropout Backward [2pt]

Please complete the `backward` pass. Again remember that the dropout is only applied during training phase, handle this in the backward pass as well.

```
In [ ]: %reload_ext autoreload

        x = np.random.randn(5, 5) + 5
        dout = np.random.randn(*x.shape)

        keep_prob = 0
        dropout_b = dropout(keep_prob, seed=100)
        out = dropout_b.forward(x, True, seed=1)
        dx = dropout_b.backward(dout)

        dx_num = eval_numerical_gradient_array(lambda xx: dropout_b.forwar
        d(xx, True, seed=1), x, dout)

        # The error should not be larger than 1e-10
        print ('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  1.892895303821259e-11
```

Testing cascaded layers: FC + Leaky ReLU [2pt]

Please find the `TestFCReLU` function in `lib/mlp/fully_conn.py` .
You only need to complete a few lines of code in the TODO block.
Please design an `Flatten -> FC -> Leaky ReLU` network where the parameters of them match the given x, w, and b.
Please insert the corresponding names you defined for each layer to param_name_w, and param_name_b respectively. Here you only modify the param_name part, the `_w` , and `_b` are automatically assigned during network setup

```
In [ ]: %reload_ext autoreload

x = np.random.randn(3, 5, 3)  # the input features
w = np.random.randn(15, 5)   # the weight of fc layer
b = np.random.randn(5)       # the bias of fc layer
dout = np.random.randn(3, 5) # the gradients to the output, notice
the shape

tiny_net = TestFCReLU()

###################################################
# TODO: param_name should be replaced accordingly #
###################################################
tiny_net.net.assign("tiny_fc_w", w)
tiny_net.net.assign("tiny_fc_b", b)
###################################################
#                END OF YOUR CODE                 #
###################################################

out = tiny_net.forward(x)
dx = tiny_net.backward(dout)

###################################################
# TODO: param_name should be replaced accordingly #
###################################################
dw = tiny_net.net.get_grads("tiny_fc_w")
db = tiny_net.net.get_grads("tiny_fc_b")
###################################################
#                END OF YOUR CODE                 #
###################################################

dx_num = eval_numerical_gradient_array(lambda x: tiny_net.forward
(x), x, dout)
dw_num = eval_numerical_gradient_array(lambda w: tiny_net.forward
(x), w, dout)
db_num = eval_numerical_gradient_array(lambda b: tiny_net.forward
(x), b, dout)

# The errors should not be larger than 1e-7
print ("dx error: ", rel_error(dx_num, dx))
print ("dw error: ", rel_error(dw_num, dw))
print ("db error: ", rel_error(db_num, db))
```

```
dx error:  2.2967180926914487e-10
dw error:  2.1929993941836986e-10
db error:  7.878626722629195e-12
```

SoftMax Function and Loss Layer [2pt]

In the `lib/mlp/layer_utils.py` , please first complete the function `softmax` , which will be used in the function `cross_entropy` . Then, implement `corss_entropy` using `softmax` . Please refer to the lecture slides of the mathematical expressions of the cross entropy loss function, and complete its forward pass and backward pass.

```
In [ ]: %reload_ext autoreload

        num_classes, num_inputs = 6, 100
        x = 0.001 * np.random.randn(num_inputs, num_classes)
        y = np.random.randint(num_classes, size=num_inputs)

        test_loss = cross_entropy()

        dx_num = eval_numerical_gradient(lambda x: test_loss.forward(x,
        y), x, verbose=False)

        loss = test_loss.forward(x, y)
        dx = test_loss.backward()

        # Test softmax_loss function. Loss should be around 1.792
        # and dx error should be at the scale of 1e-8 (or smaller)
        print ("Cross Entropy Loss: ", loss)
        print ("dx error: ", rel_error(dx_num, dx))
```

```
Cross Entropy Loss:  1.7918656600525245
dx error:  5.426680056536519e-09
```

Test a Small Fully Connected Network [2pt]

Please find the `SmallFullyConnectedNetwork` function in `lib/mlp/fully_conn.py` .
Again you only need to complete few lines of code in the TODO block.
Please design an `FC --> Leaky ReLU --> FC` network where the shapes of parameters match the given shapes.
Please insert the corresponding names you defined for each layer to param_name_w, and param_name_b respectively.
Here you only modify the param_name part, the `_w` , and `_b` are automatically assigned during network setup.

In [ ]:
```
%reload_ext autoreload

seed = 1234
np.random.seed(seed=seed)

model = SmallFullyConnectedNetwork()
loss_func = cross_entropy()

N, D, = 4, 4  # N: batch size, D: input dimension
H, C  = 30, 7 # H: hidden dimension, C: output dimension
std = 0.02
x = np.random.randn(N, D)
y = np.random.randint(C, size=N)

print ("Testing initialization ... ")

###################################################
# TODO: param_name should be replaced accordingly  #
###################################################
w1_std = abs(model.net.get_params("fc_1_w").std() - std)
b1 = model.net.get_params("fc_1_b").std()
w2_std = abs(model.net.get_params("fc_2_w").std() - std)
b2 = model.net.get_params("fc_2_b").std()
###################################################
#                END OF YOUR CODE                 #
###################################################

assert w1_std < std / 10, "First layer weights do not seem right"
assert np.all(b1 == 0), "First layer biases do not seem right"
assert w2_std < std / 10, "Second layer weights do not seem right"
assert np.all(b2 == 0), "Second layer biases do not seem right"
print ("Passed!")

print ("Testing test-time forward pass ... ")
w1 = np.linspace(-0.7, 0.3, num=D*H).reshape(D, H)
w2 = np.linspace(-0.2, 0.2, num=H*C).reshape(H, C)
b1 = np.linspace(-0.6, 0.2, num=H)
b2 = np.linspace(-0.9, 0.1, num=C)

###################################################
# TODO: param_name should be replaced accordingly  #
###################################################
model.net.assign("fc_1_w", w1)
model.net.assign("fc_1_b", b1)
model.net.assign("fc_2_w", w2)
model.net.assign("fc_2_b", b2)
###################################################
#                END OF YOUR CODE                 #
###################################################

feats = np.linspace(-5.5, 4.5, num=N*D).reshape(D, N).T
scores = model.forward(feats)
correct_scores = np.asarray([[-2.33876804, -1.92168925, -1.5046104
6, -1.08753166, -0.67045287, -0.25337408,  0.16370472],
                             [-1.57216705, -1.18571026, -0.7992534
8, -0.4127967 , -0.02633991, 0.36011687,  0.74657365],
                             [-0.80556605, -0.44973128, -0.0938965
1,  0.26193827,  0.61777304, 0.97360782,  1.32944259],
                             [-0.03896506,  0.2862477 ,  0.6114604
7,  0.93667323,  1.261886  , 1.58709876,  1.91231153]])
scores_diff = np.sum(np.abs(scores - correct_scores))
assert scores_diff < 1e-6, "Your implementation might be wrong!"
print ("Passed!")

print ("Testing the loss ...",)
y = np.asarray([0, 5, 1, 4])
loss = loss_func.forward(scores, y)
dLoss = loss_func.backward()
correct_loss = 2.4250083210516835
assert abs(loss - correct_loss) < 1e-10, "Your implementation migh
t be wrong!"
print ("Passed!")

print ("Testing the gradients (error should be no larger than 1e-
6) ...")
din = model.backward(dLoss)
for layer in model.net.layers:
    if not layer.params:
        continue
    for name in sorted(layer.grads):
        f = lambda _: loss_func.forward(model.forward(feats), y)
        grad_num = eval_numerical_gradient(f, layer.params[name],
verbose=False)
        print ('%s relative error: %.2e' % (name, rel_error(grad_n
um, layer.grads[name])))
```

```
Testing initialization ...
Passed!
Testing test-time forward pass ...
Passed!
Testing the loss ...
Passed!
Testing the gradients (error should be no larger than 1e-6) ...
fc_1_b relative error: 4.94e-09
fc_1_w relative error: 7.91e-09
fc_2_b relative error: 1.29e-10
fc_2_w relative error: 2.36e-08
```

Test a Fully Connected Network regularized with Dropout [2pt]

Please find the `DropoutNet` function in `fully_conn.py` under `lib/mlp` directory.
For this part you don't need to design a new network, just simply run the following test code.
If something goes wrong, you might want to double check your dropout implementation.

In [ ]:
```
%reload_ext autoreload

seed = 1234
np.random.seed(seed=seed)

N, D, C = 3, 15, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for keep_prob in [0, 0.25, 0.5]:
    np.random.seed(seed=seed)
    print ("Dropout p =", keep_prob)
    model = DropoutNet(keep_prob=keep_prob, seed=seed)
    loss_func = cross_entropy()
    output = model.forward(X, True, seed=seed)
    loss = loss_func.forward(output, y)
    dLoss = loss_func.backward()
    dX = model.backward(dLoss)
    grads = model.net.grads

    print ("Error of gradients should be around or less than 1e-
3")
    for name in sorted(grads):
        if name not in model.net.params.keys():
            continue
        f = lambda _: loss_func.forward(model.forward(X, True, see
d=seed), y)
        grad_num = eval_numerical_gradient(f, model.net.params[nam
e], verbose=False, h=1e-5)
        print ("{} relative error: {}".format(name, rel_error(grad
_num, grads[name])))
    print ()
```

```
Dropout p = 0
Error of gradients should be around or less than 1e-3
fc1_b relative error: 1.5562038181567602e-05
fc1_w relative error: 9.723764019716731e-05
fc2_b relative error: 2.7841895262663007e-06
fc2_w relative error: 0.0019218743762210607
fc3_b relative error: 6.9596628378975445e-09
fc3_w relative error: 0.0019961170369276215

Dropout p = 0.25
Error of gradients should be around or less than 1e-3
fc1_b relative error: 2.1950150914280595e-06
fc1_w relative error: 2.5552274036442842e-05
fc2_b relative error: 1.878084533525588e-06
fc2_w relative error: 0.0009455388205765454
fc3_b relative error: 6.521995746108614e-10
fc3_w relative error: 0.00031318161857379944

Dropout p = 0.5
Error of gradients should be around or less than 1e-3
fc1_b relative error: 2.2332156460184743e-07
fc1_w relative error: 7.381101518134411e-05
fc2_b relative error: 2.609643832664101e-07
fc2_w relative error: 0.003984285792093247
fc3_b relative error: 1.2050218326231938e-09
fc3_w relative error: 3.584680630720856e-05
```

Training a Network

In this section, we defined a `TinyNet` class for you to fill in the TODO block in `lib/mlp/fully_conn.py` .

- Here please design a two layer fully connected network with Leaky ReLU activation (`Flatten --> FC --> Leaky ReLU --> FC`).
- You can adjust the number of hidden neurons, batch_size, epochs, and learning rate decay parameters.
- Please read the `lib/train.py` carefully and complete the TODO blocks in the `train_net` function first. Codes in "Test a Small Fully Connected Network" can be helpful.
- In addition, read how the SGD function is implemented in `lib/optim.py` , you will be asked to complete three other optimization methods in the later sections.

In [ ]:
```python
# Arrange the data
data_dict = {
    "data_train": (data["data_train"], data["labels_train"]),
    "data_val": (data["data_val"], data["labels_val"]),
    "data_test": (data["data_test"], data["labels_test"])
}
```

```
In [ ]: print("Data shape:", data["data_train"].shape)
        print("Flattened data input size:", np.prod(data["data_train"].sha
        pe[1:]))
        print("Number of data classes:", max(data['labels_train']) + 1)
```

```
Data shape: (70000, 32, 32, 3)
Flattened data input size: 3072
Number of data classes: 10
```

Now train the network to achieve at least 75% validation accuracy [5pt]

You may only adjust the hyperparameters inside the TODO block

In [ ]:
```
%reload_ext autoreload

seed = 123
np.random.seed(seed=seed)

model = TinyNet()
loss_f = cross_entropy()
optimizer = SGD(model.net, 1e-2)

results = None
####################################################################
###########
# TODO: Use the train_net function you completed to train a networ
k          #
####################################################################
###########

batch_size = 64
epochs = 24
lr_decay = .99
lr_decay_every = 50

####################################################################
###########
#                             END OF YOUR CODE
#
####################################################################
###########
results = train_net(data_dict, model, loss_f, optimizer, batch_siz
e, epochs,
                    lr_decay, lr_decay_every, show_every=10000, ve
rbose=True)
opt_params, loss_hist, train_acc_hist, val_acc_hist = results
```

```
(Iteration 1 / 26232) loss: 2.3026488974526993
(Epoch 1 / 24) Training Accuracy: 0.18954285714285715, Validation
Accuracy: 0.18206938900828984
(Epoch 2 / 24) Training Accuracy: 0.1972, Validation Accuracy: 0.1
922014123426466
(Epoch 3 / 24) Training Accuracy: 0.27534285714285717, Validation
Accuracy: 0.26619588578446424
(Epoch 4 / 24) Training Accuracy: 0.3905571428571429, Validation A
ccuracy: 0.3718145532698803
(Epoch 5 / 24) Training Accuracy: 0.4703, Validation Accuracy: 0.4
6085354620816704
(Epoch 6 / 24) Training Accuracy: 0.5492142857142858, Validation A
ccuracy: 0.5455941050046055
(Epoch 7 / 24) Training Accuracy: 0.5911714285714286, Validation A
ccuracy: 0.5848940743015044
(Epoch 8 / 24) Training Accuracy: 0.6340142857142858, Validation A
ccuracy: 0.6263432606693276
(Epoch 9 / 24) Training Accuracy: 0.6592857142857143, Validation A
ccuracy: 0.6524408965305496
(Iteration 10001 / 26232) loss: 1.0412519110593301
(Epoch 10 / 24) Training Accuracy: 0.6791714285714285, Validation
Accuracy: 0.6684065090574148
(Epoch 11 / 24) Training Accuracy: 0.6914571428571429, Validation
Accuracy: 0.6782315013816396
(Epoch 12 / 24) Training Accuracy: 0.7048571428571428, Validation
Accuracy: 0.6917408658274485
(Epoch 13 / 24) Training Accuracy: 0.7139, Validation Accuracy: 0.
7052502302732576
(Epoch 14 / 24) Training Accuracy: 0.7254714285714285, Validation
Accuracy: 0.7159963156278785
(Epoch 15 / 24) Training Accuracy: 0.7333142857142857, Validation
Accuracy: 0.7224439668406509
(Epoch 16 / 24) Training Accuracy: 0.7427285714285714, Validation
Accuracy: 0.7316548971446116
(Epoch 17 / 24) Training Accuracy: 0.7477857142857143, Validation
Accuracy: 0.7420939514891004
(Epoch 18 / 24) Training Accuracy: 0.7540571428571429, Validation
Accuracy: 0.7445501995701566
(Iteration 20001 / 26232) loss: 0.6672656543682636
(Epoch 19 / 24) Training Accuracy: 0.7625571428571428, Validation
Accuracy: 0.7479275406816088
(Epoch 20 / 24) Training Accuracy: 0.7692142857142857, Validation
Accuracy: 0.7565244089653055
(Epoch 21 / 24) Training Accuracy: 0.7765, Validation Accuracy: 0.
7617439361375499
(Epoch 22 / 24) Training Accuracy: 0.7806428571428572, Validation
Accuracy: 0.762972060178078
(Epoch 23 / 24) Training Accuracy: 0.7886571428571428, Validation
Accuracy: 0.7669634633097943
(Epoch 24 / 24) Training Accuracy: 0.7928, Validation Accuracy: 0.
7749462695732269
```

In [ ]:
```
# Take a look at what names of params were stored
print (opt_params.keys())
```

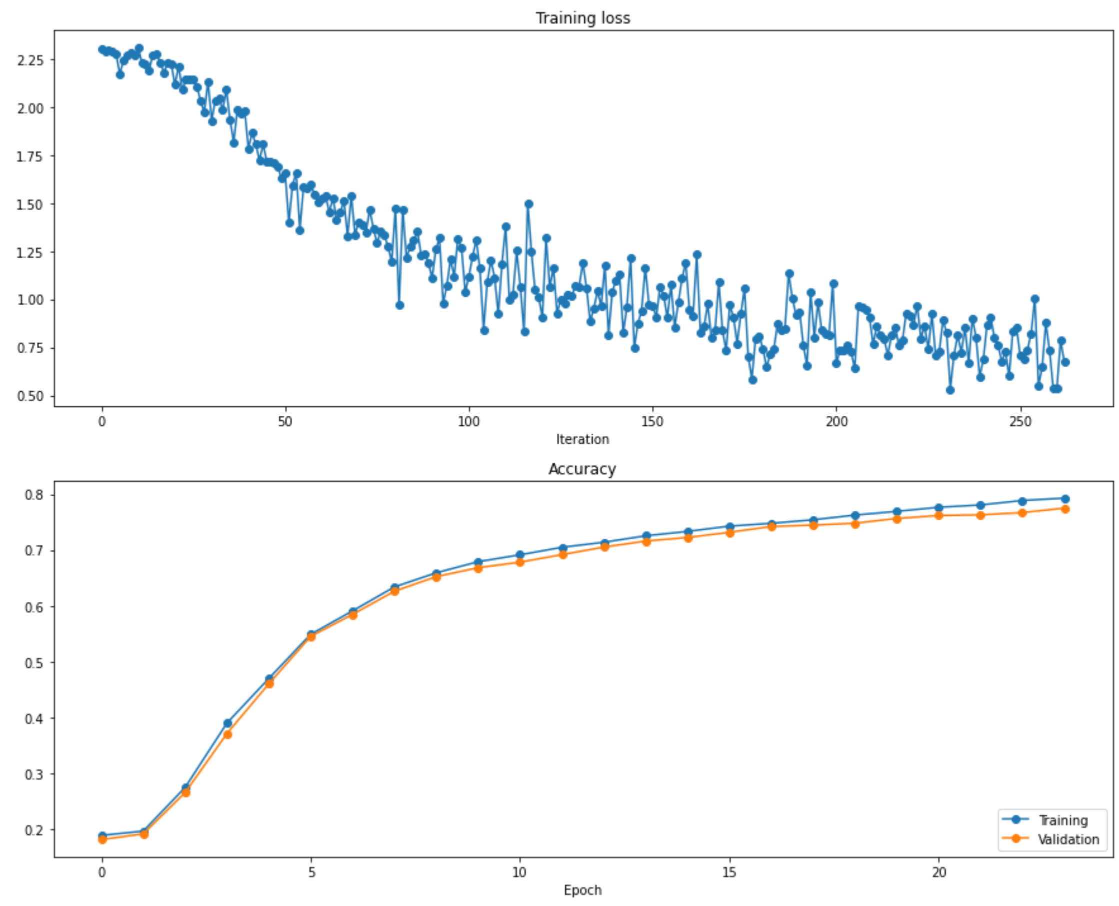
```
dict_keys(['fc1_w', 'fc1_b', 'fc2_w', 'fc2_b'])
```

In [ ]:
```
# Demo: How to load the parameters to a newly defined network
model = TinyNet()
model.net.load(opt_params)
val_acc = compute_acc(model, data["data_val"], data["labels_val"])
print ("Validation Accuracy: {}%".format(val_acc*100))
test_acc = compute_acc(model, data["data_test"], data["labels_tes
t"])
print ("Testing Accuracy: {}%".format(test_acc*100))
```

```
Loading Params: fc1_w Shape: (3072, 3072)
Loading Params: fc1_b Shape: (3072,)
Loading Params: fc2_w Shape: (3072, 10)
Loading Params: fc2_b Shape: (10,)
Validation Accuracy: 77.4946269573227%
Testing Accuracy: 75.5800553165335%
```

In [ ]:
```
# Plot the learning curves
plt.subplot(2, 1, 1)
plt.title('Training loss')
loss_hist_ = loss_hist[1::100] # sparse the curve a bit
plt.plot(loss_hist_, '-o')
plt.xlabel('Iteration')

plt.subplot(2, 1, 2)
plt.title('Accuracy')
plt.plot(train_acc_hist, '-o', label='Training')
plt.plot(val_acc_hist, '-o', label='Validation')
plt.xlabel('Epoch')
plt.legend(loc='lower right')
plt.gcf().set_size_inches(15, 12)
plt.show()
```



Different Optimizers

There are several more advanced optimizers than vanilla SGD, you will implement three more sophisticated and widely-used methods in this section.
Please complete the TODOs in the `lib/optim.py` .

SGD + Momentum [2pt]

The update rule of SGD plus momentum is as shown below:

$$\begin{aligned}
v_t &: \text{last update of the velocity} \\
\gamma &: \text{momentum} \\
\eta &: \text{learning rate} \\
v_t &= \gamma v_{t-1} - \eta \nabla_\theta J(\theta) \\
\theta &= \theta + v_t
\end{aligned}$$

The initial value of v_t is 0. Complete the `SGDM()` function in `lib/optim.py` .

```
In [ ]: %reload_ext autoreload

        # Test the implementation of SGD with Momentum
        seed = 123
        np.random.seed(seed=seed)

        N, D = 4, 5
        test_sgd = sequential(fc(N, D, name="sgd_fc"))

        w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
        dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
        v = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

        test_sgd.layers[0].params = {"sgd_fc_w": w}
        test_sgd.layers[0].grads = {"sgd_fc_w": dw}

        test_sgd_momentum = SGDM(test_sgd, 1e-3, 0.9)
        test_sgd_momentum.velocity = {"sgd_fc_w": v}
        test_sgd_momentum.step()

        updated_w = test_sgd.layers[0].params["sgd_fc_w"]
        velocity = test_sgd_momentum.velocity["sgd_fc_w"]

        expected_updated_w = np.asarray([
          [ 0.1406,      0.20738947,  0.27417895,  0.34096842,  0.4077578
        9],
          [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.7417052
        6],
          [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.0756526
        3],
          [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096
        ]])
        expected_velocity = np.asarray([
          [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.5972315
        8],
          [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.6680210
        5],
          [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.7388105
        3],
          [ 0.75296842,  0.76712632,  0.78128421,  0.79544211,  0.8096
        ]])

        print ('The following errors should be around or less than 1e-8')
        print ('updated_w error: ', rel_error(updated_w, expected_updated_
        w))
        print ('velocity error: ', rel_error(expected_velocity, velocity))
```

```
The following errors should be around or less than 1e-8
updated_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

Comparing SGD and SGD with Momentum [2pt]

Run the following code block to train a multi-layer fully connected network with both SGD and SGD plus Momentum. The network trained with SGDM optimizer should converge faster.

In [ ]:
```
seed = 123
np.random.seed(seed=seed)

# Arrange a small data
num_train = 50000
small_data_dict = {
    "data_train": (data["data_train"][:num_train], data["labels_tr
ain"][:num_train]),
    "data_val": (data["data_val"], data["labels_val"]),
    "data_test": (data["data_test"], data["labels_test"])
}

model_sgd      = FullyConnectedNetwork()
model_sgdm     = FullyConnectedNetwork()
loss_f_sgd     = cross_entropy()
loss_f_sgdm    = cross_entropy()
optimizer_sgd  = SGD(model_sgd.net, 1e-2)
optimizer_sgdm = SGDM(model_sgdm.net, 1e-2, 0.9)

print ("Training with Vanilla SGD...")
results_sgd = train_net(small_data_dict, model_sgd, loss_f_sgd, op
timizer_sgd, batch_size=32,
                        max_epochs=15, show_every=100, verbose=Tru
e)

print ("\nTraining with SGD plus Momentum...")
results_sgdm = train_net(small_data_dict, model_sgdm, loss_f_sgdm,
optimizer_sgdm, batch_size=32,
                         max_epochs=15, show_every=100, verbose=Tr
ue)

opt_params_sgd,  loss_hist_sgd,  train_acc_hist_sgd,  val_acc_hist
_sgd  = results_sgd
opt_params_sgdm, loss_hist_sgdm, train_acc_hist_sgdm, val_acc_hist
_sgdm = results_sgdm

plt.subplot(3, 1, 1)
plt.title('Training loss')
plt.xlabel('Iteration')

plt.subplot(3, 1, 2)
plt.title('Training accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 3)
plt.title('Validation accuracy')
plt.xlabel('Epoch')

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgd, 'o', label="Vanilla SGD")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgd, '-o', label="Vanilla SGD")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgd, '-o', label="Vanilla SGD")

plt.subplot(3, 1, 1)
plt.plot(loss_hist_sgdm, 'o', label="SGD with Momentum")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_sgdm, '-o', label="SGD with Momentum")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_sgdm, '-o', label="SGD with Momentum")

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
Training with Vanilla SGD...
(Iteration 1 / 23430) loss: 2.302024602405127
(Iteration 101 / 23430) loss: 2.3034601238289887
(Iteration 201 / 23430) loss: 2.268727870891892
(Iteration 301 / 23430) loss: 2.2437751202726104
(Iteration 401 / 23430) loss: 2.2383190105263284
(Iteration 501 / 23430) loss: 2.2728742533815263
(Iteration 601 / 23430) loss: 2.2451687770884394
(Iteration 701 / 23430) loss: 2.2622141329338215
(Iteration 801 / 23430) loss: 2.2573692693348812
(Iteration 901 / 23430) loss: 2.338094180590898
(Iteration 1001 / 23430) loss: 2.2025783664019407
(Iteration 1101 / 23430) loss: 2.193928573542903
(Iteration 1201 / 23430) loss: 2.142988679764224
(Iteration 1301 / 23430) loss: 2.2688718233746474
(Iteration 1401 / 23430) loss: 2.1504753777293333
(Iteration 1501 / 23430) loss: 2.27352297005684
(Epoch 1 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.1
8206938900828984
(Iteration 1601 / 23430) loss: 2.2460415185131377
(Iteration 1701 / 23430) loss: 2.3338753090342754
(Iteration 1801 / 23430) loss: 2.117519352081804
(Iteration 1901 / 23430) loss: 2.2055354583182085
(Iteration 2001 / 23430) loss: 2.143731801898431
(Iteration 2101 / 23430) loss: 2.2396229891697548
(Iteration 2201 / 23430) loss: 2.15097839809764
(Iteration 2301 / 23430) loss: 2.1239988360035857
(Iteration 2401 / 23430) loss: 2.2576888344005748
(Iteration 2501 / 23430) loss: 2.29772304584748
(Iteration 2601 / 23430) loss: 2.2162789698622034
(Iteration 2701 / 23430) loss: 2.1420435054696814
(Iteration 2801 / 23430) loss: 2.2409122687683816
(Iteration 2901 / 23430) loss: 2.2676203279797744
(Iteration 3001 / 23430) loss: 2.3097516637777327
(Iteration 3101 / 23430) loss: 2.2175169448719236
(Epoch 2 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.1
8206938900828984
(Iteration 3201 / 23430) loss: 2.194154112103643
(Iteration 3301 / 23430) loss: 2.3395208325098986
(Iteration 3401 / 23430) loss: 2.2132838015857446
(Iteration 3501 / 23430) loss: 2.2152499932798633
(Iteration 3601 / 23430) loss: 2.167542970843872
(Iteration 3701 / 23430) loss: 2.1897861389815265
(Iteration 3801 / 23430) loss: 2.131262375298502
(Iteration 3901 / 23430) loss: 2.1471137796877966
(Iteration 4001 / 23430) loss: 2.285455946890295
(Iteration 4101 / 23430) loss: 2.1982515121053874
(Iteration 4201 / 23430) loss: 2.2074600074836823
(Iteration 4301 / 23430) loss: 2.22467412692337
(Iteration 4401 / 23430) loss: 2.053039396099415
(Iteration 4501 / 23430) loss: 2.352428699409104
(Iteration 4601 / 23430) loss: 2.2293574609689446
(Epoch 3 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.1
8206938900828984
(Iteration 4701 / 23430) loss: 2.39724854712492
(Iteration 4801 / 23430) loss: 2.2141731688845034
(Iteration 4901 / 23430) loss: 2.267153104703051
(Iteration 5001 / 23430) loss: 2.085668245526734
(Iteration 5101 / 23430) loss: 2.1813810605556117
(Iteration 5201 / 23430) loss: 2.147264874797247
```

```
(Iteration 5301 / 23430) loss: 2.1776339065381904
(Iteration 5401 / 23430) loss: 2.193729981036711
(Iteration 5501 / 23430) loss: 2.1394524956297456
(Iteration 5601 / 23430) loss: 2.2202616566499858
(Iteration 5701 / 23430) loss: 2.2622337468631857
(Iteration 5801 / 23430) loss: 2.227685577402541
(Iteration 5901 / 23430) loss: 2.088278062857272
(Iteration 6001 / 23430) loss: 2.1850199494228804
(Iteration 6101 / 23430) loss: 2.1931225857353613
(Iteration 6201 / 23430) loss: 2.2734653728229812
(Epoch 4 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.1
8206938900828984
(Iteration 6301 / 23430) loss: 2.2331918410123763
(Iteration 6401 / 23430) loss: 2.1700783825659116
(Iteration 6501 / 23430) loss: 2.263550255866303
(Iteration 6601 / 23430) loss: 2.2692601821721983
(Iteration 6701 / 23430) loss: 2.1198609657360263
(Iteration 6801 / 23430) loss: 2.2073063433802984
(Iteration 6901 / 23430) loss: 2.333841586815027
(Iteration 7001 / 23430) loss: 2.24289411442957
(Iteration 7101 / 23430) loss: 2.1918140333720397
(Iteration 7201 / 23430) loss: 2.261579850472472
(Iteration 7301 / 23430) loss: 2.396177740667493
(Iteration 7401 / 23430) loss: 2.2639875708037245
(Iteration 7501 / 23430) loss: 2.2073632923493647
(Iteration 7601 / 23430) loss: 2.163729093617352
(Iteration 7701 / 23430) loss: 2.273839708919157
(Iteration 7801 / 23430) loss: 2.182653481864328
(Epoch 5 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.1
8206938900828984
(Iteration 7901 / 23430) loss: 2.279702444848493
(Iteration 8001 / 23430) loss: 2.361265166117029
(Iteration 8101 / 23430) loss: 2.1805389463231672
(Iteration 8201 / 23430) loss: 2.2969331498596617
(Iteration 8301 / 23430) loss: 2.2361079019196635
(Iteration 8401 / 23430) loss: 2.25849198286873
(Iteration 8501 / 23430) loss: 2.2413859227701622
(Iteration 8601 / 23430) loss: 2.211510471331209
(Iteration 8701 / 23430) loss: 2.3336874163253554
(Iteration 8801 / 23430) loss: 2.1460049798692906
(Iteration 8901 / 23430) loss: 2.2739742890572265
(Iteration 9001 / 23430) loss: 2.2788831350920074
(Iteration 9101 / 23430) loss: 2.1011709775162757
(Iteration 9201 / 23430) loss: 2.2175363456825545
(Iteration 9301 / 23430) loss: 2.276154867535076
(Epoch 6 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.1
8206938900828984
(Iteration 9401 / 23430) loss: 2.0642691788335013
(Iteration 9501 / 23430) loss: 2.2745090868034987
(Iteration 9601 / 23430) loss: 2.1333546288924934
(Iteration 9701 / 23430) loss: 2.1175980107656276
(Iteration 9801 / 23430) loss: 2.1546744597512304
(Iteration 9901 / 23430) loss: 2.2688888649793952
(Iteration 10001 / 23430) loss: 2.183058401210102
(Iteration 10101 / 23430) loss: 2.251299393685507
(Iteration 10201 / 23430) loss: 2.2631166633317994
(Iteration 10301 / 23430) loss: 2.167193612321144
(Iteration 10401 / 23430) loss: 2.2763755254682536
(Iteration 10501 / 23430) loss: 2.1660364964014494
(Iteration 10601 / 23430) loss: 2.1639501612675773
```

```
(Iteration 10701 / 23430) loss: 2.2584520531469434
(Iteration 10801 / 23430) loss: 2.2152285044154416
(Iteration 10901 / 23430) loss: 2.129332517760416
(Epoch 7 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.1
8206938900828984
(Iteration 11001 / 23430) loss: 2.1812958200919352
(Iteration 11101 / 23430) loss: 2.2145639536648782
(Iteration 11201 / 23430) loss: 2.1397542330464443
(Iteration 11301 / 23430) loss: 2.192299993923399
(Iteration 11401 / 23430) loss: 2.084191641961175
(Iteration 11501 / 23430) loss: 2.0391661886170698
(Iteration 11601 / 23430) loss: 2.1643675050227866
(Iteration 11701 / 23430) loss: 1.998129043286843
(Iteration 11801 / 23430) loss: 1.994050522550237
(Iteration 11901 / 23430) loss: 2.200745623648538
(Iteration 12001 / 23430) loss: 1.977341137334038
(Iteration 12101 / 23430) loss: 1.9679773805617364
(Iteration 12201 / 23430) loss: 2.207848277127094
(Iteration 12301 / 23430) loss: 2.191981004836007
(Iteration 12401 / 23430) loss: 2.126533178890107
(Epoch 8 / 15) Training Accuracy: 0.26554, Validation Accuracy: 0.
2566779244703715
(Iteration 12501 / 23430) loss: 1.986016739677467
(Iteration 12601 / 23430) loss: 2.033430994285775
(Iteration 12701 / 23430) loss: 1.8119360013391255
(Iteration 12801 / 23430) loss: 1.9390599645946671
(Iteration 12901 / 23430) loss: 2.2381560403774725
(Iteration 13001 / 23430) loss: 2.041420211593111
(Iteration 13101 / 23430) loss: 2.079549079065782
(Iteration 13201 / 23430) loss: 2.0422824733900016
(Iteration 13301 / 23430) loss: 1.811861953700057
(Iteration 13401 / 23430) loss: 1.9351495765265259
(Iteration 13501 / 23430) loss: 1.984167269978542
(Iteration 13601 / 23430) loss: 1.9181778266096534
(Iteration 13701 / 23430) loss: 1.9846246096471971
(Iteration 13801 / 23430) loss: 2.206450737473464
(Iteration 13901 / 23430) loss: 1.9779072647971965
(Iteration 14001 / 23430) loss: 1.8408176865541894
(Epoch 9 / 15) Training Accuracy: 0.2768, Validation Accuracy: 0.2
600552655818238
(Iteration 14101 / 23430) loss: 1.7569095207460457
(Iteration 14201 / 23430) loss: 1.9065341735289107
(Iteration 14301 / 23430) loss: 1.840585211203873
(Iteration 14401 / 23430) loss: 1.9077880267025016
(Iteration 14501 / 23430) loss: 1.8556381527055887
(Iteration 14601 / 23430) loss: 1.9433920970151333
(Iteration 14701 / 23430) loss: 1.9673150281977416
(Iteration 14801 / 23430) loss: 1.9219862051431764
(Iteration 14901 / 23430) loss: 1.6838468222871612
(Iteration 15001 / 23430) loss: 1.8905392116039255
(Iteration 15101 / 23430) loss: 1.858636470128303
(Iteration 15201 / 23430) loss: 1.8125553565722259
(Iteration 15301 / 23430) loss: 1.7567598757902674
(Iteration 15401 / 23430) loss: 2.167239201771908
(Iteration 15501 / 23430) loss: 1.8741552478505763
(Iteration 15601 / 23430) loss: 1.8212786941394377
(Epoch 10 / 15) Training Accuracy: 0.33108, Validation Accuracy:
0.32299662265888857
(Iteration 15701 / 23430) loss: 1.7537026231221775
(Iteration 15801 / 23430) loss: 1.9728123854944193
```

```
(Iteration 15901 / 23430) loss: 1.9557297603941097
(Iteration 16001 / 23430) loss: 1.6281151564229643
(Iteration 16101 / 23430) loss: 1.6006599826254275
(Iteration 16201 / 23430) loss: 1.8956634528890361
(Iteration 16301 / 23430) loss: 1.9344399880350918
(Iteration 16401 / 23430) loss: 1.722606558111463
(Iteration 16501 / 23430) loss: 1.607338753648624
(Iteration 16601 / 23430) loss: 1.6019310832311
(Iteration 16701 / 23430) loss: 1.7315967814801783
(Iteration 16801 / 23430) loss: 1.5204441476694617
(Iteration 16901 / 23430) loss: 1.5464829112340355
(Iteration 17001 / 23430) loss: 1.6082016849590803
(Iteration 17101 / 23430) loss: 1.8403858445214454
(Epoch 11 / 15) Training Accuracy: 0.36932, Validation Accuracy:
0.34264660730733804
(Iteration 17201 / 23430) loss: 1.7732237918799953
(Iteration 17301 / 23430) loss: 1.5170430443029288
(Iteration 17401 / 23430) loss: 1.515947789098848
(Iteration 17501 / 23430) loss: 1.7578906791866762
(Iteration 17601 / 23430) loss: 1.5339618433488769
(Iteration 17701 / 23430) loss: 1.8577912815964448
(Iteration 17801 / 23430) loss: 1.8913185969349837
(Iteration 17901 / 23430) loss: 1.9127473116810216
(Iteration 18001 / 23430) loss: 1.5610399005381488
(Iteration 18101 / 23430) loss: 1.7728833646451572
(Iteration 18201 / 23430) loss: 1.8224428314394978
(Iteration 18301 / 23430) loss: 1.6567309881707013
(Iteration 18401 / 23430) loss: 1.787513889026173
(Iteration 18501 / 23430) loss: 1.4083515989259765
(Iteration 18601 / 23430) loss: 1.4333798509491085
(Iteration 18701 / 23430) loss: 1.3818909214593609
(Epoch 12 / 15) Training Accuracy: 0.39642, Validation Accuracy:
0.37427080135093643
(Iteration 18801 / 23430) loss: 1.7807727538317262
(Iteration 18901 / 23430) loss: 1.6812143585637265
(Iteration 19001 / 23430) loss: 1.5376104706345064
(Iteration 19101 / 23430) loss: 1.3895878769254564
(Iteration 19201 / 23430) loss: 1.3947151970430236
(Iteration 19301 / 23430) loss: 1.5575912996786547
(Iteration 19401 / 23430) loss: 1.4167487841129387
(Iteration 19501 / 23430) loss: 1.5042106988888142
(Iteration 19601 / 23430) loss: 1.4679896312524396
(Iteration 19701 / 23430) loss: 1.5636968793689219
(Iteration 19801 / 23430) loss: 1.8028723857690991
(Iteration 19901 / 23430) loss: 1.7070041588825458
(Iteration 20001 / 23430) loss: 1.2881934116726914
(Iteration 20101 / 23430) loss: 1.1504785301809513
(Iteration 20201 / 23430) loss: 1.0296477227183745
(Iteration 20301 / 23430) loss: 1.7572835061743661
(Epoch 13 / 15) Training Accuracy: 0.51806, Validation Accuracy:
0.500460546515198
(Iteration 20401 / 23430) loss: 1.4946293171987717
(Iteration 20501 / 23430) loss: 0.9703649069519367
(Iteration 20601 / 23430) loss: 1.2601450201804547
(Iteration 20701 / 23430) loss: 1.3951201635990438
(Iteration 20801 / 23430) loss: 1.4451787190042535
(Iteration 20901 / 23430) loss: 1.2465183011157983
(Iteration 21001 / 23430) loss: 1.4093181209027492
(Iteration 21101 / 23430) loss: 1.729808889981196
(Iteration 21201 / 23430) loss: 1.1993450554261456
```

```
(Iteration 21301 / 23430) loss: 1.1573744131325385
(Iteration 21401 / 23430) loss: 1.0719985720710203
(Iteration 21501 / 23430) loss: 1.366212657441062
(Iteration 21601 / 23430) loss: 1.3492223846833864
(Iteration 21701 / 23430) loss: 1.311656496054532
(Iteration 21801 / 23430) loss: 1.5450930200194037
(Epoch 14 / 15) Training Accuracy: 0.57572, Validation Accuracy:
0.553269880257906
(Iteration 21901 / 23430) loss: 1.1635558037949756
(Iteration 22001 / 23430) loss: 1.3971617976313158
(Iteration 22101 / 23430) loss: 1.241129888775715
(Iteration 22201 / 23430) loss: 1.42637552895074
(Iteration 22301 / 23430) loss: 1.0883458040511138
(Iteration 22401 / 23430) loss: 1.1865591664654043
(Iteration 22501 / 23430) loss: 1.2482097753740005
(Iteration 22601 / 23430) loss: 0.9736811717017185
(Iteration 22701 / 23430) loss: 1.1026065545747212
(Iteration 22801 / 23430) loss: 0.9579572882967304
(Iteration 22901 / 23430) loss: 1.1339795484058757
(Iteration 23001 / 23430) loss: 1.0678596442400266
(Iteration 23101 / 23430) loss: 1.2637995502208796
(Iteration 23201 / 23430) loss: 1.4644910954727401
(Iteration 23301 / 23430) loss: 1.4613114716053492
(Iteration 23401 / 23430) loss: 0.8772471470512316
(Epoch 15 / 15) Training Accuracy: 0.62214, Validation Accuracy:
0.6027018728891618

Training with SGD plus Momentum...
(Iteration 1 / 23430) loss: 2.30213557079308
(Iteration 101 / 23430) loss: 2.33372822045504
(Iteration 201 / 23430) loss: 2.207808147308481
(Iteration 301 / 23430) loss: 2.1644564839379763
(Iteration 401 / 23430) loss: 2.1894932035986696
(Iteration 501 / 23430) loss: 2.2570555159616266
(Iteration 601 / 23430) loss: 2.223516855180442
(Iteration 701 / 23430) loss: 2.262123604182423
(Iteration 801 / 23430) loss: 2.252756352855614
(Iteration 901 / 23430) loss: 2.360567449447703
(Iteration 1001 / 23430) loss: 2.201613955822357
(Iteration 1101 / 23430) loss: 2.176806406888798
(Iteration 1201 / 23430) loss: 2.1124395101416136
(Iteration 1301 / 23430) loss: 2.2741097795235437
(Iteration 1401 / 23430) loss: 2.133580653190603
(Iteration 1501 / 23430) loss: 2.2789630060127357
(Epoch 1 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.1
8206938900828984
(Iteration 1601 / 23430) loss: 2.33526528092627
(Iteration 1701 / 23430) loss: 2.014204131424042
(Iteration 1801 / 23430) loss: 1.8718770885703664
(Iteration 1901 / 23430) loss: 2.0937186907549434
(Iteration 2001 / 23430) loss: 1.9480332005886376
(Iteration 2101 / 23430) loss: 2.0671319149628986
(Iteration 2201 / 23430) loss: 1.8730728006241144
(Iteration 2301 / 23430) loss: 1.9891641056727587
(Iteration 2401 / 23430) loss: 2.2327726595170216
(Iteration 2501 / 23430) loss: 1.7626232099934291
(Iteration 2601 / 23430) loss: 1.6134806921716387
(Iteration 2701 / 23430) loss: 1.694378132519821
(Iteration 2801 / 23430) loss: 1.610111703323876
(Iteration 2901 / 23430) loss: 1.9377617632129647
```

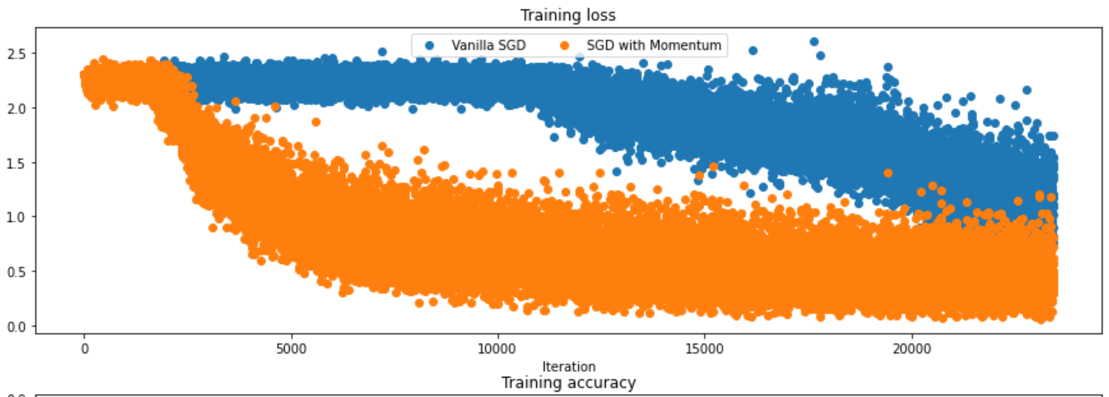
```
(Iteration 3001 / 23430) loss: 1.2946256848097841
(Iteration 3101 / 23430) loss: 1.3215964562323153
(Epoch 2 / 15) Training Accuracy: 0.47904, Validation Accuracy: 0.
4826527479275407
(Iteration 3201 / 23430) loss: 1.3755849366824517
(Iteration 3301 / 23430) loss: 1.237004950675046
(Iteration 3401 / 23430) loss: 1.5381463351333706
(Iteration 3501 / 23430) loss: 1.2181693207556
(Iteration 3601 / 23430) loss: 1.8044944573152937
(Iteration 3701 / 23430) loss: 1.4213849440837152
(Iteration 3801 / 23430) loss: 1.3122977593006293
(Iteration 3901 / 23430) loss: 1.0621525915128882
(Iteration 4001 / 23430) loss: 1.037030073370089
(Iteration 4101 / 23430) loss: 1.0297647116229374
(Iteration 4201 / 23430) loss: 1.2437481955423197
(Iteration 4301 / 23430) loss: 1.2777836734045986
(Iteration 4401 / 23430) loss: 1.0608317974712898
(Iteration 4501 / 23430) loss: 1.690590850003147
(Iteration 4601 / 23430) loss: 1.0076725460135452
(Epoch 3 / 15) Training Accuracy: 0.6523, Validation Accuracy: 0.6
266502916794596
(Iteration 4701 / 23430) loss: 0.9984468168718542
(Iteration 4801 / 23430) loss: 1.5006052981411655
(Iteration 4901 / 23430) loss: 0.8821859788711193
(Iteration 5001 / 23430) loss: 1.033284889678793
(Iteration 5101 / 23430) loss: 0.6525768378986385
(Iteration 5201 / 23430) loss: 1.3171847072664473
(Iteration 5301 / 23430) loss: 1.0265764431380004
(Iteration 5401 / 23430) loss: 1.212413213340297
(Iteration 5501 / 23430) loss: 0.8097190594205828
(Iteration 5601 / 23430) loss: 1.2869893575462292
(Iteration 5701 / 23430) loss: 1.3597573478183715
(Iteration 5801 / 23430) loss: 0.8283671258790969
(Iteration 5901 / 23430) loss: 0.932813713023838
(Iteration 6001 / 23430) loss: 0.9989597130603814
(Iteration 6101 / 23430) loss: 1.0604263668571343
(Iteration 6201 / 23430) loss: 1.0485770341632767
(Epoch 4 / 15) Training Accuracy: 0.74052, Validation Accuracy: 0.
7172244396684065
(Iteration 6301 / 23430) loss: 0.7962859010097227
(Iteration 6401 / 23430) loss: 1.0136840272988101
(Iteration 6501 / 23430) loss: 0.7745610388315698
(Iteration 6601 / 23430) loss: 1.0796702008293357
(Iteration 6701 / 23430) loss: 1.2959365119084518
(Iteration 6801 / 23430) loss: 0.5457065128384504
(Iteration 6901 / 23430) loss: 0.5387511619912678
(Iteration 7001 / 23430) loss: 0.4597487592773138
(Iteration 7101 / 23430) loss: 1.04568802185889
(Iteration 7201 / 23430) loss: 0.5052441601252374
(Iteration 7301 / 23430) loss: 0.9370747159927769
(Iteration 7401 / 23430) loss: 0.5937097910579898
(Iteration 7501 / 23430) loss: 0.5346519619360114
(Iteration 7601 / 23430) loss: 0.5872200211511391
(Iteration 7701 / 23430) loss: 0.5629030652928922
(Iteration 7801 / 23430) loss: 0.6794905762521665
(Epoch 5 / 15) Training Accuracy: 0.78008, Validation Accuracy: 0.
7617439361375499
(Iteration 7901 / 23430) loss: 0.9541257481322454
(Iteration 8001 / 23430) loss: 0.3300101388266163
(Iteration 8101 / 23430) loss: 0.9980790171649632
```

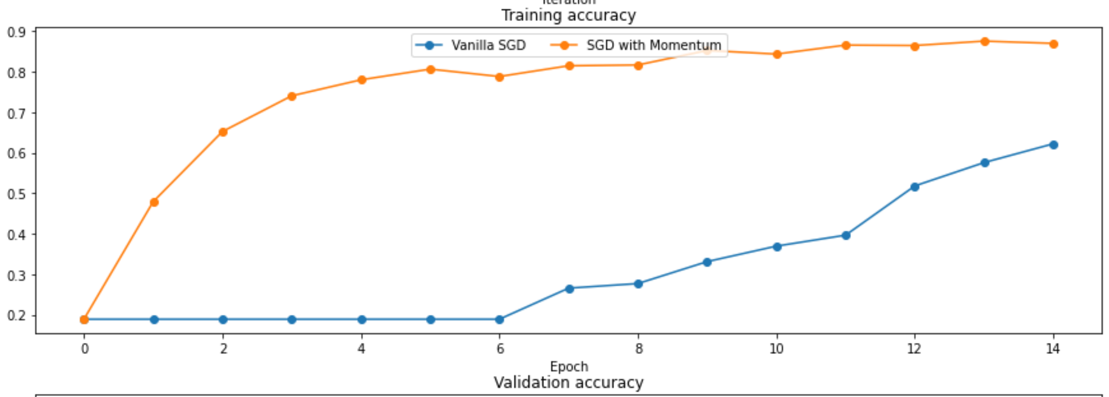
```
(Iteration 8201 / 23430) loss: 0.4777520777936344
(Iteration 8301 / 23430) loss: 0.7514339379530562
(Iteration 8401 / 23430) loss: 0.676100698656215
(Iteration 8501 / 23430) loss: 1.0169169334064119
(Iteration 8601 / 23430) loss: 0.877850961638787
(Iteration 8701 / 23430) loss: 0.4380937043142467
(Iteration 8801 / 23430) loss: 0.6216897970476348
(Iteration 8901 / 23430) loss: 0.49604568337895666
(Iteration 9001 / 23430) loss: 0.35584553205005326
(Iteration 9101 / 23430) loss: 0.5656195476840478
(Iteration 9201 / 23430) loss: 0.5882940267845291
(Iteration 9301 / 23430) loss: 0.8274494451526647
(Epoch 6 / 15) Training Accuracy: 0.80648, Validation Accuracy: 0.
7743322075529628
(Iteration 9401 / 23430) loss: 1.057533287691195
(Iteration 9501 / 23430) loss: 0.5068612520382815
(Iteration 9601 / 23430) loss: 0.43892342775380644
(Iteration 9701 / 23430) loss: 0.7685455413550067
(Iteration 9801 / 23430) loss: 0.8365666300808634
(Iteration 9901 / 23430) loss: 0.5472937444319803
(Iteration 10001 / 23430) loss: 0.39281397835645593
(Iteration 10101 / 23430) loss: 0.9042858055728241
(Iteration 10201 / 23430) loss: 0.5215174219146037
(Iteration 10301 / 23430) loss: 0.8036634105817274
(Iteration 10401 / 23430) loss: 0.4471725062129019
(Iteration 10501 / 23430) loss: 0.7963347583774952
(Iteration 10601 / 23430) loss: 0.6222351716339142
(Iteration 10701 / 23430) loss: 0.6603621062173933
(Iteration 10801 / 23430) loss: 0.8267808007969507
(Iteration 10901 / 23430) loss: 0.589404505471192
(Epoch 7 / 15) Training Accuracy: 0.78824, Validation Accuracy: 0.
7528400368437212
(Iteration 11001 / 23430) loss: 0.6436157211391498
(Iteration 11101 / 23430) loss: 0.749912225376419
(Iteration 11201 / 23430) loss: 1.2470031042928267
(Iteration 11301 / 23430) loss: 0.5381293207782023
(Iteration 11401 / 23430) loss: 0.8450844168886784
(Iteration 11501 / 23430) loss: 0.8255687885107703
(Iteration 11601 / 23430) loss: 0.3417986216800297
(Iteration 11701 / 23430) loss: 0.7493698619911429
(Iteration 11801 / 23430) loss: 1.0197973236236866
(Iteration 11901 / 23430) loss: 0.7219677977246028
(Iteration 12001 / 23430) loss: 0.24544192733530062
(Iteration 12101 / 23430) loss: 0.7476266146188114
(Iteration 12201 / 23430) loss: 0.37851579706327537
(Iteration 12301 / 23430) loss: 0.7128168891793416
(Iteration 12401 / 23430) loss: 0.7990866820613781
(Epoch 8 / 15) Training Accuracy: 0.8148, Validation Accuracy: 0.7
817009517961314
(Iteration 12501 / 23430) loss: 0.2054061573016678
(Iteration 12601 / 23430) loss: 0.4808254642078162
(Iteration 12701 / 23430) loss: 0.6669327256071251
(Iteration 12801 / 23430) loss: 0.3040797305783195
(Iteration 12901 / 23430) loss: 0.8086116343619314
(Iteration 13001 / 23430) loss: 0.8821354434851018
(Iteration 13101 / 23430) loss: 0.9365006166756629
(Iteration 13201 / 23430) loss: 0.5436601379211075
(Iteration 13301 / 23430) loss: 0.5551506253173761
(Iteration 13401 / 23430) loss: 0.28235594296232436
(Iteration 13501 / 23430) loss: 0.7477539818439073
```

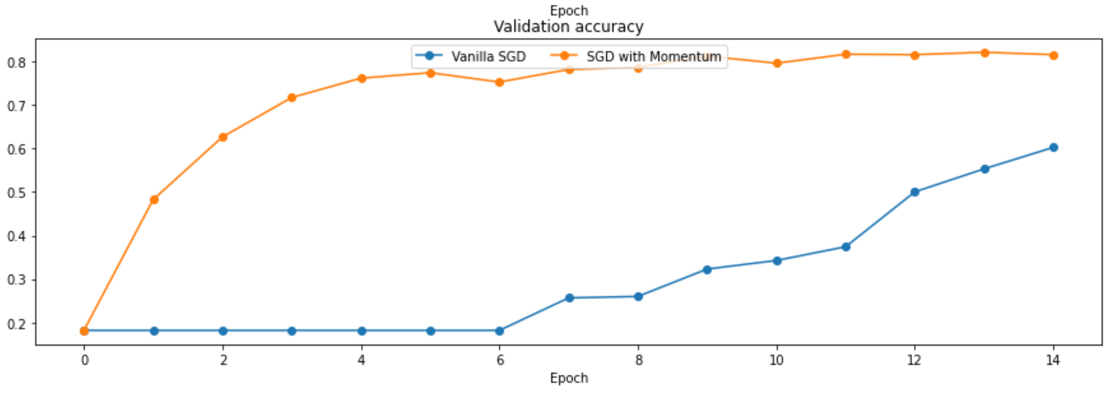
```
(Iteration 13601 / 23430) loss: 0.524932579751043
(Iteration 13701 / 23430) loss: 0.34978573521207235
(Iteration 13801 / 23430) loss: 0.515405815224768
(Iteration 13901 / 23430) loss: 0.6473355580212901
(Iteration 14001 / 23430) loss: 0.5440957306279663
(Epoch 9 / 15) Training Accuracy: 0.81674, Validation Accuracy: 0.
7869204789683758
(Iteration 14101 / 23430) loss: 0.6535034703834394
(Iteration 14201 / 23430) loss: 0.49209172413708224
(Iteration 14301 / 23430) loss: 0.492129517108089
(Iteration 14401 / 23430) loss: 0.5854936782529379
(Iteration 14501 / 23430) loss: 0.6101868071637253
(Iteration 14601 / 23430) loss: 0.6495729299228306
(Iteration 14701 / 23430) loss: 0.34326361828067975
(Iteration 14801 / 23430) loss: 0.25936697810849646
(Iteration 14901 / 23430) loss: 0.2927932510049937
(Iteration 15001 / 23430) loss: 0.5403665851507681
(Iteration 15101 / 23430) loss: 0.3035447984626038
(Iteration 15201 / 23430) loss: 0.7277879694434781
(Iteration 15301 / 23430) loss: 0.5065740510664308
(Iteration 15401 / 23430) loss: 0.865611720433287
(Iteration 15501 / 23430) loss: 0.13561011604096948
(Iteration 15601 / 23430) loss: 0.3705930660247007
(Epoch 10 / 15) Training Accuracy: 0.8526, Validation Accuracy: 0.
8124040528093337
(Iteration 15701 / 23430) loss: 0.45623304266973125
(Iteration 15801 / 23430) loss: 0.5919982510442243
(Iteration 15901 / 23430) loss: 0.3186182237126293
(Iteration 16001 / 23430) loss: 0.4259937160891114
(Iteration 16101 / 23430) loss: 0.3032687338127255
(Iteration 16201 / 23430) loss: 0.5679334025797981
(Iteration 16301 / 23430) loss: 0.3883899513295153
(Iteration 16401 / 23430) loss: 0.40665568908192473
(Iteration 16501 / 23430) loss: 0.321403132626394
(Iteration 16601 / 23430) loss: 0.3971367651632598
(Iteration 16701 / 23430) loss: 0.36181188685067683
(Iteration 16801 / 23430) loss: 0.396294768031014
(Iteration 16901 / 23430) loss: 0.5544105935994112
(Iteration 17001 / 23430) loss: 0.7597416041772647
(Iteration 17101 / 23430) loss: 0.5671881606764628
(Epoch 11 / 15) Training Accuracy: 0.8437, Validation Accuracy: 0.
7961314092723365
(Iteration 17201 / 23430) loss: 0.43850995197445775
(Iteration 17301 / 23430) loss: 0.576861741210939
(Iteration 17401 / 23430) loss: 0.35313086489278644
(Iteration 17501 / 23430) loss: 0.28181474199535905
(Iteration 17601 / 23430) loss: 0.48890517562428454
(Iteration 17701 / 23430) loss: 0.3457139367672221
(Iteration 17801 / 23430) loss: 0.35854516652181057
(Iteration 17901 / 23430) loss: 0.19925750161804368
(Iteration 18001 / 23430) loss: 0.7145500094024216
(Iteration 18101 / 23430) loss: 0.4704006429201924
(Iteration 18201 / 23430) loss: 0.3723212326039296
(Iteration 18301 / 23430) loss: 0.45683112897606915
(Iteration 18401 / 23430) loss: 0.17262738917857348
(Iteration 18501 / 23430) loss: 0.4144940311832284
(Iteration 18601 / 23430) loss: 0.7647075170597613
(Iteration 18701 / 23430) loss: 0.2687127568614884
(Epoch 12 / 15) Training Accuracy: 0.86584, Validation Accuracy:
0.8167024869511821
```

```
(Iteration 18801 / 23430) loss: 0.6268167664094798
(Iteration 18901 / 23430) loss: 0.31920923063409573
(Iteration 19001 / 23430) loss: 0.23233062117761188
(Iteration 19101 / 23430) loss: 0.5027727672378303
(Iteration 19201 / 23430) loss: 0.5612191449009536
(Iteration 19301 / 23430) loss: 0.4061725130938276
(Iteration 19401 / 23430) loss: 0.6057815922497215
(Iteration 19501 / 23430) loss: 0.6273311916956299
(Iteration 19601 / 23430) loss: 0.32240777742795734
(Iteration 19701 / 23430) loss: 0.7408989958104589
(Iteration 19801 / 23430) loss: 0.5517457024052383
(Iteration 19901 / 23430) loss: 0.20523465761871232
(Iteration 20001 / 23430) loss: 0.6143609374508561
(Iteration 20101 / 23430) loss: 0.7554016201803647
(Iteration 20201 / 23430) loss: 0.5673074949436892
(Iteration 20301 / 23430) loss: 0.6729989128827603
(Epoch 13 / 15) Training Accuracy: 0.86484, Validation Accuracy:
0.815781393920786
(Iteration 20401 / 23430) loss: 0.45979521679525054
(Iteration 20501 / 23430) loss: 0.3807507870573985
(Iteration 20601 / 23430) loss: 0.2711743049715404
(Iteration 20701 / 23430) loss: 1.1214333450612943
(Iteration 20801 / 23430) loss: 0.34163610885198203
(Iteration 20901 / 23430) loss: 0.4416225224820496
(Iteration 21001 / 23430) loss: 0.39751649801704786
(Iteration 21101 / 23430) loss: 0.5747294079104199
(Iteration 21201 / 23430) loss: 0.48907005584113833
(Iteration 21301 / 23430) loss: 0.31728259039191836
(Iteration 21401 / 23430) loss: 0.24858691187867188
(Iteration 21501 / 23430) loss: 0.6365046413400239
(Iteration 21601 / 23430) loss: 0.4424719569098372
(Iteration 21701 / 23430) loss: 0.8771648699270378
(Iteration 21801 / 23430) loss: 0.42864707242566935
(Epoch 14 / 15) Training Accuracy: 0.87568, Validation Accuracy:
0.8213079521031624
(Iteration 21901 / 23430) loss: 0.9321696010107827
(Iteration 22001 / 23430) loss: 0.5230719821191983
(Iteration 22101 / 23430) loss: 0.4254435899728492
(Iteration 22201 / 23430) loss: 0.39474109253555295
(Iteration 22301 / 23430) loss: 0.30907583749506906
(Iteration 22401 / 23430) loss: 0.23693415779386376
(Iteration 22501 / 23430) loss: 0.338534230418711
(Iteration 22601 / 23430) loss: 0.3917520488308242
(Iteration 22701 / 23430) loss: 0.614524531503239
(Iteration 22801 / 23430) loss: 0.5523313566310556
(Iteration 22901 / 23430) loss: 0.5126280822331301
(Iteration 23001 / 23430) loss: 0.48765506831545835
(Iteration 23101 / 23430) loss: 0.7970001243249464
(Iteration 23201 / 23430) loss: 0.5664391307643535
(Iteration 23301 / 23430) loss: 0.34314433728767174
(Iteration 23401 / 23430) loss: 0.36543198648538755
```

```
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:42: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:44: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:46: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:49: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:51: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:53: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
```







RMSProp [2pt]

The update rule of RMSProp is as shown below:

$$\begin{aligned}
\gamma &: \text{decay rate} \\
\epsilon &: \text{small number} \\
g_t^2 &: \text{squared gradients} \\
\eta &: \text{learning rate} \\
E[g^2]_t &: \text{decaying average of past squared gradients at update step } t \\
E[g^2]_t &= \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2 \\
\theta_{t+1} &= \theta_t - \frac{\eta \nabla_\theta J(\theta)}{\sqrt{E[g^2]_t + \epsilon}}
\end{aligned}$$

Complete the `RMSProp()` function in `lib/optim.py`

In [ ]:

```
%reload_ext autoreload

seed = 123
np.random.seed(seed=seed)

# Test RMSProp implementation; you should see errors less than 1e-
7
N, D = 4, 5
test_rms = sequential(fc(N, D, name="rms_fc"))

w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
cache = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)

test_rms.layers[0].params = {"rms_fc_w": w}
test_rms.layers[0].grads = {"rms_fc_w": dw}

opt_rms = RMSProp(test_rms, 1e-2, 0.99)
opt_rms.cache = {"rms_fc_w": cache}
opt_rms.step()

updated_w = test_rms.layers[0].params["rms_fc_w"]
cache = opt_rms.cache["rms_fc_w"]

expected_updated_w = np.asarray([
  [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.1846724
7],
  [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.0751577
4],
  [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.3353244
7],
  [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.5957661
9]])
expected_cache = np.asarray([
  [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.6580432
1],
  [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.7348437
7],
  [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.8130293
6],
  [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926
]])

print ('The following errors should be around or less than 1e-7')
print ('updated_w error: ', rel_error(expected_updated_w, updated_
w))
print ('cache error: ', rel_error(expected_cache, opt_rms.cache["r
ms_fc_w"]))
```

```
The following errors should be around or less than 1e-7
updated_w error:  9.502645229894295e-08
cache error:  2.6477955807156126e-09
```

Adam [2pt]

The update rule of Adam is as shown below:

$$\begin{aligned}
t &= t + 1 \\
g_t &: \text{gradients at update step } t \\
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \\
\hat{m}_t &= m_t / (1 - \beta_1^t) \\
\hat{v}_t &= v_t / (1 - \beta_2^t) \\
\theta_{t+1} &= \theta_t - \frac{\eta \, \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}$$

Complete the `Adam()` function in `lib/optim.py` Important Notes: 1) t must be updated before everything else 2) β_1^t is β_1 exponentiated to the t'th power

In [ ]:
```
%reload_ext autoreload

seed = 123
np.random.seed(seed=seed)

# Test Adam implementation; you should see errors around 1e-7 or l
ess
N, D = 4, 5
test_adam = sequential(fc(N, D, name="adam_fc"))

w = np.linspace(-0.4, 0.6, num=N*D).reshape(N, D)
dw = np.linspace(-0.6, 0.4, num=N*D).reshape(N, D)
m = np.linspace(0.6, 0.9, num=N*D).reshape(N, D)
v = np.linspace(0.7, 0.5, num=N*D).reshape(N, D)

test_adam.layers[0].params = {"adam_fc_w": w}
test_adam.layers[0].grads = {"adam_fc_w": dw}

opt_adam = Adam(test_adam, 1e-2, 0.9, 0.999, t=5)
opt_adam.mt = {"adam_fc_w": m}
opt_adam.vt = {"adam_fc_w": v}
opt_adam.step()

updated_w = test_adam.layers[0].params["adam_fc_w"]
mt = opt_adam.mt["adam_fc_w"]
vt = opt_adam.vt["adam_fc_w"]

expected_updated_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.1906097
7],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.072292
9],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.3351696
9],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.5980145
9]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.6573885
3,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.6046738
5,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.5520976
7,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,
]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.5578947
4],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.6552631
6],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.7526315
8],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85
]])

print ('The following errors should be around or less than 1e-7')
print ('updated_w error: ', rel_error(expected_updated_w, updated_
w))
print ('mt error: ', rel_error(expected_m, mt))
print ('vt error: ', rel_error(expected_v, vt))
```

```
The following errors should be around or less than 1e-7
updated_w error:  1.1395691798535431e-07
mt error:  4.214963193114416e-09
vt error:  4.208314038113071e-09
```

Comparing the optimizers [4pt]

Run the following code block to compare the plotted results among all the above optimizers. You should see SGD with Momentum, RMSProp, and Adam optimizers work better than Vanilla SGD optimizer.

```
In [ ]: seed = 123
        np.random.seed(seed=seed)

        model_rms      = FullyConnectedNetwork()
        model_adam     = FullyConnectedNetwork()
        loss_f_rms     = cross_entropy()
        loss_f_adam    = cross_entropy()
        optimizer_rms  = RMSProp(model_rms.net, 5e-4)
        optimizer_adam = Adam(model_adam.net, 5e-4)

        print ("Training with RMSProp...")
        results_rms = train_net(small_data_dict, model_rms, loss_f_rms, op
        timizer_rms, batch_size=32,
                                max_epochs=15, show_every=100, verbose=Tru
        e)

        print ("\nTraining with Adam...")
        results_adam = train_net(small_data_dict, model_adam, loss_f_adam,
        optimizer_adam, batch_size=32,
                                 max_epochs=15, show_every=100, verbose=Tr
        ue)

        opt_params_rms,  loss_hist_rms,  train_acc_hist_rms,  val_acc_hist
        _rms  = results_rms
        opt_params_adam, loss_hist_adam, train_acc_hist_adam, val_acc_hist
        _adam = results_adam

        plt.subplot(3, 1, 1)
        plt.title('Training loss')
        plt.xlabel('Iteration')

        plt.subplot(3, 1, 2)
        plt.title('Training accuracy')
        plt.xlabel('Epoch')

        plt.subplot(3, 1, 3)
        plt.title('Validation accuracy')
        plt.xlabel('Epoch')

        plt.subplot(3, 1, 1)
        plt.plot(loss_hist_sgd, 'o', label="Vanilla SGD")
        plt.subplot(3, 1, 2)
        plt.plot(train_acc_hist_sgd, '-o', label="Vanilla SGD")
        plt.subplot(3, 1, 3)
        plt.plot(val_acc_hist_sgd, '-o', label="Vanilla SGD")

        plt.subplot(3, 1, 1)
        plt.plot(loss_hist_sgdm, 'o', label="SGD with Momentum")
        plt.subplot(3, 1, 2)
        plt.plot(train_acc_hist_sgdm, '-o', label="SGD with Momentum")
        plt.subplot(3, 1, 3)
        plt.plot(val_acc_hist_sgdm, '-o', label="SGD with Momentum")

        plt.subplot(3, 1, 1)
        plt.plot(loss_hist_rms, 'o', label="RMSProp")
        plt.subplot(3, 1, 2)
        plt.plot(train_acc_hist_rms, '-o', label="RMSProp")
        plt.subplot(3, 1, 3)
        plt.plot(val_acc_hist_rms, '-o', label="RMSProp")
plt.subplot(3, 1, 1)
plt.plot(loss_hist_adam, 'o', label="Adam")
plt.subplot(3, 1, 2)
plt.plot(train_acc_hist_adam, '-o', label="Adam")
plt.subplot(3, 1, 3)
plt.plot(val_acc_hist_adam, '-o', label="Adam")

for i in [1, 2, 3]:
  plt.subplot(3, 1, i)
  plt.legend(loc='upper center', ncol=4)
plt.gcf().set_size_inches(15, 15)
plt.show()
```

```
Training with RMSProp...
(Iteration 1 / 23430) loss: 2.302024602405127
(Iteration 101 / 23430) loss: 2.294067022297631
(Iteration 201 / 23430) loss: 1.893342872503586
(Iteration 301 / 23430) loss: 1.759470224381714
(Iteration 401 / 23430) loss: 1.9503735310840347
(Iteration 501 / 23430) loss: 2.2738583540034556
(Iteration 601 / 23430) loss: 1.7274812520456964
(Iteration 701 / 23430) loss: 1.6806808255188614
(Iteration 801 / 23430) loss: 1.6672283455389896
(Iteration 901 / 23430) loss: 1.7881124837389342
(Iteration 1001 / 23430) loss: 1.3646501000786178
(Iteration 1101 / 23430) loss: 1.4273528384828267
(Iteration 1201 / 23430) loss: 1.6809708883056256
(Iteration 1301 / 23430) loss: 1.1991114771159426
(Iteration 1401 / 23430) loss: 1.1832867582205846
(Iteration 1501 / 23430) loss: 1.3808528439838617
(Epoch 1 / 15) Training Accuracy: 0.59628, Validation Accuracy: 0.
578753454098864
(Iteration 1601 / 23430) loss: 1.0052556697343498
(Iteration 1701 / 23430) loss: 1.2379333923390325
(Iteration 1801 / 23430) loss: 0.8143512049584081
(Iteration 1901 / 23430) loss: 0.9190943445876717
(Iteration 2001 / 23430) loss: 1.0630680460369188
(Iteration 2101 / 23430) loss: 1.0977704520428972
(Iteration 2201 / 23430) loss: 1.0259748091119232
(Iteration 2301 / 23430) loss: 1.1277537029882498
(Iteration 2401 / 23430) loss: 1.0516233791546077
(Iteration 2501 / 23430) loss: 1.0517859732243546
(Iteration 2601 / 23430) loss: 1.1145073918731958
(Iteration 2701 / 23430) loss: 1.1267738548271677
(Iteration 2801 / 23430) loss: 0.8018503116461642
(Iteration 2901 / 23430) loss: 0.6462095232236672
(Iteration 3001 / 23430) loss: 1.2576112977471636
(Iteration 3101 / 23430) loss: 1.1083835909544069
(Epoch 2 / 15) Training Accuracy: 0.7151, Validation Accuracy: 0.6
908197727970525
(Iteration 3201 / 23430) loss: 0.6635912214672188
(Iteration 3301 / 23430) loss: 0.5854400561402133
(Iteration 3401 / 23430) loss: 1.039351291243604
(Iteration 3501 / 23430) loss: 0.7022455027088499
(Iteration 3601 / 23430) loss: 0.560373779542704
(Iteration 3701 / 23430) loss: 0.6897449090796519
(Iteration 3801 / 23430) loss: 0.7940649847342381
(Iteration 3901 / 23430) loss: 1.079932640526784
(Iteration 4001 / 23430) loss: 1.294823270241326
(Iteration 4101 / 23430) loss: 0.697303569270555
(Iteration 4201 / 23430) loss: 0.8430210923497083
(Iteration 4301 / 23430) loss: 0.8869295503461858
(Iteration 4401 / 23430) loss: 0.8813039863106478
(Iteration 4501 / 23430) loss: 1.0635678669587185
(Iteration 4601 / 23430) loss: 0.9770414898258765
(Epoch 3 / 15) Training Accuracy: 0.75066, Validation Accuracy: 0.
7279705250230273
(Iteration 4701 / 23430) loss: 0.4926941797288628
(Iteration 4801 / 23430) loss: 0.8997039453662613
(Iteration 4901 / 23430) loss: 0.9474659732301252
(Iteration 5001 / 23430) loss: 0.4428209301166273
(Iteration 5101 / 23430) loss: 0.7804558159717145
(Iteration 5201 / 23430) loss: 0.8860585365584718
```

```
(Iteration 5301 / 23430) loss: 0.8876279225142616
(Iteration 5401 / 23430) loss: 0.7461874926876416
(Iteration 5501 / 23430) loss: 0.47194205901811725
(Iteration 5601 / 23430) loss: 0.3868400640250502
(Iteration 5701 / 23430) loss: 0.6995661107878017
(Iteration 5801 / 23430) loss: 1.0156554244619713
(Iteration 5901 / 23430) loss: 0.5346241260481349
(Iteration 6001 / 23430) loss: 0.7327152732077715
(Iteration 6101 / 23430) loss: 0.4348859916484709
(Iteration 6201 / 23430) loss: 0.77158142605784
(Epoch 4 / 15) Training Accuracy: 0.76888, Validation Accuracy: 0.
7442431685600246
(Iteration 6301 / 23430) loss: 0.9418467990367398
(Iteration 6401 / 23430) loss: 0.5388115494330443
(Iteration 6501 / 23430) loss: 0.591878860426356
(Iteration 6601 / 23430) loss: 0.7036796453477484
(Iteration 6701 / 23430) loss: 0.6290645215406748
(Iteration 6801 / 23430) loss: 0.5395836922625337
(Iteration 6901 / 23430) loss: 0.6660227413159074
(Iteration 7001 / 23430) loss: 0.9170356814312041
(Iteration 7101 / 23430) loss: 0.6353602314787111
(Iteration 7201 / 23430) loss: 0.7312045059157951
(Iteration 7301 / 23430) loss: 0.7547274440641598
(Iteration 7401 / 23430) loss: 0.8848942669369897
(Iteration 7501 / 23430) loss: 0.7746887819006336
(Iteration 7601 / 23430) loss: 0.6169953440095266
(Iteration 7701 / 23430) loss: 0.36479212379043163
(Iteration 7801 / 23430) loss: 0.4504958533910726
(Epoch 5 / 15) Training Accuracy: 0.79668, Validation Accuracy: 0.
7657353392692662
(Iteration 7901 / 23430) loss: 0.569083477782625
(Iteration 8001 / 23430) loss: 0.45104379676556794
(Iteration 8101 / 23430) loss: 0.5274783840844529
(Iteration 8201 / 23430) loss: 0.4521490758893326
(Iteration 8301 / 23430) loss: 0.7060777210733378
(Iteration 8401 / 23430) loss: 0.5875982555920503
(Iteration 8501 / 23430) loss: 0.7274344938715332
(Iteration 8601 / 23430) loss: 0.8212427423035693
(Iteration 8701 / 23430) loss: 0.8889450275473454
(Iteration 8801 / 23430) loss: 0.3964340775112732
(Iteration 8901 / 23430) loss: 0.6038593503619795
(Iteration 9001 / 23430) loss: 0.7633686176636664
(Iteration 9101 / 23430) loss: 0.4532081025025466
(Iteration 9201 / 23430) loss: 0.4167013318645028
(Iteration 9301 / 23430) loss: 0.7952831608977783
(Epoch 6 / 15) Training Accuracy: 0.80864, Validation Accuracy: 0.
7688056493705864
(Iteration 9401 / 23430) loss: 0.4334185642781313
(Iteration 9501 / 23430) loss: 0.6220117433415824
(Iteration 9601 / 23430) loss: 0.6329021171593973
(Iteration 9701 / 23430) loss: 0.3978822046923507
(Iteration 9801 / 23430) loss: 0.36616554118139016
(Iteration 9901 / 23430) loss: 0.8116032355584111
(Iteration 10001 / 23430) loss: 0.8675740429289918
(Iteration 10101 / 23430) loss: 0.4906492980461738
(Iteration 10201 / 23430) loss: 0.4033115046159252
(Iteration 10301 / 23430) loss: 0.6631065952481906
(Iteration 10401 / 23430) loss: 0.47172273075927
(Iteration 10501 / 23430) loss: 0.8295400486928853
(Iteration 10601 / 23430) loss: 0.6960984589366885
```

```
(Iteration 10701 / 23430) loss: 0.44952127626724375
(Iteration 10801 / 23430) loss: 0.4791530213767127
(Iteration 10901 / 23430) loss: 0.8639922977200124
(Epoch 7 / 15) Training Accuracy: 0.8259, Validation Accuracy: 0.7
924470371507523
(Iteration 11001 / 23430) loss: 0.3972655678416329
(Iteration 11101 / 23430) loss: 0.4958313802605545
(Iteration 11201 / 23430) loss: 0.9896846500236981
(Iteration 11301 / 23430) loss: 0.42499392686126863
(Iteration 11401 / 23430) loss: 0.611164027269645
(Iteration 11501 / 23430) loss: 1.043838792872848
(Iteration 11601 / 23430) loss: 0.9453959778126512
(Iteration 11701 / 23430) loss: 0.36313040065218366
(Iteration 11801 / 23430) loss: 0.2712158206695513
(Iteration 11901 / 23430) loss: 0.6572288353709436
(Iteration 12001 / 23430) loss: 0.426475216721054
(Iteration 12101 / 23430) loss: 0.38392019974430114
(Iteration 12201 / 23430) loss: 0.47747270321502044
(Iteration 12301 / 23430) loss: 0.3761936997391016
(Iteration 12401 / 23430) loss: 0.8582029441391886
(Epoch 8 / 15) Training Accuracy: 0.8373, Validation Accuracy: 0.7
9060485108996
(Iteration 12501 / 23430) loss: 0.4760155032481812
(Iteration 12601 / 23430) loss: 0.332264837677524
(Iteration 12701 / 23430) loss: 0.48114188932072927
(Iteration 12801 / 23430) loss: 0.6584945806393824
(Iteration 12901 / 23430) loss: 0.5997117763889472
(Iteration 13001 / 23430) loss: 0.4555241641716758
(Iteration 13101 / 23430) loss: 0.47367910973910565
(Iteration 13201 / 23430) loss: 0.38113043140618585
(Iteration 13301 / 23430) loss: 0.6454803175878453
(Iteration 13401 / 23430) loss: 0.2572016095043697
(Iteration 13501 / 23430) loss: 0.6383205149478877
(Iteration 13601 / 23430) loss: 0.45391218120239185
(Iteration 13701 / 23430) loss: 0.49998866988448765
(Iteration 13801 / 23430) loss: 0.6531505230332214
(Iteration 13901 / 23430) loss: 0.8398567273619184
(Iteration 14001 / 23430) loss: 0.6328387896531318
(Epoch 9 / 15) Training Accuracy: 0.85176, Validation Accuracy: 0.
7982806263432607
(Iteration 14101 / 23430) loss: 0.4149135831438119
(Iteration 14201 / 23430) loss: 0.7708854861712671
(Iteration 14301 / 23430) loss: 0.40024148127941983
(Iteration 14401 / 23430) loss: 0.3988268669555248
(Iteration 14501 / 23430) loss: 0.6836762352987622
(Iteration 14601 / 23430) loss: 0.4188064814649351
(Iteration 14701 / 23430) loss: 0.29142059623578653
(Iteration 14801 / 23430) loss: 0.3980272728802592
(Iteration 14901 / 23430) loss: 0.23984048313966483
(Iteration 15001 / 23430) loss: 0.36113495288147446
(Iteration 15101 / 23430) loss: 0.33896921106313976
(Iteration 15201 / 23430) loss: 0.519586469489863
(Iteration 15301 / 23430) loss: 0.5854906223570151
(Iteration 15401 / 23430) loss: 0.4427026158415718
(Iteration 15501 / 23430) loss: 0.5008039078062629
(Iteration 15601 / 23430) loss: 0.29384060432542397
(Epoch 10 / 15) Training Accuracy: 0.85476, Validation Accuracy:
0.8096407737181456
(Iteration 15701 / 23430) loss: 0.37527495035721875
(Iteration 15801 / 23430) loss: 0.7116441542545004
```

```
(Iteration 15901 / 23430) loss: 0.45596573250577577
(Iteration 16001 / 23430) loss: 0.48066583640179766
(Iteration 16101 / 23430) loss: 0.4006977501914045
(Iteration 16201 / 23430) loss: 0.6396420282123818
(Iteration 16301 / 23430) loss: 0.5501850864282193
(Iteration 16401 / 23430) loss: 0.2939958114003793
(Iteration 16501 / 23430) loss: 0.15653694403346555
(Iteration 16601 / 23430) loss: 0.2844788981989303
(Iteration 16701 / 23430) loss: 0.8178016522440898
(Iteration 16801 / 23430) loss: 0.11640961348013157
(Iteration 16901 / 23430) loss: 0.26157970208916803
(Iteration 17001 / 23430) loss: 0.3811662244737053
(Iteration 17101 / 23430) loss: 0.516444162154309
(Epoch 11 / 15) Training Accuracy: 0.83722, Validation Accuracy:
0.7875345409886398
(Iteration 17201 / 23430) loss: 0.4352891627423372
(Iteration 17301 / 23430) loss: 0.5932155689286975
(Iteration 17401 / 23430) loss: 0.28383361138834295
(Iteration 17501 / 23430) loss: 0.5388864090608463
(Iteration 17601 / 23430) loss: 0.540933637032997
(Iteration 17701 / 23430) loss: 0.5303415152740917
(Iteration 17801 / 23430) loss: 0.5116298489679945
(Iteration 17901 / 23430) loss: 0.6660777165084617
(Iteration 18001 / 23430) loss: 0.3893896682692089
(Iteration 18101 / 23430) loss: 0.8347372077646613
(Iteration 18201 / 23430) loss: 0.9279596652331207
(Iteration 18301 / 23430) loss: 0.4644552058583452
(Iteration 18401 / 23430) loss: 0.5547346566518084
(Iteration 18501 / 23430) loss: 0.18441430557629782
(Iteration 18601 / 23430) loss: 0.8316744630444145
(Iteration 18701 / 23430) loss: 0.4244181784308865
(Epoch 12 / 15) Training Accuracy: 0.86158, Validation Accuracy:
0.8087196806877495
(Iteration 18801 / 23430) loss: 0.6190343523765938
(Iteration 18901 / 23430) loss: 0.5669590870958643
(Iteration 19001 / 23430) loss: 0.3657855032284493
(Iteration 19101 / 23430) loss: 0.779831063507824
(Iteration 19201 / 23430) loss: 0.4405451482943167
(Iteration 19301 / 23430) loss: 0.502256401334465
(Iteration 19401 / 23430) loss: 0.29065977096022466
(Iteration 19501 / 23430) loss: 0.3465054584023689
(Iteration 19601 / 23430) loss: 0.3832692978125098
(Iteration 19701 / 23430) loss: 0.5665021935560943
(Iteration 19801 / 23430) loss: 0.7450979599307856
(Iteration 19901 / 23430) loss: 0.6298222206980159
(Iteration 20001 / 23430) loss: 0.39202885253826664
(Iteration 20101 / 23430) loss: 0.24714696931081304
(Iteration 20201 / 23430) loss: 0.2567487781675979
(Iteration 20301 / 23430) loss: 0.6088983191808037
(Epoch 13 / 15) Training Accuracy: 0.86182, Validation Accuracy:
0.803807184525637
(Iteration 20401 / 23430) loss: 0.4666124962638969
(Iteration 20501 / 23430) loss: 0.23286782789768318
(Iteration 20601 / 23430) loss: 0.24779671910501175
(Iteration 20701 / 23430) loss: 0.6689142061139716
(Iteration 20801 / 23430) loss: 0.3034728685475964
(Iteration 20901 / 23430) loss: 0.2653129190801466
(Iteration 21001 / 23430) loss: 0.551620661075795
(Iteration 21101 / 23430) loss: 0.6291386490318298
(Iteration 21201 / 23430) loss: 0.2711199542430054
```

```
(Iteration 21301 / 23430) loss: 0.24846510980219508
(Iteration 21401 / 23430) loss: 0.16967456736236045
(Iteration 21501 / 23430) loss: 0.5054520031077779
(Iteration 21601 / 23430) loss: 0.4541856300060162
(Iteration 21701 / 23430) loss: 0.35744421107390445
(Iteration 21801 / 23430) loss: 0.31182034977671547
(Epoch 14 / 15) Training Accuracy: 0.87314, Validation Accuracy:
0.8139392078599939
(Iteration 21901 / 23430) loss: 0.2610465622296579
(Iteration 22001 / 23430) loss: 0.47443615704944603
(Iteration 22101 / 23430) loss: 0.32001818030498985
(Iteration 22201 / 23430) loss: 0.409092195493904
(Iteration 22301 / 23430) loss: 0.53438783941409
(Iteration 22401 / 23430) loss: 0.377461569731471
(Iteration 22501 / 23430) loss: 0.46761182431607035
(Iteration 22601 / 23430) loss: 0.11014259465352383
(Iteration 22701 / 23430) loss: 0.2646969559966856
(Iteration 22801 / 23430) loss: 0.08558495450130214
(Iteration 22901 / 23430) loss: 0.4201695896557787
(Iteration 23001 / 23430) loss: 0.41472363527284606
(Iteration 23101 / 23430) loss: 0.4331893368058827
(Iteration 23201 / 23430) loss: 0.5098049986527498
(Iteration 23301 / 23430) loss: 0.6836915984166905
(Iteration 23401 / 23430) loss: 0.5480710253458135
(Epoch 15 / 15) Training Accuracy: 0.87596, Validation Accuracy:
0.802886091495241

Training with Adam...
(Iteration 1 / 23430) loss: 2.30213557079308
(Iteration 101 / 23430) loss: 2.3198634724140073
(Iteration 201 / 23430) loss: 1.9618958880522468
(Iteration 301 / 23430) loss: 1.585606631301271
(Iteration 401 / 23430) loss: 1.7896660759748704
(Iteration 501 / 23430) loss: 1.6129098019664687
(Iteration 601 / 23430) loss: 1.7067611315203313
(Iteration 701 / 23430) loss: 1.3276912296797112
(Iteration 801 / 23430) loss: 1.6178010018525204
(Iteration 901 / 23430) loss: 1.5866378855242758
(Iteration 1001 / 23430) loss: 1.4483594240103064
(Iteration 1101 / 23430) loss: 1.2621550801462886
(Iteration 1201 / 23430) loss: 1.3944223287725193
(Iteration 1301 / 23430) loss: 0.9467047331863577
(Iteration 1401 / 23430) loss: 1.163172707832629
(Iteration 1501 / 23430) loss: 1.2007211861998188
(Epoch 1 / 15) Training Accuracy: 0.6424, Validation Accuracy: 0.6
198956094565551
(Iteration 1601 / 23430) loss: 1.1695989450999653
(Iteration 1701 / 23430) loss: 0.6217648972274601
(Iteration 1801 / 23430) loss: 0.7650561017588391
(Iteration 1901 / 23430) loss: 1.0957411862311013
(Iteration 2001 / 23430) loss: 0.7782596242547483
(Iteration 2101 / 23430) loss: 0.7657574413169376
(Iteration 2201 / 23430) loss: 1.0061451360195492
(Iteration 2301 / 23430) loss: 1.156241400121714
(Iteration 2401 / 23430) loss: 1.1999495511878138
(Iteration 2501 / 23430) loss: 0.9426474998953062
(Iteration 2601 / 23430) loss: 0.9122671664616517
(Iteration 2701 / 23430) loss: 1.135344907606744
(Iteration 2801 / 23430) loss: 0.8982893151109528
(Iteration 2901 / 23430) loss: 1.2066398187067189
```

```
(Iteration 3001 / 23430) loss: 0.8229912031919353
(Iteration 3101 / 23430) loss: 1.0419885558822393
(Epoch 2 / 15) Training Accuracy: 0.71848, Validation Accuracy: 0.
7000307031010132
(Iteration 3201 / 23430) loss: 0.8805702818524578
(Iteration 3301 / 23430) loss: 0.8074649208648003
(Iteration 3401 / 23430) loss: 1.1711857861033752
(Iteration 3501 / 23430) loss: 0.7893376121819037
(Iteration 3601 / 23430) loss: 0.8596689189524382
(Iteration 3701 / 23430) loss: 1.0532783579073663
(Iteration 3801 / 23430) loss: 0.6482366271000934
(Iteration 3901 / 23430) loss: 0.9135016691563972
(Iteration 4001 / 23430) loss: 0.5812832872436182
(Iteration 4101 / 23430) loss: 0.6424033272987704
(Iteration 4201 / 23430) loss: 0.7391615866146246
(Iteration 4301 / 23430) loss: 0.9441097017271339
(Iteration 4401 / 23430) loss: 0.8356494442696785
(Iteration 4501 / 23430) loss: 0.9972575000311464
(Iteration 4601 / 23430) loss: 0.45354633652851856
(Epoch 3 / 15) Training Accuracy: 0.76518, Validation Accuracy: 0.
7356463002763279
(Iteration 4701 / 23430) loss: 0.6888480103687873
(Iteration 4801 / 23430) loss: 0.7770372204869729
(Iteration 4901 / 23430) loss: 0.7793976568647571
(Iteration 5001 / 23430) loss: 0.946970749662614
(Iteration 5101 / 23430) loss: 0.35026277257306143
(Iteration 5201 / 23430) loss: 0.9534633648798078
(Iteration 5301 / 23430) loss: 0.7762922653675396
(Iteration 5401 / 23430) loss: 1.4573794417091375
(Iteration 5501 / 23430) loss: 0.6059126193358364
(Iteration 5601 / 23430) loss: 0.8445126523168426
(Iteration 5701 / 23430) loss: 1.0497079551377826
(Iteration 5801 / 23430) loss: 0.6078570243273105
(Iteration 5901 / 23430) loss: 1.083782846728719
(Iteration 6001 / 23430) loss: 0.8135112664553356
(Iteration 6101 / 23430) loss: 0.896242155814864
(Iteration 6201 / 23430) loss: 0.9774523259866581
(Epoch 4 / 15) Training Accuracy: 0.78358, Validation Accuracy: 0.
7580595640159656
(Iteration 6301 / 23430) loss: 0.4289437256704275
(Iteration 6401 / 23430) loss: 0.7757521255610222
(Iteration 6501 / 23430) loss: 0.4849115062975236
(Iteration 6601 / 23430) loss: 0.6062919733817281
(Iteration 6701 / 23430) loss: 1.3224832294546722
(Iteration 6801 / 23430) loss: 0.45395810621748217
(Iteration 6901 / 23430) loss: 0.6418998513744596
(Iteration 7001 / 23430) loss: 0.45030936497286916
(Iteration 7101 / 23430) loss: 1.0260579928928084
(Iteration 7201 / 23430) loss: 0.4426094312248173
(Iteration 7301 / 23430) loss: 0.9010646614206341
(Iteration 7401 / 23430) loss: 0.45555001095114905
(Iteration 7501 / 23430) loss: 0.6683508012939745
(Iteration 7601 / 23430) loss: 0.4956375967228986
(Iteration 7701 / 23430) loss: 0.6099702344737292
(Iteration 7801 / 23430) loss: 0.6090571206374831
(Epoch 5 / 15) Training Accuracy: 0.80616, Validation Accuracy: 0.
7718759594719067
(Iteration 7901 / 23430) loss: 0.7040735466906822
(Iteration 8001 / 23430) loss: 0.507695654534123
(Iteration 8101 / 23430) loss: 0.8977962456630262
```

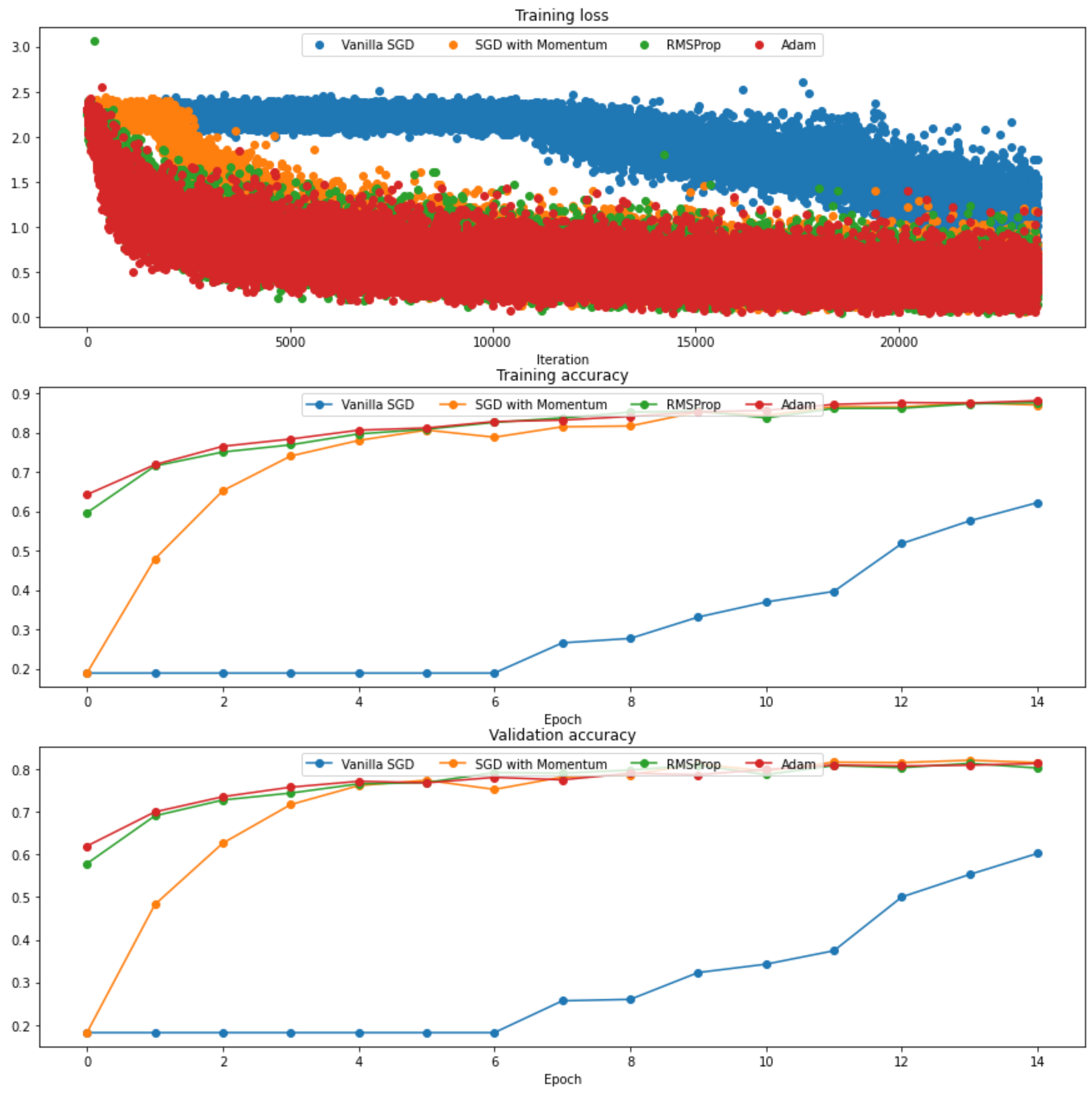
```
(Iteration 8201 / 23430) loss: 0.4157239167201861
(Iteration 8301 / 23430) loss: 0.5494966232440004
(Iteration 8401 / 23430) loss: 0.5696264243688759
(Iteration 8501 / 23430) loss: 0.8731235023319929
(Iteration 8601 / 23430) loss: 0.7048078934095701
(Iteration 8701 / 23430) loss: 0.28630971190359783
(Iteration 8801 / 23430) loss: 0.4195382432672282
(Iteration 8901 / 23430) loss: 0.40169171945313625
(Iteration 9001 / 23430) loss: 0.2200424496874236
(Iteration 9101 / 23430) loss: 0.44585782256514267
(Iteration 9201 / 23430) loss: 0.5831027137002913
(Iteration 9301 / 23430) loss: 0.5930036266949449
(Epoch 6 / 15) Training Accuracy: 0.8118, Validation Accuracy: 0.7
681915873503223
(Iteration 9401 / 23430) loss: 0.7640674034363285
(Iteration 9501 / 23430) loss: 0.410313623643202
(Iteration 9601 / 23430) loss: 0.42238107966814414
(Iteration 9701 / 23430) loss: 0.8515018481577946
(Iteration 9801 / 23430) loss: 0.5751290481736254
(Iteration 9901 / 23430) loss: 0.593071343675579
(Iteration 10001 / 23430) loss: 0.37804010096452156
(Iteration 10101 / 23430) loss: 0.9093237104675852
(Iteration 10201 / 23430) loss: 0.4763315223888851
(Iteration 10301 / 23430) loss: 0.5665155924618216
(Iteration 10401 / 23430) loss: 0.37978925159409294
(Iteration 10501 / 23430) loss: 0.6983321650724948
(Iteration 10601 / 23430) loss: 0.6151784048098548
(Iteration 10701 / 23430) loss: 0.5198568759075132
(Iteration 10801 / 23430) loss: 0.4684209829398247
(Iteration 10901 / 23430) loss: 0.5854311522495167
(Epoch 7 / 15) Training Accuracy: 0.828, Validation Accuracy: 0.78
10868897758674
(Iteration 11001 / 23430) loss: 0.5174574735115963
(Iteration 11101 / 23430) loss: 0.7380495618244347
(Iteration 11201 / 23430) loss: 0.9248628246206372
(Iteration 11301 / 23430) loss: 0.356384845098525
(Iteration 11401 / 23430) loss: 0.6387270341383339
(Iteration 11501 / 23430) loss: 0.6418139763472719
(Iteration 11601 / 23430) loss: 0.282869457592472
(Iteration 11701 / 23430) loss: 0.7280063452960892
(Iteration 11801 / 23430) loss: 0.7314928838370742
(Iteration 11901 / 23430) loss: 0.6724277607003509
(Iteration 12001 / 23430) loss: 0.25409286533644576
(Iteration 12101 / 23430) loss: 0.6704778626901557
(Iteration 12201 / 23430) loss: 0.2739670166148237
(Iteration 12301 / 23430) loss: 0.5141087646202089
(Iteration 12401 / 23430) loss: 0.631778831981185
(Epoch 8 / 15) Training Accuracy: 0.83174, Validation Accuracy: 0.
775253300583359
(Iteration 12501 / 23430) loss: 0.39452422482200333
(Iteration 12601 / 23430) loss: 0.5711682499147753
(Iteration 12701 / 23430) loss: 0.47279206753638703
(Iteration 12801 / 23430) loss: 0.28469809236780075
(Iteration 12901 / 23430) loss: 0.7571741422192575
(Iteration 13001 / 23430) loss: 0.757168996335186
(Iteration 13101 / 23430) loss: 0.6916813057421901
(Iteration 13201 / 23430) loss: 0.5969435362584367
(Iteration 13301 / 23430) loss: 0.4670087161175012
(Iteration 13401 / 23430) loss: 0.38956647062001915
(Iteration 13501 / 23430) loss: 0.6501182236980455
```

```
(Iteration 13601 / 23430) loss: 0.3843499877614884
(Iteration 13701 / 23430) loss: 0.26349074465146605
(Iteration 13801 / 23430) loss: 0.5864134199024049
(Iteration 13901 / 23430) loss: 0.4470838676328068
(Iteration 14001 / 23430) loss: 0.38236524228709484
(Epoch 9 / 15) Training Accuracy: 0.84118, Validation Accuracy: 0.
7909118821000921
(Iteration 14101 / 23430) loss: 0.49533598385866534
(Iteration 14201 / 23430) loss: 0.3888682344558458
(Iteration 14301 / 23430) loss: 0.3003842109358964
(Iteration 14401 / 23430) loss: 0.5478919085054866
(Iteration 14501 / 23430) loss: 0.5747485100277976
(Iteration 14601 / 23430) loss: 0.7627803226923896
(Iteration 14701 / 23430) loss: 0.28426358894943793
(Iteration 14801 / 23430) loss: 0.2514330618639701
(Iteration 14901 / 23430) loss: 0.2757663365722198
(Iteration 15001 / 23430) loss: 0.4477010963217078
(Iteration 15101 / 23430) loss: 0.4144170988682023
(Iteration 15201 / 23430) loss: 0.5407896601059753
(Iteration 15301 / 23430) loss: 0.5618385717393704
(Iteration 15401 / 23430) loss: 0.9249247843620947
(Iteration 15501 / 23430) loss: 0.22794788225959098
(Iteration 15601 / 23430) loss: 0.3673478178482168
(Epoch 10 / 15) Training Accuracy: 0.85262, Validation Accuracy:
0.7872275099785079
(Iteration 15701 / 23430) loss: 0.3709280527930072
(Iteration 15801 / 23430) loss: 0.5215914438265492
(Iteration 15901 / 23430) loss: 0.3463431918264248
(Iteration 16001 / 23430) loss: 0.3295875972234179
(Iteration 16101 / 23430) loss: 0.22011875285669882
(Iteration 16201 / 23430) loss: 0.3072974848643033
(Iteration 16301 / 23430) loss: 0.34442489710739616
(Iteration 16401 / 23430) loss: 0.40641301882430014
(Iteration 16501 / 23430) loss: 0.22755732498023745
(Iteration 16601 / 23430) loss: 0.3339140240568978
(Iteration 16701 / 23430) loss: 0.3349967042121684
(Iteration 16801 / 23430) loss: 0.44586050288138435
(Iteration 16901 / 23430) loss: 0.7275430135703655
(Iteration 17001 / 23430) loss: 0.7017365239633362
(Iteration 17101 / 23430) loss: 0.6444414140274795
(Epoch 11 / 15) Training Accuracy: 0.85656, Validation Accuracy:
0.7998157813939207
(Iteration 17201 / 23430) loss: 0.40034944435665787
(Iteration 17301 / 23430) loss: 0.429781566599492
(Iteration 17401 / 23430) loss: 0.365629187599824
(Iteration 17501 / 23430) loss: 0.1804573344598709
(Iteration 17601 / 23430) loss: 0.707525664718276
(Iteration 17701 / 23430) loss: 0.22698575885789604
(Iteration 17801 / 23430) loss: 0.3666810970487362
(Iteration 17901 / 23430) loss: 0.11605046578973331
(Iteration 18001 / 23430) loss: 0.8066506815778114
(Iteration 18101 / 23430) loss: 0.37130468108006753
(Iteration 18201 / 23430) loss: 0.48014047961094214
(Iteration 18301 / 23430) loss: 0.6482290830915097
(Iteration 18401 / 23430) loss: 0.14159204243638912
(Iteration 18501 / 23430) loss: 0.35339374674771135
(Iteration 18601 / 23430) loss: 0.6717047614626908
(Iteration 18701 / 23430) loss: 0.11964197183459875
(Epoch 12 / 15) Training Accuracy: 0.8717, Validation Accuracy: 0.
8099478047282775
```

```
(Iteration 18801 / 23430) loss: 0.4586146339097037
(Iteration 18901 / 23430) loss: 0.4009992206096246
(Iteration 19001 / 23430) loss: 0.2376784027949182
(Iteration 19101 / 23430) loss: 0.5388923672046414
(Iteration 19201 / 23430) loss: 0.4887996963931341
(Iteration 19301 / 23430) loss: 0.47505632777197404
(Iteration 19401 / 23430) loss: 0.6821725728219338
(Iteration 19501 / 23430) loss: 0.7301504053348807
(Iteration 19601 / 23430) loss: 0.23438690626365766
(Iteration 19701 / 23430) loss: 0.6256301091904323
(Iteration 19801 / 23430) loss: 0.47590248100548765
(Iteration 19901 / 23430) loss: 0.2893583468241379
(Iteration 20001 / 23430) loss: 0.5441848572476902
(Iteration 20101 / 23430) loss: 0.8279212783861876
(Iteration 20201 / 23430) loss: 0.6159762253971566
(Iteration 20301 / 23430) loss: 0.4527874958010594
(Epoch 13 / 15) Training Accuracy: 0.8764, Validation Accuracy: 0.
8077985876573534
(Iteration 20401 / 23430) loss: 0.40880155571977045
(Iteration 20501 / 23430) loss: 0.21767229109447067
(Iteration 20601 / 23430) loss: 0.2817191611572535
(Iteration 20701 / 23430) loss: 0.7875675026005833
(Iteration 20801 / 23430) loss: 0.4184475128153746
(Iteration 20901 / 23430) loss: 0.3663529187479895
(Iteration 21001 / 23430) loss: 0.400797070932677
(Iteration 21101 / 23430) loss: 0.4508339492173996
(Iteration 21201 / 23430) loss: 0.42567274890426177
(Iteration 21301 / 23430) loss: 0.38630941199847574
(Iteration 21401 / 23430) loss: 0.2659672429406951
(Iteration 21501 / 23430) loss: 0.541151478545976
(Iteration 21601 / 23430) loss: 0.4574121058355183
(Iteration 21701 / 23430) loss: 0.9276677605567806
(Iteration 21801 / 23430) loss: 0.21475245459610623
(Epoch 14 / 15) Training Accuracy: 0.87512, Validation Accuracy:
0.8096407737181456
(Iteration 21901 / 23430) loss: 0.6005405242059296
(Iteration 22001 / 23430) loss: 0.2806117450558887
(Iteration 22101 / 23430) loss: 0.5402668264664036
(Iteration 22201 / 23430) loss: 0.4990929882296552
(Iteration 22301 / 23430) loss: 0.20476868856302236
(Iteration 22401 / 23430) loss: 0.25093077724826246
(Iteration 22501 / 23430) loss: 0.24686692603078927
(Iteration 22601 / 23430) loss: 0.47247383492061334
(Iteration 22701 / 23430) loss: 0.5327330233497646
(Iteration 22801 / 23430) loss: 0.6243319432683416
(Iteration 22901 / 23430) loss: 0.39787820989046446
(Iteration 23001 / 23430) loss: 0.31383667436179996
(Iteration 23101 / 23430) loss: 0.6515293932569007
(Iteration 23201 / 23430) loss: 0.5058765809341539
(Iteration 23301 / 23430) loss: 0.4360468453636742
(Iteration 23401 / 23430) loss: 0.37640295406501467
```

```
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:34: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:36: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:38: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:41: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:43: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:45: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:48: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:50: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:52: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
```

```
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:55: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:57: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
ning can be suppressed, and the future behavior ensured, by passin
g a unique label to each axes instance.
/media/aditya/DriveTwo/usc/CSCI_566/assignment_01/venv/lib/python
3.6/site-packages/ipykernel_launcher.py:59: MatplotlibDeprecationW
arning: Adding an axes using the same arguments as a previous axes
currently reuses the earlier instance.  In a future version, a new
instance will always be created and returned.  Meanwhile, this war
```



Plot the Activation Functions [3pt]

In each of the activation function, use the given lambda function template to plot their corresponding curves.

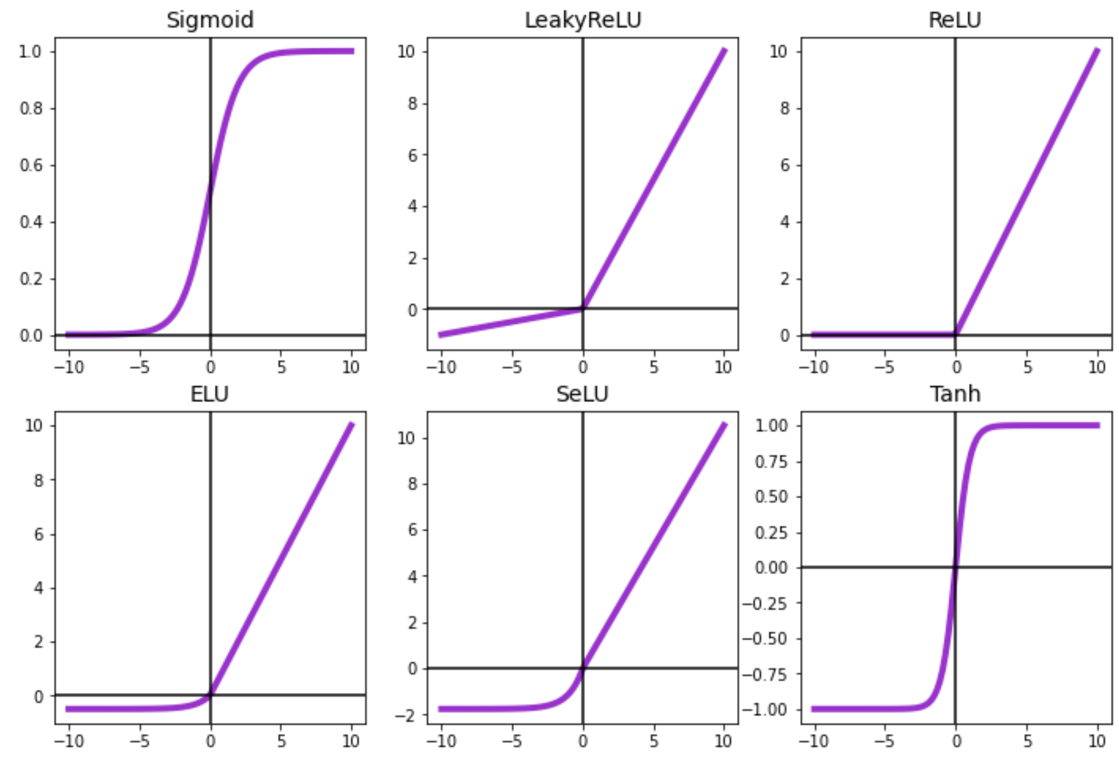
In [ ]:
```
left, right = -10, 10
X  = np.linspace(left, right, 100)
XS = np.linspace(-5, 5, 10)
lw = 4
alpha = 0.1 # alpha for leaky_relu
elu_alpha = 0.5
selu_alpha = 1.6732
selu_scale = 1.0507

###################
# TODO: YOUR CODE #
###################
sigmoid = lambda x: (1 / (1 + np.exp(x * -1)))
leaky_relu = lambda x: x * (x > 0) + (x < 0) * alpha * x
relu = lambda x: x * (x > 0) + (x<0) * 0
elu = lambda x: x * (x > 0) + (x < 0) * elu_alpha * (np.exp(x) -
1)
selu = lambda x: selu_scale * ((x) * (x > 0) + (x < 0) * selu_alph
a*(np.exp(x) - 1))
tanh = lambda x: np.tanh(x)
####################
# END OF YOUR CODE #
####################

activations = {
    "Sigmoid": sigmoid,
    "LeakyReLU": leaky_relu,
    "ReLU": relu,
    "ELU": elu,
    "SeLU": selu,
    "Tanh": tanh
}

# Ground Truth activations
GT_Act = {
    "Sigmoid": [0.00669285092428, 0.0200575365379, 0.058536902874
4, 0.158869104881, 0.364576440742,
                0.635423559258, 0.841130895119, 0.941463097126, 0.
979942463462, 0.993307149076],
    "LeakyReLU": [-0.5, -0.388888888889, -0.277777777778, -0.16666
6666667, -0.0555555555556,
                  0.555555555556, 1.66666666667, 2.77777777778, 3.
88888888889, 5.0],
    "ReLU": [-0.0, -0.0, -0.0, -0.0, -0.0, 0.555555555556, 1.66666
666667, 2.77777777778, 3.88888888889, 5.0],
    "ELU": [-0.4966310265, -0.489765962143, -0.468911737989, -0.40
5562198581, -0.213123289631,
            0.555555555556, 1.66666666667, 2.77777777778, 3.888888
88889, 5.0],
    "SeLU": [-1.74618571868, -1.72204772347, -1.64872296837, -1.42
598202974, -0.749354802287,
             0.583722222222, 1.75116666667, 2.91861111111, 4.08605
555556, 5.2535],
    "Tanh": [-0.999909204263, -0.999162466631, -0.992297935288, -
0.931109608668, -0.504672397722,
             0.504672397722, 0.931109608668, 0.992297935288, 0.999
162466631, 0.999909204263]
}
```

```
fig = plt.figure(figsize=(12,8))
for i, label in enumerate(activations):
    ax = fig.add_subplot(2, 3, i+1)
    ax.plot(X, activations[label](X), color='darkorchid', lw=lw, l
abel=label)
    assert rel_error(activations[label](XS), GT_Act[label]) < 1e-
9, \
            "Your implementation of {} might be wrong".format(labe
l)
    ax.axhline(0, color='black')
    ax.axvline(0, color='black')
    ax.set_title('{}'.format(label), fontsize=14)
plt.show()
```



Submission

Please prepare a PDF document `problem_1_solution.pdf` in the root directory of this repository with all plots and inline answers of your solution. Concretely, the document should contain the following items in strict order:

1. Training loss / accuracy curves for the simple neural network training with >75% validation accuracy
2. Plots for comparing vanilla SGD to SGD + Momentum
3. "Comparing different Optimizers" plots
4. Activation function plot

Note that you still need to submit the jupyter notebook with all generated solutions. We will randomly pick submissions and check that the plots in the PDF and in the notebook are equivalent.