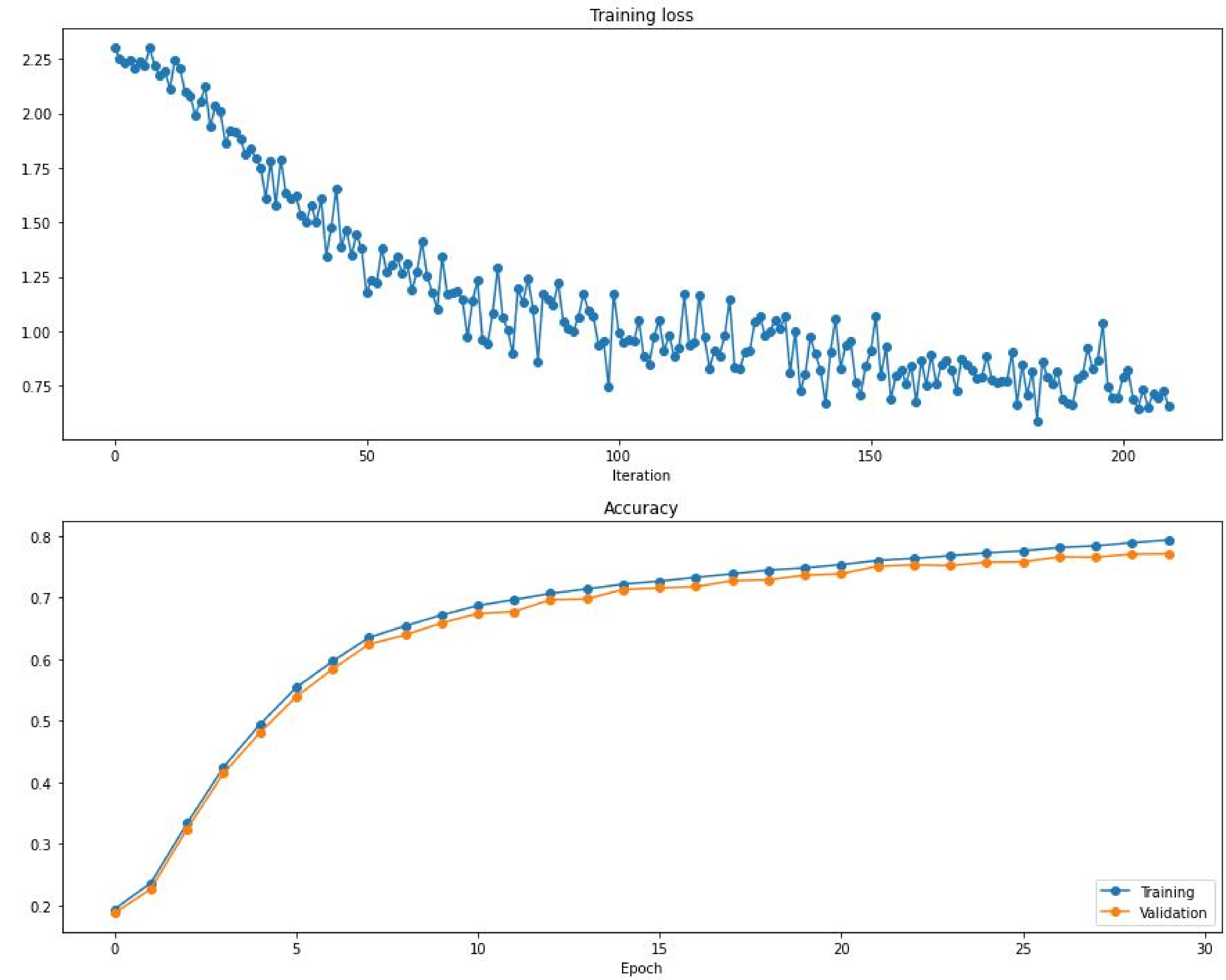
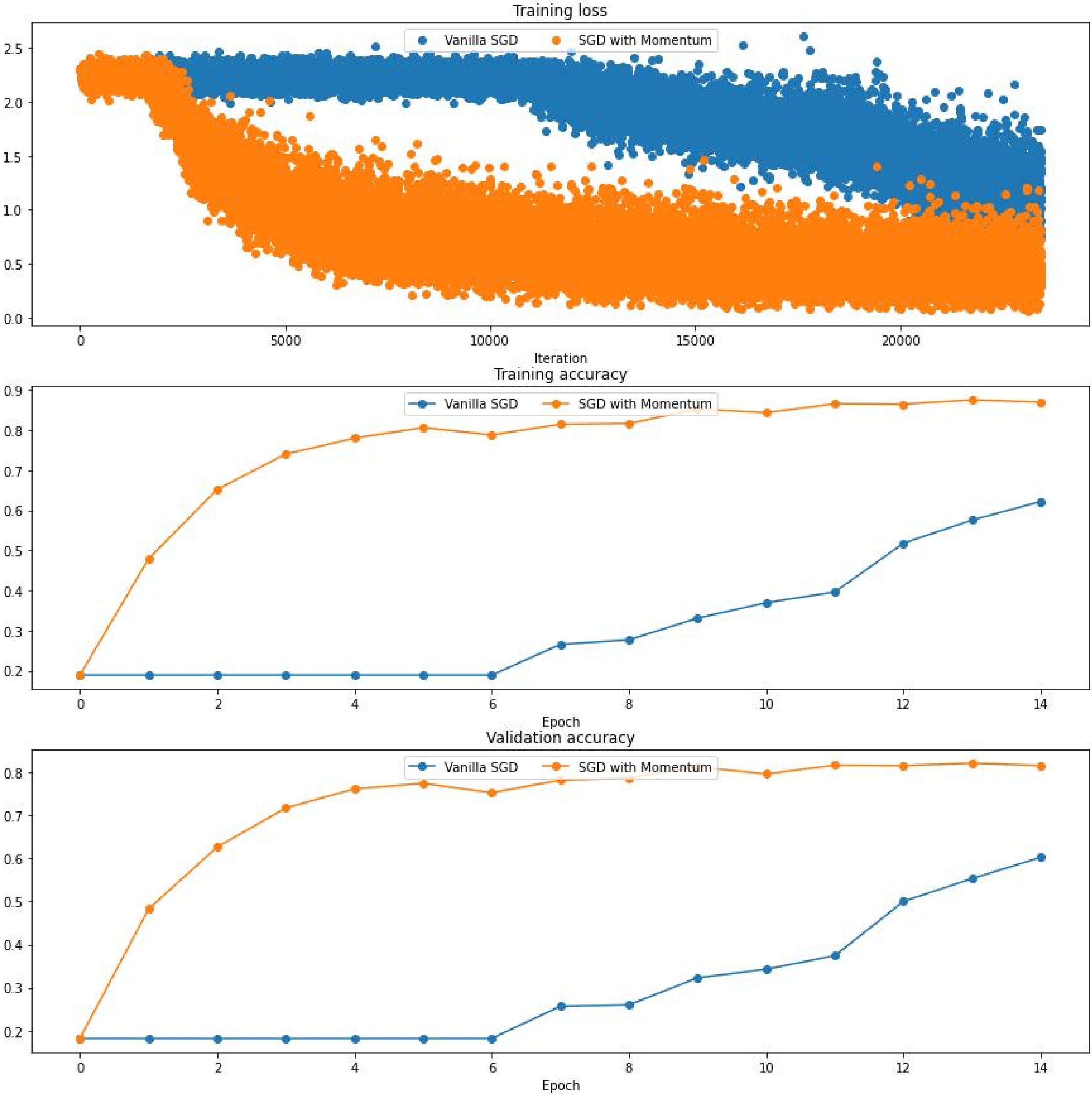
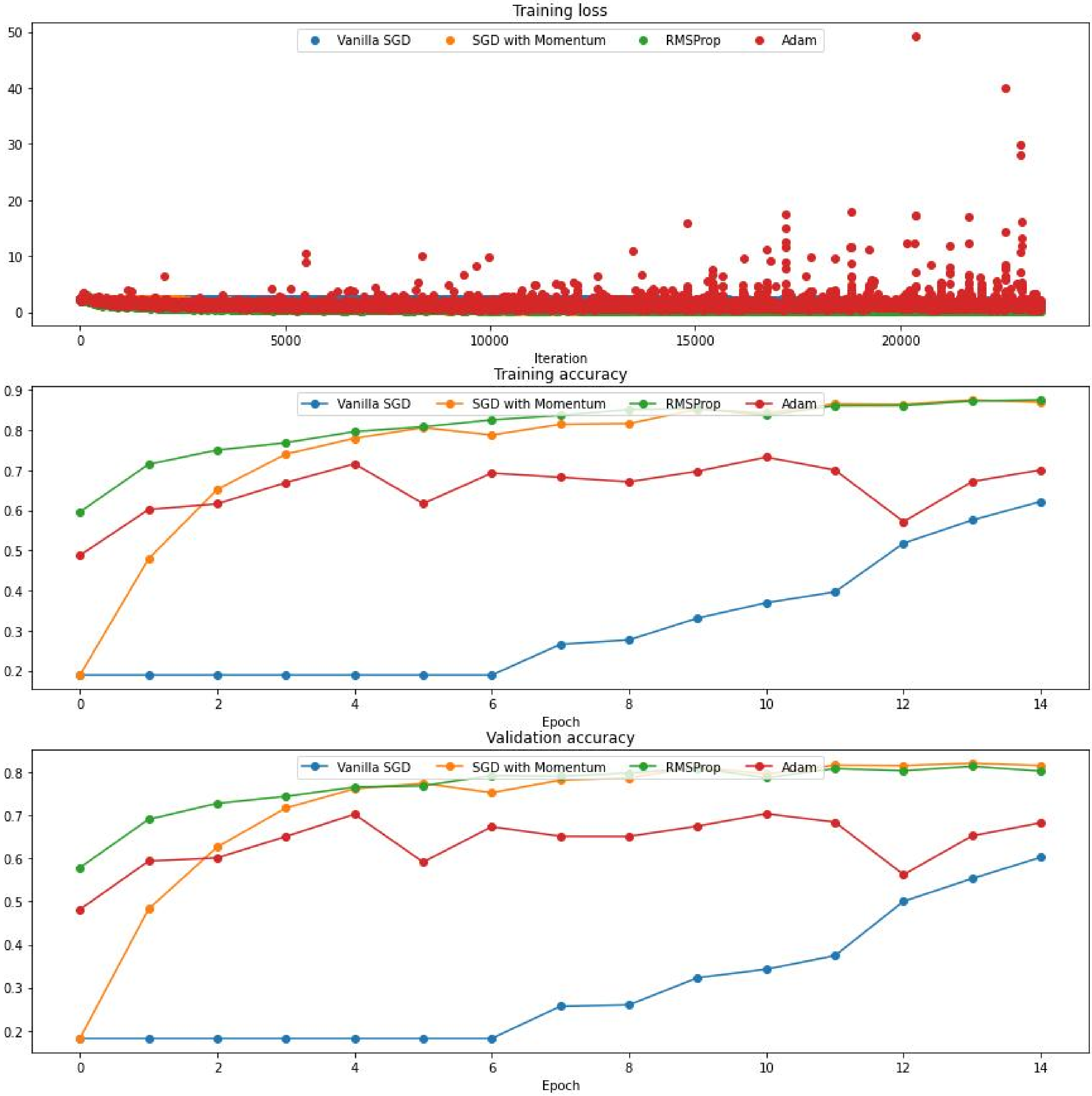
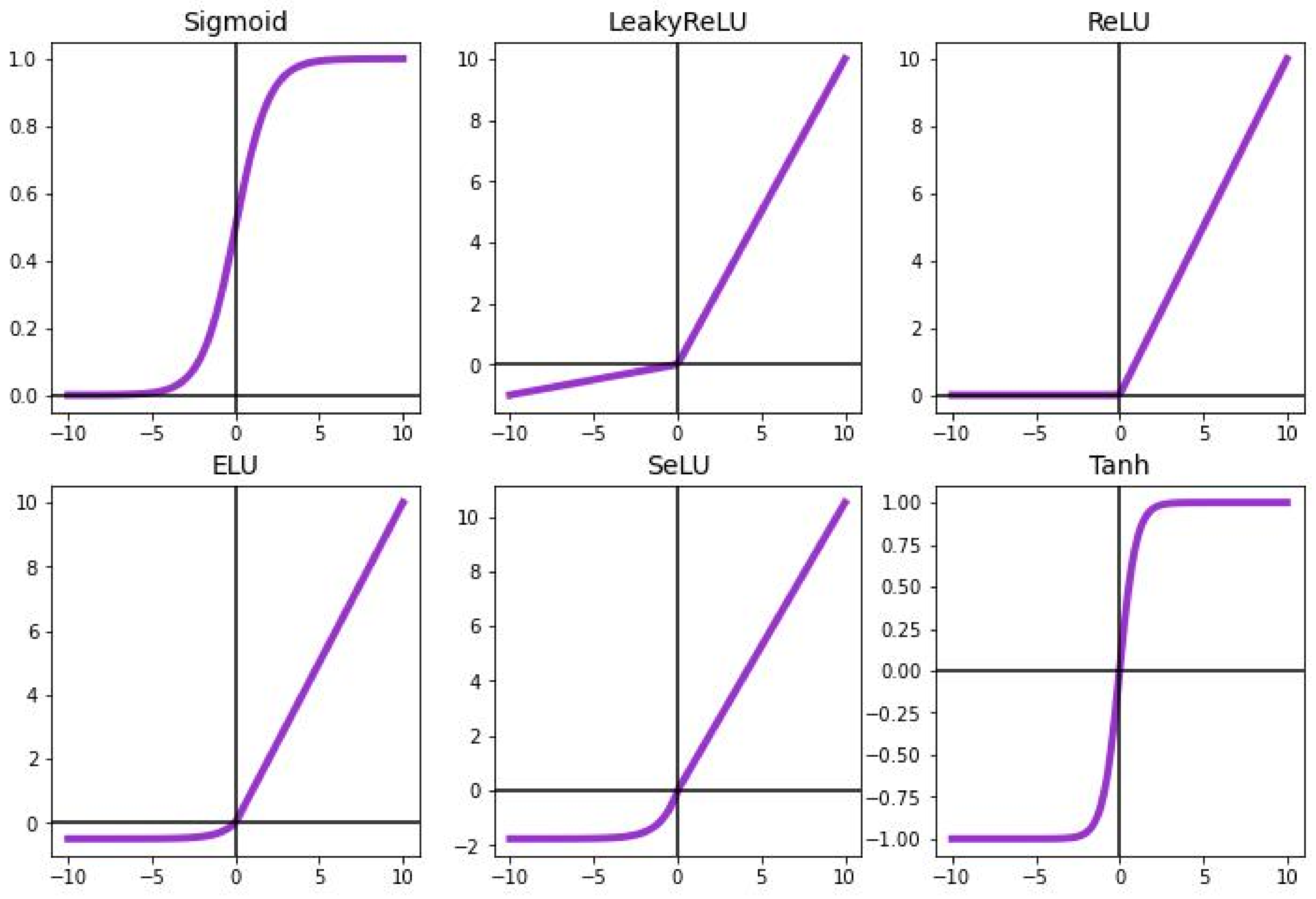


Training loss
2.25
2.00
1.75
1.50
1.25
1.00
0.75
0
50
100
150
200
Iteration
Accuracy
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
5
10
15
20
25
30
Epoch
Training
Validation



Training loss
Vanilla SGD
SGD with Momentum
2.5
2.0
1.5
1.0
0.5
0.0
0
5000
10000
15000
20000
Iteration
Training accuracy
Vanilla SGD
SGD with Momentum
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10
12
14
Epoch
Validation accuracy
Vanilla SGD
SGD with Momentum
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
2
4
6
8
10
12
14
Epoch


Training loss
Vanilla SGD
SGD with Momentum
RMSProp
Adam
Iteration
Training accuracy
Epoch
Validation accuracy
Epoch


Sigmoid
LeakyReLU
ReLU
ELU
SeLU
Tanh
1.0
0.8
0.6
0.4
0.2
0.0
10
8
6
4
2
0
−2
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−10
−5
5

Problem 1: Basics of Neural Networks

Loading the data (SVHN)

Implement Standard Layers

FC Forward [2pt]

FC Backward [2pt]

Leaky ReLU Forward [2pt]

Leaky ReLU Backward [2pt]

Dropout Forward [2pt]

Dropout Backward [2pt]

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

SoftMax Function and Loss Layer [2pt]

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 [68]:
```
%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 = 0.002
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("fullo_1_w").std() - std)
b1 = model.net.get_params("fullo_1_b").std()
w2_std = abs(model.net.get_params("fullo_2_w").std() - std)
b2 = model.net.get_params("fullo_2_b").std()
###################################################
#                END OF YOUR CODE                  #
###################################################

print(w1_std)
print(std)
print(w2_std)
print(std)
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("fullo_1_w", w1)
model.net.assign("fullo_1_b", b1)
model.net.assign("fullo_2_w", w2)
model.net.assign("fullo_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, -1.50461046, -1.08753166, -0.67045287, -0.25337408,  0.
                             [-1.57216705, -1.18571026, -0.79925348, -0.4127967 , -0.02633991, 0.36011687,  0.7
                             [-0.80556605, -0.44973128, -0.09389651,  0.26193827,  0.61777304, 0.97360782,  1.3
                             [-0.03896506,  0.2862477 ,  0.61146047,  0.93667323,  1.261886  , 1.58709876,  1.9
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 might 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_num, layer.grads[name])))
```

```
Testing initialization ...
4.732445743017475e-05
0.002
5.648012981167489e-05
0.002
Passed!
Testing test-time forward pass ...
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_11112/1238117818.py in <module>
     61                              [-0.03896506,  0.2862477 ,  0.61146047,  0.93667323,  1.261886  , 1.587098
76,  1.91231153]])
     62 scores_diff = np.sum(np.abs(scores - correct_scores))
---> 63 assert scores_diff < 1e-6, "Your implementation might be wrong!"
     64 print ("Passed!")
     65

AssertionError: Your implementation might be wrong!
```

In [72]:
```
scores
```

Out[72]:
```
array([[-2.33876804e+00, -1.92168925e+00, -1.50461046e+00,
        -1.08753166e+00, -6.70452870e-01, -2.53374077e-01,
         1.63704716e+01],
       [-1.57216705e+00, -1.18571026e+00, -7.99253481e-01,
        -4.12796698e-01, -2.63399140e-02,  3.60116870e+01,
         7.46573653e+01],
       [-8.05566054e-01, -4.49731280e-01, -9.38965060e-02,
         2.61938268e+01,  6.17773042e+01,  9.73607816e+01,
         1.32944259e+02],
       [-3.89650597e-02,  2.86247705e+01,  6.11460469e+01,
         9.36673234e+01,  1.26188600e+02,  1.58709876e+02,
         1.91231153e+02]])
```

In [73]:
```
correct_scores
```

Out[73]:
```
array([[-2.33876804, -1.92168925, -1.50461046, -1.08753166, -0.67045287,
        -0.25337408,  0.16370472],
       [-1.57216705, -1.18571026, -0.79925348, -0.4127967 , -0.02633991,
         0.36011687,  0.74657365],
       [-0.80556605, -0.44973128, -0.09389651,  0.26193827,  0.61777304,
         0.97360782,  1.32944259],
       [-0.03896506,  0.2862477 ,  0.61146047,  0.93667323,  1.261886  ,
         1.58709876,  1.91231153]])
```

In [74]:
```
scores-correct_scores
```

Out[74]:
```
array([[-3.10240900e-09,  1.37376777e-10,  3.37716011e-09,
        -3.38305806e-09, -1.43274170e-10,  3.09650888e-09,
         1.62067669e+01],
       [ 1.36236222e-09, -4.96723263e-09, -1.29682542e-09,
         2.37357967e-09, -3.95601387e-09,  3.56515701e+01,
         7.39107917e+01],
       [-4.17286750e-09, -7.18371473e-11,  4.02918955e-09,
         2.59318885e+01,  6.11595312e+01,  9.63871738e+01,
         1.31614816e+02],
       [ 2.91905618e-10,  2.83385228e+01,  6.05345865e+01,
         9.27306502e+01,  1.24926714e+02,  1.57122778e+02,
         1.89318841e+02]])
```

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 [75]:
```
%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, seed=seed), y)
        grad_num = eval_numerical_gradient(f, model.net.params[name], 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: 1.5562038182886694e-05
fc1_w relative error: 9.723764020299864e-05
fc2_b relative error: 2.7841895262663007e-06
fc2_w relative error: 0.0019218743762228804
fc3_b relative error: 6.9596628378975445e-09
fc3_w relative error: 0.0003714499355589531

Dropout p = 0.25
Error of gradients should be around or less than 1e-3
fc1_b relative error: 2.1950150920240895e-06
fc1_w relative error: 2.5552274040519906e-05
fc2_b relative error: 1.878084533525588e-06
fc2_w relative error: 0.0009455388205791924
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: 2.2332156468536366e-07
fc1_w relative error: 7.381101514624856e-05
fc2_b relative error: 2.609643832002952e-07
fc2_w relative error: 0.0039842857920920054
fc3_b relative error: 1.2050218326231938e-09
fc3_w relative error: 3.584680630505416e-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 [76]:
```
# 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 [77]:
```
print("Data shape:", data["data_train"].shape)
print("Flattened data input size:", np.prod(data["data_train"].shape[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 [79]:
```
%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 network         #
#############################################################################

batch_size = 100
epochs = 30
lr_decay = .99
lr_decay_every = 50

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

```
(Iteration 1 / 21000) loss: 2.287865881740393
(Epoch 1 / 30) Training Accuracy: 0.19422857142857142, Validation Accuracy: 0.1879029782007983
(Epoch 2 / 30) Training Accuracy: 0.23675714285714286, Validation Accuracy: 0.22689591648756524
(Epoch 3 / 30) Training Accuracy: 0.33375714285714286, Validation Accuracy: 0.32453177770954866
(Epoch 4 / 30) Training Accuracy: 0.42472857142857146, Validation Accuracy: 0.4147988946883635
(Epoch 5 / 30) Training Accuracy: 0.4939571428571429, Validation Accuracy: 0.4805035308566165
(Epoch 6 / 30) Training Accuracy: 0.5540142857142857, Validation Accuracy: 0.5388394227817009
(Epoch 7 / 30) Training Accuracy: 0.5965285714285714, Validation Accuracy: 0.5839729812717084
(Epoch 8 / 30) Training Accuracy: 0.6350714285714286, Validation Accuracy: 0.6241940435984035
(Epoch 9 / 30) Training Accuracy: 0.6541285714285714, Validation Accuracy: 0.6392385630948726
(Epoch 10 / 30) Training Accuracy: 0.6716428571428571, Validation Accuracy: 0.6588885477433221
(Epoch 11 / 30) Training Accuracy: 0.6870285714285714, Validation Accuracy: 0.6742400982499233
(Epoch 12 / 30) Training Accuracy: 0.6967714285714286, Validation Accuracy: 0.6773104083512435
(Epoch 13 / 30) Training Accuracy: 0.7066285714285714, Validation Accuracy: 0.696653361989561
(Epoch 14 / 30) Training Accuracy: 0.7141571428571428, Validation Accuracy: 0.697881486030089
(Iteration 10001 / 21000) loss: 0.9897715700557103
(Epoch 15 / 30) Training Accuracy: 0.7218428571428571, Validation Accuracy: 0.7132330365366902
(Epoch 16 / 30) Training Accuracy: 0.7268, Validation Accuracy: 0.7156892846177464
(Epoch 17 / 30) Training Accuracy: 0.7328285714285714, Validation Accuracy: 0.7178385016886706
(Epoch 18 / 30) Training Accuracy: 0.7387142857142858, Validation Accuracy: 0.7273564630027632
(Epoch 19 / 30) Training Accuracy: 0.7446285714285714, Validation Accuracy: 0.7288916180534234
(Epoch 20 / 30) Training Accuracy: 0.7483142857142857, Validation Accuracy: 0.7365673933067239
(Epoch 21 / 30) Training Accuracy: 0.7535142857142857, Validation Accuracy: 0.7387166103776481
(Epoch 22 / 30) Training Accuracy: 0.7603285714285715, Validation Accuracy: 0.750997850782929
(Epoch 23 / 30) Training Accuracy: 0.7636, Validation Accuracy: 0.7531470678538532
(Epoch 24 / 30) Training Accuracy: 0.7681142857142857, Validation Accuracy: 0.7519189438133251
(Epoch 25 / 30) Training Accuracy: 0.7725857142857143, Validation Accuracy: 0.7574455019957016
(Epoch 26 / 30) Training Accuracy: 0.7759142857142857, Validation Accuracy: 0.7580595640159656
(Epoch 27 / 30) Training Accuracy: 0.7812857142857143, Validation Accuracy: 0.7660423702793983
(Epoch 28 / 30) Training Accuracy: 0.7840571428571429, Validation Accuracy: 0.7657353392692662
(Iteration 20001 / 21000) loss: 0.7879275550312244
(Epoch 29 / 30) Training Accuracy: 0.7892142857142858, Validation Accuracy: 0.7706478354313786
(Epoch 30 / 30) Training Accuracy: 0.7936142857142857, Validation Accuracy: 0.7712618974516426
```

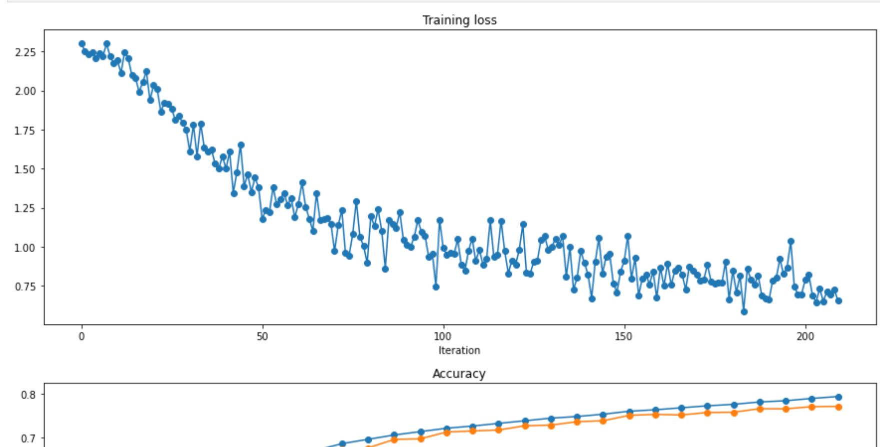
In [80]:
```
# Take a look at what names of params were stored
print (opt_params.keys())
```

```
dict_keys(['fullo_1_w', 'fullo_1_b', 'fullo_2_w', 'fullo_2_b'])
```

In [81]:
```
# 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_test"])
print ("Testing Accuracy: {}%".format(test_acc*100))
```

```
Loading Params: fullo_1_w Shape: (3072, 512)
Loading Params: fullo_1_b Shape: (512,)
Loading Params: fullo_2_w Shape: (512, 10)
Loading Params: fullo_2_b Shape: (10,)
Validation Accuracy: 77.12618974516427%
Testing Accuracy: 75.7221880762139%
```

In [82]:
```
# 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 [83]:
```
%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,  0.40775789],
  [ 0.47454737,  0.54133684,  0.60812632,  0.67491579,  0.74170526],
  [ 0.80849474,  0.87528421,  0.94207368,  1.00886316,  1.07565263],
  [ 1.14244211,  1.20923158,  1.27602105,  1.34281053,  1.4096    ]])
expected_velocity = np.asarray([
  [ 0.5406,      0.55475789,  0.56891579, 0.58307368,  0.59723158],
  [ 0.61138947,  0.62554737,  0.63970526,  0.65386316,  0.66802105],
  [ 0.68217895,  0.69633684,  0.71049474,  0.72465263,  0.73881053],
  [ 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
```

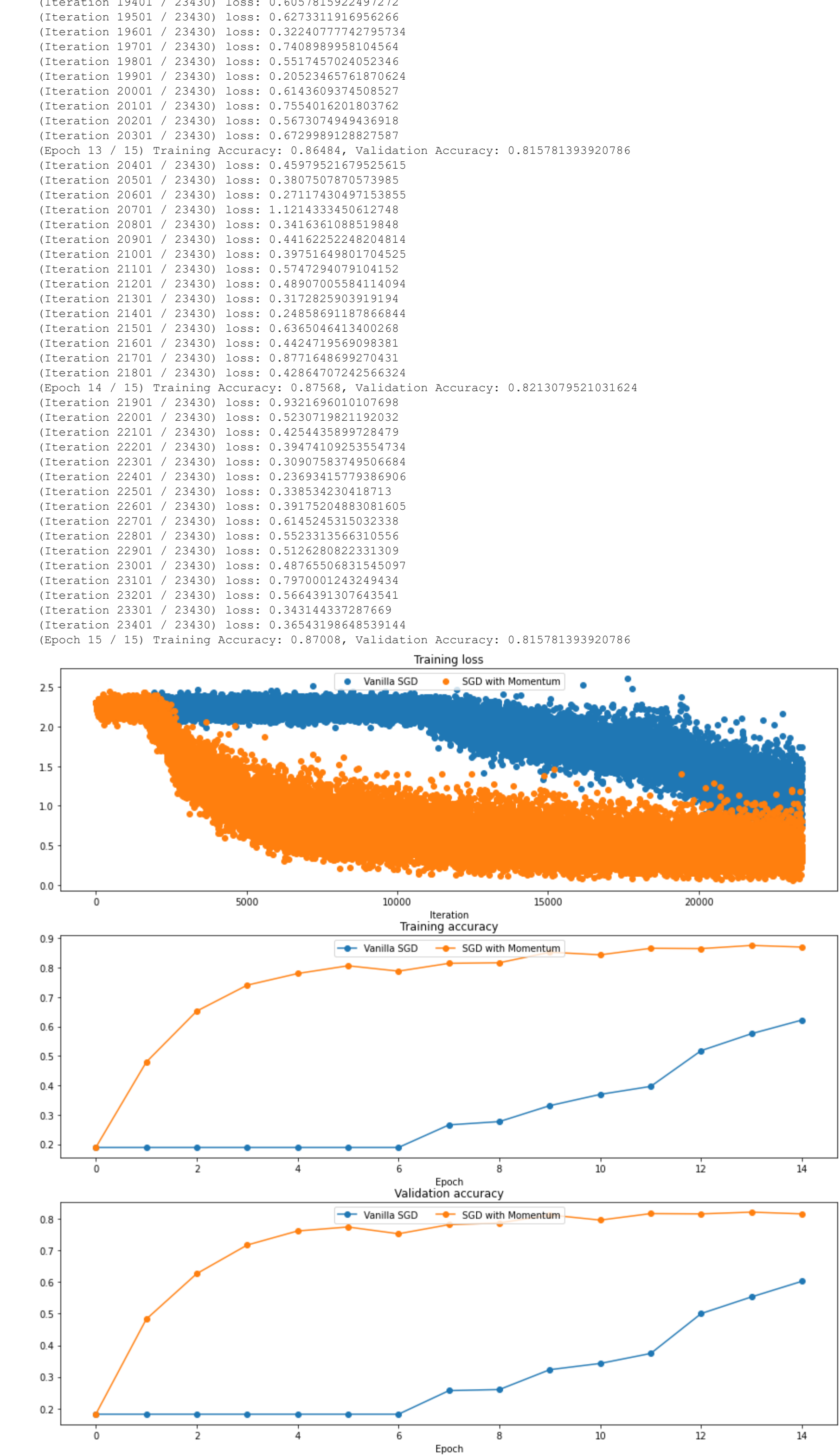
Comparing SGD and SGD with Momentum [2pt]

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 [85]:

```python
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_train"][: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, optimizer_sgd, batch_size=32,
                        max_epochs=15, show_every=100, verbose=True)

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=True)

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: 2.303460123828989
(Iteration 201 / 23430) loss: 2.268727870891892
(Iteration 301 / 23430) loss: 2.243775120272611
(Iteration 401 / 23430) loss: 2.238319010526329
(Iteration 501 / 23430) loss: 2.272874253381526
(Iteration 601 / 23430) loss: 2.2451687770884394
(Iteration 701 / 23430) loss: 2.2622141329338215
(Iteration 801 / 23430) loss: 2.2573692693348812
(Iteration 901 / 23430) loss: 2.3380941805908977
(Iteration 1001 / 23430) loss: 2.20257836640194
(Iteration 1101 / 23430) loss: 2.1939285735429026
(Iteration 1201 / 23430) loss: 2.1429886797642235
(Iteration 1301 / 23430) loss: 2.2688718233746474
(Iteration 1401 / 23430) loss: 2.1504753777293333
(Iteration 1501 / 23430) loss: 2.2735229700568405
(Epoch 1 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.18206938900828984
(Iteration 1601 / 23430) loss: 2.2460415185131373
(Iteration 1701 / 23430) loss: 2.333875309034275
(Iteration 1801 / 23430) loss: 2.117519352081804
(Iteration 1901 / 23430) loss: 2.205535458318208
(Iteration 2001 / 23430) loss: 2.1437318018984315
(Iteration 2101 / 23430) loss: 2.2396229891697548
(Iteration 2201 / 23430) loss: 2.1509783980976405
(Iteration 2301 / 23430) loss: 2.1239988360035853
(Iteration 2401 / 23430) loss: 2.2576888344005743
(Iteration 2501 / 23430) loss: 2.29772304584748
(Iteration 2601 / 23430) loss: 2.216278969862204
(Iteration 2701 / 23430) loss: 2.1420435054696814
(Iteration 2801 / 23430) loss: 2.2409122687683816
(Iteration 2901 / 23430) loss: 2.267620327979775
(Iteration 3001 / 23430) loss: 2.3097516637777322
(Iteration 3101 / 23430) loss: 2.2175169448719236
(Epoch 2 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.18206938900828984
(Iteration 3201 / 23430) loss: 2.194154112103643
(Iteration 3301 / 23430) loss: 2.3395208325098995
(Iteration 3401 / 23430) loss: 2.213283801585745
(Iteration 3501 / 23430) loss: 2.2152499932798637
(Iteration 3601 / 23430) loss: 2.167542970843872
(Iteration 3701 / 23430) loss: 2.1897861389815265
(Iteration 3801 / 23430) loss: 2.131262375298502
(Iteration 3901 / 23430) loss: 2.147113779687797
(Iteration 4001 / 23430) loss: 2.285455946890294
(Iteration 4101 / 23430) loss: 2.1982515121053874
(Iteration 4201 / 23430) loss: 2.2074600074836823
(Iteration 4301 / 23430) loss: 2.22467412692337
(Iteration 4401 / 23430) loss: 2.053039396099147
(Iteration 4501 / 23430) loss: 2.3524286994091046
(Iteration 4601 / 23430) loss: 2.229357460968944
(Epoch 3 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.18206938900828984
(Iteration 4701 / 23430) loss: 2.39724854712492
(Iteration 6801 / 23430) loss: 2.207306343380298
(Iteration 6901 / 23430) loss: 2.3338415868150264
(Iteration 7001 / 23430) loss: 2.2428941144295695
(Iteration 7101 / 23430) loss: 2.19181403337204
(Iteration 7201 / 23430) loss: 2.2615798504724722
(Iteration 7301 / 23430) loss: 2.396177740667492
(Iteration 7401 / 23430) loss: 2.263987570803724
(Iteration 7501 / 23430) loss: 2.2073632923493647
(Iteration 7601 / 23430) loss: 2.163729093617351
(Iteration 7701 / 23430) loss: 2.273839708919157
(Iteration 7801 / 23430) loss: 2.1826534818643277
(Epoch 5 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.18206938900828984
(Iteration 7901 / 23430) loss: 2.2797024448484935
(Iteration 8001 / 23430) loss: 2.361265166117029
(Iteration 8101 / 23430) loss: 2.1805389463231672
(Iteration 8201 / 23430) loss: 2.2969331498596612
(Iteration 8301 / 23430) loss: 2.2361079019196635
(Iteration 8401 / 23430) loss: 2.2584919828687298
(Iteration 8501 / 23430) loss: 2.2413859227701627
(Iteration 8601 / 23430) loss: 2.2115104713312093
(Iteration 8701 / 23430) loss: 2.333687416325356
(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: 0.18206938900828984
(Iteration 9401 / 23430) loss: 2.064269178833502
(Iteration 9501 / 23430) loss: 2.2745090868034987
(Iteration 9601 / 23430) loss: 2.1333546288924934
(Iteration 9701 / 23430) loss: 2.1175980107656276
(Iteration 9801 / 23430) loss: 2.154674459751231
(Iteration 9901 / 23430) loss: 2.2688888649793952
(Iteration 10001 / 23430) loss: 2.1830584012101015
(Iteration 10101 / 23430) loss: 2.251299393685507
(Iteration 10201 / 23430) loss: 2.2631166633317994
(Iteration 10301 / 23430) loss: 2.167193612321145
(Iteration 10401 / 23430) loss: 2.2763755254682536
(Iteration 10501 / 23430) loss: 2.16603649640145
(Iteration 10601 / 23430) loss: 2.1639501612675778
(Iteration 10701 / 23430) loss: 2.2584520531469425
(Iteration 10801 / 23430) loss: 2.2152285044154416
(Iteration 10901 / 23430) loss: 2.1293325177604157
(Epoch 7 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.18206938900828984
(Iteration 11001 / 23430) loss: 2.1812958200919343
(Iteration 11101 / 23430) loss: 2.2145639536648787
(Iteration 11201 / 23430) loss: 2.139754233046444
(Iteration 11301 / 23430) loss: 2.1922999939233985
(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: 2.2007456236485377
(Iteration 12001 / 23430) loss: 1.9773411373340384
(Iteration 12101 / 23430) loss: 1.967977380561737
(Iteration 12201 / 23430) loss: 2.207848277127094
(Iteration 12301 / 23430) loss: 2.1919810048360073
(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: 1.8119360013391257
(Iteration 12801 / 23430) loss: 1.9390599645946671
(Iteration 12901 / 23430) loss: 2.2381560403774725
(Iteration 13001 / 23430) loss: 2.0414202115931106
(Iteration 13101 / 23430) loss: 2.0795490790657816
(Iteration 13201 / 23430) loss: 2.042282473390002
(Iteration 13301 / 23430) loss: 1.811861953700057
(Iteration 13401 / 23430) loss: 1.9351495765265252
(Iteration 13501 / 23430) loss: 1.9841672699785424
(Iteration 13601 / 23430) loss: 1.9181778266096532
(Iteration 13701 / 23430) loss: 1.984624609647197
(Iteration 13801 / 23430) loss: 2.2064507374734634
(Iteration 13901 / 23430) loss: 1.9779072647971965
(Iteration 14001 / 23430) loss: 1.8408176865541896
(Epoch 9 / 15) Training Accuracy: 0.2768, Validation Accuracy: 0.2600552655818238
(Iteration 14101 / 23430) loss: 1.7569095207460452
(Iteration 14201 / 23430) loss: 1.90653417352891
(Iteration 14301 / 23430) loss: 1.8405852112038734
(Iteration 14401 / 23430) loss: 1.907788026702502
(Iteration 14501 / 23430) loss: 1.8556381527055883
(Iteration 14601 / 23430) loss: 1.9433920970151335
(Iteration 14701 / 23430) loss: 1.9673150281977414
(Iteration 14801 / 23430) loss: 1.9219862051431762
(Iteration 14901 / 23430) loss: 1.683846822287161
(Iteration 15001 / 23430) loss: 1.8905392116039246
(Iteration 15101 / 23430) loss: 1.8586364701283031
(Iteration 15201 / 23430) loss: 1.8125553565722263
(Iteration 15301 / 23430) loss: 1.7567598757902667
(Iteration 15401 / 23430) loss: 2.167239201771908
(Iteration 15501 / 23430) loss: 1.8741552478505759
(Iteration 15601 / 23430) loss: 1.8212786941394377
(Epoch 10 / 15) Training Accuracy: 0.33108, Validation Accuracy: 0.32299662265888857
(Iteration 15701 / 23430) loss: 1.753702623122177
(Iteration 15801 / 23430) loss: 1.9728123854944184
(Iteration 15901 / 23430) loss: 1.9557297603941106
(Iteration 16001 / 23430) loss: 1.6281151564229641
(Iteration 16101 / 23430) loss: 1.600659982625428
(Iteration 16201 / 23430) loss: 1.8956634528890361
(Iteration 16301 / 23430) loss: 1.9344399880350909
(Iteration 16401 / 23430) loss: 1.7226065581114627
(Iteration 16501 / 23430) loss: 1.6073387536486237
(Iteration 16601 / 23430) loss: 1.6019310832310998
(Iteration 16701 / 23430) loss: 1.7315967814801785
(Iteration 16801 / 23430) loss: 1.5204441476694615
(Iteration 16901 / 23430) loss: 1.5464829112340348
(Iteration 17001 / 23430) loss: 1.608201684959079
(Iteration 17101 / 23430) loss: 1.8403858445214452
(Epoch 11 / 15) Training Accuracy: 0.36932, Validation Accuracy: 0.34264660730733804
(Iteration 17201 / 23430) loss: 1.7732237918799951
(Iteration 17301 / 23430) loss: 1.517043044302929
(Iteration 17401 / 23430) loss: 1.5159477890988478
(Iteration 17501 / 23430) loss: 1.7578906791866782
(Iteration 17601 / 23430) loss: 1.5339618433488769
(Iteration 17701 / 23430) loss: 1.8577912815964448
(Iteration 17801 / 23430) loss: 1.8913185969349833
(Iteration 17901 / 23430) loss: 1.9127473116810205
(Iteration 18001 / 23430) loss: 1.5610399005381488
(Iteration 18101 / 23430) loss: 1.772883364645157
(Iteration 18201 / 23430) loss: 1.8224428314394987
(Iteration 18301 / 23430) loss: 1.6567309881707013
(Iteration 18401 / 23430) loss: 1.787513889026172
(Iteration 18501 / 23430) loss: 1.4083515989259765
(Iteration 18601 / 23430) loss: 1.4333798509491087
(Iteration 18701 / 23430) loss: 1.3818909214593604
(Epoch 12 / 15) Training Accuracy: 0.39642, Validation Accuracy: 0.37427080135093643
(Iteration 18801 / 23430) loss: 1.7807727538317266
(Iteration 18901 / 23430) loss: 1.681214358563727
(Iteration 19001 / 23430) loss: 1.5376104706345068
(Iteration 19101 / 23430) loss: 1.389587876925456
(Iteration 19201 / 23430) loss: 1.3947151970430236
(Iteration 19301 / 23430) loss: 1.5575912996786547
(Iteration 19401 / 23430) loss: 1.4167487841129396
(Iteration 19501 / 23430) loss: 1.504210698888814
(Iteration 19601 / 23430) loss: 1.46798963125244
(Iteration 19701 / 23430) loss: 1.5636968793689223
(Iteration 19801 / 23430) loss: 1.8028723857960987
(Iteration 19901 / 23430) loss: 1.7070041588825462
(Iteration 20001 / 23430) loss: 1.2881934116726916
(Iteration 20101 / 23430) loss: 1.1504785301809508
(Iteration 20201 / 23430) loss: 1.0296477227183738
(Iteration 20301 / 23430) loss: 1.757283506174366
(Epoch 13 / 15) Training Accuracy: 0.51806, Validation Accuracy: 0.500460546515198
(Iteration 20401 / 23430) loss: 1.4946293171987715
(Iteration 20501 / 23430) loss: 0.9703649069519366
(Iteration 20601 / 23430) loss: 1.2601450201804547
(Iteration 20701 / 23430) loss: 1.3951201635990453
(Iteration 20801 / 23430) loss: 1.445178719004254
(Iteration 20901 / 23430) loss: 1.2465183011157983
(Iteration 21001 / 23430) loss: 1.4093181209027486
(Iteration 21101 / 23430) loss: 1.729808889981196
(Iteration 21201 / 23430) loss: 1.1993450554261453
(Iteration 21301 / 23430) loss: 1.1573744131325379
(Iteration 21401 / 23430) loss: 1.0719985720710197
(Iteration 21501 / 23430) loss: 1.366212657441062
(Iteration 21601 / 23430) loss: 1.3492223846833862
(Iteration 21701 / 23430) loss: 1.3116564960545318
(Iteration 21801 / 23430) loss: 1.5450930200194046
(Epoch 14 / 15) Training Accuracy: 0.57572, Validation Accuracy: 0.553269880257906
(Iteration 21901 / 23430) loss: 1.1635580379497 51
(Iteration 22001 / 23430) loss: 1.3971617976313162
(Iteration 22101 / 23430) loss: 1.2411298887757152
(Iteration 22201 / 23430) loss: 1.426375528950741
(Iteration 22301 / 23430) loss: 1.0883458040511147
(Iteration 22401 / 23430) loss: 1.1865591664654036
(Iteration 22501 / 23430) loss: 1.2482097753740016
(Iteration 22601 / 23430) loss: 0.9736811717017174
(Iteration 22701 / 23430) loss: 1.1026065545747206
(Iteration 22801 / 23430) loss: 0.9579572882967303
(Iteration 22901 / 23430) loss: 1.1339795484058746
(Iteration 23001 / 23430) loss: 1.0678596442400266
(Iteration 23101 / 23430) loss: 1.2637995502208783
(Iteration 23201 / 23430) loss: 1.4644910954727393
(Iteration 23301 / 23430) loss: 1.4613114716053492
(Iteration 23401 / 23430) loss: 0.877247147051232
(Epoch 15 / 15) Training Accuracy: 0.62214, Validation Accuracy: 0.6027018728891618

Training with SGD plus Momentum...
(Iteration 1 / 23430) loss: 2.302135570793081
(Iteration 101 / 23430) loss: 2.3337282204550402
(Iteration 201 / 23430) loss: 2.207808147308482
(Iteration 301 / 23430) loss: 2.1644564839379763
(Iteration 401 / 23430) loss: 2.18949320359867
(Iteration 501 / 23430) loss: 2.2570555159616275
(Iteration 601 / 23430) loss: 2.223516855180442
(Iteration 701 / 23430) loss: 2.2621236041824226
(Iteration 801 / 23430) loss: 2.2527563528556134
(Iteration 901 / 23430) loss: 2.360567449447703
(Iteration 1001 / 23430) loss: 2.201613955822357
(Iteration 1101 / 23430) loss: 2.176806406888798
(Iteration 1201 / 23430) loss: 2.112439510141613
(Iteration 1301 / 23430) loss: 2.274109779523543
(Iteration 1401 / 23430) loss: 2.133580653190603
(Iteration 1501 / 23430) loss: 2.2789630060127353
(Epoch 1 / 15) Training Accuracy: 0.1888, Validation Accuracy: 0.18206938900828984
(Iteration 1601 / 23430) loss: 2.3352652809262704
(Iteration 1701 / 23430) loss: 2.0142041314240426
(Iteration 1801 / 23430) loss: 1.8718770885703664
(Iteration 1901 / 23430) loss: 2.093718690754944
(Iteration 2001 / 23430) loss: 1.9480332005886383
(Iteration 2101 / 23430) loss: 2.0671319149628995
(Iteration 2201 / 23430) loss: 1.8730728006241146
(Iteration 2301 / 23430) loss: 1.9891641056727585
(Iteration 2401 / 23430) loss: 2.2327726595170208
(Iteration 2501 / 23430) loss: 1.7626232099934296
```



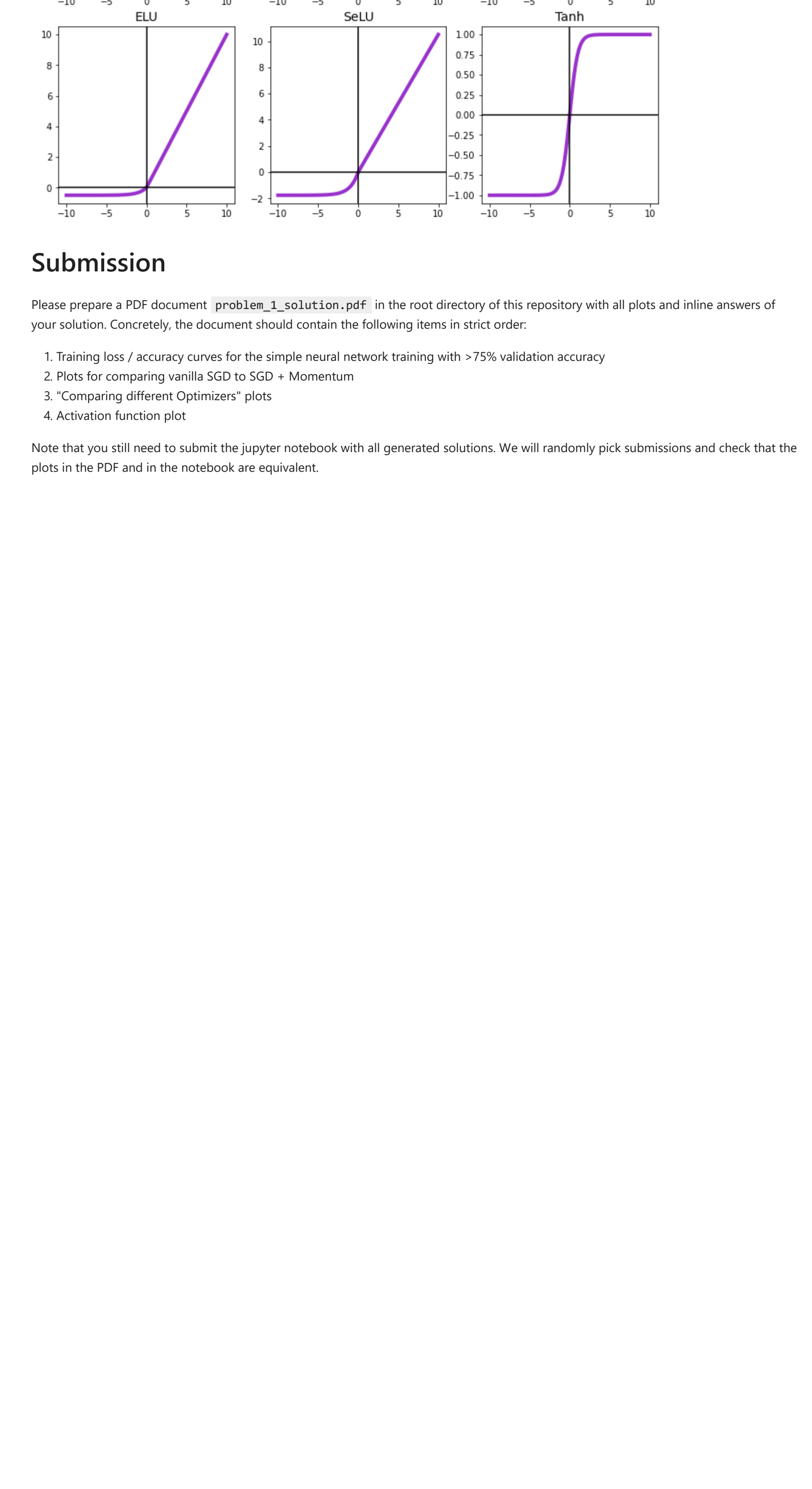
RMSProp [2pt]

Adam [2pt]

Comparing the optimizers [4pt]

```
Training with RMSProp...
(Iteration 1 / 23430) loss: 2.302024602405127
(Iteration 101 / 23430) loss: 2.294067022297631
(Iteration 201 / 23430) loss: 1.8933428725035863
(Iteration 301 / 23430) loss: 1.7594702243817137
(Iteration 401 / 23430) loss: 1.9503735310840349
(Iteration 501 / 23430) loss: 2.273858354003456
(Iteration 601 / 23430) loss: 1.7274812520456964
(Iteration 701 / 23430) loss: 1.6806808255188628
(Iteration 801 / 23430) loss: 1.6672283455389891
(Iteration 901 / 23430) loss: 1.788112483738935
(Iteration 1001 / 23430) loss: 1.3646501000786184
(Iteration 1101 / 23430) loss: 1.427352838482828
(Iteration 1201 / 23430) loss: 1.6809708883056238
(Iteration 1301 / 23430) loss: 1.199111477115944
(Iteration 1401 / 23430) loss: 1.183286758220583
(Iteration 1501 / 23430) loss: 1.3808528439838585
(Epoch 1 / 15) Training Accuracy: 0.59628, Validation Accuracy: 0.578753454098864
(Iteration 1601 / 23430) loss: 1.0052556697343458
(Iteration 1701 / 23430) loss: 1.237933392339031
(Iteration 1801 / 23430) loss: 0.8143512049584078
(Iteration 1901 / 23430) loss: 0.9190943445876718
(Iteration 2001 / 23430) loss: 1.0630680460369195
(Iteration 2101 / 23430) loss: 1.0977704520428926
(Iteration 2201 / 23430) loss: 1.0259748091119227
(Iteration 2301 / 23430) loss: 1.127753702988249
(Iteration 2401 / 23430) loss: 1.0516233791546132
(Iteration 2501 / 23430) loss: 1.0517859732243555
(Iteration 2601 / 23430) loss: 1.1145073918731951
(Iteration 2701 / 23430) loss: 1.1267738548271684
(Iteration 2801 / 23430) loss: 0.8018503116461654
(Iteration 2901 / 23430) loss: 0.6462095232236695
(Iteration 3001 / 23430) loss: 1.2576112977471592
(Iteration 3101 / 23430) loss: 1.1083835909544089
(Epoch 2 / 15) Training Accuracy: 0.7151, Validation Accuracy: 0.6908197727970525
(Iteration 3201 / 23430) loss: 0.663591221467219
(Iteration 3301 / 23430) loss: 0.5854400561402141
(Iteration 3401 / 23430) loss: 1.0393512912436043
(Iteration 3501 / 23430) loss: 0.7022455027088514
(Iteration 3601 / 23430) loss: 0.5603737795426991
(Iteration 3701 / 23430) loss: 0.6897449090796504
(Iteration 3801 / 23430) loss: 0.7940649847342406
(Iteration 3901 / 23430) loss: 1.0799326405267922
(Iteration 4001 / 23430) loss: 1.2948232702413267
(Iteration 4101 / 23430) loss: 0.6973035692705543
(Iteration 4201 / 23430) loss: 0.8430210923497083
(Iteration 4301 / 23430) loss: 0.8869295503461814
(Iteration 4401 / 23430) loss: 0.8813039863106439
(Iteration 4501 / 23430) loss: 1.063567866958722
(Iteration 4601 / 23430) loss: 0.9770414898258803
(Epoch 3 / 15) Training Accuracy: 0.75066, Validation Accuracy: 0.7279705250230273
(Iteration 4701 / 23430) loss: 0.4926941797288583
(Iteration 4801 / 23430) loss: 0.8997039453662637
(Iteration 4901 / 23430) loss: 0.9474659732301247
(Iteration 5001 / 23430) loss: 0.4428209301166264
(Iteration 5101 / 23430) loss: 0.7804558159717115
(Iteration 5201 / 23430) loss: 0.8860585365584739
(Iteration 5301 / 23430) loss: 0.8876279225142569
(Iteration 5401 / 23430) loss: 0.7461874926876383
(Iteration 5501 / 23430) loss: 0.4719420590181753
(Iteration 5601 / 23430) loss: 0.38684006402504684
(Iteration 5701 / 23430) loss: 0.6995661107877953
(Iteration 5801 / 23430) loss: 1.015655424461971
(Iteration 5901 / 23430) loss: 0.534624126048133
(Iteration 6001 / 23430) loss: 0.7327152732077709
(Iteration 6101 / 23430) loss: 0.4348859916484747
(Iteration 6201 / 23430) loss: 0.771581426057842
(Epoch 4 / 15) Training Accuracy: 0.76888, Validation Accuracy: 0.7442431685600246
(Iteration 6301 / 23430) loss: 0.9418467990367347
(Iteration 6401 / 23430) loss: 0.5388115494330454
(Iteration 6501 / 23430) loss: 0.5918788604263586
(Iteration 6601 / 23430) loss: 0.7036796453477442
(Iteration 6701 / 23430) loss: 0.6290645215406778
(Iteration 6801 / 23430) loss: 0.5395836922625323
(Iteration 6901 / 23430) loss: 0.6660227413159094
(Iteration 7001 / 23430) loss: 0.9170356814312054
(Iteration 7101 / 23430) loss: 0.6353602314787168
(Iteration 7201 / 23430) loss: 0.7312045059158002
(Iteration 7301 / 23430) loss: 0.754727444064158
(Iteration 7401 / 23430) loss: 0.8848942669369938
(Iteration 7501 / 23430) loss: 0.7746887819006347
(Iteration 7601 / 23430) loss: 0.61699534400953
(Iteration 7701 / 23430) loss: 0.3647921237904338
(Iteration 7801 / 23430) loss: 0.4504958533910714 5
(Epoch 5 / 15) Training Accuracy: 0.79668, Validation Accuracy: 0.7657353392692662
(Iteration 7901 / 23430) loss: 0.5690834777826277
(Iteration 8001 / 23430) loss: 0.4510437967657443
(Iteration 8101 / 23430) loss: 0.5274783840844547
(Iteration 8201 / 23430) loss: 0.452149075889337
(Iteration 8301 / 23430) loss: 0.7060777210733411
(Iteration 8401 / 23430) loss: 0.5875982555920508
(Iteration 8501 / 23430) loss: 0.7274344938715449
(Iteration 8601 / 23430) loss: 0.8212427423035692
(Iteration 8701 / 23430) loss: 0.8889450275473546
(Iteration 8801 / 23430) loss: 0.3964340775112729
(Iteration 8901 / 23430) loss: 0.6038593053619778
(Iteration 9001 / 23430) loss: 0.763368617663668
(Iteration 9101 / 23430) loss: 0.4532081025025479
(Iteration 9201 / 23430) loss: 0.41670133186450786
(Iteration 9301 / 23430) loss: 0.795283160897778
(Epoch 6 / 15) Training Accuracy: 0.80864, Validation Accuracy: 0.7688056493705864
(Iteration 9401 / 23430) loss: 0.4334185642781243 6
(Iteration 9501 / 23430) loss: 0.622011743341585
(Iteration 9601 / 23430) loss: 0.6329021171593978
(Iteration 9701 / 23430) loss: 0.3978822046923531 4
(Iteration 9801 / 23430) loss: 0.3661655411813911
(Iteration 9901 / 23430) loss: 0.8116032355584102
(Iteration 10001 / 23430) loss: 0.8675740429289939
(Iteration 10101 / 23430) loss: 0.4906492980461693
(Iteration 10201 / 23430) loss: 0.4033115046159259
(Iteration 10301 / 23430) loss: 0.6631065952481878
(Iteration 10401 / 23430) loss: 0.4717227307592729 6
(Iteration 10501 / 23430) loss: 0.8295400486928917
(Iteration 10601 / 23430) loss: 0.6960984589366919
(Iteration 10701 / 23430) loss: 0.4495212762672454
(Iteration 10801 / 23430) loss: 0.4791530213767098
(Iteration 10901 / 23430) loss: 0.8639922977200168
(Epoch 7 / 15) Training Accuracy: 0.8259, Validation Accuracy: 0.7924470371507523
(Iteration 11001 / 23430) loss: 0.3972655678416323
(Iteration 11101 / 23430) loss: 0.4958313802605528
(Iteration 11201 / 23430) loss: 0.9896846500236904
(Iteration 11301 / 23430) loss: 0.4249939268612713 5
(Iteration 11401 / 23430) loss: 0.611164027269647 1
(Iteration 11501 / 23430) loss: 1.0438387928728496
(Iteration 11601 / 23430) loss: 0.9453959778126568
(Iteration 11701 / 23430) loss: 0.3631304006521907
(Iteration 11801 / 23430) loss: 0.2712158206695497
(Iteration 11901 / 23430) loss: 0.657228835370939
(Iteration 12001 / 23430) loss: 0.4264752167210515
(Iteration 12101 / 23430) loss: 0.3839201997442925 3
(Iteration 12201 / 23430) loss: 0.4774727032150260 5
(Iteration 12301 / 23430) loss: 0.3761936997390973 3
(Iteration 12401 / 23430) loss: 0.8582029441391872
(Epoch 8 / 15) Training Accuracy: 0.8373, Validation Accuracy: 0.79060485108996
(Iteration 12501 / 23430) loss: 0.4760155032481815 4
(Iteration 12601 / 23430) loss: 0.3322648376775249 6
(Iteration 12701 / 23430) loss: 0.4811418893207295
(Iteration 12801 / 23430) loss: 0.6584945806393798
(Iteration 12901 / 23430) loss: 0.5997117763889415
(Iteration 13001 / 23430) loss: 0.4555241641716657
(Iteration 13101 / 23430) loss: 0.4736791097391019
(Iteration 13201 / 23430) loss: 0.3811304314061856
(Iteration 13301 / 23430) loss: 0.6454803175878547
(Iteration 13401 / 23430) loss: 0.2572016095043624
(Iteration 13501 / 23430) loss: 0.6383205149478863
(Iteration 13601 / 23430) loss: 0.4539121812023896 3
(Iteration 13701 / 23430) loss: 0.4999886698844855 4
(Iteration 13801 / 23430) loss: 0.6531505230332175
(Iteration 13901 / 23430) loss: 0.8398567273619071
(Iteration 14001 / 23430) loss: 0.6328387896531356
(Epoch 9 / 15) Training Accuracy: 0.85176, Validation Accuracy: 0.7982806263432607
(Iteration 14101 / 23430) loss: 0.4149135831438115 5
(Iteration 14201 / 23430) loss: 0.7708854861712671
(Iteration 14301 / 23430) loss: 0.4002414812794235
(Iteration 14401 / 23430) loss: 0.3988268669555259
(Iteration 14501 / 23430) loss: 0.68367623529875
(Iteration 14601 / 23430) loss: 0.4188064814649359
(Iteration 14701 / 23430) loss: 0.2914205962357875 3
(Iteration 14801 / 23430) loss: 0.398027267288025 4
(Iteration 14901 / 23430) loss: 0.2398404831396695 5
(Iteration 15001 / 23430) loss: 0.3611349528814641
(Iteration 15101 / 23430) loss: 0.3389692110631317
(Iteration 15201 / 23430) loss: 0.5195864694898584
(Iteration 15301 / 23430) loss: 0.5854906223570181
(Iteration 15401 / 23430) loss: 0.4427026158415676 7
(Iteration 15501 / 23430) loss: 0.5008039780625 94
(Iteration 15601 / 23430) loss: 0.2938406043254203 6
(Epoch 10 / 15) Training Accuracy: 0.85476, Validation Accuracy: 0.8096407737181456
(Iteration 15701 / 23430) loss: 0.3752749503572261 3
(Iteration 15801 / 23430) loss: 0.711644154254496 7
(Iteration 15901 / 23430) loss: 0.4559657325057726 6
(Iteration 16001 / 23430) loss: 0.4806658364017934
(Iteration 16101 / 23430) loss: 0.4006977501914086 3
(Iteration 16201 / 23430) loss: 0.6396420282123783
(Iteration 16301 / 23430) loss: 0.550185086428208
(Iteration 16401 / 23430) loss: 0.2939958114003768
(Iteration 16501 / 23430) loss: 0.1565369440334688
(Iteration 16601 / 23430) loss: 0.2844788981989227
(Iteration 16701 / 23430) loss: 0.8178016522440877
(Iteration 16801 / 23430) loss: 0.1164096134801280 7
(Iteration 16901 / 23430) loss: 0.2615797020891710 3
(Iteration 17001 / 23430) loss: 0.3811662244737065
(Iteration 17101 / 23430) loss: 0.5164441621543014
(Epoch 11 / 15) Training Accuracy: 0.83722, Validation Accuracy: 0.7875345409886398
(Iteration 17201 / 23430) loss: 0.4352891627423282
(Iteration 17301 / 23430) loss: 0.5932155689286972
(Iteration 17401 / 23430) loss: 0.2838336113883541 7
(Iteration 17501 / 23430) loss: 0.5388864090608441
(Iteration 17601 / 23430) loss: 0.5409336370330056
(Iteration 17701 / 23430) loss: 0.5303415152740839
(Iteration 17801 / 23430) loss: 0.5116298489679854
(Iteration 17901 / 23430) loss: 0.6660771650846 97
(Iteration 18001 / 23430) loss: 0.3893896682692177
(Iteration 18101 / 23430) loss: 0.8347372077646431
(Iteration 18201 / 23430) loss: 0.9279596652331021
(Iteration 18301 / 23430) loss: 0.4644552058583435 4
(Iteration 18401 / 23430) loss: 0.5547346566518185
(Iteration 18501 / 23430) loss: 0.1844143055762989 5
(Iteration 18601 / 23430) loss: 0.8316744630444095
(Iteration 18701 / 23430) loss: 0.4244181784308888 6
(Epoch 12 / 15) Training Accuracy: 0.86158, Validation Accuracy: 0.8087196806877495
(Iteration 18801 / 23430) loss: 0.6190343523765867
(Iteration 18901 / 23430) loss: 0.5669590870958673
(Iteration 19001 / 23430) loss: 0.3657855032284448 3
(Iteration 19101 / 23430) loss: 0.7798310635078161
(Iteration 19201 / 23430) loss: 0.4405451482954212
(Iteration 19301 / 23430) loss: 0.502256401334464
(Iteration 19401 / 23430) loss: 0.2906597709602201
(Iteration 19501 / 23430) loss: 0.3465054584023735 7
(Iteration 19601 / 23430) loss: 0.3832692978125087 4
(Iteration 19701 / 23430) loss: 0.5665021935560957
(Iteration 19801 / 23430) loss: 0.7450979599307784
(Iteration 19901 / 23430) loss: 0.6298222206980181
(Iteration 20001 / 23430) loss: 0.3920288525382706 3
(Iteration 20101 / 23430) loss: 0.2471469693108144 8
(Iteration 20201 / 23430) loss: 0.2567487781676011
(Iteration 20301 / 23430) loss: 0.6088983191808001
(Epoch 13 / 15) Training Accuracy: 0.86182, Validation Accuracy: 0.803807184525637
(Iteration 20401 / 23430) loss: 0.4666124962639021 7
(Iteration 20501 / 23430) loss: 0.2328678278976840 7
(Iteration 20601 / 23430) loss: 0.2477967191050170 2
(Iteration 20701 / 23430) loss: 0.6689142061139746
(Iteration 20801 / 23430) loss: 0.3034728685475899
(Iteration 20901 / 23430) loss: 0.2653129190801392
(Iteration 21001 / 23430) loss: 0.5516206610757932
(Iteration 21101 / 23430) loss: 0.6291386490318326
(Iteration 21201 / 23430) loss: 0.271119954243002
(Iteration 21301 / 23430) loss: 0.2484651098022108 2
(Iteration 21401 / 23430) loss: 0.1696745673623591
(Iteration 21501 / 23430) loss: 0.5054520031077838
(Iteration 21601 / 23430) loss: 0.4541856300060 15
(Iteration 21701 / 23430) loss: 0.3574442110738731
(Iteration 21801 / 23430) loss: 0.3118203497767116
(Epoch 14 / 15) Training Accuracy: 0.87314, Validation Accuracy: 0.8139392078599939
(Iteration 21901 / 23430) loss: 0.2610465622296 47
(Iteration 22001 / 23430) loss: 0.4744361570494467
(Iteration 22101 / 23430) loss: 0.3200181803049972
(Iteration 22201 / 23430) loss: 0.4080921954938907 5
(Iteration 22301 / 23430) loss: 0.534387839414032
(Iteration 22401 / 23430) loss: 0.3774615697314719 4
(Iteration 22501 / 23430) loss: 0.4676118243160577
(Iteration 22601 / 23430) loss: 0.1101425946535177 2
(Iteration 22701 / 23430) loss: 0.2646969559966893 5
(Iteration 22801 / 23430) loss: 0.0855849545013032 4
(Iteration 22901 / 23430) loss: 0.4201695896557731
(Iteration 23001 / 23430) loss: 0.4147236352728177 5
(Iteration 23101 / 23430) loss: 0.433189336805877
(Iteration 23201 / 23430) loss: 0.5098049986527676
(Iteration 23301 / 23430) loss: 0.6836915984166925
(Iteration 23401 / 23430) loss: 0.548071025345826 1
(Epoch 15 / 15) Training Accuracy: 0.87596, Validation Accuracy: 0.802886091495241

Training with Adam...
(Iteration 1 / 23430) loss: 2.302135570793081
(Iteration 101 / 23430) loss: 2.339730190754850 4
(Iteration 201 / 23430) loss: 2.218307763264379
(Iteration 301 / 23430) loss: 1.956447303098658
(Iteration 401 / 23430) loss: 2.083299469318603
(Iteration 501 / 23430) loss: 2.224377687339785 6
(Iteration 601 / 23430) loss: 2.002047214873617
(Iteration 701 / 23430) loss: 1.850581290689474 3
(Iteration 801 / 23430) loss: 2.080373867752174
(Iteration 901 / 23430) loss: 1.893185040731322
(Iteration 1001 / 23430) loss: 1.700198564500937 2
(Iteration 1101 / 23430) loss: 1.657795141952400 3
(Iteration 1201 / 23430) loss: 1.700215444960796 2
(Iteration 1301 / 23430) loss: 1.569503074665108 9
(Iteration 1401 / 23430) loss: 1.436516129493558 4
(Iteration 1501 / 23430) loss: 1.535373952551311 5
```

```
(Iteration 20701 / 23430) loss: 1.2949190314904315
(Iteration 20801 / 23430) loss: 1.025895032272282
(Iteration 20901 / 23430) loss: 0.86263868718315
(Iteration 21001 / 23430) loss: 1.907379706738212
(Iteration 21101 / 23430) loss: nan
(Iteration 21201 / 23430) loss: nan
(Iteration 21301 / 23430) loss: 1.0922950732567913
(Iteration 21401 / 23430) loss: 0.9963336169280678
(Iteration 21501 / 23430) loss: 1.318747818997291
(Iteration 21601 / 23430) loss: 0.9208030338569784
(Iteration 21701 / 23430) loss: 1.3046442947604038
(Iteration 21801 / 23430) loss: 0.9995483301522496
(Epoch 14 / 15) Training Accuracy: 0.67182, Validation Accuracy: 0.6524408965305496
(Iteration 21901 / 23430) loss: 1.4648407150757525
(Iteration 22001 / 23430) loss: 1.2581721944681952
(Iteration 22101 / 23430) loss: 1.262105122310734
(Iteration 22201 / 23430) loss: 0.6987114626817292
(Iteration 22301 / 23430) loss: 0.5626054212139843
(Iteration 22401 / 23430) loss: nan
(Iteration 22501 / 23430) loss: nan
(Iteration 22601 / 23430) loss: 1.2566499505147886
(Iteration 22701 / 23430) loss: 1.5543254845914996
(Iteration 22801 / 23430) loss: 1.4443558402956231
(Iteration 22901 / 23430) loss: 0.9980651400161022
(Iteration 23001 / 23430) loss: 1.3714654886226365
(Iteration 23101 / 23430) loss: 1.3204313611596366
(Iteration 23201 / 23430) loss: 1.0904614336759597
(Iteration 23301 / 23430) loss: 3.323154823981138
(Iteration 23401 / 23430) loss: 1.1741985060127886
(Epoch 15 / 15) Training Accuracy: 0.70096, Validation Accuracy: 0.6828369665336199
```



Plot the Activation Functions [3pt]

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

In [97]:

```
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: 1.0 / (1+np.exp(-x))
leaky_relu = lambda x: np.maximum(alpha*x, x)
relu = lambda x: np.maximum(0, x)
elu = lambda x: x*(x>0) + elu_alpha*(np.exp(x)-1)*(x<=0)
selu = lambda x: selu_scale*x*(x>0) + selu_scale*selu_alpha*(np.exp(x)-1)*(x<=0)
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, 0.0585369028744, 0.158869104881, 0.364576440742,
                0.635423559258, 0.841130895119, 0.941463097126, 0.979942463462, 0.993307149076],
    "LeakyReLU": [-0.5, -0.388888888889, -0.277777777778, -0.166666666667, -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, 1.66666666667, 2.77777777778, 3.88888888889, 5.0],
    "ELU": [-0.4966310265, -0.489765962143, -0.468911737989, -0.405562198581, -0.213123289631,
            0.555555555556, 1.66666666667, 2.77777777778, 3.88888888889, 5.0],
    "SeLU": [-1.74618571868, -1.72204772347, -1.64872296837, -1.42598202974, -0.749354802287,
             0.583722222222, 1.75116666667, 2.91861111111, 4.08605555556, 5.2535],
    "Tanh": [-0.999909204263, -0.999162466631, -0.992297935288, -0.931109608668, -0.504672397722,
             0.504672397722, 0.931109608668, 0.992297935288, 0.999162466631, 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, label=label)
    assert rel_error(activations[label](XS), GT_Act[label]) < 1e-9, \
          "Your implementation of {} might be wrong".format(label)
    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.