# FullyConnectedNets

May 12, 2025

[1]:
```
# This mounts your Google Drive to the Colab VM.
from google.colab import drive
drive.mount('/content/drive')

# TODO: Enter the foldername in your Drive where you have saved the unzipped
# assignment folder, e.g. 'cs231n/assignments/assignment2/'
FOLDERNAME = "cs231n/assignments/assignment2/"
assert FOLDERNAME is not None, "[!] Enter the foldername."

# Now that we've mounted your Drive, this ensures that
# the Python interpreter of the Colab VM can load
# python files from within it.
import sys
sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))

# This downloads the CIFAR-10 dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

```
Mounted at /content/drive
/content/drive/My Drive/cs231n/assignments/assignment2/cs231n/datasets
/content/drive/My Drive/cs231n/assignments/assignment2
```

## 1 Multi-Layer Fully Connected Network

In this exercise, you will implement a fully connected network with an arbitrary number of hidden layers.

Read through the `FullyConnectedNet` class in the file `cs231n/classifiers/fc_net.py`.

Implement the network initialization, forward pass, and backward pass. Throughout this assignment, you will be implementing layers in `cs231n/layers.py`. You can re-use your implementations for `affine_forward`, `affine_backward`, `relu_forward`, `relu_backward`, and `softmax_loss` from Assignment 1. For right now, don't worry about implementing dropout or batch/layer normalization yet, as you will add those features later.

[2]:
```
# Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifiers.fc_net import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient,␣
 ↪eval_numerical_gradient_array
from cs231n.solver import Solver

%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"

%load_ext autoreload
%autoreload 2

def rel_error(x, y):
    """Returns relative error."""
    return np.max(np.abs(x - y) / (np.maximum(1e-8, np.abs(x) + np.abs(y))))
```

[3]:
```
# Load the (preprocessed) CIFAR-10 data.
data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{k}: {v.shape}")
```

```
X_train: (49000, 3, 32, 32)
y_train: (49000,)
X_val: (1000, 3, 32, 32)
y_val: (1000,)
X_test: (1000, 3, 32, 32)
y_test: (1000,)
```

## 1.1 Initial Loss and Gradient Check

As a sanity check, run the following to check the initial loss and to gradient check the network both with and without regularization. This is a good way to see if the initial losses seem reasonable.

For gradient checking, you should expect to see errors around 1e-7 or less.

[4]:
```
np.random.seed(231)
N, D, H1, H2, C = 2, 15, 20, 30, 10
X = np.random.randn(N, D)
y = np.random.randint(C, size=(N,))

for reg in [0, 3.14]:
    print("Running check with reg = ", reg)
    model = FullyConnectedNet(
        [H1, H2],
        input_dim=D,
        num_classes=C,
        reg=reg,
        weight_scale=5e-2,
        dtype=np.float64
    )

    loss, grads = model.loss(X, y)
    print("Initial loss: ", loss)

    # Most of the errors should be on the order of e-7 or smaller.
    # NOTE: It is fine however to see an error for W2 on the order of e-5
    # for the check when reg = 0.0
    for name in sorted(grads):
        f = lambda _: model.loss(X, y)[0]
        grad_num = eval_numerical_gradient(f, model.params[name],␣
↪verbose=False, h=1e-5)
        print(f"{name} relative error: {rel_error(grad_num, grads[name])}")
```

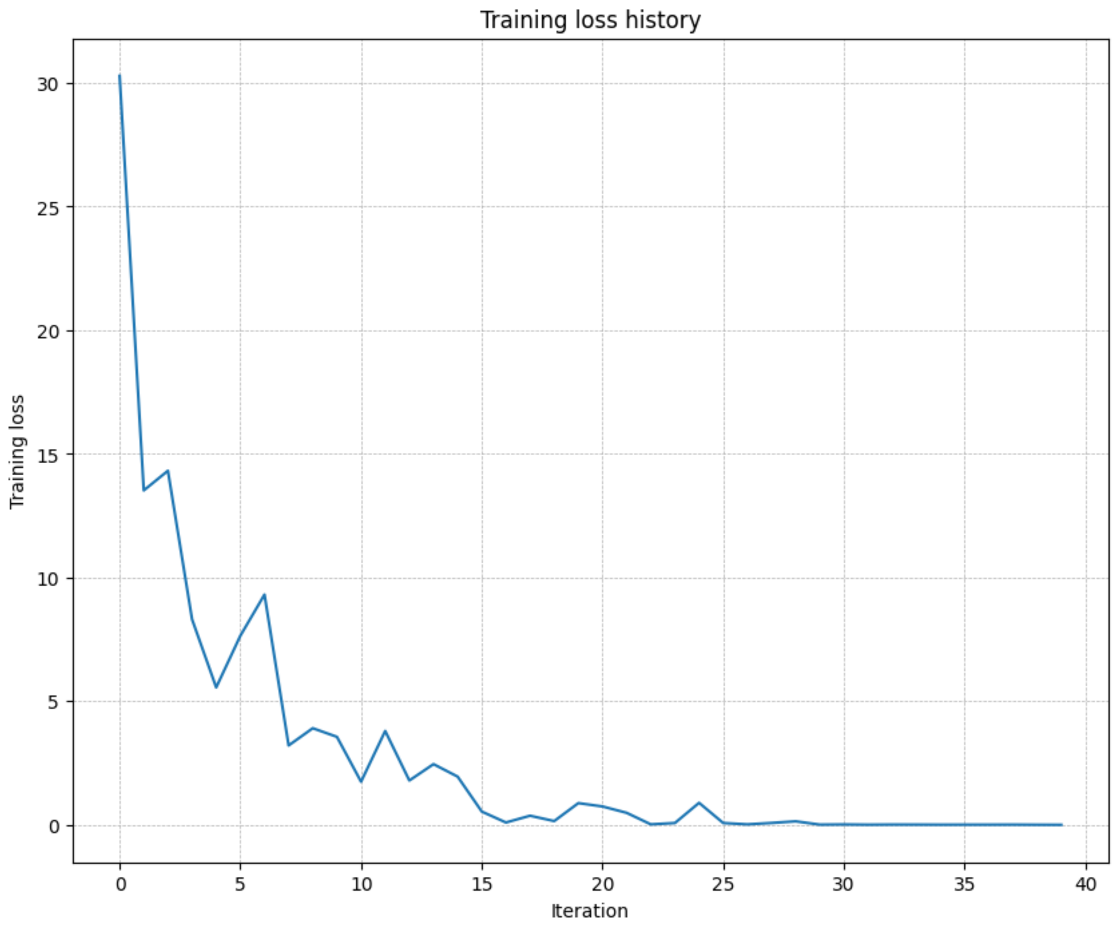
```
Running check with reg =  0
Initial loss:  2.3004790897684924
W1 relative error: 1.4839894098713283e-07
W2 relative error: 2.212047931078 52e-05
W3 relative error: 3.527252851540647e-07
b1 relative error: 5.376386228531692e-09
b2 relative error: 2.085654200257447e-09
b3 relative error: 5.7957243458479405e-11
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 3.904541941902138e-09
W2 relative error: 6.86942277940646e-08
W3 relative error: 2.131129848945024e-08
b1 relative error: 1.4752428222134868e-08
b2 relative error: 1.7223750761525226e-09
b3 relative error: 1.5702714832602802e-10
```

As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy within 20 epochs.

[16]:
```
# TODO: Use a three-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  "X_train": data["X_train"][:num_train],
  "y_train": data["y_train"][:num_train],
  "X_val": data["X_val"],
  "y_val": data["y_val"],
}

learning_rate = (10**-3) * 0.5#(2e-2)  # Experiment with this!
weight_scale = (10**-2) *  4.5 #(2e-2)*1  # Experiment with this!
model = FullyConnectedNet(
    [100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule="sgd",
    optim_config={"learning_rate": learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title("Training loss history")
plt.xlabel("Iteration")
plt.ylabel("Training loss")
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 40) loss: 30.282594
(Epoch 0 / 20) train acc: 0.120000; val_acc: 0.094000
(Epoch 1 / 20) train acc: 0.300000; val_acc: 0.102000
(Epoch 2 / 20) train acc: 0.360000; val_acc: 0.113000
(Epoch 3 / 20) train acc: 0.440000; val_acc: 0.117000
(Epoch 4 / 20) train acc: 0.500000; val_acc: 0.102000
(Epoch 5 / 20) train acc: 0.560000; val_acc: 0.113000
(Iteration 11 / 40) loss: 1.749503
(Epoch 6 / 20) train acc: 0.700000; val_acc: 0.120000
(Epoch 7 / 20) train acc: 0.740000; val_acc: 0.126000
(Epoch 8 / 20) train acc: 0.800000; val_acc: 0.134000
(Epoch 9 / 20) train acc: 0.920000; val_acc: 0.133000
(Epoch 10 / 20) train acc: 0.960000; val_acc: 0.134000
(Iteration 21 / 40) loss: 0.748622
(Epoch 11 / 20) train acc: 0.940000; val_acc: 0.135000
(Epoch 12 / 20) train acc: 0.960000; val_acc: 0.137000
```

```
(Epoch 13 / 20) train acc: 0.960000; val_acc: 0.139000
(Epoch 14 / 20) train acc: 0.980000; val_acc: 0.136000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.133000
(Iteration 31 / 40) loss: 0.021774
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.131000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.130000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.133000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.133000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.132000
```

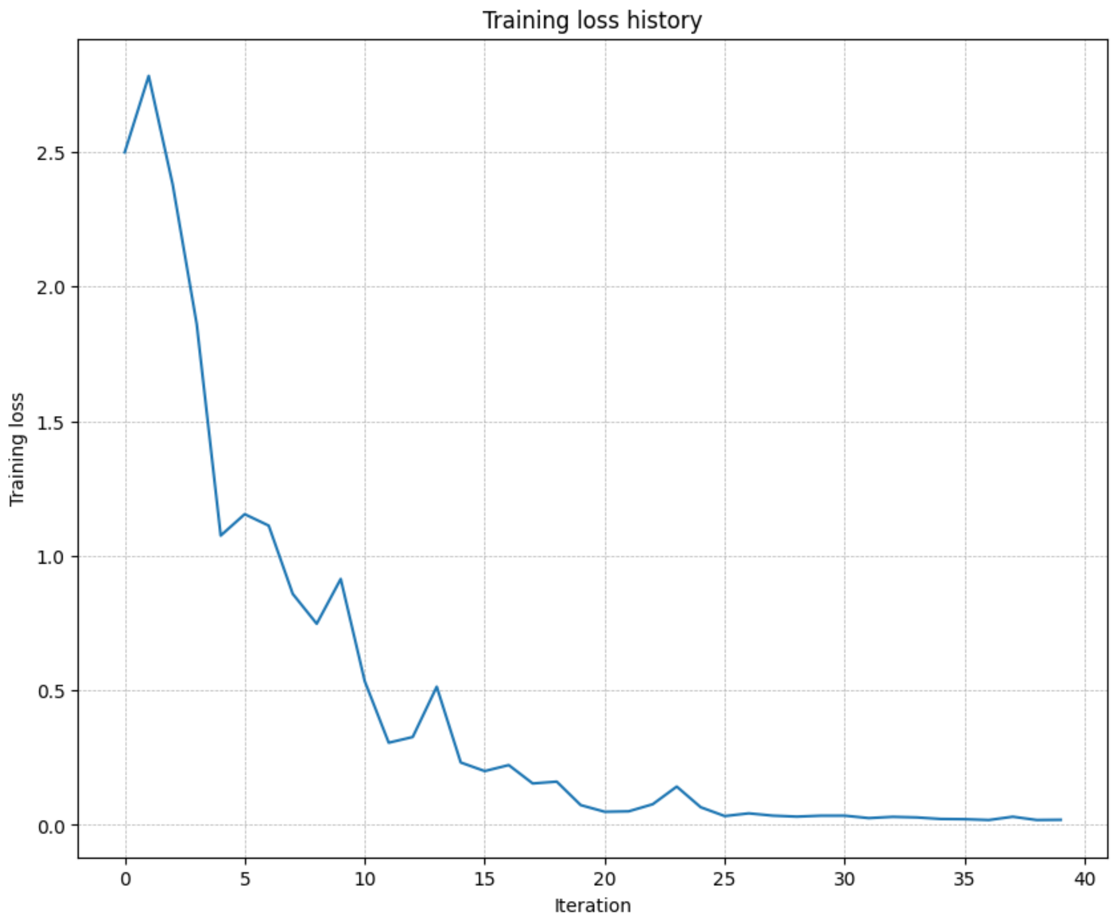


Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

[11]:
```
# TODO: Use a five-layer Net to overfit 50 training examples by
# tweaking just the learning rate and initialization scale.

num_train = 50
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

learning_rate = (10**-1) * 0.15#(2e-2)  # Experiment with this!
weight_scale = (10**-2) *  4.5 #(2e-2)*1  # Experiment with this!

model = FullyConnectedNet(
    [100, 100, 100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=20,
    batch_size=25,
    update_rule='sgd',
    optim_config={'learning_rate': learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title('Training loss history')
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 40) loss: 2.498379
(Epoch 0 / 20) train acc: 0.400000; val_acc: 0.145000
(Epoch 1 / 20) train acc: 0.320000; val_acc: 0.148000
(Epoch 2 / 20) train acc: 0.640000; val_acc: 0.169000
(Epoch 3 / 20) train acc: 0.720000; val_acc: 0.150000
(Epoch 4 / 20) train acc: 0.920000; val_acc: 0.158000
(Epoch 5 / 20) train acc: 0.840000; val_acc: 0.148000
(Iteration 11 / 40) loss: 0.533925
(Epoch 6 / 20) train acc: 0.920000; val_acc: 0.160000
(Epoch 7 / 20) train acc: 0.960000; val_acc: 0.174000
(Epoch 8 / 20) train acc: 0.980000; val_acc: 0.171000
(Epoch 9 / 20) train acc: 0.980000; val_acc: 0.166000
(Epoch 10 / 20) train acc: 1.000000; val_acc: 0.164000
(Iteration 21 / 40) loss: 0.048710
(Epoch 11 / 20) train acc: 1.000000; val_acc: 0.170000
```

```
(Epoch 12 / 20) train acc: 1.000000; val_acc: 0.166000
(Epoch 13 / 20) train acc: 1.000000; val_acc: 0.168000
(Epoch 14 / 20) train acc: 1.000000; val_acc: 0.173000
(Epoch 15 / 20) train acc: 1.000000; val_acc: 0.165000
(Iteration 31 / 40) loss: 0.034238
(Epoch 16 / 20) train acc: 1.000000; val_acc: 0.167000
(Epoch 17 / 20) train acc: 1.000000; val_acc: 0.170000
(Epoch 18 / 20) train acc: 1.000000; val_acc: 0.169000
(Epoch 19 / 20) train acc: 1.000000; val_acc: 0.169000
(Epoch 20 / 20) train acc: 1.000000; val_acc: 0.173000
```



## 1.2 Inline Question 1:

Did you notice anything about the comparative difficulty of training the three-layer network vs. training the five-layer network? In particular, based on your experience, which network seemed more sensitive to the initialization scale? Why do you think that is the case?

### 1.3 Answer:

[the five layer network seems more sensitive to initialisation scale. it often tends to run into infinity if the learning rate is too big. the reason for that i believi is that because there are more layers to pass, it is more likely that the multiplication stacks up and we end up getting a number that goes into infinity.]

## 2 Update rules

So far we have used vanilla stochastic gradient descent (SGD) as our update rule. More sophisticated update rules can make it easier to train deep networks. We will implement a few of the most commonly used update rules and compare them to vanilla SGD.

### 2.1 SGD+Momentum

Stochastic gradient descent with momentum is a widely used update rule that tends to make deep networks converge faster than vanilla stochastic gradient descent. See the Momentum Update section at http://cs231n.github.io/neural-networks-3/#sgd for more information.

Open the file `cs231n/optim.py` and read the documentation at the top of the file to make sure you understand the API. Implement the SGD+momentum update rule in the function `sgd_momentum` and run the following to check your implementation. You should see errors less than e-8.

[ ]:
```
from cs231n.optim import sgd_momentum

N, D = 4, 5
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)

config = {"learning_rate": 1e-3, "velocity": v}
next_w, _ = sgd_momentum(w, dw, config=config)

expected_next_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    ]])

# Should see relative errors around e-8 or less
print("next_w error: ", rel_error(next_w, expected_next_w))
print("velocity error: ", rel_error(expected_velocity, config["velocity"]))
```

```
next_w error:  8.882347033505819e-09
velocity error:  4.269287743278663e-09
```

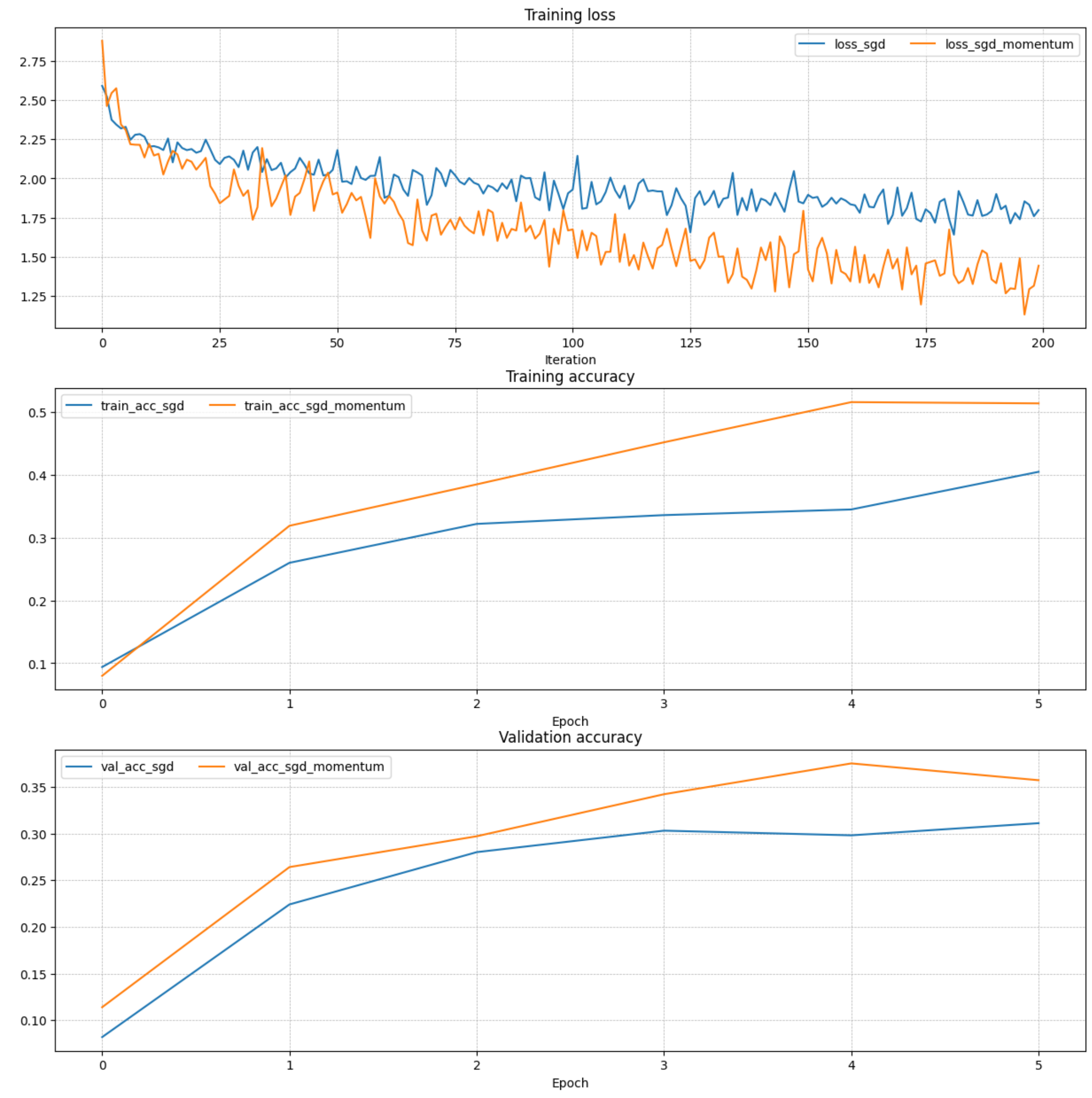
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

```
[ ]: num_train = 4000
     small_data = {
       'X_train': data['X_train'][:num_train],
       'y_train': data['y_train'][:num_train],
       'X_val': data['X_val'],
       'y_val': data['y_val'],
     }

     solvers = {}

     for update_rule in ['sgd', 'sgd_momentum']:
         print('Running with ', update_rule)
         model = FullyConnectedNet(
             [100, 100, 100, 100, 100],
             weight_scale=5e-2
         )

         solver = Solver(
             model,
             small_data,
             num_epochs=5,
             batch_size=100,
             update_rule=update_rule,
             optim_config={'learning_rate': 5e-3},
             verbose=True,
         )
         solvers[update_rule] = solver
         solver.train()

     fig, axes = plt.subplots(3, 1, figsize=(15, 15))

     axes[0].set_title('Training loss')
     axes[0].set_xlabel('Iteration')
     axes[1].set_title('Training accuracy')
     axes[1].set_xlabel('Epoch')
     axes[2].set_title('Validation accuracy')
     axes[2].set_xlabel('Epoch')

     for update_rule, solver in solvers.items():
         axes[0].plot(solver.loss_history, label=f"loss_{update_rule}")
         axes[1].plot(solver.train_acc_history, label=f"train_acc_{update_rule}")
         axes[2].plot(solver.val_acc_history, label=f"val_acc_{update_rule}")
```

```python

for ax in axes:
    ax.legend(loc="best", ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()
```

```
Running with  sgd
(Iteration 1 / 200) loss: 2.591315
(Epoch 0 / 5) train acc: 0.094000; val_acc: 0.082000
(Iteration 11 / 200) loss: 2.203128
(Iteration 21 / 200) loss: 2.164390
(Iteration 31 / 200) loss: 2.177630
(Epoch 1 / 5) train acc: 0.260000; val_acc: 0.224000
(Iteration 41 / 200) loss: 2.040117
(Iteration 51 / 200) loss: 2.181773
(Iteration 61 / 200) loss: 1.876335
(Iteration 71 / 200) loss: 1.893209
(Epoch 2 / 5) train acc: 0.322000; val_acc: 0.280000
(Iteration 81 / 200) loss: 1.960344
(Iteration 91 / 200) loss: 2.000565
(Iteration 101 / 200) loss: 1.929347
(Iteration 111 / 200) loss: 1.875478
(Epoch 3 / 5) train acc: 0.336000; val_acc: 0.303000
(Iteration 121 / 200) loss: 1.765777
(Iteration 131 / 200) loss: 1.920923
(Iteration 141 / 200) loss: 1.871054
(Iteration 151 / 200) loss: 1.895357
(Epoch 4 / 5) train acc: 0.345000; val_acc: 0.298000
(Iteration 161 / 200) loss: 1.828374
(Iteration 171 / 200) loss: 1.761285
(Iteration 181 / 200) loss: 1.738371
(Iteration 191 / 200) loss: 1.900600
(Epoch 5 / 5) train acc: 0.405000; val_acc: 0.311000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 2.878904
(Epoch 0 / 5) train acc: 0.080000; val_acc: 0.114000
(Iteration 11 / 200) loss: 2.220798
(Iteration 21 / 200) loss: 2.056227
(Iteration 31 / 200) loss: 1.888912
(Epoch 1 / 5) train acc: 0.319000; val_acc: 0.264000
(Iteration 41 / 200) loss: 1.766688
(Iteration 51 / 200) loss: 1.910477
(Iteration 61 / 200) loss: 1.839450
(Iteration 71 / 200) loss: 1.762700
(Epoch 2 / 5) train acc: 0.385000; val_acc: 0.297000
(Iteration 81 / 200) loss: 1.790721
```

```
(Iteration 91 / 200) loss: 1.660179
(Iteration 101 / 200) loss: 1.674700
(Iteration 111 / 200) loss: 1.466967
(Epoch 3 / 5) train acc: 0.452000; val_acc: 0.342000
(Iteration 121 / 200) loss: 1.678809
(Iteration 131 / 200) loss: 1.653784
(Iteration 141 / 200) loss: 1.559264
(Iteration 151 / 200) loss: 1.418575
(Epoch 4 / 5) train acc: 0.516000; val_acc: 0.375000
(Iteration 161 / 200) loss: 1.564007
(Iteration 171 / 200) loss: 1.291166
(Iteration 181 / 200) loss: 1.674087
(Iteration 191 / 200) loss: 1.331330
(Epoch 5 / 5) train acc: 0.514000; val_acc: 0.357000
```



### 2.2 RMSProp and Adam

RMSProp [1] and Adam [2] are update rules that set per-parameter learning rates by using a running average of the second moments of gradients.

In the file `cs231n/optim.py`, implement the RMSProp update rule in the `rmsprop` function and implement the Adam update rule in the `adam` function, and check your implementations using the tests below.

**NOTE:** Please implement the *complete* Adam update rule (with the bias correction mechanism), not the first simplified version mentioned in the course notes.

[1] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude." COURSERA: Neural Networks for Machine Learning 4 (2012).

[2] Diederik Kingma and Jimmy Ba, "Adam: A Method for Stochastic Optimization", ICLR 2015.

```
[ ]: # Test RMSProp implementation
     from cs231n.optim import rmsprop

     N, D = 4, 5
     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)

     config = {'learning_rate': 1e-2, 'cache': cache}
     next_w, _ = rmsprop(w, dw, config=config)

     expected_next_w = np.asarray([
       [-0.39223849, -0.34037513, -0.28849239, -0.23659121, -0.18467247],
       [-0.132737,   -0.08078555, -0.02881884,  0.02316247,  0.07515774],
       [ 0.12716641,  0.17918792,  0.23122175,  0.28326742,  0.33532447],
       [ 0.38739248,  0.43947102,  0.49155973,  0.54365823,  0.59576619]])
     expected_cache = np.asarray([
       [ 0.5976,      0.6126277,   0.6277108,   0.64284931,  0.65804321],
       [ 0.67329252,  0.68859723,  0.70395734,  0.71937285,  0.73484377],
       [ 0.75037008,  0.7659518,   0.78158892,  0.79728144,  0.81302936],
       [ 0.82883269,  0.84469141,  0.86060554,  0.87657507,  0.8926    ]])

     # You should see relative errors around e-7 or less
     print('next_w error: ', rel_error(expected_next_w, next_w))
     print('cache error: ', rel_error(expected_cache, config['cache']))
```

```
next_w error:  9.502645229894295e-08
cache error:  2.6477955807156126e-09
```

```
[ ]: # Test Adam implementation
     from cs231n.optim import adam
```

```
N, D = 4, 5
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)

config = {'learning_rate': 1e-2, 'm': m, 'v': v, 't': 5}
next_w, _ = adam(w, dw, config=config)

expected_next_w = np.asarray([
  [-0.40094747, -0.34836187, -0.29577703, -0.24319299, -0.19060977],
  [-0.1380274,  -0.08544591, -0.03286534,  0.01971428,  0.0722929],
  [ 0.1248705,   0.17744702,  0.23002243,  0.28259667,  0.33516969],
  [ 0.38774145,  0.44031188,  0.49288093,  0.54544852,  0.59801459]])
expected_v = np.asarray([
  [ 0.69966,     0.68908382,  0.67851319,  0.66794809,  0.65738853,],
  [ 0.64683452,  0.63628604,  0.6257431,   0.61520571,  0.60467385,],
  [ 0.59414753,  0.58362676,  0.57311152,  0.56260183,  0.55209767,],
  [ 0.54159906,  0.53110598,  0.52061845,  0.51013645,  0.49966,   ]])
expected_m = np.asarray([
  [ 0.48,        0.49947368,  0.51894737,  0.53842105,  0.55789474],
  [ 0.57736842,  0.59684211,  0.61631579,  0.63578947,  0.65526316],
  [ 0.67473684,  0.69421053,  0.71368421,  0.73315789,  0.75263158],
  [ 0.77210526,  0.79157895,  0.81105263,  0.83052632,  0.85      ]])

# You should see relative errors around e-7 or less
print('next_w error: ', rel_error(expected_next_w, next_w))
print('v error: ', rel_error(expected_v, config['v']))
print('m error: ', rel_error(expected_m, config['m']))
```

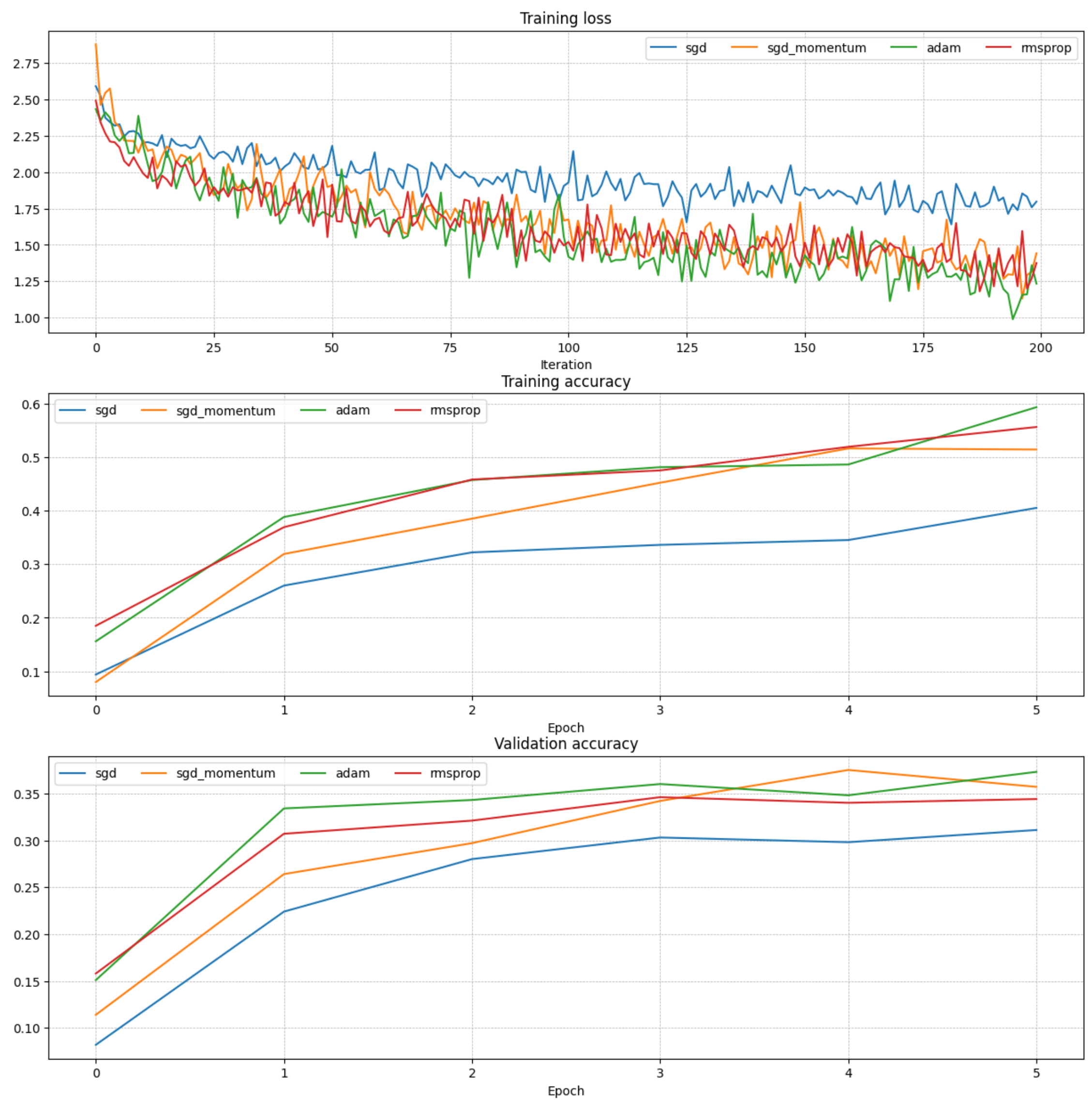
```
next_w error:  1.1398874673331344e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

```
[ ]: learning_rates = {'rmsprop': 1e-4, 'adam': 1e-3}
     for update_rule in ['adam', 'rmsprop']:
         print('Running with ', update_rule)
         model = FullyConnectedNet(
             [100, 100, 100, 100, 100],
             weight_scale=5e-2
         )
         solver = Solver(
             model,
             small_data,
             num_epochs=5,
             batch_size=100,
             update_rule=update_rule,
             optim_config={'learning_rate': learning_rates[update_rule]},
             verbose=True
         )
         solvers[update_rule] = solver
         solver.train()
         print()

     fig, axes = plt.subplots(3, 1, figsize=(15, 15))

     axes[0].set_title('Training loss')
     axes[0].set_xlabel('Iteration')
     axes[1].set_title('Training accuracy')
     axes[1].set_xlabel('Epoch')
     axes[2].set_title('Validation accuracy')
     axes[2].set_xlabel('Epoch')

     for update_rule, solver in solvers.items():
         axes[0].plot(solver.loss_history, label=f"{update_rule}")
         axes[1].plot(solver.train_acc_history, label=f"{update_rule}")
         axes[2].plot(solver.val_acc_history, label=f"{update_rule}")

     for ax in axes:
         ax.legend(loc='best', ncol=4)
         ax.grid(linestyle='--', linewidth=0.5)

     plt.show()
```

```
Running with  adam
(Iteration 1 / 200) loss: 2.433495
(Epoch 0 / 5) train acc: 0.156000; val_acc: 0.151000
(Iteration 11 / 200) loss: 2.182505
(Iteration 21 / 200) loss: 2.106663
(Iteration 31 / 200) loss: 1.686401
(Epoch 1 / 5) train acc: 0.388000; val_acc: 0.334000
(Iteration 41 / 200) loss: 1.691622
(Iteration 51 / 200) loss: 1.693744
(Iteration 61 / 200) loss: 1.718690
(Iteration 71 / 200) loss: 1.692643
(Epoch 2 / 5) train acc: 0.457000; val_acc: 0.343000
(Iteration 81 / 200) loss: 1.654635
(Iteration 91 / 200) loss: 1.507824
(Iteration 101 / 200) loss: 1.417717
(Iteration 111 / 200) loss: 1.396819
(Epoch 3 / 5) train acc: 0.481000; val_acc: 0.360000
```

```
(Iteration 121 / 200) loss: 1.587738
(Iteration 131 / 200) loss: 1.456142
(Iteration 141 / 200) loss: 1.294688
(Iteration 151 / 200) loss: 1.429624
(Epoch 4 / 5) train acc: 0.486000; val_acc: 0.348000
(Iteration 161 / 200) loss: 1.623196
(Iteration 171 / 200) loss: 1.262123
(Iteration 181 / 200) loss: 1.282647
(Iteration 191 / 200) loss: 1.374632
(Epoch 5 / 5) train acc: 0.593000; val_acc: 0.373000

Running with  rmsprop
(Iteration 1 / 200) loss: 2.491872
(Epoch 0 / 5) train acc: 0.185000; val_acc: 0.158000
(Iteration 11 / 200) loss: 1.992030
(Iteration 21 / 200) loss: 1.964633
(Iteration 31 / 200) loss: 1.874463
(Epoch 1 / 5) train acc: 0.369000; val_acc: 0.307000
(Iteration 41 / 200) loss: 1.795615
(Iteration 51 / 200) loss: 1.914654
(Iteration 61 / 200) loss: 1.686577
(Iteration 71 / 200) loss: 1.850106
(Epoch 2 / 5) train acc: 0.458000; val_acc: 0.321000
(Iteration 81 / 200) loss: 1.577173
(Iteration 91 / 200) loss: 1.600542
(Iteration 101 / 200) loss: 1.520053
(Iteration 111 / 200) loss: 1.645688
(Epoch 3 / 5) train acc: 0.475000; val_acc: 0.346000
(Iteration 121 / 200) loss: 1.436737
(Iteration 131 / 200) loss: 1.433785
(Iteration 141 / 200) loss: 1.466901
(Iteration 151 / 200) loss: 1.515337
(Epoch 4 / 5) train acc: 0.519000; val_acc: 0.340000
(Iteration 161 / 200) loss: 1.517383
(Iteration 171 / 200) loss: 1.478138
(Iteration 181 / 200) loss: 1.382041
(Iteration 191 / 200) loss: 1.215322
(Epoch 5 / 5) train acc: 0.556000; val_acc: 0.344000
```



## 2.3 Inline Question 2:

AdaGrad, like Adam, is a per-parameter optimization method that uses the following update rule:

```
cache += dw**2
w += - learning_rate * dw / (np.sqrt(cache) + eps)
```

John notices that when he was training a network with AdaGrad that the updates became very small, and that his network was learning slowly. Using your knowledge of the AdaGrad update rule, why do you think the updates would become very small? Would Adam have the same issue?

## 2.4 Answer:

[because ADAGRAD keeps adding dw to the cache, it means that the denominator in the bottom formula keeps getting smaller and smaller which means that the updates also get smaller. adam would not have this issue, because it uses an exponentially decaying average, which ensures that it never gets too small]

## 3 Train a Good Model!

Train the best fully connected model that you can on CIFAR-10, storing your best model in the `best_model` variable. We require you to get at least 50% accuracy on the validation set using a fully connected network.

If you are careful it should be possible to get accuracies above 55%, but we don't require it for this part and won't assign extra credit for doing so. Later in the assignment we will ask you to train the best convolutional network that you can on CIFAR-10, and we would prefer that you spend your effort working on convolutional networks rather than fully connected networks.

**Note:** You might find it useful to complete the `BatchNormalization.ipynb` and `Dropout.ipynb` notebooks before completing this part, since those techniques can help you train powerful models.

[28]:
```
best_model = None

################################################################################
# TODO: Train the best FullyConnectedNet that you can on CIFAR-10. You might   #
# find batch/layer normalization and dropout useful. Store your best model in  #
# the best_model variable.                                                     #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# best_val = -1     # keep track of best validation accuracy
# best_params = {} # dict to hold best model parameters
# num_train = 700  # amount of training data for tests

# # Generate small dataset
# small_data = {
#   "X_train": data["X_train"][:num_train],
#   "y_train": data["y_train"][:num_train],
#   "X_val": data["X_val"],
#   "y_val": data["y_val"],
# }

# for i in range(40):
#     lr = 10 ** np.random.uniform(-5,-3)
#     ws = 10 ** np.random.uniform(-2.5,-2)
#     reg = 10 ** np.random.uniform(-4,-2)
#     kr = np.random.uniform(0.7, 0.9)

#     model = FullyConnectedNet([256, 128, 64],
#                               weight_scale=ws,
#                               reg=reg,
#                               dropout_keep_ratio=kr,
#                                normalization='batchnorm')
#     solver = Solver(model, small_data,
#                     num_epochs=20, batch_size=256,
#                     update_rule='adam',
#                     optim_config={'learning_rate': lr},
#                     verbose=False)
#     solver.train()
#     new_val = solver.best_val_acc
#     if new_val > best_val:
#         best_val = new_val
#         best_params = {'lr':lr, 'ws':ws, 'reg':reg, 'kr':kr}

#     print(f'lr: {lr:.5f} ws: {ws:.4f}, reg: {reg:.4f}, kr: {kr:.4f}, acc:␣
↪{new_val:.4f}')

# best_model = FullyConnectedNet([500, 300,100],
#                                 weight_scale=best_params['ws'],
#                                 reg=best_params['reg'],
#                                 dropout_keep_ratio=best_params['kr'],
#                                 normalization='batchnorm')


best_model = FullyConnectedNet([500, 300,100],
                               weight_scale=0.0067,
                               reg=0.0062,
                               dropout_keep_ratio=0.8387,
                               normalization='batchnorm')

solver = Solver(best_model, data,
                num_epochs=20, batch_size=128,
                update_rule='adam',
                optim_config={'learning_rate': 0.00042},
                verbose=False)

solver.train()

print(f'validation accuracy: {solver.best_val_acc}')

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
################################################################################
#                              END OF YOUR CODE                                #
################################################################################
```

```
validation accuracy: 0.539
```

## 4 Test Your Model!

Run your best model on the validation and test sets. You should achieve at least 50% accuracy on the validation set.

[29]:
```
y_test_pred = np.argmax(best_model.loss(data['X_test']), axis=1)
y_val_pred = np.argmax(best_model.loss(data['X_val']), axis=1)
print('Validation set accuracy: ', (y_val_pred == data['y_val']).mean())
print('Test set accuracy: ', (y_test_pred == data['y_test']).mean())
```

```
Validation set accuracy:  0.533
Test set accuracy:  0.5
```