FullyConnectedNets

November 29, 2023

[ ]:
```
# 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/assignment1/'
FOLDERNAME = 'Colab Notebooks/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
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/Colab Notebooks/assignment2/cs231n/datasets
/content/drive/My Drive/Colab Notebooks/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.

[ ]:
```
# 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))))
```

[ ]:
```
# 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.

[ ]:
```
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:  2.300479089768492
W1 relative error: 1.0252674471656573e-07
W2 relative error: 2.2120479295080622e-05
W3 relative error: 4.5623278736665505e-07
b1 relative error: 4.6600944653202505e-09
b2 relative error: 2.085654276112763e-09
b3 relative error: 1.689724888469736e-10
Running check with reg =  3.14
Initial loss:  7.052114776533016
W1 relative error: 6.862884860440611e-09
W2 relative error: 3.522821562176466e-08
W3 relative error: 2.6171457283983532e-08
b1 relative error: 1.4752427965311745e-08
b2 relative error: 1.7223751746766738e-09
b3 relative error: 2.378772438198909e-10
```

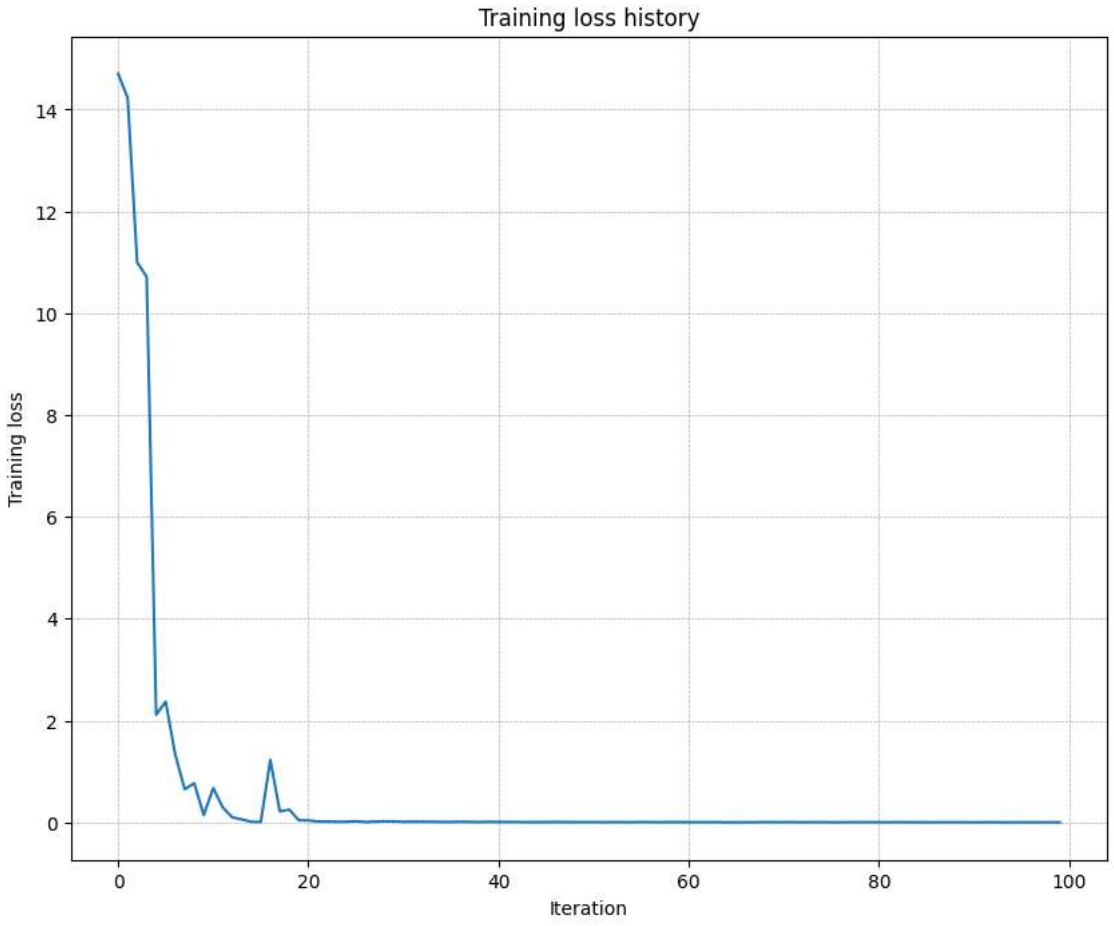
As another sanity check, make sure your network can overfit on a small dataset of 50 images. First, we will try a three-layer network with 100 units in each hidden layer. In the following cell, tweak the **learning rate** and **weight initialization scale** to overfit and achieve 100% training accuracy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

[ ]:
```
# 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"],
}

# Ağırlık ölçeği ve öğrenme oranını deneyerek ayarla
weight_scale = (12e-2) / 3.5
learning_rate = (12e-3) / 3.2
model = FullyConnectedNet(
    [100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=50,
    batch_size=20,
    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 / 100) loss: 14.700066
(Epoch 0 / 50) train acc: 0.260000; val_acc: 0.133000
(Epoch 1 / 50) train acc: 0.300000; val_acc: 0.127000
(Epoch 2 / 50) train acc: 0.440000; val_acc: 0.156000
(Epoch 3 / 50) train acc: 0.680000; val_acc: 0.169000
(Epoch 4 / 50) train acc: 0.780000; val_acc: 0.166000
(Epoch 5 / 50) train acc: 0.880000; val_acc: 0.186000
(Iteration 11 / 100) loss: 0.673543
(Epoch 6 / 50) train acc: 0.940000; val_acc: 0.189000
(Epoch 7 / 50) train acc: 0.940000; val_acc: 0.184000
(Epoch 8 / 50) train acc: 0.940000; val_acc: 0.182000
(Epoch 9 / 50) train acc: 0.980000; val_acc: 0.176000
(Epoch 10 / 50) train acc: 1.000000; val_acc: 0.167000
(Iteration 21 / 100) loss: 0.039527
(Epoch 11 / 50) train acc: 1.000000; val_acc: 0.166000
```

```
(Epoch 12 / 50) train acc: 1.000000; val_acc: 0.167000
(Epoch 13 / 50) train acc: 1.000000; val_acc: 0.167000
(Epoch 14 / 50) train acc: 1.000000; val_acc: 0.167000
(Epoch 15 / 50) train acc: 1.000000; val_acc: 0.170000
(Iteration 31 / 100) loss: 0.012571
(Epoch 16 / 50) train acc: 1.000000; val_acc: 0.173000
(Epoch 17 / 50) train acc: 1.000000; val_acc: 0.174000
(Epoch 18 / 50) train acc: 1.000000; val_acc: 0.174000
(Epoch 19 / 50) train acc: 1.000000; val_acc: 0.174000
(Epoch 20 / 50) train acc: 1.000000; val_acc: 0.173000
(Iteration 41 / 100) loss: 0.009724
(Epoch 21 / 50) train acc: 1.000000; val_acc: 0.172000
(Epoch 22 / 50) train acc: 1.000000; val_acc: 0.174000
(Epoch 23 / 50) train acc: 1.000000; val_acc: 0.175000
(Epoch 24 / 50) train acc: 1.000000; val_acc: 0.175000
(Epoch 25 / 50) train acc: 1.000000; val_acc: 0.175000
(Iteration 51 / 100) loss: 0.006237
(Epoch 26 / 50) train acc: 1.000000; val_acc: 0.176000
(Epoch 27 / 50) train acc: 1.000000; val_acc: 0.179000
(Epoch 28 / 50) train acc: 1.000000; val_acc: 0.179000
(Epoch 29 / 50) train acc: 1.000000; val_acc: 0.178000
(Epoch 30 / 50) train acc: 1.000000; val_acc: 0.177000
(Iteration 61 / 100) loss: 0.006247
(Epoch 31 / 50) train acc: 1.000000; val_acc: 0.177000
(Epoch 32 / 50) train acc: 1.000000; val_acc: 0.177000
(Epoch 33 / 50) train acc: 1.000000; val_acc: 0.177000
(Epoch 34 / 50) train acc: 1.000000; val_acc: 0.176000
(Epoch 35 / 50) train acc: 1.000000; val_acc: 0.176000
(Iteration 71 / 100) loss: 0.005212
(Epoch 36 / 50) train acc: 1.000000; val_acc: 0.177000
(Epoch 37 / 50) train acc: 1.000000; val_acc: 0.176000
(Epoch 38 / 50) train acc: 1.000000; val_acc: 0.177000
(Epoch 39 / 50) train acc: 1.000000; val_acc: 0.179000
(Epoch 40 / 50) train acc: 1.000000; val_acc: 0.179000
(Iteration 81 / 100) loss: 0.004360
(Epoch 41 / 50) train acc: 1.000000; val_acc: 0.179000
(Epoch 42 / 50) train acc: 1.000000; val_acc: 0.178000
(Epoch 43 / 50) train acc: 1.000000; val_acc: 0.177000
(Epoch 44 / 50) train acc: 1.000000; val_acc: 0.177000
(Epoch 45 / 50) train acc: 1.000000; val_acc: 0.177000
(Iteration 91 / 100) loss: 0.001510
(Epoch 46 / 50) train acc: 1.000000; val_acc: 0.177000
(Epoch 47 / 50) train acc: 1.000000; val_acc: 0.179000
(Epoch 48 / 50) train acc: 1.000000; val_acc: 0.178000
(Epoch 49 / 50) train acc: 1.000000; val_acc: 0.178000
(Epoch 50 / 50) train acc: 1.000000; val_acc: 0.178000
```



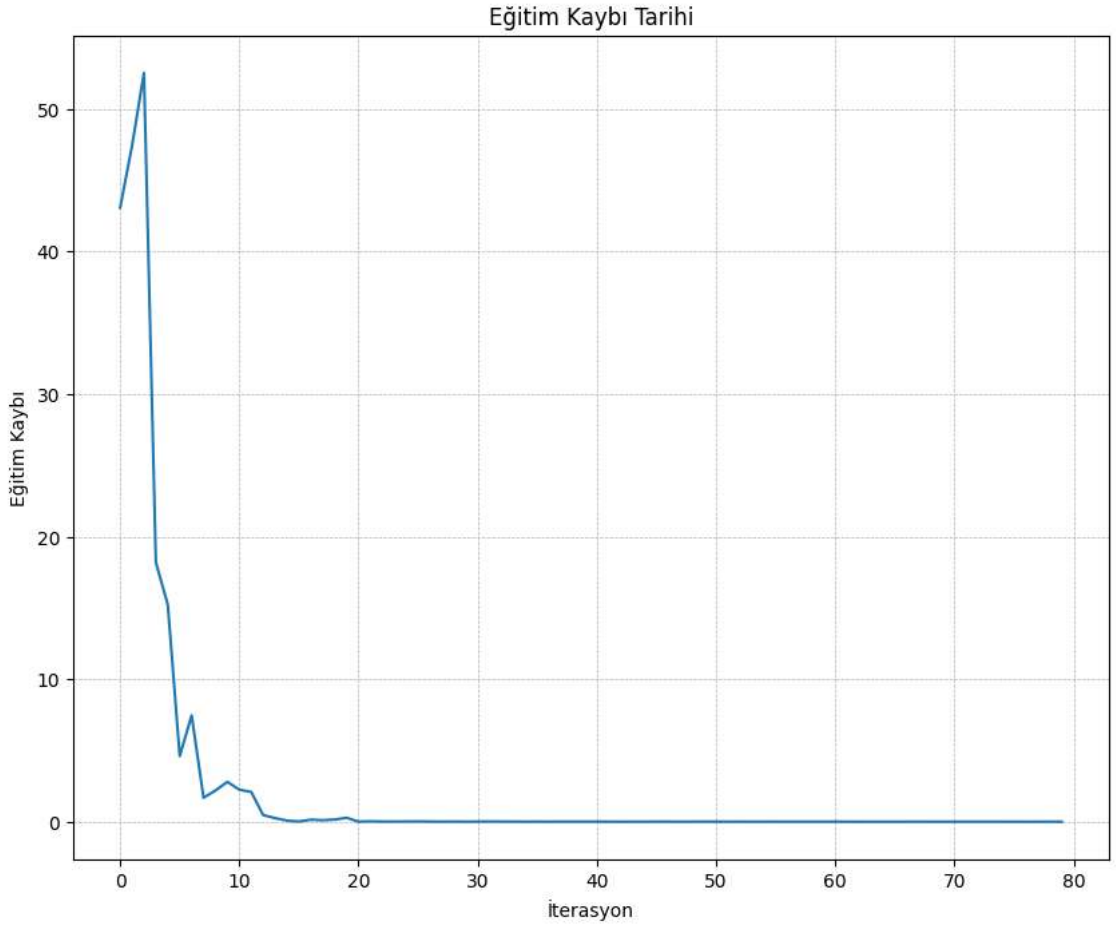
Now, try to use a five-layer network with 100 units on each layer to overfit on 50 training examples. Again, you will have to adjust the learning rate and weight initialization scale, but you should be able to achieve 100% training accuracy within 20 epochs.

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

# Beş katmanlı bir ağ kullanarak 50 eğitim örneği üzerinde aşırı uyuma ulaş

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'],
}

# Ağırlık ölçeği ve öğrenme oranını ayarla
```

```
learning_rate = (27e-3) / 9.2
weight_scale = (64e-2) / 8.1
model = FullyConnectedNet(
    [100, 100, 100, 100],
    weight_scale=weight_scale,
    dtype=np.float64
)
solver = Solver(
    model,
    small_data,
    print_every=10,
    num_epochs=40,
    batch_size=20,
    update_rule='sgd',
    optim_config={'learning_rate': learning_rate},
)
solver.train()

plt.plot(solver.loss_history)
plt.title('Eğitim Kaybı Tarihi')
plt.xlabel('İterasyon')
plt.ylabel('Eğitim Kaybı')
plt.grid(linestyle='--', linewidth=0.5)
plt.show()
```

```
(Iteration 1 / 80) loss: 43.058832
(Epoch 0 / 40) train acc: 0.180000; val_acc: 0.121000
(Epoch 1 / 40) train acc: 0.160000; val_acc: 0.100000
(Epoch 2 / 40) train acc: 0.240000; val_acc: 0.104000
(Epoch 3 / 40) train acc: 0.360000; val_acc: 0.113000
(Epoch 4 / 40) train acc: 0.600000; val_acc: 0.123000
(Epoch 5 / 40) train acc: 0.680000; val_acc: 0.119000
(Iteration 11 / 80) loss: 2.245013
(Epoch 6 / 40) train acc: 0.880000; val_acc: 0.125000
(Epoch 7 / 40) train acc: 0.940000; val_acc: 0.120000
(Epoch 8 / 40) train acc: 0.940000; val_acc: 0.118000
(Epoch 9 / 40) train acc: 0.980000; val_acc: 0.119000
(Epoch 10 / 40) train acc: 1.000000; val_acc: 0.111000
(Iteration 21 / 80) loss: 0.011416
(Epoch 11 / 40) train acc: 1.000000; val_acc: 0.112000
(Epoch 12 / 40) train acc: 1.000000; val_acc: 0.113000
(Epoch 13 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 14 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 15 / 40) train acc: 1.000000; val_acc: 0.117000
(Iteration 31 / 80) loss: 0.014125
(Epoch 16 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 17 / 40) train acc: 1.000000; val_acc: 0.117000
```

```
(Epoch 18 / 40) train acc: 1.000000; val_acc: 0.118000
(Epoch 19 / 40) train acc: 1.000000; val_acc: 0.120000
(Epoch 20 / 40) train acc: 1.000000; val_acc: 0.119000
(Iteration 41 / 80) loss: 0.010765
(Epoch 21 / 40) train acc: 1.000000; val_acc: 0.120000
(Epoch 22 / 40) train acc: 1.000000; val_acc: 0.120000
(Epoch 23 / 40) train acc: 1.000000; val_acc: 0.120000
(Epoch 24 / 40) train acc: 1.000000; val_acc: 0.120000
(Epoch 25 / 40) train acc: 1.000000; val_acc: 0.118000
(Iteration 51 / 80) loss: 0.009642
(Epoch 26 / 40) train acc: 1.000000; val_acc: 0.118000
(Epoch 27 / 40) train acc: 1.000000; val_acc: 0.117000
(Epoch 28 / 40) train acc: 1.000000; val_acc: 0.117000
(Epoch 29 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 30 / 40) train acc: 1.000000; val_acc: 0.117000
(Iteration 61 / 80) loss: 0.008980
(Epoch 31 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 32 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 33 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 34 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 35 / 40) train acc: 1.000000; val_acc: 0.116000
(Iteration 71 / 80) loss: 0.004943
(Epoch 36 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 37 / 40) train acc: 1.000000; val_acc: 0.117000
(Epoch 38 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 39 / 40) train acc: 1.000000; val_acc: 0.116000
(Epoch 40 / 40) train acc: 1.000000; val_acc: 0.117000
```



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:

Üç katmanlı ağ ile beş katmanlı ağın eğitim zorlukları arasında, derin ağların eğitiminde daha fazla zorluk yaşandığını söyleyebiliriz. Özellikle, beş katmanlı ağ, başlangıçtaki ağırlık ölçeklemesine üç katmanlı ağa göre daha duyarlıdır. Bunun başlıca nedenleri şunlardır:

Derinleşen Katmanlar ve Vanishing Gradient Problemi: Ağ derinleştikçe, geri yayılım sırasında vanishing veya exploding riski artar. Bu, ağırlık başlangıç değerlerinin önemini artırır çünkü kötü başlangıç değerleri ne kadar kötü olursa problemler o kadar artar.

Parametre Sayısının Artması ve Overfitting: Beş katmanlı ağda daha fazla parametre bulunur, bu da modelin overfitting riskini artırır. Başlangıçtaki ağırlıkların ölçeklenmesi, ağın karmaşık veri yapılarını öğrenme kapasitesini doğrudan etkiler.

Optimizasyon Zorluğu: Daha derin ağlar genellikle daha karmaşık optimizasyona sahiptirler. Bu, başlangıçtaki ağırlık değerlerinin, yerel minimumlara takılma vb. sorunları önlemede çok daha belirleyici olduğu anlamına gelir.

Bu sebeplerden ötürü, beş katmanlı ağın başlangıçtaki ağırlık ölçeklemesine daha duyarlı olduğunu ve bu ağın eğitiminin daha dikkatli bir şekilde yönetilmesi gerektiğini söyleyebiliriz.

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
```

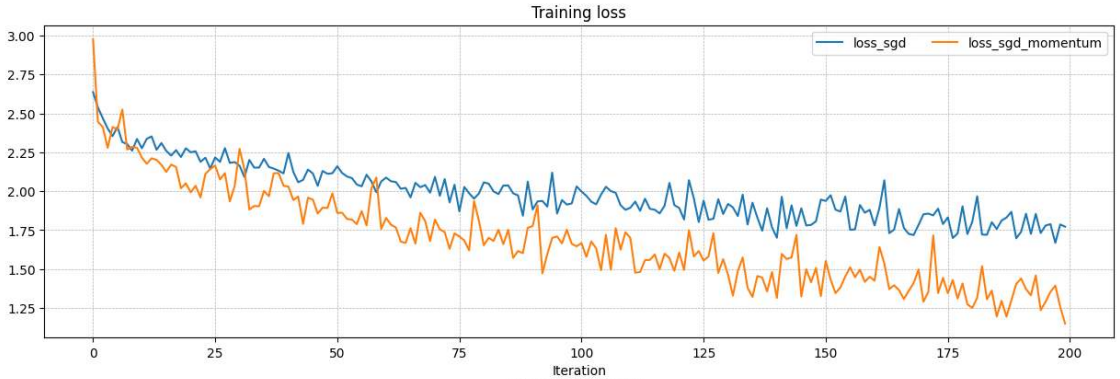
Once you have done so, run the following to train a six-layer network with both SGD and SGD+momentum. You should see the SGD+momentum update rule converge faster.

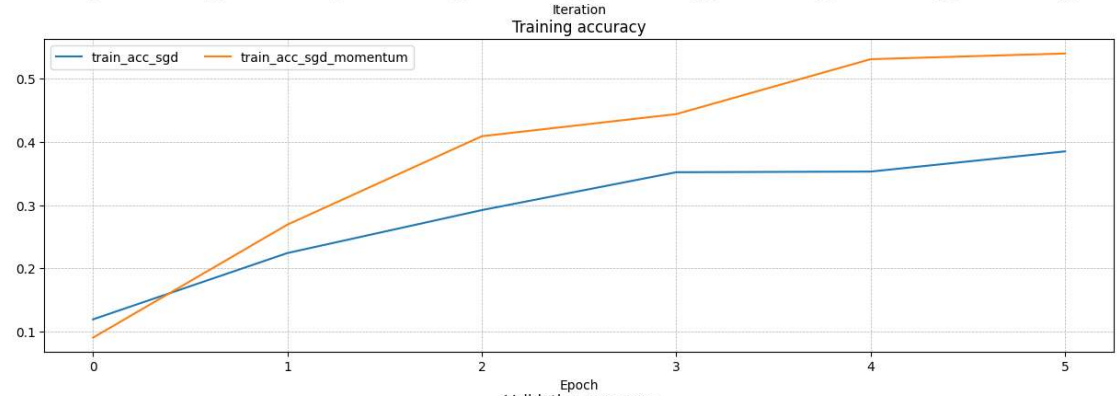
[ ]:
```
num_train = 4000
small_data = {
  'X_train': data['X_train'][:num_train],
  'y_train': data['y_train'][:num_train],
  'X_val': data['X_val'],
  'y_val': data['y_val'],
}

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}")
```

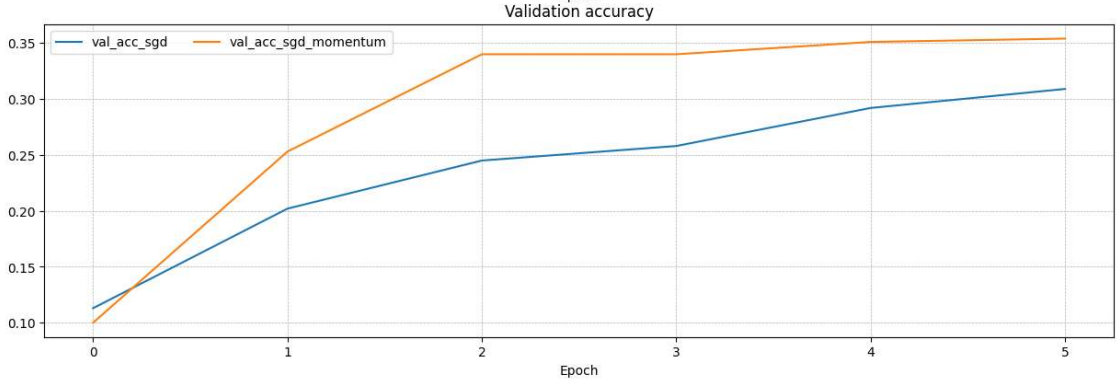
```python
for ax in axes:
    ax.legend(loc="best", ncol=4)
    ax.grid(linestyle='--', linewidth=0.5)

plt.show()
```

```
Running with  sgd
(Iteration 1 / 200) loss: 2.637031
(Epoch 0 / 5) train acc: 0.119000; val_acc: 0.113000
(Iteration 11 / 200) loss: 2.276072
(Iteration 21 / 200) loss: 2.250017
(Iteration 31 / 200) loss: 2.163727
(Epoch 1 / 5) train acc: 0.224000; val_acc: 0.202000
(Iteration 41 / 200) loss: 2.245285
(Iteration 51 / 200) loss: 2.160268
(Iteration 61 / 200) loss: 2.087614
(Iteration 71 / 200) loss: 2.092299
(Epoch 2 / 5) train acc: 0.292000; val_acc: 0.245000
(Iteration 81 / 200) loss: 2.057128
(Iteration 91 / 200) loss: 1.882858
(Iteration 101 / 200) loss: 1.998164
(Iteration 111 / 200) loss: 1.893742
(Epoch 3 / 5) train acc: 0.352000; val_acc: 0.258000
(Iteration 121 / 200) loss: 1.891981
(Iteration 131 / 200) loss: 1.918307
(Iteration 141 / 200) loss: 1.700956
(Iteration 151 / 200) loss: 1.937766
(Epoch 4 / 5) train acc: 0.353000; val_acc: 0.292000
(Iteration 161 / 200) loss: 1.780391
(Iteration 171 / 200) loss: 1.851602
(Iteration 181 / 200) loss: 1.803315
(Iteration 191 / 200) loss: 1.739500
(Epoch 5 / 5) train acc: 0.385000; val_acc: 0.309000
Running with  sgd_momentum
(Iteration 1 / 200) loss: 2.975825
(Epoch 0 / 5) train acc: 0.090000; val_acc: 0.100000
(Iteration 11 / 200) loss: 2.215814
(Iteration 21 / 200) loss: 1.992628
(Iteration 31 / 200) loss: 2.272835
(Epoch 1 / 5) train acc: 0.269000; val_acc: 0.253000
(Iteration 41 / 200) loss: 2.030072
(Iteration 51 / 200) loss: 1.860588
(Iteration 61 / 200) loss: 1.828226
(Iteration 71 / 200) loss: 1.817761
(Epoch 2 / 5) train acc: 0.409000; val_acc: 0.340000
(Iteration 81 / 200) loss: 1.652571
```

```
(Iteration 91 / 200) loss: 1.776060
(Iteration 101 / 200) loss: 1.666702
(Iteration 111 / 200) loss: 1.699783
(Epoch 3 / 5) train acc: 0.444000; val_acc: 0.340000
(Iteration 121 / 200) loss: 1.606134
(Iteration 131 / 200) loss: 1.463447
(Iteration 141 / 200) loss: 1.314465
(Iteration 151 / 200) loss: 1.552248
(Epoch 4 / 5) train acc: 0.531000; val_acc: 0.351000
(Iteration 161 / 200) loss: 1.425115
(Iteration 171 / 200) loss: 1.289689
(Iteration 181 / 200) loss: 1.249902
(Iteration 191 / 200) loss: 1.439334
(Epoch 5 / 5) train acc: 0.540000; val_acc: 0.354000
```







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:  9.524687511038133e-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:  1.1395691798535431e-07
v error:  4.208314038113071e-09
m error:  4.214963193114416e-09
```

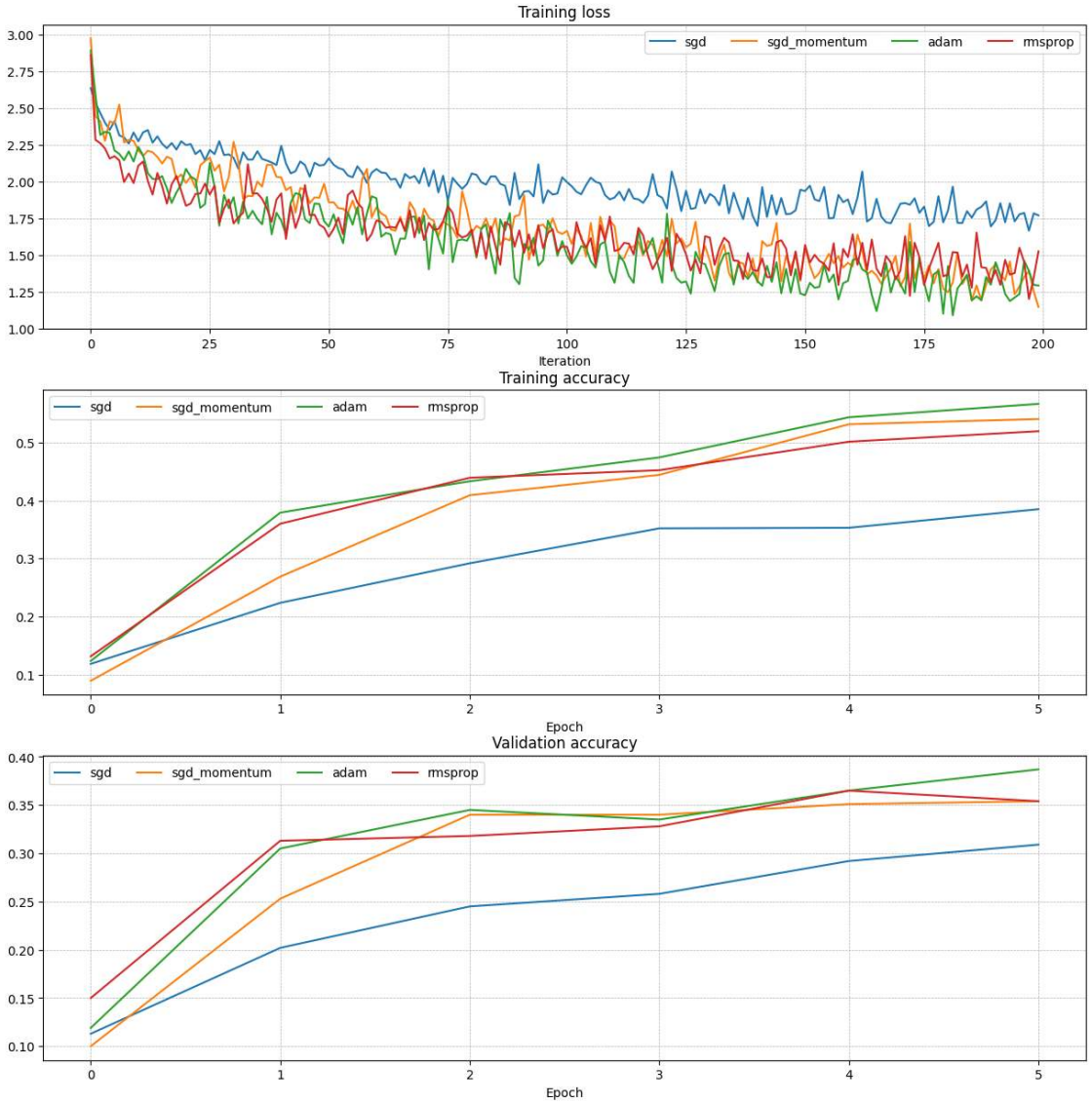
Once you have debugged your RMSProp and Adam implementations, run the following to train a pair of deep networks using these new update rules:

[ ]:
```
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: 2.893860
(Epoch 0 / 5) train acc: 0.124000; val_acc: 0.119000
(Iteration 11 / 200) loss: 2.237626
(Iteration 21 / 200) loss: 2.089030
(Iteration 31 / 200) loss: 1.896352
(Epoch 1 / 5) train acc: 0.379000; val_acc: 0.305000
(Iteration 41 / 200) loss: 1.736844
(Iteration 51 / 200) loss: 1.695719
(Iteration 61 / 200) loss: 1.888564
(Iteration 71 / 200) loss: 1.770833
(Epoch 2 / 5) train acc: 0.433000; val_acc: 0.345000
(Iteration 81 / 200) loss: 1.657866
(Iteration 91 / 200) loss: 1.304299
(Iteration 101 / 200) loss: 1.500369
(Iteration 111 / 200) loss: 1.313729
(Epoch 3 / 5) train acc: 0.474000; val_acc: 0.335000
```

```
(Iteration 121 / 200) loss: 1.314816
(Iteration 131 / 200) loss: 1.352909
(Iteration 141 / 200) loss: 1.326714
(Iteration 151 / 200) loss: 1.228922
(Epoch 4 / 5) train acc: 0.543000; val_acc: 0.365000
(Iteration 161 / 200) loss: 1.467647
(Iteration 171 / 200) loss: 1.305069
(Iteration 181 / 200) loss: 1.427209
(Iteration 191 / 200) loss: 1.453994
(Epoch 5 / 5) train acc: 0.566000; val_acc: 0.387000

Running with  rmsprop
(Iteration 1 / 200) loss: 2.862934
(Epoch 0 / 5) train acc: 0.132000; val_acc: 0.150000
(Iteration 11 / 200) loss: 2.110412
(Iteration 21 / 200) loss: 1.837542
(Iteration 31 / 200) loss: 1.717095
(Epoch 1 / 5) train acc: 0.360000; val_acc: 0.313000
(Iteration 41 / 200) loss: 1.922236
(Iteration 51 / 200) loss: 1.626868
(Iteration 61 / 200) loss: 1.737690
(Iteration 71 / 200) loss: 1.604600
(Epoch 2 / 5) train acc: 0.439000; val_acc: 0.318000
(Iteration 81 / 200) loss: 1.675216
(Iteration 91 / 200) loss: 1.670511
(Iteration 101 / 200) loss: 1.558754
(Iteration 111 / 200) loss: 1.529581
(Epoch 3 / 5) train acc: 0.452000; val_acc: 0.328000
(Iteration 121 / 200) loss: 1.533367
(Iteration 131 / 200) loss: 1.621204
(Iteration 141 / 200) loss: 1.394357
(Iteration 151 / 200) loss: 1.571472
(Epoch 4 / 5) train acc: 0.501000; val_acc: 0.365000
(Iteration 161 / 200) loss: 1.644023
(Iteration 171 / 200) loss: 1.395494
(Iteration 181 / 200) loss: 1.370596
(Iteration 191 / 200) loss: 1.400469
(Epoch 5 / 5) train acc: 0.519000; val_acc: 0.354000
```



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:

2.4.1 AdaGrad

Bu formülde, cache değişkeni geçmiş gradyanların karelerinin toplamına eşttir. Eğitim ilerledikçe cache değeri artar, bu da paydadaki np.sqrt(cache) teriminin de üstel olarak artması demektir. Bundan dolayı ağırlık güncellemesi gittikçe azalır. Yani, modelin ağırlıkları aşırı yavaş bir şekilde güncellenir ve bu da öğrenmenin aşırı yavaşlamasına yol açar.

Adam algoritması ise AdaGrad'dan farklı olarak, gradyanların geçmiş karelerinin hareketli ortalamasını kullanır ve bu hareketli ortalama bir azalma faktörü (beta2 parametresi) ile düzeltilir. Bu sayede Adam, AdaGrad'ın karşılaştığı "sürekli azalan öğrenme hızı" sıkıntısını önemsenmeyecek kadar azaltır. Adam, gradyanların geçmiş değerlerini unutmak için bir mekanizma içerir. AdaGrad geçmişteki tüm gradyanları eşit ağırlıkta hesaba katar. Dolayısıyla, Adam en azından AdaGrad'a kıyasla aşırı az etkilenir. Adam'ın bu özelliği, onun pratikte daha iyi performans vermesini sağlar.

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.

[ ]:
```
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_model = None
best_stats = None
lr = [0.001, 0.0002, 0.0007, 0.00005]  # Öğrenme oranı
reg = [0.1, 0.02, 0.005, 0.001, 0.0001]  # Düzenleme katsayısı
best_val = 0
# Generate small dataset
for l_rate in lr:  # Azaltılmış döngü sayısı
    for reg_rate in reg:
      model = FullyConnectedNet([256, 128, 100],
                                weight_scale=0.003,
                                reg=reg_rate,
```

```
                            normalization='batchnorm')

      solver = Solver(model, data,
                      num_epochs=20, batch_size=512,
                      update_rule='adam',
                      optim_config={'learning_rate': l_rate},
                      verbose=True)

      solver.train()
      new_val = solver.best_val_acc

      if new_val > best_val:
          best_val = new_val
          best_model = model
          best_stats = solver


# Print the best accuracy using small data
print(f'Best validation accuracy using a small dataset: {best_val}')
print(f'\nTraining with best parameters...')


# Print the final validation accuracy acquired with the best model
print(f'Best validation accuracy using full dataset: {best_stats.best_val_acc}')
print(f'Best Model {best_model}')

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

```
(Iteration 1 / 1900) loss: 2.679357
(Epoch 0 / 20) train acc: 0.133000; val_acc: 0.144000
(Iteration 11 / 1900) loss: 2.209749
(Iteration 21 / 1900) loss: 2.093836
(Iteration 31 / 1900) loss: 2.034794
(Iteration 41 / 1900) loss: 2.018439
(Iteration 51 / 1900) loss: 1.990886
(Iteration 61 / 1900) loss: 2.022931
(Iteration 71 / 1900) loss: 1.965182
(Iteration 81 / 1900) loss: 1.997955
(Iteration 91 / 1900) loss: 1.993196
(Epoch 1 / 20) train acc: 0.382000; val_acc: 0.397000
(Iteration 101 / 1900) loss: 1.917465
(Iteration 111 / 1900) loss: 1.976612
(Iteration 121 / 1900) loss: 1.967720
(Iteration 131 / 1900) loss: 1.961333
```

```
(Iteration 141 / 1900) loss: 1.974562
(Iteration 151 / 1900) loss: 1.910070
(Iteration 161 / 1900) loss: 1.932271
(Iteration 171 / 1900) loss: 1.932054
(Iteration 181 / 1900) loss: 1.994601
(Epoch 2 / 20) train acc: 0.369000; val_acc: 0.395000
(Iteration 191 / 1900) loss: 2.025122
(Iteration 201 / 1900) loss: 1.875690
(Iteration 211 / 1900) loss: 1.949251
(Iteration 221 / 1900) loss: 1.936501
(Iteration 231 / 1900) loss: 1.846486
(Iteration 241 / 1900) loss: 1.909235
(Iteration 251 / 1900) loss: 1.895896
(Iteration 261 / 1900) loss: 1.904108
(Iteration 271 / 1900) loss: 1.909575
(Iteration 281 / 1900) loss: 1.788905
(Epoch 3 / 20) train acc: 0.430000; val_acc: 0.414000
(Iteration 291 / 1900) loss: 1.867342
(Iteration 301 / 1900) loss: 1.973167
(Iteration 311 / 1900) loss: 1.930922
(Iteration 321 / 1900) loss: 1.957640
(Iteration 331 / 1900) loss: 1.975967
(Iteration 341 / 1900) loss: 1.982149
(Iteration 351 / 1900) loss: 1.971762
(Iteration 361 / 1900) loss: 1.868783
(Iteration 371 / 1900) loss: 1.896892
(Epoch 4 / 20) train acc: 0.409000; val_acc: 0.451000
(Iteration 381 / 1900) loss: 1.894730
(Iteration 391 / 1900) loss: 1.846210
(Iteration 401 / 1900) loss: 1.918294
(Iteration 411 / 1900) loss: 1.897439
(Iteration 421 / 1900) loss: 1.906778
(Iteration 431 / 1900) loss: 1.805065
(Iteration 441 / 1900) loss: 1.955580
(Iteration 451 / 1900) loss: 1.940690
(Iteration 461 / 1900) loss: 1.867390
(Iteration 471 / 1900) loss: 1.775204
(Epoch 5 / 20) train acc: 0.448000; val_acc: 0.431000
(Iteration 481 / 1900) loss: 1.911987
(Iteration 491 / 1900) loss: 1.850751
(Iteration 501 / 1900) loss: 1.893559
(Iteration 511 / 1900) loss: 1.893417
(Iteration 521 / 1900) loss: 1.884139
(Iteration 531 / 1900) loss: 1.872112
(Iteration 541 / 1900) loss: 1.869302
(Iteration 551 / 1900) loss: 1.892205
(Iteration 561 / 1900) loss: 1.793086
(Epoch 6 / 20) train acc: 0.440000; val_acc: 0.432000
```

```
(Iteration 571 / 1900) loss: 1.840912
(Iteration 581 / 1900) loss: 1.825457
(Iteration 591 / 1900) loss: 1.855356
(Iteration 601 / 1900) loss: 1.897190
(Iteration 611 / 1900) loss: 1.814888
(Iteration 621 / 1900) loss: 1.761882
(Iteration 631 / 1900) loss: 1.791450
(Iteration 641 / 1900) loss: 1.861751
(Iteration 651 / 1900) loss: 1.786863
(Iteration 661 / 1900) loss: 1.797008
(Epoch 7 / 20) train acc: 0.444000; val_acc: 0.442000
(Iteration 671 / 1900) loss: 1.870841
(Iteration 681 / 1900) loss: 1.833830
(Iteration 691 / 1900) loss: 1.824262
(Iteration 701 / 1900) loss: 1.802745
(Iteration 711 / 1900) loss: 1.841572
(Iteration 721 / 1900) loss: 1.796260
(Iteration 731 / 1900) loss: 1.877793
(Iteration 741 / 1900) loss: 1.790044
(Iteration 751 / 1900) loss: 1.848726
(Epoch 8 / 20) train acc: 0.465000; val_acc: 0.460000
(Iteration 761 / 1900) loss: 1.878522
(Iteration 771 / 1900) loss: 1.926338
(Iteration 781 / 1900) loss: 1.765227
(Iteration 791 / 1900) loss: 1.804239
(Iteration 801 / 1900) loss: 1.815717
(Iteration 811 / 1900) loss: 1.822980
(Iteration 821 / 1900) loss: 1.794735
(Iteration 831 / 1900) loss: 1.828056
(Iteration 841 / 1900) loss: 1.838801
(Iteration 851 / 1900) loss: 1.887106
(Epoch 9 / 20) train acc: 0.455000; val_acc: 0.444000
(Iteration 861 / 1900) loss: 1.839657
(Iteration 871 / 1900) loss: 1.884413
(Iteration 881 / 1900) loss: 1.873417
(Iteration 891 / 1900) loss: 1.867850
(Iteration 901 / 1900) loss: 1.874594
(Iteration 911 / 1900) loss: 1.777965
(Iteration 921 / 1900) loss: 1.833041
(Iteration 931 / 1900) loss: 1.832119
(Iteration 941 / 1900) loss: 1.760342
(Epoch 10 / 20) train acc: 0.462000; val_acc: 0.436000
(Iteration 951 / 1900) loss: 1.768560
(Iteration 961 / 1900) loss: 1.738371
(Iteration 971 / 1900) loss: 1.758354
(Iteration 981 / 1900) loss: 1.706288
(Iteration 991 / 1900) loss: 1.806465
(Iteration 1001 / 1900) loss: 1.728474
```

```
(Iteration 1011 / 1900) loss: 1.780598
(Iteration 1021 / 1900) loss: 1.788253
(Iteration 1031 / 1900) loss: 1.807892
(Iteration 1041 / 1900) loss: 1.789219
(Epoch 11 / 20) train acc: 0.496000; val_acc: 0.477000
(Iteration 1051 / 1900) loss: 1.834847
(Iteration 1061 / 1900) loss: 1.807945
(Iteration 1071 / 1900) loss: 1.777932
(Iteration 1081 / 1900) loss: 1.695813
(Iteration 1091 / 1900) loss: 1.837609
(Iteration 1101 / 1900) loss: 1.742449
(Iteration 1111 / 1900) loss: 1.787364
(Iteration 1121 / 1900) loss: 1.792285
(Iteration 1131 / 1900) loss: 1.810255
(Epoch 12 / 20) train acc: 0.452000; val_acc: 0.454000
(Iteration 1141 / 1900) loss: 1.755310
(Iteration 1151 / 1900) loss: 1.801077
(Iteration 1161 / 1900) loss: 1.771969
(Iteration 1171 / 1900) loss: 1.735277
(Iteration 1181 / 1900) loss: 1.798197
(Iteration 1191 / 1900) loss: 1.839270
(Iteration 1201 / 1900) loss: 1.852431
(Iteration 1211 / 1900) loss: 1.749740
(Iteration 1221 / 1900) loss: 1.818834
(Iteration 1231 / 1900) loss: 1.772838
(Epoch 13 / 20) train acc: 0.475000; val_acc: 0.471000
(Iteration 1241 / 1900) loss: 1.693778
(Iteration 1251 / 1900) loss: 1.778273
(Iteration 1261 / 1900) loss: 1.777346
(Iteration 1271 / 1900) loss: 1.789136
(Iteration 1281 / 1900) loss: 1.699529
(Iteration 1291 / 1900) loss: 1.777157
(Iteration 1301 / 1900) loss: 1.764424
(Iteration 1311 / 1900) loss: 1.716912
(Iteration 1321 / 1900) loss: 1.737463
(Epoch 14 / 20) train acc: 0.471000; val_acc: 0.468000
(Iteration 1331 / 1900) loss: 1.758377
(Iteration 1341 / 1900) loss: 1.831196
(Iteration 1351 / 1900) loss: 1.753808
(Iteration 1361 / 1900) loss: 1.776344
(Iteration 1371 / 1900) loss: 1.824623
(Iteration 1381 / 1900) loss: 1.785444
(Iteration 1391 / 1900) loss: 1.729090
(Iteration 1401 / 1900) loss: 1.730972
(Iteration 1411 / 1900) loss: 1.779591
(Iteration 1421 / 1900) loss: 1.769795
(Epoch 15 / 20) train acc: 0.472000; val_acc: 0.463000
(Iteration 1431 / 1900) loss: 1.822817
```

```
(Iteration 1441 / 1900) loss: 1.737855
(Iteration 1451 / 1900) loss: 1.762452
(Iteration 1461 / 1900) loss: 1.700954
(Iteration 1471 / 1900) loss: 1.705848
(Iteration 1481 / 1900) loss: 1.792007
(Iteration 1491 / 1900) loss: 1.832163
(Iteration 1501 / 1900) loss: 1.799956
(Iteration 1511 / 1900) loss: 1.765143
(Epoch 16 / 20) train acc: 0.488000; val_acc: 0.464000
(Iteration 1521 / 1900) loss: 1.768393
(Iteration 1531 / 1900) loss: 1.683293
(Iteration 1541 / 1900) loss: 1.787472
(Iteration 1551 / 1900) loss: 1.753784
(Iteration 1561 / 1900) loss: 1.751067
(Iteration 1571 / 1900) loss: 1.764645
(Iteration 1581 / 1900) loss: 1.718022
(Iteration 1591 / 1900) loss: 1.753017
(Iteration 1601 / 1900) loss: 1.792705
(Iteration 1611 / 1900) loss: 1.739156
(Epoch 17 / 20) train acc: 0.474000; val_acc: 0.489000
(Iteration 1621 / 1900) loss: 1.736099
(Iteration 1631 / 1900) loss: 1.736378
(Iteration 1641 / 1900) loss: 1.700109
(Iteration 1651 / 1900) loss: 1.785905
(Iteration 1661 / 1900) loss: 1.781563
(Iteration 1671 / 1900) loss: 1.786225
(Iteration 1681 / 1900) loss: 1.723514
(Iteration 1691 / 1900) loss: 1.675964
(Iteration 1701 / 1900) loss: 1.698232
(Epoch 18 / 20) train acc: 0.501000; val_acc: 0.477000
(Iteration 1711 / 1900) loss: 1.828309
(Iteration 1721 / 1900) loss: 1.685830
(Iteration 1731 / 1900) loss: 1.644116
(Iteration 1741 / 1900) loss: 1.702934
(Iteration 1751 / 1900) loss: 1.659497
(Iteration 1761 / 1900) loss: 1.677116
(Iteration 1771 / 1900) loss: 1.722388
(Iteration 1781 / 1900) loss: 1.689236
(Iteration 1791 / 1900) loss: 1.724259
(Iteration 1801 / 1900) loss: 1.767895
(Epoch 19 / 20) train acc: 0.456000; val_acc: 0.454000
(Iteration 1811 / 1900) loss: 1.703714
(Iteration 1821 / 1900) loss: 1.828681
(Iteration 1831 / 1900) loss: 1.751499
(Iteration 1841 / 1900) loss: 1.735350
(Iteration 1851 / 1900) loss: 1.627818
(Iteration 1861 / 1900) loss: 1.782480
(Iteration 1871 / 1900) loss: 1.750077
```

```
(Iteration 1881 / 1900) loss: 1.697299
(Iteration 1891 / 1900) loss: 1.684052
(Epoch 20 / 20) train acc: 0.490000; val_acc: 0.482000
(Iteration 1 / 1900) loss: 2.380519
(Epoch 0 / 20) train acc: 0.153000; val_acc: 0.171000
(Iteration 11 / 1900) loss: 2.181075
(Iteration 21 / 1900) loss: 2.010707
(Iteration 31 / 1900) loss: 1.967962
(Iteration 41 / 1900) loss: 1.867443
(Iteration 51 / 1900) loss: 1.806419
(Iteration 61 / 1900) loss: 1.843624
(Iteration 71 / 1900) loss: 1.801211
(Iteration 81 / 1900) loss: 1.840590
(Iteration 91 / 1900) loss: 1.819132
(Epoch 1 / 20) train acc: 0.386000; val_acc: 0.451000
(Iteration 101 / 1900) loss: 1.735601
(Iteration 111 / 1900) loss: 1.845518
(Iteration 121 / 1900) loss: 1.737900
(Iteration 131 / 1900) loss: 1.795683
(Iteration 141 / 1900) loss: 1.733746
(Iteration 151 / 1900) loss: 1.705122
(Iteration 161 / 1900) loss: 1.746402
(Iteration 171 / 1900) loss: 1.733175
(Iteration 181 / 1900) loss: 1.656140
(Epoch 2 / 20) train acc: 0.453000; val_acc: 0.436000
(Iteration 191 / 1900) loss: 1.703233
(Iteration 201 / 1900) loss: 1.689527
(Iteration 211 / 1900) loss: 1.717265
(Iteration 221 / 1900) loss: 1.695312
(Iteration 231 / 1900) loss: 1.697810
(Iteration 241 / 1900) loss: 1.646781
(Iteration 251 / 1900) loss: 1.745044
(Iteration 261 / 1900) loss: 1.730791
(Iteration 271 / 1900) loss: 1.641020
(Iteration 281 / 1900) loss: 1.625003
(Epoch 3 / 20) train acc: 0.496000; val_acc: 0.477000
(Iteration 291 / 1900) loss: 1.675289
(Iteration 301 / 1900) loss: 1.655974
(Iteration 311 / 1900) loss: 1.624536
(Iteration 321 / 1900) loss: 1.611670
(Iteration 331 / 1900) loss: 1.633179
(Iteration 341 / 1900) loss: 1.610627
(Iteration 351 / 1900) loss: 1.591028
(Iteration 361 / 1900) loss: 1.671826
(Iteration 371 / 1900) loss: 1.571953
(Epoch 4 / 20) train acc: 0.507000; val_acc: 0.475000
(Iteration 381 / 1900) loss: 1.545174
(Iteration 391 / 1900) loss: 1.625366
```

```
(Iteration 401 / 1900) loss: 1.576695
(Iteration 411 / 1900) loss: 1.634949
(Iteration 421 / 1900) loss: 1.709486
(Iteration 431 / 1900) loss: 1.621553
(Iteration 441 / 1900) loss: 1.674508
(Iteration 451 / 1900) loss: 1.644429
(Iteration 461 / 1900) loss: 1.635978
(Iteration 471 / 1900) loss: 1.648621
(Epoch 5 / 20) train acc: 0.502000; val_acc: 0.454000
(Iteration 481 / 1900) loss: 1.630649
(Iteration 491 / 1900) loss: 1.533806
(Iteration 501 / 1900) loss: 1.558495
(Iteration 511 / 1900) loss: 1.619530
(Iteration 521 / 1900) loss: 1.559491
(Iteration 531 / 1900) loss: 1.563270
(Iteration 541 / 1900) loss: 1.658221
(Iteration 551 / 1900) loss: 1.627345
(Iteration 561 / 1900) loss: 1.611118
(Epoch 6 / 20) train acc: 0.505000; val_acc: 0.500000
(Iteration 571 / 1900) loss: 1.630190
(Iteration 581 / 1900) loss: 1.622979
(Iteration 591 / 1900) loss: 1.574358
(Iteration 601 / 1900) loss: 1.588225
(Iteration 611 / 1900) loss: 1.595882
(Iteration 621 / 1900) loss: 1.557523
(Iteration 631 / 1900) loss: 1.650264
(Iteration 641 / 1900) loss: 1.519648
(Iteration 651 / 1900) loss: 1.677996
(Iteration 661 / 1900) loss: 1.590979
(Epoch 7 / 20) train acc: 0.509000; val_acc: 0.496000
(Iteration 671 / 1900) loss: 1.506276
(Iteration 681 / 1900) loss: 1.560702
(Iteration 691 / 1900) loss: 1.608212
(Iteration 701 / 1900) loss: 1.646595
(Iteration 711 / 1900) loss: 1.627829
(Iteration 721 / 1900) loss: 1.655222
(Iteration 731 / 1900) loss: 1.618830
(Iteration 741 / 1900) loss: 1.537690
(Iteration 751 / 1900) loss: 1.568463
(Epoch 8 / 20) train acc: 0.507000; val_acc: 0.501000
(Iteration 761 / 1900) loss: 1.571831
(Iteration 771 / 1900) loss: 1.607245
(Iteration 781 / 1900) loss: 1.645673
(Iteration 791 / 1900) loss: 1.537943
(Iteration 801 / 1900) loss: 1.540360
(Iteration 811 / 1900) loss: 1.574111
(Iteration 821 / 1900) loss: 1.548032
(Iteration 831 / 1900) loss: 1.642564
```

```
(Iteration 841 / 1900) loss: 1.561947
(Iteration 851 / 1900) loss: 1.526757
(Epoch 9 / 20) train acc: 0.567000; val_acc: 0.517000
(Iteration 861 / 1900) loss: 1.543584
(Iteration 871 / 1900) loss: 1.525594
(Iteration 881 / 1900) loss: 1.589973
(Iteration 891 / 1900) loss: 1.638133
(Iteration 901 / 1900) loss: 1.511533
(Iteration 911 / 1900) loss: 1.553492
(Iteration 921 / 1900) loss: 1.467456
(Iteration 931 / 1900) loss: 1.576132
(Iteration 941 / 1900) loss: 1.668384
(Epoch 10 / 20) train acc: 0.543000; val_acc: 0.522000
(Iteration 951 / 1900) loss: 1.589152
(Iteration 961 / 1900) loss: 1.627747
(Iteration 971 / 1900) loss: 1.567402
(Iteration 981 / 1900) loss: 1.564754
(Iteration 991 / 1900) loss: 1.487896
(Iteration 1001 / 1900) loss: 1.595423
(Iteration 1011 / 1900) loss: 1.555208
(Iteration 1021 / 1900) loss: 1.595369
(Iteration 1031 / 1900) loss: 1.517428
(Iteration 1041 / 1900) loss: 1.623296
(Epoch 11 / 20) train acc: 0.526000; val_acc: 0.478000
(Iteration 1051 / 1900) loss: 1.549756
(Iteration 1061 / 1900) loss: 1.587728
(Iteration 1071 / 1900) loss: 1.565927
(Iteration 1081 / 1900) loss: 1.480610
(Iteration 1091 / 1900) loss: 1.443694
(Iteration 1101 / 1900) loss: 1.441323
(Iteration 1111 / 1900) loss: 1.537747
(Iteration 1121 / 1900) loss: 1.520318
(Iteration 1131 / 1900) loss: 1.508557
(Epoch 12 / 20) train acc: 0.537000; val_acc: 0.520000
(Iteration 1141 / 1900) loss: 1.491301
(Iteration 1151 / 1900) loss: 1.492943
(Iteration 1161 / 1900) loss: 1.566419
(Iteration 1171 / 1900) loss: 1.625028
(Iteration 1181 / 1900) loss: 1.528595
(Iteration 1191 / 1900) loss: 1.481879
(Iteration 1201 / 1900) loss: 1.554795
(Iteration 1211 / 1900) loss: 1.549666
(Iteration 1221 / 1900) loss: 1.542212
(Iteration 1231 / 1900) loss: 1.558610
(Epoch 13 / 20) train acc: 0.530000; val_acc: 0.504000
(Iteration 1241 / 1900) loss: 1.444742
(Iteration 1251 / 1900) loss: 1.575837
(Iteration 1261 / 1900) loss: 1.464337
```

```
(Iteration 1271 / 1900) loss: 1.530099
(Iteration 1281 / 1900) loss: 1.622193
(Iteration 1291 / 1900) loss: 1.511570
(Iteration 1301 / 1900) loss: 1.607795
(Iteration 1311 / 1900) loss: 1.513013
(Iteration 1321 / 1900) loss: 1.540747
(Epoch 14 / 20) train acc: 0.556000; val_acc: 0.503000
(Iteration 1331 / 1900) loss: 1.508784
(Iteration 1341 / 1900) loss: 1.494240
(Iteration 1351 / 1900) loss: 1.558139
(Iteration 1361 / 1900) loss: 1.558598
(Iteration 1371 / 1900) loss: 1.507456
(Iteration 1381 / 1900) loss: 1.546800
(Iteration 1391 / 1900) loss: 1.487825
(Iteration 1401 / 1900) loss: 1.483195
(Iteration 1411 / 1900) loss: 1.503558
(Iteration 1421 / 1900) loss: 1.534626
(Epoch 15 / 20) train acc: 0.541000; val_acc: 0.502000
(Iteration 1431 / 1900) loss: 1.508959
(Iteration 1441 / 1900) loss: 1.598725
(Iteration 1451 / 1900) loss: 1.505104
(Iteration 1461 / 1900) loss: 1.563534
(Iteration 1471 / 1900) loss: 1.558038
(Iteration 1481 / 1900) loss: 1.538331
(Iteration 1491 / 1900) loss: 1.483556
(Iteration 1501 / 1900) loss: 1.611493
(Iteration 1511 / 1900) loss: 1.482337
(Epoch 16 / 20) train acc: 0.577000; val_acc: 0.511000
(Iteration 1521 / 1900) loss: 1.568039
(Iteration 1531 / 1900) loss: 1.552198
(Iteration 1541 / 1900) loss: 1.478373
(Iteration 1551 / 1900) loss: 1.628189
(Iteration 1561 / 1900) loss: 1.598362
(Iteration 1571 / 1900) loss: 1.568674
(Iteration 1581 / 1900) loss: 1.553155
(Iteration 1591 / 1900) loss: 1.520456
(Iteration 1601 / 1900) loss: 1.548075
(Iteration 1611 / 1900) loss: 1.456344
(Epoch 17 / 20) train acc: 0.556000; val_acc: 0.494000
(Iteration 1621 / 1900) loss: 1.550521
(Iteration 1631 / 1900) loss: 1.432146
(Iteration 1641 / 1900) loss: 1.468251
(Iteration 1651 / 1900) loss: 1.430247
(Iteration 1661 / 1900) loss: 1.461843
(Iteration 1671 / 1900) loss: 1.494911
(Iteration 1681 / 1900) loss: 1.478536
(Iteration 1691 / 1900) loss: 1.510070
(Iteration 1701 / 1900) loss: 1.514246
```

```
(Epoch 18 / 20) train acc: 0.547000; val_acc: 0.509000
(Iteration 1711 / 1900) loss: 1.607033
(Iteration 1721 / 1900) loss: 1.438459
(Iteration 1731 / 1900) loss: 1.463518
(Iteration 1741 / 1900) loss: 1.504430
(Iteration 1751 / 1900) loss: 1.541119
(Iteration 1761 / 1900) loss: 1.441804
(Iteration 1771 / 1900) loss: 1.459104
(Iteration 1781 / 1900) loss: 1.486363
(Iteration 1791 / 1900) loss: 1.573180
(Iteration 1801 / 1900) loss: 1.540023
(Epoch 19 / 20) train acc: 0.541000; val_acc: 0.500000
(Iteration 1811 / 1900) loss: 1.470533
(Iteration 1821 / 1900) loss: 1.459237
(Iteration 1831 / 1900) loss: 1.489219
(Iteration 1841 / 1900) loss: 1.434070
(Iteration 1851 / 1900) loss: 1.509014
(Iteration 1861 / 1900) loss: 1.553762
(Iteration 1871 / 1900) loss: 1.455813
(Iteration 1881 / 1900) loss: 1.438898
(Iteration 1891 / 1900) loss: 1.495462
(Epoch 20 / 20) train acc: 0.550000; val_acc: 0.513000
(Iteration 1 / 1900) loss: 2.321202
(Epoch 0 / 20) train acc: 0.157000; val_acc: 0.152000
(Iteration 11 / 1900) loss: 2.143799
(Iteration 21 / 1900) loss: 1.983889
(Iteration 31 / 1900) loss: 1.841764
(Iteration 41 / 1900) loss: 1.809212
(Iteration 51 / 1900) loss: 1.705120
(Iteration 61 / 1900) loss: 1.741039
(Iteration 71 / 1900) loss: 1.652356
(Iteration 81 / 1900) loss: 1.669533
(Iteration 91 / 1900) loss: 1.667899
(Epoch 1 / 20) train acc: 0.428000; val_acc: 0.434000
(Iteration 101 / 1900) loss: 1.596691
(Iteration 111 / 1900) loss: 1.645333
(Iteration 121 / 1900) loss: 1.592955
(Iteration 131 / 1900) loss: 1.603876
(Iteration 141 / 1900) loss: 1.599276
(Iteration 151 / 1900) loss: 1.534424
(Iteration 161 / 1900) loss: 1.510317
(Iteration 171 / 1900) loss: 1.528779
(Iteration 181 / 1900) loss: 1.462326
(Epoch 2 / 20) train acc: 0.488000; val_acc: 0.485000
(Iteration 191 / 1900) loss: 1.489540
(Iteration 201 / 1900) loss: 1.575458
(Iteration 211 / 1900) loss: 1.486248
(Iteration 221 / 1900) loss: 1.462462
```

```
(Iteration 231 / 1900) loss: 1.582497
(Iteration 241 / 1900) loss: 1.499676
(Iteration 251 / 1900) loss: 1.535904
(Iteration 261 / 1900) loss: 1.453153
(Iteration 271 / 1900) loss: 1.572479
(Iteration 281 / 1900) loss: 1.505894
(Epoch 3 / 20) train acc: 0.510000; val_acc: 0.506000
(Iteration 291 / 1900) loss: 1.466722
(Iteration 301 / 1900) loss: 1.426087
(Iteration 311 / 1900) loss: 1.443042
(Iteration 321 / 1900) loss: 1.448859
(Iteration 331 / 1900) loss: 1.390336
(Iteration 341 / 1900) loss: 1.563205
(Iteration 351 / 1900) loss: 1.460829
(Iteration 361 / 1900) loss: 1.484444
(Iteration 371 / 1900) loss: 1.498791
(Epoch 4 / 20) train acc: 0.544000; val_acc: 0.509000
(Iteration 381 / 1900) loss: 1.502293
(Iteration 391 / 1900) loss: 1.418086
(Iteration 401 / 1900) loss: 1.506798
(Iteration 411 / 1900) loss: 1.494566
(Iteration 421 / 1900) loss: 1.362517
(Iteration 431 / 1900) loss: 1.419014
(Iteration 441 / 1900) loss: 1.464054
(Iteration 451 / 1900) loss: 1.371261
(Iteration 461 / 1900) loss: 1.450421
(Iteration 471 / 1900) loss: 1.360704
(Epoch 5 / 20) train acc: 0.527000; val_acc: 0.527000
(Iteration 481 / 1900) loss: 1.432127
(Iteration 491 / 1900) loss: 1.315664
(Iteration 501 / 1900) loss: 1.462834
(Iteration 511 / 1900) loss: 1.422153
(Iteration 521 / 1900) loss: 1.434387
(Iteration 531 / 1900) loss: 1.430362
(Iteration 541 / 1900) loss: 1.343677
(Iteration 551 / 1900) loss: 1.489846
(Iteration 561 / 1900) loss: 1.464659
(Epoch 6 / 20) train acc: 0.574000; val_acc: 0.509000
(Iteration 571 / 1900) loss: 1.367320
(Iteration 581 / 1900) loss: 1.333153
(Iteration 591 / 1900) loss: 1.464906
(Iteration 601 / 1900) loss: 1.388522
(Iteration 611 / 1900) loss: 1.425034
(Iteration 621 / 1900) loss: 1.398598
(Iteration 631 / 1900) loss: 1.373417
(Iteration 641 / 1900) loss: 1.408902
(Iteration 651 / 1900) loss: 1.411965
(Iteration 661 / 1900) loss: 1.388026
```

```
(Epoch 7 / 20) train acc: 0.550000; val_acc: 0.544000
(Iteration 671 / 1900) loss: 1.384385
(Iteration 681 / 1900) loss: 1.481932
(Iteration 691 / 1900) loss: 1.461829
(Iteration 701 / 1900) loss: 1.387024
(Iteration 711 / 1900) loss: 1.437693
(Iteration 721 / 1900) loss: 1.274433
(Iteration 731 / 1900) loss: 1.415526
(Iteration 741 / 1900) loss: 1.370655
(Iteration 751 / 1900) loss: 1.416025
(Epoch 8 / 20) train acc: 0.580000; val_acc: 0.522000
(Iteration 761 / 1900) loss: 1.358626
(Iteration 771 / 1900) loss: 1.368847
(Iteration 781 / 1900) loss: 1.384688
(Iteration 791 / 1900) loss: 1.338876
(Iteration 801 / 1900) loss: 1.321971
(Iteration 811 / 1900) loss: 1.341362
(Iteration 821 / 1900) loss: 1.473528
(Iteration 831 / 1900) loss: 1.356980
(Iteration 841 / 1900) loss: 1.342896
(Iteration 851 / 1900) loss: 1.331968
(Epoch 9 / 20) train acc: 0.582000; val_acc: 0.509000
(Iteration 861 / 1900) loss: 1.331974
(Iteration 871 / 1900) loss: 1.341338
(Iteration 881 / 1900) loss: 1.309016
(Iteration 891 / 1900) loss: 1.265737
(Iteration 901 / 1900) loss: 1.236906
(Iteration 911 / 1900) loss: 1.329965
(Iteration 921 / 1900) loss: 1.349286
(Iteration 931 / 1900) loss: 1.382987
(Iteration 941 / 1900) loss: 1.270604
(Epoch 10 / 20) train acc: 0.597000; val_acc: 0.534000
(Iteration 951 / 1900) loss: 1.345976
(Iteration 961 / 1900) loss: 1.336534
(Iteration 971 / 1900) loss: 1.384524
(Iteration 981 / 1900) loss: 1.268437
(Iteration 991 / 1900) loss: 1.352073
(Iteration 1001 / 1900) loss: 1.338805
(Iteration 1011 / 1900) loss: 1.333330
(Iteration 1021 / 1900) loss: 1.335336
(Iteration 1031 / 1900) loss: 1.306046
(Iteration 1041 / 1900) loss: 1.257091
(Epoch 11 / 20) train acc: 0.646000; val_acc: 0.527000
(Iteration 1051 / 1900) loss: 1.310518
(Iteration 1061 / 1900) loss: 1.276181
(Iteration 1071 / 1900) loss: 1.230690
(Iteration 1081 / 1900) loss: 1.325739
(Iteration 1091 / 1900) loss: 1.299824
```

```
(Iteration 1101 / 1900) loss: 1.380404
(Iteration 1111 / 1900) loss: 1.330323
(Iteration 1121 / 1900) loss: 1.362317
(Iteration 1131 / 1900) loss: 1.294032
(Epoch 12 / 20) train acc: 0.629000; val_acc: 0.544000
(Iteration 1141 / 1900) loss: 1.296441
(Iteration 1151 / 1900) loss: 1.282996
(Iteration 1161 / 1900) loss: 1.339555
(Iteration 1171 / 1900) loss: 1.359952
(Iteration 1181 / 1900) loss: 1.303825
(Iteration 1191 / 1900) loss: 1.324190
(Iteration 1201 / 1900) loss: 1.437180
(Iteration 1211 / 1900) loss: 1.215503
(Iteration 1221 / 1900) loss: 1.321545
(Iteration 1231 / 1900) loss: 1.279818
(Epoch 13 / 20) train acc: 0.611000; val_acc: 0.539000
(Iteration 1241 / 1900) loss: 1.331840
(Iteration 1251 / 1900) loss: 1.310825
(Iteration 1261 / 1900) loss: 1.374704
(Iteration 1271 / 1900) loss: 1.305059
(Iteration 1281 / 1900) loss: 1.347270
(Iteration 1291 / 1900) loss: 1.347915
(Iteration 1301 / 1900) loss: 1.329464
(Iteration 1311 / 1900) loss: 1.199812
(Iteration 1321 / 1900) loss: 1.271038
(Epoch 14 / 20) train acc: 0.619000; val_acc: 0.542000
(Iteration 1331 / 1900) loss: 1.334029
(Iteration 1341 / 1900) loss: 1.252200
(Iteration 1351 / 1900) loss: 1.239886
(Iteration 1361 / 1900) loss: 1.293299
(Iteration 1371 / 1900) loss: 1.365231
(Iteration 1381 / 1900) loss: 1.382237
(Iteration 1391 / 1900) loss: 1.259520
(Iteration 1401 / 1900) loss: 1.339887
(Iteration 1411 / 1900) loss: 1.355098
(Iteration 1421 / 1900) loss: 1.284235
(Epoch 15 / 20) train acc: 0.642000; val_acc: 0.533000
(Iteration 1431 / 1900) loss: 1.264202
(Iteration 1441 / 1900) loss: 1.259360
(Iteration 1451 / 1900) loss: 1.244155
(Iteration 1461 / 1900) loss: 1.320082
(Iteration 1471 / 1900) loss: 1.302424
(Iteration 1481 / 1900) loss: 1.368368
(Iteration 1491 / 1900) loss: 1.289579
(Iteration 1501 / 1900) loss: 1.224878
(Iteration 1511 / 1900) loss: 1.310927
(Epoch 16 / 20) train acc: 0.625000; val_acc: 0.538000
(Iteration 1521 / 1900) loss: 1.360361
```

```
(Iteration 1531 / 1900) loss: 1.275177
(Iteration 1541 / 1900) loss: 1.283980
(Iteration 1551 / 1900) loss: 1.331961
(Iteration 1561 / 1900) loss: 1.314242
(Iteration 1571 / 1900) loss: 1.352035
(Iteration 1581 / 1900) loss: 1.225521
(Iteration 1591 / 1900) loss: 1.238277
(Iteration 1601 / 1900) loss: 1.309751
(Iteration 1611 / 1900) loss: 1.209846
(Epoch 17 / 20) train acc: 0.632000; val_acc: 0.555000
(Iteration 1621 / 1900) loss: 1.216556
(Iteration 1631 / 1900) loss: 1.207451
(Iteration 1641 / 1900) loss: 1.257808
(Iteration 1651 / 1900) loss: 1.302746
(Iteration 1661 / 1900) loss: 1.305491
(Iteration 1671 / 1900) loss: 1.318707
(Iteration 1681 / 1900) loss: 1.206000
(Iteration 1691 / 1900) loss: 1.309647
(Iteration 1701 / 1900) loss: 1.265377
(Epoch 18 / 20) train acc: 0.595000; val_acc: 0.518000
(Iteration 1711 / 1900) loss: 1.268620
(Iteration 1721 / 1900) loss: 1.273145
(Iteration 1731 / 1900) loss: 1.156186
(Iteration 1741 / 1900) loss: 1.282469
(Iteration 1751 / 1900) loss: 1.215969
(Iteration 1761 / 1900) loss: 1.236712
(Iteration 1771 / 1900) loss: 1.277850
(Iteration 1781 / 1900) loss: 1.285525
(Iteration 1791 / 1900) loss: 1.300698
(Iteration 1801 / 1900) loss: 1.232829
(Epoch 19 / 20) train acc: 0.640000; val_acc: 0.538000
(Iteration 1811 / 1900) loss: 1.351599
(Iteration 1821 / 1900) loss: 1.196560
(Iteration 1831 / 1900) loss: 1.305445
(Iteration 1841 / 1900) loss: 1.293984
(Iteration 1851 / 1900) loss: 1.218764
(Iteration 1861 / 1900) loss: 1.323866
(Iteration 1871 / 1900) loss: 1.206985
(Iteration 1881 / 1900) loss: 1.161490
(Iteration 1891 / 1900) loss: 1.208003
(Epoch 20 / 20) train acc: 0.664000; val_acc: 0.546000
(Iteration 1 / 1900) loss: 2.307716
(Epoch 0 / 20) train acc: 0.108000; val_acc: 0.122000
(Iteration 11 / 1900) loss: 2.113170
(Iteration 21 / 1900) loss: 1.944325
(Iteration 31 / 1900) loss: 1.866287
(Iteration 41 / 1900) loss: 1.785371
(Iteration 51 / 1900) loss: 1.708562
```

```
(Iteration 61 / 1900) loss: 1.626870
(Iteration 71 / 1900) loss: 1.598382
(Iteration 81 / 1900) loss: 1.560815
(Iteration 91 / 1900) loss: 1.564530
(Epoch 1 / 20) train acc: 0.437000; val_acc: 0.429000
(Iteration 101 / 1900) loss: 1.521536
(Iteration 111 / 1900) loss: 1.494055
(Iteration 121 / 1900) loss: 1.510559
(Iteration 131 / 1900) loss: 1.476960
(Iteration 141 / 1900) loss: 1.471340
(Iteration 151 / 1900) loss: 1.471892
(Iteration 161 / 1900) loss: 1.422993
(Iteration 171 / 1900) loss: 1.477125
(Iteration 181 / 1900) loss: 1.498318
(Epoch 2 / 20) train acc: 0.521000; val_acc: 0.493000
(Iteration 191 / 1900) loss: 1.394467
(Iteration 201 / 1900) loss: 1.364926
(Iteration 211 / 1900) loss: 1.406749
(Iteration 221 / 1900) loss: 1.342033
(Iteration 231 / 1900) loss: 1.372520
(Iteration 241 / 1900) loss: 1.398371
(Iteration 251 / 1900) loss: 1.389489
(Iteration 261 / 1900) loss: 1.299040
(Iteration 271 / 1900) loss: 1.311133
(Iteration 281 / 1900) loss: 1.441156
(Epoch 3 / 20) train acc: 0.526000; val_acc: 0.508000
(Iteration 291 / 1900) loss: 1.344449
(Iteration 301 / 1900) loss: 1.309987
(Iteration 311 / 1900) loss: 1.293808
(Iteration 321 / 1900) loss: 1.259423
(Iteration 331 / 1900) loss: 1.236342
(Iteration 341 / 1900) loss: 1.402783
(Iteration 351 / 1900) loss: 1.384869
(Iteration 361 / 1900) loss: 1.289204
(Iteration 371 / 1900) loss: 1.275307
(Epoch 4 / 20) train acc: 0.599000; val_acc: 0.524000
(Iteration 381 / 1900) loss: 1.214089
(Iteration 391 / 1900) loss: 1.362345
(Iteration 401 / 1900) loss: 1.335522
(Iteration 411 / 1900) loss: 1.400044
(Iteration 421 / 1900) loss: 1.255206
(Iteration 431 / 1900) loss: 1.289558
(Iteration 441 / 1900) loss: 1.252084
(Iteration 451 / 1900) loss: 1.226132
(Iteration 461 / 1900) loss: 1.240383
(Iteration 471 / 1900) loss: 1.247376
(Epoch 5 / 20) train acc: 0.590000; val_acc: 0.529000
(Iteration 481 / 1900) loss: 1.236137
```

```
(Iteration 491 / 1900) loss: 1.288181
(Iteration 501 / 1900) loss: 1.225854
(Iteration 511 / 1900) loss: 1.303260
(Iteration 521 / 1900) loss: 1.201134
(Iteration 531 / 1900) loss: 1.166401
(Iteration 541 / 1900) loss: 1.159550
(Iteration 551 / 1900) loss: 1.258156
(Iteration 561 / 1900) loss: 1.166865
(Epoch 6 / 20) train acc: 0.622000; val_acc: 0.532000
(Iteration 571 / 1900) loss: 1.252805
(Iteration 581 / 1900) loss: 1.252159
(Iteration 591 / 1900) loss: 1.194015
(Iteration 601 / 1900) loss: 1.203488
(Iteration 611 / 1900) loss: 1.183981
(Iteration 621 / 1900) loss: 1.188210
(Iteration 631 / 1900) loss: 1.219135
(Iteration 641 / 1900) loss: 1.190891
(Iteration 651 / 1900) loss: 1.218577
(Iteration 661 / 1900) loss: 1.160655
(Epoch 7 / 20) train acc: 0.635000; val_acc: 0.534000
(Iteration 671 / 1900) loss: 1.108675
(Iteration 681 / 1900) loss: 1.216121
(Iteration 691 / 1900) loss: 1.231043
(Iteration 701 / 1900) loss: 1.163099
(Iteration 711 / 1900) loss: 1.149161
(Iteration 721 / 1900) loss: 1.161272
(Iteration 731 / 1900) loss: 1.191199
(Iteration 741 / 1900) loss: 1.114346
(Iteration 751 / 1900) loss: 1.249426
(Epoch 8 / 20) train acc: 0.637000; val_acc: 0.562000
(Iteration 761 / 1900) loss: 1.108009
(Iteration 771 / 1900) loss: 1.129483
(Iteration 781 / 1900) loss: 1.106924
(Iteration 791 / 1900) loss: 1.171105
(Iteration 801 / 1900) loss: 1.133747
(Iteration 811 / 1900) loss: 1.167833
(Iteration 821 / 1900) loss: 1.181293
(Iteration 831 / 1900) loss: 1.229280
(Iteration 841 / 1900) loss: 1.182928
(Iteration 851 / 1900) loss: 1.160435
(Epoch 9 / 20) train acc: 0.661000; val_acc: 0.535000
(Iteration 861 / 1900) loss: 1.057785
(Iteration 871 / 1900) loss: 1.172147
(Iteration 881 / 1900) loss: 1.065744
(Iteration 891 / 1900) loss: 1.081279
(Iteration 901 / 1900) loss: 1.051673
(Iteration 911 / 1900) loss: 1.108172
(Iteration 921 / 1900) loss: 1.172235
```

```
(Iteration 931 / 1900) loss: 1.058636
(Iteration 941 / 1900) loss: 1.109729
(Epoch 10 / 20) train acc: 0.687000; val_acc: 0.550000
(Iteration 951 / 1900) loss: 1.057863
(Iteration 961 / 1900) loss: 1.092477
(Iteration 971 / 1900) loss: 1.153819
(Iteration 981 / 1900) loss: 1.071250
(Iteration 991 / 1900) loss: 1.077980
(Iteration 1001 / 1900) loss: 1.116464
(Iteration 1011 / 1900) loss: 1.085814
(Iteration 1021 / 1900) loss: 1.130790
(Iteration 1031 / 1900) loss: 1.075755
(Iteration 1041 / 1900) loss: 1.025995
(Epoch 11 / 20) train acc: 0.698000; val_acc: 0.567000
(Iteration 1051 / 1900) loss: 0.998255
(Iteration 1061 / 1900) loss: 0.999676
(Iteration 1071 / 1900) loss: 1.122715
(Iteration 1081 / 1900) loss: 1.059621
(Iteration 1091 / 1900) loss: 1.051337
(Iteration 1101 / 1900) loss: 1.060050
(Iteration 1111 / 1900) loss: 1.113312
(Iteration 1121 / 1900) loss: 1.032016
(Iteration 1131 / 1900) loss: 1.102936
(Epoch 12 / 20) train acc: 0.734000; val_acc: 0.562000
(Iteration 1141 / 1900) loss: 1.001613
(Iteration 1151 / 1900) loss: 1.115452
(Iteration 1161 / 1900) loss: 1.030879
(Iteration 1171 / 1900) loss: 1.007443
(Iteration 1181 / 1900) loss: 1.235386
(Iteration 1191 / 1900) loss: 0.999896
(Iteration 1201 / 1900) loss: 1.125823
(Iteration 1211 / 1900) loss: 1.070859
(Iteration 1221 / 1900) loss: 1.020959
(Iteration 1231 / 1900) loss: 1.059905
(Epoch 13 / 20) train acc: 0.681000; val_acc: 0.546000
(Iteration 1241 / 1900) loss: 1.027828
(Iteration 1251 / 1900) loss: 0.996160
(Iteration 1261 / 1900) loss: 1.025923
(Iteration 1271 / 1900) loss: 0.999128
(Iteration 1281 / 1900) loss: 0.978991
(Iteration 1291 / 1900) loss: 1.020598
(Iteration 1301 / 1900) loss: 0.993475
(Iteration 1311 / 1900) loss: 1.011406
(Iteration 1321 / 1900) loss: 0.983151
(Epoch 14 / 20) train acc: 0.702000; val_acc: 0.548000
(Iteration 1331 / 1900) loss: 0.990533
(Iteration 1341 / 1900) loss: 1.076437
(Iteration 1351 / 1900) loss: 0.961251
```

```
(Iteration 1361 / 1900) loss: 1.015115
(Iteration 1371 / 1900) loss: 0.973008
(Iteration 1381 / 1900) loss: 0.952081
(Iteration 1391 / 1900) loss: 1.016516
(Iteration 1401 / 1900) loss: 0.952048
(Iteration 1411 / 1900) loss: 1.027603
(Iteration 1421 / 1900) loss: 1.031521
(Epoch 15 / 20) train acc: 0.720000; val_acc: 0.557000
(Iteration 1431 / 1900) loss: 1.018373
(Iteration 1441 / 1900) loss: 1.030242
(Iteration 1451 / 1900) loss: 0.966311
(Iteration 1461 / 1900) loss: 0.989092
(Iteration 1471 / 1900) loss: 0.999386
(Iteration 1481 / 1900) loss: 1.063603
(Iteration 1491 / 1900) loss: 1.001677
(Iteration 1501 / 1900) loss: 0.894754
(Iteration 1511 / 1900) loss: 1.042933
(Epoch 16 / 20) train acc: 0.725000; val_acc: 0.548000
(Iteration 1521 / 1900) loss: 1.042550
(Iteration 1531 / 1900) loss: 0.957446
(Iteration 1541 / 1900) loss: 0.971181
(Iteration 1551 / 1900) loss: 0.988124
(Iteration 1561 / 1900) loss: 0.960835
(Iteration 1571 / 1900) loss: 0.933674
(Iteration 1581 / 1900) loss: 0.925818
(Iteration 1591 / 1900) loss: 0.968273
(Iteration 1601 / 1900) loss: 0.929999
(Iteration 1611 / 1900) loss: 0.967318
(Epoch 17 / 20) train acc: 0.749000; val_acc: 0.573000
(Iteration 1621 / 1900) loss: 0.919658
(Iteration 1631 / 1900) loss: 0.943707
(Iteration 1641 / 1900) loss: 0.940815
(Iteration 1651 / 1900) loss: 0.910518
(Iteration 1661 / 1900) loss: 0.932842
(Iteration 1671 / 1900) loss: 0.983768
(Iteration 1681 / 1900) loss: 1.011697
(Iteration 1691 / 1900) loss: 0.890348
(Iteration 1701 / 1900) loss: 0.944036
(Epoch 18 / 20) train acc: 0.745000; val_acc: 0.563000
(Iteration 1711 / 1900) loss: 0.973803
(Iteration 1721 / 1900) loss: 1.032020
(Iteration 1731 / 1900) loss: 0.964657
(Iteration 1741 / 1900) loss: 0.898551
(Iteration 1751 / 1900) loss: 0.973716
(Iteration 1761 / 1900) loss: 0.966976
(Iteration 1771 / 1900) loss: 0.886851
(Iteration 1781 / 1900) loss: 0.927954
(Iteration 1791 / 1900) loss: 0.908274
```

```
(Iteration 1801 / 1900) loss: 0.879817
(Epoch 19 / 20) train acc: 0.749000; val_acc: 0.569000
(Iteration 1811 / 1900) loss: 0.863932
(Iteration 1821 / 1900) loss: 0.915084
(Iteration 1831 / 1900) loss: 0.925110
(Iteration 1841 / 1900) loss: 0.941953
(Iteration 1851 / 1900) loss: 0.912489
(Iteration 1861 / 1900) loss: 0.943184
(Iteration 1871 / 1900) loss: 0.881262
(Iteration 1881 / 1900) loss: 0.897962
(Iteration 1891 / 1900) loss: 0.979136
(Epoch 20 / 20) train acc: 0.771000; val_acc: 0.550000
(Iteration 1 / 1900) loss: 2.305455
(Epoch 0 / 20) train acc: 0.172000; val_acc: 0.175000
(Iteration 11 / 1900) loss: 2.105816
(Iteration 21 / 1900) loss: 1.919686
(Iteration 31 / 1900) loss: 1.835128
(Iteration 41 / 1900) loss: 1.772568
(Iteration 51 / 1900) loss: 1.697457
(Iteration 61 / 1900) loss: 1.624738
(Iteration 71 / 1900) loss: 1.601003
(Iteration 81 / 1900) loss: 1.607582
(Iteration 91 / 1900) loss: 1.606423
(Epoch 1 / 20) train acc: 0.490000; val_acc: 0.477000
(Iteration 101 / 1900) loss: 1.489089
(Iteration 111 / 1900) loss: 1.438938
(Iteration 121 / 1900) loss: 1.394820
(Iteration 131 / 1900) loss: 1.457481
(Iteration 141 / 1900) loss: 1.411860
(Iteration 151 / 1900) loss: 1.417622
(Iteration 161 / 1900) loss: 1.325885
(Iteration 171 / 1900) loss: 1.358672
(Iteration 181 / 1900) loss: 1.340330
(Epoch 2 / 20) train acc: 0.539000; val_acc: 0.486000
(Iteration 191 / 1900) loss: 1.419612
(Iteration 201 / 1900) loss: 1.340938
(Iteration 211 / 1900) loss: 1.294490
(Iteration 221 / 1900) loss: 1.278183
(Iteration 231 / 1900) loss: 1.399873
(Iteration 241 / 1900) loss: 1.274180
(Iteration 251 / 1900) loss: 1.266405
(Iteration 261 / 1900) loss: 1.217744
(Iteration 271 / 1900) loss: 1.307092
(Iteration 281 / 1900) loss: 1.236610
(Epoch 3 / 20) train acc: 0.569000; val_acc: 0.541000
(Iteration 291 / 1900) loss: 1.231697
(Iteration 301 / 1900) loss: 1.228316
(Iteration 311 / 1900) loss: 1.185300
```

```
(Iteration 321 / 1900) loss: 1.226202
(Iteration 331 / 1900) loss: 1.175643
(Iteration 341 / 1900) loss: 1.172711
(Iteration 351 / 1900) loss: 1.183765
(Iteration 361 / 1900) loss: 1.161704
(Iteration 371 / 1900) loss: 1.157547
(Epoch 4 / 20) train acc: 0.579000; val_acc: 0.538000
(Iteration 381 / 1900) loss: 1.119719
(Iteration 391 / 1900) loss: 1.229962
(Iteration 401 / 1900) loss: 1.159306
(Iteration 411 / 1900) loss: 1.219422
(Iteration 421 / 1900) loss: 1.107655
(Iteration 431 / 1900) loss: 1.115620
(Iteration 441 / 1900) loss: 1.123131
(Iteration 451 / 1900) loss: 1.148539
(Iteration 461 / 1900) loss: 1.078145
(Iteration 471 / 1900) loss: 1.096561
(Epoch 5 / 20) train acc: 0.630000; val_acc: 0.539000
(Iteration 481 / 1900) loss: 1.146251
(Iteration 491 / 1900) loss: 1.139023
(Iteration 501 / 1900) loss: 1.193588
(Iteration 511 / 1900) loss: 1.100243
(Iteration 521 / 1900) loss: 1.152038
(Iteration 531 / 1900) loss: 1.047316
(Iteration 541 / 1900) loss: 1.089611
(Iteration 551 / 1900) loss: 1.164260
(Iteration 561 / 1900) loss: 1.024326
(Epoch 6 / 20) train acc: 0.641000; val_acc: 0.558000
(Iteration 571 / 1900) loss: 1.034967
(Iteration 581 / 1900) loss: 1.033546
(Iteration 591 / 1900) loss: 0.956015
(Iteration 601 / 1900) loss: 1.027366
(Iteration 611 / 1900) loss: 1.087153
(Iteration 621 / 1900) loss: 1.074229
(Iteration 631 / 1900) loss: 1.123313
(Iteration 641 / 1900) loss: 1.094240
(Iteration 651 / 1900) loss: 0.914442
(Iteration 661 / 1900) loss: 1.018126
(Epoch 7 / 20) train acc: 0.659000; val_acc: 0.565000
(Iteration 671 / 1900) loss: 0.994941
(Iteration 681 / 1900) loss: 1.021642
(Iteration 691 / 1900) loss: 0.999588
(Iteration 701 / 1900) loss: 0.966747
(Iteration 711 / 1900) loss: 1.038395
(Iteration 721 / 1900) loss: 0.982281
(Iteration 731 / 1900) loss: 0.974284
(Iteration 741 / 1900) loss: 0.944524
(Iteration 751 / 1900) loss: 0.939697
```

```
(Epoch 8 / 20) train acc: 0.693000; val_acc: 0.541000
(Iteration 761 / 1900) loss: 0.953945
(Iteration 771 / 1900) loss: 0.925437
(Iteration 781 / 1900) loss: 0.991393
(Iteration 791 / 1900) loss: 0.915751
(Iteration 801 / 1900) loss: 0.880374
(Iteration 811 / 1900) loss: 0.951572
(Iteration 821 / 1900) loss: 0.901163
(Iteration 831 / 1900) loss: 0.871545
(Iteration 841 / 1900) loss: 0.956587
(Iteration 851 / 1900) loss: 0.888810
(Epoch 9 / 20) train acc: 0.701000; val_acc: 0.573000
(Iteration 861 / 1900) loss: 0.968608
(Iteration 871 / 1900) loss: 0.906042
(Iteration 881 / 1900) loss: 0.994541
(Iteration 891 / 1900) loss: 0.881918
(Iteration 901 / 1900) loss: 0.892272
(Iteration 911 / 1900) loss: 0.906294
(Iteration 921 / 1900) loss: 0.886066
(Iteration 931 / 1900) loss: 0.904012
(Iteration 941 / 1900) loss: 0.857378
(Epoch 10 / 20) train acc: 0.744000; val_acc: 0.539000
(Iteration 951 / 1900) loss: 0.947279
(Iteration 961 / 1900) loss: 0.939349
(Iteration 971 / 1900) loss: 0.851445
(Iteration 981 / 1900) loss: 0.764188
(Iteration 991 / 1900) loss: 0.909019
(Iteration 1001 / 1900) loss: 0.860953
(Iteration 1011 / 1900) loss: 0.923893
(Iteration 1021 / 1900) loss: 0.841223
(Iteration 1031 / 1900) loss: 0.767602
(Iteration 1041 / 1900) loss: 0.848377
(Epoch 11 / 20) train acc: 0.739000; val_acc: 0.545000
(Iteration 1051 / 1900) loss: 0.807330
(Iteration 1061 / 1900) loss: 0.815570
(Iteration 1071 / 1900) loss: 0.832323
(Iteration 1081 / 1900) loss: 0.910556
(Iteration 1091 / 1900) loss: 0.776775
(Iteration 1101 / 1900) loss: 0.764774
(Iteration 1111 / 1900) loss: 0.752901
(Iteration 1121 / 1900) loss: 0.736916
(Iteration 1131 / 1900) loss: 0.735725
(Epoch 12 / 20) train acc: 0.730000; val_acc: 0.550000
(Iteration 1141 / 1900) loss: 0.767975
(Iteration 1151 / 1900) loss: 0.762449
(Iteration 1161 / 1900) loss: 0.799390
(Iteration 1171 / 1900) loss: 0.761292
(Iteration 1181 / 1900) loss: 0.699061
```

```
(Iteration 1191 / 1900) loss: 0.740978
(Iteration 1201 / 1900) loss: 0.760418
(Iteration 1211 / 1900) loss: 0.681987
(Iteration 1221 / 1900) loss: 0.725307
(Iteration 1231 / 1900) loss: 0.736645
(Epoch 13 / 20) train acc: 0.783000; val_acc: 0.556000
(Iteration 1241 / 1900) loss: 0.792097
(Iteration 1251 / 1900) loss: 0.677657
(Iteration 1261 / 1900) loss: 0.696281
(Iteration 1271 / 1900) loss: 0.725416
(Iteration 1281 / 1900) loss: 0.724483
(Iteration 1291 / 1900) loss: 0.749805
(Iteration 1301 / 1900) loss: 0.863281
(Iteration 1311 / 1900) loss: 0.721444
(Iteration 1321 / 1900) loss: 0.729753
(Epoch 14 / 20) train acc: 0.758000; val_acc: 0.555000
(Iteration 1331 / 1900) loss: 0.688732
(Iteration 1341 / 1900) loss: 0.636563
(Iteration 1351 / 1900) loss: 0.673491
(Iteration 1361 / 1900) loss: 0.682462
(Iteration 1371 / 1900) loss: 0.740041
(Iteration 1381 / 1900) loss: 0.691763
(Iteration 1391 / 1900) loss: 0.644656
(Iteration 1401 / 1900) loss: 0.680468
(Iteration 1411 / 1900) loss: 0.710947
(Iteration 1421 / 1900) loss: 0.779235
(Epoch 15 / 20) train acc: 0.794000; val_acc: 0.561000
(Iteration 1431 / 1900) loss: 0.685874
(Iteration 1441 / 1900) loss: 0.727707
(Iteration 1451 / 1900) loss: 0.641264
(Iteration 1461 / 1900) loss: 0.577309
(Iteration 1471 / 1900) loss: 0.581218
(Iteration 1481 / 1900) loss: 0.635744
(Iteration 1491 / 1900) loss: 0.751684
(Iteration 1501 / 1900) loss: 0.614137
(Iteration 1511 / 1900) loss: 0.632064
(Epoch 16 / 20) train acc: 0.817000; val_acc: 0.557000
(Iteration 1521 / 1900) loss: 0.656927
(Iteration 1531 / 1900) loss: 0.617653
(Iteration 1541 / 1900) loss: 0.607230
(Iteration 1551 / 1900) loss: 0.647745
(Iteration 1561 / 1900) loss: 0.594894
(Iteration 1571 / 1900) loss: 0.613721
(Iteration 1581 / 1900) loss: 0.608578
(Iteration 1591 / 1900) loss: 0.666748
(Iteration 1601 / 1900) loss: 0.616056
(Iteration 1611 / 1900) loss: 0.584703
(Epoch 17 / 20) train acc: 0.824000; val_acc: 0.560000
```

```
(Iteration 1621 / 1900) loss: 0.514010
(Iteration 1631 / 1900) loss: 0.576643
(Iteration 1641 / 1900) loss: 0.560082
(Iteration 1651 / 1900) loss: 0.574401
(Iteration 1661 / 1900) loss: 0.607032
(Iteration 1671 / 1900) loss: 0.597450
(Iteration 1681 / 1900) loss: 0.566730
(Iteration 1691 / 1900) loss: 0.656024
(Iteration 1701 / 1900) loss: 0.700821
(Epoch 18 / 20) train acc: 0.824000; val_acc: 0.539000
(Iteration 1711 / 1900) loss: 0.580803
(Iteration 1721 / 1900) loss: 0.551316
(Iteration 1731 / 1900) loss: 0.618107
(Iteration 1741 / 1900) loss: 0.622302
(Iteration 1751 / 1900) loss: 0.571499
(Iteration 1761 / 1900) loss: 0.528564
(Iteration 1771 / 1900) loss: 0.608328
(Iteration 1781 / 1900) loss: 0.533038
(Iteration 1791 / 1900) loss: 0.514620
(Iteration 1801 / 1900) loss: 0.537699
(Epoch 19 / 20) train acc: 0.837000; val_acc: 0.544000
(Iteration 1811 / 1900) loss: 0.585422
(Iteration 1821 / 1900) loss: 0.491696
(Iteration 1831 / 1900) loss: 0.611576
(Iteration 1841 / 1900) loss: 0.515927
(Iteration 1851 / 1900) loss: 0.514966
(Iteration 1861 / 1900) loss: 0.551652
(Iteration 1871 / 1900) loss: 0.523490
(Iteration 1881 / 1900) loss: 0.522864
(Iteration 1891 / 1900) loss: 0.557780
(Epoch 20 / 20) train acc: 0.851000; val_acc: 0.544000
(Iteration 1 / 1900) loss: 2.677361
(Epoch 0 / 20) train acc: 0.150000; val_acc: 0.191000
(Iteration 11 / 1900) loss: 2.427287
(Iteration 21 / 1900) loss: 2.254415
(Iteration 31 / 1900) loss: 2.173475
(Iteration 41 / 1900) loss: 2.085349
(Iteration 51 / 1900) loss: 2.052325
(Iteration 61 / 1900) loss: 2.029276
(Iteration 71 / 1900) loss: 1.968407
(Iteration 81 / 1900) loss: 1.950706
(Iteration 91 / 1900) loss: 1.910641
(Epoch 1 / 20) train acc: 0.422000; val_acc: 0.421000
(Iteration 101 / 1900) loss: 1.863699
(Iteration 111 / 1900) loss: 1.846198
(Iteration 121 / 1900) loss: 1.842395
(Iteration 131 / 1900) loss: 1.837377
(Iteration 141 / 1900) loss: 1.853194
```

```
(Iteration 151 / 1900) loss: 1.780809
(Iteration 161 / 1900) loss: 1.797131
(Iteration 171 / 1900) loss: 1.784334
(Iteration 181 / 1900) loss: 1.820807
(Epoch 2 / 20) train acc: 0.425000; val_acc: 0.426000
(Iteration 191 / 1900) loss: 1.767317
(Iteration 201 / 1900) loss: 1.718573
(Iteration 211 / 1900) loss: 1.669252
(Iteration 221 / 1900) loss: 1.694252
(Iteration 231 / 1900) loss: 1.662169
(Iteration 241 / 1900) loss: 1.628758
(Iteration 251 / 1900) loss: 1.685756
(Iteration 261 / 1900) loss: 1.636487
(Iteration 271 / 1900) loss: 1.664533
(Iteration 281 / 1900) loss: 1.644610
(Epoch 3 / 20) train acc: 0.457000; val_acc: 0.468000
(Iteration 291 / 1900) loss: 1.633290
(Iteration 301 / 1900) loss: 1.661189
(Iteration 311 / 1900) loss: 1.663901
(Iteration 321 / 1900) loss: 1.615911
(Iteration 331 / 1900) loss: 1.674965
(Iteration 341 / 1900) loss: 1.589949
(Iteration 351 / 1900) loss: 1.630615
(Iteration 361 / 1900) loss: 1.658397
(Iteration 371 / 1900) loss: 1.623894
(Epoch 4 / 20) train acc: 0.502000; val_acc: 0.512000
(Iteration 381 / 1900) loss: 1.579788
(Iteration 391 / 1900) loss: 1.553530
(Iteration 401 / 1900) loss: 1.598219
(Iteration 411 / 1900) loss: 1.605877
(Iteration 421 / 1900) loss: 1.520480
(Iteration 431 / 1900) loss: 1.570352
(Iteration 441 / 1900) loss: 1.567665
(Iteration 451 / 1900) loss: 1.554792
(Iteration 461 / 1900) loss: 1.492933
(Iteration 471 / 1900) loss: 1.570789
(Epoch 5 / 20) train acc: 0.500000; val_acc: 0.509000
(Iteration 481 / 1900) loss: 1.547271
(Iteration 491 / 1900) loss: 1.577489
(Iteration 501 / 1900) loss: 1.568355
(Iteration 511 / 1900) loss: 1.531391
(Iteration 521 / 1900) loss: 1.579889
(Iteration 531 / 1900) loss: 1.555991
(Iteration 541 / 1900) loss: 1.607404
(Iteration 551 / 1900) loss: 1.564916
(Iteration 561 / 1900) loss: 1.609188
(Epoch 6 / 20) train acc: 0.500000; val_acc: 0.471000
(Iteration 571 / 1900) loss: 1.575870
```

```
(Iteration 581 / 1900) loss: 1.499461
(Iteration 591 / 1900) loss: 1.552079
(Iteration 601 / 1900) loss: 1.565351
(Iteration 611 / 1900) loss: 1.542665
(Iteration 621 / 1900) loss: 1.469337
(Iteration 631 / 1900) loss: 1.501116
(Iteration 641 / 1900) loss: 1.473314
(Iteration 651 / 1900) loss: 1.492704
(Iteration 661 / 1900) loss: 1.485424
(Epoch 7 / 20) train acc: 0.527000; val_acc: 0.505000
(Iteration 671 / 1900) loss: 1.566906
(Iteration 681 / 1900) loss: 1.473813
(Iteration 691 / 1900) loss: 1.504839
(Iteration 701 / 1900) loss: 1.487284
(Iteration 711 / 1900) loss: 1.532745
(Iteration 721 / 1900) loss: 1.435215
(Iteration 731 / 1900) loss: 1.532650
(Iteration 741 / 1900) loss: 1.499644
(Iteration 751 / 1900) loss: 1.466401
(Epoch 8 / 20) train acc: 0.534000; val_acc: 0.518000
(Iteration 761 / 1900) loss: 1.556406
(Iteration 771 / 1900) loss: 1.558342
(Iteration 781 / 1900) loss: 1.497910
(Iteration 791 / 1900) loss: 1.527910
(Iteration 801 / 1900) loss: 1.530706
(Iteration 811 / 1900) loss: 1.558494
(Iteration 821 / 1900) loss: 1.528766
(Iteration 831 / 1900) loss: 1.494554
(Iteration 841 / 1900) loss: 1.471848
(Iteration 851 / 1900) loss: 1.443749
(Epoch 9 / 20) train acc: 0.558000; val_acc: 0.519000
(Iteration 861 / 1900) loss: 1.504970
(Iteration 871 / 1900) loss: 1.514629
(Iteration 881 / 1900) loss: 1.467888
(Iteration 891 / 1900) loss: 1.438421
(Iteration 901 / 1900) loss: 1.504609
(Iteration 911 / 1900) loss: 1.508677
(Iteration 921 / 1900) loss: 1.541232
(Iteration 931 / 1900) loss: 1.491433
(Iteration 941 / 1900) loss: 1.430912
(Epoch 10 / 20) train acc: 0.591000; val_acc: 0.521000
(Iteration 951 / 1900) loss: 1.491440
(Iteration 961 / 1900) loss: 1.474554
(Iteration 971 / 1900) loss: 1.491630
(Iteration 981 / 1900) loss: 1.458028
(Iteration 991 / 1900) loss: 1.503448
(Iteration 1001 / 1900) loss: 1.487702
(Iteration 1011 / 1900) loss: 1.566493
```

```
(Iteration 1021 / 1900) loss: 1.458232
(Iteration 1031 / 1900) loss: 1.454626
(Iteration 1041 / 1900) loss: 1.418745
(Epoch 11 / 20) train acc: 0.605000; val_acc: 0.540000
(Iteration 1051 / 1900) loss: 1.448347
(Iteration 1061 / 1900) loss: 1.428265
(Iteration 1071 / 1900) loss: 1.583726
(Iteration 1081 / 1900) loss: 1.441207
(Iteration 1091 / 1900) loss: 1.430194
(Iteration 1101 / 1900) loss: 1.434025
(Iteration 1111 / 1900) loss: 1.499566
(Iteration 1121 / 1900) loss: 1.421462
(Iteration 1131 / 1900) loss: 1.448447
(Epoch 12 / 20) train acc: 0.555000; val_acc: 0.506000
(Iteration 1141 / 1900) loss: 1.425626
(Iteration 1151 / 1900) loss: 1.400334
(Iteration 1161 / 1900) loss: 1.437751
(Iteration 1171 / 1900) loss: 1.454456
(Iteration 1181 / 1900) loss: 1.429677
(Iteration 1191 / 1900) loss: 1.416149
(Iteration 1201 / 1900) loss: 1.429445
(Iteration 1211 / 1900) loss: 1.445959
(Iteration 1221 / 1900) loss: 1.403389
(Iteration 1231 / 1900) loss: 1.454029
(Epoch 13 / 20) train acc: 0.572000; val_acc: 0.517000
(Iteration 1241 / 1900) loss: 1.504687
(Iteration 1251 / 1900) loss: 1.402261
(Iteration 1261 / 1900) loss: 1.372927
(Iteration 1271 / 1900) loss: 1.403877
(Iteration 1281 / 1900) loss: 1.402877
(Iteration 1291 / 1900) loss: 1.436918
(Iteration 1301 / 1900) loss: 1.480595
(Iteration 1311 / 1900) loss: 1.395126
(Iteration 1321 / 1900) loss: 1.547811
(Epoch 14 / 20) train acc: 0.604000; val_acc: 0.524000
(Iteration 1331 / 1900) loss: 1.445391
(Iteration 1341 / 1900) loss: 1.383547
(Iteration 1351 / 1900) loss: 1.392927
(Iteration 1361 / 1900) loss: 1.426307
(Iteration 1371 / 1900) loss: 1.350830
(Iteration 1381 / 1900) loss: 1.460446
(Iteration 1391 / 1900) loss: 1.398431
(Iteration 1401 / 1900) loss: 1.343622
(Iteration 1411 / 1900) loss: 1.394421
(Iteration 1421 / 1900) loss: 1.484057
(Epoch 15 / 20) train acc: 0.619000; val_acc: 0.513000
(Iteration 1431 / 1900) loss: 1.427500
(Iteration 1441 / 1900) loss: 1.380737
```

```
(Iteration 1451 / 1900) loss: 1.450722
(Iteration 1461 / 1900) loss: 1.372172
(Iteration 1471 / 1900) loss: 1.393850
(Iteration 1481 / 1900) loss: 1.409734
(Iteration 1491 / 1900) loss: 1.440352
(Iteration 1501 / 1900) loss: 1.375156
(Iteration 1511 / 1900) loss: 1.374258
(Epoch 16 / 20) train acc: 0.625000; val_acc: 0.547000
(Iteration 1521 / 1900) loss: 1.396936
(Iteration 1531 / 1900) loss: 1.396502
(Iteration 1541 / 1900) loss: 1.350169
(Iteration 1551 / 1900) loss: 1.460207
(Iteration 1561 / 1900) loss: 1.411364
(Iteration 1571 / 1900) loss: 1.375665
(Iteration 1581 / 1900) loss: 1.435971
(Iteration 1591 / 1900) loss: 1.432778
(Iteration 1601 / 1900) loss: 1.430639
(Iteration 1611 / 1900) loss: 1.350593
(Epoch 17 / 20) train acc: 0.609000; val_acc: 0.528000
(Iteration 1621 / 1900) loss: 1.365846
(Iteration 1631 / 1900) loss: 1.366520
(Iteration 1641 / 1900) loss: 1.386818
(Iteration 1651 / 1900) loss: 1.394309
(Iteration 1661 / 1900) loss: 1.350332
(Iteration 1671 / 1900) loss: 1.421707
(Iteration 1681 / 1900) loss: 1.388742
(Iteration 1691 / 1900) loss: 1.383993
(Iteration 1701 / 1900) loss: 1.397899
(Epoch 18 / 20) train acc: 0.629000; val_acc: 0.549000
(Iteration 1711 / 1900) loss: 1.356653
(Iteration 1721 / 1900) loss: 1.420472
(Iteration 1731 / 1900) loss: 1.434451
(Iteration 1741 / 1900) loss: 1.351950
(Iteration 1751 / 1900) loss: 1.390838
(Iteration 1761 / 1900) loss: 1.444750
(Iteration 1771 / 1900) loss: 1.378555
(Iteration 1781 / 1900) loss: 1.467362
(Iteration 1791 / 1900) loss: 1.440482
(Iteration 1801 / 1900) loss: 1.378571
(Epoch 19 / 20) train acc: 0.609000; val_acc: 0.569000
(Iteration 1811 / 1900) loss: 1.462423
(Iteration 1821 / 1900) loss: 1.338158
(Iteration 1831 / 1900) loss: 1.373857
(Iteration 1841 / 1900) loss: 1.344681
(Iteration 1851 / 1900) loss: 1.386747
(Iteration 1861 / 1900) loss: 1.374294
(Iteration 1871 / 1900) loss: 1.380539
(Iteration 1881 / 1900) loss: 1.378022
```

```
(Iteration 1891 / 1900) loss: 1.434949
(Epoch 20 / 20) train acc: 0.619000; val_acc: 0.564000
(Iteration 1 / 1900) loss: 2.378210
(Epoch 0 / 20) train acc: 0.187000; val_acc: 0.205000
(Iteration 11 / 1900) loss: 2.293651
(Iteration 21 / 1900) loss: 2.217826
(Iteration 31 / 1900) loss: 2.155662
(Iteration 41 / 1900) loss: 2.080833
(Iteration 51 / 1900) loss: 2.038501
(Iteration 61 / 1900) loss: 1.994932
(Iteration 71 / 1900) loss: 1.945176
(Iteration 81 / 1900) loss: 1.900292
(Iteration 91 / 1900) loss: 1.869966
(Epoch 1 / 20) train acc: 0.424000; val_acc: 0.409000
(Iteration 101 / 1900) loss: 1.809186
(Iteration 111 / 1900) loss: 1.826716
(Iteration 121 / 1900) loss: 1.764572
(Iteration 131 / 1900) loss: 1.767480
(Iteration 141 / 1900) loss: 1.715477
(Iteration 151 / 1900) loss: 1.686407
(Iteration 161 / 1900) loss: 1.646858
(Iteration 171 / 1900) loss: 1.655392
(Iteration 181 / 1900) loss: 1.625065
(Epoch 2 / 20) train acc: 0.491000; val_acc: 0.477000
(Iteration 191 / 1900) loss: 1.613628
(Iteration 201 / 1900) loss: 1.577856
(Iteration 211 / 1900) loss: 1.602751
(Iteration 221 / 1900) loss: 1.627468
(Iteration 231 / 1900) loss: 1.545607
(Iteration 241 / 1900) loss: 1.582078
(Iteration 251 / 1900) loss: 1.537229
(Iteration 261 / 1900) loss: 1.518373
(Iteration 271 / 1900) loss: 1.521401
(Iteration 281 / 1900) loss: 1.509745
(Epoch 3 / 20) train acc: 0.524000; val_acc: 0.489000
(Iteration 291 / 1900) loss: 1.503232
(Iteration 301 / 1900) loss: 1.428579
(Iteration 311 / 1900) loss: 1.516814
(Iteration 321 / 1900) loss: 1.509297
(Iteration 331 / 1900) loss: 1.410759
(Iteration 341 / 1900) loss: 1.461431
(Iteration 351 / 1900) loss: 1.378981
(Iteration 361 / 1900) loss: 1.497581
(Iteration 371 / 1900) loss: 1.403278
(Epoch 4 / 20) train acc: 0.553000; val_acc: 0.498000
(Iteration 381 / 1900) loss: 1.450500
(Iteration 391 / 1900) loss: 1.438853
(Iteration 401 / 1900) loss: 1.446537
```

```
(Iteration 411 / 1900) loss: 1.415058
(Iteration 421 / 1900) loss: 1.335298
(Iteration 431 / 1900) loss: 1.375337
(Iteration 441 / 1900) loss: 1.372587
(Iteration 451 / 1900) loss: 1.464376
(Iteration 461 / 1900) loss: 1.363586
(Iteration 471 / 1900) loss: 1.289436
(Epoch 5 / 20) train acc: 0.571000; val_acc: 0.543000
(Iteration 481 / 1900) loss: 1.373595
(Iteration 491 / 1900) loss: 1.288747
(Iteration 501 / 1900) loss: 1.287186
(Iteration 511 / 1900) loss: 1.280824
(Iteration 521 / 1900) loss: 1.288081
(Iteration 531 / 1900) loss: 1.323841
(Iteration 541 / 1900) loss: 1.360059
(Iteration 551 / 1900) loss: 1.381985
(Iteration 561 / 1900) loss: 1.356123
(Epoch 6 / 20) train acc: 0.582000; val_acc: 0.538000
(Iteration 571 / 1900) loss: 1.257999
(Iteration 581 / 1900) loss: 1.298095
(Iteration 591 / 1900) loss: 1.321750
(Iteration 601 / 1900) loss: 1.345491
(Iteration 611 / 1900) loss: 1.295138
(Iteration 621 / 1900) loss: 1.254033
(Iteration 631 / 1900) loss: 1.246075
(Iteration 641 / 1900) loss: 1.365624
(Iteration 651 / 1900) loss: 1.297065
(Iteration 661 / 1900) loss: 1.287876
(Epoch 7 / 20) train acc: 0.599000; val_acc: 0.539000
(Iteration 671 / 1900) loss: 1.267264
(Iteration 681 / 1900) loss: 1.284248
(Iteration 691 / 1900) loss: 1.223147
(Iteration 701 / 1900) loss: 1.223724
(Iteration 711 / 1900) loss: 1.241120
(Iteration 721 / 1900) loss: 1.223723
(Iteration 731 / 1900) loss: 1.219905
(Iteration 741 / 1900) loss: 1.192822
(Iteration 751 / 1900) loss: 1.254084
(Epoch 8 / 20) train acc: 0.620000; val_acc: 0.542000
(Iteration 761 / 1900) loss: 1.248330
(Iteration 771 / 1900) loss: 1.234838
(Iteration 781 / 1900) loss: 1.239934
(Iteration 791 / 1900) loss: 1.183115
(Iteration 801 / 1900) loss: 1.200401
(Iteration 811 / 1900) loss: 1.269268
(Iteration 821 / 1900) loss: 1.153659
(Iteration 831 / 1900) loss: 1.263915
(Iteration 841 / 1900) loss: 1.132440
```

```
(Iteration 851 / 1900) loss: 1.198611
(Epoch 9 / 20) train acc: 0.653000; val_acc: 0.557000
(Iteration 861 / 1900) loss: 1.196311
(Iteration 871 / 1900) loss: 1.234750
(Iteration 881 / 1900) loss: 1.202181
(Iteration 891 / 1900) loss: 1.156526
(Iteration 901 / 1900) loss: 1.246416
(Iteration 911 / 1900) loss: 1.263534
(Iteration 921 / 1900) loss: 1.269179
(Iteration 931 / 1900) loss: 1.189705
(Iteration 941 / 1900) loss: 1.198201
(Epoch 10 / 20) train acc: 0.648000; val_acc: 0.541000
(Iteration 951 / 1900) loss: 1.124494
(Iteration 961 / 1900) loss: 1.114008
(Iteration 971 / 1900) loss: 1.165950
(Iteration 981 / 1900) loss: 1.144499
(Iteration 991 / 1900) loss: 1.078968
(Iteration 1001 / 1900) loss: 1.191089
(Iteration 1011 / 1900) loss: 1.104133
(Iteration 1021 / 1900) loss: 1.171223
(Iteration 1031 / 1900) loss: 1.121804
(Iteration 1041 / 1900) loss: 1.199963
(Epoch 11 / 20) train acc: 0.622000; val_acc: 0.548000
(Iteration 1051 / 1900) loss: 1.188524
(Iteration 1061 / 1900) loss: 1.126506
(Iteration 1071 / 1900) loss: 1.059250
(Iteration 1081 / 1900) loss: 1.058651
(Iteration 1091 / 1900) loss: 1.121797
(Iteration 1101 / 1900) loss: 1.114190
(Iteration 1111 / 1900) loss: 1.151554
(Iteration 1121 / 1900) loss: 1.120244
(Iteration 1131 / 1900) loss: 1.213306
(Epoch 12 / 20) train acc: 0.640000; val_acc: 0.540000
(Iteration 1141 / 1900) loss: 1.150199
(Iteration 1151 / 1900) loss: 1.180298
(Iteration 1161 / 1900) loss: 1.165478
(Iteration 1171 / 1900) loss: 1.121483
(Iteration 1181 / 1900) loss: 1.114715
(Iteration 1191 / 1900) loss: 1.168058
(Iteration 1201 / 1900) loss: 1.040847
(Iteration 1211 / 1900) loss: 1.120137
(Iteration 1221 / 1900) loss: 1.154250
(Iteration 1231 / 1900) loss: 1.142371
(Epoch 13 / 20) train acc: 0.684000; val_acc: 0.556000
(Iteration 1241 / 1900) loss: 1.150918
(Iteration 1251 / 1900) loss: 1.037557
(Iteration 1261 / 1900) loss: 1.093035
(Iteration 1271 / 1900) loss: 1.078876
```

```
(Iteration 1281 / 1900) loss: 1.115003
(Iteration 1291 / 1900) loss: 1.087131
(Iteration 1301 / 1900) loss: 1.059307
(Iteration 1311 / 1900) loss: 1.124333
(Iteration 1321 / 1900) loss: 1.153849
(Epoch 14 / 20) train acc: 0.682000; val_acc: 0.555000
(Iteration 1331 / 1900) loss: 1.104744
(Iteration 1341 / 1900) loss: 1.136347
(Iteration 1351 / 1900) loss: 0.986924
(Iteration 1361 / 1900) loss: 1.084225
(Iteration 1371 / 1900) loss: 1.095503
(Iteration 1381 / 1900) loss: 1.019908
(Iteration 1391 / 1900) loss: 1.020660
(Iteration 1401 / 1900) loss: 0.993928
(Iteration 1411 / 1900) loss: 1.093101
(Iteration 1421 / 1900) loss: 1.077160
(Epoch 15 / 20) train acc: 0.708000; val_acc: 0.565000
(Iteration 1431 / 1900) loss: 1.114312
(Iteration 1441 / 1900) loss: 1.063671
(Iteration 1451 / 1900) loss: 1.051157
(Iteration 1461 / 1900) loss: 1.080199
(Iteration 1471 / 1900) loss: 1.098154
(Iteration 1481 / 1900) loss: 1.110601
(Iteration 1491 / 1900) loss: 1.000255
(Iteration 1501 / 1900) loss: 1.190078
(Iteration 1511 / 1900) loss: 1.035817
(Epoch 16 / 20) train acc: 0.719000; val_acc: 0.545000
(Iteration 1521 / 1900) loss: 1.097862
(Iteration 1531 / 1900) loss: 1.021171
(Iteration 1541 / 1900) loss: 1.111559
(Iteration 1551 / 1900) loss: 1.038930
(Iteration 1561 / 1900) loss: 1.093168
(Iteration 1571 / 1900) loss: 1.055279
(Iteration 1581 / 1900) loss: 1.049101
(Iteration 1591 / 1900) loss: 1.125084
(Iteration 1601 / 1900) loss: 1.047329
(Iteration 1611 / 1900) loss: 1.052982
(Epoch 17 / 20) train acc: 0.700000; val_acc: 0.563000
(Iteration 1621 / 1900) loss: 0.997640
(Iteration 1631 / 1900) loss: 0.964193
(Iteration 1641 / 1900) loss: 1.058826
(Iteration 1651 / 1900) loss: 0.951783
(Iteration 1661 / 1900) loss: 1.098896
(Iteration 1671 / 1900) loss: 0.949779
(Iteration 1681 / 1900) loss: 0.960127
(Iteration 1691 / 1900) loss: 0.996819
(Iteration 1701 / 1900) loss: 1.029653
(Epoch 18 / 20) train acc: 0.695000; val_acc: 0.535000
```

```
(Iteration 1711 / 1900) loss: 1.030517
(Iteration 1721 / 1900) loss: 1.008075
(Iteration 1731 / 1900) loss: 0.979449
(Iteration 1741 / 1900) loss: 1.009772
(Iteration 1751 / 1900) loss: 0.974038
(Iteration 1761 / 1900) loss: 1.031236
(Iteration 1771 / 1900) loss: 1.077101
(Iteration 1781 / 1900) loss: 1.032742
(Iteration 1791 / 1900) loss: 1.057800
(Iteration 1801 / 1900) loss: 0.961241
(Epoch 19 / 20) train acc: 0.740000; val_acc: 0.564000
(Iteration 1811 / 1900) loss: 1.075485
(Iteration 1821 / 1900) loss: 0.966336
(Iteration 1831 / 1900) loss: 0.991821
(Iteration 1841 / 1900) loss: 0.992368
(Iteration 1851 / 1900) loss: 0.981705
(Iteration 1861 / 1900) loss: 1.059450
(Iteration 1871 / 1900) loss: 0.963704
(Iteration 1881 / 1900) loss: 0.987543
(Iteration 1891 / 1900) loss: 0.914883
(Epoch 20 / 20) train acc: 0.743000; val_acc: 0.562000
(Iteration 1 / 1900) loss: 2.323799
(Epoch 0 / 20) train acc: 0.211000; val_acc: 0.216000
(Iteration 11 / 1900) loss: 2.254679
(Iteration 21 / 1900) loss: 2.184489
(Iteration 31 / 1900) loss: 2.130381
(Iteration 41 / 1900) loss: 2.066516
(Iteration 51 / 1900) loss: 2.012121
(Iteration 61 / 1900) loss: 1.974222
(Iteration 71 / 1900) loss: 1.916118
(Iteration 81 / 1900) loss: 1.914251
(Iteration 91 / 1900) loss: 1.843215
(Epoch 1 / 20) train acc: 0.440000; val_acc: 0.432000
(Iteration 101 / 1900) loss: 1.837782
(Iteration 111 / 1900) loss: 1.819022
(Iteration 121 / 1900) loss: 1.723029
(Iteration 131 / 1900) loss: 1.743092
(Iteration 141 / 1900) loss: 1.690598
(Iteration 151 / 1900) loss: 1.676515
(Iteration 161 / 1900) loss: 1.683766
(Iteration 171 / 1900) loss: 1.603227
(Iteration 181 / 1900) loss: 1.558714
(Epoch 2 / 20) train acc: 0.481000; val_acc: 0.479000
(Iteration 191 / 1900) loss: 1.570693
(Iteration 201 / 1900) loss: 1.574873
(Iteration 211 / 1900) loss: 1.525901
(Iteration 221 / 1900) loss: 1.530201
(Iteration 231 / 1900) loss: 1.479409
```

```
(Iteration 241 / 1900) loss: 1.467082
(Iteration 251 / 1900) loss: 1.510200
(Iteration 261 / 1900) loss: 1.456700
(Iteration 271 / 1900) loss: 1.411198
(Iteration 281 / 1900) loss: 1.422769
(Epoch 3 / 20) train acc: 0.501000; val_acc: 0.504000
(Iteration 291 / 1900) loss: 1.420156
(Iteration 301 / 1900) loss: 1.414801
(Iteration 311 / 1900) loss: 1.402514
(Iteration 321 / 1900) loss: 1.333788
(Iteration 331 / 1900) loss: 1.358248
(Iteration 341 / 1900) loss: 1.344821
(Iteration 351 / 1900) loss: 1.399796
(Iteration 361 / 1900) loss: 1.338810
(Iteration 371 / 1900) loss: 1.354858
(Epoch 4 / 20) train acc: 0.554000; val_acc: 0.540000
(Iteration 381 / 1900) loss: 1.347631
(Iteration 391 / 1900) loss: 1.273754
(Iteration 401 / 1900) loss: 1.267214
(Iteration 411 / 1900) loss: 1.246008
(Iteration 421 / 1900) loss: 1.280709
(Iteration 431 / 1900) loss: 1.215197
(Iteration 441 / 1900) loss: 1.222235
(Iteration 451 / 1900) loss: 1.217054
(Iteration 461 / 1900) loss: 1.212772
(Iteration 471 / 1900) loss: 1.213423
(Epoch 5 / 20) train acc: 0.598000; val_acc: 0.534000
(Iteration 481 / 1900) loss: 1.194270
(Iteration 491 / 1900) loss: 1.187749
(Iteration 501 / 1900) loss: 1.212511
(Iteration 511 / 1900) loss: 1.239800
(Iteration 521 / 1900) loss: 1.166033
(Iteration 531 / 1900) loss: 1.218656
(Iteration 541 / 1900) loss: 1.227740
(Iteration 551 / 1900) loss: 1.237649
(Iteration 561 / 1900) loss: 1.201103
(Epoch 6 / 20) train acc: 0.564000; val_acc: 0.532000
(Iteration 571 / 1900) loss: 1.226018
(Iteration 581 / 1900) loss: 1.258661
(Iteration 591 / 1900) loss: 1.114167
(Iteration 601 / 1900) loss: 1.216523
(Iteration 611 / 1900) loss: 1.179557
(Iteration 621 / 1900) loss: 1.025499
(Iteration 631 / 1900) loss: 1.130684
(Iteration 641 / 1900) loss: 1.169864
(Iteration 651 / 1900) loss: 1.129548
(Iteration 661 / 1900) loss: 1.063285
(Epoch 7 / 20) train acc: 0.615000; val_acc: 0.520000
```

```
(Iteration 671 / 1900) loss: 1.107173
(Iteration 681 / 1900) loss: 1.132180
(Iteration 691 / 1900) loss: 1.062288
(Iteration 701 / 1900) loss: 1.202746
(Iteration 711 / 1900) loss: 1.131543
(Iteration 721 / 1900) loss: 1.071284
(Iteration 731 / 1900) loss: 1.114349
(Iteration 741 / 1900) loss: 1.130984
(Iteration 751 / 1900) loss: 1.020091
(Epoch 8 / 20) train acc: 0.664000; val_acc: 0.551000
(Iteration 761 / 1900) loss: 1.042274
(Iteration 771 / 1900) loss: 1.088044
(Iteration 781 / 1900) loss: 1.014511
(Iteration 791 / 1900) loss: 1.026069
(Iteration 801 / 1900) loss: 1.089758
(Iteration 811 / 1900) loss: 1.087601
(Iteration 821 / 1900) loss: 1.032785
(Iteration 831 / 1900) loss: 1.075064
(Iteration 841 / 1900) loss: 1.059154
(Iteration 851 / 1900) loss: 1.083477
(Epoch 9 / 20) train acc: 0.676000; val_acc: 0.528000
(Iteration 861 / 1900) loss: 1.069679
(Iteration 871 / 1900) loss: 0.968472
(Iteration 881 / 1900) loss: 1.104207
(Iteration 891 / 1900) loss: 0.965001
(Iteration 901 / 1900) loss: 1.008559
(Iteration 911 / 1900) loss: 0.968956
(Iteration 921 / 1900) loss: 1.033882
(Iteration 931 / 1900) loss: 0.939636
(Iteration 941 / 1900) loss: 0.896143
(Epoch 10 / 20) train acc: 0.682000; val_acc: 0.557000
(Iteration 951 / 1900) loss: 0.982420
(Iteration 961 / 1900) loss: 1.105695
(Iteration 971 / 1900) loss: 1.021305
(Iteration 981 / 1900) loss: 1.002390
(Iteration 991 / 1900) loss: 0.912250
(Iteration 1001 / 1900) loss: 0.951266
(Iteration 1011 / 1900) loss: 0.950273
(Iteration 1021 / 1900) loss: 0.898502
(Iteration 1031 / 1900) loss: 0.960220
(Iteration 1041 / 1900) loss: 0.979218
(Epoch 11 / 20) train acc: 0.693000; val_acc: 0.558000
(Iteration 1051 / 1900) loss: 0.950938
(Iteration 1061 / 1900) loss: 1.001338
(Iteration 1071 / 1900) loss: 0.968002
(Iteration 1081 / 1900) loss: 0.916450
(Iteration 1091 / 1900) loss: 0.925503
(Iteration 1101 / 1900) loss: 0.937261
```

```
(Iteration 1111 / 1900) loss: 0.953230
(Iteration 1121 / 1900) loss: 0.946115
(Iteration 1131 / 1900) loss: 0.936854
(Epoch 12 / 20) train acc: 0.717000; val_acc: 0.553000
(Iteration 1141 / 1900) loss: 0.992105
(Iteration 1151 / 1900) loss: 0.905016
(Iteration 1161 / 1900) loss: 0.975625
(Iteration 1171 / 1900) loss: 0.907929
(Iteration 1181 / 1900) loss: 0.849703
(Iteration 1191 / 1900) loss: 0.929583
(Iteration 1201 / 1900) loss: 0.926948
(Iteration 1211 / 1900) loss: 0.813202
(Iteration 1221 / 1900) loss: 0.929324
(Iteration 1231 / 1900) loss: 0.895645
(Epoch 13 / 20) train acc: 0.756000; val_acc: 0.544000
(Iteration 1241 / 1900) loss: 0.948868
(Iteration 1251 / 1900) loss: 0.915394
(Iteration 1261 / 1900) loss: 0.876287
(Iteration 1271 / 1900) loss: 0.907100
(Iteration 1281 / 1900) loss: 0.853768
(Iteration 1291 / 1900) loss: 0.905423
(Iteration 1301 / 1900) loss: 0.843885
(Iteration 1311 / 1900) loss: 0.893549
(Iteration 1321 / 1900) loss: 0.871911
(Epoch 14 / 20) train acc: 0.731000; val_acc: 0.553000
(Iteration 1331 / 1900) loss: 0.818275
(Iteration 1341 / 1900) loss: 0.929515
(Iteration 1351 / 1900) loss: 0.916245
(Iteration 1361 / 1900) loss: 0.857003
(Iteration 1371 / 1900) loss: 0.751572
(Iteration 1381 / 1900) loss: 0.850707
(Iteration 1391 / 1900) loss: 0.907234
(Iteration 1401 / 1900) loss: 0.870197
(Iteration 1411 / 1900) loss: 0.823621
(Iteration 1421 / 1900) loss: 0.862234
(Epoch 15 / 20) train acc: 0.744000; val_acc: 0.557000
(Iteration 1431 / 1900) loss: 0.904432
(Iteration 1441 / 1900) loss: 0.884439
(Iteration 1451 / 1900) loss: 0.805359
(Iteration 1461 / 1900) loss: 0.823994
(Iteration 1471 / 1900) loss: 0.853532
(Iteration 1481 / 1900) loss: 0.804647
(Iteration 1491 / 1900) loss: 0.799671
(Iteration 1501 / 1900) loss: 0.827212
(Iteration 1511 / 1900) loss: 0.833224
(Epoch 16 / 20) train acc: 0.749000; val_acc: 0.562000
(Iteration 1521 / 1900) loss: 0.793141
(Iteration 1531 / 1900) loss: 0.731454
```

```
(Iteration 1541 / 1900) loss: 0.847981
(Iteration 1551 / 1900) loss: 0.790571
(Iteration 1561 / 1900) loss: 0.731424
(Iteration 1571 / 1900) loss: 0.751561
(Iteration 1581 / 1900) loss: 0.823371
(Iteration 1591 / 1900) loss: 0.809849
(Iteration 1601 / 1900) loss: 0.766200
(Iteration 1611 / 1900) loss: 0.746531
(Epoch 17 / 20) train acc: 0.769000; val_acc: 0.589000
(Iteration 1621 / 1900) loss: 0.764830
(Iteration 1631 / 1900) loss: 0.830139
(Iteration 1641 / 1900) loss: 0.796726
(Iteration 1651 / 1900) loss: 0.778753
(Iteration 1661 / 1900) loss: 0.846073
(Iteration 1671 / 1900) loss: 0.813130
(Iteration 1681 / 1900) loss: 0.760853
(Iteration 1691 / 1900) loss: 0.787361
(Iteration 1701 / 1900) loss: 0.783519
(Epoch 18 / 20) train acc: 0.756000; val_acc: 0.574000
(Iteration 1711 / 1900) loss: 0.820632
(Iteration 1721 / 1900) loss: 0.763092
(Iteration 1731 / 1900) loss: 0.765485
(Iteration 1741 / 1900) loss: 0.735716
(Iteration 1751 / 1900) loss: 0.764311
(Iteration 1761 / 1900) loss: 0.787055
(Iteration 1771 / 1900) loss: 0.730361
(Iteration 1781 / 1900) loss: 0.818016
(Iteration 1791 / 1900) loss: 0.692543
(Iteration 1801 / 1900) loss: 0.795332
(Epoch 19 / 20) train acc: 0.797000; val_acc: 0.549000
(Iteration 1811 / 1900) loss: 0.749064
(Iteration 1821 / 1900) loss: 0.737426
(Iteration 1831 / 1900) loss: 0.794444
(Iteration 1841 / 1900) loss: 0.750488
(Iteration 1851 / 1900) loss: 0.684426
(Iteration 1861 / 1900) loss: 0.738252
(Iteration 1871 / 1900) loss: 0.675885
(Iteration 1881 / 1900) loss: 0.751520
(Iteration 1891 / 1900) loss: 0.626247
(Epoch 20 / 20) train acc: 0.817000; val_acc: 0.555000
(Iteration 1 / 1900) loss: 2.305896
(Epoch 0 / 20) train acc: 0.232000; val_acc: 0.224000
(Iteration 11 / 1900) loss: 2.231616
(Iteration 21 / 1900) loss: 2.163365
(Iteration 31 / 1900) loss: 2.111610
(Iteration 41 / 1900) loss: 2.048385
(Iteration 51 / 1900) loss: 1.988604
(Iteration 61 / 1900) loss: 1.940486
```

```
(Iteration 71 / 1900) loss: 1.930142
(Iteration 81 / 1900) loss: 1.854454
(Iteration 91 / 1900) loss: 1.819968
(Epoch 1 / 20) train acc: 0.416000; val_acc: 0.446000
(Iteration 101 / 1900) loss: 1.800912
(Iteration 111 / 1900) loss: 1.753320
(Iteration 121 / 1900) loss: 1.672672
(Iteration 131 / 1900) loss: 1.703961
(Iteration 141 / 1900) loss: 1.665641
(Iteration 151 / 1900) loss: 1.602749
(Iteration 161 / 1900) loss: 1.571876
(Iteration 171 / 1900) loss: 1.598648
(Iteration 181 / 1900) loss: 1.574209
(Epoch 2 / 20) train acc: 0.509000; val_acc: 0.474000
(Iteration 191 / 1900) loss: 1.488855
(Iteration 201 / 1900) loss: 1.558625
(Iteration 211 / 1900) loss: 1.458478
(Iteration 221 / 1900) loss: 1.521189
(Iteration 231 / 1900) loss: 1.497713
(Iteration 241 / 1900) loss: 1.396543
(Iteration 251 / 1900) loss: 1.436065
(Iteration 261 / 1900) loss: 1.411522
(Iteration 271 / 1900) loss: 1.373527
(Iteration 281 / 1900) loss: 1.416420
(Epoch 3 / 20) train acc: 0.555000; val_acc: 0.514000
(Iteration 291 / 1900) loss: 1.324538
(Iteration 301 / 1900) loss: 1.336863
(Iteration 311 / 1900) loss: 1.325911
(Iteration 321 / 1900) loss: 1.357899
(Iteration 331 / 1900) loss: 1.345069
(Iteration 341 / 1900) loss: 1.322001
(Iteration 351 / 1900) loss: 1.323221
(Iteration 361 / 1900) loss: 1.279409
(Iteration 371 / 1900) loss: 1.329004
(Epoch 4 / 20) train acc: 0.578000; val_acc: 0.536000
(Iteration 381 / 1900) loss: 1.318213
(Iteration 391 / 1900) loss: 1.211801
(Iteration 401 / 1900) loss: 1.304233
(Iteration 411 / 1900) loss: 1.275771
(Iteration 421 / 1900) loss: 1.279947
(Iteration 431 / 1900) loss: 1.266914
(Iteration 441 / 1900) loss: 1.222694
(Iteration 451 / 1900) loss: 1.208769
(Iteration 461 / 1900) loss: 1.307981
(Iteration 471 / 1900) loss: 1.198246
(Epoch 5 / 20) train acc: 0.598000; val_acc: 0.532000
(Iteration 481 / 1900) loss: 1.177375
(Iteration 491 / 1900) loss: 1.188899
```

```
(Iteration 501 / 1900) loss: 1.142193
(Iteration 511 / 1900) loss: 1.117835
(Iteration 521 / 1900) loss: 1.161635
(Iteration 531 / 1900) loss: 1.144536
(Iteration 541 / 1900) loss: 1.182385
(Iteration 551 / 1900) loss: 1.129801
(Iteration 561 / 1900) loss: 1.103515
(Epoch 6 / 20) train acc: 0.621000; val_acc: 0.546000
(Iteration 571 / 1900) loss: 1.062182
(Iteration 581 / 1900) loss: 1.165487
(Iteration 591 / 1900) loss: 1.133725
(Iteration 601 / 1900) loss: 1.119143
(Iteration 611 / 1900) loss: 1.093345
(Iteration 621 / 1900) loss: 1.085413
(Iteration 631 / 1900) loss: 1.095790
(Iteration 641 / 1900) loss: 1.031393
(Iteration 651 / 1900) loss: 0.988585
(Iteration 661 / 1900) loss: 1.123682
(Epoch 7 / 20) train acc: 0.669000; val_acc: 0.563000
(Iteration 671 / 1900) loss: 0.967180
(Iteration 681 / 1900) loss: 1.068718
(Iteration 691 / 1900) loss: 1.039112
(Iteration 701 / 1900) loss: 1.099899
(Iteration 711 / 1900) loss: 1.068542
(Iteration 721 / 1900) loss: 0.961272
(Iteration 731 / 1900) loss: 1.021536
(Iteration 741 / 1900) loss: 1.011646
(Iteration 751 / 1900) loss: 1.069160
(Epoch 8 / 20) train acc: 0.664000; val_acc: 0.542000
(Iteration 761 / 1900) loss: 0.967493
(Iteration 771 / 1900) loss: 0.964808
(Iteration 781 / 1900) loss: 1.025921
(Iteration 791 / 1900) loss: 0.932893
(Iteration 801 / 1900) loss: 0.992465
(Iteration 811 / 1900) loss: 0.885791
(Iteration 821 / 1900) loss: 0.880951
(Iteration 831 / 1900) loss: 0.927054
(Iteration 841 / 1900) loss: 1.037941
(Iteration 851 / 1900) loss: 0.935012
(Epoch 9 / 20) train acc: 0.672000; val_acc: 0.540000
(Iteration 861 / 1900) loss: 0.983268
(Iteration 871 / 1900) loss: 0.938610
(Iteration 881 / 1900) loss: 0.992096
(Iteration 891 / 1900) loss: 0.938760
(Iteration 901 / 1900) loss: 0.896262
(Iteration 911 / 1900) loss: 0.897068
(Iteration 921 / 1900) loss: 1.010276
(Iteration 931 / 1900) loss: 0.936250
```

```
(Iteration 941 / 1900) loss: 0.875226
(Epoch 10 / 20) train acc: 0.693000; val_acc: 0.548000
(Iteration 951 / 1900) loss: 0.863117
(Iteration 961 / 1900) loss: 0.915683
(Iteration 971 / 1900) loss: 0.919069
(Iteration 981 / 1900) loss: 0.907657
(Iteration 991 / 1900) loss: 0.936537
(Iteration 1001 / 1900) loss: 0.942149
(Iteration 1011 / 1900) loss: 0.905303
(Iteration 1021 / 1900) loss: 0.832942
(Iteration 1031 / 1900) loss: 0.877898
(Iteration 1041 / 1900) loss: 0.875857
(Epoch 11 / 20) train acc: 0.713000; val_acc: 0.554000
(Iteration 1051 / 1900) loss: 0.856727
(Iteration 1061 / 1900) loss: 0.898390
(Iteration 1071 / 1900) loss: 0.833193
(Iteration 1081 / 1900) loss: 0.750955
(Iteration 1091 / 1900) loss: 0.822981
(Iteration 1101 / 1900) loss: 0.794184
(Iteration 1111 / 1900) loss: 0.785092
(Iteration 1121 / 1900) loss: 0.877818
(Iteration 1131 / 1900) loss: 0.819849
(Epoch 12 / 20) train acc: 0.720000; val_acc: 0.568000
(Iteration 1141 / 1900) loss: 0.830102
(Iteration 1151 / 1900) loss: 0.842728
(Iteration 1161 / 1900) loss: 0.761416
(Iteration 1171 / 1900) loss: 0.757289
(Iteration 1181 / 1900) loss: 0.871378
(Iteration 1191 / 1900) loss: 0.827703
(Iteration 1201 / 1900) loss: 0.797818
(Iteration 1211 / 1900) loss: 0.826266
(Iteration 1221 / 1900) loss: 0.797295
(Iteration 1231 / 1900) loss: 0.800391
(Epoch 13 / 20) train acc: 0.763000; val_acc: 0.572000
(Iteration 1241 / 1900) loss: 0.804943
(Iteration 1251 / 1900) loss: 0.816227
(Iteration 1261 / 1900) loss: 0.747870
(Iteration 1271 / 1900) loss: 0.716802
(Iteration 1281 / 1900) loss: 0.890606
(Iteration 1291 / 1900) loss: 0.773025
(Iteration 1301 / 1900) loss: 0.717942
(Iteration 1311 / 1900) loss: 0.792796
(Iteration 1321 / 1900) loss: 0.803199
(Epoch 14 / 20) train acc: 0.752000; val_acc: 0.549000
(Iteration 1331 / 1900) loss: 0.748498
(Iteration 1341 / 1900) loss: 0.716624
(Iteration 1351 / 1900) loss: 0.703379
(Iteration 1361 / 1900) loss: 0.732029
```

```
(Iteration 1371 / 1900) loss: 0.741618
(Iteration 1381 / 1900) loss: 0.742181
(Iteration 1391 / 1900) loss: 0.682803
(Iteration 1401 / 1900) loss: 0.699063
(Iteration 1411 / 1900) loss: 0.706992
(Iteration 1421 / 1900) loss: 0.714563
(Epoch 15 / 20) train acc: 0.732000; val_acc: 0.546000
(Iteration 1431 / 1900) loss: 0.700253
(Iteration 1441 / 1900) loss: 0.610868
(Iteration 1451 / 1900) loss: 0.718739
(Iteration 1461 / 1900) loss: 0.729212
(Iteration 1471 / 1900) loss: 0.621523
(Iteration 1481 / 1900) loss: 0.762853
(Iteration 1491 / 1900) loss: 0.686693
(Iteration 1501 / 1900) loss: 0.662835
(Iteration 1511 / 1900) loss: 0.646367
(Epoch 16 / 20) train acc: 0.791000; val_acc: 0.551000
(Iteration 1521 / 1900) loss: 0.652767
(Iteration 1531 / 1900) loss: 0.746642
(Iteration 1541 / 1900) loss: 0.602551
(Iteration 1551 / 1900) loss: 0.637320
(Iteration 1561 / 1900) loss: 0.653958
(Iteration 1571 / 1900) loss: 0.640888
(Iteration 1581 / 1900) loss: 0.571187
(Iteration 1591 / 1900) loss: 0.719457
(Iteration 1601 / 1900) loss: 0.667257
(Iteration 1611 / 1900) loss: 0.584518
(Epoch 17 / 20) train acc: 0.793000; val_acc: 0.545000
(Iteration 1621 / 1900) loss: 0.576212
(Iteration 1631 / 1900) loss: 0.678542
(Iteration 1641 / 1900) loss: 0.702209
(Iteration 1651 / 1900) loss: 0.636829
(Iteration 1661 / 1900) loss: 0.628676
(Iteration 1671 / 1900) loss: 0.592150
(Iteration 1681 / 1900) loss: 0.655360
(Iteration 1691 / 1900) loss: 0.661735
(Iteration 1701 / 1900) loss: 0.635246
(Epoch 18 / 20) train acc: 0.812000; val_acc: 0.547000
(Iteration 1711 / 1900) loss: 0.562784
(Iteration 1721 / 1900) loss: 0.627515
(Iteration 1731 / 1900) loss: 0.588821
(Iteration 1741 / 1900) loss: 0.576228
(Iteration 1751 / 1900) loss: 0.511204
(Iteration 1761 / 1900) loss: 0.619674
(Iteration 1771 / 1900) loss: 0.585093
(Iteration 1781 / 1900) loss: 0.581164
(Iteration 1791 / 1900) loss: 0.635587
(Iteration 1801 / 1900) loss: 0.659438
```

```
(Epoch 19 / 20) train acc: 0.793000; val_acc: 0.550000
(Iteration 1811 / 1900) loss: 0.582259
(Iteration 1821 / 1900) loss: 0.529198
(Iteration 1831 / 1900) loss: 0.591886
(Iteration 1841 / 1900) loss: 0.597005
(Iteration 1851 / 1900) loss: 0.591543
(Iteration 1861 / 1900) loss: 0.592453
(Iteration 1871 / 1900) loss: 0.575763
(Iteration 1881 / 1900) loss: 0.541527
(Iteration 1891 / 1900) loss: 0.548694
(Epoch 20 / 20) train acc: 0.813000; val_acc: 0.554000
(Iteration 1 / 1900) loss: 2.304187
(Epoch 0 / 20) train acc: 0.175000; val_acc: 0.169000
(Iteration 11 / 1900) loss: 2.231545
(Iteration 21 / 1900) loss: 2.174442
(Iteration 31 / 1900) loss: 2.100464
(Iteration 41 / 1900) loss: 2.033587
(Iteration 51 / 1900) loss: 2.000717
(Iteration 61 / 1900) loss: 1.933712
(Iteration 71 / 1900) loss: 1.904359
(Iteration 81 / 1900) loss: 1.873713
(Iteration 91 / 1900) loss: 1.842988
(Epoch 1 / 20) train acc: 0.452000; val_acc: 0.418000
(Iteration 101 / 1900) loss: 1.727520
(Iteration 111 / 1900) loss: 1.752297
(Iteration 121 / 1900) loss: 1.735901
(Iteration 131 / 1900) loss: 1.647615
(Iteration 141 / 1900) loss: 1.681506
(Iteration 151 / 1900) loss: 1.613191
(Iteration 161 / 1900) loss: 1.607669
(Iteration 171 / 1900) loss: 1.555623
(Iteration 181 / 1900) loss: 1.521840
(Epoch 2 / 20) train acc: 0.515000; val_acc: 0.483000
(Iteration 191 / 1900) loss: 1.523907
(Iteration 201 / 1900) loss: 1.515975
(Iteration 211 / 1900) loss: 1.495542
(Iteration 221 / 1900) loss: 1.464675
(Iteration 231 / 1900) loss: 1.424961
(Iteration 241 / 1900) loss: 1.412519
(Iteration 251 / 1900) loss: 1.411853
(Iteration 261 / 1900) loss: 1.400802
(Iteration 271 / 1900) loss: 1.366610
(Iteration 281 / 1900) loss: 1.348392
(Epoch 3 / 20) train acc: 0.520000; val_acc: 0.512000
(Iteration 291 / 1900) loss: 1.365367
(Iteration 301 / 1900) loss: 1.345637
(Iteration 311 / 1900) loss: 1.276147
(Iteration 321 / 1900) loss: 1.301540
```

```
(Iteration 331 / 1900) loss: 1.302975
(Iteration 341 / 1900) loss: 1.346389
(Iteration 351 / 1900) loss: 1.242905
(Iteration 361 / 1900) loss: 1.290442
(Iteration 371 / 1900) loss: 1.304692
(Epoch 4 / 20) train acc: 0.541000; val_acc: 0.527000
(Iteration 381 / 1900) loss: 1.330299
(Iteration 391 / 1900) loss: 1.231492
(Iteration 401 / 1900) loss: 1.205164
(Iteration 411 / 1900) loss: 1.245096
(Iteration 421 / 1900) loss: 1.194515
(Iteration 431 / 1900) loss: 1.262717
(Iteration 441 / 1900) loss: 1.187468
(Iteration 451 / 1900) loss: 1.116582
(Iteration 461 / 1900) loss: 1.174096
(Iteration 471 / 1900) loss: 1.209092
(Epoch 5 / 20) train acc: 0.584000; val_acc: 0.534000
(Iteration 481 / 1900) loss: 1.219940
(Iteration 491 / 1900) loss: 1.122597
(Iteration 501 / 1900) loss: 1.167868
(Iteration 511 / 1900) loss: 1.041952
(Iteration 521 / 1900) loss: 1.080659
(Iteration 531 / 1900) loss: 1.150470
(Iteration 541 / 1900) loss: 1.182717
(Iteration 551 / 1900) loss: 1.069998
(Iteration 561 / 1900) loss: 1.169029
(Epoch 6 / 20) train acc: 0.644000; val_acc: 0.558000
(Iteration 571 / 1900) loss: 1.110117
(Iteration 581 / 1900) loss: 1.064279
(Iteration 591 / 1900) loss: 1.006814
(Iteration 601 / 1900) loss: 1.045935
(Iteration 611 / 1900) loss: 1.087220
(Iteration 621 / 1900) loss: 1.058731
(Iteration 631 / 1900) loss: 1.106221
(Iteration 641 / 1900) loss: 1.031281
(Iteration 651 / 1900) loss: 1.095884
(Iteration 661 / 1900) loss: 1.013572
(Epoch 7 / 20) train acc: 0.650000; val_acc: 0.554000
(Iteration 671 / 1900) loss: 1.006851
(Iteration 681 / 1900) loss: 1.065733
(Iteration 691 / 1900) loss: 1.056725
(Iteration 701 / 1900) loss: 0.920623
(Iteration 711 / 1900) loss: 1.081489
(Iteration 721 / 1900) loss: 0.999628
(Iteration 731 / 1900) loss: 0.907999
(Iteration 741 / 1900) loss: 0.961914
(Iteration 751 / 1900) loss: 1.042307
(Epoch 8 / 20) train acc: 0.667000; val_acc: 0.550000
```

```
(Iteration 761 / 1900) loss: 1.054249
(Iteration 771 / 1900) loss: 1.016495
(Iteration 781 / 1900) loss: 0.959847
(Iteration 791 / 1900) loss: 0.959176
(Iteration 801 / 1900) loss: 0.930863
(Iteration 811 / 1900) loss: 0.966163
(Iteration 821 / 1900) loss: 0.929170
(Iteration 831 / 1900) loss: 0.899010
(Iteration 841 / 1900) loss: 0.957832
(Iteration 851 / 1900) loss: 0.922702
(Epoch 9 / 20) train acc: 0.689000; val_acc: 0.579000
(Iteration 861 / 1900) loss: 0.879273
(Iteration 871 / 1900) loss: 0.993834
(Iteration 881 / 1900) loss: 0.902713
(Iteration 891 / 1900) loss: 0.969531
(Iteration 901 / 1900) loss: 0.872034
(Iteration 911 / 1900) loss: 0.859995
(Iteration 921 / 1900) loss: 0.974700
(Iteration 931 / 1900) loss: 0.798352
(Iteration 941 / 1900) loss: 0.820866
(Epoch 10 / 20) train acc: 0.689000; val_acc: 0.559000
(Iteration 951 / 1900) loss: 0.913360
(Iteration 961 / 1900) loss: 0.828852
(Iteration 971 / 1900) loss: 0.866344
(Iteration 981 / 1900) loss: 0.854385
(Iteration 991 / 1900) loss: 0.887463
(Iteration 1001 / 1900) loss: 0.867746
(Iteration 1011 / 1900) loss: 0.788947
(Iteration 1021 / 1900) loss: 0.893674
(Iteration 1031 / 1900) loss: 0.811138
(Iteration 1041 / 1900) loss: 0.848001
(Epoch 11 / 20) train acc: 0.713000; val_acc: 0.561000
(Iteration 1051 / 1900) loss: 0.861388
(Iteration 1061 / 1900) loss: 0.818460
(Iteration 1071 / 1900) loss: 0.761623
(Iteration 1081 / 1900) loss: 0.751744
(Iteration 1091 / 1900) loss: 0.807852
(Iteration 1101 / 1900) loss: 0.779575
(Iteration 1111 / 1900) loss: 0.762533
(Iteration 1121 / 1900) loss: 0.757411
(Iteration 1131 / 1900) loss: 0.842471
(Epoch 12 / 20) train acc: 0.740000; val_acc: 0.567000
(Iteration 1141 / 1900) loss: 0.874469
(Iteration 1151 / 1900) loss: 0.824852
(Iteration 1161 / 1900) loss: 0.767910
(Iteration 1171 / 1900) loss: 0.749454
(Iteration 1181 / 1900) loss: 0.798119
(Iteration 1191 / 1900) loss: 0.771295
```

```
(Iteration 1201 / 1900) loss: 0.794476
(Iteration 1211 / 1900) loss: 0.723167
(Iteration 1221 / 1900) loss: 0.792289
(Iteration 1231 / 1900) loss: 0.665671
(Epoch 13 / 20) train acc: 0.732000; val_acc: 0.557000
(Iteration 1241 / 1900) loss: 0.680539
(Iteration 1251 / 1900) loss: 0.833766
(Iteration 1261 / 1900) loss: 0.687159
(Iteration 1271 / 1900) loss: 0.703133
(Iteration 1281 / 1900) loss: 0.710762
(Iteration 1291 / 1900) loss: 0.742520
(Iteration 1301 / 1900) loss: 0.745111
(Iteration 1311 / 1900) loss: 0.679584
(Iteration 1321 / 1900) loss: 0.706923
(Epoch 14 / 20) train acc: 0.752000; val_acc: 0.560000
(Iteration 1331 / 1900) loss: 0.707062
(Iteration 1341 / 1900) loss: 0.687105
(Iteration 1351 / 1900) loss: 0.662353
(Iteration 1361 / 1900) loss: 0.675900
(Iteration 1371 / 1900) loss: 0.772588
(Iteration 1381 / 1900) loss: 0.739781
(Iteration 1391 / 1900) loss: 0.818700
(Iteration 1401 / 1900) loss: 0.632764
(Iteration 1411 / 1900) loss: 0.663474
(Iteration 1421 / 1900) loss: 0.648904
(Epoch 15 / 20) train acc: 0.766000; val_acc: 0.557000
(Iteration 1431 / 1900) loss: 0.602585
(Iteration 1441 / 1900) loss: 0.649028
(Iteration 1451 / 1900) loss: 0.652931
(Iteration 1461 / 1900) loss: 0.646049
(Iteration 1471 / 1900) loss: 0.697806
(Iteration 1481 / 1900) loss: 0.644212
(Iteration 1491 / 1900) loss: 0.621145
(Iteration 1501 / 1900) loss: 0.633393
(Iteration 1511 / 1900) loss: 0.702252
(Epoch 16 / 20) train acc: 0.789000; val_acc: 0.562000
(Iteration 1521 / 1900) loss: 0.660820
(Iteration 1531 / 1900) loss: 0.674113
(Iteration 1541 / 1900) loss: 0.642965
(Iteration 1551 / 1900) loss: 0.650541
(Iteration 1561 / 1900) loss: 0.619040
(Iteration 1571 / 1900) loss: 0.598209
(Iteration 1581 / 1900) loss: 0.639649
(Iteration 1591 / 1900) loss: 0.614407
(Iteration 1601 / 1900) loss: 0.559425
(Iteration 1611 / 1900) loss: 0.582534
(Epoch 17 / 20) train acc: 0.802000; val_acc: 0.565000
(Iteration 1621 / 1900) loss: 0.534978
```

```
(Iteration 1631 / 1900) loss: 0.592256
(Iteration 1641 / 1900) loss: 0.599515
(Iteration 1651 / 1900) loss: 0.629361
(Iteration 1661 / 1900) loss: 0.602186
(Iteration 1671 / 1900) loss: 0.639540
(Iteration 1681 / 1900) loss: 0.567431
(Iteration 1691 / 1900) loss: 0.558860
(Iteration 1701 / 1900) loss: 0.606286
(Epoch 18 / 20) train acc: 0.804000; val_acc: 0.542000
(Iteration 1711 / 1900) loss: 0.657189
(Iteration 1721 / 1900) loss: 0.529515
(Iteration 1731 / 1900) loss: 0.482507
(Iteration 1741 / 1900) loss: 0.591453
(Iteration 1751 / 1900) loss: 0.644234
(Iteration 1761 / 1900) loss: 0.513917
(Iteration 1771 / 1900) loss: 0.538696
(Iteration 1781 / 1900) loss: 0.544944
(Iteration 1791 / 1900) loss: 0.551247
(Iteration 1801 / 1900) loss: 0.554515
(Epoch 19 / 20) train acc: 0.811000; val_acc: 0.558000
(Iteration 1811 / 1900) loss: 0.559489
(Iteration 1821 / 1900) loss: 0.510132
(Iteration 1831 / 1900) loss: 0.580237
(Iteration 1841 / 1900) loss: 0.462960
(Iteration 1851 / 1900) loss: 0.554792
(Iteration 1861 / 1900) loss: 0.585162
(Iteration 1871 / 1900) loss: 0.583635
(Iteration 1881 / 1900) loss: 0.467129
(Iteration 1891 / 1900) loss: 0.524568
(Epoch 20 / 20) train acc: 0.845000; val_acc: 0.545000
(Iteration 1 / 1900) loss: 2.677464
(Epoch 0 / 20) train acc: 0.132000; val_acc: 0.170000
(Iteration 11 / 1900) loss: 2.211158
(Iteration 21 / 1900) loss: 2.091085
(Iteration 31 / 1900) loss: 2.009966
(Iteration 41 / 1900) loss: 1.985939
(Iteration 51 / 1900) loss: 1.996634
(Iteration 61 / 1900) loss: 1.955622
(Iteration 71 / 1900) loss: 1.917813
(Iteration 81 / 1900) loss: 1.903735
(Iteration 91 / 1900) loss: 1.933511
(Epoch 1 / 20) train acc: 0.366000; val_acc: 0.398000
(Iteration 101 / 1900) loss: 1.921660
(Iteration 111 / 1900) loss: 1.908319
(Iteration 121 / 1900) loss: 1.935645
(Iteration 131 / 1900) loss: 1.892022
(Iteration 141 / 1900) loss: 1.927241
(Iteration 151 / 1900) loss: 1.851118
```

```
(Iteration 161 / 1900) loss: 1.949434
(Iteration 171 / 1900) loss: 1.941138
(Iteration 181 / 1900) loss: 1.975930
(Epoch 2 / 20) train acc: 0.447000; val_acc: 0.412000
(Iteration 191 / 1900) loss: 1.932625
(Iteration 201 / 1900) loss: 1.869834
(Iteration 211 / 1900) loss: 1.895421
(Iteration 221 / 1900) loss: 1.894596
(Iteration 231 / 1900) loss: 1.864090
(Iteration 241 / 1900) loss: 1.847581
(Iteration 251 / 1900) loss: 1.924528
(Iteration 261 / 1900) loss: 1.865670
(Iteration 271 / 1900) loss: 1.840750
(Iteration 281 / 1900) loss: 1.929093
(Epoch 3 / 20) train acc: 0.431000; val_acc: 0.452000
(Iteration 291 / 1900) loss: 1.937335
(Iteration 301 / 1900) loss: 1.841563
(Iteration 311 / 1900) loss: 1.805165
(Iteration 321 / 1900) loss: 1.845654
(Iteration 331 / 1900) loss: 1.791906
(Iteration 341 / 1900) loss: 1.907325
(Iteration 351 / 1900) loss: 1.797217
(Iteration 361 / 1900) loss: 1.870902
(Iteration 371 / 1900) loss: 1.636113
(Epoch 4 / 20) train acc: 0.449000; val_acc: 0.438000
(Iteration 381 / 1900) loss: 1.848046
(Iteration 391 / 1900) loss: 1.855280
(Iteration 401 / 1900) loss: 1.850866
(Iteration 411 / 1900) loss: 1.855200
(Iteration 421 / 1900) loss: 1.783834
(Iteration 431 / 1900) loss: 1.790627
(Iteration 441 / 1900) loss: 1.806453
(Iteration 451 / 1900) loss: 1.688426
(Iteration 461 / 1900) loss: 1.773692
(Iteration 471 / 1900) loss: 1.860517
(Epoch 5 / 20) train acc: 0.448000; val_acc: 0.421000
(Iteration 481 / 1900) loss: 1.820638
(Iteration 491 / 1900) loss: 1.784258
(Iteration 501 / 1900) loss: 1.811540
(Iteration 511 / 1900) loss: 1.845962
(Iteration 521 / 1900) loss: 1.747710
(Iteration 531 / 1900) loss: 1.821441
(Iteration 541 / 1900) loss: 1.834336
(Iteration 551 / 1900) loss: 1.796493
(Iteration 561 / 1900) loss: 1.778349
(Epoch 6 / 20) train acc: 0.459000; val_acc: 0.431000
(Iteration 571 / 1900) loss: 1.736306
(Iteration 581 / 1900) loss: 1.781403
```

```
(Iteration 591 / 1900) loss: 1.843942
(Iteration 601 / 1900) loss: 1.724985
(Iteration 611 / 1900) loss: 1.710534
(Iteration 621 / 1900) loss: 1.768009
(Iteration 631 / 1900) loss: 1.767417
(Iteration 641 / 1900) loss: 1.691643
(Iteration 651 / 1900) loss: 1.730879
(Iteration 661 / 1900) loss: 1.791475
(Epoch 7 / 20) train acc: 0.465000; val_acc: 0.473000
(Iteration 671 / 1900) loss: 1.837522
(Iteration 681 / 1900) loss: 1.754613
(Iteration 691 / 1900) loss: 1.774502
(Iteration 701 / 1900) loss: 1.718718
(Iteration 711 / 1900) loss: 1.684485
(Iteration 721 / 1900) loss: 1.778040
(Iteration 731 / 1900) loss: 1.744862
(Iteration 741 / 1900) loss: 1.675252
(Iteration 751 / 1900) loss: 1.731724
(Epoch 8 / 20) train acc: 0.486000; val_acc: 0.464000
(Iteration 761 / 1900) loss: 1.762653
(Iteration 771 / 1900) loss: 1.820244
(Iteration 781 / 1900) loss: 1.742290
(Iteration 791 / 1900) loss: 1.736127
(Iteration 801 / 1900) loss: 1.664838
(Iteration 811 / 1900) loss: 1.775936
(Iteration 821 / 1900) loss: 1.781778
(Iteration 831 / 1900) loss: 1.773541
(Iteration 841 / 1900) loss: 1.741270
(Iteration 851 / 1900) loss: 1.745159
(Epoch 9 / 20) train acc: 0.506000; val_acc: 0.471000
(Iteration 861 / 1900) loss: 1.680851
(Iteration 871 / 1900) loss: 1.770332
(Iteration 881 / 1900) loss: 1.711074
(Iteration 891 / 1900) loss: 1.742162
(Iteration 901 / 1900) loss: 1.778967
(Iteration 911 / 1900) loss: 1.751040
(Iteration 921 / 1900) loss: 1.701838
(Iteration 931 / 1900) loss: 1.742928
(Iteration 941 / 1900) loss: 1.734374
(Epoch 10 / 20) train acc: 0.477000; val_acc: 0.452000
(Iteration 951 / 1900) loss: 1.782757
(Iteration 961 / 1900) loss: 1.728663
(Iteration 971 / 1900) loss: 1.752902
(Iteration 981 / 1900) loss: 1.710140
(Iteration 991 / 1900) loss: 1.817302
(Iteration 1001 / 1900) loss: 1.712521
(Iteration 1011 / 1900) loss: 1.718214
(Iteration 1021 / 1900) loss: 1.785863
```

```
(Iteration 1031 / 1900) loss: 1.697317
(Iteration 1041 / 1900) loss: 1.705105
(Epoch 11 / 20) train acc: 0.482000; val_acc: 0.488000
(Iteration 1051 / 1900) loss: 1.808787
(Iteration 1061 / 1900) loss: 1.786324
(Iteration 1071 / 1900) loss: 1.759136
(Iteration 1081 / 1900) loss: 1.710600
(Iteration 1091 / 1900) loss: 1.678712
(Iteration 1101 / 1900) loss: 1.688873
(Iteration 1111 / 1900) loss: 1.693468
(Iteration 1121 / 1900) loss: 1.586477
(Iteration 1131 / 1900) loss: 1.658356
(Epoch 12 / 20) train acc: 0.515000; val_acc: 0.470000
(Iteration 1141 / 1900) loss: 1.641339
(Iteration 1151 / 1900) loss: 1.731618
(Iteration 1161 / 1900) loss: 1.745344
(Iteration 1171 / 1900) loss: 1.621152
(Iteration 1181 / 1900) loss: 1.768271
(Iteration 1191 / 1900) loss: 1.637651
(Iteration 1201 / 1900) loss: 1.766926
(Iteration 1211 / 1900) loss: 1.611059
(Iteration 1221 / 1900) loss: 1.705541
(Iteration 1231 / 1900) loss: 1.736527
(Epoch 13 / 20) train acc: 0.476000; val_acc: 0.478000
(Iteration 1241 / 1900) loss: 1.699439
(Iteration 1251 / 1900) loss: 1.766717
(Iteration 1261 / 1900) loss: 1.707815
(Iteration 1271 / 1900) loss: 1.609507
(Iteration 1281 / 1900) loss: 1.738428
(Iteration 1291 / 1900) loss: 1.693100
(Iteration 1301 / 1900) loss: 1.563122
(Iteration 1311 / 1900) loss: 1.663424
(Iteration 1321 / 1900) loss: 1.672539
(Epoch 14 / 20) train acc: 0.503000; val_acc: 0.469000
(Iteration 1331 / 1900) loss: 1.684412
(Iteration 1341 / 1900) loss: 1.698208
(Iteration 1351 / 1900) loss: 1.705358
(Iteration 1361 / 1900) loss: 1.640560
(Iteration 1371 / 1900) loss: 1.701751
(Iteration 1381 / 1900) loss: 1.617856
(Iteration 1391 / 1900) loss: 1.703166
(Iteration 1401 / 1900) loss: 1.699787
(Iteration 1411 / 1900) loss: 1.681074
(Iteration 1421 / 1900) loss: 1.706965
(Epoch 15 / 20) train acc: 0.500000; val_acc: 0.479000
(Iteration 1431 / 1900) loss: 1.630183
(Iteration 1441 / 1900) loss: 1.630451
(Iteration 1451 / 1900) loss: 1.743517
```

```
(Iteration 1461 / 1900) loss: 1.600386
(Iteration 1471 / 1900) loss: 1.733658
(Iteration 1481 / 1900) loss: 1.777904
(Iteration 1491 / 1900) loss: 1.744034
(Iteration 1501 / 1900) loss: 1.760313
(Iteration 1511 / 1900) loss: 1.710164
(Epoch 16 / 20) train acc: 0.501000; val_acc: 0.489000
(Iteration 1521 / 1900) loss: 1.643633
(Iteration 1531 / 1900) loss: 1.620099
(Iteration 1541 / 1900) loss: 1.639910
(Iteration 1551 / 1900) loss: 1.689297
(Iteration 1561 / 1900) loss: 1.677555
(Iteration 1571 / 1900) loss: 1.617017
(Iteration 1581 / 1900) loss: 1.699711
(Iteration 1591 / 1900) loss: 1.631815
(Iteration 1601 / 1900) loss: 1.647269
(Iteration 1611 / 1900) loss: 1.645295
(Epoch 17 / 20) train acc: 0.478000; val_acc: 0.483000
(Iteration 1621 / 1900) loss: 1.739529
(Iteration 1631 / 1900) loss: 1.661875
(Iteration 1641 / 1900) loss: 1.644832
(Iteration 1651 / 1900) loss: 1.656091
(Iteration 1661 / 1900) loss: 1.656875
(Iteration 1671 / 1900) loss: 1.673034
(Iteration 1681 / 1900) loss: 1.683342
(Iteration 1691 / 1900) loss: 1.646904
(Iteration 1701 / 1900) loss: 1.699102
(Epoch 18 / 20) train acc: 0.476000; val_acc: 0.493000
(Iteration 1711 / 1900) loss: 1.660940
(Iteration 1721 / 1900) loss: 1.615074
(Iteration 1731 / 1900) loss: 1.723505
(Iteration 1741 / 1900) loss: 1.630405
(Iteration 1751 / 1900) loss: 1.603777
(Iteration 1761 / 1900) loss: 1.576824
(Iteration 1771 / 1900) loss: 1.704186
(Iteration 1781 / 1900) loss: 1.652516
(Iteration 1791 / 1900) loss: 1.664139
(Iteration 1801 / 1900) loss: 1.641440
(Epoch 19 / 20) train acc: 0.505000; val_acc: 0.479000
(Iteration 1811 / 1900) loss: 1.581061
(Iteration 1821 / 1900) loss: 1.705080
(Iteration 1831 / 1900) loss: 1.709985
(Iteration 1841 / 1900) loss: 1.614724
(Iteration 1851 / 1900) loss: 1.631090
(Iteration 1861 / 1900) loss: 1.698060
(Iteration 1871 / 1900) loss: 1.734751
(Iteration 1881 / 1900) loss: 1.680861
(Iteration 1891 / 1900) loss: 1.638444
```

```
(Epoch 20 / 20) train acc: 0.510000; val_acc: 0.461000
(Iteration 1 / 1900) loss: 2.378169
(Epoch 0 / 20) train acc: 0.143000; val_acc: 0.155000
(Iteration 11 / 1900) loss: 2.206770
(Iteration 21 / 1900) loss: 2.064885
(Iteration 31 / 1900) loss: 1.934560
(Iteration 41 / 1900) loss: 1.891418
(Iteration 51 / 1900) loss: 1.821549
(Iteration 61 / 1900) loss: 1.731607
(Iteration 71 / 1900) loss: 1.752319
(Iteration 81 / 1900) loss: 1.748922
(Iteration 91 / 1900) loss: 1.693566
(Epoch 1 / 20) train acc: 0.462000; val_acc: 0.418000
(Iteration 101 / 1900) loss: 1.669485
(Iteration 111 / 1900) loss: 1.667094
(Iteration 121 / 1900) loss: 1.634182
(Iteration 131 / 1900) loss: 1.628543
(Iteration 141 / 1900) loss: 1.708237
(Iteration 151 / 1900) loss: 1.769781
(Iteration 161 / 1900) loss: 1.610571
(Iteration 171 / 1900) loss: 1.620686
(Iteration 181 / 1900) loss: 1.624516
(Epoch 2 / 20) train acc: 0.473000; val_acc: 0.478000
(Iteration 191 / 1900) loss: 1.626816
(Iteration 201 / 1900) loss: 1.607007
(Iteration 211 / 1900) loss: 1.563871
(Iteration 221 / 1900) loss: 1.674549
(Iteration 231 / 1900) loss: 1.654347
(Iteration 241 / 1900) loss: 1.658273
(Iteration 251 / 1900) loss: 1.587437
(Iteration 261 / 1900) loss: 1.659444
(Iteration 271 / 1900) loss: 1.590306
(Iteration 281 / 1900) loss: 1.643053
(Epoch 3 / 20) train acc: 0.503000; val_acc: 0.494000
(Iteration 291 / 1900) loss: 1.566466
(Iteration 301 / 1900) loss: 1.620426
(Iteration 311 / 1900) loss: 1.586500
(Iteration 321 / 1900) loss: 1.643454
(Iteration 331 / 1900) loss: 1.602681
(Iteration 341 / 1900) loss: 1.600278
(Iteration 351 / 1900) loss: 1.567217
(Iteration 361 / 1900) loss: 1.592533
(Iteration 371 / 1900) loss: 1.471845
(Epoch 4 / 20) train acc: 0.490000; val_acc: 0.498000
(Iteration 381 / 1900) loss: 1.510759
(Iteration 391 / 1900) loss: 1.551397
(Iteration 401 / 1900) loss: 1.579469
(Iteration 411 / 1900) loss: 1.593632
```

```
(Iteration 421 / 1900) loss: 1.501253
(Iteration 431 / 1900) loss: 1.499444
(Iteration 441 / 1900) loss: 1.516478
(Iteration 451 / 1900) loss: 1.484705
(Iteration 461 / 1900) loss: 1.586190
(Iteration 471 / 1900) loss: 1.588234
(Epoch 5 / 20) train acc: 0.506000; val_acc: 0.483000
(Iteration 481 / 1900) loss: 1.558565
(Iteration 491 / 1900) loss: 1.634677
(Iteration 501 / 1900) loss: 1.543589
(Iteration 511 / 1900) loss: 1.557670
(Iteration 521 / 1900) loss: 1.527208
(Iteration 531 / 1900) loss: 1.571594
(Iteration 541 / 1900) loss: 1.611429
(Iteration 551 / 1900) loss: 1.448325
(Iteration 561 / 1900) loss: 1.481429
(Epoch 6 / 20) train acc: 0.530000; val_acc: 0.492000
(Iteration 571 / 1900) loss: 1.547440
(Iteration 581 / 1900) loss: 1.418128
(Iteration 591 / 1900) loss: 1.533695
(Iteration 601 / 1900) loss: 1.463286
(Iteration 611 / 1900) loss: 1.544819
(Iteration 621 / 1900) loss: 1.478206
(Iteration 631 / 1900) loss: 1.459956
(Iteration 641 / 1900) loss: 1.448174
(Iteration 651 / 1900) loss: 1.520096
(Iteration 661 / 1900) loss: 1.561917
(Epoch 7 / 20) train acc: 0.548000; val_acc: 0.496000
(Iteration 671 / 1900) loss: 1.513444
(Iteration 681 / 1900) loss: 1.514715
(Iteration 691 / 1900) loss: 1.492748
(Iteration 701 / 1900) loss: 1.592411
(Iteration 711 / 1900) loss: 1.534883
(Iteration 721 / 1900) loss: 1.511351
(Iteration 731 / 1900) loss: 1.444269
(Iteration 741 / 1900) loss: 1.439937
(Iteration 751 / 1900) loss: 1.468074
(Epoch 8 / 20) train acc: 0.544000; val_acc: 0.492000
(Iteration 761 / 1900) loss: 1.388038
(Iteration 771 / 1900) loss: 1.381278
(Iteration 781 / 1900) loss: 1.422442
(Iteration 791 / 1900) loss: 1.526827
(Iteration 801 / 1900) loss: 1.492957
(Iteration 811 / 1900) loss: 1.555982
(Iteration 821 / 1900) loss: 1.484562
(Iteration 831 / 1900) loss: 1.577065
(Iteration 841 / 1900) loss: 1.459781
(Iteration 851 / 1900) loss: 1.504689
```

```
(Epoch 9 / 20) train acc: 0.554000; val_acc: 0.515000
(Iteration 861 / 1900) loss: 1.496050
(Iteration 871 / 1900) loss: 1.456095
(Iteration 881 / 1900) loss: 1.472479
(Iteration 891 / 1900) loss: 1.482510
(Iteration 901 / 1900) loss: 1.512925
(Iteration 911 / 1900) loss: 1.475518
(Iteration 921 / 1900) loss: 1.493364
(Iteration 931 / 1900) loss: 1.494374
(Iteration 941 / 1900) loss: 1.434188
(Epoch 10 / 20) train acc: 0.600000; val_acc: 0.515000
(Iteration 951 / 1900) loss: 1.539662
(Iteration 961 / 1900) loss: 1.473816
(Iteration 971 / 1900) loss: 1.455717
(Iteration 981 / 1900) loss: 1.457689
(Iteration 991 / 1900) loss: 1.485063
(Iteration 1001 / 1900) loss: 1.513166
(Iteration 1011 / 1900) loss: 1.512596
(Iteration 1021 / 1900) loss: 1.520184
(Iteration 1031 / 1900) loss: 1.496361
(Iteration 1041 / 1900) loss: 1.472572
(Epoch 11 / 20) train acc: 0.547000; val_acc: 0.517000
(Iteration 1051 / 1900) loss: 1.454016
(Iteration 1061 / 1900) loss: 1.460779
(Iteration 1071 / 1900) loss: 1.416280
(Iteration 1081 / 1900) loss: 1.496209
(Iteration 1091 / 1900) loss: 1.501161
(Iteration 1101 / 1900) loss: 1.548161
(Iteration 1111 / 1900) loss: 1.450298
(Iteration 1121 / 1900) loss: 1.438085
(Iteration 1131 / 1900) loss: 1.484231
(Epoch 12 / 20) train acc: 0.587000; val_acc: 0.525000
(Iteration 1141 / 1900) loss: 1.506955
(Iteration 1151 / 1900) loss: 1.401178
(Iteration 1161 / 1900) loss: 1.493962
(Iteration 1171 / 1900) loss: 1.416350
(Iteration 1181 / 1900) loss: 1.381369
(Iteration 1191 / 1900) loss: 1.414033
(Iteration 1201 / 1900) loss: 1.475962
(Iteration 1211 / 1900) loss: 1.492841
(Iteration 1221 / 1900) loss: 1.496623
(Iteration 1231 / 1900) loss: 1.315759
(Epoch 13 / 20) train acc: 0.584000; val_acc: 0.529000
(Iteration 1241 / 1900) loss: 1.440652
(Iteration 1251 / 1900) loss: 1.427809
(Iteration 1261 / 1900) loss: 1.432559
(Iteration 1271 / 1900) loss: 1.370558
(Iteration 1281 / 1900) loss: 1.377288
```

```
(Iteration 1291 / 1900) loss: 1.416219
(Iteration 1301 / 1900) loss: 1.385175
(Iteration 1311 / 1900) loss: 1.393963
(Iteration 1321 / 1900) loss: 1.433345
(Epoch 14 / 20) train acc: 0.587000; val_acc: 0.508000
(Iteration 1331 / 1900) loss: 1.393347
(Iteration 1341 / 1900) loss: 1.405571
(Iteration 1351 / 1900) loss: 1.415708
(Iteration 1361 / 1900) loss: 1.394494
(Iteration 1371 / 1900) loss: 1.379177
(Iteration 1381 / 1900) loss: 1.466260
(Iteration 1391 / 1900) loss: 1.364332
(Iteration 1401 / 1900) loss: 1.436135
(Iteration 1411 / 1900) loss: 1.379269
(Iteration 1421 / 1900) loss: 1.426440
(Epoch 15 / 20) train acc: 0.559000; val_acc: 0.527000
(Iteration 1431 / 1900) loss: 1.451974
(Iteration 1441 / 1900) loss: 1.383705
(Iteration 1451 / 1900) loss: 1.398935
(Iteration 1461 / 1900) loss: 1.405154
(Iteration 1471 / 1900) loss: 1.457273
(Iteration 1481 / 1900) loss: 1.411270
(Iteration 1491 / 1900) loss: 1.445799
(Iteration 1501 / 1900) loss: 1.441985
(Iteration 1511 / 1900) loss: 1.460915
(Epoch 16 / 20) train acc: 0.546000; val_acc: 0.528000
(Iteration 1521 / 1900) loss: 1.398939
(Iteration 1531 / 1900) loss: 1.402393
(Iteration 1541 / 1900) loss: 1.408116
(Iteration 1551 / 1900) loss: 1.363972
(Iteration 1561 / 1900) loss: 1.381195
(Iteration 1571 / 1900) loss: 1.471897
(Iteration 1581 / 1900) loss: 1.393235
(Iteration 1591 / 1900) loss: 1.361045
(Iteration 1601 / 1900) loss: 1.476291
(Iteration 1611 / 1900) loss: 1.357535
(Epoch 17 / 20) train acc: 0.612000; val_acc: 0.539000
(Iteration 1621 / 1900) loss: 1.456333
(Iteration 1631 / 1900) loss: 1.449119
(Iteration 1641 / 1900) loss: 1.450711
(Iteration 1651 / 1900) loss: 1.477103
(Iteration 1661 / 1900) loss: 1.376970
(Iteration 1671 / 1900) loss: 1.477295
(Iteration 1681 / 1900) loss: 1.494908
(Iteration 1691 / 1900) loss: 1.476434
(Iteration 1701 / 1900) loss: 1.469363
(Epoch 18 / 20) train acc: 0.594000; val_acc: 0.521000
(Iteration 1711 / 1900) loss: 1.424215
```

```
(Iteration 1721 / 1900) loss: 1.433402
(Iteration 1731 / 1900) loss: 1.372421
(Iteration 1741 / 1900) loss: 1.409447
(Iteration 1751 / 1900) loss: 1.360687
(Iteration 1761 / 1900) loss: 1.493315
(Iteration 1771 / 1900) loss: 1.469804
(Iteration 1781 / 1900) loss: 1.403662
(Iteration 1791 / 1900) loss: 1.414855
(Iteration 1801 / 1900) loss: 1.428385
(Epoch 19 / 20) train acc: 0.589000; val_acc: 0.521000
(Iteration 1811 / 1900) loss: 1.465180
(Iteration 1821 / 1900) loss: 1.392090
(Iteration 1831 / 1900) loss: 1.469168
(Iteration 1841 / 1900) loss: 1.349845
(Iteration 1851 / 1900) loss: 1.380138
(Iteration 1861 / 1900) loss: 1.408122
(Iteration 1871 / 1900) loss: 1.362765
(Iteration 1881 / 1900) loss: 1.480395
(Iteration 1891 / 1900) loss: 1.375070
(Epoch 20 / 20) train acc: 0.577000; val_acc: 0.539000
(Iteration 1 / 1900) loss: 2.322229
(Epoch 0 / 20) train acc: 0.169000; val_acc: 0.186000
(Iteration 11 / 1900) loss: 2.186585
(Iteration 21 / 1900) loss: 2.010271
(Iteration 31 / 1900) loss: 1.908499
(Iteration 41 / 1900) loss: 1.843808
(Iteration 51 / 1900) loss: 1.774696
(Iteration 61 / 1900) loss: 1.706623
(Iteration 71 / 1900) loss: 1.708348
(Iteration 81 / 1900) loss: 1.586297
(Iteration 91 / 1900) loss: 1.675813
(Epoch 1 / 20) train acc: 0.453000; val_acc: 0.453000
(Iteration 101 / 1900) loss: 1.576787
(Iteration 111 / 1900) loss: 1.516103
(Iteration 121 / 1900) loss: 1.589808
(Iteration 131 / 1900) loss: 1.591042
(Iteration 141 / 1900) loss: 1.463449
(Iteration 151 / 1900) loss: 1.491957
(Iteration 161 / 1900) loss: 1.443142
(Iteration 171 / 1900) loss: 1.497076
(Iteration 181 / 1900) loss: 1.443972
(Epoch 2 / 20) train acc: 0.475000; val_acc: 0.471000
(Iteration 191 / 1900) loss: 1.492056
(Iteration 201 / 1900) loss: 1.465365
(Iteration 211 / 1900) loss: 1.497782
(Iteration 221 / 1900) loss: 1.510655
(Iteration 231 / 1900) loss: 1.457554
(Iteration 241 / 1900) loss: 1.465071
```

```
(Iteration 251 / 1900) loss: 1.378553
(Iteration 261 / 1900) loss: 1.420076
(Iteration 271 / 1900) loss: 1.456880
(Iteration 281 / 1900) loss: 1.432184
(Epoch 3 / 20) train acc: 0.530000; val_acc: 0.517000
(Iteration 291 / 1900) loss: 1.417371
(Iteration 301 / 1900) loss: 1.458833
(Iteration 311 / 1900) loss: 1.387820
(Iteration 321 / 1900) loss: 1.338716
(Iteration 331 / 1900) loss: 1.362712
(Iteration 341 / 1900) loss: 1.391657
(Iteration 351 / 1900) loss: 1.342587
(Iteration 361 / 1900) loss: 1.353022
(Iteration 371 / 1900) loss: 1.412516
(Epoch 4 / 20) train acc: 0.554000; val_acc: 0.520000
(Iteration 381 / 1900) loss: 1.338360
(Iteration 391 / 1900) loss: 1.444371
(Iteration 401 / 1900) loss: 1.501125
(Iteration 411 / 1900) loss: 1.467343
(Iteration 421 / 1900) loss: 1.351370
(Iteration 431 / 1900) loss: 1.338647
(Iteration 441 / 1900) loss: 1.392447
(Iteration 451 / 1900) loss: 1.372388
(Iteration 461 / 1900) loss: 1.385766
(Iteration 471 / 1900) loss: 1.333303
(Epoch 5 / 20) train acc: 0.548000; val_acc: 0.540000
(Iteration 481 / 1900) loss: 1.377165
(Iteration 491 / 1900) loss: 1.273769
(Iteration 501 / 1900) loss: 1.374539
(Iteration 511 / 1900) loss: 1.313705
(Iteration 521 / 1900) loss: 1.280493
(Iteration 531 / 1900) loss: 1.348135
(Iteration 541 / 1900) loss: 1.269103
(Iteration 551 / 1900) loss: 1.304785
(Iteration 561 / 1900) loss: 1.259916
(Epoch 6 / 20) train acc: 0.582000; val_acc: 0.515000
(Iteration 571 / 1900) loss: 1.337148
(Iteration 581 / 1900) loss: 1.370554
(Iteration 591 / 1900) loss: 1.306817
(Iteration 601 / 1900) loss: 1.308152
(Iteration 611 / 1900) loss: 1.335124
(Iteration 621 / 1900) loss: 1.309728
(Iteration 631 / 1900) loss: 1.238281
(Iteration 641 / 1900) loss: 1.244635
(Iteration 651 / 1900) loss: 1.284839
(Iteration 661 / 1900) loss: 1.331694
(Epoch 7 / 20) train acc: 0.585000; val_acc: 0.556000
(Iteration 671 / 1900) loss: 1.334685
```

```
(Iteration 681 / 1900) loss: 1.349427
(Iteration 691 / 1900) loss: 1.311597
(Iteration 701 / 1900) loss: 1.343272
(Iteration 711 / 1900) loss: 1.329671
(Iteration 721 / 1900) loss: 1.253484
(Iteration 731 / 1900) loss: 1.379892
(Iteration 741 / 1900) loss: 1.236545
(Iteration 751 / 1900) loss: 1.254313
(Epoch 8 / 20) train acc: 0.608000; val_acc: 0.524000
(Iteration 761 / 1900) loss: 1.295141
(Iteration 771 / 1900) loss: 1.271415
(Iteration 781 / 1900) loss: 1.174115
(Iteration 791 / 1900) loss: 1.311366
(Iteration 801 / 1900) loss: 1.319828
(Iteration 811 / 1900) loss: 1.237708
(Iteration 821 / 1900) loss: 1.232798
(Iteration 831 / 1900) loss: 1.209745
(Iteration 841 / 1900) loss: 1.273159
(Iteration 851 / 1900) loss: 1.293949
(Epoch 9 / 20) train acc: 0.613000; val_acc: 0.534000
(Iteration 861 / 1900) loss: 1.164004
(Iteration 871 / 1900) loss: 1.288246
(Iteration 881 / 1900) loss: 1.255834
(Iteration 891 / 1900) loss: 1.342769
(Iteration 901 / 1900) loss: 1.246230
(Iteration 911 / 1900) loss: 1.195006
(Iteration 921 / 1900) loss: 1.221824
(Iteration 931 / 1900) loss: 1.318552
(Iteration 941 / 1900) loss: 1.215530
(Epoch 10 / 20) train acc: 0.617000; val_acc: 0.537000
(Iteration 951 / 1900) loss: 1.299099
(Iteration 961 / 1900) loss: 1.281669
(Iteration 971 / 1900) loss: 1.248529
(Iteration 981 / 1900) loss: 1.226420
(Iteration 991 / 1900) loss: 1.285313
(Iteration 1001 / 1900) loss: 1.243011
(Iteration 1011 / 1900) loss: 1.283571
(Iteration 1021 / 1900) loss: 1.327925
(Iteration 1031 / 1900) loss: 1.269786
(Iteration 1041 / 1900) loss: 1.157647
(Epoch 11 / 20) train acc: 0.622000; val_acc: 0.553000
(Iteration 1051 / 1900) loss: 1.277308
(Iteration 1061 / 1900) loss: 1.178237
(Iteration 1071 / 1900) loss: 1.252583
(Iteration 1081 / 1900) loss: 1.267329
(Iteration 1091 / 1900) loss: 1.215453
(Iteration 1101 / 1900) loss: 1.255122
(Iteration 1111 / 1900) loss: 1.234562
```

```
(Iteration 1121 / 1900) loss: 1.214739
(Iteration 1131 / 1900) loss: 1.114451
(Epoch 12 / 20) train acc: 0.621000; val_acc: 0.530000
(Iteration 1141 / 1900) loss: 1.329989
(Iteration 1151 / 1900) loss: 1.253620
(Iteration 1161 / 1900) loss: 1.194265
(Iteration 1171 / 1900) loss: 1.310846
(Iteration 1181 / 1900) loss: 1.230368
(Iteration 1191 / 1900) loss: 1.201901
(Iteration 1201 / 1900) loss: 1.163173
(Iteration 1211 / 1900) loss: 1.229643
(Iteration 1221 / 1900) loss: 1.191385
(Iteration 1231 / 1900) loss: 1.138066
(Epoch 13 / 20) train acc: 0.671000; val_acc: 0.541000
(Iteration 1241 / 1900) loss: 1.249343
(Iteration 1251 / 1900) loss: 1.153650
(Iteration 1261 / 1900) loss: 1.227745
(Iteration 1271 / 1900) loss: 1.196021
(Iteration 1281 / 1900) loss: 1.227040
(Iteration 1291 / 1900) loss: 1.195439
(Iteration 1301 / 1900) loss: 1.164659
(Iteration 1311 / 1900) loss: 1.148153
(Iteration 1321 / 1900) loss: 1.263479
(Epoch 14 / 20) train acc: 0.655000; val_acc: 0.549000
(Iteration 1331 / 1900) loss: 1.198541
(Iteration 1341 / 1900) loss: 1.099813
(Iteration 1351 / 1900) loss: 1.118906
(Iteration 1361 / 1900) loss: 1.145160
(Iteration 1371 / 1900) loss: 1.146973
(Iteration 1381 / 1900) loss: 1.143150
(Iteration 1391 / 1900) loss: 1.159525
(Iteration 1401 / 1900) loss: 1.248019
(Iteration 1411 / 1900) loss: 1.213679
(Iteration 1421 / 1900) loss: 1.138565
(Epoch 15 / 20) train acc: 0.684000; val_acc: 0.549000
(Iteration 1431 / 1900) loss: 1.175139
(Iteration 1441 / 1900) loss: 1.161216
(Iteration 1451 / 1900) loss: 1.123919
(Iteration 1461 / 1900) loss: 1.211806
(Iteration 1471 / 1900) loss: 1.073738
(Iteration 1481 / 1900) loss: 1.103933
(Iteration 1491 / 1900) loss: 1.187319
(Iteration 1501 / 1900) loss: 1.248554
(Iteration 1511 / 1900) loss: 1.200534
(Epoch 16 / 20) train acc: 0.646000; val_acc: 0.541000
(Iteration 1521 / 1900) loss: 1.178836
(Iteration 1531 / 1900) loss: 1.196159
(Iteration 1541 / 1900) loss: 1.156338
```

```
(Iteration 1551 / 1900) loss: 1.217053
(Iteration 1561 / 1900) loss: 1.139520
(Iteration 1571 / 1900) loss: 1.140333
(Iteration 1581 / 1900) loss: 1.178817
(Iteration 1591 / 1900) loss: 1.201295
(Iteration 1601 / 1900) loss: 1.162813
(Iteration 1611 / 1900) loss: 1.141385
(Epoch 17 / 20) train acc: 0.664000; val_acc: 0.545000
(Iteration 1621 / 1900) loss: 1.142096
(Iteration 1631 / 1900) loss: 1.117526
(Iteration 1641 / 1900) loss: 1.177933
(Iteration 1651 / 1900) loss: 1.151354
(Iteration 1661 / 1900) loss: 1.033302
(Iteration 1671 / 1900) loss: 1.091873
(Iteration 1681 / 1900) loss: 1.212679
(Iteration 1691 / 1900) loss: 1.145449
(Iteration 1701 / 1900) loss: 1.107374
(Epoch 18 / 20) train acc: 0.641000; val_acc: 0.528000
(Iteration 1711 / 1900) loss: 1.188678
(Iteration 1721 / 1900) loss: 1.100038
(Iteration 1731 / 1900) loss: 1.161299
(Iteration 1741 / 1900) loss: 1.176330
(Iteration 1751 / 1900) loss: 1.203273
(Iteration 1761 / 1900) loss: 1.236469
(Iteration 1771 / 1900) loss: 1.107718
(Iteration 1781 / 1900) loss: 1.132538
(Iteration 1791 / 1900) loss: 1.149107
(Iteration 1801 / 1900) loss: 1.134785
(Epoch 19 / 20) train acc: 0.680000; val_acc: 0.532000
(Iteration 1811 / 1900) loss: 1.198639
(Iteration 1821 / 1900) loss: 1.133540
(Iteration 1831 / 1900) loss: 1.149528
(Iteration 1841 / 1900) loss: 1.118785
(Iteration 1851 / 1900) loss: 1.153964
(Iteration 1861 / 1900) loss: 1.165914
(Iteration 1871 / 1900) loss: 1.175826
(Iteration 1881 / 1900) loss: 1.129427
(Iteration 1891 / 1900) loss: 1.191126
(Epoch 20 / 20) train acc: 0.672000; val_acc: 0.555000
(Iteration 1 / 1900) loss: 2.305920
(Epoch 0 / 20) train acc: 0.192000; val_acc: 0.189000
(Iteration 11 / 1900) loss: 2.138004
(Iteration 21 / 1900) loss: 2.004230
(Iteration 31 / 1900) loss: 1.890049
(Iteration 41 / 1900) loss: 1.797810
(Iteration 51 / 1900) loss: 1.760853
(Iteration 61 / 1900) loss: 1.713110
(Iteration 71 / 1900) loss: 1.605963
```

```
(Iteration 81 / 1900) loss: 1.570845
(Iteration 91 / 1900) loss: 1.550059
(Epoch 1 / 20) train acc: 0.419000; val_acc: 0.455000
(Iteration 101 / 1900) loss: 1.552277
(Iteration 111 / 1900) loss: 1.514028
(Iteration 121 / 1900) loss: 1.512170
(Iteration 131 / 1900) loss: 1.488210
(Iteration 141 / 1900) loss: 1.430382
(Iteration 151 / 1900) loss: 1.403107
(Iteration 161 / 1900) loss: 1.422899
(Iteration 171 / 1900) loss: 1.433105
(Iteration 181 / 1900) loss: 1.409635
(Epoch 2 / 20) train acc: 0.535000; val_acc: 0.497000
(Iteration 191 / 1900) loss: 1.403551
(Iteration 201 / 1900) loss: 1.424006
(Iteration 211 / 1900) loss: 1.333377
(Iteration 221 / 1900) loss: 1.341790
(Iteration 231 / 1900) loss: 1.372316
(Iteration 241 / 1900) loss: 1.365258
(Iteration 251 / 1900) loss: 1.356159
(Iteration 261 / 1900) loss: 1.253493
(Iteration 271 / 1900) loss: 1.213674
(Iteration 281 / 1900) loss: 1.248769
(Epoch 3 / 20) train acc: 0.571000; val_acc: 0.525000
(Iteration 291 / 1900) loss: 1.291178
(Iteration 301 / 1900) loss: 1.281860
(Iteration 311 / 1900) loss: 1.265987
(Iteration 321 / 1900) loss: 1.236308
(Iteration 331 / 1900) loss: 1.242558
(Iteration 341 / 1900) loss: 1.239098
(Iteration 351 / 1900) loss: 1.317141
(Iteration 361 / 1900) loss: 1.240446
(Iteration 371 / 1900) loss: 1.259350
(Epoch 4 / 20) train acc: 0.595000; val_acc: 0.531000
(Iteration 381 / 1900) loss: 1.175289
(Iteration 391 / 1900) loss: 1.274234
(Iteration 401 / 1900) loss: 1.238461
(Iteration 411 / 1900) loss: 1.223207
(Iteration 421 / 1900) loss: 1.209558
(Iteration 431 / 1900) loss: 1.184414
(Iteration 441 / 1900) loss: 1.238520
(Iteration 451 / 1900) loss: 1.212739
(Iteration 461 / 1900) loss: 1.137799
(Iteration 471 / 1900) loss: 1.155083
(Epoch 5 / 20) train acc: 0.572000; val_acc: 0.537000
(Iteration 481 / 1900) loss: 1.107768
(Iteration 491 / 1900) loss: 1.240335
(Iteration 501 / 1900) loss: 1.223401
```

```
(Iteration 511 / 1900) loss: 1.172261
(Iteration 521 / 1900) loss: 1.087360
(Iteration 531 / 1900) loss: 1.173597
(Iteration 541 / 1900) loss: 1.202471
(Iteration 551 / 1900) loss: 1.172937
(Iteration 561 / 1900) loss: 1.145926
(Epoch 6 / 20) train acc: 0.610000; val_acc: 0.556000
(Iteration 571 / 1900) loss: 1.186579
(Iteration 581 / 1900) loss: 1.167907
(Iteration 591 / 1900) loss: 1.219540
(Iteration 601 / 1900) loss: 1.063721
(Iteration 611 / 1900) loss: 1.123539
(Iteration 621 / 1900) loss: 1.148203
(Iteration 631 / 1900) loss: 1.181921
(Iteration 641 / 1900) loss: 1.064275
(Iteration 651 / 1900) loss: 1.089957
(Iteration 661 / 1900) loss: 1.097405
(Epoch 7 / 20) train acc: 0.634000; val_acc: 0.532000
(Iteration 671 / 1900) loss: 1.086729
(Iteration 681 / 1900) loss: 1.092285
(Iteration 691 / 1900) loss: 1.087258
(Iteration 701 / 1900) loss: 1.130845
(Iteration 711 / 1900) loss: 1.100056
(Iteration 721 / 1900) loss: 1.070400
(Iteration 731 / 1900) loss: 1.079985
(Iteration 741 / 1900) loss: 1.046288
(Iteration 751 / 1900) loss: 1.100283
(Epoch 8 / 20) train acc: 0.683000; val_acc: 0.548000
(Iteration 761 / 1900) loss: 1.112660
(Iteration 771 / 1900) loss: 1.014889
(Iteration 781 / 1900) loss: 1.069467
(Iteration 791 / 1900) loss: 1.046789
(Iteration 801 / 1900) loss: 1.034280
(Iteration 811 / 1900) loss: 1.010049
(Iteration 821 / 1900) loss: 1.045218
(Iteration 831 / 1900) loss: 1.071602
(Iteration 841 / 1900) loss: 1.058951
(Iteration 851 / 1900) loss: 1.058137
(Epoch 9 / 20) train acc: 0.656000; val_acc: 0.551000
(Iteration 861 / 1900) loss: 1.035900
(Iteration 871 / 1900) loss: 1.111878
(Iteration 881 / 1900) loss: 1.026110
(Iteration 891 / 1900) loss: 1.032056
(Iteration 901 / 1900) loss: 1.054499
(Iteration 911 / 1900) loss: 1.078244
(Iteration 921 / 1900) loss: 1.099922
(Iteration 931 / 1900) loss: 1.017187
(Iteration 941 / 1900) loss: 1.054193
```

```
(Epoch 10 / 20) train acc: 0.685000; val_acc: 0.566000
(Iteration 951 / 1900) loss: 1.028357
(Iteration 961 / 1900) loss: 0.997441
(Iteration 971 / 1900) loss: 0.959815
(Iteration 981 / 1900) loss: 0.905146
(Iteration 991 / 1900) loss: 1.019729
(Iteration 1001 / 1900) loss: 1.028556
(Iteration 1011 / 1900) loss: 1.018934
(Iteration 1021 / 1900) loss: 0.972960
(Iteration 1031 / 1900) loss: 1.029326
(Iteration 1041 / 1900) loss: 0.921450
(Epoch 11 / 20) train acc: 0.722000; val_acc: 0.529000
(Iteration 1051 / 1900) loss: 0.986764
(Iteration 1061 / 1900) loss: 0.956537
(Iteration 1071 / 1900) loss: 0.973160
(Iteration 1081 / 1900) loss: 0.956130
(Iteration 1091 / 1900) loss: 1.028587
(Iteration 1101 / 1900) loss: 0.931727
(Iteration 1111 / 1900) loss: 0.983746
(Iteration 1121 / 1900) loss: 0.976018
(Iteration 1131 / 1900) loss: 0.889591
(Epoch 12 / 20) train acc: 0.732000; val_acc: 0.548000
(Iteration 1141 / 1900) loss: 0.928608
(Iteration 1151 / 1900) loss: 1.008525
(Iteration 1161 / 1900) loss: 0.956787
(Iteration 1171 / 1900) loss: 0.959951
(Iteration 1181 / 1900) loss: 0.969922
(Iteration 1191 / 1900) loss: 0.937967
(Iteration 1201 / 1900) loss: 1.020289
(Iteration 1211 / 1900) loss: 0.946756
(Iteration 1221 / 1900) loss: 0.906480
(Iteration 1231 / 1900) loss: 0.983394
(Epoch 13 / 20) train acc: 0.735000; val_acc: 0.544000
(Iteration 1241 / 1900) loss: 0.895901
(Iteration 1251 / 1900) loss: 0.919611
(Iteration 1261 / 1900) loss: 0.994390
(Iteration 1271 / 1900) loss: 0.908746
(Iteration 1281 / 1900) loss: 0.930375
(Iteration 1291 / 1900) loss: 0.917094
(Iteration 1301 / 1900) loss: 0.970004
(Iteration 1311 / 1900) loss: 0.834261
(Iteration 1321 / 1900) loss: 0.884473
(Epoch 14 / 20) train acc: 0.713000; val_acc: 0.538000
(Iteration 1331 / 1900) loss: 0.817056
(Iteration 1341 / 1900) loss: 0.892731
(Iteration 1351 / 1900) loss: 0.867058
(Iteration 1361 / 1900) loss: 0.974781
(Iteration 1371 / 1900) loss: 0.814841
```

```
(Iteration 1381 / 1900) loss: 0.879550
(Iteration 1391 / 1900) loss: 0.949036
(Iteration 1401 / 1900) loss: 0.876946
(Iteration 1411 / 1900) loss: 0.926380
(Iteration 1421 / 1900) loss: 0.908581
(Epoch 15 / 20) train acc: 0.744000; val_acc: 0.551000
(Iteration 1431 / 1900) loss: 0.908358
(Iteration 1441 / 1900) loss: 0.887426
(Iteration 1451 / 1900) loss: 0.883570
(Iteration 1461 / 1900) loss: 0.884966
(Iteration 1471 / 1900) loss: 0.925161
(Iteration 1481 / 1900) loss: 0.841461
(Iteration 1491 / 1900) loss: 0.903132
(Iteration 1501 / 1900) loss: 0.882438
(Iteration 1511 / 1900) loss: 0.860067
(Epoch 16 / 20) train acc: 0.753000; val_acc: 0.550000
(Iteration 1521 / 1900) loss: 0.795113
(Iteration 1531 / 1900) loss: 0.875673
(Iteration 1541 / 1900) loss: 0.883972
(Iteration 1551 / 1900) loss: 0.846631
(Iteration 1561 / 1900) loss: 0.902729
(Iteration 1571 / 1900) loss: 0.842052
(Iteration 1581 / 1900) loss: 0.957303
(Iteration 1591 / 1900) loss: 0.857179
(Iteration 1601 / 1900) loss: 0.782685
(Iteration 1611 / 1900) loss: 0.844959
(Epoch 17 / 20) train acc: 0.783000; val_acc: 0.550000
(Iteration 1621 / 1900) loss: 0.882095
(Iteration 1631 / 1900) loss: 0.793881
(Iteration 1641 / 1900) loss: 0.797067
(Iteration 1651 / 1900) loss: 0.834389
(Iteration 1661 / 1900) loss: 0.734212
(Iteration 1671 / 1900) loss: 0.828244
(Iteration 1681 / 1900) loss: 0.850664
(Iteration 1691 / 1900) loss: 0.872663
(Iteration 1701 / 1900) loss: 0.728069
(Epoch 18 / 20) train acc: 0.787000; val_acc: 0.536000
(Iteration 1711 / 1900) loss: 0.810116
(Iteration 1721 / 1900) loss: 0.859068
(Iteration 1731 / 1900) loss: 0.805359
(Iteration 1741 / 1900) loss: 0.835653
(Iteration 1751 / 1900) loss: 0.862900
(Iteration 1761 / 1900) loss: 0.895476
(Iteration 1771 / 1900) loss: 0.746012
(Iteration 1781 / 1900) loss: 0.780350
(Iteration 1791 / 1900) loss: 0.855537
(Iteration 1801 / 1900) loss: 0.766571
(Epoch 19 / 20) train acc: 0.755000; val_acc: 0.544000
```

```
(Iteration 1811 / 1900) loss: 0.771078
(Iteration 1821 / 1900) loss: 0.758423
(Iteration 1831 / 1900) loss: 0.802048
(Iteration 1841 / 1900) loss: 0.798413
(Iteration 1851 / 1900) loss: 0.793603
(Iteration 1861 / 1900) loss: 0.858658
(Iteration 1871 / 1900) loss: 0.778333
(Iteration 1881 / 1900) loss: 0.860269
(Iteration 1891 / 1900) loss: 0.816228
(Epoch 20 / 20) train acc: 0.803000; val_acc: 0.547000
(Iteration 1 / 1900) loss: 2.303141
(Epoch 0 / 20) train acc: 0.142000; val_acc: 0.137000
(Iteration 11 / 1900) loss: 2.142402
(Iteration 21 / 1900) loss: 1.963843
(Iteration 31 / 1900) loss: 1.898340
(Iteration 41 / 1900) loss: 1.860348
(Iteration 51 / 1900) loss: 1.749660
(Iteration 61 / 1900) loss: 1.739788
(Iteration 71 / 1900) loss: 1.607751
(Iteration 81 / 1900) loss: 1.598171
(Iteration 91 / 1900) loss: 1.597188
(Epoch 1 / 20) train acc: 0.459000; val_acc: 0.436000
(Iteration 101 / 1900) loss: 1.527553
(Iteration 111 / 1900) loss: 1.566638
(Iteration 121 / 1900) loss: 1.460390
(Iteration 131 / 1900) loss: 1.527086
(Iteration 141 / 1900) loss: 1.409497
(Iteration 151 / 1900) loss: 1.403419
(Iteration 161 / 1900) loss: 1.349115
(Iteration 171 / 1900) loss: 1.371267
(Iteration 181 / 1900) loss: 1.424593
(Epoch 2 / 20) train acc: 0.516000; val_acc: 0.492000
(Iteration 191 / 1900) loss: 1.290901
(Iteration 201 / 1900) loss: 1.333718
(Iteration 211 / 1900) loss: 1.315536
(Iteration 221 / 1900) loss: 1.262162
(Iteration 231 / 1900) loss: 1.325322
(Iteration 241 / 1900) loss: 1.317059
(Iteration 251 / 1900) loss: 1.307147
(Iteration 261 / 1900) loss: 1.293247
(Iteration 271 / 1900) loss: 1.286405
(Iteration 281 / 1900) loss: 1.267202
(Epoch 3 / 20) train acc: 0.563000; val_acc: 0.523000
(Iteration 291 / 1900) loss: 1.312510
(Iteration 301 / 1900) loss: 1.187287
(Iteration 311 / 1900) loss: 1.210509
(Iteration 321 / 1900) loss: 1.271286
(Iteration 331 / 1900) loss: 1.183499
```

```
(Iteration 341 / 1900) loss: 1.222414
(Iteration 351 / 1900) loss: 1.162317
(Iteration 361 / 1900) loss: 1.126661
(Iteration 371 / 1900) loss: 1.218901
(Epoch 4 / 20) train acc: 0.568000; val_acc: 0.547000
(Iteration 381 / 1900) loss: 1.144188
(Iteration 391 / 1900) loss: 1.107409
(Iteration 401 / 1900) loss: 1.099427
(Iteration 411 / 1900) loss: 1.144264
(Iteration 421 / 1900) loss: 1.120113
(Iteration 431 / 1900) loss: 1.220403
(Iteration 441 / 1900) loss: 1.138269
(Iteration 451 / 1900) loss: 1.077855
(Iteration 461 / 1900) loss: 1.132523
(Iteration 471 / 1900) loss: 1.182395
(Epoch 5 / 20) train acc: 0.632000; val_acc: 0.547000
(Iteration 481 / 1900) loss: 1.144798
(Iteration 491 / 1900) loss: 1.118818
(Iteration 501 / 1900) loss: 0.990487
(Iteration 511 / 1900) loss: 1.118230
(Iteration 521 / 1900) loss: 1.117275
(Iteration 531 / 1900) loss: 1.110554
(Iteration 541 / 1900) loss: 1.092253
(Iteration 551 / 1900) loss: 1.071429
(Iteration 561 / 1900) loss: 1.106131
(Epoch 6 / 20) train acc: 0.637000; val_acc: 0.545000
(Iteration 571 / 1900) loss: 1.102273
(Iteration 581 / 1900) loss: 1.118218
(Iteration 591 / 1900) loss: 1.021900
(Iteration 601 / 1900) loss: 0.972019
(Iteration 611 / 1900) loss: 1.114268
(Iteration 621 / 1900) loss: 1.084674
(Iteration 631 / 1900) loss: 0.995870
(Iteration 641 / 1900) loss: 0.936091
(Iteration 651 / 1900) loss: 1.073814
(Iteration 661 / 1900) loss: 0.998684
(Epoch 7 / 20) train acc: 0.671000; val_acc: 0.522000
(Iteration 671 / 1900) loss: 0.992342
(Iteration 681 / 1900) loss: 1.059244
(Iteration 691 / 1900) loss: 0.942970
(Iteration 701 / 1900) loss: 1.022551
(Iteration 711 / 1900) loss: 0.977815
(Iteration 721 / 1900) loss: 0.935570
(Iteration 731 / 1900) loss: 0.967712
(Iteration 741 / 1900) loss: 1.000560
(Iteration 751 / 1900) loss: 0.984624
(Epoch 8 / 20) train acc: 0.683000; val_acc: 0.560000
(Iteration 761 / 1900) loss: 0.955683
```

```
(Iteration 771 / 1900) loss: 0.981935
(Iteration 781 / 1900) loss: 0.931488
(Iteration 791 / 1900) loss: 0.908193
(Iteration 801 / 1900) loss: 0.961302
(Iteration 811 / 1900) loss: 0.970520
(Iteration 821 / 1900) loss: 0.880088
(Iteration 831 / 1900) loss: 0.935401
(Iteration 841 / 1900) loss: 0.847116
(Iteration 851 / 1900) loss: 0.908396
(Epoch 9 / 20) train acc: 0.718000; val_acc: 0.560000
(Iteration 861 / 1900) loss: 0.945351
(Iteration 871 / 1900) loss: 0.854081
(Iteration 881 / 1900) loss: 0.885200
(Iteration 891 / 1900) loss: 0.883153
(Iteration 901 / 1900) loss: 0.876339
(Iteration 911 / 1900) loss: 0.863611
(Iteration 921 / 1900) loss: 0.899826
(Iteration 931 / 1900) loss: 0.894062
(Iteration 941 / 1900) loss: 0.915458
(Epoch 10 / 20) train acc: 0.706000; val_acc: 0.553000
(Iteration 951 / 1900) loss: 0.951042
(Iteration 961 / 1900) loss: 0.920553
(Iteration 971 / 1900) loss: 0.827172
(Iteration 981 / 1900) loss: 0.940744
(Iteration 991 / 1900) loss: 0.804757
(Iteration 1001 / 1900) loss: 0.781492
(Iteration 1011 / 1900) loss: 0.855325
(Iteration 1021 / 1900) loss: 0.835728
(Iteration 1031 / 1900) loss: 0.835912
(Iteration 1041 / 1900) loss: 0.824187
(Epoch 11 / 20) train acc: 0.737000; val_acc: 0.558000
(Iteration 1051 / 1900) loss: 0.782935
(Iteration 1061 / 1900) loss: 0.838572
(Iteration 1071 / 1900) loss: 0.891854
(Iteration 1081 / 1900) loss: 0.832974
(Iteration 1091 / 1900) loss: 0.840820
(Iteration 1101 / 1900) loss: 0.754859
(Iteration 1111 / 1900) loss: 0.851746
(Iteration 1121 / 1900) loss: 0.715082
(Iteration 1131 / 1900) loss: 0.809169
(Epoch 12 / 20) train acc: 0.719000; val_acc: 0.559000
(Iteration 1141 / 1900) loss: 0.796427
(Iteration 1151 / 1900) loss: 0.706918
(Iteration 1161 / 1900) loss: 0.818531
(Iteration 1171 / 1900) loss: 0.789685
(Iteration 1181 / 1900) loss: 0.713487
(Iteration 1191 / 1900) loss: 0.842048
(Iteration 1201 / 1900) loss: 0.786524
```

```
(Iteration 1211 / 1900) loss: 0.739856
(Iteration 1221 / 1900) loss: 0.778198
(Iteration 1231 / 1900) loss: 0.742683
(Epoch 13 / 20) train acc: 0.749000; val_acc: 0.541000
(Iteration 1241 / 1900) loss: 0.808069
(Iteration 1251 / 1900) loss: 0.687195
(Iteration 1261 / 1900) loss: 0.753800
(Iteration 1271 / 1900) loss: 0.698927
(Iteration 1281 / 1900) loss: 0.684812
(Iteration 1291 / 1900) loss: 0.694943
(Iteration 1301 / 1900) loss: 0.732437
(Iteration 1311 / 1900) loss: 0.722650
(Iteration 1321 / 1900) loss: 0.700000
(Epoch 14 / 20) train acc: 0.763000; val_acc: 0.542000
(Iteration 1331 / 1900) loss: 0.672218
(Iteration 1341 / 1900) loss: 0.771742
(Iteration 1351 / 1900) loss: 0.640784
(Iteration 1361 / 1900) loss: 0.641482
(Iteration 1371 / 1900) loss: 0.659409
(Iteration 1381 / 1900) loss: 0.659238
(Iteration 1391 / 1900) loss: 0.726246
(Iteration 1401 / 1900) loss: 0.666224
(Iteration 1411 / 1900) loss: 0.585844
(Iteration 1421 / 1900) loss: 0.654231
(Epoch 15 / 20) train acc: 0.797000; val_acc: 0.551000
(Iteration 1431 / 1900) loss: 0.580598
(Iteration 1441 / 1900) loss: 0.645904
(Iteration 1451 / 1900) loss: 0.623497
(Iteration 1461 / 1900) loss: 0.689772
(Iteration 1471 / 1900) loss: 0.566135
(Iteration 1481 / 1900) loss: 0.639157
(Iteration 1491 / 1900) loss: 0.574174
(Iteration 1501 / 1900) loss: 0.620997
(Iteration 1511 / 1900) loss: 0.685608
(Epoch 16 / 20) train acc: 0.816000; val_acc: 0.570000
(Iteration 1521 / 1900) loss: 0.601607
(Iteration 1531 / 1900) loss: 0.613583
(Iteration 1541 / 1900) loss: 0.665469
(Iteration 1551 / 1900) loss: 0.661049
(Iteration 1561 / 1900) loss: 0.676338
(Iteration 1571 / 1900) loss: 0.559207
(Iteration 1581 / 1900) loss: 0.578642
(Iteration 1591 / 1900) loss: 0.676210
(Iteration 1601 / 1900) loss: 0.627576
(Iteration 1611 / 1900) loss: 0.605198
(Epoch 17 / 20) train acc: 0.798000; val_acc: 0.563000
(Iteration 1621 / 1900) loss: 0.682779
(Iteration 1631 / 1900) loss: 0.573960
```

```
(Iteration 1641 / 1900) loss: 0.562094
(Iteration 1651 / 1900) loss: 0.610706
(Iteration 1661 / 1900) loss: 0.518476
(Iteration 1671 / 1900) loss: 0.591671
(Iteration 1681 / 1900) loss: 0.691511
(Iteration 1691 / 1900) loss: 0.575464
(Iteration 1701 / 1900) loss: 0.540666
(Epoch 18 / 20) train acc: 0.812000; val_acc: 0.553000
(Iteration 1711 / 1900) loss: 0.577172
(Iteration 1721 / 1900) loss: 0.546611
(Iteration 1731 / 1900) loss: 0.589899
(Iteration 1741 / 1900) loss: 0.589895
(Iteration 1751 / 1900) loss: 0.658333
(Iteration 1761 / 1900) loss: 0.542745
(Iteration 1771 / 1900) loss: 0.518292
(Iteration 1781 / 1900) loss: 0.561316
(Iteration 1791 / 1900) loss: 0.560604
(Iteration 1801 / 1900) loss: 0.478967
(Epoch 19 / 20) train acc: 0.846000; val_acc: 0.569000
(Iteration 1811 / 1900) loss: 0.539106
(Iteration 1821 / 1900) loss: 0.516781
(Iteration 1831 / 1900) loss: 0.499266
(Iteration 1841 / 1900) loss: 0.490641
(Iteration 1851 / 1900) loss: 0.556361
(Iteration 1861 / 1900) loss: 0.495132
(Iteration 1871 / 1900) loss: 0.547233
(Iteration 1881 / 1900) loss: 0.523273
(Iteration 1891 / 1900) loss: 0.583023
(Epoch 20 / 20) train acc: 0.824000; val_acc: 0.542000
(Iteration 1 / 1900) loss: 2.679781
(Epoch 0 / 20) train acc: 0.139000; val_acc: 0.155000
(Iteration 11 / 1900) loss: 2.598909
(Iteration 21 / 1900) loss: 2.523131
(Iteration 31 / 1900) loss: 2.455244
(Iteration 41 / 1900) loss: 2.399026
(Iteration 51 / 1900) loss: 2.353258
(Iteration 61 / 1900) loss: 2.318121
(Iteration 71 / 1900) loss: 2.270293
(Iteration 81 / 1900) loss: 2.239371
(Iteration 91 / 1900) loss: 2.208114
(Epoch 1 / 20) train acc: 0.415000; val_acc: 0.456000
(Iteration 101 / 1900) loss: 2.201106
(Iteration 111 / 1900) loss: 2.157673
(Iteration 121 / 1900) loss: 2.136703
(Iteration 131 / 1900) loss: 2.117368
(Iteration 141 / 1900) loss: 2.092777
(Iteration 151 / 1900) loss: 2.074955
(Iteration 161 / 1900) loss: 2.040712
```

```
(Iteration 171 / 1900) loss: 2.041535
(Iteration 181 / 1900) loss: 2.040134
(Epoch 2 / 20) train acc: 0.474000; val_acc: 0.460000
(Iteration 191 / 1900) loss: 2.014575
(Iteration 201 / 1900) loss: 1.990447
(Iteration 211 / 1900) loss: 1.986705
(Iteration 221 / 1900) loss: 1.929258
(Iteration 231 / 1900) loss: 1.938620
(Iteration 241 / 1900) loss: 1.920123
(Iteration 251 / 1900) loss: 1.944448
(Iteration 261 / 1900) loss: 1.864776
(Iteration 271 / 1900) loss: 1.898972
(Iteration 281 / 1900) loss: 1.865606
(Epoch 3 / 20) train acc: 0.486000; val_acc: 0.486000
(Iteration 291 / 1900) loss: 1.867159
(Iteration 301 / 1900) loss: 1.848319
(Iteration 311 / 1900) loss: 1.819529
(Iteration 321 / 1900) loss: 1.840692
(Iteration 331 / 1900) loss: 1.839407
(Iteration 341 / 1900) loss: 1.793087
(Iteration 351 / 1900) loss: 1.796001
(Iteration 361 / 1900) loss: 1.771147
(Iteration 371 / 1900) loss: 1.775079
(Epoch 4 / 20) train acc: 0.521000; val_acc: 0.496000
(Iteration 381 / 1900) loss: 1.760350
(Iteration 391 / 1900) loss: 1.739676
(Iteration 401 / 1900) loss: 1.718595
(Iteration 411 / 1900) loss: 1.749375
(Iteration 421 / 1900) loss: 1.700401
(Iteration 431 / 1900) loss: 1.745350
(Iteration 441 / 1900) loss: 1.706972
(Iteration 451 / 1900) loss: 1.729179
(Iteration 461 / 1900) loss: 1.677562
(Iteration 471 / 1900) loss: 1.673258
(Epoch 5 / 20) train acc: 0.507000; val_acc: 0.501000
(Iteration 481 / 1900) loss: 1.696017
(Iteration 491 / 1900) loss: 1.671546
(Iteration 501 / 1900) loss: 1.689225
(Iteration 511 / 1900) loss: 1.662690
(Iteration 521 / 1900) loss: 1.624106
(Iteration 531 / 1900) loss: 1.645137
(Iteration 541 / 1900) loss: 1.625478
(Iteration 551 / 1900) loss: 1.574268
(Iteration 561 / 1900) loss: 1.597252
(Epoch 6 / 20) train acc: 0.550000; val_acc: 0.499000
(Iteration 571 / 1900) loss: 1.621352
(Iteration 581 / 1900) loss: 1.635930
(Iteration 591 / 1900) loss: 1.591375
```

```
(Iteration 601 / 1900) loss: 1.598565
(Iteration 611 / 1900) loss: 1.554132
(Iteration 621 / 1900) loss: 1.549681
(Iteration 631 / 1900) loss: 1.573607
(Iteration 641 / 1900) loss: 1.593157
(Iteration 651 / 1900) loss: 1.580202
(Iteration 661 / 1900) loss: 1.568218
(Epoch 7 / 20) train acc: 0.552000; val_acc: 0.532000
(Iteration 671 / 1900) loss: 1.548334
(Iteration 681 / 1900) loss: 1.544327
(Iteration 691 / 1900) loss: 1.545940
(Iteration 701 / 1900) loss: 1.559719
(Iteration 711 / 1900) loss: 1.496407
(Iteration 721 / 1900) loss: 1.509909
(Iteration 731 / 1900) loss: 1.506477
(Iteration 741 / 1900) loss: 1.481289
(Iteration 751 / 1900) loss: 1.486350
(Epoch 8 / 20) train acc: 0.572000; val_acc: 0.499000
(Iteration 761 / 1900) loss: 1.460541
(Iteration 771 / 1900) loss: 1.519152
(Iteration 781 / 1900) loss: 1.470225
(Iteration 791 / 1900) loss: 1.478250
(Iteration 801 / 1900) loss: 1.463486
(Iteration 811 / 1900) loss: 1.492405
(Iteration 821 / 1900) loss: 1.407440
(Iteration 831 / 1900) loss: 1.489378
(Iteration 841 / 1900) loss: 1.486003
(Iteration 851 / 1900) loss: 1.454323
(Epoch 9 / 20) train acc: 0.587000; val_acc: 0.532000
(Iteration 861 / 1900) loss: 1.403332
(Iteration 871 / 1900) loss: 1.470482
(Iteration 881 / 1900) loss: 1.405728
(Iteration 891 / 1900) loss: 1.421881
(Iteration 901 / 1900) loss: 1.379688
(Iteration 911 / 1900) loss: 1.434885
(Iteration 921 / 1900) loss: 1.404273
(Iteration 931 / 1900) loss: 1.408626
(Iteration 941 / 1900) loss: 1.443806
(Epoch 10 / 20) train acc: 0.598000; val_acc: 0.520000
(Iteration 951 / 1900) loss: 1.417482
(Iteration 961 / 1900) loss: 1.391118
(Iteration 971 / 1900) loss: 1.349459
(Iteration 981 / 1900) loss: 1.351497
(Iteration 991 / 1900) loss: 1.353230
(Iteration 1001 / 1900) loss: 1.327629
(Iteration 1011 / 1900) loss: 1.348993
(Iteration 1021 / 1900) loss: 1.400586
(Iteration 1031 / 1900) loss: 1.363507
```

```
(Iteration 1041 / 1900) loss: 1.364038
(Epoch 11 / 20) train acc: 0.620000; val_acc: 0.538000
(Iteration 1051 / 1900) loss: 1.368786
(Iteration 1061 / 1900) loss: 1.289190
(Iteration 1071 / 1900) loss: 1.393054
(Iteration 1081 / 1900) loss: 1.341972
(Iteration 1091 / 1900) loss: 1.320756
(Iteration 1101 / 1900) loss: 1.336520
(Iteration 1111 / 1900) loss: 1.315813
(Iteration 1121 / 1900) loss: 1.345877
(Iteration 1131 / 1900) loss: 1.370755
(Epoch 12 / 20) train acc: 0.613000; val_acc: 0.541000
(Iteration 1141 / 1900) loss: 1.272993
(Iteration 1151 / 1900) loss: 1.261062
(Iteration 1161 / 1900) loss: 1.315892
(Iteration 1171 / 1900) loss: 1.276232
(Iteration 1181 / 1900) loss: 1.258674
(Iteration 1191 / 1900) loss: 1.286957
(Iteration 1201 / 1900) loss: 1.285952
(Iteration 1211 / 1900) loss: 1.357017
(Iteration 1221 / 1900) loss: 1.236623
(Iteration 1231 / 1900) loss: 1.284390
(Epoch 13 / 20) train acc: 0.637000; val_acc: 0.551000
(Iteration 1241 / 1900) loss: 1.292620
(Iteration 1251 / 1900) loss: 1.318201
(Iteration 1261 / 1900) loss: 1.270116
(Iteration 1271 / 1900) loss: 1.280003
(Iteration 1281 / 1900) loss: 1.240701
(Iteration 1291 / 1900) loss: 1.230456
(Iteration 1301 / 1900) loss: 1.222010
(Iteration 1311 / 1900) loss: 1.282738
(Iteration 1321 / 1900) loss: 1.244641
(Epoch 14 / 20) train acc: 0.642000; val_acc: 0.540000
(Iteration 1331 / 1900) loss: 1.282711
(Iteration 1341 / 1900) loss: 1.314918
(Iteration 1351 / 1900) loss: 1.280175
(Iteration 1361 / 1900) loss: 1.256001
(Iteration 1371 / 1900) loss: 1.206422
(Iteration 1381 / 1900) loss: 1.203706
(Iteration 1391 / 1900) loss: 1.207029
(Iteration 1401 / 1900) loss: 1.213457
(Iteration 1411 / 1900) loss: 1.269427
(Iteration 1421 / 1900) loss: 1.216774
(Epoch 15 / 20) train acc: 0.640000; val_acc: 0.540000
(Iteration 1431 / 1900) loss: 1.227508
(Iteration 1441 / 1900) loss: 1.119120
(Iteration 1451 / 1900) loss: 1.227319
(Iteration 1461 / 1900) loss: 1.193474
```

```
(Iteration 1471 / 1900) loss: 1.239609
(Iteration 1481 / 1900) loss: 1.208070
(Iteration 1491 / 1900) loss: 1.176096
(Iteration 1501 / 1900) loss: 1.199960
(Iteration 1511 / 1900) loss: 1.177379
(Epoch 16 / 20) train acc: 0.682000; val_acc: 0.553000
(Iteration 1521 / 1900) loss: 1.145354
(Iteration 1531 / 1900) loss: 1.253106
(Iteration 1541 / 1900) loss: 1.276332
(Iteration 1551 / 1900) loss: 1.222579
(Iteration 1561 / 1900) loss: 1.222752
(Iteration 1571 / 1900) loss: 1.158214
(Iteration 1581 / 1900) loss: 1.213388
(Iteration 1591 / 1900) loss: 1.216572
(Iteration 1601 / 1900) loss: 1.192618
(Iteration 1611 / 1900) loss: 1.229232
(Epoch 17 / 20) train acc: 0.670000; val_acc: 0.540000
(Iteration 1621 / 1900) loss: 1.145551
(Iteration 1631 / 1900) loss: 1.116405
(Iteration 1641 / 1900) loss: 1.146267
(Iteration 1651 / 1900) loss: 1.092320
(Iteration 1661 / 1900) loss: 1.179374
(Iteration 1671 / 1900) loss: 1.229932
(Iteration 1681 / 1900) loss: 1.210839
(Iteration 1691 / 1900) loss: 1.192182
(Iteration 1701 / 1900) loss: 1.173031
(Epoch 18 / 20) train acc: 0.709000; val_acc: 0.552000
(Iteration 1711 / 1900) loss: 1.126226
(Iteration 1721 / 1900) loss: 1.164059
(Iteration 1731 / 1900) loss: 1.126594
(Iteration 1741 / 1900) loss: 1.106313
(Iteration 1751 / 1900) loss: 1.176081
(Iteration 1761 / 1900) loss: 1.135287
(Iteration 1771 / 1900) loss: 1.154911
(Iteration 1781 / 1900) loss: 1.165737
(Iteration 1791 / 1900) loss: 1.156055
(Iteration 1801 / 1900) loss: 1.132363
(Epoch 19 / 20) train acc: 0.676000; val_acc: 0.568000
(Iteration 1811 / 1900) loss: 1.055133
(Iteration 1821 / 1900) loss: 1.095924
(Iteration 1831 / 1900) loss: 1.125380
(Iteration 1841 / 1900) loss: 1.108094
(Iteration 1851 / 1900) loss: 1.224981
(Iteration 1861 / 1900) loss: 1.148930
(Iteration 1871 / 1900) loss: 1.131108
(Iteration 1881 / 1900) loss: 1.143380
(Iteration 1891 / 1900) loss: 1.107588
(Epoch 20 / 20) train acc: 0.694000; val_acc: 0.559000
```

```
(Iteration 1 / 1900) loss: 2.378599
(Epoch 0 / 20) train acc: 0.147000; val_acc: 0.146000
(Iteration 11 / 1900) loss: 2.339163
(Iteration 21 / 1900) loss: 2.312565
(Iteration 31 / 1900) loss: 2.291681
(Iteration 41 / 1900) loss: 2.268106
(Iteration 51 / 1900) loss: 2.236292
(Iteration 61 / 1900) loss: 2.224882
(Iteration 71 / 1900) loss: 2.197369
(Iteration 81 / 1900) loss: 2.177394
(Iteration 91 / 1900) loss: 2.154441
(Epoch 1 / 20) train acc: 0.452000; val_acc: 0.437000
(Iteration 101 / 1900) loss: 2.148050
(Iteration 111 / 1900) loss: 2.127269
(Iteration 121 / 1900) loss: 2.098648
(Iteration 131 / 1900) loss: 2.080883
(Iteration 141 / 1900) loss: 2.035776
(Iteration 151 / 1900) loss: 2.025290
(Iteration 161 / 1900) loss: 2.030404
(Iteration 171 / 1900) loss: 2.012366
(Iteration 181 / 1900) loss: 1.995238
(Epoch 2 / 20) train acc: 0.479000; val_acc: 0.451000
(Iteration 191 / 1900) loss: 1.971443
(Iteration 201 / 1900) loss: 1.964018
(Iteration 211 / 1900) loss: 1.934371
(Iteration 221 / 1900) loss: 1.918791
(Iteration 231 / 1900) loss: 1.904985
(Iteration 241 / 1900) loss: 1.887422
(Iteration 251 / 1900) loss: 1.898564
(Iteration 261 / 1900) loss: 1.876002
(Iteration 271 / 1900) loss: 1.868766
(Iteration 281 / 1900) loss: 1.874394
(Epoch 3 / 20) train acc: 0.478000; val_acc: 0.468000
(Iteration 291 / 1900) loss: 1.829856
(Iteration 301 / 1900) loss: 1.825136
(Iteration 311 / 1900) loss: 1.815059
(Iteration 321 / 1900) loss: 1.786833
(Iteration 331 / 1900) loss: 1.754689
(Iteration 341 / 1900) loss: 1.776859
(Iteration 351 / 1900) loss: 1.743267
(Iteration 361 / 1900) loss: 1.764510
(Iteration 371 / 1900) loss: 1.736920
(Epoch 4 / 20) train acc: 0.498000; val_acc: 0.495000
(Iteration 381 / 1900) loss: 1.726650
(Iteration 391 / 1900) loss: 1.698713
(Iteration 401 / 1900) loss: 1.708141
(Iteration 411 / 1900) loss: 1.713471
(Iteration 421 / 1900) loss: 1.656377
```

```
(Iteration 431 / 1900) loss: 1.701891
(Iteration 441 / 1900) loss: 1.666433
(Iteration 451 / 1900) loss: 1.677001
(Iteration 461 / 1900) loss: 1.627623
(Iteration 471 / 1900) loss: 1.624398
(Epoch 5 / 20) train acc: 0.547000; val_acc: 0.504000
(Iteration 481 / 1900) loss: 1.641391
(Iteration 491 / 1900) loss: 1.617137
(Iteration 501 / 1900) loss: 1.638582
(Iteration 511 / 1900) loss: 1.597575
(Iteration 521 / 1900) loss: 1.571855
(Iteration 531 / 1900) loss: 1.582111
(Iteration 541 / 1900) loss: 1.568013
(Iteration 551 / 1900) loss: 1.538349
(Iteration 561 / 1900) loss: 1.602670
(Epoch 6 / 20) train acc: 0.569000; val_acc: 0.504000
(Iteration 571 / 1900) loss: 1.582820
(Iteration 581 / 1900) loss: 1.508789
(Iteration 591 / 1900) loss: 1.514420
(Iteration 601 / 1900) loss: 1.503613
(Iteration 611 / 1900) loss: 1.496958
(Iteration 621 / 1900) loss: 1.414003
(Iteration 631 / 1900) loss: 1.505719
(Iteration 641 / 1900) loss: 1.430599
(Iteration 651 / 1900) loss: 1.480491
(Iteration 661 / 1900) loss: 1.470797
(Epoch 7 / 20) train acc: 0.591000; val_acc: 0.526000
(Iteration 671 / 1900) loss: 1.476491
(Iteration 681 / 1900) loss: 1.401684
(Iteration 691 / 1900) loss: 1.447825
(Iteration 701 / 1900) loss: 1.392097
(Iteration 711 / 1900) loss: 1.400882
(Iteration 721 / 1900) loss: 1.379265
(Iteration 731 / 1900) loss: 1.391485
(Iteration 741 / 1900) loss: 1.408032
(Iteration 751 / 1900) loss: 1.358250
(Epoch 8 / 20) train acc: 0.610000; val_acc: 0.527000
(Iteration 761 / 1900) loss: 1.373399
(Iteration 771 / 1900) loss: 1.374101
(Iteration 781 / 1900) loss: 1.378699
(Iteration 791 / 1900) loss: 1.338715
(Iteration 801 / 1900) loss: 1.342200
(Iteration 811 / 1900) loss: 1.329110
(Iteration 821 / 1900) loss: 1.333317
(Iteration 831 / 1900) loss: 1.341599
(Iteration 841 / 1900) loss: 1.274364
(Iteration 851 / 1900) loss: 1.330996
(Epoch 9 / 20) train acc: 0.625000; val_acc: 0.532000
```

```
(Iteration 861 / 1900) loss: 1.287570
(Iteration 871 / 1900) loss: 1.337265
(Iteration 881 / 1900) loss: 1.334633
(Iteration 891 / 1900) loss: 1.302434
(Iteration 901 / 1900) loss: 1.272868
(Iteration 911 / 1900) loss: 1.324100
(Iteration 921 / 1900) loss: 1.311282
(Iteration 931 / 1900) loss: 1.293519
(Iteration 941 / 1900) loss: 1.304525
(Epoch 10 / 20) train acc: 0.608000; val_acc: 0.514000
(Iteration 951 / 1900) loss: 1.306259
(Iteration 961 / 1900) loss: 1.257145
(Iteration 971 / 1900) loss: 1.274514
(Iteration 981 / 1900) loss: 1.243965
(Iteration 991 / 1900) loss: 1.264146
(Iteration 1001 / 1900) loss: 1.202746
(Iteration 1011 / 1900) loss: 1.271420
(Iteration 1021 / 1900) loss: 1.208805
(Iteration 1031 / 1900) loss: 1.249602
(Iteration 1041 / 1900) loss: 1.195724
(Epoch 11 / 20) train acc: 0.650000; val_acc: 0.545000
(Iteration 1051 / 1900) loss: 1.181056
(Iteration 1061 / 1900) loss: 1.167650
(Iteration 1071 / 1900) loss: 1.206827
(Iteration 1081 / 1900) loss: 1.213388
(Iteration 1091 / 1900) loss: 1.182964
(Iteration 1101 / 1900) loss: 1.169081
(Iteration 1111 / 1900) loss: 1.239663
(Iteration 1121 / 1900) loss: 1.166703
(Iteration 1131 / 1900) loss: 1.131109
(Epoch 12 / 20) train acc: 0.649000; val_acc: 0.543000
(Iteration 1141 / 1900) loss: 1.147276
(Iteration 1151 / 1900) loss: 1.156357
(Iteration 1161 / 1900) loss: 1.157648
(Iteration 1171 / 1900) loss: 1.160099
(Iteration 1181 / 1900) loss: 1.061403
(Iteration 1191 / 1900) loss: 1.088702
(Iteration 1201 / 1900) loss: 1.150816
(Iteration 1211 / 1900) loss: 1.051854
(Iteration 1221 / 1900) loss: 1.159795
(Iteration 1231 / 1900) loss: 1.099424
(Epoch 13 / 20) train acc: 0.649000; val_acc: 0.543000
(Iteration 1241 / 1900) loss: 1.108614
(Iteration 1251 / 1900) loss: 1.089311
(Iteration 1261 / 1900) loss: 1.104205
(Iteration 1271 / 1900) loss: 1.099833
(Iteration 1281 / 1900) loss: 1.092801
(Iteration 1291 / 1900) loss: 1.112306
```

```
(Iteration 1301 / 1900) loss: 1.053675
(Iteration 1311 / 1900) loss: 1.054691
(Iteration 1321 / 1900) loss: 1.059219
(Epoch 14 / 20) train acc: 0.683000; val_acc: 0.548000
(Iteration 1331 / 1900) loss: 1.050655
(Iteration 1341 / 1900) loss: 1.123278
(Iteration 1351 / 1900) loss: 1.024167
(Iteration 1361 / 1900) loss: 1.084438
(Iteration 1371 / 1900) loss: 1.054826
(Iteration 1381 / 1900) loss: 1.041334
(Iteration 1391 / 1900) loss: 1.091140
(Iteration 1401 / 1900) loss: 1.079631
(Iteration 1411 / 1900) loss: 0.946731
(Iteration 1421 / 1900) loss: 1.048010
(Epoch 15 / 20) train acc: 0.698000; val_acc: 0.544000
(Iteration 1431 / 1900) loss: 1.011082
(Iteration 1441 / 1900) loss: 1.047013
(Iteration 1451 / 1900) loss: 1.010690
(Iteration 1461 / 1900) loss: 1.056452
(Iteration 1471 / 1900) loss: 0.998290
(Iteration 1481 / 1900) loss: 1.029509
(Iteration 1491 / 1900) loss: 0.957017
(Iteration 1501 / 1900) loss: 0.998871
(Iteration 1511 / 1900) loss: 0.983828
(Epoch 16 / 20) train acc: 0.701000; val_acc: 0.538000
(Iteration 1521 / 1900) loss: 0.976055
(Iteration 1531 / 1900) loss: 0.974605
(Iteration 1541 / 1900) loss: 0.980718
(Iteration 1551 / 1900) loss: 0.935529
(Iteration 1561 / 1900) loss: 0.981785
(Iteration 1571 / 1900) loss: 0.983815
(Iteration 1581 / 1900) loss: 0.929818
(Iteration 1591 / 1900) loss: 0.962723
(Iteration 1601 / 1900) loss: 0.974301
(Iteration 1611 / 1900) loss: 0.959896
(Epoch 17 / 20) train acc: 0.744000; val_acc: 0.551000
(Iteration 1621 / 1900) loss: 0.936012
(Iteration 1631 / 1900) loss: 0.928579
(Iteration 1641 / 1900) loss: 0.987467
(Iteration 1651 / 1900) loss: 0.935675
(Iteration 1661 / 1900) loss: 0.956751
(Iteration 1671 / 1900) loss: 0.917617
(Iteration 1681 / 1900) loss: 0.986125
(Iteration 1691 / 1900) loss: 0.942763
(Iteration 1701 / 1900) loss: 0.905154
(Epoch 18 / 20) train acc: 0.732000; val_acc: 0.542000
(Iteration 1711 / 1900) loss: 0.908403
(Iteration 1721 / 1900) loss: 0.950810
```

```
(Iteration 1731 / 1900) loss: 0.870486
(Iteration 1741 / 1900) loss: 0.947199
(Iteration 1751 / 1900) loss: 0.909230
(Iteration 1761 / 1900) loss: 0.887863
(Iteration 1771 / 1900) loss: 0.892660
(Iteration 1781 / 1900) loss: 0.896172
(Iteration 1791 / 1900) loss: 0.848201
(Iteration 1801 / 1900) loss: 0.943240
(Epoch 19 / 20) train acc: 0.745000; val_acc: 0.558000
(Iteration 1811 / 1900) loss: 0.916464
(Iteration 1821 / 1900) loss: 0.889620
(Iteration 1831 / 1900) loss: 0.852270
(Iteration 1841 / 1900) loss: 0.926964
(Iteration 1851 / 1900) loss: 0.880551
(Iteration 1861 / 1900) loss: 0.899015
(Iteration 1871 / 1900) loss: 0.930443
(Iteration 1881 / 1900) loss: 0.918703
(Iteration 1891 / 1900) loss: 0.880839
(Epoch 20 / 20) train acc: 0.755000; val_acc: 0.534000
(Iteration 1 / 1900) loss: 2.323405
(Epoch 0 / 20) train acc: 0.174000; val_acc: 0.176000
(Iteration 11 / 1900) loss: 2.287481
(Iteration 21 / 1900) loss: 2.264516
(Iteration 31 / 1900) loss: 2.251116
(Iteration 41 / 1900) loss: 2.221915
(Iteration 51 / 1900) loss: 2.215367
(Iteration 61 / 1900) loss: 2.184376
(Iteration 71 / 1900) loss: 2.171407
(Iteration 81 / 1900) loss: 2.157244
(Iteration 91 / 1900) loss: 2.131769
(Epoch 1 / 20) train acc: 0.455000; val_acc: 0.438000
(Iteration 101 / 1900) loss: 2.102073
(Iteration 111 / 1900) loss: 2.082935
(Iteration 121 / 1900) loss: 2.064006
(Iteration 131 / 1900) loss: 2.083166
(Iteration 141 / 1900) loss: 2.040409
(Iteration 151 / 1900) loss: 2.009662
(Iteration 161 / 1900) loss: 1.996860
(Iteration 171 / 1900) loss: 1.969292
(Iteration 181 / 1900) loss: 1.974997
(Epoch 2 / 20) train acc: 0.460000; val_acc: 0.474000
(Iteration 191 / 1900) loss: 1.950059
(Iteration 201 / 1900) loss: 1.927537
(Iteration 211 / 1900) loss: 1.912141
(Iteration 221 / 1900) loss: 1.894273
(Iteration 231 / 1900) loss: 1.856574
(Iteration 241 / 1900) loss: 1.864706
(Iteration 251 / 1900) loss: 1.872263
```

```
(Iteration 261 / 1900) loss: 1.856942
(Iteration 271 / 1900) loss: 1.841051
(Iteration 281 / 1900) loss: 1.826070
(Epoch 3 / 20) train acc: 0.492000; val_acc: 0.482000
(Iteration 291 / 1900) loss: 1.781964
(Iteration 301 / 1900) loss: 1.799371
(Iteration 311 / 1900) loss: 1.773825
(Iteration 321 / 1900) loss: 1.760326
(Iteration 331 / 1900) loss: 1.747218
(Iteration 341 / 1900) loss: 1.761031
(Iteration 351 / 1900) loss: 1.719257
(Iteration 361 / 1900) loss: 1.727394
(Iteration 371 / 1900) loss: 1.666293
(Epoch 4 / 20) train acc: 0.545000; val_acc: 0.513000
(Iteration 381 / 1900) loss: 1.687915
(Iteration 391 / 1900) loss: 1.682975
(Iteration 401 / 1900) loss: 1.653260
(Iteration 411 / 1900) loss: 1.681620
(Iteration 421 / 1900) loss: 1.640661
(Iteration 431 / 1900) loss: 1.626625
(Iteration 441 / 1900) loss: 1.628888
(Iteration 451 / 1900) loss: 1.603715
(Iteration 461 / 1900) loss: 1.550471
(Iteration 471 / 1900) loss: 1.594869
(Epoch 5 / 20) train acc: 0.526000; val_acc: 0.508000
(Iteration 481 / 1900) loss: 1.572323
(Iteration 491 / 1900) loss: 1.550729
(Iteration 501 / 1900) loss: 1.560274
(Iteration 511 / 1900) loss: 1.506930
(Iteration 521 / 1900) loss: 1.516702
(Iteration 531 / 1900) loss: 1.501407
(Iteration 541 / 1900) loss: 1.531792
(Iteration 551 / 1900) loss: 1.501443
(Iteration 561 / 1900) loss: 1.541756
(Epoch 6 / 20) train acc: 0.612000; val_acc: 0.523000
(Iteration 571 / 1900) loss: 1.503611
(Iteration 581 / 1900) loss: 1.462364
(Iteration 591 / 1900) loss: 1.473889
(Iteration 601 / 1900) loss: 1.454293
(Iteration 611 / 1900) loss: 1.423773
(Iteration 621 / 1900) loss: 1.452833
(Iteration 631 / 1900) loss: 1.442378
(Iteration 641 / 1900) loss: 1.404085
(Iteration 651 / 1900) loss: 1.425614
(Iteration 661 / 1900) loss: 1.429150
(Epoch 7 / 20) train acc: 0.582000; val_acc: 0.524000
(Iteration 671 / 1900) loss: 1.370964
(Iteration 681 / 1900) loss: 1.421101
```

```
(Iteration 691 / 1900) loss: 1.370201
(Iteration 701 / 1900) loss: 1.418143
(Iteration 711 / 1900) loss: 1.329510
(Iteration 721 / 1900) loss: 1.314017
(Iteration 731 / 1900) loss: 1.376383
(Iteration 741 / 1900) loss: 1.344712
(Iteration 751 / 1900) loss: 1.295805
(Epoch 8 / 20) train acc: 0.604000; val_acc: 0.523000
(Iteration 761 / 1900) loss: 1.317491
(Iteration 771 / 1900) loss: 1.298304
(Iteration 781 / 1900) loss: 1.315867
(Iteration 791 / 1900) loss: 1.276681
(Iteration 801 / 1900) loss: 1.285431
(Iteration 811 / 1900) loss: 1.274088
(Iteration 821 / 1900) loss: 1.309612
(Iteration 831 / 1900) loss: 1.267051
(Iteration 841 / 1900) loss: 1.279884
(Iteration 851 / 1900) loss: 1.251927
(Epoch 9 / 20) train acc: 0.626000; val_acc: 0.544000
(Iteration 861 / 1900) loss: 1.301172
(Iteration 871 / 1900) loss: 1.290637
(Iteration 881 / 1900) loss: 1.188611
(Iteration 891 / 1900) loss: 1.205968
(Iteration 901 / 1900) loss: 1.274501
(Iteration 911 / 1900) loss: 1.218280
(Iteration 921 / 1900) loss: 1.188430
(Iteration 931 / 1900) loss: 1.235120
(Iteration 941 / 1900) loss: 1.243784
(Epoch 10 / 20) train acc: 0.628000; val_acc: 0.524000
(Iteration 951 / 1900) loss: 1.194777
(Iteration 961 / 1900) loss: 1.174186
(Iteration 971 / 1900) loss: 1.186924
(Iteration 981 / 1900) loss: 1.134272
(Iteration 991 / 1900) loss: 1.135396
(Iteration 1001 / 1900) loss: 1.141790
(Iteration 1011 / 1900) loss: 1.132285
(Iteration 1021 / 1900) loss: 1.122509
(Iteration 1031 / 1900) loss: 1.153425
(Iteration 1041 / 1900) loss: 1.199351
(Epoch 11 / 20) train acc: 0.675000; val_acc: 0.550000
(Iteration 1051 / 1900) loss: 1.174729
(Iteration 1061 / 1900) loss: 1.116839
(Iteration 1071 / 1900) loss: 1.103426
(Iteration 1081 / 1900) loss: 1.148867
(Iteration 1091 / 1900) loss: 1.128280
(Iteration 1101 / 1900) loss: 1.195504
(Iteration 1111 / 1900) loss: 1.079308
(Iteration 1121 / 1900) loss: 1.091251
```

```
(Iteration 1131 / 1900) loss: 1.072048
(Epoch 12 / 20) train acc: 0.681000; val_acc: 0.536000
(Iteration 1141 / 1900) loss: 1.103403
(Iteration 1151 / 1900) loss: 1.067730
(Iteration 1161 / 1900) loss: 1.013851
(Iteration 1171 / 1900) loss: 1.033213
(Iteration 1181 / 1900) loss: 1.054578
(Iteration 1191 / 1900) loss: 1.063069
(Iteration 1201 / 1900) loss: 1.001817
(Iteration 1211 / 1900) loss: 1.071128
(Iteration 1221 / 1900) loss: 1.117571
(Iteration 1231 / 1900) loss: 1.044116
(Epoch 13 / 20) train acc: 0.700000; val_acc: 0.544000
(Iteration 1241 / 1900) loss: 1.043434
(Iteration 1251 / 1900) loss: 0.986756
(Iteration 1261 / 1900) loss: 0.978575
(Iteration 1271 / 1900) loss: 1.049646
(Iteration 1281 / 1900) loss: 0.939525
(Iteration 1291 / 1900) loss: 1.030343
(Iteration 1301 / 1900) loss: 1.072205
(Iteration 1311 / 1900) loss: 0.934998
(Iteration 1321 / 1900) loss: 0.996522
(Epoch 14 / 20) train acc: 0.698000; val_acc: 0.531000
(Iteration 1331 / 1900) loss: 1.001878
(Iteration 1341 / 1900) loss: 0.959884
(Iteration 1351 / 1900) loss: 0.996070
(Iteration 1361 / 1900) loss: 0.950386
(Iteration 1371 / 1900) loss: 0.994458
(Iteration 1381 / 1900) loss: 0.920529
(Iteration 1391 / 1900) loss: 0.946726
(Iteration 1401 / 1900) loss: 0.909135
(Iteration 1411 / 1900) loss: 0.993239
(Iteration 1421 / 1900) loss: 0.916429
(Epoch 15 / 20) train acc: 0.740000; val_acc: 0.558000
(Iteration 1431 / 1900) loss: 0.939736
(Iteration 1441 / 1900) loss: 0.927904
(Iteration 1451 / 1900) loss: 0.976885
(Iteration 1461 / 1900) loss: 0.886780
(Iteration 1471 / 1900) loss: 0.906001
(Iteration 1481 / 1900) loss: 0.956948
(Iteration 1491 / 1900) loss: 0.954785
(Iteration 1501 / 1900) loss: 0.959026
(Iteration 1511 / 1900) loss: 0.910818
(Epoch 16 / 20) train acc: 0.747000; val_acc: 0.550000
(Iteration 1521 / 1900) loss: 0.890005
(Iteration 1531 / 1900) loss: 0.923613
(Iteration 1541 / 1900) loss: 0.885894
(Iteration 1551 / 1900) loss: 0.887148
```

```
(Iteration 1561 / 1900) loss: 0.880751
(Iteration 1571 / 1900) loss: 0.871160
(Iteration 1581 / 1900) loss: 0.912216
(Iteration 1591 / 1900) loss: 0.872436
(Iteration 1601 / 1900) loss: 0.837505
(Iteration 1611 / 1900) loss: 0.855552
(Epoch 17 / 20) train acc: 0.750000; val_acc: 0.540000
(Iteration 1621 / 1900) loss: 0.820045
(Iteration 1631 / 1900) loss: 0.872690
(Iteration 1641 / 1900) loss: 0.875584
(Iteration 1651 / 1900) loss: 0.847818
(Iteration 1661 / 1900) loss: 0.791119
(Iteration 1671 / 1900) loss: 0.829524
(Iteration 1681 / 1900) loss: 0.858009
(Iteration 1691 / 1900) loss: 0.879880
(Iteration 1701 / 1900) loss: 0.870266
(Epoch 18 / 20) train acc: 0.768000; val_acc: 0.540000
(Iteration 1711 / 1900) loss: 0.796801
(Iteration 1721 / 1900) loss: 0.795914
(Iteration 1731 / 1900) loss: 0.783584
(Iteration 1741 / 1900) loss: 0.778942
(Iteration 1751 / 1900) loss: 0.807498
(Iteration 1761 / 1900) loss: 0.855940
(Iteration 1771 / 1900) loss: 0.789176
(Iteration 1781 / 1900) loss: 0.773349
(Iteration 1791 / 1900) loss: 0.815183
(Iteration 1801 / 1900) loss: 0.826628
(Epoch 19 / 20) train acc: 0.757000; val_acc: 0.545000
(Iteration 1811 / 1900) loss: 0.744539
(Iteration 1821 / 1900) loss: 0.760733
(Iteration 1831 / 1900) loss: 0.814992
(Iteration 1841 / 1900) loss: 0.712208
(Iteration 1851 / 1900) loss: 0.716935
(Iteration 1861 / 1900) loss: 0.723517
(Iteration 1871 / 1900) loss: 0.739468
(Iteration 1881 / 1900) loss: 0.751679
(Iteration 1891 / 1900) loss: 0.737354
(Epoch 20 / 20) train acc: 0.784000; val_acc: 0.554000
(Iteration 1 / 1900) loss: 2.307742
(Epoch 0 / 20) train acc: 0.142000; val_acc: 0.132000
(Iteration 11 / 1900) loss: 2.271020
(Iteration 21 / 1900) loss: 2.246181
(Iteration 31 / 1900) loss: 2.233082
(Iteration 41 / 1900) loss: 2.214687
(Iteration 51 / 1900) loss: 2.185717
(Iteration 61 / 1900) loss: 2.165905
(Iteration 71 / 1900) loss: 2.160886
(Iteration 81 / 1900) loss: 2.131084
```

```
(Iteration 91 / 1900) loss: 2.131110
(Epoch 1 / 20) train acc: 0.440000; val_acc: 0.426000
(Iteration 101 / 1900) loss: 2.101989
(Iteration 111 / 1900) loss: 2.064582
(Iteration 121 / 1900) loss: 2.060547
(Iteration 131 / 1900) loss: 2.048577
(Iteration 141 / 1900) loss: 2.014226
(Iteration 151 / 1900) loss: 2.012808
(Iteration 161 / 1900) loss: 1.996238
(Iteration 171 / 1900) loss: 1.980336
(Iteration 181 / 1900) loss: 1.949191
(Epoch 2 / 20) train acc: 0.501000; val_acc: 0.463000
(Iteration 191 / 1900) loss: 1.950850
(Iteration 201 / 1900) loss: 1.914652
(Iteration 211 / 1900) loss: 1.909788
(Iteration 221 / 1900) loss: 1.880037
(Iteration 231 / 1900) loss: 1.882840
(Iteration 241 / 1900) loss: 1.859579
(Iteration 251 / 1900) loss: 1.848856
(Iteration 261 / 1900) loss: 1.836378
(Iteration 271 / 1900) loss: 1.839636
(Iteration 281 / 1900) loss: 1.793922
(Epoch 3 / 20) train acc: 0.495000; val_acc: 0.479000
(Iteration 291 / 1900) loss: 1.787127
(Iteration 301 / 1900) loss: 1.785990
(Iteration 311 / 1900) loss: 1.773495
(Iteration 321 / 1900) loss: 1.736862
(Iteration 331 / 1900) loss: 1.740968
(Iteration 341 / 1900) loss: 1.715915
(Iteration 351 / 1900) loss: 1.687913
(Iteration 361 / 1900) loss: 1.732665
(Iteration 371 / 1900) loss: 1.713227
(Epoch 4 / 20) train acc: 0.523000; val_acc: 0.507000
(Iteration 381 / 1900) loss: 1.696169
(Iteration 391 / 1900) loss: 1.645656
(Iteration 401 / 1900) loss: 1.655710
(Iteration 411 / 1900) loss: 1.654156
(Iteration 421 / 1900) loss: 1.643680
(Iteration 431 / 1900) loss: 1.632570
(Iteration 441 / 1900) loss: 1.576174
(Iteration 451 / 1900) loss: 1.610290
(Iteration 461 / 1900) loss: 1.590506
(Iteration 471 / 1900) loss: 1.564491
(Epoch 5 / 20) train acc: 0.575000; val_acc: 0.519000
(Iteration 481 / 1900) loss: 1.583403
(Iteration 491 / 1900) loss: 1.549505
(Iteration 501 / 1900) loss: 1.579428
(Iteration 511 / 1900) loss: 1.520038
```

```
(Iteration 521 / 1900) loss: 1.499072
(Iteration 531 / 1900) loss: 1.510060
(Iteration 541 / 1900) loss: 1.515431
(Iteration 551 / 1900) loss: 1.474099
(Iteration 561 / 1900) loss: 1.431445
(Epoch 6 / 20) train acc: 0.583000; val_acc: 0.528000
(Iteration 571 / 1900) loss: 1.455023
(Iteration 581 / 1900) loss: 1.457020
(Iteration 591 / 1900) loss: 1.514594
(Iteration 601 / 1900) loss: 1.480394
(Iteration 611 / 1900) loss: 1.442481
(Iteration 621 / 1900) loss: 1.395535
(Iteration 631 / 1900) loss: 1.458506
(Iteration 641 / 1900) loss: 1.414946
(Iteration 651 / 1900) loss: 1.374853
(Iteration 661 / 1900) loss: 1.389146
(Epoch 7 / 20) train acc: 0.610000; val_acc: 0.518000
(Iteration 671 / 1900) loss: 1.360294
(Iteration 681 / 1900) loss: 1.382825
(Iteration 691 / 1900) loss: 1.393190
(Iteration 701 / 1900) loss: 1.394862
(Iteration 711 / 1900) loss: 1.390067
(Iteration 721 / 1900) loss: 1.350420
(Iteration 731 / 1900) loss: 1.318461
(Iteration 741 / 1900) loss: 1.335798
(Iteration 751 / 1900) loss: 1.324472
(Epoch 8 / 20) train acc: 0.594000; val_acc: 0.525000
(Iteration 761 / 1900) loss: 1.318859
(Iteration 771 / 1900) loss: 1.341924
(Iteration 781 / 1900) loss: 1.294231
(Iteration 791 / 1900) loss: 1.310178
(Iteration 801 / 1900) loss: 1.274419
(Iteration 811 / 1900) loss: 1.254731
(Iteration 821 / 1900) loss: 1.299024
(Iteration 831 / 1900) loss: 1.244548
(Iteration 841 / 1900) loss: 1.246494
(Iteration 851 / 1900) loss: 1.244097
(Epoch 9 / 20) train acc: 0.631000; val_acc: 0.529000
(Iteration 861 / 1900) loss: 1.248886
(Iteration 871 / 1900) loss: 1.191845
(Iteration 881 / 1900) loss: 1.180847
(Iteration 891 / 1900) loss: 1.213840
(Iteration 901 / 1900) loss: 1.179208
(Iteration 911 / 1900) loss: 1.223424
(Iteration 921 / 1900) loss: 1.186671
(Iteration 931 / 1900) loss: 1.190322
(Iteration 941 / 1900) loss: 1.175785
(Epoch 10 / 20) train acc: 0.654000; val_acc: 0.542000
```

```
(Iteration 951 / 1900) loss: 1.151077
(Iteration 961 / 1900) loss: 1.175243
(Iteration 971 / 1900) loss: 1.104742
(Iteration 981 / 1900) loss: 1.171021
(Iteration 991 / 1900) loss: 1.142526
(Iteration 1001 / 1900) loss: 1.144471
(Iteration 1011 / 1900) loss: 1.150462
(Iteration 1021 / 1900) loss: 1.106181
(Iteration 1031 / 1900) loss: 1.163990
(Iteration 1041 / 1900) loss: 1.160388
(Epoch 11 / 20) train acc: 0.660000; val_acc: 0.538000
(Iteration 1051 / 1900) loss: 1.098445
(Iteration 1061 / 1900) loss: 1.101493
(Iteration 1071 / 1900) loss: 1.073697
(Iteration 1081 / 1900) loss: 1.065375
(Iteration 1091 / 1900) loss: 1.111463
(Iteration 1101 / 1900) loss: 1.070320
(Iteration 1111 / 1900) loss: 1.129073
(Iteration 1121 / 1900) loss: 1.090626
(Iteration 1131 / 1900) loss: 1.087518
(Epoch 12 / 20) train acc: 0.675000; val_acc: 0.548000
(Iteration 1141 / 1900) loss: 1.019781
(Iteration 1151 / 1900) loss: 1.065326
(Iteration 1161 / 1900) loss: 1.027108
(Iteration 1171 / 1900) loss: 1.032178
(Iteration 1181 / 1900) loss: 0.983596
(Iteration 1191 / 1900) loss: 1.009490
(Iteration 1201 / 1900) loss: 1.029025
(Iteration 1211 / 1900) loss: 0.994719
(Iteration 1221 / 1900) loss: 1.024637
(Iteration 1231 / 1900) loss: 0.964635
(Epoch 13 / 20) train acc: 0.709000; val_acc: 0.549000
(Iteration 1241 / 1900) loss: 1.042722
(Iteration 1251 / 1900) loss: 1.007796
(Iteration 1261 / 1900) loss: 1.013787
(Iteration 1271 / 1900) loss: 0.991796
(Iteration 1281 / 1900) loss: 1.012707
(Iteration 1291 / 1900) loss: 1.019342
(Iteration 1301 / 1900) loss: 0.924864
(Iteration 1311 / 1900) loss: 0.909758
(Iteration 1321 / 1900) loss: 0.998612
(Epoch 14 / 20) train acc: 0.719000; val_acc: 0.548000
(Iteration 1331 / 1900) loss: 0.969035
(Iteration 1341 / 1900) loss: 0.975582
(Iteration 1351 / 1900) loss: 0.862387
(Iteration 1361 / 1900) loss: 0.976016
(Iteration 1371 / 1900) loss: 0.911795
(Iteration 1381 / 1900) loss: 0.882080
```

```
(Iteration 1391 / 1900) loss: 0.932098
(Iteration 1401 / 1900) loss: 0.903551
(Iteration 1411 / 1900) loss: 0.929937
(Iteration 1421 / 1900) loss: 0.915218
(Epoch 15 / 20) train acc: 0.710000; val_acc: 0.547000
(Iteration 1431 / 1900) loss: 0.904085
(Iteration 1441 / 1900) loss: 0.896870
(Iteration 1451 / 1900) loss: 0.889115
(Iteration 1461 / 1900) loss: 0.883298
(Iteration 1471 / 1900) loss: 0.909916
(Iteration 1481 / 1900) loss: 0.881192
(Iteration 1491 / 1900) loss: 0.872293
(Iteration 1501 / 1900) loss: 0.836927
(Iteration 1511 / 1900) loss: 0.905142
(Epoch 16 / 20) train acc: 0.738000; val_acc: 0.538000
(Iteration 1521 / 1900) loss: 0.856409
(Iteration 1531 / 1900) loss: 0.866541
(Iteration 1541 / 1900) loss: 0.905097
(Iteration 1551 / 1900) loss: 0.859512
(Iteration 1561 / 1900) loss: 0.882577
(Iteration 1571 / 1900) loss: 0.832192
(Iteration 1581 / 1900) loss: 0.838827
(Iteration 1591 / 1900) loss: 0.881096
(Iteration 1601 / 1900) loss: 0.776344
(Iteration 1611 / 1900) loss: 0.774334
(Epoch 17 / 20) train acc: 0.757000; val_acc: 0.550000
(Iteration 1621 / 1900) loss: 0.798750
(Iteration 1631 / 1900) loss: 0.863733
(Iteration 1641 / 1900) loss: 0.783295
(Iteration 1651 / 1900) loss: 0.826082
(Iteration 1661 / 1900) loss: 0.771152
(Iteration 1671 / 1900) loss: 0.835624
(Iteration 1681 / 1900) loss: 0.805921
(Iteration 1691 / 1900) loss: 0.790279
(Iteration 1701 / 1900) loss: 0.816087
(Epoch 18 / 20) train acc: 0.781000; val_acc: 0.549000
(Iteration 1711 / 1900) loss: 0.757714
(Iteration 1721 / 1900) loss: 0.782246
(Iteration 1731 / 1900) loss: 0.741445
(Iteration 1741 / 1900) loss: 0.795494
(Iteration 1751 / 1900) loss: 0.797462
(Iteration 1761 / 1900) loss: 0.778272
(Iteration 1771 / 1900) loss: 0.748129
(Iteration 1781 / 1900) loss: 0.765330
(Iteration 1791 / 1900) loss: 0.785642
(Iteration 1801 / 1900) loss: 0.719710
(Epoch 19 / 20) train acc: 0.775000; val_acc: 0.536000
(Iteration 1811 / 1900) loss: 0.764646
```

```
(Iteration 1821 / 1900) loss: 0.758726
(Iteration 1831 / 1900) loss: 0.748310
(Iteration 1841 / 1900) loss: 0.724422
(Iteration 1851 / 1900) loss: 0.768859
(Iteration 1861 / 1900) loss: 0.690146
(Iteration 1871 / 1900) loss: 0.657766
(Iteration 1881 / 1900) loss: 0.714203
(Iteration 1891 / 1900) loss: 0.762543
(Epoch 20 / 20) train acc: 0.783000; val_acc: 0.538000
(Iteration 1 / 1900) loss: 2.304088
(Epoch 0 / 20) train acc: 0.152000; val_acc: 0.163000
(Iteration 11 / 1900) loss: 2.264734
(Iteration 21 / 1900) loss: 2.245788
(Iteration 31 / 1900) loss: 2.224398
(Iteration 41 / 1900) loss: 2.203254
(Iteration 51 / 1900) loss: 2.184167
(Iteration 61 / 1900) loss: 2.173704
(Iteration 71 / 1900) loss: 2.143985
(Iteration 81 / 1900) loss: 2.113779
(Iteration 91 / 1900) loss: 2.100269
(Epoch 1 / 20) train acc: 0.448000; val_acc: 0.440000
(Iteration 101 / 1900) loss: 2.090337
(Iteration 111 / 1900) loss: 2.074178
(Iteration 121 / 1900) loss: 2.047773
(Iteration 131 / 1900) loss: 2.040729
(Iteration 141 / 1900) loss: 2.013208
(Iteration 151 / 1900) loss: 2.005575
(Iteration 161 / 1900) loss: 1.969378
(Iteration 171 / 1900) loss: 1.984999
(Iteration 181 / 1900) loss: 1.945341
(Epoch 2 / 20) train acc: 0.471000; val_acc: 0.464000
(Iteration 191 / 1900) loss: 1.940332
(Iteration 201 / 1900) loss: 1.905618
(Iteration 211 / 1900) loss: 1.900816
(Iteration 221 / 1900) loss: 1.890857
(Iteration 231 / 1900) loss: 1.905542
(Iteration 241 / 1900) loss: 1.844497
(Iteration 251 / 1900) loss: 1.840739
(Iteration 261 / 1900) loss: 1.835602
(Iteration 271 / 1900) loss: 1.832844
(Iteration 281 / 1900) loss: 1.835309
(Epoch 3 / 20) train acc: 0.512000; val_acc: 0.487000
(Iteration 291 / 1900) loss: 1.786005
(Iteration 301 / 1900) loss: 1.792901
(Iteration 311 / 1900) loss: 1.764422
(Iteration 321 / 1900) loss: 1.753008
(Iteration 331 / 1900) loss: 1.753582
(Iteration 341 / 1900) loss: 1.748352
```

```
(Iteration 351 / 1900) loss: 1.690958
(Iteration 361 / 1900) loss: 1.711949
(Iteration 371 / 1900) loss: 1.690728
(Epoch 4 / 20) train acc: 0.528000; val_acc: 0.487000
(Iteration 381 / 1900) loss: 1.699726
(Iteration 391 / 1900) loss: 1.653371
(Iteration 401 / 1900) loss: 1.655818
(Iteration 411 / 1900) loss: 1.607759
(Iteration 421 / 1900) loss: 1.642982
(Iteration 431 / 1900) loss: 1.573872
(Iteration 441 / 1900) loss: 1.618076
(Iteration 451 / 1900) loss: 1.617791
(Iteration 461 / 1900) loss: 1.586405
(Iteration 471 / 1900) loss: 1.577124
(Epoch 5 / 20) train acc: 0.538000; val_acc: 0.506000
(Iteration 481 / 1900) loss: 1.548741
(Iteration 491 / 1900) loss: 1.584383
(Iteration 501 / 1900) loss: 1.541883
(Iteration 511 / 1900) loss: 1.541198
(Iteration 521 / 1900) loss: 1.505780
(Iteration 531 / 1900) loss: 1.481461
(Iteration 541 / 1900) loss: 1.479929
(Iteration 551 / 1900) loss: 1.480393
(Iteration 561 / 1900) loss: 1.482670
(Epoch 6 / 20) train acc: 0.545000; val_acc: 0.518000
(Iteration 571 / 1900) loss: 1.490718
(Iteration 581 / 1900) loss: 1.486866
(Iteration 591 / 1900) loss: 1.451582
(Iteration 601 / 1900) loss: 1.440298
(Iteration 611 / 1900) loss: 1.425506
(Iteration 621 / 1900) loss: 1.436323
(Iteration 631 / 1900) loss: 1.440708
(Iteration 641 / 1900) loss: 1.416991
(Iteration 651 / 1900) loss: 1.413462
(Iteration 661 / 1900) loss: 1.380991
(Epoch 7 / 20) train acc: 0.589000; val_acc: 0.513000
(Iteration 671 / 1900) loss: 1.393565
(Iteration 681 / 1900) loss: 1.343194
(Iteration 691 / 1900) loss: 1.464759
(Iteration 701 / 1900) loss: 1.350502
(Iteration 711 / 1900) loss: 1.295518
(Iteration 721 / 1900) loss: 1.378348
(Iteration 731 / 1900) loss: 1.365553
(Iteration 741 / 1900) loss: 1.338474
(Iteration 751 / 1900) loss: 1.322449
(Epoch 8 / 20) train acc: 0.605000; val_acc: 0.525000
(Iteration 761 / 1900) loss: 1.298092
(Iteration 771 / 1900) loss: 1.245473
```

```
(Iteration 781 / 1900) loss: 1.376611
(Iteration 791 / 1900) loss: 1.283562
(Iteration 801 / 1900) loss: 1.300460
(Iteration 811 / 1900) loss: 1.262352
(Iteration 821 / 1900) loss: 1.242104
(Iteration 831 / 1900) loss: 1.241939
(Iteration 841 / 1900) loss: 1.297188
(Iteration 851 / 1900) loss: 1.258536
(Epoch 9 / 20) train acc: 0.602000; val_acc: 0.534000
(Iteration 861 / 1900) loss: 1.227745
(Iteration 871 / 1900) loss: 1.268587
(Iteration 881 / 1900) loss: 1.183902
(Iteration 891 / 1900) loss: 1.241912
(Iteration 901 / 1900) loss: 1.182783
(Iteration 911 / 1900) loss: 1.209808
(Iteration 921 / 1900) loss: 1.163352
(Iteration 931 / 1900) loss: 1.153717
(Iteration 941 / 1900) loss: 1.156499
(Epoch 10 / 20) train acc: 0.644000; val_acc: 0.532000
(Iteration 951 / 1900) loss: 1.165338
(Iteration 961 / 1900) loss: 1.154633
(Iteration 971 / 1900) loss: 1.193703
(Iteration 981 / 1900) loss: 1.203373
(Iteration 991 / 1900) loss: 1.107580
(Iteration 1001 / 1900) loss: 1.155705
(Iteration 1011 / 1900) loss: 1.191871
(Iteration 1021 / 1900) loss: 1.153879
(Iteration 1031 / 1900) loss: 1.093842
(Iteration 1041 / 1900) loss: 1.184171
(Epoch 11 / 20) train acc: 0.671000; val_acc: 0.537000
(Iteration 1051 / 1900) loss: 1.123900
(Iteration 1061 / 1900) loss: 1.111240
(Iteration 1071 / 1900) loss: 1.057605
(Iteration 1081 / 1900) loss: 1.067244
(Iteration 1091 / 1900) loss: 1.073915
(Iteration 1101 / 1900) loss: 1.064443
(Iteration 1111 / 1900) loss: 1.085774
(Iteration 1121 / 1900) loss: 1.095750
(Iteration 1131 / 1900) loss: 1.026180
(Epoch 12 / 20) train acc: 0.663000; val_acc: 0.527000
(Iteration 1141 / 1900) loss: 1.071005
(Iteration 1151 / 1900) loss: 1.073046
(Iteration 1161 / 1900) loss: 1.012072
(Iteration 1171 / 1900) loss: 1.009721
(Iteration 1181 / 1900) loss: 1.020719
(Iteration 1191 / 1900) loss: 1.041568
(Iteration 1201 / 1900) loss: 1.039113
(Iteration 1211 / 1900) loss: 1.010129
```

```
(Iteration 1221 / 1900) loss: 1.034832
(Iteration 1231 / 1900) loss: 1.035915
(Epoch 13 / 20) train acc: 0.655000; val_acc: 0.527000
(Iteration 1241 / 1900) loss: 1.035244
(Iteration 1251 / 1900) loss: 1.021402
(Iteration 1261 / 1900) loss: 0.937832
(Iteration 1271 / 1900) loss: 0.964262
(Iteration 1281 / 1900) loss: 0.974101
(Iteration 1291 / 1900) loss: 1.002775
(Iteration 1301 / 1900) loss: 0.961867
(Iteration 1311 / 1900) loss: 0.971274
(Iteration 1321 / 1900) loss: 1.008155
(Epoch 14 / 20) train acc: 0.685000; val_acc: 0.543000
(Iteration 1331 / 1900) loss: 0.946185
(Iteration 1341 / 1900) loss: 0.971097
(Iteration 1351 / 1900) loss: 0.904443
(Iteration 1361 / 1900) loss: 0.921818
(Iteration 1371 / 1900) loss: 0.946412
(Iteration 1381 / 1900) loss: 0.959453
(Iteration 1391 / 1900) loss: 0.930161
(Iteration 1401 / 1900) loss: 0.960547
(Iteration 1411 / 1900) loss: 0.910476
(Iteration 1421 / 1900) loss: 0.926228
(Epoch 15 / 20) train acc: 0.682000; val_acc: 0.526000
(Iteration 1431 / 1900) loss: 0.977615
(Iteration 1441 / 1900) loss: 0.917100
(Iteration 1451 / 1900) loss: 0.923112
(Iteration 1461 / 1900) loss: 0.900011
(Iteration 1471 / 1900) loss: 0.880424
(Iteration 1481 / 1900) loss: 0.880734
(Iteration 1491 / 1900) loss: 0.935662
(Iteration 1501 / 1900) loss: 0.855756
(Iteration 1511 / 1900) loss: 0.982644
(Epoch 16 / 20) train acc: 0.721000; val_acc: 0.526000
(Iteration 1521 / 1900) loss: 0.853583
(Iteration 1531 / 1900) loss: 0.844687
(Iteration 1541 / 1900) loss: 0.873171
(Iteration 1551 / 1900) loss: 0.878685
(Iteration 1561 / 1900) loss: 0.845417
(Iteration 1571 / 1900) loss: 0.855334
(Iteration 1581 / 1900) loss: 0.885154
(Iteration 1591 / 1900) loss: 0.864232
(Iteration 1601 / 1900) loss: 0.783653
(Iteration 1611 / 1900) loss: 0.844483
(Epoch 17 / 20) train acc: 0.732000; val_acc: 0.518000
(Iteration 1621 / 1900) loss: 0.852472
(Iteration 1631 / 1900) loss: 0.838146
(Iteration 1641 / 1900) loss: 0.818284
```

```
(Iteration 1651 / 1900) loss: 0.871789
(Iteration 1661 / 1900) loss: 0.866263
(Iteration 1671 / 1900) loss: 0.783905
(Iteration 1681 / 1900) loss: 0.822550
(Iteration 1691 / 1900) loss: 0.789717
(Iteration 1701 / 1900) loss: 0.838877
(Epoch 18 / 20) train acc: 0.741000; val_acc: 0.534000
(Iteration 1711 / 1900) loss: 0.758247
(Iteration 1721 / 1900) loss: 0.745799
(Iteration 1731 / 1900) loss: 0.766752
(Iteration 1741 / 1900) loss: 0.781263
(Iteration 1751 / 1900) loss: 0.816872
(Iteration 1761 / 1900) loss: 0.793792
(Iteration 1771 / 1900) loss: 0.792290
(Iteration 1781 / 1900) loss: 0.814911
(Iteration 1791 / 1900) loss: 0.782773
(Iteration 1801 / 1900) loss: 0.744798
(Epoch 19 / 20) train acc: 0.759000; val_acc: 0.525000
(Iteration 1811 / 1900) loss: 0.788431
(Iteration 1821 / 1900) loss: 0.771635
(Iteration 1831 / 1900) loss: 0.734369
(Iteration 1841 / 1900) loss: 0.751021
(Iteration 1851 / 1900) loss: 0.730022
(Iteration 1861 / 1900) loss: 0.747549
(Iteration 1871 / 1900) loss: 0.716268
(Iteration 1881 / 1900) loss: 0.735369
(Iteration 1891 / 1900) loss: 0.725481
(Epoch 20 / 20) train acc: 0.754000; val_acc: 0.514000
Best validation accuracy using a small dataset: 0.589

Training with best parameters…
Best validation accuracy using full dataset: 0.589
Best Model <cs231n.classifiers.fc_net.FullyConnectedNet object at
0x7b124be4a950>
```

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.

[ ]:
```
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:  0.56
Test set accuracy:  0.533
```

BatchNormalization

November 29, 2023

[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/assignment1/'
FOLDERNAME = 'Colab Notebooks/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/Colab Notebooks/assignment2/cs231n/datasets
/content/drive/My Drive/Colab Notebooks/assignment2
```

1 Batch Normalization

One way to make deep networks easier to train is to use more sophisticated optimization procedures such as SGD+momentum, RMSProp, or Adam. Another strategy is to change the architecture of the network to make it easier to train. One idea along these lines is batch normalization, proposed by [1] in 2015.

To understand the goal of batch normalization, it is important to first recognize that machine learning methods tend to perform better with input data consisting of uncorrelated features with zero mean and unit variance. When training a neural network, we can preprocess the data before feeding it to the network to explicitly decorrelate its features. This will ensure that the first layer of the network sees data that follows a nice distribution. However, even if we preprocess the input data, the activations at deeper layers of the network will likely no longer be decorrelated and will

no longer have zero mean or unit variance, since they are output from earlier layers in the network. Even worse, during the training process the distribution of features at each layer of the network will shift as the weights of each layer are updated.

The authors of [1] hypothesize that the shifting distribution of features inside deep neural networks may make training deep networks more difficult. To overcome this problem, they propose to insert into the network layers that normalize batches. At training time, such a layer uses a minibatch of data to estimate the mean and standard deviation of each feature. These estimated means and standard deviations are then used to center and normalize the features of the minibatch. A running average of these means and standard deviations is kept during training, and at test time these running averages are used to center and normalize features.

It is possible that this normalization strategy could reduce the representational power of the network, since it may sometimes be optimal for certain layers to have features that are not zero-mean or unit variance. To this end, the batch normalization layer includes learnable shift and scale parameters for each feature dimension.

[1] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

[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))))

def print_mean_std(x,axis=0):
    print(f"  means: {x.mean(axis=axis)}")
    print(f"  stds:  {x.std(axis=axis)}\n")
```

[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,)
```

2 Batch Normalization: Forward Pass

In the file `cs231n/layers.py`, implement the batch normalization forward pass in the function `batchnorm_forward`. Once you have done so, run the following to test your implementation.

Referencing the paper linked to above in [1] may be helpful!

[4]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after batch normalization

# Simulate the forward pass for a two-layer network.
np.random.seed(231)
N, D1, D2, D3 = 200, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)

print('Before batch normalization:')
print_mean_std(a,axis=0)

gamma = np.ones((D3,))
beta = np.zeros((D3,))

# Means should be close to zero and stds close to one.
print('After batch normalization (gamma=1, beta=0)')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)

gamma = np.asarray([1.0, 2.0, 3.0])
beta = np.asarray([11.0, 12.0, 13.0])

# Now means should be close to beta and stds close to gamma.
print('After batch normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = batchnorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=0)
```

```
Before batch normalization:
  means: [ -2.3814598  -13.18038246   1.91780462]
```

```
  stds:  [27.18502186 34.21455511 37.68611762]

After batch normalization (gamma=1, beta=0)
  means: [5.32907052e-17 7.04991621e-17 1.85962357e-17]
  stds:  [0.99999999 1.         1.        ]

After batch normalization (gamma= [1. 2. 3.] , beta= [11. 12. 13.] )
  means: [11. 12. 13.]
  stds:  [0.99999999 1.99999999 2.99999999]
```

[5]:
```python
# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.

np.random.seed(231)
N, D1, D2, D3 = 200, 50, 60, 3
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)

bn_param = {'mode': 'train'}
gamma = np.ones(D3)
beta = np.zeros(D3)

for t in range(50):
  X = np.random.randn(N, D1)
  a = np.maximum(0, X.dot(W1)).dot(W2)
  batchnorm_forward(a, gamma, beta, bn_param)

bn_param['mode'] = 'test'
X = np.random.randn(N, D1)
a = np.maximum(0, X.dot(W1)).dot(W2)
a_norm, _ = batchnorm_forward(a, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After batch normalization (test-time):')
print_mean_std(a_norm,axis=0)
```

```
After batch normalization (test-time):
  means: [-0.03927354 -0.04349152 -0.10452688]
  stds:  [1.01531428 1.01238373 0.97819988]
```

3 Batch Normalization: Backward Pass

Now implement the backward pass for batch normalization in the function `batchnorm_backward`.

To derive the backward pass you should write out the computation graph for batch normalization and backprop through each of the intermediate nodes. Some intermediates may have multiple outgoing branches; make sure to sum gradients across these branches in the backward pass.

Once you have finished, run the following to numerically check your backward pass.

[6]:
```
# Gradient check batchnorm backward pass.
np.random.seed(231)
N, D = 4, 5
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

bn_param = {'mode': 'train'}
fx = lambda x: batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: batchnorm_forward(x, a, beta, bn_param)[0]
fb = lambda b: batchnorm_forward(x, gamma, b, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma.copy(), dout)
db_num = eval_numerical_gradient_array(fb, beta.copy(), dout)

_, cache = batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = batchnorm_backward(dout, cache)

# You should expect to see relative errors between 1e-13 and 1e-8.
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  1.7029235612572515e-09
dgamma error:  7.420414216247087e-13
dbeta error:  2.8795057655839487e-12
```

4 Batch Normalization: Alternative Backward Pass

In class we talked about two different implementations for the sigmoid backward pass. One strategy is to write out a computation graph composed of simple operations and backprop through all intermediate values. Another strategy is to work out the derivatives on paper. For example, you can derive a very simple formula for the sigmoid function's backward pass by simplifying gradients on paper.

Surprisingly, it turns out that you can do a similar simplification for the batch normalization backward pass too!

In the forward pass, given a set of inputs $X = \begin{bmatrix} x_1 \\ x_2 \\ \cdots \\ x_N \end{bmatrix}$,

we first calculate the mean μ and variance v. With μ and v calculated, we can calculate the standard deviation σ and normalized data Y. The equations and graph illustration below describe the computation (y_i is the i-th element of the vector Y).

$$\mu = \frac{1}{N}\sum_{k=1}^{N} x_k \qquad v = \frac{1}{N}\sum_{k=1}^{N}(x_k - \mu)^2 \tag{1}$$

$$\sigma = \sqrt{v + \epsilon} \qquad y_i = \frac{x_i - \mu}{\sigma} \tag{2}$$

The meat of our problem during backpropagation is to compute $\frac{\partial L}{\partial X}$, given the upstream gradient we receive, $\frac{\partial L}{\partial Y}$. To do this, recall the chain rule in calculus gives us $\frac{\partial L}{\partial X} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial X}$.

The unknown/hard part is $\frac{\partial Y}{\partial X}$. We can find this by first deriving step-by-step our local gradients at $\frac{\partial v}{\partial X}$, $\frac{\partial \mu}{\partial X}$, $\frac{\partial \sigma}{\partial v}$, $\frac{\partial Y}{\partial \sigma}$, and $\frac{\partial Y}{\partial \mu}$, and then use the chain rule to compose these gradients (which appear in the form of vectors!) appropriately to compute $\frac{\partial Y}{\partial X}$.

If it's challenging to directly reason about the gradients over X and Y which require matrix multiplication, try reasoning about the gradients in terms of individual elements x_i and y_i first: in that case, you will need to come up with the derivations for $\frac{\partial L}{\partial x_i}$, by relying on the Chain Rule to first calculate the intermediate $\frac{\partial \mu}{\partial x_i}$, $\frac{\partial v}{\partial x_i}$, $\frac{\partial \sigma}{\partial x_i}$, then assemble these pieces to calculate $\frac{\partial y_i}{\partial x_i}$.

You should make sure each of the intermediary gradient derivations are all as simplified as possible, for ease of implementation.

After doing so, implement the simplified batch normalization backward pass in the function `batchnorm_backward_alt` and compare the two implementations by running the following. Your two implementations should compute nearly identical results, but the alternative implementation should be a bit faster.

[7]:
```
np.random.seed(231)
N, D = 100, 500
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

bn_param = {'mode': 'train'}
out, cache = batchnorm_forward(x, gamma, beta, bn_param)

t1 = time.time()
dx1, dgamma1, dbeta1 = batchnorm_backward(dout, cache)
t2 = time.time()
dx2, dgamma2, dbeta2 = batchnorm_backward_alt(dout, cache)
t3 = time.time()
```

```
print('dx difference: ', rel_error(dx1, dx2))
print('dgamma difference: ', rel_error(dgamma1, dgamma2))
print('dbeta difference: ', rel_error(dbeta1, dbeta2))
print('speedup: %.2fx' % ((t2 - t1) / (t3 - t2)))
```

```
dx difference:  5.344522019088175e-13
dgamma difference:  0.0
dbeta difference:  0.0
speedup: 1.19x
```

5 Fully Connected Networks with Batch Normalization

Now that you have a working implementation for batch normalization, go back to your `FullyConnectedNet` in the file `cs231n/classifiers/fc_net.py`. Modify your implementation to add batch normalization.

Concretely, when the `normalization` flag is set to `"batchnorm"` in the constructor, you should insert a batch normalization layer before each ReLU nonlinearity. The outputs from the last layer of the network should not be normalized. Once you are done, run the following to gradient-check your implementation.

Hint: You might find it useful to define an additional helper layer similar to those in the file `cs231n/layer_utils.py`.

[8]:
```
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,))

# You should expect losses between 1e-4~1e-10 for W,
# losses between 1e-08~1e-10 for b,
# and losses between 1e-08~1e-09 for beta and gammas.
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,
                            normalization='batchnorm')

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

  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('%s relative error: %.2e' % (name, rel_error(grad_num, grads[name])))
  if reg == 0: print()
```

```
Running check with reg =  0
Initial loss:  2.2611955101340957
W1 relative error: 1.10e-04
W2 relative error: 5.65e-06
W3 relative error: 4.14e-10
b1 relative error: 2.66e-07
b2 relative error: 2.22e-08
b3 relative error: 1.02e-10
beta1 relative error: 7.33e-09
beta2 relative error: 1.17e-09
gamma1 relative error: 7.47e-09
gamma2 relative error: 3.35e-09

Running check with reg =  3.14
Initial loss:  6.996533220108303
W1 relative error: 1.98e-06
W2 relative error: 2.29e-06
W3 relative error: 2.79e-08
b1 relative error: 1.67e-08
b2 relative error: 7.99e-07
b3 relative error: 2.10e-10
beta1 relative error: 6.32e-09
beta2 relative error: 3.48e-09
gamma1 relative error: 6.27e-09
gamma2 relative error: 4.14e-09
```

6 Batch Normalization for Deep Networks

Run the following to train a six-layer network on a subset of 1000 training examples both with and without batch normalization.

[9]:
```
np.random.seed(231)

# Try training a very deep net with batchnorm.
hidden_dims = [100, 100, 100, 100, 100]

num_train = 1000
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'],
}

weight_scale = 2e-2
bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization='batchnorm')
```

```
model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization=None)

print('Solver with batch norm:')
bn_solver = Solver(bn_model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True,print_every=20)
bn_solver.train()

print('\nSolver without batch norm:')
solver = Solver(model, small_data,
                num_epochs=10, batch_size=50,
                update_rule='adam',
                optim_config={
                  'learning_rate': 1e-3,
                },
                verbose=True, print_every=20)
solver.train()
```

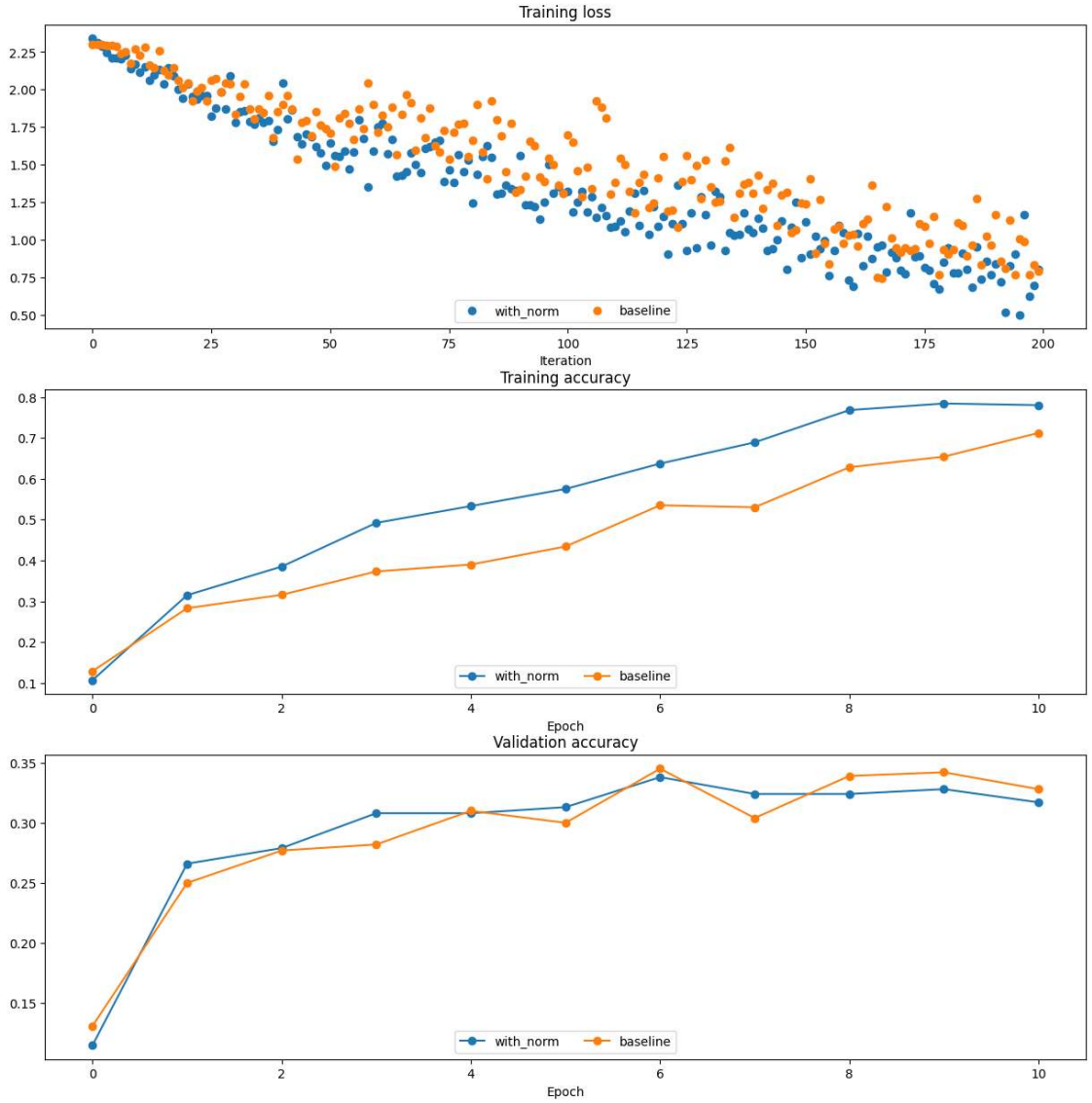
```
Solver with batch norm:
(Iteration 1 / 200) loss: 2.340974
(Epoch 0 / 10) train acc: 0.107000; val_acc: 0.115000
(Epoch 1 / 10) train acc: 0.315000; val_acc: 0.266000
(Iteration 21 / 200) loss: 2.039365
(Epoch 2 / 10) train acc: 0.385000; val_acc: 0.279000
(Iteration 41 / 200) loss: 2.041103
(Epoch 3 / 10) train acc: 0.492000; val_acc: 0.308000
(Iteration 61 / 200) loss: 1.753903
(Epoch 4 / 10) train acc: 0.533000; val_acc: 0.308000
(Iteration 81 / 200) loss: 1.246585
(Epoch 5 / 10) train acc: 0.575000; val_acc: 0.313000
(Iteration 101 / 200) loss: 1.320591
(Epoch 6 / 10) train acc: 0.637000; val_acc: 0.338000
(Iteration 121 / 200) loss: 1.157329
(Epoch 7 / 10) train acc: 0.689000; val_acc: 0.324000
(Iteration 141 / 200) loss: 1.145713
(Epoch 8 / 10) train acc: 0.768000; val_acc: 0.324000
(Iteration 161 / 200) loss: 0.689452
(Epoch 9 / 10) train acc: 0.784000; val_acc: 0.328000
(Iteration 181 / 200) loss: 0.949979
(Epoch 10 / 10) train acc: 0.780000; val_acc: 0.317000

Solver without batch norm:
```

```
(Iteration 1 / 200) loss: 2.302332
(Epoch 0 / 10) train acc: 0.129000; val_acc: 0.131000
(Epoch 1 / 10) train acc: 0.283000; val_acc: 0.250000
(Iteration 21 / 200) loss: 2.041970
(Epoch 2 / 10) train acc: 0.316000; val_acc: 0.277000
(Iteration 41 / 200) loss: 1.900473
(Epoch 3 / 10) train acc: 0.373000; val_acc: 0.282000
(Iteration 61 / 200) loss: 1.713156
(Epoch 4 / 10) train acc: 0.390000; val_acc: 0.310000
(Iteration 81 / 200) loss: 1.662209
(Epoch 5 / 10) train acc: 0.434000; val_acc: 0.300000
(Iteration 101 / 200) loss: 1.696059
(Epoch 6 / 10) train acc: 0.535000; val_acc: 0.345000
(Iteration 121 / 200) loss: 1.557987
(Epoch 7 / 10) train acc: 0.530000; val_acc: 0.304000
(Iteration 141 / 200) loss: 1.432189
(Epoch 8 / 10) train acc: 0.628000; val_acc: 0.339000
(Iteration 161 / 200) loss: 1.034116
(Epoch 9 / 10) train acc: 0.654000; val_acc: 0.342000
(Iteration 181 / 200) loss: 0.905794
(Epoch 10 / 10) train acc: 0.712000; val_acc: 0.328000
```

Run the following to visualize the results from two networks trained above. You should find that using batch normalization helps the network to converge much faster.

[10]:
```
def plot_training_history(title, label, baseline, bn_solvers, plot_fn,␣
 ↪bl_marker='.', bn_marker='.', labels=None):
    """utility function for plotting training history"""
    plt.title(title)
    plt.xlabel(label)
    bn_plots = [plot_fn(bn_solver) for bn_solver in bn_solvers]
    bl_plot = plot_fn(baseline)
    num_bn = len(bn_plots)
    for i in range(num_bn):
        label='with_norm'
        if labels is not None:
            label += str(labels[i])
        plt.plot(bn_plots[i], bn_marker, label=label)
    label='baseline'
    if labels is not None:
        label += str(labels[0])
    plt.plot(bl_plot, bl_marker, label=label)
    plt.legend(loc='lower center', ncol=num_bn+1)


plt.subplot(3, 1, 1)
plot_training_history('Training loss','Iteration', solver, [bn_solver], \
                      lambda x: x.loss_history, bl_marker='o', bn_marker='o')
```

```
plt.subplot(3, 1, 2)
plot_training_history('Training accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.train_acc_history, bl_marker='-o',␣
 ↪bn_marker='-o')
plt.subplot(3, 1, 3)
plot_training_history('Validation accuracy','Epoch', solver, [bn_solver], \
                      lambda x: x.val_acc_history, bl_marker='-o',␣
 ↪bn_marker='-o')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



7 Batch Normalization and Initialization

We will now run a small experiment to study the interaction of batch normalization and weight initialization.

The first cell will train eight-layer networks both with and without batch normalization using different scales for weight initialization. The second layer will plot training accuracy, validation set accuracy, and training loss as a function of the weight initialization scale.

[11]:
```
np.random.seed(231)

# Try training a very deep net with batchnorm.
hidden_dims = [50, 50, 50, 50, 50, 50, 50]
num_train = 1000
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'],
}

bn_solvers_ws = {}
solvers_ws = {}
weight_scales = np.logspace(-4, 0, num=20)
for i, weight_scale in enumerate(weight_scales):
    print('Running weight scale %d / %d' % (i + 1, len(weight_scales)))
    bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization='batchnorm')
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization=None)

    bn_solver = Solver(bn_model, small_data,
                    num_epochs=10, batch_size=50,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': 1e-3,
                    },
                    verbose=False, print_every=200)
    bn_solver.train()
    bn_solvers_ws[weight_scale] = bn_solver

    solver = Solver(model, small_data,
                    num_epochs=10, batch_size=50,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': 1e-3,
                    },
                    verbose=False, print_every=200)
```

```
    solver.train()
    solvers_ws[weight_scale] = solver
```

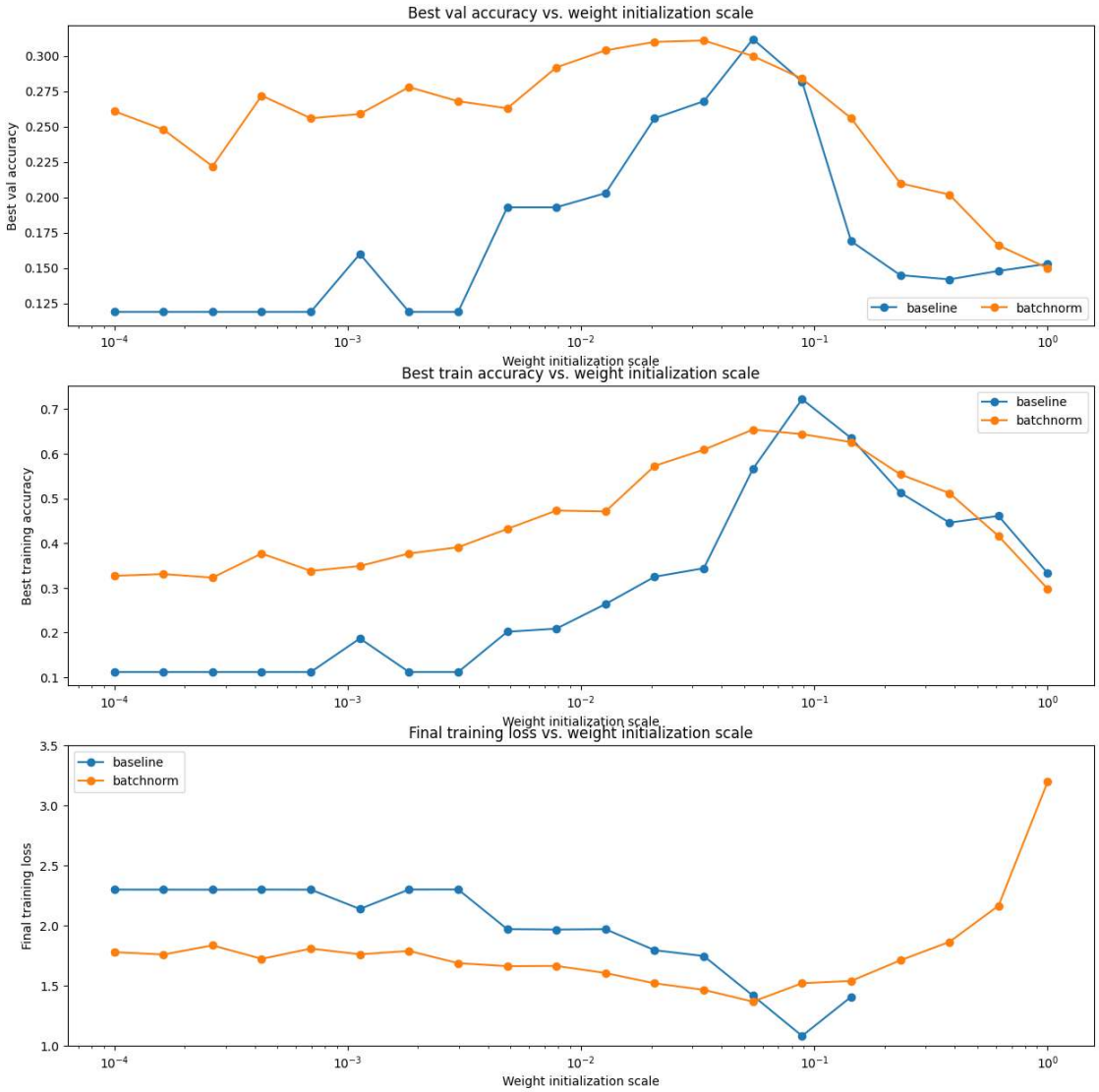
```
Running weight scale 1 / 20
Running weight scale 2 / 20
Running weight scale 3 / 20
Running weight scale 4 / 20
Running weight scale 5 / 20
Running weight scale 6 / 20
Running weight scale 7 / 20
Running weight scale 8 / 20
Running weight scale 9 / 20
Running weight scale 10 / 20
Running weight scale 11 / 20
Running weight scale 12 / 20
Running weight scale 13 / 20
Running weight scale 14 / 20
Running weight scale 15 / 20
Running weight scale 16 / 20
```

```
/content/drive/My Drive/Colab Notebooks/assignment2/cs231n/layers.py:151:
RuntimeWarning: divide by zero encountered in log
  correct_log_probs = -np.log(probabilities[np.arange(N), y])
```

```
Running weight scale 17 / 20
Running weight scale 18 / 20
Running weight scale 19 / 20
Running weight scale 20 / 20
```

[12]:
```
# Plot results of weight scale experiment.
best_train_accs, bn_best_train_accs = [], []
best_val_accs, bn_best_val_accs = [], []
final_train_loss, bn_final_train_loss = [], []

for ws in weight_scales:
  best_train_accs.append(max(solvers_ws[ws].train_acc_history))
  bn_best_train_accs.append(max(bn_solvers_ws[ws].train_acc_history))

  best_val_accs.append(max(solvers_ws[ws].val_acc_history))
  bn_best_val_accs.append(max(bn_solvers_ws[ws].val_acc_history))

  final_train_loss.append(np.mean(solvers_ws[ws].loss_history[-100:]))
  bn_final_train_loss.append(np.mean(bn_solvers_ws[ws].loss_history[-100:]))

plt.subplot(3, 1, 1)
plt.title('Best val accuracy vs. weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best val accuracy')
```

```
plt.semilogx(weight_scales, best_val_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_val_accs, '-o', label='batchnorm')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
plt.title('Best train accuracy vs. weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Best training accuracy')
plt.semilogx(weight_scales, best_train_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_train_accs, '-o', label='batchnorm')
plt.legend()

plt.subplot(3, 1, 3)
plt.title('Final training loss vs. weight initialization scale')
plt.xlabel('Weight initialization scale')
plt.ylabel('Final training loss')
plt.semilogx(weight_scales, final_train_loss, '-o', label='baseline')
plt.semilogx(weight_scales, bn_final_train_loss, '-o', label='batchnorm')
plt.legend()
plt.gca().set_ylim(1.0, 3.5)

plt.gcf().set_size_inches(15, 15)
plt.show()
```



7.1 Inline Question 1:

Describe the results of this experiment. How does the weight initialization scale affect models with/without batch normalization differently, and why?

7.2 Answer:

Bu deneyde, ağırlık başlangıç ölçeğinin, batch normalization kullanılan ve kullanılmayan modeller üzerindeki etkileri incelenmiştir. Ağırlık başlangıç ölçeği, model ağırlıklarının başlangıçtaki dağılımının genişliğini belirler ve modelin eğitim sürecini önemli ölçüde etkiler.

Batch Normalization Kullanılmayan Modeller: Batch normalization olmadan, modelin başarısı ağırlık başlangıç ölçeğine daha duyarlıdır. Çok düşük veya çok yüksek başlangıç değerleri, vanishing veya exploding neden olabilir. Bu, özellikle derin öğrenme modellerinde büyük sıkıntıdır. Düşük

başlangıç değerleri, yavaş öğrenmeye yol açarken, çok yüksek ölçek, aşırı büyük gradyanlar ve eğitimde istikrarsızlık ile sonuçlanır.

Batch Normalization Kullanılan Modeller: Batch normalization, her katmandaki aktivasyonların dağılımını normalize ederek, ağırlık başlangıç ölçeğinin etkilerini azaltır. BN, modelin farklı ağırlık ölçeklerine karşı daha esnek olmasını sağlar ve eğitim sürecini daha kararlı hale getirir. BN, ağırlıkların başlangıçtaki değerlerinden bağımsız olarak, her katmandaki aktivasyonları belirli bir dağılıma zorlar. Bu, ağırlık ölçeğinin oluşturduğu potansiyel gradyan sorunlarını giderir ve modelin daha geniş başlangıç değerleri aralığında etkili bir şekilde öğrenmesine olanak tanır.

8 Batch Normalization and Batch Size

We will now run a small experiment to study the interaction of batch normalization and batch size.

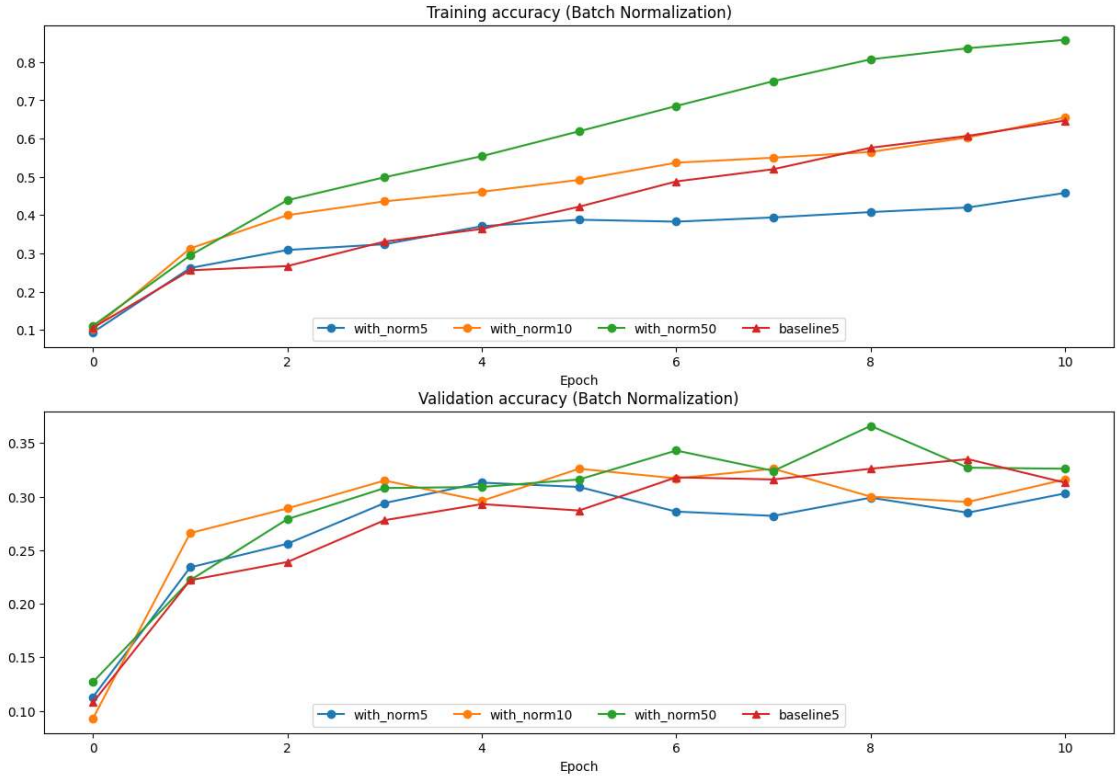
The first cell will train 6-layer networks both with and without batch normalization using different batch sizes. The second layer will plot training accuracy and validation set accuracy over time.

[13]:
```
def run_batchsize_experiments(normalization_mode):
    np.random.seed(231)

    # Try training a very deep net with batchnorm.
    hidden_dims = [100, 100, 100, 100, 100]
    num_train = 1000
    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'],
    }
    n_epochs=10
    weight_scale = 2e-2
    batch_sizes = [5,10,50]
    lr = 10**(-3.5)
    solver_bsize = batch_sizes[0]

    print('No normalization: batch size = ',solver_bsize)
    model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization=None)
    solver = Solver(model, small_data,
                    num_epochs=n_epochs, batch_size=solver_bsize,
                    update_rule='adam',
                    optim_config={
                      'learning_rate': lr,
                    },
                    verbose=False)
    solver.train()

    bn_solvers = []
```

```
    for i in range(len(batch_sizes)):
        b_size=batch_sizes[i]
        print('Normalization: batch size = ',b_size)
        bn_model = FullyConnectedNet(hidden_dims, weight_scale=weight_scale,␣
 ↪normalization=normalization_mode)
        bn_solver = Solver(bn_model, small_data,
                        num_epochs=n_epochs, batch_size=b_size,
                        update_rule='adam',
                        optim_config={
                          'learning_rate': lr,
                        },
                        verbose=False)
        bn_solver.train()
        bn_solvers.append(bn_solver)

    return bn_solvers, solver, batch_sizes

batch_sizes = [5,10,50]
bn_solvers_bsize, solver_bsize, batch_sizes =␣
 ↪run_batchsize_experiments('batchnorm')
```

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```

[14]:
```
plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Batch Normalization)','Epoch',␣
 ↪solver_bsize, bn_solvers_bsize, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Batch Normalization)','Epoch',␣
 ↪solver_bsize, bn_solvers_bsize, \
                      lambda x: x.val_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)

plt.gcf().set_size_inches(15, 10)
plt.show()
```



8.1 Inline Question 2:

Describe the results of this experiment. What does this imply about the relationship between batch normalization and batch size? Why is this relationship observed?

8.2 Answer:

Bu deneyin sonuçları, batch normalization (BN) kullanımının, farklı batch boyutlarıyla nasıl etkileşime girdiğini gösterir. Genellikle, BN'nin etkinliği batch boyutuna bağlıdır. Çok küçük batch boyutları kullanıldığında, BN'nin normalizasyon etkisi azalabilir çünkü her batch içindeki örneklerin sayısı az olduğunda, bu örnekler üzerinden hesaplanan ortalama ve varyans değerleri daha güvenilmez olur. Bu, modelin eğitim sürecinin istikrarını olumsuz etkiler. Öte yandan, daha büyük batch boyutları kullanıldığında, BN daha etkili olur çünkü ortalama ve varyans hesaplamaları daha fazla veri üzerinden yapılır ve bu da daha stabil bir normalizasyon sağlar. Bu durum, BN'nin batch boyutuna duyarlı bir özellik olduğunu ve ideal batch boyutunun seçiminin BN'nin etkinliği üzerinde önemli bir rol oynadığını gösterir.

9 Layer Normalization

Batch normalization has proved to be effective in making networks easier to train, but the dependency on batch size makes it less useful in complex networks which have a cap on the input batch size due to hardware limitations.

Several alternatives to batch normalization have been proposed to mitigate this problem; one such technique is Layer Normalization [2]. Instead of normalizing over the batch, we normalize over the features. In other words, when using Layer Normalization, each feature vector corresponding to a single datapoint is normalized based on the sum of all terms within that feature vector.

[2] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.

9.1 Inline Question 3:

Which of these data preprocessing steps is analogous to batch normalization, and which is analogous to layer normalization?

1. Scaling each image in the dataset, so that the RGB channels for each row of pixels within an image sums up to 1.
2. Scaling each image in the dataset, so that the RGB channels for all pixels within an image sums up to 1.

3. Subtracting the mean image of the dataset from each image in the dataset.
4. Setting all RGB values to either 0 or 1 depending on a given threshold.

9.2 Answer:

3. adım batch normalization'a benziyor çünkü her iki yöntem de veri setindeki tüm örnekler üzerinden hesaplanan ortalamayı kullanır.

1.adım ve 2. adım , her ikisi de layer normalization'a benzer. Çünkü, her iki yöntem de tek bir örnek içindeki tüm özellikleri dikkate aıyor. Ancak, 2. adım daha yakın bir benzerlik gösterir çünkü tüm özellikleri içeren genel bir normalleştirme yapar. 4. adım, batch normalization veya layer normalization'a benzemez. Bu adım, eşikleme işlemi olarak tanımlanır ve verileri ikili formata dönüştürür, bu da normalleştirme işleminden oldukça farklıdır.

10 Layer Normalization: Implementation

Now you'll implement layer normalization. This step should be relatively straightforward, as conceptually the implementation is almost identical to that of batch normalization. One significant difference though is that for layer normalization, we do not keep track of the moving moments, and the testing phase is identical to the training phase, where the mean and variance are directly calculated per datapoint.

Here's what you need to do:

- In `cs231n/layers.py`, implement the forward pass for layer normalization in the function `layernorm_forward`.

Run the cell below to check your results. * In `cs231n/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. * Modify `cs231n/classifiers/fc_net.py` to add layer normalization to the `FullyConnectedNet`. When the `normalization` flag is set to

"layernorm" in the constructor, you should insert a layer normalization layer before each ReLU nonlinearity.

Run the third cell below to run the batch size experiment on layer normalization.

[15]:
```
# Check the training-time forward pass by checking means and variances
# of features both before and after layer normalization.

# Simulate the forward pass for a two-layer network.
np.random.seed(231)
N, D1, D2, D3 =4, 50, 60, 3
X = np.random.randn(N, D1)
W1 = np.random.randn(D1, D2)
W2 = np.random.randn(D2, D3)
a = np.maximum(0, X.dot(W1)).dot(W2)

print('Before layer normalization:')
print_mean_std(a,axis=1)

gamma = np.ones(D3)
beta = np.zeros(D3)

# Means should be close to zero and stds close to one.
print('After layer normalization (gamma=1, beta=0)')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)

gamma = np.asarray([3.0,3.0,3.0])
beta = np.asarray([5.0,5.0,5.0])

# Now means should be close to beta and stds close to gamma.
print('After layer normalization (gamma=', gamma, ', beta=', beta, ')')
a_norm, _ = layernorm_forward(a, gamma, beta, {'mode': 'train'})
print_mean_std(a_norm,axis=1)
```

```
Before layer normalization:
  means: [-59.06673243 -47.60782686 -43.31137368 -26.40991744]
  stds:  [10.07429373 28.39478981 35.28360729  4.01831507]

After layer normalization (gamma=1, beta=0)
  means: [ 4.81096644e-16 -7.40148683e-17  2.22044605e-16 -5.92118946e-16]
  stds:  [0.99999995 0.99999999 1.         0.99999969]

After layer normalization (gamma= [3. 3. 3.] , beta= [5. 5. 5.] )
  means: [5. 5. 5. 5.]
  stds:  [2.99999985 2.99999998 2.99999999 2.99999907]
```

[16]:
```
# Gradient check batchnorm backward pass.
np.random.seed(231)
N, D = 4, 5
x = 5 * np.random.randn(N, D) + 12
gamma = np.random.randn(D)
beta = np.random.randn(D)
dout = np.random.randn(N, D)

ln_param = {}
fx = lambda x: layernorm_forward(x, gamma, beta, ln_param)[0]
fg = lambda a: layernorm_forward(x, a, beta, ln_param)[0]
fb = lambda b: layernorm_forward(x, gamma, b, ln_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma.copy(), dout)
db_num = eval_numerical_gradient_array(fb, beta.copy(), dout)

_, cache = layernorm_forward(x, gamma, beta, ln_param)
dx, dgamma, dbeta = layernorm_backward(dout, cache)

# You should expect to see relative errors between 1e-12 and 1e-8.
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  1.433616168873336e-09
dgamma error:  4.519489546032799e-12
dbeta error:  2.276445013433725e-12
```

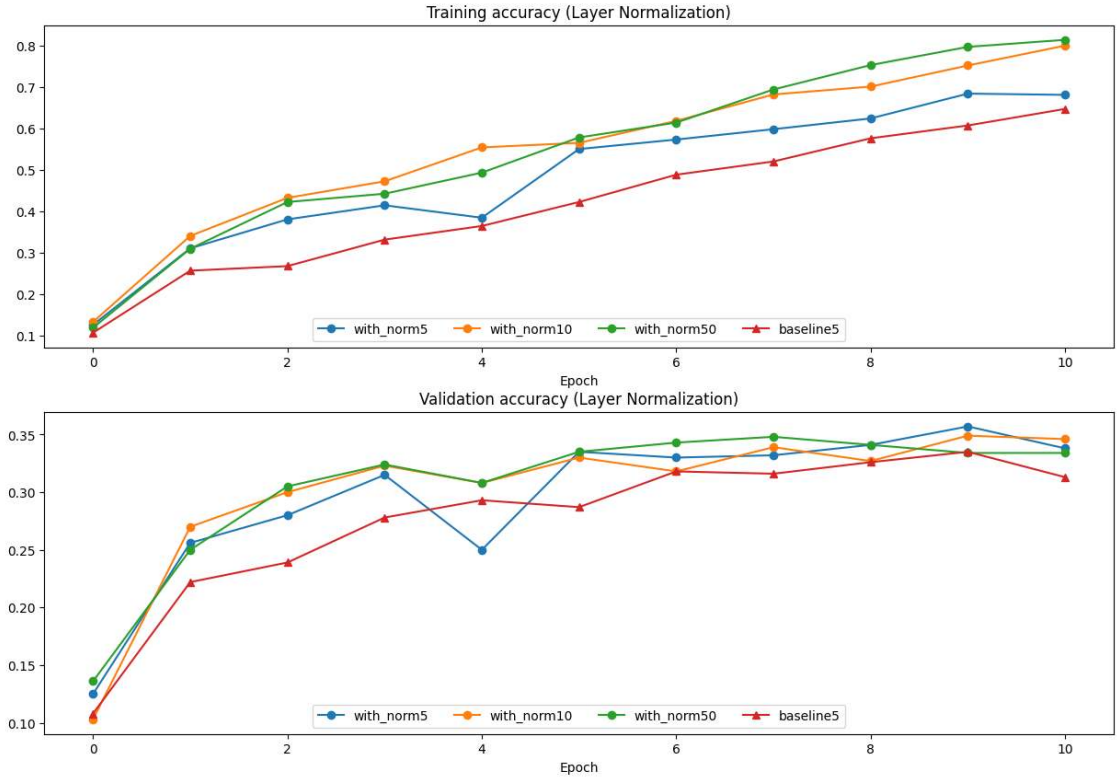
11 Layer Normalization and Batch Size

We will now run the previous batch size experiment with layer normalization instead of batch normalization. Compared to the previous experiment, you should see a markedly smaller influence of batch size on the training history!

[17]:
```
ln_solvers_bsize, solver_bsize, batch_sizes = ␣
 ↪run_batchsize_experiments('layernorm')

plt.subplot(2, 1, 1)
plot_training_history('Training accuracy (Layer Normalization)','Epoch',␣
 ↪solver_bsize, ln_solvers_bsize, \
                      lambda x: x.train_acc_history, bl_marker='-^',␣
 ↪bn_marker='-o', labels=batch_sizes)
plt.subplot(2, 1, 2)
plot_training_history('Validation accuracy (Layer Normalization)','Epoch',␣
 ↪solver_bsize, ln_solvers_bsize, \
```

```
                    lambda x: x.val_acc_history, bl_marker='-^',␣
↪bn_marker='-o', labels=batch_sizes)

plt.gcf().set_size_inches(15, 10)
plt.show()
```

```
No normalization: batch size =  5
Normalization: batch size =  5
Normalization: batch size =  10
Normalization: batch size =  50
```



11.1 Inline Question 4:

When is layer normalization likely to not work well, and why?

1. Using it in a very deep network
2. Having a very small dimension of features
3. Having a high regularization term

11.2 Answer:

Çok Derin Ağlarda Kullanımı: Katman normalizasyonu, çok derin ağlarda genellikle iyi çalışır.

Düşük Boyutlu Özelliklerin Olması: Layer normalization, özellik boyutu çok düşük olduğunda iyi

çalışmaz. Çünkü her bir katmanda sınırlı sayıda özellik üzerinden normalizasyon yapılır ve bu durumda, normalizasyon için yeterli veri olmaması nedeniyle, istenilen normalleştirme etkisi elde edilmez.

Yüksek Düzenleme Teriminin Olması: Layer normalization'ın performansını doğrudan etkileyen bir faktör değildir. Regularization terimi, modelin overfit olmasını önlemeye yardımcı olur, ancak layer normalization'ın çalışma prensibi üzerinde doğrudan bir etkisi yoktur. Layer normalization, her katmandaki aktivasyonların dağılımını düzenlemekle alakalıdır ve bu işlem düzenleme teriminden bağımsız olarak gerçekleşir.

[17]:

Dropout

November 29, 2023

[3]:
```
# 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/assignment1/'
FOLDERNAME = 'Colab Notebooks/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/Colab Notebooks/assignment2/cs231n/datasets
/content/drive/My Drive/Colab Notebooks/assignment2
```

1 Dropout

Dropout [1] is a technique for regularizing neural networks by randomly setting some output activations to zero during the forward pass. In this exercise, you will implement a dropout layer and modify your fully connected network to optionally use dropout.

[1] Geoffrey E. Hinton et al, "Improving neural networks by preventing co-adaptation of feature detectors", arXiv 2012

[4]:
```
# 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))))
```

[5]:
```
# 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,)
```

2 Dropout: Forward Pass

In the file `cs231n/layers.py`, implement the forward pass for dropout. Since dropout behaves differently during training and testing, make sure to implement the operation for both modes.

Once you have done so, run the cell below to test your implementation.

[6]:
```
np.random.seed(231)
x = np.random.randn(500, 500) + 10

for p in [0.25, 0.4, 0.7]:
    out, _ = dropout_forward(x, {'mode': 'train', 'p': p})
    out_test, _ = dropout_forward(x, {'mode': 'test', 'p': p})

    print('Running tests with p = ', p)
    print('Mean of input: ', x.mean())
    print('Mean of train-time output: ', out.mean())
```

```
    print('Mean of test-time output: ', out_test.mean())
    print('Fraction of train-time output set to zero: ', (out == 0).mean())
    print('Fraction of test-time output set to zero: ', (out_test == 0).mean())
    print()
```

```
Running tests with p =  0.25
Mean of input:  10.000207878477502
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.749784
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.4
Mean of input:  10.000207878477502
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.600796
Fraction of test-time output set to zero:  0.0

Running tests with p =  0.7
Mean of input:  10.000207878477502
Mean of train-time output:  9.987811912159426
Mean of test-time output:  10.000207878477502
Fraction of train-time output set to zero:  0.30074
Fraction of test-time output set to zero:  0.0
```

3 Dropout: Backward Pass

In the file `cs231n/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

[7]:
```
np.random.seed(231)
x = np.random.randn(10, 10) + 10
dout = np.random.randn(*x.shape)

dropout_param = {'mode': 'train', 'p': 0.2, 'seed': 123}
out, cache = dropout_forward(x, dropout_param)
dx = dropout_backward(dout, cache)
dx_num = eval_numerical_gradient_array(lambda xx: dropout_forward(xx,␣
 ↪dropout_param)[0], x, dout)

# Error should be around e-10 or less.
print('dx relative error: ', rel_error(dx, dx_num))
```

```
dx relative error:  5.44560814873387e-11
```

3.1 Inline Question 1:

What happens if we do not divide the values being passed through inverse dropout by `p` in the dropout layer? Why does that happen?

3.2 Answer:

Eğer inverse dropout işlemi sırasında değerleri p ile bölmeyip doğrudan kullanırsak, ağın aktivasyonlarının beklenen büyüklüğü değişir ve bu da modelin eğitimini olumsuz etkiler.

Dropout, her eğitim adımında rastgele bazı nöronları sıfırlayarak ağın belirli nöronlara aşırı bağımlı hale gelmesini önlemeyi amaçlar. Ters dropout yöntemi, aktif kalan nöronların çıktılarını 1/p ile çarparak, dropout uygulanmadan önceki aktivasyonların beklenen değerini korumayı hedefler. Burada p, bir nöronun aktif kalma olasılığıdır.

Eğer bu çarpanı (1/p) kullanmazsak, dropout uygulanan katmanlardan çıkan aktivasyonlar ortalama olarak daha düşük olacaktır. Bu, sonraki katmanlara daha az sinyal iletilmesine neden olur ve ağın öğrenme kapasitesini azaltır. Yani, modelin genel performansı azalır ve eğitim süreci daha yavaş veya daha az etkili hale gelebilir.

Bundan dolayı, ters dropout uygularken aktivasyonları 1/p ile çarpmak, ağın farklı dropout oranlarına rağmen tutarlı bir şekilde eğitilmesini sağlamak için önemlidir. Bu yaklaşım, eğitim sırasında ve eğitim sonrasında modelin daha iyi genelleme yapmasına yardımcı olur.

4 Fully Connected Networks with Dropout

In the file `cs231n/classifiers/fc_net.py`, modify your implementation to use dropout. Specifically, if the constructor of the network receives a value that is not 1 for the `dropout_keep_ratio` parameter, then the net should add a dropout layer immediately after every ReLU nonlinearity. After doing so, run the following to numerically gradient-check your implementation.

[8]:
```
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 dropout_keep_ratio in [1, 0.75, 0.5]:
    print('Running check with dropout = ', dropout_keep_ratio)
    model = FullyConnectedNet(
        [H1, H2],
        input_dim=D,
        num_classes=C,
        weight_scale=5e-2,
        dtype=np.float64,
        dropout_keep_ratio=dropout_keep_ratio,
        seed=123
    )

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

```
  # Relative errors should be around e-6 or less.
  # Note that it's fine if for dropout_keep_ratio=1 you have W2 error be on␣
↪the order of e-5.
  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('%s relative error: %.2e' % (name, rel_error(grad_num,␣
↪grads[name])))
  print()
```

```
Running check with dropout =  1
Initial loss:  2.300479089768492
W1 relative error: 1.03e-07
W2 relative error: 2.21e-05
W3 relative error: 4.56e-07
b1 relative error: 4.66e-09
b2 relative error: 2.09e-09
b3 relative error: 1.69e-10

Running check with dropout =  0.75
Initial loss:  2.302371489704412
W1 relative error: 1.85e-07
W2 relative error: 2.15e-06
W3 relative error: 4.56e-08
b1 relative error: 1.16e-08
b2 relative error: 1.82e-09
b3 relative error: 1.48e-10

Running check with dropout =  0.5
Initial loss:  2.30427592207859
W1 relative error: 3.11e-07
W2 relative error: 2.48e-08
W3 relative error: 6.43e-08
b1 relative error: 5.37e-09
b2 relative error: 1.91e-09
b3 relative error: 1.85e-10
```

5 Regularization Experiment

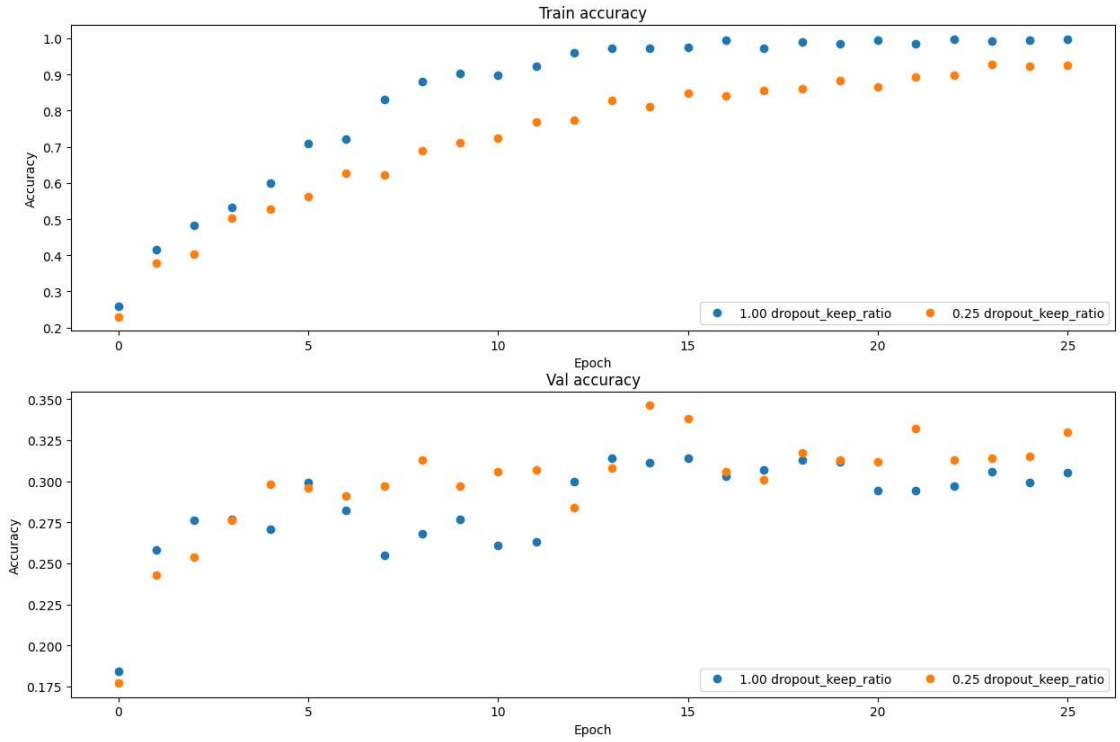
As an experiment, we will train a pair of two-layer networks on 500 training examples: one will use no dropout, and one will use a keep probability of 0.25. We will then visualize the training and validation accuracies of the two networks over time.

[9]:
```
# Train two identical nets, one with dropout and one without.
np.random.seed(231)
num_train = 500
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 = {}
dropout_choices = [1, 0.25]
for dropout_keep_ratio in dropout_choices:
    model = FullyConnectedNet(
        [500],
        dropout_keep_ratio=dropout_keep_ratio
    )
    print(dropout_keep_ratio)

    solver = Solver(
        model,
        small_data,
        num_epochs=25,
        batch_size=100,
        update_rule='adam',
        optim_config={'learning_rate': 5e-4,},
        verbose=True,
        print_every=100
    )
    solver.train()
    solvers[dropout_keep_ratio] = solver
    print()
```

```
1
(Iteration 1 / 125) loss: 7.856643
(Epoch 0 / 25) train acc: 0.260000; val_acc: 0.184000
(Epoch 1 / 25) train acc: 0.416000; val_acc: 0.258000
(Epoch 2 / 25) train acc: 0.482000; val_acc: 0.276000
(Epoch 3 / 25) train acc: 0.532000; val_acc: 0.277000
(Epoch 4 / 25) train acc: 0.600000; val_acc: 0.271000
(Epoch 5 / 25) train acc: 0.708000; val_acc: 0.299000
(Epoch 6 / 25) train acc: 0.722000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.880000; val_acc: 0.268000
(Epoch 9 / 25) train acc: 0.902000; val_acc: 0.277000
(Epoch 10 / 25) train acc: 0.898000; val_acc: 0.261000
(Epoch 11 / 25) train acc: 0.924000; val_acc: 0.263000
```

```
(Epoch 12 / 25) train acc: 0.960000; val_acc: 0.300000
(Epoch 13 / 25) train acc: 0.972000; val_acc: 0.314000
(Epoch 14 / 25) train acc: 0.972000; val_acc: 0.311000
(Epoch 15 / 25) train acc: 0.974000; val_acc: 0.314000
(Epoch 16 / 25) train acc: 0.994000; val_acc: 0.303000
(Epoch 17 / 25) train acc: 0.972000; val_acc: 0.307000
(Epoch 18 / 25) train acc: 0.990000; val_acc: 0.313000
(Epoch 19 / 25) train acc: 0.986000; val_acc: 0.312000
(Epoch 20 / 25) train acc: 0.994000; val_acc: 0.294000
(Iteration 101 / 125) loss: 0.002801
(Epoch 21 / 25) train acc: 0.986000; val_acc: 0.294000
(Epoch 22 / 25) train acc: 0.998000; val_acc: 0.297000
(Epoch 23 / 25) train acc: 0.992000; val_acc: 0.306000
(Epoch 24 / 25) train acc: 0.996000; val_acc: 0.299000
(Epoch 25 / 25) train acc: 0.998000; val_acc: 0.305000

0.25
(Iteration 1 / 125) loss: 17.318478
(Epoch 0 / 25) train acc: 0.230000; val_acc: 0.177000
(Epoch 1 / 25) train acc: 0.378000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.402000; val_acc: 0.254000
(Epoch 3 / 25) train acc: 0.502000; val_acc: 0.276000
(Epoch 4 / 25) train acc: 0.528000; val_acc: 0.298000
(Epoch 5 / 25) train acc: 0.562000; val_acc: 0.296000
(Epoch 6 / 25) train acc: 0.626000; val_acc: 0.291000
(Epoch 7 / 25) train acc: 0.622000; val_acc: 0.297000
(Epoch 8 / 25) train acc: 0.688000; val_acc: 0.313000
(Epoch 9 / 25) train acc: 0.712000; val_acc: 0.297000
(Epoch 10 / 25) train acc: 0.724000; val_acc: 0.306000
(Epoch 11 / 25) train acc: 0.768000; val_acc: 0.307000
(Epoch 12 / 25) train acc: 0.774000; val_acc: 0.284000
(Epoch 13 / 25) train acc: 0.828000; val_acc: 0.308000
(Epoch 14 / 25) train acc: 0.812000; val_acc: 0.346000
(Epoch 15 / 25) train acc: 0.848000; val_acc: 0.338000
(Epoch 16 / 25) train acc: 0.842000; val_acc: 0.306000
(Epoch 17 / 25) train acc: 0.856000; val_acc: 0.301000
(Epoch 18 / 25) train acc: 0.860000; val_acc: 0.317000
(Epoch 19 / 25) train acc: 0.882000; val_acc: 0.313000
(Epoch 20 / 25) train acc: 0.866000; val_acc: 0.312000
(Iteration 101 / 125) loss: 4.185210
(Epoch 21 / 25) train acc: 0.894000; val_acc: 0.332000
(Epoch 22 / 25) train acc: 0.898000; val_acc: 0.313000
(Epoch 23 / 25) train acc: 0.928000; val_acc: 0.314000
(Epoch 24 / 25) train acc: 0.922000; val_acc: 0.315000
(Epoch 25 / 25) train acc: 0.926000; val_acc: 0.330000
```

[10]:
```
# Plot train and validation accuracies of the two models.
train_accs = []
val_accs = []
for dropout_keep_ratio in dropout_choices:
    solver = solvers[dropout_keep_ratio]
    train_accs.append(solver.train_acc_history[-1])
    val_accs.append(solver.val_acc_history[-1])

plt.subplot(3, 1, 1)
for dropout_keep_ratio in dropout_choices:
    plt.plot(
        solvers[dropout_keep_ratio].train_acc_history, 'o', label='%.2f␣
 ↪dropout_keep_ratio' % dropout_keep_ratio)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.subplot(3, 1, 2)
for dropout_keep_ratio in dropout_choices:
    plt.plot(
        solvers[dropout_keep_ratio].val_acc_history, 'o', label='%.2f␣
 ↪dropout_keep_ratio' % dropout_keep_ratio)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

plt.gcf().set_size_inches(15, 15)
plt.show()
```



5.1 Inline Question 2:

Compare the validation and training accuracies with and without dropout – what do your results suggest about dropout as a regularizer?

5.2 Answer:

Dropout, bir regularizer olarak kullanıldığında, modelin overfit olmasını önlemeye yardımcı olur. Dropout ile ve dropoutsuz modellerin validation ve training başarı oranlarını karşılaştırmak, dropout'un bu düzenleyici etkisini ortaya koymaktadır.

Dropoutsuz Model: Eğer modelde dropout kullanılmazsa, genellikle eğitim başarısı doğrulama başarısından aşırı yüksek olur. Bu, modelin overfit olduğunu gösterir. Yani, model eğitim verilerini ezberler ve validation setinde iyi performans gösteremez.

Dropoutlu Model: Dropout uygulandığında, modelin eğitim başarısı genellikle bir miktar düşer, çünkü her eğitim anında rastgele bazı nöronlar devre dışı bırakılır. Ancak bu, modelin genelleme yeteneğini artırır ve validation setindeki başarı oranını yükseltir. Dropout, modelin overfit olmasını engeller ve daha iyi bir öğrenme sağlar.

ConvolutionalNetworks

November 29, 2023

[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/assignment1/'
FOLDERNAME = 'Colab Notebooks/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
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
/content/drive/My Drive/Colab Notebooks/assignment2/cs231n/datasets
/content/drive/My Drive/Colab Notebooks/assignment2
```

1 Convolutional Networks

So far we have worked with deep fully connected networks, using them to explore different optimization strategies and network architectures. Fully connected networks are a good testbed for experimentation because they are very computationally efficient, but in practice all state-of-the-art results use convolutional networks instead.

First you will implement several layer types that are used in convolutional networks. You will then use these layers to train a convolutional network on the CIFAR-10 dataset.

[2]:
```
# Setup cell.
import numpy as np
```

```
import matplotlib.pyplot as plt
from cs231n.classifiers.cnn import *
from cs231n.data_utils import get_CIFAR10_data
from cs231n.gradient_check import eval_numerical_gradient_array,␣
 ↪eval_numerical_gradient
from cs231n.layers import *
from cs231n.fast_layers import *
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'

# for auto-reloading external modules
# see http://stackoverflow.com/questions/1907993/
 ↪autoreload-of-modules-in-ipython
%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,)
```

2 Convolution: Naive Forward Pass

The core of a convolutional network is the convolution operation. In the file `cs231n/layers.py`, implement the forward pass for the convolution layer in the function `conv_forward_naive`.

You don't have to worry too much about efficiency at this point; just write the code in whatever way you find most clear.

You can test your implementation by running the following:

[4]:
```
!pip install numpy --upgrade
from numpy.lib.stride_tricks import sliding_window_view
```

```
x = np.arange(4*4*3).reshape(3, 4, 4)
x = np.pad(x, ((0,), (1,), (1,)))

# print(x)

# 1 - up/down; 2- left/right, 3 - forth/back
# N, C, H, W
filters_dim = (1, 1, 2, 2)
M3, K3, K1, K2 = filters_dim

inputs_dim = (1, 1, 4, 4)
N, N3, N1, N2 = inputs_dim



# H, W
stride = (1, 1)
S1, S2 = stride

n_every1 = append_after_every_index = S1 - 1
n_every2 = append_after_every_index = S2 - 1

pad = (0, 0)
P1, P2 = pad

# Shapes
M1 = (N1 - K1 + 2 * P1) // S1 + 1
M2 = (N2 - K2 + 2 * P2) // S2 + 1

transform_shape = (N, K3, K1, K2, 1, 1, M1, M2)
is1 = np.arange(1, M1)
is2 = np.arange(1, M2)

# Filters
filters = np.arange(np.prod(filters_dim)).reshape(filters_dim)
print('Original filters:\n', filters)

filters_reshaped = filters.reshape(M3, K1*K2*K3)
# print('\nReshaped filters:\n', filters_reshaped)

# Images
inputs = np.arange(np.prod(inputs_dim)).reshape(inputs_dim)
inputs = np.pad(inputs, pad_width=((0,), (0,), (P1,), (P2,)), mode='constant')
print(f'\nOriginal inputs {inputs.shape}:\n', inputs)
```

```
inputs_reshaped = sliding_window_view(inputs.T, window_shape=(K2, K1, K3, N)).
 ↪T#[..., ::S1, ::S2]#.reshape(N, K1*K2*K3, M1*M2)
#print(f'\nUnshaped inputs: {inputs_reshaped.shape}\n', inputs_reshaped)
inputs_reshaped = inputs_reshaped[..., ::S1, ::S2].reshape(N, K1*K2*K3, M1*M2)
# inputs_reshaped = sliding_window_view(inputs, window_shape=(K3, K1, K2),␣
 ↪axis=(1,2,3))[:, :, ::S1, ::S2]#.reshape((N, M1*M2, K1*K2*K3)).transpose((0,␣
 ↪2, 1))
#print(f'\nUntransposed inputs: {inputs_reshaped.shape}\n', inputs_reshaped)

# inputs_reshaped = inputs_reshaped.reshape((N, M1*M2, K1*K2*K3)).transpose((0,␣
 ↪2, 1))
# print(f'\nReshaped inputs: {inputs_reshaped.shape}\n', inputs_reshaped)

# reshaped_images = sliding_window_view(images, window_shape=(2, 2, 2),␣
 ↪axis=(1, 2, 3))[:, :, ::2, ::2].reshape((2, 4, 2*2*2))


# Activation maps
activation_maps = filters_reshaped @ inputs_reshaped
# activation_maps = np.tensordot(inputs_reshaped, filters_reshaped, axes=1)
# print('\nReshaped activation maps:\n', activation_maps)

activation_maps_reshaped = activation_maps.reshape((N, M3, M1, M2))
activation_maps_reshaped = np.ones_like(activation_maps_reshaped)
print(f'\nActivation maps {activation_maps_reshaped.shape}:\n',␣
 ↪activation_maps_reshaped)

# Derivatives
dx_reshaped = filters_reshaped.T @ activation_maps
# print('\ndx unshaped:\n', dx_reshaped)


dx_reshaped = np.expand_dims(dx_reshaped, axis=(0, 1, 2, 3)).
 ↪reshape(transform_shape)
# print('\ndx reshaped:\n', dx_reshaped)


if S1 > 1:
  dx_reshaped = np.insert(dx_reshaped, is1, [[0]]*n_every1, axis=6)

if S2 > 1:
  dx_reshaped = np.insert(dx_reshaped, is2, [[0]]*n_every2, axis=7)


dx_reshaped = dx_reshaped
```

```
#dx_reshaped = np.insert(dx_reshaped, i2, [[0]]*n_last2, axis=7)
#dx_reshaped = np.insert(dx_reshaped, is2, [[0]]*n_every2, axis=7)


#dx_reshaped = np.insert(dx_reshaped, (1, 2), 0, axis=7)
# print(f'\ndx unshaped: {dx_reshaped.shape}\n', dx_reshaped)


filter_new = np.rot90(filters, 2, axes=(2, 3))
# print(f'\nfilter flipped: {filter_new.shape}\n', filter_new)


filter_new_reshaped = filter_new.reshape(M3, K3*K1*K2)
filter_new_reshaped = np.concatenate(filter_new_reshaped.reshape(M3, K3, -1),␣
 ↪axis=1)
# print(f'\nfilter flipped reshaped: {filter_new_reshaped.shape}\n',␣
 ↪filter_new_reshaped)


activations_new = np.copy(activation_maps_reshaped)

if S1 > 1:
    activations_new = np.insert(activations_new, range(1, M1), [[0]]*n_every1,␣
 ↪axis=2)

if S2 > 1:
    activations_new = np.insert(activations_new, range(1, M2), [[0]]*n_every2,␣
 ↪axis=3)


# dout_row = np.concatenate(activations_new.reshape(N, M3, -1), axis=1)
dout_row = activations_new.reshape(N, 1, M3, -1)
#print(f'\ndout_row {dout_row .shape}:\n', dout_row)

x_col = sliding_window_view(inputs.T, window_shape=(activations_new.shape[3],␣
 ↪activations_new.shape[2], K3, N)).T.reshape((N, K3, activations_new.shape[3]␣
 ↪* activations_new.shape[2], -1))
#print(f'\nx_col {x_col.shape}:\n', x_col)

dw = np.moveaxis((dout_row @ x_col).sum(axis=0), 1, 0).reshape((M3, K3, K1, K2))
#print(f'\ndw {dw.shape}:\n', dw)


inputs_third = sliding_window_view(inputs.T, window_shape=(activations_new.
 ↪shape[3], activations_new.shape[2], K3, N)).T.reshape((N, K3,␣
 ↪activations_new.shape[3] * activations_new.shape[2], -1))
```

```
# print(f'\ndw {dw.shape}:\n', dw)

inputs_alt = sliding_window_view(inputs, window_shape=(N, K3, activations_new.
 ↪shape[2], activations_new.shape[3]))
# print(f'\nInputs reshaped {inputs_alt.shape}:\n', inputs_alt)
dw_alt = np.einsum('ijkl,mnopiqkl->jqop', activations_new, inputs_alt)
print(f'\ndw alt {dw_alt.shape}:\n', dw_alt)

activations_new = np.pad(activations_new, pad_width=((0,), (0,), (K1-1,),␣
 ↪(K2-1,)), mode='constant')
# print(f'\nactivations padded: {activations_new.shape}\n', activations_new)

M11 = (M1 - K1 + 2 * (K1-1)) + 1
M21 = (M2 - K2 + 2 * (K2-1)) + 1
# print(f'Expected area: {(M11, M21)}')

activations_reshaped = sliding_window_view(activations_new.T, window_shape=(K2,␣
 ↪K1, M3, N)).T.reshape(N, M3*K1*K2, -1)
# print(f'\nactivations reshaped: {activations_reshaped.shape}\n',␣
 ↪activations_reshaped)

activations_alt = sliding_window_view(activations_new, window_shape=(N, M3, K1,␣
 ↪K2))
# print(f'\nactivations alt: {activations_alt.shape}\n', activations_alt)

dx_reshaped = filter_new_reshaped @ activations_reshaped
dx = dx_reshaped.reshape((N, K3, N1, N2))
# print(f'\ndx: {dx.shape}\n', dx)

# M3, K3, K1, K2 | 1, 1, N1 N2 N M3 K1 K2
dx_alt = np.einsum('ijkl,mnopqikl->qjop', filter_new, activations_alt)
# print(f'\ndx alt: {dx_alt.shape}\n', dx_alt)
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages
(1.26.2)
Original filters:
 [[[[0 1]
    [2 3]]]]

Original inputs (1, 1, 4, 4):
 [[[[ 0  1  2  3]
    [ 4  5  6  7]
    [ 8  9 10 11]
    [12 13 14 15]]]]

Activation maps (1, 1, 3, 3):
 [[[[1 1 1]
```

```
   [1 1 1]
   [1 1 1]]]]

dw alt (1, 1, 2, 2):
 [[[[45 54]
   [81 90]]]]
```

[5]:
```
x_shape = (2, 3, 4, 4)
w_shape = (3, 3, 4, 4)
x = np.linspace(-0.1, 0.5, num=np.prod(x_shape)).reshape(x_shape)
w = np.linspace(-0.2, 0.3, num=np.prod(w_shape)).reshape(w_shape)
b = np.linspace(-0.1, 0.2, num=3)

conv_param = {'stride': 2, 'pad': 1}
out, _ = conv_forward_naive(x, w, b, conv_param)
correct_out = np.array([[[[-0.08759809, -0.10987781],
                          [-0.18387192, -0.2109216 ]],
                         [[ 0.21027089,  0.21661097],
                          [ 0.22847626,  0.23004637]],
                         [[ 0.50813986,  0.54309974],
                          [ 0.64082444,  0.67101435]]],
                        [[[-0.98053589, -1.03143541],
                          [-1.19128892, -1.24695841]],
                         [[ 0.69108355,  0.66880383],
                          [ 0.59480972,  0.56776003]],
                         [[ 2.36270298,  2.36904306],
                          [ 2.38090835,  2.38247847]]]])

# Compare your output to ours; difference should be around e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

```
Testing conv_forward_naive
difference:  2.2121476417505994e-08
```

2.1 Aside: Image Processing via Convolutions

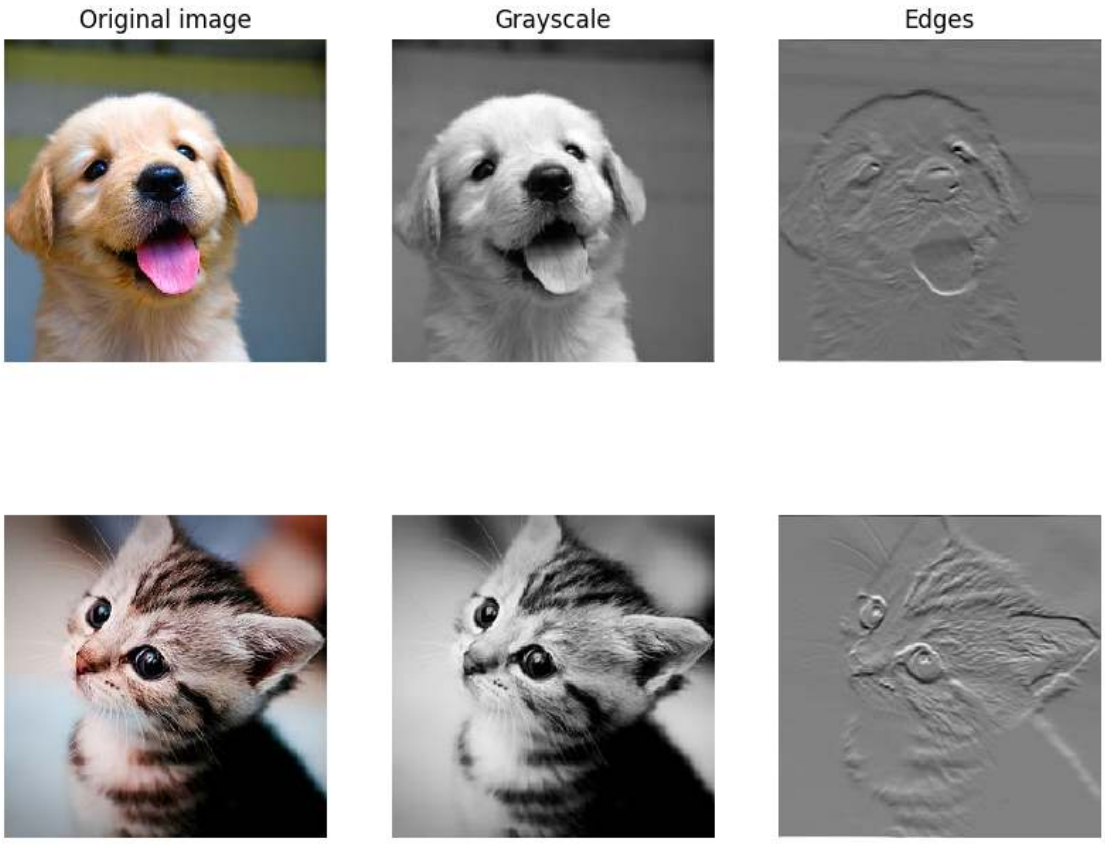
As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection). The convolution forward pass will apply these operations to each of the input images. We can then visualize the results as a sanity check.

[6]:
```
from imageio import imread
from PIL import Image

kitten = imread('cs231n/notebook_images/kitten.jpg')
puppy = imread('cs231n/notebook_images/puppy.jpg')
```

```
# kitten is wide, and puppy is already square
d = kitten.shape[1] - kitten.shape[0]
kitten_cropped = kitten[:, d//2:-d//2, :]

img_size = 200   # Make this smaller if it runs too slow
resized_puppy = np.array(Image.fromarray(puppy).resize((img_size, img_size)))
resized_kitten = np.array(Image.fromarray(kitten_cropped).resize((img_size,␣
 ↪img_size)))
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = resized_puppy.transpose((2, 0, 1))
x[1, :, :, :] = resized_kitten.transpose((2, 0, 1))

# Set up a convolutional weights holding 2 filters, each 3x3
w = np.zeros((2, 3, 3, 3))

# The first filter converts the image to grayscale.
# Set up the red, green, and blue channels of the filter.
w[0, 0, :, :] = [[0, 0, 0], [0, 0.3, 0], [0, 0, 0]]
w[0, 1, :, :] = [[0, 0, 0], [0, 0.6, 0], [0, 0, 0]]
w[0, 2, :, :] = [[0, 0, 0], [0, 0.1, 0], [0, 0, 0]]

# Second filter detects horizontal edges in the blue channel.
w[1, 2, :, :] = [[1, 2, 1], [0, 0, 0], [-1, -2, -1]]

# Vector of biases. We don't need any bias for the grayscale
# filter, but for the edge detection filter we want to add 128
# to each output so that nothing is negative.
b = np.array([0, 128])

# Compute the result of convolving each input in x with each filter in w,
# offsetting by b, and storing the results in out.
out, _ = conv_forward_naive(x, w, b, {'stride': 1, 'pad': 1})

def imshow_no_ax(img, normalize=True):
    """ Tiny helper to show images as uint8 and remove axis labels """
    if normalize:
        img_max, img_min = np.max(img), np.min(img)
        img = 255.0 * (img - img_min) / (img_max - img_min)
    plt.imshow(img.astype('uint8'))
    plt.gca().axis('off')

# Show the original images and the results of the conv operation
plt.subplot(2, 3, 1)
imshow_no_ax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_no_ax(out[0, 0])
```

```
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_no_ax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_no_ax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_no_ax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_no_ax(out[1, 1])
plt.show()
```

```
<ipython-input-6-7950733600c3>:4: DeprecationWarning: Starting with ImageIO v3
the behavior of this function will switch to that of iio.v3.imread. To keep the
current behavior (and make this warning disappear) use `import imageio.v2 as
imageio` or call `imageio.v2.imread` directly.
  kitten = imread('cs231n/notebook_images/kitten.jpg')
<ipython-input-6-7950733600c3>:5: DeprecationWarning: Starting with ImageIO v3
the behavior of this function will switch to that of iio.v3.imread. To keep the
current behavior (and make this warning disappear) use `import imageio.v2 as
imageio` or call `imageio.v2.imread` directly.
  puppy = imread('cs231n/notebook_images/puppy.jpg')
```



3 Convolution: Naive Backward Pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `cs231n/layers.py`. Again, you don't need to worry too much about computational efficiency.

When you are done, run the following to check your backward pass with a numeric gradient check.

[7]:
```
np.random.seed(231)
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

dx_num = eval_numerical_gradient_array(lambda x: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b,␣
 ↪conv_param)[0], b, dout)

out, cache = conv_forward_naive(x, w, b, conv_param)
dx, dw, db = conv_backward_naive(dout, cache)

# Your errors should be around e-8 or less.
print('Testing conv_backward_naive function')
print('dx error: ', rel_error(dx, dx_num))
print('dw error: ', rel_error(dw, dw_num))
print('db error: ', rel_error(db, db_num))
```

```
Testing conv_backward_naive function
dx error:  1.159803161159293e-08
dw error:  2.2471264748452487e-10
db error:  3.37264006649648e-11
```

4 Max-Pooling: Naive Forward Pass

Implement the forward pass for the max-pooling operation in the function `max_pool_forward_naive` in the file `cs231n/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

[8]:
```
x_shape = (2, 3, 4, 4)
x = np.linspace(-0.3, 0.4, num=np.prod(x_shape)).reshape(x_shape)
pool_param = {'pool_width': 2, 'pool_height': 2, 'stride': 2}

out, _ = max_pool_forward_naive(x, pool_param)

correct_out = np.array([[[[-0.26315789, -0.24842105],
                          [-0.20421053, -0.18947368]],
                         [[-0.14526316, -0.13052632],
                          [-0.08631579, -0.07157895]],
                         [[-0.02736842, -0.01263158],
                          [ 0.03157895,  0.04631579]]],
                        [[[ 0.09052632,  0.10526316],
                          [ 0.14947368,  0.16421053]],
                         [[ 0.20842105,  0.22315789],
                          [ 0.26736842,  0.28210526]],
                         [[ 0.32631579,  0.34105263],
                          [ 0.38526316,  0.4       ]]]])

# Compare your output with ours. Difference should be on the order of e-8.
print('Testing max_pool_forward_naive function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing max_pool_forward_naive function:
difference:  4.1666665157267834e-08
```

5 Max-Pooling: Naive Backward

Implement the backward pass for the max-pooling operation in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. You don't need to worry about computational efficiency.

Check your implementation with numeric gradient checking by running the following:

[9]:
```
np.random.seed(231)
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

dx_num = eval_numerical_gradient_array(lambda x: max_pool_forward_naive(x,␣
 ↪pool_param)[0], x, dout)

out, cache = max_pool_forward_naive(x, pool_param)
dx = max_pool_backward_naive(dout, cache)

# Your error should be on the order of e-12
print('Testing max_pool_backward_naive function:')
```

```
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  1.0
```

6 Fast Layers

Making convolution and pooling layers fast can be challenging. To spare you the pain, we've provided fast implementations of the forward and backward passes for convolution and pooling layers in the file `cs231n/fast_layers.py`.

6.0.1 Execute the below cell, save the notebook, and restart the runtime

The fast convolution implementation depends on a Cython extension; to compile it, run the cell below. Next, save the Colab notebook (`File > Save`) and **restart the runtime** (`Runtime > Restart runtime`). You can then re-execute the preceeding cells from top to bottom and skip the cell below as you only need to run it once for the compilation step.

[10]:
```
# Remember to restart the runtime after executing this cell!
%cd /content/drive/My\ Drive/$FOLDERNAME/cs231n/
!python setup.py build_ext --inplace
%cd /content/drive/My\ Drive/$FOLDERNAME/
```

```
/content/drive/My Drive/Colab Notebooks/assignment2/cs231n
/content/drive/My Drive/Colab Notebooks/assignment2
```

The API for the fast versions of the convolution and pooling layers is exactly the same as the naive versions that you implemented above: the forward pass receives data, weights, and parameters and produces outputs and a cache object; the backward pass recieves upstream derivatives and the cache object and produces gradients with respect to the data and weights.

Note: The fast implementation for pooling will only perform optimally if the pooling regions are non-overlapping and tile the input. If these conditions are not met then the fast pooling implementation will not be much faster than the naive implementation.

You can compare the performance of the naive and fast versions of these layers by running the following:

[11]:
```
# Rel errors should be around e-9 or less.
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time
np.random.seed(231)
x = np.random.randn(100, 3, 31, 31)
w = np.random.randn(25, 3, 3, 3)
b = np.random.randn(25,)
dout = np.random.randn(100, 25, 16, 16)
conv_param = {'stride': 2, 'pad': 1}

t0 = time()
```

```
out_naive, cache_naive = conv_forward_naive(x, w, b, conv_param)
t1 = time()
out_fast, cache_fast = conv_forward_fast(x, w, b, conv_param)
t2 = time()

print('Testing conv_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('Difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive, dw_naive, db_naive = conv_backward_naive(dout, cache_naive)
t1 = time()
dx_fast, dw_fast, db_fast = conv_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting conv_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('Fast: %fs' % (t2 - t1))
print('Speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
print('dw difference: ', rel_error(dw_naive, dw_fast))
print('db difference: ', rel_error(db_naive, db_fast))
```

```
Testing conv_forward_fast:
Naive: 8.628166s
Fast: 0.015547s
Speedup: 554.980232x
Difference:  4.926407851494105e-11

Testing conv_backward_fast:
Naive: 16.447939s
Fast: 0.010773s
Speedup: 1526.782251x
dx difference:  1.949764775345631e-11
dw difference:  3.681156828004736e-13
db difference:  3.481354613192702e-14
```

[12]:
```
# Relative errors should be close to 0.0.
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast
np.random.seed(231)
x = np.random.randn(100, 3, 32, 32)
dout = np.random.randn(100, 3, 16, 16)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

t0 = time()
```

```
out_naive, cache_naive = max_pool_forward_naive(x, pool_param)
t1 = time()
out_fast, cache_fast = max_pool_forward_fast(x, pool_param)
t2 = time()

print('Testing pool_forward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('difference: ', rel_error(out_naive, out_fast))

t0 = time()
dx_naive = max_pool_backward_naive(dout, cache_naive)
t1 = time()
dx_fast = max_pool_backward_fast(dout, cache_fast)
t2 = time()

print('\nTesting pool_backward_fast:')
print('Naive: %fs' % (t1 - t0))
print('fast: %fs' % (t2 - t1))
print('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.377097s
fast: 0.004930s
speedup: 76.486242x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.887300s
fast: 0.020490s
speedup: 43.304174x
dx difference:  1.0
```

7 Convolutional "Sandwich" Layers

In the previous assignment, we introduced the concept of "sandwich" layers that combine multiple operations into commonly used patterns. In the file `cs231n/layer_utils.py` you will find sandwich layers that implement a few commonly used patterns for convolutional networks. Run the cells below to sanity check their usage.

[13]:
```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward
np.random.seed(231)
x = np.random.randn(2, 3, 16, 16)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
```

```
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

out, cache = conv_relu_pool_forward(x, w, b, conv_param, pool_param)
dx, dw, db = conv_relu_pool_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_pool_forward(x, w,␣
 ↪b, conv_param, pool_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w,␣
 ↪b, conv_param, pool_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w,␣
 ↪b, conv_param, pool_param)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu_pool')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu_pool
dx error:  9.591132621921372e-09
dw error:  5.802391137330214e-09
db error:  1.0146343411762047e-09
```

[14]:
```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward
np.random.seed(231)
x = np.random.randn(2, 3, 8, 8)
w = np.random.randn(3, 3, 3, 3)
b = np.random.randn(3,)
dout = np.random.randn(2, 3, 8, 8)
conv_param = {'stride': 1, 'pad': 1}

out, cache = conv_relu_forward(x, w, b, conv_param)
dx, dw, db = conv_relu_backward(dout, cache)

dx_num = eval_numerical_gradient_array(lambda x: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b,␣
 ↪conv_param)[0], b, dout)

# Relative errors should be around e-8 or less
print('Testing conv_relu:')
print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
```

```
print('db error: ', rel_error(db_num, db))
```

```
Testing conv_relu:
dx error:  1.5218619980349303e-09
dw error:  2.702022646099404e-10
db error:  1.451272393591721e-10
```

8 Three-Layer Convolutional Network

Now that you have implemented all the necessary layers, we can put them together into a simple convolutional network.

Open the file `cs231n/classifiers/cnn.py` and complete the implementation of the `ThreeLayerConvNet` class. Remember you can use the fast/sandwich layers (already imported for you) in your implementation. Run the following cells to help you debug:

8.1 Sanity Check Loss

After you build a new network, one of the first things you should do is sanity check the loss. When we use the softmax loss, we expect the loss for random weights (and no regularization) to be about `log(C)` for `C` classes. When we add regularization the loss should go up slightly.

[15]:
```
model = ThreeLayerConvNet()

N = 50
X = np.random.randn(N, 3, 32, 32)
y = np.random.randint(10, size=N)

loss, grads = model.loss(X, y)
print('Initial loss (no regularization): ', loss)

model.reg = 0.5
loss, grads = model.loss(X, y)
print('Initial loss (with regularization): ', loss)
```

```
Initial loss (no regularization):  2.302586071243987
Initial loss (with regularization):  2.508255638232932
```

8.2 Gradient Check

After the loss looks reasonable, use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer. Note: correct implementations may still have relative errors up to the order of e-2.

[16]:
```
num_inputs = 2
input_dim = (3, 16, 16)
reg = 0.0
num_classes = 10
```

```
np.random.seed(231)
X = np.random.randn(num_inputs, *input_dim)
y = np.random.randint(num_classes, size=num_inputs)

model = ThreeLayerConvNet(
    num_filters=3,
    filter_size=3,
    input_dim=input_dim,
    hidden_dim=7,
    dtype=np.float64
)
loss, grads = model.loss(X, y)
# Errors should be small, but correct implementations may have
# relative errors up to the order of e-2
for param_name in sorted(grads):
    f = lambda _: model.loss(X, y)[0]
    param_grad_num = eval_numerical_gradient(f, model.params[param_name],␣
 ↪verbose=False, h=1e-6)
    e = rel_error(param_grad_num, grads[param_name])
    print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
 ↪grads[param_name])))
```

```
W1 max relative error: 3.053965e-04
W2 max relative error: 1.822723e-02
W3 max relative error: 3.422399e-04
b1 max relative error: 3.397321e-06
b2 max relative error: 2.517459e-03
b3 max relative error: 9.711800e-10
```

8.3 Overfit Small Data

A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

[17]:
```
np.random.seed(231)

num_train = 100
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'],
}

model = ThreeLayerConvNet(weight_scale=1e-2)

solver = Solver(
```

```
    model,
    small_data,
    num_epochs=15,
    batch_size=50,
    update_rule='adam',
    optim_config={'learning_rate': 1e-3,},
    verbose=True,
    print_every=1
)
solver.train()
```

```
(Iteration 1 / 30) loss: 2.414060
(Epoch 0 / 15) train acc: 0.200000; val_acc: 0.137000
(Iteration 2 / 30) loss: 3.102925
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270330
(Iteration 4 / 30) loss: 2.096705
(Epoch 2 / 15) train acc: 0.240000; val_acc: 0.094000
(Iteration 5 / 30) loss: 1.838880
(Iteration 6 / 30) loss: 1.934188
(Epoch 3 / 15) train acc: 0.510000; val_acc: 0.173000
(Iteration 7 / 30) loss: 1.827912
(Iteration 8 / 30) loss: 1.639574
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.188000
(Iteration 9 / 30) loss: 1.330082
(Iteration 10 / 30) loss: 1.756115
(Epoch 5 / 15) train acc: 0.630000; val_acc: 0.167000
(Iteration 11 / 30) loss: 1.024162
(Iteration 12 / 30) loss: 1.041826
(Epoch 6 / 15) train acc: 0.750000; val_acc: 0.229000
(Iteration 13 / 30) loss: 1.142777
(Iteration 14 / 30) loss: 0.835706
(Epoch 7 / 15) train acc: 0.790000; val_acc: 0.247000
(Iteration 15 / 30) loss: 0.587786
(Iteration 16 / 30) loss: 0.645509
(Epoch 8 / 15) train acc: 0.820000; val_acc: 0.252000
(Iteration 17 / 30) loss: 0.786844
(Iteration 18 / 30) loss: 0.467054
(Epoch 9 / 15) train acc: 0.820000; val_acc: 0.178000
(Iteration 19 / 30) loss: 0.429880
(Iteration 20 / 30) loss: 0.635498
(Epoch 10 / 15) train acc: 0.900000; val_acc: 0.206000
(Iteration 21 / 30) loss: 0.365807
(Iteration 22 / 30) loss: 0.284220
(Epoch 11 / 15) train acc: 0.820000; val_acc: 0.201000
(Iteration 23 / 30) loss: 0.469343
(Iteration 24 / 30) loss: 0.509369
```

```
(Epoch 12 / 15) train acc: 0.920000; val_acc: 0.211000
(Iteration 25 / 30) loss: 0.111638
(Iteration 26 / 30) loss: 0.145388
(Epoch 13 / 15) train acc: 0.930000; val_acc: 0.213000
(Iteration 27 / 30) loss: 0.155575
(Iteration 28 / 30) loss: 0.143398
(Epoch 14 / 15) train acc: 0.960000; val_acc: 0.212000
(Iteration 29 / 30) loss: 0.158160
(Iteration 30 / 30) loss: 0.118934
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.220000
```

[18]:
```
# Print final training accuracy.
print(
    "Small data training accuracy:",
    solver.check_accuracy(small_data['X_train'], small_data['y_train'])
)
```

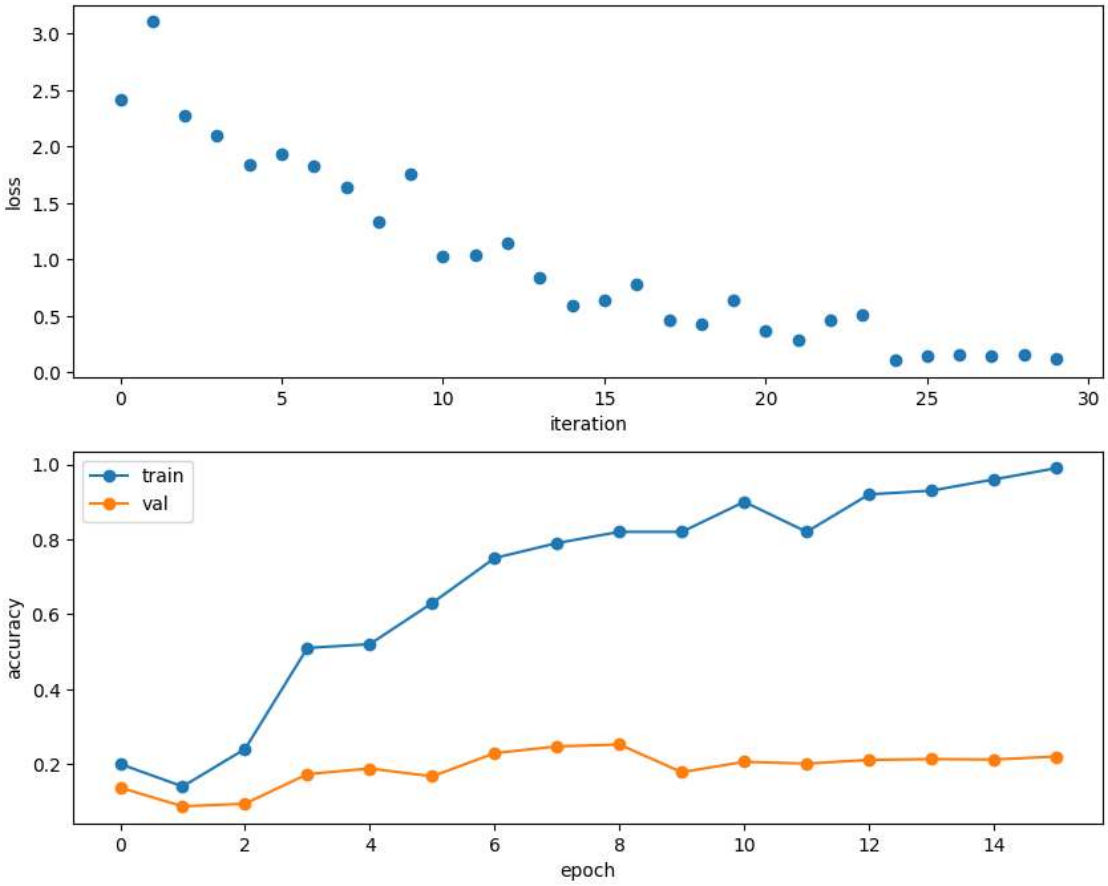
```
Small data training accuracy: 0.82
```

[19]:
```
# Print final validation accuracy.
print(
    "Small data validation accuracy:",
    solver.check_accuracy(small_data['X_val'], small_data['y_val'])
)
```

```
Small data validation accuracy: 0.252
```

Plotting the loss, training accuracy, and validation accuracy should show clear overfitting:

[20]:
```
plt.subplot(2, 1, 1)
plt.plot(solver.loss_history, 'o')
plt.xlabel('iteration')
plt.ylabel('loss')

plt.subplot(2, 1, 2)
plt.plot(solver.train_acc_history, '-o')
plt.plot(solver.val_acc_history, '-o')
plt.legend(['train', 'val'], loc='upper left')
plt.xlabel('epoch')
plt.ylabel('accuracy')
plt.show()
```



8.4 Train the Network

By training the three-layer convolutional network for one epoch, you should achieve greater than 40% accuracy on the training set:

[21]:
```
model = ThreeLayerConvNet(weight_scale=0.001, hidden_dim=500, reg=0.001)

solver = Solver(
    model,
    data,
    num_epochs=1,
    batch_size=50,
    update_rule='adam',
    optim_config={'learning_rate': 1e-3,},
    verbose=True,
    print_every=20
)
solver.train()
```

```
(Iteration 1 / 980) loss: 2.304740
(Epoch 0 / 1) train acc: 0.103000; val_acc: 0.107000
(Iteration 21 / 980) loss: 2.098229
(Iteration 41 / 980) loss: 1.949788
(Iteration 61 / 980) loss: 1.888398
(Iteration 81 / 980) loss: 1.877093
(Iteration 101 / 980) loss: 1.851877
(Iteration 121 / 980) loss: 1.859353
(Iteration 141 / 980) loss: 1.800181
(Iteration 161 / 980) loss: 2.143292
(Iteration 181 / 980) loss: 1.830573
(Iteration 201 / 980) loss: 2.037280
(Iteration 221 / 980) loss: 2.020304
(Iteration 241 / 980) loss: 1.823728
(Iteration 261 / 980) loss: 1.692679
(Iteration 281 / 980) loss: 1.882594
(Iteration 301 / 980) loss: 1.798261
(Iteration 321 / 980) loss: 1.851960
(Iteration 341 / 980) loss: 1.716323
(Iteration 361 / 980) loss: 1.897655
(Iteration 381 / 980) loss: 1.319744
(Iteration 401 / 980) loss: 1.738790
(Iteration 421 / 980) loss: 1.488866
(Iteration 441 / 980) loss: 1.718409
(Iteration 461 / 980) loss: 1.744440
(Iteration 481 / 980) loss: 1.605460
(Iteration 501 / 980) loss: 1.494847
(Iteration 521 / 980) loss: 1.835179
(Iteration 541 / 980) loss: 1.483923
(Iteration 561 / 980) loss: 1.676871
(Iteration 581 / 980) loss: 1.438325
(Iteration 601 / 980) loss: 1.443469
(Iteration 621 / 980) loss: 1.529369
(Iteration 641 / 980) loss: 1.763475
(Iteration 661 / 980) loss: 1.790329
(Iteration 681 / 980) loss: 1.693343
(Iteration 701 / 980) loss: 1.637078
(Iteration 721 / 980) loss: 1.644564
(Iteration 741 / 980) loss: 1.708919
(Iteration 761 / 980) loss: 1.494252
(Iteration 781 / 980) loss: 1.901751
(Iteration 801 / 980) loss: 1.898991
(Iteration 821 / 980) loss: 1.489988
(Iteration 841 / 980) loss: 1.377615
(Iteration 861 / 980) loss: 1.763751
(Iteration 881 / 980) loss: 1.540284
(Iteration 901 / 980) loss: 1.525582
(Iteration 921 / 980) loss: 1.674166
```

```
(Iteration 941 / 980) loss: 1.714316
(Iteration 961 / 980) loss: 1.534668
(Epoch 1 / 1) train acc: 0.504000; val_acc: 0.499000
```

[22]:
```
# Print final training accuracy.
print(
    "Full data training accuracy:",
    solver.check_accuracy(data['X_train'], data['y_train'])
)
```

```
Full data training accuracy: 0.4761836734693878
```

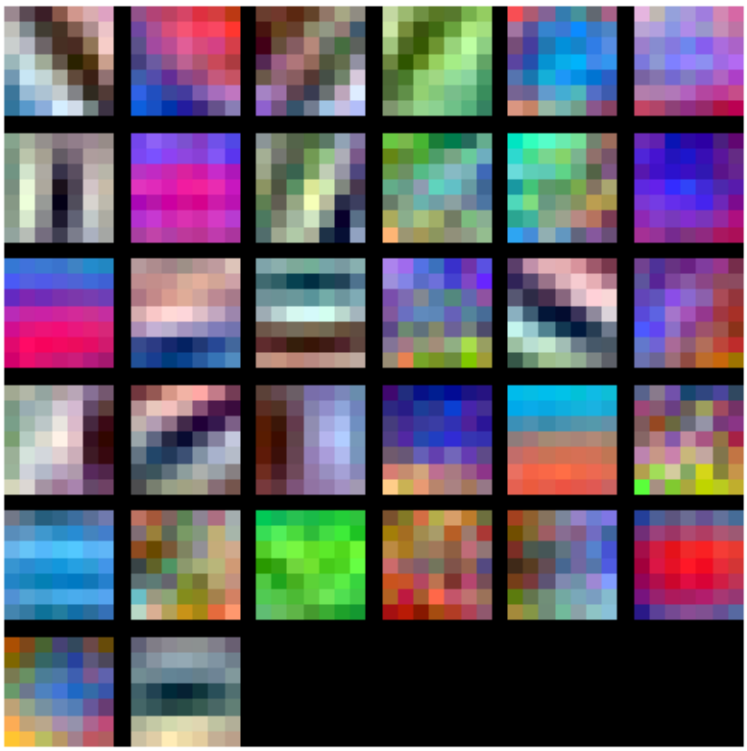
[23]:
```
# Print final validation accuracy.
print(
    "Full data validation accuracy:",
    solver.check_accuracy(data['X_val'], data['y_val'])
)
```

```
Full data validation accuracy: 0.499
```

8.5 Visualize Filters

You can visualize the first-layer convolutional filters from the trained network by running the following:

[24]:
```
from cs231n.vis_utils import visualize_grid

grid = visualize_grid(model.params['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
plt.axis('off')
plt.gcf().set_size_inches(5, 5)
plt.show()
```



9 Spatial Batch Normalization

We already saw that batch normalization is a very useful technique for training deep fully connected networks. As proposed in the original paper (link in `BatchNormalization.ipynb`), batch normalization can also be used for convolutional networks, but we need to tweak it a bit; the modification will be called "spatial batch normalization."

Normally, batch-normalization accepts inputs of shape `(N, D)` and produces outputs of shape `(N, D)`, where we normalize across the minibatch dimension `N`. For data coming from convolutional layers, batch normalization needs to accept inputs of shape `(N, C, H, W)` and produce outputs of shape `(N, C, H, W)` where the `N` dimension gives the minibatch size and the `(H, W)` dimensions give the spatial size of the feature map.

If the feature map was produced using convolutions, then we expect every feature channel's statistics e.g. mean, variance to be relatively consistent both between different images, and different locations within the same image – after all, every feature channel is produced by the same convolutional filter! Therefore, spatial batch normalization computes a mean and variance for each of the `C` feature channels by computing statistics over the minibatch dimension `N` as well the spatial dimensions `H` and `W`.

[1] Sergey Ioffe and Christian Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift", ICML 2015.

10 Spatial Batch Normalization: Forward Pass

In the file `cs231n/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

[25]:
```
np.random.seed(231)

# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization.
N, C, H, W = 2, 3, 4, 5
x = 4 * np.random.randn(N, C, H, W) + 10

print('Before spatial batch normalization:')
print('  shape: ', x.shape)
print('  means: ', x.mean(axis=(0, 2, 3)))
print('  stds: ', x.std(axis=(0, 2, 3)))

# Means should be close to zero and stds close to one
gamma, beta = np.ones(C), np.zeros(C)
bn_param = {'mode': 'train'}
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization:')
print('  shape: ', out.shape)
print('  means: ', out.mean(axis=(0, 2, 3)))
print('  stds: ', out.std(axis=(0, 2, 3)))

# Means should be close to beta and stds close to gamma
gamma, beta = np.asarray([3, 4, 5]), np.asarray([6, 7, 8])
out, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)
print('After spatial batch normalization (nontrivial gamma, beta):')
print('  shape: ', out.shape)
print('  means: ', out.mean(axis=(0, 2, 3)))
print('  stds: ', out.std(axis=(0, 2, 3)))
```

```
Before spatial batch normalization:
  shape:  (2, 3, 4, 5)
  means:  [9.33463814 8.90909116 9.11056338]
  stds:  [3.61447857 3.19347686 3.5168142 ]
After spatial batch normalization:
  shape:  (2, 3, 4, 5)
  means:  [ 6.18949336e-16  5.99520433e-16 -1.22124533e-16]
  stds:  [0.99999962 0.99999951 0.9999996 ]
After spatial batch normalization (nontrivial gamma, beta):
  shape:  (2, 3, 4, 5)
  means:  [6. 7. 8.]
  stds:  [2.99999885 3.99999804 4.99999798]
```

[26]:
```
np.random.seed(231)

# Check the test-time forward pass by running the training-time
# forward pass many times to warm up the running averages, and then
# checking the means and variances of activations after a test-time
# forward pass.
N, C, H, W = 10, 4, 11, 12

bn_param = {'mode': 'train'}
gamma = np.ones(C)
beta = np.zeros(C)
for t in range(50):
  x = 2.3 * np.random.randn(N, C, H, W) + 13
  spatial_batchnorm_forward(x, gamma, beta, bn_param)
bn_param['mode'] = 'test'
x = 2.3 * np.random.randn(N, C, H, W) + 13
a_norm, _ = spatial_batchnorm_forward(x, gamma, beta, bn_param)

# Means should be close to zero and stds close to one, but will be
# noisier than training-time forward passes.
print('After spatial batch normalization (test-time):')
print('  means: ', a_norm.mean(axis=(0, 2, 3)))
print('  stds: ', a_norm.std(axis=(0, 2, 3)))
```

```
After spatial batch normalization (test-time):
  means:  [-0.08034406  0.07562881  0.05716371  0.04378383]
  stds:  [0.96718744 1.0299714  1.02887624 1.00585577]
```

11 Spatial Batch Normalization: Backward Pass

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_batchnorm_backward`. Run the following to check your implementation using a numeric gradient check:

[27]:
```
np.random.seed(231)
N, C, H, W = 2, 3, 4, 5
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(C)
beta = np.random.randn(C)
dout = np.random.randn(N, C, H, W)

bn_param = {'mode': 'train'}
fx = lambda x: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fg = lambda a: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]
fb = lambda b: spatial_batchnorm_forward(x, gamma, beta, bn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
```

```
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

#You should expect errors of magnitudes between 1e-12~1e-06
_, cache = spatial_batchnorm_forward(x, gamma, beta, bn_param)
dx, dgamma, dbeta = spatial_batchnorm_backward(dout, cache)
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  2.786648193872555e-07
dgamma error:  7.0974817113608705e-12
dbeta error:  3.275608725278405e-12
```

12 Spatial Group Normalization

In the previous notebook, we mentioned that Layer Normalization is an alternative normalization technique that mitigates the batch size limitations of Batch Normalization. However, as the authors of [2] observed, Layer Normalization does not perform as well as Batch Normalization when used with Convolutional Layers:

> With fully connected layers, all the hidden units in a layer tend to make similar contributions to the final prediction, and re-centering and rescaling the summed inputs to a layer works well. However, the assumption of similar contributions is no longer true for convolutional neural networks. The large number of the hidden units whose receptive fields lie near the boundary of the image are rarely turned on and thus have very different statistics from the rest of the hidden units within the same layer.

The authors of [3] propose an intermediary technique. In contrast to Layer Normalization, where you normalize over the entire feature per-datapoint, they suggest a consistent splitting of each per-datapoint feature into G groups and a per-group per-datapoint normalization instead.

Visual comparison of the normalization techniques discussed so far (image edited from [3])

Even though an assumption of equal contribution is still being made within each group, the authors hypothesize that this is not as problematic, as innate grouping arises within features for visual recognition. One example they use to illustrate this is that many high-performance handcrafted features in traditional computer vision have terms that are explicitly grouped together. Take for example Histogram of Oriented Gradients [4] – after computing histograms per spatially local block, each per-block histogram is normalized before being concatenated together to form the final feature vector.

You will now implement Group Normalization.

[2] Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton. "Layer Normalization." stat 1050 (2016): 21.

[3] Wu, Yuxin, and Kaiming He. "Group Normalization." arXiv preprint arXiv:1803.08494 (2018).

[4] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In Computer Vision and Pattern Recognition (CVPR), 2005.

13 Spatial Group Normalization: Forward Pass

In the file `cs231n/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

[28]:
```
np.random.seed(231)

# Check the training-time forward pass by checking means and variances
# of features both before and after spatial batch normalization.
N, C, H, W = 2, 6, 4, 5
G = 2
x = 4 * np.random.randn(N, C, H, W) + 10
x_g = x.reshape((N*G,-1))
print('Before spatial group normalization:')
print('  shape: ', x.shape)
print('  means: ', x_g.mean(axis=1))
print('  stds: ', x_g.std(axis=1))

# Means should be close to zero and stds close to one
gamma, beta = np.ones((1,C,1,1)), np.zeros((1,C,1,1))
bn_param = {'mode': 'train'}

out, _ = spatial_groupnorm_forward(x, gamma, beta, G, bn_param)
out_g = out.reshape((N*G,-1))
print('After spatial group normalization:')
print('  shape: ', out.shape)
print('  means: ', out_g.mean(axis=1))
print('  stds: ', out_g.std(axis=1))
```

```
Before spatial group normalization:
  shape:  (2, 6, 4, 5)
  means:  [9.72505327 8.51114185 8.9147544  9.43448077]
  stds:  [3.67070958 3.09892597 4.27043622 3.97521327]
After spatial group normalization:
  shape:  (2, 6, 4, 5)
  means:  [-2.14643118e-16  5.25505565e-16  2.65528340e-16 -3.38618023e-16]
  stds:  [0.99999963 0.99999948 0.99999973 0.99999968]
```

14 Spatial Group Normalization: Backward Pass

In the file `cs231n/layers.py`, implement the backward pass for spatial batch normalization in the function `spatial_groupnorm_backward`. Run the following to check your implementation using a numeric gradient check:

[29]:
```
np.random.seed(231)

N, C, H, W = 2, 6, 4, 5
G = 2
```

```
x = 5 * np.random.randn(N, C, H, W) + 12
gamma = np.random.randn(1,C,1,1)
beta = np.random.randn(1,C,1,1)
dout = np.random.randn(N, C, H, W)

gn_param = {}
fx = lambda x: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fg = lambda a: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]
fb = lambda b: spatial_groupnorm_forward(x, gamma, beta, G, gn_param)[0]

dx_num = eval_numerical_gradient_array(fx, x, dout)
da_num = eval_numerical_gradient_array(fg, gamma, dout)
db_num = eval_numerical_gradient_array(fb, beta, dout)

_, cache = spatial_groupnorm_forward(x, gamma, beta, G, gn_param)
dx, dgamma, dbeta = spatial_groupnorm_backward(dout, cache)

# You should expect errors of magnitudes between 1e-12 and 1e-07.
print('dx error: ', rel_error(dx_num, dx))
print('dgamma error: ', rel_error(da_num, dgamma))
print('dbeta error: ', rel_error(db_num, dbeta))
```

```
dx error:  1.0
dgamma error:  9.468195772749234e-12
dbeta error:  3.354494437653335e-12
```

PyTorch

November 29, 2023

[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/assignment1/'
FOLDERNAME = 'Colab Notebooks/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/Colab Notebooks/assignment2/cs231n/datasets
/content/drive/My Drive/Colab Notebooks/assignment2
```

1 Introduction to PyTorch

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, PyTorch (or TensorFlow, if you choose to work with that notebook).

1.1 Why do we use deep learning frameworks?

- Our code will now run on GPUs! This will allow our models to train much faster. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- In this class, we want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- Finally, we want you to be exposed to the sort of deep learning code you might run into in academia or industry.

1.2 What is PyTorch?

PyTorch is a system for executing dynamic computational graphs over Tensor objects that behave similarly as numpy ndarray. It comes with a powerful automatic differentiation engine that removes the need for manual back-propagation.

1.3 How do I learn PyTorch?

One of our former instructors, Justin Johnson, made an excellent tutorial for PyTorch.

You can also find the detailed API doc here. If you have other questions that are not addressed by the API docs, the PyTorch forum is a much better place to ask than StackOverflow.

2 Table of Contents

This assignment has 5 parts. You will learn PyTorch on **three different levels of abstraction**, which will help you understand it better and prepare you for the final project.

1. Part I, Preparation: we will use CIFAR-10 dataset.
2. Part II, Barebones PyTorch: **Abstraction level 1**, we will work directly with the lowest-level PyTorch Tensors.
3. Part III, PyTorch Module API: **Abstraction level 2**, we will use `nn.Module` to define arbitrary neural network architecture.
4. Part IV, PyTorch Sequential API: **Abstraction level 3**, we will use `nn.Sequential` to define a linear feed-forward network very conveniently.
5. Part V, CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`nn.Module`	High	Medium

API	Flexibility	Convenience
`nn.Sequential`	Low	High

3 GPU

You can manually switch to a GPU device on Colab by clicking `Runtime -> Change runtime type` and selecting `GPU` under `Hardware Accelerator`. You should do this before running the following cells to import packages, since the kernel gets restarted upon switching runtimes.

[2]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler

import torchvision.datasets as dset
import torchvision.transforms as T

import numpy as np

USE_GPU = True
dtype = torch.float32 # We will be using float throughout this tutorial.

if USE_GPU and torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

# Constant to control how frequently we print train loss.
print_every = 100
print('using device:', device)
```

```
using device: cuda
```

4 Part I. Preparation

Now, let's load the CIFAR-10 dataset. This might take a couple minutes the first time you do it, but the files should stay cached after that.

In previous parts of the assignment we had to write our own code to download the CIFAR-10 dataset, preprocess it, and iterate through it in minibatches; PyTorch provides convenient tools to automate this process for us.

[3]:
```
NUM_TRAIN = 49000

# The torchvision.transforms package provides tools for preprocessing data
# and for performing data augmentation; here we set up a transform to
```

```
# preprocess the data by subtracting the mean RGB value and dividing by the
# standard deviation of each RGB value; we've hardcoded the mean and std.
transform = T.Compose([
                T.ToTensor(),
                T.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
            ])

# We set up a Dataset object for each split (train / val / test); Datasets load
# training examples one at a time, so we wrap each Dataset in a DataLoader which
# iterates through the Dataset and forms minibatches. We divide the CIFAR-10
# training set into train and val sets by passing a Sampler object to the
# DataLoader telling how it should sample from the underlying Dataset.
cifar10_train = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                             transform=transform)
loader_train = DataLoader(cifar10_train, batch_size=64,
                          sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN)))

cifar10_val = dset.CIFAR10('./cs231n/datasets', train=True, download=True,
                           transform=transform)
loader_val = DataLoader(cifar10_val, batch_size=64,
                        sampler=sampler.SubsetRandomSampler(range(NUM_TRAIN,␣
↪50000)))

cifar10_test = dset.CIFAR10('./cs231n/datasets', train=False, download=True,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)
```

```
Files already downloaded and verified
Files already downloaded and verified
Files already downloaded and verified
```

5 Part II. Barebones PyTorch

PyTorch ships with high-level APIs to help us define model architectures conveniently, which we will cover in Part II of this tutorial. In this section, we will start with the barebone PyTorch elements to understand the autograd engine better. After this exercise, you will come to appreciate the high-level model API more.

We will start with a simple fully-connected ReLU network with two hidden layers and no biases for CIFAR classification. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. It is important that you understand every line, because you will write a harder version after the example.

When we create a PyTorch Tensor with `requires_grad=True`, then operations involving that Tensor will not just compute values; they will also build up a computational graph in the background, allowing us to easily backpropagate through the graph to compute gradients of some Tensors with respect to a downstream loss. Concretely if x is a Tensor with `x.requires_grad == True` then after backpropagation `x.grad` will be another Tensor holding the gradient of x with respect to the

scalar loss at the end.

5.0.1 PyTorch Tensors: Flatten Function

A PyTorch Tensor is conceptionally similar to a numpy array: it is an n-dimensional grid of numbers, and like numpy PyTorch provides many functions to efficiently operate on Tensors. As a simple example, we provide a `flatten` function below which reshapes image data for use in a fully-connected neural network.

Recall that image data is typically stored in a Tensor of shape N x C x H x W, where:

- N is the number of datapoints
- C is the number of channels
- H is the height of the intermediate feature map in pixels
- W is the height of the intermediate feature map in pixels

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector – it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `C x H x W` values per representation into a single long vector. The flatten function below first reads in the N, C, H, and W values from a given batch of data, and then returns a "view" of that data. "View" is analogous to numpy's "reshape" method: it reshapes x's dimensions to be N x ??, where ?? is allowed to be anything (in this case, it will be C x H x W, but we don't need to specify that explicitly).

[4]:
```
def flatten(x):
    N = x.shape[0] # read in N, C, H, W
    return x.view(N, -1)  # "flatten" the C * H * W values into a single vector␣
 ↪per image

def test_flatten():
    x = torch.arange(12).view(2, 1, 3, 2)
    print('Before flattening: ', x)
    print('After flattening: ', flatten(x))

test_flatten()
```

```
Before flattening:  tensor([[[[ 0,  1],
          [ 2,  3],
          [ 4,  5]]],


        [[[ 6,  7],
          [ 8,  9],
          [10, 11]]]])
After flattening:  tensor([[ 0,  1,  2,  3,  4,  5],
        [ 6,  7,  8,  9, 10, 11]])
```

5.0.2 Barebones PyTorch: Two-Layer Network

Here we define a function `two_layer_fc` which performs the forward pass of a two-layer fully-connected ReLU network on a batch of image data. After defining the forward pass we check that it doesn't crash and that it produces outputs of the right shape by running zeros through the network.

You don't have to write any code here, but it's important that you read and understand the implementation.

[5]:
```
import torch.nn.functional as F  # useful stateless functions

def two_layer_fc(x, params):
    """
    A fully-connected neural networks; the architecture is:
    NN is fully connected -> ReLU -> fully connected layer.
    Note that this function only defines the forward pass;
    PyTorch will take care of the backward pass for us.

    The input to the network will be a minibatch of data, of shape
    (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have H␣
 ↪units,
    and the output layer will produce scores for C classes.

    Inputs:
    - x: A PyTorch Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
    - params: A list [w1, w2] of PyTorch Tensors giving weights for the network;
      w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A PyTorch Tensor of shape (N, C) giving classification scores for
      the input data x.
    """
    # first we flatten the image
    x = flatten(x)  # shape: [batch_size, C x H x W]

    w1, w2 = params

    # Forward pass: compute predicted y using operations on Tensors. Since w1␣
 ↪and
    # w2 have requires_grad=True, operations involving these Tensors will cause
    # PyTorch to build a computational graph, allowing automatic computation of
    # gradients. Since we are no longer implementing the backward pass by hand␣
 ↪we
    # don't need to keep references to intermediate values.
    # you can also use `.clamp(min=0)`, equivalent to F.relu()
    x = F.relu(x.mm(w1))
```

```
    x = x.mm(w2)
    return x


def two_layer_fc_test():
    hidden_layer_size = 42
    x = torch.zeros((64, 50), dtype=dtype)  # minibatch size 64, feature␣
 ↪dimension 50
    w1 = torch.zeros((50, hidden_layer_size), dtype=dtype)
    w2 = torch.zeros((hidden_layer_size, 10), dtype=dtype)
    scores = two_layer_fc(x, [w1, w2])
    print(scores.size())  # you should see [64, 10]

two_layer_fc_test()
```

```
torch.Size([64, 10])
```

5.0.3 Barebones PyTorch: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet`, which will perform the forward pass of a three-layer convolutional network. Like above, we can immediately test our implementation by passing zeros through the network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for C classes.

Note that we have **no softmax activation** here after our fully-connected layer: this is because PyTorch's cross entropy loss performs a softmax activation for you, and by bundling that step in makes computation more efficient.

HINT: For convolutions: http://pytorch.org/docs/stable/nn.html#torch.nn.functional.conv2d; pay attention to the shapes of convolutional filters!

[6]:
```
def three_layer_convnet(x, params):
    """
    Performs the forward pass of a three-layer convolutional network with the
    architecture defined above.

    Inputs:
    - x: A PyTorch Tensor of shape (N, 3, H, W) giving a minibatch of images
    - params: A list of PyTorch Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: PyTorch Tensor of shape (channel_1, 3, KH1, KW1) giving weights
```

```
      for the first convolutional layer
    - conv_b1: PyTorch Tensor of shape (channel_1,) giving biases for the␣
↪first
      convolutional layer
    - conv_w2: PyTorch Tensor of shape (channel_2, channel_1, KH2, KW2) giving
      weights for the second convolutional layer
    - conv_b2: PyTorch Tensor of shape (channel_2,) giving biases for the␣
↪second
      convolutional layer
    - fc_w: PyTorch Tensor giving weights for the fully-connected layer. Can␣
↪you
      figure out what the shape should be?
    - fc_b: PyTorch Tensor giving biases for the fully-connected layer. Can␣
↪you
      figure out what the shape should be?

  Returns:
  - scores: PyTorch Tensor of shape (N, C) giving classification scores for x
  """
  conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
  scores = None
  ␣
↪##############################################################################
  # TODO: Implement the forward pass for the three-layer ConvNet.             ␣
↪  #
  ␣
↪##############################################################################
  # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

  # İlk evrişimli katman + ReLU aktivasyonu
  x = F.relu(F.conv2d(x, conv_w1, conv_b1, padding=2))

  # İkinci evrişimli katman + ReLU aktivasyonu
  x = F.relu(F.conv2d(x, conv_w2, conv_b2, padding=1))

  # Düzleştirme işlemi ve tam bağlantılı katman
  x = x.view(x.size(0), -1)  # Flatten
  scores = x.mm(fc_w) + fc_b

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

```
    return scores
```

After defining the forward pass of the ConvNet above, run the following cell to test your implementation.

When you run this function, scores should have shape (64, 10).

[7]:
```
def three_layer_convnet_test():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image␣
 ↪size [3, 32, 32]

    conv_w1 = torch.zeros((6, 3, 5, 5), dtype=dtype)  # [out_channel,␣
 ↪in_channel, kernel_H, kernel_W]
    conv_b1 = torch.zeros((6,))  # out_channel
    conv_w2 = torch.zeros((9, 6, 3, 3), dtype=dtype)  # [out_channel,␣
 ↪in_channel, kernel_H, kernel_W]
    conv_b2 = torch.zeros((9,))  # out_channel

    # you must calculate the shape of the tensor after two conv layers, before␣
 ↪the fully-connected layer
    fc_w = torch.zeros((9 * 32 * 32, 10))
    fc_b = torch.zeros(10)

    scores = three_layer_convnet(x, [conv_w1, conv_b1, conv_w2, conv_b2, fc_w,␣
 ↪fc_b])
    print(scores.size())  # you should see [64, 10]
three_layer_convnet_test()
```

```
torch.Size([64, 10])
```

5.0.4 Barebones PyTorch: Initialization

Let's write a couple utility methods to initialize the weight matrices for our models.

- `random_weight(shape)` initializes a weight tensor with the Kaiming normalization method.
- `zero_weight(shape)` initializes a weight tensor with all zeros. Useful for instantiating bias parameters.

The `random_weight` function uses the Kaiming normal initialization method, described in:

He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, ICCV 2015, https://arxiv.org/abs/1502.01852

[8]:
```
def random_weight(shape):
    """
    Create random Tensors for weights; setting requires_grad=True means that we
    want to compute gradients for these Tensors during the backward pass.
    We use Kaiming normalization: sqrt(2 / fan_in)
    """
    if len(shape) == 2:  # FC weight
```

```
        fan_in = shape[0]
    else:
        fan_in = np.prod(shape[1:]) # conv weight [out_channel, in_channel, kH,␣
 ↪kW]
    # randn is standard normal distribution generator.
    w = torch.randn(shape, device=device, dtype=dtype) * np.sqrt(2. / fan_in)
    w.requires_grad = True
    return w

def zero_weight(shape):
    return torch.zeros(shape, device=device, dtype=dtype, requires_grad=True)

# create a weight of shape [3 x 5]
# you should see the type `torch.cuda.FloatTensor` if you use GPU.
# Otherwise it should be `torch.FloatTensor`
random_weight((3, 5))
```

[8]:
```
tensor([[-1.0707,  0.0917, -0.1488, -1.3135,  0.7490],
        [-0.3843, -0.5356, -0.0632, -0.5950, -0.5485],
        [ 0.4475, -0.6179,  1.0626, -0.1411, -0.8810]], device='cuda:0',
       requires_grad=True)
```

5.0.5 Barebones PyTorch: Check Accuracy

When training the model we will use the following function to check the accuracy of our model on the training or validation sets.

When checking accuracy we don't need to compute any gradients; as a result we don't need PyTorch to build a computational graph for us when we compute scores. To prevent a graph from being built we scope our computation under a `torch.no_grad()` context manager.

[9]:
```
def check_accuracy_part2(loader, model_fn, params):
    """
    Check the accuracy of a classification model.

    Inputs:
    - loader: A DataLoader for the data split we want to check
    - model_fn: A function that performs the forward pass of the model,
      with the signature scores = model_fn(x, params)
    - params: List of PyTorch Tensors giving parameters of the model

    Returns: Nothing, but prints the accuracy of the model
    """
    split = 'val' if loader.dataset.train else 'test'
    print('Checking accuracy on the %s set' % split)
    num_correct, num_samples = 0, 0
    with torch.no_grad():
        for x, y in loader:
```

```
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.int64)
            scores = model_fn(x, params)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 *␣
↪acc))
```

5.0.6 BareBones PyTorch: Training Loop

We can now set up a basic training loop to train our network. We will train the model using stochastic gradient descent without momentum. We will use `torch.functional.cross_entropy` to compute the loss; you can read about it here.

The training loop takes as input the neural network function, a list of initialized parameters (`[w1, w2]` in our example), and learning rate.

[10]:
```
def train_part2(model_fn, params, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model.
      It should have the signature scores = model_fn(x, params) where x is a
      PyTorch Tensor of image data, params is a list of PyTorch Tensors giving
      model weights, and scores is a PyTorch Tensor of shape (N, C) giving
      scores for the elements in x.
    - params: List of PyTorch Tensors giving weights for the model
    - learning_rate: Python scalar giving the learning rate to use for SGD

    Returns: Nothing
    """
    for t, (x, y) in enumerate(loader_train):
        # Move the data to the proper device (GPU or CPU)
        x = x.to(device=device, dtype=dtype)
        y = y.to(device=device, dtype=torch.long)

        # Forward pass: compute scores and loss
        scores = model_fn(x, params)
        loss = F.cross_entropy(scores, y)

        # Backward pass: PyTorch figures out which Tensors in the computational
        # graph has requires_grad=True and uses backpropagation to compute the
        # gradient of the loss with respect to these Tensors, and stores the
        # gradients in the .grad attribute of each Tensor.
        loss.backward()
```

```
        # Update parameters. We don't want to backpropagate through the
        # parameter updates, so we scope the updates under a torch.no_grad()
        # context manager to prevent a computational graph from being built.
        with torch.no_grad():
            for w in params:
                w -= learning_rate * w.grad

                # Manually zero the gradients after running the backward pass
                w.grad.zero_()

        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss.item()))
            check_accuracy_part2(loader_val, model_fn, params)
            print()
```

5.0.7 BareBones PyTorch: Train a Two-Layer Network

Now we are ready to run the training loop. We need to explicitly allocate tensors for the fully connected weights, `w1` and `w2`.

Each minibatch of CIFAR has 64 examples, so the tensor shape is `[64, 3, 32, 32]`.

After flattening, `x` shape should be `[64, 3 * 32 * 32]`. This will be the size of the first dimension of `w1`. The second dimension of `w1` is the hidden layer size, which will also be the first dimension of `w2`.

Finally, the output of the network is a 10-dimensional vector that represents the probability distribution over 10 classes.

You don't need to tune any hyperparameters but you should see accuracies above 40% after training for one epoch.

[11]:
```
hidden_layer_size = 4000
learning_rate = 1e-2

w1 = random_weight((3 * 32 * 32, hidden_layer_size))
w2 = random_weight((hidden_layer_size, 10))

train_part2(two_layer_fc, [w1, w2], learning_rate)
```

```
Iteration 0, loss = 2.8408
Checking accuracy on the val set
Got 134 / 1000 correct (13.40%)

Iteration 100, loss = 1.7337
Checking accuracy on the val set
Got 311 / 1000 correct (31.10%)

Iteration 200, loss = 1.9684
```

```
Checking accuracy on the val set
Got 344 / 1000 correct (34.40%)

Iteration 300, loss = 1.9384
Checking accuracy on the val set
Got 405 / 1000 correct (40.50%)

Iteration 400, loss = 1.9320
Checking accuracy on the val set
Got 428 / 1000 correct (42.80%)

Iteration 500, loss = 1.6331
Checking accuracy on the val set
Got 415 / 1000 correct (41.50%)

Iteration 600, loss = 1.6422
Checking accuracy on the val set
Got 409 / 1000 correct (40.90%)

Iteration 700, loss = 1.3888
Checking accuracy on the val set
Got 455 / 1000 correct (45.50%)
```

5.0.8 BareBones PyTorch: Training a ConvNet

In the below you should use the functions defined above to train a three-layer convolutional network on CIFAR. The network should have the following architecture:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You should initialize your weight matrices using the `random_weight` function defined above, and you should initialize your bias vectors using the `zero_weight` function above.

You don't need to tune any hyperparameters, but if everything works correctly you should achieve an accuracy above 42% after one epoch.

[12]:
```
learning_rate = 3e-3

channel_1 = 32
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
```

```
conv_b2 = None
fc_w = None
fc_b = None

################################################################################
# TODO: Initialize the parameters of a three-layer ConvNet.                    #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


# Parametreleri başlat
conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight(channel_1)
conv_w2 = random_weight((channel_2, channel_1, 3, 3))
conv_b2 = zero_weight(channel_2)
# Tam bağlantılı katman için boyutlar: (channel_2 * feature_map_height *␣
 ↪feature_map_width, class_count)
fc_w = random_weight((channel_2 * 32 * 32, 10))
fc_b = zero_weight(10)

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)


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

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)
```

```
Iteration 0, loss = 3.1659
Checking accuracy on the val set
Got 117 / 1000 correct (11.70%)

Iteration 100, loss = 1.9195
Checking accuracy on the val set
Got 363 / 1000 correct (36.30%)

Iteration 200, loss = 1.7475
Checking accuracy on the val set
Got 390 / 1000 correct (39.00%)

Iteration 300, loss = 1.6560
Checking accuracy on the val set
Got 429 / 1000 correct (42.90%)
```

```
Iteration 400, loss = 1.6852
Checking accuracy on the val set
Got 439 / 1000 correct (43.90%)

Iteration 500, loss = 1.3455
Checking accuracy on the val set
Got 450 / 1000 correct (45.00%)

Iteration 600, loss = 1.5280
Checking accuracy on the val set
Got 474 / 1000 correct (47.40%)

Iteration 700, loss = 1.6983
Checking accuracy on the val set
Got 475 / 1000 correct (47.50%)

Iteration 0, loss = 1.4383
Checking accuracy on the val set
Got 468 / 1000 correct (46.80%)

Iteration 100, loss = 1.1845
Checking accuracy on the val set
Got 497 / 1000 correct (49.70%)

Iteration 200, loss = 1.3240
Checking accuracy on the val set
Got 492 / 1000 correct (49.20%)

Iteration 300, loss = 1.4815
Checking accuracy on the val set
Got 505 / 1000 correct (50.50%)

Iteration 400, loss = 1.3945
Checking accuracy on the val set
Got 492 / 1000 correct (49.20%)

Iteration 500, loss = 1.2743
Checking accuracy on the val set
Got 514 / 1000 correct (51.40%)

Iteration 600, loss = 1.1216
Checking accuracy on the val set
Got 511 / 1000 correct (51.10%)

Iteration 700, loss = 1.2109
Checking accuracy on the val set
Got 514 / 1000 correct (51.40%)
```

6 Part III. PyTorch Module API

Barebone PyTorch requires that we track all the parameter tensors by hand. This is fine for small networks with a few tensors, but it would be extremely inconvenient and error-prone to track tens or hundreds of tensors in larger networks.

PyTorch provides the `nn.Module` API for you to define arbitrary network architectures, while tracking every learnable parameters for you. In Part II, we implemented SGD ourselves. PyTorch also provides the `torch.optim` package that implements all the common optimizers, such as RMSProp, Adagrad, and Adam. It even supports approximate second-order methods like L-BFGS! You can refer to the doc for the exact specifications of each optimizer.

To use the Module API, follow the steps below:

1. Subclass `nn.Module`. Give your network class an intuitive name like `TwoLayerFC`.
2. In the constructor `__init__()`, define all the layers you need as class attributes. Layer objects like `nn.Linear` and `nn.Conv2d` are themselves `nn.Module` subclasses and contain learnable parameters, so that you don't have to instantiate the raw tensors yourself. `nn.Module` will track these internal parameters for you. Refer to the doc to learn more about the dozens of builtin layers. **Warning**: don't forget to call the `super().__init__()` first!
3. In the `forward()` method, define the *connectivity* of your network. You should use the attributes defined in `__init__` as function calls that take tensor as input and output the "transformed" tensor. Do *not* create any new layers with learnable parameters in `forward()`! All of them must be declared upfront in `__init__`.

After you define your Module subclass, you can instantiate it as an object and call it just like the NN forward function in part II.

6.0.1 Module API: Two-Layer Network

Here is a concrete example of a 2-layer fully connected network:

[13]:
```
class TwoLayerFC(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        # assign layer objects to class attributes
        self.fc1 = nn.Linear(input_size, hidden_size)
        # nn.init package contains convenient initialization methods
        # http://pytorch.org/docs/master/nn.html#torch-nn-init
        nn.init.kaiming_normal_(self.fc1.weight)
        self.fc2 = nn.Linear(hidden_size, num_classes)
        nn.init.kaiming_normal_(self.fc2.weight)

    def forward(self, x):
        # forward always defines connectivity
        x = flatten(x)
        scores = self.fc2(F.relu(self.fc1(x)))
```

```
        return scores

def test_TwoLayerFC():
    input_size = 50
    x = torch.zeros((64, input_size), dtype=dtype)  # minibatch size 64,␣
 ↪feature dimension 50
    model = TwoLayerFC(input_size, 42, 10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_TwoLayerFC()
```

```
torch.Size([64, 10])
```

6.0.2 Module API: Three-Layer ConvNet

It's your turn to implement a 3-layer ConvNet followed by a fully connected layer. The network architecture should be the same as in Part II:

1. Convolutional layer with `channel_1` 5x5 filters with zero-padding of 2
2. ReLU
3. Convolutional layer with `channel_2` 3x3 filters with zero-padding of 1
4. ReLU
5. Fully-connected layer to `num_classes` classes

You should initialize the weight matrices of the model using the Kaiming normal initialization method.

HINT: http://pytorch.org/docs/stable/nn.html#conv2d

After you implement the three-layer ConvNet, the `test_ThreeLayerConvNet` function will run your implementation; it should print `(64, 10)` for the shape of the output scores.

[14]:
```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ########################################################################
        # TODO: Set up the layers you need for a three-layer ConvNet with the  #
        # architecture defined above.                                          #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # İki evrişimli katman ve bir tam bağlantılı katman tanımla
        self.conv1 = nn.Conv2d(in_channel, channel_1, 5, padding=2)
        nn.init.kaiming_normal_(self.conv1.weight)
        self.conv2 = nn.Conv2d(channel_1, channel_2, 3, padding=1)
        nn.init.kaiming_normal_(self.conv2.weight)
        self.fc = nn.Linear(channel_2 * 32 * 32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight)
        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        ########################################################################
```

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

    def forward(self, x):
        scores = None
        ########################################################################
        # TODO: Implement the forward function for a 3-layer ConvNet. you      #
        # should use the layers you defined in __init__ and specify the        #
        # connectivity of those layers in forward()                            #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # İleri geçiş fonksiyonunu tanımla
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        N = x.shape[0]  # x'in batch boyutu
        x = x.view(N, -1)
        scores = self.fc(x)

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


def test_ThreeLayerConvNet():
    x = torch.zeros((64, 3, 32, 32), dtype=dtype)  # minibatch size 64, image␣
 ↪size [3, 32, 32]
    model = ThreeLayerConvNet(in_channel=3, channel_1=12, channel_2=8,␣
 ↪num_classes=10)
    scores = model(x)
    print(scores.size())  # you should see [64, 10]
test_ThreeLayerConvNet()
```

```
torch.Size([64, 10])
```

6.0.3 Module API: Check Accuracy

Given the validation or test set, we can check the classification accuracy of a neural network.

This version is slightly different from the one in part II. You don't manually pass in the parameters anymore.

[15]:
```
def check_accuracy_part34(loader, model):
    if loader.dataset.train:
        print('Checking accuracy on validation set')
    else:
```

```
        print('Checking accuracy on test set')
    num_correct = 0
    num_samples = 0
    model.eval()  # set model to evaluation mode
    with torch.no_grad():
        for x, y in loader:
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)
            scores = model(x)
            _, preds = scores.max(1)
            num_correct += (preds == y).sum()
            num_samples += preds.size(0)
        acc = float(num_correct) / num_samples
        print('Got %d / %d correct (%.2f)' % (num_correct, num_samples, 100 *␣
↪acc))
```

6.0.4 Module API: Training Loop

We also use a slightly different training loop. Rather than updating the values of the weights ourselves, we use an Optimizer object from the `torch.optim` package, which abstract the notion of an optimization algorithm and provides implementations of most of the algorithms commonly used to optimize neural networks.

[16]:
```
def train_part34(model, optimizer, epochs=1):
    """
    Train a model on CIFAR-10 using the PyTorch Module API.

    Inputs:
    - model: A PyTorch Module giving the model to train.
    - optimizer: An Optimizer object we will use to train the model
    - epochs: (Optional) A Python integer giving the number of epochs to train␣
↪for

    Returns: Nothing, but prints model accuracies during training.
    """
    model = model.to(device=device)  # move the model parameters to CPU/GPU
    for e in range(epochs):
        for t, (x, y) in enumerate(loader_train):
            model.train()  # put model to training mode
            x = x.to(device=device, dtype=dtype)  # move to device, e.g. GPU
            y = y.to(device=device, dtype=torch.long)

            scores = model(x)
            loss = F.cross_entropy(scores, y)

            # Zero out all of the gradients for the variables which the␣
↪optimizer
```

```
            # will update.
            optimizer.zero_grad()

            # This is the backwards pass: compute the gradient of the loss with
            # respect to each  parameter of the model.
            loss.backward()

            # Actually update the parameters of the model using the gradients
            # computed by the backwards pass.
            optimizer.step()

            if t % print_every == 0:
                print('Iteration %d, loss = %.4f' % (t, loss.item()))
                check_accuracy_part34(loader_val, model)
                print()
```

6.0.5 Module API: Train a Two-Layer Network

Now we are ready to run the training loop. In contrast to part II, we don't explicitly allocate parameter tensors anymore.

Simply pass the input size, hidden layer size, and number of classes (i.e. output size) to the constructor of `TwoLayerFC`.

You also need to define an optimizer that tracks all the learnable parameters inside `TwoLayerFC`.

You don't need to tune any hyperparameters, but you should see model accuracies above 40% after training for one epoch.

[17]:
```
hidden_layer_size = 4000
learning_rate = 1e-2
model = TwoLayerFC(3 * 32 * 32, hidden_layer_size, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.1275
Checking accuracy on validation set
Got 118 / 1000 correct (11.80)

Iteration 100, loss = 2.3795
Checking accuracy on validation set
Got 352 / 1000 correct (35.20)

Iteration 200, loss = 2.3971
Checking accuracy on validation set
Got 364 / 1000 correct (36.40)

Iteration 300, loss = 1.5516
```

```
Checking accuracy on validation set
Got 402 / 1000 correct (40.20)

Iteration 400, loss = 1.5722
Checking accuracy on validation set
Got 407 / 1000 correct (40.70)

Iteration 500, loss = 1.8731
Checking accuracy on validation set
Got 459 / 1000 correct (45.90)

Iteration 600, loss = 1.7641
Checking accuracy on validation set
Got 391 / 1000 correct (39.10)

Iteration 700, loss = 1.8771
Checking accuracy on validation set
Got 431 / 1000 correct (43.10)
```

6.0.6 Module API: Train a Three-Layer ConvNet

You should now use the Module API to train a three-layer ConvNet on CIFAR. This should look very similar to training the two-layer network! You don't need to tune any hyperparameters, but you should achieve above above 45% after training for one epoch.

You should train the model using stochastic gradient descent without momentum.

[18]:
```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
################################################################################
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****



# ThreeLayerConvNet modelini ve SGD optimizatörünü başlat
model = ThreeLayerConvNet(3, channel_1, channel_2, 10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)

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

```
################################################################################

train_part34(model, optimizer)
```

```
Iteration 0, loss = 3.6065
Checking accuracy on validation set
Got 106 / 1000 correct (10.60)

Iteration 100, loss = 1.6847
Checking accuracy on validation set
Got 355 / 1000 correct (35.50)

Iteration 200, loss = 1.8266
Checking accuracy on validation set
Got 404 / 1000 correct (40.40)

Iteration 300, loss = 1.6893
Checking accuracy on validation set
Got 398 / 1000 correct (39.80)

Iteration 400, loss = 1.5038
Checking accuracy on validation set
Got 424 / 1000 correct (42.40)

Iteration 500, loss = 1.9133
Checking accuracy on validation set
Got 451 / 1000 correct (45.10)

Iteration 600, loss = 1.4977
Checking accuracy on validation set
Got 451 / 1000 correct (45.10)

Iteration 700, loss = 1.5469
Checking accuracy on validation set
Got 464 / 1000 correct (46.40)
```

7 Part IV. PyTorch Sequential API

Part III introduced the PyTorch Module API, which allows you to define arbitrary learnable layers and their connectivity.

For simple models like a stack of feed forward layers, you still need to go through 3 steps: subclass `nn.Module`, assign layers to class attributes in `__init__`, and call each layer one by one in `forward()`. Is there a more convenient way?

Fortunately, PyTorch provides a container Module called `nn.Sequential`, which merges the above steps into one. It is not as flexible as `nn.Module`, because you cannot specify more complex topology

than a feed-forward stack, but it's good enough for many use cases.

7.0.1 Sequential API: Two-Layer Network

Let's see how to rewrite our two-layer fully connected network example with `nn.Sequential`, and train it using the training loop defined above.

Again, you don't need to tune any hyperparameters here, but you shoud achieve above 40% accuracy after one epoch of training.

[19]:
```
# We need to wrap `flatten` function in a module in order to stack it
# in nn.Sequential
class Flatten(nn.Module):
    def forward(self, x):
        return flatten(x)

hidden_layer_size = 4000
learning_rate = 1e-2

model = nn.Sequential(
    Flatten(),
    nn.Linear(3 * 32 * 32, hidden_layer_size),
    nn.ReLU(),
    nn.Linear(hidden_layer_size, 10),
)

# you can use Nesterov momentum in optim.SGD
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                      momentum=0.9, nesterov=True)

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3200
Checking accuracy on validation set
Got 183 / 1000 correct (18.30)

Iteration 100, loss = 1.6248
Checking accuracy on validation set
Got 383 / 1000 correct (38.30)

Iteration 200, loss = 1.9203
Checking accuracy on validation set
Got 371 / 1000 correct (37.10)

Iteration 300, loss = 1.5882
Checking accuracy on validation set
Got 433 / 1000 correct (43.30)

Iteration 400, loss = 1.6217
```

```
Checking accuracy on validation set
Got 414 / 1000 correct (41.40)

Iteration 500, loss = 1.7291
Checking accuracy on validation set
Got 422 / 1000 correct (42.20)

Iteration 600, loss = 1.6650
Checking accuracy on validation set
Got 449 / 1000 correct (44.90)

Iteration 700, loss = 1.4557
Checking accuracy on validation set
Got 435 / 1000 correct (43.50)
```

7.0.2 Sequential API: Three-Layer ConvNet

Here you should use `nn.Sequential` to define and train a three-layer ConvNet with the same architecture we used in Part III:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You can use the default PyTorch weight initialization.

You should optimize your model using stochastic gradient descent with Nesterov momentum 0.9.

Again, you don't need to tune any hyperparameters but you should see accuracy above 55% after one epoch of training.

[20]:
```
channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None

################################################################################
# TODO: Rewrite the 2-layer ConvNet with bias from Part III with the           #
# Sequential API.                                                              #
################################################################################
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


model = nn.Sequential(
```

```
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2),  # Kernel boyutunu␣
 ↪ayarlayabilirsiniz
    nn.BatchNorm2d(channel_1),  # Batch Normalization ekleyin
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),  # Kernel␣
 ↪boyutunu ayarlayabilirsiniz
    nn.BatchNorm2d(channel_2),  # Batch Normalization ekleyin
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2 * 32 * 32, 10)  # Burada tam bağlantılı katmanın boyutu␣
 ↪değiştirilebilir
)

optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9,␣
 ↪nesterov=True)

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.3746
Checking accuracy on validation set
Got 132 / 1000 correct (13.20)

Iteration 100, loss = 2.3036
Checking accuracy on validation set
Got 105 / 1000 correct (10.50)

Iteration 200, loss = 2.2984
Checking accuracy on validation set
Got 105 / 1000 correct (10.50)

Iteration 300, loss = 2.3021
Checking accuracy on validation set
Got 79 / 1000 correct (7.90)

Iteration 400, loss = 2.3047
Checking accuracy on validation set
Got 98 / 1000 correct (9.80)

Iteration 500, loss = 2.3013
Checking accuracy on validation set
Got 98 / 1000 correct (9.80)
```

```
Iteration 600, loss = 2.3073
Checking accuracy on validation set
Got 87 / 1000 correct (8.70)

Iteration 700, loss = 2.3047
Checking accuracy on validation set
Got 107 / 1000 correct (10.70)
```

8 Part V. CIFAR-10 open-ended challenge

In this section, you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

Now it's your job to experiment with architectures, hyperparameters, loss functions, and optimizers to train a model that achieves **at least 70%** accuracy on the CIFAR-10 **validation** set within 10 epochs. You can use the check_accuracy and train functions from above. You can use either `nn.Module` or `nn.Sequential` API.

Describe what you did at the end of this notebook.

Here are the official API documentation for each component. One note: what we call in the class "spatial batch norm" is called "BatchNorm2D" in PyTorch.

- Layers in torch.nn package: http://pytorch.org/docs/stable/nn.html
- Activations: http://pytorch.org/docs/stable/nn.html#non-linear-activations
- Loss functions: http://pytorch.org/docs/stable/nn.html#loss-functions
- Optimizers: http://pytorch.org/docs/stable/optim.html

8.0.1 Things you might try:

- **Filter size**: Above we used 5x5; would smaller filters be more efficient?
- **Number of filters**: Above we used 32 filters. Do more or fewer do better?
- **Pooling vs Strided Convolution**: Do you use max pooling or just stride convolutions?
- **Batch normalization**: Try adding spatial batch normalization after convolution layers and vanilla batch normalization after affine layers. Do your networks train faster?
- **Network architecture**: The network above has two layers of trainable parameters. Can you do better with a deep network? Good architectures to try include:
  - [conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [conv-relu-conv-relu-pool]xN -> [affine]xM -> [softmax or SVM]
  - [batchnorm-relu-conv]xN -> [affine]xM -> [softmax or SVM]
- **Global Average Pooling**: Instead of flattening and then having multiple affine layers, perform convolutions until your image gets small (7x7 or so) and then perform an average pooling operation to get to a 1x1 image picture (1, 1 , Filter#), which is then reshaped into a (Filter#) vector. This is used in Google's Inception Network (See Table 1 for their architecture).
- **Regularization**: Add l2 weight regularization, or perhaps use Dropout.

8.0.2 Tips for training

For each network architecture that you try, you should tune the learning rate and other hyperparameters. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.
- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

8.0.3 Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time!

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
  - ResNets where the input from the previous layer is added to the output.
  - DenseNets where inputs into previous layers are concatenated together.
  - This blog has an in-depth overview

8.0.4 Have fun and happy training!

[21]:
```
# DenseNet approach based on paper https://arxiv.org/abs/1608.06993
# Implementation details: https://amaarora.github.io/2020/08/02/densenets.html

from collections import OrderedDict

class _Transition(nn.Sequential):
    """A transition layer used between each dense block."""

    def __init__(self, in_channels, out_channels):
        """Initializes the transition layer.

        Only `1` convolutional layer is used with kernel size of `1` to reduce␣
 ↪the
        _depth_ of the _activation maps_ from `in_channels` to `out_channels`.

        Args:
            in_channels (int): The number of input channels
```

```
            out_channels (int): The number of output channels
        """
        super().__init__()

        # Pass raw inputs through batchnorm and relu to get activations
        self.add_module('norm', nn.BatchNorm2d(in_channels))
        self.add_module('relu', nn.ReLU(inplace=True))

        # Perform channel and spacial area downsampling
        self.add_module('conv', nn.Conv2d(in_channels, out_channels, 1,␣
↪bias=False))
        self.add_module('pool', nn.AvgPool2d(2, 2))


class _DenseLayer(nn.Module):
    """A bulding layer used in each dense block."""

    def __init__(self, in_channels, growth_rate, bottleneck_size, drop_rate):
        """Initializes the dense layer.

        The layer takes in a batch of inputs of depth `in_channels`, and,
        instead of producing `growth_rate` feature maps, it firstly reduces
        the input depth to `bottleneck_size * growth_rate` (by performing a
        convolution with filter size `(1, 1)`) and only then performs the main
        convolution with filter size `(3, 3)`.

        Args:
            in_channels (int):     The number of input channels
            growth_rate (int):     The number of output channels
            bottleneck_size (int): The bottleneck before main convolution
            drop_rate (int):       The dropout hyperparameter
        """
        super().__init__()

        # Bottleneck layer to do initial depth downsampling
        self.norm1 = nn.BatchNorm2d(in_channels)
        self.relu1 = nn.ReLU(inplace=True)
        self.conv1 = nn.Conv2d(in_channels, growth_rate * bottleneck_size, 1,␣
↪bias=False)

        # Main layer to extract features and produce growth_rate activation maps
        self.norm2 = nn.BatchNorm2d(growth_rate * bottleneck_size)
        self.relu2 = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(growth_rate * bottleneck_size, growth_rate, 3,␣
↪padding=1, bias=False)

        # We may use dropout
```

```
        self.drop_rate = float(drop_rate)

    def forward(self, x):
        """Performs forward pass on the given input.

        Args:
            x (Tensor): Input data of dim (M, N, in_channels, H, W)

        Returns:
            output (Tensor): Output data of dim (N, growth_rate, H, W)
        """
        x = [x] if torch.is_tensor(x) else x                    # assure it's a␣
↪tensor so we could concatenate
        x = self.conv1(self.relu1(self.norm1(torch.cat(x, 1)))) # perform first␣
↪(bottleneck) pass
        output = self.conv2(self.relu2(self.norm2(x)))          # perform␣
↪second (feaure extraction) pass

        # Use dropout if drop rate is provided
        if self.drop_rate > 0:
            output = F.dropout(output, p=self.drop_rate, training=self.training)

        return output


class _DenseBlock(nn.ModuleDict):
    """A building block used in DenseNet."""

    def __init__(self, num_layers, in_channels, growth_rate, bottleneck_size,␣
↪drop_rate):
        """Initializes the dense block.

        The block constructs `num_layers` densely connected layers with shared
        hyperparameters, taking into account that the output of each layer is
        fed into _all_ other subsequent layers.

        Args:
            num_layers (int):      The number of layers this block will have
            in_channels (int):     The number of input channels
            growth_rate (int):     The number of output channels for each␣
↪layer
            bottleneck_size (int): The bottleneck before main convolution for␣
↪each layer
            drop_rate (float):     The dropout hyperparameter
        """
        super().__init__()
```

```
        # Loop through every layer and iniatialize it
        for i in range(num_layers):
            layer = _DenseLayer(in_channels + i*growth_rate, growth_rate,␣
↪bottleneck_size, drop_rate)
            self.add_module(f'denselayer{i+1}', layer)

    def forward(self, x):
        """Performs forward pass for each layer.

        Args:
            x (Tensor): Input data of dim (N, in_channels, H, W)

        Returns:
            Output data of dim (N, growth_rate * num_layers, H, W)
        """
        xs = [x] # use a list of tensors that will be concatinated as inputs

        # Loop through every layer providing concatinated previous outputs as␣
↪inputs
        for name, layer in self.items():
            x_new = layer(xs)
            xs.append(x_new)

        return torch.cat(xs, 1)


class DenseNet(nn.Module):
    """Densely connected network

    The network has the following architecture:
        1. `CONV->NORM->RELU` to preprocess the input for a chain of dense␣
↪blocks
        2. `BLOCK->[TRANS->BLOCK] x N` where each block consists of arbitrary
           number of densely connected layers
        3. `NORM->RELU->POOL->LINEAR` where global average pooling is performed
           before calculating raw scores
    """

    def __init__(self, in_channels=32, growth_rate=16, bottleneck_size=4,
                 block_config=(6, 12, 8), drop_rate=0, num_classes=10):
        """Initializes the dense network.

        The first layer produces `in_channels` activation maps which are then␣
↪fed to a
        sequence of dense blocks containing a specified number of layers. There␣
↪is a
```

```
        transition layer between each block. At the end the _global average␣
↪pooling_
        layer is used to flatten the activations for the linear softmax␣
↪classifier.

        Args:
            block_config (tuple):  The number of layers each bloack should have␣
↪in sequence
            in_channels (int):     The number of input channels for the␣
↪sequence of blocks
            growth_rate (int):     The number of output channels for each layer␣
↪per block
            bottleneck_size (int): The bottleneck before main convolution for␣
↪each layer per block
            drop_rate (float):     The dropout hyperparameter
            num_classes (int):     The total number of classes
        """
        super().__init__()

        # Initialize the layers for preprocessing the input (preserves its␣
↪spacial dim)
        self.features = nn.Sequential(OrderedDict([
            ('conv0', nn.Conv2d(3, in_channels, 7, padding=3, bias=False)),
            ('norm0', nn.BatchNorm2d(in_channels)),
            ('relu0', nn.ReLU(inplace=True)),
        ]))

        num_features = in_channels # input size that will be updated for each␣
↪block

        # Create the specified number of blocks (should be 3 for spacial dim␣
↪32x32)
        for i, num_layers in enumerate(block_config):
            # Create and add the dense block and update the number of channels␣
↪for the next block
            block = _DenseBlock(num_layers, num_features, growth_rate,␣
↪bottleneck_size, drop_rate)
            self.features.add_module(f'denseblock{i+1}', block)
            num_features += num_layers * growth_rate

            # Add a transition if it is not the last block
            if i != len(block_config) - 1:
                trans = _Transition(num_features, num_features // 2)
                self.features.add_module(f'transition{i+1}', trans)
                num_features = num_features // 2
```

```
        # Add the final batchnorm and ReLU layers before global average pooling
        self.features.add_module(f'norm{i+2}', nn.BatchNorm2d(num_features))
        self.features.add_module(f'relu{i+2}', nn.ReLU(inplace=True))

        # The final layer is our linear classifier
        self.classifier = nn.Linear(num_features, num_classes)

        # Official init from torch repo
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        """Performs forward pass for the whole network.

        Args:
            x (Tensor): input data of dim (N, 3, H, W)

        Returns:
            Output data of dim (N, 10)
        """
        out = F.adaptive_avg_pool2d(self.features(x), (1, 1)) # global average␣
↪pooling to flatten activations
        out = self.classifier(flatten(out))                   # classifier to␣
↪produces raw scores for every class

        return out
```

[23]:
```
################################################################################
# TODO:                                                                        #
# Experiment with any architectures, optimizers, and hyperparameters.          #
# Achieve AT LEAST 70% accuracy on the *validation set* within 10 epochs.      #
#                                                                              #
# Note that you can use the check_accuracy function to evaluate on either      #
# the test set or the validation set, by passing either loader_test or         #
# loader_val as the second argument to check_accuracy. You should not touch    #
# the test set until you have finished your architecture and  hyperparameter   #
# tuning, and only run the test set once at the end to report a final value.   #
################################################################################
model = None
optimizer = None
```

```python
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
import torch
import torchvision.models as models
import torch.nn as nn
learning_rate = .0015

# PyTorch'un hazır DenseNet modelini kullanarak modeli oluştur
model = models.densenet121(pretrained=False)  # Eğer önceden eğitilmiş bir␣
 ↪model kullanmak istiyorsanız 'pretrained=True' kullanabilirsiniz.
model.classifier = nn.Linear(model.classifier.in_features, 10)  # CIFAR-10 için␣
 ↪sınıf sayısını 10 olarak ayarla

# Adam optimizatörü kullanarak optimizatörü oluştur
optimizer = optim.Adam(model.parameters(), lr=learning_rate)

# Modeli eğit
train_part34(model, optimizer, epochs=10)

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

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)
```

```
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:208:
UserWarning: The parameter 'pretrained' is deprecated since 0.13 and may be
removed in the future, please use 'weights' instead.
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/torchvision/models/_utils.py:223:
UserWarning: Arguments other than a weight enum or `None` for 'weights' are
deprecated since 0.13 and may be removed in the future. The current behavior is
equivalent to passing `weights=None`.
  warnings.warn(msg)
```

```
Iteration 0, loss = 2.3299
Checking accuracy on validation set
Got 99 / 1000 correct (9.90)

Iteration 100, loss = 1.6429
Checking accuracy on validation set
Got 394 / 1000 correct (39.40)

Iteration 200, loss = 1.5578
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)
```

```
Iteration 300, loss = 1.4472
Checking accuracy on validation set
Got 520 / 1000 correct (52.00)

Iteration 400, loss = 1.3004
Checking accuracy on validation set
Got 543 / 1000 correct (54.30)

Iteration 500, loss = 1.1829
Checking accuracy on validation set
Got 576 / 1000 correct (57.60)

Iteration 600, loss = 1.1388
Checking accuracy on validation set
Got 555 / 1000 correct (55.50)

Iteration 700, loss = 1.0828
Checking accuracy on validation set
Got 610 / 1000 correct (61.00)

Iteration 0, loss = 0.9438
Checking accuracy on validation set
Got 607 / 1000 correct (60.70)

Iteration 100, loss = 1.4488
Checking accuracy on validation set
Got 537 / 1000 correct (53.70)

Iteration 200, loss = 0.9468
Checking accuracy on validation set
Got 645 / 1000 correct (64.50)

Iteration 300, loss = 0.9167
Checking accuracy on validation set
Got 618 / 1000 correct (61.80)

Iteration 400, loss = 1.1793
Checking accuracy on validation set
Got 631 / 1000 correct (63.10)

Iteration 500, loss = 0.8819
Checking accuracy on validation set
Got 643 / 1000 correct (64.30)

Iteration 600, loss = 0.7541
Checking accuracy on validation set
Got 653 / 1000 correct (65.30)
```

```
Iteration 700, loss = 0.8082
Checking accuracy on validation set
Got 671 / 1000 correct (67.10)

Iteration 0, loss = 0.8613
Checking accuracy on validation set
Got 697 / 1000 correct (69.70)

Iteration 100, loss = 0.7452
Checking accuracy on validation set
Got 710 / 1000 correct (71.00)

Iteration 200, loss = 0.8885
Checking accuracy on validation set
Got 690 / 1000 correct (69.00)

Iteration 300, loss = 0.6886
Checking accuracy on validation set
Got 676 / 1000 correct (67.60)

Iteration 400, loss = 0.9776
Checking accuracy on validation set
Got 701 / 1000 correct (70.10)

Iteration 500, loss = 0.6314
Checking accuracy on validation set
Got 709 / 1000 correct (70.90)

Iteration 600, loss = 0.9144
Checking accuracy on validation set
Got 750 / 1000 correct (75.00)

Iteration 700, loss = 0.7401
Checking accuracy on validation set
Got 693 / 1000 correct (69.30)

Iteration 0, loss = 0.7823
Checking accuracy on validation set
Got 679 / 1000 correct (67.90)

Iteration 100, loss = 0.5686
Checking accuracy on validation set
Got 757 / 1000 correct (75.70)

Iteration 200, loss = 0.5813
Checking accuracy on validation set
Got 736 / 1000 correct (73.60)
```

```
Iteration 300, loss = 0.7314
Checking accuracy on validation set
Got 738 / 1000 correct (73.80)

Iteration 400, loss = 0.6043
Checking accuracy on validation set
Got 740 / 1000 correct (74.00)

Iteration 500, loss = 0.4694
Checking accuracy on validation set
Got 751 / 1000 correct (75.10)

Iteration 600, loss = 0.9439
Checking accuracy on validation set
Got 759 / 1000 correct (75.90)

Iteration 700, loss = 0.6353
Checking accuracy on validation set
Got 745 / 1000 correct (74.50)

Iteration 0, loss = 0.6486
Checking accuracy on validation set
Got 770 / 1000 correct (77.00)

Iteration 100, loss = 0.4146
Checking accuracy on validation set
Got 752 / 1000 correct (75.20)

Iteration 200, loss = 0.7363
Checking accuracy on validation set
Got 750 / 1000 correct (75.00)

Iteration 300, loss = 0.4194
Checking accuracy on validation set
Got 760 / 1000 correct (76.00)

Iteration 400, loss = 0.6774
Checking accuracy on validation set
Got 750 / 1000 correct (75.00)

Iteration 500, loss = 0.6957
Checking accuracy on validation set
Got 739 / 1000 correct (73.90)

Iteration 600, loss = 0.4751
Checking accuracy on validation set
Got 780 / 1000 correct (78.00)
```

```
Iteration 700, loss = 0.5136
Checking accuracy on validation set
Got 770 / 1000 correct (77.00)

Iteration 0, loss = 0.4590
Checking accuracy on validation set
Got 777 / 1000 correct (77.70)

Iteration 100, loss = 0.2122
Checking accuracy on validation set
Got 777 / 1000 correct (77.70)

Iteration 200, loss = 0.4542
Checking accuracy on validation set
Got 766 / 1000 correct (76.60)

Iteration 300, loss = 0.2859
Checking accuracy on validation set
Got 752 / 1000 correct (75.20)

Iteration 400, loss = 0.6130
Checking accuracy on validation set
Got 755 / 1000 correct (75.50)

Iteration 500, loss = 0.5189
Checking accuracy on validation set
Got 751 / 1000 correct (75.10)

Iteration 600, loss = 0.3743
Checking accuracy on validation set
Got 779 / 1000 correct (77.90)

Iteration 700, loss = 0.5837
Checking accuracy on validation set
Got 712 / 1000 correct (71.20)

Iteration 0, loss = 0.6621
Checking accuracy on validation set
Got 781 / 1000 correct (78.10)

Iteration 100, loss = 0.3598
Checking accuracy on validation set
Got 794 / 1000 correct (79.40)

Iteration 200, loss = 0.4177
Checking accuracy on validation set
Got 775 / 1000 correct (77.50)
```

```
Iteration 300, loss = 0.3309
Checking accuracy on validation set
Got 778 / 1000 correct (77.80)

Iteration 400, loss = 0.4209
Checking accuracy on validation set
Got 776 / 1000 correct (77.60)

Iteration 500, loss = 0.4293
Checking accuracy on validation set
Got 766 / 1000 correct (76.60)

Iteration 600, loss = 0.6324
Checking accuracy on validation set
Got 785 / 1000 correct (78.50)

Iteration 700, loss = 0.2493
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 0, loss = 0.4688
Checking accuracy on validation set
Got 789 / 1000 correct (78.90)

Iteration 100, loss = 0.3228
Checking accuracy on validation set
Got 784 / 1000 correct (78.40)

Iteration 200, loss = 0.3002
Checking accuracy on validation set
Got 783 / 1000 correct (78.30)

Iteration 300, loss = 0.3271
Checking accuracy on validation set
Got 787 / 1000 correct (78.70)

Iteration 400, loss = 0.5229
Checking accuracy on validation set
Got 789 / 1000 correct (78.90)

Iteration 500, loss = 0.2769
Checking accuracy on validation set
Got 776 / 1000 correct (77.60)

Iteration 600, loss = 0.2935
Checking accuracy on validation set
Got 771 / 1000 correct (77.10)
```

```
Iteration 700, loss = 0.2541
Checking accuracy on validation set
Got 777 / 1000 correct (77.70)

Iteration 0, loss = 0.3178
Checking accuracy on validation set
Got 789 / 1000 correct (78.90)

Iteration 100, loss = 0.3965
Checking accuracy on validation set
Got 808 / 1000 correct (80.80)

Iteration 200, loss = 0.2472
Checking accuracy on validation set
Got 802 / 1000 correct (80.20)

Iteration 300, loss = 0.1515
Checking accuracy on validation set
Got 782 / 1000 correct (78.20)

Iteration 400, loss = 0.3214
Checking accuracy on validation set
Got 790 / 1000 correct (79.00)

Iteration 500, loss = 0.1850
Checking accuracy on validation set
Got 787 / 1000 correct (78.70)

Iteration 600, loss = 0.2331
Checking accuracy on validation set
Got 776 / 1000 correct (77.60)

Iteration 700, loss = 0.0950
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 0, loss = 0.2172
Checking accuracy on validation set
Got 780 / 1000 correct (78.00)

Iteration 100, loss = 0.1606
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 200, loss = 0.2611
Checking accuracy on validation set
Got 817 / 1000 correct (81.70)
```

```
Iteration 300, loss = 0.2856
Checking accuracy on validation set
Got 789 / 1000 correct (78.90)

Iteration 400, loss = 0.1038
Checking accuracy on validation set
Got 786 / 1000 correct (78.60)

Iteration 500, loss = 0.2489
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 600, loss = 0.2233
Checking accuracy on validation set
Got 795 / 1000 correct (79.50)

Iteration 700, loss = 0.2476
Checking accuracy on validation set
Got 793 / 1000 correct (79.30)

Iteration 0, loss = 0.1276
Checking accuracy on validation set
Got 775 / 1000 correct (77.50)

Iteration 100, loss = 0.1136
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 200, loss = 0.1716
Checking accuracy on validation set
Got 804 / 1000 correct (80.40)

Iteration 300, loss = 0.1503
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 400, loss = 0.2089
Checking accuracy on validation set
Got 795 / 1000 correct (79.50)

Iteration 500, loss = 0.2314
Checking accuracy on validation set
Got 797 / 1000 correct (79.70)

Iteration 600, loss = 0.2014
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)
```

```
Iteration 700, loss = 0.1610
Checking accuracy on validation set
Got 783 / 1000 correct (78.30)

Iteration 0, loss = 0.1679
Checking accuracy on validation set
Got 811 / 1000 correct (81.10)

Iteration 100, loss = 0.0572
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)

Iteration 200, loss = 0.1744
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 300, loss = 0.0876
Checking accuracy on validation set
Got 804 / 1000 correct (80.40)

Iteration 400, loss = 0.1557
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 500, loss = 0.0804
Checking accuracy on validation set
Got 790 / 1000 correct (79.00)

Iteration 600, loss = 0.2516
Checking accuracy on validation set
Got 815 / 1000 correct (81.50)

Iteration 700, loss = 0.1223
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 0, loss = 0.1332
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 100, loss = 0.2312
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)

Iteration 200, loss = 0.0633
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)
```

```
Iteration 300, loss = 0.1458
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 400, loss = 0.1642
Checking accuracy on validation set
Got 795 / 1000 correct (79.50)

Iteration 500, loss = 0.2758
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 600, loss = 0.1176
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 700, loss = 0.1016
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)

Iteration 0, loss = 0.0844
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 100, loss = 0.0686
Checking accuracy on validation set
Got 813 / 1000 correct (81.30)

Iteration 200, loss = 0.2820
Checking accuracy on validation set
Got 814 / 1000 correct (81.40)

Iteration 300, loss = 0.0258
Checking accuracy on validation set
Got 800 / 1000 correct (80.00)

Iteration 400, loss = 0.1306
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 500, loss = 0.0433
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 600, loss = 0.0864
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)
```

```
Iteration 700, loss = 0.0573
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)

Iteration 0, loss = 0.1016
Checking accuracy on validation set
Got 807 / 1000 correct (80.70)

Iteration 100, loss = 0.0756
Checking accuracy on validation set
Got 821 / 1000 correct (82.10)

Iteration 200, loss = 0.1790
Checking accuracy on validation set
Got 820 / 1000 correct (82.00)

Iteration 300, loss = 0.1381
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 400, loss = 0.1508
Checking accuracy on validation set
Got 785 / 1000 correct (78.50)

Iteration 500, loss = 0.1161
Checking accuracy on validation set
Got 802 / 1000 correct (80.20)

Iteration 600, loss = 0.2060
Checking accuracy on validation set
Got 785 / 1000 correct (78.50)

Iteration 700, loss = 0.0142
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 0, loss = 0.0355
Checking accuracy on validation set
Got 802 / 1000 correct (80.20)

Iteration 100, loss = 0.0937
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 200, loss = 0.0606
Checking accuracy on validation set
Got 799 / 1000 correct (79.90)
```

```
Iteration 300, loss = 0.1453
Checking accuracy on validation set
Got 792 / 1000 correct (79.20)

Iteration 400, loss = 0.1130
Checking accuracy on validation set
Got 792 / 1000 correct (79.20)

Iteration 500, loss = 0.2172
Checking accuracy on validation set
Got 814 / 1000 correct (81.40)

Iteration 600, loss = 0.1503
Checking accuracy on validation set
Got 812 / 1000 correct (81.20)

Iteration 700, loss = 0.1923
Checking accuracy on validation set
Got 802 / 1000 correct (80.20)

Iteration 0, loss = 0.0252
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 100, loss = 0.0370
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 200, loss = 0.0869
Checking accuracy on validation set
Got 800 / 1000 correct (80.00)

Iteration 300, loss = 0.0903
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 400, loss = 0.1585
Checking accuracy on validation set
Got 800 / 1000 correct (80.00)

Iteration 500, loss = 0.1017
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 600, loss = 0.0755
Checking accuracy on validation set
Got 781 / 1000 correct (78.10)
```

```
Iteration 700, loss = 0.0844
Checking accuracy on validation set
Got 793 / 1000 correct (79.30)

Iteration 0, loss = 0.0186
Checking accuracy on validation set
Got 792 / 1000 correct (79.20)

Iteration 100, loss = 0.0468
Checking accuracy on validation set
Got 812 / 1000 correct (81.20)

Iteration 200, loss = 0.0240
Checking accuracy on validation set
Got 806 / 1000 correct (80.60)

Iteration 300, loss = 0.0199
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 400, loss = 0.0559
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 500, loss = 0.0948
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 600, loss = 0.1686
Checking accuracy on validation set
Got 805 / 1000 correct (80.50)

Iteration 700, loss = 0.0130
Checking accuracy on validation set
Got 812 / 1000 correct (81.20)

Iteration 0, loss = 0.0431
Checking accuracy on validation set
Got 806 / 1000 correct (80.60)

Iteration 100, loss = 0.0692
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)

Iteration 200, loss = 0.0319
Checking accuracy on validation set
Got 802 / 1000 correct (80.20)
```

```
Iteration 300, loss = 0.1269
Checking accuracy on validation set
Got 791 / 1000 correct (79.10)

Iteration 400, loss = 0.1984
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)

Iteration 500, loss = 0.0451
Checking accuracy on validation set
Got 801 / 1000 correct (80.10)

Iteration 600, loss = 0.1815
Checking accuracy on validation set
Got 815 / 1000 correct (81.50)

Iteration 700, loss = 0.1933
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 0, loss = 0.0569
Checking accuracy on validation set
Got 798 / 1000 correct (79.80)

Iteration 100, loss = 0.1248
Checking accuracy on validation set
Got 811 / 1000 correct (81.10)

Iteration 200, loss = 0.0403
Checking accuracy on validation set
Got 798 / 1000 correct (79.80)

Iteration 300, loss = 0.1448
Checking accuracy on validation set
Got 804 / 1000 correct (80.40)

Iteration 400, loss = 0.1387
Checking accuracy on validation set
Got 796 / 1000 correct (79.60)

Iteration 500, loss = 0.0106
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)

Iteration 600, loss = 0.0228
Checking accuracy on validation set
Got 809 / 1000 correct (80.90)
```

```
Iteration 700, loss = 0.1066
Checking accuracy on validation set
Got 808 / 1000 correct (80.80)
```

8.1 Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

Answer:

8.1.1 Network design

I used a **DenseNet** (paper) with the following architecture: 1. `CONV->NORM->RELU` * As the input is of shape `32x32x3`, initially I used a convolution layer with filter size `7x7` to produce `32` *activation maps*. Padding of `3` was used to preserve the input spacial dimension. * The `32` *activation maps*, each of shape `32x32`, were then passed through *batchnorm* layer and through *ReLU* nonlinearity 2. `BLOCK->TRANS->BLOCK->TRANS->BLOCK` * In general each block has a certain number of layers, each of which perfom a *convolution*, *batchnorm* and *ReLU*. Each layer concatinates its input with every other layer's output within the same block. Throughout the convolutions in the block, the spacial size of each *activation map* is preserved, however, the channel size grows by a constant of `16` after each layer produces an output. * Each transition layer simply performs a *convolution* to reduce the channel size by a factor of `2`. We also have *average pooling* whith reducs the spatial dimension also by a factor of `2`. As in any other group of layers, there is a *normalization* layer and *ReLU* nonlinearity. * I used `6` layers in the first block, `12` in the second and `8` in the final one. Thus the 'preprocessed' input of size `32x32x32` after each block and transition had its shape modified in had its dimensions changed in the following way `32x32x128->16x16x64->16x16x256->8x8x128->8x8x256` (based on the provided sequence of blocks and transitions). 3. `NORM->RELU->POOL->SOFTMAX` * The raw activation maps are passed through *batchnorm* and *ReLU*, a *global average pooling* is performed to get the final activation shape of `1x1x256` which is then flattened to a linear classifier which produces scores based on the *Softmax* loss function.

> Key motivation is that by allowing subsequent layers to see the outputs of every previous layer it makes the network more rubust in recognizing certain details. For example, if one layer detects edges and a subsequent layer detects shapes, sometimes it may be better to see the shape before deciding on the edge.

8.1.2 Hyperparameters

I only fine-tuned 3 hyperparameters - *learning rate*, *weight decay* and *drop rate*. As it turns out, dropout and regularization do not impact performance much, in many cases - even make it worse, possibly due to heavy use of batch normalization which already make sthe network robust. I settled on the learning rate of `0.0015`, larger values make the validation accuracy too unstable during training.

8.2 Test set – run this only once

Now that we've gotten a result we're happy with, we test our final model on the test set (which you should store in best_model). Think about how this compares to your validation set accuracy.

[24]:
```
best_model = model
check_accuracy_part34(loader_test, best_model)
```

```
Checking accuracy on test set
Got 7860 / 10000 correct (78.60)
```

TensorFlow

November 29, 2023

[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/assignment1/'
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 Introduction to TensorFlow

You've written a lot of code in this assignment to provide a whole host of neural network functionality. Dropout, Batch Norm, and 2D convolutions are some of the workhorses of deep learning in computer vision. You've also worked hard to make your code efficient and vectorized.

For the last part of this assignment, though, we're going to leave behind your beautiful codebase and instead migrate to one of two popular deep learning frameworks: in this instance, TensorFlow (or PyTorch, if you choose to work with that notebook).

1.1 Why do we use deep learning frameworks?

- Our code will now run on GPUs! This will allow our models to train much faster. When using a framework like PyTorch or TensorFlow you can harness the power of the GPU for your own custom neural network architectures without having to write CUDA code directly (which is beyond the scope of this class).
- In this class, we want you to be ready to use one of these frameworks for your project so you can experiment more efficiently than if you were writing every feature you want to use by hand.
- We want you to stand on the shoulders of giants! TensorFlow and PyTorch are both excellent frameworks that will make your lives a lot easier, and now that you understand their guts, you are free to use them :)
- Finally, we want you to be exposed to the sort of deep learning code you might run into in academia or industry.

1.2 What is TensorFlow?

TensorFlow is a system for executing computational graphs over Tensor objects, with native support for performing backpropogation for its Variables. In it, we work with Tensors which are n-dimensional arrays analogous to the numpy ndarray.

1.3 How do I learn TensorFlow?

TensorFlow has many excellent tutorials available, including those from Google themselves.

Otherwise, this notebook will walk you through much of what you need to do to train models in TensorFlow. See the end of the notebook for some links to helpful tutorials if you want to learn more or need further clarification on topics that aren't fully explained here.

Note: This notebook is meant to teach you Tensorflow 2.x. Most examples on the web today are still in 1.x, so be careful not to confuse the two when looking up documentation.

2 Table of Contents

This notebook has 5 parts. We will walk through TensorFlow at **three different levels of abstraction**, which should help you better understand it and prepare you for working on your project.

1. Part I, Preparation: load the CIFAR-10 dataset.
2. Part II, Barebone TensorFlow: **Abstraction Level 1**, we will work directly with low-level TensorFlow graphs.
3. Part III, Keras Model API: **Abstraction Level 2**, we will use `tf.keras.Model` to define arbitrary neural network architecture.
4. Part IV, Keras Sequential + Functional API: **Abstraction Level 3**, we will use `tf.keras.Sequential` to define a linear feed-forward network very conveniently, and then explore the functional libraries for building unique and uncommon models that require more flexibility.
5. Part V, CIFAR-10 open-ended challenge: please implement your own network to get as high accuracy as possible on CIFAR-10. You can experiment with any layer, optimizer, hyperparameters or other advanced features.

We will discuss Keras in more detail later in the notebook.

Here is a table of comparison:

API	Flexibility	Convenience
Barebone	High	Low
`tf.keras.Model`	High	Medium
`tf.keras.Sequential`	Low	High

3 GPU

You can manually switch to a GPU device on Colab by clicking `Runtime -> Change runtime type` and selecting `GPU` under `Hardware Accelerator`. You should do this before running the following cells to import packages, since the kernel gets restarted upon switching runtimes.

[2]:
```
import os
import tensorflow as tf
import numpy as np
import math
import timeit
import matplotlib.pyplot as plt

%matplotlib inline

USE_GPU = True

if USE_GPU:
    device = '/device:GPU:0'
else:
    device = '/cpu:0'

# Constant to control how often we print when training models.
print_every = 100
print('Using device: ', device)
```

```
Using device:  /device:GPU:0
```

4 Part I: Preparation

First, we load the CIFAR-10 dataset. This might take a few minutes to download the first time you run it, but after that the files should be cached on disk and loading should be faster.

In previous parts of the assignment we used CS231N-specific code to download and read the CIFAR-10 dataset; however the `tf.keras.datasets` package in TensorFlow provides prebuilt utility functions for loading many common datasets.

For the purposes of this assignment we will still write our own code to preprocess the data and iterate through it in minibatches. The `tf.data` package in TensorFlow provides tools for automating this

process, but working with this package adds extra complication and is beyond the scope of this notebook. However using `tf.data` can be much more efficient than the simple approach used in this notebook, so you should consider using it for your project.

[3]:
```
def load_cifar10(num_training=49000, num_validation=1000, num_test=10000):
    """
    Fetch the CIFAR-10 dataset from the web and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 dataset and use appropriate data types and shapes
    cifar10 = tf.keras.datasets.cifar10.load_data()
    (X_train, y_train), (X_test, y_test) = cifar10
    X_train = np.asarray(X_train, dtype=np.float32)
    y_train = np.asarray(y_train, dtype=np.int32).flatten()
    X_test = np.asarray(X_test, dtype=np.float32)
    y_test = np.asarray(y_test, dtype=np.int32).flatten()

    # Subsample the data
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean pixel and divide by std
    mean_pixel = X_train.mean(axis=(0, 1, 2), keepdims=True)
    std_pixel = X_train.std(axis=(0, 1, 2), keepdims=True)
    X_train = (X_train - mean_pixel) / std_pixel
    X_val = (X_val - mean_pixel) / std_pixel
    X_test = (X_test - mean_pixel) / std_pixel

    return X_train, y_train, X_val, y_val, X_test, y_test

# If there are errors with SSL downloading involving self-signed certificates,
# it may be that your Python version was recently installed on the current␣
 ↪machine.
# See: https://github.com/tensorflow/tensorflow/issues/10779
# To fix, run the command: /Applications/Python\ 3.7/Install\ Certificates.
 ↪command
#   ...replacing paths as necessary.

# Invoke the above function to get our data.
```

```
NHW = (0, 1, 2)
X_train, y_train, X_val, y_val, X_test, y_test = load_cifar10()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape, y_train.dtype)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170500096/170498071 [==============================] - 3s 0us/step
170508288/170498071 [==============================] - 3s 0us/step
Train data shape:  (49000, 32, 32, 3)
Train labels shape:  (49000,) int32
Validation data shape:  (1000, 32, 32, 3)
Validation labels shape:  (1000,)
Test data shape:  (10000, 32, 32, 3)
Test labels shape:  (10000,)
```

[4]:
```
class Dataset(object):
    def __init__(self, X, y, batch_size, shuffle=False):
        """
        Construct a Dataset object to iterate over data X and labels y

        Inputs:
        - X: Numpy array of data, of any shape
        - y: Numpy array of labels, of any shape but with y.shape[0] == X.
 ↪shape[0]
        - batch_size: Integer giving number of elements per minibatch
        - shuffle: (optional) Boolean, whether to shuffle the data on each epoch
        """
        assert X.shape[0] == y.shape[0], 'Got different numbers of data and␣
 ↪labels'
        self.X, self.y = X, y
        self.batch_size, self.shuffle = batch_size, shuffle

    def __iter__(self):
        N, B = self.X.shape[0], self.batch_size
        idxs = np.arange(N)
        if self.shuffle:
            np.random.shuffle(idxs)
        return iter((self.X[i:i+B], self.y[i:i+B]) for i in range(0, N, B))


train_dset = Dataset(X_train, y_train, batch_size=64, shuffle=True)
val_dset = Dataset(X_val, y_val, batch_size=64, shuffle=False)
test_dset = Dataset(X_test, y_test, batch_size=64)
```

[5]:
```
# We can iterate through a dataset like this:
for t, (x, y) in enumerate(train_dset):
    print(t, x.shape, y.shape)
    if t > 5: break
```

```
0 (64, 32, 32, 3) (64,)
1 (64, 32, 32, 3) (64,)
2 (64, 32, 32, 3) (64,)
3 (64, 32, 32, 3) (64,)
4 (64, 32, 32, 3) (64,)
5 (64, 32, 32, 3) (64,)
6 (64, 32, 32, 3) (64,)
```

5 Part II: Barebones TensorFlow

TensorFlow ships with various high-level APIs which make it very convenient to define and train neural networks; we will cover some of these constructs in Part III and Part IV of this notebook. In this section we will start by building a model with basic TensorFlow constructs to help you better understand what's going on under the hood of the higher-level APIs.

"Barebones Tensorflow" is important to understanding the building blocks of TensorFlow, but much of it involves concepts from TensorFlow 1.x. We will be working with legacy modules such as `tf.Variable`.

Therefore, please read and understand the differences between legacy (1.x) TF and the new (2.0) TF.

5.0.1 Historical background on TensorFlow 1.x

TensorFlow 1.x is primarily a framework for working with **static computational graphs**. Nodes in the computational graph are Tensors which will hold n-dimensional arrays when the graph is run; edges in the graph represent functions that will operate on Tensors when the graph is run to actually perform useful computation.

Before Tensorflow 2.0, we had to configure the graph into two phases. There are plenty of tutorials online that explain this two-step process. The process generally looks like the following for TF 1.x: 1. **Build a computational graph that describes the computation that you want to perform**. This stage doesn't actually perform any computation; it just builds up a symbolic representation of your computation. This stage will typically define one or more `placeholder` objects that represent inputs to the computational graph. 2. **Run the computational graph many times.** Each time the graph is run (e.g. for one gradient descent step) you will specify which parts of the graph you want to compute, and pass a `feed_dict` dictionary that will give concrete values to any `placeholder`s in the graph.

5.0.2 The new paradigm in Tensorflow 2.0

Now, with Tensorflow 2.0, we can simply adopt a functional form that is more Pythonic and similar in spirit to PyTorch and direct Numpy operation. Instead of the 2-step paradigm with computation

graphs, making it (among other things) easier to debug TF code. You can read more details at https://www.tensorflow.org/guide/eager.

The main difference between the TF 1.x and 2.0 approach is that the 2.0 approach doesn't make use of `tf.Session`, `tf.run`, `placeholder`, `feed_dict`. To get more details of what's different between the two version and how to convert between the two, check out the official migration guide: https://www.tensorflow.org/alpha/guide/migration_guide

Later, in the rest of this notebook we'll focus on this new, simpler approach.

5.0.3 TensorFlow warmup: Flatten Function

We can see this in action by defining a simple `flatten` function that will reshape image data for use in a fully-connected network.

In TensorFlow, data for convolutional feature maps is typically stored in a Tensor of shape N x H x W x C where:

- N is the number of datapoints (minibatch size)
- H is the height of the feature map
- W is the width of the feature map
- C is the number of channels in the feature map

This is the right way to represent the data when we are doing something like a 2D convolution, that needs spatial understanding of where the intermediate features are relative to each other. When we use fully connected affine layers to process the image, however, we want each datapoint to be represented by a single vector – it's no longer useful to segregate the different channels, rows, and columns of the data. So, we use a "flatten" operation to collapse the `H x W x C` values per representation into a single long vector.

Notice the `tf.reshape` call has the target shape as `(N, -1)`, meaning it will reshape/keep the first dimension to be N, and then infer as necessary what the second dimension is in the output, so we can collapse the remaining dimensions from the input properly.

NOTE: TensorFlow and PyTorch differ on the default Tensor layout; TensorFlow uses N x H x W x C but PyTorch uses N x C x H x W.

[6]:
```
def flatten(x):
    """
    Input:
    - TensorFlow Tensor of shape (N, D1, ..., DM)

    Output:
    - TensorFlow Tensor of shape (N, D1 * ... * DM)
    """
    N = tf.shape(x)[0]
    return tf.reshape(x, (N, -1))
```

[7]:
```
def test_flatten():
    # Construct concrete values of the input data x using numpy
    x_np = np.arange(24).reshape((2, 3, 4))
```

```
    print('x_np:\n', x_np, '\n')
    # Compute a concrete output value.
    x_flat_np = flatten(x_np)
    print('x_flat_np:\n', x_flat_np, '\n')

test_flatten()
```

```
x_np:
 [[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]

x_flat_np:
 tf.Tensor(
[[ 0  1  2  3  4  5  6  7  8  9 10 11]
 [12 13 14 15 16 17 18 19 20 21 22 23]], shape=(2, 12), dtype=int64)
```

5.0.4 Barebones TensorFlow: Define a Two-Layer Network

We will now implement our first neural network with TensorFlow: a fully-connected ReLU network with two hidden layers and no biases on the CIFAR10 dataset. For now we will use only low-level TensorFlow operators to define the network; later we will see how to use the higher-level abstractions provided by `tf.keras` to simplify the process.

We will define the forward pass of the network in the function `two_layer_fc`; this will accept TensorFlow Tensors for the inputs and weights of the network, and return a TensorFlow Tensor for the scores.

After defining the network architecture in the `two_layer_fc` function, we will test the implementation by checking the shape of the output.

It's important that you read and understand this implementation.

[14]:
```
def two_layer_fc(x, params):
    """
    A fully-connected neural network; the architecture is:
    fully-connected layer -> ReLU -> fully connected layer.
    Note that we only need to define the forward pass here; TensorFlow will take
    care of computing the gradients for us.

    The input to the network will be a minibatch of data, of shape
    (N, d1, ..., dM) where d1 * ... * dM = D. The hidden layer will have H␣
 ↪units,
    and the output layer will produce scores for C classes.
```

```
    Inputs:
    - x: A TensorFlow Tensor of shape (N, d1, ..., dM) giving a minibatch of
      input data.
    - params: A list [w1, w2] of TensorFlow Tensors giving weights for the
      network, where w1 has shape (D, H) and w2 has shape (H, C).

    Returns:
    - scores: A TensorFlow Tensor of shape (N, C) giving classification scores
      for the input data x.
    """
    w1, w2 = params                   # Unpack the parameters
    x = flatten(x)                    # Flatten the input; now x has shape (N,␣
↪D)
    h = tf.nn.relu(tf.matmul(x, w1))  # Hidden layer: h has shape (N, H)
    scores = tf.matmul(h, w2)         # Compute scores of shape (N, C)
    return scores
```

[15]:
```
def two_layer_fc_test():
    hidden_layer_size = 42

    # Scoping our TF operations under a tf.device context manager
    # lets us tell TensorFlow where we want these Tensors to be
    # multiplied and/or operated on, e.g. on a CPU or a GPU.
    with tf.device(device):
        x = tf.zeros((64, 32, 32, 3))
        w1 = tf.zeros((32 * 32 * 3, hidden_layer_size))
        w2 = tf.zeros((hidden_layer_size, 10))

        # Call our two_layer_fc function for the forward pass of the network.
        scores = two_layer_fc(x, [w1, w2])

    print(scores.shape)

two_layer_fc_test()
```

```
(64, 10)
```

5.0.5 Barebones TensorFlow: Three-Layer ConvNet

Here you will complete the implementation of the function `three_layer_convnet` which will perform the forward pass of a three-layer convolutional network. The network should have the following architecture:

1. A convolutional layer (with bias) with `channel_1` filters, each with shape `KW1 x KH1`, and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape `KW2 x KH2`, and

zero-padding of one
4. ReLU nonlinearity
5. Fully-connected layer with bias, producing scores for `C` classes.

HINT: For convolutions: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/nn/conv2d; be careful with padding!

HINT: For biases: https://www.tensorflow.org/performance/xla/broadcasting

[16]:
```
def three_layer_convnet(x, params):
    """
    A three-layer convolutional network with the architecture described above.

    Inputs:
    - x: A TensorFlow Tensor of shape (N, H, W, 3) giving a minibatch of images
    - params: A list of TensorFlow Tensors giving the weights and biases for the
      network; should contain the following:
      - conv_w1: TensorFlow Tensor of shape (KH1, KW1, 3, channel_1) giving
        weights for the first convolutional layer.
      - conv_b1: TensorFlow Tensor of shape (channel_1,) giving biases for the
        first convolutional layer.
      - conv_w2: TensorFlow Tensor of shape (KH2, KW2, channel_1, channel_2)
        giving weights for the second convolutional layer
      - conv_b2: TensorFlow Tensor of shape (channel_2,) giving biases for the
        second convolutional layer.
      - fc_w: TensorFlow Tensor giving weights for the fully-connected layer.
        Can you figure out what the shape should be?
      - fc_b: TensorFlow Tensor giving biases for the fully-connected layer.
        Can you figure out what the shape should be?
    """
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None
    ################################################################################
    # TODO: Implement the forward pass for the three-layer ConvNet.                #
    ################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    x = tf.nn.relu(tf.nn.conv2d(x, conv_w1, 1, 'SAME') + conv_b1)
    x = tf.nn.relu(tf.nn.conv2d(x, conv_w2, 1, 'SAME') + conv_b2)
    scores = tf.matmul(flatten(x), fc_w) + fc_b

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

After defing the forward pass of the three-layer ConvNet above, run the following cell to test your implementation. Like the two-layer network, we run the graph on a batch of zeros just to make

sure the function doesn't crash, and produces outputs of the correct shape.

When you run this function, `scores_np` should have shape `(64, 10)`.

[17]:
```
def three_layer_convnet_test():

    with tf.device(device):
        x = tf.zeros((64, 32, 32, 3))
        conv_w1 = tf.zeros((5, 5, 3, 6))
        conv_b1 = tf.zeros((6,))
        conv_w2 = tf.zeros((3, 3, 6, 9))
        conv_b2 = tf.zeros((9,))
        fc_w = tf.zeros((32 * 32 * 9, 10))
        fc_b = tf.zeros((10,))
        params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
        scores = three_layer_convnet(x, params)

    # Inputs to convolutional layers are 4-dimensional arrays with shape
    # [batch_size, height, width, channels]
    print('scores_np has shape: ', scores.shape)

three_layer_convnet_test()
```

```
scores_np has shape:  (64, 10)
```

5.0.6 Barebones TensorFlow: Training Step

We now define the `training_step` function performs a single training step. This will take three basic steps:

1. Compute the loss
2. Compute the gradient of the loss with respect to all network weights
3. Make a weight update step using (stochastic) gradient descent.

We need to use a few new TensorFlow functions to do all of this: - For computing the cross-entropy loss we'll use `tf.nn.sparse_softmax_cross_entropy_with_logits`: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits

- For averaging the loss across a minibatch of data we'll use `tf.reduce_mean`: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/reduce_mean
- For computing gradients of the loss with respect to the weights we'll use `tf.GradientTape` (useful for Eager execution): https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/GradientTape
- We'll mutate the weight values stored in a TensorFlow Tensor using `tf.assign_sub` ("sub" is for subtraction): https://www.tensorflow.org/api_docs/python/tf/assign_sub

[8]:
```
def training_step(model_fn, x, y, params, learning_rate):
    with tf.GradientTape() as tape:
        scores = model_fn(x, params) # Forward pass of the model
```

```
        loss = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,␣
↪logits=scores)
        total_loss = tf.reduce_mean(loss)
        grad_params = tape.gradient(total_loss, params)

        # Make a vanilla gradient descent step on all of the model parameters
        # Manually update the weights using assign_sub()
        for w, grad_w in zip(params, grad_params):
            w.assign_sub(learning_rate * grad_w)

        return total_loss
```

[9]:
```
def train_part2(model_fn, init_fn, learning_rate):
    """
    Train a model on CIFAR-10.

    Inputs:
    - model_fn: A Python function that performs the forward pass of the model
      using TensorFlow; it should have the following signature:
      scores = model_fn(x, params) where x is a TensorFlow Tensor giving a
      minibatch of image data, params is a list of TensorFlow Tensors holding
      the model weights, and scores is a TensorFlow Tensor of shape (N, C)
      giving scores for all elements of x.
    - init_fn: A Python function that initializes the parameters of the model.
      It should have the signature params = init_fn() where params is a list
      of TensorFlow Tensors holding the (randomly initialized) weights of the
      model.
    - learning_rate: Python float giving the learning rate to use for SGD.
    """


    params = init_fn()  # Initialize the model parameters

    for t, (x_np, y_np) in enumerate(train_dset):
        # Run the graph on a batch of training data.
        loss = training_step(model_fn, x_np, y_np, params, learning_rate)

        # Periodically print the loss and check accuracy on the val set.
        if t % print_every == 0:
            print('Iteration %d, loss = %.4f' % (t, loss))
            check_accuracy(val_dset, x_np, model_fn, params)
```

[10]:
```
def check_accuracy(dset, x, model_fn, params):
    """
    Check accuracy on a classification model, e.g. for validation.

    Inputs:
```

```
    - dset: A Dataset object against which to check accuracy
    - x: A TensorFlow placeholder Tensor where input images should be fed
    - model_fn: the Model we will be calling to make predictions on x
    - params: parameters for the model_fn to work with

    Returns: Nothing, but prints the accuracy of the model
    """
    num_correct, num_samples = 0, 0
    for x_batch, y_batch in dset:
        scores_np = model_fn(x_batch, params).numpy()
        y_pred = scores_np.argmax(axis=1)
        num_samples += x_batch.shape[0]
        num_correct += (y_pred == y_batch).sum()
    acc = float(num_correct) / num_samples
    print('Got %d / %d correct (%.2f%%)' % (num_correct, num_samples, 100 *␣
↪acc))
```

5.0.7 Barebones TensorFlow: Initialization

We'll use the following utility method to initialize the weight matrices for our models using Kaiming's normalization method.

[1] He et al, *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification* , ICCV 2015, https://arxiv.org/abs/1502.01852

[11]:
```
def create_matrix_with_kaiming_normal(shape):
    if len(shape) == 2:
        fan_in, fan_out = shape[0], shape[1]
    elif len(shape) == 4:
        fan_in, fan_out = np.prod(shape[:3]), shape[3]
    return tf.keras.backend.random_normal(shape) * np.sqrt(2.0 / fan_in)
```

5.0.8 Barebones TensorFlow: Train a Two-Layer Network

We are finally ready to use all of the pieces defined above to train a two-layer fully-connected network on CIFAR-10.

We just need to define a function to initialize the weights of the model, and call `train_part2`.

Defining the weights of the network introduces another important piece of TensorFlow API: `tf.Variable`. A TensorFlow Variable is a Tensor whose value is stored in the graph and persists across runs of the computational graph; however unlike constants defined with `tf.zeros` or `tf.random_normal`, the values of a Variable can be mutated as the graph runs; these mutations will persist across graph runs. Learnable parameters of the network are usually stored in Variables.

You don't need to tune any hyperparameters, but you should achieve validation accuracies above 40% after one epoch of training.

[18]:
```
def two_layer_fc_init():
    """
    Initialize the weights of a two-layer network, for use with the
    two_layer_network function defined above.
    You can use the `create_matrix_with_kaiming_normal` helper!

    Inputs: None

    Returns: A list of:
    - w1: TensorFlow tf.Variable giving the weights for the first layer
    - w2: TensorFlow tf.Variable giving the weights for the second layer
    """
    hidden_layer_size = 4000
    w1 = tf.Variable(create_matrix_with_kaiming_normal((3 * 32 * 32, 4000)))
    w2 = tf.Variable(create_matrix_with_kaiming_normal((4000, 10)))
    return [w1, w2]

learning_rate = 1e-2
train_part2(two_layer_fc, two_layer_fc_init, learning_rate)
```

```
Iteration 0, loss = 3.6422
Got 132 / 1000 correct (13.20%)
Iteration 100, loss = 1.8063
Got 363 / 1000 correct (36.30%)
Iteration 200, loss = 1.4058
Got 400 / 1000 correct (40.00%)
Iteration 300, loss = 1.7331
Got 358 / 1000 correct (35.80%)
Iteration 400, loss = 1.7765
Got 408 / 1000 correct (40.80%)
Iteration 500, loss = 1.8330
Got 431 / 1000 correct (43.10%)
Iteration 600, loss = 1.9183
Got 421 / 1000 correct (42.10%)
Iteration 700, loss = 1.9694
Got 458 / 1000 correct (45.80%)
```

5.0.9 Barebones TensorFlow: Train a three-layer ConvNet

We will now use TensorFlow to train a three-layer ConvNet on CIFAR-10.

You need to implement the `three_layer_convnet_init` function. Recall that the architecture of the network is:

1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding 2
2. ReLU
3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding 1
4. ReLU
5. Fully-connected layer (with bias) to compute scores for 10 classes

You don't need to do any hyperparameter tuning, but you should see validation accuracies above 43% after one epoch of training.

[24]:
```
def three_layer_convnet_init():
    """
    Initialize the weights of a Three-Layer ConvNet, for use with the
    three_layer_convnet function defined above.
    You can use the `create_matrix_with_kaiming_normal` helper!

    Inputs: None

    Returns a list containing:
    - conv_w1: TensorFlow tf.Variable giving weights for the first conv layer
    - conv_b1: TensorFlow tf.Variable giving biases for the first conv layer
    - conv_w2: TensorFlow tf.Variable giving weights for the second conv layer
    - conv_b2: TensorFlow tf.Variable giving biases for the second conv layer
    - fc_w: TensorFlow tf.Variable giving weights for the fully-connected layer
    - fc_b: TensorFlow tf.Variable giving biases for the fully-connected layer
    """
    params = None
    ############################################################################
    # TODO: Initialize the parameters of the three-layer network.              #
    ############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    conv_w1 = tf.Variable(create_matrix_with_kaiming_normal((5, 5, 3, 32)))
    conv_b1 = tf.Variable(tf.zeros(32))
    conv_w2 = tf.Variable(create_matrix_with_kaiming_normal((3, 3, 32, 16)))
    conv_b2 = tf.Variable(tf.zeros(16))
    fc_w = tf.Variable(create_matrix_with_kaiming_normal((32 * 32 * 16, 10)))
    fc_b = tf.Variable(tf.zeros(10))

    params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]

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

learning_rate = 3e-3
train_part2(three_layer_convnet, three_layer_convnet_init, learning_rate)
```

```
Iteration 0, loss = 3.3666
Got 93 / 1000 correct (9.30%)
Iteration 100, loss = 1.7840
Got 349 / 1000 correct (34.90%)
Iteration 200, loss = 1.5946
```

```
Got 391 / 1000 correct (39.10%)
Iteration 300, loss = 1.7177
Got 395 / 1000 correct (39.50%)
Iteration 400, loss = 1.7431
Got 411 / 1000 correct (41.10%)
Iteration 500, loss = 1.6669
Got 449 / 1000 correct (44.90%)
Iteration 600, loss = 1.7225
Got 457 / 1000 correct (45.70%)
Iteration 700, loss = 1.6917
Got 461 / 1000 correct (46.10%)
```

6 Part III: Keras Model Subclassing API

Implementing a neural network using the low-level TensorFlow API is a good way to understand how TensorFlow works, but it's a little inconvenient - we had to manually keep track of all Tensors holding learnable parameters. This was fine for a small network, but could quickly become unweildy for a large complex model.

Fortunately TensorFlow 2.0 provides higher-level APIs such as `tf.keras` which make it easy to build models out of modular, object-oriented layers. Further, TensorFlow 2.0 uses eager execution that evaluates operations immediately, without explicitly constructing any computational graphs. This makes it easy to write and debug models, and reduces the boilerplate code.

In this part of the notebook we will define neural network models using the `tf.keras.Model` API. To implement your own model, you need to do the following:

1. Define a new class which subclasses `tf.keras.Model`. Give your class an intuitive name that describes it, like `TwoLayerFC` or `ThreeLayerConvNet`.
2. In the initializer `__init__()` for your new class, define all the layers you need as class attributes. The `tf.keras.layers` package provides many common neural-network layers, like `tf.keras.layers.Dense` for fully-connected layers and `tf.keras.layers.Conv2D` for convolutional layers. Under the hood, these layers will construct `Variable` Tensors for any learnable parameters. **Warning**: Don't forget to call `super(YourModelName, self).__init__()` as the first line in your initializer!
3. Implement the `call()` method for your class; this implements the forward pass of your model, and defines the *connectivity* of your network. Layers defined in `__init__()` implement `__call__()` so they can be used as function objects that transform input Tensors into output Tensors. Don't define any new layers in `call()`; any layers you want to use in the forward pass should be defined in `__init__()`.

After you define your `tf.keras.Model` subclass, you can instantiate it and use it like the model functions from Part II.

6.0.1 Keras Model Subclassing API: Two-Layer Network

Here is a concrete example of using the `tf.keras.Model` API to define a two-layer network. There are a few new bits of API to be aware of here:

We use an `Initializer` object to set up the initial values of the learnable parameters

of the layers; in particular `tf.initializers.VarianceScaling` gives behavior similar to the Kaiming initialization method we used in Part II. You can read more about it here: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/initializers/VarianceScaling

We construct `tf.keras.layers.Dense` objects to represent the two fully-connected layers of the model. In addition to multiplying their input by a weight matrix and adding a bias vector, these layer can also apply a nonlinearity for you. For the first layer we specify a ReLU activation function by passing `activation='relu'` to the constructor; the second layer uses softmax activation function. Finally, we use `tf.keras.layers.Flatten` to flatten the output from the previous fully-connected layer.

[25]:
```
class TwoLayerFC(tf.keras.Model):
    def __init__(self, hidden_size, num_classes):
        super(TwoLayerFC, self).__init__()
        initializer = tf.initializers.VarianceScaling(scale=2.0)
        self.fc1 = tf.keras.layers.Dense(hidden_size, activation='relu',
                                         kernel_initializer=initializer)
        self.fc2 = tf.keras.layers.Dense(num_classes, activation='softmax',
                                         kernel_initializer=initializer)
        self.flatten = tf.keras.layers.Flatten()

    def call(self, x, training=False):
        x = self.flatten(x)
        x = self.fc1(x)
        x = self.fc2(x)
        return x


def test_TwoLayerFC():
    """ A small unit test to exercise the TwoLayerFC model above. """
    input_size, hidden_size, num_classes = 50, 42, 10
    x = tf.zeros((64, input_size))
    model = TwoLayerFC(hidden_size, num_classes)
    with tf.device(device):
        scores = model(x)
        print(scores.shape)

test_TwoLayerFC()
```

```
(64, 10)
```

6.0.2 Keras Model Subclassing API: Three-Layer ConvNet

Now it's your turn to implement a three-layer ConvNet using the `tf.keras.Model` API. Your model should have the same architecture used in Part II:

1. Convolutional layer with 5 x 5 kernels, with zero-padding of 2
2. ReLU nonlinearity
3. Convolutional layer with 3 x 3 kernels, with zero-padding of 1

4. ReLU nonlinearity
5. Fully-connected layer to give class scores
6. Softmax nonlinearity

You should initialize the weights of your network using the same initialization method as was used in the two-layer network above.

Hint: Refer to the documentation for `tf.keras.layers.Conv2D` and `tf.keras.layers.Dense`:

https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Conv2D

https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Dense

[26]:
```
class ThreeLayerConvNet(tf.keras.Model):
    def __init__(self, channel_1, channel_2, num_classes):
        super(ThreeLayerConvNet, self).__init__()
        ########################################################################
        # TODO: Implement the __init__ method for a three-layer ConvNet. You   #
        # should instantiate layer objects to be used in the forward pass.     #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        initializer = tf.initializers.VarianceScaling(scale=2.0)
        self.conv1 = tf.keras.layers.Conv2D(channel_1, 5, padding='same',␣
↪activation='relu', kernel_initializer=initializer)
        self.conv2 = tf.keras.layers.Conv2D(channel_2, 3, padding='same',␣
↪activation='relu', kernel_initializer=initializer)
        self.fc = tf.keras.layers.Dense(num_classes, activation='softmax',␣
↪kernel_initializer=initializer)
        self.flatten = tf.keras.layers.Flatten()

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

    def call(self, x, training=False):
        scores = None
        ########################################################################
        # TODO: Implement the forward pass for a three-layer ConvNet. You      #
        # should use the layer objects defined in the __init__ method.         #
        ########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        scores = self.fc(self.flatten(self.conv2(self.conv1(x))))

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

```
    ␣
↪############################################################################ ␣
↪
        return scores
```

Once you complete the implementation of the `ThreeLayerConvNet` above you can run the following to ensure that your implementation does not crash and produces outputs of the expected shape.

[27]:
```
def test_ThreeLayerConvNet():
    channel_1, channel_2, num_classes = 12, 8, 10
    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)
    with tf.device(device):
        x = tf.zeros((64, 3, 32, 32))
        scores = model(x)
        print(scores.shape)

test_ThreeLayerConvNet()
```

```
(64, 10)
```

6.0.3 Keras Model Subclassing API: Eager Training

While keras models have a builtin training loop (using the `model.fit`), sometimes you need more customization. Here's an example, of a training loop implemented with eager execution.

In particular, notice `tf.GradientTape`. Automatic differentiation is used in the backend for implementing backpropagation in frameworks like TensorFlow. During eager execution, `tf.GradientTape` is used to trace operations for computing gradients later. A particular `tf.GradientTape` can only compute one gradient; subsequent calls to tape will throw a runtime error.

TensorFlow 2.0 ships with easy-to-use built-in metrics under `tf.keras.metrics` module. Each metric is an object, and we can use `update_state()` to add observations and `reset_state()` to clear all observations. We can get the current result of a metric by calling `result()` on the metric object.

[11]:
```
def train_part34(model_init_fn, optimizer_init_fn, num_epochs=1,␣
↪is_training=False):
    """
    Simple training loop for use with models defined using tf.keras. It trains
    a model for one epoch on the CIFAR-10 training set and periodically checks
    accuracy on the CIFAR-10 validation set.

    Inputs:
    - model_init_fn: A function that takes no parameters; when called it
      constructs the model we want to train: model = model_init_fn()
    - optimizer_init_fn: A function which takes no parameters; when called it
      constructs the Optimizer object we will use to optimize the model:
      optimizer = optimizer_init_fn()
```

```
    - num_epochs: The number of epochs to train for

    Returns: Nothing, but prints progress during trainingn
    """
    with tf.device(device):

        # Compute the loss like we did in Part II
        loss_fn = tf.keras.losses.SparseCategoricalCrossentropy()

        model = model_init_fn()
        optimizer = optimizer_init_fn()

        train_loss = tf.keras.metrics.Mean(name='train_loss')
        train_accuracy = tf.keras.metrics.
↪SparseCategoricalAccuracy(name='train_accuracy')

        val_loss = tf.keras.metrics.Mean(name='val_loss')
        val_accuracy = tf.keras.metrics.
↪SparseCategoricalAccuracy(name='val_accuracy')

        t = 0
        for epoch in range(num_epochs):

            # Reset the metrics - https://www.tensorflow.org/alpha/guide/
↪migration_guide#new-style_metrics
            train_loss.reset_states()
            train_accuracy.reset_states()

            for x_np, y_np in train_dset:
                with tf.GradientTape() as tape:

                    # Use the model function to build the forward pass.
                    scores = model(x_np, training=is_training)
                    loss = loss_fn(y_np, scores)

                    gradients = tape.gradient(loss, model.trainable_variables)
                    optimizer.apply_gradients(zip(gradients, model.
↪trainable_variables))

                    # Update the metrics
                    train_loss.update_state(loss)
                    train_accuracy.update_state(y_np, scores)

                    if t % print_every == 0:
                        val_loss.reset_states()
                        val_accuracy.reset_states()
                        for test_x, test_y in val_dset:
```

```
                            # During validation at end of epoch, training set␣
↪to False
                            prediction = model(test_x, training=False)
                            t_loss = loss_fn(test_y, prediction)

                            val_loss.update_state(t_loss)
                            val_accuracy.update_state(test_y, prediction)

                        template = 'Iteration {}, Epoch {}, Loss: {}, Accuracy:␣
↪{}, Val Loss: {}, Val Accuracy: {}'
                        print (template.format(t, epoch+1,
                                               train_loss.result(),
                                               train_accuracy.result()*100,
                                               val_loss.result(),
                                               val_accuracy.result()*100))
                    t += 1
```

6.0.4 Keras Model Subclassing API: Train a Two-Layer Network

We can now use the tools defined above to train a two-layer network on CIFAR-10. We define the `model_init_fn` and `optimizer_init_fn` that construct the model and optimizer respectively when called. Here we want to train the model using stochastic gradient descent with no momentum, so we construct a `tf.keras.optimizers.SGD` function; you can read about it here.

You don't need to tune any hyperparameters here, but you should achieve validation accuracies above 40% after one epoch of training.

[29]:
```
hidden_size, num_classes = 4000, 10
learning_rate = 1e-2

def model_init_fn():
    return TwoLayerFC(hidden_size, num_classes)

def optimizer_init_fn():
    return tf.keras.optimizers.SGD(learning_rate=learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 2.6609761714935303, Accuracy: 9.375, Val Loss:
2.8042850494384766, Val Accuracy: 12.5
Iteration 100, Epoch 1, Loss: 2.216717481613159, Accuracy: 28.97586441040039,
Val Loss: 1.9253003597259521, Val Accuracy: 37.79999923706055
Iteration 200, Epoch 1, Loss: 2.059610605239868, Accuracy: 32.8125, Val Loss:
1.8415443897247314, Val Accuracy: 40.5
Iteration 300, Epoch 1, Loss: 1.9900760650634766, Accuracy: 34.50996398925781,
Val Loss: 1.9108893871307373, Val Accuracy: 36.19999694824219
Iteration 400, Epoch 1, Loss: 1.924764633178711, Accuracy: 36.33494567871094,
Val Loss: 1.7669795751571655, Val Accuracy: 39.099998474121094
```

```
Iteration 500, Epoch 1, Loss: 1.8809505701065063, Accuracy: 37.384605407714844,
Val Loss: 1.691422700881958, Val Accuracy: 41.5
Iteration 600, Epoch 1, Loss: 1.8505853414535522, Accuracy: 38.287750244140625,
Val Loss: 1.7025686502456665, Val Accuracy: 41.60000228881836
Iteration 700, Epoch 1, Loss: 1.8255977630615234, Accuracy: 38.91984558105469,
Val Loss: 1.6738944053649902, Val Accuracy: 42.599998474121094
```

6.0.5 Keras Model Subclassing API: Train a Three-Layer ConvNet

Here you should use the tools we've defined above to train a three-layer ConvNet on CIFAR-10. Your ConvNet should use 32 filters in the first convolutional layer and 16 filters in the second layer.

To train the model you should use gradient descent with Nesterov momentum 0.9.

HINT: https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/optimizers/SGD

You don't need to perform any hyperparameter tuning, but you should achieve validation accuracies above 50% after training for one epoch.

[30]:
```
learning_rate = 3e-3
channel_1, channel_2, num_classes = 32, 16, 10

def model_init_fn():
    model = None
    ################################################################################
    # TODO: Complete the implementation of model_fn.                               #
    ################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    model = ThreeLayerConvNet(channel_1, channel_2, num_classes)

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

def optimizer_init_fn():
    optimizer = None
    ################################################################################
    # TODO: Complete the implementation of model_fn.                               #
    ################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate, momentum=0.
    ↪9, nesterov=True)

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

```
    #                               END OF YOUR CODE                              #
    ############################################################################
    return optimizer

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 2.9922893047332764, Accuracy: 10.9375, Val Loss:
6.65677547454834, Val Accuracy: 7.90000057220459
Iteration 100, Epoch 1, Loss: 2.173110246658325, Accuracy: 26.670793533325195,
Val Loss: 1.7641805410385132, Val Accuracy: 37.79999923706055
Iteration 200, Epoch 1, Loss: 1.915182113647461, Accuracy: 34.406097412109375,
Val Loss: 1.5217888355255127, Val Accuracy: 47.29999923706055
Iteration 300, Epoch 1, Loss: 1.7899959087371826, Accuracy: 38.024295806884766,
Val Loss: 1.4604946374893188, Val Accuracy: 50.0
Iteration 400, Epoch 1, Loss: 1.6932916641235352, Accuracy: 41.22116470336914,
Val Loss: 1.4003156423568726, Val Accuracy: 50.400001525878906
Iteration 500, Epoch 1, Loss: 1.6279816627502441, Accuracy: 43.2728271484375,
Val Loss: 1.3361924886703491, Val Accuracy: 52.39999771118164
Iteration 600, Epoch 1, Loss: 1.5825644731521606, Accuracy: 44.65734100341797,
Val Loss: 1.2787823677062988, Val Accuracy: 54.599998474121094
Iteration 700, Epoch 1, Loss: 1.5452487468719482, Accuracy: 45.883113861083984,
Val Loss: 1.2654927968978882, Val Accuracy: 54.400001525878906
```

7 Part IV: Keras Sequential API

In Part III we introduced the `tf.keras.Model` API, which allows you to define models with any number of learnable layers and with arbitrary connectivity between layers.

However for many models you don't need such flexibility - a lot of models can be expressed as a sequential stack of layers, with the output of each layer fed to the next layer as input. If your model fits this pattern, then there is an even easier way to define your model: using `tf.keras.Sequential`. You don't need to write any custom classes; you simply call the `tf.keras.Sequential` constructor with a list containing a sequence of layer objects.

One complication with `tf.keras.Sequential` is that you must define the shape of the input to the model by passing a value to the `input_shape` of the first layer in your model.

7.0.1 Keras Sequential API: Two-Layer Network

In this subsection, we will rewrite the two-layer fully-connected network using `tf.keras.Sequential`, and train it using the training loop defined above.

You don't need to perform any hyperparameter tuning here, but you should see validation accuracies above 40% after training for one epoch.

[31]:
```
learning_rate = 1e-2

def model_init_fn():
    input_shape = (32, 32, 3)
```

```
    hidden_layer_size, num_classes = 4000, 10
    initializer = tf.initializers.VarianceScaling(scale=2.0)
    layers = [
        tf.keras.layers.Flatten(input_shape=input_shape),
        tf.keras.layers.Dense(hidden_layer_size, activation='relu',
                              kernel_initializer=initializer),
        tf.keras.layers.Dense(num_classes, activation='softmax',
                              kernel_initializer=initializer),
    ]
    model = tf.keras.Sequential(layers)
    return model

def optimizer_init_fn():
    return tf.keras.optimizers.SGD(learning_rate=learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 3.060246467590332, Accuracy: 10.9375, Val Loss:
2.8321943283081055, Val Accuracy: 13.300000190734863
Iteration 100, Epoch 1, Loss: 2.2500174045562744, Accuracy: 28.7592830657959,
Val Loss: 1.838533878326416, Val Accuracy: 39.89999771118164
Iteration 200, Epoch 1, Loss: 2.0877583026885986, Accuracy: 32.45491409301758,
Val Loss: 1.7922143936157227, Val Accuracy: 39.89999771118164
Iteration 300, Epoch 1, Loss: 2.0124216079711914, Accuracy: 33.959716796875, Val
Loss: 1.8211208581924438, Val Accuracy: 37.599998474121094
Iteration 400, Epoch 1, Loss: 1.9414657354354858, Accuracy: 35.85956954956055,
Val Loss: 1.693195104598999, Val Accuracy: 43.39999771118164
Iteration 500, Epoch 1, Loss: 1.8957719802856445, Accuracy: 36.93550491333008,
Val Loss: 1.6461632251739502, Val Accuracy: 43.79999923706055
Iteration 600, Epoch 1, Loss: 1.8634772300720215, Accuracy: 37.88217544555664,
Val Loss: 1.6569775342941284, Val Accuracy: 43.39999771118164
Iteration 700, Epoch 1, Loss: 1.8376551866531372, Accuracy: 38.5208625793457,
Val Loss: 1.597198486328125, Val Accuracy: 44.900001525878906
```

7.0.2 Abstracting Away the Training Loop

In the previous examples, we used a customised training loop to train models (e.g. `train_part34`). Writing your own training loop is only required if you need more flexibility and control during training your model. Alternately, you can also use built-in APIs like `tf.keras.Model.fit()` and `tf.keras.Model.evaluate` to train and evaluate a model. Also remember to configure your model for training by calling 'tf.keras.Model.compile.

You don't need to perform any hyperparameter tuning here, but you should see validation and test accuracies above 42% after training for one epoch.

[34]:
```
model = model_init_fn()
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=learning_rate),
              loss='sparse_categorical_crossentropy',
```

```
                metrics=[tf.keras.metrics.sparse_categorical_accuracy])
model.fit(X_train, y_train, batch_size=64, epochs=1, validation_data=(X_val,␣
 ↪y_val))
model.evaluate(X_test, y_test)
```

```
766/766 [==============================] - 48s 62ms/step - loss: 1.8184 -
sparse_categorical_accuracy: 0.3899 - val_loss: 1.6502 -
val_sparse_categorical_accuracy: 0.4270
313/313 [==============================] - 6s 18ms/step - loss: 1.6070 -
sparse_categorical_accuracy: 0.4326
```

[34]:
```
[1.6069886684417725, 0.4325999915599823]
```

7.0.3 Keras Sequential API: Three-Layer ConvNet

Here you should use `tf.keras.Sequential` to reimplement the same three-layer ConvNet architecture used in Part II and Part III. As a reminder, your model should have the following architecture:

1. Convolutional layer with 32 5x5 kernels, using zero padding of 2
2. ReLU nonlinearity
3. Convolutional layer with 16 3x3 kernels, using zero padding of 1
4. ReLU nonlinearity
5. Fully-connected layer giving class scores
6. Softmax nonlinearity

You should initialize the weights of the model using a `tf.initializers.VarianceScaling` as above.

You should train the model using Nesterov momentum 0.9.

You don't need to perform any hyperparameter search, but you should achieve accuracy above 45% after training for one epoch.

[35]:
```
def model_init_fn():
    model = None
    ##############################################################################
    # TODO: Construct a three-layer ConvNet using tf.keras.Sequential.           #
    ##############################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    initializer = tf.initializers.VarianceScaling(scale=2.0)
    model = tf.keras.Sequential([
        tf.keras.layers.Conv2D(channel_1, 5, input_shape=(32, 32, 3),␣
 ↪padding='same', activation='relu', kernel_initializer=initializer),
        tf.keras.layers.Conv2D(channel_2, 3, padding='same', activation='relu',␣
 ↪kernel_initializer=initializer),
        tf.keras.layers.Flatten(),
        tf.keras.layers.Dense(num_classes, activation='softmax',␣
 ↪kernel_initializer=initializer)
```

```
    ])

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

learning_rate = 5e-4
def optimizer_init_fn():
    optimizer = None
    ################################################################################
    # TODO: Complete the implementation of model_fn.                               #
    ################################################################################
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    optimizer = tf.keras.optimizers.SGD(learning_rate=learning_rate, momentum=0.
 ↪9, nesterov=True)

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

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 2.826967239379883, Accuracy: 10.9375, Val Loss:
2.590606212615967, Val Accuracy: 10.0
Iteration 100, Epoch 1, Loss: 2.0122900009155273, Accuracy: 27.150371551513672,
Val Loss: 1.8256518840789795, Val Accuracy: 36.29999923706055
Iteration 200, Epoch 1, Loss: 1.8841396570205688, Accuracy: 32.501556396484375,
Val Loss: 1.7027852535247803, Val Accuracy: 40.900001525878906
Iteration 300, Epoch 1, Loss: 1.8150813579559326, Accuracy: 35.558555603027344,
Val Loss: 1.64588463306427, Val Accuracy: 42.29999923706055
Iteration 400, Epoch 1, Loss: 1.7569448947906494, Accuracy: 37.881858825683594,
Val Loss: 1.5895328521728516, Val Accuracy: 45.29999923706055
Iteration 500, Epoch 1, Loss: 1.714806318283081, Accuracy: 39.28081512451172,
Val Loss: 1.5428310632705688, Val Accuracy: 44.60000228881836
Iteration 600, Epoch 1, Loss: 1.6873254776000977, Accuracy: 40.33121871948242,
Val Loss: 1.5165183544158936, Val Accuracy: 47.099998474121094
Iteration 700, Epoch 1, Loss: 1.661790132522583, Accuracy: 41.28477096557617,
Val Loss: 1.4688769578933716, Val Accuracy: 47.5
```

We will also train this model with the built-in training loop APIs provided by TensorFlow.

[36]:
```
model = model_init_fn()
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=[tf.keras.metrics.sparse_categorical_accuracy])
model.fit(X_train, y_train, batch_size=64, epochs=1, validation_data=(X_val,␣
 ↪y_val))
model.evaluate(X_test, y_test)
```

```
766/766 [==============================] - 117s 153ms/step - loss: 1.6081 -
sparse_categorical_accuracy: 0.4391 - val_loss: 1.4005 -
val_sparse_categorical_accuracy: 0.5010
313/313 [==============================] - 6s 20ms/step - loss: 1.4170 -
sparse_categorical_accuracy: 0.4993
```

[36]:
```
[1.416999101638794, 0.4993000030517578]
```

7.1 Part IV: Functional API

7.1.1 Demonstration with a Two-Layer Network

In the previous section, we saw how we can use `tf.keras.Sequential` to stack layers to quickly build simple models. But this comes at the cost of losing flexibility.

Often we will have to write complex models that have non-sequential data flows: a layer can have **multiple inputs and/or outputs**, such as stacking the output of 2 previous layers together to feed as input to a third! (Some examples are residual connections and dense blocks.)

In such cases, we can use Keras functional API to write models with complex topologies such as:

1. Multi-input models
2. Multi-output models
3. Models with shared layers (the same layer called several times)
4. Models with non-sequential data flows (e.g. residual connections)

Writing a model with Functional API requires us to create a `tf.keras.Model` instance and explicitly write input tensors and output tensors for this model.

[37]:
```
def two_layer_fc_functional(input_shape, hidden_size, num_classes):
    initializer = tf.initializers.VarianceScaling(scale=2.0)
    inputs = tf.keras.Input(shape=input_shape)
    flattened_inputs = tf.keras.layers.Flatten()(inputs)
    fc1_output = tf.keras.layers.Dense(hidden_size, activation='relu',
                                       ␣
 ↪kernel_initializer=initializer)(flattened_inputs)
    scores = tf.keras.layers.Dense(num_classes, activation='softmax',
                                   kernel_initializer=initializer)(fc1_output)

    # Instantiate the model given inputs and outputs.
    model = tf.keras.Model(inputs=inputs, outputs=scores)
    return model
```

```
def test_two_layer_fc_functional():
    """ A small unit test to exercise the TwoLayerFC model above. """
    input_size, hidden_size, num_classes = 50, 42, 10
    input_shape = (50,)

    x = tf.zeros((64, input_size))
    model = two_layer_fc_functional(input_shape, hidden_size, num_classes)

    with tf.device(device):
        scores = model(x)
        print(scores.shape)

test_two_layer_fc_functional()
```

```
(64, 10)
```

7.1.2 Keras Functional API: Train a Two-Layer Network

You can now train this two-layer network constructed using the functional API.

You don't need to perform any hyperparameter tuning here, but you should see validation accuracies above 40% after training for one epoch.

[38]:
```
input_shape = (32, 32, 3)
hidden_size, num_classes = 4000, 10
learning_rate = 1e-2

def model_init_fn():
    return two_layer_fc_functional(input_shape, hidden_size, num_classes)

def optimizer_init_fn():
    return tf.keras.optimizers.SGD(learning_rate=learning_rate)

train_part34(model_init_fn, optimizer_init_fn)
```

```
Iteration 0, Epoch 1, Loss: 3.112236261367798, Accuracy: 7.8125, Val Loss:
2.903177261352539, Val Accuracy: 14.100000381469727
Iteration 100, Epoch 1, Loss: 2.216944694519043, Accuracy: 29.223390579223633,
Val Loss: 1.8776451349258423, Val Accuracy: 40.79999923706055
Iteration 200, Epoch 1, Loss: 2.0730960369110107, Accuracy: 32.532649993896484,
Val Loss: 1.868657112121582, Val Accuracy: 40.099998474121094
Iteration 300, Epoch 1, Loss: 1.9974868297576904, Accuracy: 34.369808197021484,
Val Loss: 1.8381285667419434, Val Accuracy: 38.80000305175781
Iteration 400, Epoch 1, Loss: 1.9319634437561035, Accuracy: 35.925811767578125,
Val Loss: 1.7609485387802124, Val Accuracy: 41.10000228881836
Iteration 500, Epoch 1, Loss: 1.888472080230713, Accuracy: 37.06337356567383,
Val Loss: 1.6645240783691406, Val Accuracy: 43.0
Iteration 600, Epoch 1, Loss: 1.8600658178329468, Accuracy: 37.80937957763672,
```

```
Val Loss: 1.6867358684539795, Val Accuracy: 41.5
Iteration 700, Epoch 1, Loss: 1.8349647521972656, Accuracy: 38.48965835571289,
Val Loss: 1.6277226209640503, Val Accuracy: 44.400001525878906
```

8 Part V: CIFAR-10 open-ended challenge

In this section you can experiment with whatever ConvNet architecture you'd like on CIFAR-10.

You should experiment with architectures, hyperparameters, loss functions, regularization, or anything else you can think of to train a model that achieves **at least 70%** accuracy on the **validation** set within 10 epochs. You can use the built-in train function, the `train_part34` function from above, or implement your own training loop.

Describe what you did at the end of the notebook.

8.0.1 Some things you can try:

- **Filter size**: Above we used 5x5 and 3x3; is this optimal?
- **Number of filters**: Above we used 16 and 32 filters. Would more or fewer do better?
- **Pooling**: We didn't use any pooling above. Would this improve the model?
- **Normalization**: Would your model be improved with batch normalization, layer normalization, group normalization, or some other normalization strategy?
- **Network architecture**: The ConvNet above has only three layers of trainable parameters. Would a deeper model do better?
- **Global average pooling**: Instead of flattening after the final convolutional layer, would global average pooling do better? This strategy is used for example in Google's Inception network and in Residual Networks.
- **Regularization**: Would some kind of regularization improve performance? Maybe weight decay or dropout?

8.0.2 NOTE: Batch Normalization / Dropout

If you are using Batch Normalization and Dropout, remember to pass `is_training=True` if you use the `train_part34()` function. BatchNorm and Dropout layers have different behaviors at training and inference time. `training` is a specific keyword argument reserved for this purpose in any `tf.keras.Model`'s `call()` function. Read more about this here : https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/BatchNormalization#methods
https://www.tensorflow.org/versions/r2.0/api_docs/python/tf/keras/layers/Dropout#methods

8.0.3 Tips for training

For each network architecture that you try, you should tune the learning rate and other hyperparameters. When doing this there are a couple important things to keep in mind:

- If the parameters are working well, you should see improvement within a few hundred iterations
- Remember the coarse-to-fine approach for hyperparameter tuning: start by testing a large range of hyperparameters for just a few training iterations to find the combinations of parameters that are working at all.

- Once you have found some sets of parameters that seem to work, search more finely around these parameters. You may need to train for more epochs.
- You should use the validation set for hyperparameter search, and save your test set for evaluating your architecture on the best parameters as selected by the validation set.

8.0.4 Going above and beyond

If you are feeling adventurous there are many other features you can implement to try and improve your performance. You are **not required** to implement any of these, but don't miss the fun if you have time!

- Alternative optimizers: you can try Adam, Adagrad, RMSprop, etc.
- Alternative activation functions such as leaky ReLU, parametric ReLU, ELU, or MaxOut.
- Model ensembles
- Data augmentation
- New Architectures
  - ResNets where the input from the previous layer is added to the output.
  - DenseNets where inputs into previous layers are concatenated together.
  - This blog has an in-depth overview

8.0.5 Have fun and happy training!

[31]:
```
# ResNet approach based on paper https://arxiv.org/abs/1512.03385
# Implementation details: https://www.analyticsvidhya.com/blog/2021/08/
 ↪how-to-code-your-resnet-from-scratch-in-tensorflow%E2%80%8A/

class _IdentityBlock(tf.keras.Model):
    """Identity block utilizing skip connections."""

    def __init__(self, out_channels):
        super().__init__()
        """Initializes the identiy block.

        Here we simply initialize 2 layers which process the input and after␣
 ↪the output
        is produces it is added together with the input whcih is the final␣
 ↪output.

        Args:
            out_channels (int): The number of activation maps this block should␣
 ↪produce
        """
        # Acts as Kaiming weight initalization
        initializer = tf.initializers.VarianceScaling(scale=2.0)

        # Part 1 of the convolution, normalization and non-linearity
        self.conv1 = tf.keras.layers.Conv2D(out_channels, 3, padding='same',␣
 ↪use_bias=False, kernel_initializer=initializer)
```

```
        self.norm1 = tf.keras.layers.BatchNormalization(axis=3)
        self.relu1 = tf.keras.layers.Activation('relu')

        # Part 2 of the convolution, normalization and non-linearity
        self.conv2 = tf.keras.layers.Conv2D(out_channels, 3, padding='same',␣
↪use_bias=False, kernel_initializer=initializer)
        self.norm2 = tf.keras.layers.BatchNormalization(axis=3)
        self.relu2 = tf.keras.layers.Activation('relu')

        # Add layer will add together the input and the output
        self.add = tf.keras.layers.Add()

    def call(self, x, training=False):
        """Performs forward pass on the given input.

        Args:
            x (Tensor):      The input of dimensions (N, H, W, C)
            training (bool): Indicates whether the forward pass happens in the␣
↪training mode

        Returns:
              out (Tensor): Output data of dim (N, H, W, C)
        """
        x_skip = tf.identity(x)                    # prepare to add the input to␣
↪the output
        x = self.relu1(self.norm1(self.conv1(x))) # pass input through the␣
↪first layer
        x = self.norm2(self.conv2(x))              # pass input through the␣
↪second layer (without ReLU)
        out = self.relu2(self.add([x, x_skip]))    # perform ReLU on the␣
↪processed input added with the raw input

        return out


class _BottleneckBlock(tf.keras.Model):
    """Same as identity block except it reduces the spacial area before␣
↪processing the input."""

    def __init__(self, out_channels):
        """Initializes the bottleneck block.

        Unlike *_IdentityBlock*, the first convolution here reduces the spacial␣
↪size of the input
        by a factor of `2`. Then, it performs the main convolution after which␣
↪the output maps
```

```
        are added together with the input maps to produce final activations.

        Args:
            out_channels (int): The number of activation maps this block should␣
↪produce
        """
        super().__init__()

        # Acts as Kaiming weight initalization
        initializer = tf.initializers.VarianceScaling(scale=2.0)

        # Reduce the input size by 2 to match output size
        self.skip1 = tf.keras.layers.Conv2D(out_channels, 2, strides=2,␣
↪use_bias=False, kernel_initializer=initializer)

        # Part 1 of the convolution which reduces the spacial area
        self.conv1 = tf.keras.layers.Conv2D(out_channels, 3, strides=2,␣
↪padding='same', use_bias=False, kernel_initializer=initializer)
        self.norm1 = tf.keras.layers.BatchNormalization(axis=3)
        self.relu1 = tf.keras.layers.Activation('relu')

        # Part 2 of the convolution which extracts features from the reduced␣
↪input
        self.conv2 = tf.keras.layers.Conv2D(out_channels, 3, padding='same',␣
↪use_bias=False, kernel_initializer=initializer)
        self.norm2 = tf.keras.layers.BatchNormalization(axis=3)
        self.relu2 = tf.keras.layers.Activation('relu')

        # Add layer will add together the input and the output
        self.add = tf.keras.layers.Add()

    def call(self, x, training=False):
        """Performs forward pass on the given input.

        Args:
            x (Tensor):      The input of dimensions (N, H, W, C)
            training (bool): Indicates whether the forward pass happens in the␣
↪training mode

        Returns:
            out (Tensor): Output data of dim (N, H/2, W/2, out_channels)
        """
        x_skip = self.skip1(x)                    # prepare to add the input to␣
↪the output
        x = self.relu1(self.norm1(self.conv1(x))) # pass input through the␣
↪first layer
```

```
        x = self.norm2(self.conv2(x))              # pass input through the␣
↪second layer (without ReLU)
        out = self.relu2(self.add([x, x_skip]))    # perform ReLU on the␣
↪processed input added with the raw input

        return out

class ResNet(tf.keras.Model):
    """Residual network

    The network has the following architecture:
        1. `CONV->NORM->RELU->CONV->NORM->RELU` to preprocess the input for a␣
↪chain of layer blocks
        2. `IDENTITY->[BOTTLENECK->IDENTITY] x N` where each block consists of␣
↪arbitrary
           number of layers which use 'skip' connections
        3. `POOL->DENSE->SOFTMAX` where global average pooling is performed
           before calculating raw scores
    """

    def __init__(self, in_channels=32, block_config=(2, 2, 2, 2),␣
↪num_classes=10):
        """Initializes the residual network.

        The first layer produces `in_channels` activation maps which are then␣
↪fed to a
        sequence of blocks containing a specified number of identity sub-blocks␣
↪(first
        block is always *_BottleneckBlock*). At the end the _global average␣
↪pooling_
        layer is used to flatten the activations for the linear softmax␣
↪classifier.

        Args:
            in_channels (int):    The number of channels to extract after the␣
↪first convolution
            block_config (tuple): The number of layers each bloack should have␣
↪in sequence
            num_classes (int):    The total number of classes
        """
        super().__init__()

        # Acts as Kaiming weight initalization
        initializer = tf.initializers.VarianceScaling(scale=2.0)

        # Prepare the input for the chains of identity blocks
```

```
        self.features = tf.keras.Sequential([
            tf.keras.layers.Conv2D(in_channels, 5, padding='same',␣
↪use_bias=False, kernel_initializer=initializer),
            tf.keras.layers.BatchNormalization(axis=3),
            tf.keras.layers.Activation('relu'),
        ])

        num_features = in_channels # num feaure maps to produce after each␣
↪group of identity blocks

        # Loop through every group of blocks
        for i, num_layers in enumerate(block_config):
            # Use bottleneck block as the first block in every group (except␣
↪first)
            if i != 0:
                self.features.add(_BottleneckBlock(num_features))
            else:
                self.features.add(_IdentityBlock(num_features))

            # Create the specified number of identity blocks for i'th group
            for j in range(num_layers-1):
                self.features.add(_IdentityBlock(num_features))

            num_features *= 2 # increase the nuber of features to be produced

        # Flatten the final activation maps using global average pooling
        self.features.add(tf.keras.layers.GlobalAveragePooling2D())

        # Softmax classifier is used as the final layer
        self.classifier = tf.keras.layers.Dense(num_classes,␣
↪activation='softmax', kernel_initializer=initializer)

    def call(self, x, training=False):
        """Performs forward pass on the given input.

        Args:
            x (Tensor):      The input of dimensions (N, H, W, C)
            training (bool): Indicates whether the forward pass happens in the␣
↪training mode

        Returns:
            out (Tensor): Output data of dim (N, 10)
        """
        out = self.features(x)     # Get the extracted features for the linear␣
↪classfier
```

```
        out = self.classifier(out) # Perform classification with softmax␣
↪activation

        return out
```

[32]:
```
class CustomConvNet(tf.keras.Model):
    def __init__(self):
        super(CustomConvNet, self).__init__()
        ␣
↪#########################################################################
        # TODO: Construct a model that performs well on CIFAR-10                ␣
↪    #
        ␣
↪#########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        self.model = ResNet()

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

    def call(self, input_tensor, training=False):
        ␣
↪#########################################################################
        # TODO: Construct a model that performs well on CIFAR-10                ␣
↪    #
        ␣
↪#########################################################################
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        x = self.model.call(input_tensor, training)

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

```
print_every = 100
num_epochs = 10

model = CustomConvNet()

def model_init_fn():
    return CustomConvNet()

def optimizer_init_fn():
    learning_rate = 1e-3
    return tf.keras.optimizers.Adam(learning_rate)

train_part34(model_init_fn, optimizer_init_fn, num_epochs=num_epochs,␣
 ↪is_training=True)
```

```
Iteration 0, Epoch 1, Loss: 5.634993553161621, Accuracy: 6.25, Val Loss:
35.548118591308594, Val Accuracy: 11.200000762939453
Iteration 100, Epoch 1, Loss: 1.8992418050765991, Accuracy: 34.79269790649414,
Val Loss: 5.219505786895752, Val Accuracy: 18.299999237060547
Iteration 200, Epoch 1, Loss: 1.695570468902588, Accuracy: 40.003108978271484,
Val Loss: 1.5958833694458008, Val Accuracy: 41.60000228881836
Iteration 300, Epoch 1, Loss: 1.5840414762496948, Accuracy: 43.687705993652344,
Val Loss: 1.6454240083694458, Val Accuracy: 42.89999771118164
Iteration 400, Epoch 1, Loss: 1.4977669715881348, Accuracy: 46.450279235839844,
Val Loss: 1.456979513168335, Val Accuracy: 49.599998474121094
Iteration 500, Epoch 1, Loss: 1.4332406520843506, Accuracy: 48.67140579223633,
Val Loss: 1.291126012802124, Val Accuracy: 56.0
Iteration 600, Epoch 1, Loss: 1.3800538778305054, Accuracy: 50.582359313964844,
Val Loss: 1.2686375379562378, Val Accuracy: 57.400001525878906
Iteration 700, Epoch 1, Loss: 1.335140347480774, Accuracy: 52.242332458496094,
Val Loss: 1.155949354171753, Val Accuracy: 60.20000076293945
Iteration 800, Epoch 2, Loss: 0.9661270380020142, Accuracy: 66.5625, Val Loss:
1.143396258354187, Val Accuracy: 63.30000305175781
Iteration 900, Epoch 2, Loss: 0.9268084764480591, Accuracy: 67.24536895751953,
Val Loss: 1.2269514799118042, Val Accuracy: 58.0
Iteration 1000, Epoch 2, Loss: 0.9152790904045105, Accuracy: 67.55319213867188,
Val Loss: 1.2005499601364136, Val Accuracy: 60.5
Iteration 1100, Epoch 2, Loss: 0.8981556296348572, Accuracy: 68.03638458251953,
Val Loss: 1.2360095977783203, Val Accuracy: 59.60000228881836
Iteration 1200, Epoch 2, Loss: 0.8798803091049194, Accuracy: 68.69612121582031,
Val Loss: 0.8850542902946472, Val Accuracy: 71.10000610351562
Iteration 1300, Epoch 2, Loss: 0.8638664484024048, Accuracy: 69.35163879394531,
Val Loss: 0.9424600601196289, Val Accuracy: 68.5999984741211
Iteration 1400, Epoch 2, Loss: 0.8460219502449036, Accuracy: 69.92617797851562,
Val Loss: 1.096124529838562, Val Accuracy: 65.4000015258789
Iteration 1500, Epoch 2, Loss: 0.8271858096122742, Accuracy: 70.70365905761719,
```

Val Loss: 0.841564953327179, Val Accuracy: 72.0999984741211
Iteration 1600, Epoch 3, Loss: 0.6442562341690063, Accuracy: 77.42300415039062,
Val Loss: 0.8527148962020874, Val Accuracy: 69.9000015258789
Iteration 1700, Epoch 3, Loss: 0.6585096716880798, Accuracy: 76.63646697998047,
Val Loss: 0.9143608212471008, Val Accuracy: 68.5
Iteration 1800, Epoch 3, Loss: 0.6590654253959656, Accuracy: 76.71932983398438,
Val Loss: 1.4682292938232422, Val Accuracy: 59.500003814697266
Iteration 1900, Epoch 3, Loss: 0.6505727767944336, Accuracy: 77.17225646972656,
Val Loss: 0.927227795124054, Val Accuracy: 69.5
Iteration 2000, Epoch 3, Loss: 0.6332799196243286, Accuracy: 77.74520874023438,
Val Loss: 0.9183900952339172, Val Accuracy: 70.5999984741211
Iteration 2100, Epoch 3, Loss: 0.6281720399856567, Accuracy: 77.88335418701172,
Val Loss: 0.990812361240387, Val Accuracy: 68.5999984741211
Iteration 2200, Epoch 3, Loss: 0.6138661503791809, Accuracy: 78.40994262695312,
Val Loss: 0.7790083289146423, Val Accuracy: 74.0999984741211
Iteration 2300, Epoch 4, Loss: 0.4864637851715088, Accuracy: 83.33332824707031,
Val Loss: 1.0910351276397705, Val Accuracy: 67.4000015258789
Iteration 2400, Epoch 4, Loss: 0.47865355014801025, Accuracy: 83.84405517578125,
Val Loss: 0.820149302482605, Val Accuracy: 72.79999542236328
Iteration 2500, Epoch 4, Loss: 0.4841322600841522, Accuracy: 83.32820129394531,
Val Loss: 0.9401343464851379, Val Accuracy: 69.0
Iteration 2600, Epoch 4, Loss: 0.47872093319892883, Accuracy: 83.53960418701172,
Val Loss: 1.051696538925171, Val Accuracy: 69.5999984741211
Iteration 2700, Epoch 4, Loss: 0.4718254506587982, Accuracy: 83.75852966308594,
Val Loss: 1.6867735385894775, Val Accuracy: 58.20000076293945
Iteration 2800, Epoch 4, Loss: 0.4620145559310913, Accuracy: 84.12959289550781,
Val Loss: 0.8847664594650269, Val Accuracy: 74.19999694824219
Iteration 2900, Epoch 4, Loss: 0.4580515921115875, Accuracy: 84.21175384521484,
Val Loss: 1.0884151458740234, Val Accuracy: 69.70000457763672
Iteration 3000, Epoch 4, Loss: 0.4465528130531311, Accuracy: 84.61726379394531,
Val Loss: 0.9101166725158691, Val Accuracy: 71.9000015258789
Iteration 3100, Epoch 5, Loss: 0.3343348503112793, Accuracy: 88.64019775390625,
Val Loss: 1.1402682065963745, Val Accuracy: 68.5
Iteration 3200, Epoch 5, Loss: 0.3379545509815216, Accuracy: 88.50364685058594,
Val Loss: 0.9418537616729736, Val Accuracy: 72.29999542236328
Iteration 3300, Epoch 5, Loss: 0.35348764061927795, Accuracy: 87.93512725830078,
Val Loss: 0.7520752549171448, Val Accuracy: 77.70000457763672
Iteration 3400, Epoch 5, Loss: 0.3452610373497009, Accuracy: 88.11665344238281,
Val Loss: 1.2003525495529175, Val Accuracy: 68.5999984741211
Iteration 3500, Epoch 5, Loss: 0.34819963574409485, Accuracy: 87.97196960449219,
Val Loss: 1.0770049095153809, Val Accuracy: 68.19999694824219
Iteration 3600, Epoch 5, Loss: 0.3434656262397766, Accuracy: 88.07029724121094,
Val Loss: 1.3848072290420532, Val Accuracy: 66.79999542236328
Iteration 3700, Epoch 5, Loss: 0.3356703221797943, Accuracy: 88.35606384277344,
Val Loss: 0.9750781059265137, Val Accuracy: 73.5999984741211
Iteration 3800, Epoch 5, Loss: 0.3261083662509918, Accuracy: 88.63212585449219,
Val Loss: 1.4661262035369873, Val Accuracy: 64.9000015258789
Iteration 3900, Epoch 6, Loss: 0.2567446529865265, Accuracy: 90.8450698852539,

Val Loss: 1.0090537071228027, Val Accuracy: 74.0
Iteration 4000, Epoch 6, Loss: 0.25472185015678406, Accuracy: 90.83515930175781,
Val Loss: 0.9656975269317627, Val Accuracy: 73.9000015258789
Iteration 4100, Epoch 6, Loss: 0.25628578662872314, Accuracy: 90.9421157836914,
Val Loss: 1.2046126127243042, Val Accuracy: 70.0999984741211
Iteration 4200, Epoch 6, Loss: 0.25789424777030945, Accuracy: 90.88611602783203,
Val Loss: 1.1936237812042236, Val Accuracy: 70.5999984741211
Iteration 4300, Epoch 6, Loss: 0.2531624436378479, Accuracy: 90.98991394042969,
Val Loss: 1.107487678527832, Val Accuracy: 71.80000305175781
Iteration 4400, Epoch 6, Loss: 0.24822482466697693, Accuracy: 91.150390625, Val
Loss: 1.388395071029663, Val Accuracy: 66.79999542236328
Iteration 4500, Epoch 6, Loss: 0.24032381176948547, Accuracy: 91.44467163085938,
Val Loss: 1.1176267862319946, Val Accuracy: 73.0
Iteration 4600, Epoch 7, Loss: 0.18045249581336975, Accuracy: 93.125, Val Loss:
1.4311168193817139, Val Accuracy: 67.69999694824219
Iteration 4700, Epoch 7, Loss: 0.18860596418380737, Accuracy: 93.37797546386719,
Val Loss: 1.117013692855835, Val Accuracy: 73.5999984741211
Iteration 4800, Epoch 7, Loss: 0.19260868430137634, Accuracy: 93.23933410644531,
Val Loss: 1.0524531602859497, Val Accuracy: 73.5
Iteration 4900, Epoch 7, Loss: 0.19207222759723663, Accuracy: 93.16598510742188,
Val Loss: 1.0800105333328247, Val Accuracy: 73.79999542236328
Iteration 5000, Epoch 7, Loss: 0.1872919648885727, Accuracy: 93.37577056884766,
Val Loss: 1.1237258911132812, Val Accuracy: 73.5
Iteration 5100, Epoch 7, Loss: 0.18036894500255585, Accuracy: 93.59529876708984,
Val Loss: 1.1738204956054688, Val Accuracy: 72.89999389648438
Iteration 5200, Epoch 7, Loss: 0.17783138155937195, Accuracy: 93.69576263427734,
Val Loss: 1.1361266374588013, Val Accuracy: 74.0999984741211
Iteration 5300, Epoch 7, Loss: 0.17345276474952698, Accuracy: 93.89405822753906,
Val Loss: 1.0958597660064697, Val Accuracy: 75.70000457763672
Iteration 5400, Epoch 8, Loss: 0.1459358185529709, Accuracy: 94.31089782714844,
Val Loss: 1.2481788396835327, Val Accuracy: 73.0999984741211
Iteration 5500, Epoch 8, Loss: 0.1542244851589203, Accuracy: 94.52562713623047,
Val Loss: 1.3518459796905518, Val Accuracy: 71.10000610351562
Iteration 5600, Epoch 8, Loss: 0.15384559333324432, Accuracy: 94.51490783691406,
Val Loss: 1.156948208808899, Val Accuracy: 71.10000610351562
Iteration 5700, Epoch 8, Loss: 0.1501299887895584, Accuracy: 94.73635864257812,
Val Loss: 1.0998262166976929, Val Accuracy: 74.69999694824219
Iteration 5800, Epoch 8, Loss: 0.14638641476631165, Accuracy: 94.83912658691406,
Val Loss: 1.0847746133804321, Val Accuracy: 76.4000015258789
Iteration 5900, Epoch 8, Loss: 0.14413852989673615, Accuracy: 94.91825103759766,
Val Loss: 1.233810305595398, Val Accuracy: 73.5
Iteration 6000, Epoch 8, Loss: 0.14326871931552887, Accuracy: 94.9212646484375,
Val Loss: 1.149505853652954, Val Accuracy: 75.4000015258789
Iteration 6100, Epoch 8, Loss: 0.13885453343391418, Accuracy: 95.0672378540039,
Val Loss: 1.2692354917526245, Val Accuracy: 73.69999694824219
Iteration 6200, Epoch 9, Loss: 0.1250068098306656, Accuracy: 95.50513458251953,
Val Loss: 1.2530502080917358, Val Accuracy: 74.0999984741211
Iteration 6300, Epoch 9, Loss: 0.11732673645019531, Accuracy: 95.61958312988281,

```
Val Loss: 1.1713415384292603, Val Accuracy: 73.5999984741211
Iteration 6400, Epoch 9, Loss: 0.11977241188287735, Accuracy: 95.69024658203125,
Val Loss: 1.2268697023391724, Val Accuracy: 72.5
Iteration 6500, Epoch 9, Loss: 0.12163068354129791, Accuracy: 95.63086700439453,
Val Loss: 1.2410246133804321, Val Accuracy: 72.19999694824219
Iteration 6600, Epoch 9, Loss: 0.11849036812782288, Accuracy: 95.76837158203125,
Val Loss: 1.108953595161438, Val Accuracy: 75.0999984741211
Iteration 6700, Epoch 9, Loss: 0.11831071227788925, Accuracy: 95.78152465820312,
Val Loss: 1.1682968139648438, Val Accuracy: 75.19999694824219
Iteration 6800, Epoch 9, Loss: 0.11855020374059677, Accuracy: 95.77915954589844,
Val Loss: 1.246986746788025, Val Accuracy: 74.0999984741211
Iteration 6900, Epoch 10, Loss: 0.11535124480724335, Accuracy:
96.20536041259766, Val Loss: 1.2228147983551025, Val Accuracy: 74.69999694824219
Iteration 7000, Epoch 10, Loss: 0.10356191545724869, Accuracy:
96.23247528076172, Val Loss: 1.1028289794921875, Val Accuracy: 74.69999694824219
Iteration 7100, Epoch 10, Loss: 0.10001878440380096, Accuracy:
96.52777862548828, Val Loss: 1.1288849115371704, Val Accuracy: 76.4000015258789
Iteration 7200, Epoch 10, Loss: 0.09920195490121841, Accuracy:
96.50346374511719, Val Loss: 0.971367597579956, Val Accuracy: 77.20000457763672
Iteration 7300, Epoch 10, Loss: 0.09890499711036682, Accuracy:
96.54100036621094, Val Loss: 1.2718911170959473, Val Accuracy: 73.69999694824219
Iteration 7400, Epoch 10, Loss: 0.09917673468589783, Accuracy:
96.52366638183594, Val Loss: 1.3118406534194946, Val Accuracy: 74.5999984741211
Iteration 7500, Epoch 10, Loss: 0.09908019006252289, Accuracy:
96.53778839111328, Val Loss: 1.0309803485870361, Val Accuracy: 76.20000457763672
Iteration 7600, Epoch 10, Loss: 0.09895777702331543, Accuracy:
96.55233764648438, Val Loss: 1.1956337690353394, Val Accuracy: 76.20000457763672
```

8.1 Describe what you did

In the cell below you should write an explanation of what you did, any additional features that you implemented, and/or any graphs that you made in the process of training and evaluating your network.

Answer:

8.1.1 Network design

I used a **ResNet** (paper) with the following architecture: 1. `CONV->NORM->RELU` * As the input is of shape `32x32x3`, initially performed a convolution with filter size `5x5` to produce `32` *activation maps*. Padding of `2` was used to preserve the input spacial area size. * The `32` *activation maps*, each of shape `32x32`, were then passed through *batchnorm* layer and through *ReLU* nonlinearity. 2. `[IDENTITY->IDENTITY]->[BOTLENECK->IDENTITY]x3` * In general each block has `2` layers, each of which perfom a *convolution*, *batchnorm* and *ReLU*. The second layer adds together its output with block's raw input to produce the final activations. The channel size grows by a factor of `2` after each block. * Each *identity* block doe not change the spacial area of the input, however, the *bottleneck* block reduces the spacial size by a factor of `2`. * The 'preprocessed' input of size `32x32x32` after each block had its shape modified and its dimensions changed in the following way `32x32x64->16x16x128->8x8x256->4x4x512` (based on the provided sequence of blocks). 3.

`POOL->SOFTMAX` * The activation maps are passed through *global average pooling* to get the final activation shape of `1x1x512` which is then flattened to a linear classifier which produces scores based on the *Softmax* loss function.

> Key motivation is that by adding together the outputs of the second layer with the inputs of the first layer (per block), it allows the network to extract more features when many layers are used because the vanishing gradient problem disappears.

8.1.2 Hyperparameters

I did not fine-tune any parameters, simply used the provided *learning rate* of `0.001` with already defined **Adam** optimizer.

[ ]: