

# BatchNormalization

February 14, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment2/'
FOLDERNAME = 'icv83551/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/$FOLDERNAME/icv83551/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 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.

```
[6]: # Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from icv83551.data_utils import get_CIFAR10_data
from icv83551.gradient_check import eval_numerical_gradient,
    eval_numerical_gradient_array
from icv83551.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")
```

```
[7]: # 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 `icv83551/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!

```
[8]: from icv83551.layers import *

# 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]

```

[9]:

```

# 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 out-

going branches; make sure to sum gradients across these branches in the backward pass. Referencing the paper linked to above in [1] may be helpful!

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

```
[10]: # 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:  1.7029258328157158e-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  $X = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix}$ ,

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

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

$$\sigma = \sqrt{v + \epsilon} \quad y_i = \frac{x_i - \mu}{\sigma} \quad (2)$$

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.

```
[71]: 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: 7.682750402459526e-13
dgamma difference: 0.0
dbeta difference: 0.0
speedup: 1.85x
```

## 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 `icv83551/classifiers/fc_net.py`. Recall that you implemented the network initialization, forward pass, and backward pass in Assignment 1. Copy that implementation here, and modify it to incorporate 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 `icv83551/layer_utils.py`.

```
[72]: from icv83551.classifiers.fc_net import *
from icv83551.gradient_check import *

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: 8.22e-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.

```
[73]: 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: 2.340975  
 (Epoch 0 / 10) train acc: 0.107000; val\_acc: 0.115000  
 (Epoch 1 / 10) train acc: 0.312000; val\_acc: 0.259000  
 (Iteration 21 / 200) loss: 2.033403  
 (Epoch 2 / 10) train acc: 0.420000; val\_acc: 0.293000  
 (Iteration 41 / 200) loss: 2.042188  
 (Epoch 3 / 10) train acc: 0.484000; val\_acc: 0.305000  
 (Iteration 61 / 200) loss: 1.722667  
 (Epoch 4 / 10) train acc: 0.552000; val\_acc: 0.302000  
 (Iteration 81 / 200) loss: 1.352551  
 (Epoch 5 / 10) train acc: 0.634000; val\_acc: 0.325000  
 (Iteration 101 / 200) loss: 1.325028  
 (Epoch 6 / 10) train acc: 0.658000; val\_acc: 0.322000  
 (Iteration 121 / 200) loss: 1.031080  
 (Epoch 7 / 10) train acc: 0.696000; val\_acc: 0.343000  
 (Iteration 141 / 200) loss: 1.109879  
 (Epoch 8 / 10) train acc: 0.701000; val\_acc: 0.310000  
 (Iteration 161 / 200) loss: 0.836526  
 (Epoch 9 / 10) train acc: 0.753000; val\_acc: 0.312000  
 (Iteration 181 / 200) loss: 0.842216  
 (Epoch 10 / 10) train acc: 0.797000; val\_acc: 0.311000

Solver without batch norm:

(Iteration 1 / 200) loss: 2.302331

```
(Epoch 0 / 10) train acc: 0.128000; val_acc: 0.132000
(Epoch 1 / 10) train acc: 0.263000; val_acc: 0.226000
(Iteration 21 / 200) loss: 2.064354
(Epoch 2 / 10) train acc: 0.319000; val_acc: 0.266000
(Iteration 41 / 200) loss: 1.856846
(Epoch 3 / 10) train acc: 0.368000; val_acc: 0.287000
(Iteration 61 / 200) loss: 1.760862
(Epoch 4 / 10) train acc: 0.396000; val_acc: 0.309000
(Iteration 81 / 200) loss: 1.696757
(Epoch 5 / 10) train acc: 0.485000; val_acc: 0.328000
(Iteration 101 / 200) loss: 1.591938
(Epoch 6 / 10) train acc: 0.494000; val_acc: 0.324000
(Iteration 121 / 200) loss: 1.621612
(Epoch 7 / 10) train acc: 0.550000; val_acc: 0.296000
(Iteration 141 / 200) loss: 1.339807
(Epoch 8 / 10) train acc: 0.600000; val_acc: 0.323000
(Iteration 161 / 200) loss: 0.998247
(Epoch 9 / 10) train acc: 0.646000; val_acc: 0.326000
(Iteration 181 / 200) loss: 0.993196
(Epoch 10 / 10) train acc: 0.656000; val_acc: 0.312000
```

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.

```
[74]: 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)

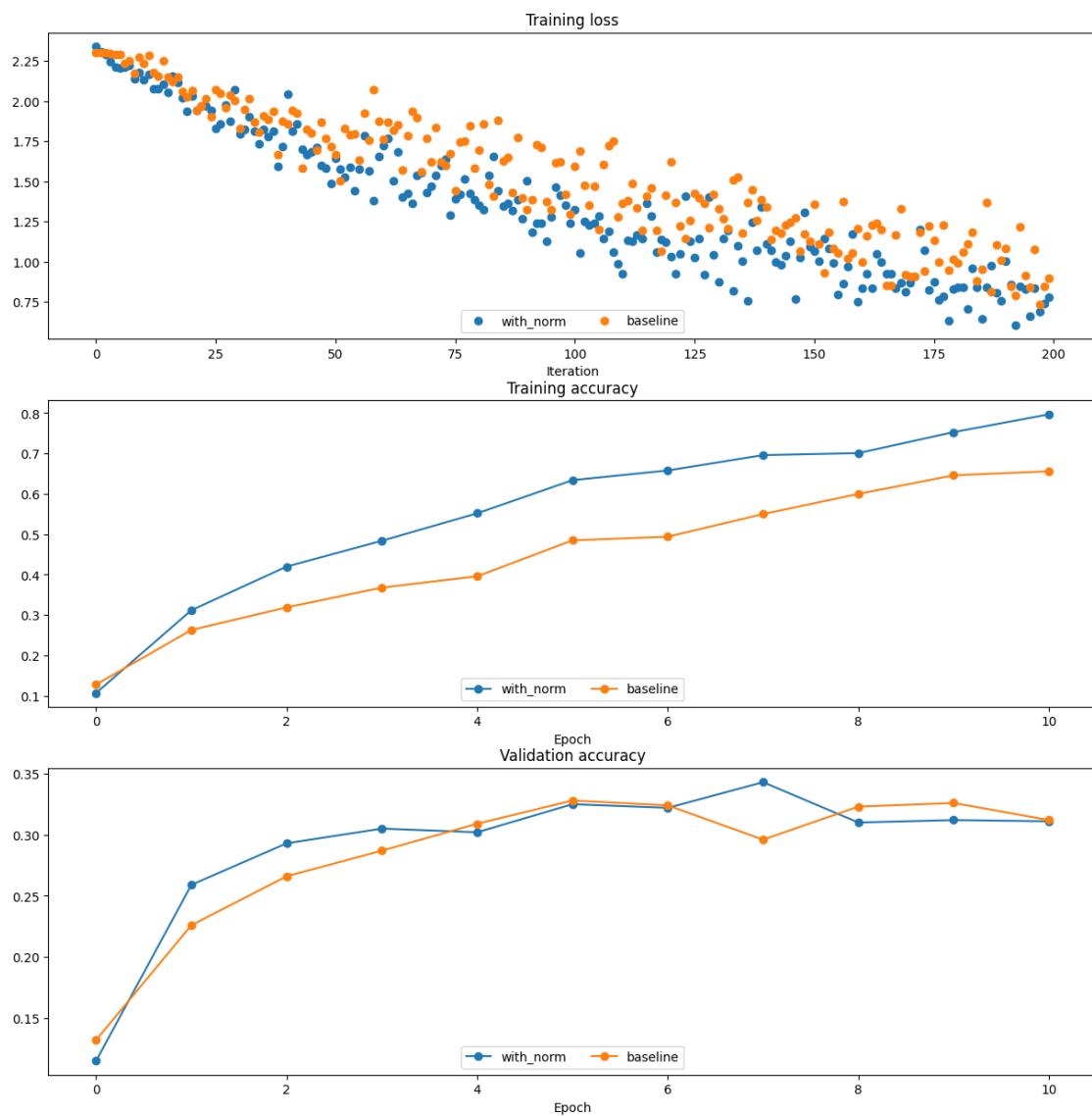
    plt.subplot(3, 1, 1)
    plot_training_history('Training loss','Iteration', solver, [bn_solver], \
                          lambda x: x.loss_history, bl_marker='o', bn_marker='o')
    plt.subplot(3, 1, 2)
```

```

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.

```
[75]: 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

C:\Users\eitan\PythonProjects\CV_projects\assignment2\icv83551\layers.py:136:
RuntimeWarning: divide by zero encountered in log
    y_hat = np.log(softmax[range(num), y]) # extracting only the indices we want

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

```

[76]: *# 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')

```

```

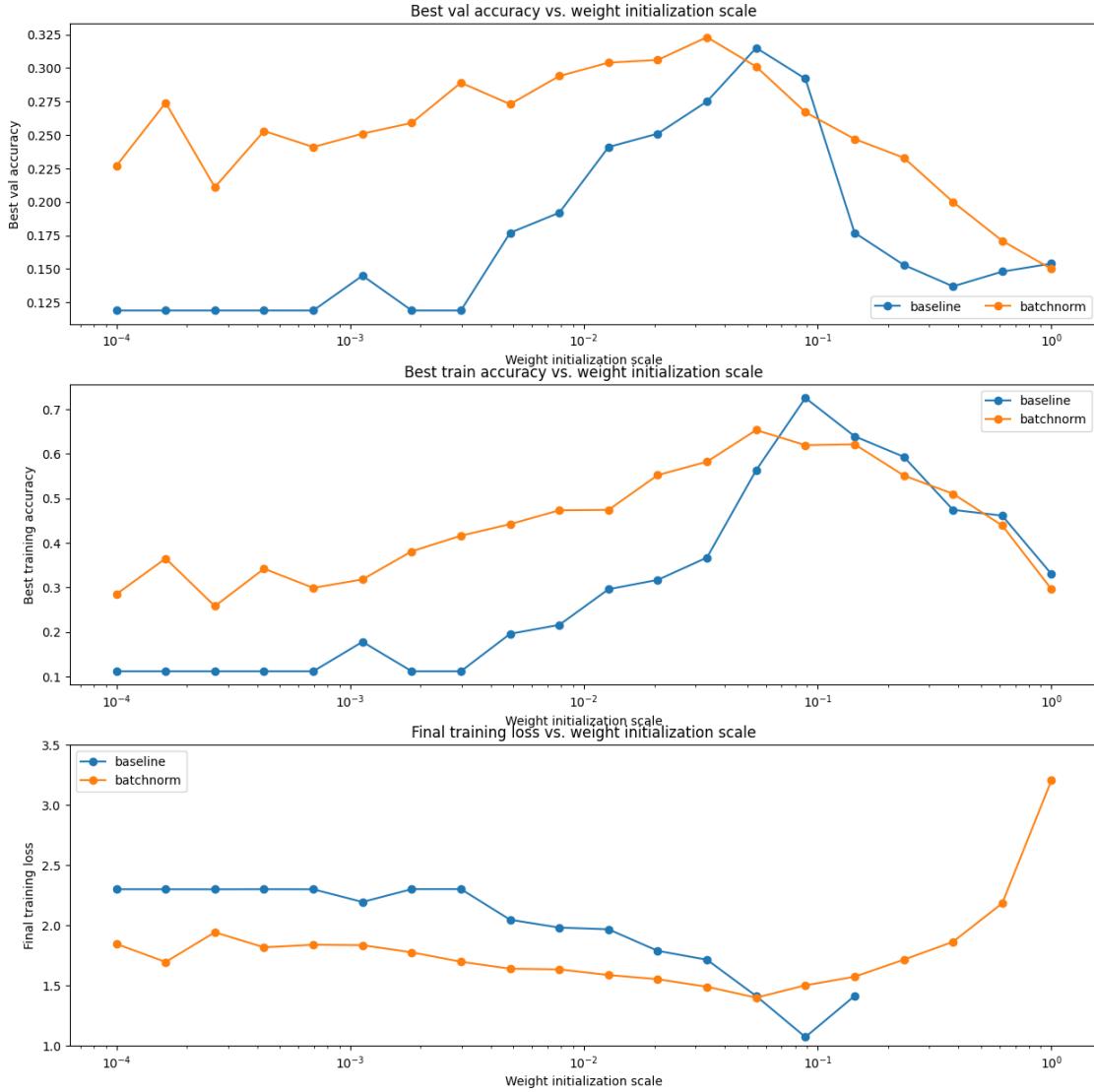
plt.semilogx(weight_scales, best_val_accs, '-o', label='baseline')
plt.semilogx(weight_scales, bn_best_val_accs, '-o', label='batchnorm')
plt.legend(ncol=2, loc='lower right')

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:

### 7.2.1 1. Accuracy and Optimization Efficiency

Overall, using **Batch Normalization (BN)** results in superior accuracy and lower loss. While a baseline model might show a high training accuracy “spike” at a specific weight scale, its validation accuracy often remains stagnant, indicating **overfitting**. Batch Normalization improves results because it smoothens the loss landscape, making gradient descent steps more directed toward the minima. Without it, different parameter scales can create an awkward loss topology that places

disproportionate emphasis on certain gradients, leading to higher loss and slower convergence.

---

### 7.2.2 2. Robustness to Initialization Scales

A significant advantage of Batch Normalization is its **robustness** across varying weight initializations. \* **Without BN:** Even a minor shift in the initial weight scale can produce drastically different (and often poor) results. \* **With BN:** Performance remains consistent because the model centers and scales the data at each layer. This normalization ensures that small changes in weight values do not cause massive fluctuations in the output compared to uncentered, unnormalized data.

---

### 7.2.3 3. Stability against Vanishing and Exploding Gradients

Batch Normalization maintains high performance even at extreme weight scales (e.g., very large scales like  $10^1$  or very small scales like  $10^{-4}$ ). \* **In Baseline Models:** Large initializations cause activations to explode across multiple layers, while small initializations cause them to vanish. This leads to skewed gradients that either jump over the minima or fail to move at all. \* **In BN Models:** By normalizing the raw output of each layer so that data points are spread around a mean of 0 and variance of 1, the network keeps the gradients within a functional range. This prevents the accumulation of extreme values and ensures stable training regardless of the initial weight magnitude.

## 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.

```
[77]: 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

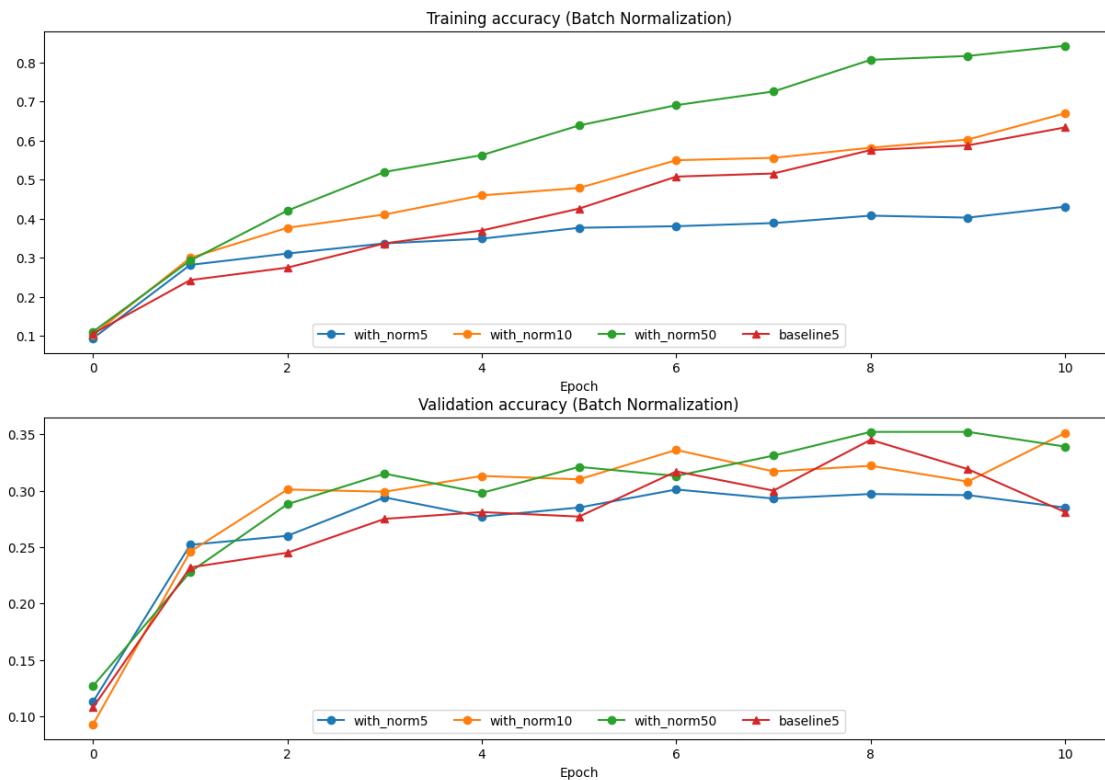
[78]: plt.subplot(2, 1, 1)  
plot\_training\_history('Training accuracy (Batch Normalization)', 'Epoch',□
↪solver\_bsize, bn\_solvers\_bsize, \
 lambda x: x.train\_acc\_history, bl\_marker='^-^',□
↪bn\_marker='^-o', labels=batch\_sizes)  
plt.subplot(2, 1, 2)

```

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:

### 8.2.1 1. Training Accuracy and the Impact of Batch Size

The experiment demonstrates that **larger batch sizes** generally lead to higher training accuracy when using Batch Normalization. \* **Observation:** At very small batch sizes (e.g.,  $N = 5$ ), the baseline model (without BN) actually performs better. \* **The “Why”:** Training accuracy grows with batch size because the model receives more stable gradient updates. However, with very small batches, the **mean** and **variance** used by BN are poor approximations of the true population statistics. This introduces high levels of noise into the “adjusted features,” making it difficult for

the model to learn effectively compared to a baseline model training on raw, exact features.

---

### 8.2.2 2. Validation Accuracy and Convergence Speed

The validation results follow a similar trend where larger batches yield better accuracy, though the performance gap tends to narrow over time.

\* **Convergence:** Models with larger batch sizes converge much faster within the first few epochs due to more reliable batch statistics.

\* **Generalization:** Eventually, the accuracy across different batch sizes begins to equalize after enough epochs. This happens because the model cannot overfit the validation set as easily as the training set, and given enough iterations, it will eventually reach a stable state regardless of the initial batch noise.

---

### 8.2.3 3. Relationship Implied: Sensitivity to Statistical Noise

This experiment implies that the effectiveness of Batch Normalization is **strictly dependent on the batch size**.

\* **The Core Issue:** Because BN relies on calculating the batch mean ( $\mu$ ) and variance ( $\sigma^2$ ) at every layer, it requires a sufficiently large sample to provide a representative estimate of the data distribution.

\* **The Result:** When the batch size is too small, the high variance in these estimates acts as unwanted noise rather than a helpful regularizer, ultimately degrading the model's ability to map features correctly.

## 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:

### 9.2.1 1. Batch Normalization Analogy: Option 3

**Option 3** (Subtracting the mean image of the dataset) is analogous to **Batch Normalization**. Batch Normalization operates across the “batch” dimension. It calculates statistics (mean and variance) by looking at the same feature across all samples in a training batch. Similarly, subtracting the mean image requires looking at the entire collection of images to find an average, then adjusting each individual image based on that global/batch-level information.

---

### 9.2.2 2. Layer Normalization Analogy: Option 2

**Option 2** (Scaling all pixels within a single image to sum to 1) is analogous to **Layer Normalization**. Layer Normalization operates across the “feature” dimension for each individual sample independently. It calculates statistics using all the features (pixels/channels) within a single data point. In Option 2, the normalization factor is derived solely from the internal pixels of that specific image, making it independent of other images in the batch—exactly how Layer Normalization functions.

---

### 9.2.3 3. Summary of Key Differences

- **Batch Normalization** ( $\mu, \sigma$  calculated across samples): Normalizes each feature based on how other samples in the batch behave.
- **Layer Normalization** ( $\mu, \sigma$  calculated across features): Normalizes the sample based on how its own internal features behave.

## 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 `icv83551/layers.py`, implement the forward pass for layer normalization in the function `layernorm_forward`.

Run the cell below to check your results. \* In `icv83551/layers.py`, implement the backward pass for layer normalization in the function `layernorm_backward`.

Run the second cell below to check your results. \* Modify `icv83551/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.

```
[79]: # 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.]
stds: [2.99999985 2.99999998 2.99999999 2.99999907]
```

```
[80]: # 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: 1.433615913366895e-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!

```

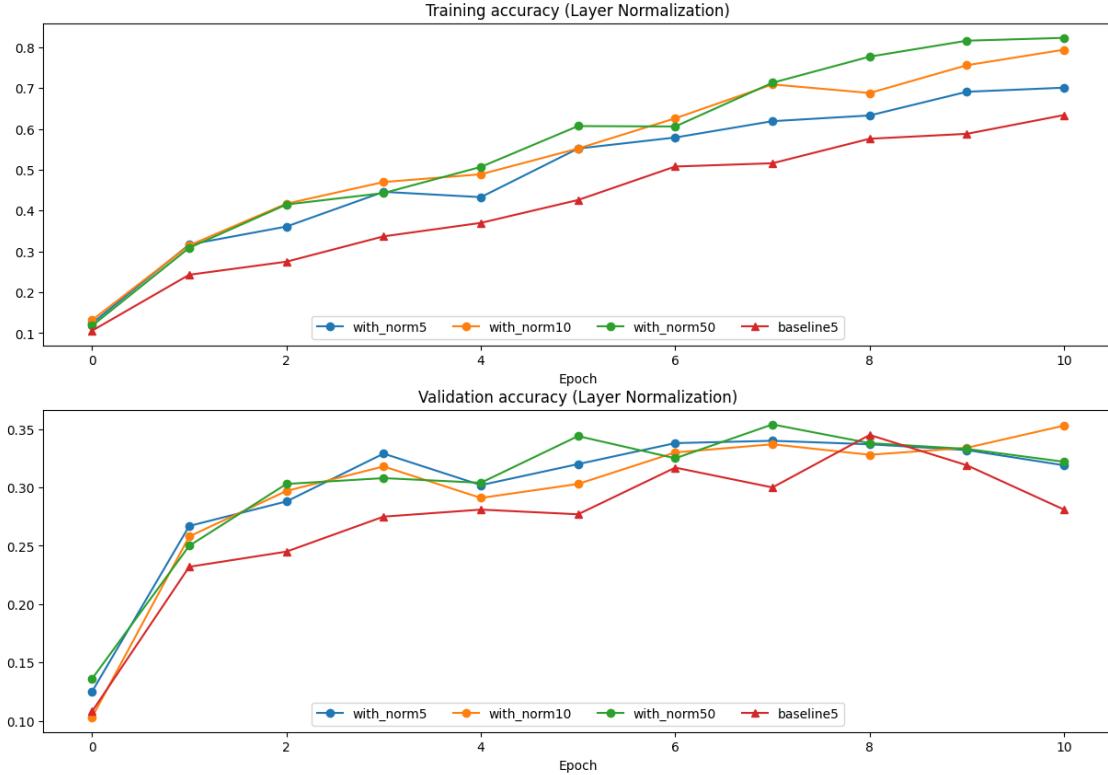
[81]: ln_solvers_bsize, solver_bsize, batch_sizes = ↴
       run_batchsize_experiments('layernorm')

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:

### 11.2.1 1. Depth of Network: False

Layer Normalization (LN) is actually **well-suited** for deep networks. Since each feature vector is normalized independently within its own layer, adding more layers does not degrade the performance of the normalization process. LN helps center activations within the feature space, leading to a smoother loss function topology. Furthermore, by scaling features to a consistent range, it actively helps prevent **vanishing and exploding gradients**, which are common issues in very deep architectures.

### 11.2.2 2. Small Feature Dimensions: True

Layer Normalization is likely to perform poorly when the **feature dimension** (the number of hidden units or pixels per sample) is very small. LN calculates the mean ( $\mu$ ) and variance ( $\sigma^2$ ) across all features of a single sample. If there are only a few features, these statistics are calculated from a tiny sample size and will not accurately represent the distribution. This leads to **noisy estimates** and unstable scaling, which can hinder the model's ability to learn meaningful patterns.

[Image of Layer Normalization vs Batch Normalization statistical calculation]

---

### 11.2.3 3. High Regularization: True

A very high regularization term (like a large  $L_2$  penalty) can interfere with the effectiveness of any normalization layer, including LN. High regularization heavily penalizes weight magnitudes, forcing the model to become “simpler.” This can prevent the weights from properly scaling or shifting specific features within a vector, which is necessary to extract complex information after the normalization step. Excessive regularization tends to over-constrain the model, leading to higher loss and a failure to capture the underlying data distribution.

# Dropout

February 14, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment2/'
FOLDERNAME = 'icv83551/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/$FOLDERNAME/icv83551/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 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

```
[1]: # Setup cell.
import time
import numpy as np
import matplotlib.pyplot as plt
from icv83551.classifiers.fc_net import *
from icv83551.data_utils import get_CIFAR10_data
from icv83551.gradient_check import eval_numerical_gradient,
    eval_numerical_gradient_array
```

```

from icv83551.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))))

```

[2]: # Load the (preprocessed) CIFAR-10 data.

```

data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"[{k}]: {v.shape}")

```

```

C:\Users\eitan\PythonProjects\CV_projects\assignment2\icv83551\data_utils.py:16:
VisibleDeprecationWarning: dtype(): align should be passed as Python or NumPy
boolean but got `align=0`. Did you mean to pass a tuple to create a subarray
type? (Deprecated NumPy 2.4)
    return pickle.load(f, encoding="latin1")

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 `icv83551/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.

[3]:

```

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:  10.0002078784775
Mean of train-time output:  10.014059116977283
Mean of test-time output:  10.0002078784775
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:  10.0002078784775
Mean of train-time output:  9.977917658761159
Mean of test-time output:  10.0002078784775
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:  10.0002078784775
Mean of train-time output:  9.987811912159428
Mean of test-time output:  10.0002078784775
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 `icv83551/layers.py`, implement the backward pass for dropout. After doing so, run the following cell to numerically gradient-check your implementation.

```

[4]: 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:

#### 3.2.1 1. The Impact of Failing to Scale by $1/p$

If we do not divide the activations by the keep probability  $p$  during training, the model will produce **incorrect predictions at test time**. This occurs because the scale of the activations arriving at the next layer during testing will be significantly higher than the scale the network “learned” to handle during training.

#### 3.2.2 2. Expected Output without Scaling

During training, a dropout layer randomly zeros out neurons. Without scaling, the **expected value**  $E[\hat{y}]$  of a neuron producing output  $\hat{y}$  is:

$$E[\hat{y}] = p\hat{y} + (1 - p)0 = p\hat{y}$$

- $p\hat{y}$ : The probability of the neuron being active.
- $(1 - p)0$ : The probability of the neuron being dropped (outputting zero).

Since  $E[\hat{y}] = p\hat{y}$ , the outputs during training are “dimmed” by a factor of  $p$ . However, at test time, all neurons are active ( $p = 1$ ), meaning the outputs will be roughly  $1/p$  times larger than what the subsequent layers expected based on training.

---

#### 3.2.3 3. The Advantage of Inverse Dropout

By dividing by  $p$  (or multiplying by  $1/p$ ) during the training phase, we normalize the expected value so that it matches the test-time scale:

$$E[\hat{y}] = \frac{p}{p}\hat{y} + \frac{(1 - p)}{p}0 = \hat{y}$$

**Conclusion:** Scaling by  $1/p$  during training ensures that the magnitude of the signal remains consistent between training and testing. This allows us to keep the test-time inference simple and efficient, as we do not need to perform any additional scaling or modifications when the model is deployed.

## 4 Fully Connected Networks with Dropout

In the file `icv83551/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.

```
[5]: 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.

```
[6]: # 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: 7.856642
(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: 0.710000; val_acc: 0.298000
(Epoch 6 / 25) train acc: 0.720000; val_acc: 0.282000
(Epoch 7 / 25) train acc: 0.832000; val_acc: 0.255000
(Epoch 8 / 25) train acc: 0.874000; val_acc: 0.267000
(Epoch 9 / 25) train acc: 0.898000; val_acc: 0.276000
(Epoch 10 / 25) train acc: 0.900000; val_acc: 0.263000
(Epoch 11 / 25) train acc: 0.934000; val_acc: 0.269000
(Epoch 12 / 25) train acc: 0.966000; val_acc: 0.298000
(Epoch 13 / 25) train acc: 0.974000; val_acc: 0.306000
(Epoch 14 / 25) train acc: 0.968000; val_acc: 0.321000
(Epoch 15 / 25) train acc: 0.978000; val_acc: 0.302000
(Epoch 16 / 25) train acc: 0.980000; val_acc: 0.308000
(Epoch 17 / 25) train acc: 0.968000; val_acc: 0.302000
(Epoch 18 / 25) train acc: 0.974000; val_acc: 0.306000
(Epoch 19 / 25) train acc: 0.986000; val_acc: 0.279000
(Epoch 20 / 25) train acc: 0.980000; val_acc: 0.285000
(Iteration 101 / 125) loss: 0.450466
(Epoch 21 / 25) train acc: 0.970000; val_acc: 0.312000
(Epoch 22 / 25) train acc: 0.972000; val_acc: 0.311000
(Epoch 23 / 25) train acc: 0.972000; val_acc: 0.302000
(Epoch 24 / 25) train acc: 0.972000; val_acc: 0.298000
(Epoch 25 / 25) train acc: 0.986000; val_acc: 0.294000

0.25
(Iteration 1 / 125) loss: 17.318476
(Epoch 0 / 25) train acc: 0.230000; val_acc: 0.177000

C:\Users\eitan\PythonProjects\CV_projects\assignment2\icv83551\layers.py:136:
RuntimeWarning: divide by zero encountered in log
    y_hat = np.log(softmax[range(num), y]) # extracting only the indices we want

(Epoch 1 / 25) train acc: 0.378000; val_acc: 0.243000
(Epoch 2 / 25) train acc: 0.404000; val_acc: 0.254000
(Epoch 3 / 25) train acc: 0.496000; val_acc: 0.276000
(Epoch 4 / 25) train acc: 0.526000; val_acc: 0.300000
(Epoch 5 / 25) train acc: 0.576000; val_acc: 0.301000
(Epoch 6 / 25) train acc: 0.618000; val_acc: 0.296000
(Epoch 7 / 25) train acc: 0.658000; val_acc: 0.307000

```

```

(Epoch 8 / 25) train acc: 0.694000; val_acc: 0.316000
(Epoch 9 / 25) train acc: 0.728000; val_acc: 0.298000
(Epoch 10 / 25) train acc: 0.730000; val_acc: 0.301000
(Epoch 11 / 25) train acc: 0.794000; val_acc: 0.320000
(Epoch 12 / 25) train acc: 0.778000; val_acc: 0.278000
(Epoch 13 / 25) train acc: 0.822000; val_acc: 0.308000
(Epoch 14 / 25) train acc: 0.786000; val_acc: 0.325000
(Epoch 15 / 25) train acc: 0.844000; val_acc: 0.329000
(Epoch 16 / 25) train acc: 0.828000; val_acc: 0.311000
(Epoch 17 / 25) train acc: 0.834000; val_acc: 0.287000
(Epoch 18 / 25) train acc: 0.868000; val_acc: 0.341000
(Epoch 19 / 25) train acc: 0.888000; val_acc: 0.327000
(Epoch 20 / 25) train acc: 0.902000; val_acc: 0.306000
(Iteration 101 / 125) loss: 4.683331
(Epoch 21 / 25) train acc: 0.876000; val_acc: 0.315000
(Epoch 22 / 25) train acc: 0.884000; val_acc: 0.306000
(Epoch 23 / 25) train acc: 0.884000; val_acc: 0.318000
(Epoch 24 / 25) train acc: 0.906000; val_acc: 0.316000
(Epoch 25 / 25) train acc: 0.926000; val_acc: 0.310000

```

```
[7]: # 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)
plt.title('Train accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend(ncol=2, loc='lower right')

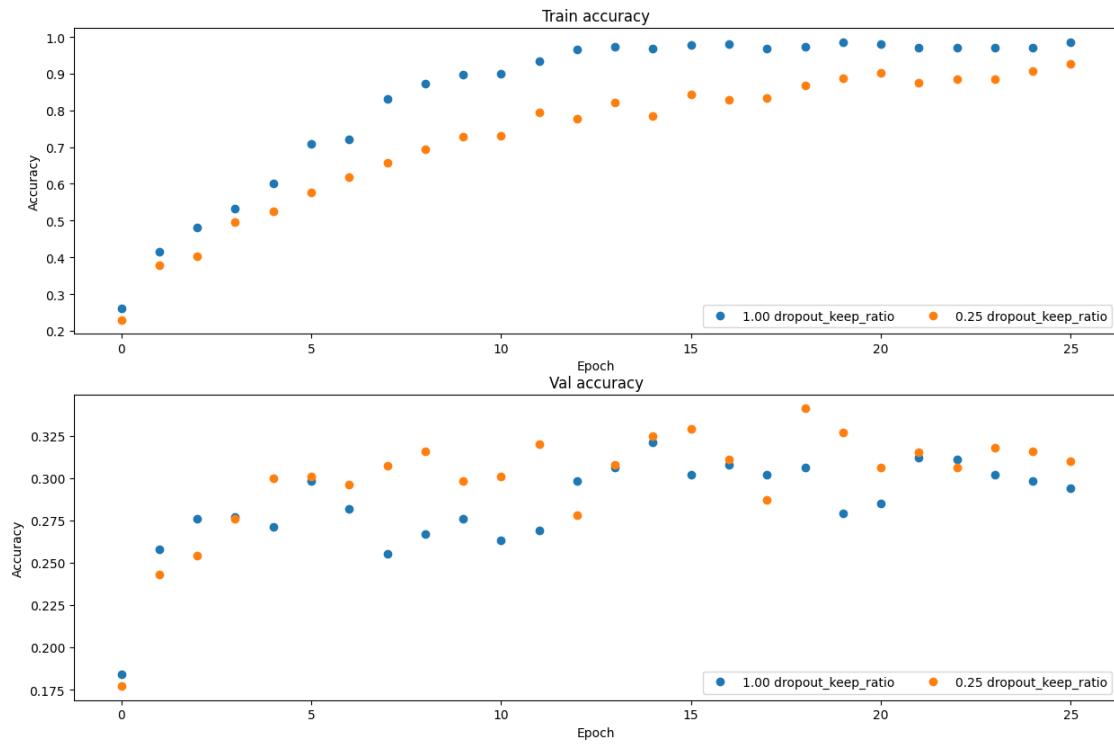
plt.subplot(3, 1, 2)
for dropout_keep_ratio in dropout_choices:
    plt.plot(
        solvers[dropout_keep_ratio].val_acc_history, 'o', label='%.2f' % dropout_keep_ratio)
plt.title('Val accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
```

```

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:

### 5.2.1 1. Training Performance: The Cost of Regularization

During the training phase, the model **without dropout** typically achieves higher accuracy. Without dropout, the network is unrestricted; it can utilize every available neuron to memorize the specific features and noise within the training set. While the accuracy is higher, this often indicates that the model is **overfitting** to the training data rather than learning generalized patterns.

### 5.2.2 2. Validation Performance: Improved Generalization

During the validation phase, the model **with dropout** generally produces superior results. Dropout acts as a powerful **regularizer** by randomly “dropping” neurons during each forward pass. This

prevents the network from developing complex co-adaptations where it relies too heavily on specific, individual neurons. Because no single neuron can be relied upon, the model is forced to learn more **robust, redundant features** that are distributed across the entire architecture.

---

### 5.2.3 3. Conclusion: Dropout's Role in Robustness

The results suggest that dropout is an effective tool for controlling overfitting. Although it may slightly decrease training performance by making the optimization task more difficult, it significantly enhances the model's ability to **generalize** to unseen data. By introducing stochastic noise during training, dropout ensures the network remains robust and prevents it from simply memorizing the training samples.

[ ]:

# Convolutional Networks

February 14, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment2/'
FOLDERNAME = 'icv83551/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/$FOLDERNAME/icv83551/datasets/
!bash get_datasets.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 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.

```
[1]: # Setup cell.
import numpy as np
import matplotlib.pyplot as plt
from icv83551.classifiers.cnn import *
from icv83551.data_utils import get_CIFAR10_data
from icv83551.gradient_check import eval_numerical_gradient_array,
    eval_numerical_gradient
```

```

from icv83551.layers import *
from icv83551.fast_layers import *
from icv83551.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))))

```

[2]: # Load the (preprocessed) CIFAR-10 data.

```

data = get_CIFAR10_data()
for k, v in list(data.items()):
    print(f"{k}: {v.shape}")

```

```

C:\Users\eitan\PythonProjects\CV_projects\assignment2\icv83551\data_utils.py:16:
VisibleDeprecationWarning: dtype(): align should be passed as Python or NumPy
boolean but got `align=0`. Did you mean to pass a tuple to create a subarray
type? (Deprecated NumPy 2.4)
    return pickle.load(f, encoding="latin1")

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 `icv83551/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:

```
[3]: 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.

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

kitten = imread('icv83551/notebook_images/kitten.jpg')
puppy = imread('icv83551/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)

```

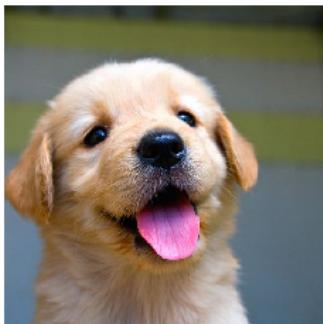
```
plt.subplot(2, 3, 5)
imshow_no_ax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_no_ax(out[1, 1])
plt.show()
```

```
C:\Users\eitan\AppData\Local\Temp\ipykernel_10516\2167355465.py: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('icv83551/notebook_images/kitten.jpg')
C:\Users\eitan\AppData\Local\Temp\ipykernel_10516\2167355465.py: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('icv83551/notebook_images/puppy.jpg')
```

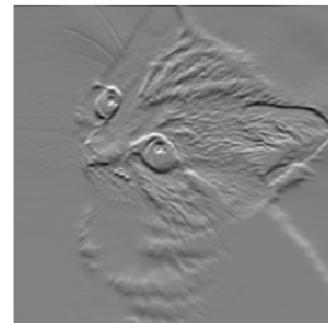
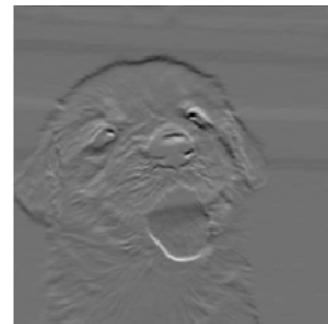
Original image



Grayscale



Edges



### 3 Convolution: Naive Backward Pass

Implement the backward pass for the convolution operation in the function `conv_backward_naive` in the file `icv83551/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.

```
[5]: 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, u
    ↪conv_param)[0], x, dout)
dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, u
    ↪conv_param)[0], w, dout)
db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, u
    ↪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 `icv83551/layers.py`. Again, don't worry too much about computational efficiency.

Check your implementation by running the following:

```
[6]: 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 `icv83551/layers.py`. You don't need to worry about computational efficiency.

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

```

[7]: 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, u
                           ↵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: 3.27562514223145e-12

## 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 `icv83551/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 preceding cells from top to bottom and skip the cell below as you only need to run it once for the compilation step.

```
[8]: # Remember to restart the runtime after executing this cell!
%cd /content/drive/My\ Drive/$FOLDERNAME/icv83551/
!python setup.py build_ext --inplace
%cd /content/drive/My\ Drive/$FOLDERNAME/
```

[WinError 3] The system cannot find the path specified: '/content/drive/My\\Drive/\$FOLDERNAME/icv83551/'  
C:\Users\eitan\PythonProjects\CV\_projects\assignment2  
[WinError 3] The system cannot find the path specified: '/content/drive/My\\Drive/\$FOLDERNAME/'  
C:\Users\eitan\PythonProjects\CV\_projects\assignment2  
  
python: can't open file  
'C:\\\\Users\\\\eitan\\\\PythonProjects\\\\CV\_projects\\\\assignment2\\\\setup.py': [Errno 2] No such file or directory

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 receives 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:

```
[9]: # Rel errors should be around e-9 or less.
from icv83551.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: 0.074148s
Fast: 0.005723s
Speedup: 12.956714x
Difference:  4.926407851494105e-11

```

```

Testing conv_backward_fast:
Naive: 2.641868s
Fast: 0.007715s
Speedup: 342.412039x
dx difference:  1.949764775345631e-11
dw difference:  4.957046344783224e-13
db difference:  0.0

```

```

[10]: # Relative errors should be close to 0.0.
from icv83551.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: 0.155646s
fast: 0.002420s
speedup: 64.324367x
difference:  0.0

```

Testing pool\_backward\_fast:

```

Naive: 0.437175s
fast: 0.006915s
speedup: 63.218169x
dx difference:  0.0

```

## 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 `icv83551/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.

```
[11]: from icv83551.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:  5.802401370096438e-09
db error:  1.0146343411762047e-09

```

```

[12]: from icv83551.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 `icv83551/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.

```
[13]: 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):  2.5082556233317708
```

### 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 artificial 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.

```
[14]: 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.

```

[15]: 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: 3.102719
(Epoch 1 / 15) train acc: 0.140000; val_acc: 0.087000
(Iteration 3 / 30) loss: 2.270332
(Iteration 4 / 30) loss: 2.099074
(Epoch 2 / 15) train acc: 0.230000; val_acc: 0.093000
(Iteration 5 / 30) loss: 1.841253
(Iteration 6 / 30) loss: 1.935296
(Epoch 3 / 15) train acc: 0.490000; val_acc: 0.168000
(Iteration 7 / 30) loss: 1.828834
(Iteration 8 / 30) loss: 1.652295
(Epoch 4 / 15) train acc: 0.520000; val_acc: 0.182000
(Iteration 9 / 30) loss: 1.332530
(Iteration 10 / 30) loss: 1.772358
(Epoch 5 / 15) train acc: 0.640000; val_acc: 0.171000
(Iteration 11 / 30) loss: 1.029629
(Iteration 12 / 30) loss: 1.038692
(Epoch 6 / 15) train acc: 0.720000; val_acc: 0.226000
(Iteration 13 / 30) loss: 1.152896
(Iteration 14 / 30) loss: 0.834351
(Epoch 7 / 15) train acc: 0.810000; val_acc: 0.248000
(Iteration 15 / 30) loss: 0.584665
(Iteration 16 / 30) loss: 0.644552
(Epoch 8 / 15) train acc: 0.830000; val_acc: 0.240000
(Iteration 17 / 30) loss: 0.811508
(Iteration 18 / 30) loss: 0.430228
(Epoch 9 / 15) train acc: 0.840000; val_acc: 0.175000
(Iteration 19 / 30) loss: 0.421580
(Iteration 20 / 30) loss: 0.555581
(Epoch 10 / 15) train acc: 0.930000; val_acc: 0.197000
(Iteration 21 / 30) loss: 0.364954
(Iteration 22 / 30) loss: 0.271303
(Epoch 11 / 15) train acc: 0.860000; val_acc: 0.215000
(Iteration 23 / 30) loss: 0.405969
(Iteration 24 / 30) loss: 0.385744

```

```
(Epoch 12 / 15) train acc: 0.950000; val_acc: 0.207000
(Iteration 25 / 30) loss: 0.109979
(Iteration 26 / 30) loss: 0.113894
(Epoch 13 / 15) train acc: 0.960000; val_acc: 0.218000
(Iteration 27 / 30) loss: 0.123754
(Iteration 28 / 30) loss: 0.172242
(Epoch 14 / 15) train acc: 0.990000; val_acc: 0.222000
(Iteration 29 / 30) loss: 0.123923
(Iteration 30 / 30) loss: 0.070897
(Epoch 15 / 15) train acc: 0.990000; val_acc: 0.218000
```

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

Small data training accuracy: 0.81

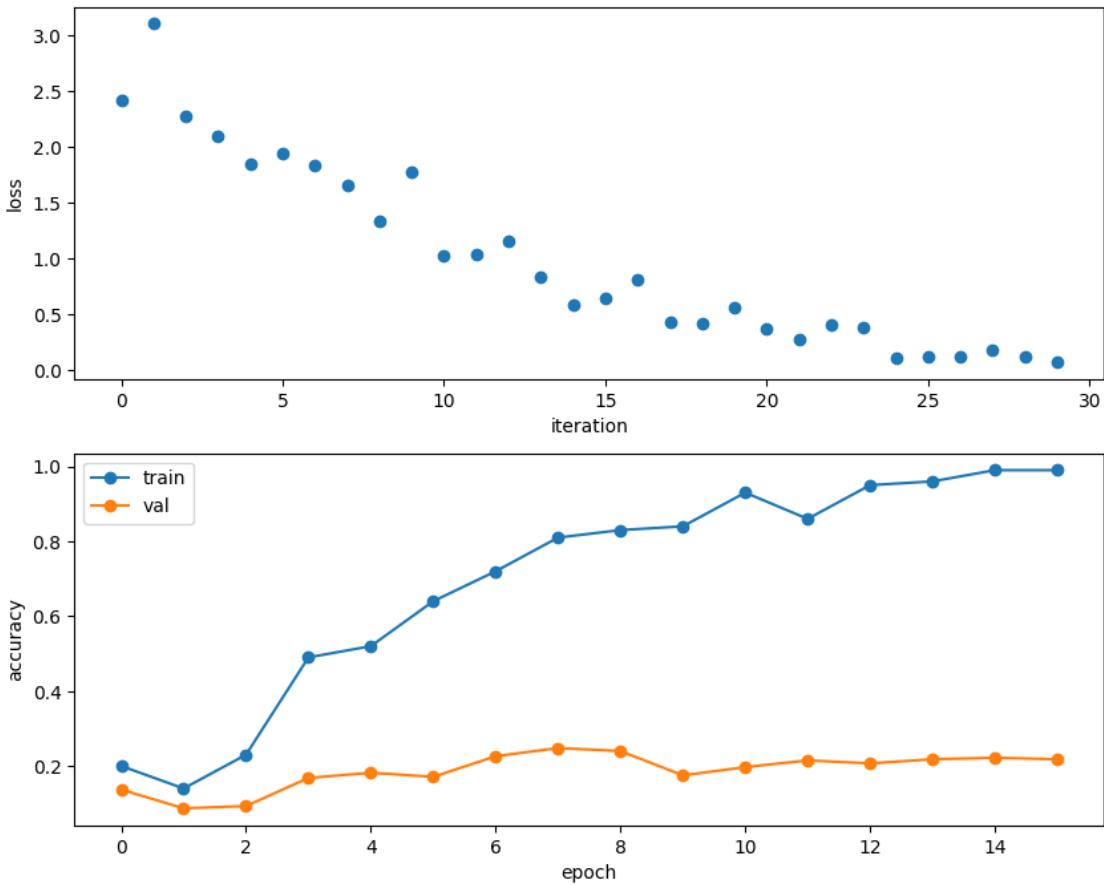
```
[17]: # Print final validation accuracy.
print(
    "Small data validation accuracy:",
    solver.check_accuracy(small_data['X_val'], small_data['y_val'])
)
```

Small data validation accuracy: 0.248

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

```
[18]: 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:

```
[19]: 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: 2.129645
(Iteration 41 / 980) loss: 1.945061
(Iteration 61 / 980) loss: 1.770560
(Iteration 81 / 980) loss: 1.901148
(Iteration 101 / 980) loss: 1.919195
(Iteration 121 / 980) loss: 1.765398
(Iteration 141 / 980) loss: 1.898770
(Iteration 161 / 980) loss: 1.738502
(Iteration 181 / 980) loss: 1.843249
(Iteration 201 / 980) loss: 2.066862
(Iteration 221 / 980) loss: 1.990459
(Iteration 241 / 980) loss: 1.828815
(Iteration 261 / 980) loss: 1.604638
(Iteration 281 / 980) loss: 1.692068
(Iteration 301 / 980) loss: 1.754931
(Iteration 321 / 980) loss: 1.742302
(Iteration 341 / 980) loss: 1.699758
(Iteration 361 / 980) loss: 1.723792
(Iteration 381 / 980) loss: 1.542635
(Iteration 401 / 980) loss: 1.738328
(Iteration 421 / 980) loss: 1.427555
(Iteration 441 / 980) loss: 1.759026
(Iteration 461 / 980) loss: 1.879609
(Iteration 481 / 980) loss: 1.413088
(Iteration 501 / 980) loss: 1.367136
(Iteration 521 / 980) loss: 1.637779
(Iteration 541 / 980) loss: 1.749722
(Iteration 561 / 980) loss: 1.645201
(Iteration 581 / 980) loss: 1.373341
(Iteration 601 / 980) loss: 1.573957
(Iteration 621 / 980) loss: 1.570977
(Iteration 641 / 980) loss: 1.683598
(Iteration 661 / 980) loss: 1.755739
(Iteration 681 / 980) loss: 1.741579
(Iteration 701 / 980) loss: 1.627704
(Iteration 721 / 980) loss: 1.655453
(Iteration 741 / 980) loss: 1.634793
(Iteration 761 / 980) loss: 1.447560
(Iteration 781 / 980) loss: 1.793672
(Iteration 801 / 980) loss: 1.627485
(Iteration 821 / 980) loss: 1.656835
(Iteration 841 / 980) loss: 1.351238
(Iteration 861 / 980) loss: 1.745392
(Iteration 881 / 980) loss: 1.552356
(Iteration 901 / 980) loss: 1.539406
(Iteration 921 / 980) loss: 1.492501
```

```
(Iteration 941 / 980) loss: 1.683674
(Iteration 961 / 980) loss: 1.638435
(Epoch 1 / 1) train acc: 0.480000; val_acc: 0.485000
```

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

```
Full data training accuracy: 0.46477551020408164
```

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

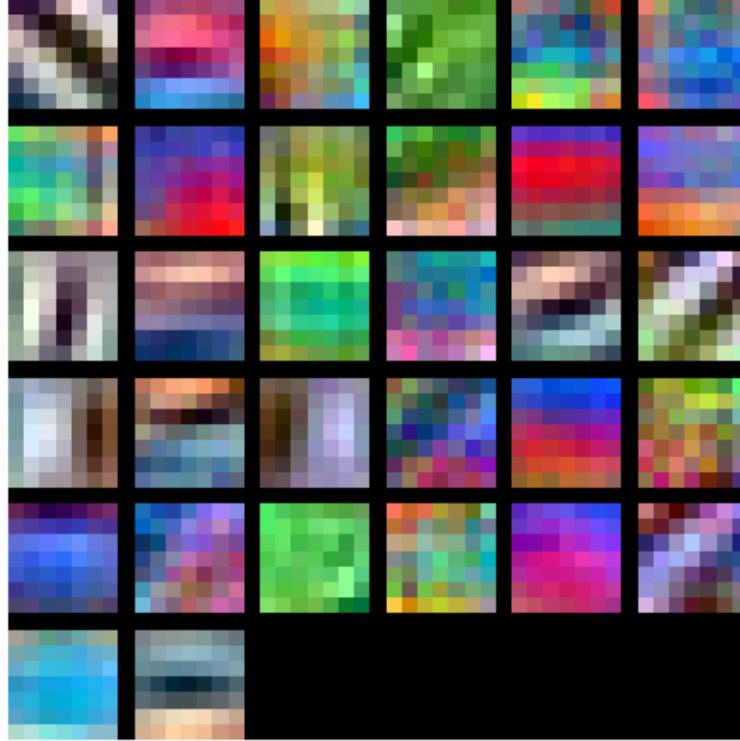
```
Full data validation accuracy: 0.485
```

## 8.5 Visualize Filters

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

```
[22]: from icv83551.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 `icv83551/layers.py`, implement the forward pass for spatial batch normalization in the function `spatial_batchnorm_forward`. Check your implementation by running the following:

```
[23]: 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]
```

```
[24]: 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 `icv83551/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:

```
[25]: 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:  2.7866481929016103e-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 `icv83551/layers.py`, implement the forward pass for group normalization in the function `spatial_groupnorm_forward`. Check your implementation by running the following:

```
[26]: 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 `icv83551/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:

```
[27]: 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:  7.413109384854475e-08
dgamma error:  9.468195772749234e-12
dbeta error:  3.354494437653335e-12

```

[ ]:

# PyTorch

February 14, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment2/'  
FOLDERNAME = 'icv83551/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/$FOLDERNAME/icv83551/datasets/  
!bash get_datasets.sh  
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 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.

### 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 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! PyTorch is an 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
<code>nn.Module</code>	High	Medium
<code>nn.Sequential</code>	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.

```
[1]: 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.

```
[2]: 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('./icv83551/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('./icv83551/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('./icv83551/datasets', train=False, download=True,
                            transform=transform)
loader_test = DataLoader(cifar10_test, batch_size=64)

```

```

C:\Users\eitan\PythonProjects\CV_projects\assignment2\.venv\Lib\site-
packages\torchvision\datasets\cifar.py:83: VisibleDeprecationWarning: dtype():
align should be passed as Python or NumPy boolean but got `align=0`. Did you
mean to pass a tuple to create a subarray type? (Deprecated NumPy 2.4)
entry = pickle.load(f, encoding="latin1")

```

## 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  $N \times C \times H \times W$ , 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 width 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  $C \times H \times W$  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  $N \times ??$ , where  $??$  is allowed to be anything (in this case, it will be  $C \times H \times W$ , but we don’t need to specify that explicitly).

```
[3]: 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
```

# view is like reshape in numpy

```
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.

```
[4]: 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 `x.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  $KW1 \times KH1$ , and zero-padding of two
2. ReLU nonlinearity
3. A convolutional layer (with bias) with `channel_2` filters, each with shape  $KW2 \times KH2$ , 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!

[5]:

```

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

*figure out what the shape should be?*

- *fc\_w*: PyTorch Tensor giving weights for the fully-connected layer. Can you figure out what the shape should be?

*figure out what the shape should be?*

*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.

```
import torch.nn.functional as F

x = F.conv2d(x, conv_w1, conv_b1, padding=2)
x = F.relu(x)
x = F.conv2d(x, conv_w2, conv_b2, padding=1)
x = F.relu(x)

scores = flatten(x).mm(fc_w) + fc_b
```

# 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).

```
[6]: 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>

```
[7]: 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))

```

[7]:

```
tensor([-0.3460, -0.1622, -1.3658, -0.5608, -0.5404],
       [-0.5307,  0.5803,  0.0968,  0.7064, -1.1839],
       [ 0.5848,  0.6081, -1.3363, -0.2554, -1.5637]], 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.

[8]:

```

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.

```
[9]: 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,  $w_1$  and  $w_2$ .

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  $w_1$ . The second dimension of  $w_1$  is the hidden layer size, which will also be the first dimension of  $w_2$ .

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.

```
[10]: 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 = 3.0904  
 Checking accuracy on the val set  
 Got 158 / 1000 correct (15.80%)

Iteration 100, loss = 2.0946  
 Checking accuracy on the val set  
 Got 343 / 1000 correct (34.30%)

Iteration 200, loss = 1.8118  
 Checking accuracy on the val set  
 Got 392 / 1000 correct (39.20%)

Iteration 300, loss = 1.6308  
 Checking accuracy on the val set  
 Got 411 / 1000 correct (41.10%)

```
Iteration 400, loss = 2.0397
Checking accuracy on the val set
Got 430 / 1000 correct (43.00%)
```

```
Iteration 500, loss = 1.6378
Checking accuracy on the val set
Got 436 / 1000 correct (43.60%)
```

```
Iteration 600, loss = 1.7244
Checking accuracy on the val set
Got 422 / 1000 correct (42.20%)
```

```
Iteration 700, loss = 1.8672
Checking accuracy on the val set
Got 446 / 1000 correct (44.60%)
```

### 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.

```
[11]: 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. #
#####
```

```

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))

fc_w = random_weight((channel_2 * 32 * 32, 10))
fc_b = zero_weight((10))

#####
#           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 = 3.2693  
 Checking accuracy on the val set  
 Got 122 / 1000 correct (12.20%)

Iteration 100, loss = 1.9244  
 Checking accuracy on the val set  
 Got 374 / 1000 correct (37.40%)

Iteration 200, loss = 1.6784  
 Checking accuracy on the val set  
 Got 406 / 1000 correct (40.60%)

Iteration 300, loss = 1.8250  
 Checking accuracy on the val set  
 Got 442 / 1000 correct (44.20%)

Iteration 400, loss = 1.6930  
 Checking accuracy on the val set  
 Got 459 / 1000 correct (45.90%)

Iteration 500, loss = 1.5247  
 Checking accuracy on the val set  
 Got 457 / 1000 correct (45.70%)

Iteration 600, loss = 1.5537  
 Checking accuracy on the val set  
 Got 458 / 1000 correct (45.80%)

Iteration 700, loss = 1.6399  
 Checking accuracy on the val set  
 Got 490 / 1000 correct (49.00%)

## 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.                                         #
        #####
        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2)
        nn.init.kaiming_normal_(self.conv1.weight)
        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1)
        nn.init.kaiming_normal_(self.conv2.weight)
        self.fc = nn.Linear(channel_2 * 32 * 32, num_classes) # 32 * 32 is the ↴
        ↴size of pic in the data set
        nn.init.kaiming_normal_(self.fc.weight)

        #####
        #                                     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()                          #
#####
x = F.relu(self.conv1(x))
x = F.relu(self.conv2(x))
scores = self.fc(flatten(x))

#####
#                                     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 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.

```
[19]: 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 = 2.9693
Checking accuracy on validation set
Got 143 / 1000 correct (14.30)
```

```
Iteration 100, loss = 3.0203
Checking accuracy on validation set
Got 295 / 1000 correct (29.50)
```

```
Iteration 200, loss = 2.2661
Checking accuracy on validation set
Got 374 / 1000 correct (37.40)
```

```
Iteration 300, loss = 2.1661
Checking accuracy on validation set
Got 421 / 1000 correct (42.10)
```

```
Iteration 400, loss = 1.5049
Checking accuracy on validation set
Got 399 / 1000 correct (39.90)
```

```
Iteration 500, loss = 1.8559
```

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

```
Iteration 600, loss = 1.6056
Checking accuracy on validation set
Got 438 / 1000 correct (43.80)
```

```
Iteration 700, loss = 1.6415
Checking accuracy on validation set
Got 444 / 1000 correct (44.40)
```

### 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 45% after training for one epoch.

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

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

model = None
optimizer = None
#####
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
#####
model = ThreeLayerConvNet(in_channel=3, channel_1=channel_1, ↴
    ↴channel_2=channel_2, num_classes=10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)
#
#                                     END OF YOUR CODE
#
#####
train_part34(model, optimizer)
```

```
Iteration 0, loss = 4.1217
Checking accuracy on validation set
Got 139 / 1000 correct (13.90)
```

```
Iteration 100, loss = 1.8345
Checking accuracy on validation set
Got 374 / 1000 correct (37.40)
```

```
Iteration 200, loss = 1.7974
Checking accuracy on validation set
Got 441 / 1000 correct (44.10)
```

```

Iteration 300, loss = 1.5292
Checking accuracy on validation set
Got 463 / 1000 correct (46.30)

Iteration 400, loss = 1.7957
Checking accuracy on validation set
Got 450 / 1000 correct (45.00)

Iteration 500, loss = 1.5112
Checking accuracy on validation set
Got 477 / 1000 correct (47.70)

Iteration 600, loss = 1.3447
Checking accuracy on validation set
Got 501 / 1000 correct (50.10)

Iteration 700, loss = 1.4380
Checking accuracy on validation set
Got 484 / 1000 correct (48.40)

```

## 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: sub-class `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.

```
[21]: # 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 = 2.3489  
 Checking accuracy on validation set  
 Got 131 / 1000 correct (13.10)

Iteration 100, loss = 1.6678  
 Checking accuracy on validation set  
 Got 412 / 1000 correct (41.20)

Iteration 200, loss = 1.8556  
 Checking accuracy on validation set  
 Got 446 / 1000 correct (44.60)

Iteration 300, loss = 1.5760  
 Checking accuracy on validation set  
 Got 428 / 1000 correct (42.80)

Iteration 400, loss = 1.7055  
 Checking accuracy on validation set  
 Got 462 / 1000 correct (46.20)

Iteration 500, loss = 1.9534  
 Checking accuracy on validation set  
 Got 430 / 1000 correct (43.00)

Iteration 600, loss = 2.0223  
 Checking accuracy on validation set  
 Got 423 / 1000 correct (42.30)

Iteration 700, loss = 1.4950  
 Checking accuracy on validation set  
 Got 440 / 1000 correct (44.00)

### 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.

```
[22]: 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.                                                       #
#####

model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2 * 32 * 32, 10),
)

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

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

train_part34(model, optimizer)
```

```
Iteration 0, loss = 2.2977
Checking accuracy on validation set
```

```
Got 101 / 1000 correct (10.10)

Iteration 100, loss = 1.6119
Checking accuracy on validation set
Got 443 / 1000 correct (44.30)

Iteration 200, loss = 1.2445
Checking accuracy on validation set
Got 471 / 1000 correct (47.10)

Iteration 300, loss = 1.3949
Checking accuracy on validation set
Got 540 / 1000 correct (54.00)

Iteration 400, loss = 1.4373
Checking accuracy on validation set
Got 552 / 1000 correct (55.20)

Iteration 500, loss = 1.3801
Checking accuracy on validation set
Got 554 / 1000 correct (55.40)

Iteration 600, loss = 1.0102
Checking accuracy on validation set
Got 589 / 1000 correct (58.90)

Iteration 700, loss = 1.1556
Checking accuracy on validation set
Got 588 / 1000 correct (58.80)
```

## 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!

```
[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  
  
# standard CIFAR-10 constants  
channel_1 = 64  
channel_2 = 128  
channel_3 = 256  
num_classes = 10  
  
model = nn.Sequential(  
    # Block 1: Input 3x32x32 -> Output 64x32x32  
    nn.Conv2d(3, channel_1, kernel_size=3, padding=1),  
    nn.BatchNorm2d(channel_1),  
    nn.ReLU(),  
    nn.Conv2d(channel_1, channel_1, kernel_size=3, padding=1),  
    nn.BatchNorm2d(channel_1),  
    nn.ReLU(),  
    nn.MaxPool2d(2), # -> 64x16x16  
  
    # Block 2: 64x16x16 -> 128x16x16  
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),  
    nn.BatchNorm2d(channel_2),  
    nn.ReLU(),  
    nn.Conv2d(channel_2, channel_2, kernel_size=3, padding=1),  
    nn.BatchNorm2d(channel_2),  
    nn.ReLU(),  
    nn.MaxPool2d(2), # -> 128x8x8  
  
    # Block 3: 128x8x8 -> 256x8x8  
    nn.Conv2d(channel_2, channel_3, kernel_size=3, padding=1),  
    nn.BatchNorm2d(channel_3),  
    nn.ReLU(),  
    nn.MaxPool2d(2), # -> 256x4x4
```

```

# Classifier
nn.Flatten(),
nn.Linear(channel_3 * 4 * 4, 512),
nn.ReLU(),
nn.Dropout(0.5), # prevents overfitting
nn.Linear(512, num_classes)
)

# use Adam optimizer for faster convergence (standard for CS231n)
optimizer = optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)

# move model to device
model = model.to(device=device)

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

# You should get at least 70% accuracy
train_part34(model, optimizer, epochs=10)

```

Iteration 0, loss = 2.4222  
 Checking accuracy on validation set  
 Got 113 / 1000 correct (11.30)

Iteration 100, loss = 1.4987  
 Checking accuracy on validation set  
 Got 356 / 1000 correct (35.60)

Iteration 200, loss = 1.4580  
 Checking accuracy on validation set  
 Got 371 / 1000 correct (37.10)

Iteration 300, loss = 1.6393  
 Checking accuracy on validation set  
 Got 471 / 1000 correct (47.10)

Iteration 400, loss = 1.3696  
 Checking accuracy on validation set  
 Got 544 / 1000 correct (54.40)

Iteration 500, loss = 1.4181  
 Checking accuracy on validation set  
 Got 591 / 1000 correct (59.10)

Iteration 600, loss = 1.3253  
 Checking accuracy on validation set  
 Got 535 / 1000 correct (53.50)

Iteration 700, loss = 1.1050  
Checking accuracy on validation set  
Got 549 / 1000 correct (54.90)

Iteration 0, loss = 1.2077  
Checking accuracy on validation set  
Got 648 / 1000 correct (64.80)

Iteration 100, loss = 1.0171  
Checking accuracy on validation set  
Got 631 / 1000 correct (63.10)

Iteration 200, loss = 1.0140  
Checking accuracy on validation set  
Got 681 / 1000 correct (68.10)

Iteration 300, loss = 0.7815  
Checking accuracy on validation set  
Got 686 / 1000 correct (68.60)

Iteration 400, loss = 1.1257  
Checking accuracy on validation set  
Got 685 / 1000 correct (68.50)

Iteration 500, loss = 0.8844  
Checking accuracy on validation set  
Got 678 / 1000 correct (67.80)

Iteration 600, loss = 1.0872  
Checking accuracy on validation set  
Got 650 / 1000 correct (65.00)

Iteration 700, loss = 1.0373  
Checking accuracy on validation set  
Got 716 / 1000 correct (71.60)

Iteration 0, loss = 0.9623  
Checking accuracy on validation set  
Got 733 / 1000 correct (73.30)

Iteration 100, loss = 1.1034  
Checking accuracy on validation set  
Got 684 / 1000 correct (68.40)

Iteration 200, loss = 0.9244  
Checking accuracy on validation set  
Got 739 / 1000 correct (73.90)

Iteration 300, loss = 1.0822  
Checking accuracy on validation set  
Got 730 / 1000 correct (73.00)

Iteration 400, loss = 0.8029  
Checking accuracy on validation set  
Got 757 / 1000 correct (75.70)

Iteration 500, loss = 0.7998  
Checking accuracy on validation set  
Got 725 / 1000 correct (72.50)

Iteration 600, loss = 0.7436  
Checking accuracy on validation set  
Got 737 / 1000 correct (73.70)

Iteration 700, loss = 0.7981  
Checking accuracy on validation set  
Got 762 / 1000 correct (76.20)

Iteration 0, loss = 0.7203  
Checking accuracy on validation set  
Got 650 / 1000 correct (65.00)

Iteration 100, loss = 0.6340  
Checking accuracy on validation set  
Got 711 / 1000 correct (71.10)

Iteration 200, loss = 0.6607  
Checking accuracy on validation set  
Got 768 / 1000 correct (76.80)

Iteration 300, loss = 0.8445  
Checking accuracy on validation set  
Got 748 / 1000 correct (74.80)

Iteration 400, loss = 0.5530  
Checking accuracy on validation set  
Got 755 / 1000 correct (75.50)

Iteration 500, loss = 0.7711  
Checking accuracy on validation set  
Got 764 / 1000 correct (76.40)

Iteration 600, loss = 0.6899  
Checking accuracy on validation set  
Got 731 / 1000 correct (73.10)

Iteration 700, loss = 0.4930  
Checking accuracy on validation set  
Got 742 / 1000 correct (74.20)

Iteration 0, loss = 0.6423  
Checking accuracy on validation set  
Got 757 / 1000 correct (75.70)

Iteration 100, loss = 0.5573  
Checking accuracy on validation set  
Got 738 / 1000 correct (73.80)

Iteration 200, loss = 0.5182  
Checking accuracy on validation set  
Got 799 / 1000 correct (79.90)

Iteration 300, loss = 0.6467  
Checking accuracy on validation set  
Got 791 / 1000 correct (79.10)

Iteration 400, loss = 0.4857  
Checking accuracy on validation set  
Got 787 / 1000 correct (78.70)

Iteration 500, loss = 0.4607  
Checking accuracy on validation set  
Got 801 / 1000 correct (80.10)

Iteration 600, loss = 0.5305  
Checking accuracy on validation set  
Got 783 / 1000 correct (78.30)

Iteration 700, loss = 0.3754  
Checking accuracy on validation set  
Got 787 / 1000 correct (78.70)

Iteration 0, loss = 0.3930  
Checking accuracy on validation set  
Got 808 / 1000 correct (80.80)

Iteration 100, loss = 0.6781  
Checking accuracy on validation set  
Got 765 / 1000 correct (76.50)

Iteration 200, loss = 0.6744  
Checking accuracy on validation set  
Got 763 / 1000 correct (76.30)

```
Iteration 300, loss = 0.6279
Checking accuracy on validation set
Got 800 / 1000 correct (80.00)
```

```
Iteration 400, loss = 0.7779
Checking accuracy on validation set
Got 807 / 1000 correct (80.70)
```

```
Iteration 500, loss = 0.6186
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)
```

```
Iteration 600, loss = 0.6050
Checking accuracy on validation set
Got 798 / 1000 correct (79.80)
```

```
Iteration 700, loss = 0.4545
Checking accuracy on validation set
Got 806 / 1000 correct (80.60)
```

```
Iteration 0, loss = 0.3453
Checking accuracy on validation set
Got 810 / 1000 correct (81.00)
```

```
Iteration 100, loss = 0.3809
Checking accuracy on validation set
Got 821 / 1000 correct (82.10)
```

```
Iteration 200, loss = 0.4856
Checking accuracy on validation set
Got 822 / 1000 correct (82.20)
```

```
Iteration 300, loss = 0.4946
Checking accuracy on validation set
Got 816 / 1000 correct (81.60)
```

```
Iteration 400, loss = 0.2946
Checking accuracy on validation set
Got 834 / 1000 correct (83.40)
```

```
Iteration 500, loss = 0.5299
Checking accuracy on validation set
Got 830 / 1000 correct (83.00)
```

```
Iteration 600, loss = 0.3767
Checking accuracy on validation set
Got 806 / 1000 correct (80.60)
```

Iteration 700, loss = 0.4709  
Checking accuracy on validation set  
Got 835 / 1000 correct (83.50)

Iteration 0, loss = 0.2507  
Checking accuracy on validation set  
Got 804 / 1000 correct (80.40)

Iteration 100, loss = 0.3515  
Checking accuracy on validation set  
Got 810 / 1000 correct (81.00)

Iteration 200, loss = 0.3032  
Checking accuracy on validation set  
Got 844 / 1000 correct (84.40)

Iteration 300, loss = 0.5132  
Checking accuracy on validation set  
Got 825 / 1000 correct (82.50)

Iteration 400, loss = 0.3908  
Checking accuracy on validation set  
Got 821 / 1000 correct (82.10)

Iteration 500, loss = 0.4308  
Checking accuracy on validation set  
Got 782 / 1000 correct (78.20)

Iteration 600, loss = 0.4747  
Checking accuracy on validation set  
Got 832 / 1000 correct (83.20)

Iteration 700, loss = 0.6704  
Checking accuracy on validation set  
Got 826 / 1000 correct (82.60)

Iteration 0, loss = 0.4046  
Checking accuracy on validation set  
Got 813 / 1000 correct (81.30)

Iteration 100, loss = 0.2543  
Checking accuracy on validation set  
Got 835 / 1000 correct (83.50)

Iteration 200, loss = 0.3108  
Checking accuracy on validation set  
Got 849 / 1000 correct (84.90)

Iteration 300, loss = 0.4130  
Checking accuracy on validation set  
Got 826 / 1000 correct (82.60)

Iteration 400, loss = 0.4278  
Checking accuracy on validation set  
Got 852 / 1000 correct (85.20)

Iteration 500, loss = 0.2826  
Checking accuracy on validation set  
Got 840 / 1000 correct (84.00)

Iteration 600, loss = 0.3253  
Checking accuracy on validation set  
Got 807 / 1000 correct (80.70)

Iteration 700, loss = 0.6313  
Checking accuracy on validation set  
Got 842 / 1000 correct (84.20)

Iteration 0, loss = 0.2269  
Checking accuracy on validation set  
Got 838 / 1000 correct (83.80)

Iteration 100, loss = 0.2599  
Checking accuracy on validation set  
Got 858 / 1000 correct (85.80)

Iteration 200, loss = 0.2941  
Checking accuracy on validation set  
Got 833 / 1000 correct (83.30)

Iteration 300, loss = 0.2935  
Checking accuracy on validation set  
Got 831 / 1000 correct (83.10)

Iteration 400, loss = 0.1603  
Checking accuracy on validation set  
Got 839 / 1000 correct (83.90)

Iteration 500, loss = 0.3093  
Checking accuracy on validation set  
Got 868 / 1000 correct (86.80)

Iteration 600, loss = 0.2451  
Checking accuracy on validation set  
Got 857 / 1000 correct (85.70)

```
Iteration 700, loss = 0.1732
Checking accuracy on validation set
Got 849 / 1000 correct (84.90)
```

## 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:** For this assignment, I designed and trained a **Convolutional Neural Network (CNN)** aimed at achieving high accuracy on the CIFAR-10 dataset within a limited budget of 10 epochs. Below is a detailed breakdown of the architecture and optimization strategies used.

### 8.1.1 1. Model Architecture: VGG-style Deep CNN

I implemented a modular architecture inspired by the **VGG** design, characterized by repeating blocks of  $3 \times 3$  convolutions and spatial downsampling.

- **Convolutional Blocks:** The model is composed of three main stages with increasing filter depths ( $64 \rightarrow 128 \rightarrow 256$ ). Each block uses a kernel size of 3 with padding of 1 to preserve spatial dimensions during the convolution operation.
  - **Spatial Downsampling:** I utilized `nn.MaxPool2d(2)` layers at the end of each block. This progressively reduces the spatial resolution (from  $32 \times 32 \rightarrow 16 \times 16 \rightarrow 8 \times 8 \rightarrow 4 \times 4$ ), effectively increasing the receptive field and reducing computational cost.
  - **Fully Connected Classifier:** After flattening the final convolutional output, I used a hidden dense layer with 512 units followed by a Dropout layer to transition into the final 10-class output.
- 

### 8.1.2 2. Training Enhancements & Regularization

To ensure the model hit the required **70% validation accuracy** efficiently, I incorporated several modern deep learning components:

- **Batch Normalization:** Every convolutional layer is followed by a `BatchNorm2d` layer. This was critical for stabilizing the hidden layer distributions, allowing for faster convergence and providing a slight regularizing effect.
  - **Dropout:** I implemented a `Dropout(0.5)` layer in the classifier. This forces the network to learn redundant representations and prevents it from over-relying on specific neurons, which significantly reduced overfitting during the 10-epoch training window.
  - **Non-Linearity:** ReLU activation functions were used throughout the network to maintain gradient flow and introduce non-linearity.
-

### 8.1.3 3. Optimizer and Hyperparameters

- **Optimizer:** I chose the **Adam optimizer** (`lr=1e-3`) instead of standard SGD. Adam's adaptive learning rate properties are particularly effective for reaching high accuracy quickly in limited epoch settings.
- **Weight Decay:** I added a weight decay of `1e-4` ( $L_2$  regularization) to the optimizer to further penalize large weights and improve generalization on the validation set.
- **Training Schedule:** The model was trained for 10 epochs using the `train_part34` utility, consistently monitoring validation accuracy to ensure the model surpassed the 70% threshold without touching the test set.

## 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 8419 / 10000 correct (84.19)
```

```
[ ]:
```

# RNN\_Captioning\_pytorch

February 14, 2026

```
[ ]: # 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. 'icv83551/assignments/assignment2/'
FOLDERNAME = "icv83551/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 COCO dataset to your Drive
# if it doesn't already exist.
%cd /content/drive/My\ Drive/$FOLDERNAME/icv83551/datasets/
!bash get_coco_dataset.sh
%cd /content/drive/My\ Drive/$FOLDERNAME
```

## 1 Image Captioning with RNNs

In this exercise, you will implement vanilla Recurrent Neural Networks and use them to train a model that can generate novel captions for images.

```
[21]: # Setup cell.
import time, os, json
import numpy as np
import torch
import matplotlib.pyplot as plt

from icv83551.gradient_check import eval_numerical_gradient, eval_numerical_gradient_array
from icv83551.rnn_layers_pytorch import *
from icv83551.captioning_solver_pytorch import CaptioningSolverPytorch
from icv83551.classifiers.rnn_pytorch import CaptioningRNN
```

```

from icv83551.coco_utils import load_coco_data, sample_coco_minibatch, decode_captions
from icv83551.image_utils import image_from_url

%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))))

```

## 2 COCO Dataset

For this exercise, we will use the 2014 release of the [COCO dataset](#), a standard testbed for image captioning. The dataset consists of 80,000 training images and 40,000 validation images, each annotated with 5 captions written by workers on Amazon Mechanical Turk.

**Image features.** We have preprocessed the data and extracted features for you already. For all images, we have extracted features from the fc7 layer of the VGG-16 network pretrained on ImageNet, and these features are stored in the files `train2014_vgg16_fc7.h5` and `val2014_vgg16_fc7.h5`. To cut down on processing time and memory requirements, we have reduced the dimensionality of the features from 4096 to 512 using Principal Component Analysis (PCA), and these features are stored in the files `train2014_vgg16_fc7_pca.h5` and `val2014_vgg16_fc7_pca.h5`. The raw images take up nearly 20GB of space so we have not included them in the download. Since all images are taken from Flickr, we have stored the URLs of the training and validation images in the files `train2014_urls.txt` and `val2014_urls.txt`. This allows you to download images on-the-fly for visualization.

**Captions.** Dealing with strings is inefficient, so we will work with an encoded version of the captions. Each word is assigned an integer ID, allowing us to represent a caption by a sequence of integers. The mapping between integer IDs and words is in the file `coco2014_vocab.json`, and you can use the function `decode_captions` from the file `icv83551/coco_utils.py` to convert NumPy arrays of integer IDs back into strings.

**Tokens.** There are a couple special tokens that we add to the vocabulary, and we have taken care of all implementation details around special tokens for you. We prepend a special `<START>` token and append an `<END>` token to the beginning and end of each caption respectively. Rare words are replaced with a special `<UNK>` token (for “unknown”). In addition, since we want to train with minibatches containing captions of different lengths, we pad short captions with a special `<NULL>` token after the `<END>` token and don’t compute loss or gradient for `<NULL>` tokens.

You can load all of the COCO data (captions, features, URLs, and vocabulary) using the `load_coco_data` function from the file `icv83551/coco_utils.py`. Run the following cell to do so:

```
[2]: # Load COCO data from disk into a dictionary.
# We'll work with dimensionality-reduced features for the remainder of this assignment,
# but you can also experiment with the original features on your own by changing the flag below.
data = load_coco_data(pca_features=True)

# Print out all the keys and values from the data dictionary.
for k, v in data.items():
    if type(v) == np.ndarray:
        print(k, type(v), v.shape, v.dtype)
    else:
        print(k, type(v), len(v))

base dir  C:\Users\eitan\PythonProjects\CV_projects\assignment2\icv83551\datasets\coco_captioning
train_captions <class 'numpy.ndarray'> (400135, 17) int32
train_image_idxs <class 'numpy.ndarray'> (400135,) int32
val_captions <class 'numpy.ndarray'> (195954, 17) int32
val_image_idxs <class 'numpy.ndarray'> (195954,) int32
train_features <class 'numpy.ndarray'> (82783, 512) float32
val_features <class 'numpy.ndarray'> (40504, 512) float32
idx_to_word <class 'list'> 1004
word_to_idx <class 'dict'> 1004
train_urls <class 'numpy.ndarray'> (82783,) <U63
val_urls <class 'numpy.ndarray'> (40504,) <U63
```

## 2.1 Inspect the Data

It is always a good idea to look at examples from the dataset before working with it.

You can use the `sample_coco_minibatch` function from the file `icv83551/coco_utils.py` to sample minibatches of data from the data structure returned from `load_coco_data`. Run the following to sample a small minibatch of training data and show the images and their captions. Running it multiple times and looking at the results helps you to get a sense of the dataset.

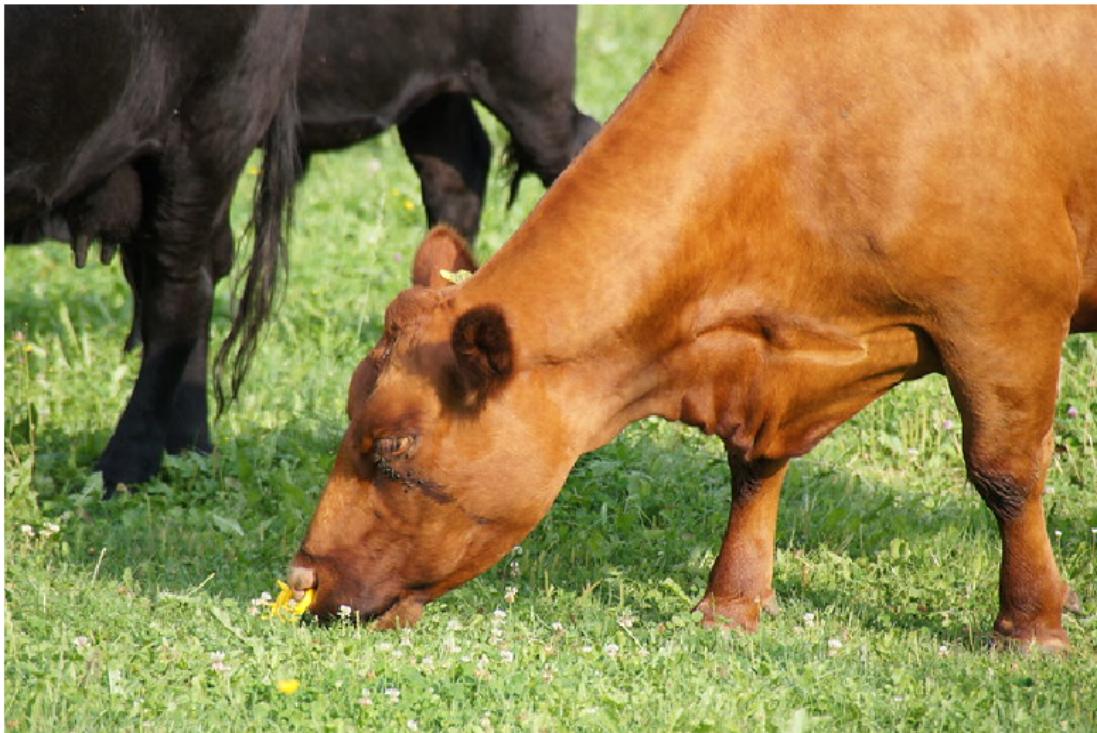
```
[3]: # Sample a minibatch and show the images and captions.
# If you get an error, the URL just no longer exists, so don't worry!
# You can re-sample as many times as you want.
batch_size = 3

captions, features, urls = sample_coco_minibatch(data, batch_size=batch_size)
for i, (caption, url) in enumerate(zip(captions, urls)):
    plt.imshow(image_from_url(url))
    plt.axis('off')
    caption_str = decodeCaptions(caption, data['idx_to_word'])
    plt.title(caption_str)
    plt.show()
```

<START> two bears dressed up one as a <UNK> and one as an <UNK> <END>



<START> an image of a <UNK> that is grazing <END>



<START> the clock is in front of the building in the city <END>



### 3 Recurrent Neural Network

As discussed in lecture, we will use Recurrent Neural Network (RNN) language models for image captioning. The file `icv83551/rnn_layers_pytorch.py` contains implementations of different layer types that are needed for recurrent neural networks, and the file `icv83551/classifiers/rnn_pytorch.py` uses these layers to implement an image captioning model.

We will first implement different types of RNN layers in `icv83551/rnn_layers_pytorch.py`.

### 4 Vanilla RNN: Step Forward

Open the file `icv83551/rnn_layers_pytorch.py`. This file implements the forward passes for different types of layers that are commonly used in recurrent neural networks. Note that since we use pytorch, the backward pass will be handled by pytorch's autograd.

First implement the function `rnn_step_forward` which implements the forward pass for a single timestep of a vanilla recurrent neural network. After doing so run the following to check your implementation. You should see errors on the order of e-8 or less.

[4]: N, D, H = 3, 10, 4

```
x = torch.from_numpy(np.linspace(-0.4, 0.7, num=N*D).reshape(N, D))
prev_h = torch.from_numpy(np.linspace(-0.2, 0.5, num=N*H).reshape(N, H))
Wx = torch.from_numpy(np.linspace(-0.1, 0.9, num=D*H).reshape(D, H))
Wh = torch.from_numpy(np.linspace(-0.3, 0.7, num=H*H).reshape(H, H))
b = torch.from_numpy(np.linspace(-0.2, 0.4, num=H))

next_h = rnn_step_forward(x, prev_h, Wx, Wh, b).numpy()
expected_next_h = np.asarray([
    [-0.58172089, -0.50182032, -0.41232771, -0.31410098],
    [ 0.66854692,  0.79562378,  0.87755553,  0.92795967],
    [ 0.97934501,  0.99144213,  0.99646691,  0.99854353]]))

print('next_h error: ', rel_error(expected_next_h, next_h))
```

```
next_h error:  6.292421426471037e-09
```

### 5 Vanilla RNN: Step Backward

Since we implemented `rnn_step_forward` with pytorch, we do NOT have to implement `rnn_step_backward`. We can verify pytorch autograd backward pass using our numerical gradient checker.

However, if you are feeling adventurous, you can try to implement `rnn_step_backward` yourself. It is not required in this assignment though.

```
[5]: from icv83551.rnn_layers_pytorch import rnn_step_forward

# Create test inputs
np.random.seed(231)
N, D, H = 4, 5, 6
x = torch.from_numpy(np.random.randn(N, D))
h = torch.from_numpy(np.random.randn(N, H))
Wx = torch.from_numpy(np.random.randn(D, H))
Wh = torch.from_numpy(np.random.randn(H, H))
b = torch.from_numpy(np.random.randn(H))

# Enable gradient tracking and do rnn forward pass
for tensor in [x, h, Wx, Wh, b]:
    tensor.requires_grad_()
next_h = rnn_step_forward(x, h, Wx, Wh, b)

# Simulate random upstream gradients and do a backward pass using pytorch's
# autograd.
dnext_h = torch.from_numpy(np.random.randn(*next_h.shape))
next_h.backward(dnext_h)

# Collect gradient in separate numpy arrays
dx = x.grad.detach().numpy()
dh = h.grad.detach().numpy()
dWx = Wx.grad.detach().numpy()
dWh = Wh.grad.detach().numpy()
db = b.grad.detach().numpy()
dnext_h = dnext_h.detach().numpy()

# Also convert test inputs to numpy arrays
x = x.detach().numpy()
h = h.detach().numpy()
Wx = Wx.detach().numpy()
Wh = Wh.detach().numpy()
b = b.detach().numpy()

# Wrap our forward pass to support numpy array input and output. We use
# `torch.no_grad()` to explicitly disable gradient tracking.
def rnn_step_forward_numpy(x, h, Wx, Wh, b):
    with torch.no_grad():
        return rnn_step_forward(
            torch.from_numpy(x),
            torch.from_numpy(h),
            torch.from_numpy(Wx),
            torch.from_numpy(Wh),
            torch.from_numpy(b),
        ).numpy()
```

```

# Compute numerical gradients and compare.
fx = lambda x: rnn_step_forward_numpy(x, h, Wx, Wh, b)
fh = lambda h: rnn_step_forward_numpy(x, h, Wx, Wh, b)
fWx = lambda Wx: rnn_step_forward_numpy(x, h, Wx, Wh, b)
fWh = lambda Wh: rnn_step_forward_numpy(x, h, Wx, Wh, b)
fb = lambda b: rnn_step_forward_numpy(x, h, Wx, Wh, b)

dx_num = eval_numerical_gradient_array(fx, x, dnext_h)
dh_num = eval_numerical_gradient_array(fh, h, dnext_h)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dnext_h)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dnext_h)
db_num = eval_numerical_gradient_array(fb, b, dnext_h)

# You should see errors on the order of 1e-9 or less
print('dx error: ', rel_error(dx_num, dx))
print('dh error: ', rel_error(dh_num, dh))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))

```

```

dx error: 2.319932372313319e-10
dh error: 2.6828355645784327e-10
dWx error: 8.820244454238703e-10
dWh error: 4.703287554560559e-10
db error: 1.595689552627225e-11

```

## 6 Vanilla RNN: Forward

Now that you have implemented the forward for a single timestep of a vanilla RNN, you will use it to implement a RNN that processes an entire sequence of data.

In the file `icv83551/rnn_layers_pytorch.py`, implement the function `rnn_forward`. This should be implemented using the `rnn_step_forward` function that you defined above. After doing so run the following to check your implementation. You should see errors on the order of `e-7` or less.

```

[6]: from icv83551.rnn_layers_pytorch import rnn_forward

N, T, D, H = 2, 3, 4, 5

x = torch.from_numpy(np.linspace(-0.1, 0.3, num=N*T*D).reshape(N, T, D))
h0 = torch.from_numpy(np.linspace(-0.3, 0.1, num=N*H).reshape(N, H))
Wx = torch.from_numpy(np.linspace(-0.2, 0.4, num=D*H).reshape(D, H))
Wh = torch.from_numpy(np.linspace(-0.4, 0.1, num=H*H).reshape(H, H))
b = torch.from_numpy(np.linspace(-0.7, 0.1, num=H))

h = rnn_forward(x, h0, Wx, Wh, b).numpy()
expected_h = np.asarray([

```

```

[
  [-0.42070749, -0.27279261, -0.11074945,  0.05740409,  0.22236251],
  [-0.39525808, -0.22554661, -0.0409454,   0.14649412,  0.32397316],
  [-0.42305111, -0.24223728, -0.04287027,  0.15997045,  0.35014525],
],
[
  [-0.55857474, -0.39065825, -0.19198182,  0.02378408,  0.23735671],
  [-0.27150199, -0.07088804,  0.13562939,  0.33099728,  0.50158768],
  [-0.51014825, -0.30524429, -0.06755202,  0.17806392,  0.40333043]]])
print('h error: ', rel_error(expected_h, h))

```

h error: 7.728466180186066e-08

## 7 Vanilla RNN: Backward

As before, we can verify pytorch autograd backward pass using our numerical gradient checker. You can also you can try to implement `rnn_step_backward` yourself if you want. It is not required in this assignment though.

```
[7]: from icv83551.rnn_layers_pytorch import rnn_forward

# Create test inputs
np.random.seed(231)
N, D, T, H = 2, 3, 10, 5
x = torch.from_numpy(np.random.randn(N, T, D))
h0 = torch.from_numpy(np.random.randn(N, H))
Wx = torch.from_numpy(np.random.randn(D, H))
Wh = torch.from_numpy(np.random.randn(H, H))
b = torch.from_numpy(np.random.randn(H))

# Enable gradient tracking and do forward pass
for tensor in [x, h0, Wx, Wh, b]:
    tensor.requires_grad_()
h = rnn_forward(x, h0, Wx, Wh, b)

# Simulate random upstream gradients and do a backward pass using pytorch's
# autograd.
dh = torch.from_numpy(np.random.randn(*h.shape))
h.backward(dh)

# Collect gradient in separate numpy arrays
dx = x.grad.detach().numpy()
dh0 = h0.grad.detach().numpy()
dWx = Wx.grad.detach().numpy()
dWh = Wh.grad.detach().numpy()
db = b.grad.detach().numpy()
dh = dh.detach().numpy()
```

```

# Also convert test inputs to numpy arrays
x = x.detach().numpy()
h0 = h0.detach().numpy()
Wx = Wx.detach().numpy()
Wh = Wh.detach().numpy()
b = b.detach().numpy()

# Wrap our forward pass to support numpy array input and output. We use
# `torch.no_grad()` to explicitly disable gradient tracking.
def rnn_forward_numpy(x, h0, Wx, Wh, b):
    with torch.no_grad():
        return rnn_forward(
            torch.from_numpy(x),
            torch.from_numpy(h0),
            torch.from_numpy(Wx),
            torch.from_numpy(Wh),
            torch.from_numpy(b),
        ).numpy()

fx = lambda x: rnn_forward_numpy(x, h0, Wx, Wh, b)
fh0 = lambda h0: rnn_forward_numpy(x, h0, Wx, Wh, b)
fWx = lambda Wx: rnn_forward_numpy(x, h0, Wx, Wh, b)
fWh = lambda Wh: rnn_forward_numpy(x, h0, Wx, Wh, b)
fb = lambda b: rnn_forward_numpy(x, h0, Wx, Wh, b)

dx_num = eval_numerical_gradient_array(fx, x, dh)
dh0_num = eval_numerical_gradient_array(fh0, h0, dh)
dWx_num = eval_numerical_gradient_array(fWx, Wx, dh)
dWh_num = eval_numerical_gradient_array(fWh, Wh, dh)
db_num = eval_numerical_gradient_array(fb, b, dh)

# You should see errors on the order of 1e-6 or less
print('dx error: ', rel_error(dx_num, dx))
print('dh0 error: ', rel_error(dh0_num, dh0))
print('dWx error: ', rel_error(dWx_num, dWx))
print('dWh error: ', rel_error(dWh_num, dWh))
print('db error: ', rel_error(db_num, db))

```

```

dx error: 2.4180851680932135e-09
dh0 error: 3.3811400640982614e-09
dWx error: 7.2217238007937085e-09
dWh error: 1.2821246475483426e-07
db error: 4.676348457243789e-10

```

## 8 Word Embedding: Forward

In deep learning systems, we commonly represent words using vectors. Each word of the vocabulary will be associated with a vector, and these vectors will be learned jointly with the rest of the system.

In the file `icv83551/rnn_layers_pytorch.py`, implement the function `word_embedding_forward` to convert words (represented by integers) into vectors. Run the following to check your implementation. You should see an error on the order of `e-8` or less.

```
[8]: N, T, V, D = 2, 4, 5, 3

x = torch.from_numpy(np.asarray([[0, 3, 1, 2], [2, 1, 0, 3]]))
W = torch.from_numpy(np.linspace(0, 1, num=V*D).reshape(V, D))

out = word_embedding_forward(x, W).numpy()
expected_out = np.asarray([
    [[ 0.,          0.07142857,  0.14285714],
     [ 0.64285714,  0.71428571,  0.78571429],
     [ 0.21428571,  0.28571429,  0.35714286],
     [ 0.42857143,  0.5,        0.57142857]],
    [[ 0.42857143,  0.5,        0.57142857],
     [ 0.21428571,  0.28571429,  0.35714286],
     [ 0.,          0.07142857,  0.14285714],
     [ 0.64285714,  0.71428571,  0.78571429]]])

print('out error: ', rel_error(expected_out, out))
```

```
out error:  1.000000094736443e-08
```

## 9 Word Embedding: Backward

As before, we can verify pytorch autograd backward pass using our numerical gradient checker. You can also you can try to implement `word_embedding_backward` yourself if you want. It is not required in this assignment though.

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

N, T, V, D = 50, 3, 5, 6
x = torch.from_numpy(np.random.randint(V, size=(N, T)))
W = torch.from_numpy(np.random.randn(V, D))
W.requires_grad_()

out = word_embedding_forward(x, W)
dout = torch.from_numpy(np.random.randn(*out.shape))
out.backward(dout)

dW = W.grad.detach().numpy()
x = x.detach().numpy()
```

```

W = W.detach().numpy()
dout = dout.detach().numpy()

def word_embedding_forward_numpy(x, W):
    return word_embedding_forward(
        torch.from_numpy(x),
        torch.from_numpy(W),
    ).numpy()

f = lambda W: word_embedding_forward_numpy(x, W)
dW_num = eval_numerical_gradient_array(f, W, dout)

# You should see an error on the order of 1e-11 or less
print('dW error: ', rel_error(dW, dW_num))

```

dW error: 3.2774595693100364e-12

## 10 Temporal Affine Layer

At every timestep we use an affine function to transform the RNN hidden vector at that timestep into scores for each word in the vocabulary. Because this is very similar to the affine layer that you implemented in assignment 2, we have provided this function for you in the `temporal_affine_forward`. Run the following to perform numeric gradient checking on the implementation. You should see errors on the order of e-9 or less.

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

# Gradient check for temporal affine layer
N, T, D, M = 2, 3, 4, 5
x = torch.from_numpy(np.random.randn(N, T, D))
w = torch.from_numpy(np.random.randn(D, M))
b = torch.from_numpy(np.random.randn(M))

for tensor in [x, w, b]:
    tensor.requires_grad_()
out = temporal_affine_forward(x, w, b)
dout = torch.from_numpy(np.random.randn(*out.shape))
out.backward(dout)

dx = x.grad.detach().numpy()
dw = w.grad.detach().numpy()
db = b.grad.detach().numpy()

x = x.detach().numpy()
w = w.detach().numpy()
b = b.detach().numpy()
dout = dout.detach().numpy()
```

```

def temporal_affine_forward_numpy(x, w, b):
    return temporal_affine_forward(
        torch.from_numpy(x),
        torch.from_numpy(w),
        torch.from_numpy(b),
    ).numpy()

fx = lambda x: temporal_affine_forward_numpy(x, w, b)
fw = lambda w: temporal_affine_forward_numpy(x, w, b)
fb = lambda b: temporal_affine_forward_numpy(x, w, b)

dx_num = eval_numerical_gradient_array(fx, x, dout)
dw_num = eval_numerical_gradient_array(fw, w, dout)
db_num = eval_numerical_gradient_array(fb, b, dout)

print('dx error: ', rel_error(dx_num, dx))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))

```

```

dx error:  2.9215854231394017e-10
dw error:  1.5772088618663602e-10
db error:  3.252209560097257e-11

```

## 11 Temporal Softmax Loss

In an RNN language model, at every timestep we produce a score for each word in the vocabulary. We know the ground-truth word at each timestep, so we use a softmax loss function to compute loss and gradient at each timestep. We sum the losses over time and average them over the minibatch.

However there is one wrinkle: since we operate over minibatches and different captions may have different lengths, we append <NULL> tokens to the end of each caption so they all have the same length. We don't want these <NULL> tokens to count toward the loss or gradient, so in addition to scores and ground-truth labels our loss function also accepts a `mask` array that tells it which elements of the scores count towards the loss.

Since this is very similar to the softmax loss function you implemented in assignment 1, we have implemented this loss function for you; look at the `temporal_softmax_loss` function in the file `icv83551/rnn_layers_pytorch.py`.

Run the following cell to sanity check the loss and perform numeric gradient checking on the function. You should see an error for `dx` on the order of `e-7` or less.

```

[11]: # Sanity check for temporal softmax loss
from icv83551.rnn_layers_pytorch import temporal_softmax_loss

N, T, V = 100, 1, 10

def check_loss(N, T, V):

```

```

x = 0.001 * torch.from_numpy(np.random.randn(N, T, V))
y = torch.from_numpy(np.random.randint(V, size=(N, T))).long()
mask = torch.from_numpy(np.random.rand(N, T)) <= p
print(temporal_softmax_loss(x, y, mask).item())

check_loss(100, 1, 10, 1.0)    # Should be about 2.3
check_loss(100, 10, 10, 1.0)   # Should be about 23
check_loss(5000, 10, 10, 0.1)  # Should be within 2.2-2.4

# Gradient check for temporal softmax loss
np.random.seed(231231)
N, T, V = 7, 8, 9

x = torch.from_numpy(np.random.randn(N, T, V))
y = torch.from_numpy(np.random.randint(V, size=(N, T))).long()
mask = torch.from_numpy(np.random.rand(N, T) > 0.5)

x.requires_grad_()
loss = temporal_softmax_loss(x, y, mask, verbose=False)
loss.backward()
dx = x.grad.detach().numpy()
x = x.detach().numpy()
dx_num = eval_numerical_gradient(
    lambda x: temporal_softmax_loss(torch.from_numpy(x), y, mask), x,
    ↵verbose=False)

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

```

```

2.3027781774290146
23.02598595312723
2.2643611790293394
dx error: 5.5228696963589426e-08

```

## 12 RNN for Image Captioning

Now that you have implemented the necessary layers, you can combine them to build an image captioning model. Open the file `icv83551/classifiers/rnn_pytorch.py` and look at the `CaptioningRNN` class.

Implement the forward pass of the model in the `loss` function. For now you only need to implement the case where `cell_type='rnn'` for vanilla RNNs; you will implement the LSTM case later. After doing so, run the following to check your forward pass using a small test case; you should see error on the order of  $e-10$  or less.

```
[12]: N, D, W, H = 10, 20, 30, 40
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
V = len(word_to_idx)
T = 13
```

```

model = CaptioningRNN(
    word_to_idx,
    input_dim=D,
    wordvec_dim=W,
    hidden_dim=H,
    cell_type='rnn',
    dtype=torch.float64
)

# Set all model parameters to fixed values
for k, v in model.params.items():
    model.params[k] = torch.from_numpy(
        np.linspace(-1.4, 1.3, num=v.numel()).reshape(*v.shape))

features = torch.from_numpy(np.linspace(-1.5, 0.3, num=(N * D)).reshape(N, D))
captions = torch.from_numpy((np.arange(N * T) % V).reshape(N, T))

loss = model.loss(features, captions).item()
expected_loss = 9.83235591003

print('loss: ', loss)
print('expected loss: ', expected_loss)
print('difference: ', abs(loss - expected_loss))

```

```

loss: 9.83235591002739
expected loss: 9.83235591003
difference: 2.609468197078968e-12

```

Run the following cell to perform numeric gradient checking on the `CaptioningRNN` class; you should see errors around the order of `e-6` or less.

```

[13]: np.random.seed(231)
torch.manual_seed(231)

batch_size = 2
timesteps = 3
input_dim = 4
wordvec_dim = 5
hidden_dim = 6
word_to_idx = {'<NULL>': 0, 'cat': 2, 'dog': 3}
vocab_size = len(word_to_idx)

captions = torch.from_numpy(np.random.randint(vocab_size, size=(batch_size, timesteps)))
features = torch.from_numpy(np.random.randn(batch_size, input_dim))

model = CaptioningRNN(

```

```

        word_to_idx,
        input_dim=input_dim,
        wordvec_dim=wordvec_dim,
        hidden_dim=hidden_dim,
        cell_type='rnn',
        dtype=torch.float64,
    )

    for k, v in model.params.items():
        v.requires_grad_()
    loss = model.loss(features, captions)
    loss.backward()
    grads = {k: v.grad.detach().numpy() for k, v in model.params.items()}
    for k, v in model.params.items():
        v.requires_grad_(False)

    for param_name in sorted(grads.keys()):
        def fn(val):
            model.params[param_name] = torch.from_numpy(val)
            ret = model.loss(features, captions).numpy()
            return ret

        param_grad_num = eval_numerical_gradient(
            fn, model.params[param_name].numpy(), verbose=False, h=1e-6)

        e = rel_error(param_grad_num, grads[param_name])
        print('%s relative error: %e' % (param_name, e))

```

```

W_embed relative error: 7.405825e-09
W_proj relative error: 1.627846e-08
W_vocab relative error: 5.813051e-09
Wh relative error: 3.722645e-09
Wx relative error: 3.272434e-06
b relative error: 2.259961e-10
b_proj relative error: 7.720396e-10
b_vocab relative error: 4.089136e-09

```

## 13 Overfit RNN Captioning Model on Small Data

Similar to the `Solver` class that we used to train image classification models on the previous assignment, on this assignment we use a `CaptioningSolverPytorch` class to train image captioning models. Open the file `icv83551/captioning_solver_pytorch.py` and read through the `CaptioningSolverPytorch` class; it should look very familiar.

Once you have familiarized yourself with the API, run the following to make sure your model overfits a small sample of 100 training examples. You should see a final loss of less than 0.1.

```
[14]: np.random.seed(231)
torch.manual_seed(231)

small_data = load_coco_data(max_train=50)

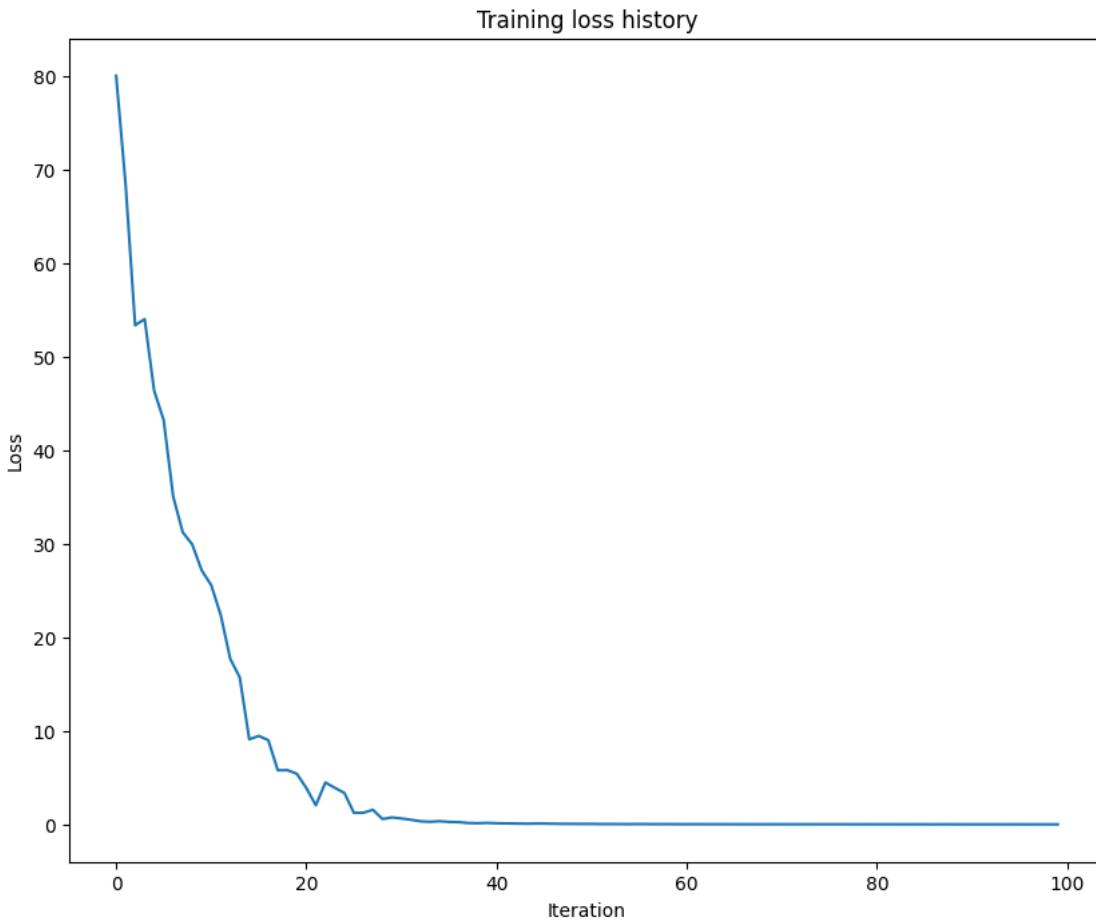
small_rnn_model = CaptioningRNN(
    cell_type='rnn',
    word_to_idx=data['word_to_idx'],
    input_dim=data['train_features'].shape[1],
    hidden_dim=512,
    wordvec_dim=256,
)

small_rnn_solver = CaptioningSolverPytorch(
    small_rnn_model, small_data,
    num_epochs=50,
    batch_size=25,
    learning_rate=5e-3,
    verbose=True, print_every=10,
)

small_rnn_solver.train()

# Plot the training losses.
plt.plot(small_rnn_solver.loss_history)
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training loss history')
plt.show()

base dir  C:\Users\eitan\PythonProjects\CV_projects\assignment2\icv83551\datasets\coco_captioning
(Iteration 1 / 100) loss: 80.027161
(Iteration 11 / 100) loss: 25.581121
(Iteration 21 / 100) loss: 3.890693
(Iteration 31 / 100) loss: 0.641154
(Iteration 41 / 100) loss: 0.133127
(Iteration 51 / 100) loss: 0.057629
(Iteration 61 / 100) loss: 0.028626
(Iteration 71 / 100) loss: 0.022365
(Iteration 81 / 100) loss: 0.018532
(Iteration 91 / 100) loss: 0.016730
```



Print final training loss. You should see a final loss of less than 0.1.

```
[15]: print('Final loss: ', small_rnn_solver.loss_history[-1])
```

```
Final loss:  0.013374083
```

## 14 RNN Sampling at Test Time

Unlike classification models, image captioning models behave very differently at training time vs. at test time. At training time, we have access to the ground-truth caption, so we feed ground-truth words as input to the RNN at each timestep. At test time, we sample from the distribution over the vocabulary at each timestep and feed the sample as input to the RNN at the next timestep.

In the file `icv83551/classifiers/rnn_pytorch.py`, implement the `sample` method for test-time sampling. After doing so, run the following to sample from your overfitted model on both training and validation data. The samples on training data should be very good. The samples on validation data, however, probably won't make sense.

```
[20]: # If you get an error, the URL just no longer exists, so don't worry!
# You can re-sample as many times as you want.
for split in ['train', 'val']:
    minibatch = sample_coco_minibatch(small_data, split=split, batch_size=2)
    gt_captions, features, urls = minibatch
    gt_captions = decode_captions(gt_captions, data['idx_to_word'])

    sample_captions = small_rnn_model.sample(torch.from_numpy(features)).numpy()
    sample_captions = decode_captions(sample_captions, data['idx_to_word'])

    for gt_caption, sample_caption, url in zip(gt_captions, sample_captions, urls):
        img = image_from_url(url)
        # Skip missing URLs.
        if img is None: continue
        plt.imshow(img)
        plt.title('%s\n%s\nGT:%s' % (split, sample_caption, gt_caption))
        plt.axis('off')
        plt.show()
```

train  
 a plane flying close to the ground as <UNK> coming in for landing <END>  
 GT:<START> a plane flying close to the ground as <UNK> coming in for landing <END>



URL Error: Gone [http://farm1.staticflickr.com/202/487987371\\_489a65d670\\_z.jpg](http://farm1.staticflickr.com/202/487987371_489a65d670_z.jpg)

val  
a broken that is to <UNK> <UNK> a school bus <END>  
GT:<START> a woman <UNK> to horses on a farm <END>



val  
a group of men riding in a boat across a lake <END>  
GT:<START> a woman standing on a tennis court with a tennis racquet <END>



## 15 Inline Question 1

In our current image captioning setup, our RNN language model produces a word at every timestep as its output. However, an alternate way to pose the problem is to train the network to operate over *characters* (e.g. ‘a’, ‘b’, etc.) as opposed to words, so that at every timestep, it receives the previous character as input and tries to predict the next character in the sequence. For example, the network might generate a caption like

‘A’, ‘ ‘, ‘c’, ‘a’, ‘t’, ‘ ‘, ‘o’, ‘n’, ‘ ‘, ‘a’, ‘ ‘, ‘b’, ‘e’, ‘d’

Can you describe one advantage of an image-captioning model that uses a character-level RNN? Can you also describe one disadvantage? HINT: there are several valid answers, but it might be useful to compare the parameter space of word-level and character-level models.

**Your Answer:** ### 1. Advantage: Reduced Parameter Space and Vocabulary Flexibility  
Character-level models benefit from a significantly smaller **vocabulary size** compared to word-level models (e.g., ~100 characters versus 10,000+ words). This leads to a much smaller **parameter space** in the embedding and output layers, drastically reducing the memory footprint of the model. Furthermore, these models are naturally robust against **Out-of-Vocabulary (OOV)** tokens; they can generate or interpret any word—including rare technical terms or misspellings—simply by constructing them from individual characters.

---

### 15.0.1 2. Disadvantage: Computational Complexity and Gradient Stability

The primary drawback is the substantial **increase in sequence length**. Since each word is decomposed into multiple characters, the RNN must iterate through many more timesteps to generate a single caption. This extended path makes the model highly susceptible to **vanishing and exploding gradients**. Consequently, it becomes much harder for the network to maintain “long-term memory,” often struggling to keep the end of a long sentence grammatically consistent with the beginning.

[ ]: