1. Let us define $g(w) = f(w^{(t)}) + (w - w^{(t)}) f'(w^{(t)}) + \frac{\lambda}{2} \|w - w^{(t)}\|^2$

as the function to minimize.

$\therefore g'(w) = f'(w^{(t)}) + \lambda(w - w^{(t)})$

Setting $g'(w) = 0$

$$w = w^{(t)} - \frac{1}{\lambda} f'(w^{(t)})$$

$$\therefore \eta = \frac{1}{\lambda}$$

Scanned by CamScanner

(2) $\langle w^{(t)} - w^*, v_t \rangle = \frac{1}{\eta} \langle w^{(t)} - w^*, \eta v_t \rangle$

$$= \frac{1}{2\eta}\left(\|w^{(t)} - w^*\|^2 + \eta^2\|v_t\|^2 - \|w^{(t)} - w^* - \eta v_t\|^2\right)$$

$$= \frac{1}{2\eta}\left(\|w^{(t)} - w^*\|^2 - \|w^{(t+1)} - w^*\|^2\right) + \frac{\eta}{2}\|v_t\|^2$$

Summing over $t = 1$ to T

$$\sum_{t=1}^{T} \langle w^{(t)} - w^*, v_t \rangle = \frac{1}{2\eta}\sum_{t=1}^{T}\left(\|w^{(t)} - w^*\|^2 - \|w^{(t+1)} - w^*\|^2\right) + \frac{\eta}{2}\sum_{t=1}^{T}\|v_t\|^2$$

$$= \frac{1}{2\eta}\left(\|w^{(1)} - w^*\|^2 - \|w^{(T+1)} - w^*\|^2\right) + \frac{\eta}{2}\sum_{t=1}^{T}\|v_t\|^2$$

$$\leq \frac{\|w^*\|^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{T}\|v_t\|^2$$

Hence Proved.

Scanned by CamScanner

(3) $f(\bar{w}) - f(w^*) = f\left(\frac{1}{T}\sum_{t=1}^{T} w^{(t)}\right) - f(w^*)$

Using Jensen's inequality

$$\leq \frac{1}{T}\sum_{t=1}^{T} f(w^{(t)}) - f(w^*)$$

$$\leq \frac{1}{T}\sum_{t=1}^{T}\left(f(w^{(t)}) - f(w^*)\right)$$

As f is convex

$$f(w^{(t)}) - f(w^*) \leq (w^{(t)} - w^*)\nabla f(w^{(t)})$$

$$\therefore f(\bar{w}) - f(w^*) \leq \frac{1}{T}\sum_{t=1}^{T} \langle w^{(t)} - w^*, \nabla f(w^{(t)})\rangle$$

$$\leq \frac{1}{T}\left(\frac{\|w^*\|^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{T}\|\nabla f(w^{(t)})\|^2\right)$$

$$\leq \frac{1}{T}\left(\frac{B^{\cancel{2}}}{2\cancel{B}}\rho\sqrt{T} + \frac{B}{2\cancel{\rho}\cancel{\sqrt{T}}}\sqrt{T}\cdot\rho^{\cancel{2}}\right)$$

$$\leq \frac{1}{T} B\rho\sqrt{T} = \frac{B\rho}{\sqrt{T}}$$

$$\therefore f(\bar{w}) - f(w^*) \propto \frac{1}{\sqrt{T}}$$

$$\sim O\left(\frac{1}{\sqrt{T}}\right)$$

Scanned by CamScanner

(4) So (4.1) doesn't guarantee to decrease $f(w)$. On ~~simplyfying~~ simplifying -

$f(w) = \ln(e^{w} + e^{-w} + 2)$, $f_1(w) = \ln(e^{w}+1)$

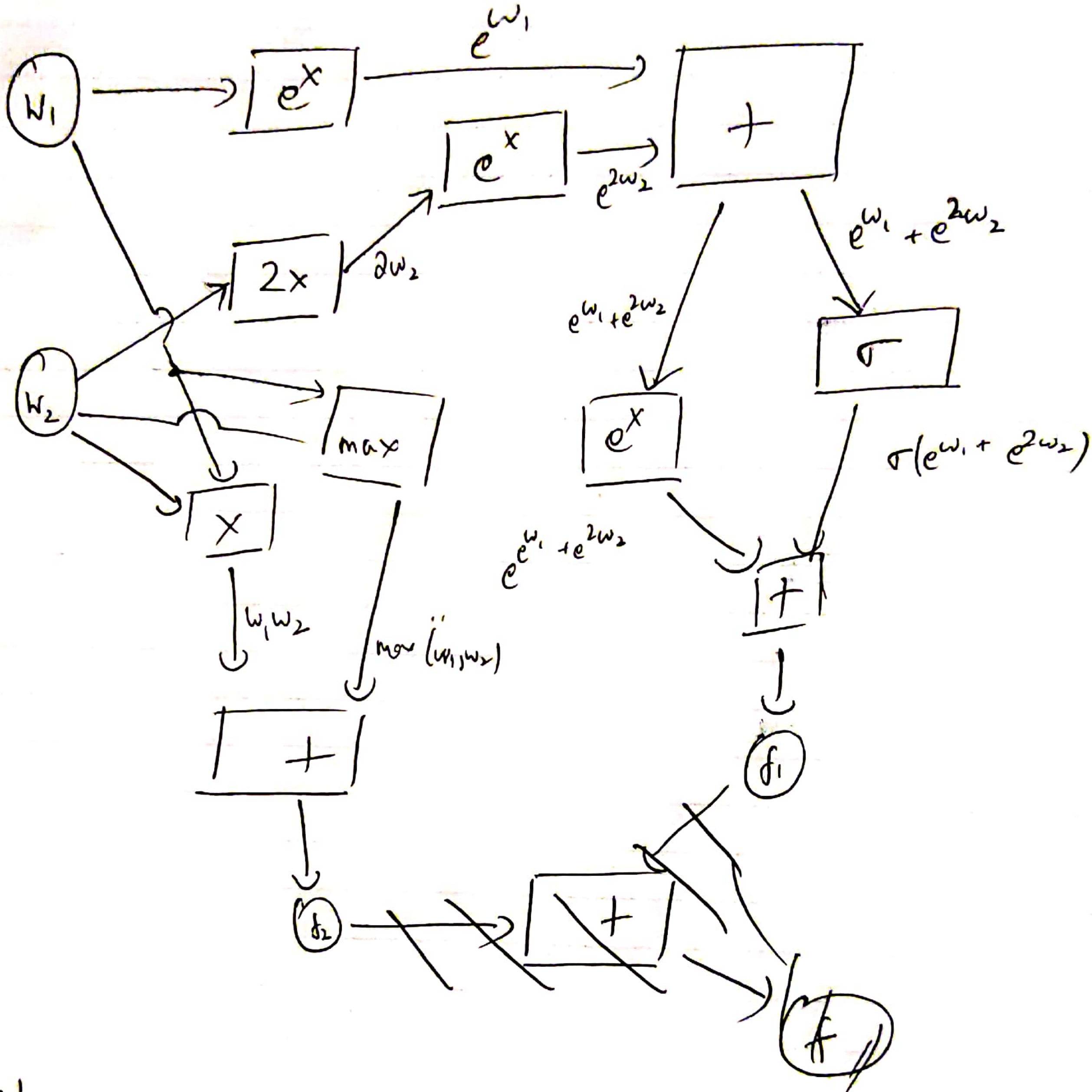
$f'(w) = \dfrac{1-e^{-w}}{1+e^{-w}}$, $f_1'(w) = \dfrac{1}{1+e^{-w}}$ and $f_2'(w) = \dfrac{-e^{-w}}{1+e^{-w}}$

$\therefore$ If we set $w^0 = 0$ & choose to descend along f_2,

then $w' = \eta f_2'(w) = \dfrac{\eta}{2}$ and $f\left(\dfrac{\eta}{2}\right) > f(0)$ as 0 is the minima of the function f.

Scanned by CamScanner

5. (a)



$w_1 = 1$

$w_2 = -1$

$e^{w_1} + e^{2w_2} = e + e^{-2}$

$f_1(\vec{w}) = e^{e+e^{-2}} + \sigma(e+e^{-2})$

$f_2(\vec{w}) = -1 + 1 = 0$

$\therefore f(\vec{w}) = \langle 18.296, 0 \rangle$

$\therefore \cancel{f(\vec{w}) = e^{e+e^{-2}} + \sigma(e+e^{-2})}$

$\cancel{= 18.296}$

5 (b) $\frac{\partial f}{\partial w} = \left[\frac{f(1+\Delta w, -1) - f(1,-1)}{\Delta w}, \frac{f(1, -1+\Delta w) - f(1,-1)}{\Delta w} \right]$

$= \left[\frac{f(1.01, -1) - f(1,1)}{.01}, \frac{f(1, -0.99) - f(1,-1)}{0.01} \right]$

$= \left[\frac{\langle 18.778, 0\rangle - \langle 18.256, 0\rangle}{.01}, \frac{\langle 18.344, 0.01\rangle - \langle 18.256, 0\rangle}{0.01} \right]$

$= \begin{bmatrix} 48.2 & 4.8 \\ 0 & 1 \end{bmatrix}$

5. (c) For input w_1 –

Let $a = w_1$

$$\mathring{i}_1 = e^{w_1}$$
$$i_2 = e^{w_1} + e^{2w_2} = i_1 + e^{2w_2}$$
$$i_3 = e^{i_2}$$
$$i_4 = \sigma(i_2)$$
$$f_1 = i_3 + i_4$$

$$i_5 = w_1 \cdot w_2$$
$$i_6 = \max(w_1, w_2) = w_1 \text{ at } \vec{\omega}$$
$$f_2 = i_5 + i_6$$

$$\mathring{w}_1 = \frac{\partial w_1}{\partial a} = 1$$

$$\mathring{i}_1 = \frac{\partial i_1}{\partial a} = \frac{\partial i_1}{\partial w_1} \cdot \frac{\partial w_1}{\partial a} = e^{w_1} \cdot \mathring{w}_1 = e^{w_1}$$

And using the above chain rule

$$\mathring{i}_2 = \frac{\partial i_2}{\partial a} = \frac{\partial i_1}{\partial a} = e^{w_1}$$

$$\mathring{i}_3 = \frac{\partial i_3}{\partial a} = e^{i_2} \cdot \mathring{i}_2 = e^{i_2} \cdot e^{w_1}$$

$$\mathring{i}_4 = \frac{\partial i_4}{\partial a} = \sigma(i_2)\left(1 - \sigma(i_2)\right) e^{w_1}$$

$$\frac{\partial f_1}{\partial a} = \mathring{i}_3 + \mathring{i}_4 = e^{w_1}\left(e^{i_2} + \sigma(i_2)\left(1 - \sigma(i_2)\right)\right)$$

where $i_2 = e^{w_1} + e^{2w_2}$

$$\mathring{i}_5 = \frac{\partial i_5}{\partial a} = w_2$$

$$\mathring{i}_6 = \frac{\partial i_6}{\partial a} = 1$$

$$\frac{\partial f_2}{\partial a} = \mathring{i}_5 + \mathring{i}_6 = w_2 + 1$$

Scanned by CamScanner

For input w_2

Let $a = w_2$

$$i_1 = 2w_2$$
$$i_2 = e^{i_1}$$
$$i_3 = e^{w_1} + i_2$$
$$i_4 = e^{i_3}$$
$$i_5 = \sigma(i_3)$$
$$f_1 = i_4 + i_5$$

$$i_6 = w_1 \cdot w_2$$
$$i_7 = \max(w_1, w_2) = w_1 \quad \text{at } \vec{\omega}$$
$$f_2 = i_6 + i_7$$

$$\dot{w}_2 = \frac{\partial w_2}{\partial a} = 1$$

$$\dot{i}_1 = 2$$
$$\dot{i}_2 = 2 \cdot e^{2w_2}$$
$$\dot{i}_3 = \dot{i}_2 = 2 \cdot e^{2w_2}$$
$$\dot{i}_4 = e^{i_3} \cdot \dot{i}_3 = e^{i_3} \cdot 2 \cdot e^{2w_2}$$
$$\dot{i}_5 = \sigma(i_3)\left(1 - \sigma(i_3)\right) \dot{i}_3$$

$$\dot{f}_1 = 2e^{2w_2}\left(e^{i_3} + \sigma(i_3)\left(1 - \sigma(i_3)\right)\right)$$

$$\dot{i}_6 = w_1$$
$$\dot{i}_7 = 0$$
$$\dot{f}_2 = w_1$$

$$\therefore \text{Jacobian} = \begin{bmatrix} 47.303 & 4.7102 \\ 0 & 1 \end{bmatrix}$$

Scanned by CamScanner

5. (d)

$f_1 = i_1 + i_2$

$i_1 = e^{i_3}$ ~~+ ...~~

$i_2 = \sigma(i_3)$

$i_3 = i_4 + i_5$

$i_4 = e^{w_1}$

$i_5 = e^{i_6}$

$i_6 = 2w_2$

$f_2 = i_7 + i_8$

$i_7 = w_1 \cdot w_2$

$i_8 = \max(w_1, w_2) = w_1$ at $\vec{w}$

For input w_1 -

$f_1' = i_1' + i_2'$

$i_1' = e^{i_3} \cdot i_3'$

$i_2' = \sigma(i_3)(1-\sigma(i_3))\, i_3'$

$i_3' = i_4' + i_5'$

$i_4 = e^{w_1}$

$i_5 = e^{i_6} \cdot i_6'$

$i_6' = 0$

$f_2' = i_7' + i_8'$

$i_7 = w_2$

$i_8 = 1$

$\therefore f_1' = \dfrac{\partial f_1}{\partial w_1} = e^{w_1}\left(e^{i_3} + \sigma(i_3)(1-\sigma(i_3))\right)$

$f_2' = w_2 + 1$

For input w_2 -

$f_1^o = i_1^o + i_2^o$

$i_1^o = e^{i_3} \cdot i_3^o$

$i_2^o = \sigma(i_3)(1-\sigma(i_3))\, i_3^o$

$i_3^o = i_4^o + i_5^o$

$i_4^o = 0$

$i_5^o = e^{i_6} \cdot i_6^o = 2e^{i_6}$

$f_1^o = 2e^{2w_2}\left(e^{i_3} + \sigma(i_3)(1-\sigma(i_3))\right)$

$f_2^o = i_7^o + i_8^o$

$i_7^o = w_1$

$i_8^o = 0$

$f_2^o = w_1$

$\therefore$ Jacobian $= \begin{bmatrix} 47.303 & 407.102 \\ 0 & 1 \end{bmatrix}$

Scanned by CamScanner

Paper Review

This was a very interesting paper to read as it was different from other papers and delved into an area I did not know of. The paper tried to highlight the importance of architectures by taking random weights and sharing them between different architectures to search for optimal architectures for specific problems. These architectures were tested without training the weights and a novel variant of NEAT was used to find **minimal** optimal architectures. This made the paper slightly stronger as the paper kept in mind the complexity of a model and used an incremental approach to building such models. They also tried their search algorithm to build models for reinforcement learning based games as well as supervised learning problems like MNIST. While the accuracies they were able to achieve seemed impressive, it wasn't clear as to how long it took to search for the optimal structures. Moreover, I believe once the architectures start getting **deep**, the algorithm should become exponentially slower and that means it probably remains practical for easier problems. I was interested to see if a reversed approach of reducing complexity can be used on current SOTA models to verify if we can retain their performance with lower complexity. One of the obvious uses of this approach is to help find optimal architectures before one starts training their weights for a specific problem. But here a question that arises is that does the architecture found keep performing well if the weights are trained on it or while finding the optimal architecture on random weights, have we skipped an architecture that will perform better with trained weights. That is to say, is architecture search followed by weight training always optimal or do we have to actually do both together?

softmax

February 11, 2020

1 Softmax Classifier

This exercise guides you through the process of classifying images using a Softmax classifier. As part of this you will:

- Implement a fully vectorized loss function for the Softmax classifier
- Calculate the analytical gradient using vectorized code
- Tune hyperparameters on a validation set
- Optimize the loss function with Stochastic Gradient Descent (SGD)
- Visualize the learned weights

[2]:
```
# start-up code!
import random

import matplotlib.pyplot as plt
import numpy as np

%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 extenrnal modules
# see http://stackoverflow.com/questions/1907993/
↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
```

[3]:
```
from load_cifar10_tvt import load_cifar10_train_val

X_train, y_train, X_val, y_val, X_test, y_test = load_cifar10_train_val()
print("Train data shape: ", X_train.shape)
print("Train labels shape: ", y_train.shape)
print("Val data shape: ", X_val.shape)
print("Val labels shape: ", y_val.shape)
print("Test data shape: ", X_test.shape)
print("Test labels shape: ", y_test.shape)
```

```
Train, validation and testing sets have been created as
```

```
 X_i and y_i where i=train,val,test
Train data shape:  (3073, 49000)
Train labels shape:  (49000,)
Val data shape:  (3073, 1000)
Val labels shape:  (1000,)
Test data shape:  (3073, 1000)
Test labels shape:  (1000,)
```

Code for this section is to be written in `cs231n/classifiers/softmax.py`

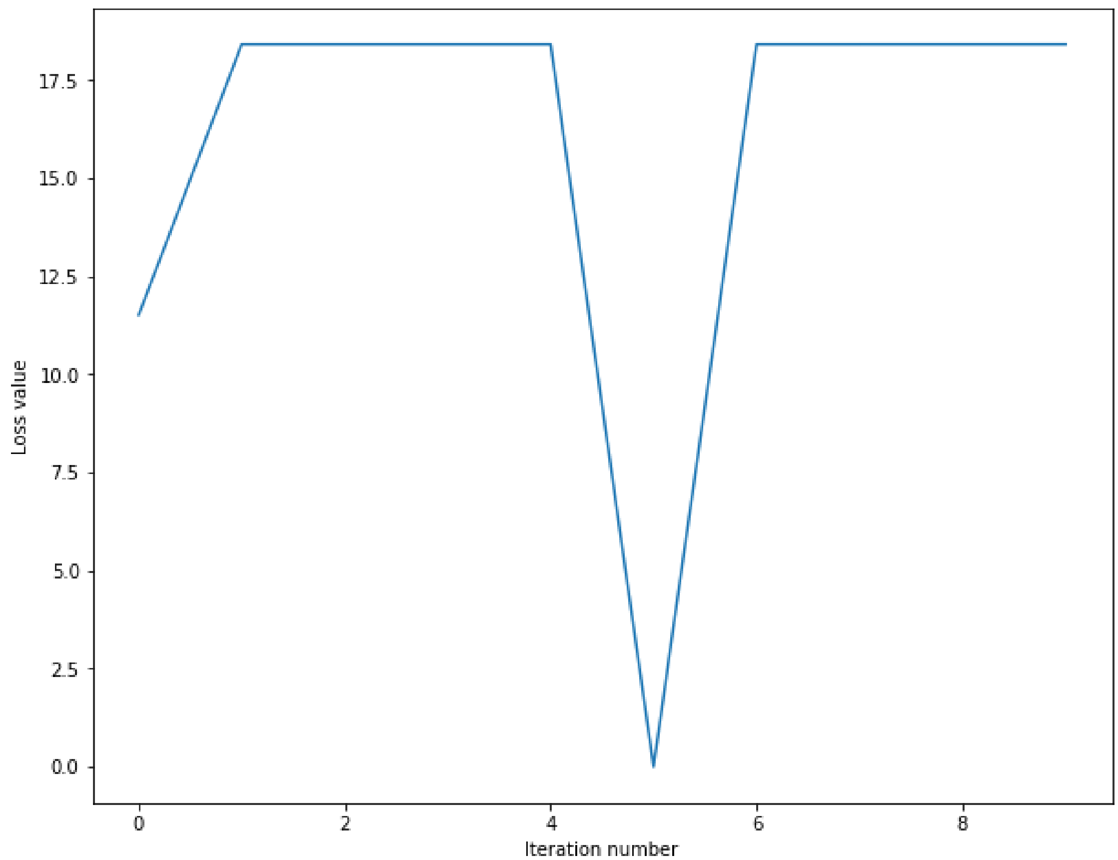
[47]:
```python
# Now, implement the vectorized version in softmax_loss_vectorized.

import time

from cs231n.classifiers.softmax import softmax_loss_vectorized

# gradient check.
from cs231n.gradient_check import grad_check_sparse

W = np.random.randn(10, 3073) * 0.0001

tic = time.time()
loss, grad = softmax_loss_vectorized(W, X_train, y_train, 0.00001)
toc = time.time()
print("vectorized loss: %e computed in %fs" % (loss, toc - tic))

# As a rough sanity check, our loss should be something close to -log(0.1).
print("loss: %f" % loss)
print("sanity check: %f" % (-np.log(0.1)))

f = lambda w: softmax_loss_vectorized(w, X_train, y_train, 0.0)[0]
grad_numerical = grad_check_sparse(f, W, grad, 10)
```

```
vectorized loss: 2.374608e+00 computed in 0.809769s
loss: 2.374608
sanity check: 2.302585
numerical: -1.470689 analytic: -1.470689, relative error: 5.385235e-08
numerical: 0.546594 analytic: 0.546594, relative error: 5.151508e-08
numerical: 0.212101 analytic: 0.212101, relative error: 6.510477e-08
numerical: -0.580468 analytic: -0.580468, relative error: 9.473899e-08
numerical: 0.813494 analytic: 0.813494, relative error: 1.757073e-08
numerical: 0.397953 analytic: 0.397953, relative error: 2.808242e-08
numerical: -0.694171 analytic: -0.694171, relative error: 1.431749e-07
numerical: -4.441120 analytic: -4.441121, relative error: 6.575908e-08
numerical: 0.739192 analytic: 0.739192, relative error: 3.235504e-08
numerical: 2.383121 analytic: 2.383122, relative error: 7.219369e-08
```

Code for this section is to be written in`cs231n/classifiers/linear_classifier.py`

[58]:
```python
# Now that efficient implementations to calculate loss function and gradient of
↪the softmax are ready,
# use it to train the classifier on the cifar-10 data
# Complete the `train` function in cs231n/classifiers/linear_classifier.py

from cs231n.classifiers.linear_classifier import Softmax

classifier = Softmax()
loss_hist = classifier.train(
    X_train,
    y_train,
    learning_rate=1e-3,
    reg=1e-5,
    num_iters=10,
    batch_size=200,
    verbose=False,
)
# Plot loss vs. iterations
plt.plot(loss_hist)
plt.xlabel("Iteration number")
plt.ylabel("Loss value")
```

[58]:
```
Text(0, 0.5, 'Loss value')
```



[59]:
```
# Complete the `predict` function in cs231n/classifiers/linear_classifier.py
# Evaluate on test set
y_test_pred = classifier.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print("softmax on raw pixels final test set accuracy: %f" % (test_accuracy,))
```

```
softmax on raw pixels final test set accuracy: 0.000000
```

[60]:
```
# Visualize the learned weights for each class
w = classifier.W[:, :-1]  # strip out the bias
w = w.reshape(10, 32, 32, 3)

w_min, w_max = np.min(w), np.max(w)

classes = [
    "plane",
    "car",
    "bird",
    "cat",
    "deer",
```

```
        "dog",
        "frog",
        "horse",
        "ship",
        "truck",
]
for i in range(10):
    plt.subplot(2, 5, i + 1)

    # Rescale the weights to be between 0 and 255
    wimg = 255.0 * (w[i].squeeze() - w_min) / (w_max - w_min)
    plt.imshow(wimg.astype("uint8"))
    plt.axis("off")
    plt.title(classes[i])
```

[ ]:

```
```

two_layer_net

February 11, 2020

1 Implementing a Neural Network

In this exercise we will develop a neural network with fully-connected layers to perform classification, and test it out on the CIFAR-10 dataset.

[ ]:
```
# A bit of setup

import numpy as np
import matplotlib.pyplot as plt

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

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

The neural network parameters will be stored in a dictionary (`model` below), where the keys are the parameter names and the values are numpy arrays. Below, we initialize toy data and a toy model that we will use to verify your implementations.

[ ]:
```
# Create some toy data to check your implementations
input_size = 4
hidden_size = 10
num_classes = 3
num_inputs = 5

def init_toy_model():
  model = {}
```

```
    model['W1'] = np.linspace(-0.2, 0.6, num=input_size*hidden_size).
 ↪reshape(input_size, hidden_size)
    model['b1'] = np.linspace(-0.3, 0.7, num=hidden_size)
    model['W2'] = np.linspace(-0.4, 0.1, num=hidden_size*num_classes).
 ↪reshape(hidden_size, num_classes)
    model['b2'] = np.linspace(-0.5, 0.9, num=num_classes)
    return model

def init_toy_data():
    X = np.linspace(-0.2, 0.5, num=num_inputs*input_size).reshape(num_inputs,␣
 ↪input_size)
    y = np.array([0, 1, 2, 2, 1])
    return X, y

model = init_toy_model()
X, y = init_toy_data()
```

2 Forward pass: compute scores

Open the file `cs231n/classifiers/neural_net.py` and look at the function `two_layer_net`. This function is very similar to the loss functions you have written for the Softmax exercise in HW0: It takes the data and weights and computes the class scores, the loss, and the gradients on the parameters.

Implement the first part of the forward pass which uses the weights and biases to compute the scores for all inputs.

[ ]:
```
from cs231n.classifiers.neural_net import two_layer_net

scores = two_layer_net(X, model)
print(scores)
correct_scores = [[-0.5328368, 0.20031504, 0.93346689],
 [-0.59412164, 0.15498488, 0.9040914 ],
 [-0.67658362, 0.08978957, 0.85616275],
 [-0.77092643, 0.01339997, 0.79772637],
 [-0.89110401, -0.08754544, 0.71601312]]

# the difference should be very small. We get 3e-8
print('Difference between your scores and correct scores:')
print(np.sum(np.abs(scores - correct_scores)))
```

3 Forward pass: compute loss

In the same function, implement the second part that computes the data and regularizaion loss.

```
[ ]: reg = 0.1
loss, _ = two_layer_net(X, model, y, reg)
correct_loss = 1.38191946092

# should be very small, we get 5e-12
print('Difference between your loss and correct loss:')
print(np.sum(np.abs(loss - correct_loss)))
```

4 Backward pass

Implement the rest of the function. This will compute the gradient of the loss with respect to the variables W1, b1, W2, and b2. Now that you (hopefully!) have a correctly implemented forward pass, you can debug your backward pass using a numeric gradient check:

```
[ ]: from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backward␣
↪pass.
# If your implementation is correct, the difference between the numeric and
# analytic gradients should be less than 1e-8 for each of W1, W2, b1, and b2.

loss, grads = two_layer_net(X, model, y, reg)

# these should all be less than 1e-8 or so
for param_name in grads:
  param_grad_num = eval_numerical_gradient(lambda W: two_layer_net(X, model, y,␣
↪reg)[0], model[param_name], verbose=False)
  print('%s max relative error: %e' % (param_name, rel_error(param_grad_num,␣
↪grads[param_name])))
```

5 Train the network

To train the network we will use SGD with Momentum. Last assignment you implemented vanilla SGD. You will now implement the momentum update and the RMSProp update. Open the file classifier_trainer.py and familiarize yourself with the ClassifierTrainer class. It performs optimization given an arbitrary cost function data, and model. By default it uses vanilla SGD, which we have already implemented for you. First, run the optimization below using Vanilla SGD:

```
[ ]: from cs231n.classifier_trainer import ClassifierTrainer

model = init_toy_model()
trainer = ClassifierTrainer()
# call the trainer to optimize the loss
# Notice that we're using sample_batches=False, so we're performing Gradient␣
↪Descent (no sampled batches of data)
```

```
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                               model, two_layer_net,
                                               reg=0.001,
                                               learning_rate=1e-1, momentum=0.0,␣
 ↪learning_rate_decay=1,
                                               update='sgd', sample_batches=False,
                                               num_epochs=100,
                                               verbose=False)
print('Final loss with vanilla SGD: %f' % (loss_history[-1], ))
```

Now fill in the **momentum update** in the first missing code block inside the `train` function, and run the same optimization as above but with the momentum update. You should see a much better result in the final obtained loss:

[ ]:
```
model = init_toy_model()
trainer = ClassifierTrainer()
# call the trainer to optimize the loss
# Notice that we're using sample_batches=False, so we're performing Gradient␣
 ↪Descent (no sampled batches of data)
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                               model, two_layer_net,
                                               reg=0.001,
                                               learning_rate=1e-1, momentum=0.9,␣
 ↪learning_rate_decay=1,
                                               update='momentum',␣
 ↪sample_batches=False,
                                               num_epochs=100,
                                               verbose=False)
correct_loss = 0.494394
print('Final loss with momentum SGD: %f. We get: %f' % (loss_history[-1],␣
 ↪correct_loss))
```

The **RMSProp** update step is given as follows:

```
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / np.sqrt(cache + 1e-8)
```

Here, `decay_rate` is a hyperparameter and typical values are [0.9, 0.99, 0.999].

Implement the **RMSProp** update rule inside the `train` function and rerun the optimization:

[ ]:
```
model = init_toy_model()
trainer = ClassifierTrainer()
# call the trainer to optimize the loss
# Notice that we're using sample_batches=False, so we're performing Gradient␣
 ↪Descent (no sampled batches of data)
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                               model, two_layer_net,
                                               reg=0.001,
```

```
                                                   learning_rate=1e-1, momentum=0.9,␣
↪learning_rate_decay=1,
                                                   update='rmsprop',␣
↪sample_batches=False,
                                                   num_epochs=100,
                                                   verbose=False)
correct_loss = 0.439368
print('Final loss with RMSProp: %f. We get: %f' % (loss_history[-1],␣
↪correct_loss))
```

6 Load the data

Now that you have implemented a two-layer network that passes gradient checks, it's time to load up our favorite CIFAR-10 data so we can use it to train a classifier.

[ ]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Reshape data to rows
    X_train = X_train.reshape(num_training, -1)
    X_val = X_val.reshape(num_validation, -1)
```

```
    X_test = X_test.reshape(num_test, -1)

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

7 Train a network

To train our network we will use SGD with momentum. In addition, we will adjust the learning rate with an exponential learning rate schedule as optimization proceeds; after each epoch, we will reduce the learning rate by multiplying it by a decay rate.

[ ]:
```
from cs231n.classifiers.neural_net import init_two_layer_model

model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number␣
↪of classes
trainer = ClassifierTrainer()
best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_train,␣
↪X_val, y_val,
                                             model, two_layer_net,
                                             num_epochs=5, reg=1.0,
                                             momentum=0.9, learning_rate_decay␣
↪= 0.95,
                                             learning_rate=1e-5, verbose=True)
```

8 Debug the training

With the default parameters we provided above, you should get a validation accuracy of about 0.37 on the validation set. This isn't very good.

One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[ ]:
```
# Plot the loss function and train / validation accuracies
plt.subplot(2, 1, 1)
```

```
plt.plot(loss_history)
plt.title('Loss history')
plt.xlabel('Iteration')
plt.ylabel('Loss')

plt.subplot(2, 1, 2)
plt.plot(train_acc)
plt.plot(val_acc)
plt.legend(['Training accuracy', 'Validation accuracy'], loc='lower right')
plt.xlabel('Epoch')
plt.ylabel('Clasification accuracy')
```

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

# Visualize the weights of the network

def show_net_weights(model):
    plt.imshow(visualize_grid(model['W1'].T.reshape(-1, 32, 32, 3), padding=3).
 ↪astype('uint8'))
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```

9 Tune your hyperparameters

What's wrong?. Looking at the visualizations above, we see that the loss is decreasing more or less linearly, which seems to suggest that the learning rate may be too low. Moreover, there is no gap between the training and validation accuracy, suggesting that the model we used has low capacity, and that we should increase its size. On the other hand, with a very large model we would expect to see more overfitting, which would manifest itself as a very large gap between the training and validation accuracy.

Tuning. Tuning the hyperparameters and developing intuition for how they affect the final performance is a large part of using Neural Networks, so we want you to get a lot of practice. Below, you should experiment with different values of the various hyperparameters, including hidden layer size, learning rate, numer of training epochs, and regularization strength. You might also consider tuning the momentum and learning rate decay parameters, but you should be able to get good performance using the default values.

Approximate results. You should be aim to achieve a classification accuracy of greater than 50% on the validation set. Our best network gets over 56% on the validation set.

Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. For every 1% above 56% on the Test set we will award you with one extra bonus point. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding dropout, or adding features to the solver, etc.).

[ ]:
```
from itertools import product

best_model = None # store the best model into this
best_acc = 0.0

H = np.linspace(500, 5000, 5)
learning_rates = np.linspace(1e-6, 1e-2, 5)
E = np.linspace(1, 10, 10)
regs = np.linspace(0, 1, 5)
for h, lr, e, r in list(product(H, learning_rates, E, regs)):
    model = init_two_layer_model(32*32*3, int(h), 10)
    trainer = ClassifierTrainer()
    current_model, loss_history, train_acc, val_acc = trainer.train(X_train,␣
 ↪y_train,
                                                    X_val, y_val,
                                                    model, two_layer_net,
                                                    num_epochs=int(e), reg=r,
                                                    momentum=0.0,
                                                    learning_rate_decay=0.0,
                                                    learning_rate=lr, verbose=True)
#     print(val_acc)
    if max(val_acc) > best_acc:
        best_model = current_model
        best_acc = max(val_acc)
```

[ ]:
```
# visualize the weights
show_net_weights(best_model)
```

10 Run on the test set

When you are done experimenting, you should evaluate your final trained network on the test set.

[ ]:
```
scores_test = two_layer_net(X_test, best_model)
print('Test accuracy: ', np.mean(np.argmax(scores_test, axis=1) == y_test))
```

layers

February 11, 2020

1 Modular neural nets

In the previous exercise, we computed the loss and gradient for a two-layer neural network in a single monolithic function. This isn't very difficult for a small two-layer network, but would be tedious and error-prone for larger networks. Ideally we want to build networks using a more modular design so that we can snap together different types of layers and loss functions in order to quickly experiment with different architectures.

In this exercise we will implement this approach, and develop a number of different layer types in isolation that can then be easily plugged together. For each layer we will implement `forward` and `backward` functions. The `forward` function will receive data, weights, and other parameters, and will return both an output and a `cache` object that stores data needed for the backward pass. The `backward` function will recieve upstream derivatives and the cache object, and will return gradients with respect to the data and all of the weights. This will allow us to write code that looks like this:

```
def two_layer_net(X, W1, b1, W2, b2, reg):
    # Forward pass; compute scores
    s1, fc1_cache = affine_forward(X, W1, b1)
    a1, relu_cache = relu_forward(s1)
    scores, fc2_cache = affine_forward(a1, W2, b2)

    # Loss functions return data loss and gradients on scores
    data_loss, dscores = svm_loss(scores, y)

    # Compute backward pass
    da1, dW2, db2 = affine_backward(dscores, fc2_cache)
    ds1 = relu_backward(da1, relu_cache)
    dX, dW1, db1 = affine_backward(ds1, fc1_cache)

    # A real network would add regularization here

    # Return loss and gradients
    return loss, dW1, db1, dW2, db2
```

[1]:
```
# As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
```

```
from cs231n.gradient_check import eval_numerical_gradient_array,␣
 ↪eval_numerical_gradient
from cs231n.layers import *

%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 Affine layer: forward

Open the file `cs231n/layers.py` and implement the `affine_forward` function.

Once you are done we will test your can test your implementation by running the following:

[2]:
```
# Test the affine_forward function

num_inputs = 2
input_shape = (4, 5, 6)
output_dim = 3

input_size = num_inputs * np.prod(input_shape)
weight_size = output_dim * np.prod(input_shape)

x = np.linspace(-0.1, 0.5, num=input_size).reshape(num_inputs, *input_shape)
w = np.linspace(-0.2, 0.3, num=weight_size).reshape(np.prod(input_shape),␣
 ↪output_dim)
b = np.linspace(-0.3, 0.1, num=output_dim)

out, _ = affine_forward(x, w, b)
correct_out = np.array([[ 1.49834967,  1.70660132,  1.91485297],
                        [ 3.25553199,  3.5141327,   3.77273342]])

# Compare your output with ours. The error should be around 1e-9.
print('Testing affine_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing affine_forward function:
difference:  9.769847728806635e-10
```

3 Affine layer: backward

Now implement the `affine_backward` function. You can test your implementation using numeric gradient checking.

[10]:
```python
# Test the affine_backward function

x = np.random.randn(10, 2, 3)
w = np.random.randn(6, 5)
b = np.random.randn(5)
dout = np.random.randn(10, 5)

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

_, cache = affine_forward(x, w, b)
dx, dw, db = affine_backward(dout, cache)

# The error should be less than 1e-10
print('Testing affine_backward function:')
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 affine_backward function:
dx error:  6.229740884137868e-10
dw error:  8.216698307090757e-11
db error:  1.537238419837524e-10
```

4 ReLU layer: forward

Implement the `relu_forward` function and test your implementation by running the following:

[11]:
```python
# Test the relu_forward function

x = np.linspace(-0.5, 0.5, num=12).reshape(3, 4)

out, _ = relu_forward(x)
correct_out = np.array([[ 0.,          0.,          0.,          0.,         ],
                        [ 0.,          0.,          0.04545455,  0.13636364,],
```

```
                    [ 0.22727273,  0.31818182,  0.40909091,  0.5,       ]])

# Compare your output with ours. The error should be around 1e-8
print('Testing relu_forward function:')
print('difference: ', rel_error(out, correct_out))
```

```
Testing relu_forward function:
difference:  4.999999798022158e-08
```

5 ReLU layer: backward

Implement the `relu_backward` function and test your implementation using numeric gradient checking:

[13]:
```
x = np.random.randn(10, 10)
dout = np.random.randn(*x.shape)

dx_num = eval_numerical_gradient_array(lambda x: relu_forward(x)[0], x, dout)

_, cache = relu_forward(x)
dx = relu_backward(dout, cache)

# The error should be around 1e-12
print('Testing relu_backward function:')
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing relu_backward function:
dx error:  3.275612814312212e-12
```

6 Loss layers: Softmax and SVM

You implemented these loss functions in the last assignment, so we'll give them to you for free here. It's still a good idea to test them to make sure they work correctly.

[14]:
```
num_classes, num_inputs = 10, 50
x = 0.001 * np.random.randn(num_inputs, num_classes)
y = np.random.randint(num_classes, size=num_inputs)

dx_num = eval_numerical_gradient(lambda x: svm_loss(x, y)[0], x, verbose=False)
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be 1e-9
print('Testing svm_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
dx_num = eval_numerical_gradient(lambda x: softmax_loss(x, y)[0], x,␣
 ↪verbose=False)
loss, dx = softmax_loss(x, y)

# Test softmax_loss function. Loss should be 2.3 and dx error should be 1e-8
print('\nTesting softmax_loss:')
print('loss: ', loss)
print('dx error: ', rel_error(dx_num, dx))
```

```
Testing svm_loss:
loss:  8.999901006028317
dx error:  3.6226026387582733e-09

Testing softmax_loss:
loss:  2.3025756811914233
dx error:  1.0773563075545908e-08
```

7 Convolution layer: forward naive

We are now ready to implement the forward pass for a convolutional layer. Implement the function `conv_forward_naive` in the file `cs231n/layers.py`.

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:

[62]:
```
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)
print(out)
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 1e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

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

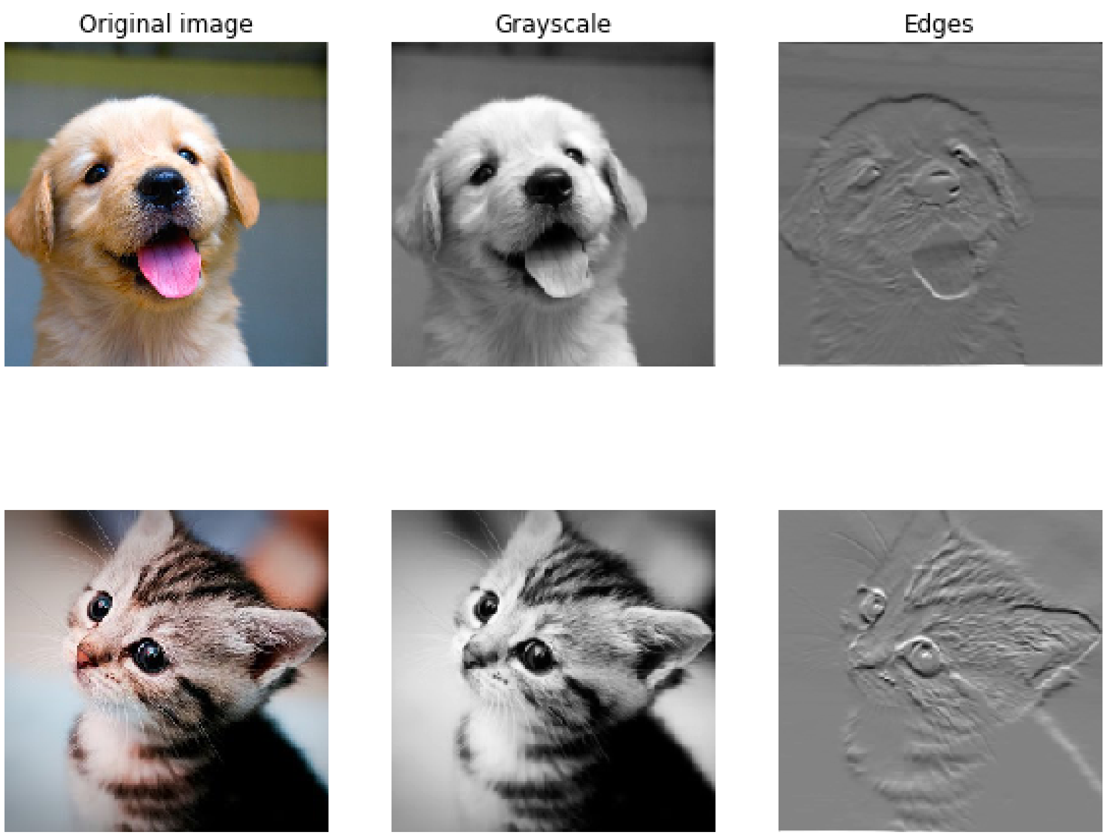
8 Aside: Image processing via convolutions

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.

[63]:
```
from scipy.misc import imread, imresize

kitten, puppy = imread('kitten.jpg'), imread('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
x = np.zeros((2, 3, img_size, img_size))
x[0, :, :, :] = imresize(puppy, (img_size, img_size)).transpose((2, 0, 1))
x[1, :, :, :] = imresize(kitten_cropped, (img_size, img_size)).transpose((2, 0,
 ↪1))
```

```python
# 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_noax(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_noax(puppy, normalize=False)
plt.title('Original image')
plt.subplot(2, 3, 2)
imshow_noax(out[0, 0])
plt.title('Grayscale')
plt.subplot(2, 3, 3)
imshow_noax(out[0, 1])
plt.title('Edges')
plt.subplot(2, 3, 4)
imshow_noax(kitten_cropped, normalize=False)
plt.subplot(2, 3, 5)
imshow_noax(out[1, 0])
plt.subplot(2, 3, 6)
imshow_noax(out[1, 1])
plt.show()
```

```
/Users/ssipani/miniconda3/envs/cs7643/lib/python3.7/site-
packages/ipykernel_launcher.py:3: DeprecationWarning: `imread` is deprecated!
`imread` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``imageio.imread`` instead.
  This is separate from the ipykernel package so we can avoid doing imports
until
/Users/ssipani/miniconda3/envs/cs7643/lib/python3.7/site-
packages/ipykernel_launcher.py:10: DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # Remove the CWD from sys.path while we load stuff.
/Users/ssipani/miniconda3/envs/cs7643/lib/python3.7/site-
packages/ipykernel_launcher.py:11: DeprecationWarning: `imresize` is deprecated!
`imresize` is deprecated in SciPy 1.0.0, and will be removed in 1.2.0.
Use ``skimage.transform.resize`` instead.
  # This is added back by InteractiveShellApp.init_path()
```



9 Convolution layer: backward naive

Next you need to implement the function `conv_backward_naive` in the file `cs231n/layers.py`. As usual, we will check your implementation with numeric gradient checking.

[70]:
```
x = np.random.randn(4, 3, 5, 5)
w = np.random.randn(2, 3, 3, 3)
b = np.random.randn(2,)
dout = np.random.randn(4, 2, 5, 5)
conv_param = {'stride': 1, 'pad': 1}

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

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

# print(dx)
# print(dw)

# Your errors should be around 1e-9'
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:  1.562511213608746e-09
dw error:  1.3934435161566938e-08
db error:  8.164009949533664e-12
```

10 Max pooling layer: forward naive

The last layer we need for a basic convolutional neural network is the max pooling layer. First implement the forward pass in the function `max_pool_forward_naive` in the file `cs231n/layers.py`.

[72]:
```
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 around 1e-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
```

11 Max pooling layer: backward naive

Implement the backward pass for a max pooling layer in the function `max_pool_backward_naive` in the file `cs231n/layers.py`. As always we check the correctness of the backward pass using numerical gradient checking.

[73]:
```
x = np.random.randn(3, 2, 8, 8)
dout = np.random.randn(3, 2, 4, 4)
pool_param = {'pool_height': 2, 'pool_width': 2, 'stride': 2}

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

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

# Your error should be around 1e-12
print('Testing max_pool_backward_naive function:')
print('dx error: ', rel_error(dx, dx_num))
```

```
Testing max_pool_backward_naive function:
dx error:  3.275630137940067e-12
```

12 Fast layers

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

The fast convolution implementation depends on a Cython extension; to compile it you need to run the following from the `cs231n` directory:

```
python setup.py build_ext --inplace
```

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

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

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

[74]:
```
from cs231n.fast_layers import conv_forward_fast, conv_backward_fast
from time import time

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: 10.193678s
Fast: 0.027950s
Speedup: 364.713683x
Difference:  1.1552716289304518e-11

Testing conv_backward_fast:
Naive: 17.836557s
Fast: 0.025802s
Speedup: 691.288595x
dx difference:  1.6750554469420988e-11
dw difference:  8.182503010119549e-13
db difference:  0.0
```

[75]:
```
from cs231n.fast_layers import max_pool_forward_fast, max_pool_backward_fast

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('speedup: %fx' % ((t1 - t0) / (t2 - t1)))
print('dx difference: ', rel_error(dx_naive, dx_fast))
```

```
Testing pool_forward_fast:
Naive: 0.792710s
```

```
fast: 0.005289s
speedup: 149.883605x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.754867s
speedup: 31.264054x
dx difference:  0.0
```

13 Sandwich layers

There are a couple common layer "sandwiches" that frequently appear in ConvNets. For example convolutional layers are frequently followed by ReLU and pooling, and affine layers are frequently followed by ReLU. To make it more convenient to use these common patterns, we have defined several convenience layers in the file `cs231n/layer_utils.py`. Lets grad-check them to make sure that they work correctly:

[76]:
```
from cs231n.layer_utils import conv_relu_pool_forward, conv_relu_pool_backward

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)

print('Testing conv_relu_pool_forward:')
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_forward:
dx error:  1.4306251613522052e-08
dw error:  5.693903828917013e-10
db error:  3.699192285144267e-11
```

[77]:
```
from cs231n.layer_utils import conv_relu_forward, conv_relu_backward

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)

print('Testing conv_relu_forward:')
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_forward:
dx error:  4.518922954146391e-09
dw error:  9.909016154816753e-09
db error:  4.308314302730245e-11
```

[78]:
```
from cs231n.layer_utils import affine_relu_forward, affine_relu_backward

x = np.random.randn(2, 3, 4)
w = np.random.randn(12, 10)
b = np.random.randn(10)
dout = np.random.randn(2, 10)

out, cache = affine_relu_forward(x, w, b)
dx, dw, db = affine_relu_backward(dout, cache)

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

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

```python
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward:
dx error:  1.4000743724844122e-10
dw error:  2.004992303924378e-10
db error:  7.434255696947127e-10
```

[ ]:

```python

```

convnet

February 11, 2020

1 Train a ConvNet!

We now have a generic solver and a bunch of modularized layers. It's time to put it all together, and train a ConvNet to recognize the classes in CIFAR-10. In this notebook we will walk you through training a simple two-layer ConvNet and then set you free to build the best net that you can to perform well on CIFAR-10.

Open up the file `cs231n/classifiers/convnet.py`; you will see that the `two_layer_convnet` function computes the loss and gradients for a two-layer ConvNet. Note that this function uses the "sandwich" layers defined in `cs231n/layer_utils.py`.

[1]:
```
# As usual, a bit of setup

import numpy as np
import matplotlib.pyplot as plt
from cs231n.classifier_trainer import ClassifierTrainer
from cs231n.gradient_check import eval_numerical_gradient
from cs231n.classifiers.convnet import *

%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]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
```

```
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
    we used for the SVM, but condensed to a single function.
    """
    # Load the raw CIFAR-10 data
    cifar10_dir = 'cs231n/datasets/cifar-10-batches-py'
    X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

    # Subsample the data
    mask = range(num_training, num_training + num_validation)
    X_val = X_train[mask]
    y_val = y_train[mask]
    mask = range(num_training)
    X_train = X_train[mask]
    y_train = y_train[mask]
    mask = range(num_test)
    X_test = X_test[mask]
    y_test = y_test[mask]

    # Normalize the data: subtract the mean image
    mean_image = np.mean(X_train, axis=0)
    X_train -= mean_image
    X_val -= mean_image
    X_test -= mean_image

    # Transpose so that channels come first
    X_train = X_train.transpose(0, 3, 1, 2).copy()
    X_val = X_val.transpose(0, 3, 1, 2).copy()
    x_test = X_test.transpose(0, 3, 1, 2).copy()

    return X_train, y_train, X_val, y_val, X_test, y_test


# Invoke the above function to get our data.
X_train, y_train, X_val, y_val, X_test, y_test = get_CIFAR10_data()
print('Train data shape: ', X_train.shape)
print('Train labels shape: ', y_train.shape)
print('Validation data shape: ', X_val.shape)
print('Validation labels shape: ', y_val.shape)
print('Test data shape: ', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Train data shape:  (49000, 3, 32, 32)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3, 32, 32)
Validation labels shape:  (1000,)
Test data shape:  (1000, 32, 32, 3)
```

```
Test labels shape:  (1000,)
```

2 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 this should go up.

[3]:
```
model = init_two_layer_convnet()

X = np.random.randn(100, 3, 32, 32)
y = np.random.randint(10, size=100)

loss, _ = two_layer_convnet(X, model, y, reg=0)

# Sanity check: Loss should be about log(10) = 2.3026
print('Sanity check loss (no regularization): ', loss)

# Sanity check: Loss should go up when you add regularization
loss, _ = two_layer_convnet(X, model, y, reg=1)
print('Sanity check loss (with regularization): ', loss)
```

```
Sanity check loss (no regularization):  2.302596029068058
Sanity check loss (with regularization):  2.3446760601234597
```

3 Gradient check

After the loss looks reasonable, you should always use numeric gradient checking to make sure that your backward pass is correct. When you use numeric gradient checking you should use a small amount of artifical data and a small number of neurons at each layer.

[6]:
```
num_inputs = 2
input_shape = (3, 16, 16)
reg = 0.0
num_classes = 10
X = np.random.randn(num_inputs, *input_shape)
y = np.random.randint(num_classes, size=num_inputs)

model = init_two_layer_convnet(num_filters=3, filter_size=3,␣
 ↪input_shape=input_shape)
loss, grads = two_layer_convnet(X, model, y)
for param_name in sorted(grads):
    f = lambda _: two_layer_convnet(X, model, y)[0]
    param_grad_num = eval_numerical_gradient(f, model[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: 3.047767e-06
W2 max relative error: 1.008414e-05
b1 max relative error: 3.320790e-08
b2 max relative error: 1.653092e-09
```

4 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.

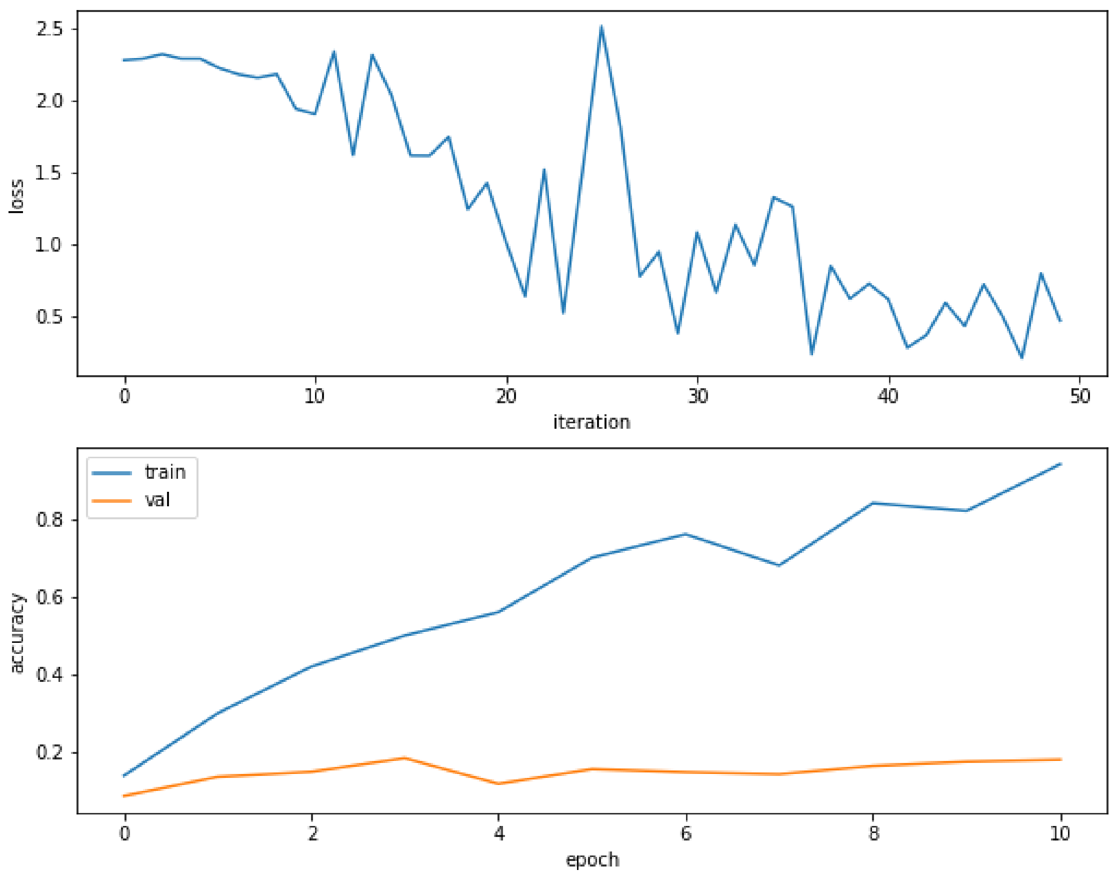
[7]:
```
# Use a two-layer ConvNet to overfit 50 training examples.

model = init_two_layer_convnet()
trainer = ClassifierTrainer()
best_model, loss_history, train_acc_history, val_acc_history = trainer.train(
          X_train[:50], y_train[:50], X_val, y_val, model, two_layer_convnet,
          reg=0.001, momentum=0.9, learning_rate=0.0001, batch_size=10,␣
↪num_epochs=10,
          verbose=True)
```

```
starting iteration  0
Finished epoch 0 / 10: cost 2.280898, train: 0.140000, val 0.088000, lr
1.000000e-04
Finished epoch 1 / 10: cost 2.290638, train: 0.300000, val 0.137000, lr
9.500000e-05
Finished epoch 2 / 10: cost 1.941838, train: 0.420000, val 0.150000, lr
9.025000e-05
starting iteration  10
Finished epoch 3 / 10: cost 2.038384, train: 0.500000, val 0.185000, lr
8.573750e-05
Finished epoch 4 / 10: cost 1.424962, train: 0.560000, val 0.119000, lr
8.145062e-05
starting iteration  20
Finished epoch 5 / 10: cost 1.506822, train: 0.700000, val 0.157000, lr
7.737809e-05
Finished epoch 6 / 10: cost 0.378818, train: 0.760000, val 0.149000, lr
7.350919e-05
starting iteration  30
Finished epoch 7 / 10: cost 1.325558, train: 0.680000, val 0.144000, lr
6.983373e-05
Finished epoch 8 / 10: cost 0.723883, train: 0.840000, val 0.165000, lr
6.634204e-05
starting iteration  40
```

```
Finished epoch 9 / 10: cost 0.428771, train: 0.820000, val 0.176000, lr
6.302494e-05
Finished epoch 10 / 10: cost 0.466106, train: 0.940000, val 0.181000, lr
5.987369e-05
finished optimization. best validation accuracy: 0.185000
```

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

[8]:
```
plt.subplot(2, 1, 1)
plt.plot(loss_history)
plt.xlabel('iteration')
plt.ylabel('loss')

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



5 Train the net

Once the above works, training the net is the next thing to try. You can set the `acc_frequency` parameter to change the frequency at which the training and validation set accuracies are tested. If your parameters are set properly, you should see the training and validation accuracy start to improve within a hundred iterations, and you should be able to train a reasonable model with just one epoch.

Using the parameters below you should be able to get around 50% accuracy on the validation set.

[9]:
```
model = init_two_layer_convnet(filter_size=7)
trainer = ClassifierTrainer()
best_model, loss_history, train_acc_history, val_acc_history = trainer.train(
          X_train, y_train, X_val, y_val, model, two_layer_convnet,
          reg=0.001, momentum=0.9, learning_rate=0.0001, batch_size=50,␣
 ↪num_epochs=1,
          acc_frequency=50, verbose=True)
```

```
starting iteration  0
Finished epoch 0 / 1: cost 2.307156, train: 0.078000, val 0.093000, lr
1.000000e-04
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
Finished epoch 0 / 1: cost 1.730877, train: 0.301000, val 0.313000, lr
1.000000e-04
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
starting iteration  100
Finished epoch 0 / 1: cost 1.635925, train: 0.376000, val 0.396000, lr
1.000000e-04
starting iteration  110
starting iteration  120
starting iteration  130
starting iteration  140
starting iteration  150
Finished epoch 0 / 1: cost 1.315248, train: 0.410000, val 0.426000, lr
1.000000e-04
starting iteration  160
starting iteration  170
starting iteration  180
starting iteration  190
starting iteration  200
Finished epoch 0 / 1: cost 1.557095, train: 0.433000, val 0.419000, lr
```

```
1.000000e-04
starting iteration  210
starting iteration  220
starting iteration  230
starting iteration  240
starting iteration  250
Finished epoch 0 / 1: cost 1.663908, train: 0.418000, val 0.428000, lr
1.000000e-04
starting iteration  260
starting iteration  270
starting iteration  280
starting iteration  290
starting iteration  300
Finished epoch 0 / 1: cost 1.382058, train: 0.415000, val 0.388000, lr
1.000000e-04
starting iteration  310
starting iteration  320
starting iteration  330
starting iteration  340
starting iteration  350
Finished epoch 0 / 1: cost 1.608989, train: 0.439000, val 0.417000, lr
1.000000e-04
starting iteration  360
starting iteration  370
starting iteration  380
starting iteration  390
starting iteration  400
Finished epoch 0 / 1: cost 1.741164, train: 0.494000, val 0.436000, lr
1.000000e-04
starting iteration  410
starting iteration  420
starting iteration  430
starting iteration  440
starting iteration  450
Finished epoch 0 / 1: cost 1.561337, train: 0.454000, val 0.448000, lr
1.000000e-04
starting iteration  460
starting iteration  470
starting iteration  480
starting iteration  490
starting iteration  500
Finished epoch 0 / 1: cost 1.512908, train: 0.485000, val 0.443000, lr
1.000000e-04
starting iteration  510
starting iteration  520
starting iteration  530
starting iteration  540
starting iteration  550
```

```
Finished epoch 0 / 1: cost 2.036114, train: 0.434000, val 0.431000, lr
1.000000e-04
starting iteration  560
starting iteration  570
starting iteration  580
starting iteration  590
starting iteration  600
Finished epoch 0 / 1: cost 1.244206, train: 0.476000, val 0.447000, lr
1.000000e-04
starting iteration  610
starting iteration  620
starting iteration  630
starting iteration  640
starting iteration  650
Finished epoch 0 / 1: cost 1.884902, train: 0.466000, val 0.446000, lr
1.000000e-04
starting iteration  660
starting iteration  670
starting iteration  680
starting iteration  690
starting iteration  700
Finished epoch 0 / 1: cost 1.754295, train: 0.459000, val 0.476000, lr
1.000000e-04
starting iteration  710
starting iteration  720
starting iteration  730
starting iteration  740
starting iteration  750
Finished epoch 0 / 1: cost 2.353327, train: 0.467000, val 0.472000, lr
1.000000e-04
starting iteration  760
starting iteration  770
starting iteration  780
starting iteration  790
starting iteration  800
Finished epoch 0 / 1: cost 1.850225, train: 0.463000, val 0.461000, lr
1.000000e-04
starting iteration  810
starting iteration  820
starting iteration  830
starting iteration  840
starting iteration  850
Finished epoch 0 / 1: cost 1.859495, train: 0.457000, val 0.441000, lr
1.000000e-04
starting iteration  860
starting iteration  870
starting iteration  880
starting iteration  890
```

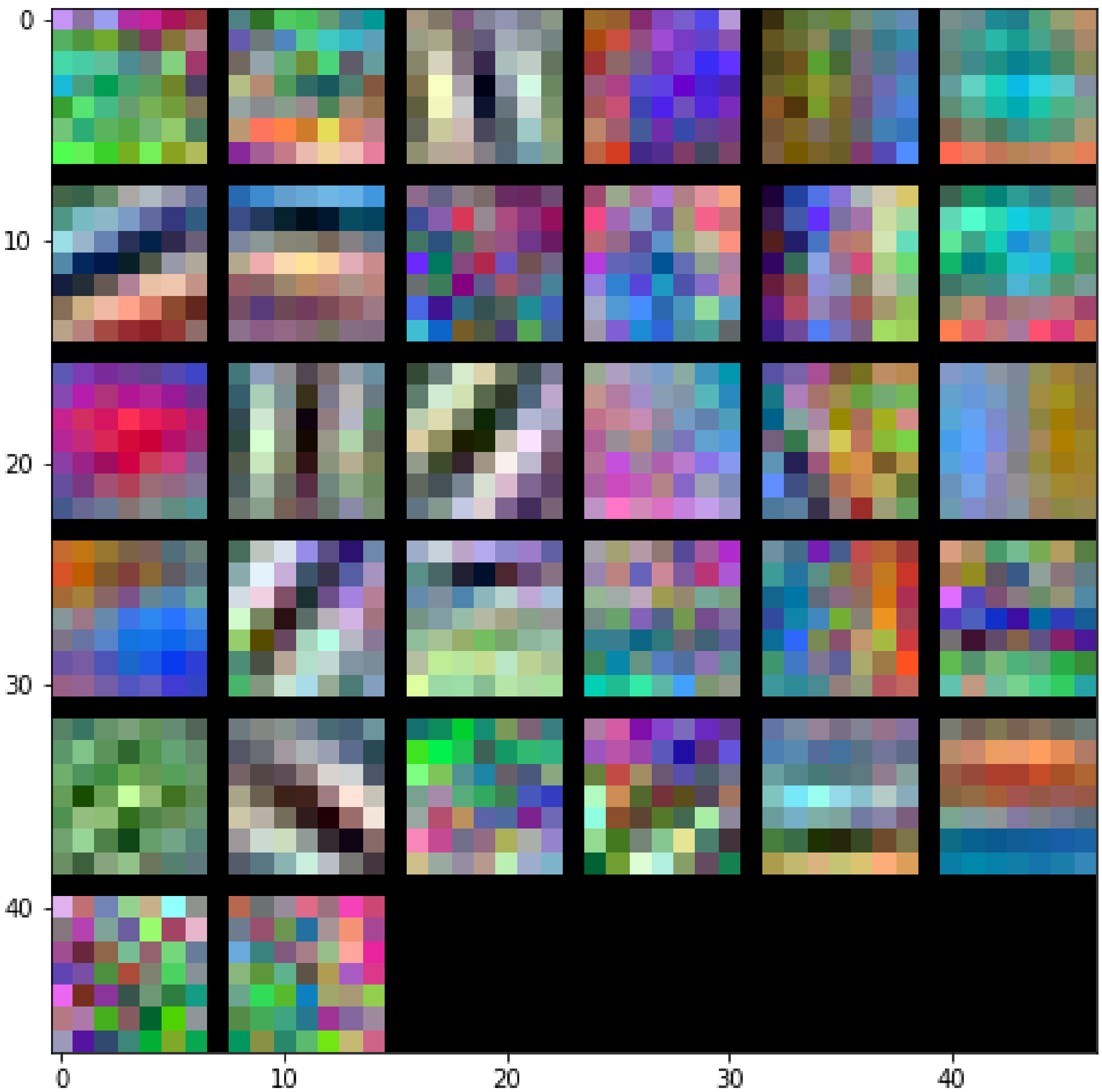
```
starting iteration  900
Finished epoch 0 / 1: cost 1.547902, train: 0.462000, val 0.488000, lr
1.000000e-04
starting iteration  910
starting iteration  920
starting iteration  930
starting iteration  940
starting iteration  950
Finished epoch 0 / 1: cost 1.704826, train: 0.441000, val 0.458000, lr
1.000000e-04
starting iteration  960
starting iteration  970
Finished epoch 1 / 1: cost 1.784207, train: 0.490000, val 0.458000, lr
9.500000e-05
finished optimization. best validation accuracy: 0.488000
```

6 Visualize weights

We can visualize the convolutional weights from the first layer. If everything worked properly, these will usually be edges and blobs of various colors and orientations.

[10]:
```
from cs231n.vis_utils import visualize_grid

grid = visualize_grid(best_model['W1'].transpose(0, 2, 3, 1))
plt.imshow(grid.astype('uint8'))
```

[10]:
```
<matplotlib.image.AxesImage at 0x113b60e90>
```


0
10
20
30
40
0
10
20
30
40

[ ]:

softmax-classifier

February 11, 2020

0.1 PyTorch data

PyTorch comes with a nice paradigm for dealing with data which we'll use here. A PyTorch `Dataset` knows where to find data in its raw form (files on disk) and how to load individual examples into Python datastructures. A PyTorch `DataLoader` takes a dataset and offers a variety of ways to sample batches from that dataset.

Take a moment to browse through the `CIFAR10 Dataset` in `2_pytorch/cifar10.py`, read the `DataLoader` documentation linked above, and see how these are used in the section of `train.py` that loads data. Note that in the first part of the homework we subtracted a mean CIFAR10 image from every image before feeding it in to our models. Here we subtract a constant color instead. Both methods are seen in practice and work equally well.

PyTorch provides lots of vision datasets which can be imported directly from `torchvision.datasets`. Also see `torchtext` for natural language datasets.

0.2 Softmax Classifier in PyTorch

In PyTorch Deep Learning building blocks are implemented in the neural network module `torch.nn` (usually imported as `nn`). A PyTorch model is typically a subclass of `nn.Module` and thereby gains a multitude of features. Because your logistic regressor is an `nn.Module` all of its parameters and sub-modules are accessible through the `.parameters()` and `.modules()` methods.

Now implement a softmax classifier by filling in the marked sections of `models/softmax.py`.

The main driver for this question is `train.py`. It reads arguments and model hyperparameter from the command line, loads CIFAR10 data and the specified model (in this case, softmax). Using the optimizer initialized with appropriate hyperparameters, it trains the model and reports performance on test data.

Complete the following couple of sections in `train.py`: 1. Initialize an optimizer from the torch.optim package 2. Update the parameters in model using the optimizer initialized above

At this point all of the components required to train the softmax classifer are complete for the softmax classifier. Now run

```
$ run_softmax.sh
```

to train a model and save it to `softmax.pt`. This will also produce a `softmax.log` file which contains training details which we will visualize below.

Note: You may want to adjust the hyperparameters specified in `run_softmax.sh` to get reasonable performance.

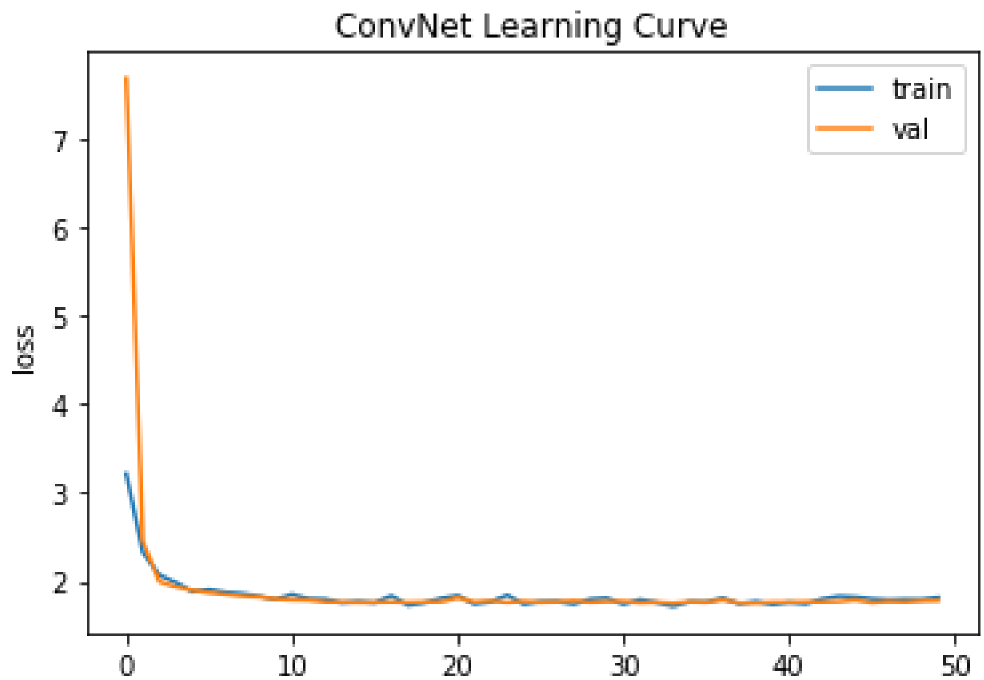
0.3 Visualizing the PyTorch model

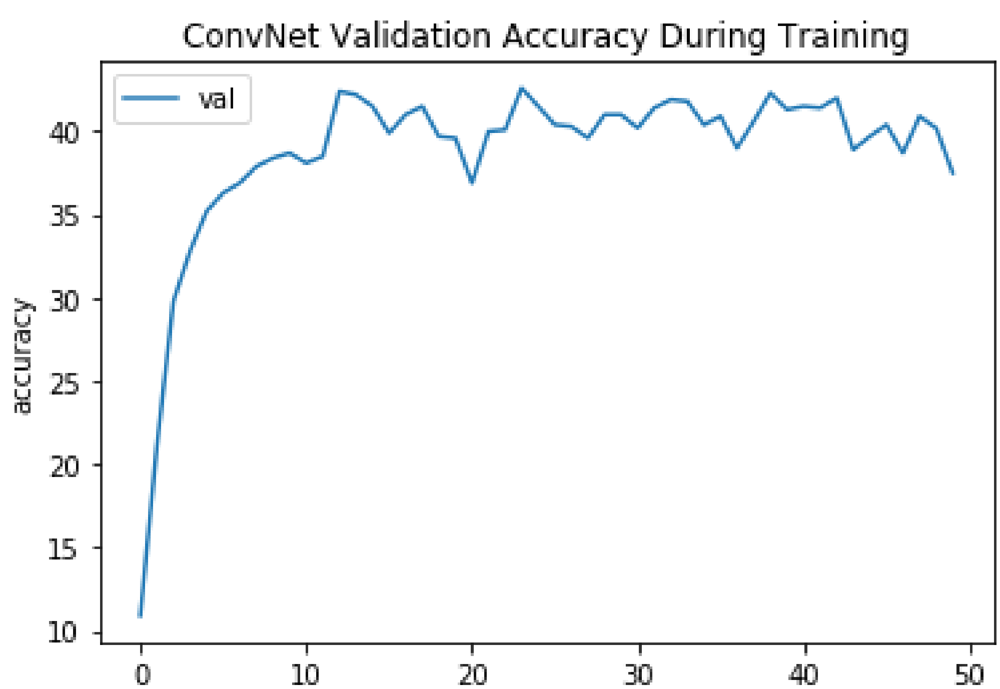
[9]:
```
# Assuming that you have completed training the classifer, let us plot the␣
↪training loss vs. iteration. This is an
# example to show a simple way to log and plot data from PyTorch.

# we neeed matplotlib to plot the graphs for us!
import matplotlib
# This is needed to save images
matplotlib.use('Agg')
import matplotlib.pyplot as plt
%matplotlib inline
```

[10]:
```
# Parse the train and val losses one line at a time.
import re
# regexes to find train and val losses on a line
float_regex = r'[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?'
train_loss_re = re.compile('.*Train Loss: ({})'.format(float_regex))
val_loss_re = re.compile('.*Val Loss: ({})'.format(float_regex))
val_acc_re = re.compile('.*Val Acc: ({})'.format(float_regex))
# extract one loss for each logged iteration
train_losses = []
val_losses = []
val_accs = []
# NOTE: You may need to change this file name.
with open('convnet.log', 'r') as f:
    for line in f:
        train_match = train_loss_re.match(line)
        val_match = val_loss_re.match(line)
        val_acc_match = val_acc_re.match(line)
        if train_match:
            train_losses.append(float(train_match.group(1)))
        if val_match:
            val_losses.append(float(val_match.group(1)))
        if val_acc_match:
            val_accs.append(float(val_acc_match.group(1)))
```

[11]:
```
fig = plt.figure()
plt.plot(train_losses, label='train')
plt.plot(val_losses, label='val')
plt.title('ConvNet Learning Curve')
plt.ylabel('loss')
plt.legend()
fig.savefig('convnet_lossvstrain.png')

fig = plt.figure()
plt.plot(val_accs, label='val')
```

```
plt.title('ConvNet Validation Accuracy During Training')
plt.ylabel('accuracy')
plt.legend()
fig.savefig('convnet_valaccuracy.png')
```





```
[ ]:
```

filter-viz

February 11, 2020

0.1 Visualizing the trained filters

[14]:
```python
# some startup!
import numpy as np
import matplotlib
# This is needed to save images
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import torch
```

[32]:
```python
# load the model saved by train.py
# This will be an instance of models.softmax.Softmax.
# NOTE: You may need to change this file name.
softmax_model = torch.load('convnet.pt')
```

[38]:
```python
# collect all the weights
w = None

w = softmax_model.conv.weight.data.numpy().transpose(0,2,3,1)
print(w.shape)

w_min, w_max = np.min(w), np.max(w)
# classes
classes = ['plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse',␣
↪'ship', 'truck']
# init figure
fig = plt.figure(figsize=(6,6))
for i in range(10):
    wimg = 255.0*(w[i].squeeze() - w_min) / (w_max - w_min)
    fig.add_subplot(9,2,i+1).imshow(wimg.astype('uint8'))
# save fig!
fig.savefig('convnet_filt.png')
print('figure saved')
```

```
(10, 1, 1, 3)
```

```
/Users/ssipani/miniconda3/envs/cs7643/lib/python3.7/site-
packages/ipykernel_launcher.py:11: RuntimeWarning: More than 20 figures have
been opened. Figures created through the pyplot interface
```

```
(`matplotlib.pyplot.figure`) are retained until explicitly closed and may
consume too much memory. (To control this warning, see the rcParam
`figure.max_open_warning`).
  # This is added back by InteractiveShellApp.init_path()


 ↪---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call␣
 ↪last)

        <ipython-input-38-0900badacd3d> in <module>
         12 for i in range(10):
         13     wimg = 255.0*(w[i].squeeze() - w_min) / (w_max - w_min)
    ---> 14     fig.add_subplot(9,2,i+1).imshow(wimg.astype('uint8'))
         15 # save fig!
         16 fig.savefig('convnet_filt.png')


        ~/miniconda3/envs/cs7643/lib/python3.7/site-packages/matplotlib/__init__.
 ↪py in inner(ax, data, *args, **kwargs)
       1597     def inner(ax, *args, data=None, **kwargs):
       1598         if data is None:
    -> 1599             return func(ax, *map(sanitize_sequence, args), **kwargs)
       1600
       1601         bound = new_sig.bind(ax, *args, **kwargs)


        ~/miniconda3/envs/cs7643/lib/python3.7/site-packages/matplotlib/cbook/
 ↪deprecation.py in wrapper(*args, **kwargs)
        367                 f"%(removal)s.  If any parameter follows {name!r},␣
 ↪they "
        368                 f"should be pass as keyword, not positionally.")
    --> 369         return func(*args, **kwargs)
        370
        371     return wrapper


        ~/miniconda3/envs/cs7643/lib/python3.7/site-packages/matplotlib/cbook/
 ↪deprecation.py in wrapper(*args, **kwargs)
        367                 f"%(removal)s.  If any parameter follows {name!r},␣
 ↪they "
        368                 f"should be pass as keyword, not positionally.")
    --> 369         return func(*args, **kwargs)
        370
        371     return wrapper
```

```
~/miniconda3/envs/cs7643/lib/python3.7/site-packages/matplotlib/axes/
↪_axes.py in imshow(self, X, cmap, norm, aspect, interpolation, alpha, vmin,␣
↪vmax, origin, extent, shape, filternorm, filterrad, imlim, resample, url,␣
↪**kwargs)
   5677                                 resample=resample, **kwargs)
   5678
-> 5679         im.set_data(X)
   5680         im.set_alpha(alpha)
   5681         if im.get_clip_path() is None:


~/miniconda3/envs/cs7643/lib/python3.7/site-packages/matplotlib/image.py␣
↪in set_data(self, A)
    688                 or self._A.ndim == 3 and self._A.shape[-1] in [3,␣
↪4]):
    689             raise TypeError("Invalid shape {} for image data"
--> 690                             .format(self._A.shape))
    691
    692         if self._A.ndim == 3:


TypeError: Invalid shape (3,) for image data
```

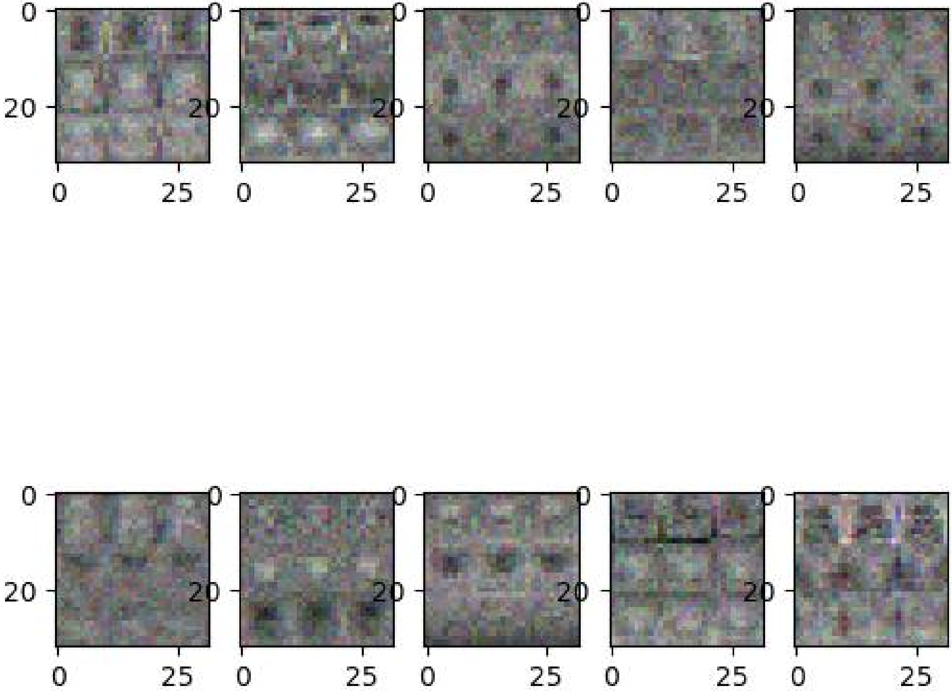
[39]:

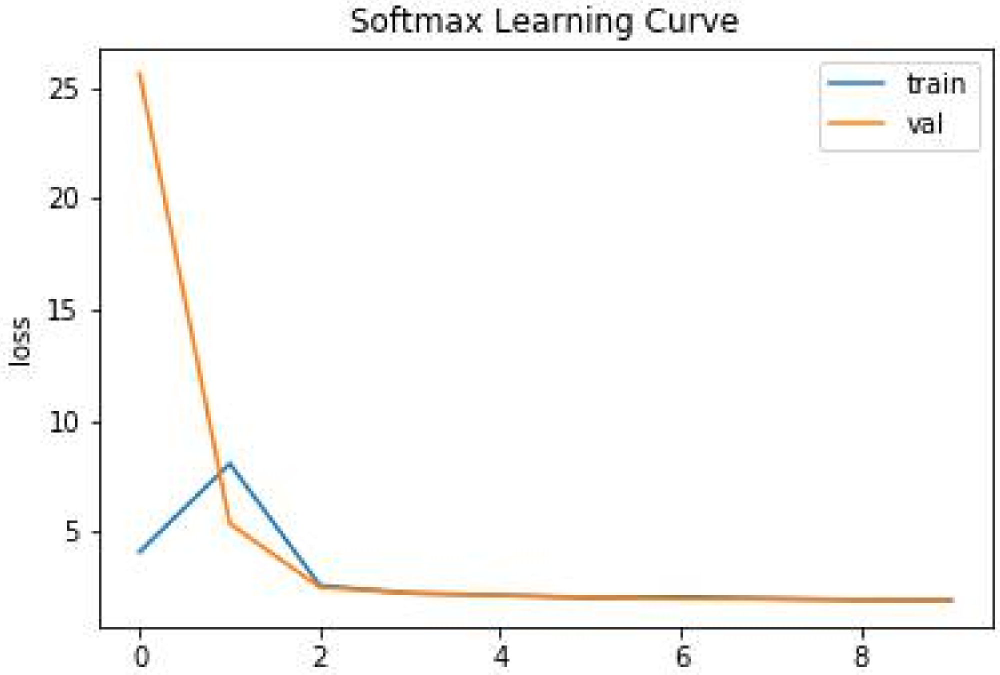
```
# vis_utils.py has helper code to view multiple filters in single image. Use␣
↪this to visuzlize
# neural network adn convnets.
# import vis_utils
from vis_utils import visualize_grid
# saving the weights is now as simple as:
plt.imsave('convnet_gridfilt.png',visualize_grid(w, padding=3).astype('uint8'))
# padding is the space between images. Make sure that w is of shape: (N,H,W,C)
print('figure saved as a grid!')
```

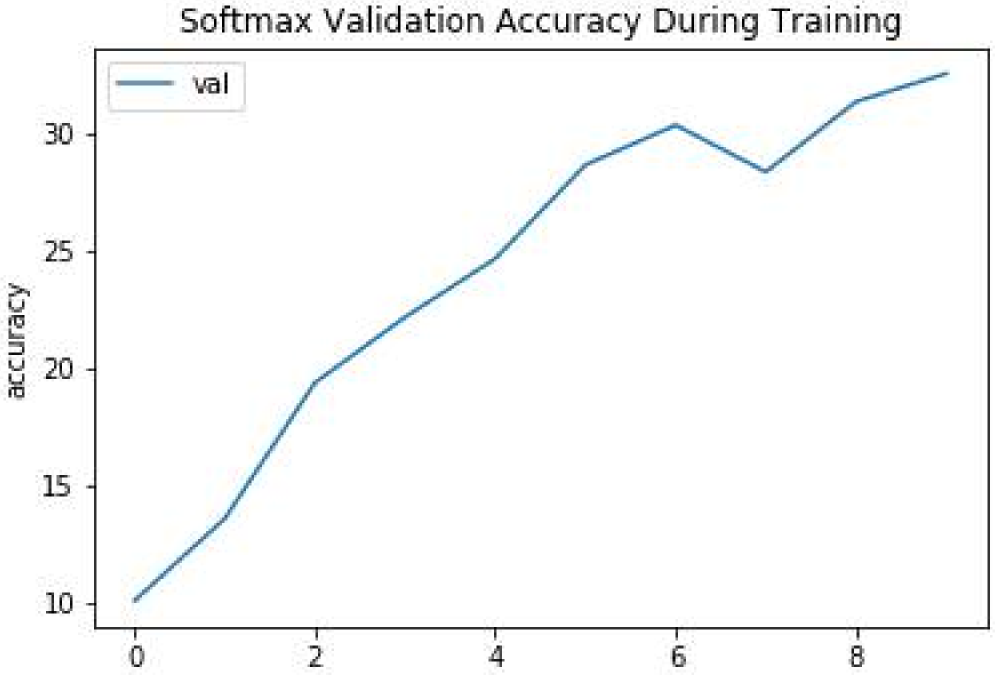
```
figure saved as a grid!
```

[ ]:

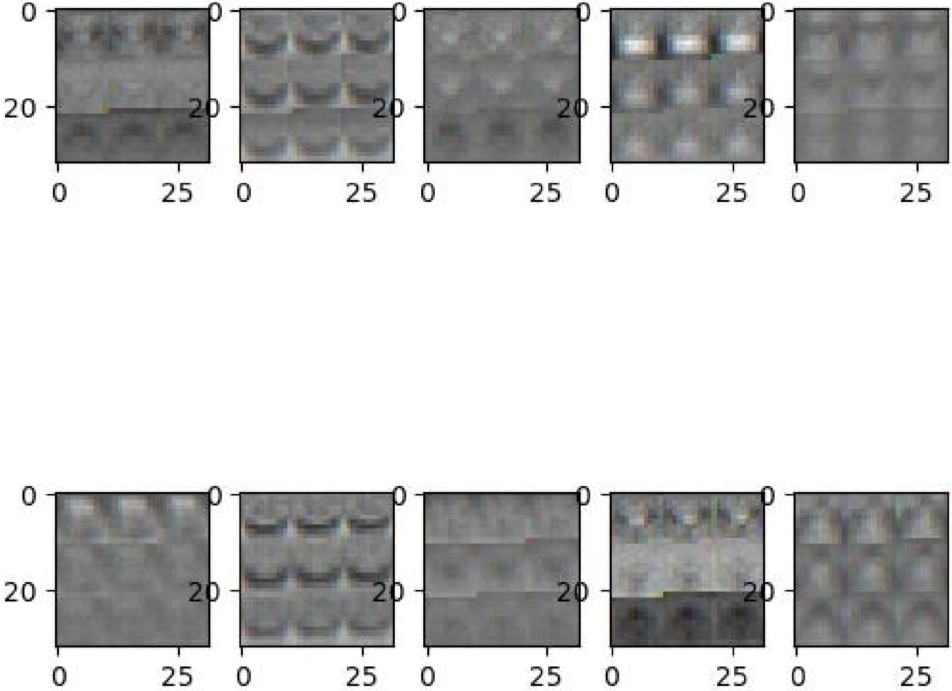
```

```



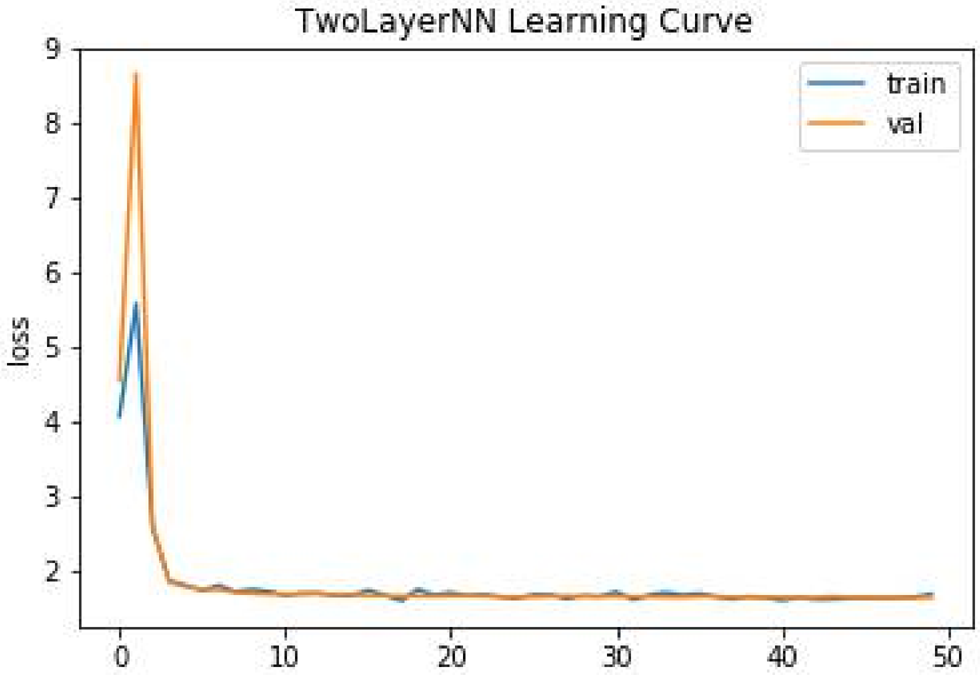

Softmax Learning Curve
loss
25
20
15
10
5
0
2
4
6
8
train
val

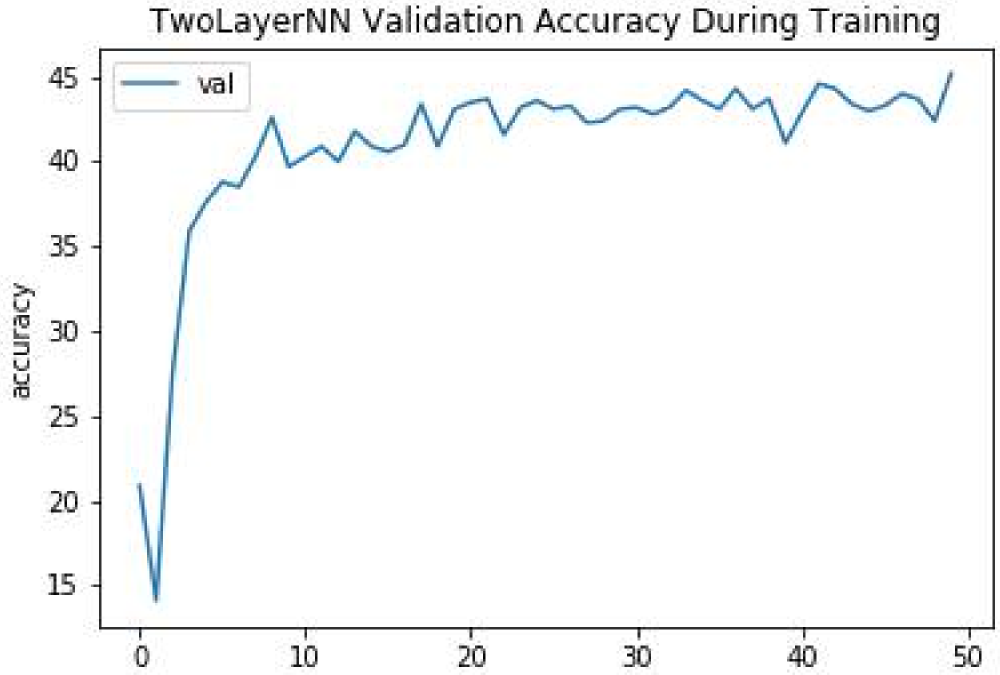


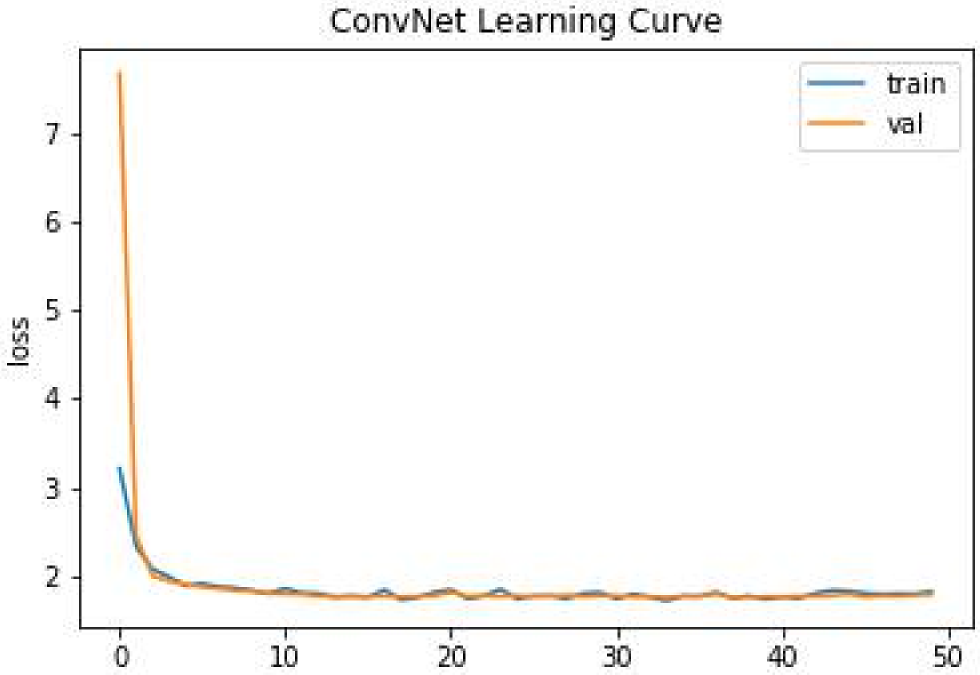
Softmax Validation Accuracy During Training
val
accuracy
10
15
20
25
30
0
2
4
6
8

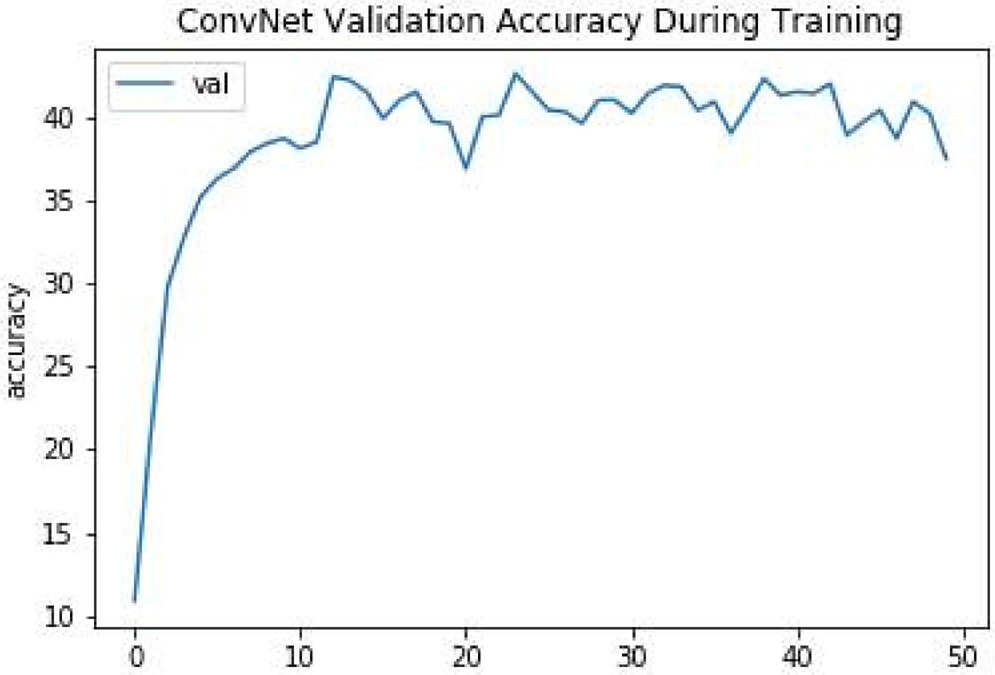


0
20
0
25


TwoLayerNN Learning Curve
loss
train
val
9
8
7
6
5
4
3
2
0
10
20
30
40
50


TwoLayerNN Validation Accuracy During Training
val
accuracy
45
40
35
30
25
20
15
0
10
20
30
40
50


ConvNet Learning Curve
train
val
loss
7
6
5
4
3
2
0
10
20
30
40
50


ConvNet Validation Accuracy During Training
val
accuracy
10
15
20
25
30
35
40
0
10
20
30
40
50