CS7643: Deep Learning
Spring 2020
Problem Set 1

Instructor: Zsolt Kira
TAs: Yihao Chen, Sameer Dharur, Rahul Duggal, Patrick Grady, Harish Kamath
Yinquan Lu, Anishi Mehta, Manas Sahni, Jiachen Yang, Zhuoran Yu
Discussions: `https://piazza.com/gatech/spring2020/cs4803dl7643a/home`

Due: Tuesday, February 11, 11:55pm

Instructions

1. We will be using Gradescope to collect your assignments. Please read the following instructions for submitting to Gradescope carefully!
   - Each subproblem must be submitted on a separate page. When submitting to Gradescope, make sure to mark which page(s) corresponds to each problem/sub-problem. For instance, Q5 has 5 subproblems, and the solution to each must start on a new page. Similarly, Q8 has 8 subproblems, and the writeup for each should start on a new page.
   - For the coding problems (Q8), please use the provided `collect_submission.sh` script and upload `hw1.zip` to the HW1 Code assignment on Gradescope. While we will not be explicitly grading your code, you are still required to submit it. Please make sure you have saved the most recent version of your jupyter notebook before running this script. Further, append the writeup for each Q8 subproblem to your PS1 solution PDF.
   - Note: This is a large class and Gradescope's assignment segmentation features are essential. Failure to follow these instructions may result in parts of your assignment not being graded. We will not entertain regrading requests for failure to follow instructions.
2. LaTeX'd solutions are strongly encouraged (solution template available at cc.gatech.edu/classes/AY2020/cs7643_fall/assets/sol1.tex), but scanned handwritten copies are acceptable. Hard copies are **not** accepted.
3. We generally encourage you to collaborate with other students.

   You may talk to a friend, discuss the questions and potential directions for solving them. However, you need to write your own solutions and code separately, and *not* as a group activity. Please list the students you collaborated with.

1 Gradient Descent

1. (3 points) We often use iterative optimization algorithms such as Gradient Descent to find $\mathbf{w}$ that minimizes a loss function $f(\mathbf{w})$. Recall that in gradient descent, we start with an initial

value of $\mathbf{w}$ (say $\mathbf{w}^{(1)}$) and iteratively take a step in the direction of the negative of the gradient of the objective function *i.e.*

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla f(\mathbf{w}^{(t)}) \tag{1}$$

for learning rate $\eta > 0$.

In this question, we will develop a slightly deeper understanding of this update rule, in particular for minimizing a convex function $f(\mathbf{w})$. Note: this analysis will not directly carry over to training neural networks since loss functions for training neural networks are typically not convex, but this will (a) develop intuition and (b) provide a starting point for research in non-convex optimization (which is beyond the scope of this class).

Recall the first-order Taylor approximation of f at $\mathbf{w}^{(t)}$:

$$f(\mathbf{w}) \approx f(\mathbf{w}^{(t)}) + \langle \mathbf{w} - \mathbf{w}^{(t)}, \nabla f(\mathbf{w}^{(t)}) \rangle \tag{2}$$

When f is convex, this approximation forms a lower bound of f, *i.e.*

$$f(\mathbf{w}) \geq \underbrace{f(\mathbf{w}^{(t)}) + \langle \mathbf{w} - \mathbf{w}^{(t)}, \nabla f(\mathbf{w}^{(t)}) \rangle}_{\text{affine lower bound to } f(\cdot)} \quad \forall \mathbf{w} \tag{3}$$

Since this approximation is a 'simpler' function than $f(\cdot)$, we could consider minimizing the approximation instead of $f(\cdot)$. Two immediate problems: (1) the approximation is affine (thus unbounded from below) and (2) the approximation is faithful for $\mathbf{w}$ close to $\mathbf{w}^{(t)}$. To solve both problems, we add a squared ℓ_2 *proximity term* to the approximation minimization:

$$\underset{\mathbf{w}}{\operatorname{argmin}} \underbrace{f(\mathbf{w}^{(t)}) + \langle \mathbf{w} - \mathbf{w}^{(t)}, \nabla f(\mathbf{w}^{(t)}) \rangle}_{\text{affine lower bound to } f(\cdot)} + \underbrace{\frac{\lambda}{2}}_{\text{trade-off}} \underbrace{\left\|\mathbf{w} - \mathbf{w}^{(t)}\right\|^2}_{\text{proximity term}} \tag{4}$$

Notice that the optimization problem above is an unconstrained quadratic programming problem, meaning that it can be solved in closed form (hint: gradients).

What is the solution $\mathbf{w}^*$ of the above optimization? What does that tell you about the gradient descent update rule? What is the relationship between λ and η?
Answer: Take the derivative of w:

$$f'(w) = \nabla f(\mathbf{w}^{(t)} + \lambda(\mathbf{w} - \mathbf{w}^{(t)}) = 0 \tag{5}$$

$$\mathbf{w}^* = \mathbf{w}^{(t)} - \frac{1}{\lambda} \nabla f(\mathbf{w}^{(t)}) \tag{6}$$

$$\eta = \frac{1}{\lambda} \tag{7}$$

Gradient decent can minimize the approximation function at each step. The value of λ means the penalty of proximity term. When λ increases, the learning rate will decrease, which means the step along GD can be small, and vice versa.

2. (3 points) Let's prove a lemma that will initially seem devoid of the rest of the analysis but will come in handy in the next sub-question when we start combining things. Specifically, the analysis in this sub-question holds for any $\mathbf{w}^\star$, but in the next sub-question we will use it for $\mathbf{w}^\star$ that minimizes $f(\mathbf{w})$.

Consider a sequence of vectors $\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_T$, and an update equation of the form $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \mathbf{v}_t$ with $\mathbf{w}^{(1)} = 0$. Show that:

$$\sum_{t=1}^{T} \langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \mathbf{v}_t \rangle \leq \frac{\|\mathbf{w}^\star\|^2}{2\eta} + \frac{\eta}{2} \sum_{t=1}^{T} \|\mathbf{v}_t\|^2 \tag{8}$$

Answer:

$$\begin{aligned}
\langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \mathbf{v}_t \rangle &= \frac{1}{\eta} \langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \eta \mathbf{v}_t \rangle & (9)\\
&= \left\|\mathbf{w}^{(t)} - \mathbf{w}^\star\right\|^2 + \|\eta \mathbf{v}_t\|^2 + \left\|\mathbf{w}^{(t)} - \mathbf{w}^\star - \eta \mathbf{v}_t\right\|^2 & (10)\\
&= \frac{1}{4\eta} (\left\|\mathbf{w}^{(t)} - \mathbf{w}^\star + \eta \mathbf{v}_t\right\|^2 - \left\|\mathbf{w}^{(t)} - \mathbf{w}^\star - \eta \mathbf{v}_t\right\|^2) & (11)\\
&= \frac{1}{2\eta} (\left\|\mathbf{w}^{(t)} - \mathbf{w}^\star\right\|^2 - \left\|\mathbf{w}^{(t+1)} - \mathbf{w}^\star\right\|^2) + \frac{\eta}{2} \|\mathbf{v}_t\|^2 & (12)
\end{aligned}$$

$$\sum_{t=1}^{T} \langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \mathbf{v}_t \rangle = \sum_{t=1}^{T} \frac{1}{2\eta} (\left\|\mathbf{w}^{(t)} - \mathbf{w}^\star\right\|^2 - \left\|\mathbf{w}^{(t+1)} - \mathbf{w}^\star\right\|^2) + \frac{\eta}{2} \sum_{t=1}^{T} \|\mathbf{v}_t\|^2 \tag{13}$$

we can find:

$$\sum_{t=1}^{T} \frac{1}{2\eta} (\left\|\mathbf{w}^{(t)} - \mathbf{w}^\star\right\|^2 - \left\|\mathbf{w}^{(t+1)} - \mathbf{w}^\star\right\|^2) = \frac{1}{2\eta} (\|\mathbf{w}^\star\|^2 - \left\|\mathbf{w}^{(t+1)} - \mathbf{w}^\star\right\|^2) \leq \frac{\|\mathbf{w}^\star\|^2}{2\eta} \tag{14}$$

Thus we can prove:

$$\sum_{t=1}^{T} \langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \mathbf{v}_t \rangle \leq \frac{\|\mathbf{w}^\star\|^2}{2\eta} + \frac{\eta}{2} \sum_{t=1}^{T} \|\mathbf{v}_t\|^2 \tag{15}$$

3. (3 points) Now let's start putting things together and analyze the convergence rate of gradient descent *i.e.* how fast it converges to $\mathbf{w}^\star$.

   First, show that for $\bar{\mathbf{w}} = \frac{1}{T} \sum_{t=1}^{T} \mathbf{w}^{(t)}$

$$f(\bar{\mathbf{w}}) - f(\mathbf{w}^\star) \leq \frac{1}{T} \sum_{t=1}^{T} \langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \nabla f(\mathbf{w}^{(t)}) \rangle \tag{16}$$

   Next, use the result from part 2, with upper bounds B and ρ for $\|\mathbf{w}^\star\|$ and $\left\|\nabla f(\mathbf{w}^{(t)})\right\|$ respectively and show that for fixed $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$, the convergence rate of gradient descent is $\mathcal{O}(1/\sqrt{T})$ *i.e.* the upper bound for $f(\bar{\mathbf{w}}) - f(\mathbf{w}^\star) \propto \frac{1}{\sqrt{T}}$.

   Answer:

$$f(\bar{\mathbf{w}}) - f(\mathbf{w}^\star) = f(\frac{1}{T}\sum_{t=1}^{T}\mathbf{w}^{(t)}) - f(\mathbf{w}^\star) \tag{17}$$

$$\leq \frac{1}{T} f(\sum_{t=1}^{T}\mathbf{w}^{(t)}) - f(\mathbf{w}^\star) \tag{18}$$

Because of the convexity in question 1, we have:

$$f(\sum_{t=1}^{T}\mathbf{w}^{(t)}) - f(\mathbf{w}^\star) \leq \langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \nabla f(\mathbf{w}^{(t)})\rangle \tag{19}$$

Thus we can prove:

$$f(\bar{\mathbf{w}}) - f(\mathbf{w}^\star) \leq \frac{1}{T}\sum_{t=1}^{T}\langle \mathbf{w}^{(t)} - \mathbf{w}^\star, \nabla f(\mathbf{w}^{(t)})\rangle \tag{20}$$

Using the conclusion from part2, we can get:

$$f(\bar{\mathbf{w}}) - f(\mathbf{w}^\star) \leq \frac{1}{T}(\frac{\|\mathbf{w}^\star\|^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{T}\left\|\nabla f(\mathbf{w}^{(t)}\right\|^2) \tag{21}$$

$$\leq \frac{1}{T}(\frac{B^2}{2\eta} + \frac{\eta}{2}Tp) \tag{22}$$

$$= \frac{B^2}{2\eta T} + \frac{\eta p}{2} \tag{23}$$

$$= \frac{1}{\sqrt{T}}\frac{BP + P}{2} \tag{24}$$

Thus the convergence rate of gradient descent is $\mathcal{O}(1/\sqrt{T})$

4. (2 points) Consider an objective function $f(w) := f_1(w) + f_2(w)$ comprised of $N = 2$ terms:

$$f_1(w) = -\ln\left(1 - \frac{1}{1+\exp(-w)}\right) \quad \text{and} \quad f_2(w) = -\ln\left(\frac{1}{1+\exp(-w)}\right) \tag{25}$$

Now consider using SGD (with a batch-size $B = 1$) to minimize $f(w)$. Specifically, in each iteration, we will pick one of the two terms (uniformly at random), and take a step in the direction of the negative gradient, with a constant step-size of η. You can assume η is small enough that every update does result in improvement (aka descent) on the sampled term. Is SGD guaranteed to decrease the overall loss function in every iteration? If yes, provide a proof. If no, provide a counter-example.

Answer: No, SGD does not guarantee to decrease the objective function at every iteration. For example, let $w^{(0)} = 0$, $w^{(1)} = w^{(0)} - \eta(1 - \frac{1}{(2\mathbf{e}^{-\mathbf{w}_0}(1+)\mathbf{e}^{-\mathbf{w}_0})^3}) = -\frac{\eta}{2}$. Since η is positive, f(w(1)) $>$ f(w(0))

2 Automatic Differentiation

5. (4 points) In practice, writing the closed-form expression of the derivative of a loss function f w.r.t. the parameters of a deep neural network is hard (and mostly unnecessary) as f becomes complex. Instead, we define computation graphs and use the automatic differentiation algorithms (typically backpropagation) to compute gradients using the chain rule. For example, consider the expression

$$f(x, y) = (x + y)(y + 1) \tag{26}$$

Let's define intermediate variables a and b such that

$$a = x + y \tag{27}$$
$$b = y + 1 \tag{28}$$
$$f = a \times b \tag{29}$$

A computation graph for the "forward pass" through f is shown in Fig. 1.

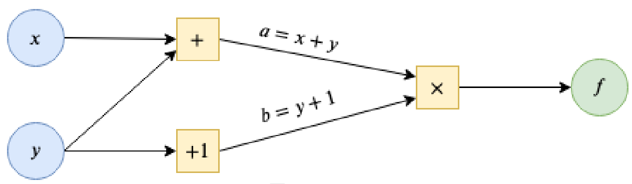


Figure 1

We can then work backwards and compute the derivative of f w.r.t. each intermediate variable ($\frac{\partial f}{\partial a}$ and $\frac{\partial f}{\partial b}$) and chain them together to get $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$.

Let $\sigma(\cdot)$ denote the standard sigmoid function. Now, for the following vector function:

$$f_1(w_1, w_2) = e^{e^{w_1} + e^{2w_2}} + \sigma(e^{w_1} + e^{2w_2}) \tag{30}$$
$$f_2(w_1, w_2) = w_1 w_2 + \max(w_1, w_2) \tag{31}$$

(a) Draw the computation graph. Compute the value of f at $\vec{w} = (1, -1)$.

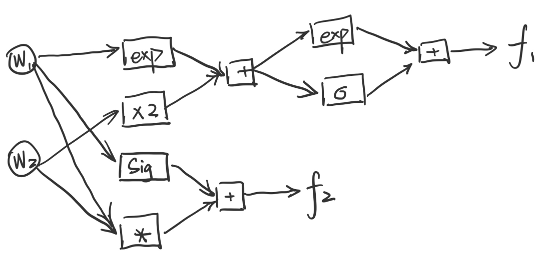


Figure 2

(b) At this $\vec{w}$, compute the Jacobian $\frac{\partial \vec{f}}{\partial \vec{w}}$ using numerical differentiation (using $\Delta w = 0.01$).
Answer:

$$\begin{bmatrix} \frac{\partial f_1}{\partial w_1} & \frac{\partial f_1}{\partial w_2} \\ \frac{\partial f_2}{\partial w_1} & \frac{\partial f_2}{\partial w_2} \end{bmatrix} = \begin{bmatrix} \frac{f_1(w_1+0.01,w_2)-f_1(w_1,w_2)}{0.01} & \frac{f_2(w_1,w_2+0.01)-f_2(w_1,w_2)}{0.01} \\ \frac{f_2(w_1+0.01,w_2)-f_1(w_1,w_2)}{0.01} & \frac{f_2(w_1,w_2+0.01)-f_2(w_1,w_2)}{0.01} \end{bmatrix} = \begin{bmatrix} 48.19 & 4.76 \\ 0 & 1 \end{bmatrix}$$

(c) At this $\vec{w}$, compute the Jacobian using forward mode auto-differentiation.
Answer:

$$\begin{aligned} y_1 &= e^{w_1} && (32) \\ y_2 &= e^{w_1} + e^{2w_2} && (33) \\ y_3 &= e^{y_2} && (34) \\ y_4 &= \sigma(y_3) && (35) \\ \frac{\partial f_1}{\partial w_1} &= \frac{\partial y_5}{\partial w_1} && (36) \\ &= \frac{\partial y_3}{\partial w_1} + \frac{\partial y_4}{\partial w_1} && (37) \\ &= 47.28739 && (38) \end{aligned}$$

Similarly,

$$\begin{aligned} \frac{\partial f_1}{\partial w_2} &= 4.710 && (39) \\ \frac{\partial f_2}{\partial w_1} &= 1 && (40) \\ \frac{\partial f_2}{\partial w_2} &= 0 && (41) \end{aligned}$$

$$\begin{bmatrix} 47.28739 & 4.71 \\ 0 & 1 \end{bmatrix}$$

(d) At this $\vec{w}$, compute the Jacobian using backward mode auto-differentiation.
Answer:

$$\begin{aligned} \frac{\partial f_1}{\partial w_1} &= \vec{a_1}\vec{b_1} && (42) \\ &= \vec{a_1}\frac{exp(a)}{1+exp(a)^2} + \vec{d_2}exp(c_2) && (43) \\ &= 47.28739 && (44) \end{aligned}$$

Similarly,

$$\begin{bmatrix} 47.28739 & 4.71 \\ 0 & 1 \end{bmatrix}$$

(e) Don't you love that software exists to do this for us?

3 Paper Review

The first of our paper reviews for this course comes from a much acclaimed spotlight presentation at NeurIPS 2019 on the topic 'Weight Agnostic Neural Networks' by Adam Gaier and David Ha from Google Brain.
The paper presents a very interesting proposition that, through a series of experiments, re-examines some fundamental notions about neural networks - in particular, the comparative importance of architectures and weights in a network's predictive performance.
The paper can be viewed here. The authors have also written a blog post with intuitive visualizations to help understand its key concepts better.
Guidelines: Please restrict your reviews to no more than 350 words. The evaluation rubric for this section is as follows :

6. (2 points) What is the main contribution of this paper? Briefly summarize its key insights, strengths and weaknesses.
   Contributions:
   In this paper, author starts using deemphasizing weights to search neural network. They aim to search for weight agnostic neural networks, architectures with strong inductive biases that can already perform various tasks with random weights.
   First they created an initial population having minimal neural network topologies. Each rollout has shared weight values and can be used to evaluate the network.Then they can rank these networks. They repeat evaluation and eventually they can create a new population.
   They use 3 models to test, which is CartPoleSwingUp, BipedalWalker-v2 and CarRacing-v0. Researchers found the results were surprisingly good, as the WANN models with the best-performing shared weight values reached an upright pole position on the CartPoleSwingUp task after only after a few swings. Experiment results also proved that WANNs are no match for convolutional neural networks, which was an expected outcome.

7. (2 points) What is your personal takeaway from this paper? This could be expressed either in terms of relating the approaches adopted in this paper to your traditional understanding of learning parameterized models, or potential future directions of research in the area which the authors haven't addressed, or anything else that struck you as being noteworthy.
   Personal takeaway:

   This is a brand new method for searching neural network without using gradient descent. It's not like traditional neural network and it may lessen the computational resources. As with the age-old ânature versus nurtureâ debate, AI researchers want to know whether architecture or weights play the main role in the performance of neural networks. This paper definitely provide a promising start. For me, I think it is interesting but still need to bring it to actual practice to show its performance of the untrained neural network.

4 Implement and train a network on CIFAR-10

Setup Instructions: Before attempting this question, look at setup instructions at here.

8. (Upto 29 points) Now, we will learn how to implement a softmax classifier, vanilla neural networks (or Multi-Layer Perceptrons), and ConvNets. You will begin by writing the forward and backward passes for different types of layers (including convolution and pooling), and

then go on to train a shallow ConvNet on the CIFAR-10 dataset in Python. Next you will learn to use PyTorch, a popular open-source deep learning framework, and use it to replicate the experiments from before.

Follow the instructions provided here

softmax

February 10, 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

[127]:
```
# 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
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

[128]:
```
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)
```

```python
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`

[144]:
```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)
```

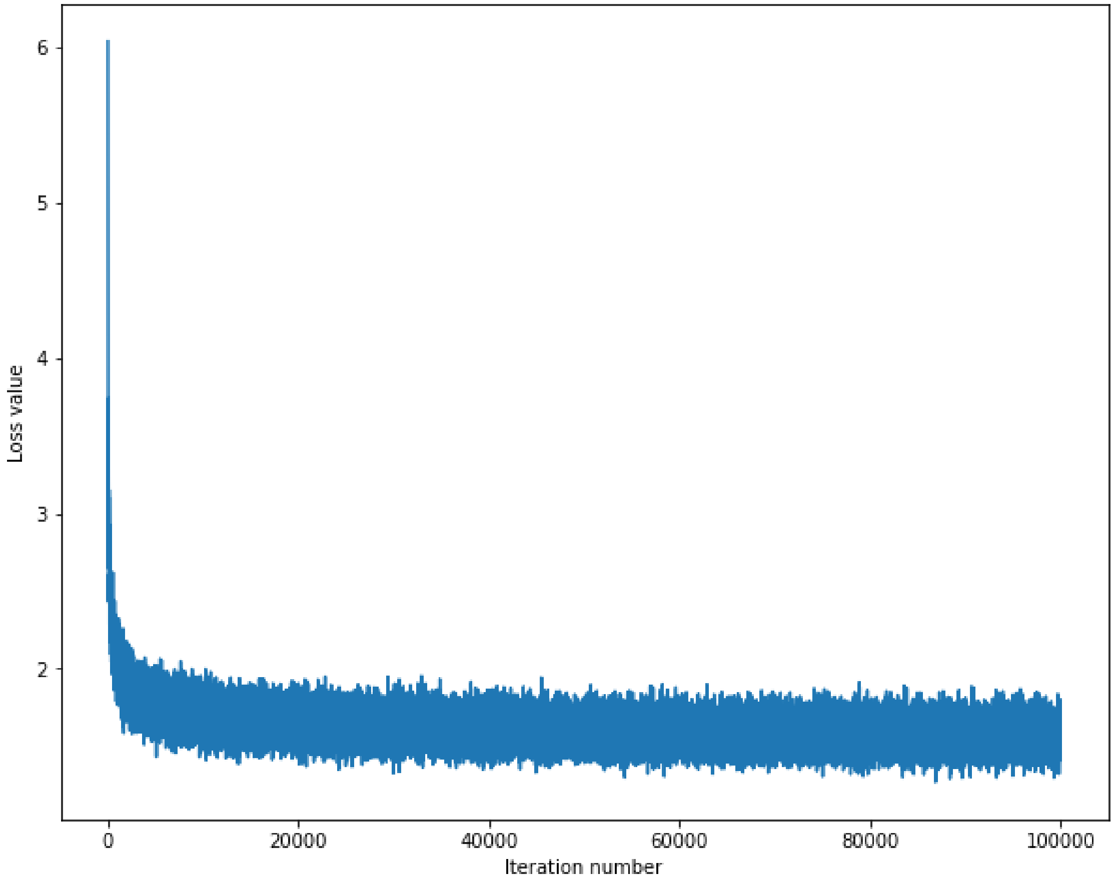
```
vectorized loss: 2.383807e+00 computed in 0.147605s
loss: 2.383807
sanity check: 2.302585
numerical: -2.929964 analytic: -2.929964, relative error: 4.616225e-09
numerical: -3.135085 analytic: -3.135085, relative error: 2.552106e-08
numerical: -0.570058 analytic: -0.570058, relative error: 1.404715e-07
numerical: 0.337006 analytic: 0.337006, relative error: 4.889899e-08
numerical: 1.682049 analytic: 1.682049, relative error: 2.496364e-08
numerical: -0.297519 analytic: -0.297519, relative error: 8.559922e-08
numerical: 0.558544 analytic: 0.558544, relative error: 2.837774e-08
numerical: 0.629610 analytic: 0.629610, relative error: 2.700313e-08
```

```
numerical: -2.363835 analytic: -2.363835, relative error: 1.094664e-08
numerical: -1.217689 analytic: -1.217689, relative error: 2.245683e-09
```

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

[219]:
```
# 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-6,
    reg=1e-5,
    num_iters=100,
    batch_size=200,
    verbose=False,
)
# Plot loss vs. iterations
plt.plot(loss_hist)
plt.xlabel("Iteration number")
plt.ylabel("Loss value")
```

[219]:
```
Text(0, 0.5, 'Loss value')
```

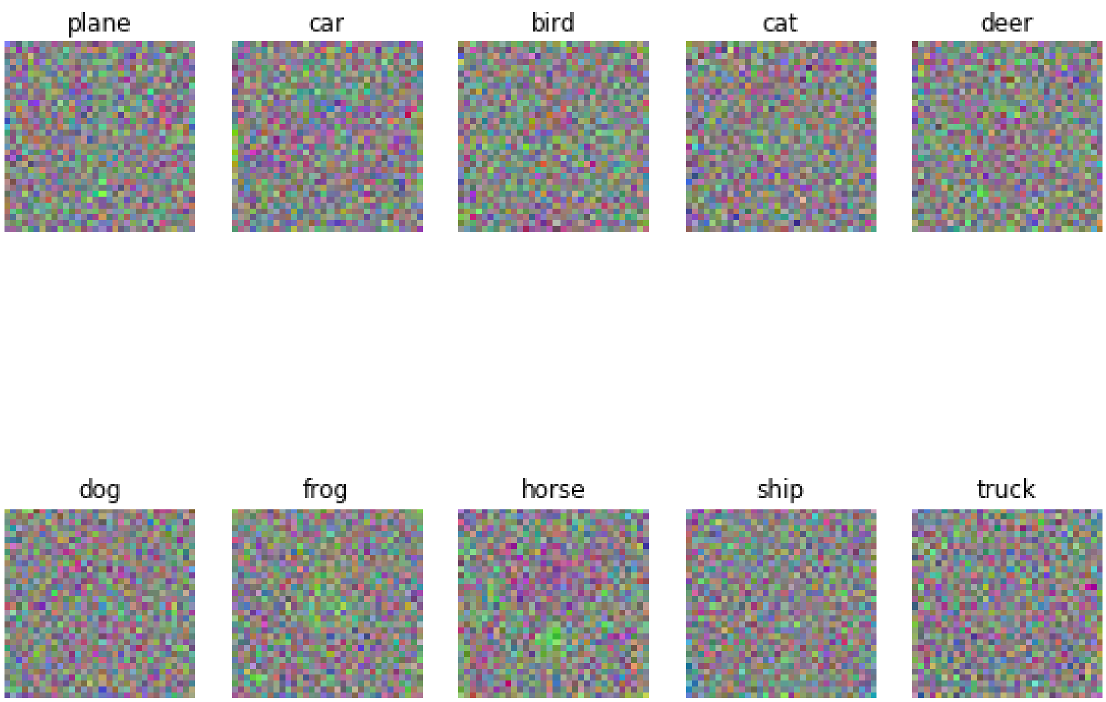


[220]:
```
# Complete the `predict` function in cs231n/classifiers/linear_classifier.py
# Evaluate on test set
y_test_pred = classifier.predict(X_test)
test_accuracy = np.mean(y_test == y_test_pred)
print("softmax on raw pixels final test set accuracy: %f" % (test_accuracy,))
```

```
softmax on raw pixels final test set accuracy: 0.379000
```

[221]:
```
# 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 10, 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.

[1]:
```
# 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.

[2]:
```
# 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.

[12]:
```
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)))
```

```
[[-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]]
```

```
Difference between your scores and correct scores:
3.848682278081994e-08
```

3 Forward pass: compute loss

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

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

```
Difference between your loss and correct loss:
4.6769255135359344e-12
```

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:

[14]:
```
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])))
```

```
W2 max relative error: 9.913918e-10
b2 max relative error: 8.190173e-11
W1 max relative error: 4.426512e-09
b1 max relative error: 5.435431e-08
```

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:

[17]:
```
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], ))
```

```
starting iteration  0
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
Final loss with vanilla SGD: 0.940686
```

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:

[18]:
```
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))
```

```
starting iteration  0
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
Final loss with momentum SGD: 0.494394. We get: 0.494394
```

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:

[19]:
```
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))
```

```
starting iteration  0
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
Final loss with RMSProp: 0.439368. We get: 0.439368
```

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.

[20]:
```python
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)
```

```
Train data shape:  (49000, 3072)
Train labels shape:  (49000,)
Validation data shape:  (1000, 3072)
Validation labels shape:  (1000,)
Test data shape:  (1000, 3072)
Test labels shape:  (1000,)
```

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.

[21]:
```
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)
```

```
starting iteration  0
Finished epoch 0 / 5: cost 2.302593, train: 0.094000, val 0.108000, lr
1.000000e-05
starting iteration  10
```

```
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
starting iteration  100
starting iteration  110
starting iteration  120
starting iteration  130
starting iteration  140
starting iteration  150
starting iteration  160
starting iteration  170
starting iteration  180
starting iteration  190
starting iteration  200
starting iteration  210
starting iteration  220
starting iteration  230
starting iteration  240
starting iteration  250
starting iteration  260
starting iteration  270
starting iteration  280
starting iteration  290
starting iteration  300
starting iteration  310
starting iteration  320
starting iteration  330
starting iteration  340
starting iteration  350
starting iteration  360
starting iteration  370
starting iteration  380
starting iteration  390
starting iteration  400
starting iteration  410
starting iteration  420
starting iteration  430
starting iteration  440
starting iteration  450
starting iteration  460
starting iteration  470
starting iteration  480
Finished epoch 1 / 5: cost 2.285003, train: 0.193000, val 0.173000, lr
```

```
9.500000e-06
starting iteration  490
starting iteration  500
starting iteration  510
starting iteration  520
starting iteration  530
starting iteration  540
starting iteration  550
starting iteration  560
starting iteration  570
starting iteration  580
starting iteration  590
starting iteration  600
starting iteration  610
starting iteration  620
starting iteration  630
starting iteration  640
starting iteration  650
starting iteration  660
starting iteration  670
starting iteration  680
starting iteration  690
starting iteration  700
starting iteration  710
starting iteration  720
starting iteration  730
starting iteration  740
starting iteration  750
starting iteration  760
starting iteration  770
starting iteration  780
starting iteration  790
starting iteration  800
starting iteration  810
starting iteration  820
starting iteration  830
starting iteration  840
starting iteration  850
starting iteration  860
starting iteration  870
starting iteration  880
starting iteration  890
starting iteration  900
starting iteration  910
starting iteration  920
starting iteration  930
starting iteration  940
starting iteration  950
```

```
starting iteration  960
starting iteration  970
Finished epoch 2 / 5: cost 2.154377, train: 0.256000, val 0.240000, lr
9.025000e-06
starting iteration  980
starting iteration  990
starting iteration  1000
starting iteration  1010
starting iteration  1020
starting iteration  1030
starting iteration  1040
starting iteration  1050
starting iteration  1060
starting iteration  1070
starting iteration  1080
starting iteration  1090
starting iteration  1100
starting iteration  1110
starting iteration  1120
starting iteration  1130
starting iteration  1140
starting iteration  1150
starting iteration  1160
starting iteration  1170
starting iteration  1180
starting iteration  1190
starting iteration  1200
starting iteration  1210
starting iteration  1220
starting iteration  1230
starting iteration  1240
starting iteration  1250
starting iteration  1260
starting iteration  1270
starting iteration  1280
starting iteration  1290
starting iteration  1300
starting iteration  1310
starting iteration  1320
starting iteration  1330
starting iteration  1340
starting iteration  1350
starting iteration  1360
starting iteration  1370
starting iteration  1380
starting iteration  1390
starting iteration  1400
starting iteration  1410
```

```
starting iteration  1420
starting iteration  1430
starting iteration  1440
starting iteration  1450
starting iteration  1460
Finished epoch 3 / 5: cost 1.928161, train: 0.264000, val 0.282000, lr
8.573750e-06
starting iteration  1470
starting iteration  1480
starting iteration  1490
starting iteration  1500
starting iteration  1510
starting iteration  1520
starting iteration  1530
starting iteration  1540
starting iteration  1550
starting iteration  1560
starting iteration  1570
starting iteration  1580
starting iteration  1590
starting iteration  1600
starting iteration  1610
starting iteration  1620
starting iteration  1630
starting iteration  1640
starting iteration  1650
starting iteration  1660
starting iteration  1670
starting iteration  1680
starting iteration  1690
starting iteration  1700
starting iteration  1710
starting iteration  1720
starting iteration  1730
starting iteration  1740
starting iteration  1750
starting iteration  1760
starting iteration  1770
starting iteration  1780
starting iteration  1790
starting iteration  1800
starting iteration  1810
starting iteration  1820
starting iteration  1830
starting iteration  1840
starting iteration  1850
starting iteration  1860
starting iteration  1870
```

```
starting iteration  1880
starting iteration  1890
starting iteration  1900
starting iteration  1910
starting iteration  1920
starting iteration  1930
starting iteration  1940
starting iteration  1950
Finished epoch 4 / 5: cost 1.799440, train: 0.329000, val 0.331000, lr
8.145063e-06
starting iteration  1960
starting iteration  1970
starting iteration  1980
starting iteration  1990
starting iteration  2000
starting iteration  2010
starting iteration  2020
starting iteration  2030
starting iteration  2040
starting iteration  2050
starting iteration  2060
starting iteration  2070
starting iteration  2080
starting iteration  2090
starting iteration  2100
starting iteration  2110
starting iteration  2120
starting iteration  2130
starting iteration  2140
starting iteration  2150
starting iteration  2160
starting iteration  2170
starting iteration  2180
starting iteration  2190
starting iteration  2200
starting iteration  2210
starting iteration  2220
starting iteration  2230
starting iteration  2240
starting iteration  2250
starting iteration  2260
starting iteration  2270
starting iteration  2280
starting iteration  2290
starting iteration  2300
starting iteration  2310
starting iteration  2320
starting iteration  2330
```

```
starting iteration  2340
starting iteration  2350
starting iteration  2360
starting iteration  2370
starting iteration  2380
starting iteration  2390
starting iteration  2400
starting iteration  2410
starting iteration  2420
starting iteration  2430
starting iteration  2440
Finished epoch 5 / 5: cost 1.724871, train: 0.351000, val 0.356000, lr
7.737809e-06
finished optimization. best validation accuracy: 0.356000
```

8 Debug the training

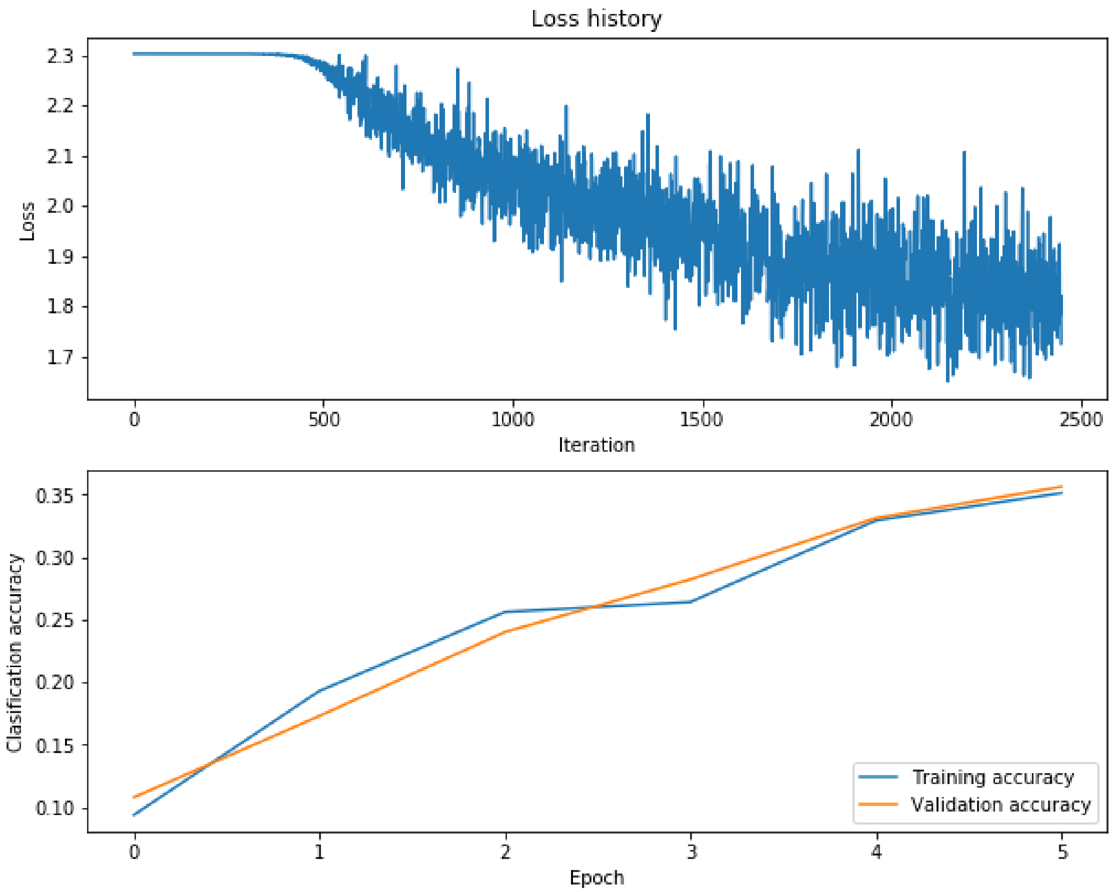
With the default parameters we provided above, you should get a validation accuracy of about 0.37 on the validation set. This isn't very good.

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.

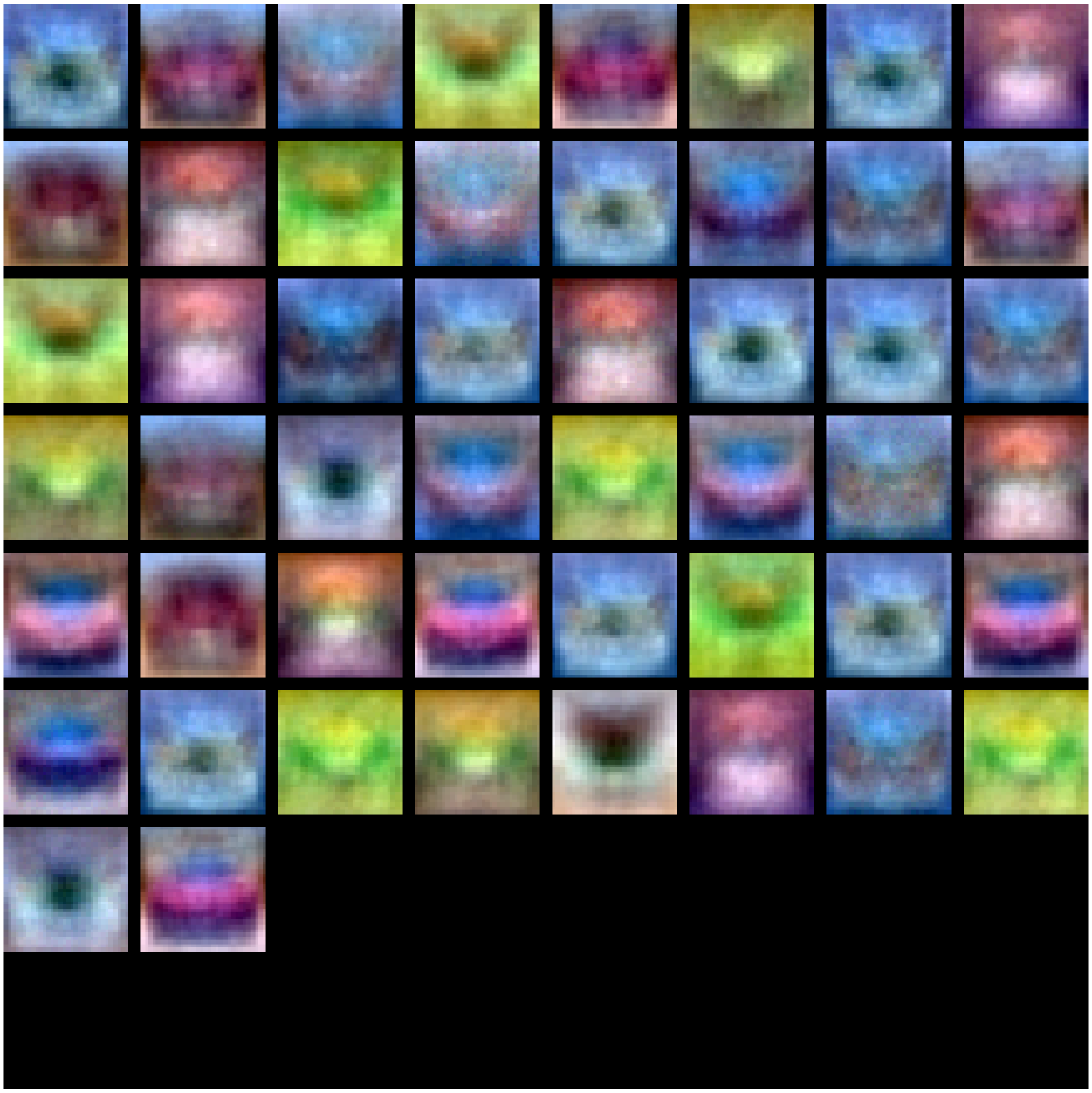
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

[22]:
```python
# 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')
```

[22]:
```
Text(0, 0.5, 'Clasification accuracy')
```



[23]:
```
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.).

[28]:
```
best_model = None # store the best model into this

#################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trained ␣
↪#
# model in best_model.                                                          ␣
↪#
#                                                                               ␣
↪#
# To help debug your network, it may help to use visualizations similar to the ␣
↪#
# ones we used above; these visualizations will have significant qualitative   ␣
↪#
# differences from the ones we saw above for the poorly tuned network.         ␣
↪#
#                                                                               ␣
↪#
# Tweaking hyperparameters by hand can be fun, but you might find it useful to ␣
↪#
# write code to sweep through possible combinations of hyperparameters         ␣
↪#
# automatically like we did on the previous assignment.                        ␣
↪#
#################################################################################
# input size, hidden size, number of classes
model = init_two_layer_model(32*32*3, 1000, 10)
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=10, reg=0.0001,
                                             momentum=0.9,
                                             learning_rate_decay=0.7,
                                             learning_rate=1e-4, verbose=True)
#################################################################################
```

```
#                                    END OF YOUR CODE                           ␣
↪#
################################################################################
```

```
starting iteration  0
Finished epoch 0 / 10: cost 2.302586, train: 0.137000, val 0.105000, lr
1.000000e-04
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
starting iteration  100
starting iteration  110
starting iteration  120
starting iteration  130
starting iteration  140
starting iteration  150
starting iteration  160
starting iteration  170
starting iteration  180
starting iteration  190
starting iteration  200
starting iteration  210
starting iteration  220
starting iteration  230
starting iteration  240
starting iteration  250
starting iteration  260
starting iteration  270
starting iteration  280
starting iteration  290
starting iteration  300
starting iteration  310
starting iteration  320
starting iteration  330
starting iteration  340
starting iteration  350
starting iteration  360
starting iteration  370
starting iteration  380
starting iteration  390
starting iteration  400
```

```
starting iteration  410
starting iteration  420
starting iteration  430
starting iteration  440
starting iteration  450
starting iteration  460
starting iteration  470
starting iteration  480
Finished epoch 1 / 10: cost 1.498791, train: 0.451000, val 0.450000, lr
7.000000e-05
starting iteration  490
starting iteration  500
starting iteration  510
starting iteration  520
starting iteration  530
starting iteration  540
starting iteration  550
starting iteration  560
starting iteration  570
starting iteration  580
starting iteration  590
starting iteration  600
starting iteration  610
starting iteration  620
starting iteration  630
starting iteration  640
starting iteration  650
starting iteration  660
starting iteration  670
starting iteration  680
starting iteration  690
starting iteration  700
starting iteration  710
starting iteration  720
starting iteration  730
starting iteration  740
starting iteration  750
starting iteration  760
starting iteration  770
starting iteration  780
starting iteration  790
starting iteration  800
starting iteration  810
starting iteration  820
starting iteration  830
starting iteration  840
starting iteration  850
starting iteration  860
```

```
starting iteration  870
starting iteration  880
starting iteration  890
starting iteration  900
starting iteration  910
starting iteration  920
starting iteration  930
starting iteration  940
starting iteration  950
starting iteration  960
starting iteration  970
Finished epoch 2 / 10: cost 1.667420, train: 0.510000, val 0.482000, lr
4.900000e-05
starting iteration  980
starting iteration  990
starting iteration  1000
starting iteration  1010
starting iteration  1020
starting iteration  1030
starting iteration  1040
starting iteration  1050
starting iteration  1060
starting iteration  1070
starting iteration  1080
starting iteration  1090
starting iteration  1100
starting iteration  1110
starting iteration  1120
starting iteration  1130
starting iteration  1140
starting iteration  1150
starting iteration  1160
starting iteration  1170
starting iteration  1180
starting iteration  1190
starting iteration  1200
starting iteration  1210
starting iteration  1220
starting iteration  1230
starting iteration  1240
starting iteration  1250
starting iteration  1260
starting iteration  1270
starting iteration  1280
starting iteration  1290
starting iteration  1300
starting iteration  1310
starting iteration  1320
```

```
starting iteration  1330
starting iteration  1340
starting iteration  1350
starting iteration  1360
starting iteration  1370
starting iteration  1380
starting iteration  1390
starting iteration  1400
starting iteration  1410
starting iteration  1420
starting iteration  1430
starting iteration  1440
starting iteration  1450
starting iteration  1460
Finished epoch 3 / 10: cost 1.220875, train: 0.513000, val 0.484000, lr
3.430000e-05
starting iteration  1470
starting iteration  1480
starting iteration  1490
starting iteration  1500
starting iteration  1510
starting iteration  1520
starting iteration  1530
starting iteration  1540
starting iteration  1550
starting iteration  1560
starting iteration  1570
starting iteration  1580
starting iteration  1590
starting iteration  1600
starting iteration  1610
starting iteration  1620
starting iteration  1630
starting iteration  1640
starting iteration  1650
starting iteration  1660
starting iteration  1670
starting iteration  1680
starting iteration  1690
starting iteration  1700
starting iteration  1710
starting iteration  1720
starting iteration  1730
starting iteration  1740
starting iteration  1750
starting iteration  1760
starting iteration  1770
starting iteration  1780
```

```
starting iteration  1790
starting iteration  1800
starting iteration  1810
starting iteration  1820
starting iteration  1830
starting iteration  1840
starting iteration  1850
starting iteration  1860
starting iteration  1870
starting iteration  1880
starting iteration  1890
starting iteration  1900
starting iteration  1910
starting iteration  1920
starting iteration  1930
starting iteration  1940
starting iteration  1950
Finished epoch 4 / 10: cost 1.415856, train: 0.548000, val 0.508000, lr
2.401000e-05
starting iteration  1960
starting iteration  1970
starting iteration  1980
starting iteration  1990
starting iteration  2000
starting iteration  2010
starting iteration  2020
starting iteration  2030
starting iteration  2040
starting iteration  2050
starting iteration  2060
starting iteration  2070
starting iteration  2080
starting iteration  2090
starting iteration  2100
starting iteration  2110
starting iteration  2120
starting iteration  2130
starting iteration  2140
starting iteration  2150
starting iteration  2160
starting iteration  2170
starting iteration  2180
starting iteration  2190
starting iteration  2200
starting iteration  2210
starting iteration  2220
starting iteration  2230
starting iteration  2240
```

```
starting iteration  2250
starting iteration  2260
starting iteration  2270
starting iteration  2280
starting iteration  2290
starting iteration  2300
starting iteration  2310
starting iteration  2320
starting iteration  2330
starting iteration  2340
starting iteration  2350
starting iteration  2360
starting iteration  2370
starting iteration  2380
starting iteration  2390
starting iteration  2400
starting iteration  2410
starting iteration  2420
starting iteration  2430
starting iteration  2440
Finished epoch 5 / 10: cost 1.094622, train: 0.580000, val 0.527000, lr
1.680700e-05
starting iteration  2450
starting iteration  2460
starting iteration  2470
starting iteration  2480
starting iteration  2490
starting iteration  2500
starting iteration  2510
starting iteration  2520
starting iteration  2530
starting iteration  2540
starting iteration  2550
starting iteration  2560
starting iteration  2570
starting iteration  2580
starting iteration  2590
starting iteration  2600
starting iteration  2610
starting iteration  2620
starting iteration  2630
starting iteration  2640
starting iteration  2650
starting iteration  2660
starting iteration  2670
starting iteration  2680
starting iteration  2690
starting iteration  2700
```

```
starting iteration  2710
starting iteration  2720
starting iteration  2730
starting iteration  2740
starting iteration  2750
starting iteration  2760
starting iteration  2770
starting iteration  2780
starting iteration  2790
starting iteration  2800
starting iteration  2810
starting iteration  2820
starting iteration  2830
starting iteration  2840
starting iteration  2850
starting iteration  2860
starting iteration  2870
starting iteration  2880
starting iteration  2890
starting iteration  2900
starting iteration  2910
starting iteration  2920
starting iteration  2930
Finished epoch 6 / 10: cost 1.398334, train: 0.613000, val 0.532000, lr
1.176490e-05
starting iteration  2940
starting iteration  2950
starting iteration  2960
starting iteration  2970
starting iteration  2980
starting iteration  2990
starting iteration  3000
starting iteration  3010
starting iteration  3020
starting iteration  3030
starting iteration  3040
starting iteration  3050
starting iteration  3060
starting iteration  3070
starting iteration  3080
starting iteration  3090
starting iteration  3100
starting iteration  3110
starting iteration  3120
starting iteration  3130
starting iteration  3140
starting iteration  3150
starting iteration  3160
```

```
starting iteration  3170
starting iteration  3180
starting iteration  3190
starting iteration  3200
starting iteration  3210
starting iteration  3220
starting iteration  3230
starting iteration  3240
starting iteration  3250
starting iteration  3260
starting iteration  3270
starting iteration  3280
starting iteration  3290
starting iteration  3300
starting iteration  3310
starting iteration  3320
starting iteration  3330
starting iteration  3340
starting iteration  3350
starting iteration  3360
starting iteration  3370
starting iteration  3380
starting iteration  3390
starting iteration  3400
starting iteration  3410
starting iteration  3420
Finished epoch 7 / 10: cost 1.084483, train: 0.562000, val 0.544000, lr
8.235430e-06
starting iteration  3430
starting iteration  3440
starting iteration  3450
starting iteration  3460
starting iteration  3470
starting iteration  3480
starting iteration  3490
starting iteration  3500
starting iteration  3510
starting iteration  3520
starting iteration  3530
starting iteration  3540
starting iteration  3550
starting iteration  3560
starting iteration  3570
starting iteration  3580
starting iteration  3590
starting iteration  3600
starting iteration  3610
starting iteration  3620
```

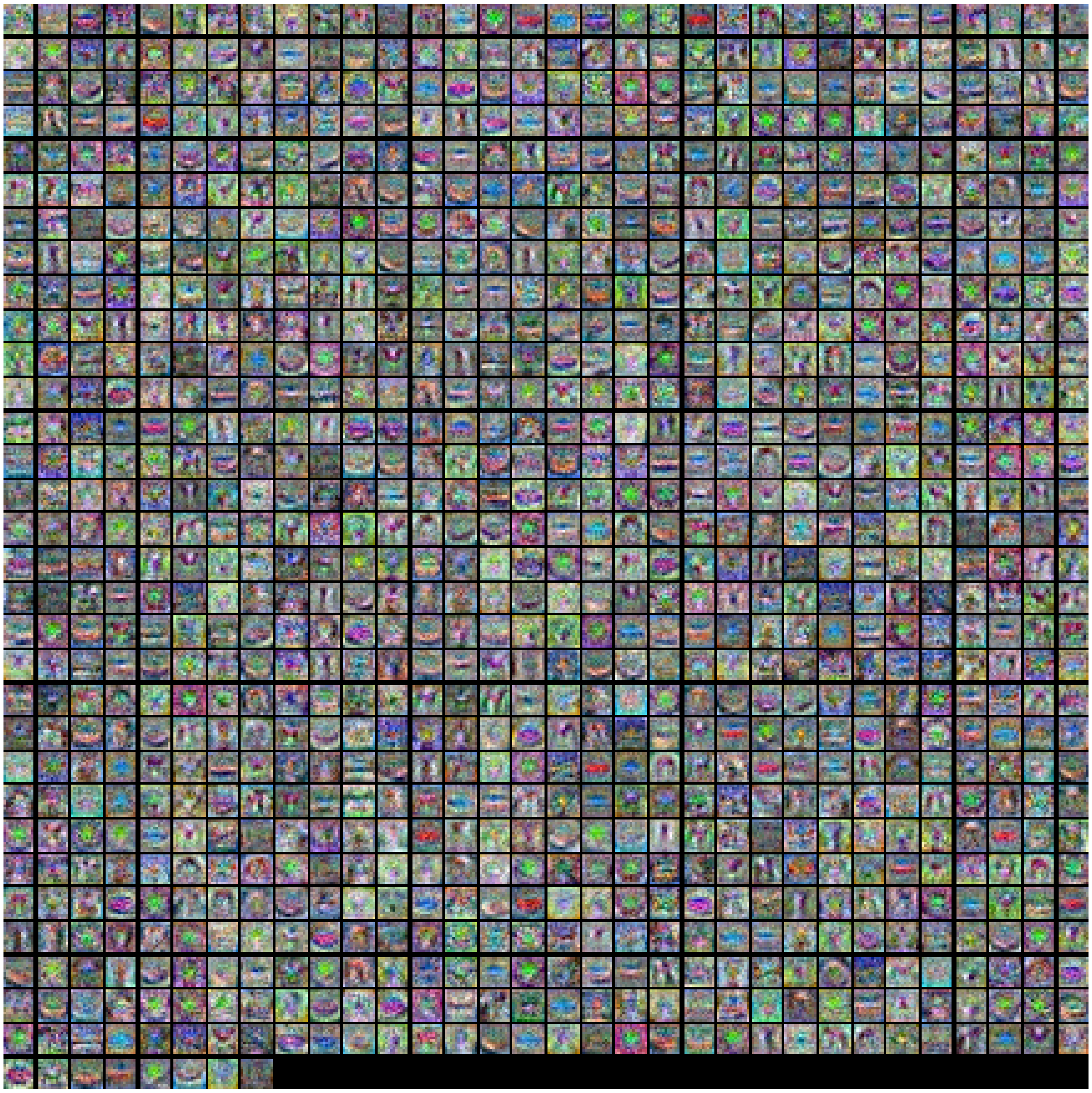
```
starting iteration  3630
starting iteration  3640
starting iteration  3650
starting iteration  3660
starting iteration  3670
starting iteration  3680
starting iteration  3690
starting iteration  3700
starting iteration  3710
starting iteration  3720
starting iteration  3730
starting iteration  3740
starting iteration  3750
starting iteration  3760
starting iteration  3770
starting iteration  3780
starting iteration  3790
starting iteration  3800
starting iteration  3810
starting iteration  3820
starting iteration  3830
starting iteration  3840
starting iteration  3850
starting iteration  3860
starting iteration  3870
starting iteration  3880
starting iteration  3890
starting iteration  3900
starting iteration  3910
Finished epoch 8 / 10: cost 1.069327, train: 0.624000, val 0.534000, lr
5.764801e-06
starting iteration  3920
starting iteration  3930
starting iteration  3940
starting iteration  3950
starting iteration  3960
starting iteration  3970
starting iteration  3980
starting iteration  3990
starting iteration  4000
starting iteration  4010
starting iteration  4020
starting iteration  4030
starting iteration  4040
starting iteration  4050
starting iteration  4060
starting iteration  4070
starting iteration  4080
```

```
starting iteration  4090
starting iteration  4100
starting iteration  4110
starting iteration  4120
starting iteration  4130
starting iteration  4140
starting iteration  4150
starting iteration  4160
starting iteration  4170
starting iteration  4180
starting iteration  4190
starting iteration  4200
starting iteration  4210
starting iteration  4220
starting iteration  4230
starting iteration  4240
starting iteration  4250
starting iteration  4260
starting iteration  4270
starting iteration  4280
starting iteration  4290
starting iteration  4300
starting iteration  4310
starting iteration  4320
starting iteration  4330
starting iteration  4340
starting iteration  4350
starting iteration  4360
starting iteration  4370
starting iteration  4380
starting iteration  4390
starting iteration  4400
Finished epoch 9 / 10: cost 1.126785, train: 0.651000, val 0.557000, lr
4.035361e-06
starting iteration  4410
starting iteration  4420
starting iteration  4430
starting iteration  4440
starting iteration  4450
starting iteration  4460
starting iteration  4470
starting iteration  4480
starting iteration  4490
starting iteration  4500
starting iteration  4510
starting iteration  4520
starting iteration  4530
starting iteration  4540
```

```
starting iteration  4550
starting iteration  4560
starting iteration  4570
starting iteration  4580
starting iteration  4590
starting iteration  4600
starting iteration  4610
starting iteration  4620
starting iteration  4630
starting iteration  4640
starting iteration  4650
starting iteration  4660
starting iteration  4670
starting iteration  4680
starting iteration  4690
starting iteration  4700
starting iteration  4710
starting iteration  4720
starting iteration  4730
starting iteration  4740
starting iteration  4750
starting iteration  4760
starting iteration  4770
starting iteration  4780
starting iteration  4790
starting iteration  4800
starting iteration  4810
starting iteration  4820
starting iteration  4830
starting iteration  4840
starting iteration  4850
starting iteration  4860
starting iteration  4870
starting iteration  4880
starting iteration  4890
Finished epoch 10 / 10: cost 1.272121, train: 0.601000, val 0.543000, lr
2.824752e-06
finished optimization. best validation accuracy: 0.557000
```

[29]:

```
# 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.

[30]:
```
scores_test = two_layer_net(X_test, best_model)
print('Test accuracy: ', np.mean(np.argmax(scores_test, axis=1) == y_test))
```

```
Test accuracy:  0.538
```

[ ]:
```
```

layers

February 10, 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.

[3]:
```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:  8.382793903902256e-11
dw error:  1.558761275742988e-10
db error:  4.6002441594006085e-11
```

4 ReLU layer: forward

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

[4]:
```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:

[5]:
```
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:  3.275618483368625e-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.

[6]:
```
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:')
```

```python
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:  9.000597824304572
dx error:  8.182894472887002e-10

Testing softmax_loss:
loss:  2.3026453494071113
dx error:  7.485321900683551e-09
```

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:

[7]:
```python
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 1e-8
print('Testing conv_forward_naive')
print('difference: ', rel_error(out, correct_out))
```

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

8 Aside: Image processing via convolutions

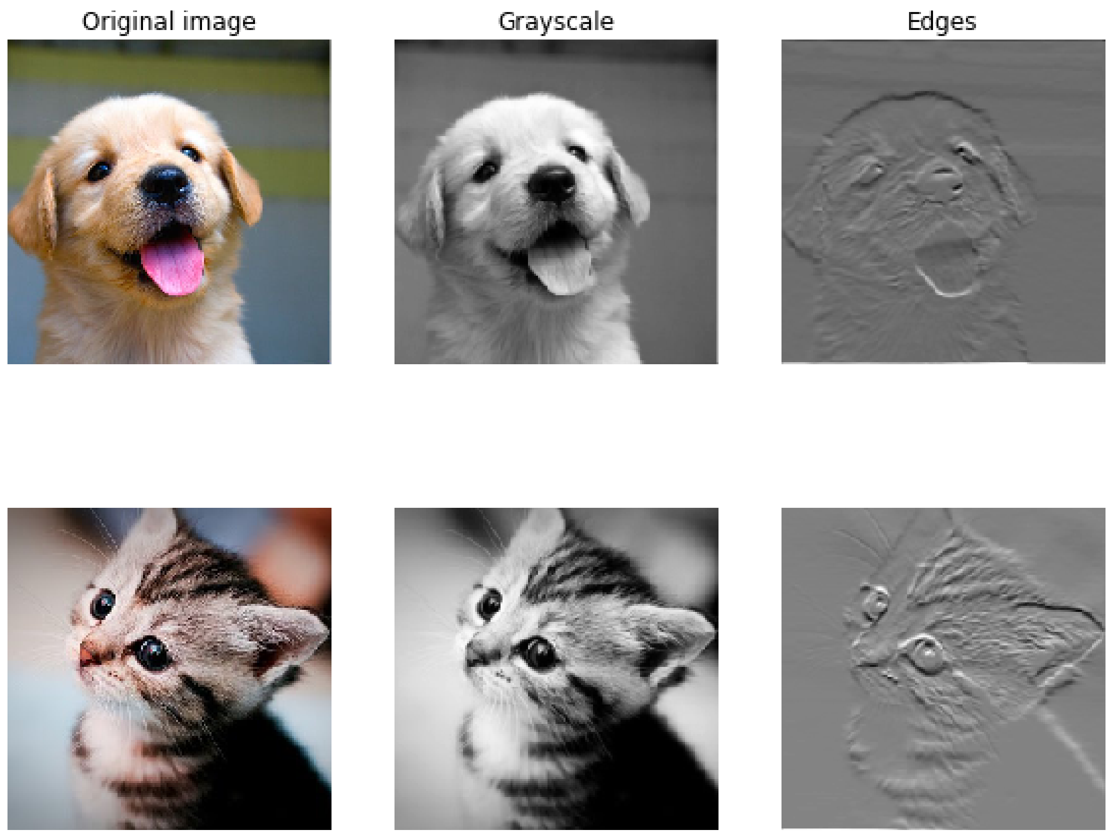
As fun way to both check your implementation and gain a better understanding of the type of operation that convolutional layers can perform, we will set up an input containing two images and manually set up filters that perform common image processing operations (grayscale conversion and edge detection).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

[8]:
```
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))

# 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
```

```python
# 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()
```

```
C:\Users\82120\.conda\envs\7643\lib\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
C:\Users\82120\.conda\envs\7643\lib\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.
C:\Users\82120\.conda\envs\7643\lib\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.

[25]:

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

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

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

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

# Your errors should be around 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))
```

```
5
Testing conv_backward_naive function
dx error:  3.16025486039339e-09
dw error:  5.03599497021093e-10
db error:  1.2370846328888066e-10
```

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`.

[26]:
```
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.

[33]:
```
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:  3.275628496082982e-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:

[30]:

```python
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: 3.885641s
Fast: 0.007948s
Speedup: 488.872994x
Difference:  3.978240059466707e-11

Testing conv_backward_fast:
Naive: 14.729645s
Fast: 0.008946s
Speedup: 1646.516977x
dx difference:  1.2479237101602112e-11
dw difference:  1.5251920225304764e-12
```

```
db difference:  0.0
```

[34]:
```python
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: 0.141594s
fast: 0.002992s
speedup: 47.321753x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.387990s
speedup: 43.224229x
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:

[11]:
```
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:  2.2041086893381568e-07
dw error:  2.717511280801651e-10
db error:  3.502737338077941e-11
```

[12]:
```
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:  2.6296328828998218e-08
dw error:  1.5342896567838248e-09
db error:  7.688682989110673e-12
```

[13]:
```
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))
print('dw error: ', rel_error(dw_num, dw))
print('db error: ', rel_error(db_num, db))
```

```
Testing affine_relu_forward:
dx error:  4.5035126837662996e-09
dw error:  6.687598198525906e-11
db error:  3.2755814073540085e-12
```

[ ]:
```
```

convnet

February 10, 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))))
```

[9]:
```
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.

[10]:
```
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):  2.3026302984328333
Sanity check loss (with regularization):  2.344375763367764
```

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.

[11]:
```
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: 3.023856e-07
W2 max relative error: 1.418324e-05
b1 max relative error: 2.668192e-08
b2 max relative error: 1.995789e-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.

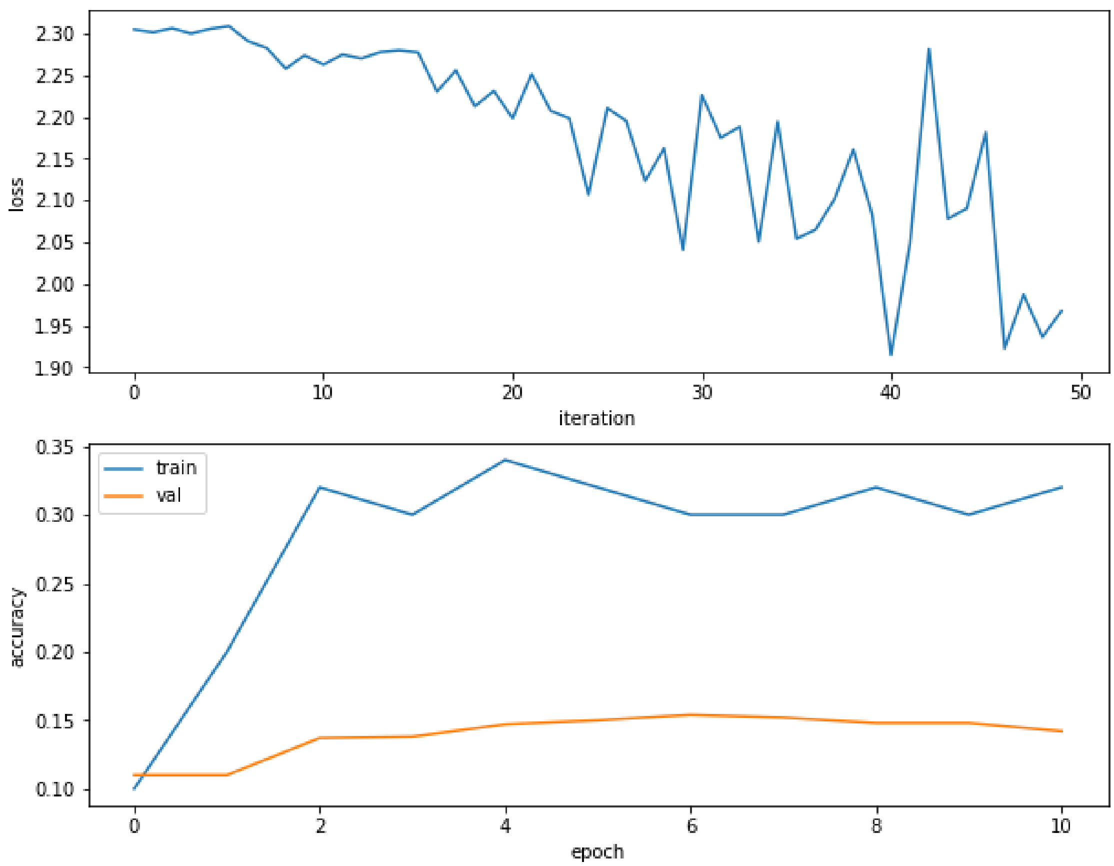
[13]:
```
# Use a two-layer ConvNet to overfit 50 training examples.

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=0.05, momentum=0.9, learning_rate=0.00001, batch_size=10,␣
↪num_epochs=10,
          verbose=True)
```

```
starting iteration  0
Finished epoch 0 / 10: cost 2.304627, train: 0.100000, val 0.110000, lr
1.000000e-05
Finished epoch 1 / 10: cost 2.305375, train: 0.200000, val 0.110000, lr
9.500000e-06
Finished epoch 2 / 10: cost 2.273784, train: 0.320000, val 0.137000, lr
9.025000e-06
starting iteration  10
Finished epoch 3 / 10: cost 2.279856, train: 0.300000, val 0.138000, lr
8.573750e-06
Finished epoch 4 / 10: cost 2.231050, train: 0.340000, val 0.147000, lr
8.145063e-06
starting iteration  20
Finished epoch 5 / 10: cost 2.107130, train: 0.320000, val 0.150000, lr
7.737809e-06
Finished epoch 6 / 10: cost 2.040741, train: 0.300000, val 0.154000, lr
7.350919e-06
starting iteration  30
Finished epoch 7 / 10: cost 2.194671, train: 0.300000, val 0.152000, lr
6.983373e-06
Finished epoch 8 / 10: cost 2.082568, train: 0.320000, val 0.148000, lr
```

```
6.634204e-06
starting iteration  40
Finished epoch 9 / 10: cost 2.090183, train: 0.300000, val 0.148000, lr
6.302494e-06
Finished epoch 10 / 10: cost 1.967252, train: 0.320000, val 0.142000, lr
5.987369e-06
finished optimization. best validation accuracy: 0.154000
```

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

[14]:
```python
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.

[15]:
```
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 2.294376, train: 0.105000, val 0.108000, lr
1.000000e-04
starting iteration  10
starting iteration  20
starting iteration  30
starting iteration  40
starting iteration  50
Finished epoch 0 / 1: cost 1.934034, train: 0.301000, val 0.320000, lr
1.000000e-04
starting iteration  60
starting iteration  70
starting iteration  80
starting iteration  90
starting iteration  100
Finished epoch 0 / 1: cost 1.768556, train: 0.373000, val 0.355000, lr
1.000000e-04
starting iteration  110
starting iteration  120
starting iteration  130
starting iteration  140
starting iteration  150
Finished epoch 0 / 1: cost 1.515819, train: 0.381000, val 0.378000, lr
1.000000e-04
starting iteration  160
starting iteration  170
```

```
starting iteration  180
starting iteration  190
starting iteration  200
Finished epoch 0 / 1: cost 1.677729, train: 0.375000, val 0.411000, lr
1.000000e-04
starting iteration  210
starting iteration  220
starting iteration  230
starting iteration  240
starting iteration  250
Finished epoch 0 / 1: cost 1.849573, train: 0.388000, val 0.413000, lr
1.000000e-04
starting iteration  260
starting iteration  270
starting iteration  280
starting iteration  290
starting iteration  300
Finished epoch 0 / 1: cost 1.854043, train: 0.429000, val 0.426000, lr
1.000000e-04
starting iteration  310
starting iteration  320
starting iteration  330
starting iteration  340
starting iteration  350
Finished epoch 0 / 1: cost 1.815773, train: 0.456000, val 0.441000, lr
1.000000e-04
starting iteration  360
starting iteration  370
starting iteration  380
starting iteration  390
starting iteration  400
Finished epoch 0 / 1: cost 1.995193, train: 0.424000, val 0.427000, lr
1.000000e-04
starting iteration  410
starting iteration  420
starting iteration  430
starting iteration  440
starting iteration  450
Finished epoch 0 / 1: cost 2.085419, train: 0.426000, val 0.436000, lr
1.000000e-04
starting iteration  460
starting iteration  470
starting iteration  480
starting iteration  490
starting iteration  500
Finished epoch 0 / 1: cost 1.864259, train: 0.441000, val 0.432000, lr
1.000000e-04
starting iteration  510
```

```
starting iteration  520
starting iteration  530
starting iteration  540
starting iteration  550
Finished epoch 0 / 1: cost 1.329013, train: 0.484000, val 0.436000, lr
1.000000e-04
starting iteration  560
starting iteration  570
starting iteration  580
starting iteration  590
starting iteration  600
Finished epoch 0 / 1: cost 1.509924, train: 0.464000, val 0.473000, lr
1.000000e-04
starting iteration  610
starting iteration  620
starting iteration  630
starting iteration  640
starting iteration  650
Finished epoch 0 / 1: cost 1.282254, train: 0.454000, val 0.472000, lr
1.000000e-04
starting iteration  660
starting iteration  670
starting iteration  680
starting iteration  690
starting iteration  700
Finished epoch 0 / 1: cost 1.523986, train: 0.464000, val 0.436000, lr
1.000000e-04
starting iteration  710
starting iteration  720
starting iteration  730
starting iteration  740
starting iteration  750
Finished epoch 0 / 1: cost 2.197756, train: 0.484000, val 0.469000, lr
1.000000e-04
starting iteration  760
starting iteration  770
starting iteration  780
starting iteration  790
starting iteration  800
Finished epoch 0 / 1: cost 1.176381, train: 0.495000, val 0.479000, lr
1.000000e-04
starting iteration  810
starting iteration  820
starting iteration  830
starting iteration  840
starting iteration  850
Finished epoch 0 / 1: cost 1.670690, train: 0.478000, val 0.488000, lr
1.000000e-04
```

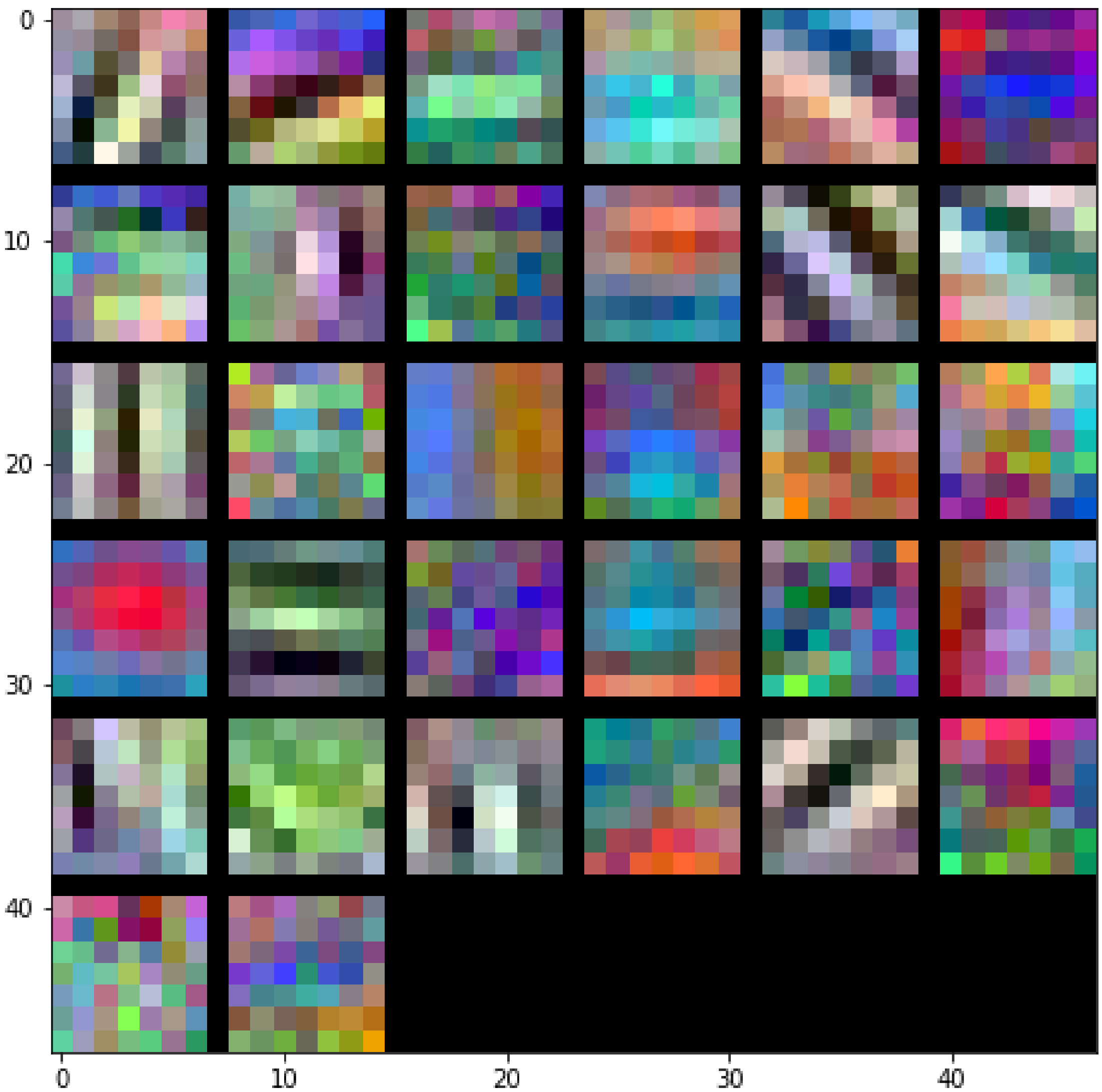
```
starting iteration  860
starting iteration  870
starting iteration  880
starting iteration  890
starting iteration  900
Finished epoch 0 / 1: cost 1.255807, train: 0.451000, val 0.476000, lr
1.000000e-04
starting iteration  910
starting iteration  920
starting iteration  930
starting iteration  940
starting iteration  950
Finished epoch 0 / 1: cost 1.237256, train: 0.470000, val 0.462000, lr
1.000000e-04
starting iteration  960
starting iteration  970
Finished epoch 1 / 1: cost 1.929538, train: 0.485000, val 0.481000, 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.

[16]:
```
from cs231n.vis_utils import visualize_grid

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

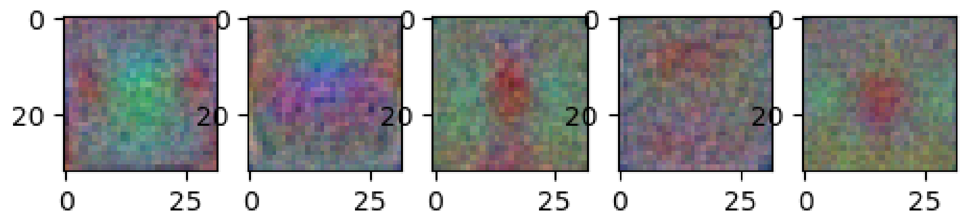
[16]:
```
<matplotlib.image.AxesImage at 0x1bb4e1c3ec8>
```

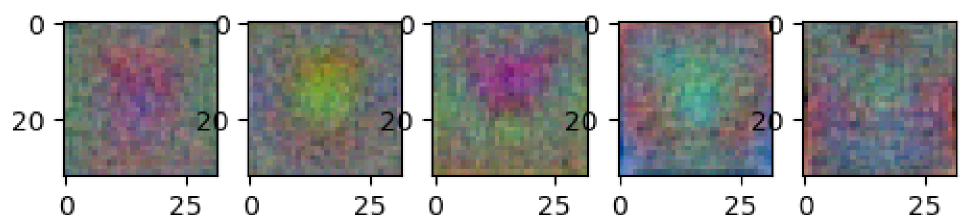


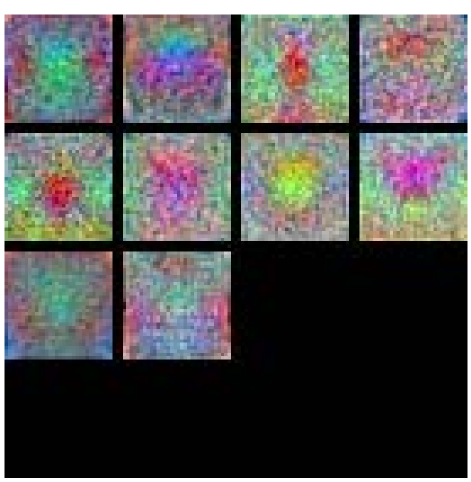
0
10
20
30
40
0
10
20
30
40

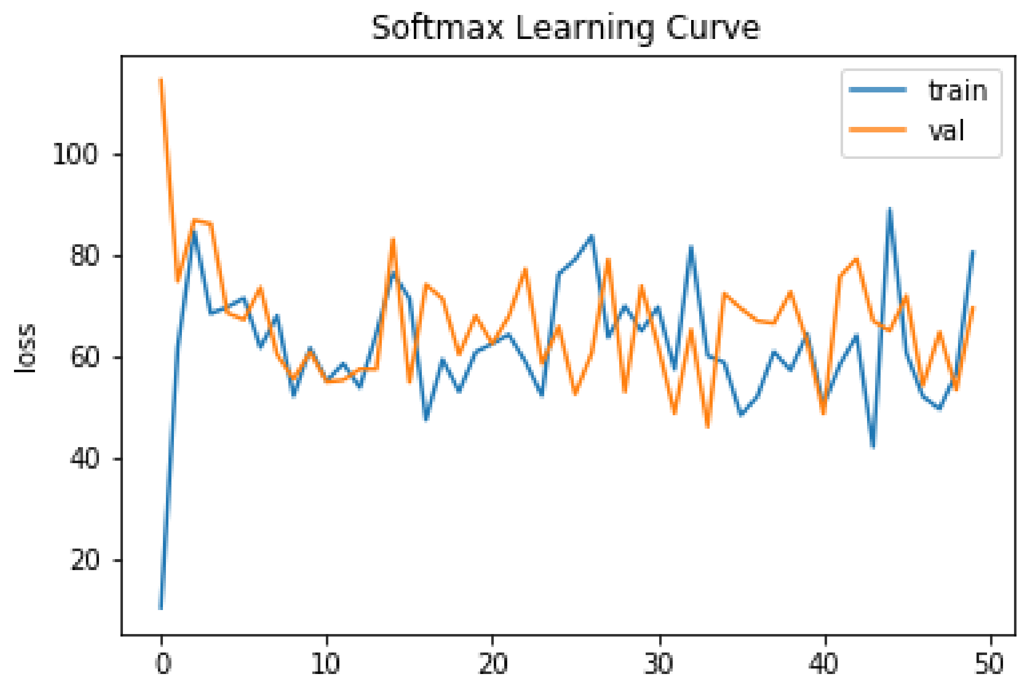

```
[ ]:
```

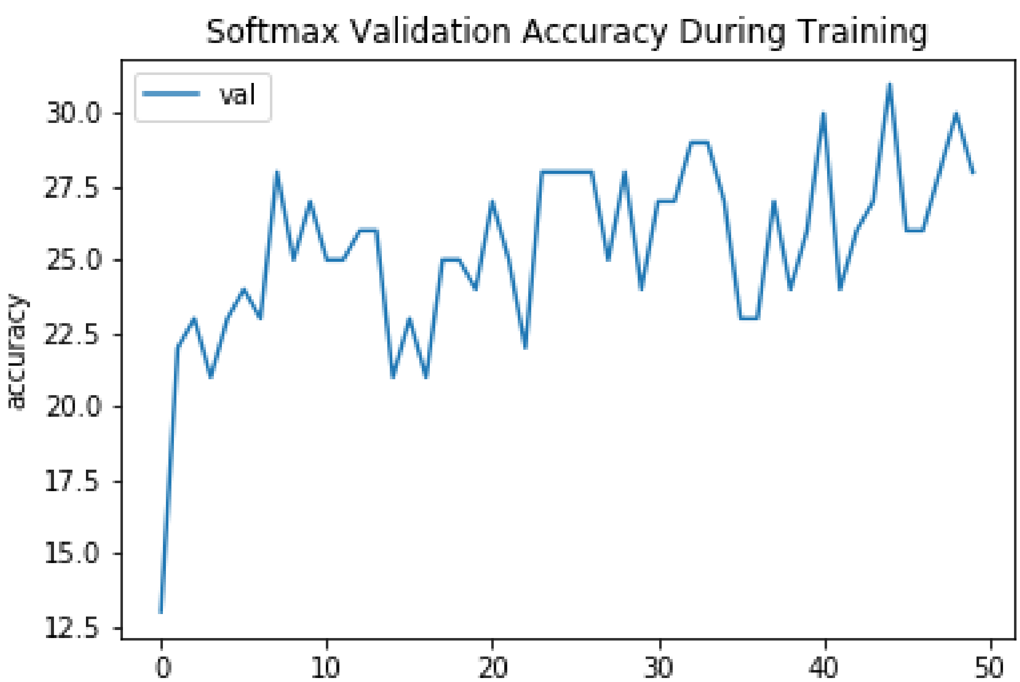
Softmax



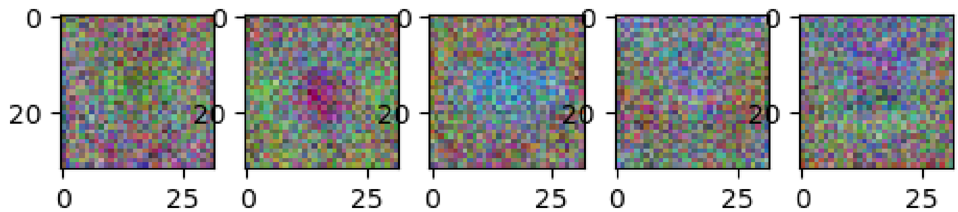


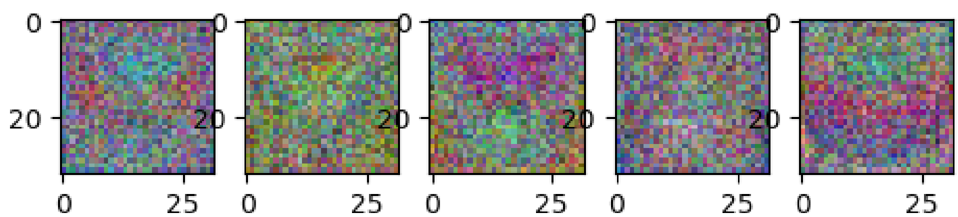


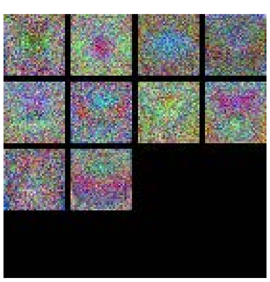

Softmax Learning Curve
train
val
loss
100
80
60
40
20
0
10
20
30
40
50

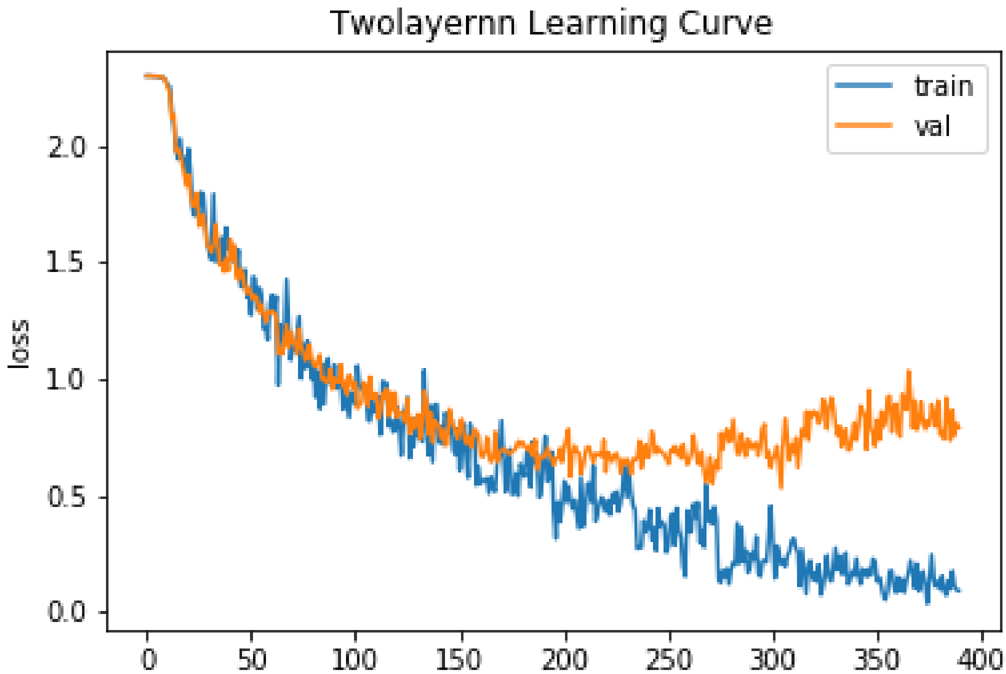

Softmax Validation Accuracy During Training
val
accuracy
30.0
27.5
25.0
22.5
20.0
17.5
15.0
12.5
0
10
20
30
40
50

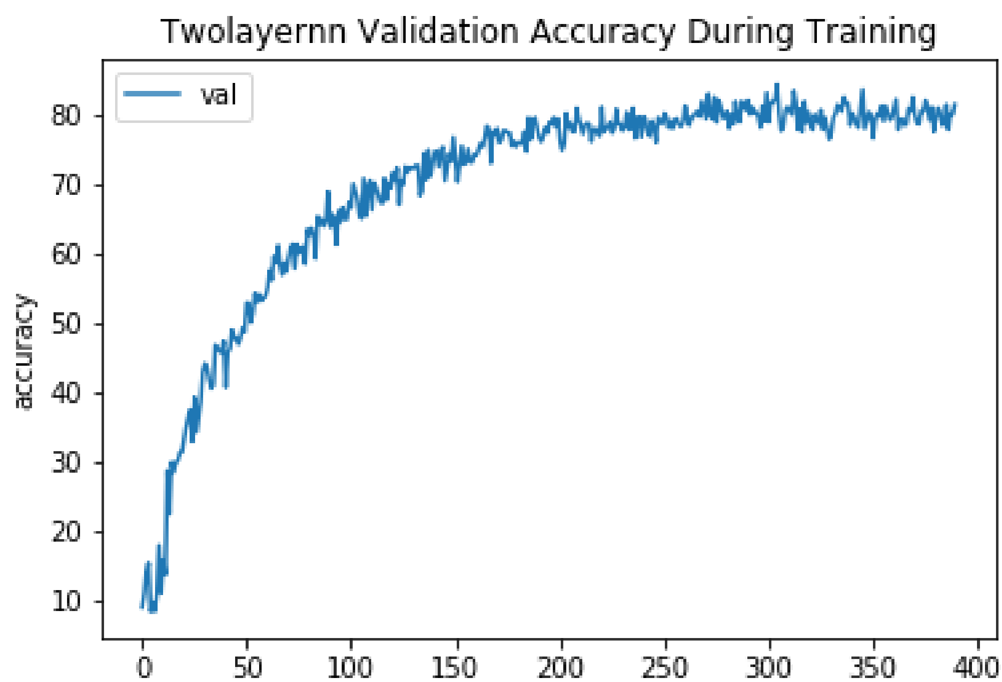
Twolayernn:



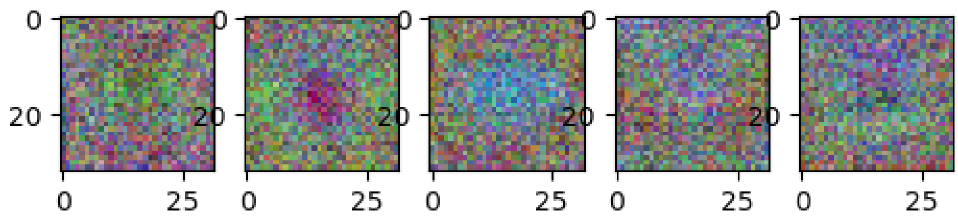


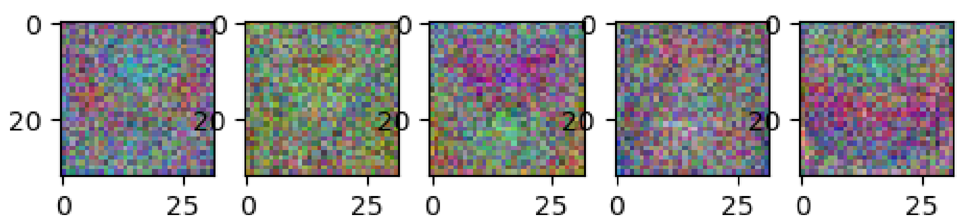


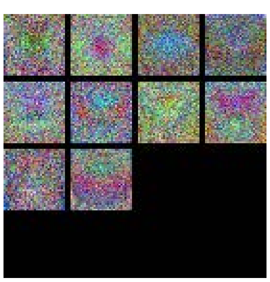

Twolayernn Learning Curve
train
val
loss
2.0
1.5
1.0
0.5
0.0
0
50
100
150
200
250
300
350
400

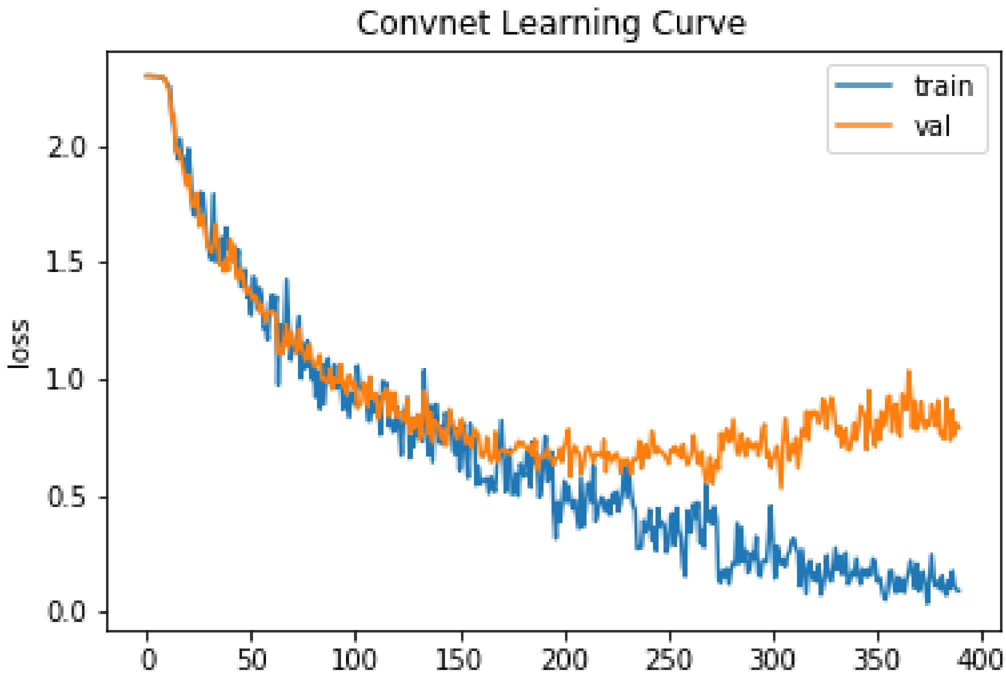

Twolayernn Validation Accuracy During Training
val
accuracy
80
70
60
50
40
30
20
10
0
50
100
150
200
250
300
350
400

Convnet:

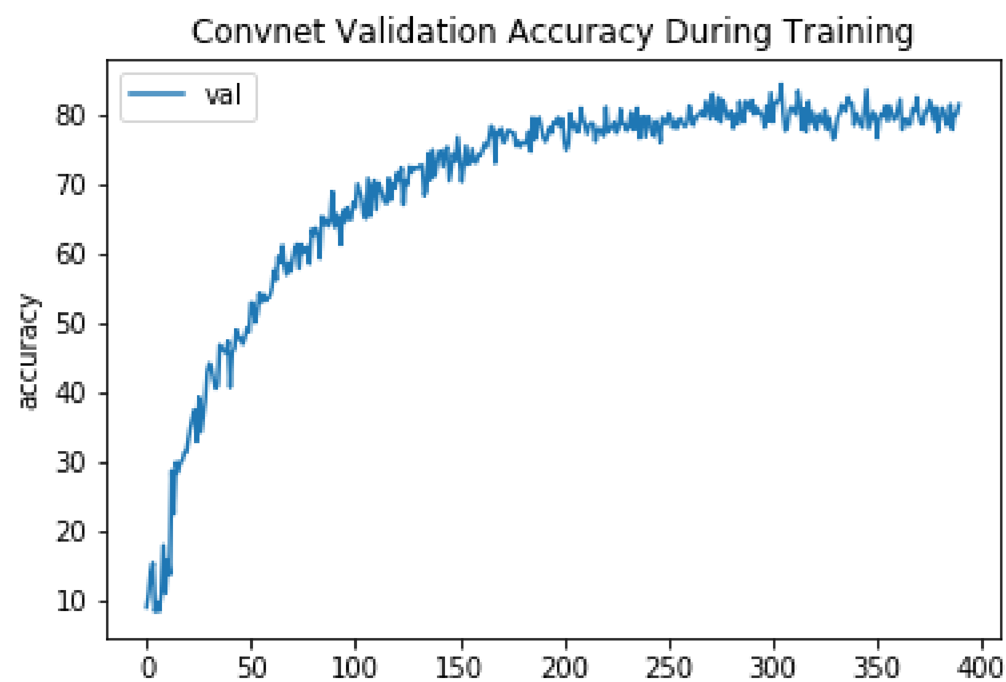






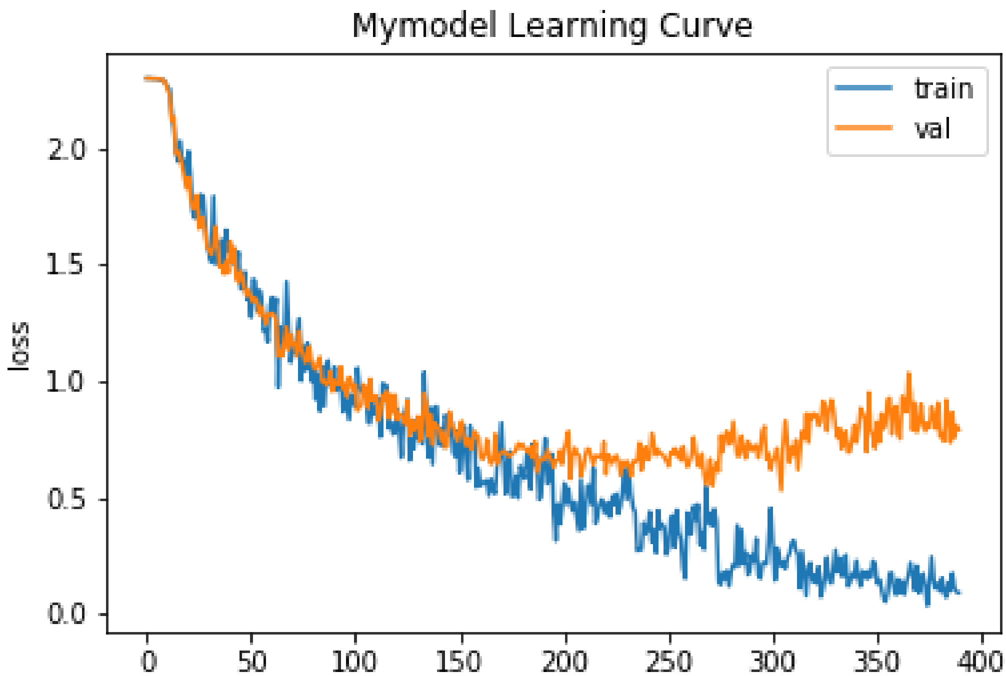


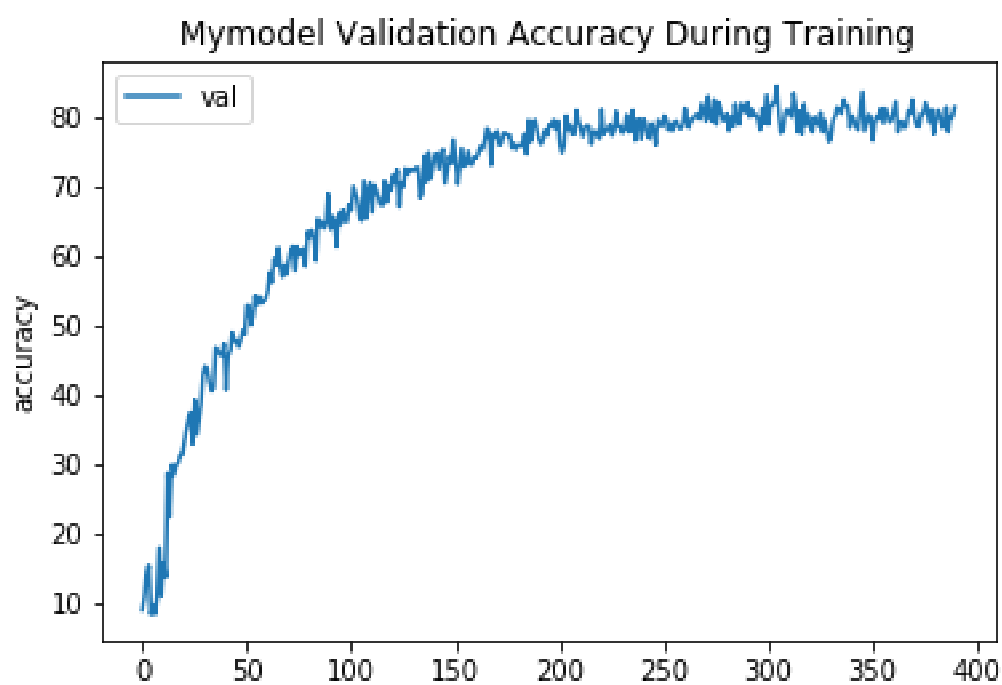
Convnet Learning Curve
train
val
loss
2.0
1.5
1.0
0.5
0.0
0
50
100
150
200
250
300
350
400



Convnet Validation Accuracy During Training
val
accuracy
80
70
60
50
40
30
20
10
0
50
100
150
200
250
300
350
400

Mymodel:


Mymodel Learning Curve
train
val
loss
2.0
1.5
1.0
0.5
0.0
0
50
100
150
200
250
300
350
400


Mymodel Validation Accuracy During Training
val
accuracy
80
70
60
50
40
30
20
10
0
50
100
150
200
250
300
350
400