CS7643: Deep Learning
Fall 2019
HW1 Solutions

Nicolas Six

September 26, 2019

1 Gradient Descent

1.1 Minimisation of $\vec{w^*}$

If this function has a minimum, it has to be when its partial derivative regarding to $\vec{w}$ is null. In addition, as the this function is convex, so admit only one point where $\vec{w}$ is null, and so only one extrema. As this function is clearly unbounded from above this extrema is the global minimum.

$$
\begin{aligned}
\frac{\partial\left(f\left(\vec{w^{(t)}}\right)+\left\langle\vec{w}-\vec{w^{(t)}},\nabla f\left(\vec{w^{(t)}}\right)\right\rangle+\frac{\lambda}{2}\left\|\vec{w}-\vec{w^{(t)}}\right\|^2\right)}{\partial\vec{w}} &= 0\\
\Leftrightarrow \frac{\partial\left(\left\langle\vec{w},\nabla f\left(\vec{w^{(t)}}\right)\right\rangle\right)}{\partial\vec{w}}+\frac{\partial\left(\frac{\lambda}{2}\left\langle\vec{w}-\vec{w^{(t)}},\vec{w}-\vec{w^{(t)}}\right\rangle\right)}{\partial\vec{w}} &= 0\\
\Leftrightarrow \left\langle\frac{\partial\vec{w}}{\partial\vec{w}},\nabla f\left(\vec{w^{(t)}}\right)\right\rangle+\lambda\left\langle\frac{\partial\vec{w}}{\partial\vec{w}},\vec{w}-\vec{w^{(t)}}\right\rangle &= 0\\
\Leftrightarrow \left\langle\frac{\partial\vec{w}}{\partial\vec{w}},\nabla f\left(\vec{w^{(t)}}\right)+\lambda\left(\vec{w}-\vec{w^{(t)}}\right)\right\rangle &= 0\\
\Leftrightarrow \nabla f\left(\vec{w^{(t)}}\right)+\lambda\left(\vec{w}-\vec{w^{(t)}}\right) &= \vec{0}\\
\Leftrightarrow \vec{w} = \vec{w^{(t)}}-\frac{1}{\lambda}\nabla f\left(\vec{w^{(t)}}\right)
\end{aligned}
$$

In conclusion we get the following:

$$
\begin{aligned}
\vec{w^*} &= \vec{w^{(t)}}-\frac{1}{\lambda}\nabla f\left(\vec{w^{(t)}}\right)\\
\eta &= \frac{1}{\lambda}
\end{aligned}
$$

This show us that under our current set of assumption, the gradient descent is leading us the the optimal solution.

1.2 Lema proof

We have:

$$\vec{w^{(t+1)}} = \vec{w^{(t)}} - \eta\vec{v_t}$$
$$\Leftrightarrow \vec{v_t} = \frac{1}{\eta}\left(\vec{w^{(t)}} - \vec{w^{(t+1)}}\right)$$

So:

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

In conclusion

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

1.3 Convergence rate

We have:

$$f\left(\vec{w^*}\right) \geqslant f\left(\vec{w}\right) + \left\langle \vec{w^*} - \vec{w}, \nabla f\left(\vec{w}\right)\right\rangle$$
$$\vec{w} = \frac{1}{T}\sum_{t=1}^{T} \vec{w^{(t)}}$$

So:

$$\begin{aligned}
& f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant f\left(\vec{w}\right) - f\left(\vec{w}\right) - \left\langle \vec{w^*} - \vec{w}, \nabla f\left(\vec{w}\right)\right\rangle \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \left\langle \vec{w} - \vec{w^*}, \nabla f\left(\vec{w}\right)\right\rangle \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \left\langle \frac{1}{T}\sum_{t=1}^{T} \vec{w^{(t)}} - \vec{w^*}, \nabla f\left(\frac{1}{T}\sum_{t=1}^{T} \vec{w^{(t)}}\right)\right\rangle \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \frac{1}{T}\sum_{t=1}^{T} \left\langle \vec{w^{(t)}} - \vec{w^*}, \nabla f\left(\vec{w^{(t)}}\right)\right\rangle \\
\Rightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \frac{1}{T}\left(\frac{\left\|\vec{w^*}\right\|^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{T}\left\|\nabla f\left(\vec{w^{(t)}}\right)\right\|^2\right) \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \frac{1}{T}\left(\frac{B^2}{2\eta} + \frac{\eta}{2}\sum_{t=1}^{T}\rho^2\right) \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \frac{1}{T}\left(\frac{B^2}{2\eta} + \frac{\eta}{2}T\rho^2\right) \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \frac{B^2}{2\eta T} + \frac{\eta}{2}\rho^2 \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \frac{B^2}{2\sqrt{\frac{B^2}{\rho^2 T}}T} + \frac{\sqrt{\frac{B^2}{\rho^2 T}}}{2}\rho^2 \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \frac{B\rho}{2\sqrt{T}} + \frac{B}{2\sqrt{T}}\rho \\
\Leftrightarrow\ & f\left(\vec{w}\right) - f\left(\vec{w^*}\right) \leqslant \frac{B\rho}{\sqrt{T}}
\end{aligned}$$

So for $\eta = \sqrt{\frac{B^2}{\rho^2 T}}$ the convergence rate of gradient decent is in the order of $O\left(\frac{1}{\sqrt{T}}\right)$

1.4 Convergence guarantee

Counter example:

$$
\begin{aligned}
f(w) &= (w - \frac{1}{2})^2 + \frac{9}{4} \\
\frac{\partial f}{\partial w}\left(\frac{1}{2}\right) &= 0
\end{aligned}
$$

Given that SGD is at the global optimal, $\frac{1}{2}$, the gradiant for both terms should be zero in order to guarantee this property.

$$
\begin{aligned}
\frac{\partial\left(\frac{1}{2}(w-2)^2\right)}{\partial w}\left(\frac{1}{2}\right) &= (w-2)\left(\frac{1}{2}\right) \\
&= -\frac{3}{2} \\
&\neq 0 \\
\frac{\partial\left(\frac{1}{2}(w+1)^2\right)}{\partial w}\left(\frac{1}{2}\right) &= (w+1)\left(\frac{1}{2}\right) \\
&= \frac{3}{2} \\
&\neq 0
\end{aligned}
$$

So in this case, for any η not equal to zero, SGD will increase the overall loss function.
In conclusion SGD doesn't guarantee to reach the global optimum after a given time under this conditions.

2 Automatic Differentiation

2.1 Computation graph

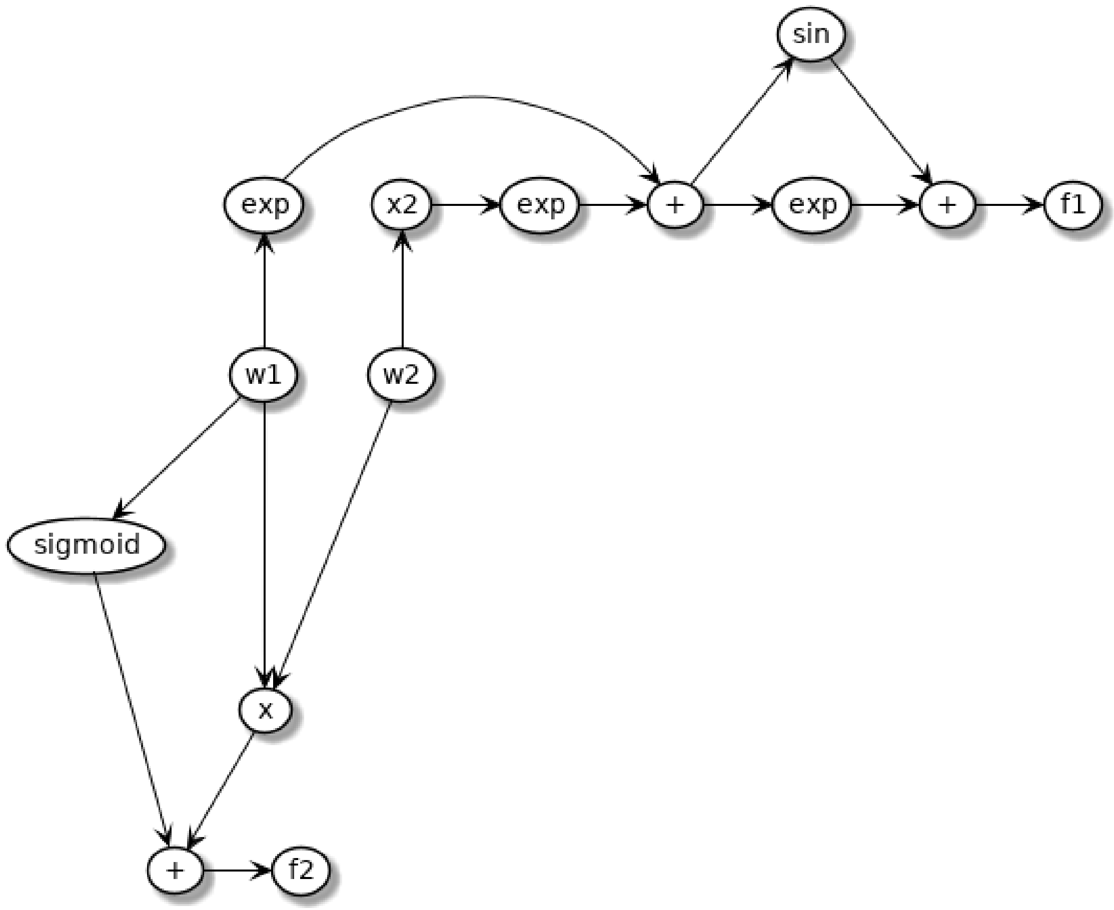


Figure 1: Computation graph for $\vec{f}$

2.2 Jacobian and numerical differentiation

With:

$$
\begin{aligned}
f_1(1,2) &= 24521.662861313 \\
f_2(1,2) &= 2.73105857863 \\
f_1(1.01,2) &= 25200.8058793246 \\
f_2(1.01,2) &= 2.75302014923886 \\
f_1(1,2.01) &= 26411.9854338567 \\
f_2(1,2.01) &= 2.74105857863
\end{aligned}
$$

We have:

$$
\begin{aligned}
\frac{\partial \vec{f}}{\partial \vec{w}} &= \begin{bmatrix} \frac{f_1(1+\Delta,2)-f_1(1,2)}{\Delta} & \frac{f_1(1,2+\Delta)-f_1(1,2)}{\Delta} \\ \frac{f_2(1+\Delta,2)-f_2(1,2)}{\Delta} & \frac{f_2(1,2+\Delta)-f_2(1,2)}{\Delta} \end{bmatrix} \\
&= \begin{bmatrix} 67914.3018011622 & 189032.25725437 \\ 2.19615706088527 & 0.999999999999979 \end{bmatrix}
\end{aligned}
$$

Note: This computation weren't done in a formal way and are so subject to numerical approximations. Adding some more imprecision to already not very accurate method.

2.3 Forward mode auto-differentiation

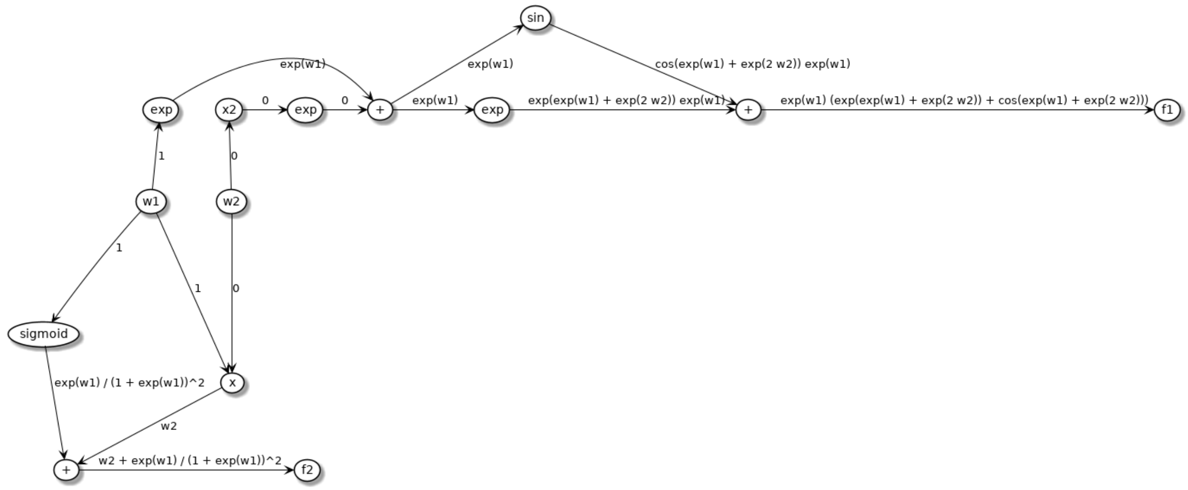


Figure 2: Forward gradient decent on $\vec{f}$ for $w1$

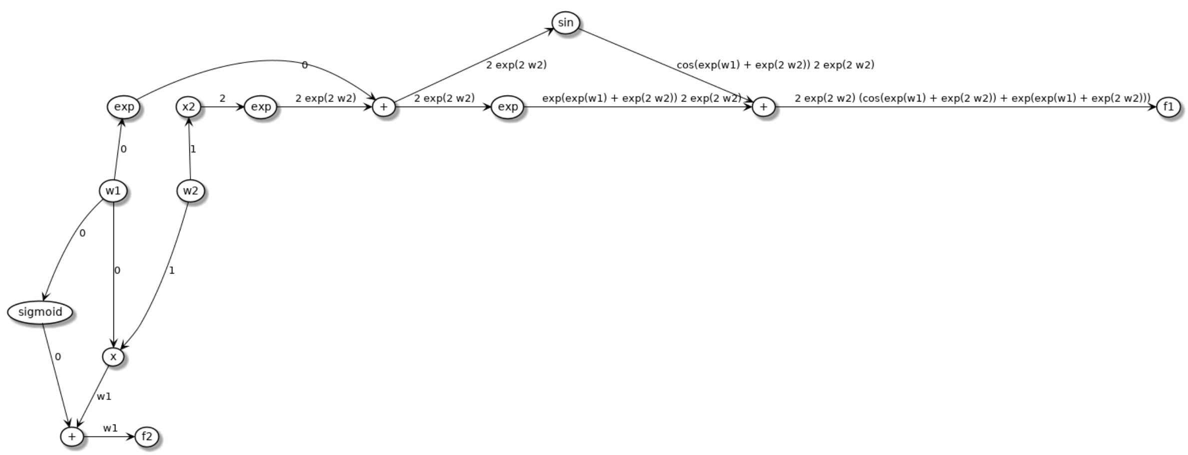


Figure 3: Forward gradient decent on $\vec{f}$ for $w2$

Which in a more mathematical format give us:

$$\frac{\partial \vec{f}}{\partial \vec{w}} = \begin{bmatrix} (\cos(e^{w_1} + e^{2w_2}) + e^{e^{w_1} + e^{2w_2}})e^{w_1} & 2(\cos(e^{w_1} + e^{2w_2}) + e^{e^{w_1} + e^{2w_2}})e^{2w_2} \\ \frac{e^{w_1}}{(1+e^{w_1})^2} + w_2 & w_1 \end{bmatrix}$$

2.4 Backward mode auto-differentiation

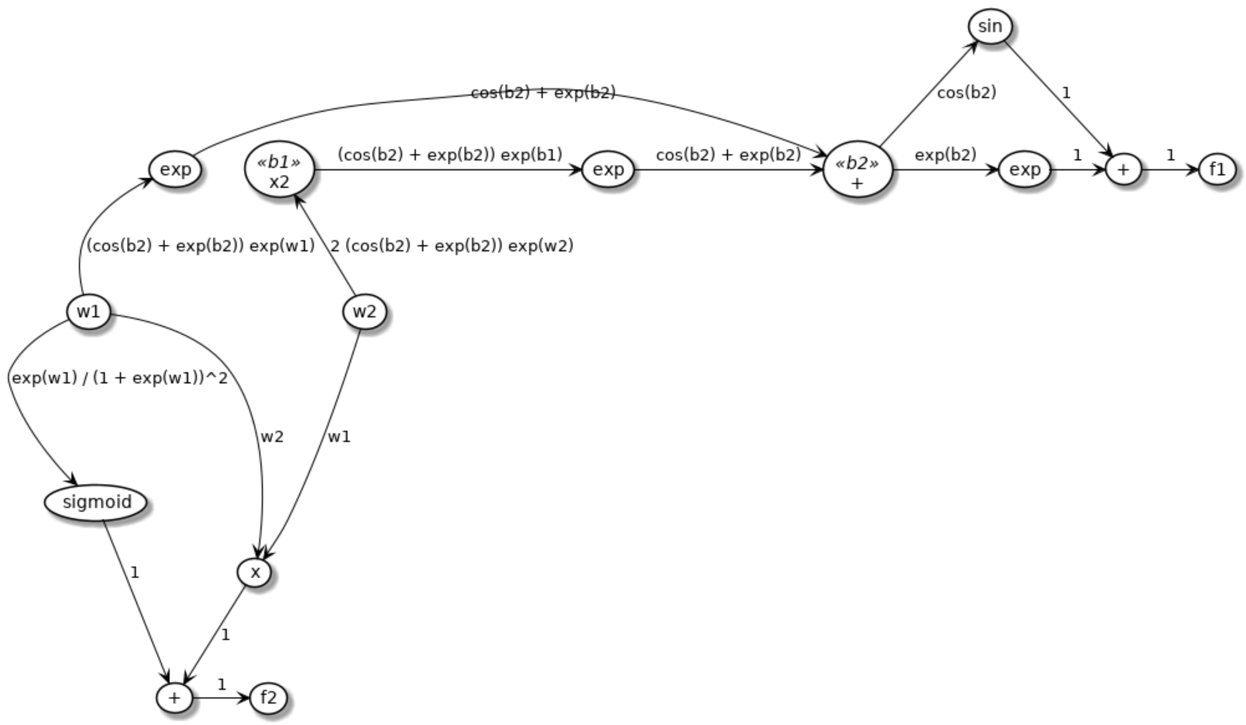


Figure 4: Backward gradient decent on $\vec{f}$

Which in a more mathematical format give us:

$$\frac{\partial \vec{f}}{\partial \vec{w}} = \begin{bmatrix} (\cos(e^{w_1} + e^{2w_2}) + e^{e^{w_1}+e^{2w_2}})e^{w_1} & 2(\cos(e^{w_1} + e^{2w_2}) + e^{e^{w_1}+e^{2w_2}})e^{2w_2} \\ \frac{e^{w_1}}{(1+e^{w_1})^2} + w_2 & w_1 \end{bmatrix}$$

2.5 Automation

Sure!

3 Directed Acyclic Graphs (DAG)

3.1 DAG imply topological ordering

If G is a DAG, then according to the lemma, at least one node has on input edges. If we took such a node out of the graph as well as all the edges going from this node, the remaning graph would be a DAG too (we only removed edges, we can't create cycle be removing edges). We can now repeat the previous step on the newly created graph, until we got an empty graph.
The order in which we took the node out is a possible topological ordering.

3.2 topological ordering imply DAG

If a graph has a topological ordering, then there is no edges going backward following this ordering. Which mean that there is no cycle on the graph, as you can't go back to the place where you started to try creating your graph, no matter where you try. For the same reason there is either no edge in the graph or all the edges are directed.
So, if we have a graph with a valid topological ordering, we have a graph with directed edges and no cycle, a DAG.

4 Implement and train a network on CIFAR-10

4.1 cs231n

softmax

September 26, 2019

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

```
In [1]: # 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
```

```
In [2]: from load_cifar10_tvt import load_cifar10_train_val

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

```
Train, validation and testing sets have been created as
 X_i and y_i where i=train,val,test
Train data shape:  (3073, 49000)
Train labels shape:  (49000,)
Val data shape:  (3073, 1000)
Val labels shape:  (1000,)
Test data shape:  (3073, 1000)
Test labels shape:  (1000,)
```

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

In [3]:
```
# 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: 2.382422e+00 computed in 0.585892s
loss: 2.382422
sanity check: 2.302585
numerical: -4.046185 analytic: -4.046185, relative error: 9.531160e-09
numerical: -1.985439 analytic: -1.985439, relative error: 1.308065e-09
numerical: -1.685963 analytic: -1.685963, relative error: 3.072309e-09
numerical: -0.192502 analytic: -0.192502, relative error: 4.025777e-07
numerical: -1.057372 analytic: -1.057372, relative error: 1.039151e-08
numerical: -2.068152 analytic: -2.068152, relative error: 5.768658e-09
numerical: 1.733033 analytic: 1.733033, relative error: 2.407816e-08
numerical: 0.965264 analytic: 0.965264, relative error: 6.165870e-08
numerical: 0.956019 analytic: 0.956019, relative error: 2.961977e-08
```

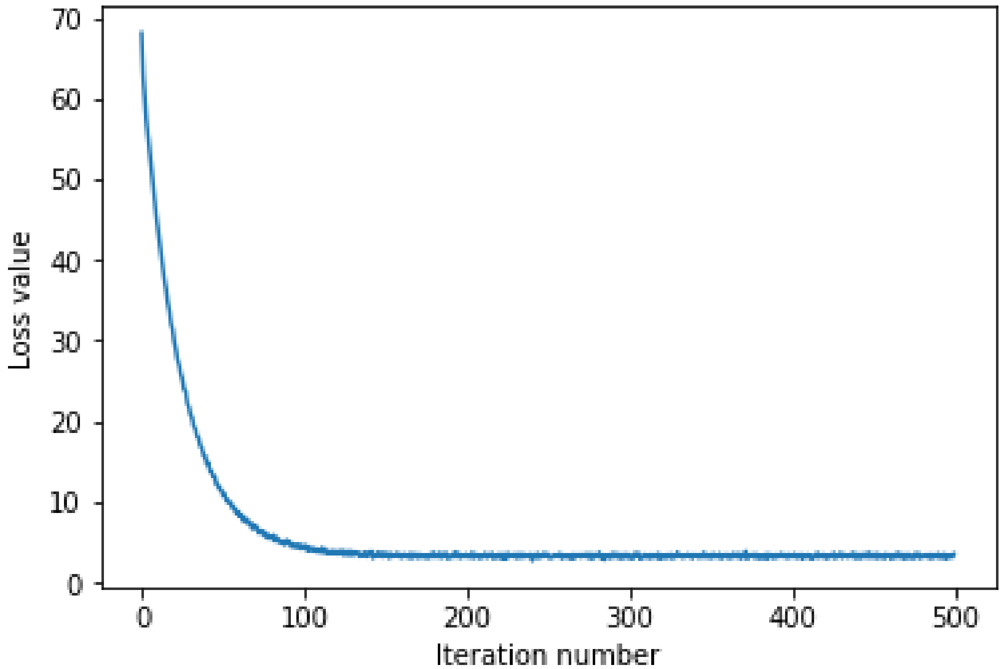
```
numerical: -3.049205 analytic: -3.049205, relative error: 2.257660e-10
```

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

```
In [27]: # Now that efficient implementations to calculate loss function and gradient of the s
         # 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-5,
             reg=2000,
             num_iters=500,
             batch_size=8000,
             verbose=True,
         )
         # Plot loss vs. iterations
         plt.plot(loss_hist)
         plt.xlabel("Iteration number")
         plt.ylabel("Loss value")
```

```
iteration 0 / 500: loss 68.132452
iteration 100 / 500: loss 4.206699
iteration 200 / 500: loss 3.127493
iteration 300 / 500: loss 2.996467
iteration 400 / 500: loss 3.226845
```

```
Out[27]: Text(0,0.5,'Loss value')
```

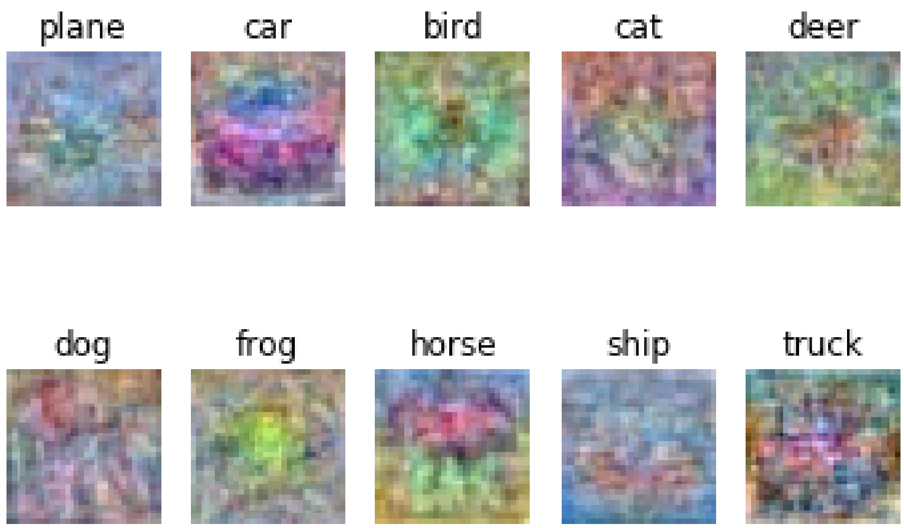



```
In [28]: # Complete the `predict` function in cs231n/classifiers/linear_classifier.py
         # Evaluate on test set
         y_test_pred = classifier.predict(X_test)
         test_accuracy = np.mean(y_test == y_test_pred)
         print("softmax on raw pixels final test set accuracy: %f" % (test_accuracy,))
```

```
softmax on raw pixels final test set accuracy: 0.254000
```

```
In [29]: # 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

September 26, 2019

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.

```
In [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.

```
In [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,
```

```
    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
    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.

```
In [11]: 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:
3.848682303062012e-08
```

3 Forward pass: compute loss

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

```
In [15]: from cs231n.classifiers.neural_net import two_layer_net

         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)))
```

```
(5, 3) (5, 10) (10, 3)
(5, 3) (4, 10)
(5, 10) (5, 4)
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:

```
In [23]: 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)[
           print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[pa
```

```
W2 max relative error: 8.023743e-10
b2 max relative error: 8.190173e-11
W1 max relative error: 4.426512e-09
b1 max relative error: 5.435430e-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:

```
In [24]: 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
         best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                                   model, two_layer_net,
                                                   reg=0.001,
                                                   learning_rate=1e-1, momentum=0.0, learnin
                                                   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:

```
In [41]: 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
         best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                                   model, two_layer_net,
                                                   reg=0.001,
                                                   learning_rate=1e-1, momentum=0.9, learnin
                                                   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:

```
In [43]: 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
         best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                                    model, two_layer_net,
                                                    reg=0.001,
                                                    learning_rate=1e-1, momentum=0.9, learnin
                                                    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.

```
In [44]: 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.

```
In [45]: 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 cl
         trainer = ClassifierTrainer()
         best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_train, X_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: 0.100000, val 0.113000, 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 2.287704, train: 0.159000, val 0.153000, 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 2.081489, train: 0.249000, val 0.243000, 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 1.972086, train: 0.311000, val 0.297000, 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 1.863844, train: 0.308000, val 0.340000, 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 1.774964, train: 0.350000, val 0.370000, lr 7.737809e-06
finished optimization. best validation accuracy: 0.370000
```

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.

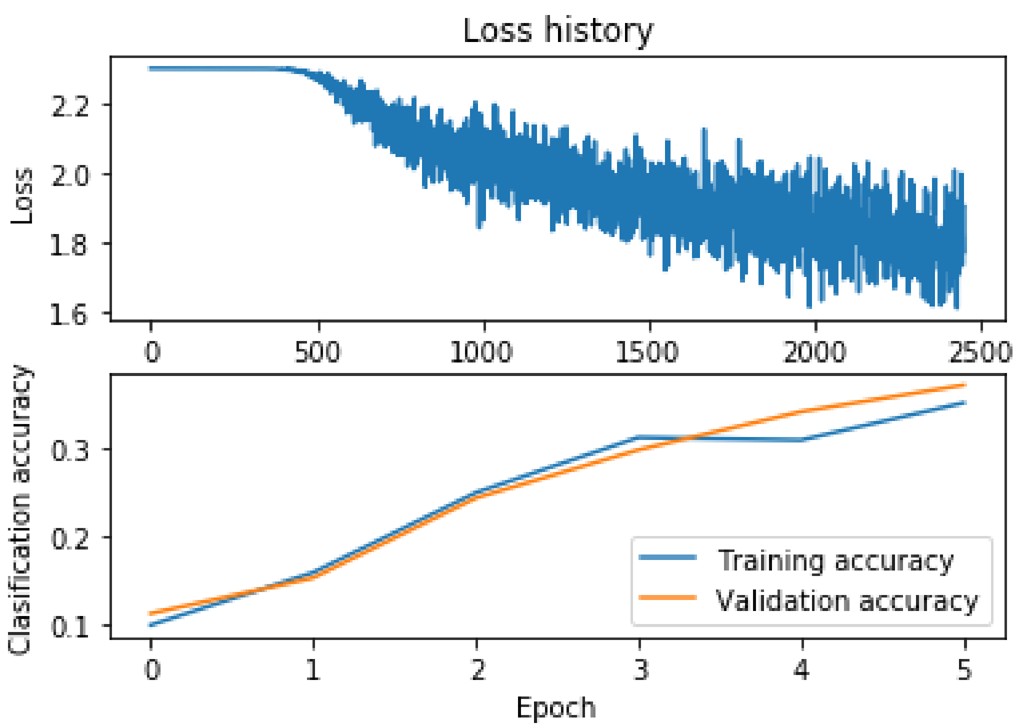
One strategy for getting insight into what's wrong is to plot the loss function and the accuracies on the training and validation sets during optimization.

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.

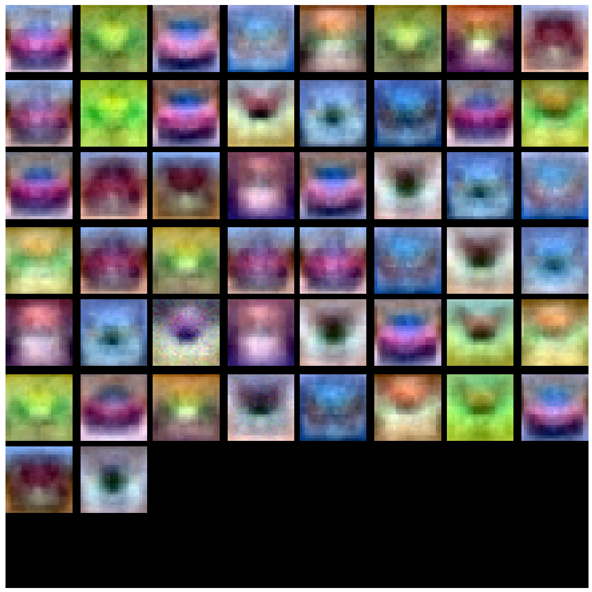
```
In [46]: # Plot the loss function and train / validation accuracies
         plt.subplot(2, 1, 1)
         plt.plot(loss_history)
         plt.title('Loss history')
         plt.xlabel('Iteration')
         plt.ylabel('Loss')

         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')
```

```
Out[46]: Text(0,0.5,'Clasification accuracy')
```




```
In [47]: 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
    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.

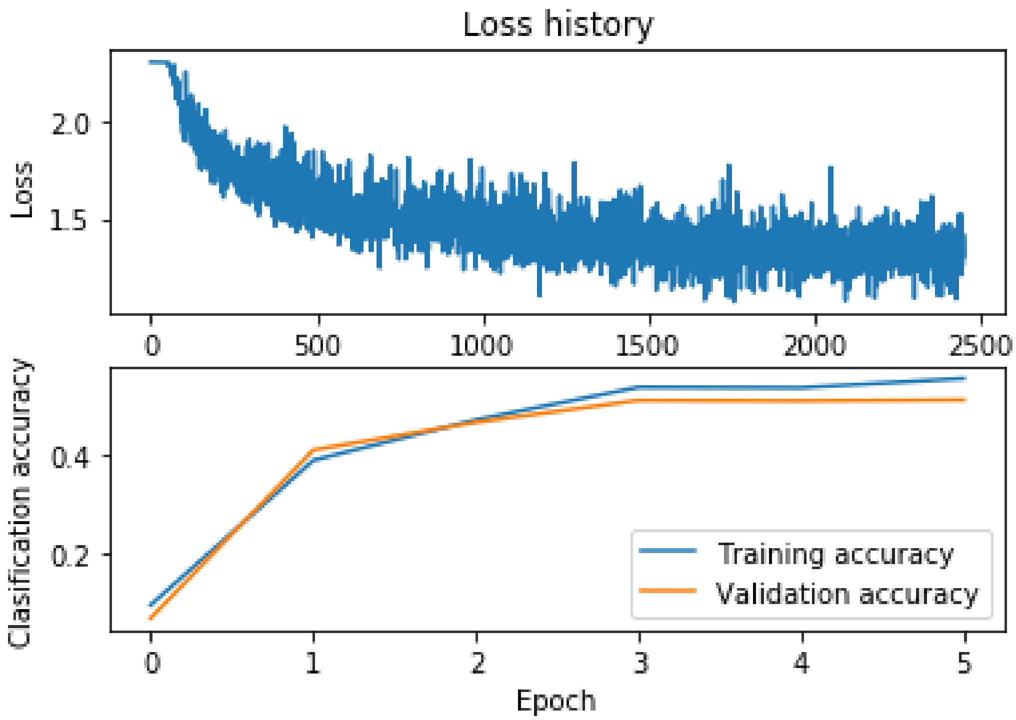
Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. For every 1% above 56% on the Test set we will award you with one extra bonus point. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding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

In [138]:
```python
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, 100, 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=5, reg=0.2,
                                              momentum=0.95,
                                              learning_rate_decay=0.5,
                                              learning_rate=1e-4, verbose=True)
################################################################################
#                                 END OF YOUR CODE                              #
################################################################################
```

```
Finished epoch 0 / 5: cost 2.302588, train: 0.094000, val 0.067000, lr 1.000000e-04
Finished epoch 1 / 5: cost 1.742135, train: 0.391000, val 0.412000, lr 5.000000e-05
Finished epoch 2 / 5: cost 1.244785, train: 0.474000, val 0.469000, lr 2.500000e-05
Finished epoch 3 / 5: cost 1.499905, train: 0.541000, val 0.514000, lr 1.250000e-05
Finished epoch 4 / 5: cost 1.532929, train: 0.540000, val 0.513000, lr 6.250000e-06
Finished epoch 5 / 5: cost 1.305601, train: 0.559000, val 0.515000, lr 3.125000e-06
finished optimization. best validation accuracy: 0.515000
```

In [139]:
```python
# visualize the weights
show_net_weights(best_model)
```

```
In [140]: # 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')

Out[140]: Text(0,0.5,'Clasification accuracy')
```



10 Run on the test set

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

```
In [141]: scores_test = two_layer_net(X_test, best_model)
          print('Test accuracy: ', np.mean(np.argmax(scores_test, axis=1) == y_test))
```

```
Test accuracy:  0.515
```

layers

September 26, 2019

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
```

```
In [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_gradie
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:

```
In [2]: # Test the affine_forward function
        from cs231n.layers import *

        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:  9.769849468192957e-10
```

3 Affine layer: backward

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

```
In [3]: # 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:  8.193994779273482e-10
dw error:  5.73045306371578e-11
db error:  3.1766143326853046e-11
```

4 ReLU layer: forward

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

```
In [4]: # 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:

```
In [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:  3.2756077431014307e-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.

```
In [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:')
        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:  9.000323981992523
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.30261796361517
dx error:  8.29724704294113e-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:

```
In [7]: 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
```

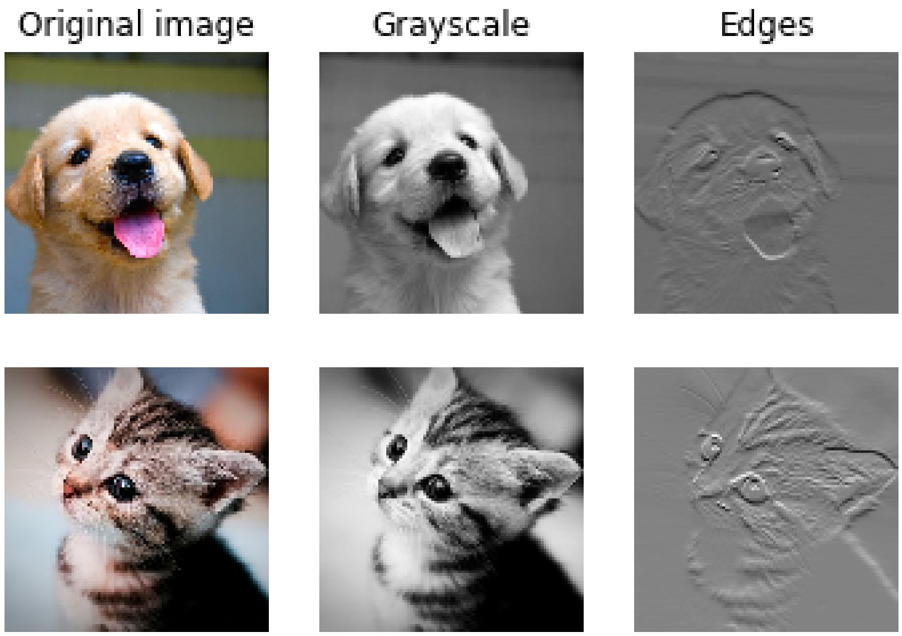
8 Aside: Image processing via convolutions

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.

In [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
# 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')
```

```python
# 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()
```

```
/home/nicolas/.local/lib/python3.6/site-packages/ipykernel_launcher.py:3: DeprecationWarning:
`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
/home/nicolas/.local/lib/python3.6/site-packages/ipykernel_launcher.py:10: DeprecationWarning:
`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.
/home/nicolas/.local/lib/python3.6/site-packages/ipykernel_launcher.py:11: DeprecationWarning:
`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.

```
In [9]: 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
        dw_num = eval_numerical_gradient_array(lambda w: conv_forward_naive(x, w, b, conv_param
        db_num = eval_numerical_gradient_array(lambda b: conv_forward_naive(x, w, b, conv_param

        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))
```

```
Testing conv_backward_naive function
dx error:  9.462213210626812e-09
dw error:  2.7957966454622364e-10
db error:  1.3426584939780371e-11
```

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`.

```
In [10]: 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.

```
In [11]: 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)

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:  3.2756242663824225e-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:

```
In [12]: 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: 3.033225s
Fast: 0.006476s
Speedup: 468.402047x
Difference:  1.8988659336214176e-11

Testing conv_backward_fast:
Naive: 4.840777s
Fast: 0.010355s
Speedup: 467.482317x
dx difference:  2.8711588418023044e-11
dw difference:  2.9630590969060655e-13
db difference:  1.2146998481782131e-14
```

In [13]:
```
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: 0.010396s
fast: 0.002350s
speedup: 4.423760x
difference:  0.0

Testing pool_backward_fast:
Naive: 0.147004s
speedup: 20.530084x
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:

```
In [14]: 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_
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_pool_forward(x, w, b, conv_
db_num = eval_numerical_gradient_array(lambda b: conv_relu_pool_forward(x, w, b, conv_

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

```
Testing conv_relu_pool_forward:
dx error:  1.268538101543734e-08
dw error:  2.1978135228663684e-10
db error:  9.132374492533775e-12
```

In [15]:
```
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
dw_num = eval_numerical_gradient_array(lambda w: conv_relu_forward(x, w, b, conv_param
db_num = eval_numerical_gradient_array(lambda b: conv_relu_forward(x, w, b, conv_param

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:  1.683050969419403e-09
dw error:  5.194184067114542e-10
db error:  1.3387377343956088e-11
```

In [16]:
```
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, d
dw_num = eval_numerical_gradient_array(lambda w: affine_relu_forward(x, w, b)[0], w, d
db_num = eval_numerical_gradient_array(lambda b: affine_relu_forward(x, w, b)[0], b, d

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:  8.43401795782027e-10
dw error:  9.119546076538304e-10
db error:  2.5480057504049307e-11
```

convnet

September 26, 2019

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`.

In [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))))
```

In [2]:
```
from cs231n.data_utils import load_CIFAR10

def get_CIFAR10_data(num_training=49000, num_validation=1000, num_test=1000):
    """
    Load the CIFAR-10 dataset from disk and perform preprocessing to prepare
    it for the two-layer neural net classifier. These are the same steps as
```

```python
    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.

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

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

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

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

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

```
Sanity check loss (no regularization):  2.302547349828836
Sanity check loss (with regularization):  2.3447415665637705
```

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.

```
In [4]: 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
            e = rel_error(param_grad_num, grads[param_name])
            print('%s max relative error: %e' % (param_name, rel_error(param_grad_num, grads[pa
```

```
W1 max relative error: 7.991980e-07
W2 max relative error: 3.118514e-06
b1 max relative error: 1.335139e-07
```

```
b2 max relative error: 1.577013e-09
```

4 Overfit small data

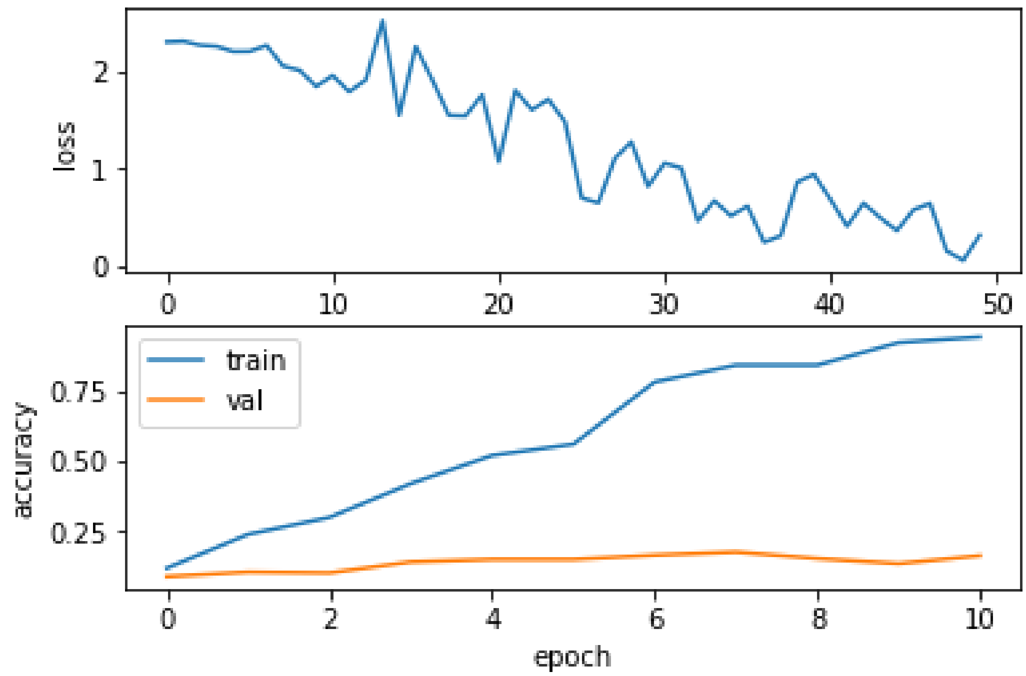
A nice trick is to train your model with just a few training samples. You should be able to overfit small datasets, which will result in very high training accuracy and comparatively low validation accuracy.

```
In [5]: # Use a two-layer ConvNet to overfit 50 training examples.

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

```
Finished epoch 0 / 10: cost 2.295300, train: 0.120000, val 0.090000, lr 1.000000e-04
Finished epoch 1 / 10: cost 2.197694, train: 0.240000, val 0.106000, lr 9.500000e-05
Finished epoch 2 / 10: cost 1.841652, train: 0.300000, val 0.103000, lr 9.025000e-05
Finished epoch 3 / 10: cost 1.543712, train: 0.420000, val 0.142000, lr 8.573750e-05
Finished epoch 4 / 10: cost 1.755838, train: 0.520000, val 0.151000, lr 8.145062e-05
Finished epoch 5 / 10: cost 1.478216, train: 0.560000, val 0.151000, lr 7.737809e-05
Finished epoch 6 / 10: cost 0.813337, train: 0.780000, val 0.166000, lr 7.350919e-05
Finished epoch 7 / 10: cost 0.512724, train: 0.840000, val 0.177000, lr 6.983373e-05
Finished epoch 8 / 10: cost 0.938180, train: 0.840000, val 0.154000, lr 6.634204e-05
Finished epoch 9 / 10: cost 0.360220, train: 0.920000, val 0.136000, lr 6.302494e-05
Finished epoch 10 / 10: cost 0.313373, train: 0.940000, val 0.164000, lr 5.987369e-05
finished optimization. best validation accuracy: 0.177000
```

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

```
In [6]: 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.

```
In [17]: 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)
```

```
Finished epoch 0 / 1: cost 2.301658, train: 0.092000, val 0.086000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.835203, train: 0.309000, val 0.312000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.710324, train: 0.343000, val 0.387000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.771179, train: 0.402000, val 0.398000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.802380, train: 0.435000, val 0.413000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.634422, train: 0.432000, val 0.424000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.836016, train: 0.453000, val 0.455000, lr 1.000000e-04
```

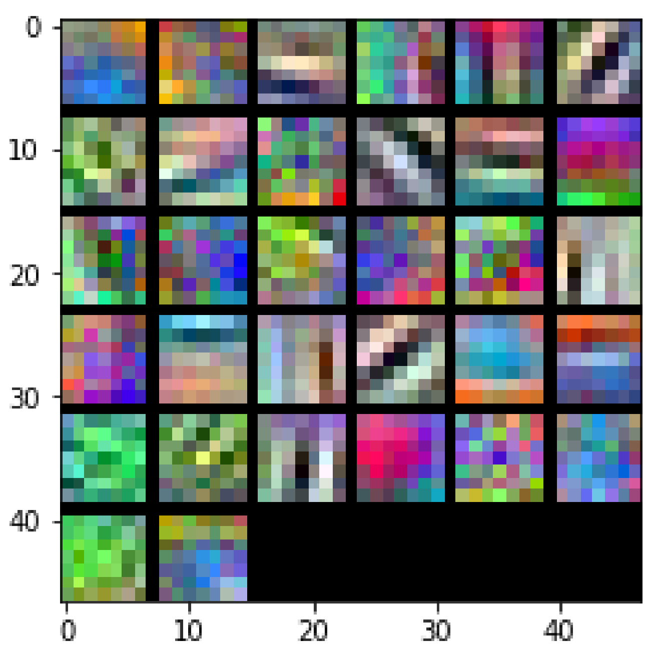
```
Finished epoch 0 / 1: cost 1.496180, train: 0.408000, val 0.411000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.692247, train: 0.456000, val 0.494000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.401887, train: 0.466000, val 0.493000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.516343, train: 0.478000, val 0.486000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.507644, train: 0.457000, val 0.496000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.967142, train: 0.476000, val 0.477000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.806298, train: 0.478000, val 0.484000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.660422, train: 0.475000, val 0.459000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.628083, train: 0.488000, val 0.474000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.359960, train: 0.470000, val 0.487000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.604342, train: 0.484000, val 0.440000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.022019, train: 0.523000, val 0.501000, lr 1.000000e-04
Finished epoch 0 / 1: cost 1.772906, train: 0.492000, val 0.480000, lr 1.000000e-04
Finished epoch 1 / 1: cost 2.590106, train: 0.496000, val 0.480000, lr 9.500000e-05
finished optimization. best validation accuracy: 0.501000
```

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.

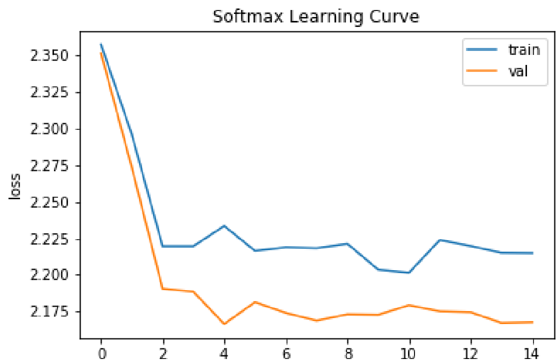
In [18]:
```
from cs231n.vis_utils import visualize_grid

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

Out[18]: <matplotlib.image.AxesImage at 0x7f393751c438>

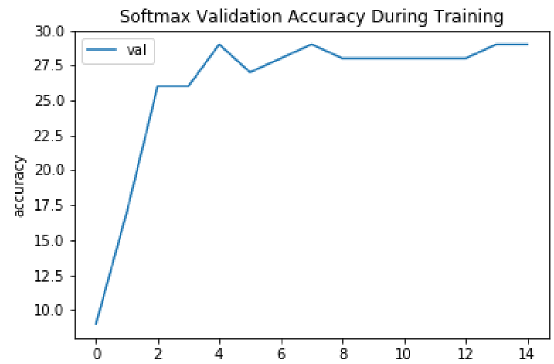


4.2 PyTorch

4.2.1 Softmax

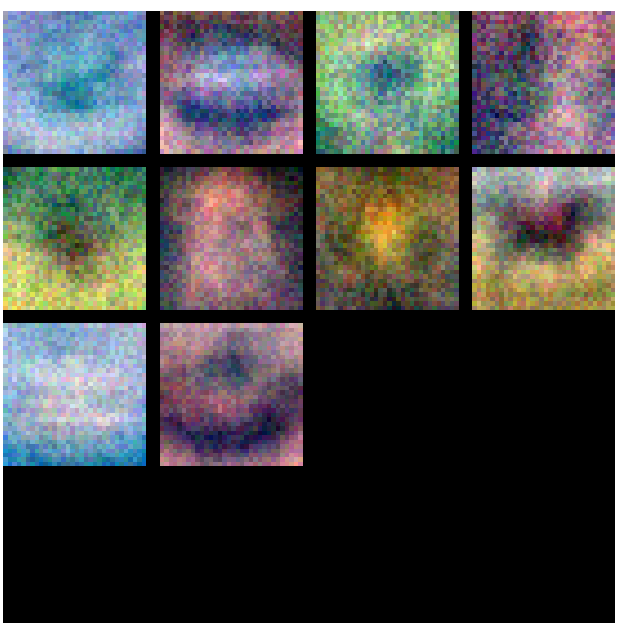


(a) Evolution of loss during training

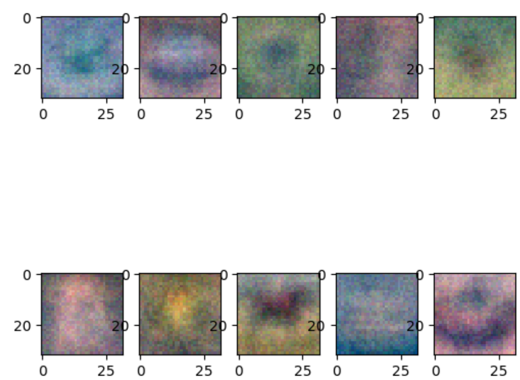


(b) Evolution of validation accuracy during training

Figure 5: Evolution of different indicators during training



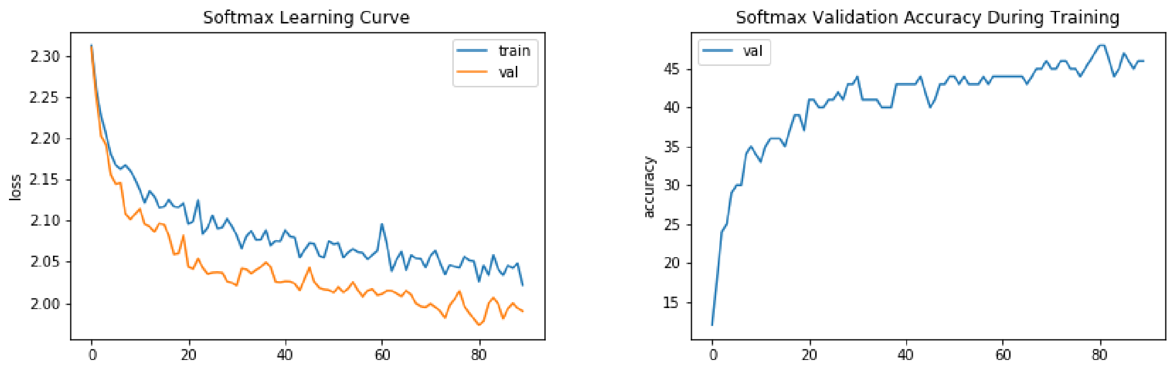
(a) Filter of the trained softmax model displayed as a grid



(b) Filter of the trained softmax model

Figure 6: Weights visualisation after training

4.2.2 Two layers NN



(a) Evolution of loss during training (b) Evolution of validation accuracy during training

Figure 7: Evolution of different indicators during training

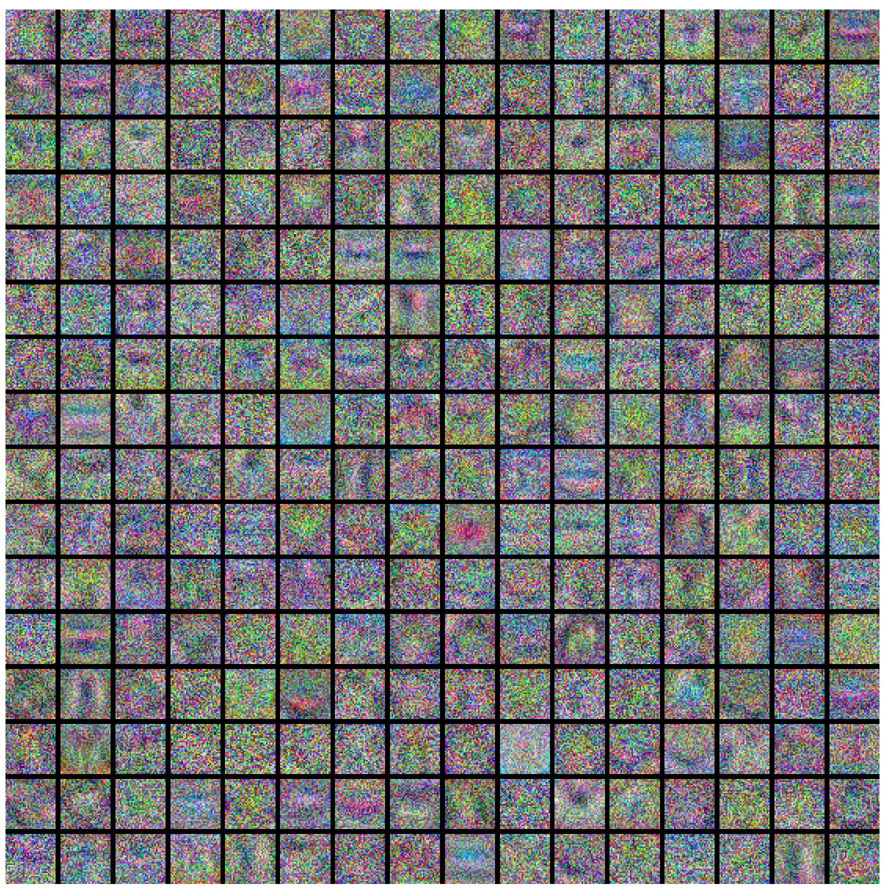
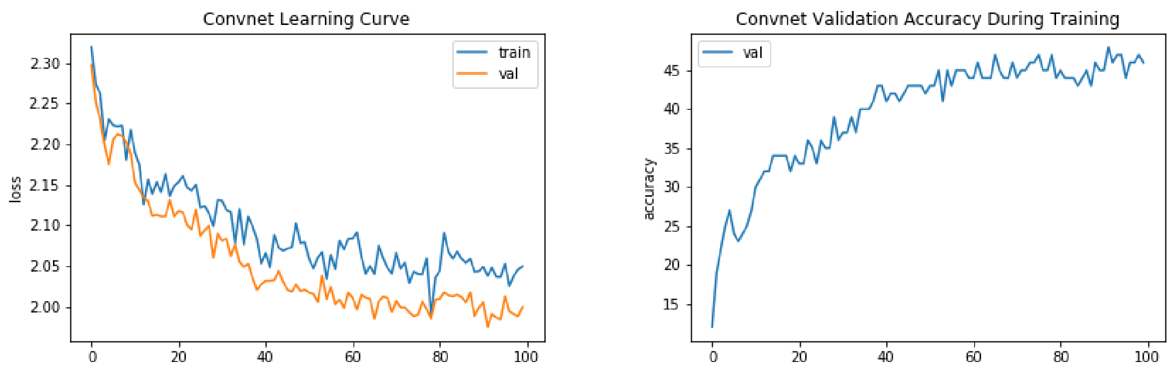


Figure 8: Filter of the trained softmax model displayed as a grid

4.2.3 Convnet



(a) Evolution of loss during training (b) Evolution of validation accuracy during training

Figure 9: Evolution of different indicators during training

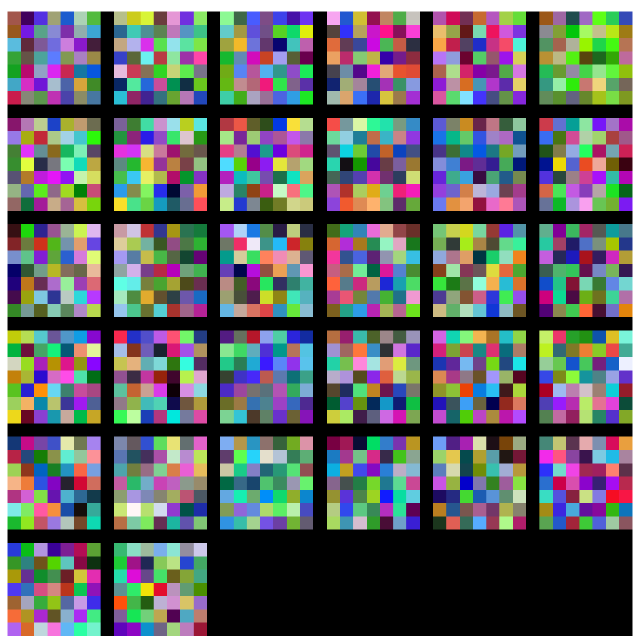
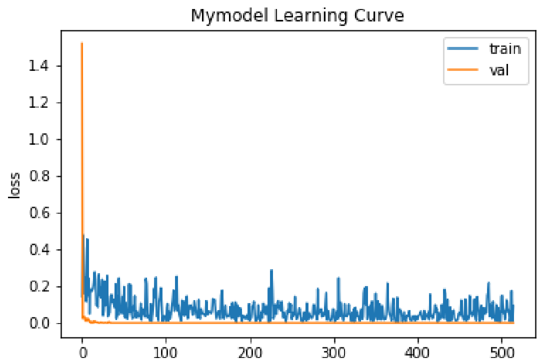
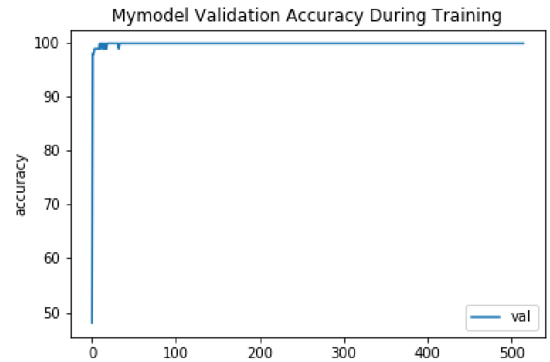


Figure 10: Filter of the trained softmax model displayed as a grid

4.2.4 Experimentation



(a) Evolution of loss during training



(b) Evolution of validation accuracy during training

Figure 11: Evolution of different indicators during training

For this section I used the code and pre-trained weights from https://github.com/quark0/darts I reimplemented the authors network on the homework format, modified the `train.py` script to allow fine tuning of existing weights and added data augmentation on the training data. The pre-trained model being already trained on Cifar-10 using architecture search, no much training were needed, one epoch was enough for the network to take into account the difference on dataset loading.
I expect one or two percent accuracy can still be gained by re running the architecture search and using more fine tuned data augmentation.

- Name: Nicolas Six
- Email ID: nsix6 (nsix6@gatech.edu)
- Best accuracy: 0.962 (EvalAI public set)