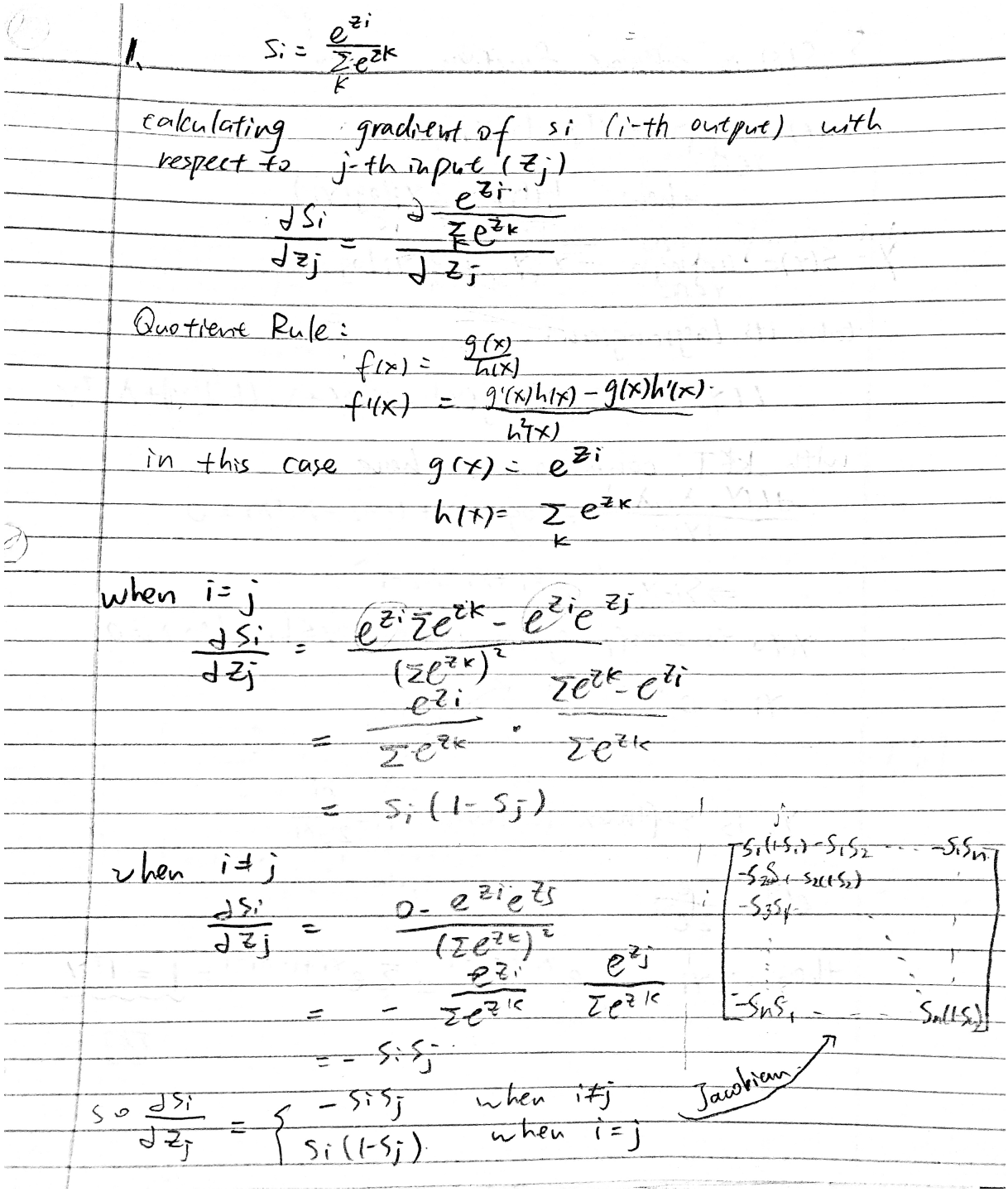
CS7643: Deep Learning
Fall 2020
HW1 Solutions

Tianxue Hu ID:903519328

September 9, 2020

1 Optimization

1.1 1.



1.2 2.

2.

prove "→"

If g has a local minimum at some point w^T.

let $w = (w_1 \cdots w_n)$ $\quad w^T = \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$

when g has local minimum at w^T.

then $g_{w_1}(w_1 \cdots w_n) = 0$

$\vdots$

$g_{w_n}(w_1 \cdots w_n) = 0$

then $\nabla g(w_1 \cdots w_n)$

$$= \begin{bmatrix} g_{w_1}(w_1 \cdots w_n) \\ \vdots \\ g_{w_n}(w_1 \cdots w_n) \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}$$

$\Rightarrow \nabla g(w^T) = 0$

"←" $\nabla f(w^T) = 0$, w^T may also be a reflection point or saddle point. not guaranteed as local min/max

e.g. $g(w_1, w_2) = w_1^2 - w_2^2$

$$\nabla g = \begin{bmatrix} g_{w_1}(w_1, w_2) \\ g_{w_2}(w_1, w_2) \end{bmatrix} = \begin{bmatrix} 2w_1 \\ -2w_2 \end{bmatrix} = 0$$

$\Rightarrow w_1 = 0, \ w_2 = 0 \quad w^T = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

$g(0,0) = 0$

However. $g(w_1, w_2) \to -\infty$ when $w_2 \to \infty, w_1 = 0$

or $g(w_1, w_2) \to \infty$ when $w_1 \to \infty, w_2 = 0$

$\nabla g = 0$ at w^T not guarantee w^T is a local min

1.3 3.

3. If g is differentiable $:\mathbb{R}^n \to \mathbb{R}$ and $\nabla g(w^*)=0$.
we want to prove w^* is a global minimum of g.
then we want to prove that for any w :
$$g(\vec{w}) \geq g(\vec{w^*})$$

If g is convex. by definition, for $\forall \lambda \in (0,1)$, $\vec{x}, \vec{y} \in$ dom(g)
$$g(\lambda\vec{y}+(1-\lambda)\vec{x}) \leq \lambda g(\vec{y})+(1-\lambda)g(\vec{x})$$
which is $g(\vec{x}+\lambda(\vec{y}-\vec{x})) \leq g(\vec{x})+\lambda(g(\vec{y})-g(\vec{x}))$
$$\Rightarrow \quad g(\vec{y})-g(\vec{x}) \geq \frac{1}{\lambda}(g(\vec{x}+\lambda(\vec{y}-\vec{x}))-g(\vec{x}))$$

as $\lambda \to 0$, we get
$$g(\vec{y})-g(\vec{x}) \geq \nabla g^T(\vec{x})(\vec{y}-\vec{x}).$$

when $\nabla g(\vec{x})=0$,
$$g(\vec{y}) \geq g(\vec{x}) \quad \text{for all } \vec{y}$$
plugin $\vec{w}$ as $\vec{y}$, $\vec{w^*}$ as $\vec{x}$
$$g(\vec{w}) \geq g(w^*) \quad \text{for all } \vec{w}$$

1.4 4.

4.

$f(w) = \frac{1}{2}(w-2)^2 + \frac{1}{2}(w+1)^2$

$f_1(w) = \frac{1}{2}(w-2)^2$ $\quad$ $f_2(w) = \frac{1}{2}(w+1)^2$

$f_1'(w) = w-2$ $\quad$ $f_2'(w) = w+1$

$P(f_1(w)) = P(f_2(w)) = \frac{1}{2}$

update SGD by $w^{(t+1)} = w^{(t)} - \eta f'(w)$

let $w^{(1)} = 0$

$f(w^{(1)}) = \frac{1}{2}(0-2)^2 + \frac{1}{2}(0+1)^2 = 2.5$

gradient $f'(w) = -2$ if $f_1'(w)$

1 if $f_2'(w)$

If using $f_1(w)$:

$w^{(2)} = w^{(1)} - \eta f_1'(w) = 2\eta$

$f(w^{(2)}) = \frac{1}{2}(2\eta-2)^2 + \frac{1}{2}(2\eta+1)^2$

$= 4\eta^2 - 4\eta + 5 = (2\eta+2)^2 + 1 \geq 1$

If using $f_2(w)$:

$w^{(2)} = w^{(1)} - \eta f_2'(w) = -\eta$

$f(w^{(2)}) = \frac{1}{2}(-\eta-2)^2 + \frac{1}{2}(-\eta+1)^2$

$= \eta^2 + \eta + \frac{5}{2} = (\eta+\frac{1}{2})^2 + \frac{9}{4} \geq 2.25$

Thus, we cannot guarantee SGD will decrease

1.5 5.

5. $S(x)$ is softmax function.

$$S(x) = \underset{Y \in \mathbb{R}^d}{\operatorname{argmin}} \; -X^T Y - H(Y)$$

where $H(Y) = -\sum_i Y_i \log(Y_i)$

$$Y = S(x) = \underset{Y \in \mathbb{R}^d}{\operatorname{argmin}} \; -X^T Y + \sum_i Y_i \log(Y_i)$$

take its Lagrangian:

$$L(Y, \lambda_1, \lambda_2) = Y^T \log(Y) - X^T Y + \lambda_1 (1 - 1^T Y) + \lambda_2^T Y$$

with KKT condition, we have

$$\frac{dL(Y, \lambda_1, \lambda_2)}{dY_i} = \log(Y_i) + 1 - X_i - \lambda_1 + \lambda_2 = 0$$

$$\Rightarrow S_i = Y_i = e^{X_i + \lambda_1 - \lambda_2 - 1}$$

Y_i to to satisfy $\log(Y_i)$, and $0 < Y \leq 1$, then $\lambda_2 = 0$

$$Y_i = e^{X_i + \lambda_1 - 1}$$

plug in

Y_i is softmax function $Y_i = \frac{e^i}{\sum_i e^{X_i}}$

$$e^{\lambda_1} = \frac{e}{\sum_i e^{X_i}}$$

then $\sum_i Y_i = \sum_i e^{X_i + \lambda_1 - 1} = \sum_i e^{X_i + 1 - X_i - 1} = 1 = 1^T Y$

2 Directed Acyclic Graphs (DAG)

2.1 6.

6. If graph G is a DAG, then G has a topological ordering

Proof: by induction, n is num of nodes in G

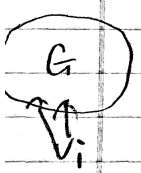
Base case: if n=1, then G has topological order.

Inductive steps:

Assume G' is DAG of size n+1, G has topological order, G' have nodes $\{v_1, \dots v_{n+1}\}$

Prove G is DAG of size n, also has topological order.

Proof: finding a node v_i with no incoming edges in G'



$G' - v_i$ is also DAG as deleting v wouldn't creating cycles.

for v_i, An edge (v_i, v_j) must be deleted before deletion of v_j

v_i must be deleted before v_j

then G of nodes $\{v_1 \dots v_n\}$

for any $v_i v_j$ in $\{v_1 \dots v_n\}$, $i<j$ holds for every edge (v_i, v_j)

so G is also a DAG has topological order.

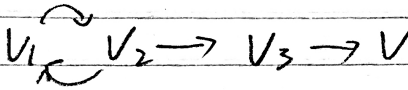
Inductive case proved.

2.2 7.

7. If G has a topological ordering, then G is DAG.

Proof: By contradiction

suppose G has topological ordering $\{v_1 \dots v_n\}$ and G is direct cyclic graph like this:



we have $v_1 \to v_2$ and $1<2$, which is true
However, $v_2 \to v_1$ and $2<1$ is not true.
which doesn't satisfied the definition of topological ordering.
By contradiction. G must be direct Asyclic graph

3 Paper Review

3.1 8. Briefly summarize the key contributions, strengths and weaknesses of this paper.

Key contributions

The paper proposed a search method for neural networks architectures that can already perform tasks without explicit weight training. And the method can generate minimal neural network architectures to perform some reinforcement learning tasks without training weight. This is done by assigning a single shared weight parameter to every network connection, evaluating the network on a wide range of this single weight parameter, rank networks by performance and complexity, repeat the process by create new population by varying best networks, and choosing probabilistically through tournament selection.

Strengths

Previous neural architecture search (NAS) methods never claimed that the solution is innate to the structure of the network. This weight agnostic neural network search is efficient and simple by assigning a single shared weight parameter to every network connectionï¼and this single parameter can easily be tuned to increase performance.

Weaknesses

One weakness of this method is that, as they mentioned, it is highly sensitive to the initial weight. From the result table, WANN fails when individual weight values are assigned randomly. Also, adding larger noise to the weight the WANN performs poorly.

3.2 9. What is your personal takeaway from this paper?

I think it is really good of them to think outside the box. While other NAS methods are more complicated focusing on evaluating by training the models, their method focusing on a shared weight and testing performances through tournament selection. I'm wondering how this methods works for NLP tasks and related models such as seq2seq and transformer model.

4 CIFAR-10 Implementation

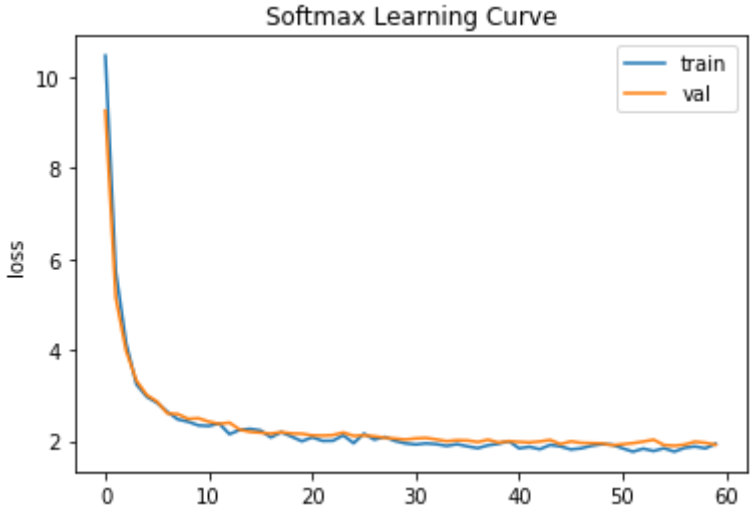
4.1 10.1 Softmax Regression

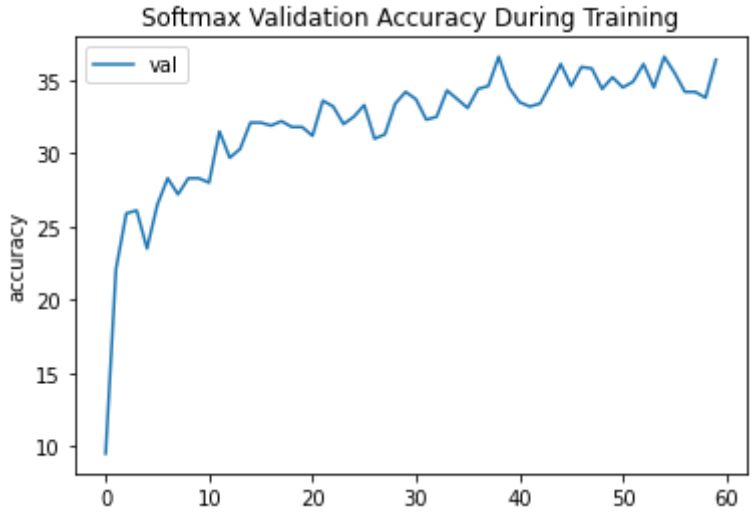
Visualizing the PyTorch model

In [8]:
```
# Assuming that you have completed training the classifer, let us plot the
# example to show a simple way to log and plot data from PyTorch.

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

In [9]:
```
# Parse the train and val losses one line at a time.
import re
# regexes to find train and val losses on a line
float_regex = r'[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?'
train_loss_re = re.compile('.*Train Loss: ({})'.format(float_regex))
val_loss_re = re.compile('.*Val Loss: ({})'.format(float_regex))
val_acc_re = re.compile('.*Val Acc: ({})'.format(float_regex))
# extract one loss for each logged iteration
train_losses = []
val_losses = []
val_accs = []
# NOTE: You may need to change this file name.
with open('softmax.log', 'r') as f:
    for line in f:
        train_match = train_loss_re.match(line)
        val_match = val_loss_re.match(line)
        val_acc_match = val_acc_re.match(line)
        if train_match:
            train_losses.append(float(train_match.group(1)))
        if val_match:
            val_losses.append(float(val_match.group(1)))
        if val_acc_match:
            val_accs.append(float(val_acc_match.group(1)))
```

In [10]:
```
fig = plt.figure()
plt.plot(train_losses, label='train')
plt.plot(val_losses, label='val')
plt.title('Softmax Learning Curve')
plt.ylabel('loss')
plt.legend()
fig.savefig('softmax_lossvstrain.png')

fig = plt.figure()
plt.plot(val_accs, label='val')
plt.title('Softmax Validation Accuracy During Training')
plt.ylabel('accuracy')
plt.legend()
fig.savefig('softmax_valaccuracy.png')
```

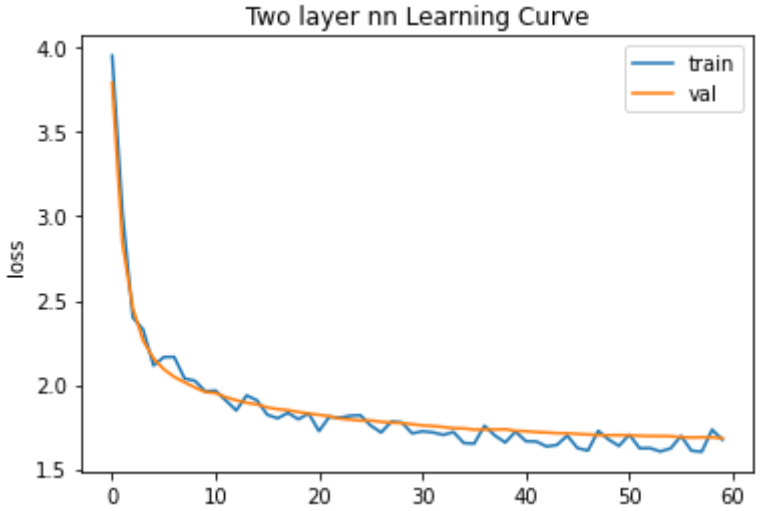


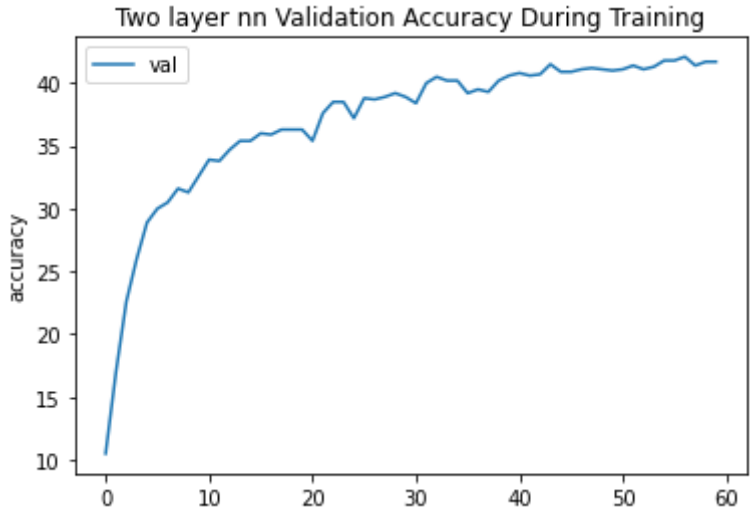


two layer nn

In [11]:
```
# Parse the train and val losses one line at a time.
import re
# regexes to find train and val losses on a line
float_regex = r'[-+]?(\d+(\.\d*)?|\.\d+)([eE][-+]?\d+)?'
train_loss_re = re.compile('.*Train Loss: ({})'.format(float_regex))
val_loss_re = re.compile('.*Val Loss: ({})'.format(float_regex))
val_acc_re = re.compile('.*Val Acc: ({})'.format(float_regex))
# extract one loss for each logged iteration
train_losses = []
val_losses = []
val_accs = []
# NOTE: You may need to change this file name.
with open('twolayernn.log', 'r') as f:
    for line in f:
        train_match = train_loss_re.match(line)
        val_match = val_loss_re.match(line)
        val_acc_match = val_acc_re.match(line)
        if train_match:
            train_losses.append(float(train_match.group(1)))
        if val_match:
            val_losses.append(float(val_match.group(1)))
        if val_acc_match:
            val_accs.append(float(val_acc_match.group(1)))
```

In [12]:
```
fig = plt.figure()
plt.plot(train_losses, label='train')
plt.plot(val_losses, label='val')
plt.title('Two layer nn Learning Curve')
plt.ylabel('loss')
plt.legend()
fig.savefig('twolayernn_lossvstrain.png')

fig = plt.figure()
plt.plot(val_accs, label='val')
plt.title('Two layer nn Validation Accuracy During Training')
plt.ylabel('accuracy')
plt.legend()
fig.savefig('twolayernn_valaccuracy.png')
```





In [ ]:

4.2 10.2 Two-layer Neural Network

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 [125]:
```
# 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-i
%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 autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

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 [126]:
```
# 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(
  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
  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,
  y = np.array([0, 1, 2, 2, 1])
  return X, y

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

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 [127]:
```
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
```

Forward pass: compute loss

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

In [128]:
```
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:
3.605849741239453e-06
```

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 [129]:
```
from cs231n.gradient_check import eval_numerical_gradient

# Use numeric gradient checking to check your implementation of the backwar
# 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 b

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
  print('%s max relative error: %e' % (param_name, rel_error(param_grad_num
```

```
W2 max relative error: 5.260597e-05
b2 max relative error: 6.317662e-06
W1 max relative error: 2.493902e-05
b1 max relative error: 1.449991e-03
```

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 [136]:
```
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 Gradien
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                               model, two_layer_net,
                                               reg=0.001,
                                               learning_rate=1e-1, momentum=0
                                               update='sgd', sample_batches=F
                                               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: 0.940683
```

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 [137]:
```
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 Gradien
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                               model, two_layer_net,
                                               reg=0.001,
                                               learning_rate=1e-1, momentum=0
                                               update='momentum', sample_batc
                                               num_epochs=100,
                                               verbose=False)
correct_loss = 0.494394
print('Final loss with momentum SGD: %f. We get: %f' % (loss_history[-1], c
```

```
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: 0.494391. 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 [138]:
```
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 Gradien
best_model, loss_history, _, _ = trainer.train(X, y, X, y,
                                               model, two_layer_net,
                                               reg=0.001,
                                               learning_rate=1e-1, momentum=0
                                               update='rmsprop', sample_batch
                                               num_epochs=100,
                                               verbose=False)
correct_loss = 0.439368
print('Final loss with RMSProp: %f. We get: %f' % (loss_history[-1], correc
```

```
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: 0.439363. We get: 0.439368
```

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 [139]:
```
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 prepar
    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,)
```

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 [140]:
```
from cs231n.classifiers.neural_net import init_two_layer_model

model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, nu
trainer = ClassifierTrainer()
best_model, loss_history, train_acc, val_acc = trainer.train(X_train, y_tra
                                                model, two_layer_net,
                                                num_epochs=5, reg=1.0,
                                                momentum=0.9, learning_rate_de
                                                learning_rate=1e-5, verbose=Tr
```

```
starting iteration  0
Finished epoch 0 / 5: cost 2.302583, train: 0.104000, val 0.100000, 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  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 1.690442, train: 0.375000, val 0.364000, lr 7.
737809e-06
finished optimization. best validation accuracy: 0.364000
```

Debug the training

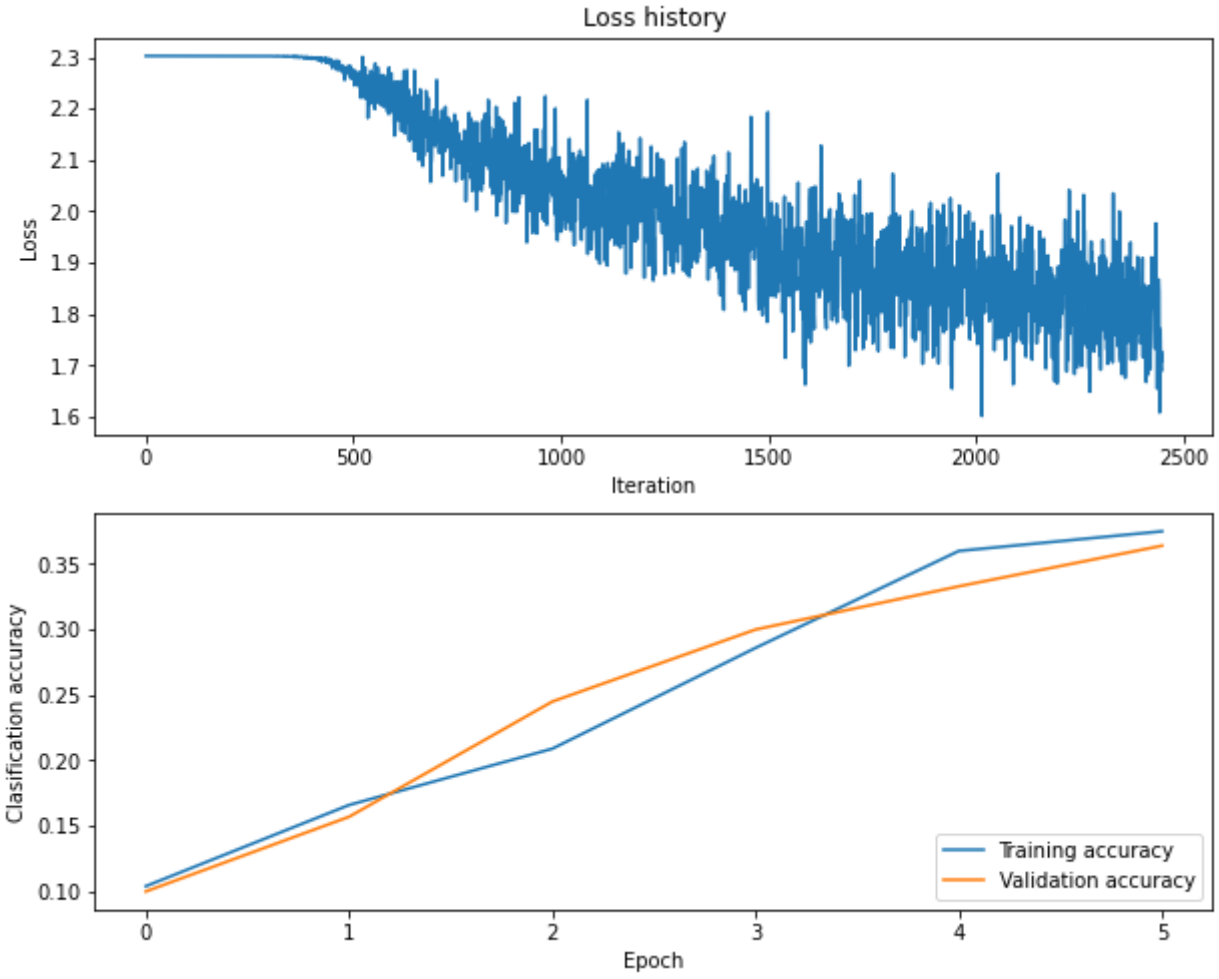
With the default parameters we provided above, you should get a validation accuracy of about 0.37 on the validation set. This isn't very good.

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.

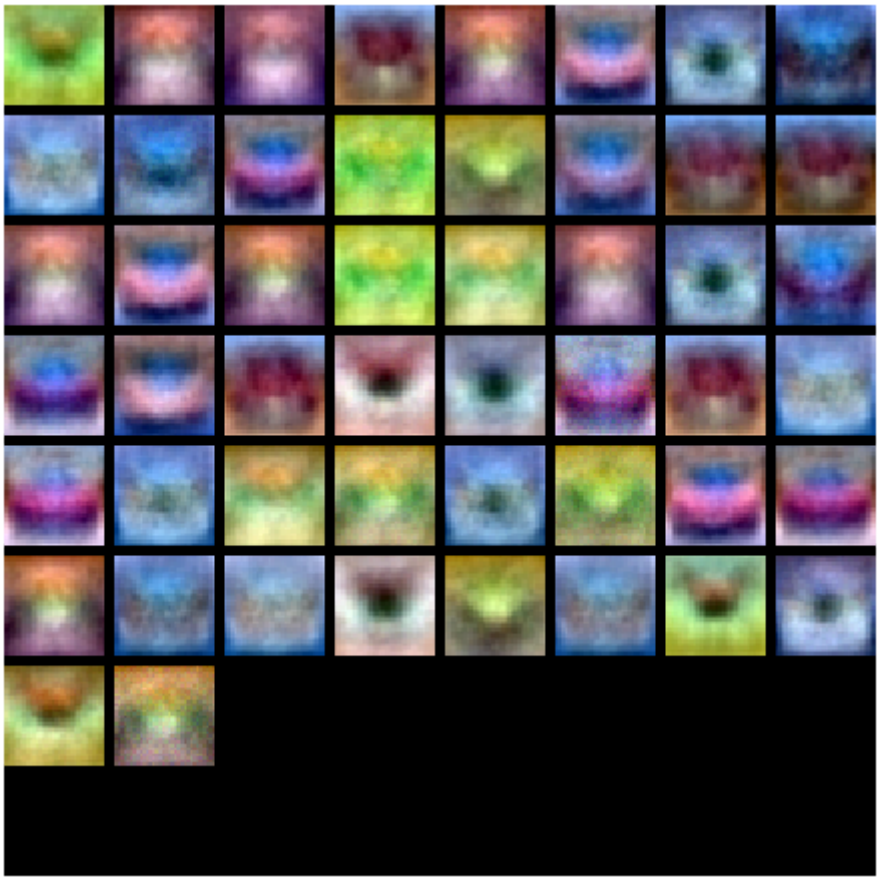
Another strategy is to visualize the weights that were learned in the first layer of the network. In most neural networks trained on visual data, the first layer weights typically show some visible structure when visualized.

In [141]:
```
# 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[141]:
```
Text(0, 0.5, 'Clasification accuracy')
```



In [142]:
```
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
    plt.gca().axis('off')
    plt.show()

show_net_weights(model)
```



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.

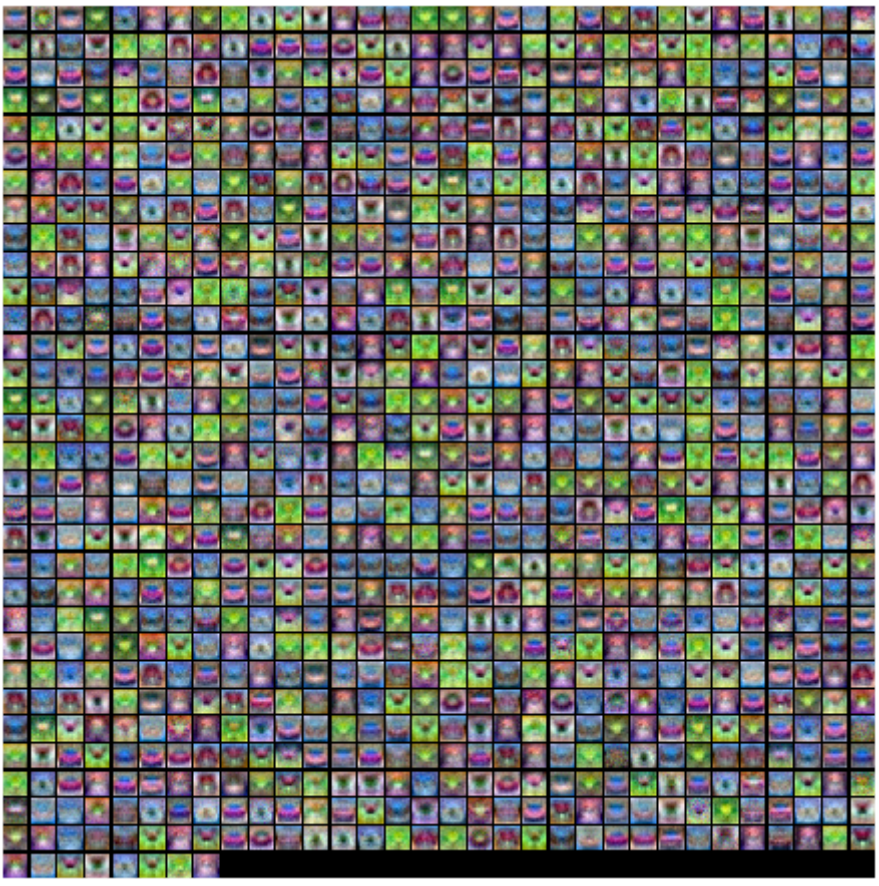
Experiment: You goal in this exercise is to get as good of a result on CIFAR-10 as you can, with a fully-connected Neural Network. For every 1% above 56% on the Test set we will award you with one extra bonus point. Feel free implement your own techniques (e.g. PCA to reduce dimensionality, or adding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

In [143]:
```
best_model = None # store the best model into this

################################################################################
# TODO: Tune hyperparameters using the validation set. Store your best trai
# model in best_model.
#
# To help debug your network, it may help to use visualizations similar to
# ones we used above; these visualizations will have significant qualitativ
# 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
# 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_tra
                                                X_val, y_val,
                                                model, two_layer_net,
                                                num_epochs=50, reg=0.01,
                                                momentum=0.92,
                                                learning_rate_decay=0.999,
                                                learning_rate=1e-6, verbose=Tr
################################################################################
#                                END OF YOUR CODE
################################################################################
```

```
starting iteration  0
Finished epoch 0 / 50: cost 2.302578, train: 0.108000, val 0.113000, lr
1.000000e-06
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  24320
starting iteration  24330
starting iteration  24340
starting iteration  24350
starting iteration  24360
starting iteration  24370
starting iteration  24380
starting iteration  24390
starting iteration  24400
starting iteration  24410
starting iteration  24420
starting iteration  24430
starting iteration  24440
starting iteration  24450
starting iteration  24460
starting iteration  24470
starting iteration  24480
starting iteration  24490
Finished epoch 50 / 50: cost 1.699020, train: 0.469000, val 0.428000, lr
9.512056e-07
finished optimization. best validation accuracy: 0.428000
```

In [144]:
```
# visualize the weights
show_net_weights(best_model)
```



Run on the test set

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

In [145]:
```
scores_test = two_layer_net(X_test, best_model)
print('Test accuracy: ', np.mean(np.argmax(scores_test, axis=1) == y_test))
```

```
Test accuracy:  0.402
```

In [ ]:
```

```

4.3 10.3 Modular Neural Network

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_numer
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-i
%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))))
```

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 [3]:
```
# 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), o
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))
```

```
(2, 4, 5, 6)
(2, 120)
Testing affine_forward function:
difference:  9.769849468192957e-10
```

Affine layer: backward

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

In [4]:
```
# 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]
dw_num = eval_numerical_gradient_array(lambda w: affine_forward(x, w, b)[0]
db_num = eval_numerical_gradient_array(lambda b: affine_forward(x, w, b)[0]

_, 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:  5.965971776753866e-09
dw error:  2.7373542425098575e-10
db error:  8.990651411333356e-12
```

ReLU layer: forward

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

In [21]:
```
# 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
```

ReLU layer: backward

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

In [22]:
```
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, dou

_, 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:  3.2756056332714853e-12
```

Loss layers: Softmax and SVM

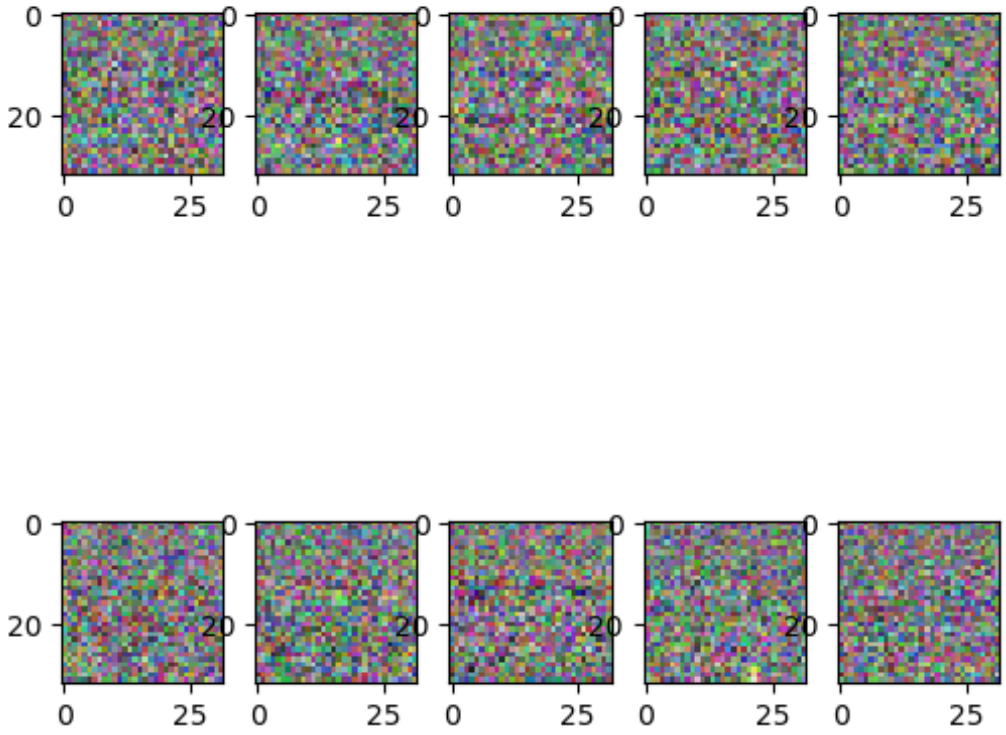
You implemented these loss functions in the last assignment, so we'll give them to you for free here. It's still a good idea to test them to make sure they work correctly.

In [23]:
```
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=Fa
loss, dx = svm_loss(x, y)

# Test svm_loss function. Loss should be around 9 and dx error should be 1e
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, verbos
loss, dx = softmax_loss(x, y)

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

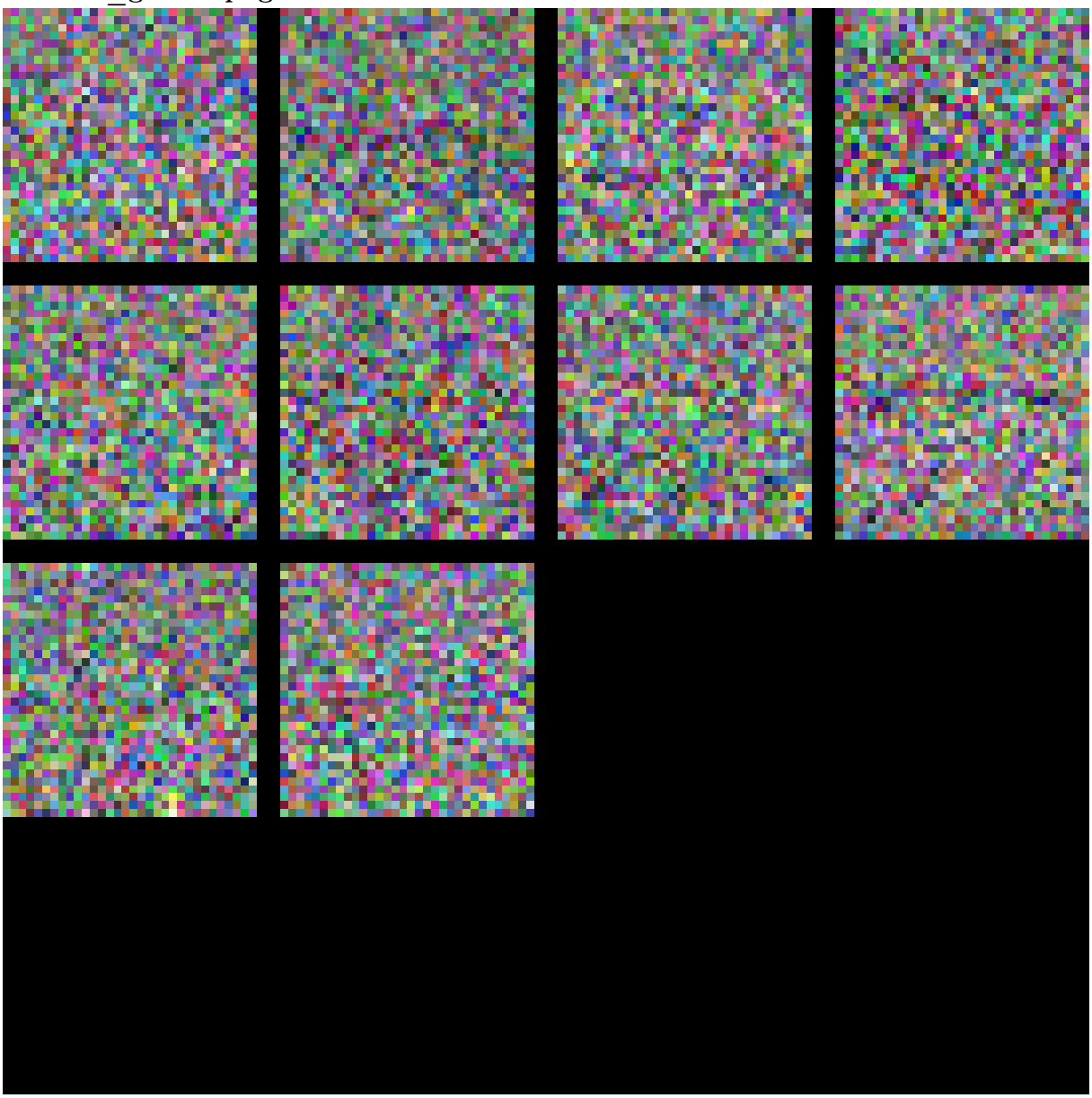
```
Testing svm_loss:
loss:  9.001184382766617
dx error:  1.4021566006651672e-09

Testing softmax_loss:
loss:  2.302704018730277
dx error:  9.499035653390327e-09
```

4.4 10.4 Softmax Classifier using PyTorch

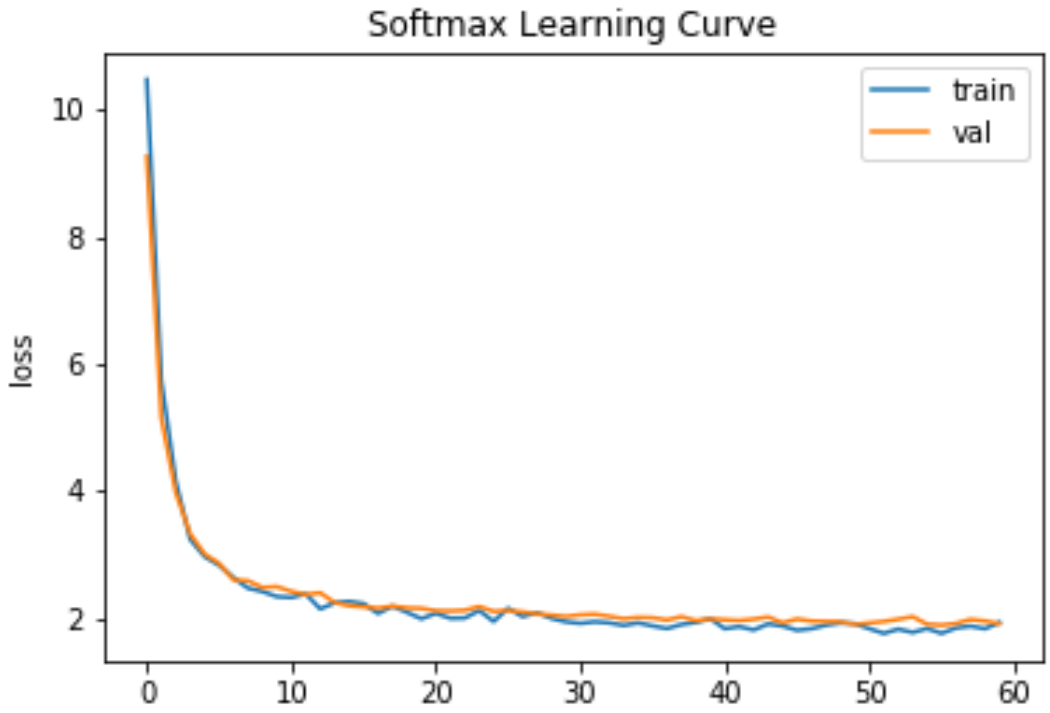
softmax_filt.png



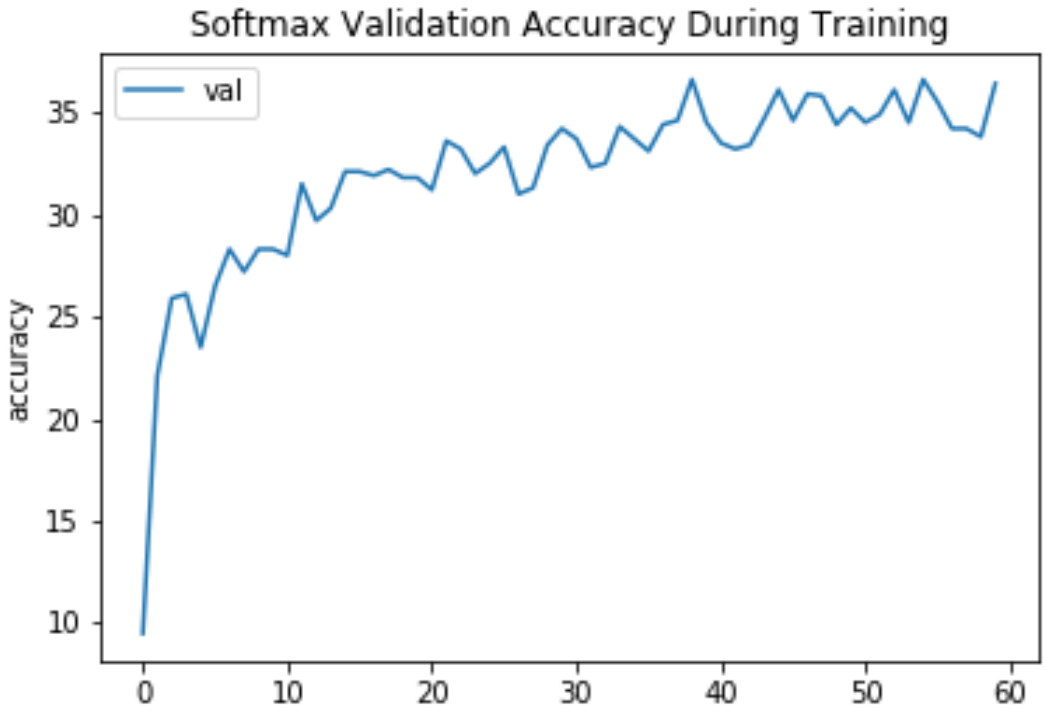
softmax_gridfilt.png



softmax_lossvstrain.png

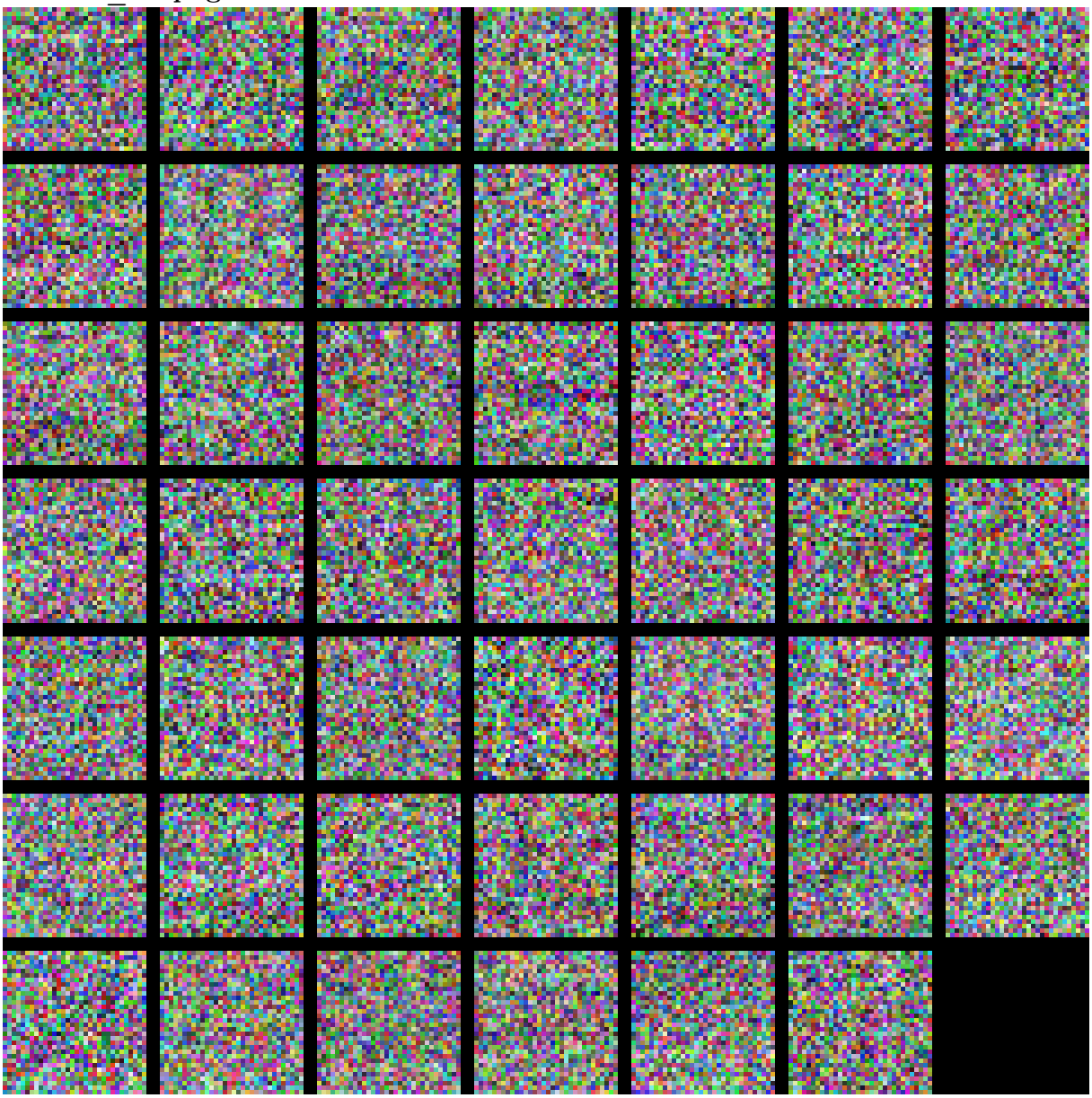


softmax_valaccuracy.png

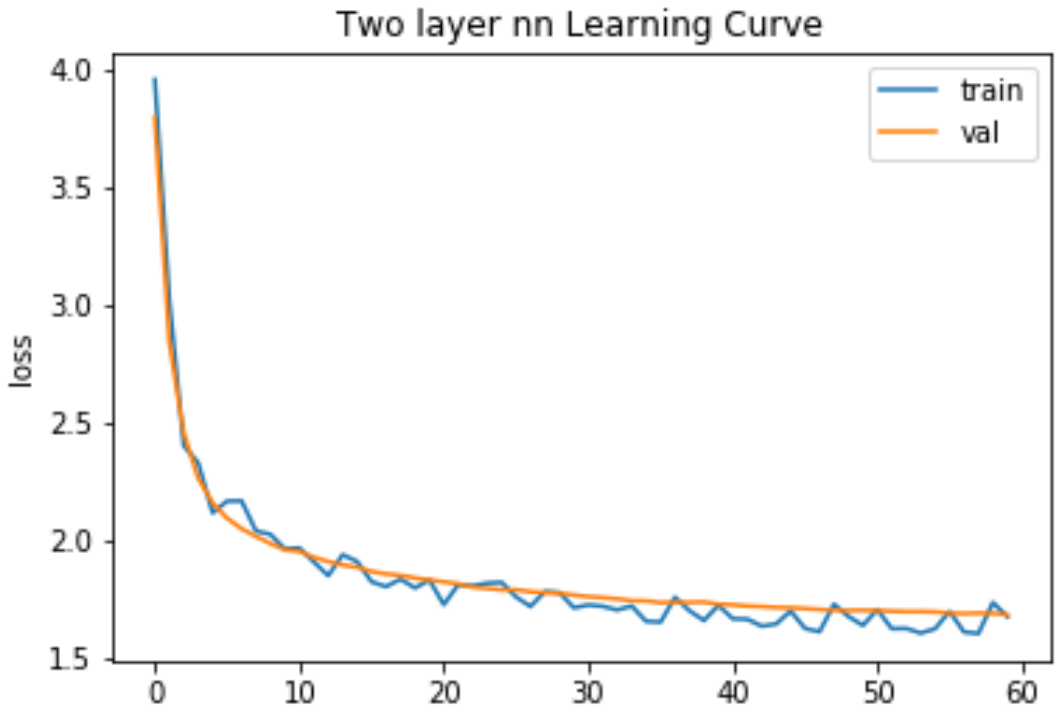


4.5 10.5 Two-layer Neural Network using PyTorch

softmax_filt.png



softmax_lossvstrain.png



softmax_valaccuracy.png