Lecture 6:
Training Neural Networks, Part I
Administrative

Assignment 1 due Thursday (today), 11:59pm on Canvas

Assignment 2 out today

Project proposal due Tuesday April 25

Notes on backprop for a linear layer and vector/tensor derivatives linked to Lecture 4 on syllabus
Where we are now...

Computational graphs

\[ f = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]
Neural Networks
Linear score function:
2-layer Neural Network

\[ f = W x \]
\[ f = W_2 \max(0, W_1 x) \]
Where we are now...

Convolutional Neural Networks
Where we are now...

Convolutional Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map
Where we are now...

**Convolutional Layer**

For example, if we had 6 5x5 filters, we’ll get 6 separate activation maps:

We stack these up to get a “new image” of size 28x28x6!
Where we are now...

**Learning network parameters through optimization**

```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
Where we are now...

**Mini-batch SGD**

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient
Next: Training Neural Networks
Overview

1. One time setup
   activation functions, preprocessing, weight initialization, regularization, gradient checking

2. Training dynamics
   babysitting the learning process, parameter updates, hyperparameter optimization

3. Evaluation
   model ensembles
Part 1

- Activation Functions
- Data Preprocessing
- Weight Initialization
- Batch Normalization
- Babysitting the Learning Process
- Hyperparameter Optimization
Activation Functions
Activation Functions

\[ f \left( \sum_{i} w_{i}x_{i} + b \right) \]

- **axon from a neuron**: \( x_{0} \)
- **synapse**: \( w_{0}x_{0} \)
- **dendrite**: \( w_{i}x_{i} \)
- **cell body**: \( \sum_{i} w_{i}x_{i} + b \)
- **output axon**: \( f \left( \sum_{i} w_{i}x_{i} + b \right) \)
- **activation function**: \( f \)
Activation Functions

**Sigmoid**
\[ \sigma(x) = \frac{1}{1+e^{-x}} \]

**tanh**
\[ \tanh(x) \]

**ReLU**
\[ \max(0, x) \]

**Leaky ReLU**
\[ \max(0.1x, x) \]

**Maxout**
\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

**ELU**
\[ \begin{cases} 
    x & \text{if } x \geq 0 \\
    \alpha(e^x - 1) & \text{if } x < 0
\end{cases} \]
Activation Functions

Sigmoid

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron
Activation Functions

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
What happens when $x = -10$?
What happens when $x = 0$?
What happens when $x = 10$?
Activation Functions

Sigmoid

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Squashes numbers to range \([0, 1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
Consider what happens when the input to a neuron \((x)\) is always positive:

What can we say about the gradients on \(w\)?
Consider what happens when the input to a neuron is always positive...

$$f \left( \sum_i w_i x_i + b \right)$$

What can we say about the gradients on $w$?
Always all positive or all negative :(
(this is also why you want zero-mean data!)
Activation Functions

**Sigmoid**

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Squashes numbers to range \([0,1]\)
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. \(\exp()\) is a bit compute expensive
Activation Functions

- Squashes numbers to range [-1, 1]
- Zero centered (nice)
- Still kills gradients when saturated :(

$tanh(x)$

[LeCun et al., 1991]
Activation Functions

- Computes $f(x) = \max(0, x)$
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]
Activation Functions

ReLU
(Rectified Linear Unit)

- Computes \( f(x) = \max(0, x) \)
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance:

hint: what is the gradient when \( x < 0 \)?
What happens when $x = -10$?
What happens when $x = 0$?
What happens when $x = 10$?
DATA CLOUD

active ReLU

dead ReLU
will never activate
=> never update
=> people like to initialize ReLU neurons with slightly positive biases (e.g. 0.01)

dead ReLU will never activate => never update
Activation Functions

Leaky ReLU

\[ f(x) = \max(0.01x, x) \]

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

[Mass et al., 2013]
[He et al., 2015]
Activation Functions

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- will not “die”.

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into $\alpha$ (parameter)

[Mass et al., 2013]
[He et al., 2015]
Activation Functions

Exponential Linear Units (ELU)

- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

\[ f(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  \alpha (\exp(x) - 1) & \text{if } x \leq 0 
\end{cases} \]

\[ [\text{Clevert et al., 2015}] \]
Maxout “Neuron”

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

\[
\max(w_1^T x + b_1, w_2^T x + b_2)
\]

Problem: doubles the number of parameters/neuron :(
TLDR: In practice:

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don’t expect much
- Don’t use sigmoid
Data Preprocessing
Step 1: Preprocess the data

(Assume X [NxD] is data matrix, each example in a row)
Remember: Consider what happens when the input to a neuron is always positive...

\[ f \left( \sum_i w_i x_i + b \right) \]

What can we say about the gradients on \( w \)?

Always all positive or all negative :( (this is also why you want zero-mean data!)
Step 1: Preprocess the data

(Assume $X$ [NxD] is data matrix, each example in a row)

```
X -= np.mean(X, axis = 0)  
X /= np.std(X, axis = 0)
```
Step 1: Preprocess the data

In practice, you may also see **PCA** and **Whitening** of the data.

![original data](image1)

![decorrelated data](image2)

![whitened data](image3)

- **original data**
- **decorrelated data** (data has diagonal covariance matrix)
- **whitened data** (covariance matrix is the identity matrix)
TLDR: In practice for Images: center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)

Not common to normalize variance, to do PCA or whitening
Weight Initialization
Q: what happens when $W=0$ init is used?
- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[
W = 0.01 \times \text{np.random.randn(D,H)}
\]
First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

\[
W = 0.01 \times \text{np.random.randn}(D,H)
\]

Works ~okay for small networks, but problems with deeper networks.
Let's look at some activation statistics.

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.
All activations become zero!

Q: think about the backward pass. What do the gradients look like?

Hint: think about backward pass for a W*X gate.
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

*1.0 instead of *0.01
"Xavier initialization"  
[Glorot et al., 2010]  

Reasonable initialization.  
(Mathematical derivation assumes linear activations)
but when using the ReLU nonlinearity it breaks.
\[ W = \text{np.random.randn}(\text{fan\_in}, \text{fan\_out}) / \text{np.sqrt}(\text{fan\_in}/2) \] # layer initialization

He et al., 2015 (note additional /2)
He et al., 2015
(note additional /2)
Proper initialization is an active area of research…

*Understanding the difficulty of training deep feedforward neural networks* by Glorot and Bengio, 2010

*Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* by Saxe et al, 2013

*Random walk initialization for training very deep feedforward networks* by Sussillo and Abbott, 2014

*Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* by He et al., 2015

*Data-dependent Initializations of Convolutional Neural Networks* by Krähenbühl et al., 2015

*All you need is a good init*, Mishkin and Matas, 2015
Batch Normalization
Batch Normalization

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer. To make each dimension unit gaussian, apply:

\[
\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}
\]

this is a vanilla differentiable function...
Batch Normalization

“you want unit gaussian activations? just make them so.”

1. compute the empirical mean and variance independently for each dimension.

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]
Batch Normalization

Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]
Batch Normalization

Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

Problem: do we necessarily want a unit gaussian input to a tanh layer?

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]  

[Ioffe and Szegedy, 2015]
Batch Normalization

Normalize:

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]

And then allow the network to squash the range if it wants to:

\[
y(k) = \gamma(k) \hat{x}(k) + \beta(k)
\]

Note, the network can learn:

\[
\gamma(k) = \sqrt{\text{Var}[x(k)]}
\]

\[
\beta(k) = E[x(k)]
\]

to recover the identity mapping.
Batch Normalization

Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_1...m\}$; Parameters to be learned: $\gamma, \beta$

Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

\[
\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad // \text{mini-batch mean}
\]

\[
\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2 \quad // \text{mini-batch variance}
\]

\[
\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{normalize}
\]

\[
y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad // \text{scale and shift}
\]

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe
Batch Normalization

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)
Babysitting the Learning Process
Step 1: Preprocess the data

(Assume $X [NxD]$ is data matrix, each example in a row)
Step 2: Choose the architecture: say we start with one hidden layer of 50 neurons:

- **CIFAR-10 images, 3072 numbers**
- **50 hidden neurons**
- **10 output neurons, one per class**

The diagram illustrates a neural network with:
- **Input layer**
- **Hidden layer**
- **Output layer**

The input layer is connected to the hidden layer, and the hidden layer is connected to the output layer.
Double check that the loss is reasonable:

```python
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}  
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```python
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train) # disable regularization
```

loss ~2.3. “correct” for 10 classes

returns the loss and the gradient for all parameters
Double check that the loss is reasonable:

```python
def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {}
    model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
    model['b1'] = np.zeros(hidden_size)
    model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
    model['b2'] = np.zeros(output_size)
    return model
```

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)  # crank up regularization
print loss
```

3.06859716482

loss went up, good. (sanity check)
Let's try to train now…

**Tip:** Make sure that you can overfit very small portion of the training data

The above code:
- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla ‘sgd’
Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!
Let's try to train now...

Start with small regularization and find learning rate that makes the loss go down.

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                   model, two_layer_net,
                                   num_epochs=10, reg=0.000001,
                                   update='sgd', learning_rate_decay=1,
                                   sample_batches=True,
                                   learning_rate=1e-6, verbose=True)
```
Let's try to train now...

Start with small regularization and find learning rate that makes the loss go down.

Loss barely changing
Let's try to train now…

Start with small regularization and find learning rate that makes the loss go down.

Loss not going down: learning rate too low
Let's try to train now...

Start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)
Lets try to train now...

Start with small regularization and find learning rate that makes the loss go down.

**loss not going down:**
learning rate too low

```python
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  )
```

Now let's try learning rate 1e6.
Let's try to train now…

Start with small regularization and find learning rate that makes the loss go down.

**Cost not going down:** learning rate too low

**Loss exploding:** learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()

best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                 model, TwoLayerNet,
                                 num_epochs=10, reg=0.000001,
                                 update='sgd', learning_rate_decay=1,
                                 sample_batches = True,
                                 learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero encountered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value encountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
```

Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+00
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+00
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+00

Cost: NaN almost always means high learning rate...
Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low
loss exploding: learning rate too high

3e-3 is still too high. Cost explodes….

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]
Hyperparameter Optimization
Cross-validation strategy

coarse -> fine cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
... (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early
For example: run coarse search for 5 epochs

```python
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)

    trainer = ClassifierTrainer()
    model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
    trainer = ClassifierTrainer()
    best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                            model, two_layer_net,
                                            num_epochs=5, reg=reg,
                                            update='momentum', learning_rate_decay=0.9,
                                            sample_batches = True, batch_size = 100,
                                            learning_rate=lr, verbose=False)

    val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
    val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
    val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
    val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
    val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
    val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
    val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
    val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
    val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
    val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
    val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

Note it's best to optimize in log space!
Now run finer search...

```
for count in xrange(max_count):
    reg = 10**uniform(-5, 5)
    lr = 10**uniform(-3, -6)
```

- adjust range

```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

<table>
<thead>
<tr>
<th>val_acc</th>
<th>lr</th>
<th>reg</th>
<th>(0 / 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.527000</td>
<td>5.340517e-04</td>
<td>4.097824e-01</td>
<td></td>
</tr>
<tr>
<td>0.492000</td>
<td>2.279484e-04</td>
<td>9.991348e-04</td>
<td></td>
</tr>
<tr>
<td>0.512000</td>
<td>8.688827e-04</td>
<td>1.349727e-02</td>
<td></td>
</tr>
<tr>
<td>0.461000</td>
<td>1.028377e-04</td>
<td>1.228193e-02</td>
<td></td>
</tr>
<tr>
<td>0.460000</td>
<td>1.11373e-04</td>
<td>5.244309e-02</td>
<td></td>
</tr>
<tr>
<td>0.490000</td>
<td>9.477776e-04</td>
<td>2.601293e-03</td>
<td></td>
</tr>
<tr>
<td>0.460000</td>
<td>1.484369e-04</td>
<td>4.328313e-01</td>
<td></td>
</tr>
<tr>
<td>0.522000</td>
<td>5.586261e-04</td>
<td>2.312685e-04</td>
<td></td>
</tr>
<tr>
<td>0.530000</td>
<td>5.86183e-04</td>
<td>8.259964e-02</td>
<td></td>
</tr>
<tr>
<td>0.489000</td>
<td>1.979168e-04</td>
<td>1.918889e-04</td>
<td></td>
</tr>
<tr>
<td>0.490000</td>
<td>2.036031e-04</td>
<td>2.466271e-03</td>
<td></td>
</tr>
<tr>
<td>0.475000</td>
<td>2.021162e-04</td>
<td>2.287807e-01</td>
<td></td>
</tr>
<tr>
<td>0.460000</td>
<td>1.135527e-04</td>
<td>3.905649e-02</td>
<td></td>
</tr>
<tr>
<td>0.515000</td>
<td>6.947668e-04</td>
<td>1.562808e-02</td>
<td></td>
</tr>
<tr>
<td>0.531000</td>
<td>9.471549e-04</td>
<td>1.433895e-03</td>
<td></td>
</tr>
<tr>
<td>0.509000</td>
<td>3.148888e-04</td>
<td>2.857518e-01</td>
<td></td>
</tr>
<tr>
<td>0.514000</td>
<td>6.438349e-04</td>
<td>3.033781e-01</td>
<td></td>
</tr>
<tr>
<td>0.502000</td>
<td>3.921764e-04</td>
<td>2.707126e-04</td>
<td></td>
</tr>
<tr>
<td>0.509000</td>
<td>9.752279e-04</td>
<td>2.850865e-03</td>
<td></td>
</tr>
<tr>
<td>0.500000</td>
<td>2.412048e-04</td>
<td>4.997821e-04</td>
<td></td>
</tr>
<tr>
<td>0.466000</td>
<td>1.319314e-04</td>
<td>1.189915e-02</td>
<td></td>
</tr>
<tr>
<td>0.516000</td>
<td>8.039527e-04</td>
<td>1.528291e-02</td>
<td></td>
</tr>
</tbody>
</table>

53% - relatively good for a 2-layer neural net with 50 hidden neurons.
Now run finer search...

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

But this best cross-validation result is worrying. Why?
Random Search vs. Grid Search

Grid Layout

Random Layout

Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

Random Search for Hyper-Parameter Optimization Bergstra and Bengio, 2012
Hyperparameters to play with:
- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

neural networks practitioner
music = loss function

This image by Paolo Guereta is licensed under CC-BY 2.0
Cross-validation
“command center”
Monitor and visualize the loss curve

![Loss curve graph with different learning rates](image)
Bad initialization
a prime suspect
Monitor and visualize the accuracy:

- big gap = overfitting
  => increase regularization strength?
- no gap
  => increase model capacity?
Track the ratio of weight updates / weight magnitudes:

```python
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW  # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update  # the actual update
print update_scale / param_scale  # want ~1e-3
```

ratio between the updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so
Summary

We looked in detail at:

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Weight Initialization (use Xavier init)
- Batch Normalization (use)
- Babysitting the Learning process
- Hyperparameter Optimization (random sample hyperparams, in log space when appropriate)
Next time:
Training Neural Networks, Part 2

- Parameter update schemes
- Learning rate schedules
- Gradient checking
- Regularization (Dropout etc.)
- Evaluation (Ensembles etc.)
- Transfer learning / fine-tuning