Lecture 5: Training Neural Networks, Part I

Thursday February 2, 2017
Announcements!

- HW1 due today!
- Because of website typo, will accept homework 1 until Saturday with no late penalty.
- HW2 comes out tomorrow. It is very large.
Python/Numpy of the Day

- `numpy.random.uniform(low, high)`

```python
while solver.best_val_acc < 0.50:
    weight_scale = np.random.uniform(1e-5, 1e-1)
    learning_rate = np.random.uniform(1e-8, 1e-1)
    model = FullyConnectedNet([100, 100],
                              weight_scale=weight_scale, dtype=np.float64)
    solver = Solver(model, data,
                    num_epochs=<small number>...)
    solver.train()
print 'Best val_acc = {} : lr was {} ws was {}' .format(solver.best_val_acc
                                                      learning_rate, weight_scale)
```
The effects of step size (or “learning rate”)

Regularization effect can be observed this way also.

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Things you should know for your Project Proposal

“ConvNets need a lot of data to train”
Things you should know for your Project Proposal

“ConvNets need a lot of data to train”

finetuning! we rarely ever train ConvNets from scratch.
1. Train on ImageNet

2. Finetune network on your own data

ImageNet data

your data

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Transfer Learning with CNNs

1. Train on ImageNet

2. If small dataset: fix all weights (treat CNN as fixed feature extractor), retrain only the classifier
   i.e. swap the Softmax layer at the end

3. If you have medium sized dataset, “finetune” instead: use the old weights as initialization, train the full network or only some of the higher layers
   retrain bigger portion of the network, or even all of it.
E.g. Caffe Model Zoo: Lots of pretrained ConvNets
https://github.com/BVLC/caffe/wiki/Model-Zoo
Things you should know for your Project Proposal

“We have infinite compute available on AWS GPU machines.”
Things you should know for your Project Proposal

“We have infinite compute available on AWS GPU machines.”

You have finite compute. Don’t be overly ambitious.

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Where we are now...

Mini-batch SGD

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph, get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient
Where we are now...
Convolutional Network (AlexNet)

input image
weights
loss
activations

\[ \frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x} \]

"local gradient"

\[ \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial y} \]

gradients
Implementation: forward/backward API

Graph (or Net) object. (Rough pseudo code)

```python
class ComputationalGraph(object):
    # ...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss  # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```
Implementation: forward/backward API

```
class MultiplyGate(object):
    def forward(x, y):
        z = x * y
        self.x = x  # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz  # [dz/dx * dL/dz]
        dy = self.x * dz  # [dz/dy * dL/dz]
        return [dx, dy]
```

(x, y, z are scalars)
Example: Torch Layers

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Neural Network: without the brain stuff

(Before) Linear score function:

\[ f = Wx \]

(Now) 2-layer Neural Network or 3-layer Neural Network

\[ f = W_2 \max(0, W_1 x) \]

\[ f = W_3 \max(0, W_2 \max(0, W_1 x)) \]
class Neuron:
    # ...
    def neuron_tick(inputs):
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum)) # sigmoid activation function
        return firing_rate

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Neural Networks: Architectures

“2-layer Neural Net”, or “1-hidden-layer Neural Net”

“Fully-connected” layers

“3-layer Neural Net”, or “2-hidden-layer Neural Net”
Training Neural Networks

A bit of history...
A bit of history

The **Mark I Perceptron** machine was the first implementation of the perceptron algorithm.

The machine was connected to a camera that used $20 \times 20$ cadmium sulfide photocells to produce a 400-pixel image.

recognized letters of the alphabet

update rule:

$$w_i(t+1) = w_i(t) + \alpha (d_j - y_j(t))x_{j,i}$$

*Frank Rosenblatt, ~1957: Perceptron*
A bit of history

Widrow and Hoff, ~1960: Adaline/Madaline
A bit of history

Rumelhart et al. 1986: First time back-propagation became popular
Yann and his friends: CNNs in 1993

https://youtu.be/FwFduRA_L6Q
A bit of history

[Hinton and Salakhutdinov 2006]

Reinvigorated research in Deep Learning

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
“Autoencoders are useful for some things, but turned out not to be nearly as necessary as we once thought. Around 10 years ago, we thought that deep nets would not learn correctly if trained with only backprop of the supervised cost. We thought that deep nets would also need an unsupervised cost, like the autoencoder cost, to regularize them. When Google Brain built their first very large neural network to recognize objects in images, it was an autoencoder (and it didn’t work very well at recognizing objects compared to later approaches). Today, we know we are able to recognize images just by using backprop on the supervised cost as long as there is enough labeled data. There are other tasks where we do still use autoencoders, but they’re not the fundamental solution to training deep nets that people once thought they were going to be.”
First strong results

**Context-Dependent Pre-trained Deep Neural Networks for Large Vocabulary Speech Recognition**
George Dahl, Dong Yu, Li Deng, Alex Acero, 2010

**Imagenet classification with deep convolutional neural networks**

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Overview

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

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

1. Evaluation
   model ensembles
Activation Functions
Activation Functions
Activation Functions

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

\[ \tanh \quad \tanh(x) \]

ReLU \quad \text{max}(0, x)

Leaky ReLU \quad \text{max}(0.1x, x)

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

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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Activation Functions

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

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

Sigmoid

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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

---

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
What happens when \( x = -10? \)
What happens when \( x = 0? \)
What happens when \( x = 10? \)
Sigmoid

Activation Functions

- 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

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

- 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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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)

ReLU
(Rectified Linear Unit)

[Krizhevsky et al., 2012]
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)
- Not zero-centered output
- An annoyance:

hint: what is the gradient when \( x < 0 \)?

ReLU
(Rectified Linear Unit)

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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

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

Leaky ReLU

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

[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]

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Activation Functions

Exponential Linear Units (ELU)

- All benefits of ReLU
- Does not die
- Closer to zero mean outputs
- Computation requires \( \exp() \)

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

[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)

```python
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.
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 can lead to non-homogeneous distributions of activations across the layers of a network.
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.
Input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000082 and std 0.000332
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

comp150dl

Tufts University

61
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.

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

*1.0 instead of *0.01
Reasonable initialization. (Mathematical derivation assumes linear activations)

* Xavier initialization * [Glorot et al., 2010]
but when using the ReLU nonlinearity it breaks.

\[
W = \text{np.random.randn(fan\ in, fan\ out)} / \text{np.sqrt(fan\ in)} \quad \#\ \text{layer\ initialization}
\]
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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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.

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

[loffe and Szegedy, 2015]
Batch Normalization

Usually inserted after Fully Connected / (or Convolutional, as we’ll see soon) layers, and before nonlinearity.

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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Batch Normalization

- 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

\[ \mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad // \text{mini-batch mean} \]
\[ \sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad // \text{mini-batch variance} \]
\[ \hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad // \text{normalize} \]
\[ y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad // \text{scale and shift} \]
Batch Normalization

Input: Values of $x$ over a mini-batch: $B = \{x_1...m\}$; Parameters to be learned: $\gamma, \beta$  
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$
\mu_B \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i \quad \text{// mini-batch mean}
$$

$$
\sigma_B^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 \quad \text{// mini-batch variance}
$$

$$
\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{// normalize}
$$

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

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)

[ioffe and Szegedy, 2015]
Babysitting the Learning Process
Step 1: Preprocess the data

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

- Input layer
- Hidden layer with 50 neurons
- Output layer with 10 neurons, one per class

CIFAR-10 images, 3072 numbers
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, 0.0)  # disable regularization
print loss
2.30261216167
```

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
```

```
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)
print loss
```

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'
Let's 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!

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
I like to 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...

I like to 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
Lets try to train now…

I like to 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)
Let's try to train now…

I like to start with small regularization and find learning rate that makes the loss go down.

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

Okay now let's try learning rate 1e6. What could possibly go wrong?
Let's try to train now…

I like to 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

Cost: NaN almost always means high learning rate…
Let's try to train now...

I like to 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 $[1e-3 \ldots 1e-5]$
Hyperparameter Optimization
Cross-validation strategy

Try **coarse to 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

```
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)
```

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

note it’s best to optimize in log space!
Now run finer search...

adjust range

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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
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

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

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
- lots of connections to make
- lots of knobs to turn
- want to get the best test performance
Andrej Karpathy’s cross-validation “command center”
Monitor and visualize the loss curve

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n
Loss

\[ \text{time} \]
Bad initialization
a prime suspect

Loss

Time
Loss function specimen

validation loss  LR step function  “This RNN smoothly forgets everything it has learned.”

* Original slides borrowed from Andrej Karpathy and Li Fei-Fei, Stanford cs231n

comp150dl
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 values and updates: \( \sim 0.0002 / 0.02 = 0.01 \) (about okay)
want this to be somewhere around 0.001 or so
Parameters $p$: no learning

Parameters $p'$: lots of learning

How Weights Change

Parameters $p$: learning rate or regularization are too big and some weights are exploding.
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
Look at:

- Parameter update schemes
- Learning rate schedules
- Gradient Checking
- Regularization (Dropout etc)
- Evaluation (Ensembles etc)