Lecture 5:

Training Neural Networks, Part I
Administrative

A1 is due **today** (midnight)
I’m holding make up office hours on today: 5pm @ Gates 259

A2 will be released ~tomorrow. It’s meaty, but educational!

Also:
- We are shuffling the course schedule around a bit
- the grading scheme is subject to few % changes
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”

\textbf{MYTH!} we rarely ever train ConvNets from scratch.
1. Train on ImageNet

2. Fine-tune network on your own data
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 because Terminal.”
Things you should know for your Project Proposal

“We have infinite compute available because Terminal.”

You have finite compute. Don’t be overly ambitious.
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...
Neural Turing Machine

input tape

loss
activations

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

gradients

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

“local gradient”
Implementation: forward/backward API

Graph (or Net) object. *(Rough psuedo 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(self, x, y):
        z = x * y
        self.x = x  # must keep these around!
        self.y = y
        return z
    def backward(self, 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
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
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 20×20 cadmium sulfide photocells to produce a 400-pixel image.

\[
    f(x) = \begin{cases} 
    1 & \text{if } w \cdot x + b > 0 \\
    0 & \text{otherwise} 
    \end{cases}
\]

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
A bit of history

[Hinton and Salakhutdinov 2006]

Reinvigorated research in Deep Learning
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**  
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*
Activation Functions
Activation Functions

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

- \( x_0 \) : axon from a neuron
- \( w_0 \) : synapse
- \( w_0 x_0 \) : dendrite
- \( w_1 x_1 \)
- \( w_2 x_2 \)
- \( \sum_i w_i x_i + b \)
- Output axon
- Activation function
### Activation Functions

**Sigmoid**

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

![Sigmoid Graph](image1)

**tanh**

\[ \tanh(x) \]

![tanh Graph](image2)

**ReLU**

\[ \max(0, x) \]

![ReLU Graph](image3)

**Leaky ReLU**

\[ \max(0.1x, x) \]

![Leaky ReLU Graph](image4)

**Maxout**

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

![Maxout Graph](image5)

**ELU**

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

![ELU Graph](image6)
Activation Functions

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

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]
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
What happens when $x = -10$?
What happens when $x = 0$?
What happens when $x = 10$?

The sigmoid function is given by:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
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
Consider what happens when the input to a neuron (x) is always positive:

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

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 :(

\[
\text{tanh}(x)
\]

[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:

ReLU
(Rectified Linear Unit)

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

active ReLU
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]
Activation Functions

**Exponential Linear Units (ELU)**

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

\[
f(x) = \begin{cases} 
  x & \text{if } x > 0 \\
  \alpha (\exp(x) - 1) & \text{if } x \leq 0 
\end{cases}
\]
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 :( [Goodfellow et al., 2013]
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)
Step 1: Preprocess the data

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

- **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 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.
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.
\[
W = \text{np.random.randn}(\text{fan\_in}, \text{fan\_out}) / \text{np.sqrt}(\text{fan\_in}/2) \quad \# \text{layer initialization}
\]

He et al., 2015
(note additional /2)
He et al., 2015
(note additional /2)

\[ W = \text{np.random.randn}(\text{fan\_in}, \text{fan\_out}) / \text{np.sqrt}(\text{fan\_in}/2) \]  # layer initialization
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

"you want unit gaussian activations? just make them so."

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

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

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.

2. Normalize

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

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

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

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]
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:** $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$
\begin{align*}
\mu_B & \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i & \text{// mini-batch mean} \\
\sigma_B^2 & \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2 & \text{// mini-batch variance} \\
\hat{x}_i & \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} & \text{// normalize} \\
y_i & \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) & \text{// scale and shift}
\end{align*}
$$

- 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

[ioffe and Szegedy, 2015]
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 $ \hspace{1cm} // mini-batch mean

$\sigma^2_\mathcal{B} \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 $ \hspace{1cm} // mini-batch variance

$\hat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma^2_\mathcal{B} + \epsilon}} $ \hspace{1cm} // normalize

$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) $ \hspace{1cm} // 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 a data matrix, each example in a row)

$$X \leftarrow \text{np.mean}(X, \text{axis} = 0)$$  $$X \leftarrow \text{np.std}(X, \text{axis} = 0)$$
Step 2: Choose the architecture: 
say we start with one hidden layer of 50 neurons:

CIFAR-10 images, 3072 numbers

input layer

hidden layer

output layer

50 hidden neurons

10 output neurons, one per class
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, 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
```

```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)  # crank up regularization
print loss
```

```
3.06859716482
```

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

I like to start with small regularization and find learning rate that makes the loss go down.
Let's try to train now...

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

Loss barely changing
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
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)
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

Okay now lets 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 ... 1e-5]
Hyperparameter Optimization
Cross-validation strategy

I like to do coarse -&gt; 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 &gt; 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)
```
Now run finer search...

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

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?

<table>
<thead>
<tr>
<th>max_count</th>
<th>for count in xrange(max_count):</th>
<th>val acc: 0.527600, lr: 5.340517e-04, reg: 4.697824e-01, (0 / 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>val acc: 0.492600, lr: 2.279484e-04, reg: 9.391345e-04, (1 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.512800, lr: 8.688827e-04, reg: 1.349727e-02, (2 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.461600, lr: 1.028377e-04, reg: 1.228193e-02, (3 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.490000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.460000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.522800, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.530000, lr: 5.808183e-04, reg: 8.259644e-02, (8 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.480000, lr: 1.979168e-04, reg: 1.018889e-04, (9 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.490600, lr: 2.036031e-04, reg: 2.466271e-03, (10 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.475600, lr: 2.021162e-04, reg: 2.287007e-01, (11 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.460000, lr: 1.135527e-04, reg: 3.905640e-02, (12 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.516000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.531600, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.509600, lr: 3.148888e-04, reg: 2.857518e-01, (15 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.514600, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.502000, lr: 3.921764e-04, reg: 2.707126e-04, (17 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.509600, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.460000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>val acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)</td>
</tr>
</tbody>
</table>
Random Search vs. Grid Search

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
My cross-validation
“command center”
Monitor and visualize the loss curve
Loss

Bad initialization
a prime suspect
Loss function specimen

![Graphs showing loss functions over epochs](lossfunctions.tumblr.com)
Monitor and visualize the accuracy:

![Graph showing training and validation accuracy over epochs.]

- **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
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)

TLDRs
TODO
Look at:

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