I didn't have any accurate numbers so I just made up this one.

Studies have shown that accurate numbers aren't any more useful than the ones you make up.

How many studies showed that? Eighty-seven.

http://www.cs.vu.nl/~frankh/dilbert.html
How do you actually train these things?

[Szegedy et al, 2014]
How do you actually train these things?

[Szegedy et al, 2014]

Network in network

We need to go deeper
How do you actually train these things?

... why so many layers?

[ Szegedy et al, 2014 ]

Network in network

We need to go deeper
How do you actually train these things?
How do you actually train these things?

Roughly speaking:
How do you actually train these things?

Roughly speaking:

Gather labeled data
How do you actually train these things?

Roughly speaking:

- Gather labeled data
- Find a CNN architecture
How do you actually train these things?

Roughly speaking:

Gather labeled data  Find a CNN architecture  Minimize the loss
Training a convolutional neural network
Training a convolutional neural network

• Split and preprocess your data
Training a convolutional neural network

- Split and preprocess your data
- Choose your network architecture
Training a convolutional neural network

• Split and preprocess your data
• Choose your network architecture
• Initialize the weights
Training a convolutional neural network

- Split and preprocess your data
- Choose your network architecture
- Initialize the weights
- Find a learning rate and regularization strength
Training a convolutional neural network

- Split and preprocess your data
- Choose your network architecture
- Initialize the weights
- Find a learning rate and regularization strength
- Minimize the loss and monitor progress
Training a convolutional neural network

- Split and preprocess your data
- Choose your network architecture
- Initialize the weights
- Find a learning rate and regularization strength
- Minimize the loss and monitor progress
- Fiddle with knobs
Mini-batch Gradient Descent
Mini-batch Gradient Descent

Loop:
Mini-batch Gradient Descent

Loop:

1. Sample a batch of training data (~100 images)
Mini-batch Gradient Descent

Loop:

1. Sample a batch of training data (~100 images)
2. Forwards pass: compute loss (avg. over batch)
Mini-batch Gradient Descent

Loop:

1. Sample a batch of training data (~100 images)
2. Forwards pass: compute loss (avg. over batch)
3. Backwards pass: compute gradient
Mini-batch Gradient Descent

Loop:

1. Sample a batch of training data (~100 images)
2. Forwards pass: compute loss (avg. over batch)
3. Backwards pass: compute gradient
4. Update all parameters
Mini-batch Gradient Descent

Loop:

1. Sample a batch of training data (~100 images)
2. Forwards pass: compute loss (avg. over batch)
3. Backwards pass: compute gradient
4. Update all parameters

Note: usually called “stochastic gradient descent” even though SGD has a batch size of 1
Regularization
Regularization reduces overfitting:
Regularization

Regularization reduces overfitting:

\[ L = L_{\text{data}} + L_{\text{reg}} \]

\[ L_{\text{reg}} = \lambda \frac{1}{2} \| W \|_2^2 \]
Regularization

Regularization reduces overfitting:

\[ L = L_{\text{data}} + L_{\text{reg}} \]

\[ L_{\text{reg}} = \lambda \frac{1}{2} \| W \|_2^2 \]
(0) Dataset split

Split your data into “train”, “validation”, and “test”:

Dataset
(0) Dataset split

Split your data into “train”, “validation”, and “test”:
(0) Dataset split

Train

Validation

Test
(0) Dataset split

Train:

Train: gradient descent and fine-tuning of parameters
(0) Dataset split

Train: gradient descent and fine-tuning of parameters

Validation: determining hyper-parameters (learning rate, regularization strength, etc) and picking an architecture
(0) Dataset split

**Train**: gradient descent and fine-tuning of parameters

**Validation**: determining hyper-parameters (learning rate, regularization strength, etc) and picking an architecture

**Test**: estimate real-world performance (e.g. accuracy = fraction correctly classified)
(0) Dataset split

Be careful with false discovery:
Be careful with false discovery:

To avoid false discovery, once we have used a test set once, we should not use it again (but nobody follows this rule, since it’s expensive to collect datasets)
(0) Dataset split

Be careful with false discovery:

To avoid false discovery, once we have used a test set once, we should *not use it again* (but nobody follows this rule, since it’s expensive to collect datasets)

Instead, try and avoid looking at the test score until the end
(0) Dataset split

**Cross-validation:** cycle which data is used as validation
(0) Dataset split

Cross-validation: cycle which data is used as validation
(0) Dataset split

**Cross-validation:** cycle which data is used as validation
(0) Dataset split

**Cross-validation:** cycle which data is used as validation
(0) Dataset split

**Cross-validation:** cycle which data is used as validation

Average scores across validation splits
(1) Data preprocessing

Preprocess the data so that learning is better conditioned:

Figure: Andrej Karpathy
(1) Data preprocessing

Preprocess the data so that learning is better conditioned:

```
X -= np.mean(axis=0, keepdims=True)
```
(1) Data preprocessing

Preprocess the data so that learning is better conditioned:

- Original data
- Zero-centered data
- Normalized data

```python
X -= np.mean(axis=0, keepdims=True)
X /= np.std(axis=0, keepdims=True)
```

*Figure: Andrej Karpathy*
(1) Data preprocessing

In practice, you may also see **PCA** and **Whitening** of the data:
In practice, you may also see **PCA** and **Whitening** of the data:
(1) Data preprocessing

In practice, you may also see **PCA** and **Whitening** of the data:
(1) Data preprocessing

For CNNs, typically only the mean is subtracted.

Figure: Alex Krizhevsky
(1) **Data preprocessing**

For CNNs, typically only the mean is subtracted.

*Figure: Alex Krizhevsky*
(1) Data preprocessing

For CNNs, typically only the mean is subtracted.

A per-channel mean also works (one value per R,G,B).

*Figure: Alex Krizhevsky*
(1) Data preprocessing

**Augment the data** — extract random crops from the input, with slightly jittered offsets. Without this, typical CNNs (e.g. [Krizhevsky 2012]) overfit the data.

*Figure: Alex Krizhevsky*
(1) Data preprocessing

Augment the data — extract random crops from the input, with slightly jittered offsets. Without this, typical CNNs (e.g. [Krizhevsky 2012]) overfit the data.

E.g. 224x224 patches extracted from 256x256 images

Figure: Alex Krizhevsky
(1) Data preprocessing

Augment the data — extract random crops from the input, with slightly jittered offsets. Without this, typical CNNs (e.g. [Krizhevsky 2012]) overfit the data.

E.g. 224x224 patches extracted from 256x256 images

Randomly reflect horizontally

Figure: Alex Krizhevsky
(1) Data preprocessing

**Augment the data** — extract random crops from the input, with slightly jittered offsets. Without this, typical CNNs (e.g. [Krizhevsky 2012]) overfit the data.

**E.g.** 224x224 patches extracted from 256x256 images

Randomly reflect horizontally

Perform the augmentation live during training

*Figure: Alex Krizhevsky*
(2) Choose your architecture

Toy example: one hidden layer of size 50
(2) Choose your architecture

Toy example: one hidden layer of size 50

CIFAR-10 images, 3072 numbers
(3) Initialize your weights

Set the weights to small random numbers:
(3) Initialize your weights

Set the weights to small random numbers:

\[ W = \text{np.random.randn}(D, H) \times 0.001 \]

(matrix of small random numbers drawn from a Gaussian distribution)
(3) Initialize your weights

Set the weights to small random numbers:

```python
W = np.random.randn(D, H) * 0.001
```

(matrix of small random numbers drawn from a Gaussian distribution)

(the magnitude is important — more on this later)
(3) Initialize your weights

Set the weights to small random numbers:

\[ W = \text{np.random.randn(D, H)} \times 0.001 \]

(matrix of small random numbers drawn from a Gaussian distribution)

(the magnitude is important — more on this later)

Set the bias to zero (or small nonzero):

\[ b = \text{np.zeros(H)} \]
(3) Check that the loss is reasonable

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
(3) 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
```

returns the loss and the gradient for all parameters
(3) 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)
```

returns the loss and the gradient for all parameters

Slide: Andrej Karpathy
(3) Check that the loss is reasonable

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

Slide: Andrej Karpathy
(3) Check that the loss is reasonable

def init_two_layer_model(input_size, hidden_size, output_size):
    # initialize a model
    model = {
        'W1': 0.0001 * np.random.randn(input_size, hidden_size),
        'b1': np.zeros(hidden_size),
        'W2': 0.0001 * np.random.randn(hidden_size, output_size),
        '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)  # crank up regularization
print loss
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
```
(3) Check that the loss is reasonable

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)

3.06859716482  
loss went up, good. (sanity check)

Slide: Andrej Karpathy
(4) Overfit a small portion of the data

```python
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
    model, two_layer_net,
    num_epochs=200, reg=0.0,
    update='sgd', learning_rate_decay=1,
    sample_batches = False,
    learning_rate=1e-3, verbose=True)
```
(4) Overfit a small portion of the data

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20]  # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```
(4) Overfit a small portion of the data

Details:

```python
model = init_two_layer_model(32*32*3, 50, 10)  # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20]  # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```
(4) Overfit a small portion of the data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                           model, two_layer_net,
                                           num_epochs=200, reg=0.0,
                                           update='sgd', learning_rate_decay=1,
                                           sample_batches = False,
                                           learning_rate=1e-3, verbose=True)
```

**Details:**

‘s SGD’: vanilla gradient descent (no momentum etc)
(4) Overfit a small portion of the data

Details:
‘sgd’: vanilla gradient descent (no momentum etc)
learning_rate_decay = 1: constant learning rate
(4) Overfit a small portion of the data

```python
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
model, two_layer_net,
num_epochs=200, reg=0.0,
update='sgd', learning_rate_decay=1,
sample_batches = False,
learning_rate=1e-3, verbose=True)
```

Details:

‘sgd’: vanilla gradient descent (no momentum etc)

learning_rate_decay = 1: constant learning rate

sample_batches = False (full gradient descent, no batches)
(4) Overfit a small portion of the data

Details:

‘sgd’: vanilla gradient descent (no momentum etc)
learning_rate_decay = 1: constant learning rate
sample_batches = False (full gradient descent, no batches)
ePOCHS = 200: number of passes through the data
(4) Overfit a small portion of the data
(4) Overfit a small portion of the data
(4) Overfit a small portion of the data

100% accuracy on the training set (good)
(4) Find a learning rate

Let’s start with small regularization and find the learning rate that makes the loss decrease:

```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)
```
(4) Find a learning rate

Let’s start with small regularization and find the learning rate that makes the loss decrease:

```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)
```
(4) Find a learning rate

```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)
```
(4) Find a learning rate
(4) Find a learning rate

Loss barely changes
(learning rate is too low or regularization too high)
(4) Find a learning rate

---

**Loss barely changes**

Why is the accuracy 20%?

(learning rate is too low or regularization too high)

---

*Slide: Andrej Karpathy*
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

```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=1e6, verbose=True)
```
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

```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=1e6, verbose=True)

/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero encountered in log
```
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

```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=1e6, verbose=True)
```

Loss is NaN $\rightarrow$ learning rate is too high
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

Loss $L$ vs. A weight somewhere in the network
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

A weight somewhere in the network
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

A weight somewhere in the network
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

A weight somewhere in the network
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

A weight somewhere in the network
(4) Find a learning rate

Learning rate: $1e6$ — what could go wrong?

A weight somewhere in the network
(4) Find a learning rate

Learning rate: 3e-3
(4) Find a learning rate

Learning rate: 3e-3
(4) Find a learning rate

Learning rate: 3e-3

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=3e-3, verbose=True)

Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03

Loss is inf —> still too high
(4) Find a learning rate

Learning rate: 3e-3

Loss is inf —> still too high
But now we know we should be searching the range [1e-3 ... 1e-5]

Slide: Andrej Karpathy
(4) Find a learning rate

Coarse to fine search
(4) Find a learning rate

Coarse to fine search

First stage: only a few epochs (passes through the data) to get a rough idea
(4) Find a learning rate

Coarse to fine search

First stage: only a few epochs (passes through the data) to get a rough idea

Second stage: longer running time, finer search
(4) Find a learning rate

Coarse to fine search

First stage: only a few epochs (passes through the data) to get a rough idea

Second stage: longer running time, finer search

Tip: if loss > 3 * original loss, quit early (learning rate too high)
Find a learning rate

Coarse to fine search

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

note it’s best to optimize in log space
(4) Find a learning rate

Coarse to fine search

```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)
```

*note it’s best to optimize in log space*
(4) Find a learning rate

Coarse to fine search

```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

Slide: Andrej Karpathy
(4) Find a learning rate

Coarse to fine search

```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)

Slide: Andrej Karpathy
(4) Find a learning rate

Coarse to fine search

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

Coarse to fine search

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

adjust range
(4) Find a learning rate

Coarse to fine search

```
max_count = 100
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)
```
(4) Find a learning rate

Coarse to fine search

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

<table>
<thead>
<tr>
<th>count</th>
<th>val_acc</th>
<th>lr</th>
<th>reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.527000</td>
<td>5.340517e-04</td>
<td>4.097824e-01</td>
</tr>
<tr>
<td>1</td>
<td>0.492000</td>
<td>2.279484e-04</td>
<td>9.991345e-04</td>
</tr>
<tr>
<td>2</td>
<td>0.512000</td>
<td>8.680827e-04</td>
<td>1.349727e-02</td>
</tr>
<tr>
<td>3</td>
<td>0.461000</td>
<td>1.028377e-04</td>
<td>1.220193e-02</td>
</tr>
<tr>
<td>4</td>
<td>0.460000</td>
<td>1.113730e-04</td>
<td>5.244309e-02</td>
</tr>
<tr>
<td>5</td>
<td>0.498000</td>
<td>9.477776e-04</td>
<td>2.001293e-03</td>
</tr>
<tr>
<td>6</td>
<td>0.469000</td>
<td>1.484369e-04</td>
<td>4.328313e-01</td>
</tr>
<tr>
<td>7</td>
<td>0.522000</td>
<td>5.586261e-04</td>
<td>2.312685e-04</td>
</tr>
<tr>
<td>8</td>
<td>0.530000</td>
<td>5.808183e-04</td>
<td>8.259964e-02</td>
</tr>
<tr>
<td>9</td>
<td>0.489000</td>
<td>1.979168e-04</td>
<td>1.010889e-04</td>
</tr>
<tr>
<td>10</td>
<td>0.490000</td>
<td>2.036031e-04</td>
<td>2.406271e-03</td>
</tr>
<tr>
<td>11</td>
<td>0.475000</td>
<td>2.021162e-04</td>
<td>2.287807e-01</td>
</tr>
<tr>
<td>12</td>
<td>0.460000</td>
<td>1.135527e-04</td>
<td>3.905040e-02</td>
</tr>
<tr>
<td>13</td>
<td>0.515000</td>
<td>6.947668e-04</td>
<td>1.562808e-02</td>
</tr>
<tr>
<td>14</td>
<td>0.531000</td>
<td>9.471549e-04</td>
<td>1.433895e-03</td>
</tr>
<tr>
<td>15</td>
<td>0.509000</td>
<td>3.140888e-04</td>
<td>2.857518e-01</td>
</tr>
<tr>
<td>16</td>
<td>0.514000</td>
<td>6.438349e-04</td>
<td>3.033781e-01</td>
</tr>
<tr>
<td>17</td>
<td>0.502000</td>
<td>3.921784e-04</td>
<td>2.707126e-04</td>
</tr>
<tr>
<td>18</td>
<td>0.509000</td>
<td>9.752279e-04</td>
<td>2.850865e-03</td>
</tr>
<tr>
<td>19</td>
<td>0.500000</td>
<td>2.412048e-04</td>
<td>4.997821e-04</td>
</tr>
<tr>
<td>20</td>
<td>0.466000</td>
<td>1.319314e-04</td>
<td>1.189915e-02</td>
</tr>
<tr>
<td>21</td>
<td>0.516000</td>
<td>8.039527e-04</td>
<td>1.528291e-02</td>
</tr>
</tbody>
</table>
```

Slide: Andrej Karpathy
(4) Find a learning rate

Coarse to fine search

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

<table>
<thead>
<tr>
<th>count</th>
<th>val_acc</th>
<th>lr</th>
<th>reg</th>
<th>max_count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.527000</td>
<td>5.340517e-04</td>
<td>4.097824e-01</td>
<td>100</td>
</tr>
<tr>
<td>1</td>
<td>0.492000</td>
<td>2.279484e-04</td>
<td>9.991345e-04</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>0.512000</td>
<td>8.680827e-04</td>
<td>1.349727e-02</td>
<td>100</td>
</tr>
<tr>
<td>3</td>
<td>0.461000</td>
<td>1.028377e-04</td>
<td>1.220193e-02</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>0.460000</td>
<td>1.173300e-04</td>
<td>5.244309e-02</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>0.498000</td>
<td>9.477766e-04</td>
<td>2.001293e-03</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>0.490000</td>
<td>1.484369e-04</td>
<td>4.328313e-01</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>0.522000</td>
<td>5.586261e-04</td>
<td>2.312685e-04</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>0.530000</td>
<td>5.808183e-04</td>
<td>8.259964e-02</td>
<td>100</td>
</tr>
<tr>
<td>9</td>
<td>0.489000</td>
<td>1.979168e-04</td>
<td>1.010889e-04</td>
<td>100</td>
</tr>
<tr>
<td>10</td>
<td>0.490000</td>
<td>2.036031e-04</td>
<td>2.406271e-03</td>
<td>100</td>
</tr>
<tr>
<td>11</td>
<td>0.475000</td>
<td>2.021162e-04</td>
<td>2.287007e-01</td>
<td>100</td>
</tr>
<tr>
<td>12</td>
<td>0.460000</td>
<td>1.135527e-04</td>
<td>3.905040e-02</td>
<td>100</td>
</tr>
<tr>
<td>13</td>
<td>0.515000</td>
<td>6.947668e-04</td>
<td>1.562808e-02</td>
<td>100</td>
</tr>
<tr>
<td>14</td>
<td>0.531000</td>
<td>9.471549e-04</td>
<td>1.433895e-03</td>
<td>100</td>
</tr>
<tr>
<td>15</td>
<td>0.509000</td>
<td>3.140888e-04</td>
<td>2.857518e-01</td>
<td>100</td>
</tr>
<tr>
<td>16</td>
<td>0.514000</td>
<td>6.438349e-04</td>
<td>3.033781e-01</td>
<td>100</td>
</tr>
<tr>
<td>17</td>
<td>0.502000</td>
<td>3.921784e-04</td>
<td>2.707126e-04</td>
<td>100</td>
</tr>
<tr>
<td>18</td>
<td>0.509000</td>
<td>9.752279e-04</td>
<td>2.850856e-03</td>
<td>100</td>
</tr>
<tr>
<td>19</td>
<td>0.500000</td>
<td>2.412048e-04</td>
<td>4.997821e-04</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>0.466000</td>
<td>1.319314e-04</td>
<td>1.189915e-02</td>
<td>100</td>
</tr>
<tr>
<td>21</td>
<td>0.516000</td>
<td>8.039527e-04</td>
<td>1.528291e-02</td>
<td>100</td>
</tr>
</tbody>
</table>

53%
(4) Find a learning rate

Coarse to fine search

Remember this is just a 2 layer neural net with 50 neurons

53%
(4) Find a learning rate
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Plot the loss

Figure: Andrej Karpathy
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Plot the loss

Figure: Andrej Karpathy
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —→ visualize as you go

Plot the loss

For very small learning rates, the loss decreases linearly and slowly.
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Plot the loss

For very small learning rates, the loss decreases linearly and slowly

(Why linearly?)

Figure: Andrej Karpathy
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Plot the loss

For very small learning rates, the loss decreases linearly and slowly

(*Why linearly?*)

Larger learning rates tend to look more exponential

*Figure: Andrej Karpathy*
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Typical training loss:

Figure: Andrej Karpathy
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Typical training loss:

Why is it varying so rapidly?

Figure: Andrej Karpathy
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation — visualize as you go

Typical training loss:

Why is it varying so rapidly?

The width of the curve is related to the batchsize — if too noisy, increase the batch size

Figure: Andrej Karpathy
(4) Find a learning rate

Normally, you don’t have the budget for lots of cross-validation —> visualize as you go

Typical training loss:

Why is it varying so rapidly?

The width of the curve is related to the batch size — if too noisy, increase the batch size

Possibly too linear (learning rate too small)

Figure: Andrej Karpathy
(4) Find a learning rate

Visualize the accuracy

Figure: Andrej Karpathy
(4) Find a learning rate

Visualize the accuracy

Figure: Andrej Karpathy
(4) Find a learning rate

Visualize the accuracy

Big gap: overfitting (increase regularization)

Figure: Andrej Karpathy
(4) Find a learning rate

Visualize the accuracy

**Big gap:** overfitting
(increase regularization)

**No gap:** underfitting
(increase model capacity, make layers bigger or decrease regularization)

Figure: Andrej Karpathy
(4) Find a learning rate

Visualize the weights

Figure: Andrej Karpathy
(4) Find a learning rate

Visualize the weights

Noisy weights: possibly regularization not strong enough

Figure: Andrej Karpathy
(4) Find a learning rate

Visualize the weights

Nice clean weights: training is proceeding well

Figure: Alex Krizhevsky, Andrej Karpathy
Summary of things to fiddle
Summary of things to fiddle

- Network architecture
Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type
Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, …)
Summary of things to fiddle

• Network architecture
• Learning rate, decay schedule, update type
• Regularization (L2, L1, maxnorm, dropout, …)
• Loss function (softmax, SVM, …)
Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, …)
- Loss function (softmax, SVM, …)
- Weight initialization
Summary of things to fiddle

- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, …)
- Loss function (softmax, SVM, …)
- Weight initialization
Questions?
30s owl break

http://globe-views.com/dreams/owl.html
Tricks for making training work better
Momentum

Simple but powerful improvement:
Give some “momentum” to the parameters
Momentum

Simple but powerful improvement:
Give some “momentum” to the parameters

Figure: Andrej Karpathy
Momentum

Simple but powerful improvement:
Give some “momentum” to the parameters

\[ v_{i+1} = 0.9 v_i - \alpha \frac{\partial L}{\partial \theta} (\theta_i) \]

\[ \theta_{i+1} = \theta_i + v_{i+1} \]
Momentum

Simple but powerful improvement:
Give some “momentum” to the parameters

\[ v_{i+1} = 0.9v_i - \alpha \frac{\partial L}{\partial \theta}(\theta_i) \]

\[ \theta_{i+1} = \theta_i + v_{i+1} \]
Momentum

Simple but powerful improvement:
Give some “momentum” to the parameters

\[ v_{i+1} = 0.9v_i - \alpha \frac{\partial L}{\partial \theta}(\theta_i) \]

\[ \theta_{i+1} = \theta_i + v_{i+1} \]
Momentum

Simple but powerful improvement: Give some “momentum” to the parameters

\[ v_{i+1} = 0.9v_i - \alpha \frac{\partial L}{\partial \theta}(\theta_i) \]
\[ \theta_{i+1} = \theta_i + v_{i+1} \]

Unfortunate nomenclature: the damping factor is called “momentum”
Momentum

Simple but powerful improvement:
Give some “momentum” to the parameters

\[
\begin{align*}
    v_{i+1} &= 0.9v_i - \alpha \frac{\partial L}{\partial \theta}(\theta_i) \\
    \theta_{i+1} &= \theta_i + v_{i+1}
\end{align*}
\]

Unfortunate nomenclature: the damping factor is called “momentum”

“Lesson from the trenches”: well-tuned SGD with Momentum is very hard to beat for CNNs

Figure: Andrej Karpathy
Momentum

Intuition behind momentum:

Figure: Andrej Karpathy
Momentum

Intuition behind momentum:

- Imagine a ball on the loss surface (its position is the current weight settings)
Momentum

Intuition behind momentum:

- Imagine a ball on the loss surface (its position is the current weight settings)

Figure: Andrej Karpathy
Momentum

Intuition behind momentum:

- Imagine a ball on the loss surface (its position is the current weight settings)
- Directions with lots of oscillations are damped

Figure: Andrej Karpathy
Momentum

Intuition behind momentum:

- Imagine a ball on the loss surface (its position is the current weight settings)
- Directions with lots of oscillations are damped
- Builds up speed in directions with a consistent gradient

Figure: Andrej Karpathy
“RMSprop”

On Geoff Hinton’s coursera lecture 6a, he mentioned various “tricks” including “rmsprop”
“RMSprop”

On Geoff Hinton’s coursera lecture 6a, he mentioned various “tricks” including “rmsprop”

Idea: track the moving average of squared gradients
“RMSprop”

On Geoff Hinton’s coursera lecture 6a, he mentioned various “tricks” including “rmsprop”

**Idea:** track the moving average of squared gradients

```python
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += -learning_rate * dx / np.sqrt(cache + 1e-8)
```
“RMSprop”

On Geoff Hinton’s coursera lecture 6a, he mentioned various “tricks” including “rmsprop”

**Idea:** track the moving average of squared gradients

```python
cache = decay_rate * cache + (1 - decay_rate) * dx**2
x += - learning_rate * dx / np.sqrt(cache + 1e-8)
```

decay_rate is a hyper-parameter (typically 0.9, 0.99, or 0.999)
Weight Initialization

For deep nets, initialization is subtle and important:

Weight Initialization

For deep nets, initialization is subtle and important:

Initialize weights to be smaller if there are more input connections:

\[ W = \text{np.random.randn}(n) \times \sqrt{2.0 / n} \]

For deep nets, initialization is subtle and important:

For neural nets with ReLU, this will ensure all activations have the same variance

\[ W = np.random.randn(n) * \sqrt{2.0 / n} \]

Initialization matters

Training can take much longer if not carefully initialized:

Initialization matters

Training can take much longer if not carefully initialized:

22 layer model

Initialization matters

Training can take much longer if not carefully initialized:

Regularization

L2 regularization

\[ L_{\text{reg}} = \lambda \frac{1}{2} \| W \|_2^2 \]

(L2 regularization encourages small weights)
Regularization

L2 regularization

\[ L_{\text{reg}} = \lambda \frac{1}{2} ||W||_2^2 \]

(L2 regularization encourages small weights)

L1 regularization

\[ L_{\text{reg}} = \lambda ||W||_1 = \lambda \sum_{ij} |W_{ij}| \]

(L1 regularization encourages sparse weights: weights are encouraged to reduce to exactly zero)
Regularization

**L2 regularization**

\[ L_{\text{reg}} = \lambda \frac{1}{2} ||W||_2^2 \]

(L2 regularization encourages small weights)

**L1 regularization**

\[ L_{\text{reg}} = \lambda ||W||_1 = \lambda \sum_{ij} |W_{ij}| \]

(L1 regularization encourages sparse weights: weights are encouraged to reduce to exactly zero)

**“Elastic net”**

\[ L_{\text{reg}} = \lambda_1 ||W||_1 + \lambda_2 ||W||_2^2 \]

(Combine L1 and L2 regularization)
Regularization

L2 regularization

\[ L_{\text{reg}} = \lambda \frac{1}{2} \| W \|_2^2 \]

(L2 regularization encourages small weights)

L1 regularization

\[ L_{\text{reg}} = \lambda \| W \|_1 = \lambda \sum_{ij} |W_{ij}| \]

(L1 regularization encourages sparse weights: weights are encouraged to reduce to exactly zero)

“Elastic net”

\[ L_{\text{reg}} = \lambda_1 \| W \|_1 + \lambda_2 \| W \|_2^2 \]

(combine L1 and L2 regularization)

Max norm

Clamp weights to some max norm

\[ \| W \|_2^2 \leq c \]
“Weight decay”

Regularization is also called “weight decay” because the weights “decay” each iteration:

\[ L_{\text{reg}} = \lambda \frac{1}{2} \| W \|_2^2 \]

[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]
“Weight decay”

Regularization is also called “weight decay” because the weights “decay” each iteration:

$$L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2 \quad \rightarrow \quad \frac{\partial L}{\partial W} = \lambda W$$
“Weight decay”

Regularization is also called “weight decay” because the weights “decay” each iteration:

\[
L_{\text{reg}} = \lambda \frac{1}{2} \|W\|^2 \quad \longrightarrow \quad \frac{\partial L}{\partial W} = \lambda W
\]

Gradient descent step:

\[
W \leftarrow W - \alpha \lambda W - \frac{\partial L_{\text{data}}}{\partial W}
\]

[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]
“Weight decay”

Regularization is also called “weight decay” because the weights “decay” each iteration:

\[
L_{\text{reg}} = \lambda \frac{1}{2} ||W||_2^2 \quad \rightarrow \quad \frac{\partial L}{\partial W} = \lambda W
\]

Gradient descent step:

\[
W \leftarrow W - \alpha \lambda W - \frac{\partial L_{\text{data}}}{\partial W}
\]

Weight decay: \(\alpha \lambda\) (weights always decay by this amount)

[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]
“Weight decay”

Regularization is also called “weight decay” because the weights “decay” each iteration:

\[ L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2 \quad \rightarrow \quad \frac{\partial L}{\partial W} = \lambda W \]

Gradient descent step:

\[ W \leftarrow W - \alpha \lambda W - \frac{\partial L_{\text{data}}}{\partial W} \]

Weight decay: \( \alpha \lambda \) (weights always decay by this amount)

**Note:** typically, biases are excluded from regularization

[Andrej Karpathy http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html]
Dropout

Simple but powerful technique to reduce overfitting:

Dropout

Simple but powerful technique to reduce overfitting:

Dropout

Simple but powerful technique to reduce overfitting:

Dropout

Simple but powerful technique to reduce overfitting:

(a) Standard Neural Net

Dropout

Simple but powerful technique to reduce overfitting:

(a) Standard Neural Net
(b) After applying dropout.

Dropout

Simple but powerful technique to reduce overfitting:

Note: Dropout can be interpreted as an approximation to taking the geometric mean of an ensemble of exponentially many models

Dropout

How much dropout?

Dropout

How much dropout?

(a) Keeping $n$ fixed.
(b) Keeping $pn$ fixed.

Dropout

How much dropout?  Around $p = 0.5$

Dropout

Case study: [Krizhevsky 2012]

“Without dropout, our network exhibits substantial overfitting.”

[Krizhevsky et al, “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012]
Case study: [Krizhevsky 2012]

“Without dropout, our network exhibits substantial overfitting.”

[Krizhevsky et al, “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012]
Dropout

Case study: [Krizhevsky 2012]
“Without dropout, our network exhibits substantial overfitting.”

Dropout here

But not here — why?

[Krizhevsky et al, “ImageNet Classification with Deep Convolutional Neural Networks”, NIPS 2012]
Dropout

$p = 0.5$ # probability of keeping a unit active. higher = less dropout

```python
def train_step(X):
    """ X contains the data """

    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p # first dropout mask
    H1 *= U1 # drop!
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2 # drop!
    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```

(note, here $X$ is a single input)

Figure: Andrej Karpathy
Dropout

Test time: scale the activations

Figure: Andrej Karpathy
Dropout

Test time: scale the activations

Expected value of a neuron $h$ with dropout:

$$E[h] = ph + (1 - p)0 = ph$$
Dropout

**Test time:** scale the activations

Expected value of a neuron $h$ with dropout:

$$E[h] = ph + (1 - p)0 = ph$$

```python
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```
Dropout

**Test time:** scale the activations

Expected value of a neuron $h$ with dropout:

$$E[h] = ph + (1 - p)0 = ph$$

```python
def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

We want to keep the same expected value
Learning rate schedule

How do we change the learning rate over time?

Various choices:
Learning rate schedule

How do we change the learning rate over time?

Various choices:

• Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])
Learning rate schedule

How do we change the learning rate over time?

Various choices:

• Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])

• Decrease by a factor of 0.97 every epoch (used by GoogLeNet [Szegedy 2014])
Learning rate schedule

How do we change the learning rate over time?

Various choices:

- Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])
- Decrease by a factor of 0.97 every epoch (used by GoogLeNet [Szegedy 2014])
- Scale by $\sqrt{1-t/\max_t}$ (used by BVLC to re-implement GoogLeNet)
Learning rate schedule

How do we change the learning rate over time?

Various choices:

- Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])
- Decrease by a factor of 0.97 every epoch (used by GoogLeNet [Szegedy 2014])
- Scale by \sqrt{1-t/\text{max}_t}
  (used by BVLC to re-implement GoogLeNet)
- Scale by $1/t$
Learning rate schedule

How do we change the learning rate over time?

Various choices:

• Step down by a factor of 0.1 every 50,000 mini-batches (used by SuperVision [Krizhevsky 2012])

• Decrease by a factor of 0.97 every epoch (used by GoogLeNet [Szegedy 2014])

• Scale by sqrt(1-t/max_t) (used by BVLC to re-implement GoogLeNet)

• Scale by 1/t

• Scale by exp(-t)
Hints for PA5
A note on weight sharing

How does backprop work for shared parameters?
A note on weight sharing

How does backprop work for shared parameters?

Consider a “copy” layer that replicates its input:
A note on weight sharing

How does backprop work for shared parameters?
Consider a “copy” layer that replicates its input:

\[
x \rightarrow \text{Copy} \rightarrow x_1, \ldots, x_m
\]
A note on weight sharing

How does backprop work for shared parameters?

Consider a “copy” layer that replicates its input:

\[ x \rightarrow \text{Copy} \rightarrow x_1 \rightarrow \cdots \rightarrow x_m \]

The backwards pass is:
A note on weight sharing

How does backprop work for shared parameters?

Consider a “copy” layer that replicates its input:

\[ x \rightarrow \text{Copy} \]

The backwards pass is:

\[
\frac{\partial L}{\partial x} = \sum_i \frac{\partial L}{\partial x_i} \frac{\partial x_i}{\partial x}
\]
A note on weight sharing

How does backprop work for shared parameters?

Consider a “copy” layer that replicates its input:

\[ x \rightarrow \text{Copy} \]

The backwards pass is:

\[
\frac{\partial L}{\partial x} = \sum_i \frac{\partial L}{\partial x_i} \frac{\partial x_i}{\partial x} = \sum_i \frac{\partial L}{\partial x_i} 1
\]
A note on weight sharing

How does backprop work for shared parameters?

Consider a “copy” layer that replicates its input:

The backwards pass is:

$$\frac{\partial L}{\partial x} = \sum_i \frac{\partial L}{\partial x_i} \frac{\partial x_i}{\partial x} = \sum_i \frac{\partial L}{\partial x_i} \cdot 1 = \sum_i \frac{\partial L}{\partial x_i}$$
A note on weight sharing

How does backprop work for shared parameters?

Consider a “copy” layer that replicates its input:

The backwards pass is:

$$
\frac{\partial L}{\partial x} = \sum_i \frac{\partial L}{\partial x_i} \frac{\partial x_i}{\partial x} = \sum_i \frac{\partial L}{\partial x_i} \cdot 1 = \sum_i \frac{\partial L}{\partial x_i}
$$

Thus, when values are shared, their gradients get added.
A note on weight sharing
A note on weight sharing

Backwards pass for convolution:
A note on weight sharing

Backwards pass for convolution:

- Convolution has weight sharing across positions
A note on weight sharing

Backwards pass for convolution:

- Convolution has weight sharing across positions

- Thus, to figure out the backwards pass, figure out the gradient for one position, and accumulate the gradients across positions
A note on weight sharing

Backwards pass for convolution:

- Convolution has **weight sharing** across positions

- Thus, to figure out the backwards pass, figure out the gradient for one position, and accumulate the gradients across positions

- Don’t try and derive a closed-form expression
A note on weight sharing

Backwards pass for convolution:

- Convolution has weight sharing across positions

- Thus, to figure out the backwards pass, figure out the gradient for one position, and accumulate the gradients across positions

- Don’t try and derive a closed-form expression

Heads up: we might make the backwards pass of convolution required (not extra-credit)
Wow! That was a great lecture!

I'm so confused.

GREAT MOMENTS IN TEACHING

2008©

http://brownsharpie.courtneygibbons.org/?p=746