

ITCS 6156/8156 Fall 2024

Machine Learning

Convolutional Neural Networks

Instructor: Hongfei Xue

Email: hongfei.xue@charlotte.edu

Class Meeting: Tue & Thu, 4:00 PM – 5:15 PM, WWH 130

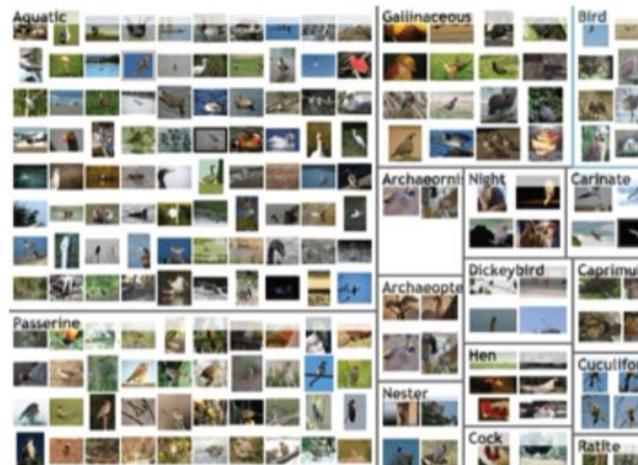


Some content in the slides is based on DeepMind's lecture

Training ConvNets

Roughly speaking:

Gather
labeled data



Find a ConvNet
architecture



Minimize
the loss



Training a ConvNet

- 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

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:

$$L = L_{\text{data}} + L_{\text{reg}}$$

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

$$\lambda = 0.001$$

$$\lambda = 0.01$$

$$\lambda = 0.1$$



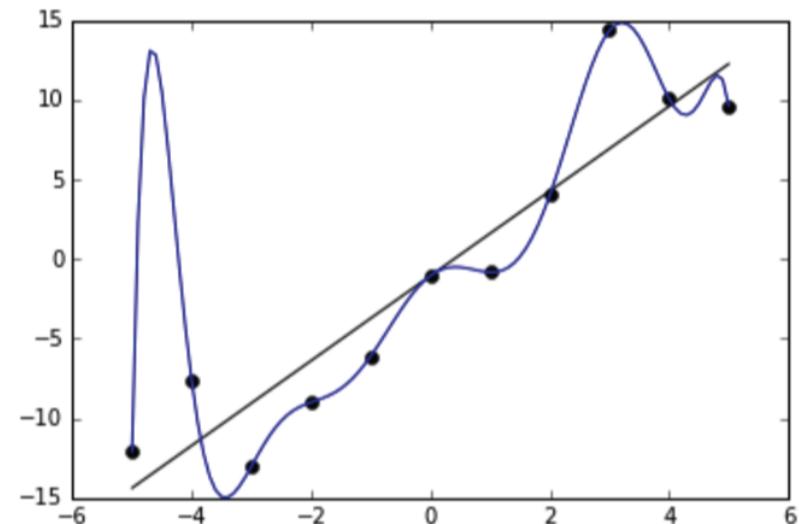
[Andrej Karpathy <http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>]

Overfitting

Overfitting: modeling noise in the training set instead of the “true” underlying relationship

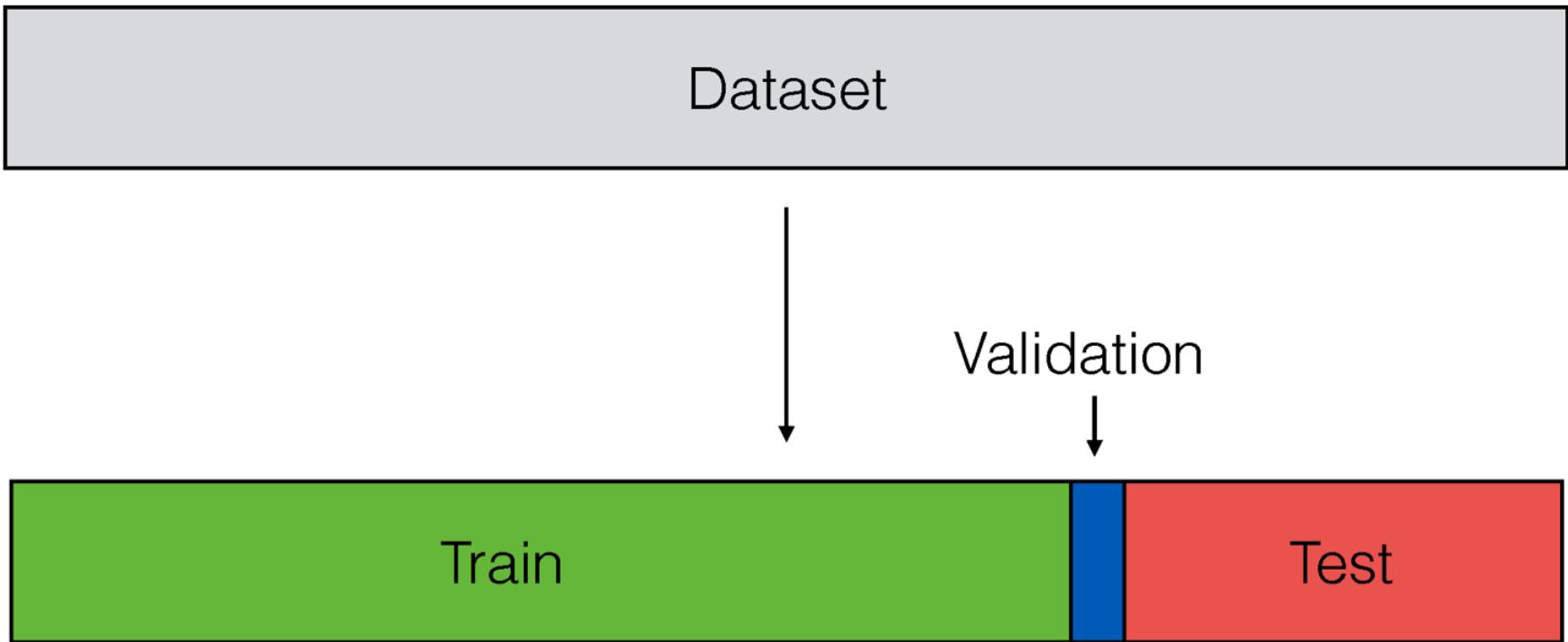
Underfitting: insufficiently modeling the relationship in the training set

General rule: models that are “bigger” or have more capacity are more likely to overfit



Dataset Split

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



Dataset Split



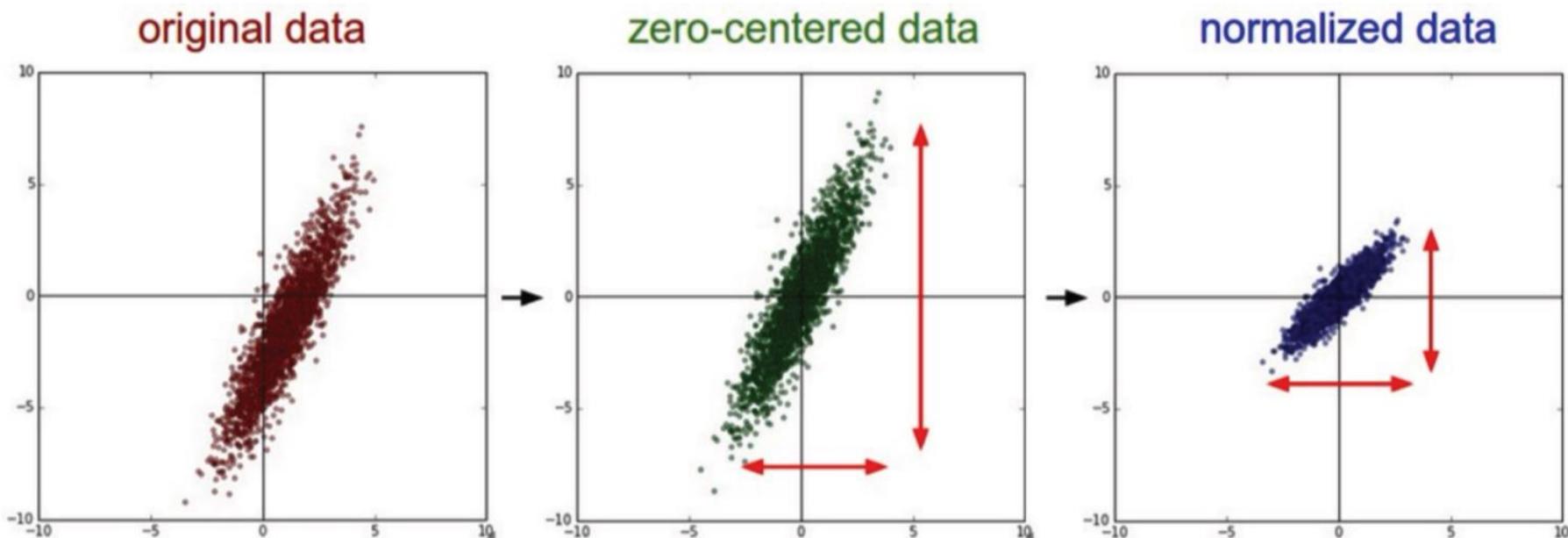
Train: gradient descent and fine-tuning of parameters

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

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

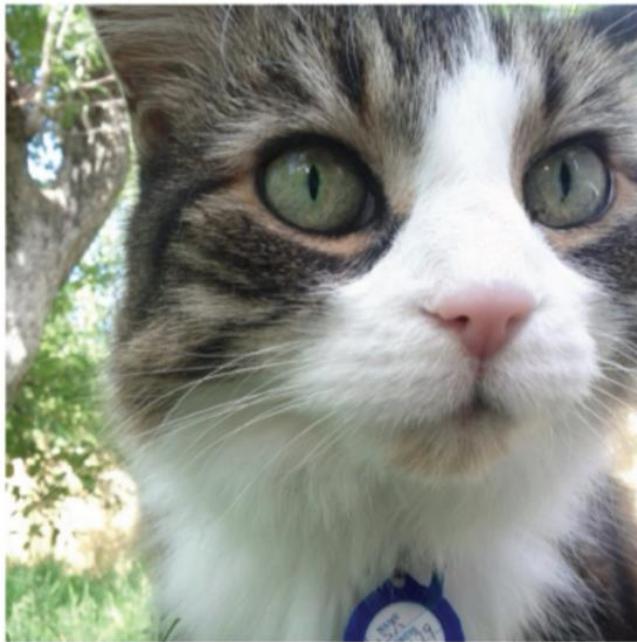
Data Preprocessing

Preprocess the data so that learning is better conditioned:



Data Preprocessing

For ConvNets, typically only the mean is subtracted.



An input image (256x256)



Minus sign



The mean input image

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

Data Preprocessing

Data augmentation



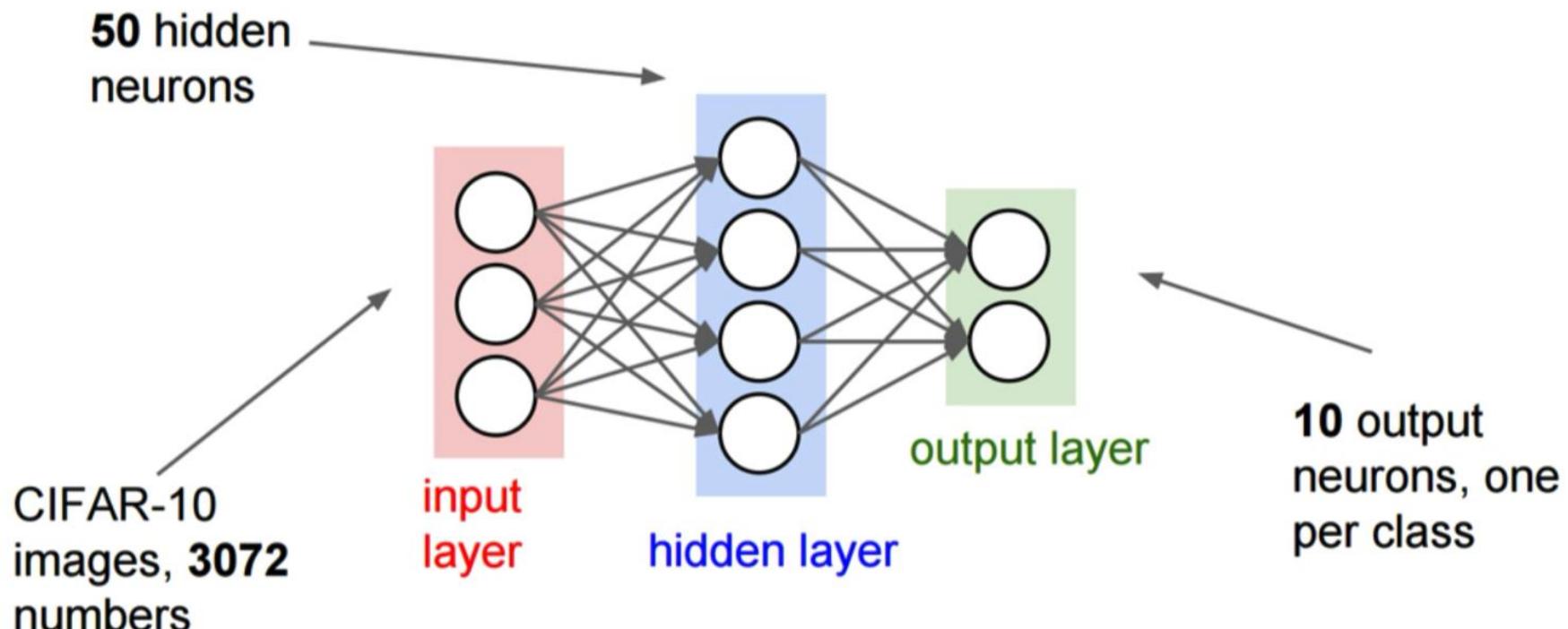
By design, convnets are only robust against translation

Data augmentation makes them robust against other transformations: rotation, scaling, shearing, warping, ...



Choose the Architecture

Toy example: one hidden layer of size 50



Initialize the Weights

Set the weights to small random numbers:

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

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

(the magnitude is important and this is not optimal — more on this later)

Set the bias to zero (or small nonzero):

```
b = np.zeros(H)
```

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
```

returns the loss and the
gradient for all parameters

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)    crank up regularization
print loss
```



loss went up, good. (sanity check)

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:

'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

Overfit a Small Portion of the Data

100% accuracy on the training set (good)

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 20 / 200: cost 1.305760, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 195 / 200: cost 0.002694, train: 1.000000 val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000 val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000 val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000 val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000 val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000 val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Find a Learning Rate

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

```
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)
```

Find a Learning Rate

```
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)
```

Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000

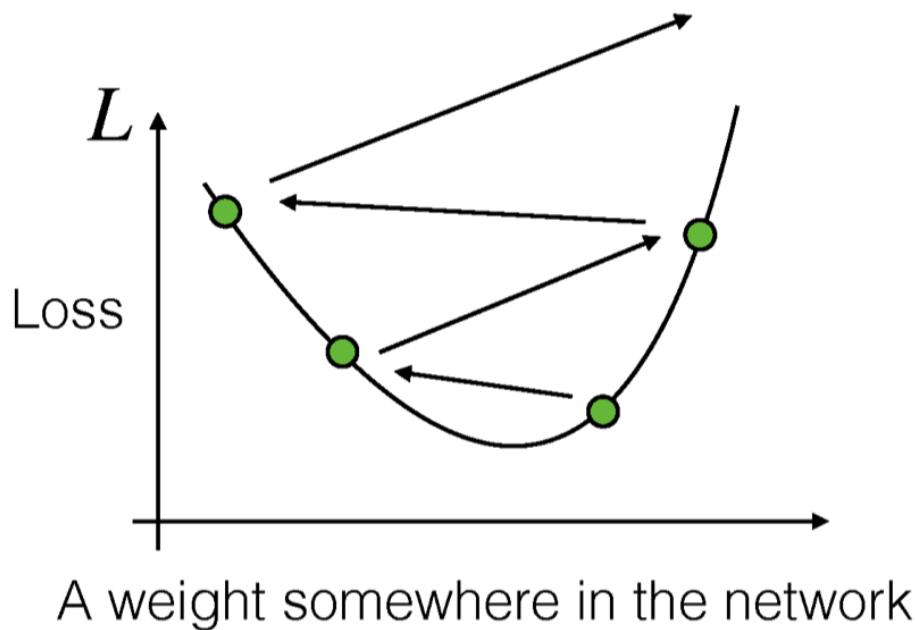
Loss barely changes

Why is the accuracy 20%?

(learning rate is too low or regularization too high)

Find a Learning Rate

Learning rate: 1e6 — what could go wrong?



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 $\text{loss} > 3 * \text{original loss}$, quit early
(learning rate too high)

Find a Learning Rate

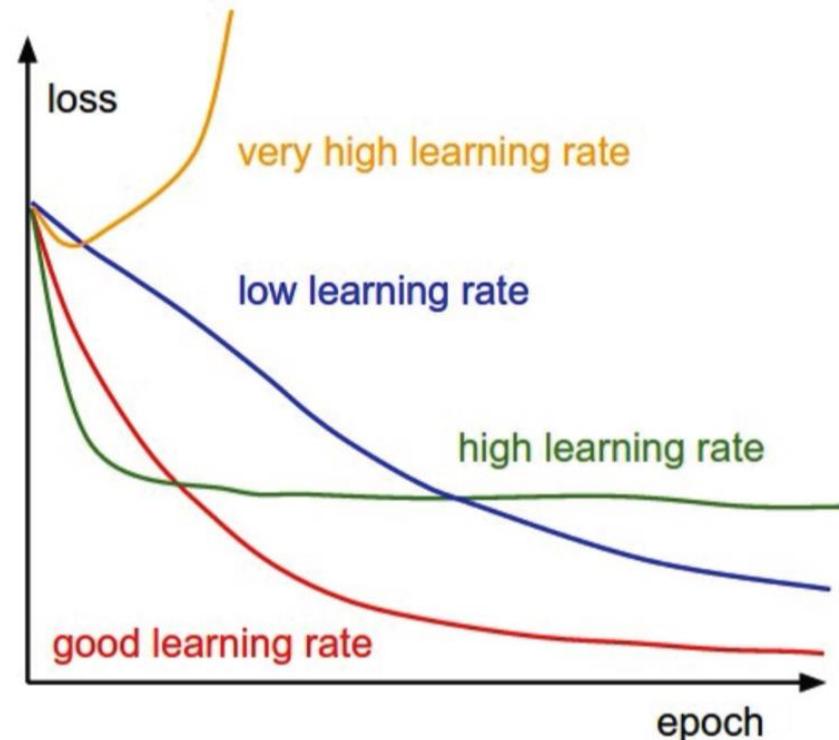
Normally, you don't have the budget for lots of cross-validation —> visualize as you go

Plot the loss

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

(Why linear?)

Larger learning rates tend to look more exponential



Find a Learning Rate

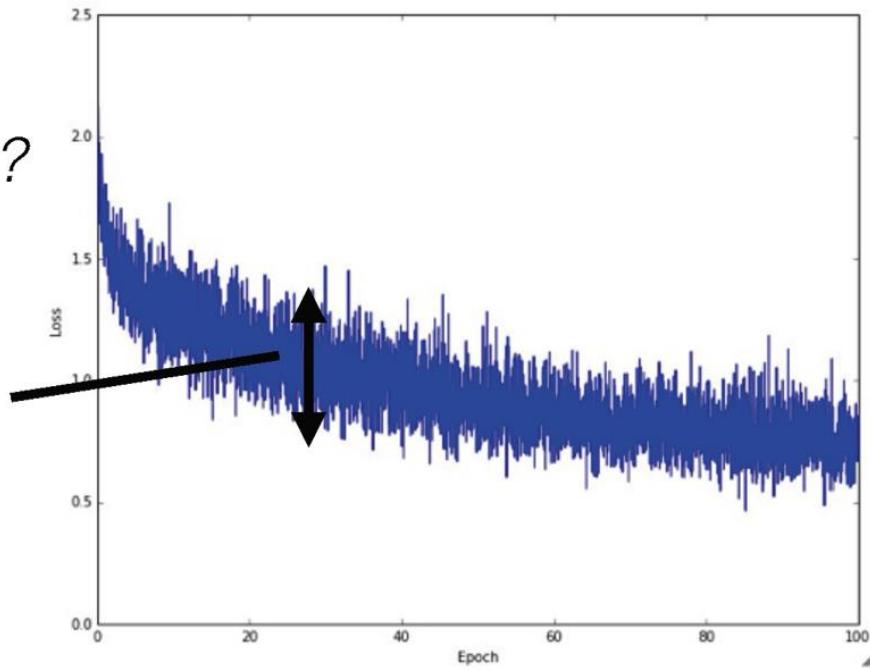
Normally, you don't have the budget for lots of cross-validation —> visualize as you go

Typical training loss:

Why is it varying so rapidly?

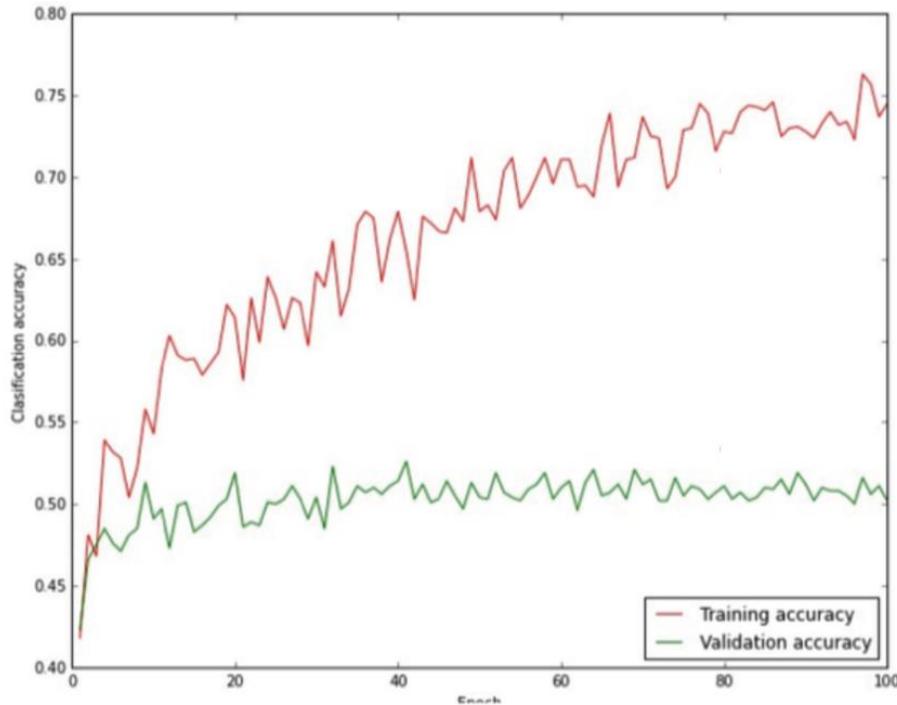
The width of the curve is related to the batchsize — if too noisy, increase the batch size

Possibly too linear
(learning rate too small)



Find a Learning Rate

Visualize the accuracy



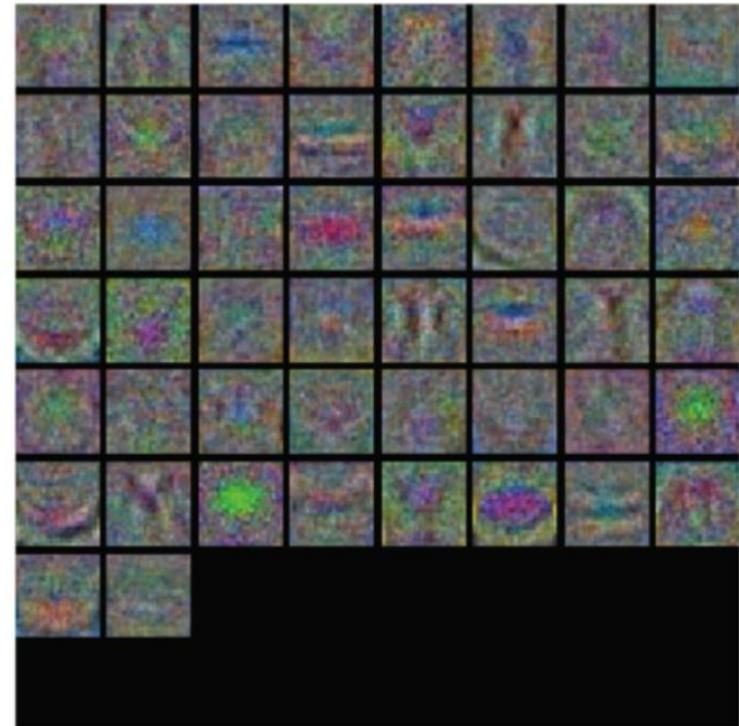
Big gap: overfitting
(increase regularization)

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

Find a Learning Rate

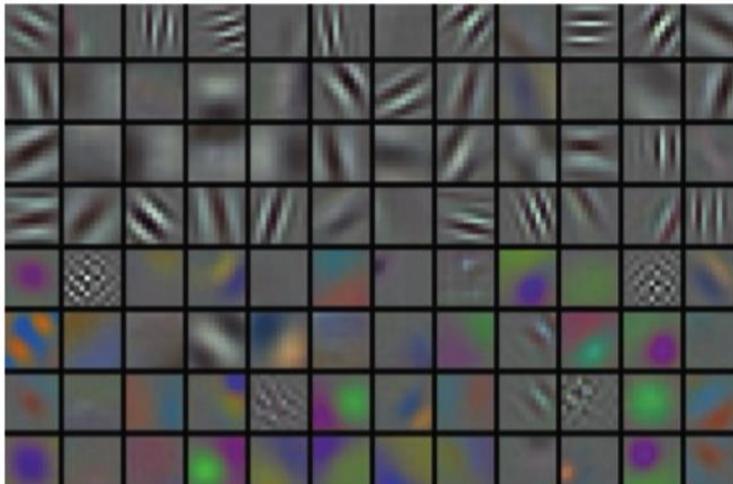
Visualize the weights

Noisy weights: possibly regularization not strong enough

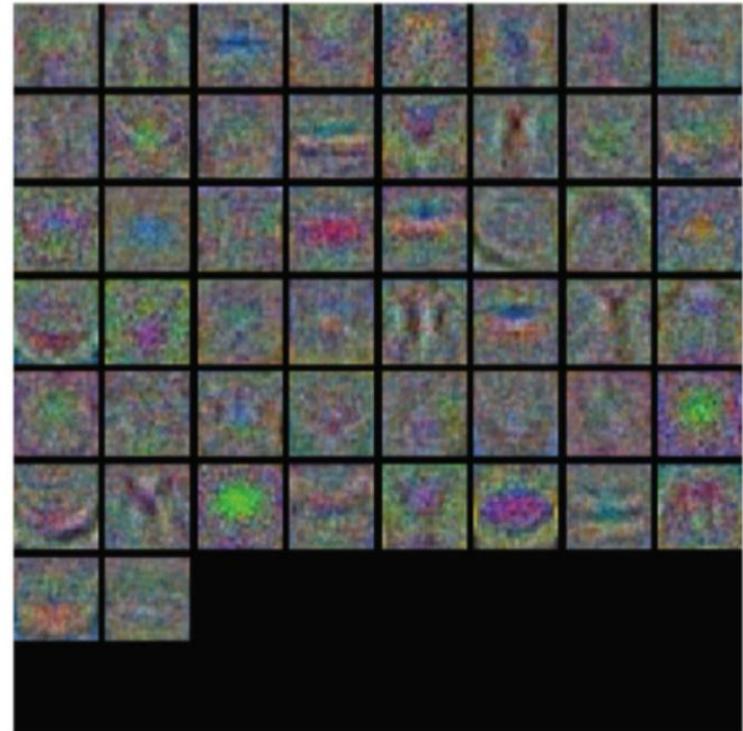


Find a Learning Rate

Visualize the weights



Nice clean weights:
training is proceeding well



Find a Learning Rate

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 $\exp(-t)$

Summary of Things to Fiddle

- Network architecture
- Learning rate, decay schedule, update type
- Regularization (L2, L1, maxnorm, dropout, ...)
- Loss function (softmax, SVM, ...)
- Weight initialization

Neural network
parameters



(Recall) Regularization Reduces Overfitting

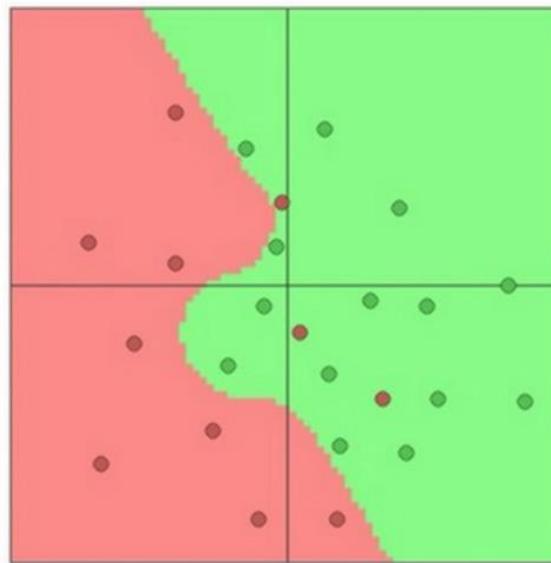
$$L = L_{\text{data}} + L_{\text{reg}}$$

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

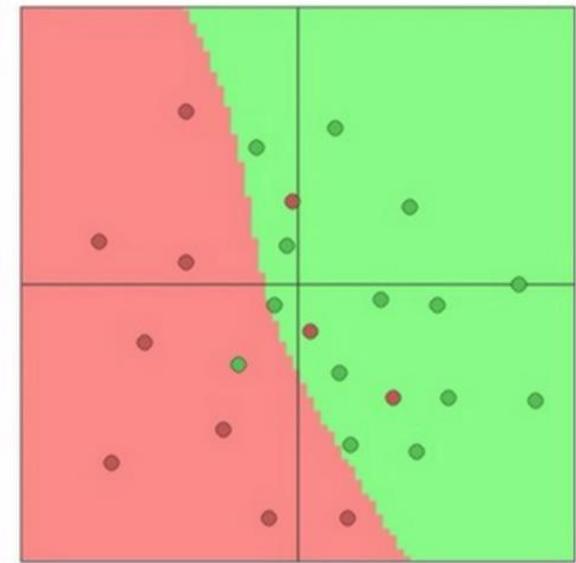
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$



Example Regularizers

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:

$$L_{\text{reg}} = \lambda \frac{1}{2} \|W\|_2^2 \quad \longrightarrow \quad \frac{\partial L}{\partial W} = \lambda W$$

Gradient descent step:

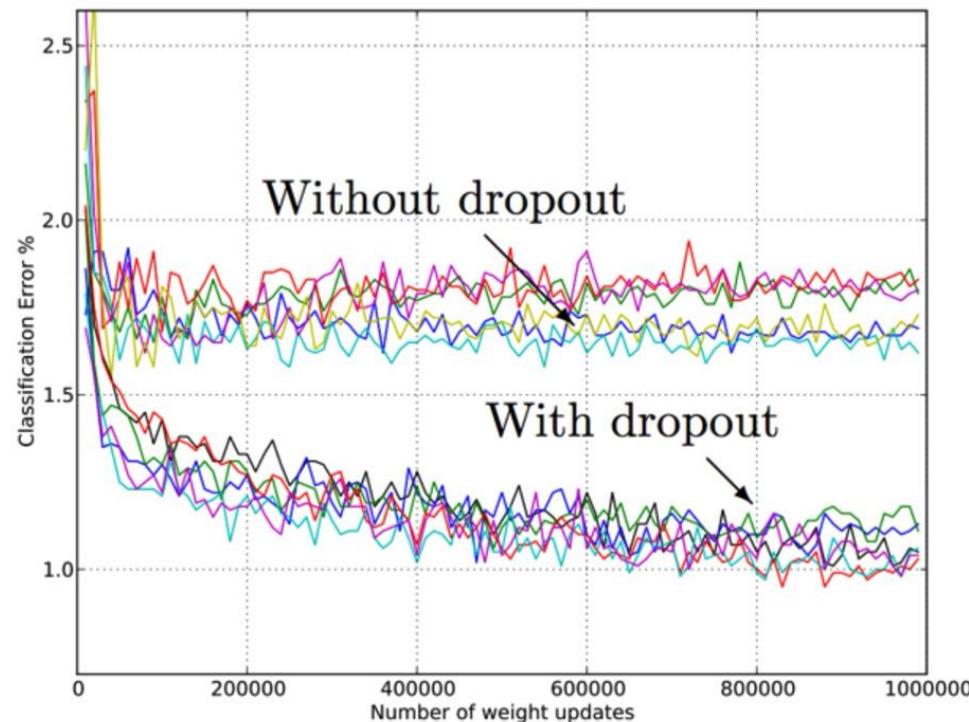
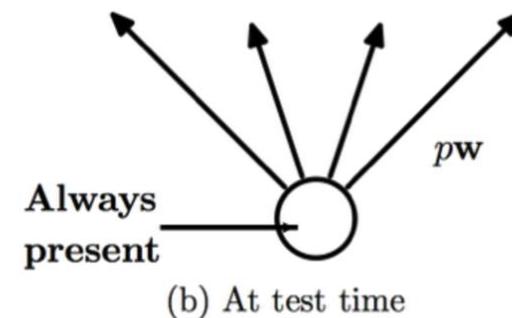
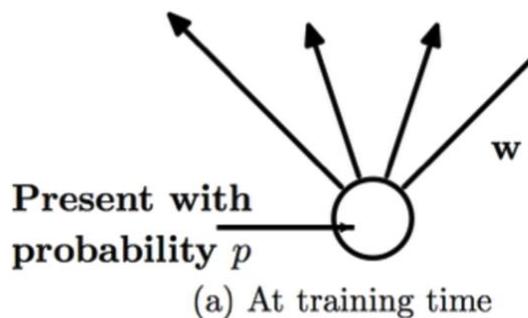
$$W \leftarrow W - \alpha \lambda W - \frac{\partial L_{\text{data}}}{\partial W}$$

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

Note: biases are sometimes excluded from regularization

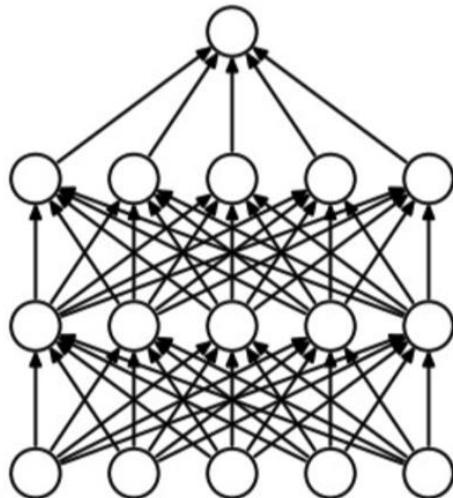
Dropout

Simple but powerful technique to reduce overfitting:

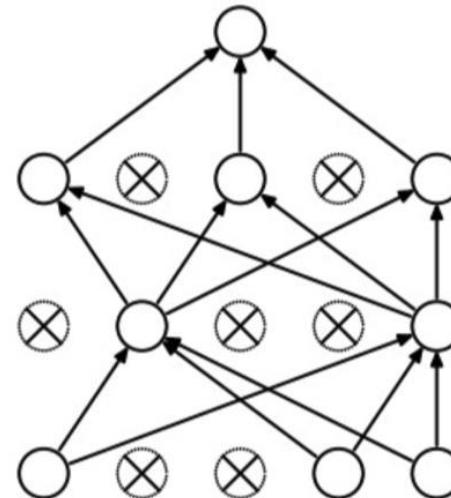


Dropout

Simple but powerful technique to reduce overfitting:



(a) Standard Neural Net

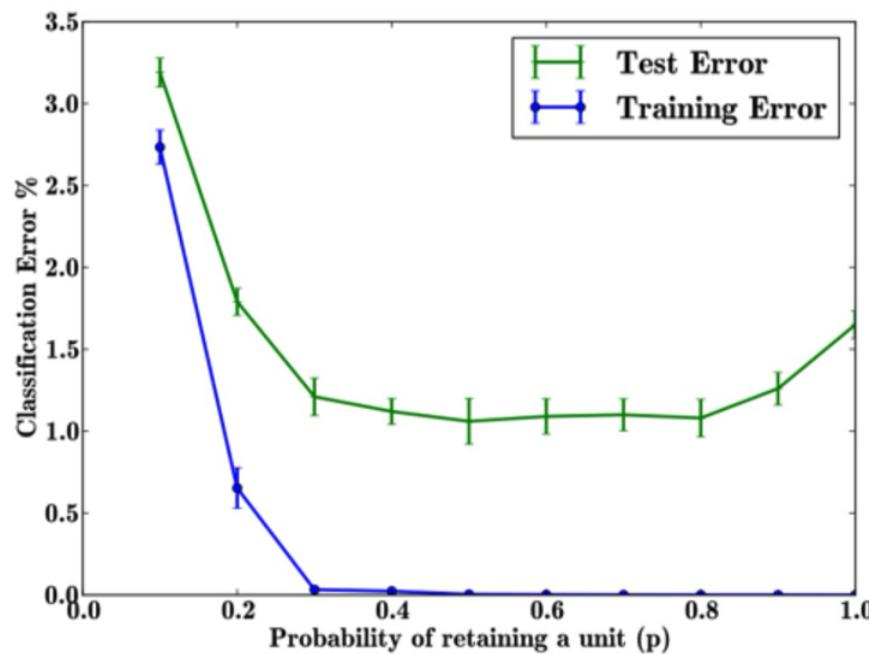


(b) After applying dropout.

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

Dropout

How much dropout? Around $p = 0.5$



(a) Keeping n fixed.

Dropout

Test time: scale the activations

Expected value of a neuron h with dropout:

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

```
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

Summary

- Preprocess the data (subtract mean, sub-crops)
- Initialize weights carefully
- Use Dropout
- Use SGD + Momentum
- Fine-tune from ImageNet
- Babysit the network as it trains

Questions?