Training convolutional neural networks
Readings

• Stochastic Gradient Descent & Backpropagation
  – http://cs231n.github.io/optimization-1/
  – http://cs231n.github.io/optimization-2/

• Best practices for training CNNs
Announcements

• Final exam in class, May 9
  – Will provide study materials Wednesday
  – Final is open book / open note (please use your judgement – see Piazza for more info)
  – No laptops / iPads / phones. Calculator is OK.

• Project 5 (CNNs) to be released by Wednesday
  – Likely due Monday, 5/14
Last time

- Backpropagation algorithm
- Training networks via gradient descent
Today

• Best practices for training CNNs
Where we are now...

Learning network parameters through optimization

```python
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += - step_size * weights_grad # perform parameter update
```
How do you actually train these things?

Roughly speaking:

- Gather labeled data
- Find a ConvNet architecture
- Minimize the loss
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

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 reduces overfitting:

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

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

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

(0) Dataset split

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

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
(1) Data preprocessing

Preprocess the data so that learning is better conditioned:

Original data → zero-centered data → normalized data

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

Figure: Andrej Karpathy
(1) 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).

Figure: Alex Krizhevsky
(1) Data preprocessing

Augment the data — extract random crops from the input, with slightly jittered offsets. Without this, typical ConvNets (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

CIFAR-10 images, 3072 numbers

50 hidden neurons

input layer

hidden layer

output layer

10 output neurons, one per class

Slide: Andrej Karpathy
Demo time

https://playground.tensorflow.org/
(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)

Set the bias to zero (or small nonzero):

\[ b = \text{np.zeros}(H) \]

(if you use ReLU activations, folks tend to initialize bias to small positive number)
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
(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 = {}
    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)

3.06859716482

loss went up, good. (sanity check)
(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,
um_epochs=200, reg=0.0,
update='sgd', learning_rate_decay=1,
sample_batches = False,
learning_rate=1e-3, verbose=True)
```

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

‘sparse coding 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

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

100% accuracy on the training set (good)

<table>
<thead>
<tr>
<th>Finished epoch</th>
<th>cost</th>
<th>train:</th>
<th>val:</th>
<th>lr</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 / 200</td>
<td>2.302603</td>
<td>0.40000</td>
<td>0.40000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>2 / 200</td>
<td>2.302258</td>
<td>0.45000</td>
<td>0.45000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>3 / 200</td>
<td>2.301849</td>
<td>0.60000</td>
<td>0.60000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>4 / 200</td>
<td>2.301196</td>
<td>0.65000</td>
<td>0.65000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>5 / 200</td>
<td>2.300044</td>
<td>0.65000</td>
<td>0.65000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>6 / 200</td>
<td>2.297864</td>
<td>0.55000</td>
<td>0.55000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>7 / 200</td>
<td>2.293595</td>
<td>0.60000</td>
<td>0.60000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>8 / 200</td>
<td>2.285096</td>
<td>0.55000</td>
<td>0.55000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>9 / 200</td>
<td>2.268094</td>
<td>0.55000</td>
<td>0.55000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>10 / 200</td>
<td>2.234787</td>
<td>0.50000</td>
<td>0.50000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>11 / 200</td>
<td>2.173187</td>
<td>0.50000</td>
<td>0.50000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>12 / 200</td>
<td>2.076862</td>
<td>0.50000</td>
<td>0.50000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>13 / 200</td>
<td>1.974090</td>
<td>0.40000</td>
<td>0.40000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>14 / 200</td>
<td>1.895885</td>
<td>0.40000</td>
<td>0.40000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>15 / 200</td>
<td>1.820876</td>
<td>0.45000</td>
<td>0.45000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>16 / 200</td>
<td>1.737430</td>
<td>0.45000</td>
<td>0.45000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>17 / 200</td>
<td>1.642356</td>
<td>0.50000</td>
<td>0.50000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>18 / 200</td>
<td>1.535239</td>
<td>0.60000</td>
<td>0.60000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>19 / 200</td>
<td>1.421527</td>
<td>0.60000</td>
<td>0.60000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>20 / 200</td>
<td>1.307760</td>
<td>0.65000</td>
<td>0.65000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>195 / 200</td>
<td>0.002694</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>196 / 200</td>
<td>0.002674</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>197 / 200</td>
<td>0.002655</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>198 / 200</td>
<td>0.002635</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>199 / 200</td>
<td>0.002617</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.000000e-03</td>
</tr>
<tr>
<td>200 / 200</td>
<td>0.002597</td>
<td>1.00000</td>
<td>1.00000</td>
<td>1.000000e-03</td>
</tr>
</tbody>
</table>

finished optimization. best validation accuracy: 1.000000
Babysitting learning

Q: Which one of these learning rates is best to use?
(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

Loss barely changes
(leaving rate is too low or regularization too high)

Slide: Andrej Karpathy
(4) Find a learning rate

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

```
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

Loss is NaN $\rightarrow$ learning rate is too high
(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

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 3.0000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.0000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.0000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.0000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.0000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.0000000e-03

Loss is inf —> still too high
But now we know we should be searching the range [1e-5 ... 1e-3]
(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 $> 3 \times$ original loss, quit early (learning rate too high)
(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)
```
(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>val_acc</th>
<th>lr</th>
<th>reg</th>
<th>(count / max_count)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.527000</td>
<td>5.340517e-04</td>
<td>4.097824e-01</td>
<td>(0 / 100)</td>
</tr>
<tr>
<td>0.492000</td>
<td>2.279484e-04</td>
<td>9.991345e-04</td>
<td>(1 / 100)</td>
</tr>
<tr>
<td>0.512000</td>
<td>8.680827e-04</td>
<td>1.349727e-02</td>
<td>(2 / 100)</td>
</tr>
<tr>
<td>0.461000</td>
<td>1.028377e-04</td>
<td>1.220193e-02</td>
<td>(3 / 100)</td>
</tr>
<tr>
<td>0.460000</td>
<td>1.113730e-04</td>
<td>5.244309e-02</td>
<td>(4 / 100)</td>
</tr>
<tr>
<td>0.498000</td>
<td>9.477756e-04</td>
<td>2.001293e-03</td>
<td>(5 / 100)</td>
</tr>
<tr>
<td>0.469000</td>
<td>1.484369e-04</td>
<td>4.328313e-01</td>
<td>(6 / 100)</td>
</tr>
<tr>
<td>0.522000</td>
<td>5.586261e-04</td>
<td>2.312685e-04</td>
<td>(7 / 100)</td>
</tr>
<tr>
<td>0.530000</td>
<td>5.808183e-04</td>
<td>8.259964e-02</td>
<td>(8 / 100)</td>
</tr>
<tr>
<td>0.489000</td>
<td>1.979168e-04</td>
<td>1.010889e-04</td>
<td>(9 / 100)</td>
</tr>
<tr>
<td>0.490000</td>
<td>2.036031e-04</td>
<td>2.406271e-03</td>
<td>(10 / 100)</td>
</tr>
<tr>
<td>0.475000</td>
<td>2.021162e-04</td>
<td>2.287007e-01</td>
<td>(11 / 100)</td>
</tr>
<tr>
<td>0.460000</td>
<td>1.135527e-04</td>
<td>3.905040e-02</td>
<td>(12 / 100)</td>
</tr>
<tr>
<td>0.515000</td>
<td>6.947668e-04</td>
<td>1.562808e-02</td>
<td>(13 / 100)</td>
</tr>
<tr>
<td>0.531000</td>
<td>9.471549e-04</td>
<td>1.433895e-03</td>
<td>(14 / 100)</td>
</tr>
<tr>
<td>0.509000</td>
<td>3.140888e-04</td>
<td>2.857518e-01</td>
<td>(15 / 100)</td>
</tr>
<tr>
<td>0.514000</td>
<td>6.438349e-04</td>
<td>3.033581e-01</td>
<td>(16 / 100)</td>
</tr>
<tr>
<td>0.502000</td>
<td>3.921784e-04</td>
<td>2.707126e-04</td>
<td>(17 / 100)</td>
</tr>
<tr>
<td>0.509000</td>
<td>9.752279e-04</td>
<td>2.850865e-03</td>
<td>(18 / 100)</td>
</tr>
<tr>
<td>0.500000</td>
<td>2.412048e-04</td>
<td>4.997821e-04</td>
<td>(19 / 100)</td>
</tr>
<tr>
<td>0.466000</td>
<td>1.319314e-04</td>
<td>1.189915e-02</td>
<td>(20 / 100)</td>
</tr>
<tr>
<td>0.516000</td>
<td>8.039527e-04</td>
<td>1.528291e-02</td>
<td>(21 / 100)</td>
</tr>
</tbody>
</table>

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

53%
(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

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 \(\rightarrow\) 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

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

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
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)
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?
(Recall) Regularization reduces overfitting

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

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

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

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

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: biases are sometimes excluded from regularization

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

Simple but powerful technique to reduce overfitting:

(a) At training time

Present with probability $p$

(b) At test time

Always present

$pw$

Dropout

Simple but powerful technique to reduce overfitting:

Regularization: Dropout
How can this possibly be a good idea?

Forces the network to have a redundant representation;
Prevents co-adaptation of features

- has an ear
- has a tail
- is furry
- has claws
- mischievous look

X

cat score

X

X
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

Case study: [Krizhevsky 2012]

"Without dropout, our network exhibits substantial overfitting."

[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

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

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

*Figure: Andrej Karpathy*
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?
Transfer Learning

“You need a lot of a data if you want to train/use CNNs”
Transfer Learning

“You need a lot of data if you want to train/use CNNs”
Transfer Learning with CNNs

1. Train on Imagenet

- FC-1000
- FC-4096
- FC-4096
- MaxPool
- Conv-512
- Conv-512
- MaxPool
- Conv-512
- Conv-512
- MaxPool
- Conv-256
- Conv-256
- MaxPool
- Conv-128
- Conv-128
- MaxPool
- Conv-64
- Conv-64

Image
Transfer Learning with CNNs

1. Train on Imagenet
   - FC-1000
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-266
   - Conv-266
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image

2. Small Dataset (C classes)
   - FC-C
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-266
   - Conv-266
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image

Reinitialize this and train
Freeze these
Transfer Learning with CNNs

1. Train on Imagenet
   - FC-1000
   - FC-4096
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-266
   - Conv-266
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image

2. Small Dataset (c classes)
   - FC-C
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-266
   - Conv-266
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image
   - Reinitialize this and train
   - Freeze these

3. Bigger dataset
   - FC-C
   - FC-4096
   - MaxPool
   - Conv-512
   - Conv-512
   - MaxPool
   - Conv-266
   - Conv-266
   - MaxPool
   - Conv-128
   - Conv-128
   - MaxPool
   - Conv-64
   - Conv-64
   - Image
   - Train these
   - With bigger dataset, train more layers
   - Freeze these

   Lower learning rate when fine-tuning; 1/10 of original LR is good starting point
<table>
<thead>
<tr>
<th>More specific</th>
<th>More generic</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FC-1000</strong></td>
<td><strong>Image</strong></td>
</tr>
<tr>
<td><strong>FC-4096</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>MaxPool</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
<tr>
<td><strong>Conv-512</strong></td>
<td><strong>Conv-64</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>very similar dataset</th>
<th>very different dataset</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>very little data</strong></td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>very similar dataset</td>
<td>very different dataset</td>
</tr>
<tr>
<td>----------------------</td>
<td>--------------------------------------------------------------------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td><strong>very little data</strong></td>
<td>Use Linear Classifier on top layer</td>
<td>?</td>
</tr>
<tr>
<td><strong>quite a lot of data</strong></td>
<td>Finetune a few layers</td>
<td>?</td>
</tr>
<tr>
<td>More specific</td>
<td>More generic</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td></td>
</tr>
<tr>
<td><strong>very little data</strong></td>
<td><strong>quite a lot of data</strong></td>
<td><strong>very similar dataset</strong></td>
</tr>
<tr>
<td>Use Linear Classifier on top layer</td>
<td>Finetune a few layers</td>
<td>Finetune a larger number of layers</td>
</tr>
</tbody>
</table>
Transfer learning with CNNs is pervasive…
(it’s the norm, not an exception)

Object Detection
(Fast R-CNN)

Image Captioning: CNN + RNN

Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Transfer learning with CNNs is pervasive... (it’s the norm, not an exception)

Object Detection (Fast R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

Girshick, "Fast R-CNN", ICCV 2015
Figure copyright Ross Girshick, 2015. Reproduced with permission.

Figure copyright IEEE, 2015. Reproduced for educational purposes.
Transfer learning with CNNs is pervasive... (it’s the norm, not an exception)

Object Detection (Fast R-CNN)

CNN pretrained on ImageNet

Image Captioning: CNN + RNN

Word vectors pretrained with word2vec
Takeaway for your projects and beyond:
Have some dataset of interest but it has < ~1M images?

1. Find a very large dataset that has similar data, train a big ConvNet there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

Caffe: https://github.com/BVLC/caffe/wiki/Model-Zoo
TensorFlow: https://github.com/tensorflow/models
PyTorch: https://github.com/pytorch/vision
Questions?