Training ConvNets
How do you actually train these things?

Roughly speaking:

- Gather labeled data
- Find a ConvNet architecture
- Minimize the loss
Microsoft COCO dataset
Tsung-Yi Lin
Cornell Tech

Michael Maire
TTI Chicago

Serge Belongie
Cornell Tech

Lubomir Bourdev
Facebook

James Hays
Brown University

Pietro Perona
Caltech

Deva Ramanan
UC Irvine

Ross Girshick
Microsoft Research

Piotr Dollar
Microsoft Research

Larry Zitnick
Microsoft Research

http://mscoco.org
✓ Instance segmentation
✓ Non-iconic Images
Iconic object images
Iconic scene images
Non-iconic images
Object categories

- Person & Accessory
- Animal
- Vehicle
- Outdoor Obj.
- Sports
- Kitchenware

- Food
- Furniture
- Appliance
- Electronics
- Indoor objects
flickr
(All creative commons)

330,000 images
“Dog”
“Dog + Car”

Im2Text: Describing Images Using 1 Million Captioned Photographs, V. Ordonez, G. Kulkarni, T. L. Berg  NIPS’11

http://mscoco.org
Annotation pipeline
Amazon Mechanical Turk

Get Started with Amazon Mechanical Turk

Create Tasks
Human intelligence through an API. Access a global, on-demand, 24/7 workforce.

Make Money
Make money in your spare time. Get paid for completing simple tasks.

Create a Requester account
Create a Worker account
In the following you will have to listen to instructions. Please put on your headphones or turn on your speakers. Next, play the audio file below and adjust the volume to a level that is comfortable to you. You can listen to the audio file as often as you want to make volume adjustments. When the volume is at a comfortable level, please proceed.
Amazon Mechanical Turk
Divide and Conquer

1. Category Labeling
2. Instance spotting
3. Instance segmentation

dog, bottle

http://mscoco.org
1. Category Labeling

Image 4:

Task: select **person** and **accessory** items shown in the image (if any):
2. Instance Spotting

0 car(s) found in this image.

car
3. Instance Segmentation

http://opensurfaces.cs.cornell.edu/

Sean Bell, Paul Upchurch, Noah Snavely, Kavita Bala, Cornell University.

http://mscoco.org
Properties
Number of categories vs. number of instances

Instances per category vs. Number of categories

Segmentations

Caltech Ped
PASCAL VOC
COCO
ImageNet Detection
Caltech 256
Caltech 101
SUN
ImageNet Classification
• 330,000 images
• >2 million instances (700k people)
• Every instance is segmented
• 7.7 instances per image (3.5 categories)
## Detection Performance

( DPM V5 )

<table>
<thead>
<tr>
<th></th>
<th>Person (mAP)</th>
<th>Average (mAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PASCAL VOC</td>
<td>41.3</td>
<td>29.6</td>
</tr>
<tr>
<td>MS COCO</td>
<td>17.5</td>
<td>16.9</td>
</tr>
</tbody>
</table>

[http://mscoco.org](http://mscoco.org)
Beyond detection
Beyond detection

✓ Sentences

two giraffe standing next to each other in front of a wooden fence.
two giraffes standing in the dirt near a gate.
two giraffes stand by a food box awaiting the goods.
two giraffes are standing next to a wooden fence.
two giraffes standing alone by a picket fence.

Collecting Image Annotations Using Amazon’s Mechanical Turk,
C. Rashtchian, P. Young, M. Hodosh, J. Hockenmaier, NAACL HLT
Workshop on Creating Speech and Language Data with Amazon’s
Mechanical Turk, 2010

http://mscoco.org
Beyond detection

✓ Keypoints
  (provided by Facebook)

http://mscoco.org
Beyond detection

✓ Attributes

<table>
<thead>
<tr>
<th>dog</th>
<th>giraffe</th>
<th>person</th>
<th>dog</th>
</tr>
</thead>
<tbody>
<tr>
<td>jumping, catching, happy, exercising</td>
<td>eating, grazing, bending, peaceful, spotted, wild</td>
<td>traveling, bending, riding, moving</td>
<td>thinking, leaning, smelling, sniffing, watching, tame, loving, curious, family-friendly</td>
</tr>
<tr>
<td>floating, enjoying, hairy, playing, athletic, socializing</td>
<td>competitive</td>
<td>driving, adult, athletic, male</td>
<td>public</td>
</tr>
</tbody>
</table>

Genevieve Patterson, James Hays.  
COCO Attributes: Attributes for People, Animals, and Objects. ECCV 2016.

http://mscoco.org
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
(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)
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

Figure: Andrej Karpathy
(1) **Data preprocessing**

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

For ConvNets, 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 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
(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 and this is not optimal — 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

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
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]
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

$$||W||_2^2 \leq c$$
(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

loss goes up or goes down, good. (sanity check)
(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, 1e3)  # crank up regularization
print loss
```

loss went up, good. (sanity check)
(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:

‘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

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>1 / 200: cost 2.302603, train: 0.400000, val: 0.400000, lr 1.000000e-03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finished epoch</td>
<td>2 / 200: cost 2.302258, train: 0.450000, val: 0.450000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>3 / 200: cost 2.301849, train: 0.600000, val: 0.600000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>4 / 200: cost 2.301196, train: 0.650000, val: 0.650000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>5 / 200: cost 2.300044, train: 0.650000, val: 0.650000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>6 / 200: cost 2.297864, train: 0.550000, val: 0.550000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>7 / 200: cost 2.293595, train: 0.600000, val: 0.600000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>8 / 200: cost 2.285096, train: 0.550000, val: 0.550000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>9 / 200: cost 2.268094, train: 0.550000, val: 0.550000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>10 / 200: cost 2.234787, train: 0.500000, val: 0.500000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>11 / 200: cost 2.173187, train: 0.500000, val: 0.500000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>12 / 200: cost 2.076862, train: 0.500000, val: 0.500000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>13 / 200: cost 1.974090, train: 0.400000, val: 0.400000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>14 / 200: cost 1.895885, train: 0.400000, val: 0.400000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>15 / 200: cost 1.820876, train: 0.450000, val: 0.450000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>16 / 200: cost 1.737430, train: 0.450000, val: 0.450000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>17 / 200: cost 1.642356, train: 0.500000, val: 0.500000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>18 / 200: cost 1.535239, train: 0.600000, val: 0.600000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>19 / 200: cost 1.421527, train: 0.600000, val: 0.600000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>20 / 200: cost 1.367760, train: 0.650000, val: 0.650000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>195 / 200: cost 0.002694, train: 1.000000, val: 1.000000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>196 / 200: cost 0.002674, train: 1.000000, val: 1.000000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>197 / 200: cost 0.002655, train: 1.000000, val: 1.000000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>198 / 200: cost 0.002635, train: 1.000000, val: 1.000000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>199 / 200: cost 0.002617, train: 1.000000, val: 1.000000, lr 1.000000e-03</td>
</tr>
<tr>
<td>Finished epoch</td>
<td>200 / 200: cost 0.002597, train: 1.000000, val: 1.000000, lr 1.000000e-03</td>
</tr>
</tbody>
</table>

finished optimization. best validation accuracy: 1.000000
(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

```
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=0.0000001,
                                 update='sgd', learning_rate_decay=1,
                                 sample_batches=True,
                                 learning_rate=1e-6, verbose=True)
```

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: small — what could go wrong?

![Diagram](#)

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

Learning rate: large — 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

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
A typical phenomenon

- Why does the learning curve look like this?

Image source: Stanford CS231n
Debugging learning curves

- Not training: Bug in update calculation?
- Error increasing: Bug in update calculation?
- Error decreasing: Not converged yet
- Slow start: Suboptimal initialization?
- Possible overfitting
- Definite overfitting

Image source: Stanford CS231n
Early stopping

- Idea: do not train a network to achieve too low training error
- Monitor validation error to decide when to stop

Figure from Deep Learning Book
(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

Possibly too linear (learning rate too small)

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

Neural network parameters
Dropout

Simple but powerful technique to reduce overfitting:

Dropout

Simple but powerful technique to reduce overfitting:

[2.5]

[2.0]

[1.5]

[1.0]

0 200000 400000 600000 800000 1000000

Number of weight updates

Classification Error %

Without dropout

With dropout

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.

(b) Keeping $pn$ fixed.

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

Expected value of a neuron $h$ with dropout:

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

```python
def predict(X):
    # enssembled 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*
Advanced optimizers

- SGD with momentum
- Adagrad
SGD with momentum

What will SGD do?
SGD with momentum

• Introduce a “momentum” variable and associated “friction” coefficient:

\[ \beta m \]

• Typically start with 0, gradually increase over time
SGD with momentum

- Introduce a “momentum” variable and associated “friction” coefficient:
  - Move faster in directions with consistent gradient
  - Avoid oscillating in directions with large but inconsistent gradients

Image source
Adagrad
Adaptive per-parameter learning rates

- Gradients of different layers have different magnitudes
- Want an automatic way to set different learning rates for different parameters
Which optimizer to use in practice?

- Adagrad better than SGD initially
  - Adam with default parameters is a popular choice, SGD+momentum may work better but requires more tuning
- However, adaptive methods may quickly plateau on the validation set or generalize more poorly
  - Use Adam first, then switch to SGD?
  - Or just stick with plain old SGD? (Wilson et al., 2017)
- All methods require careful tuning and learning rate control
Other optimization tricks

- **Adding noise to gradients**: SGD with Langevin dynamics helps to jump out of local minima (e.g., Welling and Teh, 2011)
- **Cyclical learning rates**: increase learning rate from time to time to jump out of local minima (Smith 2017)
References

Basic reading: No standard textbooks yet! Some good resources:
• https://sites.google.com/site/deeplearningsummerschool/
• http://www.deeplearningbook.org/