CSCI 497P/597P: Computer Vision

Convolutional Neural Networks and some of the practicalities that make them work
Readings

with a great deal more detail...

• http://cs231n.github.io/convolutional-networks/
Announcements

- P2 grades out
- P3 grading underway
- Midterm grades also still in process
Demo

• A hand-rolled linear classifier in pytorch.

• Takeaways:
  – compute loss = my_loss_fn(X, y, W, ...)
  – call backward()
  – W.grad now contains the gradient!
Regularization – Linear Classifiers

E.g. Suppose that we found a $W$ such that $L = 0$. Is this $W$ unique?
Regularization

E.g. Suppose that we found a $W$ such that $L = 0$. Is this $W$ unique?

No! $2W$ is also has $L = 0$!
Which do we prefer – $W$, or $2W$?
Regularization: Prefer Simpler Models
Regularization: Prefer Simpler Models
Regularization: Prefer Simpler Models

Regularization pushes against fitting the data too well so we don’t fit noise in the data
A more interesting example of non-uniqueness...
Regularization

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) \]

**Data loss**: Model predictions should match training data
Regularization

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

**Data loss**: Model predictions should match training data

**Regularization**: Prevent the model from doing *too* well on training data
Regularization

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i) + \lambda R(W) \]

- **Data loss**: Model predictions should match training data
- **Regularization**: Prevent the model from doing too well on training data

\[ \lambda \text{ = regularization strength (hyperparameter)} \]

**Simple examples**
- **L2 regularization**:  \[ R(W) = \sum_k \sum_l W_{k,l}^2 \]
- **L1 regularization**:  \[ R(W) = \sum_k \sum_l |W_{k,l}| \]
- **Elastic net (L1 + L2)**:  \[ R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \]
Neural Networks

Linear classifiers

Neural Network
Neural Networks

Neural Network

Linear classifiers

Nonlinearities!
Neural Networks

Matrix multiplications

Nonlinearities!
Convolutional Neural Networks

Neural Network

Convolutions

Nonlinearities!
Convolutional Layers

• Feature maps (“hidden layers”, “activations”, etc.) are no longer column vectors but 3D blobs:
  – Input # 256x256x3
  – Conv2d(in: 3, out:10) # 255x255x10
  – Conv2d(in: 10, out:20) # 255x255x20
  – ...

[Diagram of convolutional layers]
Convolution as a general layer

For example, if we had 6 5x5 filters, we’ll get 6 separate activation maps:

We stack these up to get a “new image” of size 28x28x6!
CNNs before they were cool: LeNet-5 [LeCun et al., 1998]

• Today’s architectures still look a lot like this!
The CNN that made them cool: AlexNet

[Krizhevsky et al. 2012]
The CNN that made them cool: AlexNet [Krizhevsky et al. 2012]

• What happened?
The CNN that made them cool: AlexNet [Krizhevsky et al. 2012]

• What changed?
  – Bigger training data: ImageNet has 14 million images and 20,000 categories.
    • (performance numbers are on a 1000-category subset)
  – GPU implementation of ConvNets
    • Train bigger, deeper networks for longer than before
  – ReLU
    • Not new in AlexNet, but a necessary design choice to avoid vanishing gradients in deep network

• Hence “deep learning”:
  – a rebranding of formerly unfashionable neural networks
What do all these feature maps mean?

The filters:

Some image patches that have high activations on those filters:

Visualizations from
[M.D. Zeiler and R. Fergus: Visualizing and Understanding Convolutional Networks, ECCV 2014]
What do all these feature maps mean?

The filters, “deconvolved” back into pixel space (see the paper):

Some image patches that have high activations on those filters:

[M.D. Zeiler and R. Fergus: Visualizing and Understanding Convolutional Networks, ECCV 2014]
What do all these feature maps mean?

The filters, “deconvolved” back into pixel space (see the paper):

Some image patches that have high activations on those filters:

[M.D. Zeiler and R. Fergus: Visualizing and Understanding Convolutional Networks, ECCV 2014]
What do all these feature maps *mean*?

[M.D. Zeiler and R. Fergus: Visualizing and Understanding Convolutional Networks, ECCV 2014]
What do all these feature maps mean?

M.D. Zeiler and R. Fergus: Visualizing and Understanding Convolutional Networks, ECCV 2014
Another View: Visualizing AlexNet in 2D with t-SNE

Linear Classifier

(c) DeCAF\textsubscript{1}

(d) DeCAF\textsubscript{6}

How do you get this to work?

• Basic version:
  – Download the 1281167 images in ImageNet
  – Feed an image into network, compute gradient of loss wrt parameters, update parameters.
  – Repeat a few times (1.5 billion should do it)
How do you get this to work?

Mini-batch SGD

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient
Batched Training

• Stochastic gradient descent, technically:
  – Sample a single random datapoint
  – Compute the loss
  – Update the parameters
• What people actually mean when they say SGD: Minibatch Gradient Descent
  – Shuffle your dataset
  – Iterate over batches of (batch_size) images:
    • Feed the whole batch through the network
    • Compute loss and update parameters
• What size batches?
  – Whatever your GPU can push through the model at once. 16, 32, 64, 256, ...
There’s a bit more to it.

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Dropout
  – Weight initialization and batch normalization
Networks learn better on zero-centered data.

Consider what happens when the input to a neuron is always positive...

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

What can we say about the gradients on \( w \)?
Always all positive or all negative :( (this is also why you want zero-mean data!)
Preprocessing

**Step 1: Preprocess the data**

(Assume X [NxD] is data matrix, each example in a row)

In practice: Average all images in the dataset and subtract that from each input. Dividing by stdev isn’t usually done.
Data Augmentation

• When >1 million training images is not enough:
  – Randomly Flip, Scale, Crop, Rotate, Perturb brightness and color
  – Example:

```python
import torchvision.transforms as tvt
transforms = tvt.Compose([
    tvt.Resize((224,224)),
    tvt.ColorJitter(hue=.05, saturation=.05),
    tvt.RandomHorizontalFlip(),
    tvt.RandomRotation(20, resample=PIL.Image.BILINEAR)
])
```
Data Augmentation
There’s a bit more to it.

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Dropout
  – Weight initialization and batch normalization
Mini-batch SGD

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient
Momentum combines the gradient update with a direction based on the average of recent update direction.

Update on $v$ is usually something like:

$$ v = (1 - b) v + b \times dx $$
Momentum combines the gradient update with a direction based on the average of recent updates.

Update on $v$ is usually something like:

$$v = (1 - b) v + b * dx$$
There’s a bit more to it.

- Most of these things are practical heuristics that have been empirically discovered to work well:
  - Batched training
  - Preprocessing / data augmentation
  - Momentum
  - Learning rate decay
  - Weight initialization and batch normalization
  - Dropout
Learning Rate Decay (Annealing)

- Reduce learning rate as training continues.
  - Step decay:
  - Exponential decay
  - $1/t$ decay
Training CNNs

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Weight initialization and batch normalization
  – Ensembling
  – Dropout
Weight Initialization

- Q: what happens when W=constant init is used?
Weight Initialization

- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[ W = 0.01 \times \text{np.random.randn}(D,H) \]
Weight Initialization

- First idea: **Small random numbers**
  (gaussian with zero mean and 1e-2 standard deviation)

\[ W = 0.01 \times \text{np.random.randn}(D,H) \]

Works ~okay for small networks, but problems with deeper networks.
Let's look at some activation statistics.

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
Activations become zero!

What do the gradients look like?
Weight Initialization

\[ W = \frac{\text{np.random.randn(fan_in, fan_out)}}{\sqrt{2/\text{fan_in}}} \]

# fan_in = numel(input)
# fan_out = numel(output)
Proper initialization is an active area of research...

*Understanding the difficulty of training deep feedforward neural networks* by Glorot and Bengio, 2010

*Exact solutions to the nonlinear dynamics of learning in deep linear neural networks* by Saxe et al, 2013

*Random walk initialization for training very deep feedforward networks* by Sussillo and Abbott, 2014

*Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification* by He et al., 2015

*Data-dependent Initializations of Convolutional Neural Networks* by Krähenbühl et al., 2015

*All you need is a good init*, Mishkin and Matas, 2015

...
Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

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

this is a vanilla differentiable function...
Batch Normalization

“you want zero-mean unit-variance activations? just make them so.”

1. compute the empirical mean and variance independently for each dimension.

2. Normalize

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

Problem: do we necessarily want a zero-mean unit-variance input?

Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

Normalize:

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

And then allow the network to squash the range if it wants to:

\[ y(k) = \gamma(k) \hat{x}(k) + \beta(k) \]

Note, the network can learn:

\[ \gamma(k) = \sqrt{\text{Var}[x(k)]} \]
\[ \beta(k) = E[x(k)] \]

to recover the identity mapping.

- At test time, the answer shouldn’t depend on the batch:
  - Instead, use a global average (computed during training) of activation means and variances
Batch Normalization

BatchNorm2d

**CLASS** torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)

Applies Batch Normalization over a 4D input (a mini-batch of 2D inputs with additional channel dimension) as described in the paper Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.

\[
y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} \ast \gamma + \beta
\]

**TL;DR:** Using batch normalization speeds up training and makes it less sensitive to weight initialization.
Training CNNs

- Most of these things are practical heuristics that have been empirically discovered to work well:
  - Batched training
  - Preprocessing / data augmentation
  - Momentum
  - Learning rate decay
  - Weight initialization and batch normalization
  - Ensembling
  - Dropout
Model Ensembles

1. Train multiple independent models
2. At test time average their results
   (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

Why would this work?
• Using different random initializations results in training arriving at different local minima.
• Remarkable (empirical) fact: performance of each one is similar!
Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yikuan Li and Geoff Pleiss, 2017. Reproduced with permission.
Model Ensembles: Tips and Tricks

Instead of training independent models, use multiple snapshots of a single model during training!

Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Cyclic learning rate schedules can make this work even better!
Training CNNs

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Weight initialization and batch normalization
  – Ensembling
  – Dropout
Regularization: Recall

• Penalizes large weights to prevent the model from fitting training data too closely (overfitting)
  – Helps network generalize to unseen data
• L2 regularization forces parameters to be used “equally”
  – parameters with similar magnitudes will have a lower regularization cost than mostly zero with a few huge values.
• Another way to force the network to use all its parameters equally: randomly drop parameters each training iteration!
Another way to force the network to use all its parameters equally: randomly drop parameters each training iteration!

Regularization: Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common

Srivastava et al., "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014
Regularization: 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)
```

Example forward pass with a 3-layer network using dropout
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

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
Regularization: Dropout
How can this possibly be a good idea?

Another interpretation:

Dropout is training a large *ensemble* of models (that share parameters).

Each binary mask is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...
Dropout: Test time

```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
```

At test time all neurons are active always => We must scale the activations so that for each neuron: output at test time = expected output at training time
"Vanilla Dropout: Not recommended implementation (see notes below)"

\[ p = 0.5 \] # probability of keeping a unit active. higher = less dropout

```python
def train_step(X):
    
    # 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)

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 Summary**

- **Drop in forward pass**
- **Scale at test time**
More common: “Inverted dropout”

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

def train_step(X):
    # forward pass for example 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
    H1 *= U1 # drop!

    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
    H2 *= U2 # drop!

    out = np.dot(W3, H2) + b3

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

def predict(X):
    # ensembled forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
```

test time is unchanged!
Training CNNs

• Most of these things are practical heuristics that have been empirically discovered to work well:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Weight initialization and batch normalization
  – Ensembling
  – Dropout
Next Up: CNN Architecture Tour

• What happened since AlexNet?
• There’s a general theme:

WE NEED TO GO DEEPER