CSCI 497P/597P: Computer Vision

Regularization
CNNs: Interpretation, Practicalities
Readings

with a great deal more detail...

• https://cs231n.github.io/neural-networks-2/
• https://cs231n.github.io/convolutional-networks/
Goals (Last Lecture)

• Understand why we need activation functions.
• Understand the motivation and behavior of convolutional layers in neural networks.
• Understand the degrees of freedom available in setting up a convolution layer:
  – Output channels, kernel size, padding, stride
• Know the meaning of the various basic layers involved in standard CNN architectures
  – Conv, ReLU, Pool, Fully Connected
Goals (Today)

• Understand the purpose of applying regularization in machine learning training.
• Know what overfitting means and why it’s bad.
• Gain intuition for the meaning of intermediate layers in CNNs.
• Know the idea and purpose of each of the following tricks used when training CNNs:
  – Batched training
  – Preprocessing / data augmentation
  – Momentum
  – Learning rate decay
  – Dropout
  – Weight initialization and batch normalization
Announcements

• HW5 still doesn’t exist. It will be short, optional, or both, with the goal of helping you prepare for the final.
• P2 grading is in progress.
Announcements

• P4 is out. You will:
  – Modify a trained 1000-class classifier to turn it into a 2-class “dog vs food” classifier.
  – Misuse backpropagation to:
    • see which input pixels are most influential in classifying it
    • trick the classifier into predicting the wrong class
    • synthesize images that maximize a chosen class score
Announcements

• P4 is out. You will:
  – Modify a trained 1000-class classifier to turn it into a 2-class “dog vs food” classifier.
  – Misuse backpropagation to:
    • see which input pixels are most influential in classifying it
    • trick the classifier into predicting the wrong class
    • synthesize images that maximize a chosen class score
Announcements

– Misuse backpropagation to:
  • see which input pixels are most influential (saliency)
  • trick the classifier into predicting the wrong class
  • synthesize images that maximize a chosen class score
Announcements

– Misuse backpropagation to:
  • see which input pixels are most influential in classifying it
  • **trick the classifier into predicting the wrong class**
  • synthesize images that maximize a chosen class score

---

![Images of dogs and snails with differences shown]

dog  snail  difference
Announcements

– Misuse backpropagation to:
  • see which input pixels are most influential in classifying it
  • trick the classifier into predicting the wrong class
  • synthesize images that maximize a chosen class score
Machine Learning Aside: Regularization

• Suppose we’ve learned a linear classifier $W$ such that $L = 0$: it classifies everything perfectly.

• Is this $W$ unique?
Regularization

• Suppose we’ve learned a linear classifier $W$ such that $L = 0$: it classifies everything perfectly.

• Is this $W$ unique?

No! $2W$ is also has $L = 0$!
Which do we prefer – $W$, or $2W$?

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
Regularization: Prefer Simpler Models
Regularization: Prefer Simpler Models
overfitting: learning the training data too well, to the detriment of performance on unseen 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

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
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

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)**: \( R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}| \)

\( \lambda \) = regularization strength (hyperparameter)
Regularization in CNNs

• AKA “weight decay”
Convolutional Neural Networks

Neural Network

Convolutions

Nonlinearities!
Consider a second, green filter.

- **Convolution Layer**
  - 32x32x3 image
  - 5x5x3 filter

  **Activation Maps**
  - convolution (slide) over all spatial locations
  - 28x28x1
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:

[Layer 3]

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

Layer 4

Layer 5

[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

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)
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
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.

**Before normalization**: classification loss very sensitive to changes in weight matrix; hard to optimize

**After normalization**: less sensitive to small changes in weights; easier to optimize
Step 1: Preprocess the data

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

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

# Vanilla update
```python
x += - learning_rate * dx
```

# Momentum update
```python
v = mu * v - learning_rate * dx # integrate velocity
x += v # integrate position
```

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 * dx
```
Momentum combines the gradient update with a direction based on the average of recent update directions.

Update on $v$ is usually something like:

$$v = (1 - b) \ v + b \ * \ dx$$
30 Second Red Panda Break
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
Question for you

- 2 questions on Socrative: How do convolution layers work?
- Breakout groups of 3
  - socrative.com
  - room name: CSCI497P
Convolution Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map
Convolution Layer

Consider a second, green filter

32x32x3 image
5x5x3 filter

Convolve (slide) over all spatial locations

Activation maps

32 32
3 3

1 28
28
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 \sim okay for small networks, but problems with deeper networks.

Slide: Fei-Fei Li, Justin Johnson, & Serena Yeung
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.

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['relu'] * len(hidden_layer_sizes)

act = {'relu': lambda x: np.maximum(0, x), 'tanh': lambda x: np.tanh(x)}
Hs = []
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1]  # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01  # layer initialization
    H = np.dot(X, W)  # matrix multiply
    H = act[nonlinearities[i]](H)  # nonlinearity
    Hs[i] = H  # cache result on this layer

# look at distributions at each layer
print '{:14} : mean {:.3f} std {:.3f}'.format('input layer', np.mean(D), np.std(D))
layer_means = [np.mean(H) for i, H in Hs.iteritems()]
layer_stds = [np.std(H) for i, H in Hs.iteritems()]
for i, H in Hs.iteritems():
    print '{:14} : mean {:.3f} std {:.3f}'.format('layer %d' % (i+1), layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i, H in Hs.iteritems():
    plt.subplot(1, len(Hs), i+1)
    plt.hist(H.ravel(), 30, range=(-1, 1))
```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213801
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 -0.000000 and std 0.002378
hidden layer 5 had mean -0.000000 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

\[ W = \frac{\text{np.random.randn}(\text{fan\_in}, \text{fan\_out})}{\sqrt{2/\text{fan\_in}}} \]

# \text{fan\_in} = \text{numel}(\text{input})
# \text{fan\_out} = \text{numel}(\text{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

*Fixup Initialization: Residual Learning Without Normalization*, Zhang et al, 2019

*The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks*, Frankle and Carbin, 2019

...
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
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.

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

2. Normalize
Batch Normalization

[ioffe and Szegedy, 2015]

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

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

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

Normalize:

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

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:

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

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

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: Reminder

• 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

Regularization: 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)
```

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)

```python
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)

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