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

with a great deal more detail...

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

• 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 out; due Monday 11/30. Lowest HW grade is dropped.
Convolutional Neural Networks

Neural Network

Convolutions

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

- What happened?
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
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:

$$v = (1 - b)v + b \cdot \Delta x$$
Momentum combines the gradient update with a direction based on the average of recent update directions. 

Update on $v$ is usually something like:

$$v = (1 - b) v + b * dx$$
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)

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

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

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

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

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

…
Question for you

• The input to a network is a 3-channel RGB image. The first layer of the network is a convolution layer. This layer learns 8 filters, each of which is 3x3. How many parameters (weights) need to be learned for this layer?
  – A: 9
  – B: 72
  – C: 216
  – D: Depends on the input image dimensions

27 weights \cdot 8
Convolution Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map
The input to a network is a 3-channel RGB image. The first layer of the network is a convolution layer. This layer learns 8 filters, each of which is 3x3. What is the channel dimension of the output feature map?

- A: 1
- B: 3
- C: 8
- D: 24
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.

2. Normalize

\[
\hat{x}(k) = \frac{x(k) - E[x(k)]}{\sqrt{\text{Var}[x(k)]}}
\]
Problem: do we necessarily want a zero-mean unit-variance input?

Batch Normalization

[ioffe and Szegedy, 2015]

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

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

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.

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

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

[SOURCE]
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!

Cyclic learning rate schedules can make this work even better!

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

---

Srivastava et al., "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014
Regularization: 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)
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
# probability of keeping a unit active. higher = less dropout
p = 0.5

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