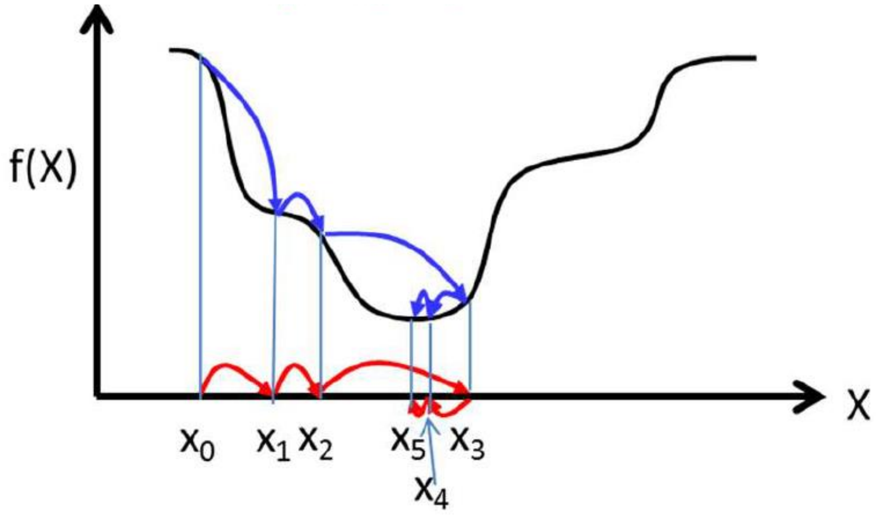
Lecture 8: CNNs III – Model Training and Regularization

Xuming He
SIST, ShanghaiTech
Fall, 2019

Training overview

- Supervised learning paradigm
- Mini-batch SGD

  Loop:
  - Sample a (mini-)batch of data
  - Forward propagation it through the network, compute loss
  - Backpropagation to calculate the gradients
  - Update the parameters using the gradient

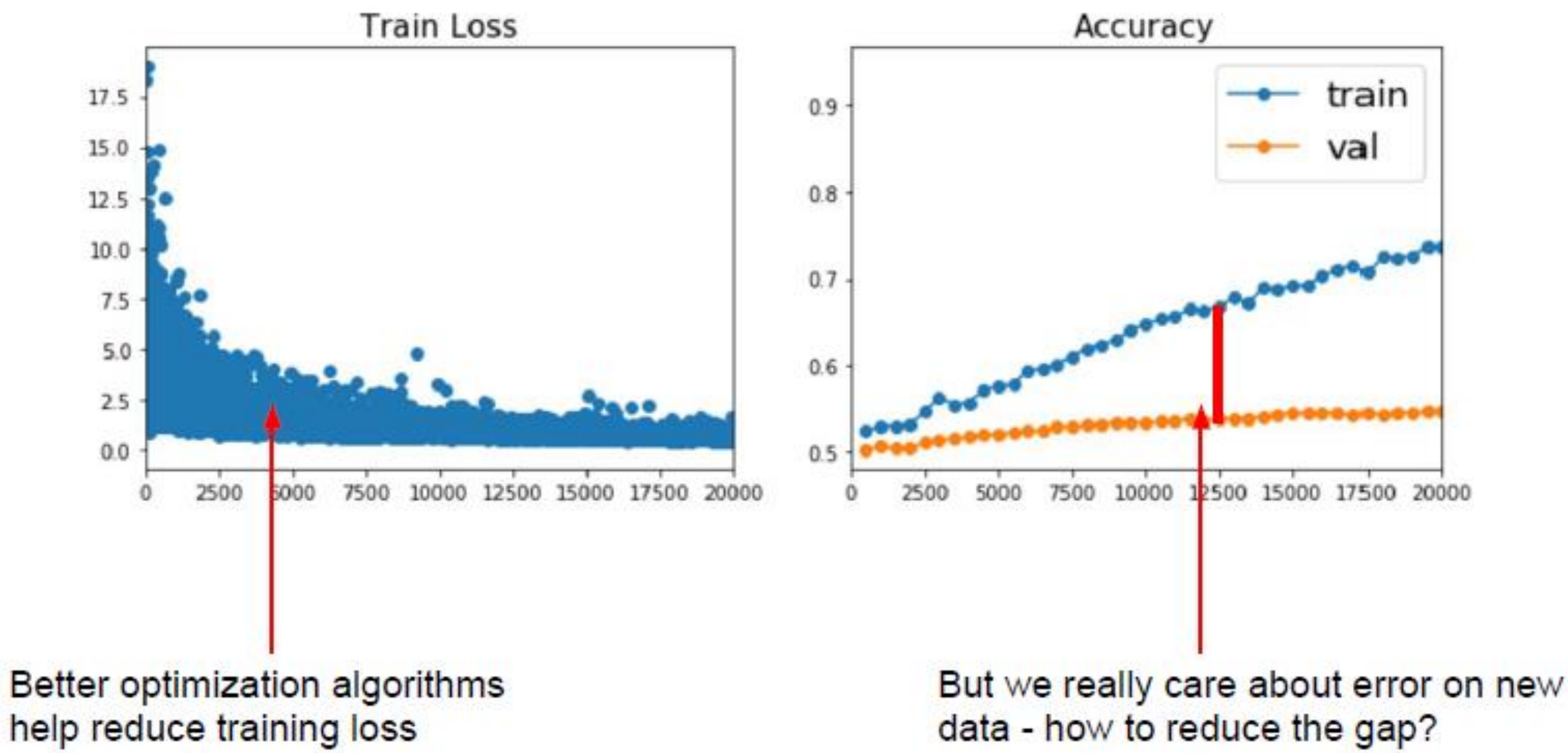


Training overview

- Two aspects of training networks
  - Optimization
    - How do we minimize the loss function effectively?
  - Generalization
    - How do we avoid overfitting?
- Optimization – this lecture
  - Data pre-processing, weight initialization, parameter updates, batch normalization
- Generalization – next lecture
  - Data augmentation, dropout, model ensembles, hyper-parameter optimization

Beyond Training Error

- How do we generalize to unseen data?
  - Well studied but still poorly understood



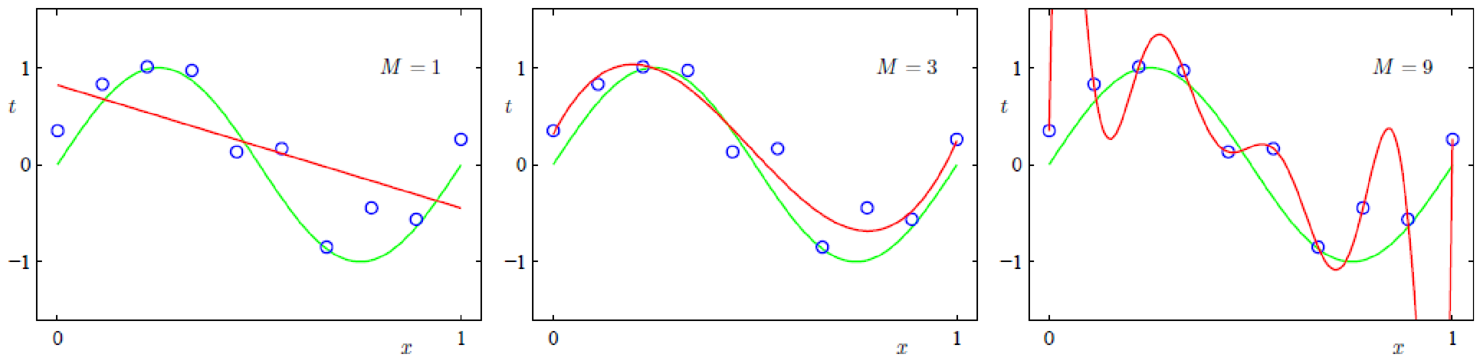
Outline

- Overview of generalization
- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization
  - Stochastic Regularization
  - Model Ensembles
  - Transfer Learning
- Basic network training in practice
  - CNN learning pipeline
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Generalization

- Recall: overfitting and underfitting



- Deep network learning
  - Avoid overfitting due to large # of parameters

Some Theoretic Guidance

- Bias/Variance decomposition
  - For squared error loss
  - Want to minimize the expected loss

$$\mathbb{E}_{p_{\mathcal{D}}}[(y-t)^2 \mid \mathbf{x}]$$

  - Best prediction is $y_\star = \mathbb{E}_{p_{\mathcal{D}}}[t \mid \mathbf{x}]$

  - Derivation:

$$\begin{aligned}
\mathbb{E}[(y-t)^2 \mid \mathbf{x}] &= \mathbb{E}[y^2 - 2yt + t^2 \mid \mathbf{x}] \\
&= y^2 - 2y\mathbb{E}[t \mid \mathbf{x}] + \mathbb{E}[t^2 \mid \mathbf{x}] \\
&= y^2 - 2y\mathbb{E}[t \mid \mathbf{x}] + \mathbb{E}[t \mid \mathbf{x}]^2 + \mathrm{Var}[t \mid \mathbf{x}] \\
&= (y - y_\star)^2 + \mathrm{Var}[t \mid \mathbf{x}]
\end{aligned}$$

  - The term Var[t|x], called Bayes error, is the best achievable risk.

Some Theoretic Guidance

- Bias/Variance decomposition
  - For squared error loss
  - Given a training set and train a model, which predicts y from x
  - Aim to minimize the expected loss $\mathbb{E}[(y-t)^2]$
  - We can decompose it into bias, variance and Bayes error (suppress the conditioning on x for clarity)

$$
\begin{aligned}
\mathbb{E}[(y-t)^2] &= \mathbb{E}[(y-y_\star)^2] + \mathrm{Var}(t) \\
&= \mathbb{E}[y_\star^2 - 2y_\star y + y^2] + \mathrm{Var}(t) \\
&= y_\star^2 - 2y_\star \mathbb{E}[y] + \mathbb{E}[y^2] + \mathrm{Var}(t) \\
&= y_\star^2 - 2y_\star \mathbb{E}[y] + \mathbb{E}[y]^2 + \mathrm{Var}(y) + \mathrm{Var}(t) \\
&= \underbrace{(y_\star - \mathbb{E}[y])^2}_{\text{bias}} + \underbrace{\mathrm{Var}(y)}_{\text{variance}} + \underbrace{\mathrm{Var}(t)}_{\text{Bayes error}}
\end{aligned}
$$

Bag of Tricks

- How can we train a model that's complex enough to model the structure in the data, but prevent it from overfitting?
  - How to achieve low bias and low variance?
- Our bag of tricks
  - Data augmentation
  - Reduce the number of parameters
  - Weight decay
  - Early stopping
  - Ensembles (combine predictions of different models)
  - Stochastic regularization (e.g. dropout)
- The best-performing models on most benchmarks use some or all of these tricks

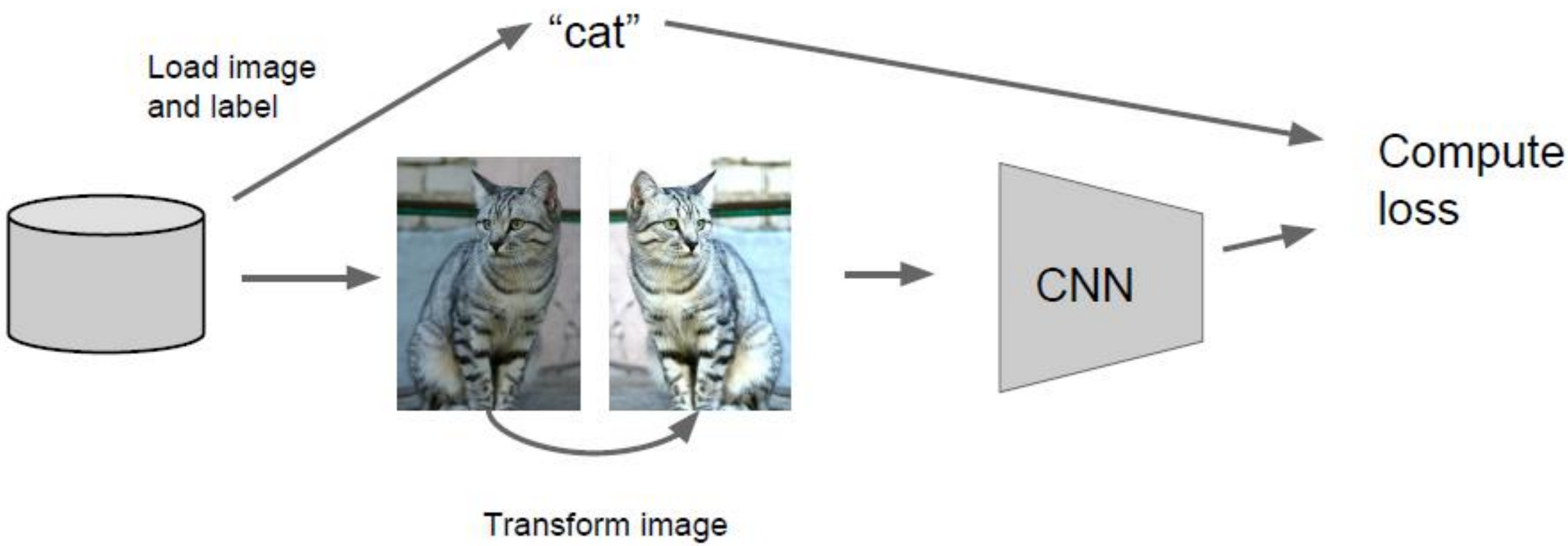
Outline

- Overview of generalization
- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization
  - Stochastic Regularization
  - Model Ensembles
  - Transfer Learning
- Basic network training in practice
  - CNN learning pipeline
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Data Augmentation

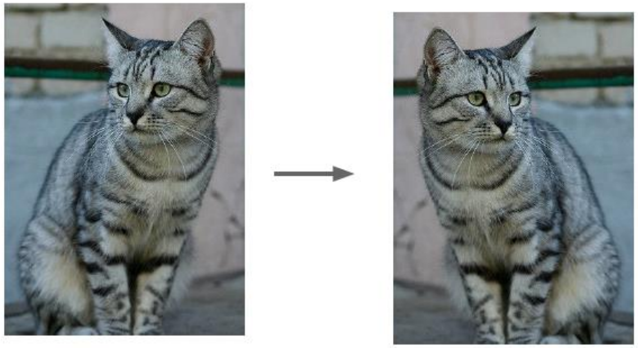
- Create more data for regularization



Data Augmentation

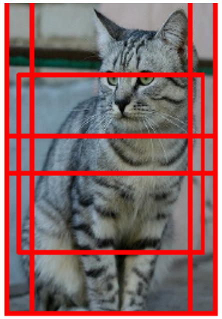
- Create more data for regularization

Horizontal Flips



Random crops and scales

Training: sample random crops / scales
ResNet:
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



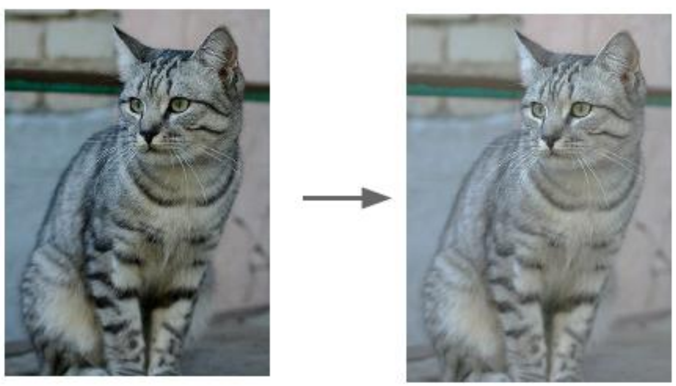
Testing: average a fixed set of crops
ResNet:
1. Resize image at 5 scales: {224, 256, 384, 480, 640}
2. For each size, use 10 224 x 224 crops: 4 corners + center, + flips

Data Augmentation

- Create more data for regularization

Color Jitter

Simple: Randomize contrast and brightness



More Complex:

1. Apply PCA to all [R, G, B] pixels in training set
2. Sample a “color offset” along principal component directions
3. Add offset to all pixels of a training image

(As seen in *[Krizhevsky et al. 2012]*, ResNet, etc)

Data Augmentation

- Create more data for regularization
- Examples (for visual recognition)
  - translation
  - horizontal or vertical
  - flip
  - rotation
  - smooth warping
  - noise (e.g. flip random pixels)
- The choice of transformations depends on the task.
  - E.g. horizontal flip for object recognition, but not handwritten digit recognition.

Data Augmentation

- AutoAugment (Cubuk et al, Arxiv 2018)
  - An automatic way to design custom data augmentation policies for computer vision datasets,
  - Selecting an optimal policy from a search space of 2.9×10^{32} image transformation possibilities.
    - E.g., guiding the selection of basic image transformation operations, such as flipping an image horizontally/vertically, rotating an image, changing the color of an image, etc.
  - Using reinforcement learning strategy (More later…)
- Results
  - New state of the art: ImageNet: 83.54% top1 accuracy; SVHN: error rate 1.02%.
  - AutoAugment policies are found to be transferable to other vision datasets.

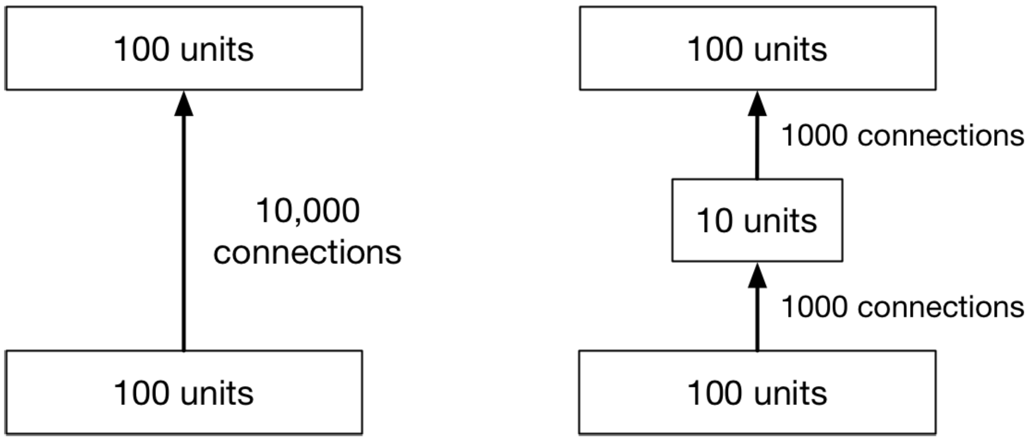
Outline

- Overview of generalization
- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization
  - Stochastic Regularization
  - Model Ensembles
  - Transfer Learning
- Basic network training in practice
  - CNN learning pipeline
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Reducing # of Parameters

- Reducing the number of layers or the number of parameters per layer.
- Adding a linear bottleneck layer:



  - The first network is strictly more expressive than the second (i.e. it can represent a strictly larger class of functions). (Why?)
  - Remember how linear layers don't make a network more expressive? They might still improve generalization.

Weight Regularization

- Basic form

Regularization: Add term to loss

$$L = \frac{1}{N}\sum_{i=1}^{N} \boxed{loss(f_{net}(x_i, W), y_i)} + \boxed{\lambda R(W)}$$

In common use:

L2 regularization	$R(W) = \sum_k \sum_l W_{k,l}^2$	(Weight decay)
L1 regularization	$R(W) = \sum_k \sum_l \lvert W_{k,l} \rvert$	
Elastic net (L1 + L2)	$R(W) = \sum_k \sum_l \beta W_{k,l}^2 + \lvert W_{k,l} \rvert$	

Weight Regularization

- L_2 regularization / weight decay
  - Encouraging the weights to be small in magnitude

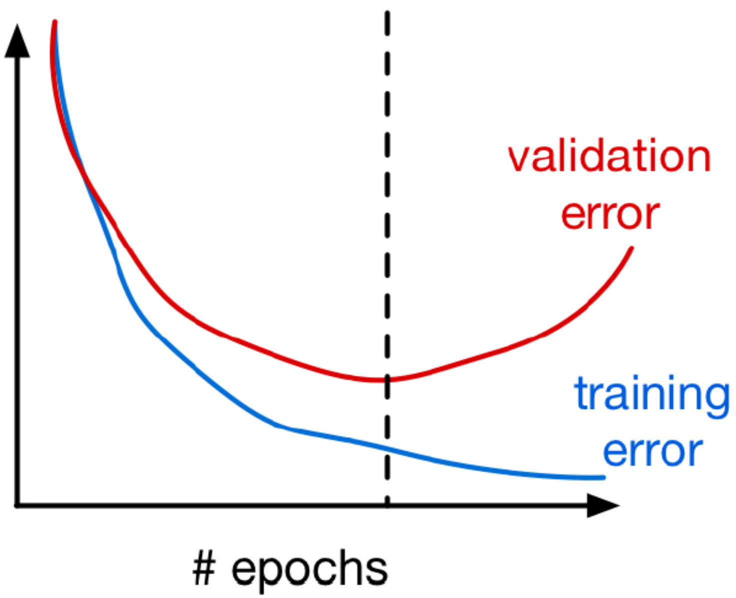
$$\mathcal{E}_{\text{reg}} = \mathcal{E} + \lambda \mathcal{R} = \mathcal{E} + \frac{\lambda}{2} \sum_j w_j^2$$

  - The gradient update can be interpreted as weight decay

$$\begin{aligned} \mathbf{w} \leftarrow \mathbf{w} - \alpha \left(\frac{\partial \mathcal{E}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\ = \mathbf{w} - \alpha \left(\frac{\partial \mathcal{E}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\ = (1 - \alpha\lambda)\mathbf{w} - \alpha \frac{\partial \mathcal{E}}{\partial \mathbf{w}} \end{aligned}$$

Early Stopping

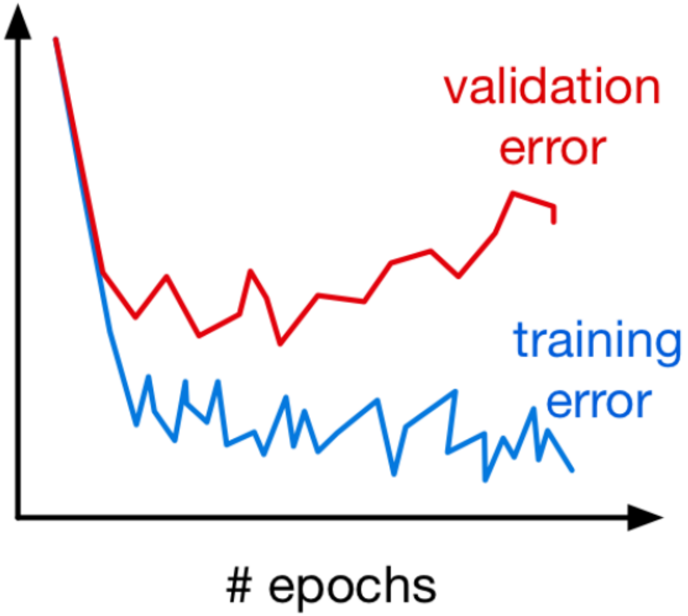
- Early stopping: monitor performance on a validation set, stop training when the validation error starts going up.
  - We don't always want to find a global (or even local) optimum of our cost function.



  - Weights start out small, so it takes time for them to grow large. Therefore, it has a similar effect to weight decay.

Early Stopping

- A slight catch: validation error fluctuates because of stochasticity in the updates.
  - Determining when the validation error has actually leveled off can be tricky.
  - May use temporal smoothing



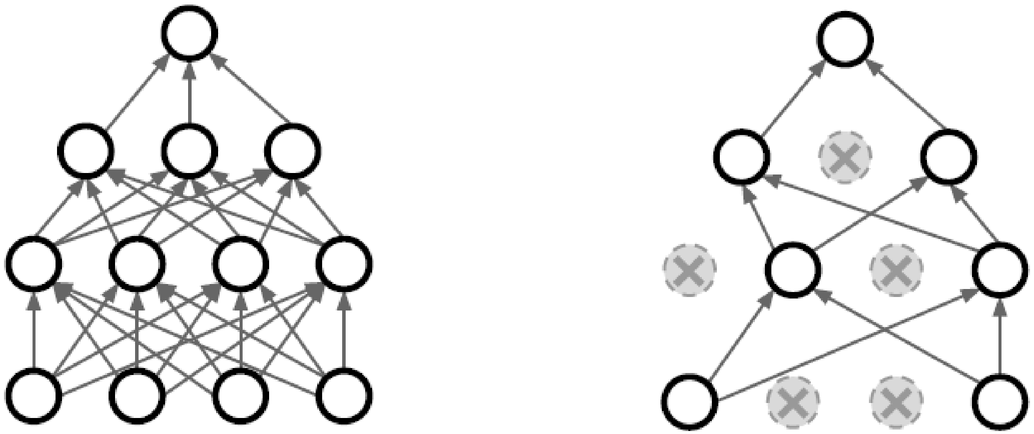
Outline

- Overview of generalization
- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization
  - Stochastic Regularization
  - Model Ensembles
  - Transfer Learning
- Basic network training in practice
  - CNN learning pipeline
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Stochastic Regularization

- For a network to overfit, its computations need to be really precise. This suggests regularizing them by injecting noise into the computations, a strategy known as stochastic regularization.
- Dropout is a stochastic regularizer which randomly deactivates a subset of the units



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

Dropout

- Operations

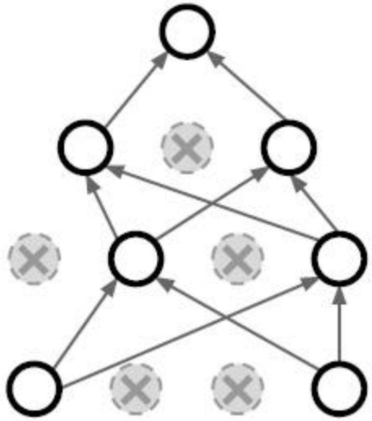
$$h_i = m_i \cdot \phi(z_i),$$

where m_i is a Bernoulli random variable, independent for each hidden unit.

Regularization: Dropout

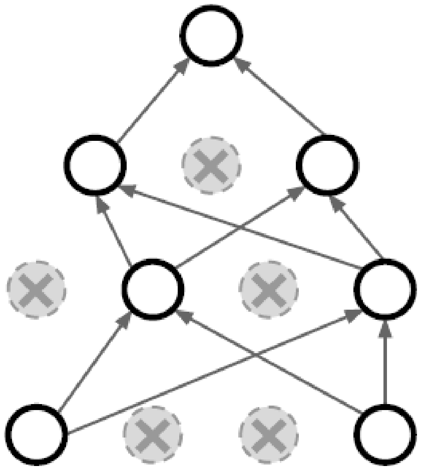
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

Example forward pass with a 3-layer network using dropout

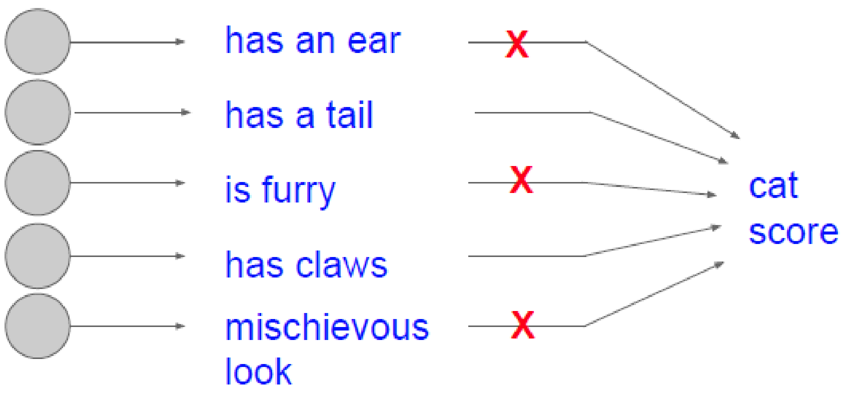


Regularization: Dropout

How can this possibly be a good idea?

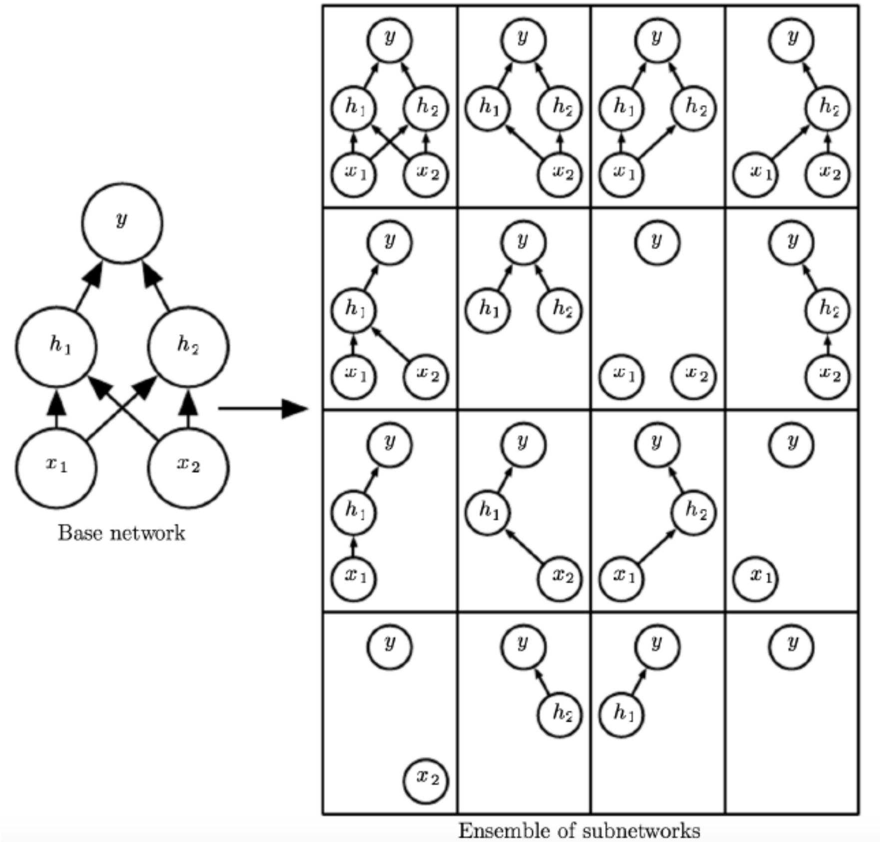


Forces the network to have a redundant representation;
Prevents co-adaptation of features



Understanding Dropout

- Dropout can be seen as training an ensemble of 2^D different architectures with shared weights (where D is the number of units):



— Goodfellow et al., *Deep Learning*

Dropout

- Dropout at test time

Dropout makes our output random!

$$\underset{\text{(label)}}{\overset{\text{Output}}{\boxed{y}}} = f_W(\underset{\text{(image)}}{\overset{\text{Input}}{\boxed{x}}}, \underset{\text{mask}}{\overset{\text{Random}}{\boxed{z}}})$$

Want to "average out" the randomness at test-time

$$y = f(x) = E_z[f(x, z)] = \int p(z) f(x, z) dz$$

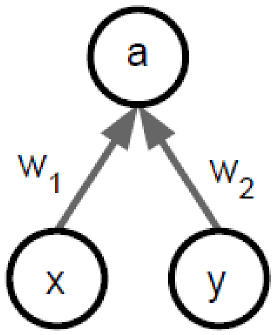
But this integral seems hard …

Dropout

- Dropout at test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x, z)\big] = \int p(z) f(x, z)\, dz$$



Consider a single neuron.

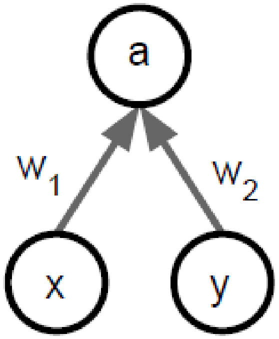
At test time we have: $E\big[a\big] = w_1 x + w_2 y$

Dropout

- Dropout at test time

Want to approximate the integral

$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$



Consider a single neuron.

At test time we have: $E\big[a\big] = w_1 x + w_2 y$

During training we have:

$$\begin{aligned} E\big[a\big] =& \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ =& \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

At test time, **multiply** by dropout probability

Dropout

- Dropout at test time

```
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:
<u>output at test time</u> = <u>expected output at training time</u>

Dropout

- Implementation: Inverted dropout

```
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!

Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - DropConnect drops connections instead of activations.
  - Batch normalization (mentioned last time for its optimization benefits) also introduces stochasticity, thereby acting as a regularizer.
  - The stochasticity in SGD updates has been observed to act as a regularizer, helping generalization.
    - Increasing the mini-batch size may improve training error at the expense of test error!

Outline

- Overview of generalization
- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization
  - Stochastic Regularization
  - Model Ensembles
  - Transfer Learning
- Basic network training in practice
  - CNN learning pipeline
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

Model Ensembles

- If a loss function is convex (with respect to the predictions)

$$\mathcal{L}(\lambda_1 y_1 + \cdots + \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \cdots + \lambda_N \mathcal{L}(y_N, t) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1$$

  - True no matter where they came from
  - Examples: squared error, cross-entropy, hinge loss
- If you have multiple candidate models and average their predictions on the test data, the set of models is called an ensemble.
  - Averaging often helps even when the loss is nonconvex

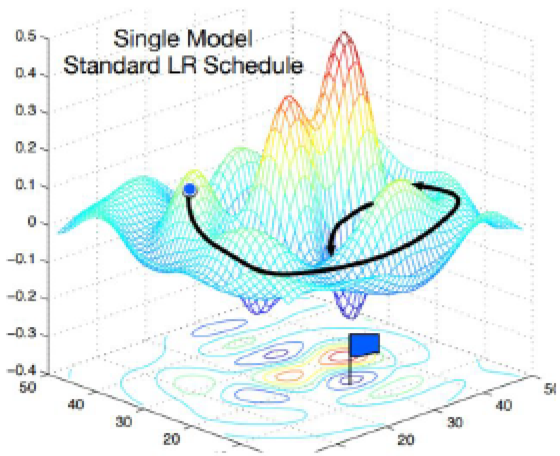
Model Ensembles

- Some examples of ensembles:
  - Train networks starting from different random initializations. But this might not give enough diversity to be useful.
  - Train networks on different subsets of the training data. This is called bagging.
  - Train networks with different architectures or hyperparameters, or even use other algorithms which aren't neural nets.
- Ensembles can improve generalization quite a bit, and the winning systems for most machine learning benchmarks are ensembles.
- But they are expensive, and the predictions can be hard to interpret.

Model Ensembles

- Tips and Tricks

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

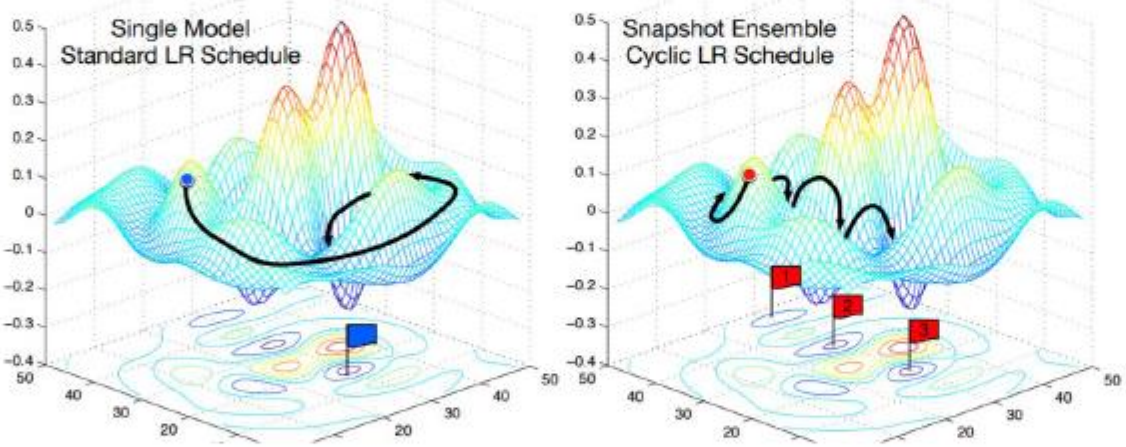


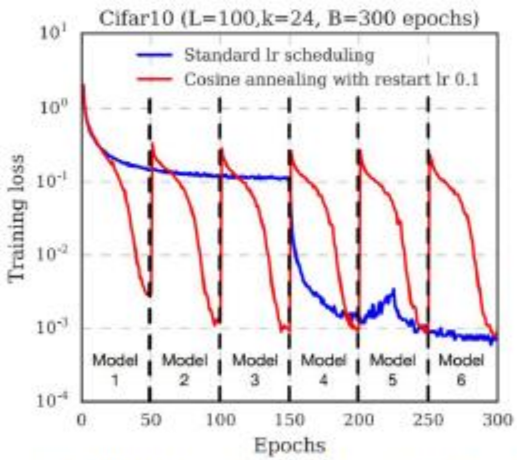
Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
Huang et al, "Snapshot ensembles: train 1, get M for free", ICLR 2017

Model Ensembles

- Tips and Tricks

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





Loshchilov and Hutter, "SGDR: Stochastic gradient descent with restarts", arXiv 2016
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!

Model Ensembles

- Tips and Tricks

Instead of using actual parameter vector, keep a moving average of the parameter vector and use that at test time (Polyak averaging)

```
while True:
  data_batch = dataset.sample_data_batch()
  loss = network.forward(data_batch)
  dx = network.backward()
  x += - learning_rate * dx
  x_test = 0.995*x_test + 0.005*x # use for test set
```

Outline

- Overview of generalization
- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization
  - Stochastic Regularization
  - Model Ensembles
  - Transfer Learning
- Basic network training in practice
  - CNN learning pipeline
  - Hyper-parameter optimization

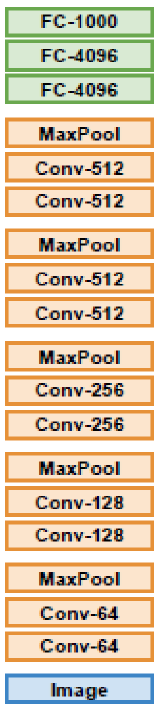
Acknowledgement: Feifei Li's cs231n notes

Transfer Learning

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs

1. Train on Imagenet



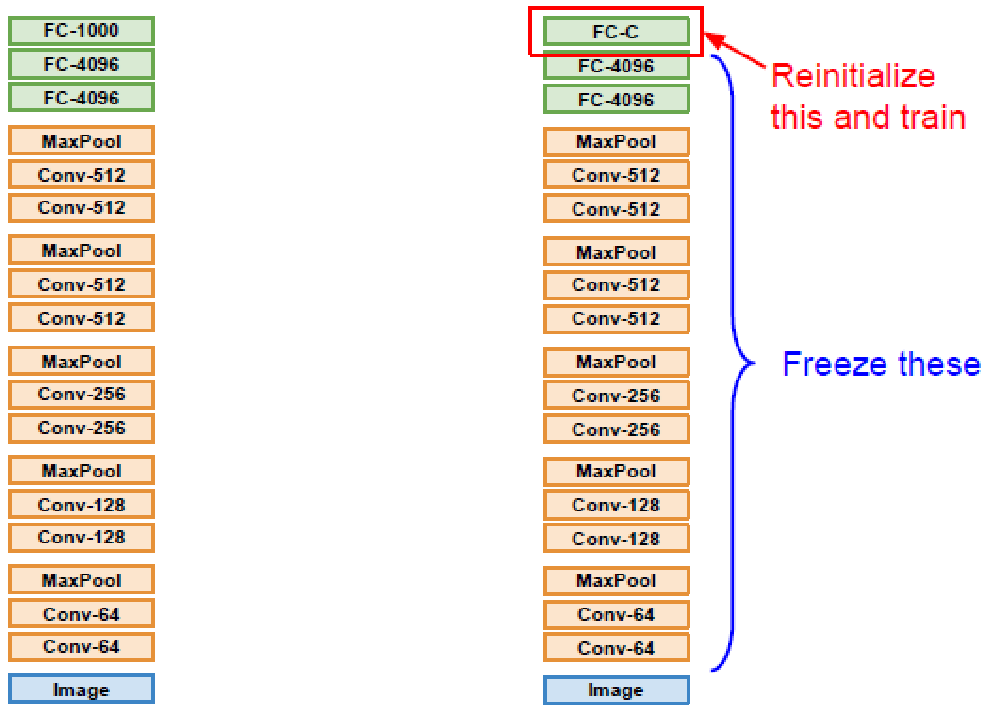
Transfer Learning

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

Transfer Learning with CNNs

1. Train on Imagenet

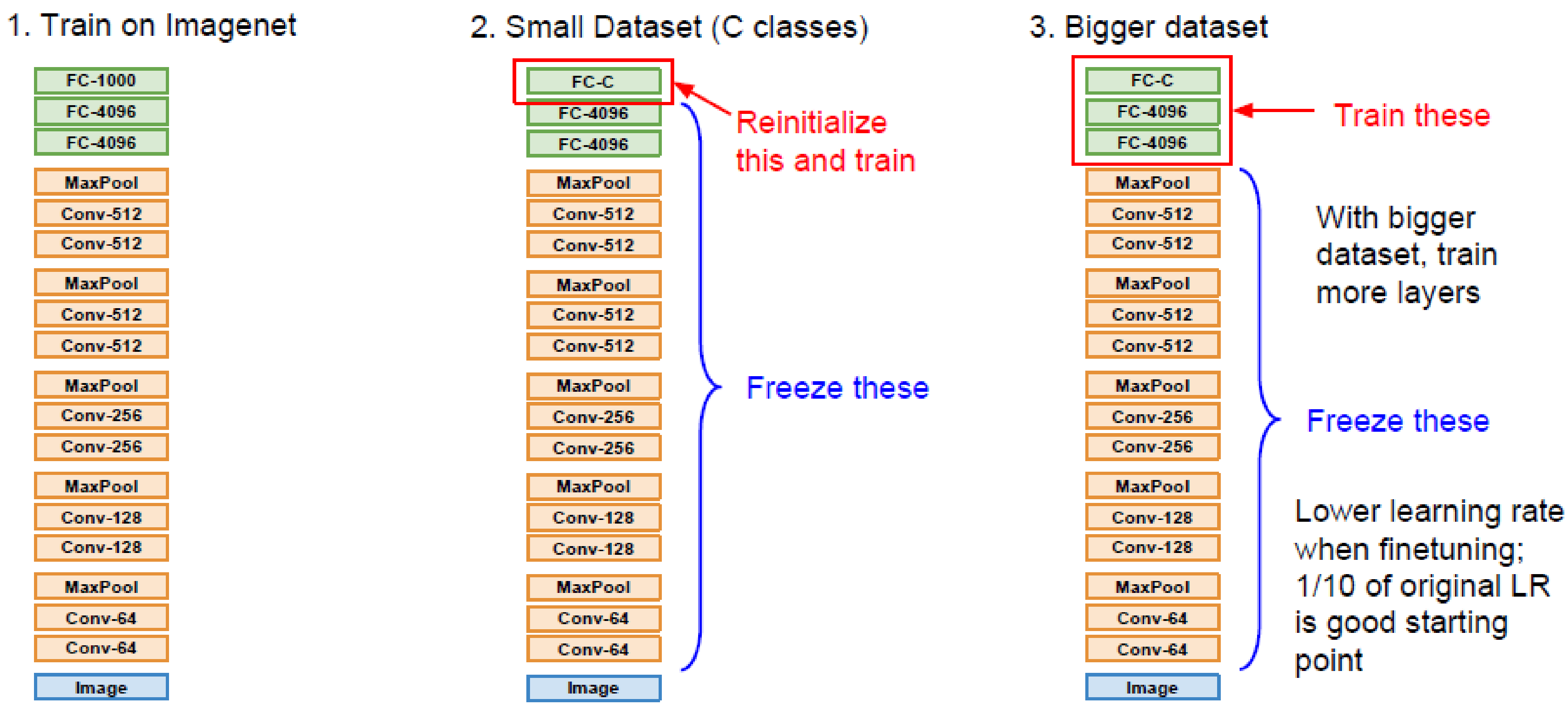
2. Small Dataset (C classes)



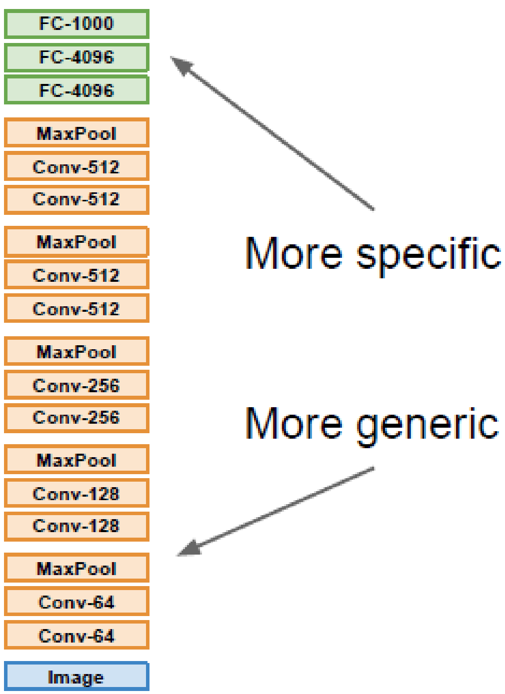
Transfer Learning

Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014



Transfer Learning

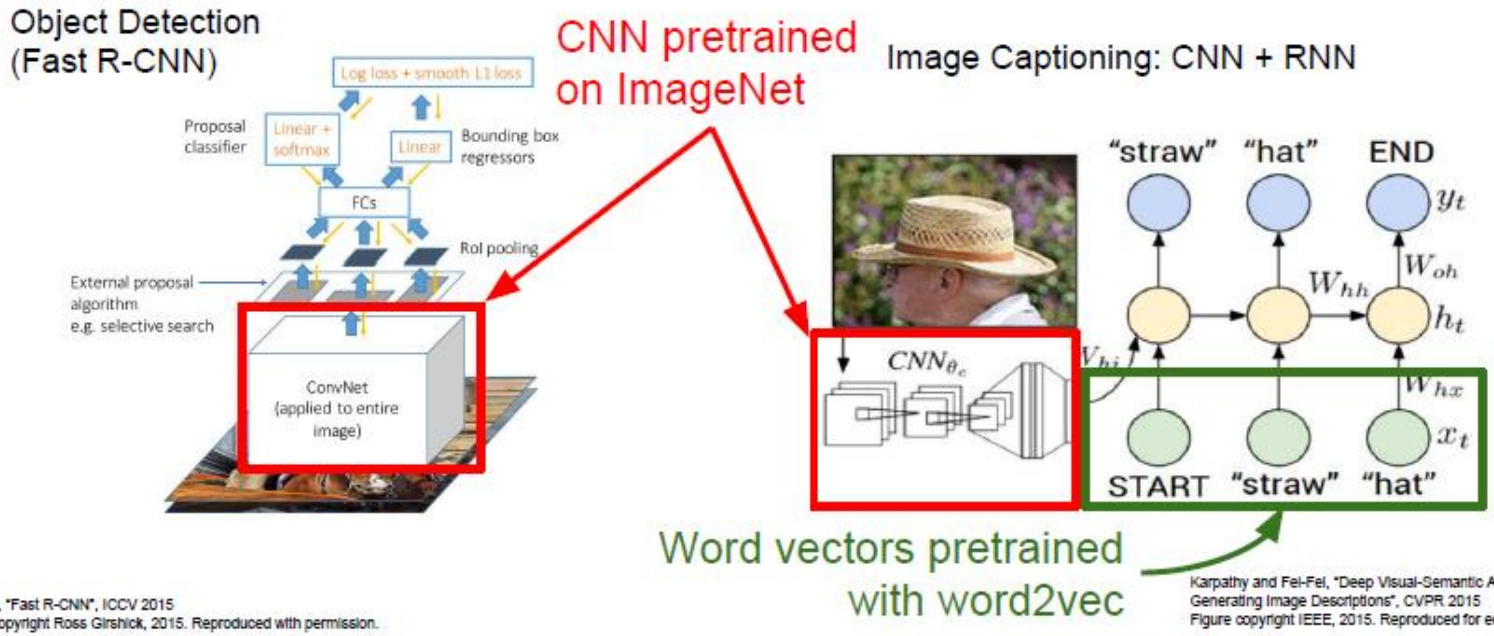


	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Transfer Learning

- Transfer learning with CNNs is pervasive

(it's the norm, not an exception)



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

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015
Figure copyright IEEE, 2015. Reproduced for educational purposes.

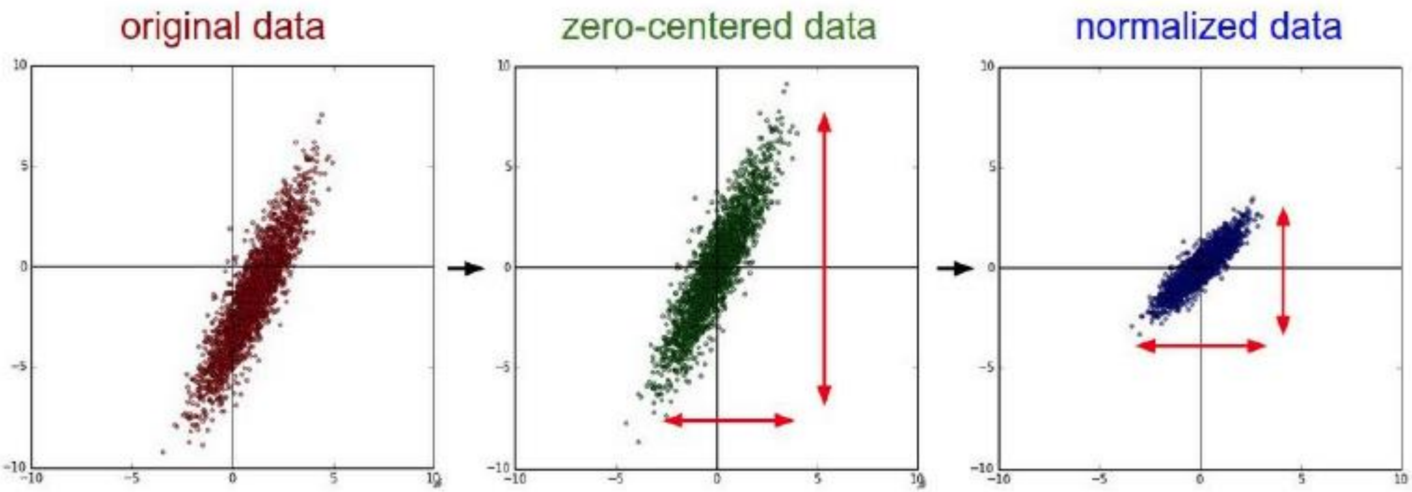
Outline

- Overview of generalization
- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization
  - Stochastic Regularization
  - Model Ensembles
  - Transfer Learning
- Basic network training in practice
  - CNN learning pipeline
  - Hyper-parameter optimization

Acknowledgement: Feifei Li's cs231n notes

CNN training pipeline: example

Step 1: Preprocess the data




```
X -= np.mean(X, axis = 0)
```

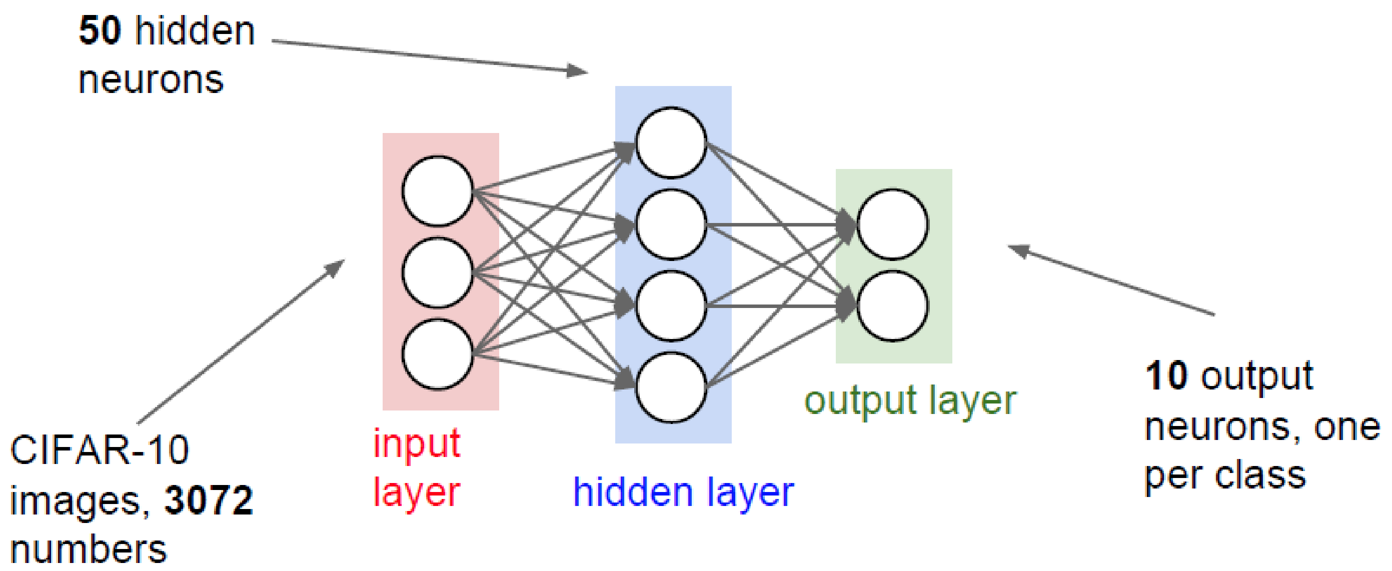
```
X /= np.std(X, axis = 0)
```

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

CNN training pipeline: example

Step 2: Choose the architecture:

say we start with one hidden layer of 50 neurons:



CNN training pipeline: example

Double 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)
print loss
```

disable regularization

```
2.30261216167
```

loss ~2.3.
“correct “ for
10 classes

returns the loss and the
gradient for all parameters

CNN training pipeline: example

Double 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)
print loss
```

crank up regularization

```
3.06859716482
```

loss went up, good. (sanity check)

CNN training pipeline: example

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training 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)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla 'sgd'

CNN training pipeline: example

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training 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)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
```

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Very small loss, train accuracy 1.00, nice!

CNN training pipeline: example

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

```
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)
```

CNN training pipeline: example

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low

```
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)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

CNN training pipeline: example

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
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 en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost always means high learning rate...

CNN training pipeline: example

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
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 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

3e-3 is still too high. Cost explodes….

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]

Hyperparameter optimization

Cross-validation strategy

coarse -> fine cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

Hyperparameter optimization

For example: run coarse search for 5 epochs

```
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)
```

note it's best to optimize in log space!

```
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)
```

nice

Hyperparameter optimization

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

adjust range

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

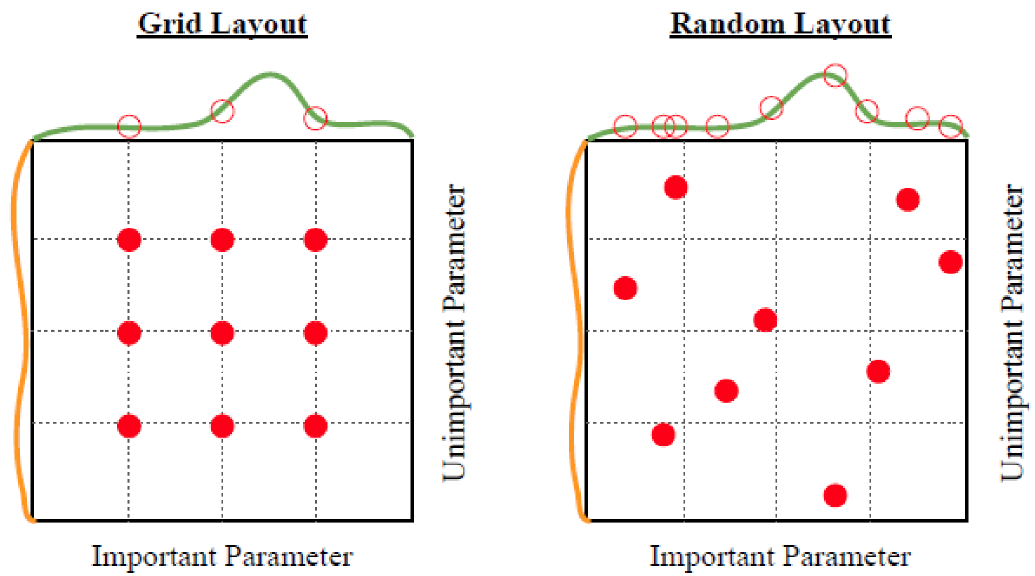
```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

Hyperparameter optimization

Random Search vs. Grid Search

Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012



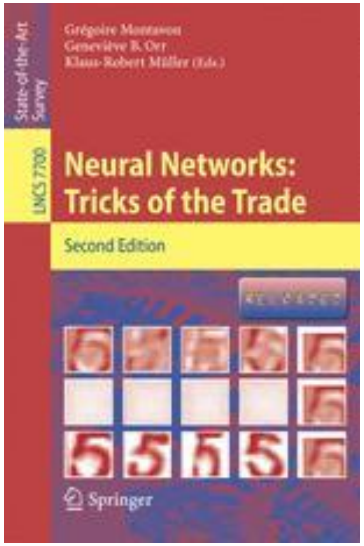
Hyperparameter optimization

- **Hyperparameters to play with:**
  - network architecture
  - learning rate, its decay schedule, update type
  - regularization (L2/Dropout strength)

- Other hyperparameter optimization methods
  - Shahriari, et al. "Taking the human out of the loop: A review of Bayesian optimization." Proceedings of the IEEE 104.1 (2016): 148-175.

More Tricks on CNN Training

- Book:
  - Neural Networks: Tricks of the Trade. Montavon, Grégoire, Orr, Geneviève, Müller, Klaus-Robert (Eds.) Springer 2012.



Summary

- Bag of tricks for improving generalization
  - Pros: you have a toolbox to use
  - Cons: many trial and error, tedious process

- Seeking fully automatic approaches to model selection
  - Bayesian optimization
  - Reinforcement learning