

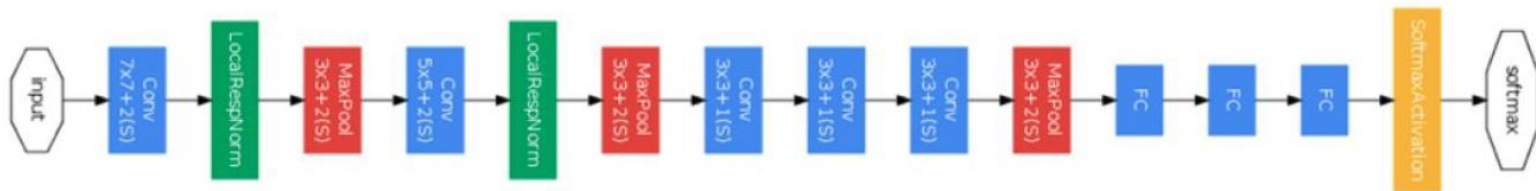


# Lecture 06: CNNs III – Network Regularization & Training

Lan Xu  
SIST, ShanghaiTech  
Fall, 2022

# Batch Normalization

- Problem in deep network learning



$$\ell = F_2(F_1(u, \Theta_1), \Theta_2)$$

- Change of distribution in activation across layers

# Batch Normalization

- Normalize the inputs to a layer:

“you want unit gaussian activations? just make them so.”

consider a batch of activations at some layer.  
To make each dimension unit gaussian, apply:

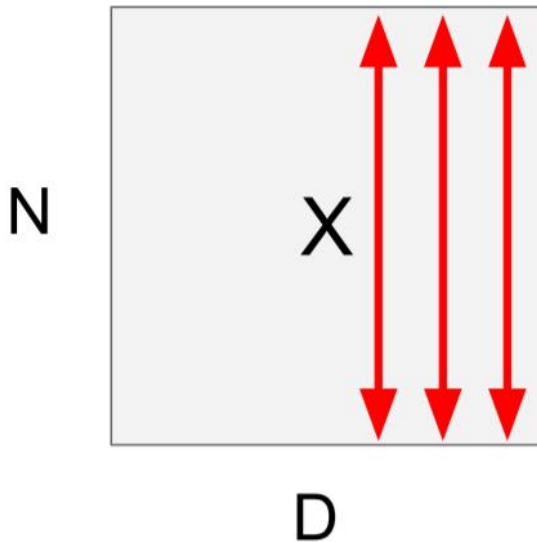
$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

this is a vanilla  
differentiable function...

# Batch Normalization

- Layer details

**Input:**  $x : N \times D$



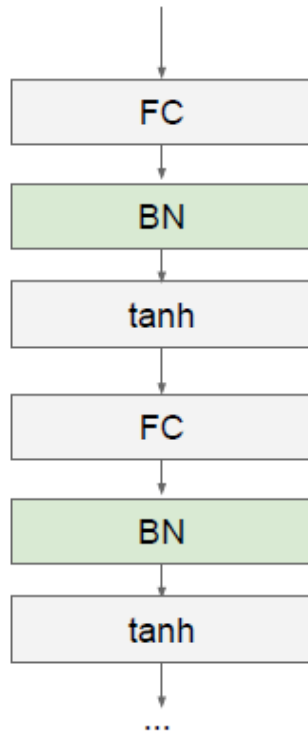
$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j} \quad \text{Per-channel mean, shape is D}$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is D}$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \quad \text{Normalized x, Shape is N x D}$$

# Batch Normalization

## ■ Layer details



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

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

# Batch Normalization

- Extra capacity:

**Input:**  $x : N \times D$

**Learnable scale and shift parameters:**

$$\gamma, \beta : D$$

Learning  $\gamma = \sigma$ ,  
 $\beta = \mu$ , will recover the  
identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j} \quad \text{Per-channel mean, shape is } D$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2 \quad \text{Per-channel var, shape is } D$$

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}} \quad \text{Normalized } x, \text{ Shape is } N \times D$$

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j \quad \text{Output, Shape is } N \times D$$

# Batch Normalization

## ■ Algorithm

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_1 \dots x_m\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

# Batch Normalization

## ■ Test time

**Input:**  $x : N \times D$

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean,  
shape is D

**Learnable scale and shift parameters:**

$$\gamma, \beta : D$$

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel var,  
shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

Normalized x,  
Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

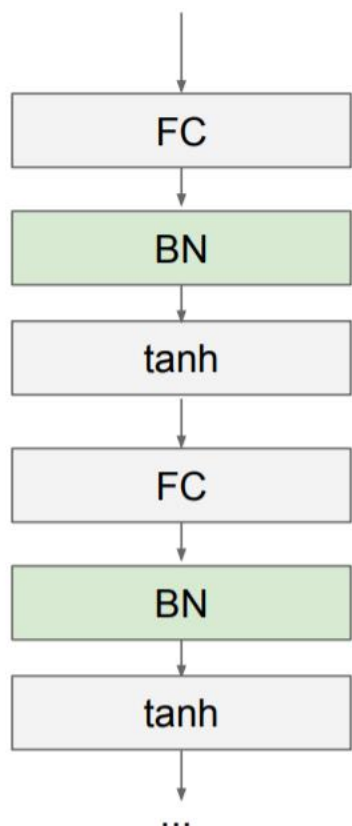
Output,  
Shape is N x D

During testing batchnorm  
becomes a linear operator!  
Can be fused with the previous  
fully-connected or conv layer



# Batch Normalization

## ■ Benefits



- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

# Batch Normalization

## ■ ConvNets

Batch Normalization for  
**fully-connected** networks

$$\begin{array}{l} \mathbf{x}: \mathbf{N} \times \mathbf{D} \\ \text{Normalize} \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{D} \\ \boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times \mathbf{D} \\ \mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

Batch Normalization for  
**convolutional** networks  
(Spatial Batchnorm, BatchNorm2D)

$$\begin{array}{l} \mathbf{x}: \mathbf{N} \times \mathbf{C} \times \mathbf{H} \times \mathbf{W} \\ \text{Normalize} \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times \mathbf{C} \times 1 \times 1 \\ \boldsymbol{\gamma}, \boldsymbol{\beta}: 1 \times \mathbf{C} \times 1 \times 1 \\ \mathbf{y} = \boldsymbol{\gamma}(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \boldsymbol{\beta} \end{array}$$

# Layer Normalization

## ■ Batch Normalization vs. Layer Normalization

**Batch Normalization** for  
fully-connected networks

$$\mathbf{x}: \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{1} \times \mathbf{D}$$

$$\gamma, \beta: \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \beta$$

**Layer Normalization** for  
fully-connected networks  
Same behavior at train and test!  
Can be used in recurrent networks

$$\mathbf{x}: \mathbf{N} \times \mathbf{D}$$

Normalize



$$\boldsymbol{\mu}, \boldsymbol{\sigma}: \mathbf{N} \times \mathbf{1}$$

$$\gamma, \beta: \mathbf{1} \times \mathbf{D}$$

$$\mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \beta$$

Ba, Kiros, and Hinton, "Layer Normalization", arXiv 2016

# Instance Normalization

**Batch Normalization** for  
convolutional networks

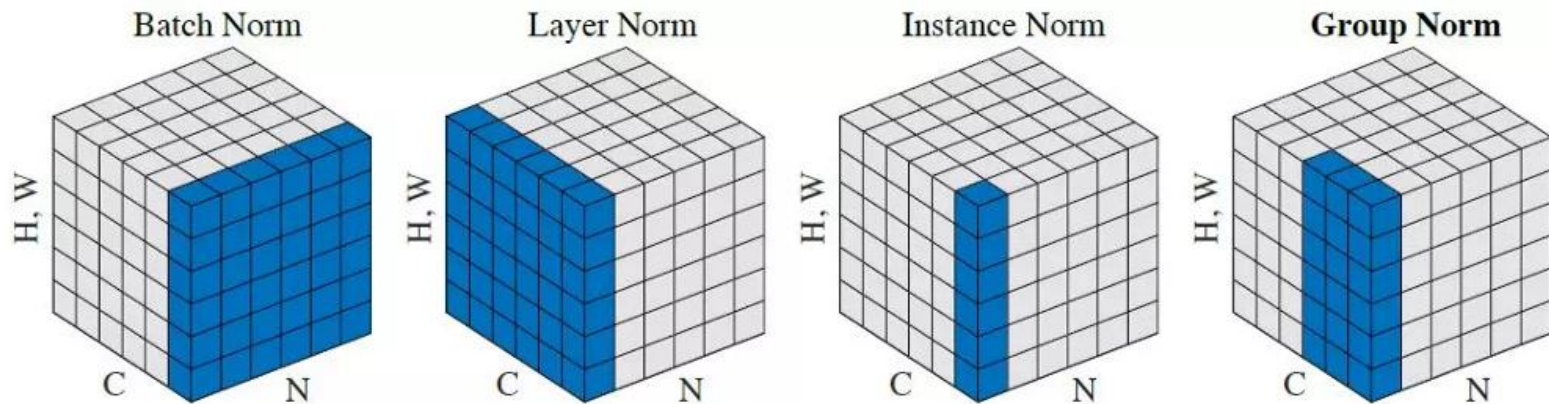
$$\begin{array}{l} \mathbf{x}: N \times C \times H \times W \\ \text{Normalize} \quad \downarrow \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: 1 \times C \times 1 \times 1 \\ \gamma, \beta: 1 \times C \times 1 \times 1 \\ \mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \beta \end{array}$$

**Instance Normalization** for  
convolutional networks  
Same behavior at train / test!

$$\begin{array}{l} \mathbf{x}: N \times C \times H \times W \\ \text{Normalize} \quad \quad \downarrow \quad \downarrow \\ \boldsymbol{\mu}, \boldsymbol{\sigma}: N \times C \times 1 \times 1 \\ \gamma, \beta: 1 \times C \times 1 \times 1 \\ \mathbf{y} = \gamma(\mathbf{x} - \boldsymbol{\mu}) / \boldsymbol{\sigma} + \beta \end{array}$$

# Batch-like Normalization

- Layer normalization (Ba, Kiros, Hinton, 2016)
- Instance normalization (Ulyanov, Vedaldi, Lempitsky, 2016)
- Group normalization (Wu and He, 2018)

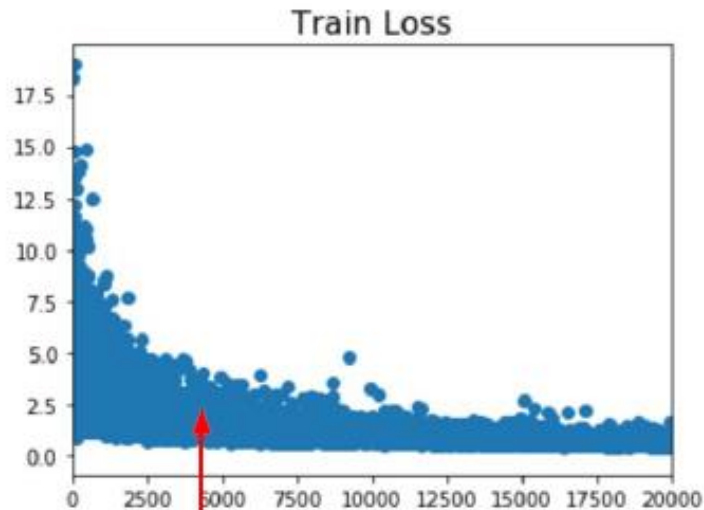


# Training overview

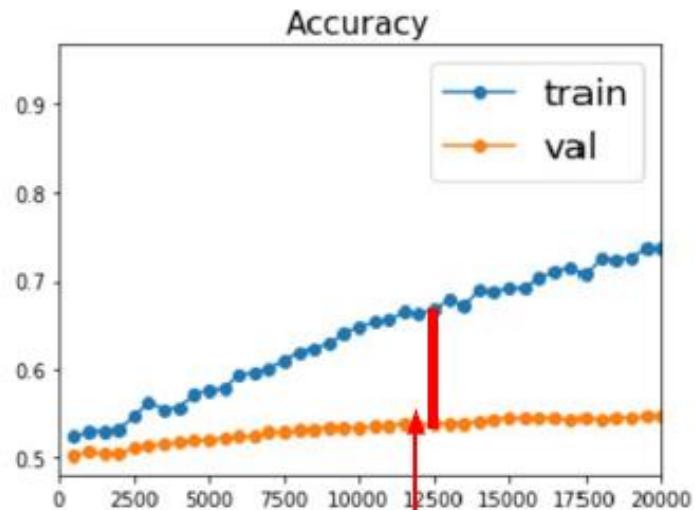
- Two aspects of training networks
  - Optimization
    - How do we minimize the loss function effectively?
  - Generalization
    - How do we avoid overfitting?
- CNN training pipeline
  - Data processing
  - Weight initialization
  - Parameter updates
  - Batch normalization
- Avoid overfitting: Regularization

# Beyond Training Error

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



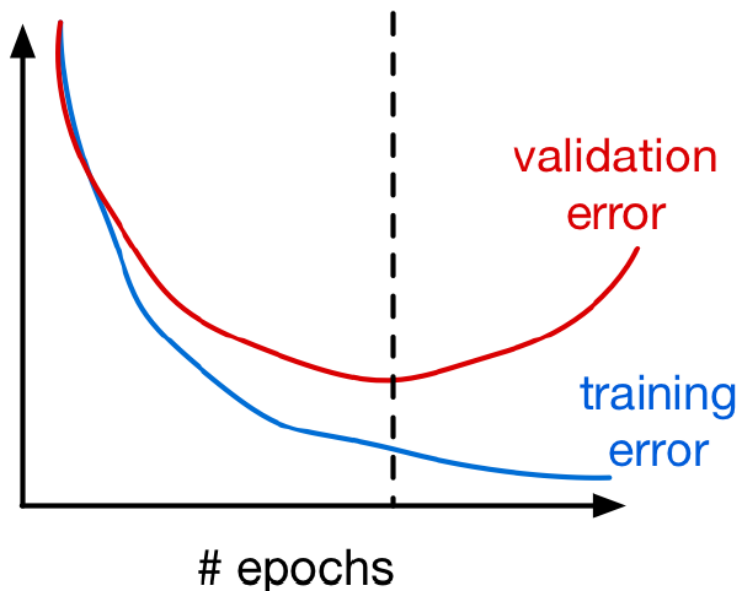
Better optimization algorithms  
help reduce training loss



But we really care about error on new  
data - how to reduce the gap?

# 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.

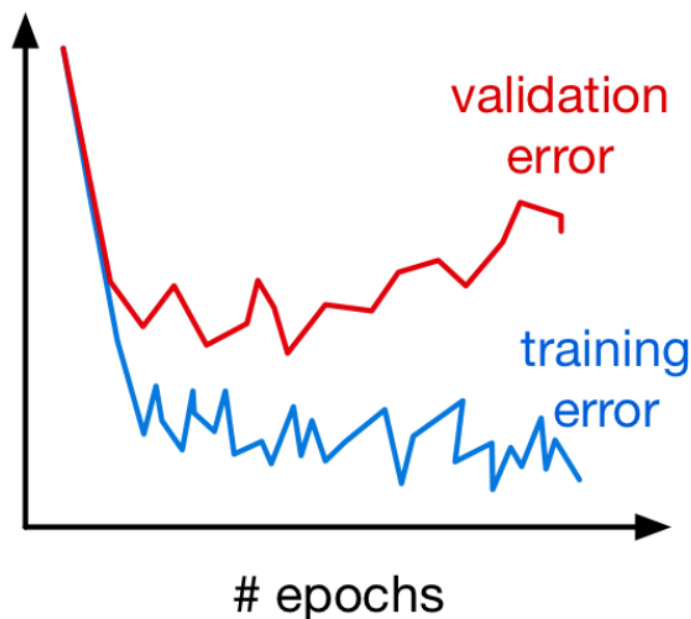


- Weights start out small, so it takes time for them to grow large. Therefore, it has a similar effect to weight decay.



# 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

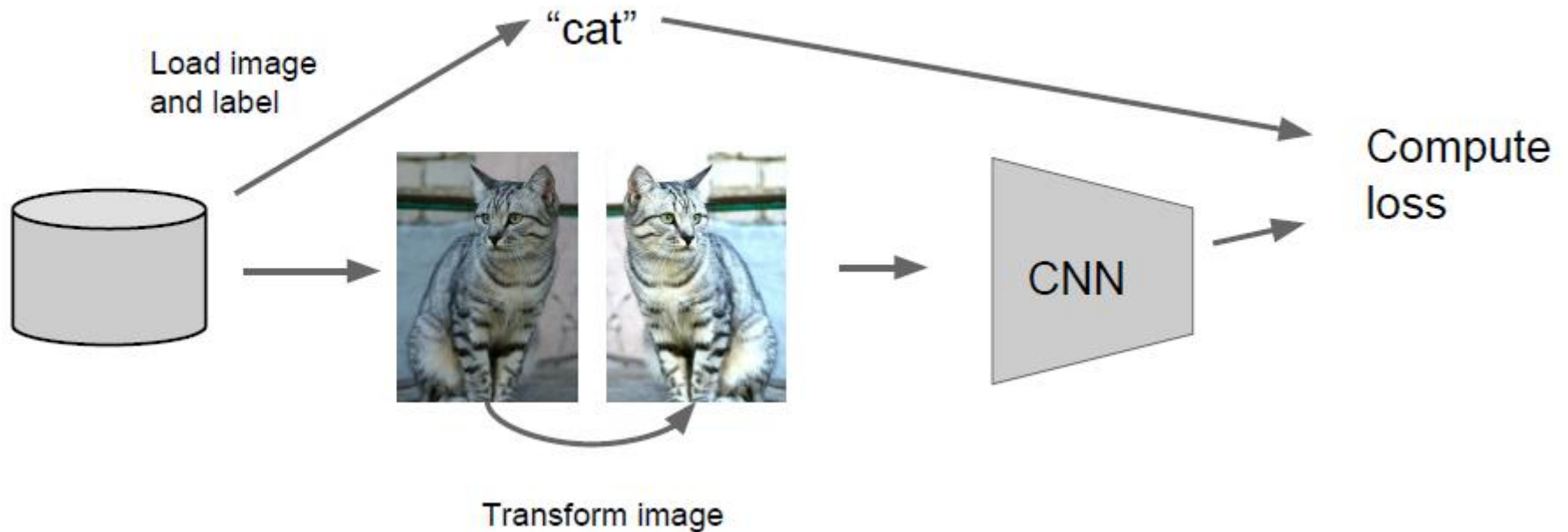


# Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization

# Data Augmentation

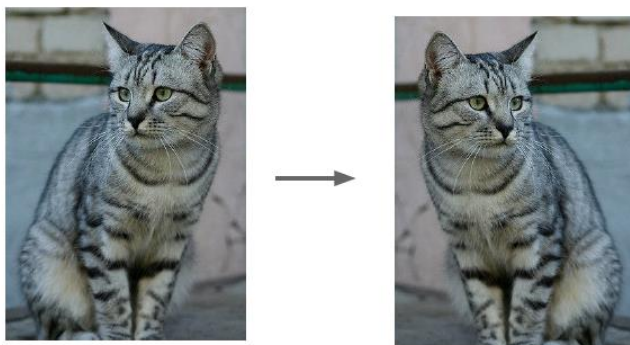
- Create more data for regularization



# Data Augmentation

## ■ Create more data for regularization

### Horizontal Flips



### Random crops and scales

**Training:** sample random crops / scales

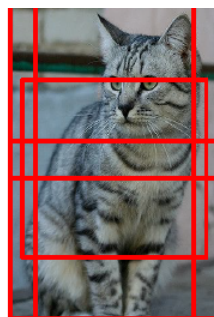
ResNet:

1. Pick random  $L$  in range  $[256, 480]$
2. Resize training image, short side =  $L$
3. Sample random  $224 \times 224$  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  $224 \times 224$  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  $2.9 \times 10^{32}$  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.

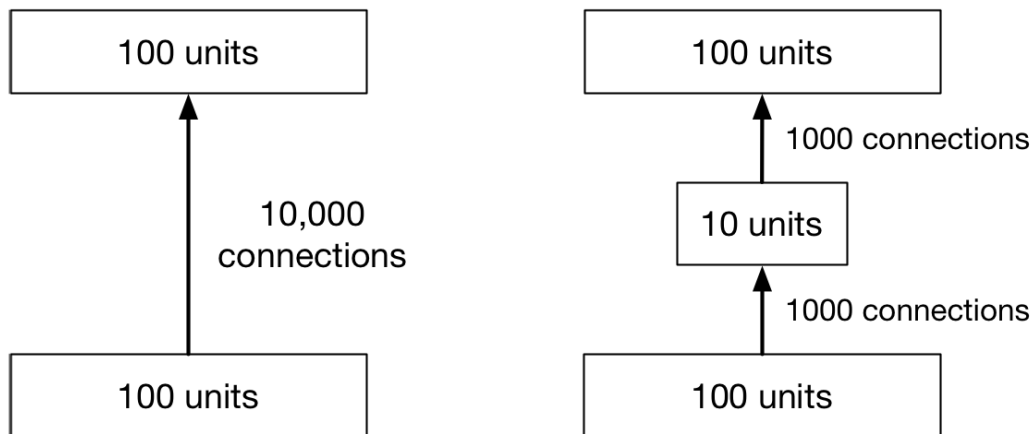
# Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization



# 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

- $L_2$  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

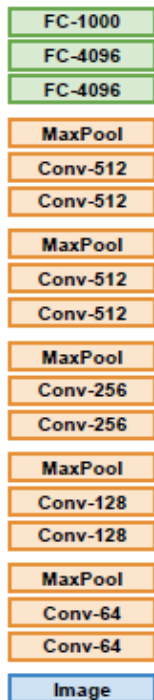
$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \left( \frac{\partial \mathcal{E}}{\partial \mathbf{w}} + \lambda \frac{\partial \mathcal{R}}{\partial \mathbf{w}} \right) \\ &= \mathbf{w} - \alpha \left( \frac{\partial \mathcal{E}}{\partial \mathbf{w}} + \lambda \mathbf{w} \right) \\ &=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

# Transfer Learning

## Transfer Learning with CNNs

### 1. Train on Imagenet

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

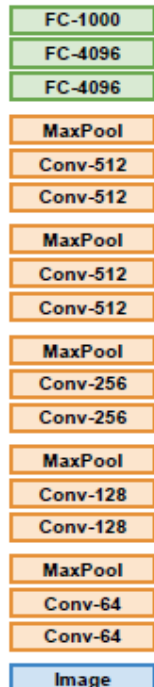


# Transfer Learning

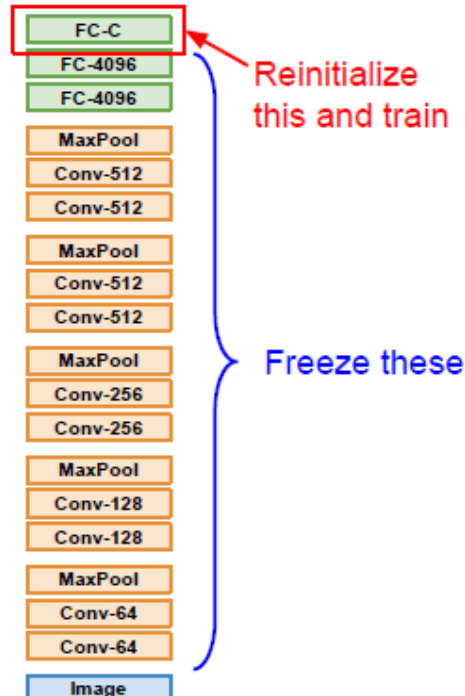
## Transfer Learning with CNNs

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

### 1. Train on Imagenet



### 2. Small Dataset (C classes)

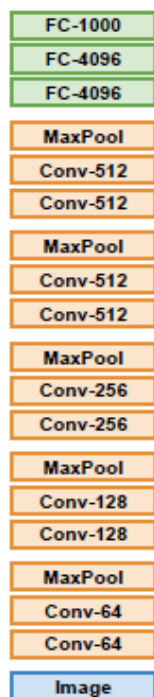


# Transfer Learning

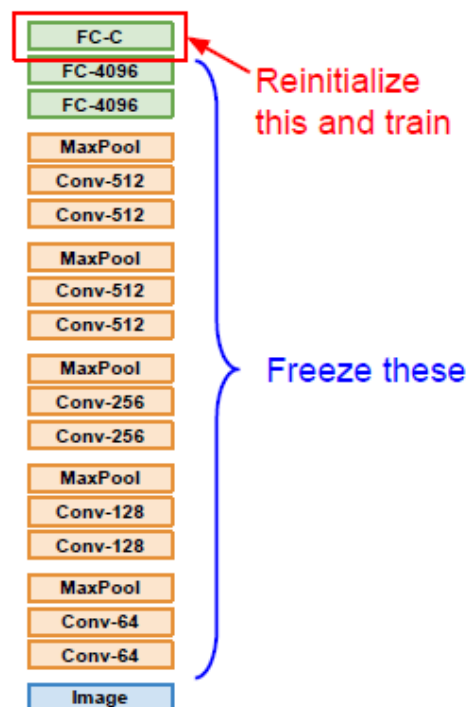
## Transfer Learning with CNNs

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

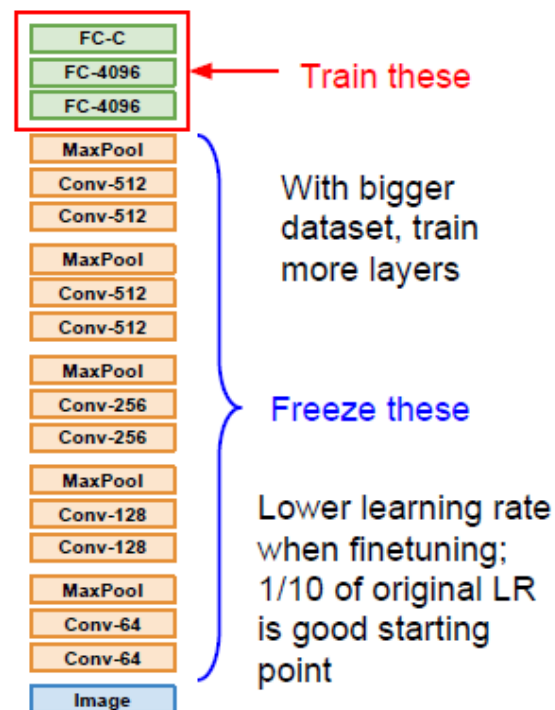
### 1. Train on Imagenet



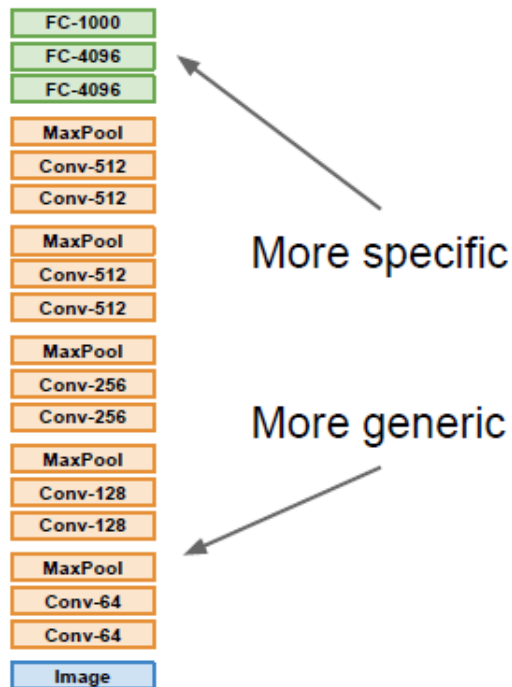
### 2. Small Dataset (C classes)



### 3. Bigger dataset



# Transfer Learning



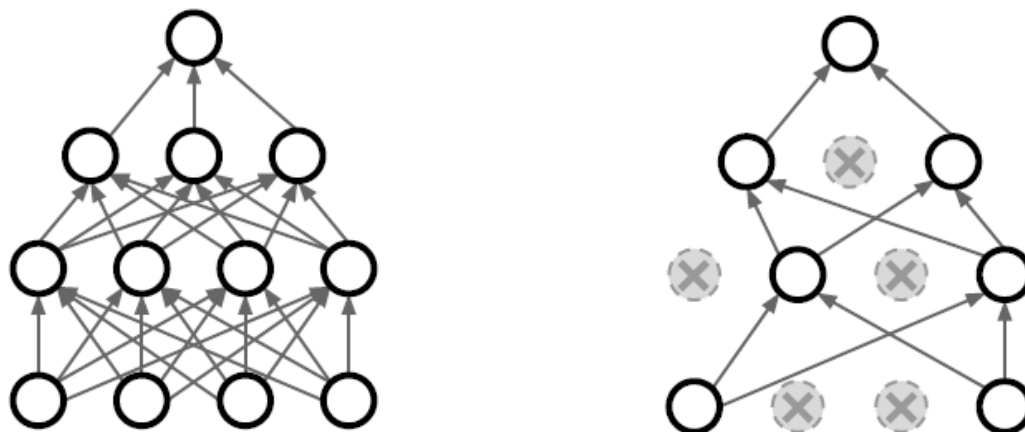
	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble... Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

# Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization
- Network Architectures

# 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

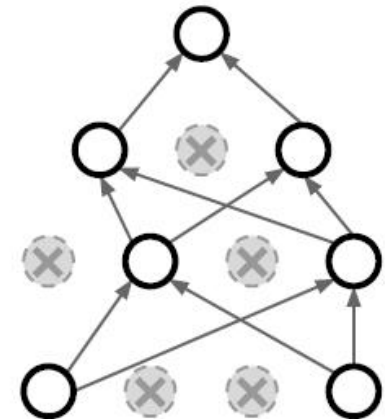
```
p = 0.5 # probability of keeping a unit active. higher = less dropout

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)
```

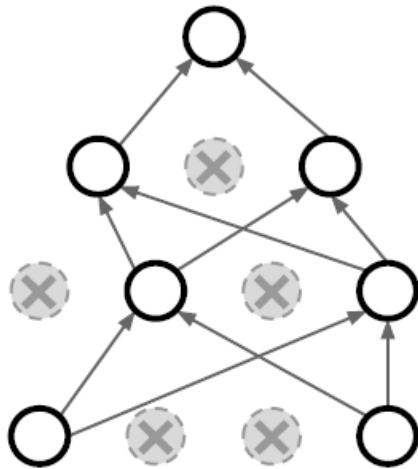
Example forward pass with a 3-layer network using dropout



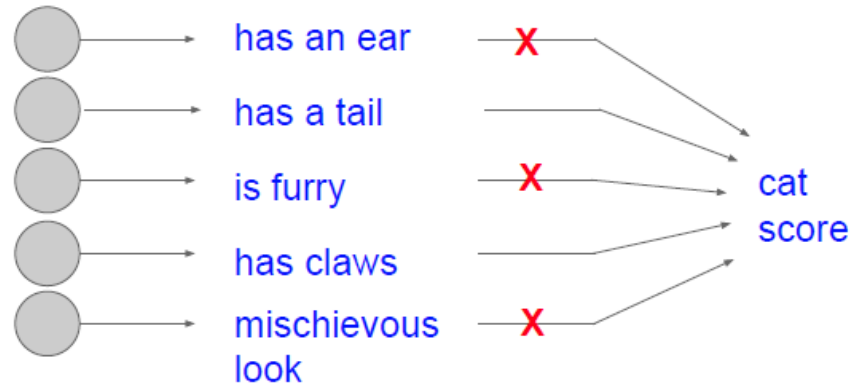
# Understanding Dropout

## Regularization: Dropout

How can this possibly be a good idea?

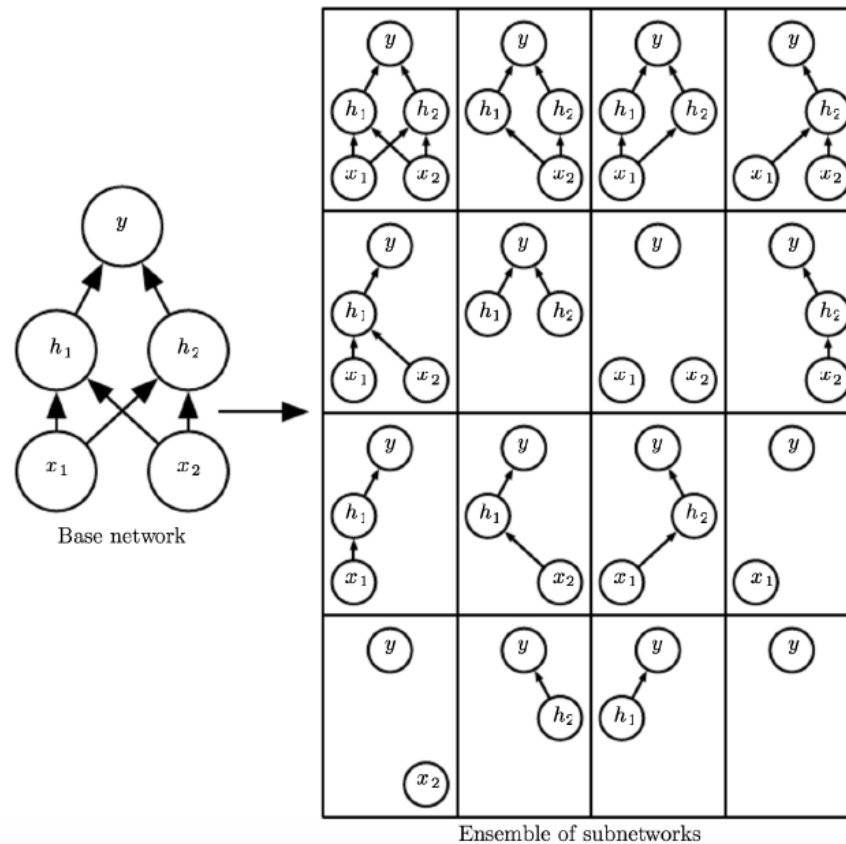


Forces the network to have a redundant representation;  
Prevents co-adaptation of features



# 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!

$$\text{Output (label)} \quad \boxed{y} = f_W(\text{Input (image)} \quad \boxed{x}, \boxed{z})$$

Random mask

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

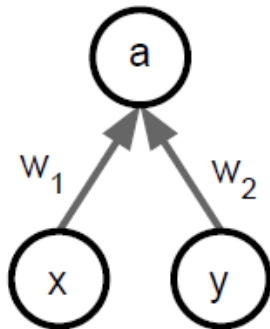
- Dropout at test time

Want to approximate  
the integral

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

Consider a single neuron.

At test time we have:  $E[a] = w_1x + w_2y$



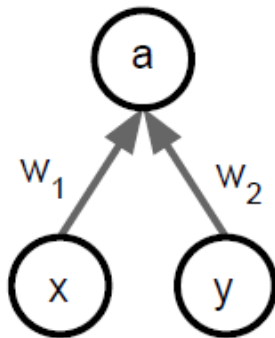
# Dropout

- Dropout at test time

Want to approximate the integral

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

Consider a single neuron.



At test time we have:  $E[a] = w_1x + w_2y$

During training we have: 
$$E[a] = \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) = \frac{1}{2}(w_1x + w_2y)$$

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:

output at test time = expected output at training time

# 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
```

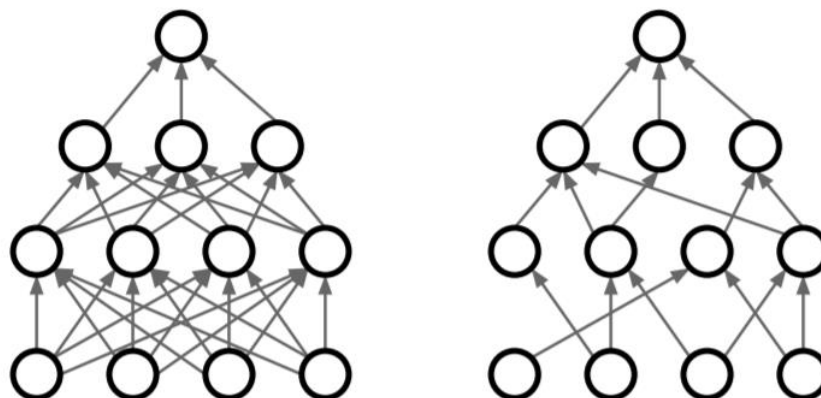
test time is unchanged!



# Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - **DropConnect** drops connections instead of activations.

- Training: Drop connections between neurons (set weights to 0)
- Testing: Use all the connections



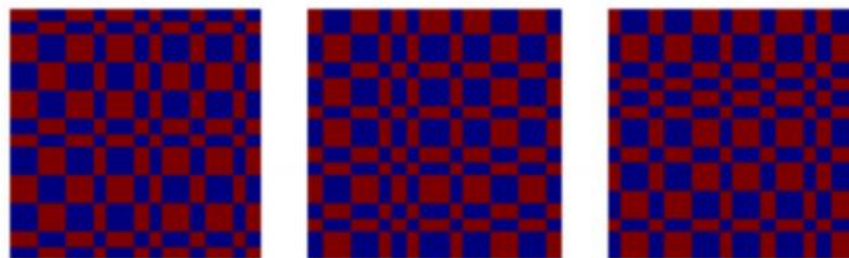
Wan et al, "Regularization of Neural Networks using DropConnect", ICML 2013

# Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:

- Fractional Pooling

- Training: Use randomized pooling regions
    - Testing: Average predictions from several regions



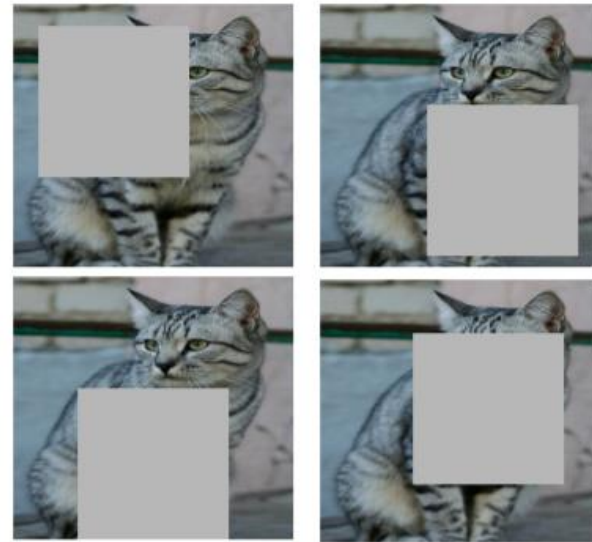
Graham, “Fractional Max Pooling”, arXiv 2014

# Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:

- Cutout

- Training: Set random image regions to zero
    - Testing: Use full image predictions from several regions



Works very well for small datasets like CIFAR,  
less common for large datasets like ImageNet

DeVries and Taylor, “Improved Regularization of Convolutional Neural Networks with Cutout”,  
arXiv 2017

# Stochastic Regularization

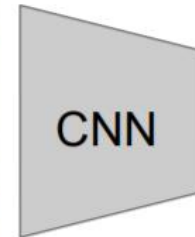
- Lots of other stochastic regularizers have been proposed:

- Mixup

- Training: Train on random blends of images
    - Testing: Use original images



Randomly blend the pixels of pairs of training images, e.g. 40% cat, 60% dog



Target label:  
cat: 0.4  
dog: 0.6

Zhang et al, “mixup: Beyond Empirical Risk Minimization”, ICLR 2018

# Stochastic Regularization

- Lots of other stochastic regularizers have been proposed:
  - Training: Add random noise
  - Testing: Marginalize over the noise
- In practice
  - Consider **dropout** for large fully-connected layers
  - **Batch normalization** and **data augmentation** almost always a good idea
  - Try **cutout** and **mixup** especially for small classification datasets

# Outline

- Regularization in CNN training
  - Data Augmentation
  - Weight Regularization & Transfer Learning
  - Stochastic Regularization
  - Hyper-parameter optimization

# 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  $> 3 \times$  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

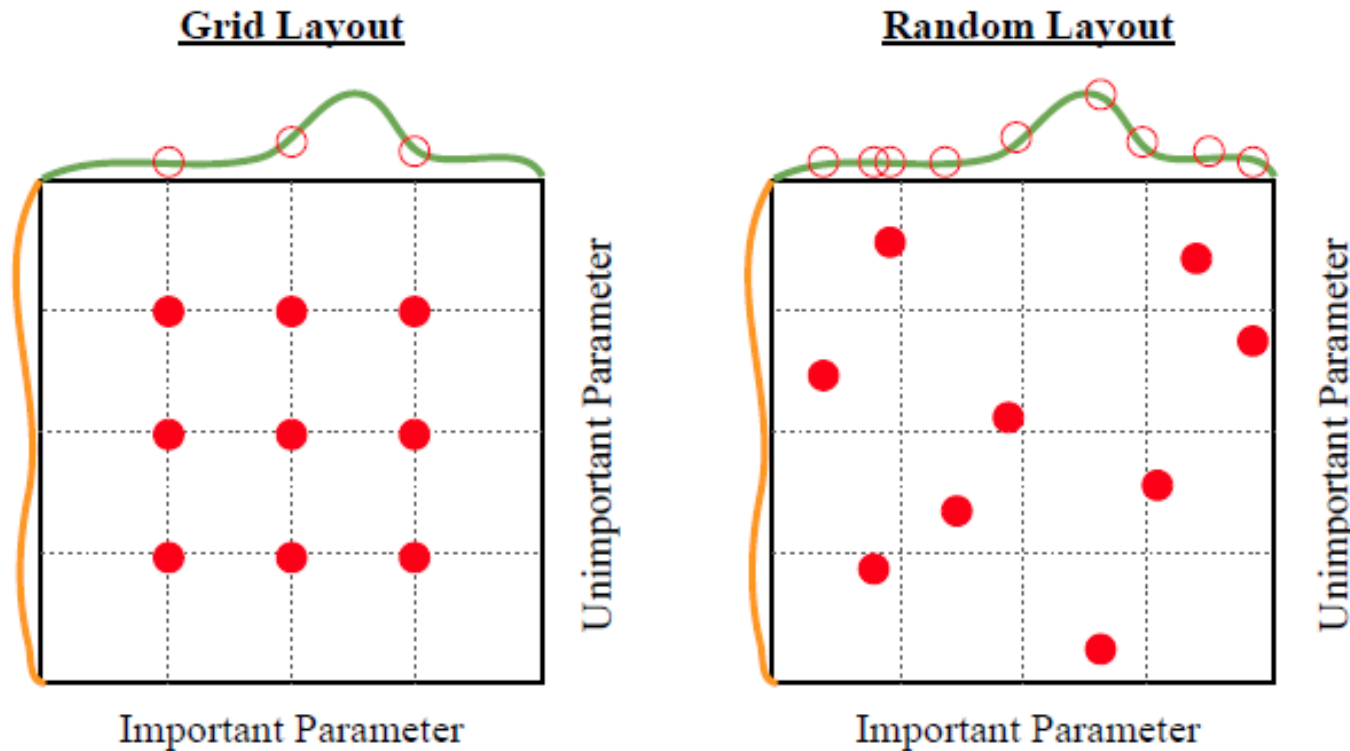
```
max_count = 100
for count in xrange(max_count):
    reg = 10**uniform(-4, 0)
    lr = 10**uniform(-3, -4)
```

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.

# 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
- Next time
  - CNN in Vision
- Reference
  - CS231n course notes