

CS 4644-DL / 7643-A: LECTURE 12

DANFEI XU

Topics:

- Training Neural Networks (Part 3)

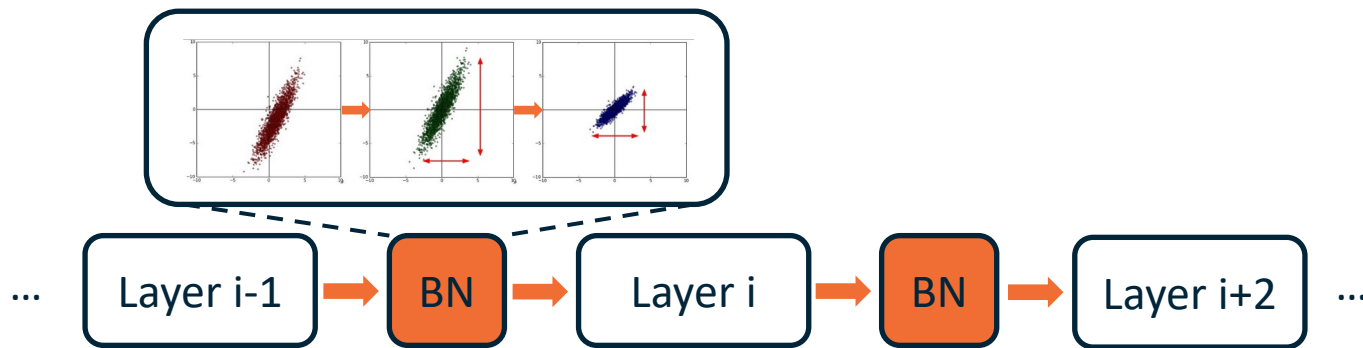
Administrative

- Project Proposal deadline **postponed to Oct 3rd (Monday)**
 - No grace period

Batch Normalization

Batch Normalization

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



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

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: $x : N \times D$
Learnable scale and shift parameters:

$$\gamma, \beta : \mathbb{R}^D$$

We want to give the model a chance to **adjust batchnorm** if the default is not optimal.

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

$$\mu_j = \frac{1}{N} \sum_{i=1}^N x_{i,j}$$

Per-channel mean,
shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (x_{i,j} - \mu_j)^2$$

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

$$y_{i,j} = \underline{\gamma}_j \hat{x}_{i,j} + \underline{\beta}_j$$

Output,
Shape is N x D

Batch Normalization: Test-Time

Input: $x : N \times D$
Learnable scale and shift parameters:

$$\gamma, \beta: \mathbb{R}^D$$

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

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

Per-channel mean, shape is D

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

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

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

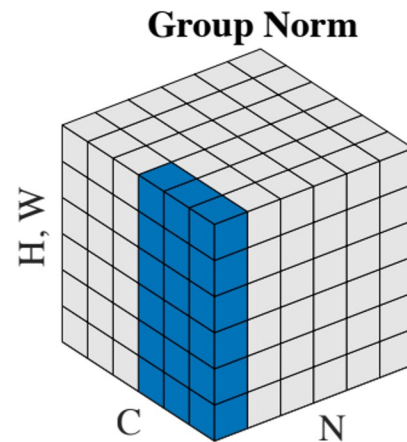
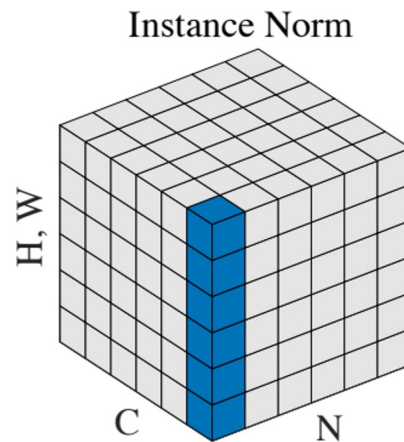
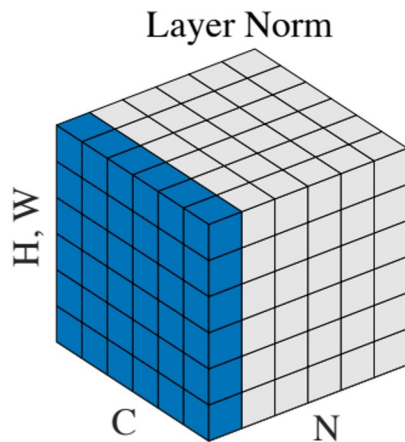
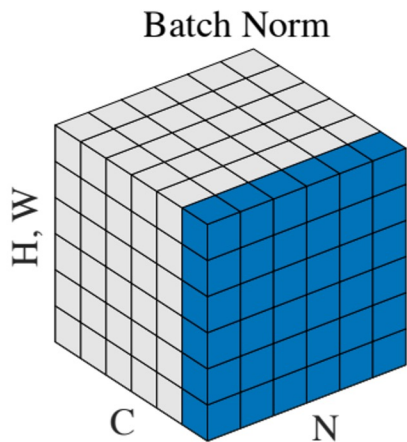
Output, Shape is N x D

Batch Normalization

[Ioffe and Szegedy, 2015]

- Makes deep networks **much** easier to train!
 - If you are interested in the theory, read <https://arxiv.org/abs/1805.11604>
 - TL;DR: makes optimization landscape smoother
- Allows higher learning rates, faster convergence
- More useful in deeper networks
- Networks become more robust to initialization
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!
- Needs large batch size to calculate accurate stats

Group Normalization



SGD + Momentum

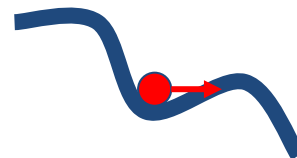
Intuitions:

- Think of a ball (set of parameters) moving in space (loss landscape), with momentum keeping it going in a direction.
- Individual gradient step may be noisy, the general trend accumulated over a few steps will point to the right direction.
- Momentum can “push” the ball over saddle points or local minima.

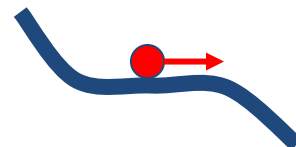
Noisy gradients



Local Minima



Saddle points



SGD + Momentum:

continue moving in the general direction as the previous iterations

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
    dx = compute_gradient(x)
    x -= learning_rate * dx
```

SGD+Momentum

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

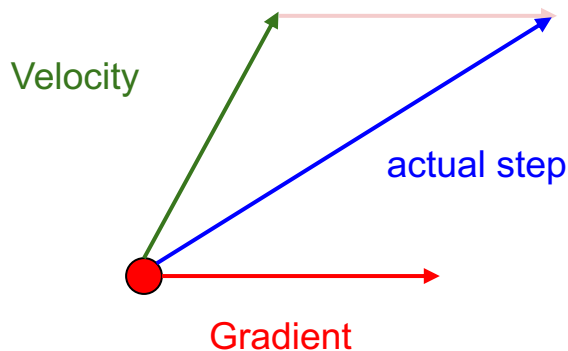
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
    dx = compute_gradient(x)
    vx = rho * vx + dx
    x -= learning_rate * vx
```

- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

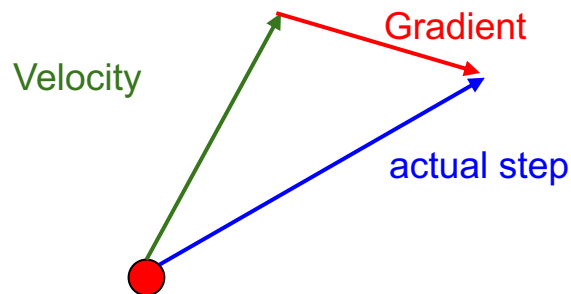
Nesterov Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov Momentum



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

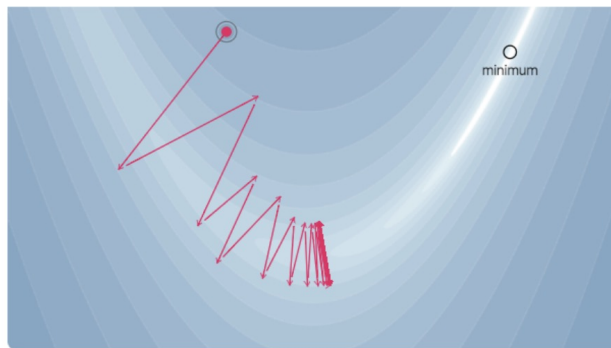
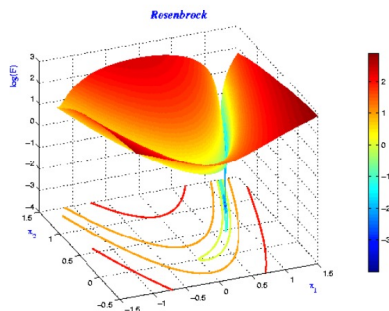
Nesterov, “A method of solving a convex programming problem with convergence rate $O(1/k^2)$ ”, 1983
Nesterov, “Introductory lectures on convex optimization: a basic course”, 2004
Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Optimization: Problem #3 with SGD

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?

Very slow progress along shallow dimension, jitter along steep direction

Long, narrow ravines:

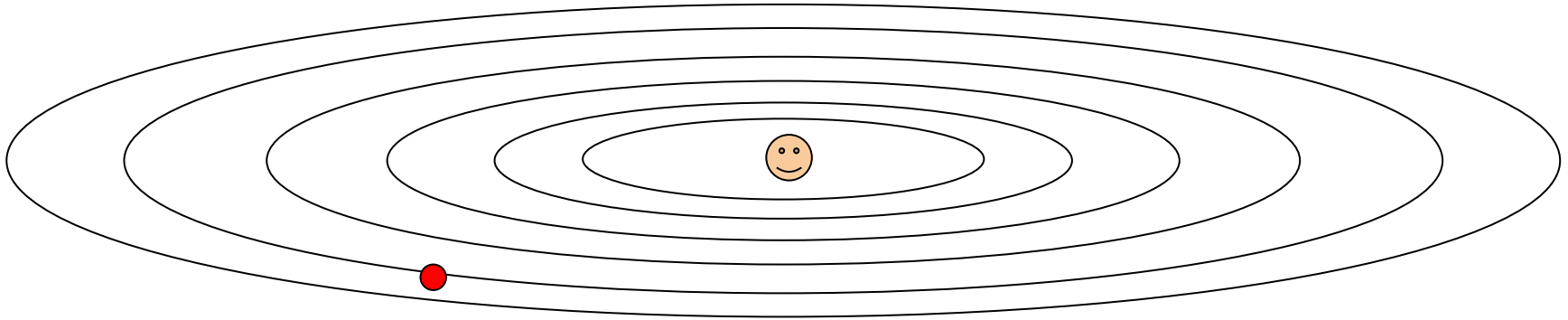


https://www.cs.toronto.edu/~rgrosse/courses/csc421_2019/slides/lec07.pdf

Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q2: What happens to the step size over long time? Decays to zero

RMSProp: “Leaky AdaGrad”

AdaGrad

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared += dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



RMSProp

```
grad_squared = 0
while True:
    dx = compute_gradient(x)
    grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
    x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7)
```

Momentum

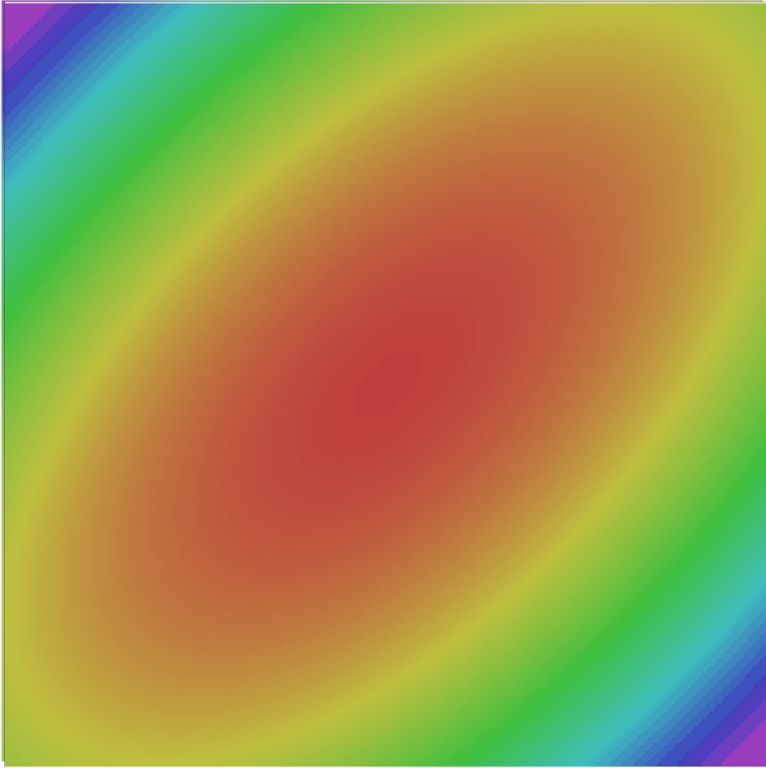
Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

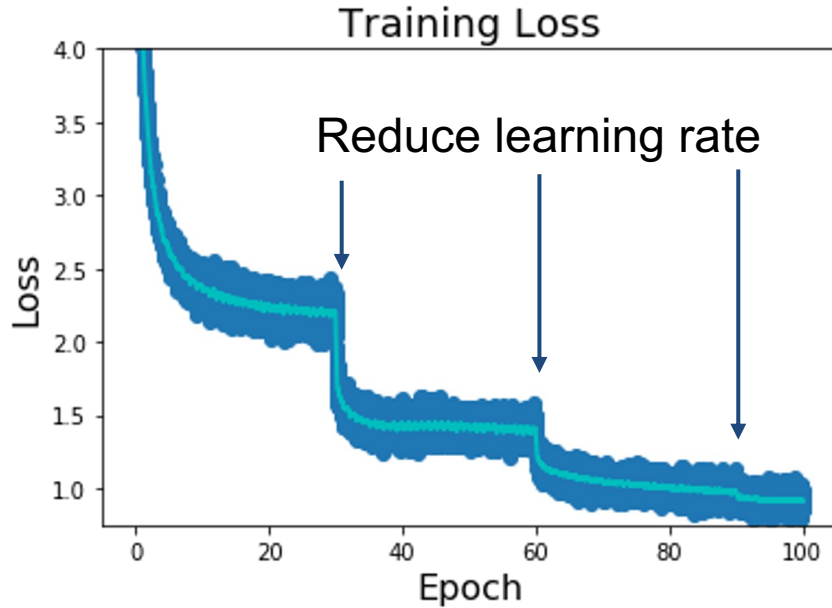
Adam with $\text{beta1} = 0.9$, $\text{beta2} = 0.999$, and $\text{learning_rate} = 1e-3$ or $5e-4$ is a great starting point for many models!

Adam



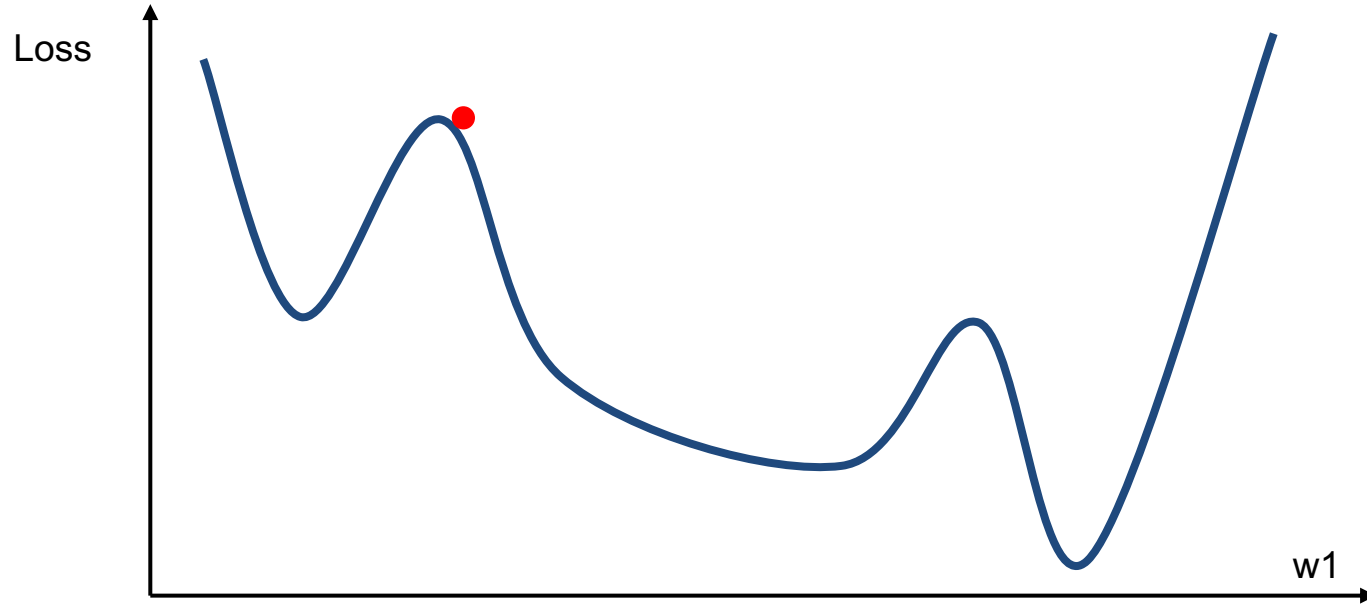
- SGD
- SGD+Momentum
- RMSProp
- Adam

Learning rate decays over time



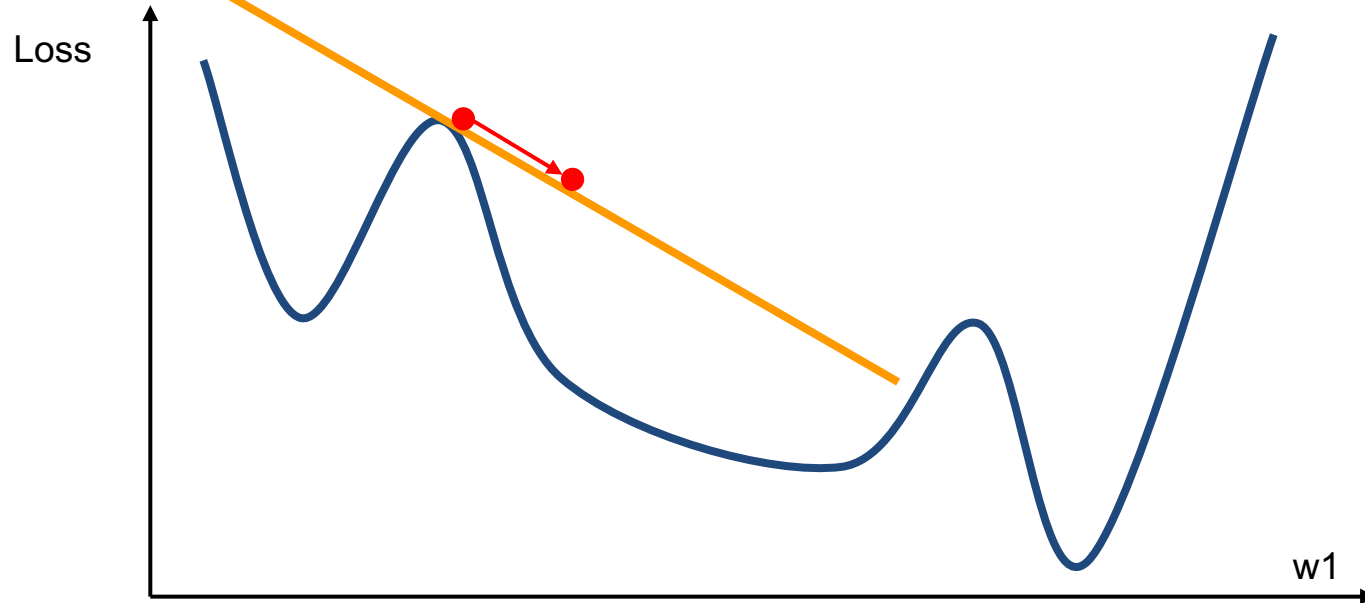
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

First-Order Optimization



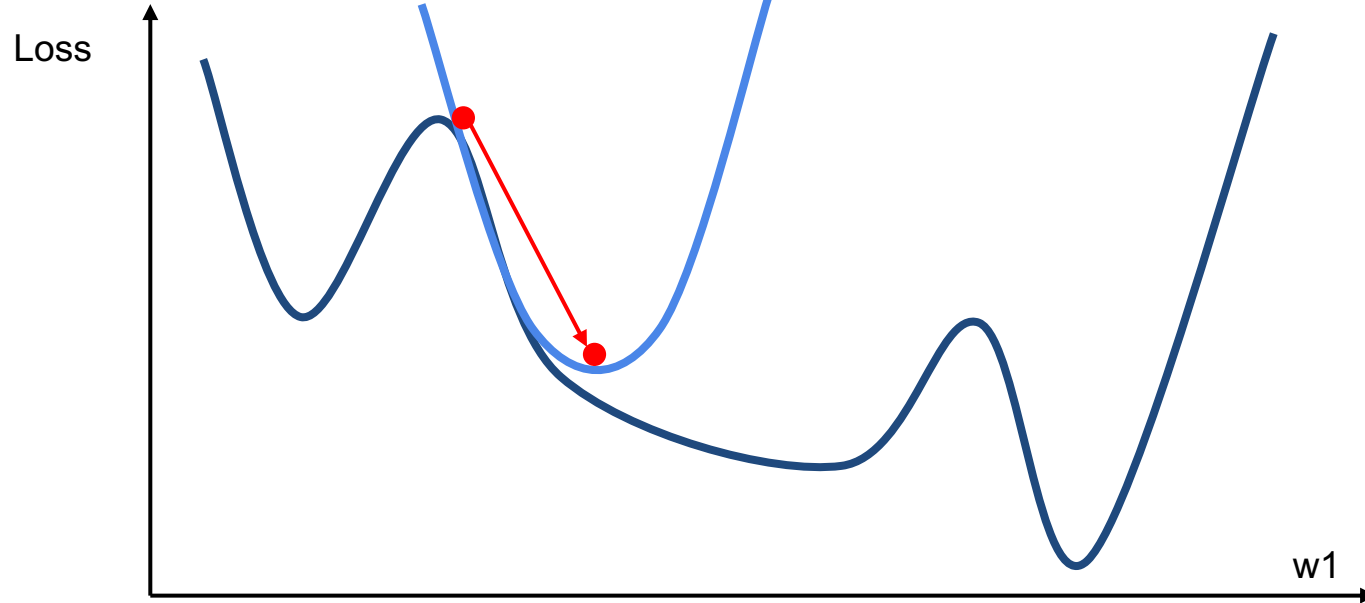
First-Order Optimization

- (1) Use gradient form linear approximation
- (2) Step to minimize the approximation



Second-Order Optimization

- (1) Use gradient **and Hessian** to form **quadratic** approximation
- (2) Step to the **minima** of the approximation



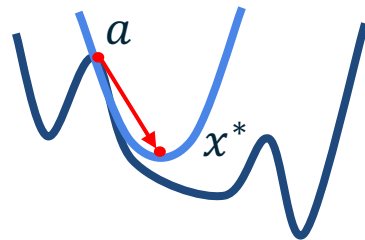
Second-Order Optimization

second-order Taylor Expansion of $f(x)$ at a :

$$f(x) = f(a) + \frac{f'(a)}{1!} (x - a) + \frac{f''(a)}{2!} (x - a)^2$$

Newton's method for optimization: solving for the critical point $f'(x) = 0$, we obtain the Newton update rule

$$f'(x) = f'(a) + f''(a)(x - a) = 0$$
$$x^* = a - \frac{1}{f''(a)} f'(a)$$



Think of a as the current params, x^* as the updated params

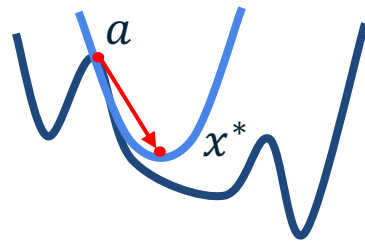
Second-Order Optimization (multivariate)

second-order Taylor Expansion of $f(\mathbf{x})$ at \mathbf{a} :

$$f(\mathbf{w}) = f(\mathbf{a}) + (\mathbf{x} - \mathbf{a})^T \nabla f + \frac{1}{2} (\mathbf{x} - \mathbf{a})^T H (\mathbf{x} - \mathbf{a})$$

Newton's method for optimization: solving for the critical point we obtain the Newton update rule:

$$\mathbf{x}^* = \mathbf{a} - H^{-1} \nabla f$$



Q: Why is this bad for deep learning?

Hessian Matrix

$$\mathbf{H}_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Second-Order Optimization

second-order Taylor expansion:

$$f(\mathbf{x}) = f(\mathbf{a}) + (\mathbf{x} - \mathbf{a})^T \nabla f + \frac{1}{2} (\mathbf{x} - \mathbf{a})^T H (\mathbf{x} - \mathbf{a})$$

Solving for the critical point we obtain the Newton parameter update:

$$\mathbf{x}^* = \mathbf{a} - H^{-1} \nabla f$$

Hessian has $O(N^2)$ elements
Inverting takes $O(N^3)$
N = Millions

Q: Why is this bad for deep learning?

Second-Order Optimization

- Quasi-Newton methods (**BFGS** most popular):
instead of inverting the Hessian ($O(n^3)$), approximate inverse Hessian with rank 1 updates over time ($O(n^2)$ each).
Still pretty expensive
- **L-BFGS** (Limited memory BFGS):
Does not form/store the full inverse Hessian.

L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic $f(x)$ then L-BFGS will probably work very nicely
- **Does not transfer very well to mini-batch setting.** Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Le et al, "On optimization methods for deep learning, ICML 2011"

Ba et al, "Distributed second-order optimization using Kronecker-factored approximations", ICLR 2017

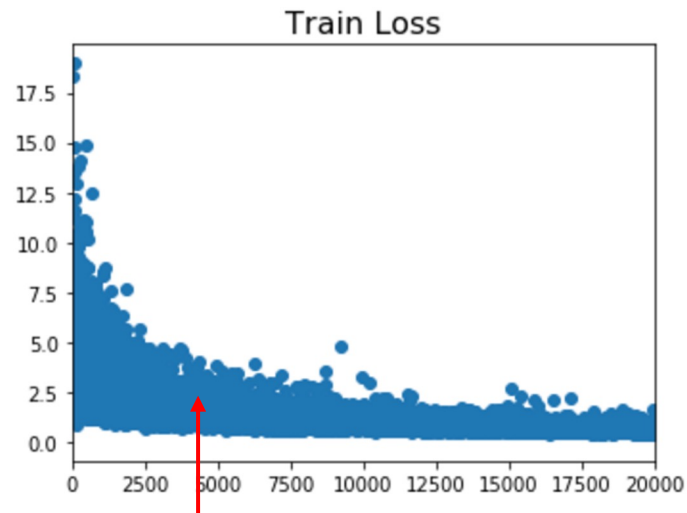
This Time:

Training Deep Neural Networks

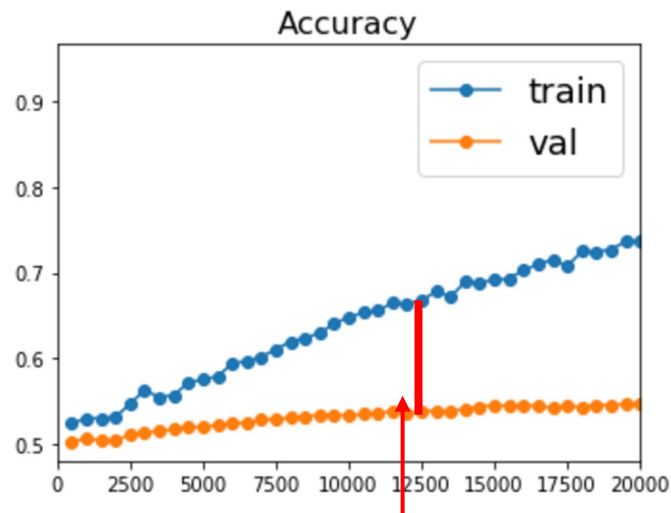
- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- **Regularization**
- **Data Augmentation**
- **Transfer learning**
- **Hyperparameter Tuning**

Regularization

Beyond Training Error

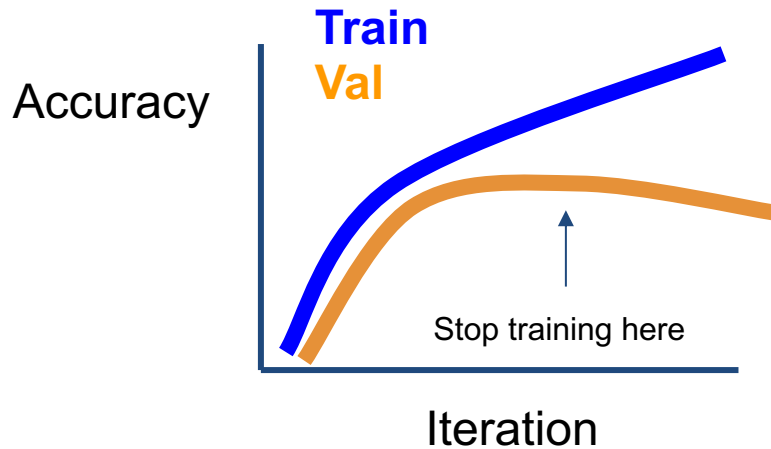
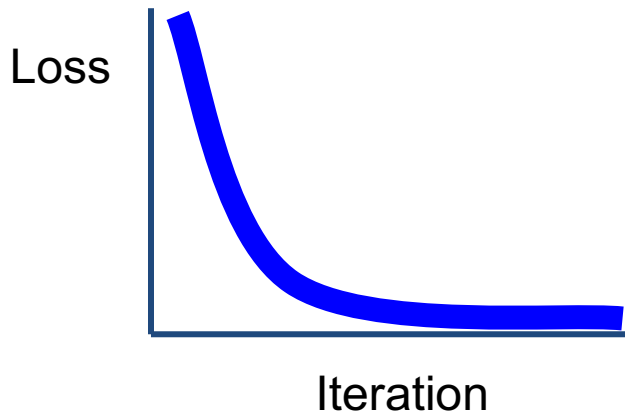


Better optimization algorithms help reduce training loss



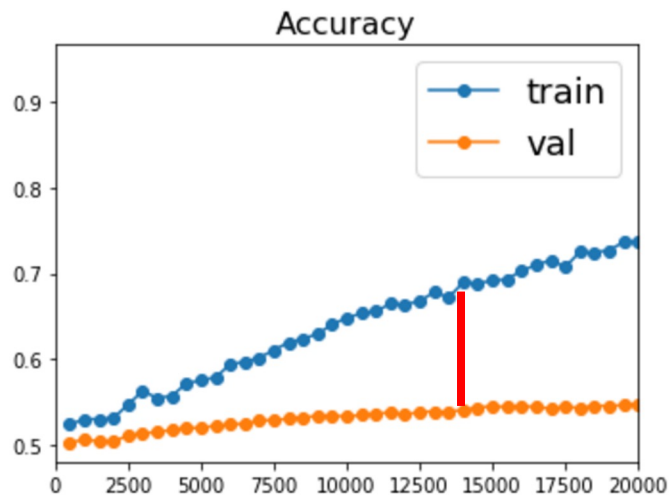
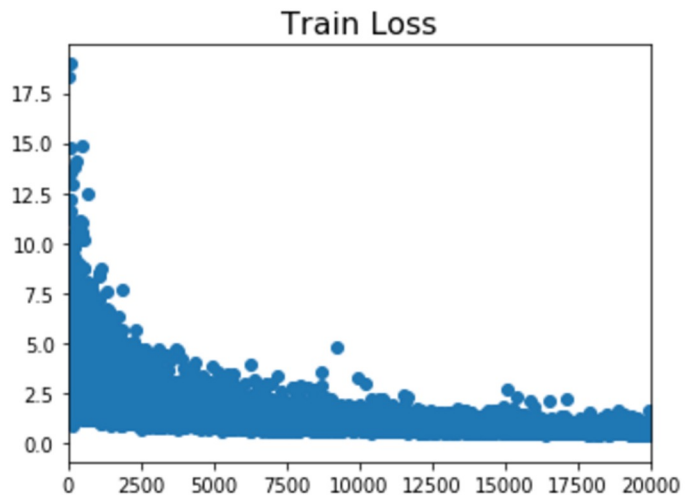
But we really care about error on new data - how to reduce the gap?

Early Stopping: Always do this



Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot
that worked best on val

How to improve single-model performance?



Regularization

Regularization: Add term to loss

$$L = \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

In common use:

L2 regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2 \quad (\text{Weight decay})$$

L1 regularization

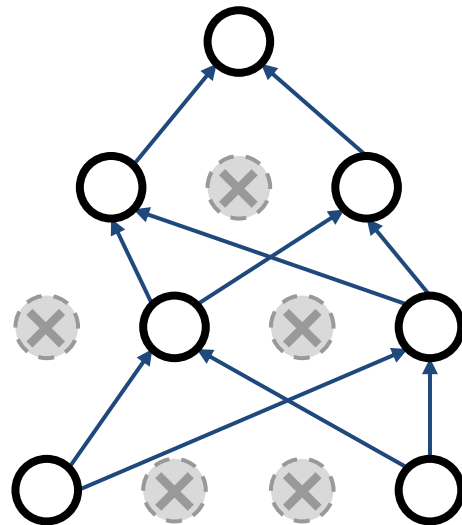
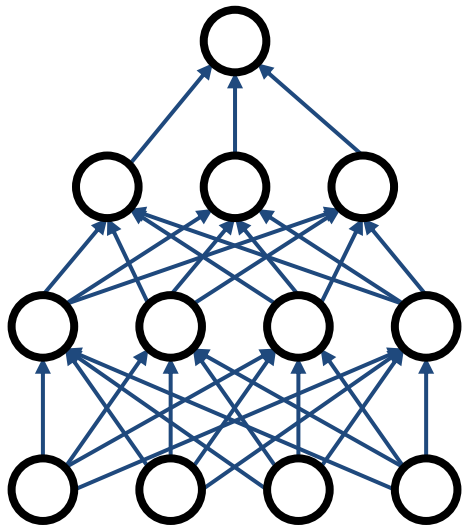
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

Elastic net (L1 + L2)

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

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
```

```
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
```

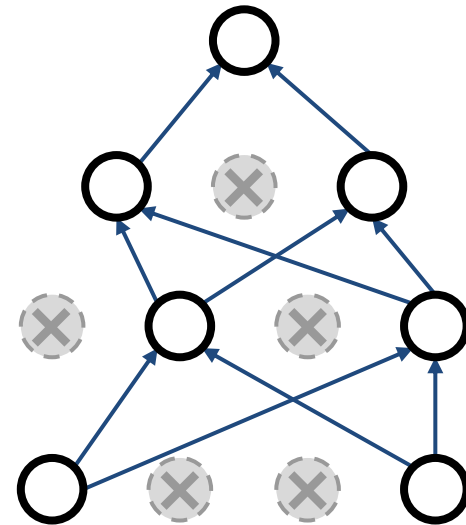
```
    H2 *= U2 # drop!
```

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

```
    # backward pass: compute gradients... (not shown)
```

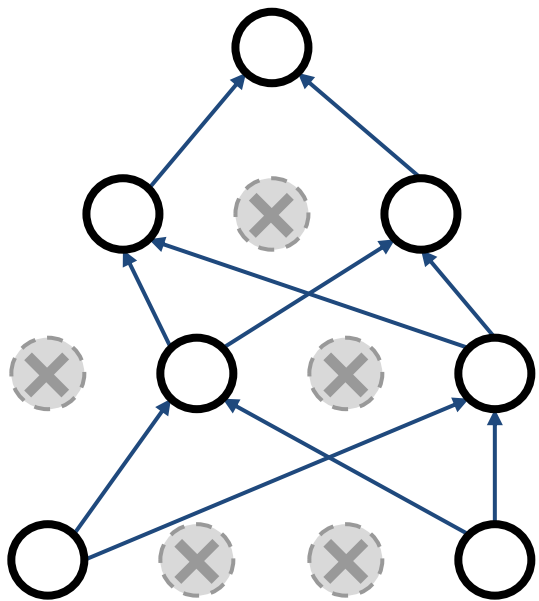
```
    # perform parameter update... (not shown)
```

Example forward pass with a 3-layer network using dropout



Regularization: Dropout

How can this possibly be a good idea?

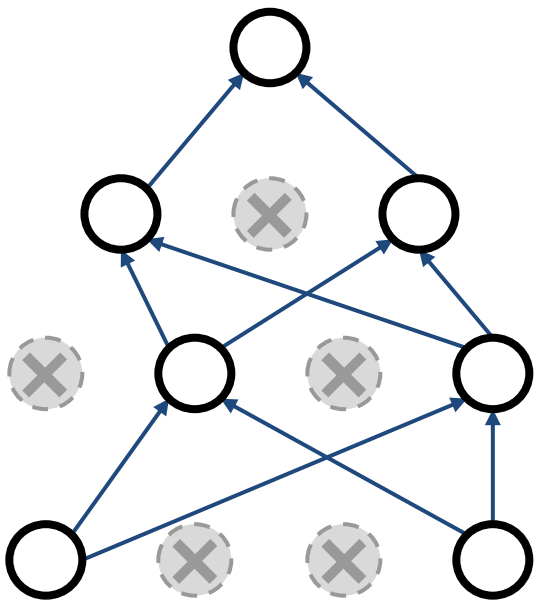


Forces the network to have a redundant representation;
Prevents co-adaptation of features



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

Dropout makes our output random!

$$\begin{array}{l} \text{Output} \\ \text{(label)} \end{array} \quad \boxed{y} = f_W \left(\begin{array}{l} \text{Input} \\ \text{(image)} \end{array} \boxed{x}, \begin{array}{l} \text{Random} \\ \text{mask} \end{array} \boxed{z} \right)$$

Want to “average out” the randomness at test-time

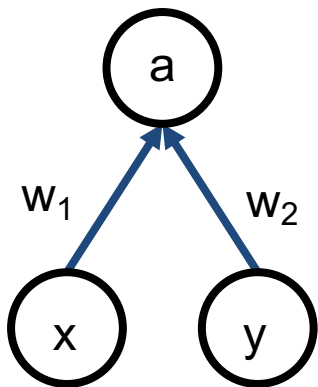
$$y = f(x) = E_z [f(x, z)] = \int p(z) f(x, z) dz$$

Dropout: Test time

Compute the expectation

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

Consider a single neuron.

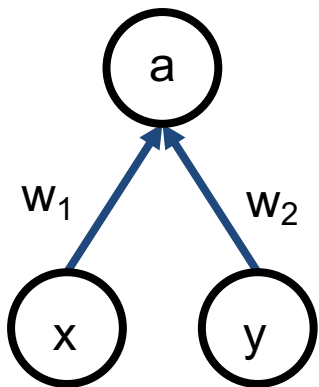


Dropout: Test time

Compute the expectation

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

Consider a single neuron.



Without dropout:

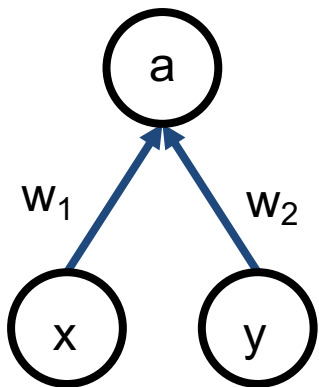
$$E[a] = w_1x + w_2y$$

Dropout: Test time

Compute the expectation

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

Consider a single neuron.



Without dropout:

$$E[a] = w_1 x + w_2 y$$

With dropout we have:

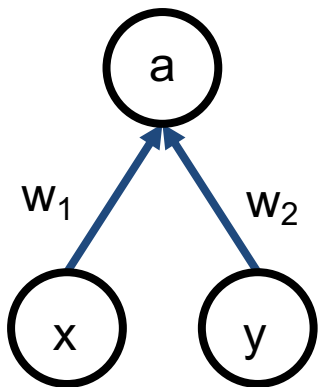
$$\begin{aligned} E[a] &= \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y) \\ &= \frac{1}{2}(w_1 x + w_2 y) \end{aligned}$$

Dropout: Test time

Compute the expectation

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

Consider a single neuron.



Without dropout:

$$E[a] = w_1x + w_2y$$

With dropout we have:

$$\begin{aligned} E[a] &= \frac{1}{4}(w_1x + w_2y) + \frac{1}{4}(w_1x + 0y) \\ &\quad + \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2y) \\ &= \frac{1}{2}(w_1x + w_2y) \end{aligned}$$

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

Dropout: 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 Summary

```
""" Vanilla Dropout: Not recommended implementation (see notes below) """
```

```
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
```

drop in train time

scale at test time

More common: “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!

Similar to BatchNorm, different behavior train vs test!

Regularization: A common pattern

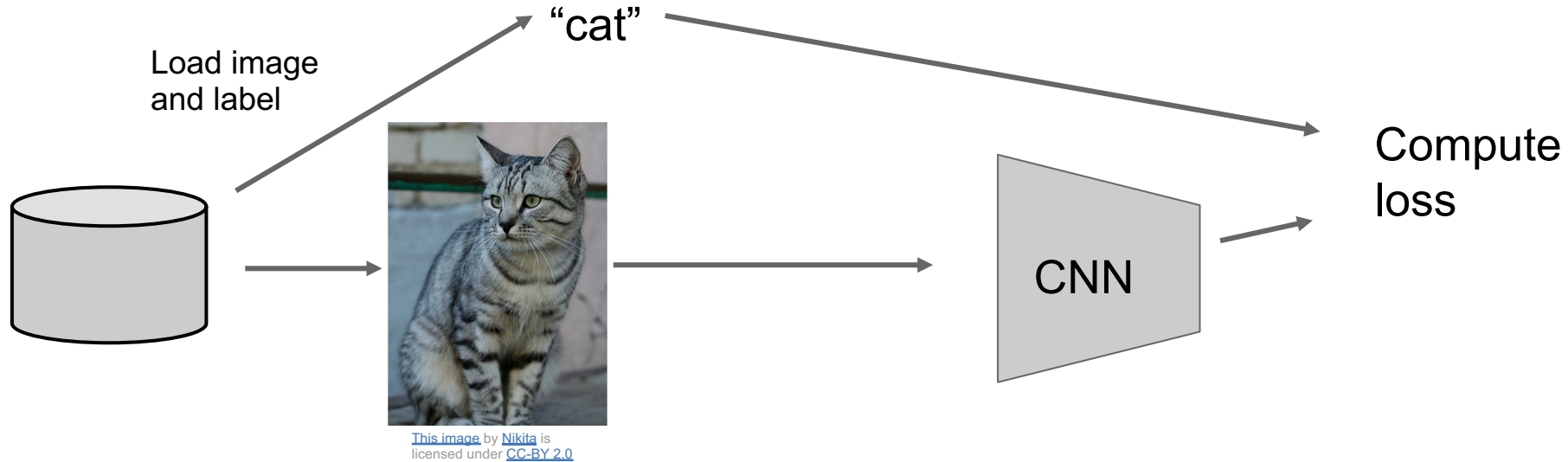
Training: Add some kind of randomness

$$y = f_W(x, z)$$

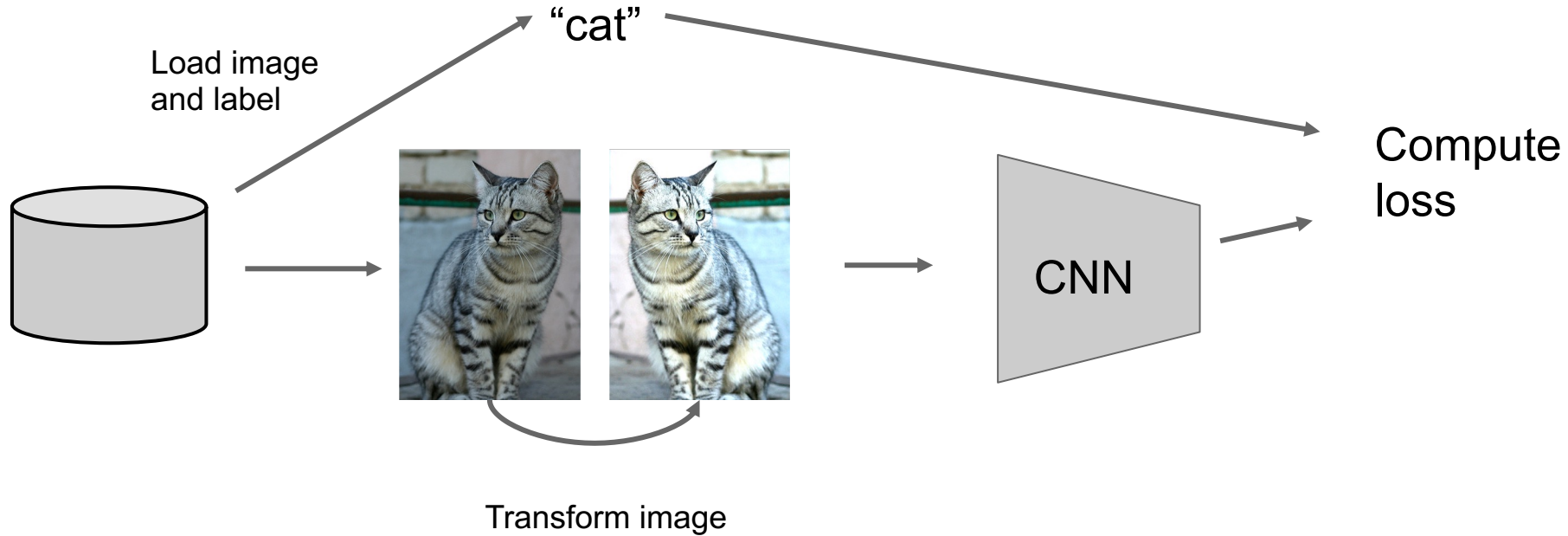
Testing: Average out randomness (sometimes approximate)

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

Regularization: Data Augmentation



Regularization: Data Augmentation



Data Augmentation

Horizontal Flips



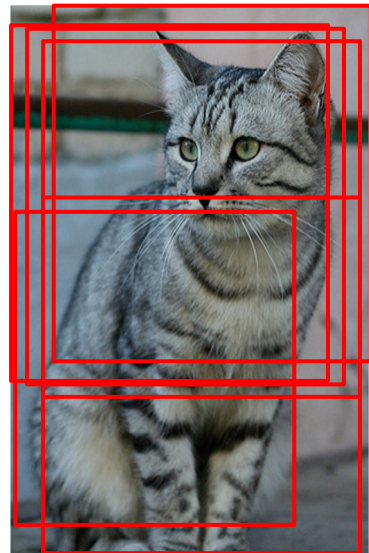
Data Augmentation

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 224×224 patch



Data Augmentation

Random crops and scales

Training: sample random crops / scales

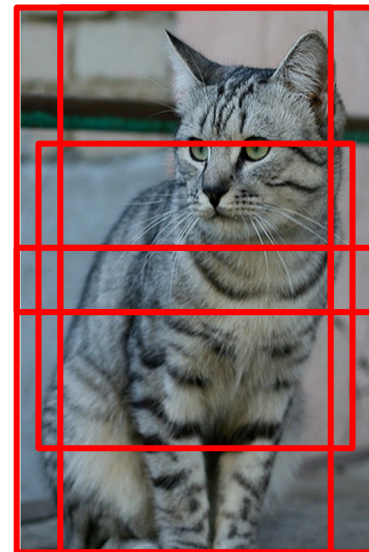
ResNet:

1. Pick random L in range $[256, 480]$
2. Resize training image, short side = L
3. Sample random 224×224 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 10 224×224 crops: 4 corners + center, + flips



Data Augmentation

Color Jitter

Simple: Randomize
contrast and brightness



Data Augmentation

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
1. Add offset to all pixels of a training image

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

Data Augmentation

Get creative for your problem!

Examples of data augmentations:

- translation
- rotation
- stretching
- shearing,
- lens distortions, ... (go crazy)

Automatic Data Augmentation

	Original	Sub-policy 1	Sub-policy 2	Sub-policy 3	Sub-policy 4	Sub-policy 5
Batch 1						
Batch 2						
Batch 3						
		ShearX, 0.9, 7 Invert, 0.2, 3	ShearY, 0.7, 6 Solarize, 0.4, 8	ShearX, 0.9, 4 AutoContrast, 0.8, 3	Invert, 0.9, 3 Equalize, 0.6, 3	ShearY, 0.8, 5 AutoContrast, 0.7, 3

Cubuk et al., "AutoAugment: Learning Augmentation Strategies from Data", CVPR 2019

Regularization: A common pattern

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Batch Normalization

Data Augmentation

Regularization: DropConnect

Training: Drop connections between neurons (set weights to 0)

Testing: Use all the connections

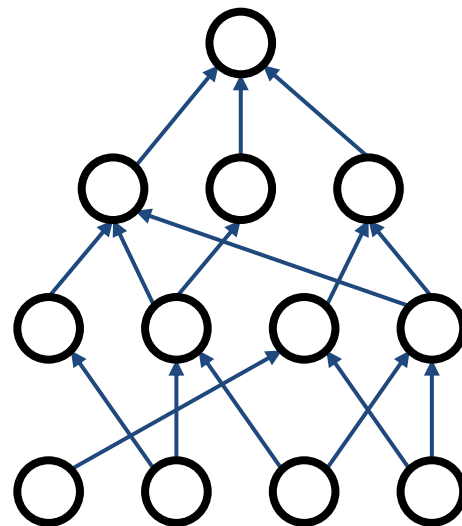
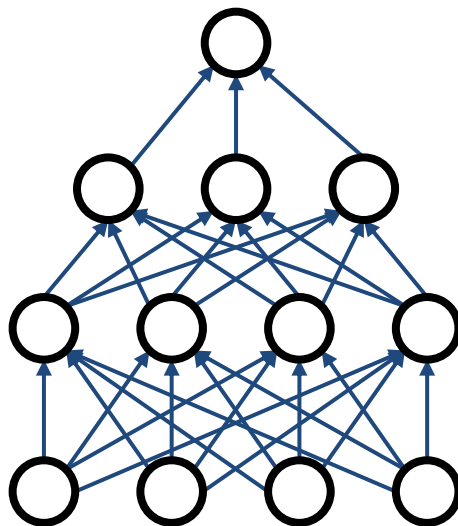
Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect



Regularization: Fractional Pooling

Training: Use randomized pooling regions

Testing: Average predictions from several regions

Examples:

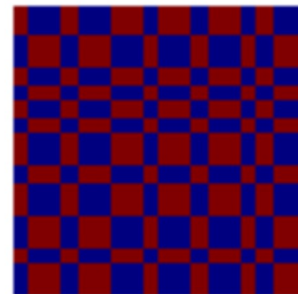
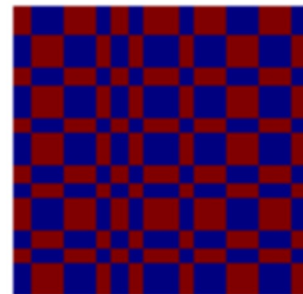
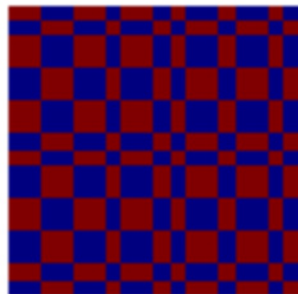
Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling



Regularization: Stochastic Depth

Training: Skip some layers in the network

Testing: Use all the layer

Examples:

Dropout

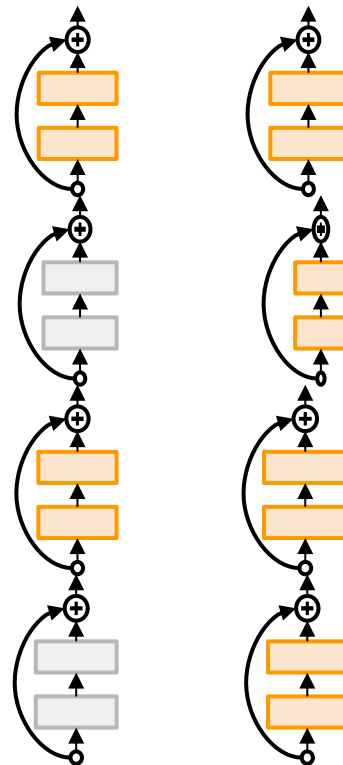
Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth (with residual)



Regularization - In practice

Training: Add random noise

Testing: Marginalize over the noise

Examples:

Dropout

Batch Normalization

Data Augmentation

DropConnect

Fractional Max Pooling

Stochastic Depth

- 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

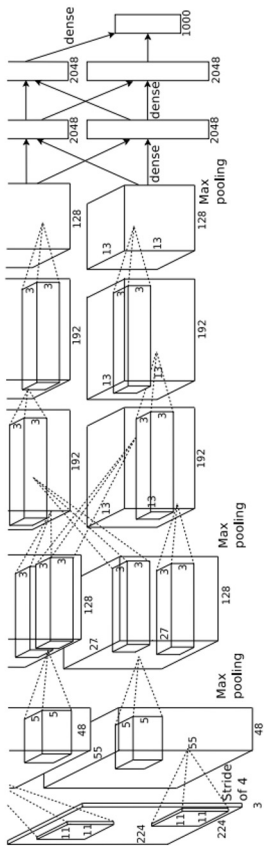
Transfer learning / Pretraining

“You need a lot of a data if you want to train/use deep neural networks”

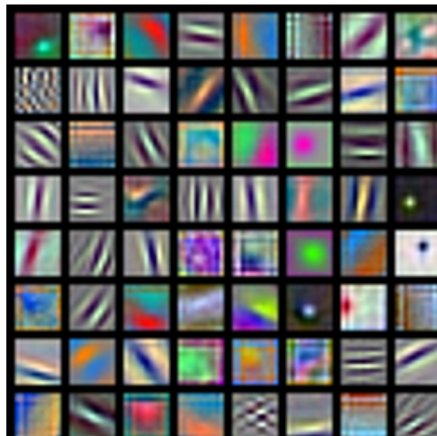
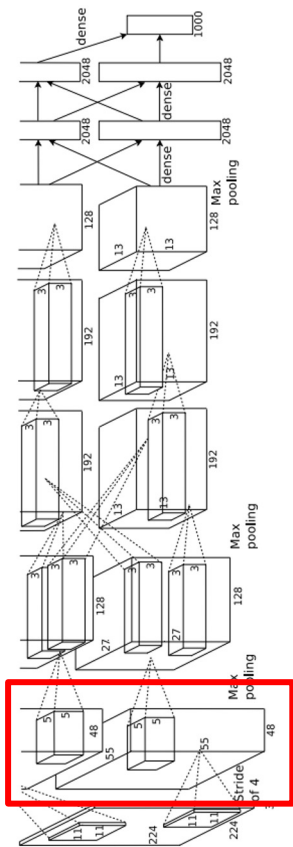
“You need a lot of a data if you want to train/use deep neural networks”

BUSTED

Transfer Learning with CNNs



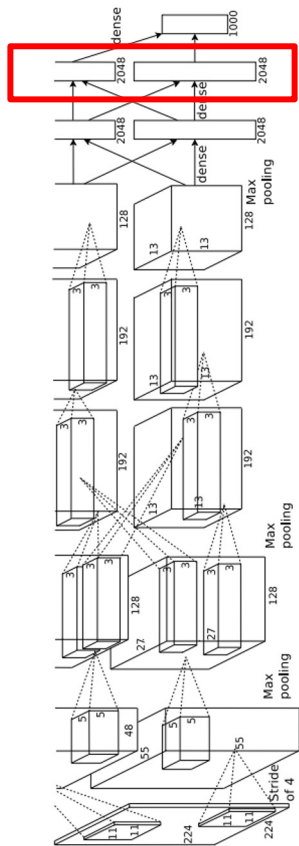
Transfer Learning with CNNs



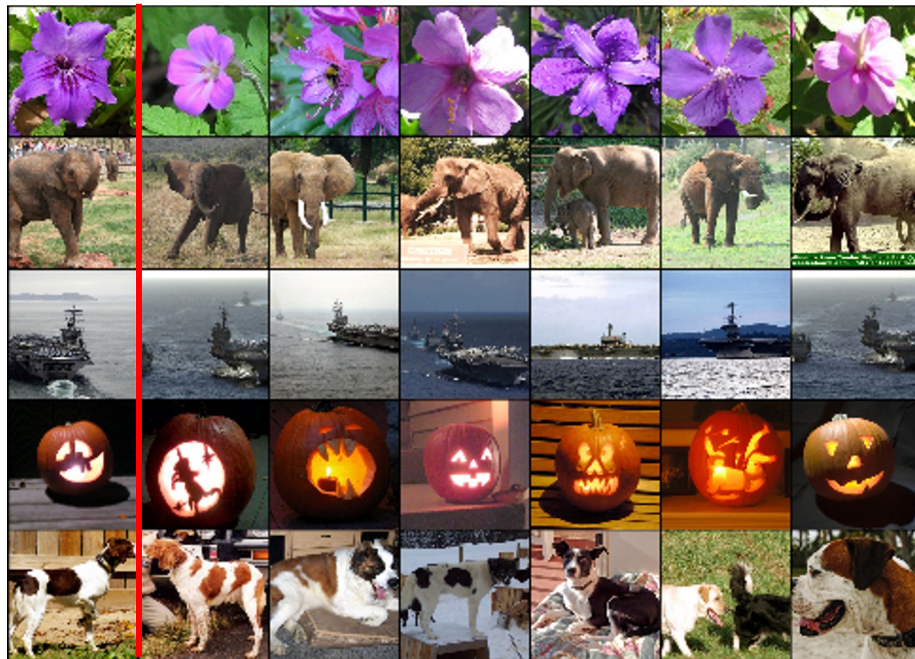
AlexNet:
64 x 3 x 11 x 11

(More on this in Lecture 13)

Transfer Learning with CNNs



Test image L2 Nearest neighbors in feature space



(More on this in Lecture 13)

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

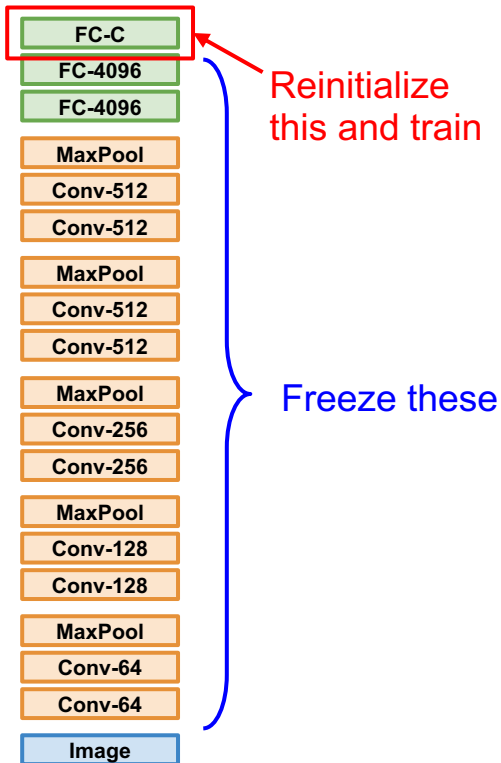


Transfer Learning with CNNs

1. Train on Imagenet



2. Small Dataset (C classes)



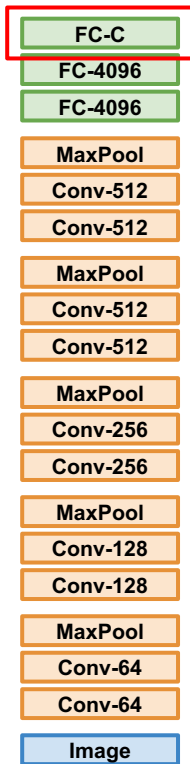
Transfer Learning with CNNs

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



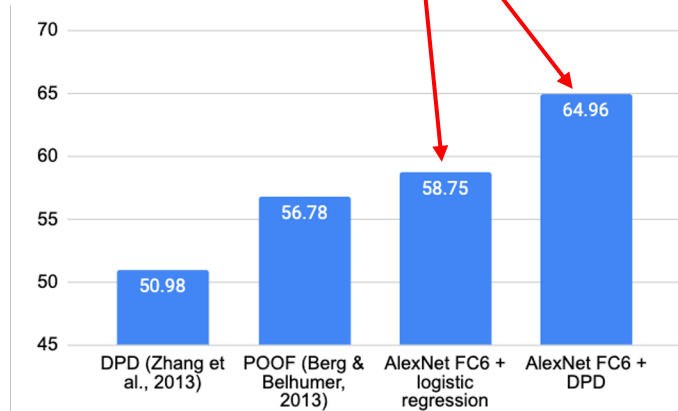
2. Small Dataset (C classes)



Reinitialize this and train

Freeze these

Finetuned from AlexNet



Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

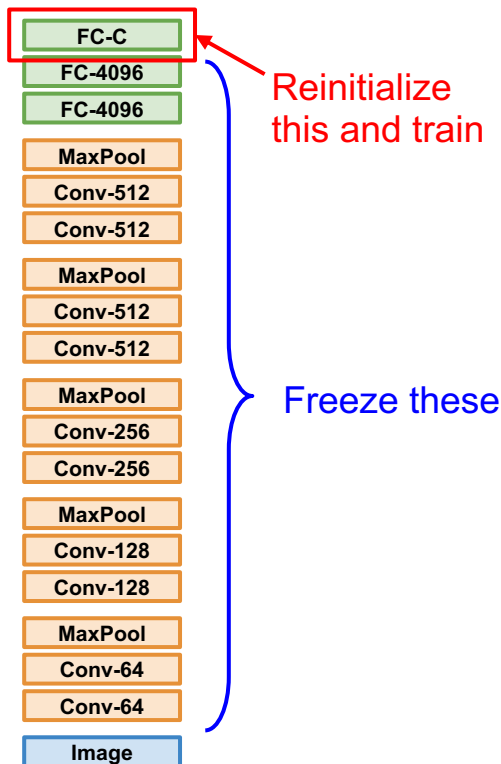
Transfer Learning with CNNs

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

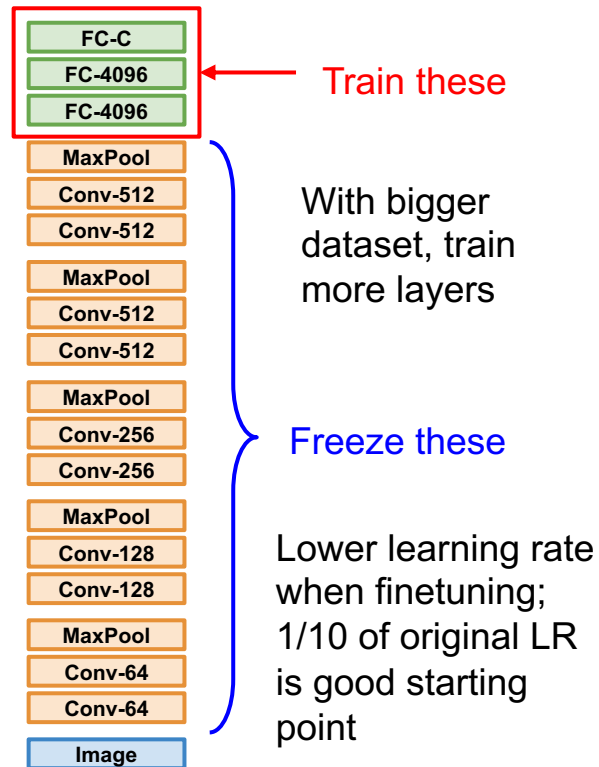
1. Train on Imagenet



2. Small Dataset (C classes)



3. Bigger dataset





Task-specific

Task-agnostic

	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?



Task-specific

Task-agnostic

	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



Task-specific

Task-agnostic

	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 is pervasive... (it's the norm, not an exception)

Object Detection (Fast R-CNN)

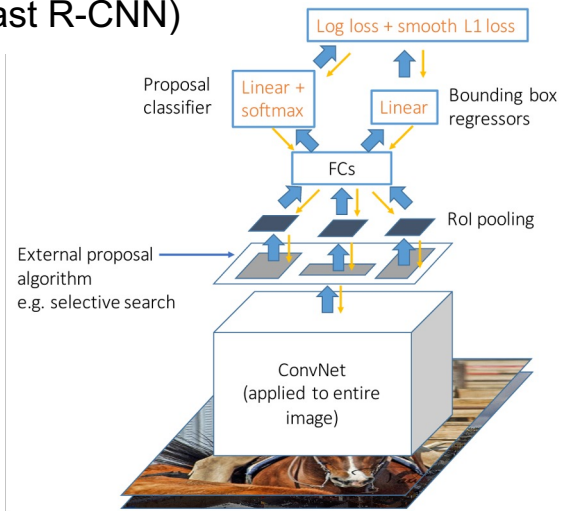
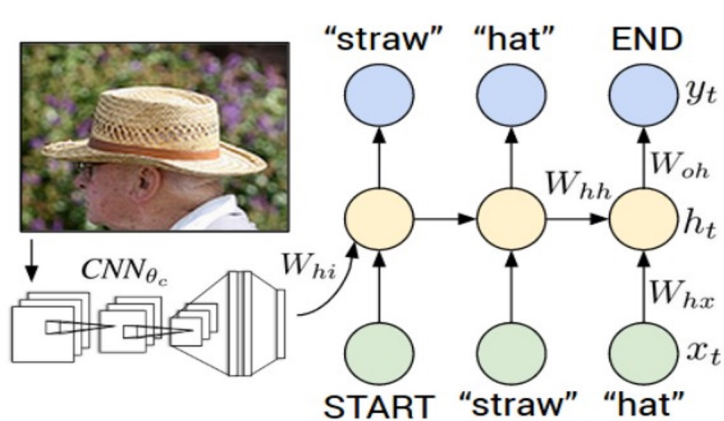
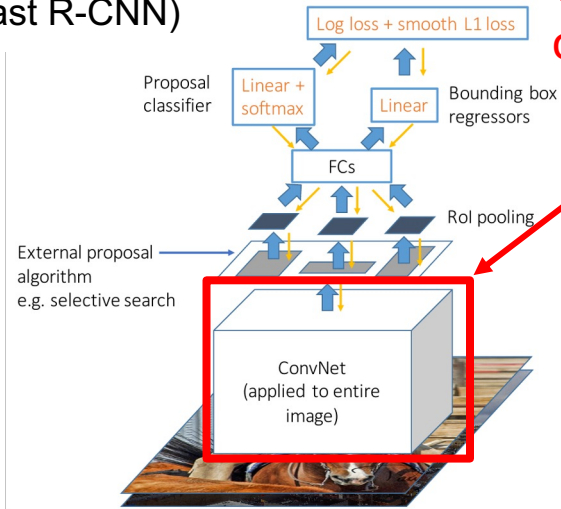


Image Captioning: CNN + RNN



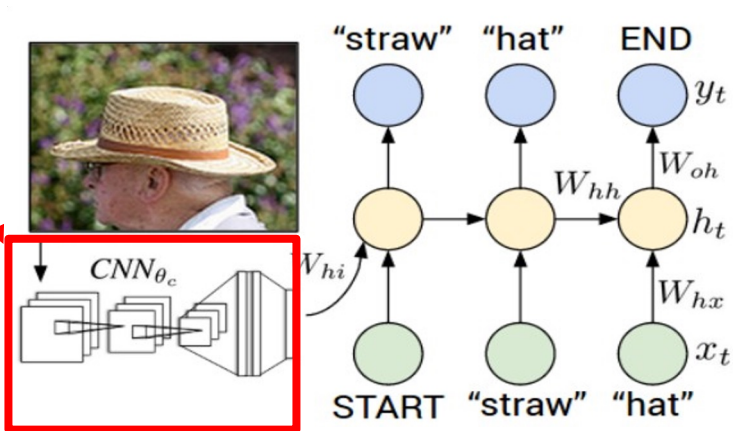
Transfer learning is pervasive... (it's the norm, not an exception)

Object Detection
(Fast R-CNN)



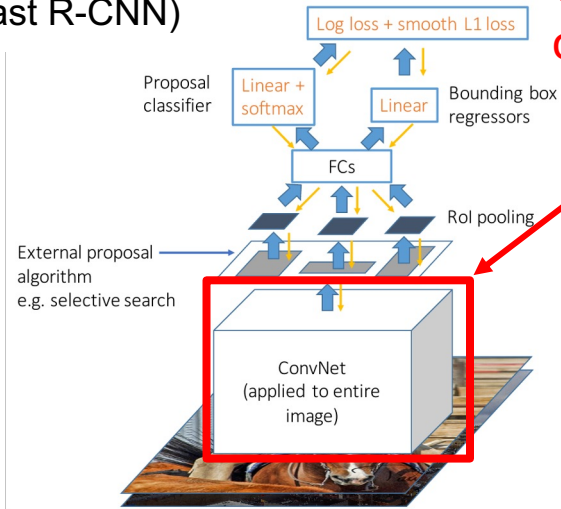
CNN pretrained
on ImageNet

Image Captioning: CNN + RNN



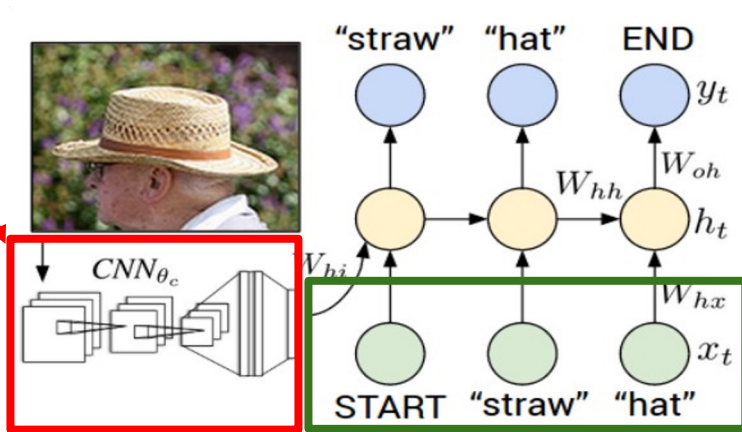
Transfer learning is pervasive... (it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained
on ImageNet

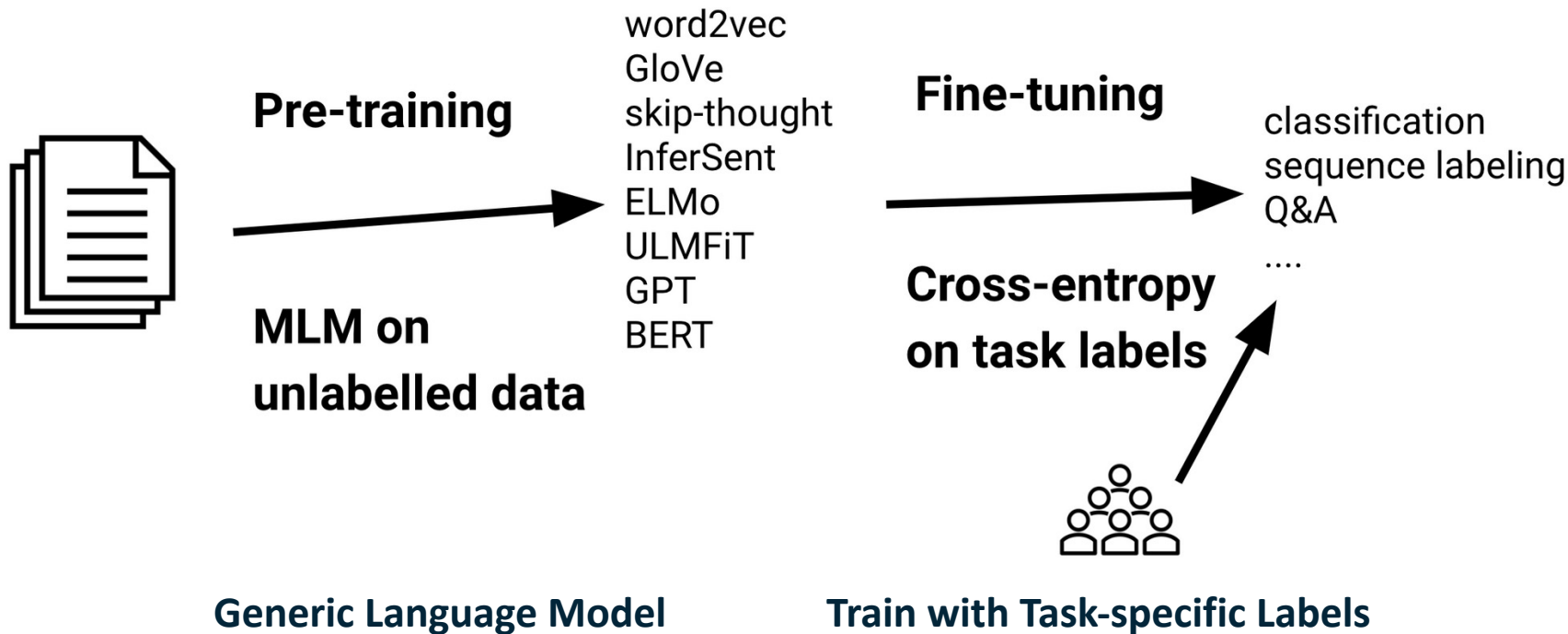
Image Captioning: CNN + RNN



Word vectors pretrained
with word2vec

Transfer learning is pervasive...

(it's the norm, not an exception)



Preview: Self-Supervised Pretraining

(pretraining tasks that do not need labels)

Example: learn to predict image transformations / complete corrupted images

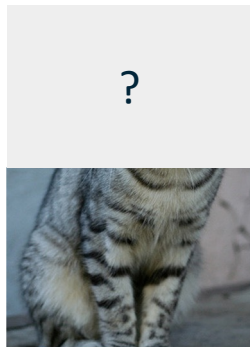
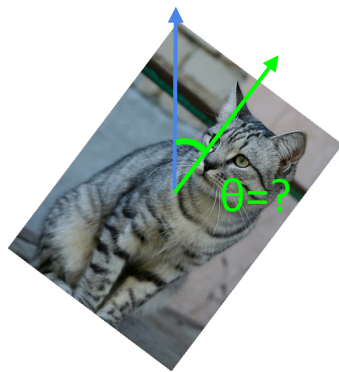
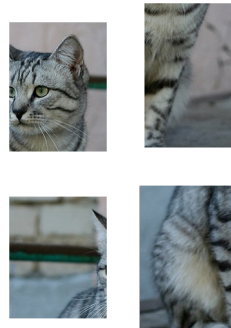


image
completion



rotation
prediction



“jigsaw puzzle”



colorization

1. Solving the pretext tasks allow the model to learn good features.
2. We can automatically generate labels for the pretext tasks.

Takeaway for your projects and beyond:

Transfer learning be like



Source: AI & Deep Learning Memes For Back-propagated Poets

Takeaway for your projects and beyond:

Have some dataset of interest but not big enough to train deep models?

1. Find a very large dataset that has similar data, train a big model there
2. Transfer learn to your dataset

Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own

TensorFlow: <https://github.com/tensorflow/models>

PyTorch (Vision): <https://github.com/pytorch/vision>

PyTorch (NLP): <https://github.com/pytorch/text>

Diagnose your training

(without tons of GPUs)

Diagnose your training

Step 1: Check initial loss

Turn off weight decay, sanity check loss at initialization
e.g. $\log(C)$ for softmax with C classes

Diagnose your training

Step 1: Check initial loss

Step 2: **Overfit a small sample**

Try to train to 100% training accuracy on a small sample of training data (~5-10 minibatches); fiddle with architecture, learning rate, weight initialization

Loss not going down? LR too low, bad initialization, bug in code or errors in training labels

Loss explodes to Inf or NaN? LR too high, bad initialization, bug in code

Diagnose your training

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~ 100 iterations

Good learning rates to try: $1e-3$, $3e-4$, $1e-4$

Diagnose your training

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.

Good weight decay to try: $1e-4$, $1e-5$, 0

Diagnose your training

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

Pick best models from Step 4, train them for longer (~10-20 epochs) without learning rate decay

Diagnose your training

Step 1: Check initial loss

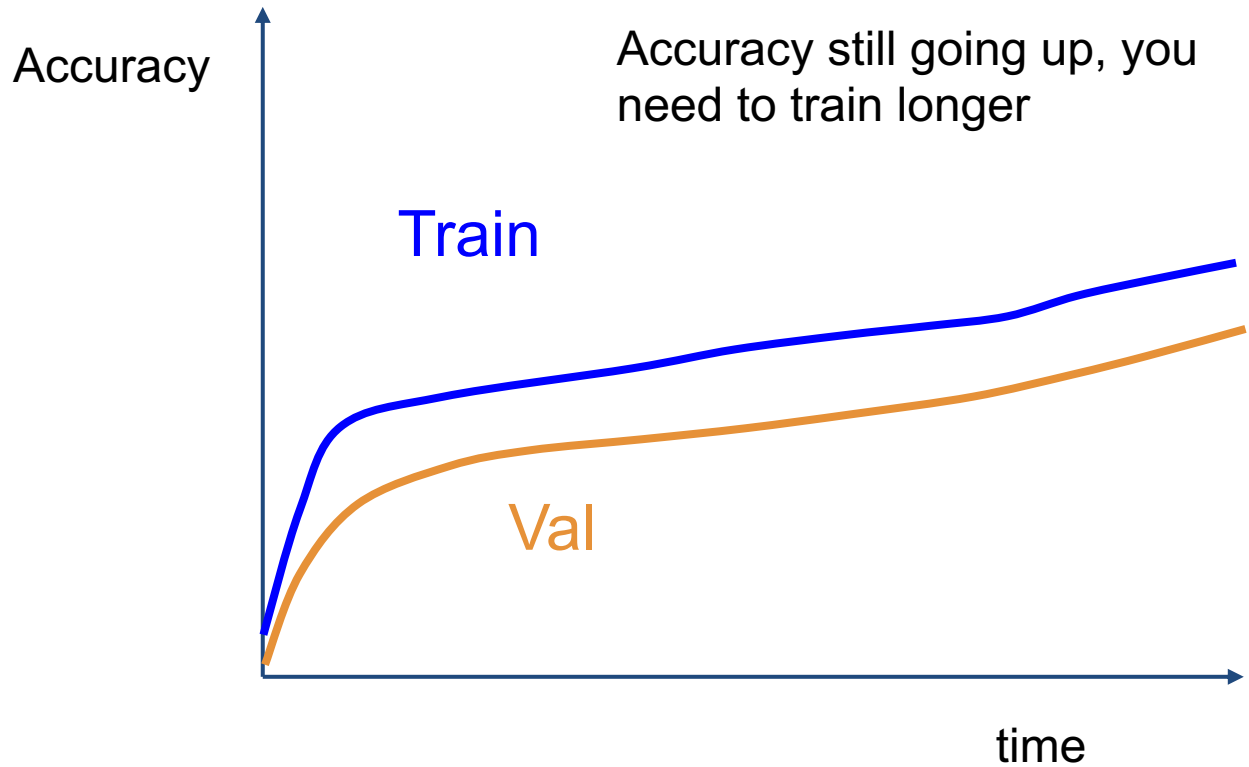
Step 2: Overfit a small sample

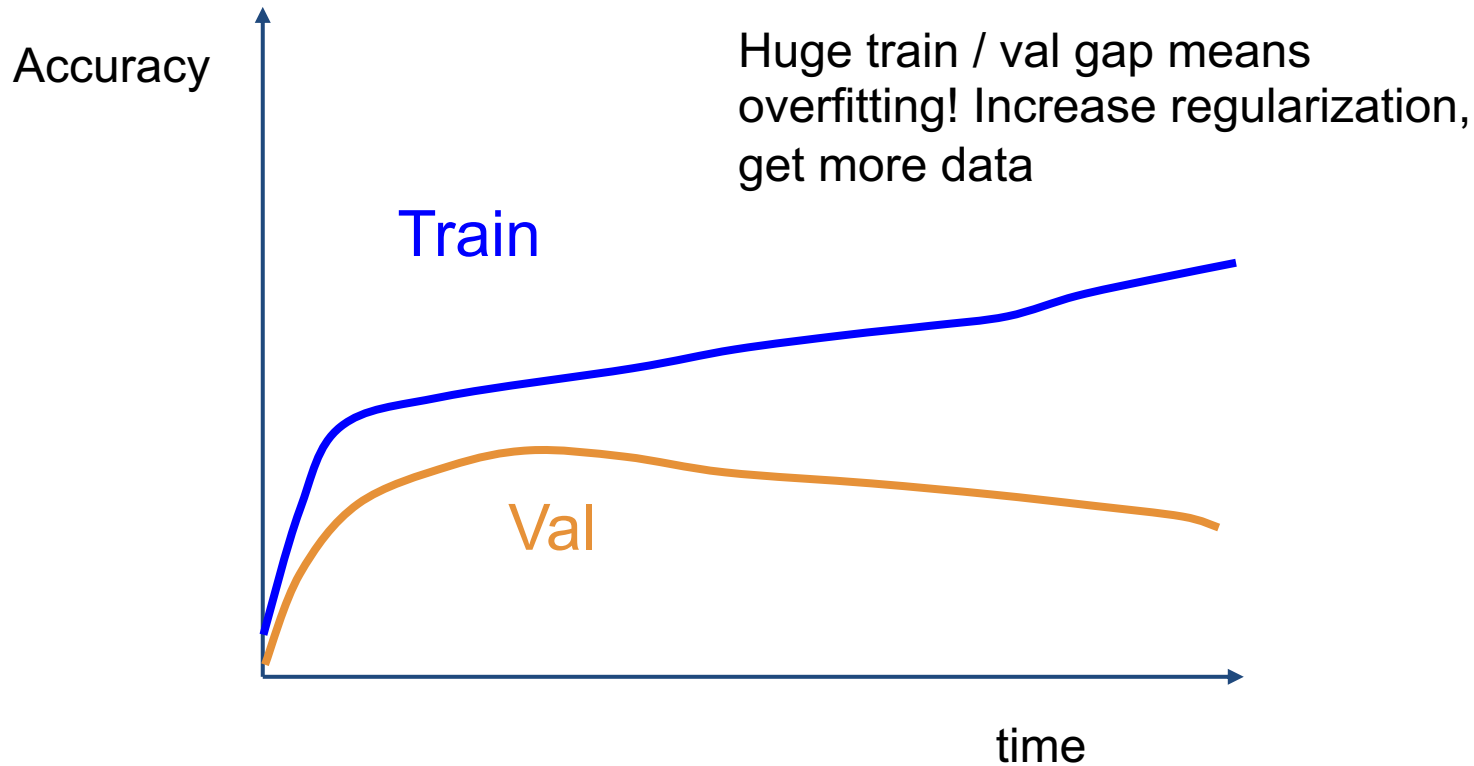
Step 3: Find LR that makes loss go down

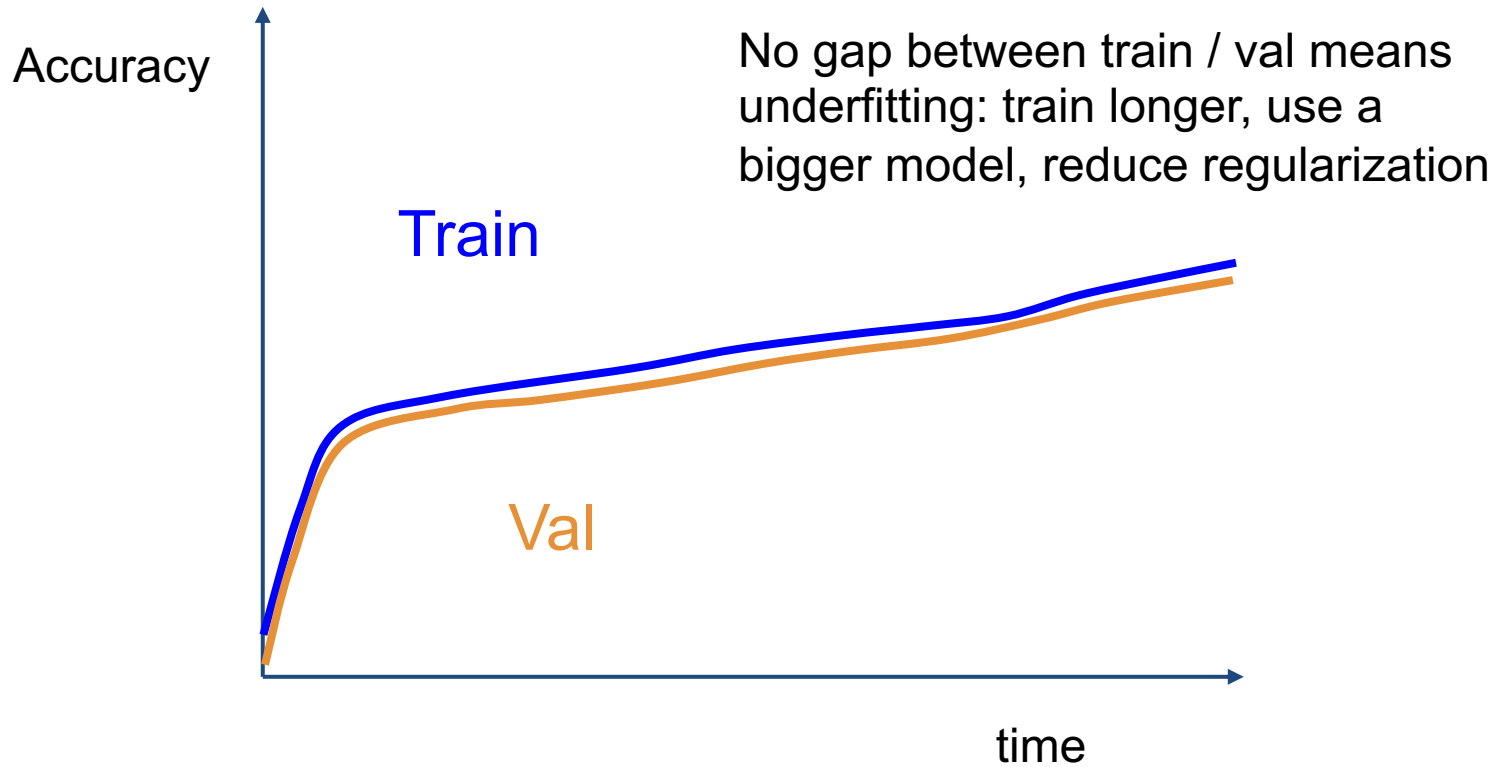
Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

Step 6: Look at loss and accuracy curves

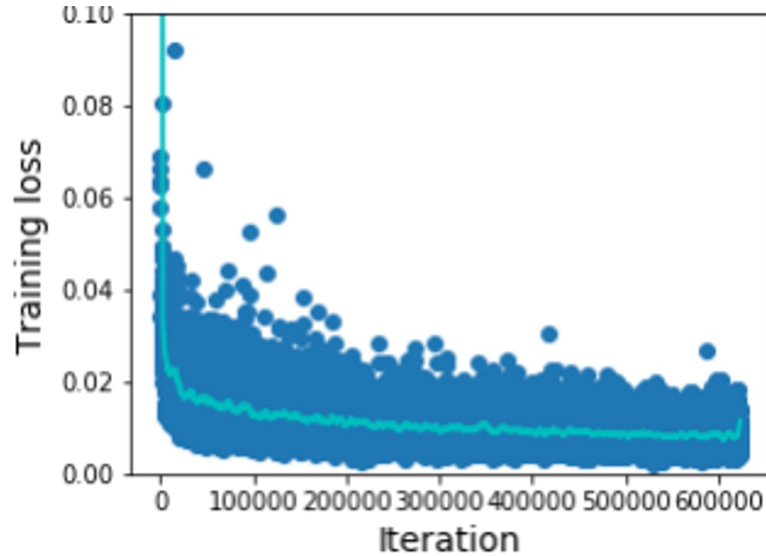




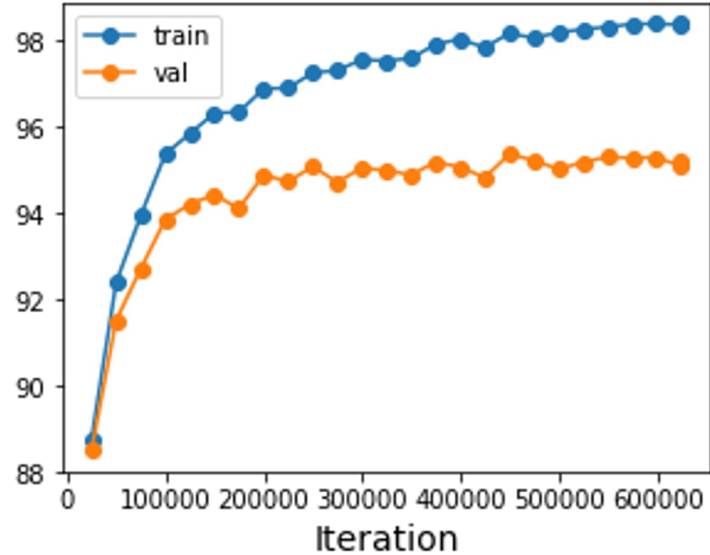


Look at learning curves!

Training Loss



Train / Val Accuracy

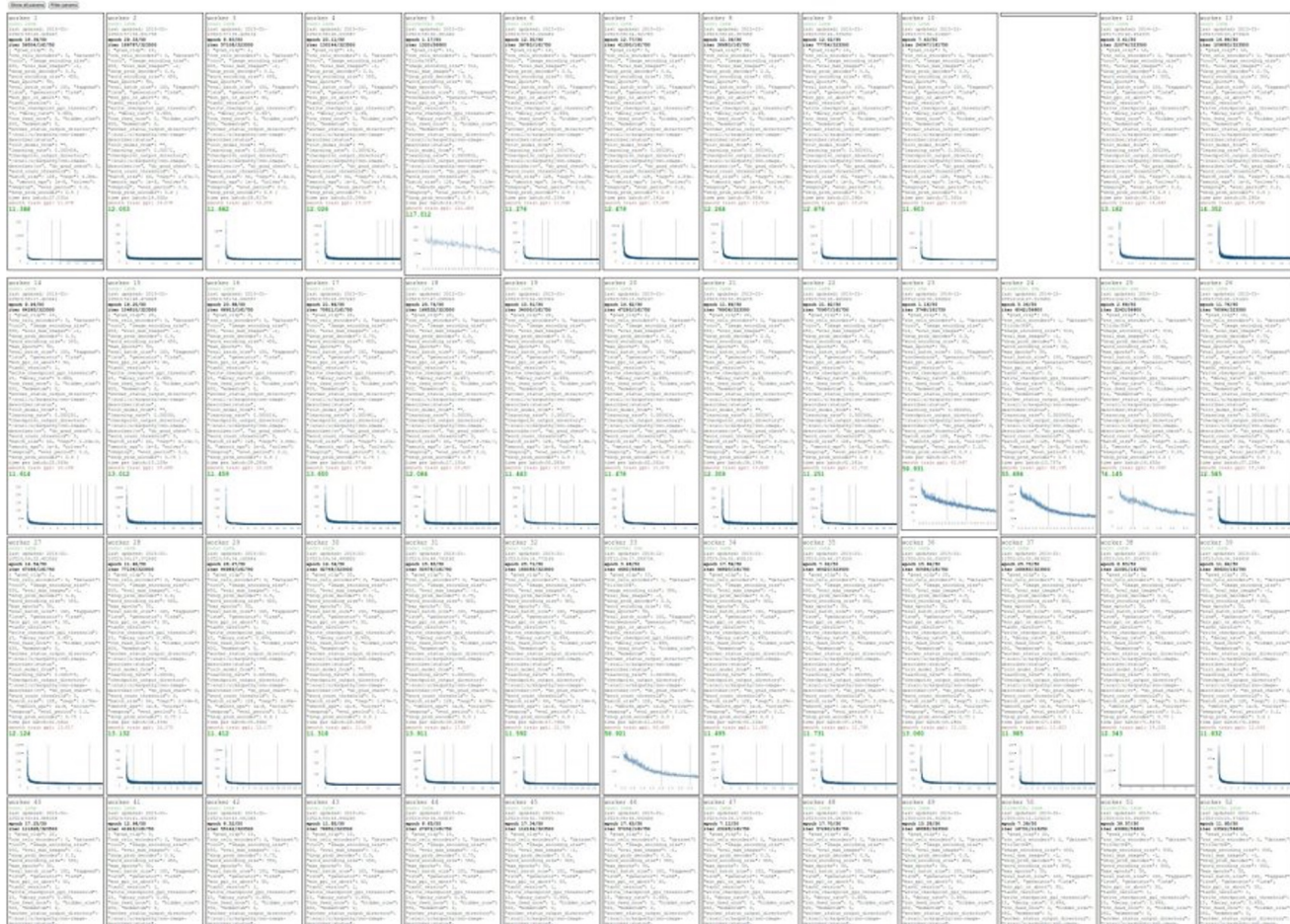


Losses may be noisy, use a scatter plot and also plot moving average to see trends better

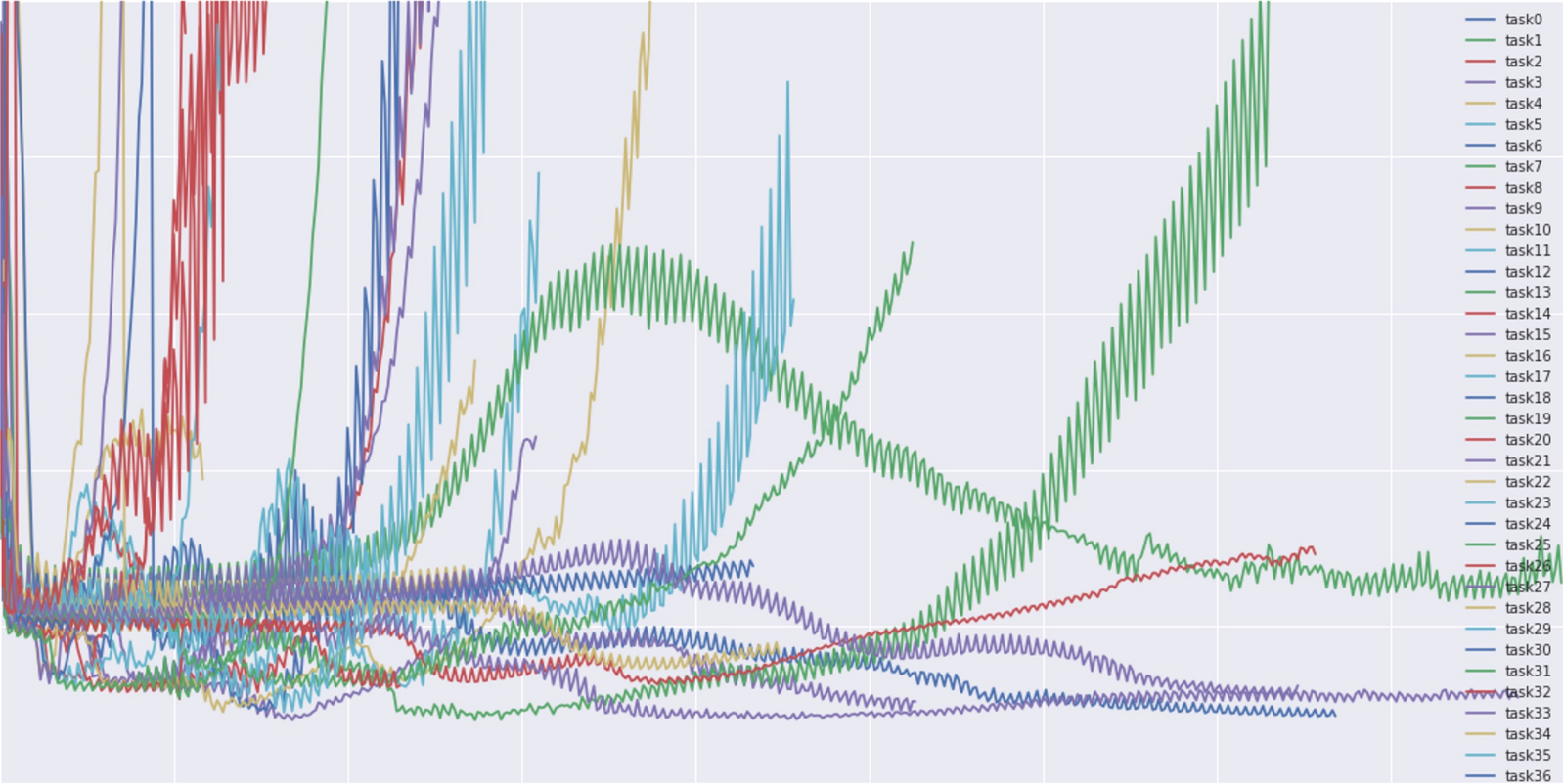
Cross-validation

We develop "command centers" to visualize all our models training with different hyperparameters

check out [weights and biases](#)



You can plot all your loss curves for different hyperparameters on a single plot



Don't look at accuracy or loss curves for too long!



Choosing Hyperparameters

Step 1: Check initial loss

Step 2: Overfit a small sample

Step 3: Find LR that makes loss go down

Step 4: Coarse grid, train for ~1-5 epochs

Step 5: Refine grid, train longer

Step 6: Look at loss and accuracy curves

Step 7: GOTO step 5

Hyperparameters to play with:

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

Random Search vs. Grid Search

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

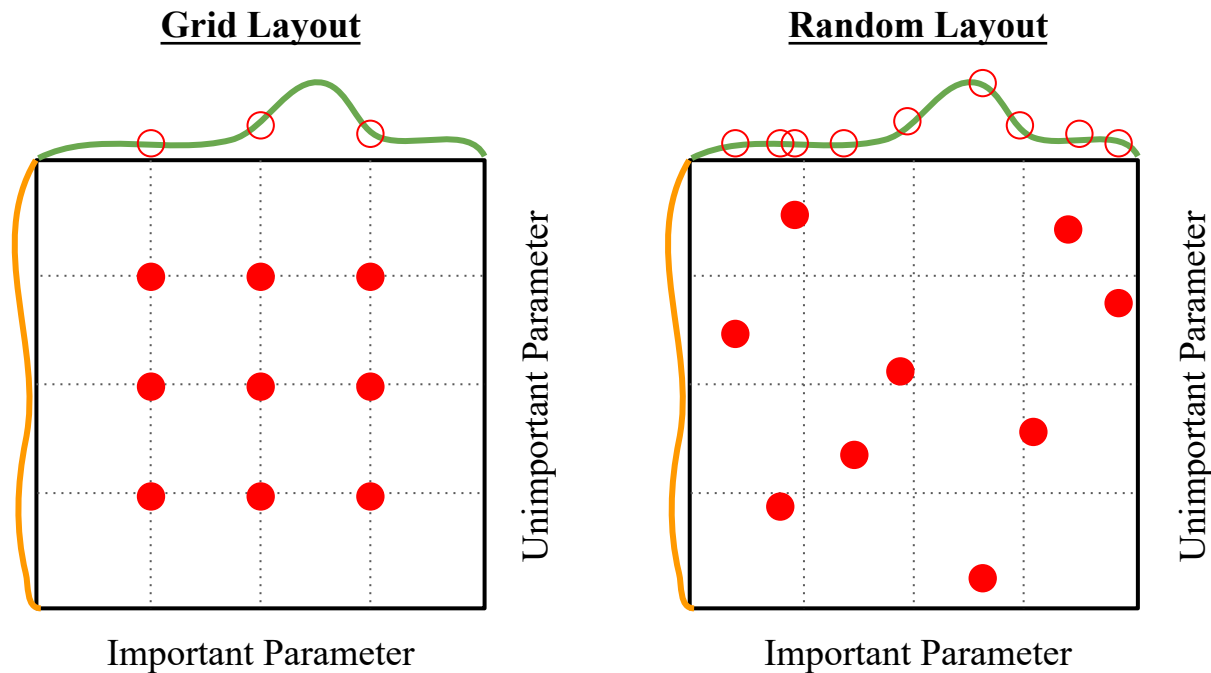


Illustration of Bergstra et al., 2012 by Shayne Longpre, copyright CS231n 2017

Summary

- Improve your training error:
 - Optimizers
 - Learning rate schedules
- Improve your test error:
 - Regularization
 - Choosing Hyperparameters

Summary

Training Deep Neural Networks

- Details of the non-linear activation functions
- Data normalization
- Weight Initialization
- Batch Normalization
- Advanced Optimization
- Regularization
- Data Augmentation
- Transfer learning
- Hyperparameter Tuning

Next time: Deep Learning Hardware
and Software