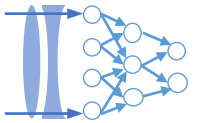


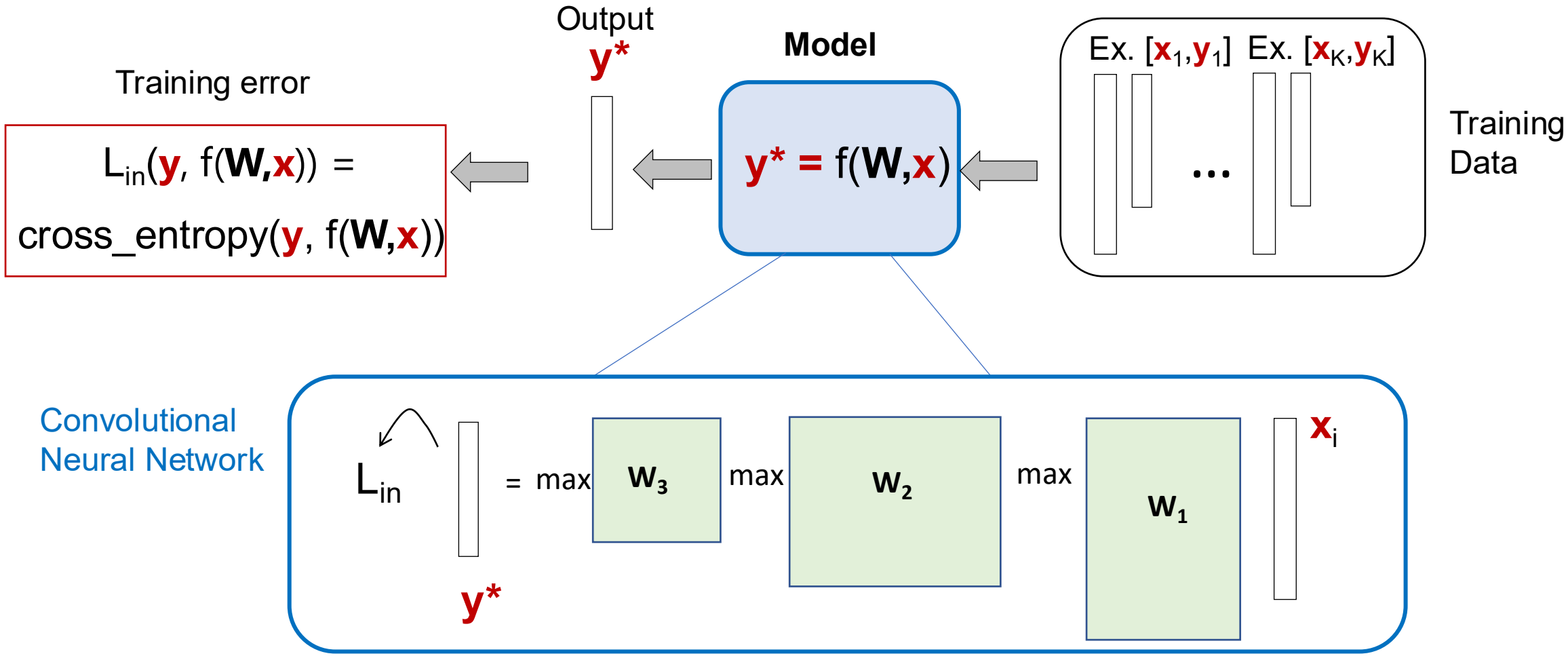
Lecture 10: Tools for your deep learning toolbox – Part III

Machine Learning and Imaging

BME 548L
Roarke Horstmeyer

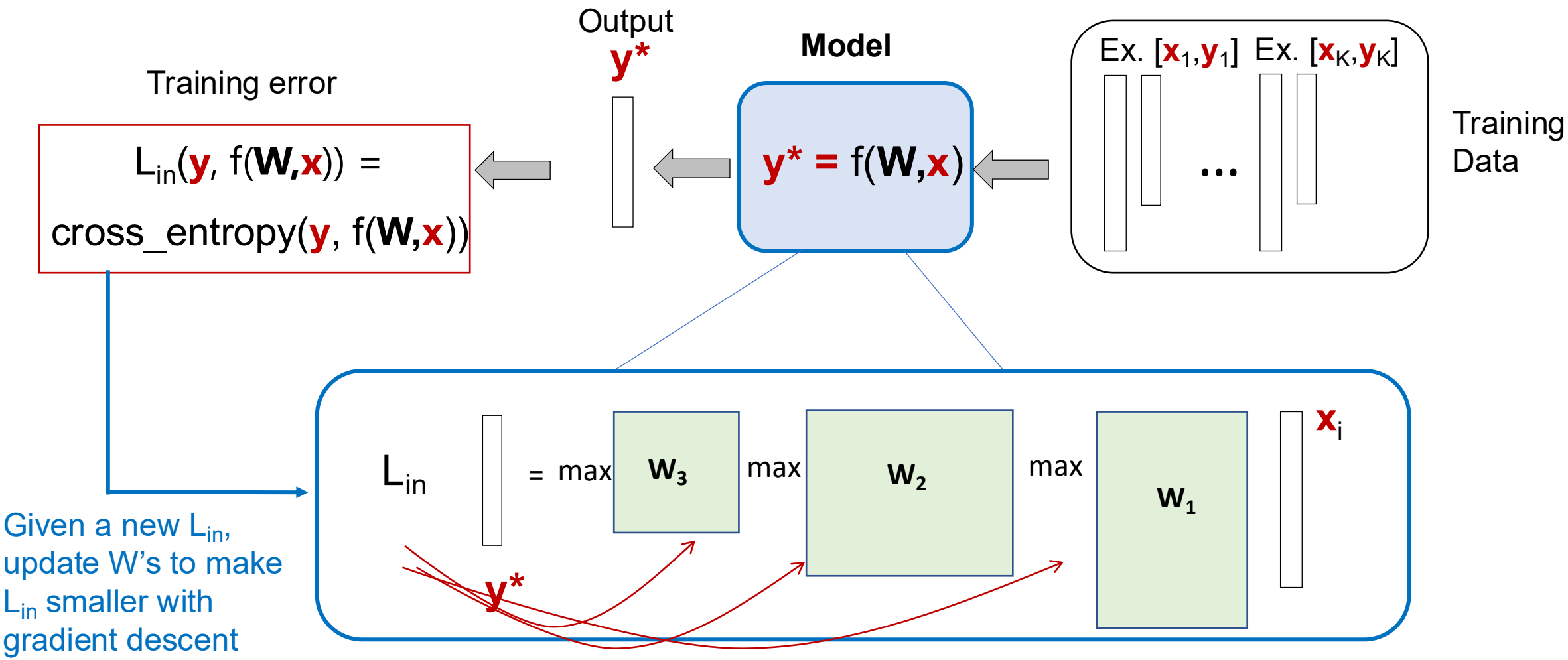
Thanks to Dr .Kevin Zhou for helping with material preparation

Our very basic convolutional neural network



Forward pass: from $\mathbf{x}_i$ and current $\mathbf{W}$'s, find L_{in}

Our very basic convolutional neural network



Next Class: Effectively achieve this with automatic differentiation (backprop)

Important components of a CNN

deep imaging

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

Common loss functions used for CNN optimization

- Cross-entropy loss function
  - Softmax cross-entropy – use with single-entry labels
  - Weighted cross-entropy – use to bias towards true pos./false neg.
  - Sigmoid cross-entropy
  - KL Divergence
- Pseudo-Huber loss function
- L1 loss loss function
- MSE (Euclidean error, L2 loss function)
- Mixtures of the above functions

Important components of a CNN

deep imaging

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

Important components of a CNN

deep imaging

CNN Architecture

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Very quick outline

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

Implementation detail #1 – method for gradient descent

```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent (SGD)

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

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

Implementation detail #1 – method for gradient descent

```
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

Stochastic Gradient Descent (SGD)

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

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Full sum expensive when N is large!

Approximate sum using a **minibatch** of examples
32 / 64 / 128 common

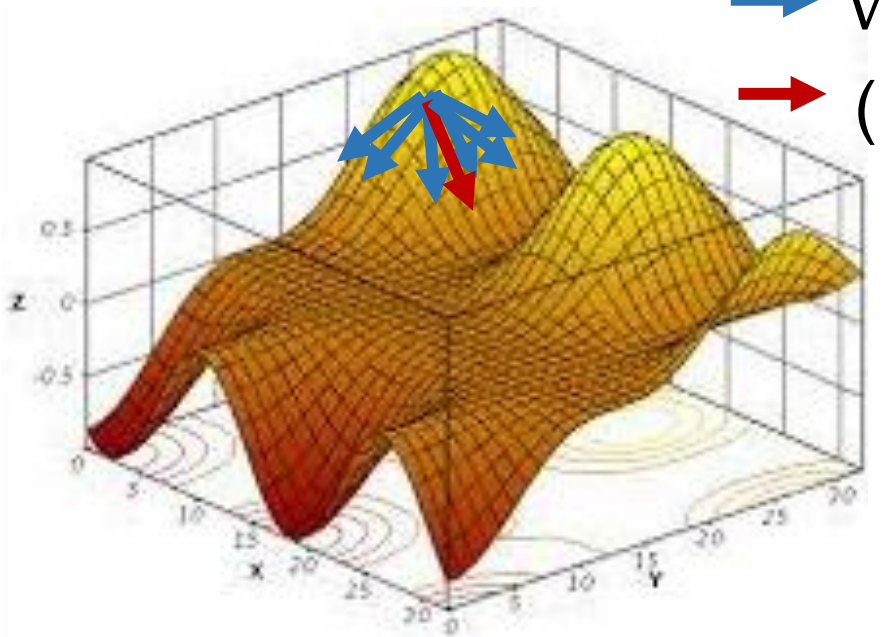
Question: Why does gradient descent still work with mini-batches?

Answer: With stochastic gradient descent, random sub-set averaging of gradients still allows one to find their way down the hill to global minimum, at least with convex and quasi-convex functions [1].

Stochastic Gradient Descent (SGD)

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

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$



[1] Bottou, Léon (1998). "Online Algorithms and Stochastic Approximations”: https://leon.bottou.org/publications/pdf/online-1998.pdf

deep imaging

Question: Why does gradient descent still work with mini-batches?

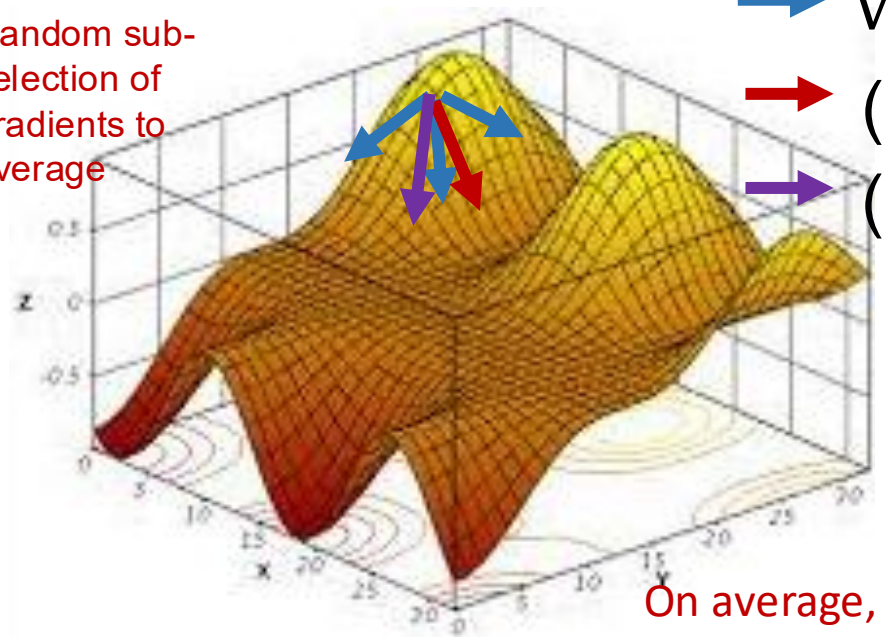
Answer: With stochastic gradient descent, random sub-set averaging of gradients still allows one to find their way down the hill to global minimum, at least with convex and quasi-convex functions [1].

Stochastic Gradient Descent (SGD)

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

$$\nabla_W L(W) = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W)$$

Random sub-selection of gradients to average



∇L_i

$(1/N)\Sigma_N \nabla L_i$

$(1/M)\Sigma_M \nabla L_i$

On average, purple arrow points in same direction as red arrow

[1] Bottou, Léon (1998). "Online Algorithms and Stochastic Approximations": https://leon.bottou.org/publications/pdf/online-1998.pdf

A variety of gradient descent solvers available in Pytorch

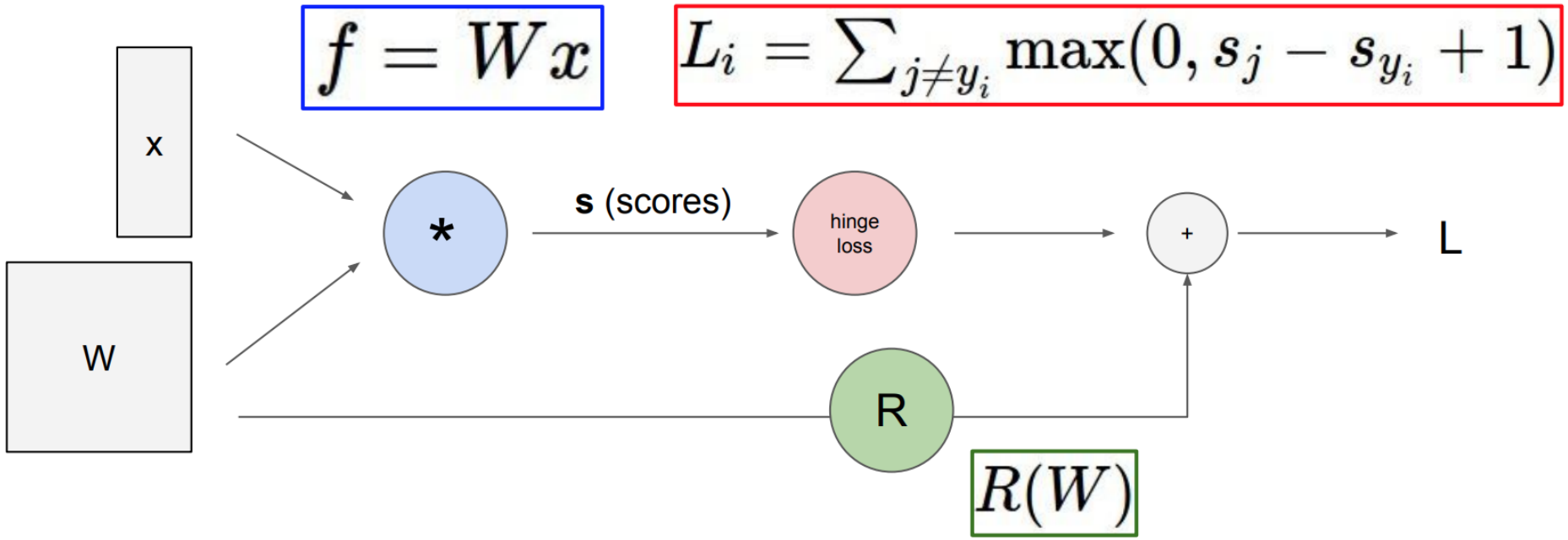
https://docs.pytorch.org/docs/stable/optim.html

Algorithms

`Adadelta`	Implements Adadelta algorithm.
`Adafactor`	Implements Adafactor algorithm.
`Adagrad`	Implements Adagrad algorithm.
`Adam`	Implements Adam algorithm.
`AdamW`	Implements AdamW algorithm, where weight decay does not accumulate in the momentum nor variance.
`SparseAdam`	SparseAdam implements a masked version of the Adam algorithm suitable for sparse gradients.
`Adamax`	Implements Adamax algorithm (a variant of Adam based on infinity norm).
`ASGD`	Implements Averaged Stochastic Gradient Descent.
`LBFGS`	Implements L-BFGS algorithm.
`Muon`	Implements Muon algorithm.
`NAdam`	Implements NAdam algorithm.
`RAdam`	Implements RAdam algorithm.
`RMSprop`	Implements RMSprop algorithm.
`Rprop`	Implements the resilient backpropagation algorithm.
`SGD`	Implements stochastic gradient descent (optionally with momentum).

Next lecture: how Tensorflow actually solves gradient descent for you

Computational Graphs and the Chain Rule!



Important components of a CNN

CNN Architecture

Architecture choices

- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Optimization choices

Other specifics: Variable Initialization, augmentation, batch normalization, dropout, gradient descent params.

The rest of this lecture: final details about deep CNN implementation

Let's view some code!

deep imaging

CNN Architecture

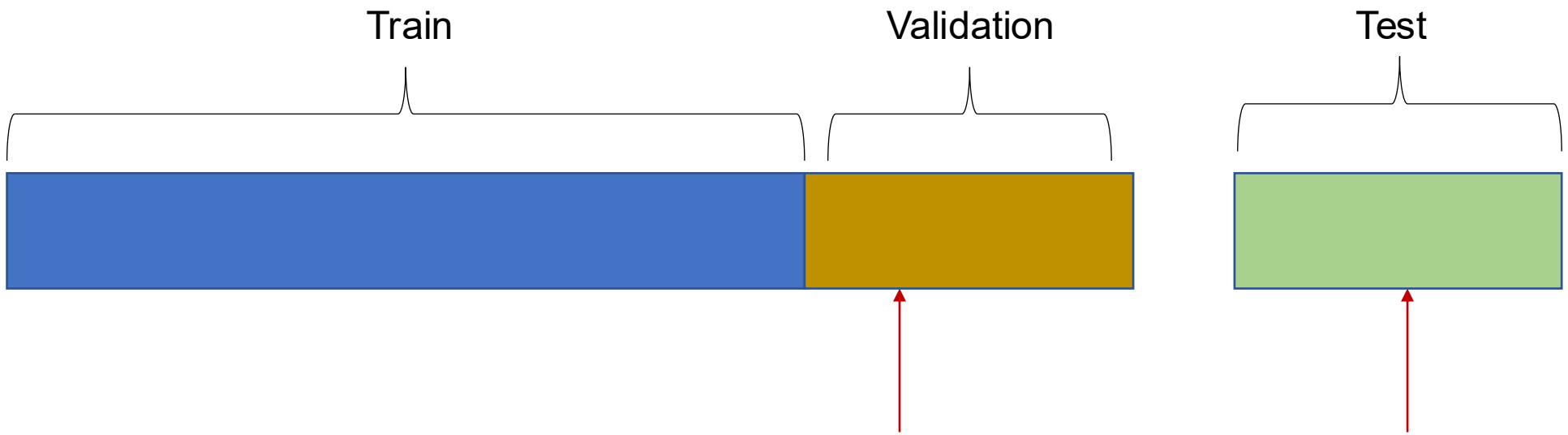
- CONV size, stride, pad, depth
- ReLU & other nonlinearities
- POOL methods
- # of layers, dimensions per layer
- Fully connected layers

Loss function & optimization

- Type of loss function
- Regularization
- Gradient descent method
- SGD batch and step size

Other specifics: Pre-processing, initialization, dropout, batch normalization, augmentation

Training dataset, test dataset and validation dataset



Use to evaluate while tuning hyperparameters

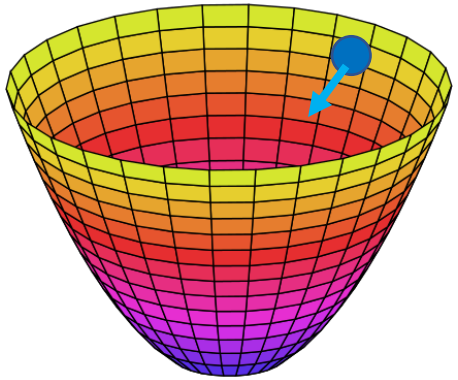
- effect will creep into model as you continue to use it

Final test set is always separate!
Don't touch until the end!

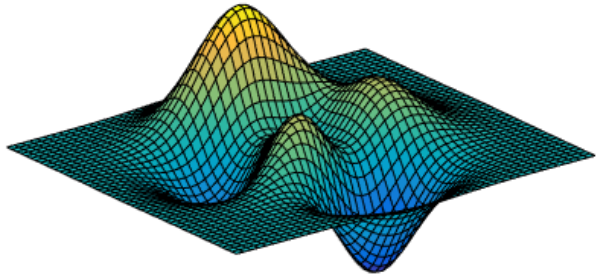
Weights initialization

- Need to start somewhere – typically best to use an appropriate random guess

Convex problem: doesn't really matter where you start



Non-convex problem: certainly matters, but you don't know where is best…



- Need to start somewhere – typically best to use an appropriate random guess sampled from a Gaussian distribution

Weights initialization

- Often it is helpful to take variance of weights into account
  - Having very large and very small weights leads to instabilities

- Desire: variance of inputs (x) remain unchanged as they transfer through network

Weights initialization

- Often it is helpful to take variance of weights into account
  - Having very large and very small weights leads to instabilities

- Desire: variance of inputs (x) remain unchanged as they transfer through network

$$\mathbf{y} = \mathbf{w}^T\mathbf{x}$$

$$\text{var}(\mathbf{y}) = \text{var}(\mathbf{w}^T\mathbf{x}) = \text{var}(w_1x_1 + \ldots w_Nx_N) = N\ \text{var}(w_1x_1) \quad \text{(IID)}$$

$$\text{var}(wx) = E(w)^2\text{var}(x) + E(x)^2\text{var}(w) + \text{var}(w)\text{var}(x) = \text{var}(w)\text{var}(x)$$

Weights initialization

- Often it is helpful to take variance of weights into account
  - Having very large and very small weights leads to instabilities

- Desire: variance of inputs (x) remain unchanged as they transfer through network

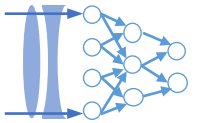
$$\mathbf{y} = \mathbf{w}^T\mathbf{x}$$

$$\text{var}(\mathbf{y}) = \text{var}(\mathbf{w}^T\mathbf{x}) = \text{var}(w_1x_1 + \ldots w_Nx_N) = N\ \text{var}(w_1x_1) \quad \text{(IID)}$$

$$\text{var}(wx) = E(w)^2\text{var}(x) + E(x)^2\text{var}(w) + \text{var}(w)\text{var}(x) = \text{var}(w)\text{var}(x)$$

$$\text{var}(y) = N\ \text{var}(w)\text{var}(x)$$

$$\text{var}(y) = \text{var}(x) \text{ when } \text{var}(w) = 1/N$$ **Xavier Initialization**



Data augmentation

- Machine learning is data-driven – the more data, the better!
- Nothing beats collecting more data, but that can be expensive and/or time consuming
- Data augmentation is the next best thing, and it's free!

deep imaging

Data augmentation one image at a time

Still a cat?

Flip left/right

Random rotation

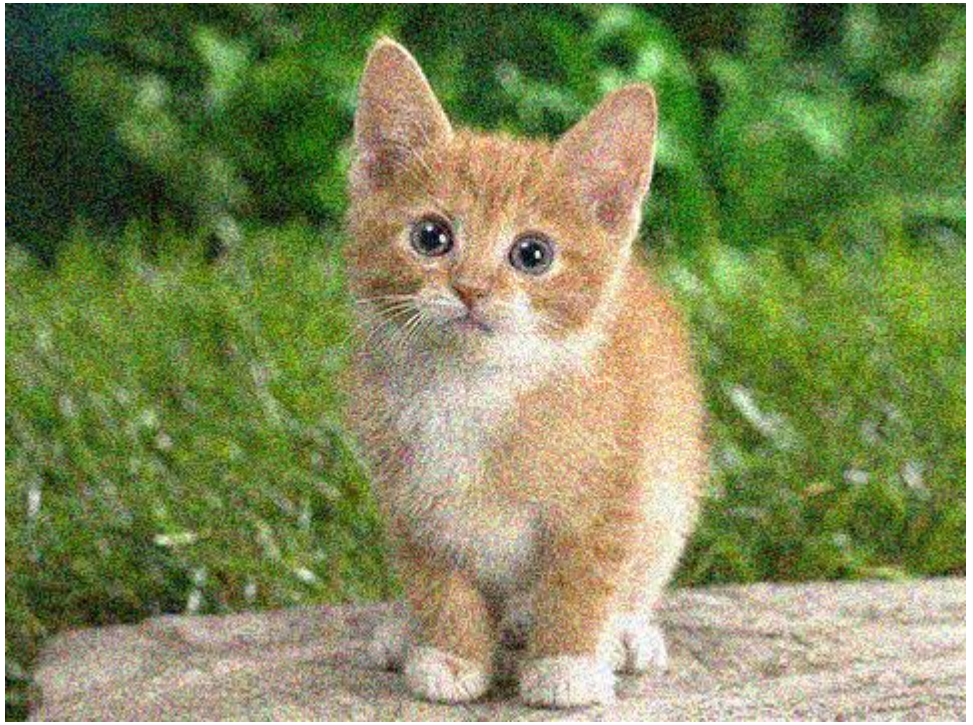
Still a cat?

Flip up/down

Random affine transformation

Still a cat?

Change color scheme



Add random noise

Data augmentation

- Basic idea: to simulate variation that you might actually see in real life
- It's a form of regularization
- Not an exact science, but try it out – it's free!

Normalization: data preprocessing

- If you use sigmoid activations, inputs that are too large could saturate them at early layers (vanishing gradient problem)
- Good practice to normalize your inputs
  - e.g. normalize to 0 mean, 1 variance; normalize to between 0 and 1 or -1 and 1
  - $X_i \leftarrow \frac{X_i - \mu}{\sigma}$
- Depending on the dataset, normalization can be done per instance or across entire dataset
  - Datasets with instances that have inconsistent ranges, although theoretically not a problem, in practice could speed up learning

Generalizing normalization to hidden layers

deep imaging

- Batch normalization
- Layer normalization
- Instance normalization
- Group normalization

- All of these normalize hidden layers to 0 mean and 1 variance, but these means and variances are computed across different dimensions
  - $X_i \leftarrow \frac{X_i - \mu}{\sigma}$

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

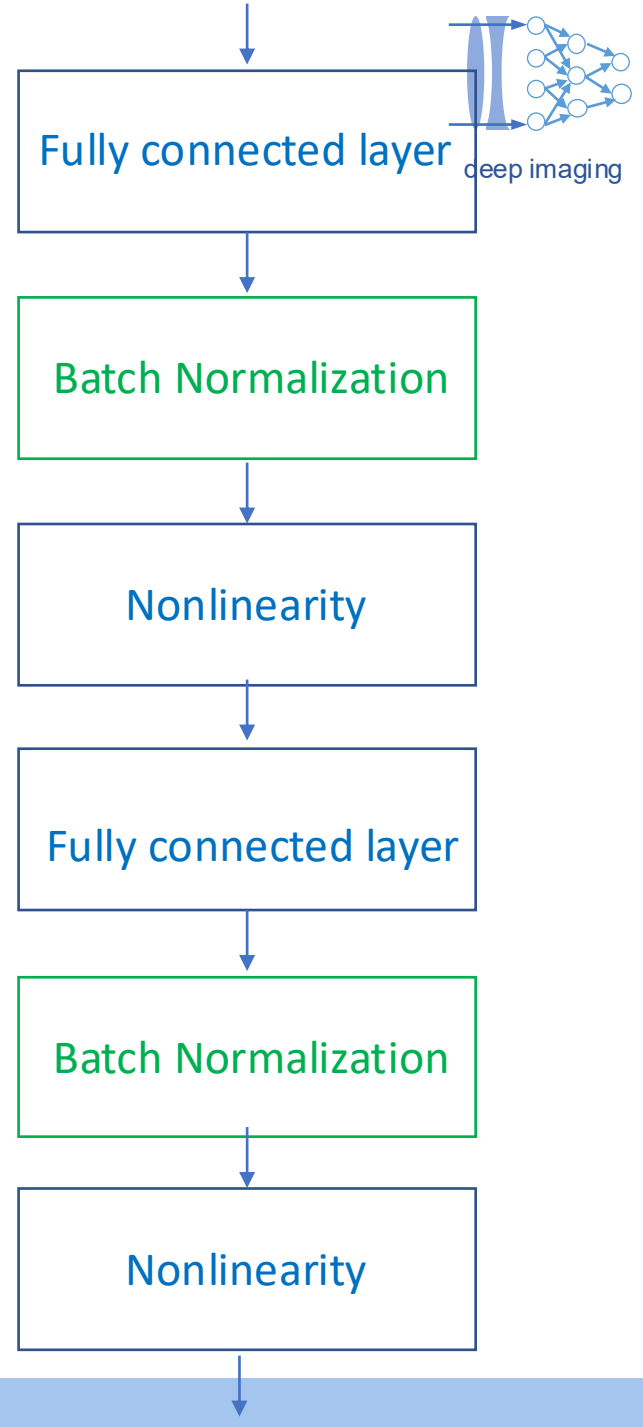
Batch normalization (BN)

- Before BN, training very deep networks was hard
  - If using sigmoid activations, large weights could result in saturation
  - Updating earlier layers' weights causes the distribution of weights in later layers to shift – the *internal covariate shift*
- To address this covariate shift, BN "resets" the layer it is applied to by normalizing to 0 mean, 1 variance
  - Mean and variance are computed over the batch at the current iteration

Batch normalization update for inputs x:

$$x'(i) = (x(i) - E[x(i)]) / STD[x(i)]$$

- Mean subtract
- Normalize by standard deviation



Problems

- Normalizing to 0 mean 1 variance reduces the expressivity of the layer
  - E.g., if using a sigmoid activation, you're stuck in the linear regime
- Solution: reintroduce mean (β) and standard deviation (γ) parameters:
  - $X_i \leftarrow \frac{X_i - \mu}{\sigma}$ #normalize
  - $X_i \leftarrow \gamma X_i + \beta$ #new mean and standard deviations
  - γ and β are trainable parameters

- Accuracy of μ and σ depends on the batch size being large

Other hidden layer normalizations (for CNNs)

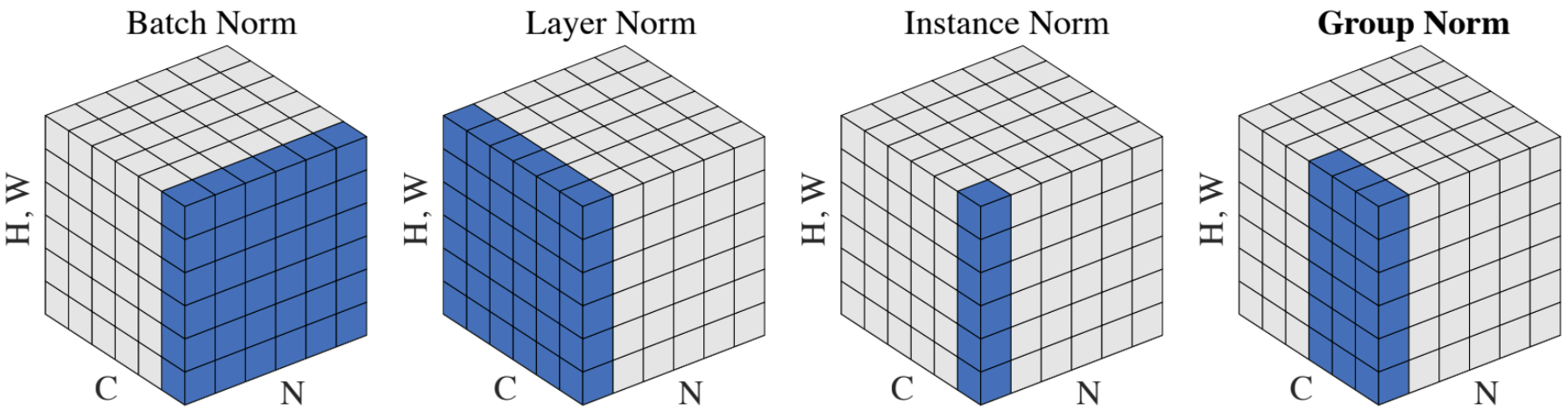
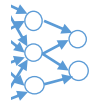


Figure 2. **Normalization methods**. Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.

https://nealjean.com/ml/neural-network-normalization/

Journal of Machine Learning Research 15 (2014) 1929-1958 Submitted 11/13; Published 6/14



aging

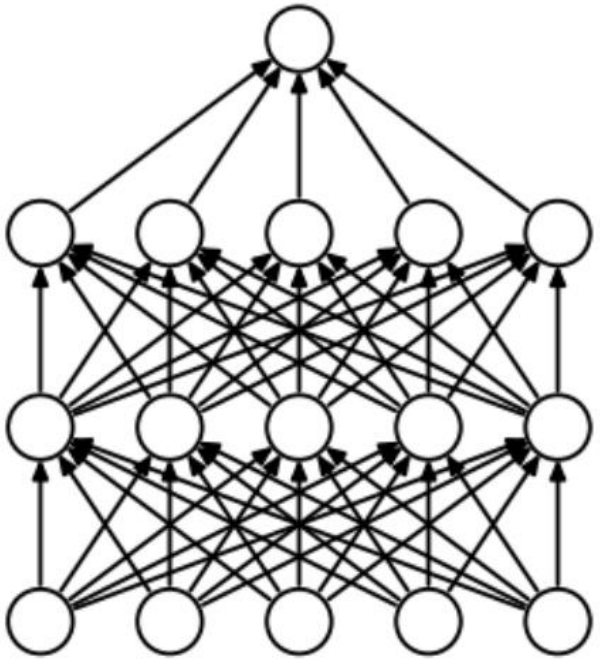
Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava NITISH@CS.TORONTO.EDU
Geoffrey Hinton HINTON@CS.TORONTO.EDU
Alex Krizhevsky KRIZ@CS.TORONTO.EDU
Ilya Sutskever ILYA@CS.TORONTO.EDU
Ruslan Salakhutdinov RSALAKHU@CS.TORONTO.EDU
Department of Computer Science
University of Toronto
10 Kings College Road, Rm 3302
Toronto, Ontario, M5S 3G4, Canada.

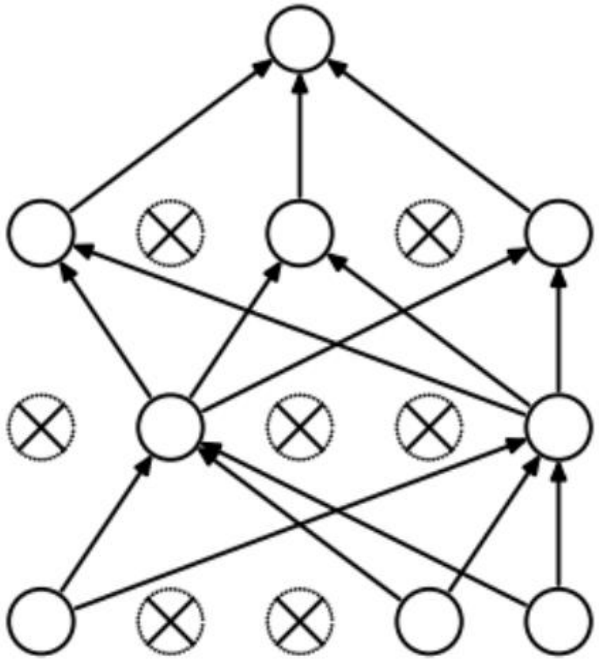
Editor: Yoshua Bengio

Dropout

- At each train iteration, randomly delete a fraction p of the nodes
- Prevents neurons from being lazy
- A form of model averaging
- (related: DropConnect – drop the connections instead of nodes)



(a) Standard Neural Net



(b) After applying dropout.

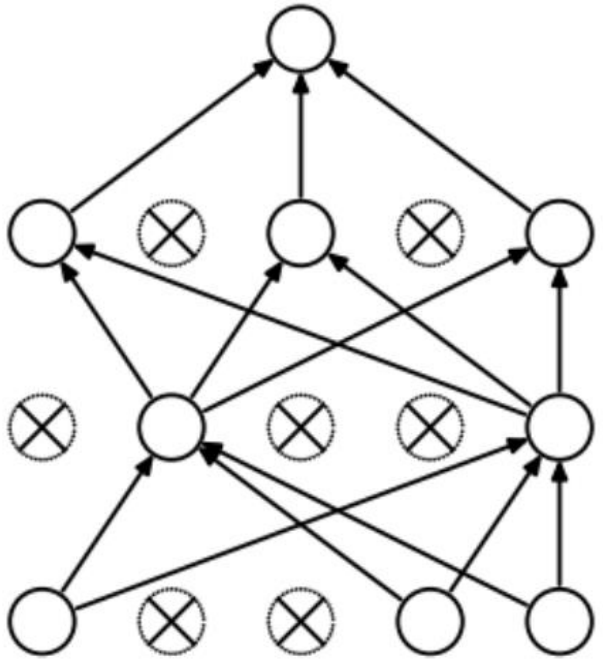
https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

Dropout

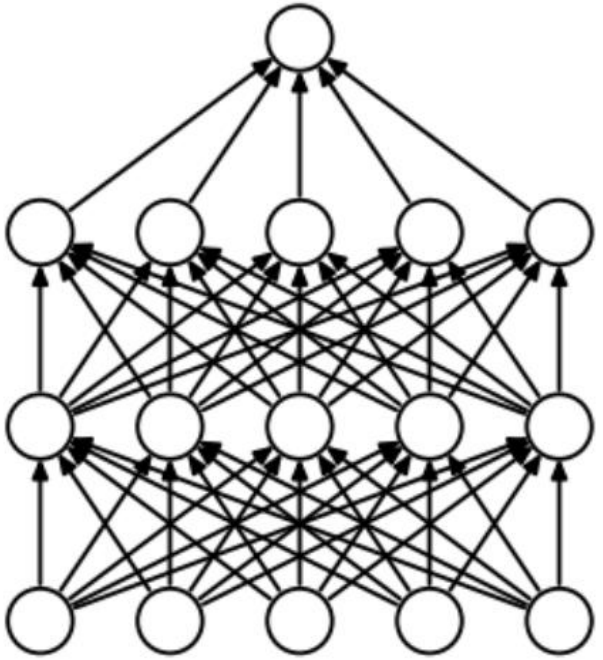
- Only one hyperparameter “rate” = p, the expected fraction of neurons to drop in a given layer
- In Pytorch:
  - nn.Dropout(0.5)
- Common practices:
  - Set p=0.5
  - Apply to fully connected layers, not convolutional layers (already sparse)

Dropout training vs testing

- Training: at a given layer, each node is dropped with probability p
- Testing: multiply the outgoing weights by 1-p (*weight scaling inference rule*)
- As a model averaging technique, other possibilities exist



Training
(each node dropped with probability)



Testing
(all weights multiplied by 1-p)

https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

Regularization: A common pattern

Training: Add some kind of randomness, z:

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

Testing: Average out randomness (sometimes approximate)

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

Example: Batch Normalization

Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Slide from http://cs231n.stanford.edu/

Regularization: A common pattern

Training: Add some kind of randomness, z:

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

Testing: Average out randomness (sometimes approximate)

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

Example: Batch Normalization

Training: Normalize using stats from random minibatches

Testing: Use fixed stats to normalize

Obvious examples: Dropout, data augmentation

Advanced examples: DropConnect, Fractional Max Pooling, Stochastic Depth

Wan et al, “Regularization of Neural Networks using DropConnect”, ICML 2013
Graham, “Fractional Max Pooling”, arXiv 2014
Huang et al, “Deep Networks with Stochastic Depth”, ECCV 2016

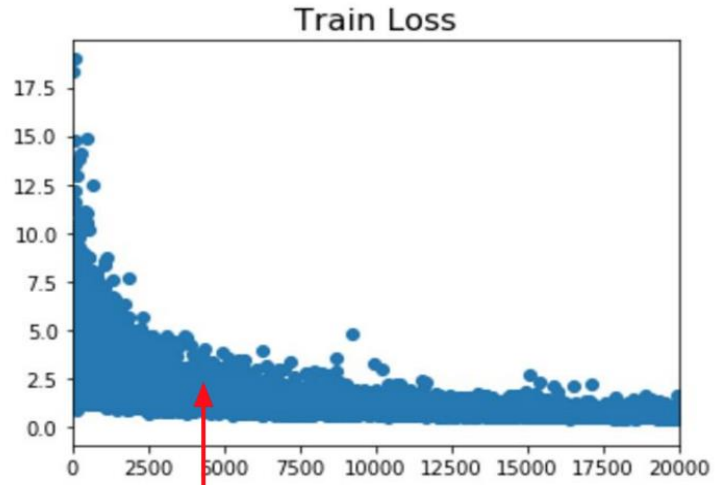
Slide from http://cs231n.stanford.edu/

Please try to identify the following in our example code

- Structure of input/output
- Train/Validation/Test split
- Cost function
- Optimization method, steps (epochs)
- Batch size
- Dropout
- Data augmentation?

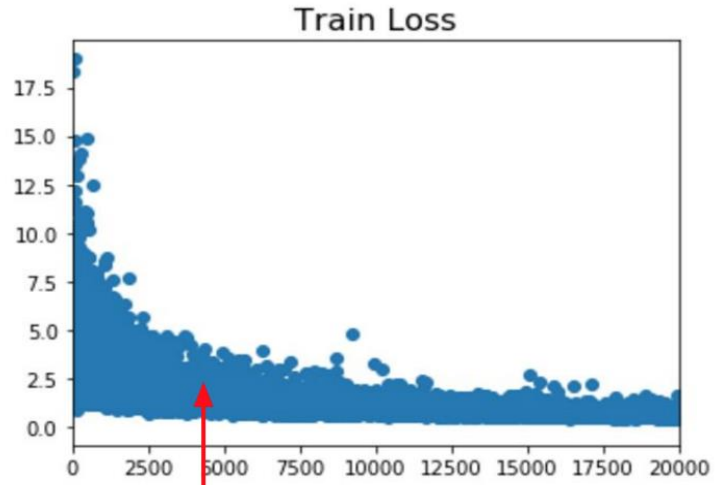
https://deepimaging.github.io/data/high_level_pytorch_intro.ipynb

What you'll typically see…



Better optimization algorithms
help reduce training loss

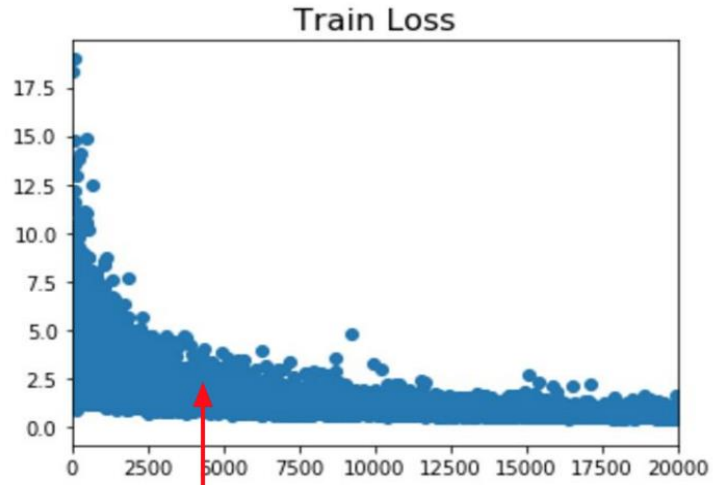
What you'll typically see…

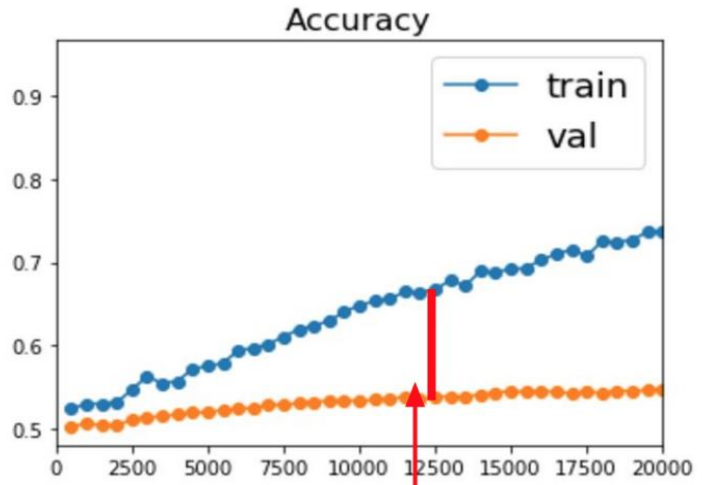


What you can do to help out training error:

- Optimizer choice
- Optimizer step size

What you'll typically see…





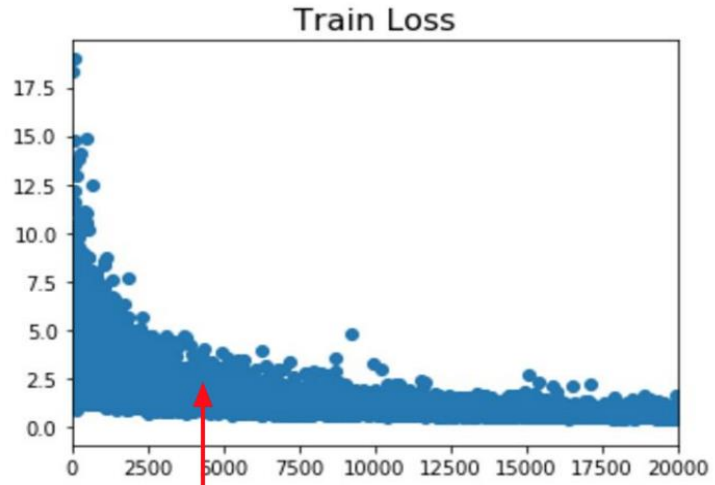
Better optimization algorithms help reduce training loss

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

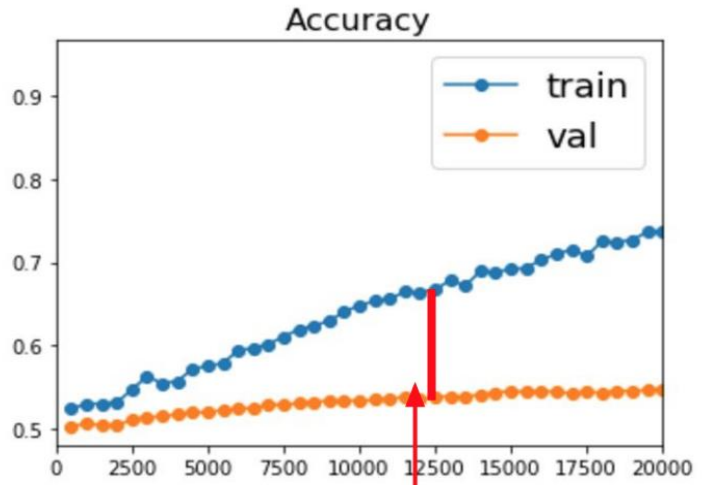
What you can do to help out training error:

- Optimizer choice
- Optimizer step size

What you'll typically see…



Better optimization algorithms help reduce training loss

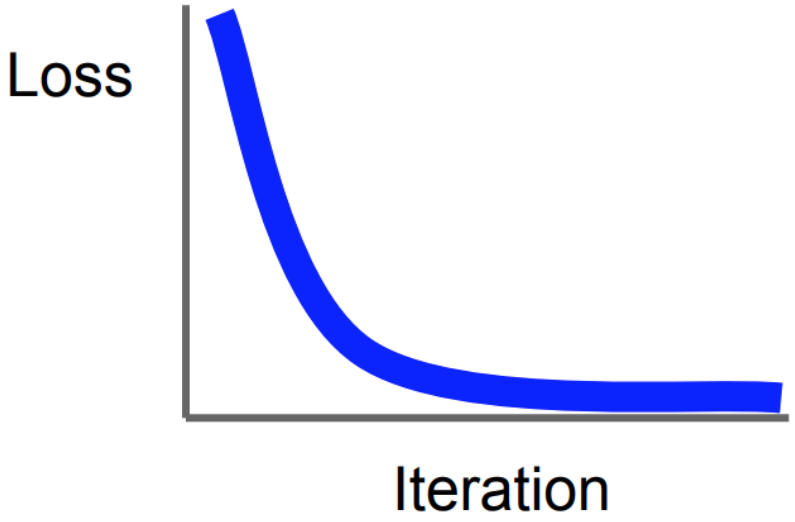


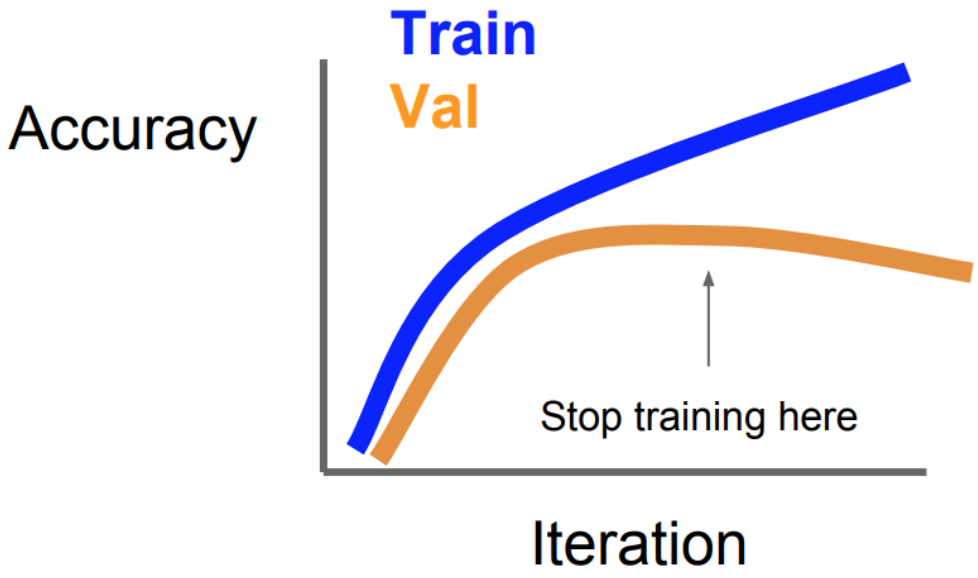
But we really care about error on new data - how to reduce the gap?

What you can do to help out training error:
- Optimizer choice
- Optimizer step size

What you can do to help out training error:
- More regularization!
  - Dropout
- Data normalization
- Data augmentation
- A few other tricks..

Trick #1: Early stopping

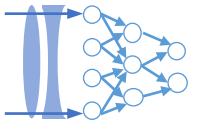




Stop training the model when accuracy on the validation set decreases
Or train for a long time, but always keep track of the model snapshot that worked best on val

Slide from http://cs231n.stanford.edu/

Trick #2: Use Model Ensembles



1. Train multiple independent models
2. At test time average their results
   (Take average of predicted probability distributions, then choose argmax)

Enjoy 2% extra performance

Related concept/term: majority voting

E.g., look at *same* dog image from test data 9X, each w/ uniquely trained model

- Get (let's say) [6, 3] for output classification
- so guess [1,0] = it's a dog
- Will do better than running model once!

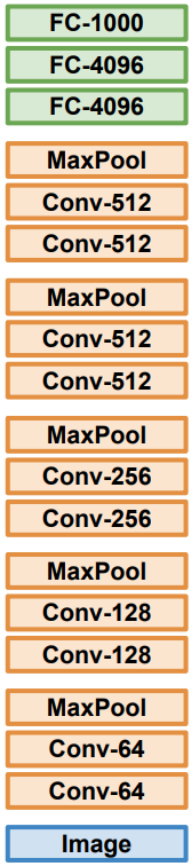
Related technique: Test Time Augmentation
Also relevant: Dropout-type methods

Trick #3: Transfer learning

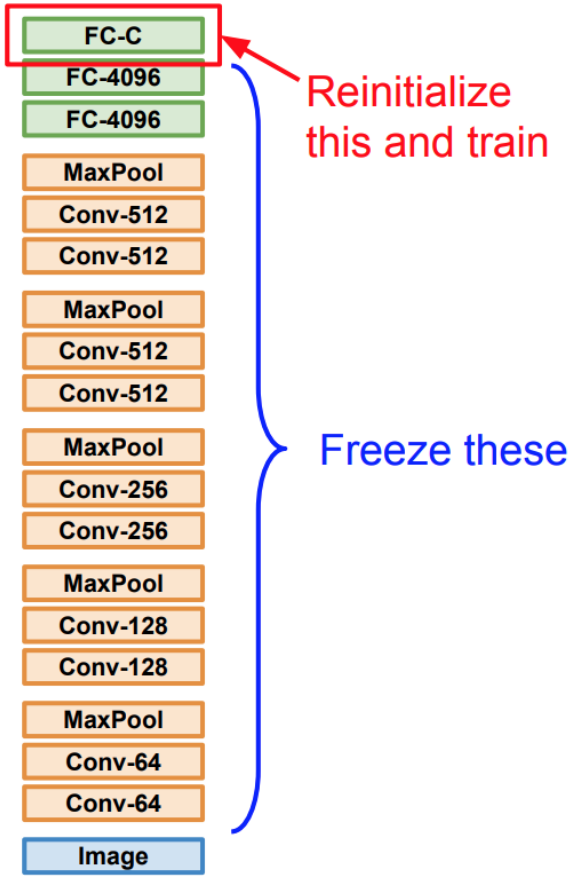
Transfer Learning with CNNs

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

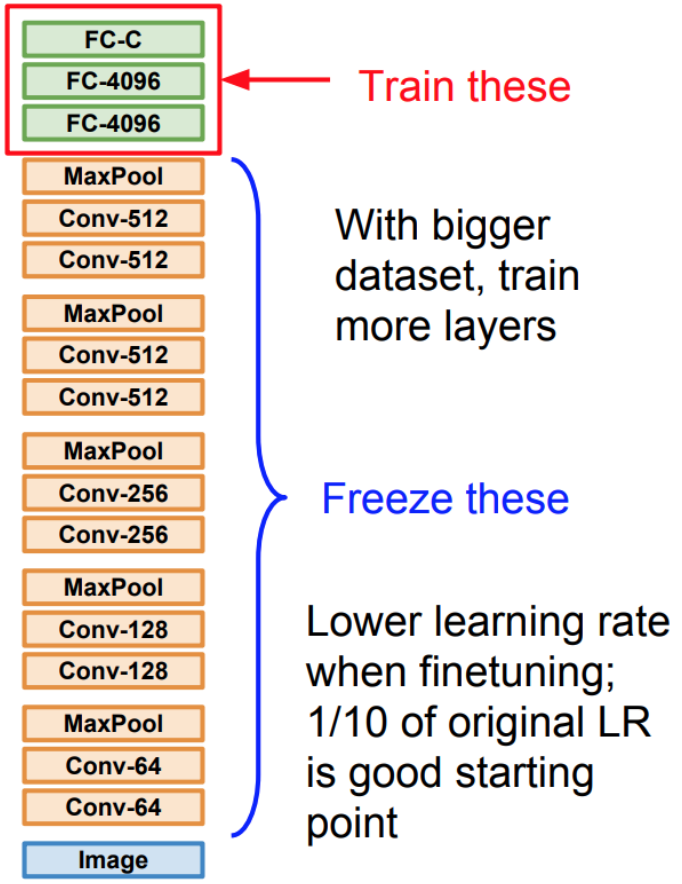
1. Train on Imagenet



2. Small Dataset (C classes)

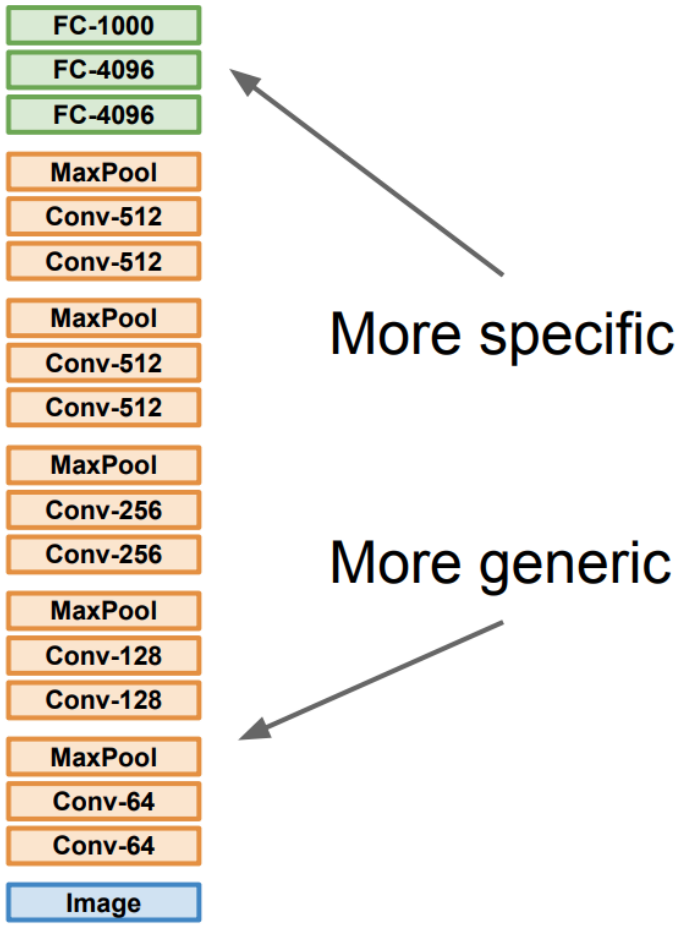


3. Bigger dataset



Slide from http://cs231n.stanford.edu/

Trick #3: Transfer learning

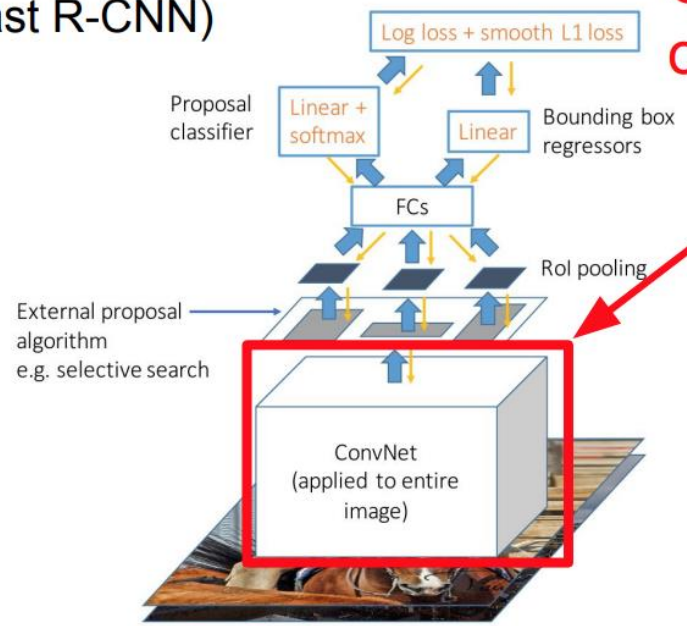


	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Slide from http://cs231n.stanford.edu/

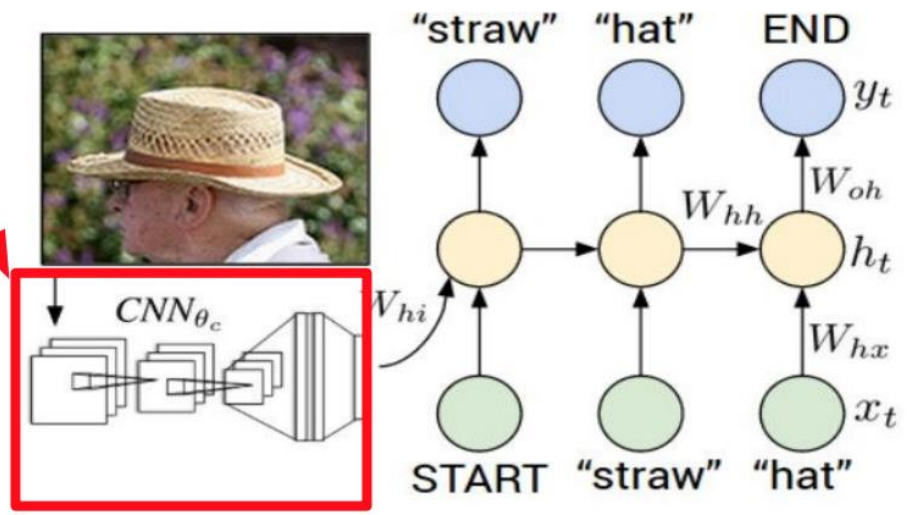
Transfer learning with CNNs is pervasive…
(it's the norm, not an exception)

Object Detection
(Fast R-CNN)



CNN pretrained on ImageNet

Image Captioning: CNN + RNN



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.

Slide from http://cs231n.stanford.edu/