

Lecture 11: Tools for your deep learning toolbox

Machine Learning and Imaging

BME 590L
Roarke Horstmeyer

Thank you to Kevin Zhou for preparing much of this material

Final Presentation Scheduling – Survey

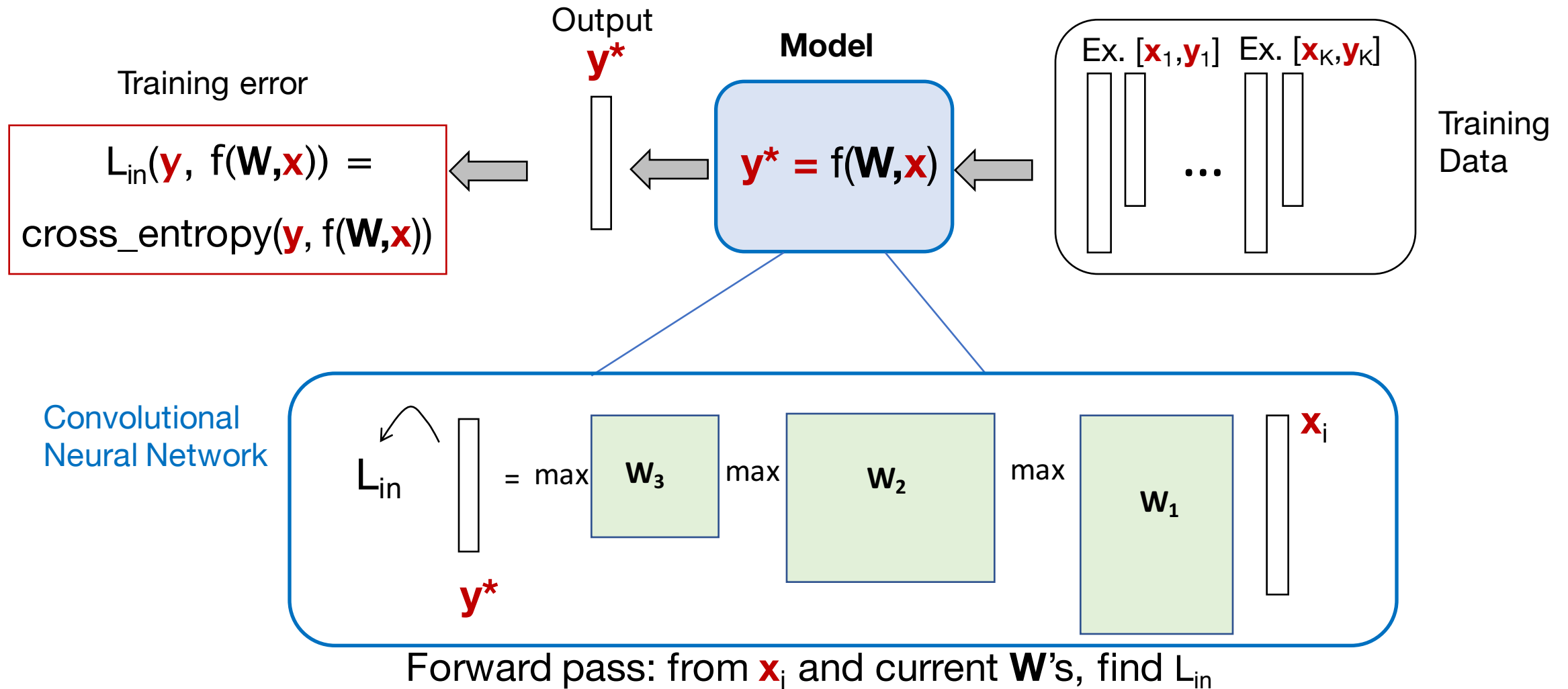
Option #1: Final presentations over 3 (or slightly more) hours on Wednesday 4/29 at 9AM (Finals slot). Everyone has to attend and listen to all projects

Option #2: Split final presentations into 3 slots, each ~1 1/4 hours in duration. Here are the approximate time slots:

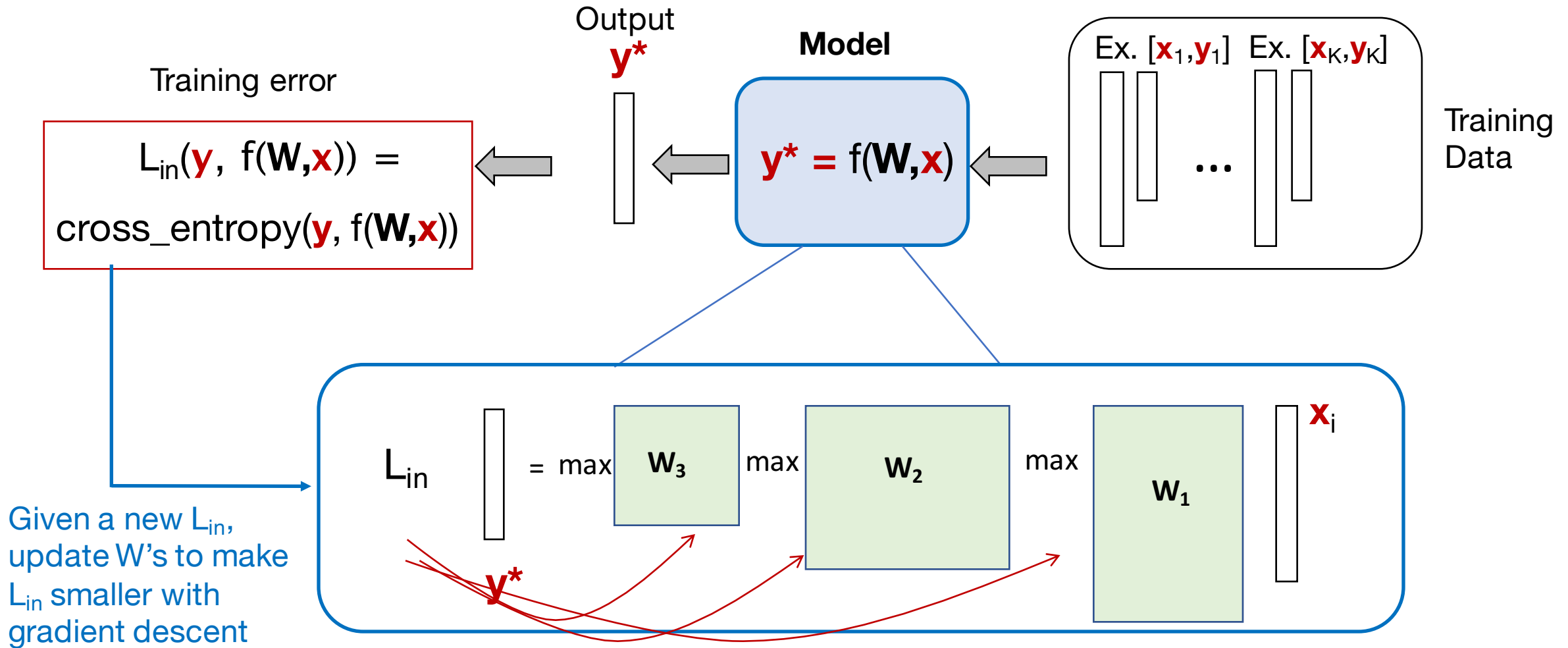
- Thursday 4/23 from 11:45am – 1pm
- Friday 4/24 from 4pm – 5:15 pm
- Monday 4/27 from 2:00pm – 3:15 pm

You would only have to come to one slot (the one you'll present at), but are certainly invited to come to all of them as well.

Our very basic convolutional neural network



Our very basic convolutional neural network



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

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.

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

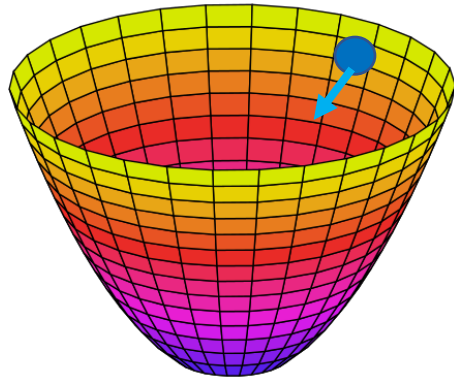
Other specifics: Variable Initialization, augmentation, batch normalization, dropout, gradient descent params.

This class: final details about deep CNN implementation

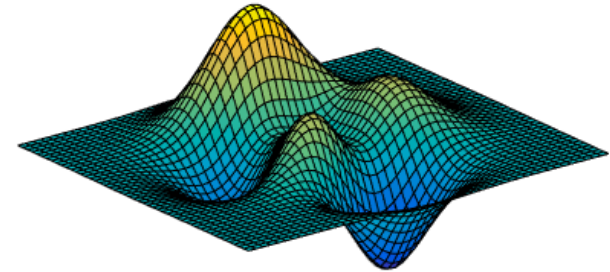
Weights initialization

- Need to start somewhere –

**Convex
problem:
doesn't really
matter where
you start**



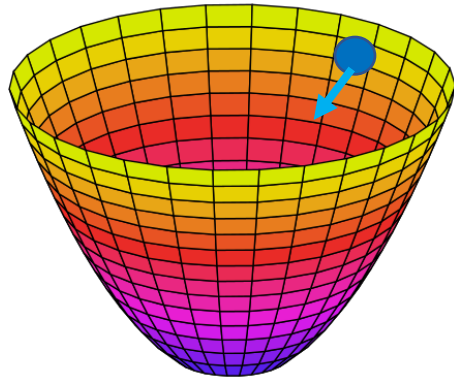
**Non-convex problem:
certainly matters, but
you don't know where
is best...**



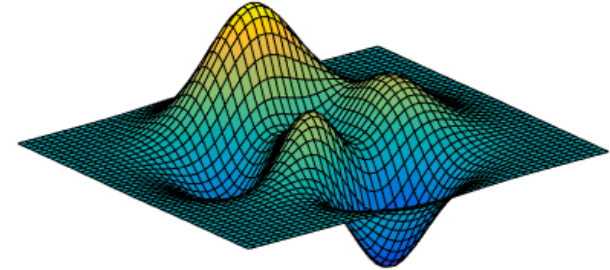
Weights initialization

- Need to start somewhere –

Convex problem:
doesn't really matter where you start



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



- Typically best to use an appropriate random guess sampled from a Gaussian distribution:

```
layer1_weight = tf.Variable(tf.truncated_normal([5,5, 1, 32], stddev = 0.1))
```


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

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

$$\text{var}(\mathbf{y}) = \text{var}(\mathbf{w}^T \mathbf{x}) = \text{var}(w_1 x_1 + \dots w_N x_N) = N \text{var}(w_1 x_1) \quad (\text{IID})$$

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

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

$$\text{var}(\mathbf{y}) = \text{var}(\mathbf{w}^T \mathbf{x}) = \text{var}(w_1 x_1 + \dots w_N x_N) = N \text{var}(w_1 x_1) \quad (\text{IID})$$

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

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

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

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

$$\text{var}(\mathbf{y}) = \text{var}(\mathbf{w}^T \mathbf{x}) = \text{var}(w_1 x_1 + \dots w_N x_N) = N \text{var}(w_1 x_1) \quad (\text{IID})$$

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

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

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

`layer1_weight = tf.Variable(tf.truncated_normal([5,5, 1, 32], stddev = 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!

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

- 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

Cited ~16,000 times! (as
of Feb 2020)

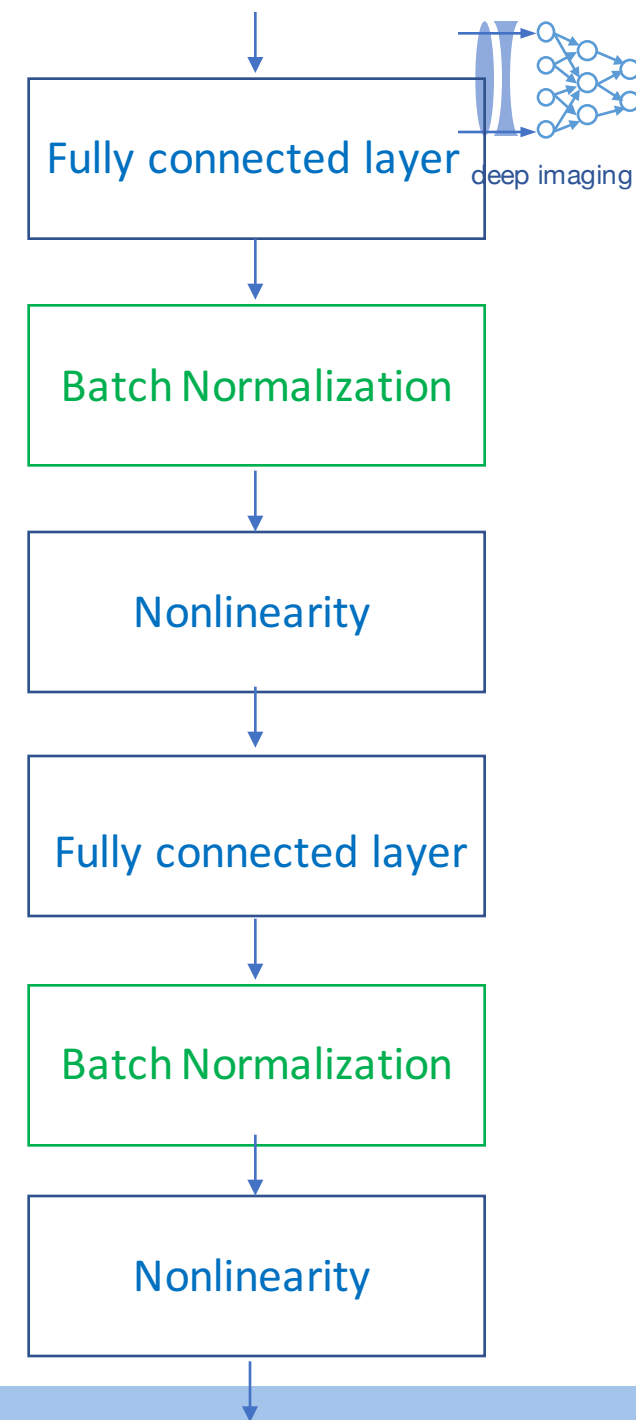
Batch normalization (BN)

- 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 :

$$x'(i) = (x(i) - E[x(i)]) / \text{STD}[x(i)]$$

- 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)

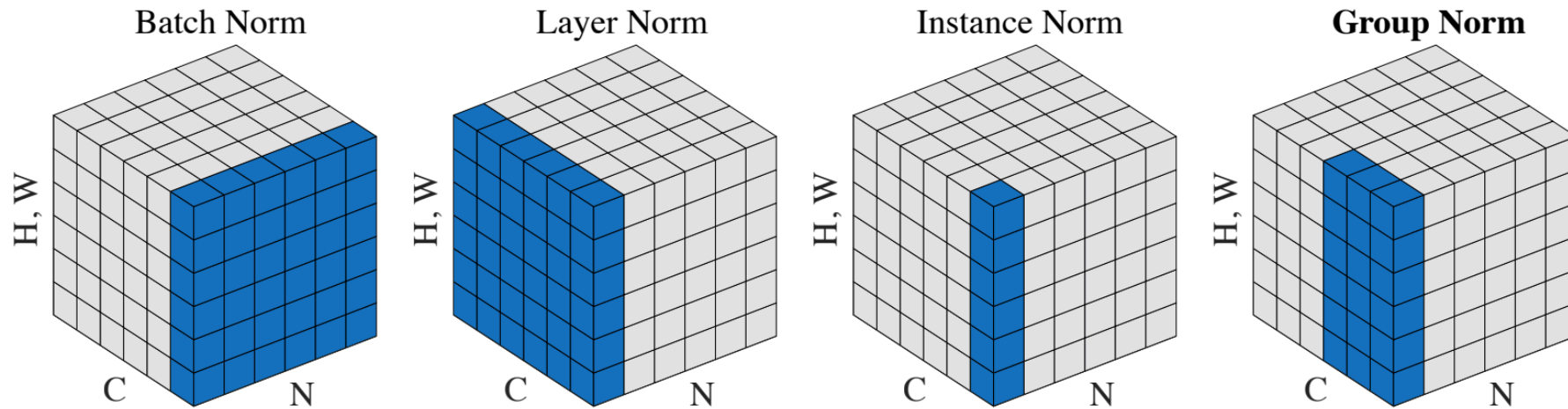
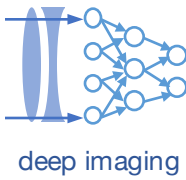


Figure 2. **Normalization methods.** Each subplot shows a feature map tensor, with N as the batch axis, C as the channel axis, and (H, W) as the spatial axes. The pixels in blue are normalized by the same mean and variance, computed by aggregating the values of these pixels.



Dropout: A Simple Way to Prevent Neural Networks from Overfitting

Nitish Srivastava

Geoffrey Hinton

Alex Krizhevsky

Ilya Sutskever

Ruslan Salakhutdinov

Department of Computer Science

University of Toronto

10 Kings College Road, Rm 3302

Toronto, Ontario, M5S 3G4, Canada.

NITISH@CS.TORONTO.EDU

HINTON@CS.TORONTO.EDU

KRIZ@CS.TORONTO.EDU

ILYA@CS.TORONTO.EDU

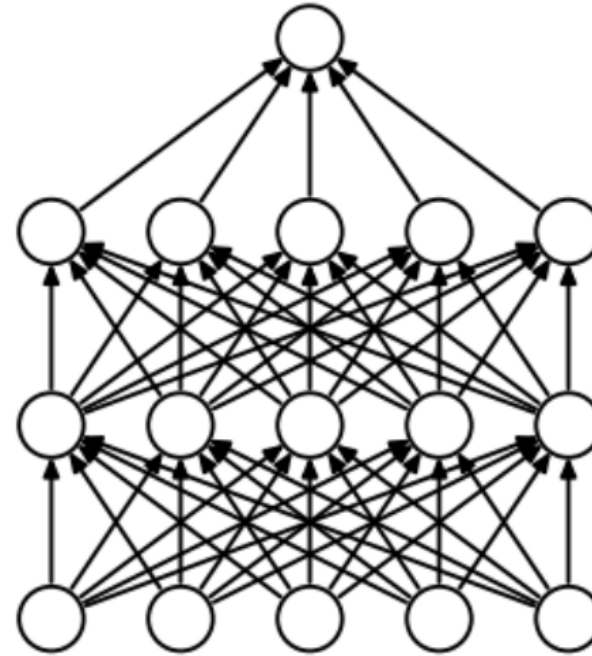
RSALAKHU@CS.TORONTO.EDU

Editor: Yoshua Bengio

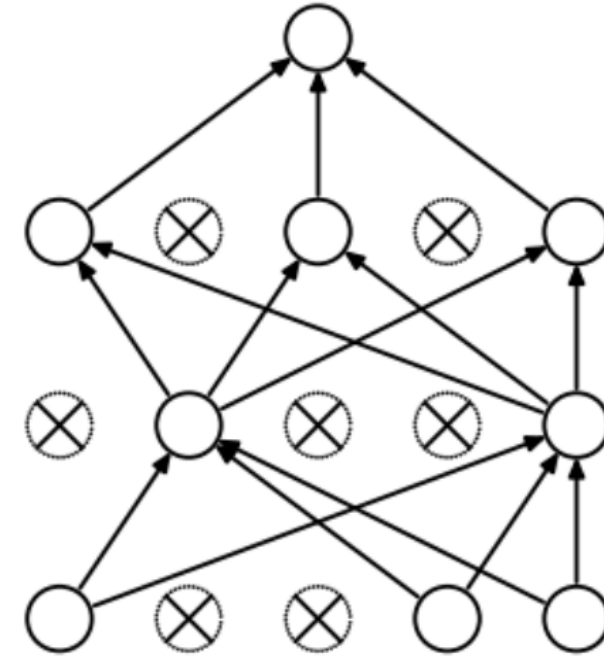
**Cited almost 18,000
times! (as of Feb. 2020)**

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



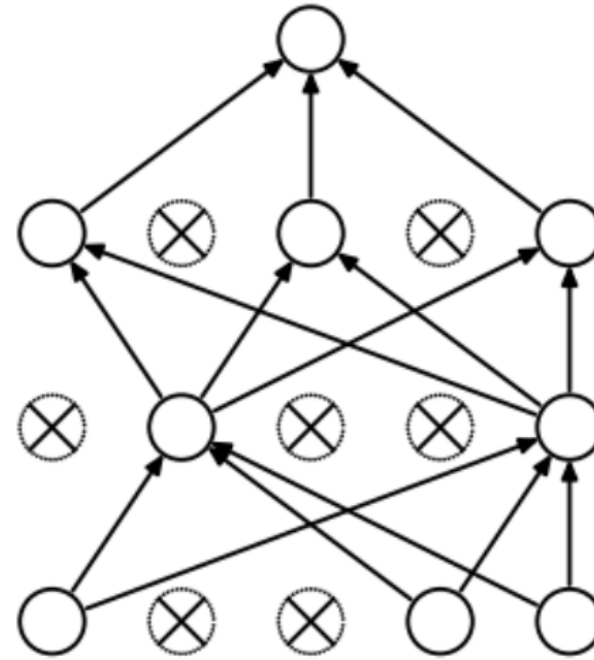
(b) After applying dropout.

Dropout

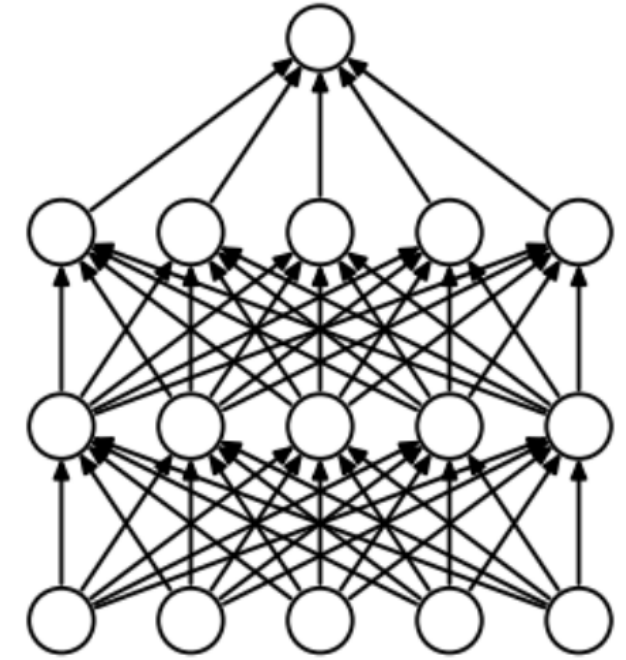
- Only one hyperparameter “rate” = p , the expected fraction of neurons to drop in a given layer
- In TensorFlow:
 - `next_layer = tf.layers.dropout(previous_layer, rate=0.5)`
- Common practices:
 - Set $p=0.5$
 - Make the layer wider (more units/neurons)
 - Apply to fully connected layers, not convolutional 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

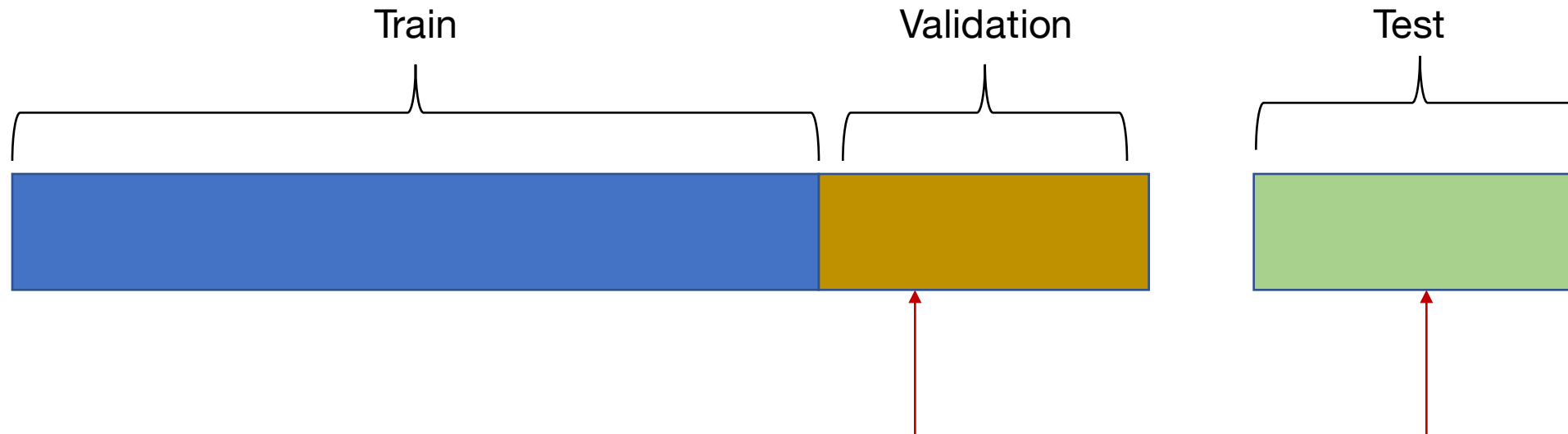


Training
(each node dropped with probability)



Testing
(all weights multiplied by $1-p$)

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!

Let's examine our first CNN implemented in Tensorflow!

http://deepimaging.github.io/data/basic_tensorflow_eager_example.ipynb

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