

Deep Learning in Apache MxNet Gluon

Imperative, Dynamic and Distributed

gluon.mxnet.io

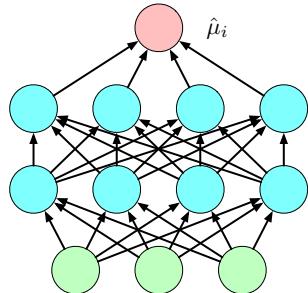
Mu Li, Zachary Lipton

AWS Machine Learning



What are we doing here?

For novices



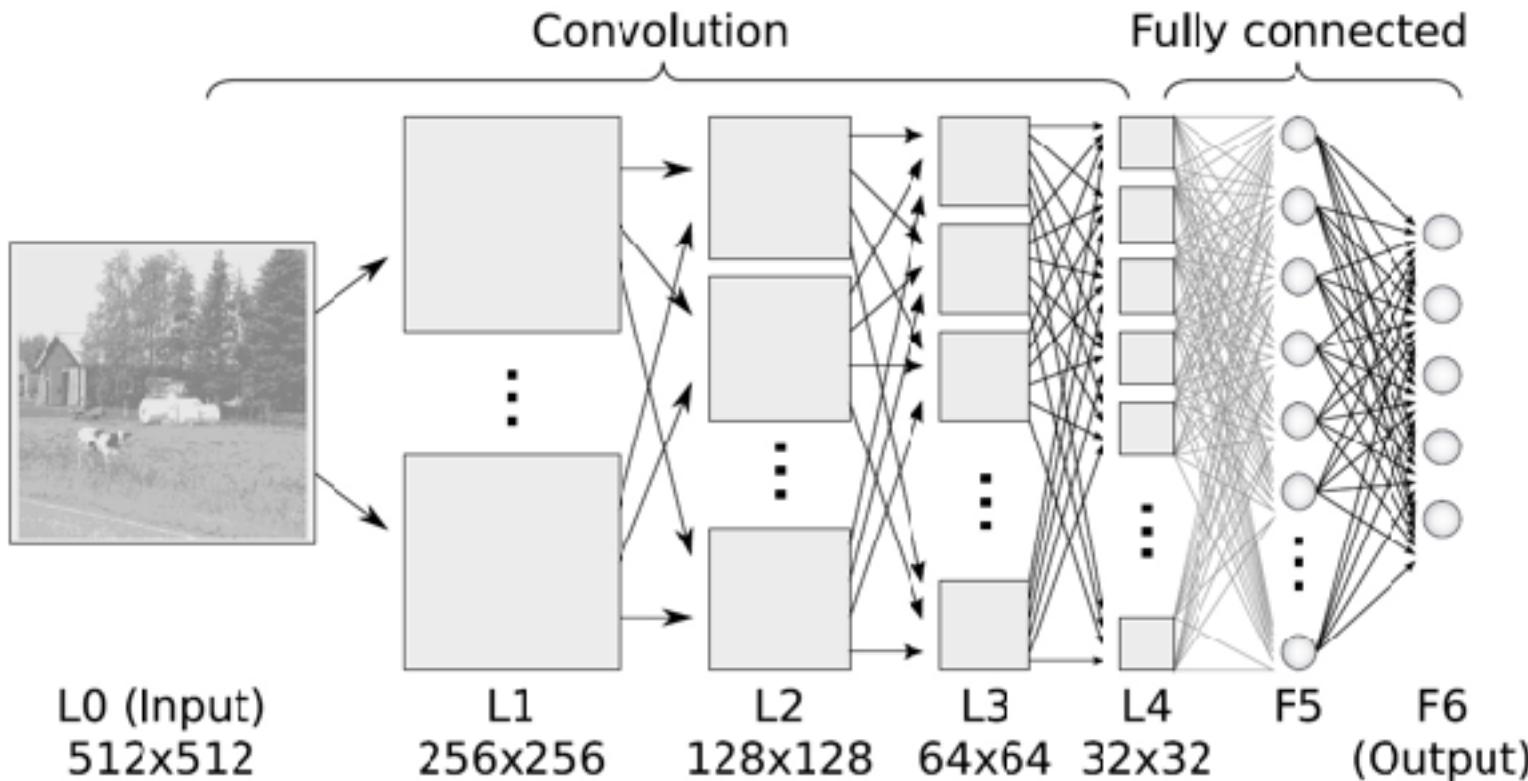
For deep learning
experts



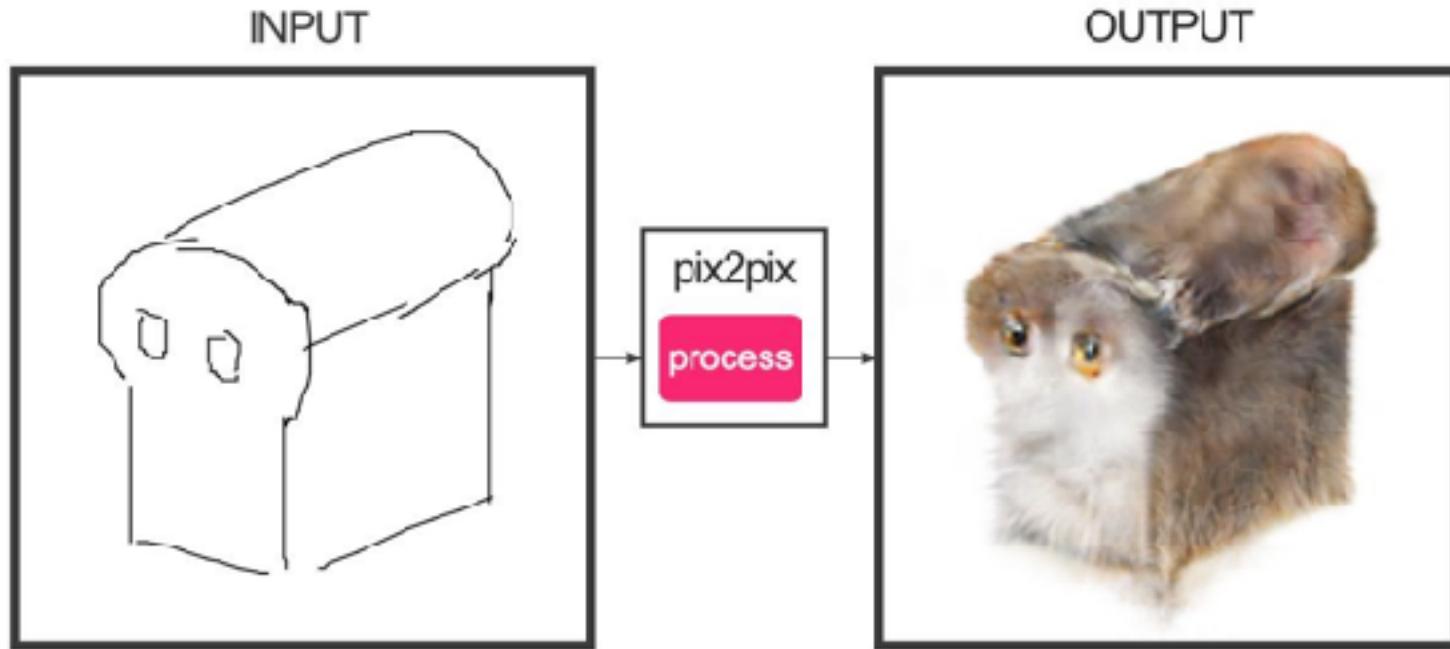
For experts in both



Deep learning can classify images



... and translate sketches into cats



<https://arxiv.org/pdf/1611.07004v1.pdf>

... and generate beer reviews

Poured from 12oz bottle into half-liter Pilsner Urquell branded pilsner glass. **Appearance:** Pours a cloudy golden-orange color with a small, quickly dissipating white head that leaves a bit of lace behind. **Smell:** Smells HEAVILY of citrus. By heavily, I mean that this smells like kitchen cleaner with added wheat. **Taste:** Tastes heavily of citrus- lemon, lime, and orange with a hint of wheat at the end. **Mouthfeel:** Thin, with a bit too much carbonation. Refreshing. **Drinkability:** If I wanted lemonade, then I would have bought that.

(a) Real Review

Poured from a 12oz bottle into a 16oz Samuel Adams Perfect Pint glass. **Appearance:** Very pale golden color with a thin, white head that leaves little lacing. **Smell:** Very mild and incoffensive aromas of citrus. **Taste:** Starts with the same tastes of the citrus and fruit flavors of orange and lemon and the orange taste is all there. There is a little bit of wheat that is pretty weak, but it is sort of harsh (in a good way) and ends with a slightly bitter aftertaste. **Mouthfeel:** Light body with a little alcohol burn. Finish is slightly dry with some lingering spice. **Drinkability:** A decent beer, but not great. I don't think I would rate this anytime soon as it says that there are other Belgian beers out there, but this is a good choice for a warm day when it's always available in the North Coast Brewing Company party.

(b) Synthetic review

<http://deepx.ucsd.edu/#/home/beermind>



... and [???]



 **DeepDrumpf**
@DeepDrumpf

[Follow](#) ▾

[Despite the negative press [#covfefe](#)] look at what's going on. They shoot media. Usually that's a bad sign of things to come.

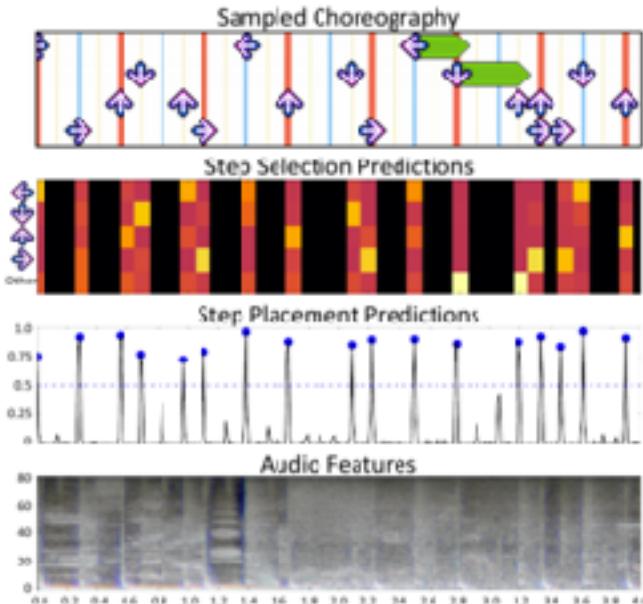
12:10 AM - 31 May 2017

Still Want That Peny

31 Retweets 96 Likes

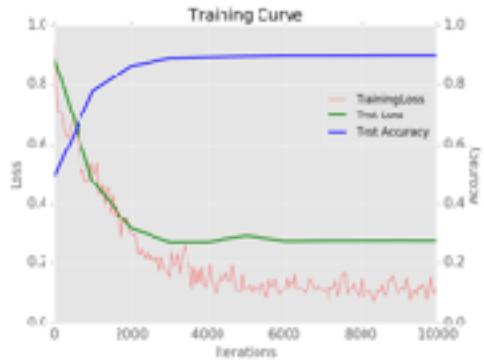
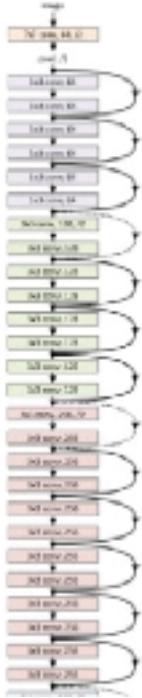
3 31 96

... and generate DDR choreography



<https://arxiv.org/abs/1703.06891>

DL is cool, but expensive (computationally)



The languages we like are slow...



Call down to fast libraries



python



PYTORCH



theano

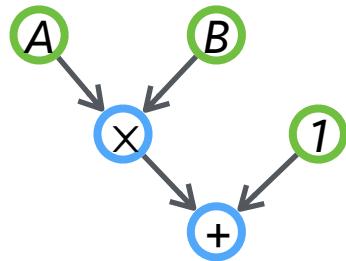
model.forward(data)



[crazy GPU stuff]



Declarative (Symbolic) Programs



```
A = Variable('A')
B = Variable('B')
C = B * A
D = C + 1
f = compile(D)
d = f(A=np.ones(10),
      B=np.ones(10)*2)
```

C can share memory with D

- **Advantages:**

- Opportunities for optimization
- Easy to serialize models
- More portable across languages

- **Disadvantages:**

- Hard to debug
- Unsuitable for dynamic graphs
- Can't use native code

Imperative Programs



```
import numpy as np  
a = np.ones(10)  
b = np.ones(10) * 2  
c = b * a  
print c  
d = c + 1
```

Easy to debug,
easy to code

- **Advantages**

- Straightforward and flexible
- Use language natively (loops, control flow, debugging)

- **Disadvantages**

- Hard to optimize
 - **Fixed with JIT compiler!
(covered in this tutorial)**

Motivation

Q: Why GLUON?

A:



+



imperative

symbolic

theano

Caffe

Microsoft
CNTK

before

2012

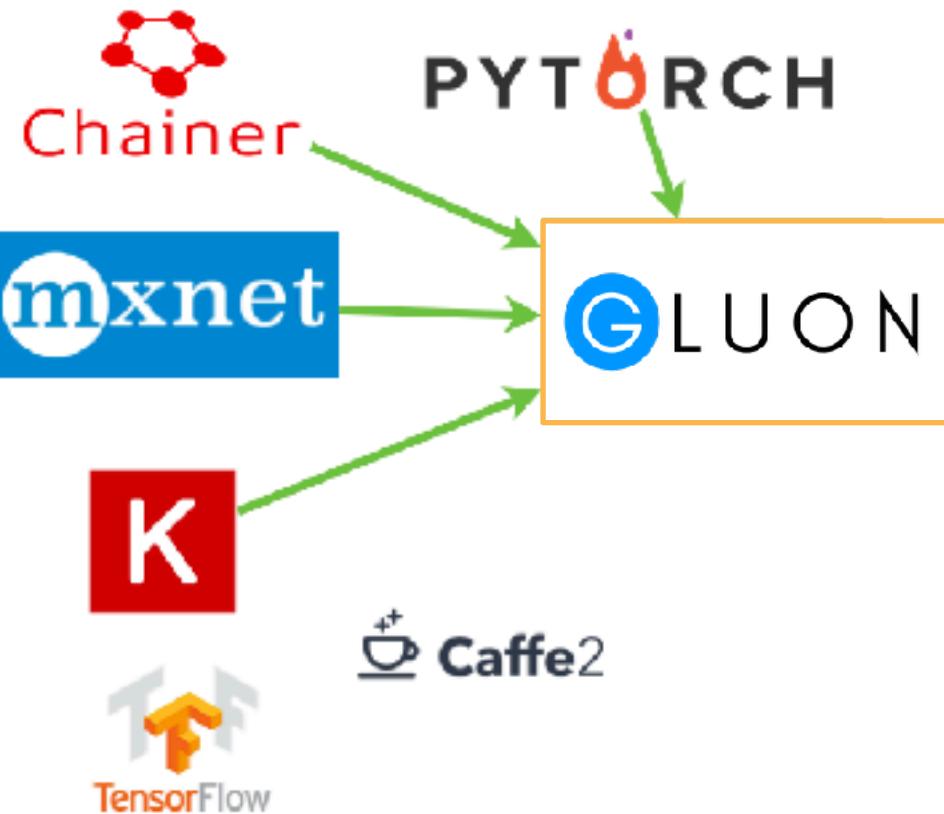
2013

2014

2015

2016

2017

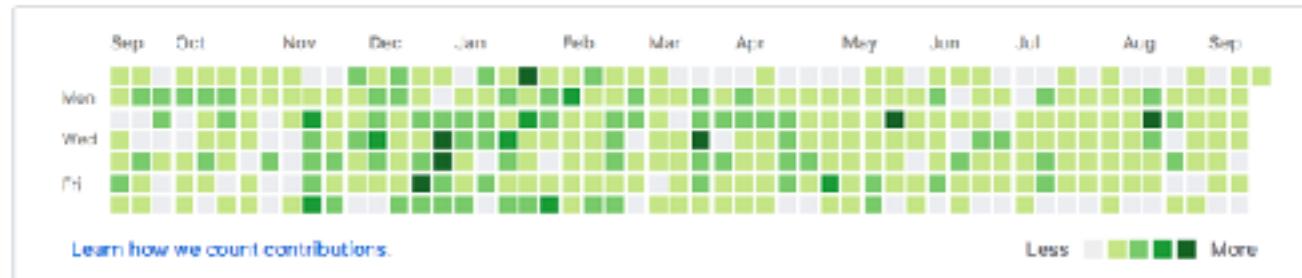


Shoutout

- Developed by Amazon scientist and GitHub superhero Eric (Junyuan) Xie @piiswrong



1,594 contributions in the last year



Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

[SMTG like this...]

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

Installation

- To use gluon, you'll need the latest version of MxNet!

- **PIP install** (Better to use Python 3)

```
pip install --upgrade pip  
pip install --upgrade setuptools  
pip install mxnet -pre --user
```

- pip install mxnet-cu80 # for cuda 8.0

- pip install mxnet-cu75 # for cuda 7.5

- python

- from mxnet import nd

- a = nd.array([1,2])

- print(a)

without GPUs



Documentation & Slides

- You'll also need the latest version of MxNet!

- Notebooks

```
git clone
```

```
git@github.com:zackchase/mxnet-the-straight-dope
```

- Gluon documentation

gluon.mxnet.io

mxnet.io/api/python/gluon.html

- Deep Learning AMI

bit.ly/deepami and bit.ly/deepubuntu



Amazon Machine Image for Deep Learning

- AWS instance **fully set up** for deep learning research
- Updated **MXNet**, TensorFlow, Caffe, Torch, Theano, Keras
- Anaconda, Jupyter, Python 2 and 3
- **NVIDIA** Drivers for GPU instances (G2, P2)
- **Intel MKL** Drivers for all other instances (C4, M4, etc.)
- Available at: bit.ly/deepami and bit.ly/deepubuntu

Getting started

```
Last login: Mon Aug 14 14:38:19 2017 from 173.252.63.84
```

```
=====
 _ | _ | _ )  
_ | ( _ / Deep Learning AMI for Amazon Linux  
_ \_ | _ |
```

```
The README file for the AMI ➔➔➔➔➔➔➔➔➔➔➔➔➔➔➔➔➔➔➔➔ ➔ /home/ec2-user/src/README.md  
Tests for deep learning frameworks ➔➔➔➔➔➔➔➔➔ ➔ /home/ec2-user/src/bin  
=====
```

```
No packages needed for security; 2 packages available
```

```
Run "sudo yum update" to apply all updates.
```

```
Amazon Linux version 2017.03 is available.
```

```
[ec2-user@ip-10-0-0-77 ~]$ ls src
```

anaconda2	caffe	caffe3	caffe_cpu	keras	Nvidia_Cloud_EULA.pdf	tensorflow
tensorflowanaconda3			anaconda3	caffe2	caffe_anaconda2	cntk
logs		OpenBLAS	tensorflow3		theano	bin
caffe2_anaconda2		caffe_anaconda3	demos	mxnet	README.md	tensorflowanaconda torch

Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

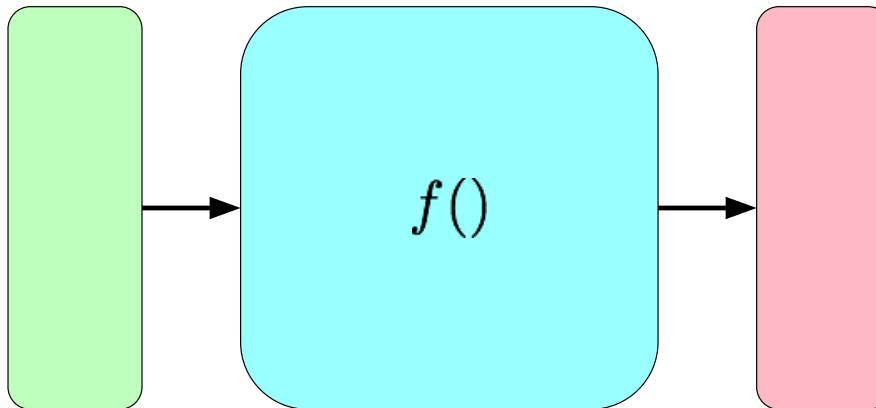
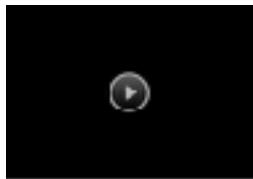
4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

Supervised Learning

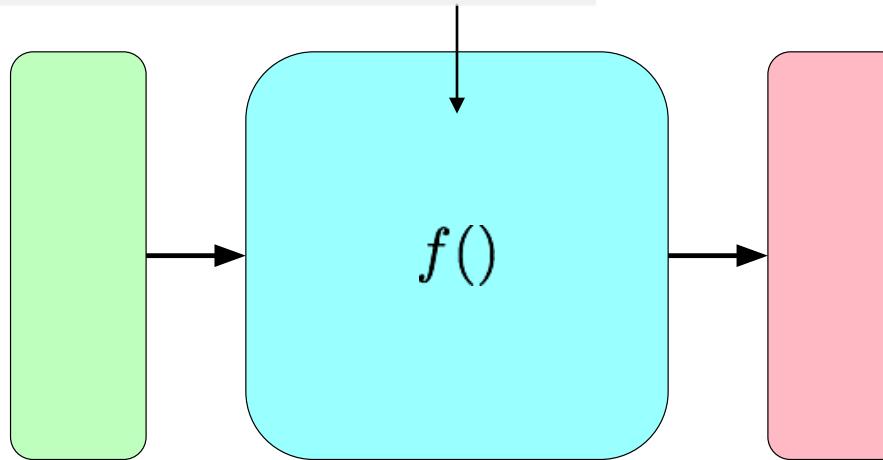
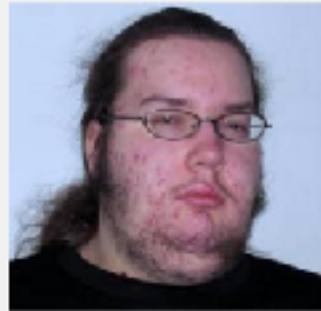


“Time underlies many interesting human behaviors”



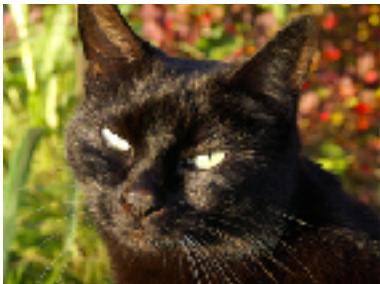
{0,1}
{A,B,C...}
captions,
email mom,
fire nukes,
eject pop tart

Programming with Programmers



Limits of Conventional Programming

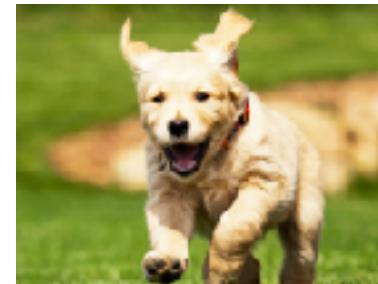
- Given some input x we want to output correct value y
 - When you know a simple rule, OK
 - For many, problems we don't know the rule**
 - Still, we might have examples of correct (x, y) pairs**



cat



cat



dog

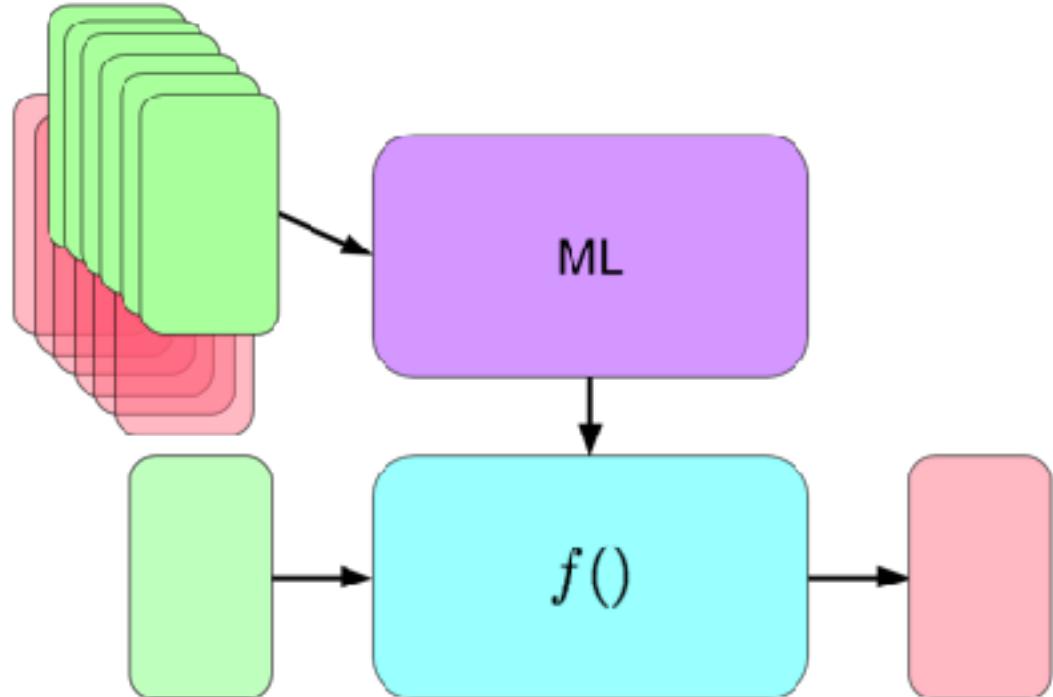


dog

image credit - wikipedia

Programming with Data

- **Core components**
 - Input data
 - Targets
 - Model
 - Loss function
 - Learning rule



FizzBuzz

Programming with Data

- **Generate training data**
`(9,'Fizz'),(10,'Buzz'),(2,'2'),
(15,'Fizz Buzz') ...`
- **Extract input features**
 $x \rightarrow (x \% 3, x \% 5, x \% 15)$
- **Train a classifier**
mapping input x to output y
using training data

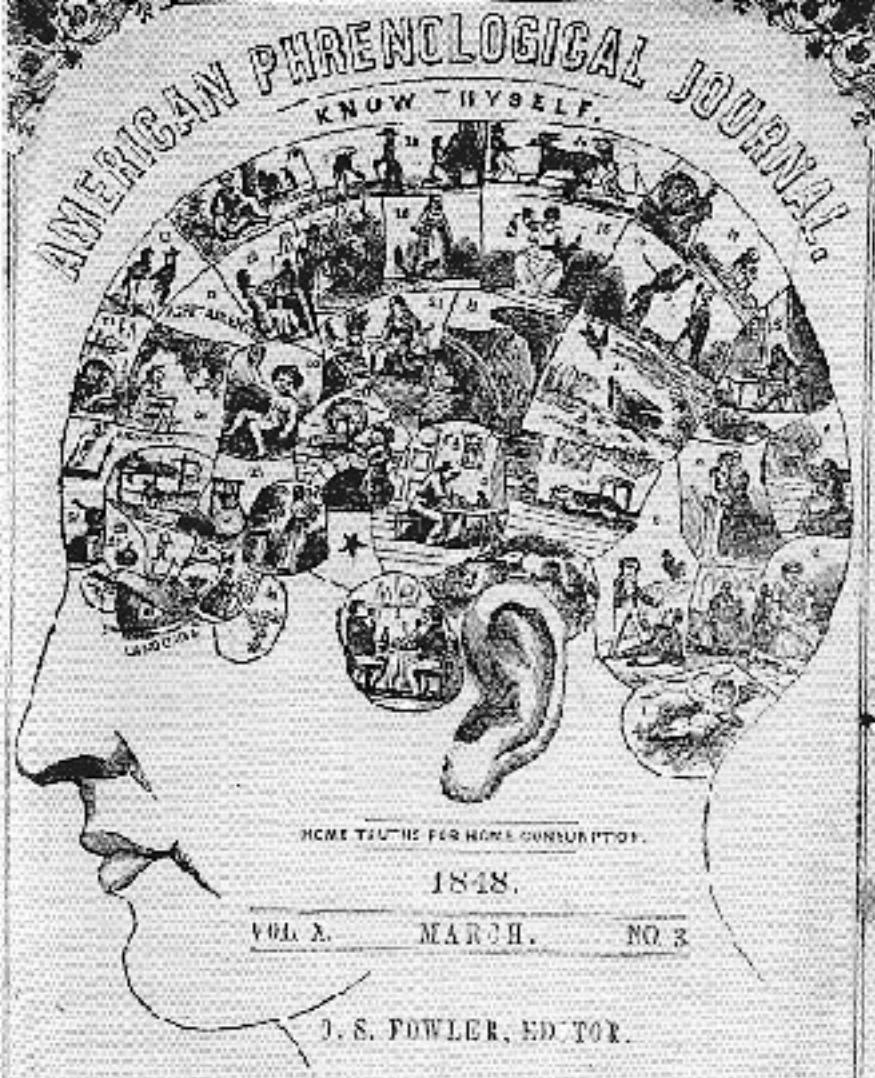
Code

```
var o='';  
  
for (var i=1; i<=100; i++) {  
    i%3 || (o+='Fizz ');  
    i%5 || (o+='Buzz ');  
    !(i%3 && i%5) || (o+=(i+' '));  
}  
  
console.log(o);
```

That was silly.
Why would you do this?

Inspired by joelgrus.com

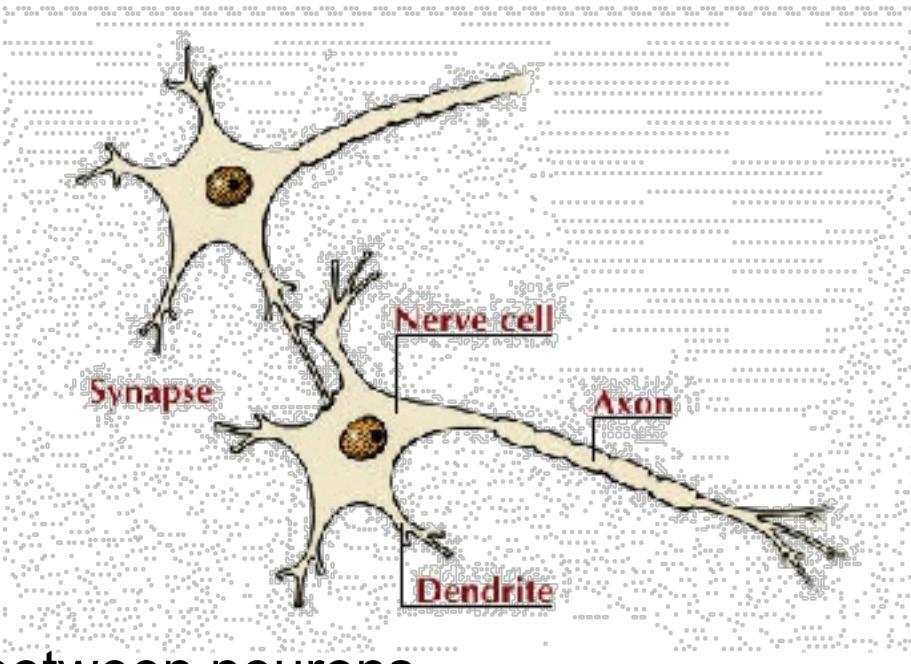




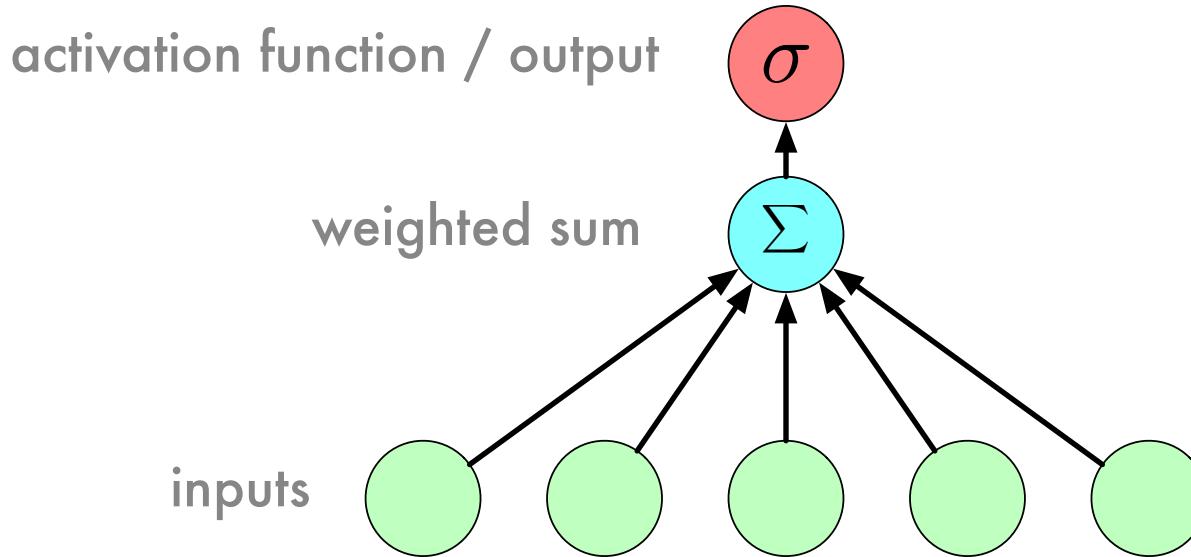
Neurons and Learning

Neurons

- **Soma** (CPU)
Cell body - combines signals
- **Dendrite** (input bus)
Combines the inputs from several other nerve cells
- **Synapse** (interface)
Interface and **parameter store** between neurons
- **Axon** (cable)
May be up to 1m long and will transport the activation signal to neurons at different locations



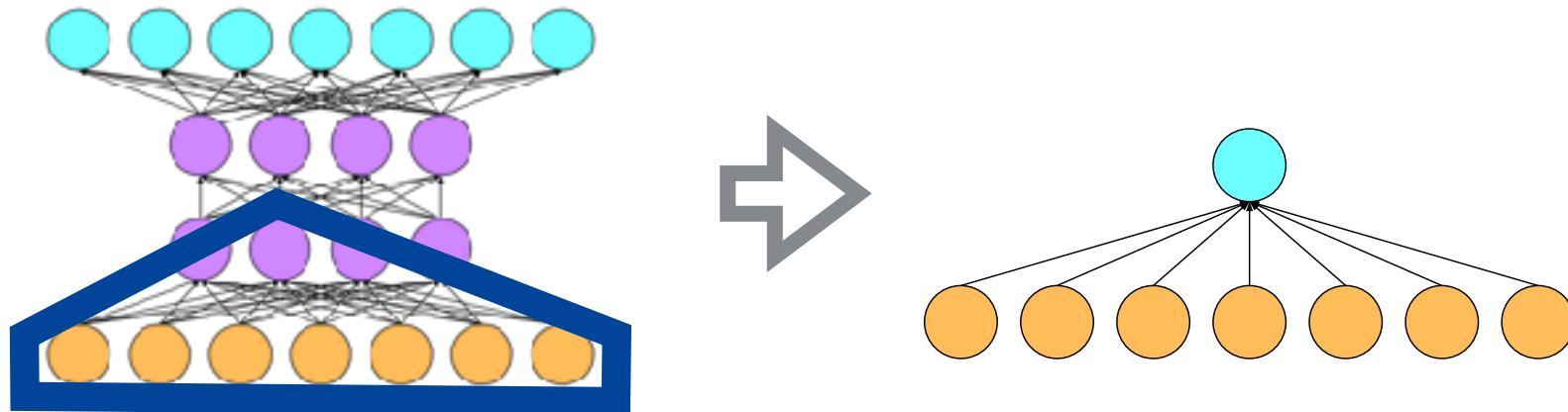
Neurons



$$f(x) = \sum_i w_i x_i = \langle w, x \rangle$$

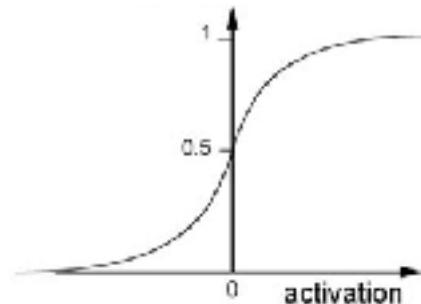
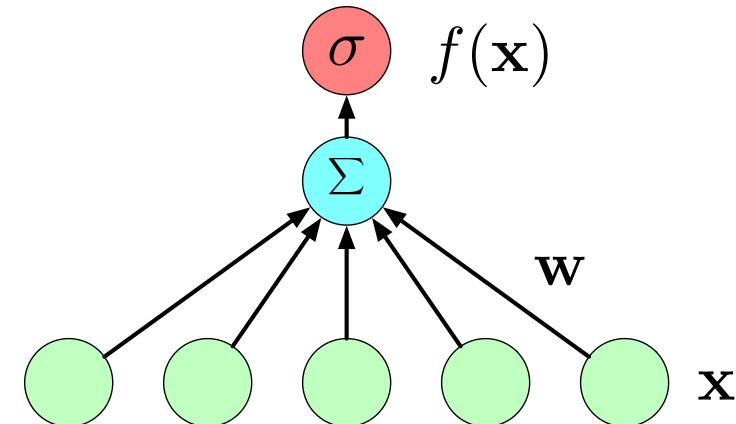
Neural networks

- Neural nets are **models** composed of artificial neurons
- The simplest neural nets consist of one neuron



Components of a neural net

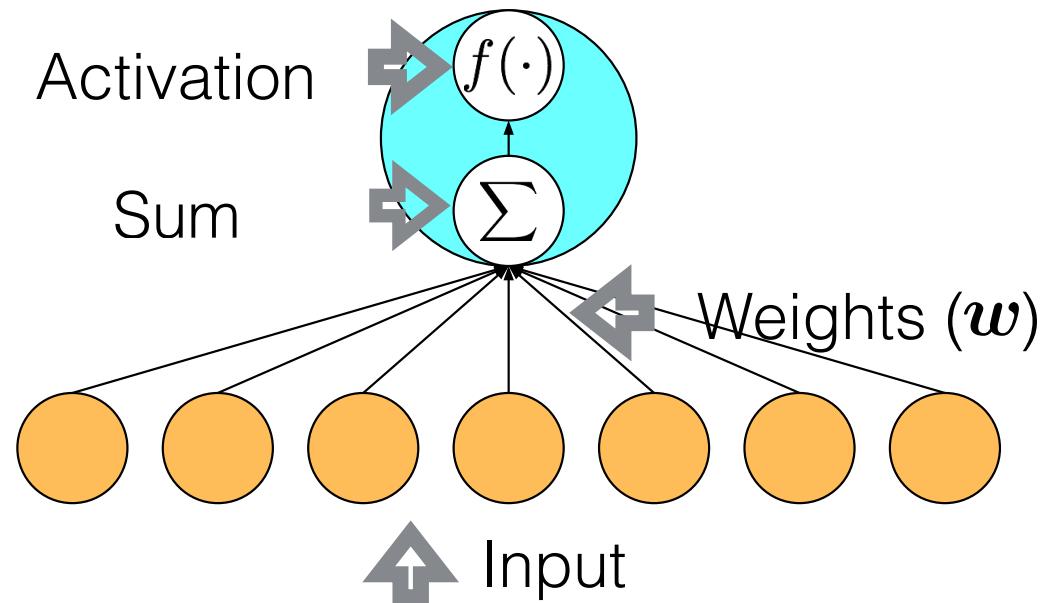
- **Inputs**
Data vector \mathbf{x}
- **Targets**
What we want model to output
- **Architecture**
Connectivity pattern and activation functions
- **Learning rule**
Updates the parameters



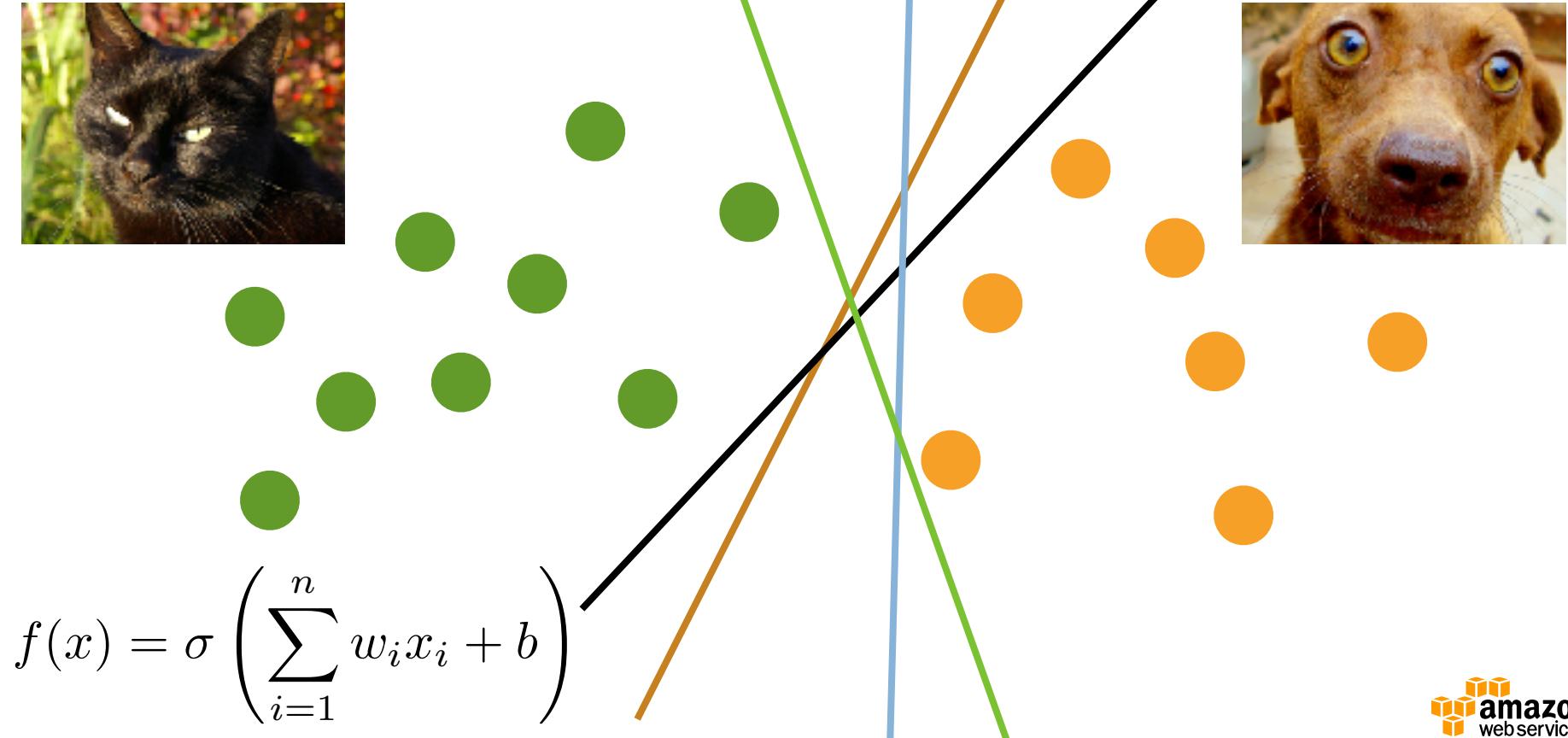
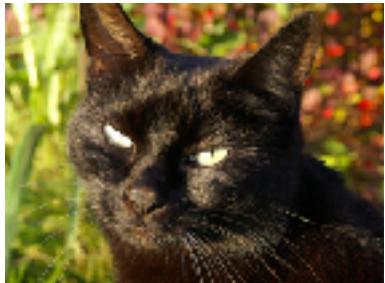
$$f(x) = \sigma \left(\sum_{i=1}^n w_i x_i + b \right)$$

Some single-neuron models

- **Linear Regression**
Linear activation
Square loss
- **Perceptron**
Linear
Hinge loss
- **Logistic Regression**
Sigmoid Activation
Cross-entropy loss



Perceptron, logistic regression, linear SVM



MxNet Code

```
import mxnet as mx  
from mxnet import gluon  
  
net = gluon.nn.Sequential()  
net.add(gluon.nn.Dense(1))  
loss = gluon.loss.SoftmaxCrossEntropyLoss()  
  
+ iterator over data (and loader)  
+ training loop
```

create a new model

add fully-connected layer

instantiate a loss

A large, fluffy white and black panda is the central focus, standing on its hind legs and looking down at a small, yellowish-orange figure crouching on the grass. The setting is a vibrant green field dotted with small flowers, with a traditional Chinese town featuring red-roofed buildings in the background under a clear blue sky.

The Perceptron

The Perceptron

initialize $w = 0$ and $b = 0$

repeat

if $y_i [\langle w, x_i \rangle + b] \leq 0$ **then**

$w \leftarrow w + y_i x_i$ and $b \leftarrow b + y_i$

end if

until all classified correctly

- Nothing happens if classified correctly
- Weight vector is linear combination
- Classifier is linear combination of inner products

$$w = \sum_{i \in I} y_i x_i$$

$$f(x) = \sum_{i \in I} y_i \langle x_i, x \rangle + b$$

Convergence Theorem

- If there exists some (w^*, b^*) with unit length and

$$y_i [\langle x_i, w^* \rangle + b^*] \geq \rho \text{ for all } i$$

then the perceptron converges to a linear separator after a number of steps bounded by

$$\frac{2(R^2 + 1)}{\epsilon^2} \text{ where } \|x_i\| \leq R \text{ for all } i$$

- Dimensionality independent
- Order independent (i.e. also worst case)
- Scales with ‘difficulty’ of problem

Consequences

- Only need to store errors
- This gives the perceptron a compression bound
- This is stochastic gradient descent on hinge loss

$$l(y, f(x)) = \max(0, yf(x))$$

- Fails miserably with noisy data

do NOT train your
avatar with perceptrons

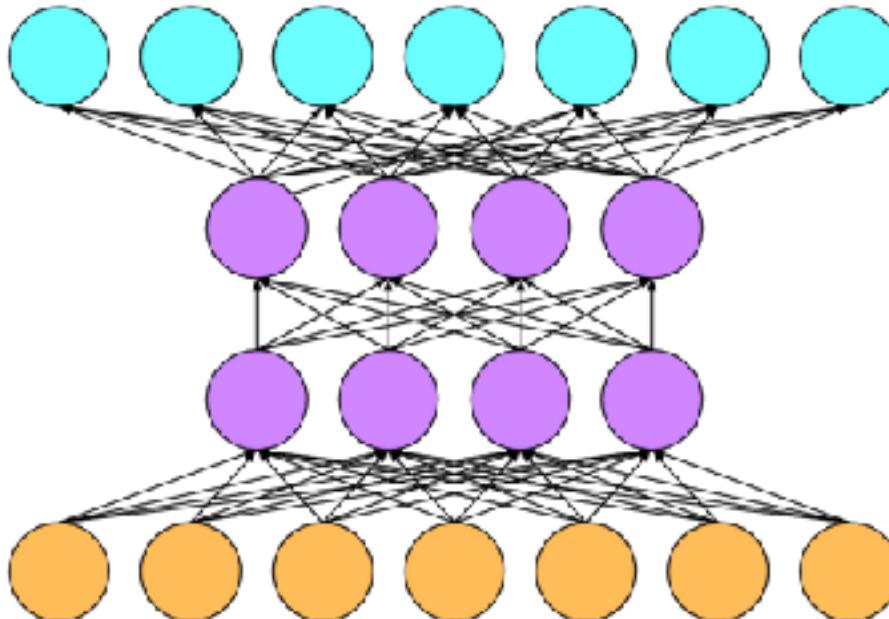


Black & White

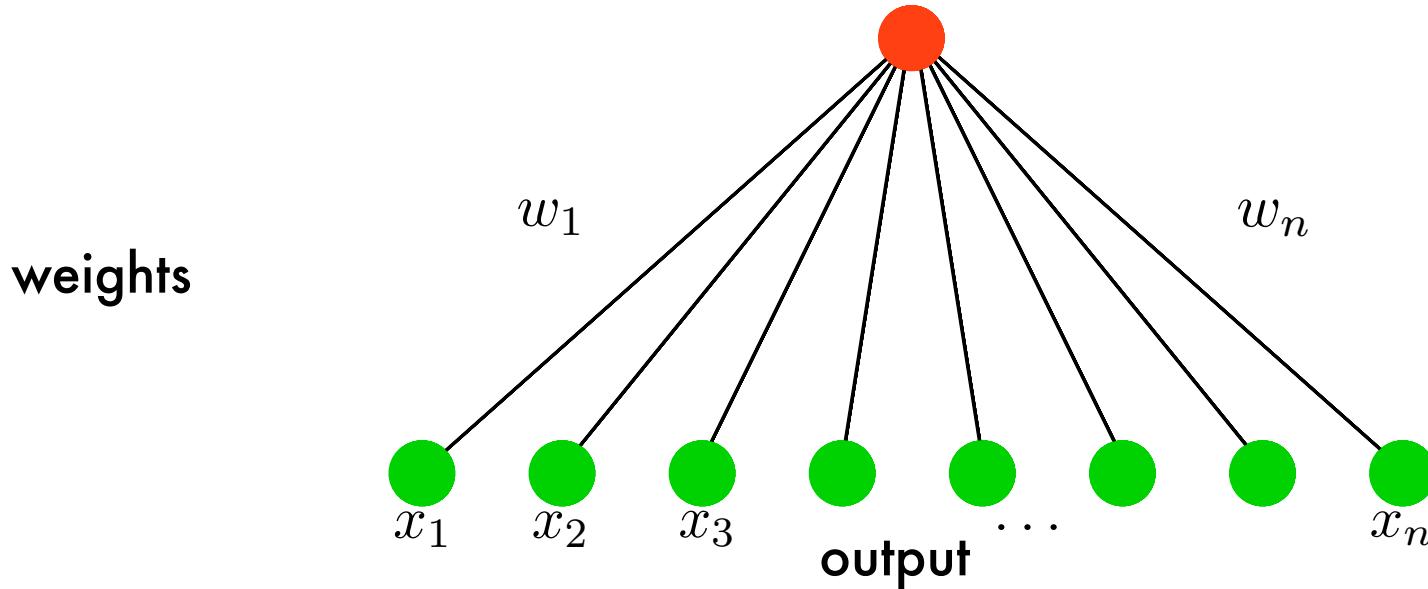
Links & Notebooks

[gluon.mxnet.io/chapter02_supervised-learning/
perceptron.html](http://gluon.mxnet.io/chapter02_supervised-learning/perceptron.html)

Multilayer Perceptron and Backpropagation



Linear models

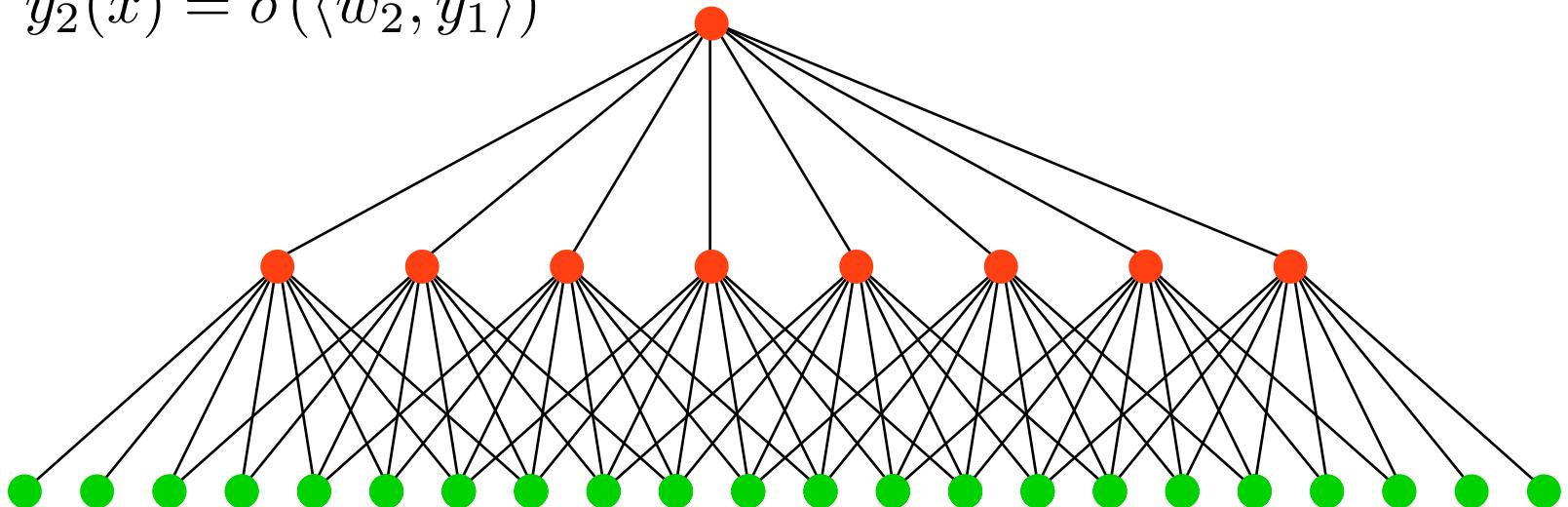


$$y(x) = \sigma(\langle w, x \rangle)$$

Multilayer Perceptron

$$y_{1i}(x) = \sigma(\langle w_{1i}, x \rangle)$$

$$y_2(x) = \sigma(\langle w_2, y_1 \rangle)$$

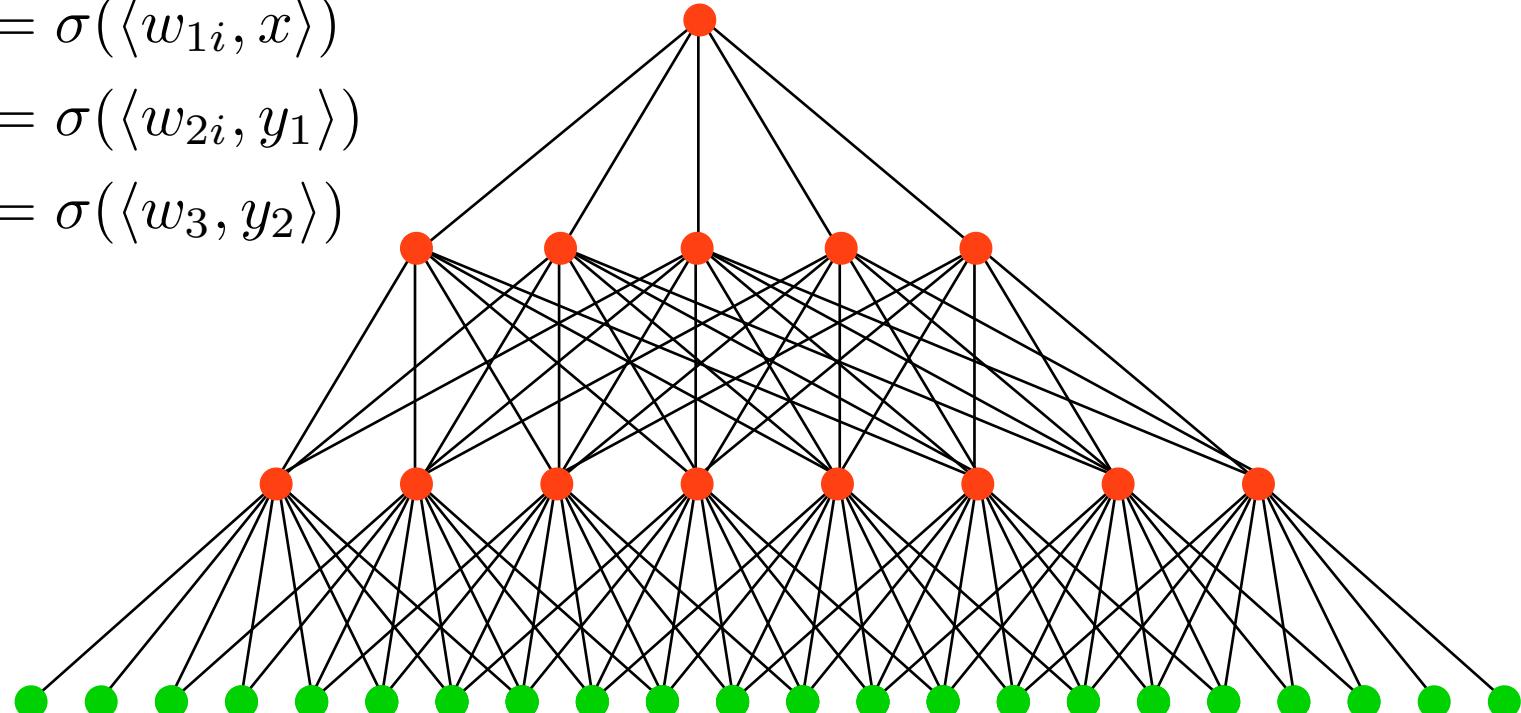


Multilayer Perceptron

$$y_{1i}(x) = \sigma(\langle w_{1i}, x \rangle)$$

$$y_{2i}(x) = \sigma(\langle w_{2i}, y_1 \rangle)$$

$$y_3(x) = \sigma(\langle w_3, y_2 \rangle)$$



Multilayer Perceptron Training

- **Hidden Representations**

$$y_i = W_i x_i + b_i$$

fully connected

$$y_i = \sigma(x_i)$$

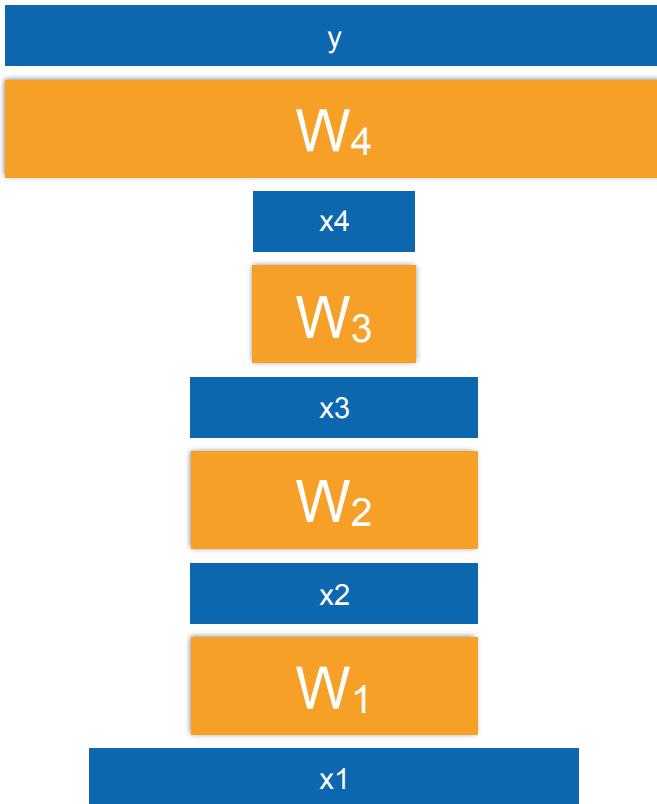
nonlinearity

- **Loss Function is just another layer**

$$y_i = l(\text{label}, x_i)$$

- **Optimization**

$$w \leftarrow w - \eta \partial_w l(\text{label}, \text{input})$$



Multilayer Perceptron Training

- **Layer Representation**
(each layer performs a transformation)

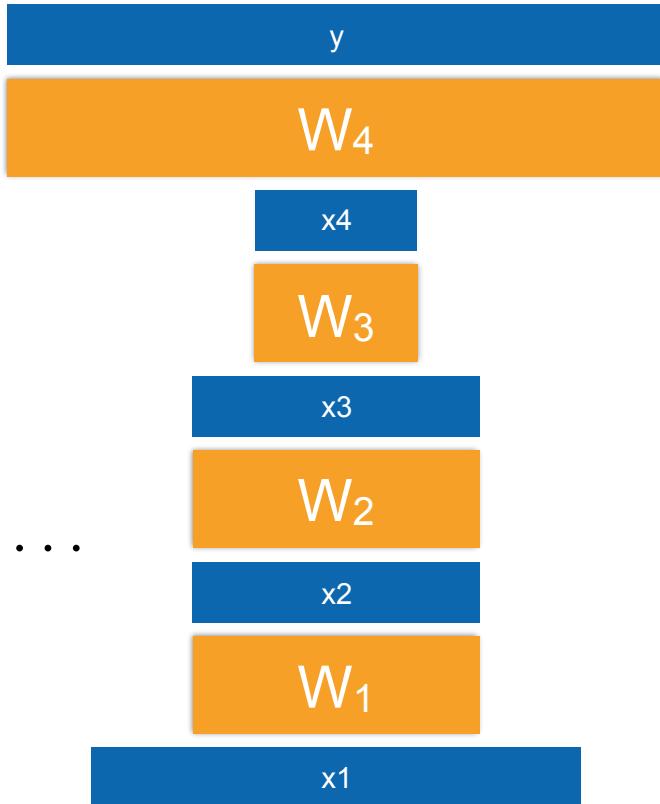
$$x_{i+1} = f_i(x_i, w_i)$$

- **Gradient**

$$x_{i+1} = f_i(x_i, w_i) = f_i(f_{i-1}(x_{i-1}, w_{i-1})) \dots$$

- **Use the chain rule**

$$\partial_x f(g(x)) = f'(g(x))g'(x)$$



Backpropagation

- **Layer Representation**
(each layer performs a transformation)

$$x_{i+1} = f_i(x_i, w_i)$$

- **Gradient**

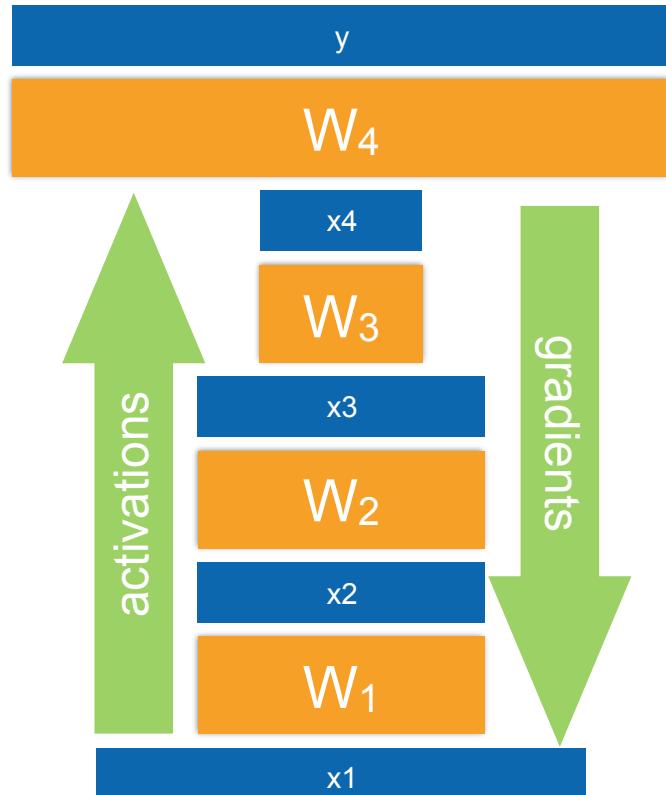
$$\partial_{w_1} x_2(x) = \partial_{w_1} f_1(x_1, w_1)$$

$$\partial_{w_1} x_3(x) = \partial_{x_2} f_2(x_2, w_2) \partial_{w_1} x_2(x)$$

...

$$\partial_{w_i} x_{i+1}(x) = \partial_{w_i} f_i(x_i, w_i)$$

$$\partial_{w_i} x_{j+1}(x) = \partial_{x_j} f_j(x_j, w_j) \partial_{w_i} x_j(x) \text{ if } i > j$$



Backpropagation

- **Layer Representation**
(each layer performs a transformation)

$$x_{i+1} = f_i(x_i, w_i)$$

- **Gradient**

$$\partial_{w_i} x_{i+1}(x) = \partial_{w_i} f_i(x_i, w_i)$$

$$\partial_{w_i} x_{j+1}(x) = \partial_{x_j} f_j(x_j, w_j) \partial_{w_i} x_j(x)$$

$$= \partial_{x_j} f_j(x_j, w_j) \partial_{x_{j-1}} f_{j-1}(x_{j-1}, w_{j-1}) \partial_{w_i} x_{j-1}(x)$$

incoming
gradient

multiply
with this

still to
compute

Backpropagation by hand

- **Two quantities**

$$\partial_w x'(x) = \partial_w f(x, w)$$

$$\partial_x x'(x) = \partial_x f(x, w)$$

- **Sigmoid**

$$x'(x) = \sigma(x) \implies \partial_x x'(x) = \sigma'(x)$$

(element-wise multiply)

- **Fully connected layer**

$$x'(x) = Wx \implies \begin{aligned} \partial_x x'(x) &= W^\top \\ \partial_W x'(x) &= 1 \cdot x \end{aligned}$$

That was complicated! MXNet takes care of this for you ...

```
import mxnet as mx
from mxnet import gluon, autograd
... #load up some data

net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.Dense(128, activation='relu'))
    net.add(gluon.Dense(64, activation='relu'))
    net.add(gluon.Dense(1))

cross_entropy = gluon.loss.SoftmaxCrossEntropyLoss()

with autograd.record():
    output = net(data)
    loss = cross_entropy(output, label)
loss.backward()
```

forward pass

nonlinearity

second layer

loss function

backward pass



Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

Matrices and Vectors

- A Simple ML algorithm

```
initialize function f(x,w)
```

```
for (x,y) in data:
```

```
    l = loss(y, f(x,w))
```

```
    g = gradient(l)
```

```
    w = w - eta * g
```

- Need vectors to deal with w
- Need matrices to deal with chain rule for loss gradient
- Need high performance linear algebra



NDArray (Linear Algebra on GPUs in Python)

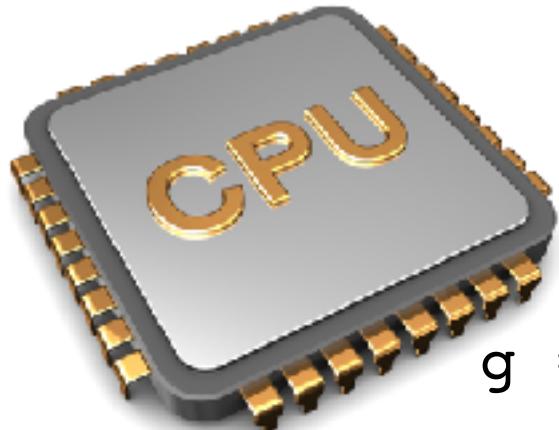
- NumPy
 - Limited to CPUs
 - No automatic differentiation
 - **Blocking (returns only once computation is performed)**
- NDArray
 - Multiple CPUs / GPUs via device context
 - Distributed systems in the cloud
 - Nonblocking Python frontend (no GIL)
 - Lazy evaluation

NDArray Device Contexts

```
context = mx.cpu()
```

```
context = mx.gpu(0)
```

```
context = mx.gpu(1)
```



```
g = copyto(c)
```

```
g = c.as_in_context(mx.gpu(0))
```

Automatic Differentiation

```
x = nd.array([[1, 2], [3, 4]])  
x.attach_grad()  
with ag.record():  
    y = x * 2  
    z = y * x  
z.backward()
```

**AutoGrad computes gradients
without a declared compute graph**

Links & Notebooks

http://gluon.mxnet.io/chapter01_crashcourse/ndarray.html

http://gluon.mxnet.io/chapter01_crashcourse/linear-algebra.html

http://gluon.mxnet.io/chapter01_crashcourse/autograd.html

Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

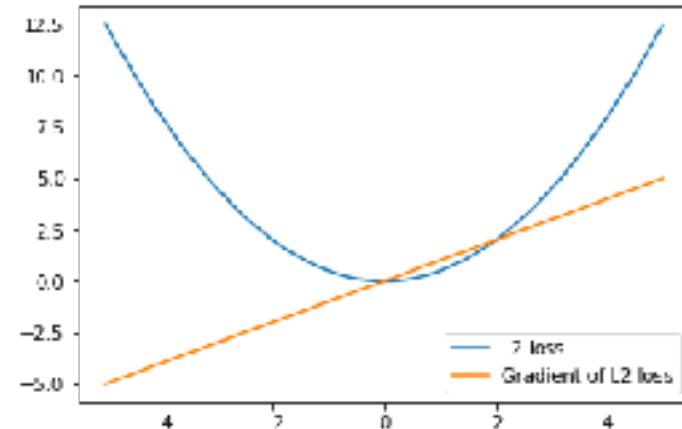
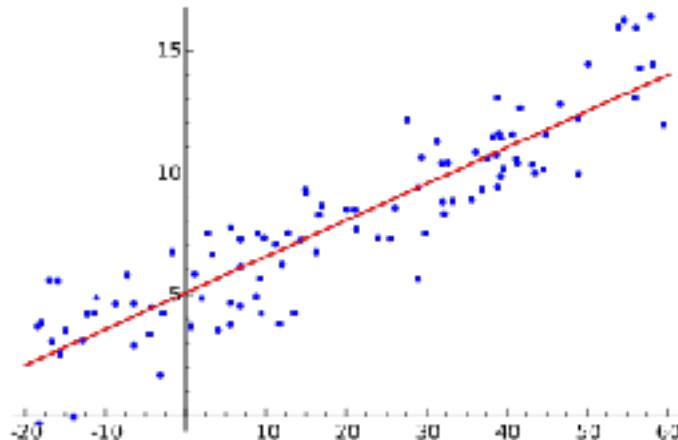
Linear Regression — for “How Much?” problems

- Model

$$f(x) = \langle w, x \rangle + b$$

- Loss

$$l(y, f(x)) = \frac{1}{2}(y - f(x))^2$$



Linear regression (analytic solution)

$$\ell(\mathbf{y}, \hat{\mathbf{y}}) = \|\mathbf{y} - X\mathbf{w}\|_2^2$$

$$= \sum_{i=1}^n (y_i - \mathbf{w}^T \mathbf{x}_i)^2$$

$$= (\mathbf{y} - X\mathbf{w})^T (\mathbf{y} - X\mathbf{w})$$

$$\frac{\partial \ell(\mathbf{y}, \hat{\mathbf{y}})}{\partial \mathbf{w}} = 2(-X)^T (\mathbf{y} - X\mathbf{w}) = 0$$

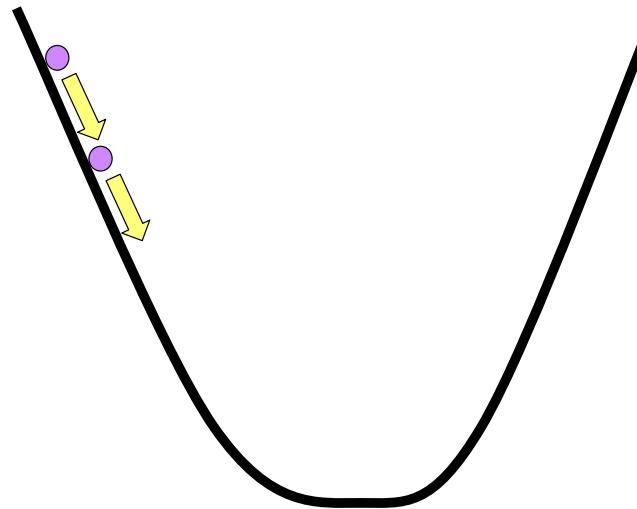
$$-2X^T \mathbf{y} + 2X^T X\mathbf{w} = 0$$

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

$$\mathbf{w} = (X^T X)^{-1} X^T \mathbf{y}$$

Stochastic gradient descent

- Initialize w, b at random (or at zero)
- For each data point, calculate gradient $g = dl/dw$
- Apply update to minimize loss $w \leftarrow w - \eta g$



Logistic Regression — for “yes/no?” problems

- Model (single layer)

$$f(x) = \sigma(Wx + b)$$

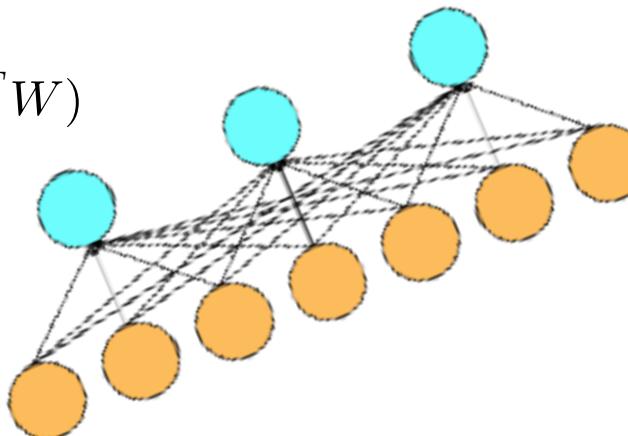
- Loss (negative log likelihood)

$$-\sum y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)$$

Multiclass logistic regression

For a Single Example:

- Label: $y \in 1, \dots, k$
- Output: $\hat{y} = \text{softmax}(\mathbf{x}^T W)$
- Weights: $W \in \mathbb{R}^{d \times k}$
- Input: $\mathbf{x} \in \mathbb{R}^d$



The softmax operation

- Now we have k outputs
- We want our output layer to assign *probabilities* to each output
- The softmax operation enforces that our output is a valid multinomial distribution

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Multiclass logistic regression

- We represent our label as a one-hot over all the possible outputs
- A one hot has value one for true label, 0 elsewhere:
[0 0 0 0 0 0 0 1 0 0 0]
- To perform maximum likelihood, we try to maximize the probability assigned to the correct class
- This loss function is called cross-entropy

$$l(\hat{\mathbf{y}}, \mathbf{y}) = -\sum y_i \log \hat{y}_i$$

Links & Notebooks

http://gluon.mxnet.io/chapter02_supervised-learning/linear-regression-scratch.html

http://gluon.mxnet.io/chapter02_supervised-learning/linear-regression-gluon.html

http://gluon.mxnet.io/chapter02_supervised-learning/softmax-regression-scratch.html

http://gluon.mxnet.io/chapter03_deep-neural-networks/mlp-scratch.html

http://gluon.mxnet.io/chapter03_deep-neural-networks/mlp-gluon.html

Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

Where is Waldo?



image credit - wikipedia

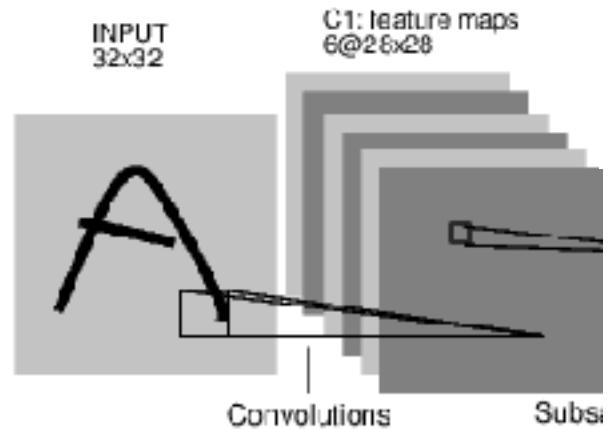
Where is Waldo?

Waldo model is global

Waldo information is local.

Convolutional Layers

- Images have translation invariance (to some extent)
- Low level is mostly edge and feature detectors
- Usually via convolution (plus nonlinearity)



Convolutional Layers

- **Feature Locality**

Relevant information only in neighborhood of pixel

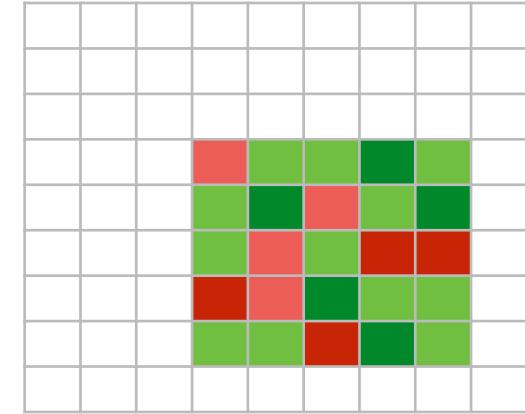
$$y_{ij} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} W_{ij,ab} x_{i+a,j+b}$$



- **Translation Invariance**

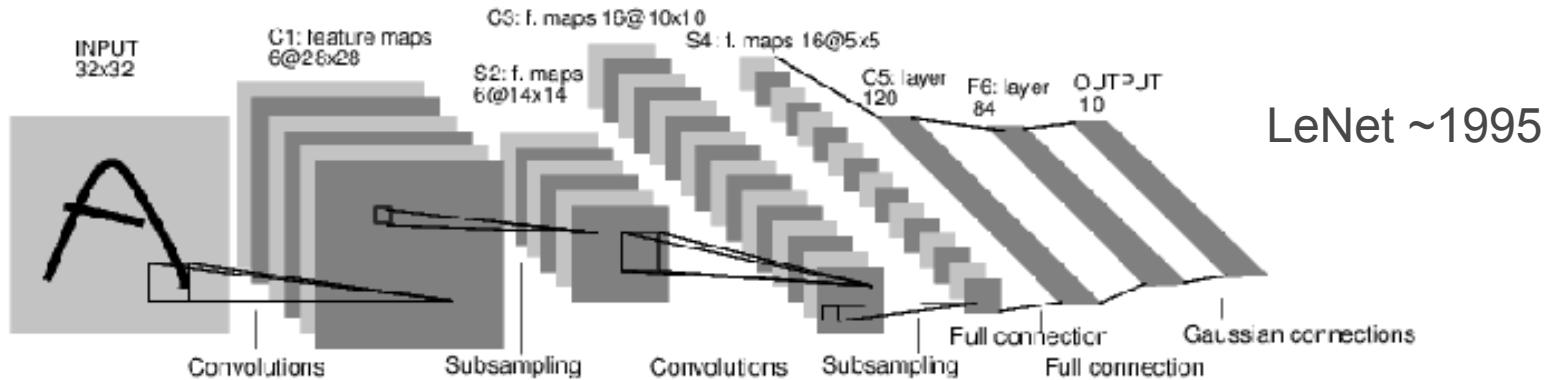
Weights invariant relative to shift in point of view

$$y_{ij} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} W_{ab} x_{i+a,j+b}$$



Subsampling & MaxPooling

- Multiple convolutions blow up dimensionality



- Subsampling - average over patches (works OK)
- MaxPooling - pick the maximum over patches (much better)

image credit - Le Cun et al, 1998

LeNet in MXNet

```
net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.nn.Conv2D(channels=20, kernel_size=5, activation='tanh'))
    net.add(gluon.nn.AvgPool2D(pool_size=2))
    net.add(gluon.nn.Conv2D(channels=50, kernel_size=5, activation='tanh'))
    net.add(gluon.nn.AvgPool2D(pool_size=2))
    net.add(gluon.nn.Flatten())
    net.add(gluon.nn.Dense(500, activation='tanh'))
    net.add(gluon.nn.Dense(10))

loss = gluon.loss.SoftmaxCrossEntropyLoss()

(size and shape inference is automatic)
```

More Layers

- **The usual suspects**

- gluon.nn.Dense(units=50, ...)
- gluon.nn.Activation(activation='relu')
- gluon.nn.Dropout(rate=0.3)
- gluon.nn.BatchNorm(...)
- gluon.nn.Embedding(...) for text

- **Convolutions**

- gluon.nn.Conv1D, 2D, 3D
- gluon.nn.Conv1DTranspose, 2D, 3D for Deconvolution

- **Pooling Layers**

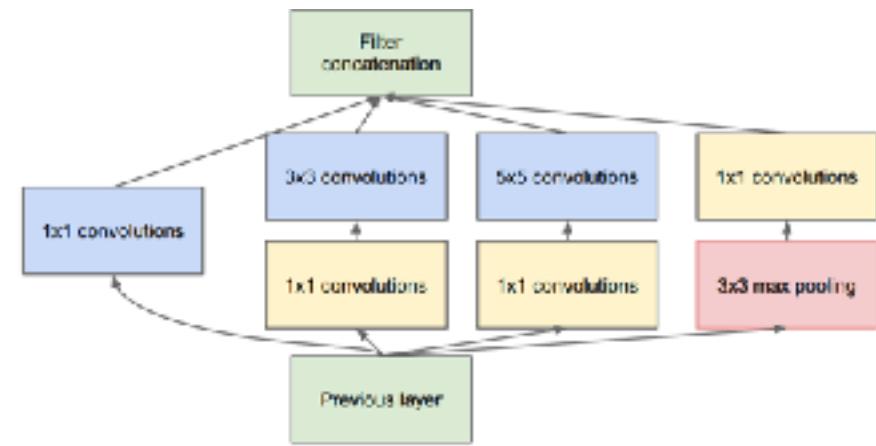
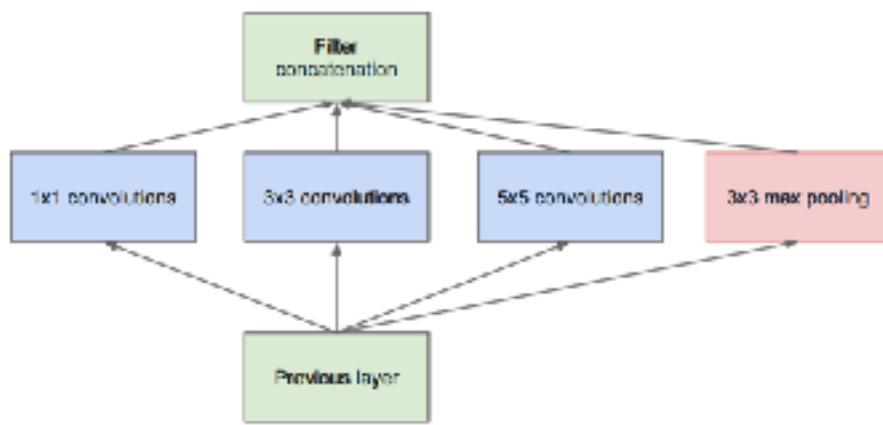
- gluon.nn.MaxPool1D, 2D, 3D, gluon.nn.AvgPool1D, 2D, 3D

Links & Notebooks

http://gluon.mxnet.io/chapter04_convolutional-neural-networks/cnn-gluon.html

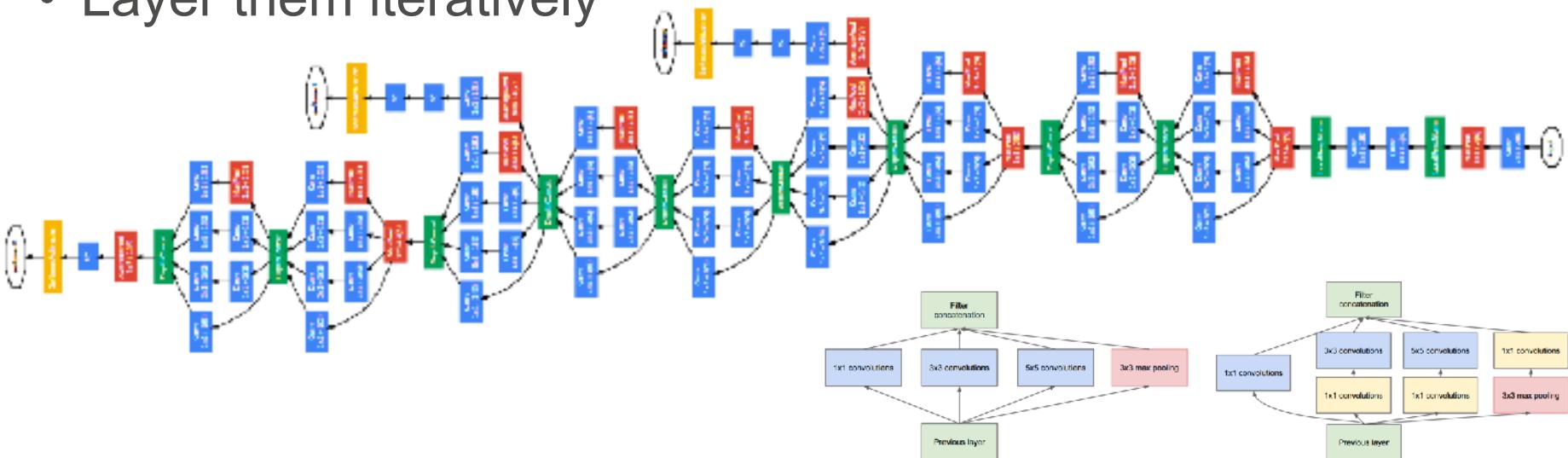
Fancy structures

- Compute different filters
- Compose one big vector from all of them
- Layer them iteratively



Fancy structures

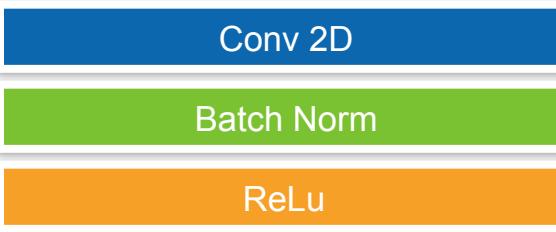
- Compute different filters
- Compose one big vector from all of them
- Layer them iteratively



Networks of networks - Sequential Composition

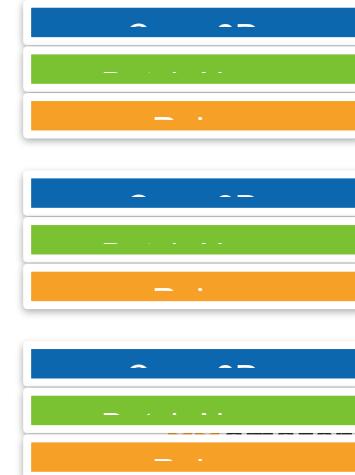
- Define Single Block

```
def _make_basic_conv(**kwargs):  
    out = nn.HybridSequential(prefix='')  
    out.add(nn.Conv2D(use_bias=False, **kwargs))  
    out.add(nn.BatchNorm(epsilon=0.001))  
    out.add(nn.Activation('relu'))  
    return out
```



- Compose Multiple Blocks into Branch

```
def _make_branch(*conv_settings):  
    out = nn.HybridSequential(prefix='')  
    out.add(nn.MaxPool2D(pool_size=3, strides=2))  
    setting_names = ['channels', 'kernel_size', 'strides', 'padding']  
    for setting in conv_settings:  
        kwargs = {}  
        for i, value in enumerate(setting):  
            if value is not None:  
                kwargs[setting_names[i]] = value  
        out.add(_make_basic_conv(**kwargs))  
    return out
```



Networks of networks - Parallel Composition

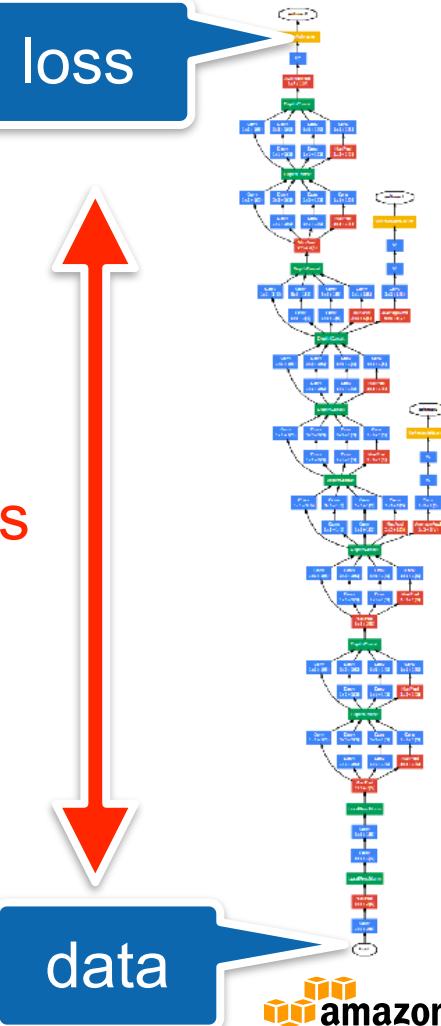
- Parallel composition of branches

```
def _make_A(pool_features, prefix):
    out = HybridConcurrent(concat_dim=1, prefix=prefix)
    with out.name_scope():
        out.add(_make_branch(None, (64, 1, None, None)))
        out.add(_make_branch(None, (48, 1, None, None),
                            (64, 5, None, 2)))
        out.add(_make_branch(None, (64, 1, None, None),
                            (96, 3, None, 1),
                            (96, 3, None, 1)))
    out.add(_make_branch('avg', (pool_features, 1, None, None)))
    return out
```



Batch Normalization

- Loss occurs at last layer
 - Last layers learn quickly
- Data is inserted at bottom layer
 - Bottom layers change - **everything** changes
 - Last layers need to relearn many times
 - Slow convergence
- This is like covariate shift
Can we avoid changing last layers while learning first layers?



Batch Normalization

- Can we avoid changing last layers while learning first layers?
- Fix mean and variance

$$\mu_B = \frac{1}{|B|} \sum_{i \in B} x_i \text{ and } \sigma_B^2 = \frac{1}{|B|} \sum_{i \in B} (x_i - \mu_B)^2 + \epsilon$$

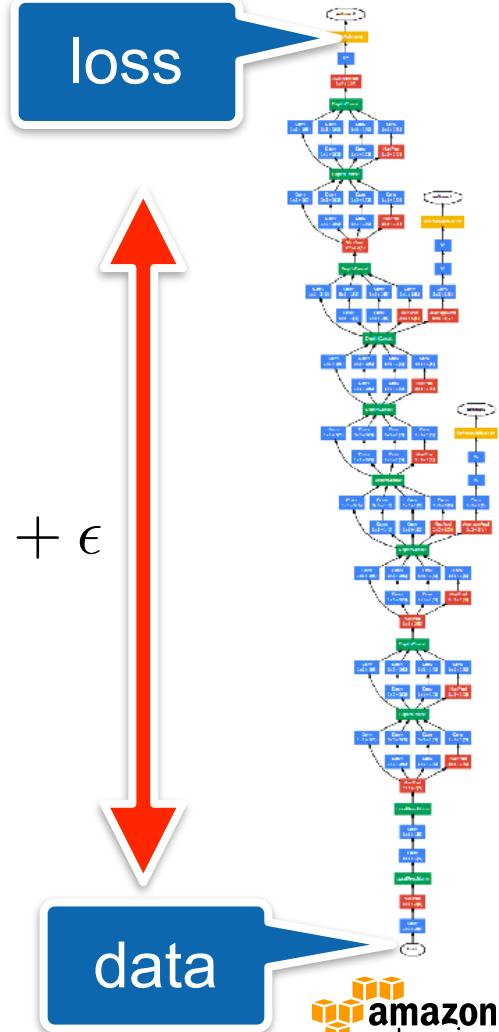
and adjust it separately

$$x_{i+1} = \gamma \frac{x_i - \mu_B}{\sigma_B} + \beta$$

variance

mean

data



Model Zoo

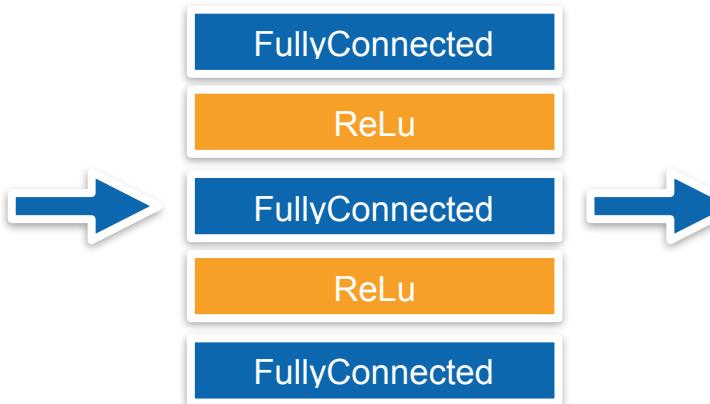
- Lots more networks at `mxnet.gluon.model_zoo.vision` (with all the model parameters you want)
 - AlexNet (because, why not)
 - SqueezeNet
 - Inception
 - VGG
 - DenseNet
 - ResNet

... import from PyTorch is trivial ...

Hybridization (a JIT compiler for Networks)

- Loops and control flow are easy to understand
- Runtime engine needs to parse them each time (slow)
- Use JIT compiler to convert into compute graph
(no need to parse the code more than once)

```
x = F.relu(self.fc1(x))  
x = F.relu(self.fc2(x))  
return self.fc3(x)
```



JSON
Network definition
Serialize weights

Links & Notebooks

http://gluon.mxnet.io/chapter04_convolutional-neural-networks/deep-cnns-alexnet.html
http://gluon.mxnet.io/chapter04_convolutional-neural-networks/very-deep-nets-vgg.html

Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

Recall - Stochastic Gradient Descent

- **Optimization Problem**

$$\underset{w}{\text{minimize}} \frac{1}{m} \sum_{i=1}^m l(y_i, f(x_i, w)) + \lambda \|w\|_2^2$$

- **Stochastic Gradient Approximation**

- **Gradient on Minibatch**

$$g_B := \frac{1}{b} \sum_{i \in B} \partial_w l(y_i, f(x_i, w)) \text{ and } w \leftarrow \text{Optimizer}(w, g_B)$$

- **Update w with advanced first order solver**

(Adam, AdaGrad, Momentum, Eve, SGD, SGLD ...)



Stochastic Gradient Descent

- **Stochastic Gradient Descent (SGD)**

$$w \leftarrow (1 - 2\eta\lambda)w - \eta g_b$$

- **SGD with Momentum and Clipping**

$$s \leftarrow \mu s + \eta \text{clip}(g_b) + \eta\lambda w$$

$$w \leftarrow w - s$$

Quite often learning rate is piecewise constant (this yields better accuracy in practice, albeit convergence is slower)

- **Improved conditioning, momentum, precision ...**

- SGD
- DCASGD
- NAG
- Adam
- SGLD
- AdaGrad
- RMSProp
- AdaDelta
- Ftrl
- Adamax
- Nadam

Momentum

- Average over recent gradients

- Helps with local minima
- Flat (noisy) gradients



momentum

$$m_t = (1 - \lambda)m_{t-1} + \lambda g_t$$

$$w_t \leftarrow w_t - \eta_t g_t - \tilde{\eta}_t m_t$$

- Can lead to oscillations for large momentum
- Nesterov's accelerated gradient

$$m_{t+1} = \mu m_t + \epsilon g(w_t - \mu m_t)$$

$$w_{t+1} = w_t - m_{t+1}$$

Minibatch

- Aggregate gradients over a few instances before updating
 - Reduces variance in gradients
 - Better for vectorization (GPUs)
(vector, vector) < (vector, matrix) < (matrix, matrix)
 $\langle x, x' \rangle < Mx < MX$
 - Large minibatch may need lots of memory
(and may slow updates).
- **For multiple GPUs you need to scale up mini batch size
(256GPUs and 20 samples/GPU = mini batch of 5120!)**

Learning rate decay

- **Constant**

(requires schedule for piecewise constant, tricky)

- **Useful hack**

Keep learning rate constant until no improvement on validation data, then reduce rate.

- **Polynomial decay**

Canonical choice for convex solvers

$$\eta = \frac{\alpha}{\sqrt{\beta + t}}$$

- **Exponential decay**

not recommended

AdaGrad

- **Adaptive learning rate (preconditioner)**

$$\eta_{ij}(t) = \frac{\eta_0}{\sqrt{K + \sum_t g_{ij}^2(t)}}$$

- For directions with large gradient, decrease learning rate aggressively to avoid instability
- If gradients start vanishing, learning rate decrease reduces, too
- **Local variant**

$$\eta_{ij}(t) = \frac{\eta_t}{\sqrt{K + \sum_{t'=t-\tau}^t g_{ij}^2(t')}}$$

Capacity control

- Minimizing loss can lead to overfitting
- **Weight decay**

$$w_{t+1} \leftarrow w_t - \eta_t g_t$$

$$w_{t+1} \leftarrow (1 - \lambda)w_t - \eta_t g_t$$

- **Gradient clipping**

- Overheated GPU
- Numerical instabilities
- Hacky but necessary (e.g. deep architectures)

prevents parameters
from diverging

Dropout

Goals

- **Avoid parameter sensitivity**
- **Distributed representation** (info in > 1 dim)

Solution

- Zero out half of the nodes in each layer (Dropout)
- Or zero-out weights (DropConnect), slightly better

Srivastava, Hinton, Krizhevski, Sutskever, Salakhutdinov <http://jmlr.org/papers/v15/srivastava14a.html>
<http://cs.nyu.edu/~wanli/dropc/>



Dropout

- Without dropout

$$y_i = Wx_i$$

$$x_{i+1} = \sigma(y_i)$$

- With dropout

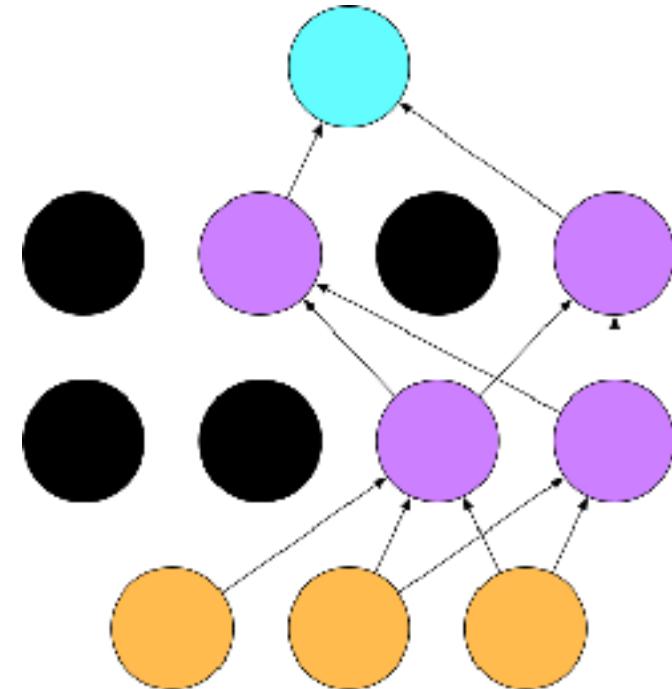
$$z_{ij} = \xi_{ij}x_{ij}$$

$$y_i = Wz_i$$

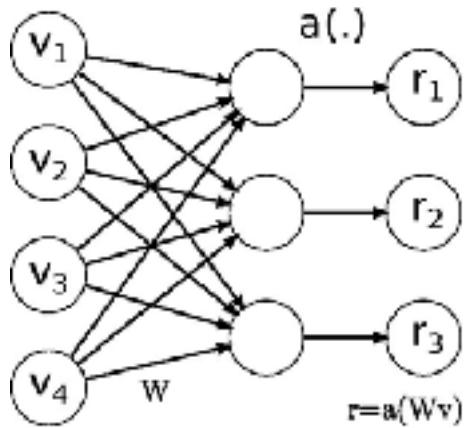
$$x_{i+1} = \sigma(y_i)$$

Draw a new mask for every step

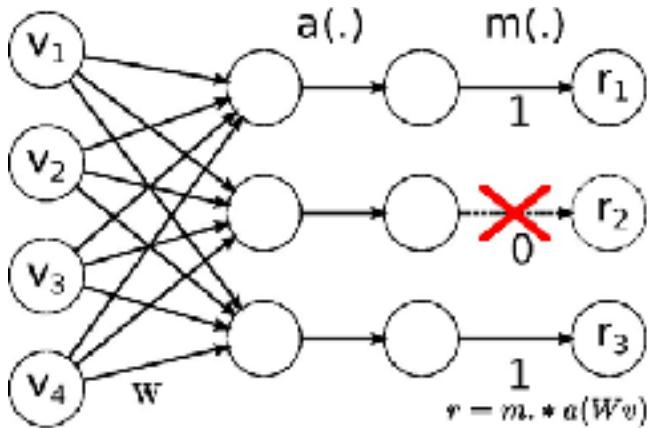
- Update via backprop



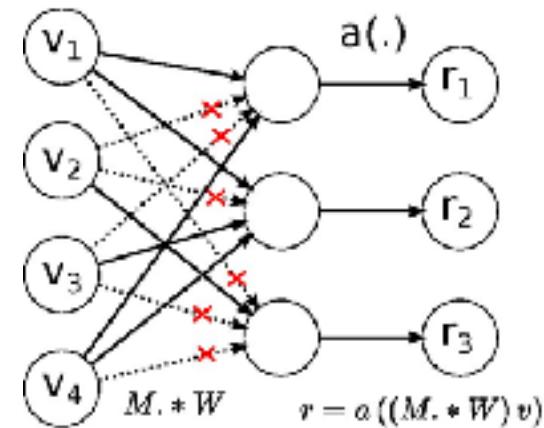
Dropout & DropConnect



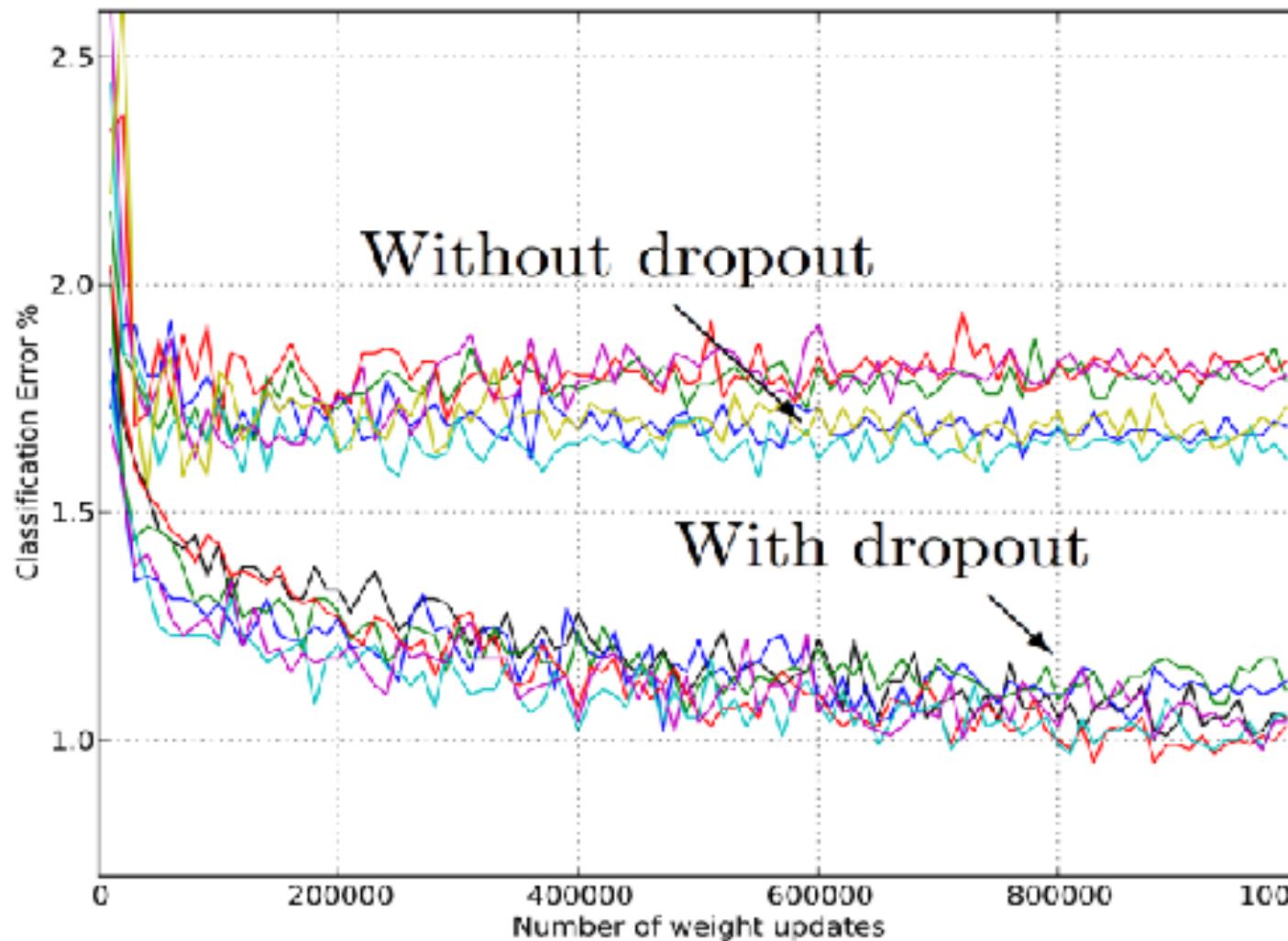
Regular



Dropout



DropConnect



Links & Notebooks

http://gluon.mxnet.io/chapter03_deep-neural-networks/mlp-dropout-scratch.html
http://gluon.mxnet.io/chapter03_deep-neural-networks/mlp-dropout-gluon.html

Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

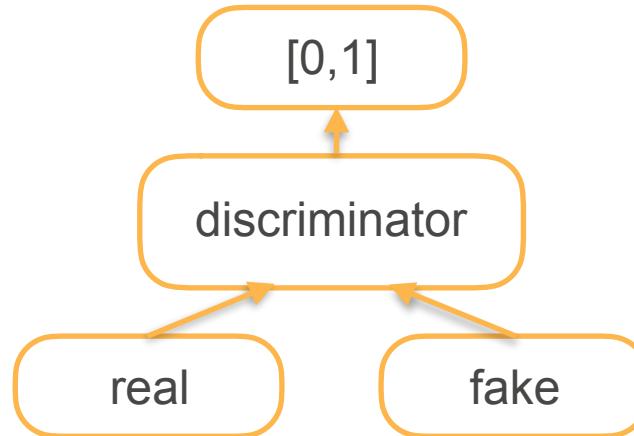
4.3. Parallel and Distributed Training

Some goals of unsupervised learning

- **Traditional Statistics**
Given data X , try to find some distribution p that would have generated X .
- **Why?**
 - Assess how well some observation x' fits with the data, by evaluating $p(x')$.
 - Draw new fake data x' that looks like it came from X
 - Summarize X concisely (sufficient statistics)

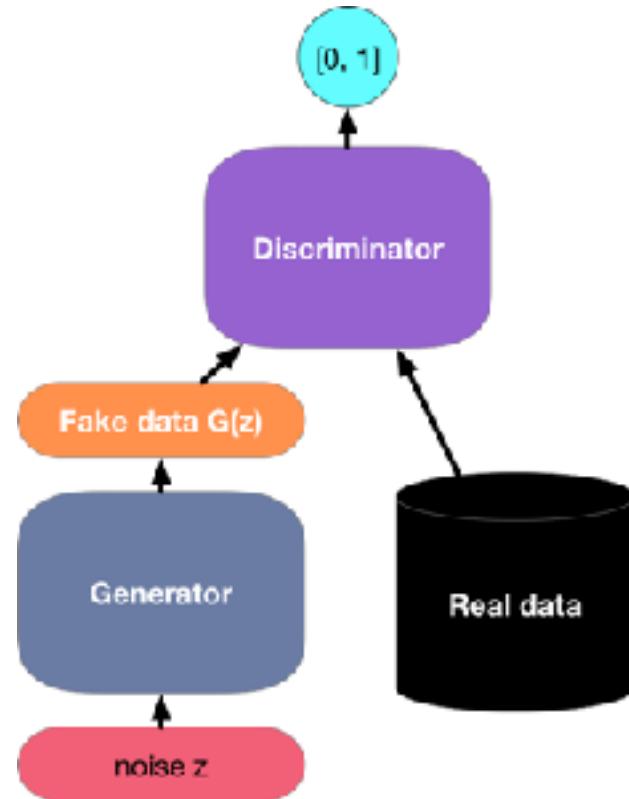
Generative adversarial networks

- **From discrimination to generation**
 - Since 2012, we've known that convnets are great at classifying images
 - Can leverage this capacity for discrimination to learn generative models?



Key Idea - Focus on generating data

- **Generator network**
 - Start with some random z
 - Transform it via neural network
 - $x' \sim g(z)$
- **Discriminator network**
 - distinguishes x from fake x'



Objective Functions

- Discriminator Network uses logistic regression
(binary classification between real and fake data)

$$\frac{1}{n} \sum_{i=1}^n \log(1 + \exp(d(x_i))) + \mathbf{E}_{x' \sim q(x')} [\log(1 + \exp(-d(x')))]$$

real data

fake data

- Generative Network tries to fool data
(maximize acceptance or minimize rejection)

$$\mathbf{E}_u [\log(1 + \exp(d(g(z)))]$$

Multiple Gradients

- ‘Normal’ SGD loop
 - Iterate over data and compute loss
 - Update parameters w.r.t. loss gradient
- GAN loop
 - Iterate over data and sample from generator
 - Compute loss and gradient on real & fake data
 - Update parameters for discriminator
 - Sample from generator
 - Compute loss and gradient on real data
 - Update parameters for generator



different
parameters

Links & Notebooks

http://gluon.mxnet.io/chapter14_generative-adversarial-networks/gan-intro.html

http://gluon.mxnet.io/chapter14_generative-adversarial-networks/dcgan.html

http://gluon.mxnet.io/chapter14_generative-adversarial-networks/pixel2pixel.html

Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

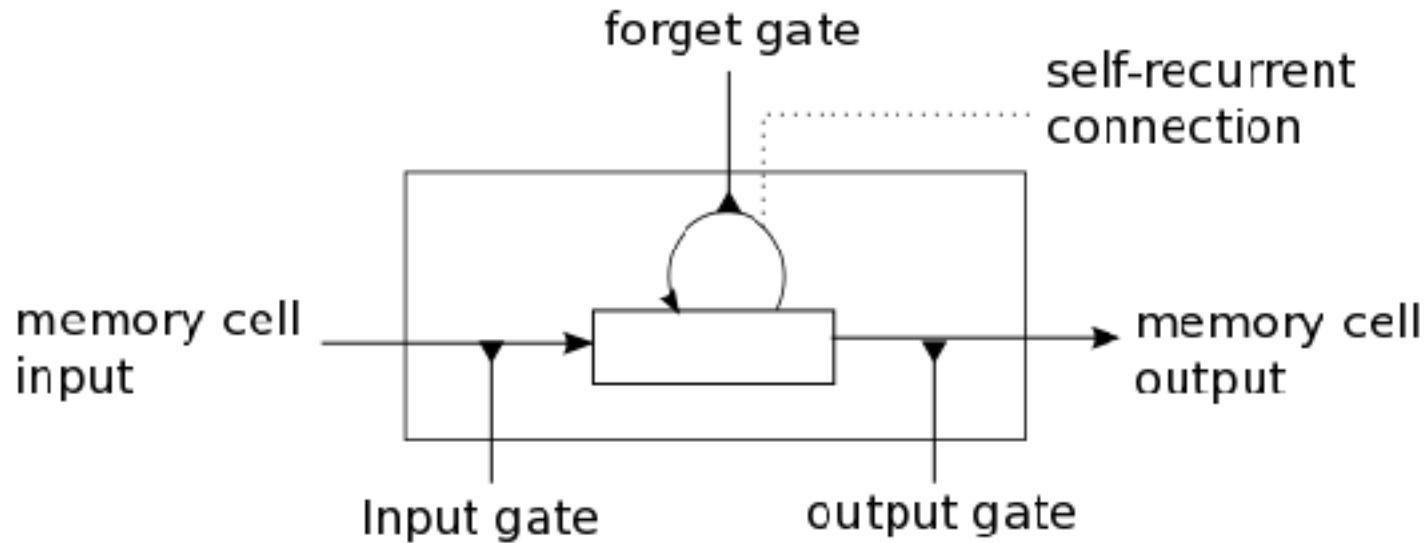
4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

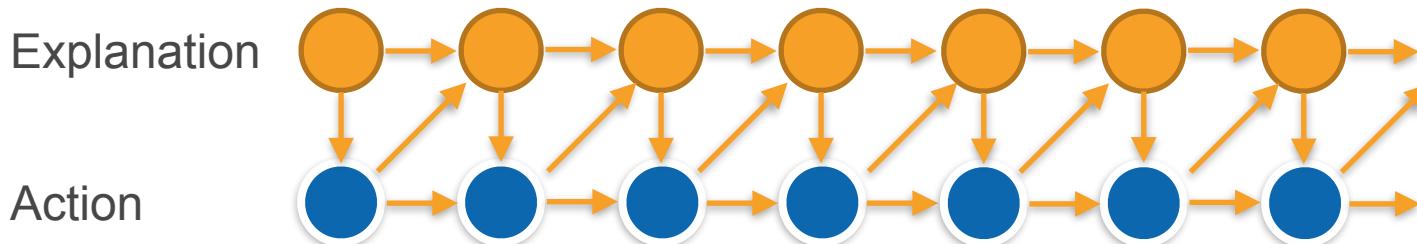
4.3. Parallel and Distributed Training

Sequence Models



Latent Variable Models

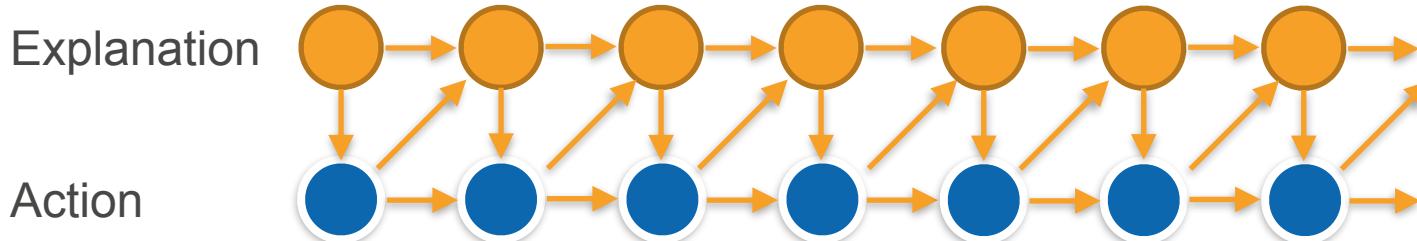
- **Temporal sequence of observations**
Purchases, likes, app use, e-mails, ad clicks, queries, ratings
- **Latent state to explain behavior**
 - Clusters (navigational, informational queries in search)
 - Topics (interest distributions for users over time)
 - Kalman Filter (trajectory and location modeling)



Latent Variable Models

- **Temporal sequence of observations**
Purchases, likes, app use, e-mails, ad clicks, queries, ratings
- **Latent state to explain behavior**

Are the parametric models really true?

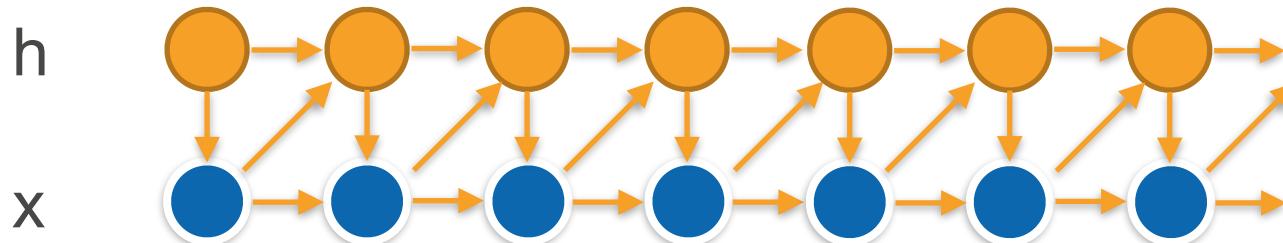


Latent Variable Models

- **Temporal sequence of observations**
Purchases, likes, app use, e-mails, ad clicks, queries, ratings
- **Latent state to explain behavior**
 - Nonparametric model / spectral
 - Use data to determine shape
 - Sidestep approximate inference

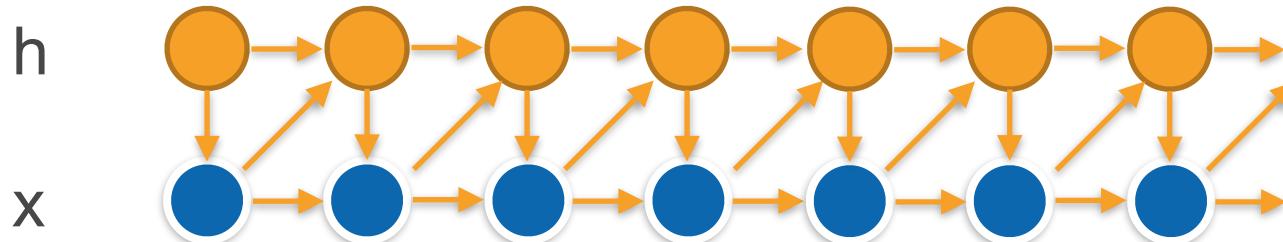
$$h_t = f(x_{t-1}, h_{t-1})$$

$$x_t = g(x_{t-1}, h_t)$$



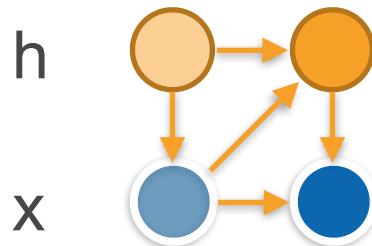
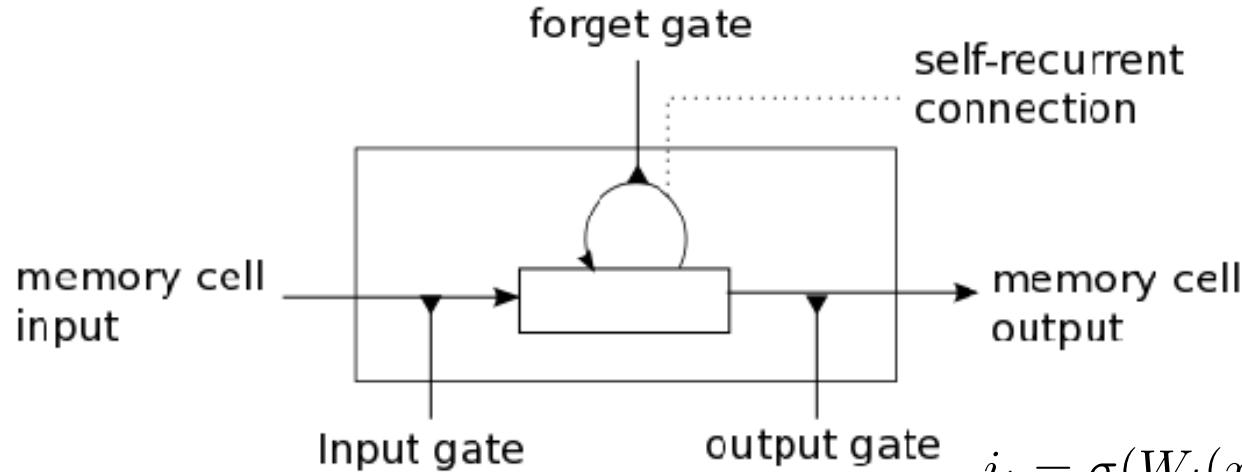
Latent Variable Models

- **Temporal sequence of observations**
Purchases, likes, app use, e-mails, ad clicks, queries, ratings
- **Latent state to explain behavior**
 - Plain deep network = RNN
 - Deep network with attention = LSTM / GRU ...
(learn when to update state, how to read out)



Long Short Term Memory

Hochreiter & Schmidhuber, 1997



$$i_t = \sigma(W_i(x_t, h_t) + b_i)$$

$$f_t = \sigma(W_f(x_t, h_t) + b_f)$$

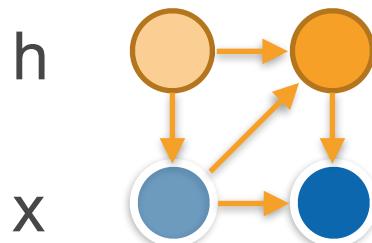
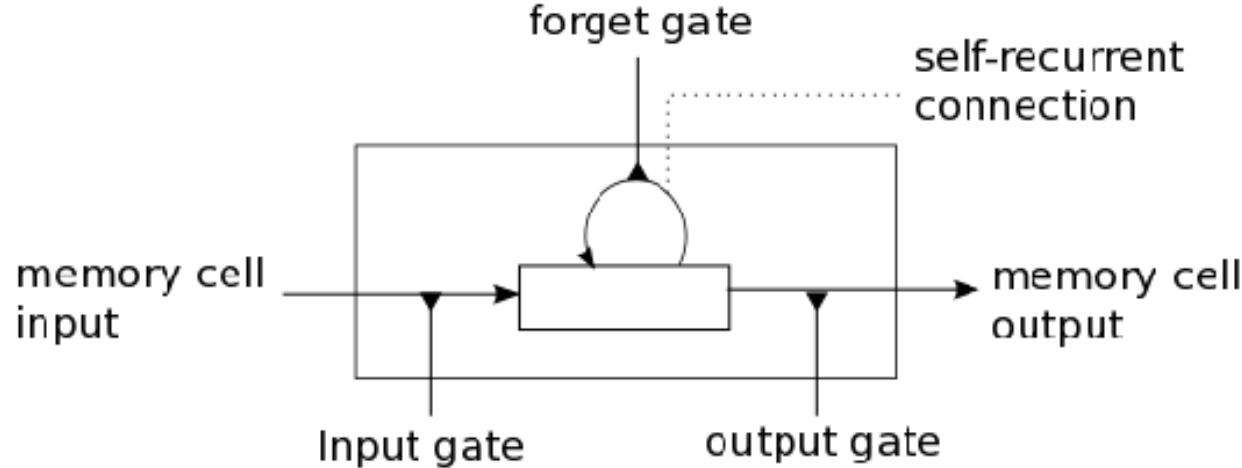
$$z_{t+1} = f_t \cdot z_t + i_t \cdot \tanh(W_z(x_t, h_t) + b_z)$$

$$o_t = \sigma(W_o(x_t, h_t, z_{t+1}) + b_o)$$

$$h_{t+1} = o_t \cdot \tanh z_{t+1}$$

Long Short Term Memory

Hochreiter & Schmidhuber, 1997



$$(z_{t+1}, h_{t+1}, o_t) = \text{LSTM}(z_t, h_t, x_t)$$

```
rnn.LSTM(num_hidden, num_layers, dropout, input_size)
```

The anatomy of an RNN language model

- **Statistical Model**

$$p(x) = \prod_{i=0}^{n-1} p(x_{i+1} | x_i \dots x_1) = \prod_{i=0}^{n-1} p(x_{i+1} | \text{LSTM}(x_i, h_i))$$

- **Truncated Backpropagation Through Time (BPTT)**

- Sequence is often too long for exact computation
- Truncate and predict smaller segments (approximation)

$$p(x) = \prod_{j=0}^{\lfloor n/L \rfloor} \prod_{i=0}^{L-1} p(x_{i+1+jL} | \text{LSTM}(x_{i+jL}, h_{i+jL}))$$

The anatomy of an RNN language model

- **Data**
 - Some sequence to predict the next ...
 - Break into batches
 - Tokenize words (or characters)
 - Truncate for limited Backprop through Time (BPTT)
- **Encoder**
 - Map token i into vector v_i
(one-hot encoding followed by an MLP)

The anatomy of an RNN language model

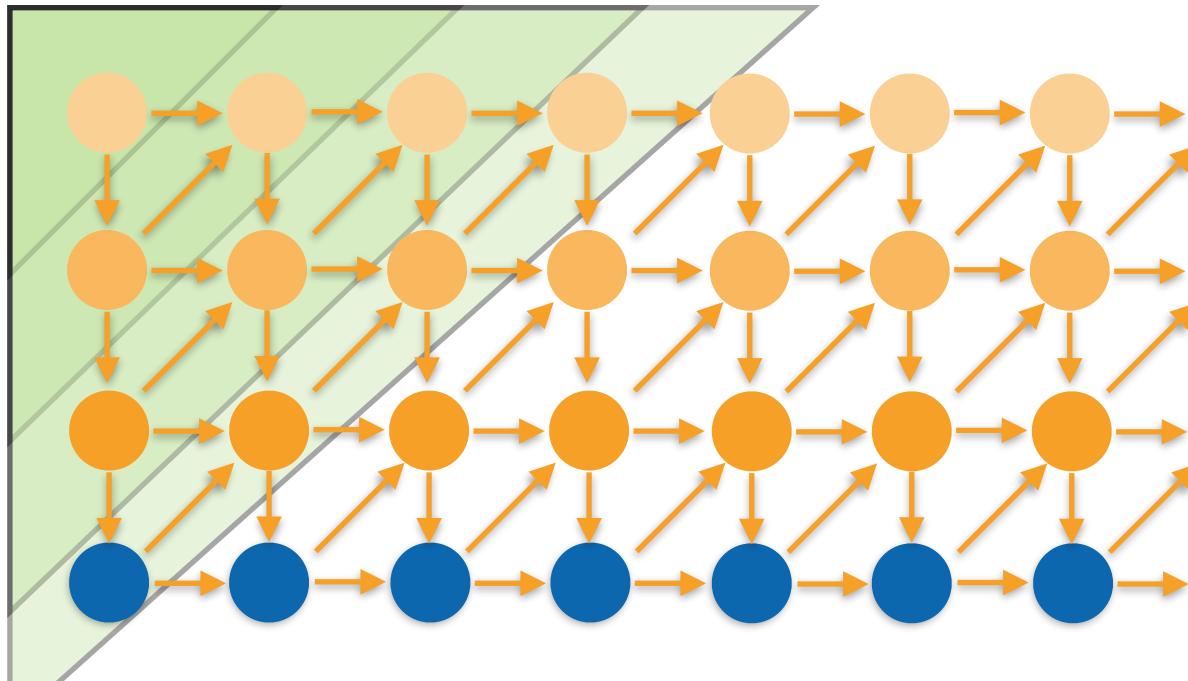
- **RNN**
 - Initialize with some state s
 - Recurse through sequence to compute
$$(h_{i+1}, o_{i+1}) = \text{LSTM}(h_i, x_i)$$
- **Decoder**
 - Softmax decoding for output, i.e
$$p(x_i | o_i) = \text{softmax}(\text{MLP}(o_i))$$
 - Better decoders possible (beam search, WFST ...)

Links & Notebooks

http://gluon.mxnet.io/chapter05_recurrent-neural-networks/simple-rnn.html
http://gluon.mxnet.io/chapter05_recurrent-neural-networks/lstm-scratch.html
http://gluon.mxnet.io/chapter05_recurrent-neural-networks/rnns-gluon.html

Fusion for RNNs

- GPUs too fast to call a matrix-vector product every time
- Fuse CUDA kernels (automatic if you use Gluon RNNs)



Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

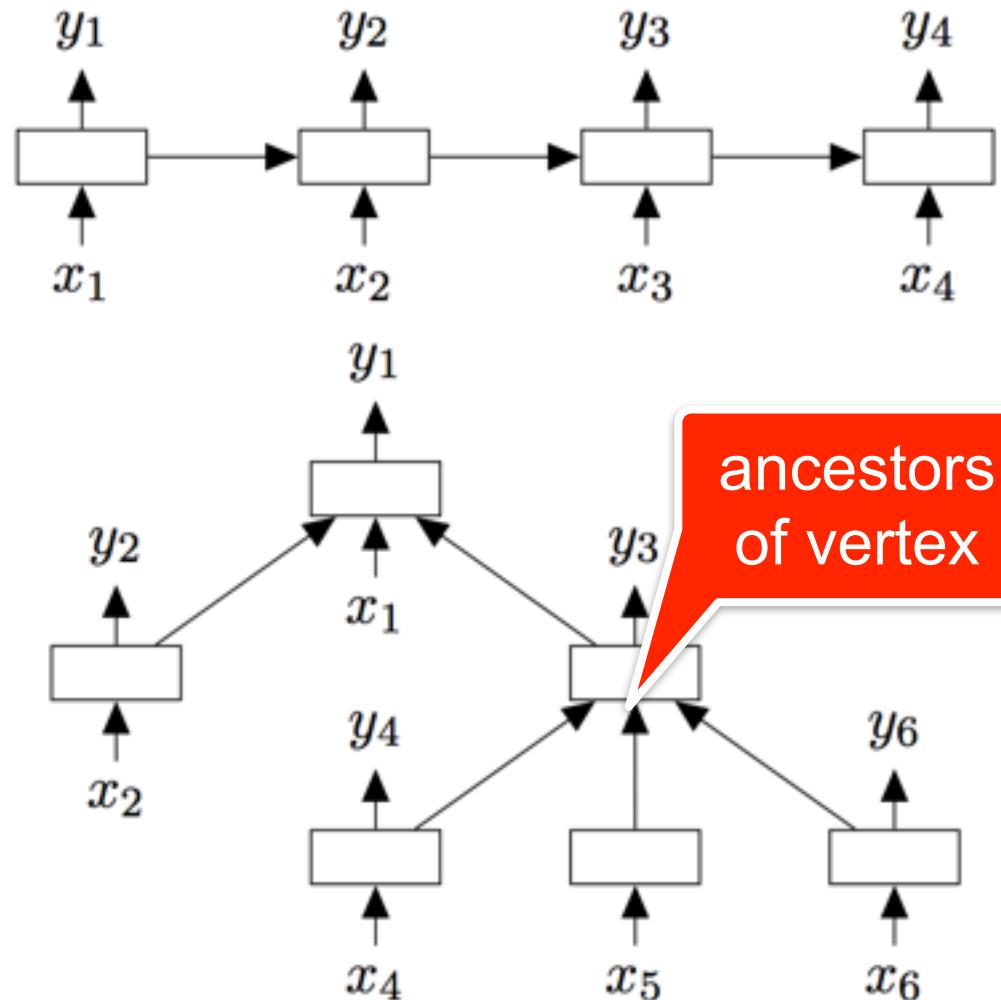
4.3. Parallel and Distributed Training

Tree LSTMs

- **LSTM**
sequential dependence
between states
- $o_t = f(x_t, h_{t-1})$
- $(c_t, h_t) = g(x_t, h_{t-1}, c_{t-1})$
- **Tree LSTM**
hierarchical dependence

$$o_t = f(x_t, H_{t-1})$$

$$(c_t, h_t) = g(x_t, H_{t-1}, C_{t-1})$$



Tree LSTMs (Tsai et al, 2015)

- **Applications**

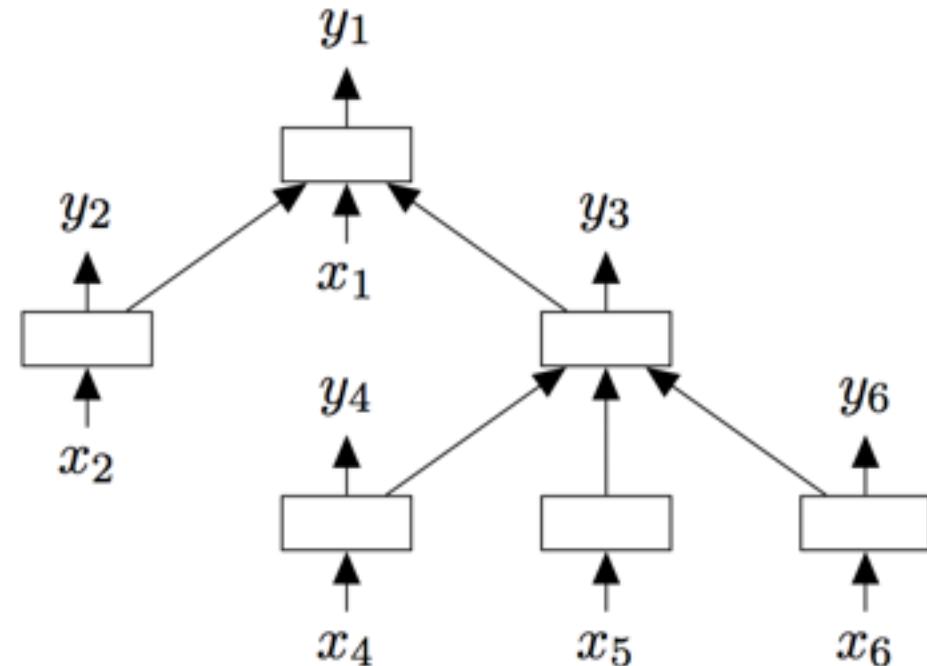
- dependency parsing
- sentiment
- document similarity

- **Problems**

- **Variable inputs H and C**
- Specialization (left, right)
- **Model has variable dependency structure**

$$o_t = f(x_t, H_{t-1})$$

$$(c_t, h_t) = g(x_t, H_{t-1}, C_{t-1})$$



LSTM

$$z = Wx_t + Uh_{t-1} + b$$

$$[i_t, f_t, o_t, u_t] = [\sigma(z_i), \sigma(z_f), \sigma(z_o), \tanh(z_u)]$$

$$c_t = i_t \circ u_t + f_t \circ c_{t-1}$$

$$h_t = o_t \circ \tanh(c_t)$$

Tree LSTM

individual
forget weights

$$\tilde{h} = \sum_{j \in C(t)} h_j$$

aggregate
hidden state

$$z = Wx_t + U\tilde{h} + b$$

$$f_{tj} = \sigma(W_f x_t + U_f h_j + b_f)$$

$$[i_t, o_t, u_t] = [\sigma(z_i), \sigma(z_o), \tanh(z_u)]$$

$$c_t = i_t \circ u_t + \sum_{j \in C(t)} f_{tj} \circ c_j$$

$$h_t = o_t \circ \tanh(c_t)$$

LSTM

$$z = Wx_t + Uh_{t-1} + b$$

$$[i_t, f_t, o_t, u_t] = [\sigma(z_i), \sigma(z_f), \sigma(z_o), \tanh(z_u)]$$

$$c_t = i_t \circ u_t + f_t \circ c_{t-1}$$

$$h_t = o_t \circ \tanh(c_t)$$

Tree LSTM (individual weights)

individual weight

$$z = Wx_t + U [h_j, \dots h_{j'}] + b$$

$$[i_t, o_t, [f_{tj} \dots f_{tj'}], u_t] = [\sigma(z_i), \sigma(z_o), [\sigma(z_{fj}) \dots \sigma(z_{fj'})], \tanh(z_u)]$$

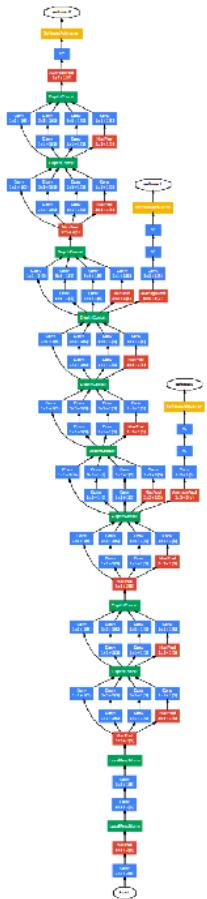
$$c_t = i_t \circ u_t + \sum_{j \in C(t)} f_{tj} \circ c_{tj}$$

$$h_t = o_t \circ \tanh(c_t)$$

individual forget weight

120

Static and Dynamic Graphs



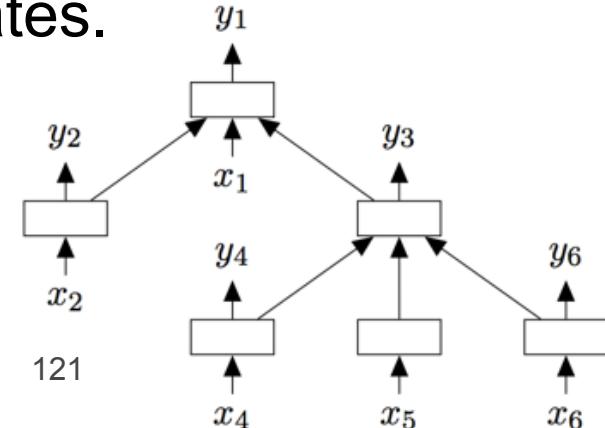
- **Static Graph**

Predefined concatenation of layers. Easy to do with a symbolic framework.

- **Dynamic Graph**

Implicitly defined via code in **imperative** framework.
Automatic differentiation for updates.

- Compute solution
- Update parameters



Links & Notebooks

http://gluon.mxnet.io/chapter09_natural-language-processing/tree-lstm.html

Hybridization - (JIT for Deep Networks)

- **Forward Definition**

```
def hybrid_forward(self, F, x):  
    print('type(x): {}, F: {}'.format(type(x).__name__, F.__name__))  
    x = F.relu(self.fc1(x))  
    x = F.relu(self.fc2(x))  
    return self.fc3(x)
```

- **Without Hybridization**

Executes the full program (including print)

- **With Hybridization**

Executes only the compute graph that was **implicitly** defined. JIT for compute graphs (no need to parse repeatedly), as efficient as symbolic definition.

123

Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

A brief history of computers

	1970	1980	1990	2000	2010	2020
Data (samples)	10^2 (e.g. iris)	10^3	10^4 OCR	10^{7-8} web	10^{10} advertising	10^{12} social nets
RAM	1kB	100kB	10MB	100MB	1GB	100GB
CPU	100kF (8080)	1MF (80186)	10MF (80486)	1GF (Intel Core)	100GF NVIDIA	>200TF

A brief history of computers

	1970	1980	1990	2000	2010	2020
Data (samples)	10^2 (e.g. iris)	10^3	old school neural nets	10^{7-8} web	10^{10} advertising	modern deep nets
RAM	1kB	100kB	10MB	100MB	1GB	100GB
CPU	100kF (8080)	1MF (80186)	10MF (80486)	Intel Core	NVIDIA GF	>200TF

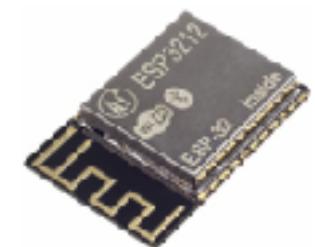
A brief history of clusters

	1970	1980	1990	2000	2010	2020
Data (samples)	10^2 (e.g. Iris)	10^3	10^4 OCR	10^{7-8} web	10^{10} advertising	10^{12} social+ads
RAM	1kB	100kB	10MB	10GB	1TB	>10TB
CPU	100kF (8080)	1MF (80186)	10MF (80486)	100GF (Intel Core)	10^{TF} NVIDIA	$>100^{PF}$
Cluster size			10	100	32x4 GPU 1000 CPU	128×8 GPU
Network		10Mb	100Mb	1Gb	10Gb	100Gb

Mobile & Sensors

Check out the
CoreML exporter

- **Mobile phones** (\$500) - Android, iOS
 - 1TF GPU / 10-100 Gflops CPU compute
 - 4GB RAM (lower bandwidth)
- **IoT** (\$50) - **Greengrass** category (Raspberry Pi)
 - 10x less (low grade mobile SoC or similar)
- **Embedded** (\$5) - **FreeRTOS** (Dash Button, ESP32)
 - 32bit, 2x200MHz, 128kB, BTLE, WiFi
- **Sensors**
 - 4k video camera (\$10), microphone array, gyro, compass, are ubiquitous (e.g. SnowballEdge)



Scaling and Machine Learning

- **Compute** grows at highest rate (**1000x due to GPUs**)
- Data grows at Kryder's law (100x)
- Memory grows more slowly (20x)
- Network grows slowly **between computers** (10x)
- Network grows rapidly **within** (100x - NVLink/QPI)

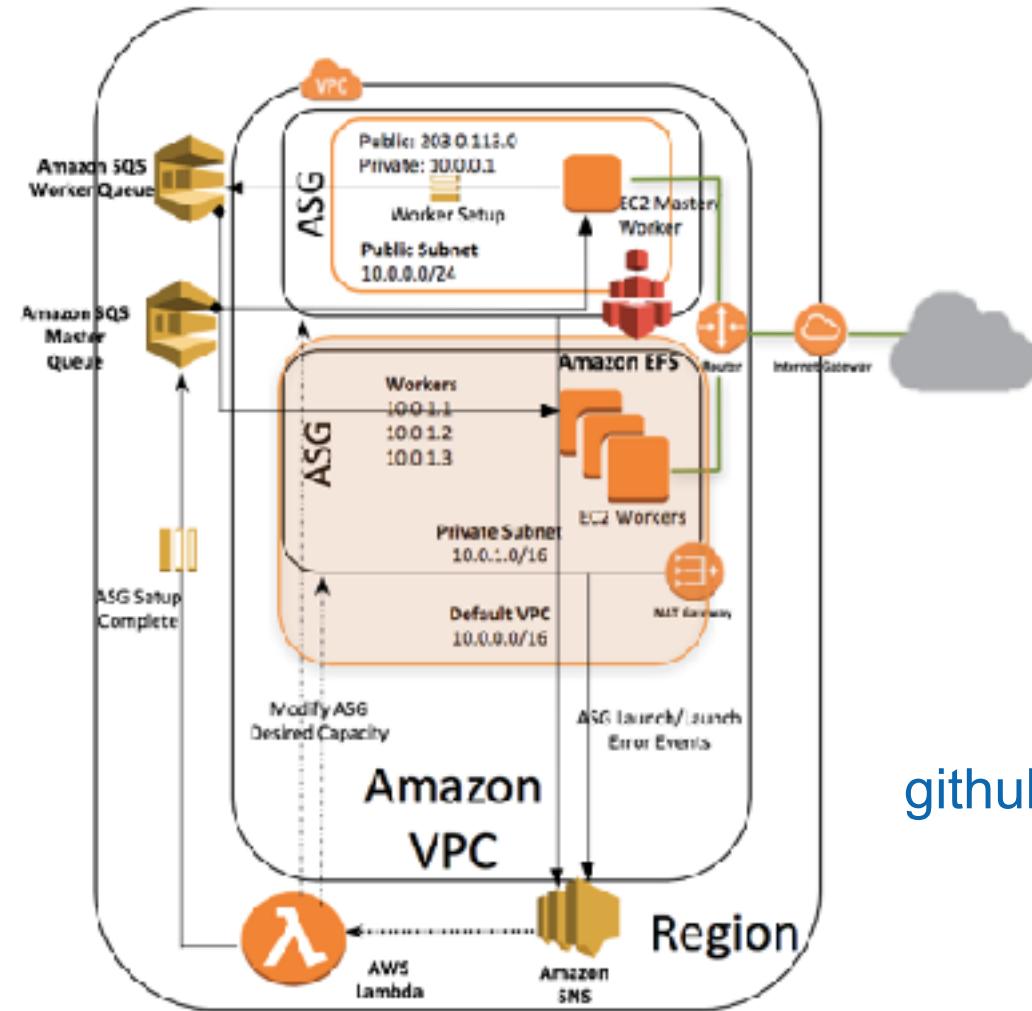
deep
learning

- Sweet spot for algorithms for data analysis
compute intensive, memory frugal, modest I/O
- **Nonparametric models** possible due to plenty of data

AWS Instances

- CPU
 - **c4.8xlarge** (36 threads, 60GB RAM, 4Gbit)
 - **m4.16xlarge** (64 threads, 256GB RAM, 10Gbit)
- GPU
 - **p3.16xlarge**
(16xNVIDIA Kepler K80, 64 threads, 732GB RAM, 20Gbit)
 - **g3.16xlarge**
(4xNVIDIA Maxwell M60, 64 threads, 488GB RAM, 20Gbit)
 - **NVIDIA Volta in the future ...**

AWS CloudFormation Template for Deep Learning



github.com/awslabs/deeplearning-cfn



AWS CloudFormation Components

- A VPC in the customer account.
- The requested number or available number of worker instances in an [Auto Scaling](#) group within the VPC. These worker instances are launched in a private subnet.
- A master instance in a separate Auto Scaling group that acts as a proxy for the workers. AWS CloudFormation places this instance within the VPC and connects it to the worker instances. This instance has both public IP addresses and DNS.
- An Amazon EFS file storage system configured in General Purpose performance mode.
- A mount target to mount Amazon EFS on the instances.
- A security group that allows external SSH access to the master instance.
- A security group that allows the master and worker instances to mount and access Amazon EFS through NFS port 2049.
- Two security groups that open ports on the private subnet for communication between the master and workers.
- An [AWS Identity and Access Management \(IAM\)](#) role that allows instances to poll [Amazon Simple Queue Service \(Amazon SQS\)](#) and access and query Auto Scaling groups and the private IP addresses of the EC2 instances.
- A NAT gateway used by the instances within the VPC to talk to the outside world.
- Two Amazon SQS queues to configure the metadata at startup on the master and the workers.
- An [AWS Lambda](#) function that monitors the Auto Scaling group's launch activities and modifies the desired capacity of the Auto Scaling group based on availability.
- An [Amazon Simple Notification Service \(Amazon SNS\)](#) topic to trigger the Lambda function on Auto Scaling events.
- AWS CloudFormation WaitCondition and WaitHandler, with a stack creation timeout of 55 minutes to complete metadata setup.

Handles all this
behind the scenes



Demo using <https://github.com/awslabs/deeplearning-cfn>



Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

Optimization Basics (Last Time)

- **Optimization Problem**

$$\underset{w}{\text{minimize}} \frac{1}{m} \sum_{i=1}^m l(y_i, f(x_i, w)) + \lambda \|w\|_2^2$$

- **Stochastic Gradient Descent**

- **Gradient on Minibatch**

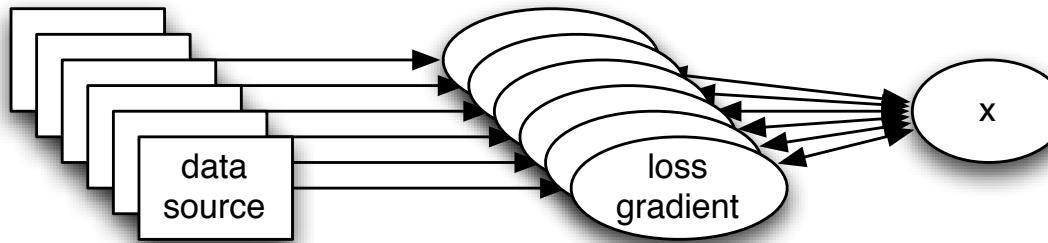
$$g_B := \frac{1}{b} \sum_{i \in B} \partial_w l(y_i, f(x_i, w)) \text{ and } w \leftarrow \text{Optimizer}(w, g_B)$$

- **More GPUs, more machines = Larger Minibatch
Synchronization - data, model, gradients**

Parallelization

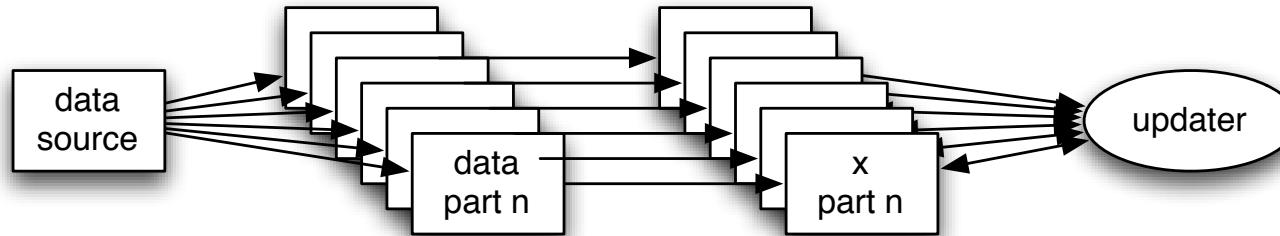
- **Data Parallel (easier & more efficient)**

Distribute data over multiple GPUs / CPUs / machines



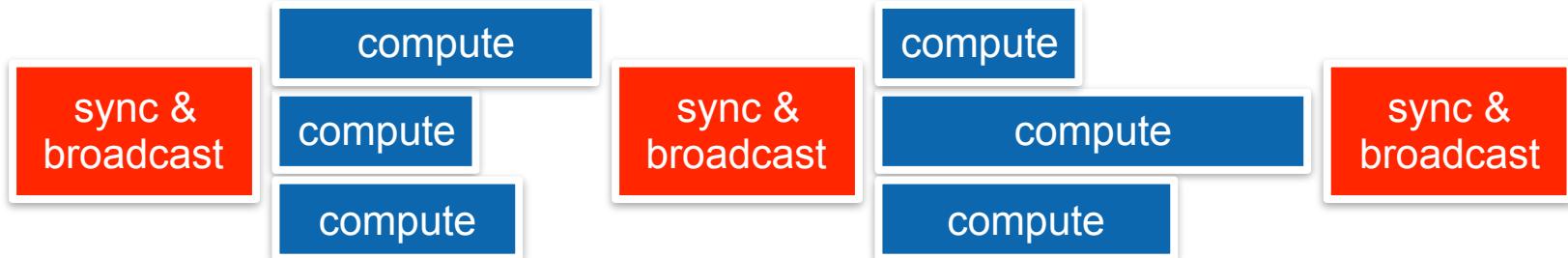
- **Model Parallel (only for huge models)**

Distribute parts of the computation over multiple GPUs / CPUs

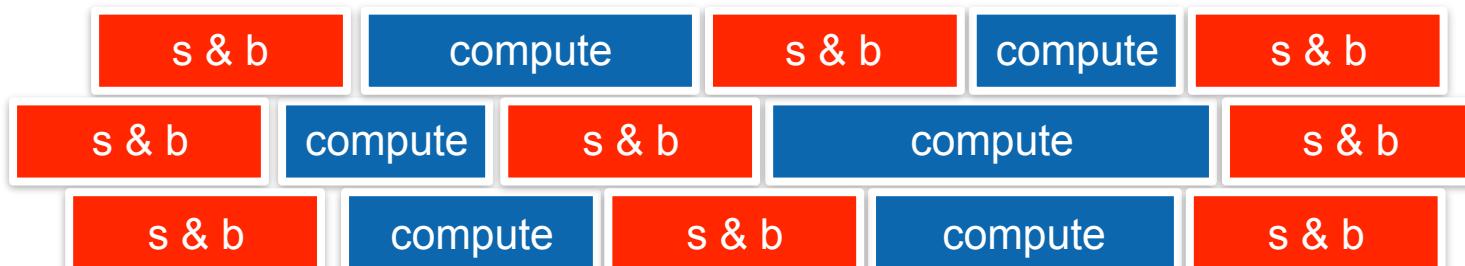


Parallelization

- **Synchronous - easy for homogeneous jobs**



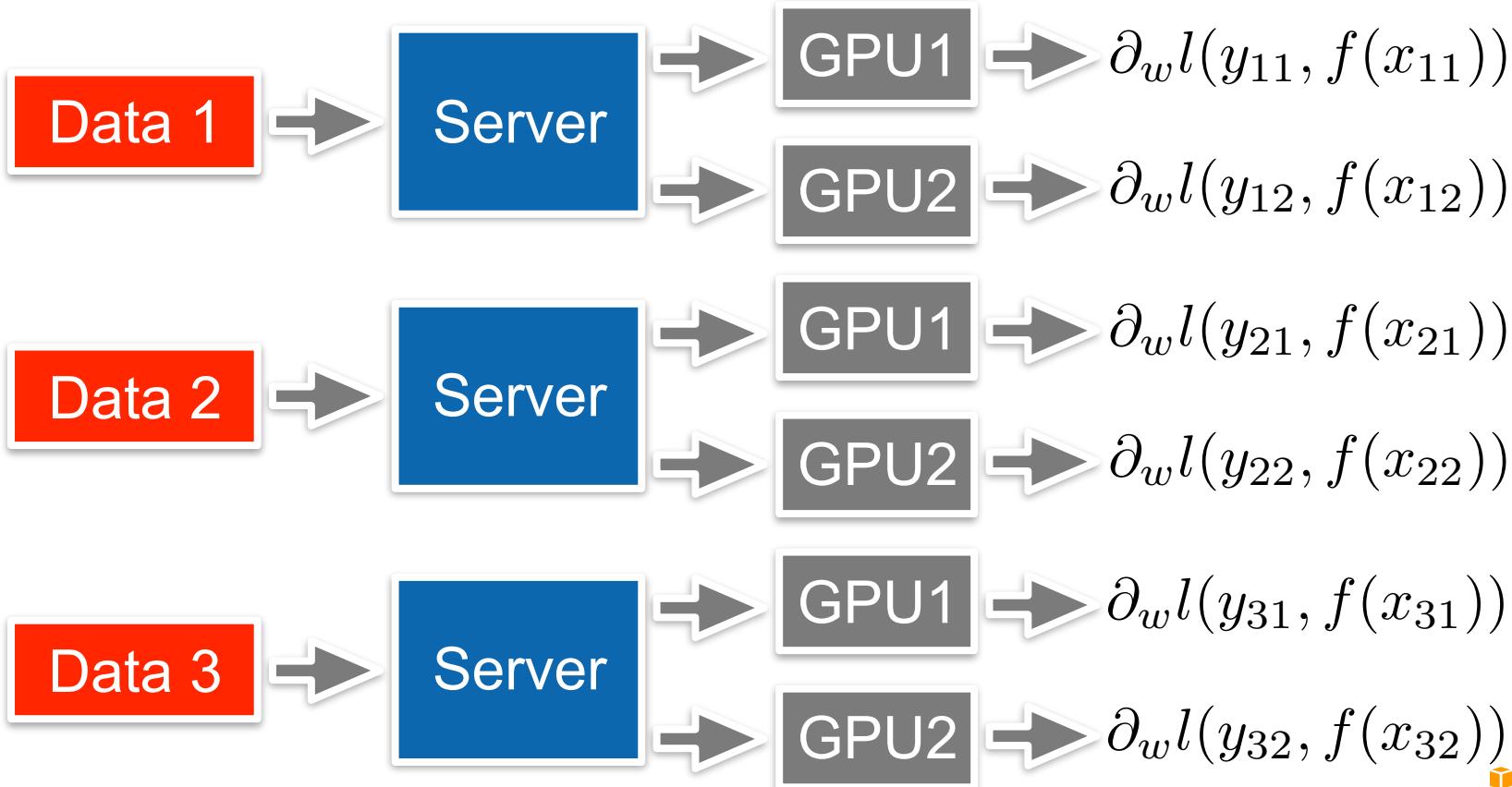
- **Asynchronous - efficient for high time dispersion**



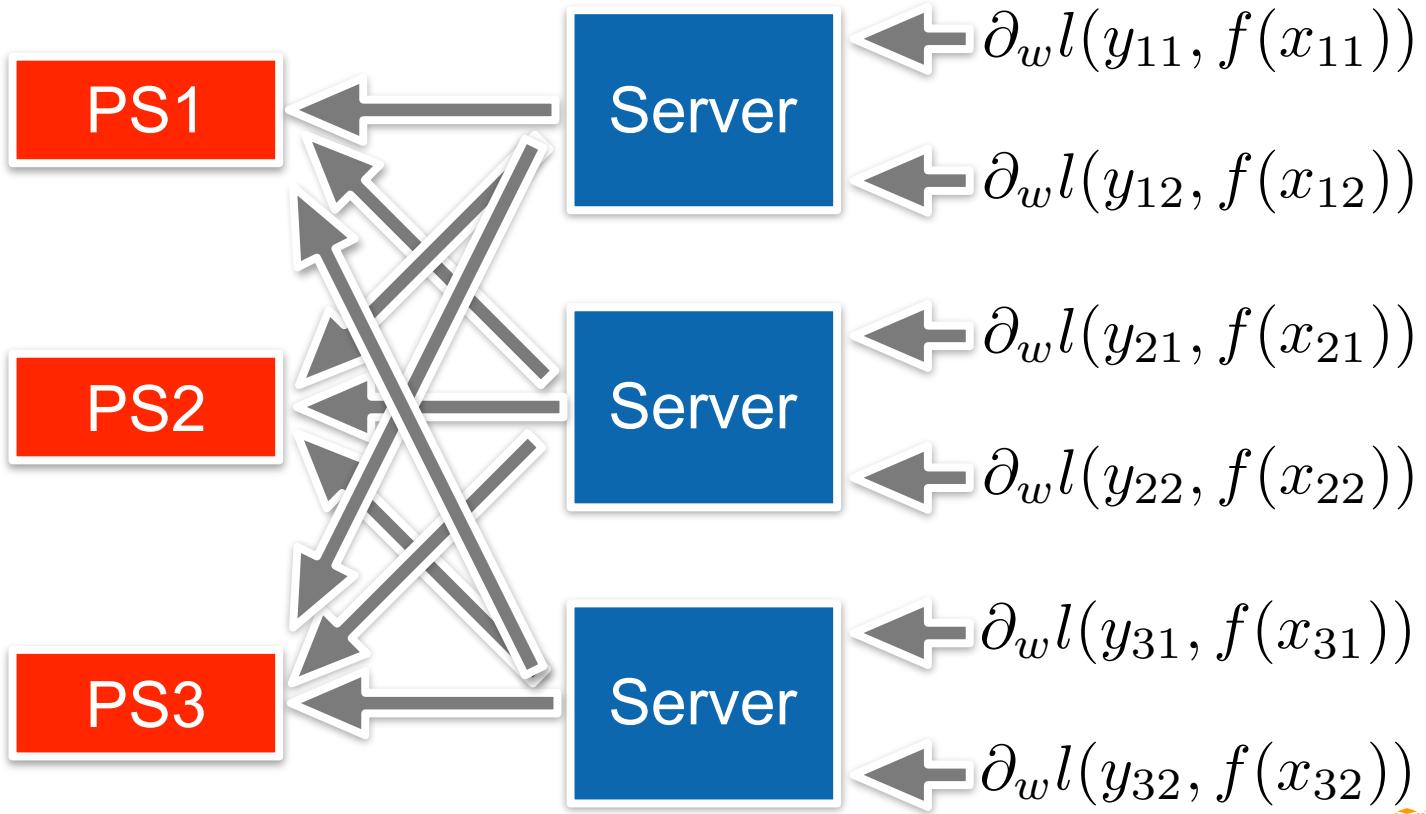
Parallelization - Pragmatic Strategy

- **Data Parallel & Synchronous**
 - Even for 512 GPUs you only need 32 P2.16xlarge (so the speed dispersion is negligible).
 - GPUs have plenty of RAM
(only an inefficient framework needs model parallelization)
- **Algorithm Template** (shard data over machines)
 - Compute gradient on mini batch on each GPU
 - Aggregate gradients into (key,value) store
 - Broadcast them back to GPUs

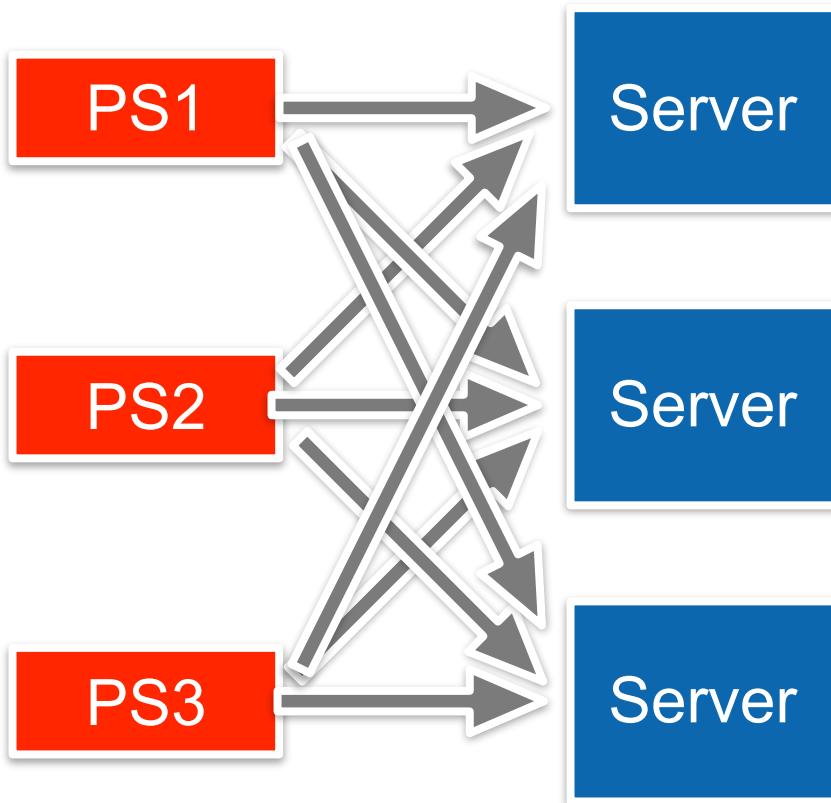
Parallelization - Pragmatic Strategy



Parallelization - Pragmatic Strategy

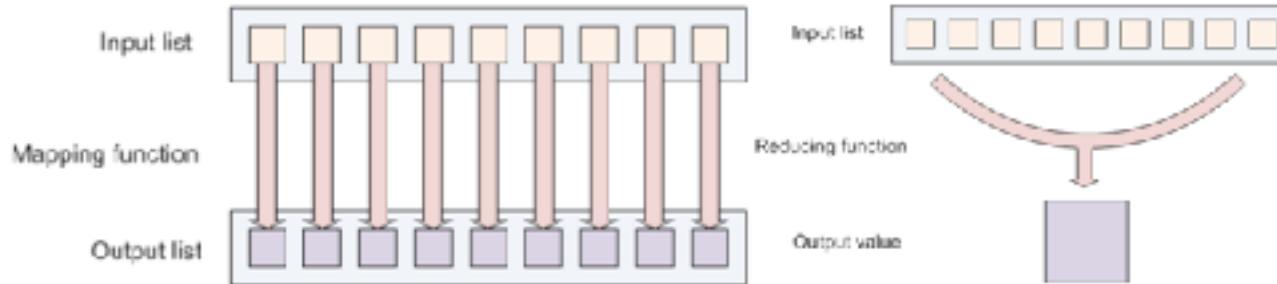


Parallelization - Pragmatic Strategy



Why (not) MapReduce?

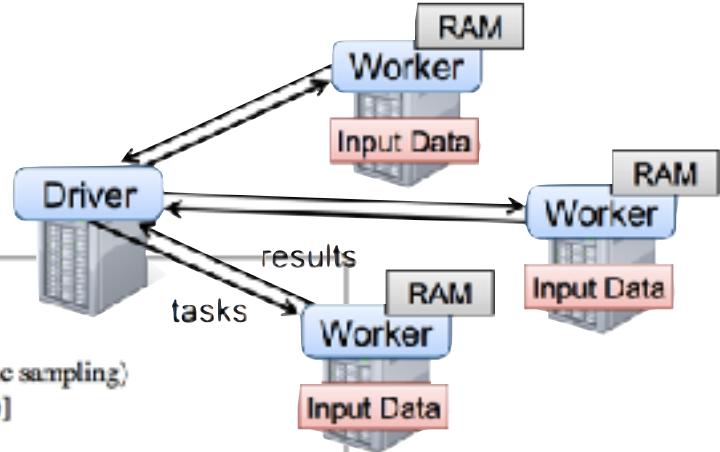
- **Map(key, value)**
Process subset of the data (and emit gradients)
- **Reduce(key, value)**
Aggregate gradients from machines and update parameters
- **Simple parameter exchange**
 - Great if only small number of syncs needed (**slow mixing**)
 - Huge overhead per iteration
 - Fully synchronous (**bad for large number of machines**)



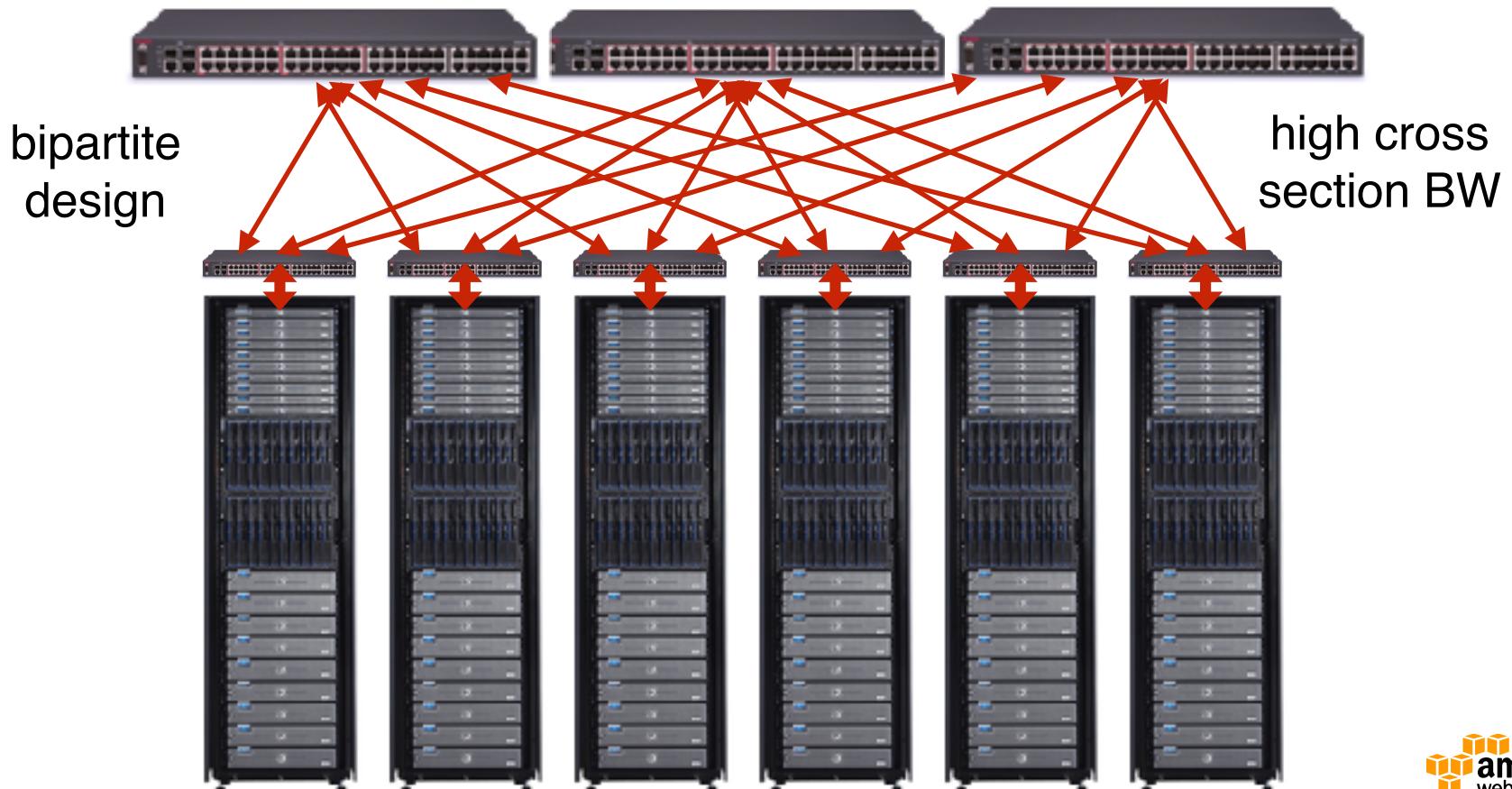
Why (not) Spark (Zaharia et al, 2012)

- Data is transformed by processing
- Store intermediate data using lineage
- Driver controls work

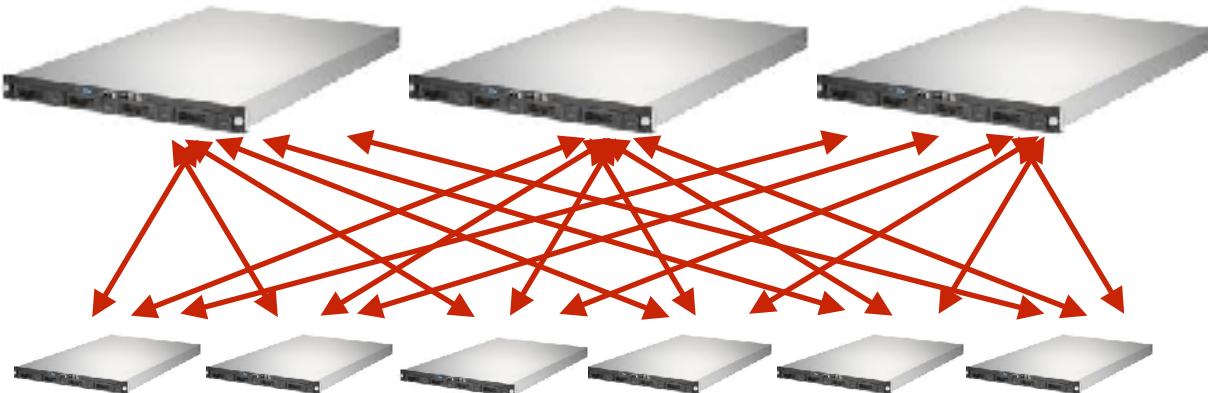
	Transformations	
	$\text{map}(f : T \rightarrow U)$: $\text{RDD}[T] \rightarrow \text{RDD}[U]$ $\text{filter}(f : T \rightarrow \text{Bool})$: $\text{RDD}[T] \rightarrow \text{RDD}[T]$ $\text{flatMap}(f : T \rightarrow \text{Seq}[U])$: $\text{RDD}[T] \rightarrow \text{RDD}[U]$ $\text{sample}(\text{fraction} : \text{Float})$: $\text{RDD}[T] \rightarrow \text{RDD}[T]$ (Deterministic sampling) $\text{groupByKey}()$: $\text{RDD}[(K, V)] \rightarrow \text{RDD}[(K, \text{Seq}[V])]$ $\text{reduceByKey}(f : (V, V) \rightarrow V)$: $\text{RDD}[(K, V)] \rightarrow \text{RDD}[(K, V)]$ $\text{union}()$: $(\text{RDD}[T], \text{RDD}[T]) \rightarrow \text{RDD}[T]$ $\text{join}()$: $(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \rightarrow \text{RDD}[(K, (V, W))]$ $\text{cogroup}()$: $\text{crossProduct}()$: $\text{mapValues}(f : V \Rightarrow W)$: $\text{sort}(c : \text{Comparator}[K])$: $\text{partitionBy}(p : \text{Partitioner}[K])$:	
ACTIONS		<p>great for ETL bad for many low latency updates</p> $\text{count}()$: $\text{collect}()$: $\text{RDD}[T] \rightarrow \text{Seq}[T]$ $\text{reduce}(f : (T, T) \rightarrow T)$: $\text{RDD}[T] \rightarrow T$ $\text{lookup}(k : K)$: $\text{RDD}[(K, V)] \rightarrow \text{Seq}[V]$ (On hash/range partitioned RDDs) $\text{save(path : String)}$: Outputs RDD to a storage system, e.g., HDFS



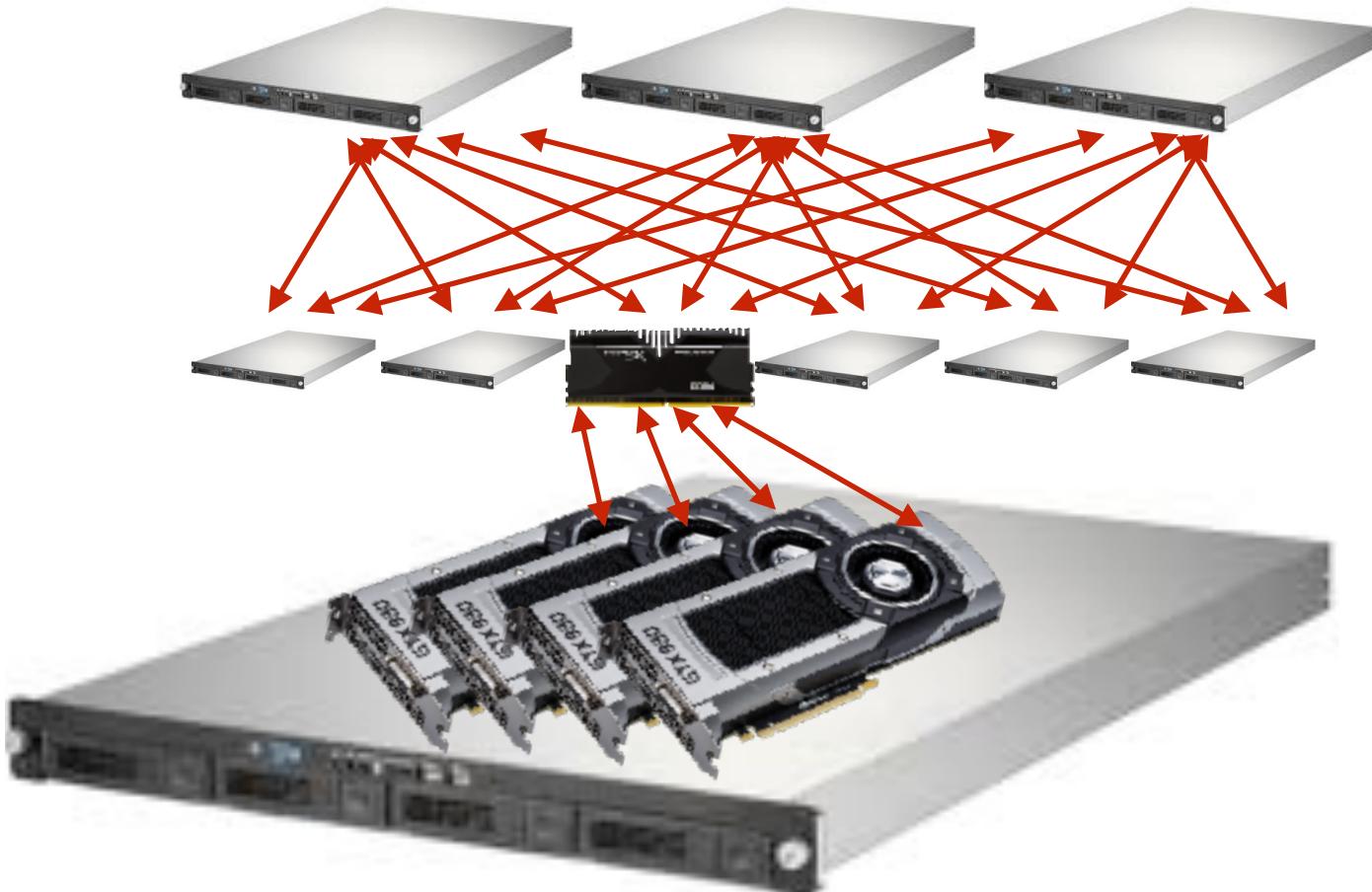
An idea from network design (Clos Networks)



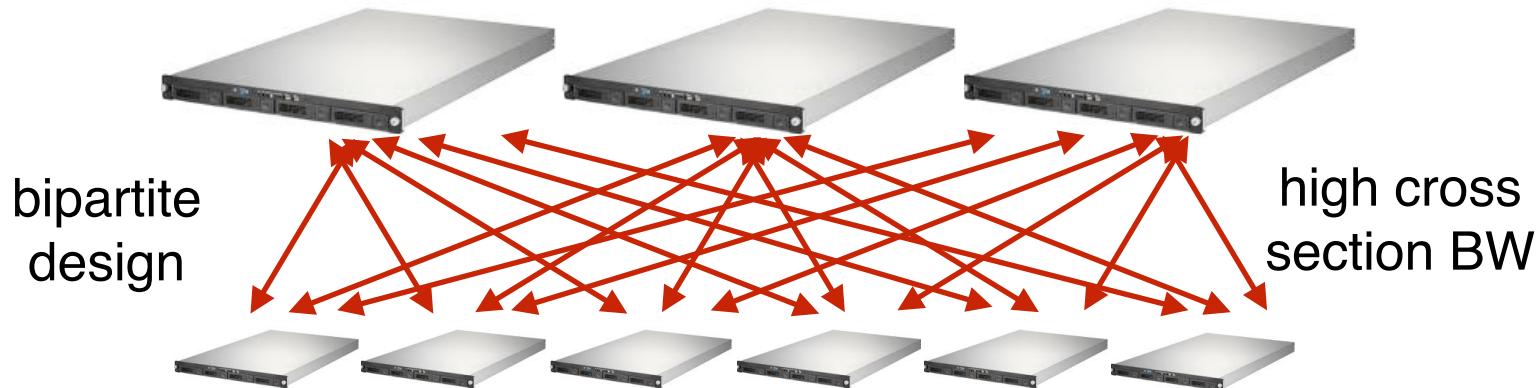
Parameter Server aka (key,value) store



Parameter Server aka (key, value) store



Parameter Server (hierarchical for DL)



Clients have local parameter view

Servers have shard of parameter space

Client-server synchronization

- Reconciliation protocol
- Synchronization schedule
- Load distribution algorithm

Smola & Narayananamurthy, 2010, VLDB

Gonzalez et al., 2012, WSDM

Dean et al, 2012, NIPS

Shervashidze et al., 2013, WWW

Google, Baidu, Facebook,

Amazon, Yahoo, Microsoft

put(keys,values,clock)
get(keys,values,clock)

Applications of the Parameter Server

- Large scale Logistic Regression models
(100B parameters, 100B observations)
- Latent Variable Models
(100B words, 10k topics)
- Deep Learning (e.g. TensorFlow, MxNet)
(sync between GPUs on machine via NCCL, bipartite PS between machines)
- Sketching
(1B/s inserts into 10 machines)

Outline

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Convolutional Neural Networks

2.3. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

Links & Notebooks

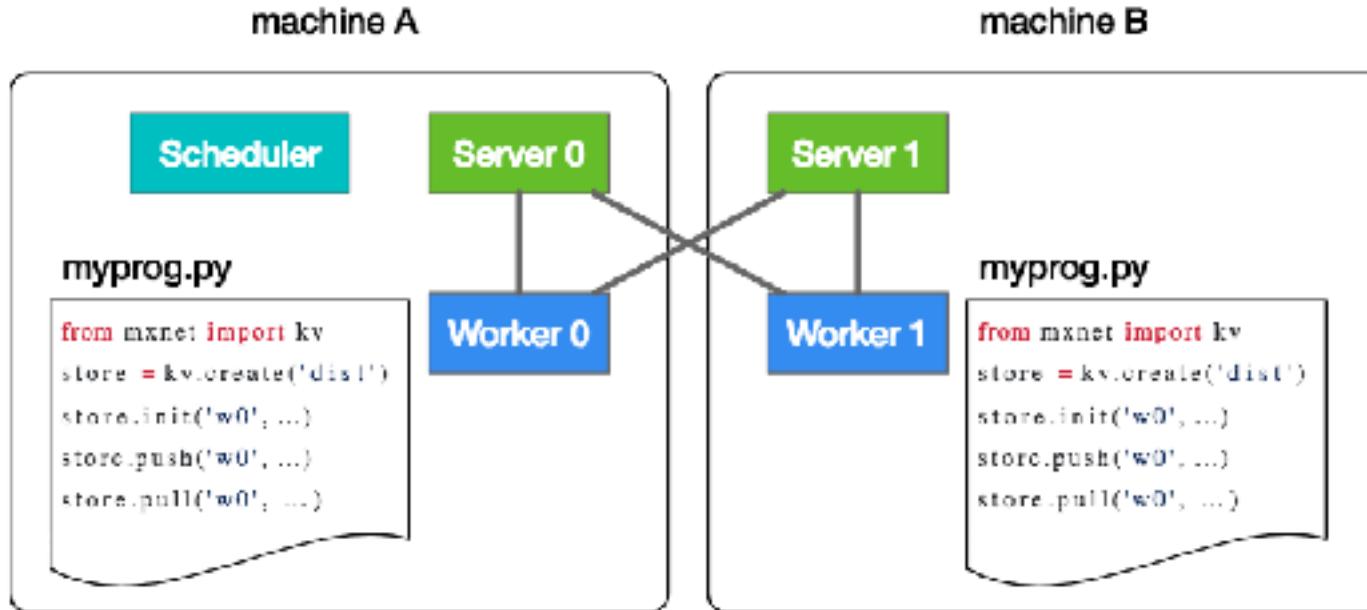
gluon.mxnet.io/P14-C02-multiple-gpus-scratch.html

gluon.mxnet.io/P14-C03-multiple-gpus-gluon.html

mxnet.io/api/python/kvstore.html

Multiple Machines (in practice)

- Each machine has server and worker job
- One machine has scheduler for task distribution



Summary

1. Getting Started

1.1. Installation

1.2. Deep Learning 101

1.3. NDArray

2. Neural Networks in MXNet

2.1. Linear Models & MLPs

2.2. Loss Functions

2.3. Convolutional Neural Networks

2.4. Optimization & Capacity Control

3. Advanced Topics

3.1. Generative Adversarial Networks

3.2. LSTMs

3.3. TreeLSTMs

4. Scaling Learning

4.1. Infrastructure on AWS

4.2. Parallel Optimization Algorithms

4.3. Parallel and Distributed Training

gluon.mxnet.org

