Deep Learning in Apache MxNet Gluon

Imperative, Dynamic and Distributed

gluon.mxnet.io

Zachary Lipton, Alex Smola

AWS Machine Learning

Schedule

8:30-9:30 Part I

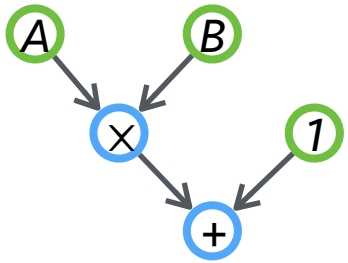
9:30-10:00 Coffee

10:00-12:00 Part 2

12:00-13:30 Lunch

13:30-15:30 Part 3

Declarative 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*

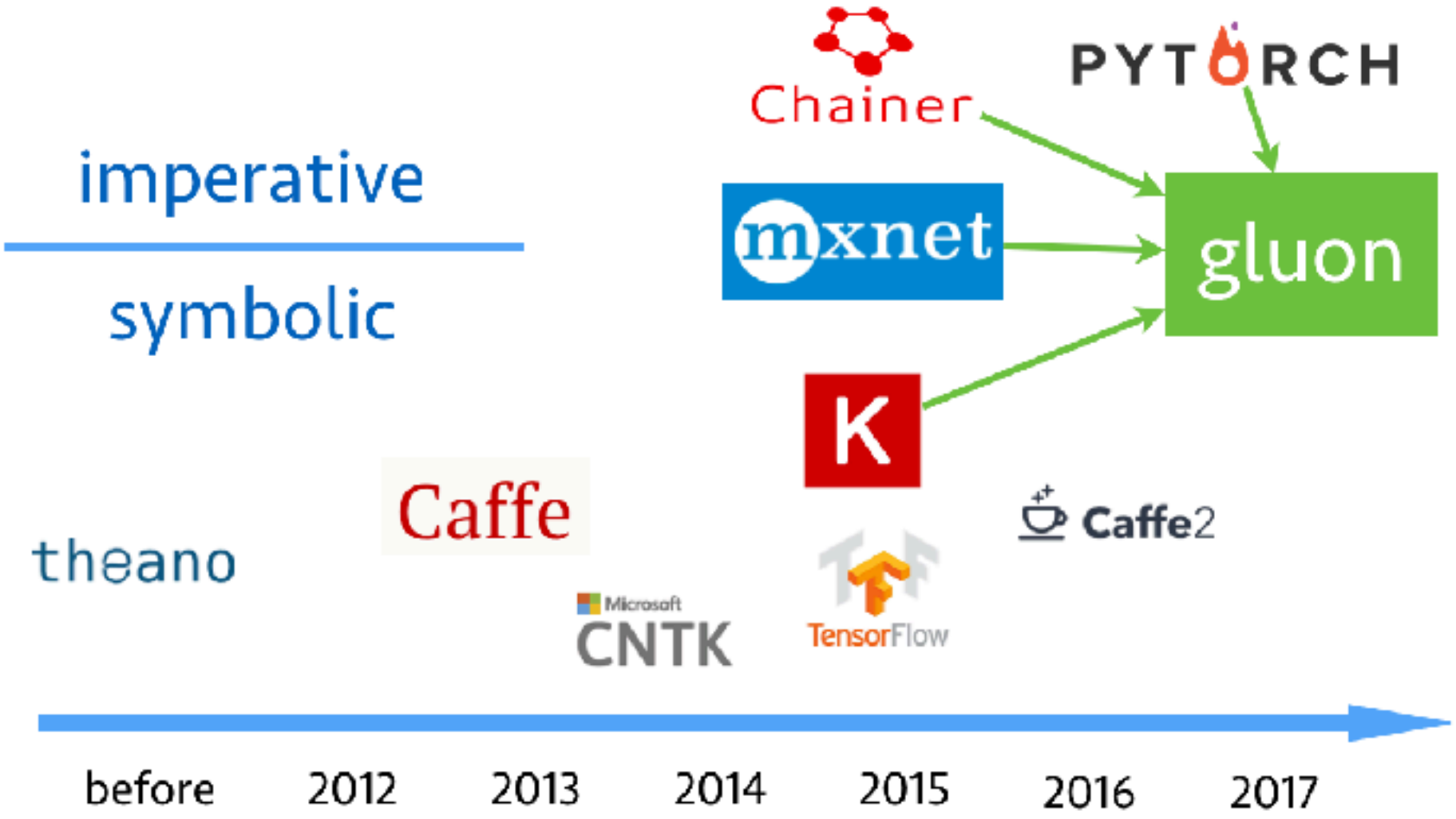
- **Advantages**:
  - Opportunities for optimization
  - Easy to serialize models
  - More portable across languages
- **Disadvantages**:
  - Hard to debug
  - Unsuitable for dynamic graphs
  - Can't use native code

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)**


imperative
symbolic
Chainer
PYTORCH
mxnet
gluon
K
Caffe
Caffe2
theano
Microsoft
CNTK
TensorFlow
before
2012
2013
2014
2015
2016
2017

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. 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

Installation

- **To use gluon, you'll need the latest version of MxNet!**
- **Build from source**

```
git clone git@github.com:apache/incubator-mxnet.git
git submodule update --init
mxnet.io/get_started/install.html
```

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

```
pip install --upgrade pip
pip install --upgrade setuptools
pip install mxnet_cuda80 --pre
# pip install mxnet --pre
```

without GPUs

amazon web services

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 web services

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
```

amazon web services

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. Loss Functions
- 2.3. Convolutional Neural Networks
- 2.4. Optimization & Capacity Control

3. Advanced Topics

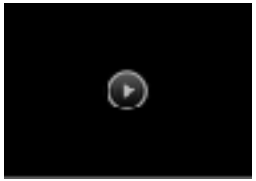
- 3.1. Generative Adversarial Networks
- 3.2. LSTMs
- 3.3.TreeLSTMs

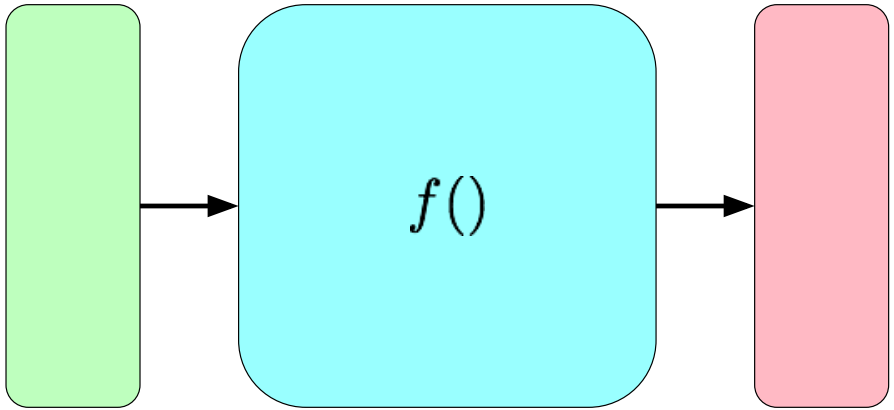
4. Scaling Learning

- 4.1. Infrastructure on AWS
- 4.2. Parallel Optimization Algorithms
- 4.3. Parallel and Distributed Training

Supervised Learning

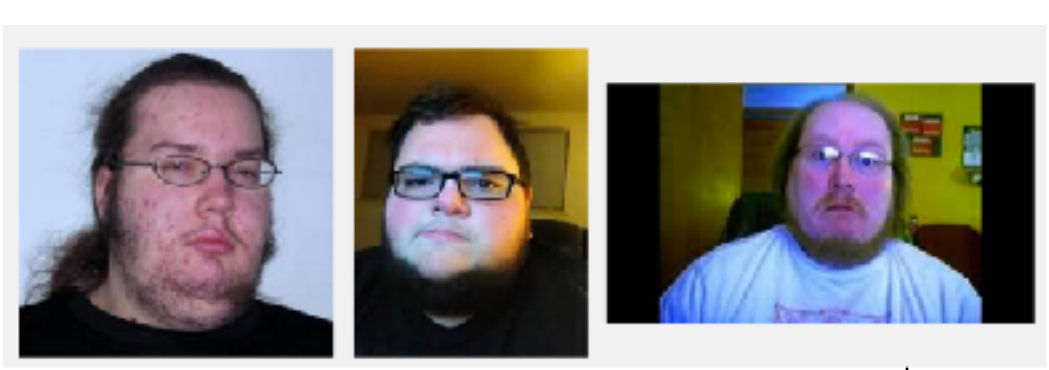
“Time underlies many interesting human behaviors”

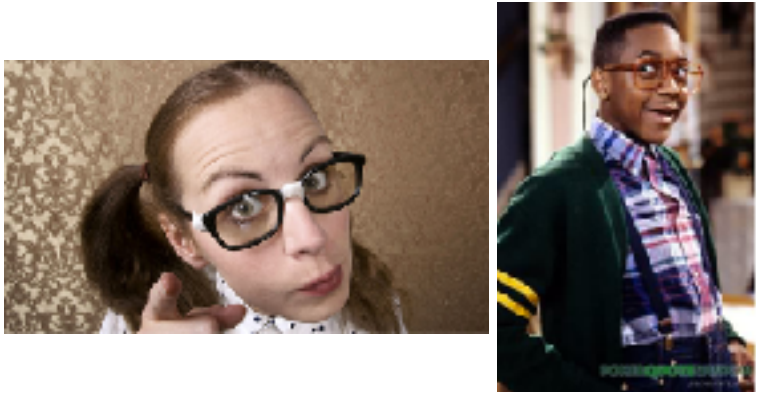


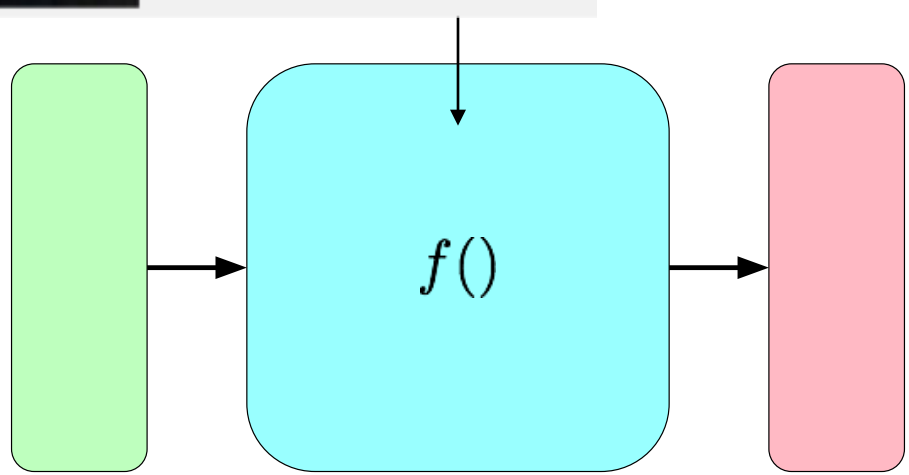


{0,1}
{A,B,C…}
captions,
email mom,
fire nukes,
eject pop tart

Programming with Programmers







Limits of Conventional Programming

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

cat

cat

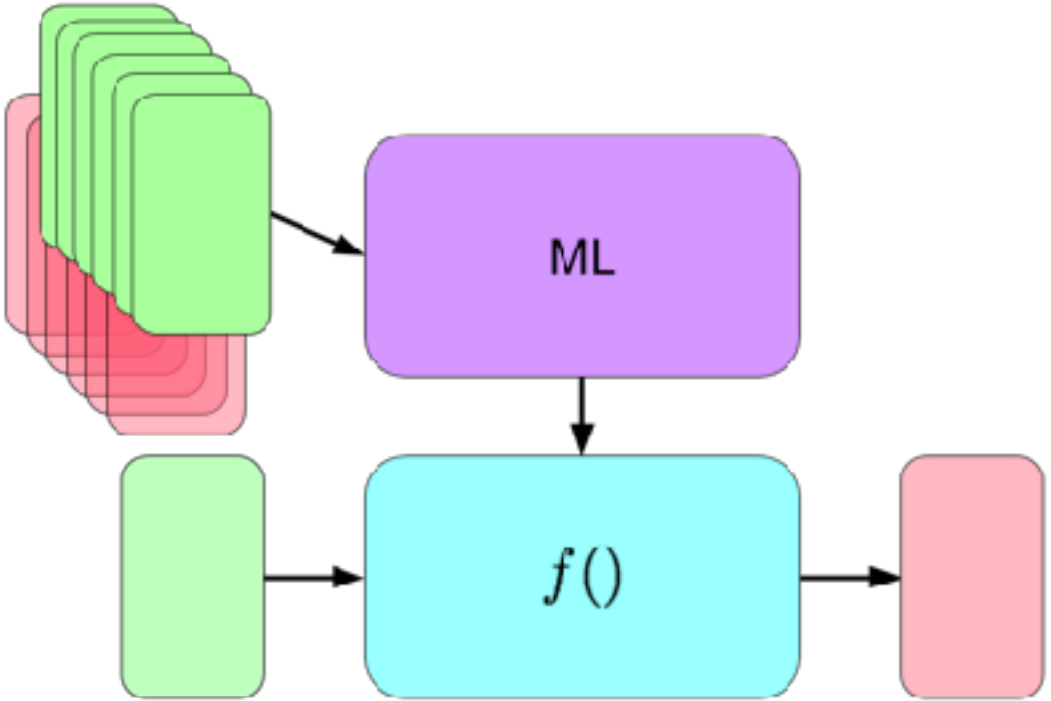
dog

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 -> (x%3, x%5, x%15)
- **Train a classifier**
  mapping input x to output y
  using training data

Code

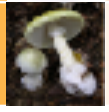
```
var o='';
for (var i=1; i<=100; i++) {
    i%3 || (o+='Fizz ');
    i%5 || (o+='Buzz ');
    !(i%3 && i%5) || (o+=(i+' '));
}
console.log(o);
```

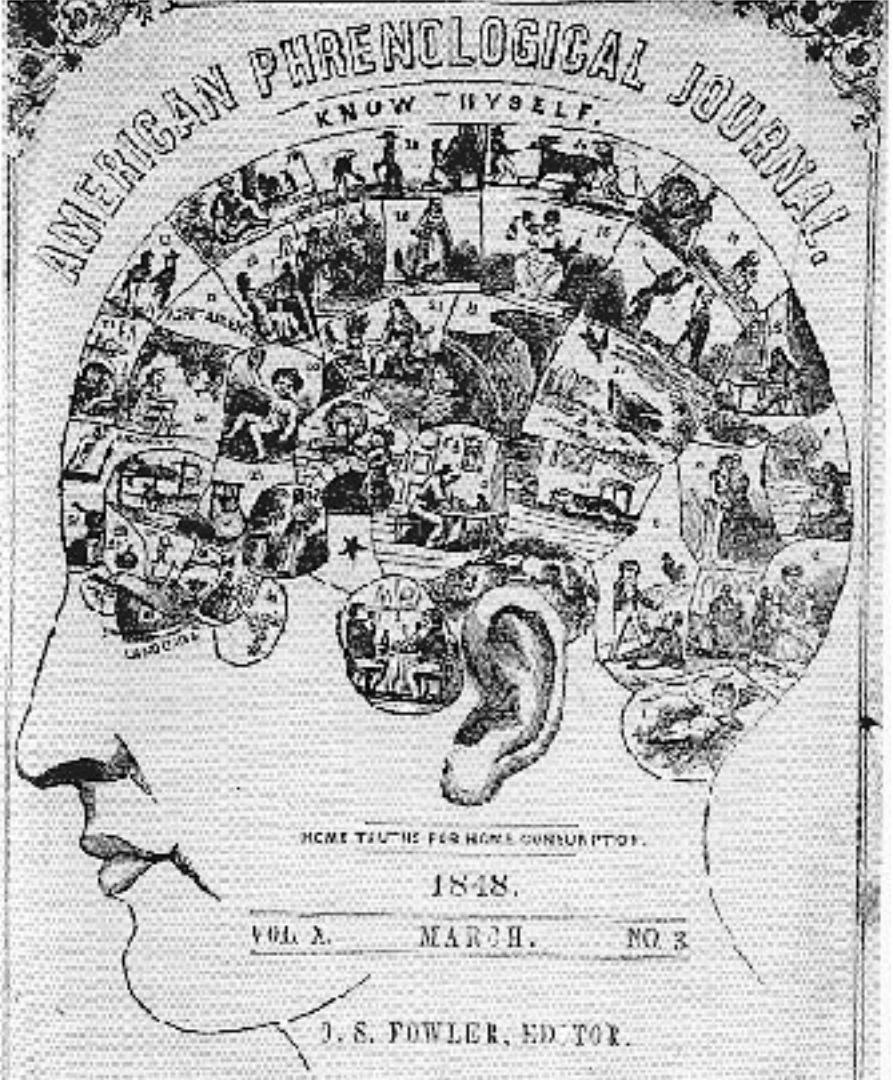
That was silly.
Why would you do this?

Inspired by joelgrus.com

Programming with Data

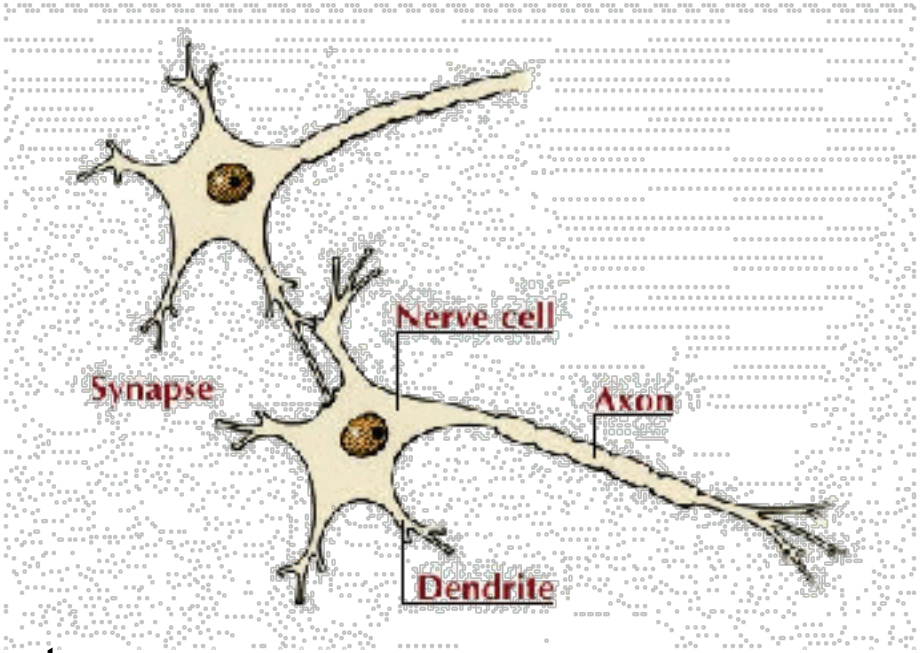
- **Data** (patterns, labels)
  - Images (pictures / cat, dog, dinosaur, chicken, human)
  - Text (email / spam, ham)
  - Sound (recording / recognized speech)
  - Video (sequence / break-in, OK)
- **Goal** Learn mapping data to outputs
- **Loss function**
  measure how well we're doing

		Edible	Poison
Truffle		0	10
Death cap		1,000,000	0



Neurons and Learning

Neurons

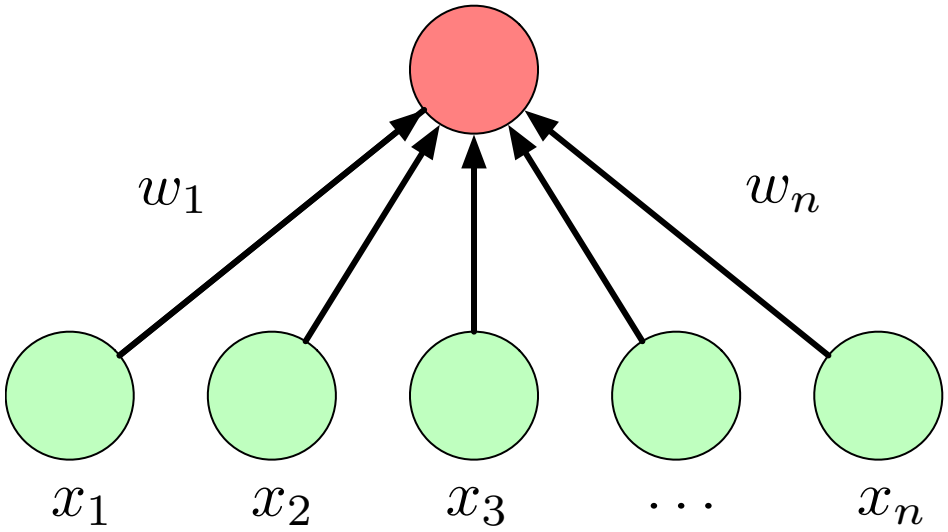


- **Soma** (CPU)
  Cell body - combines signals
- **Dendrite** (input bus)
  Combines the inputs from several other nerve cells
- **Synapse** (interface)
  Interface and parameter store between neurons
- **Axon** (cable)
  May be up to 1m long and will transport the activation signal to neurons at different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

Neurons

synaptic weights

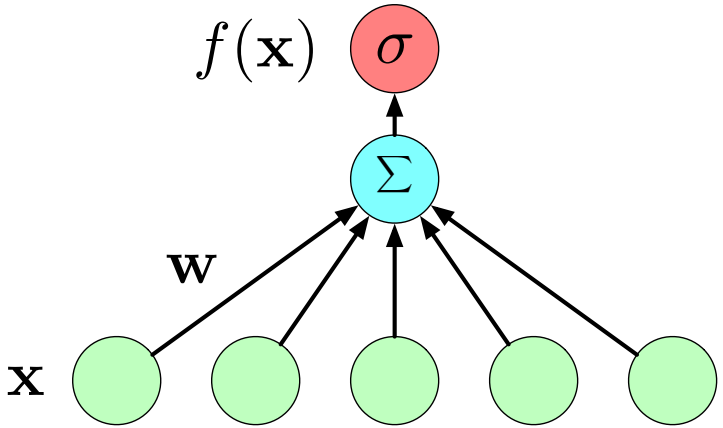
inputs

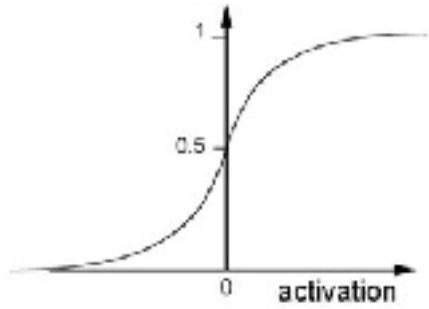


$$f(x) = \sum_i w_i x_i = \langle w, x \rangle$$

Neural Networks

- **Input**
  Data vector $\mathbf{x}$
- **Output**
  Linear function of inputs
- **Nonlinearity**
  Transform output into desired range of values, e.g. probabilities [0, 1]
- **Training**
  Learn the weights $\mathbf{w}$ and bias b





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

Perceptron

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

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


The Perceptron

The Perceptron

initialize $w = 0$ and $b = 0$
repeat
  if $y_i \left[\langle w, x_i \rangle + b\right] \leq 0$ **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

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

  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

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

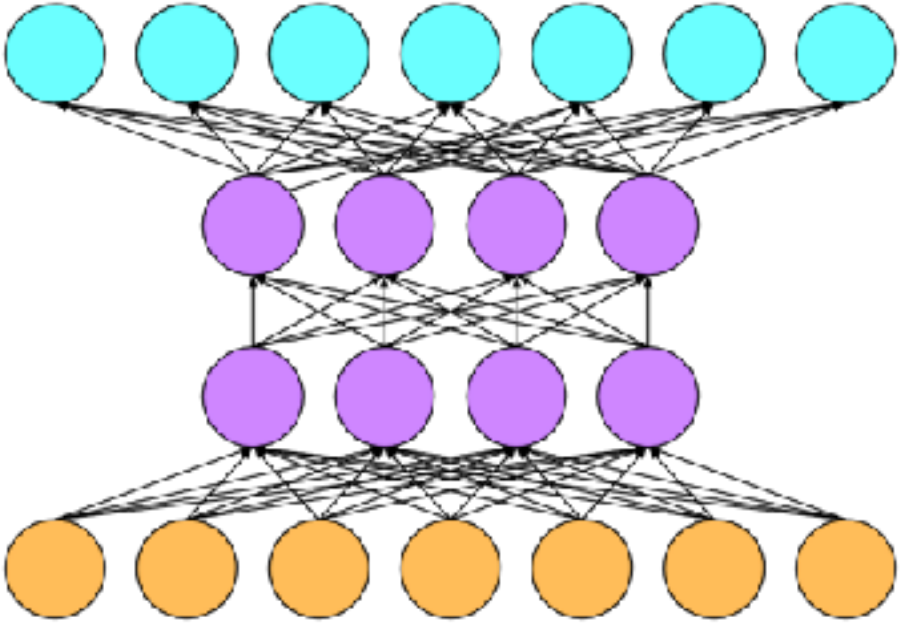
- Fails miserably with noisy data

do NOT train your avatar with perceptrons

Links & Notebooks

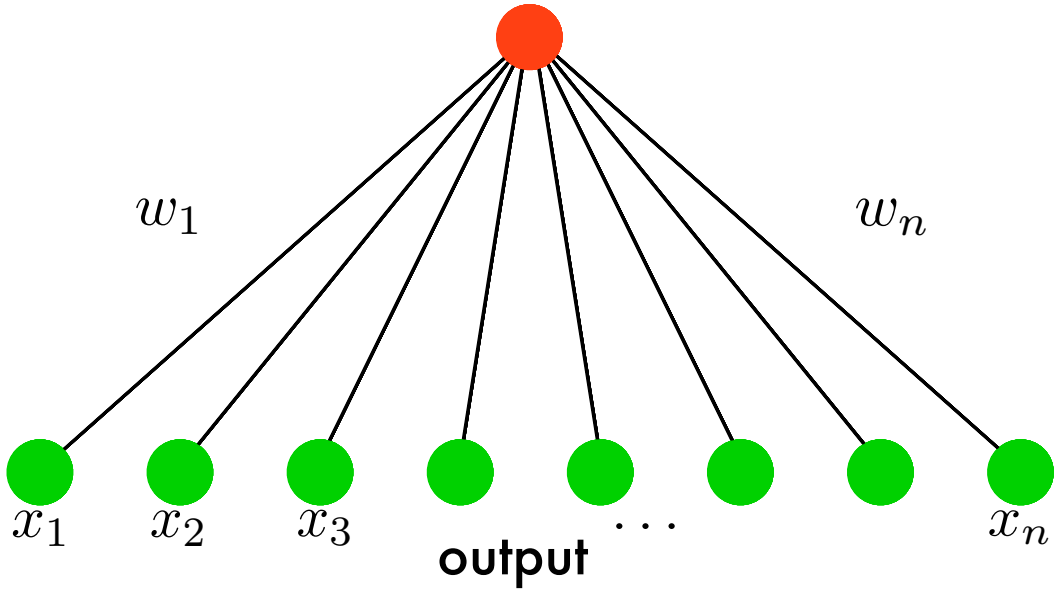
gluon.mxnet.io/P02-C02.5-perceptron.html

Multilayer Perceptron and Backpropagation



Perceptron

weights

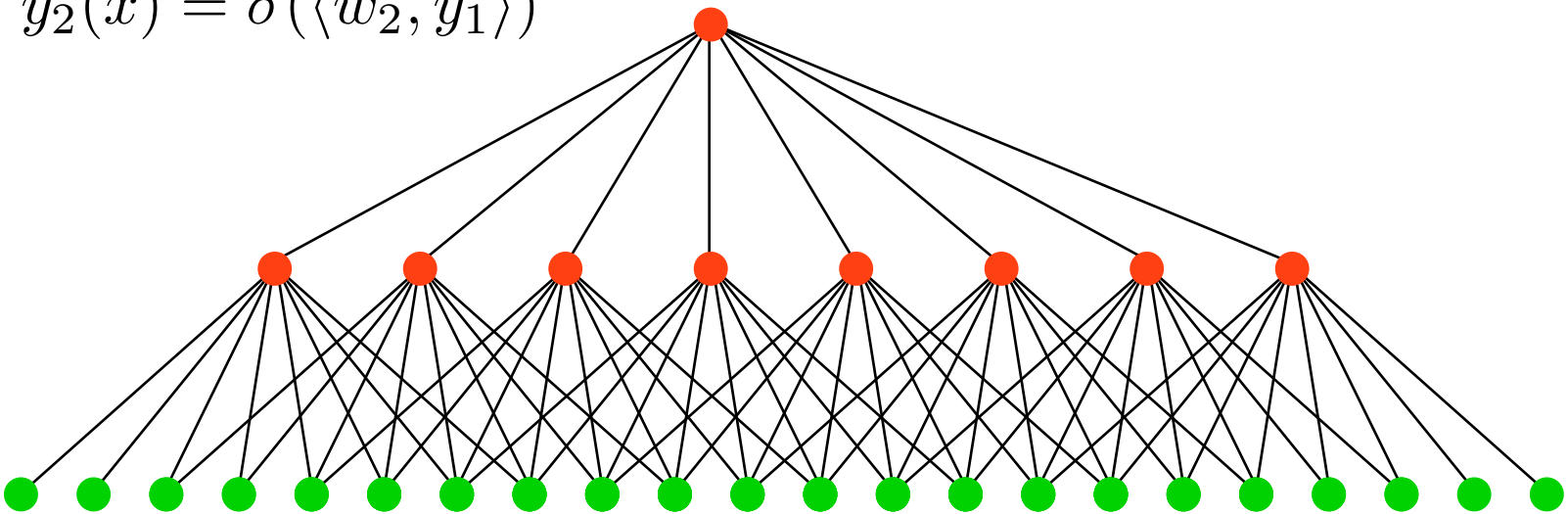


$$y(x) = \sigma(\langle w, x \rangle)$$

Multilayer Perceptron

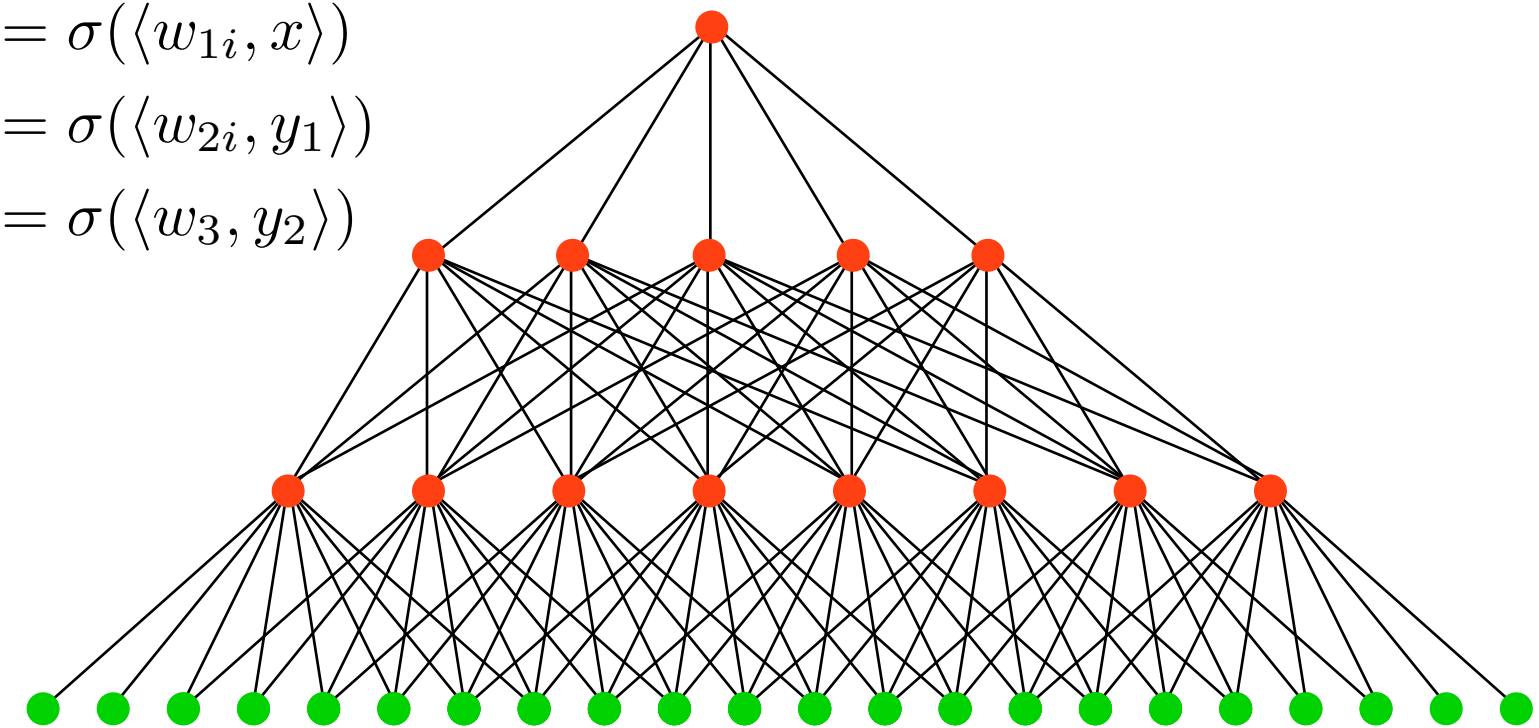
$$y_{1i}(x) = \sigma(\langle w_{1i}, x \rangle)$$

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



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**

$$y_i = W_i x_i + b_i \qquad \text{fully connected}$$

$$y_i = \sigma(x_i) \qquad \text{nonlinearity}$$

- **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**

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

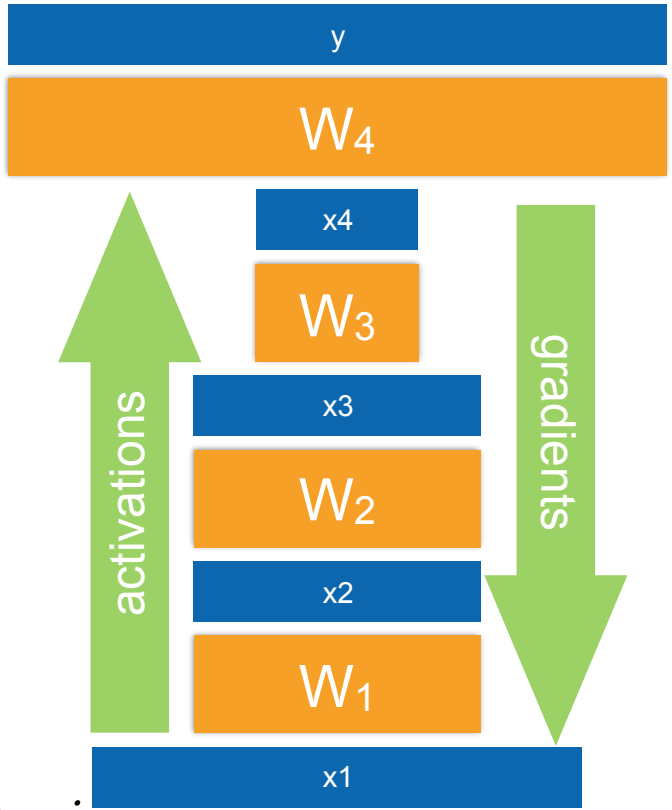
- **Use the chain rule**

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

Backpropagation

- **Layer Representation**
  (each layer performs a transformation)
  $x_{i+1} = f_i(x_i, w_i)$
- **Gradient**

$$\partial_{w_1} x_2(x) = \partial_{w_1} f_1(x_1, w_1)$$
$$\partial_{w_1} x_3(x) = \partial_{x_2} f_2(x_2, w_2) \partial_{w_1} x_2(x)$$
$$\dots$$
$$\partial_{w_i} x_{i+1}(x) = \partial_{w_i} f_i(x_i, w_i)$$
$$\partial_{w_i} x_{j+1}(x) = \partial_{x_j} f_j(x_j, w_j) \partial_{w_i} x_j(x) \text{ if } i > j$$



Backpropagation

- **Layer Representation**
  (each layer performs a transformation)
  $x_{i+1} = f_i(x_i, w_i)$
- **Gradient**

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

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

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

still to compute

incoming gradient

multiply with this

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

net = gluon.nn.Sequential()
with net.name_scope():
    net.add(gluon.Dense(128, activation='relu'))
    net.add(gluon.Dropout(rate=0.5))
    net.add(gluon.Dense(64, activation='relu'))
    net.add(gluon.Dropout(rate=0.5))
    net.add(gluon.Dense(1))

loss = gluon.loss.SoftmaxCrossEntropyLoss()
```

nonlinearity

regularization

second layer

amazon web services

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. 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

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

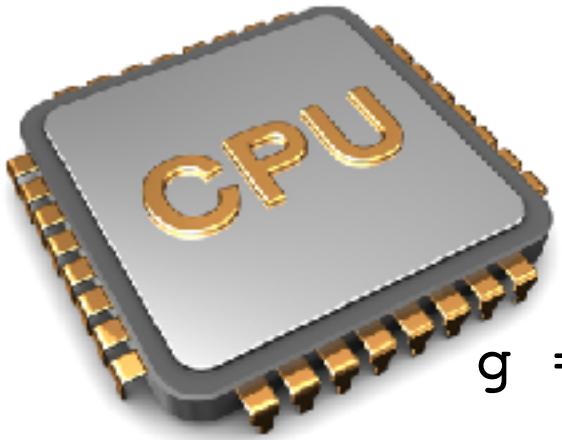
NDArray (Linear Algebra on GPUs in Python)

- **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

Automatic Differentiation

```
b = a * 2
while (nd.norm(b) < 1000).asscalar():
    b = b * 2
if (mx.nd.sum(b) > 0).asscalar():
    c = b
else :
    c = 100 * b
```

AutoGrad computes gradients through arbitrary control flow

Links & Notebooks

http://gluon.mxnet.io/P01-C02-ndarray.ipynb
http://gluon.mxnet.io/P01-C03-linear-algebra.html
http://gluon.mxnet.io/P01-C05-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. 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

Linear Regression

- Model $$f(x) = \langle w, x \rangle + b$$
- Loss $$l(y, f(x)) = \frac{1}{2}(y - f(x))^2$$

- Optimization algorithm - SGD
  - Initialize w, b at random (or at zero)
  - Iterate over data

$$(w, b) \leftarrow (w, b) - \eta \partial_{(w,b)} l(y, f(x))$$

Logistic Regression

- Model $$f(x) = \langle w, x \rangle + b$$
- Loss $$p(y|f(x)) \propto \exp\left(\frac{1}{2} y f(x)\right) \text{ and hence } p(y|f(x)) = \frac{1}{1+\exp(-yf(x))}$$
$$l(y, f(x)) = -\log p(y|f(x)) = \log(1+\exp(-yf(x)))$$

- Stochastic Gradient Descent (SGD)
  - 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 (Multiclass)

- Model (single layer)

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

- Multilayer Perceptron

$$x_1 = x_{\text{input}} \text{ and } x_{l+1} = \sigma\left(W_l x_l + b_l\right)$$

- Loss (exponential family)

$$-\log p(y|x) = \log \sum_{y'} \exp\left(f(x)_{y'}\right) - f(x)_y$$

Links & Notebooks

http://gluon.mxnet.io/P02-C01-linear-regression-scratch.html
http://gluon.mxnet.io/P02-C02-linear-regression-gluon.html
http://gluon.mxnet.io/P02-C04-softmax-regression-gluon.html
http://gluon.mxnet.io/P03-C02-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. 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

Least Mean Squares

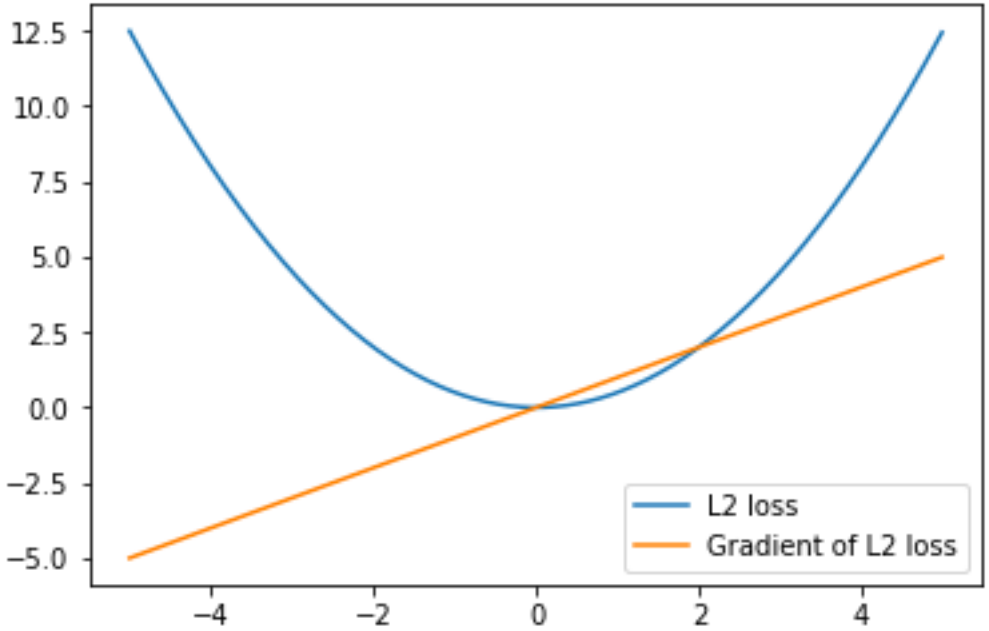
- **Loss**

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

- **Gradient**

$$\partial_f l(y, f(x)) = f(x) - y$$

(loss is differentiable)



L1 Loss

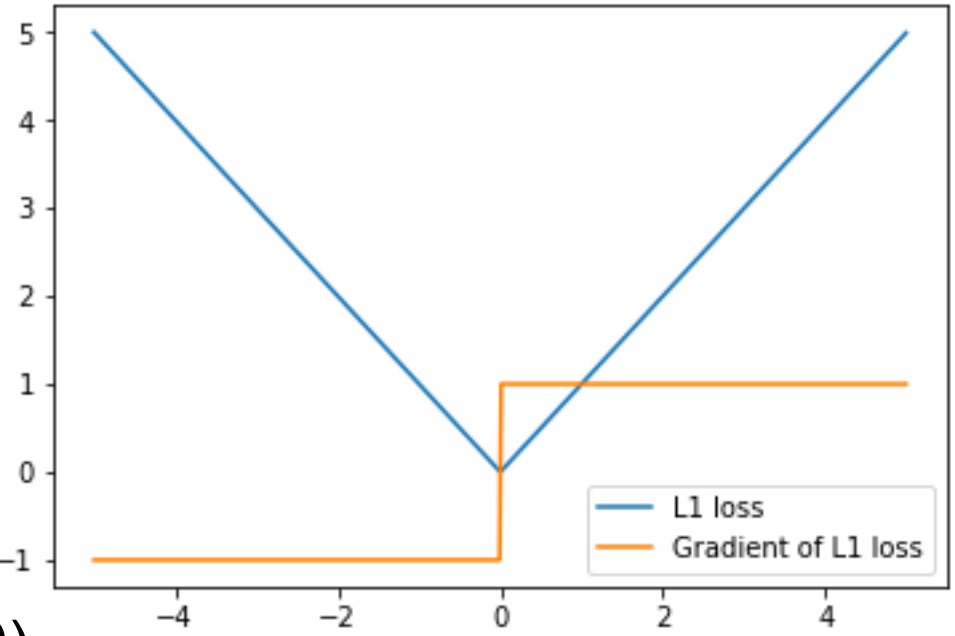
- **Loss**

$$l(y, f(x)) = |f(x) - y|$$

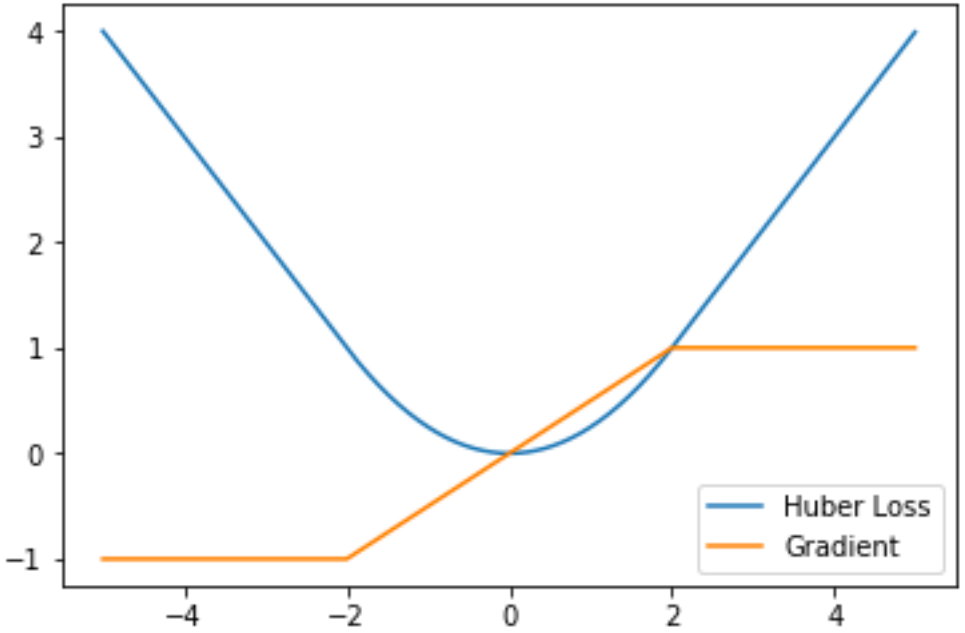
- **Gradient**

$$\partial_f l(y, f(x)) = \text{sgn}\,[f(x) - y]$$

(loss is not differentiable at 0)



Huber's Robust Loss



$$l(y, f) = \begin{cases} \frac{1}{2\rho}(y - f)^2 & \text{for } |y - f| < \rho \\ |y - f| - \frac{\rho}{2} & \text{otherwise} \end{cases}$$

$$\partial_f l(y, f) = \begin{cases} \frac{1}{\rho}(f - y) & \text{for } |y - f| < \rho \\ \text{sgn}(y - f) & \text{otherwise} \end{cases}$$

Logistic Loss

- **Loss**

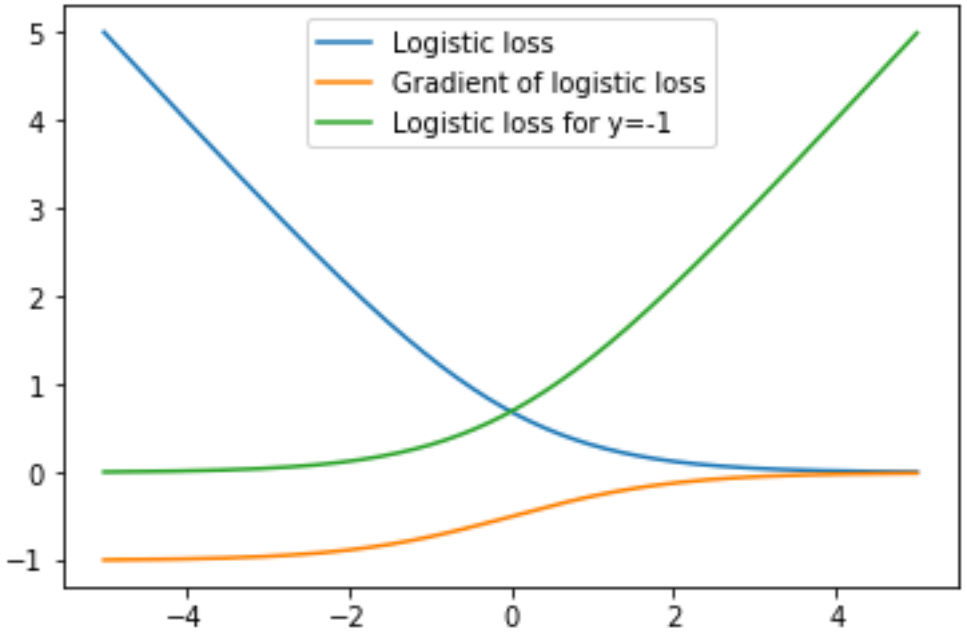
$$p(y|f) = \frac{1}{1+\exp(-yf)}$$

$$l(y,f) = -\log p(y|f)$$

$$= \log(1+\exp(-yf))$$

- **Gradient**

$$\partial_f l(y,f) = \frac{-y}{1+\exp(yf)}$$



Multinomial Logistic Loss

- **Many classes** (assume exponential dependency)

$$p(y|f) \propto \exp(f_y)$$

$$p(y|f) = \frac{\exp(f_y)}{\sum_{y'} \exp(f_{y'})}$$

$$l(y, f) = -\log p(y|f) = \log \sum_{y'} \exp(f_{y'}) - f_y$$

- **Gradient**

$$\partial_f l(y, f) = \frac{\exp(f)}{\sum_{y'} \exp(f_{y'})} - e_y$$

Links & Notebooks

gluon.mxnet.io/P02-C02.6-loss.html

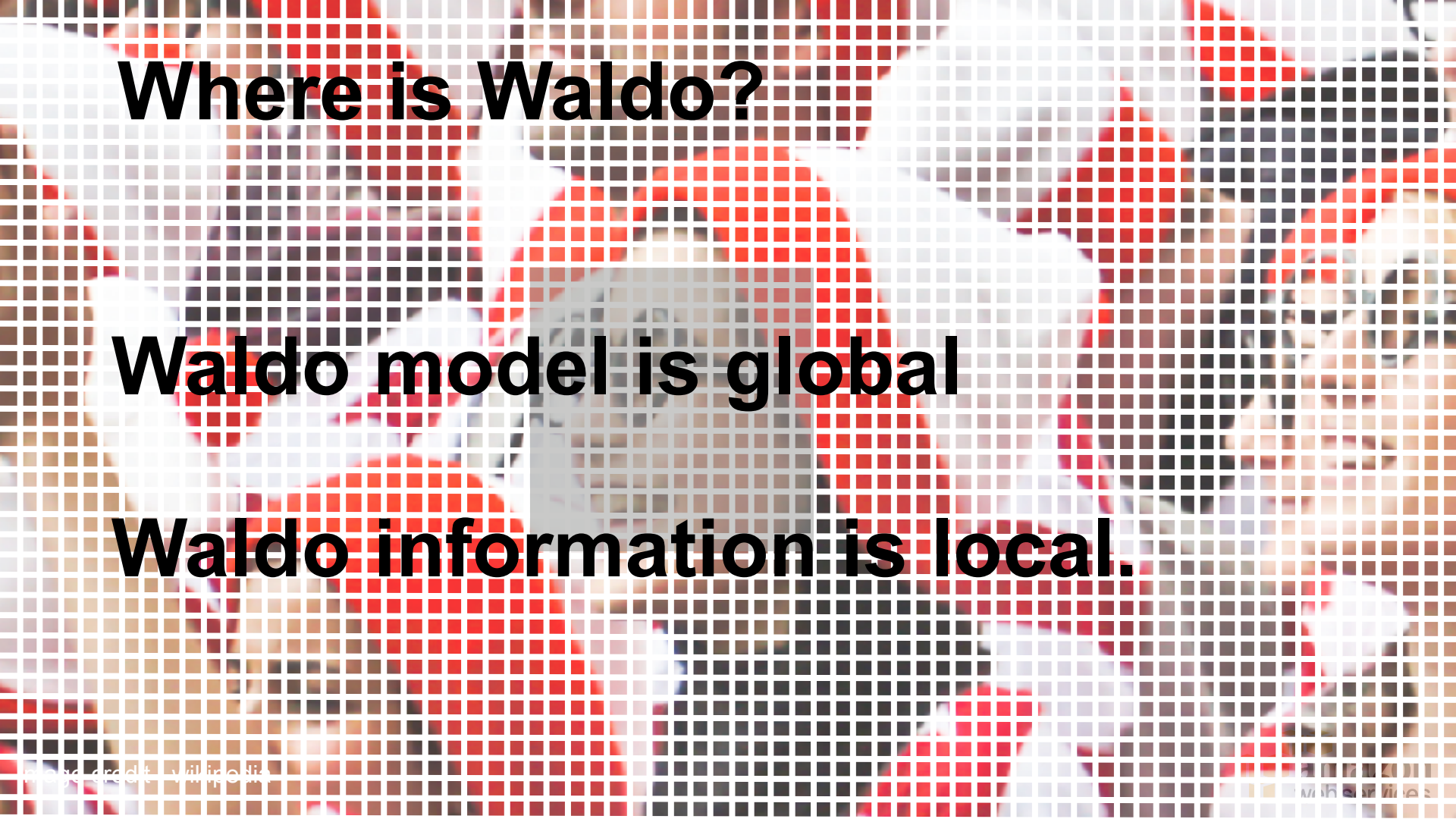
Outline

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. 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

Where is Waldo?

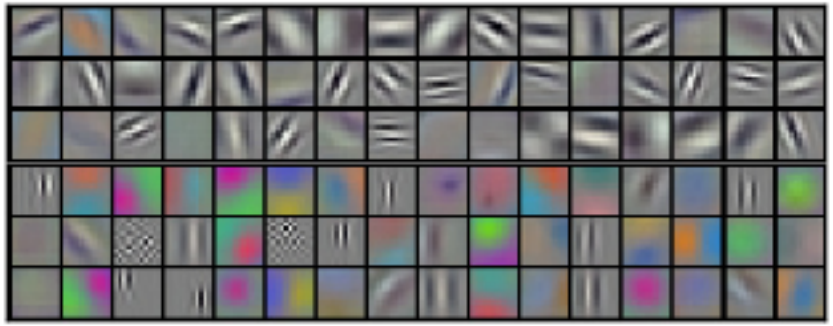
image credit - wikipedia

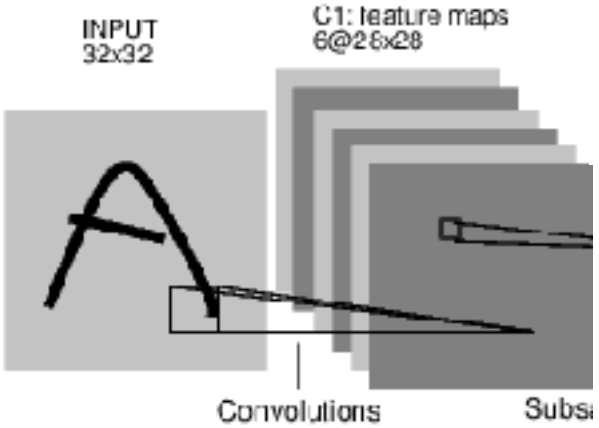
amazon web services


Where is Waldo?
Waldo model is global
Waldo information is local.

Convolutional Layers

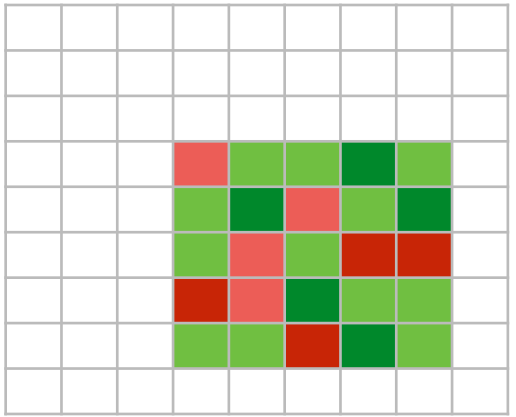
- Images have translation invariance (to some extent)
- Low level is mostly edge and feature detectors
- Usually via convolution (plus nonlinearity)





Convolutional Layers

- **Feature Locality**
  Relevant information only in neighborhood of pixel



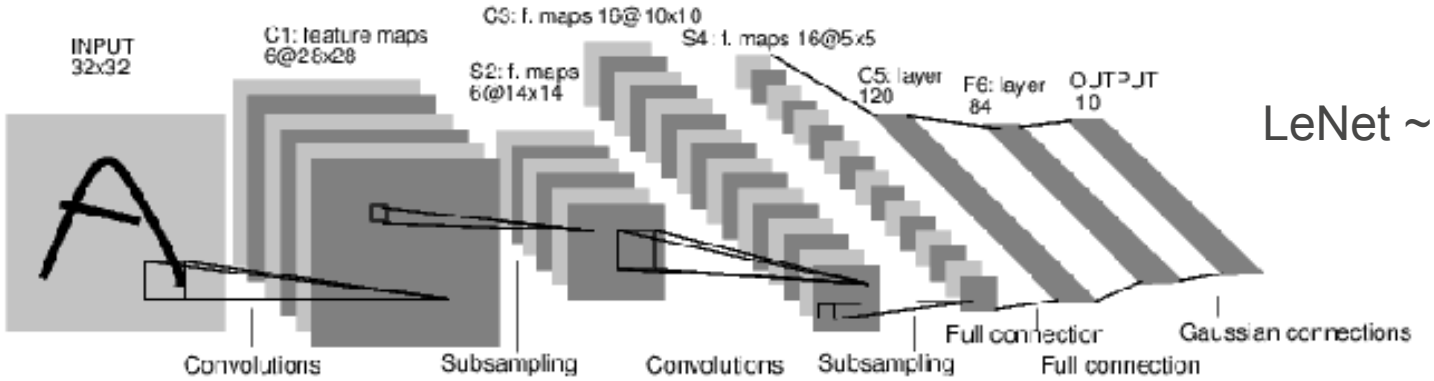
$$y_{ij} = \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} W_{ij,ab} x_{i+a,j+b}$$

- **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



LeNet ~1995

- 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

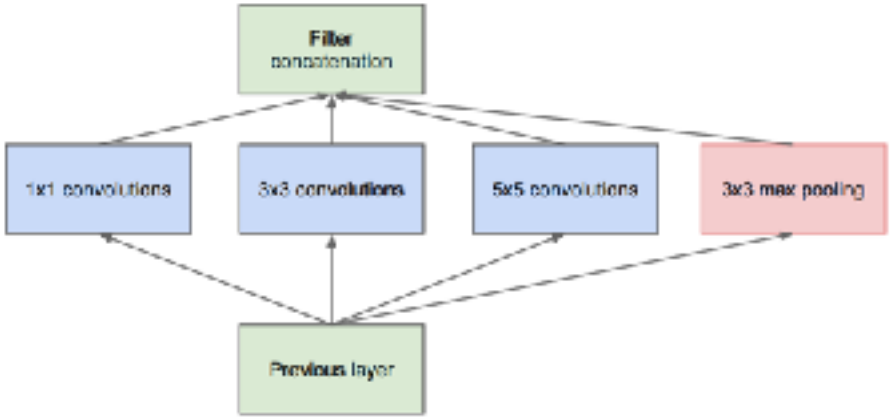
- **The usual suspects**
  - gluon.nn.Dense(units=50, …)
  - gluon.nn.Activation(activation='relu')
  - gluon.nn.Dropout(rate=0.3)
  - gluon.nn.BatchNorm(…)
  - gluon.nn.Embedding(…) for text
- **Convolutions**
  - gluon.nn.Conv1D, 2D, 3D
  - gluon.nn.Conv1DTranspose, 2D, 3D for Deconvolution
- **Pooling Layers**
  - gluon.nn.MaxPool1D, 2D, 3D, gluon.nn.AvgPool1D, 2D, 3D

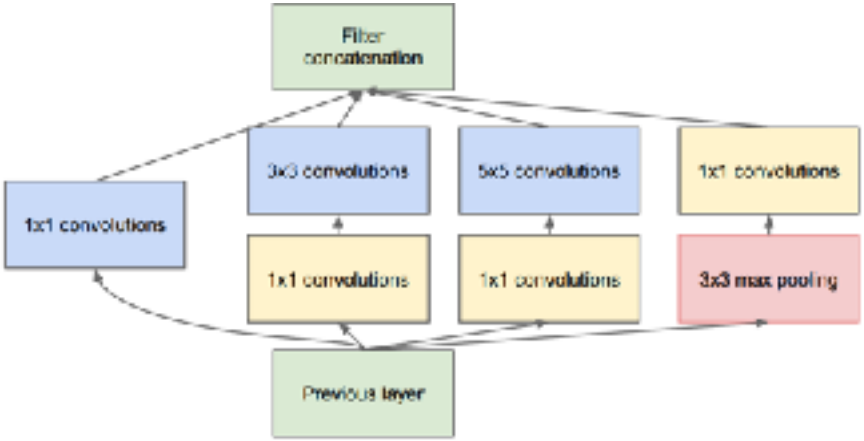
Links & Notebooks

http://gluon.mxnet.io/P04-C02-cnn-gluon.html
http://gluon.mxnet.io/P14-C05-hybridize.html

Fancy structures

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

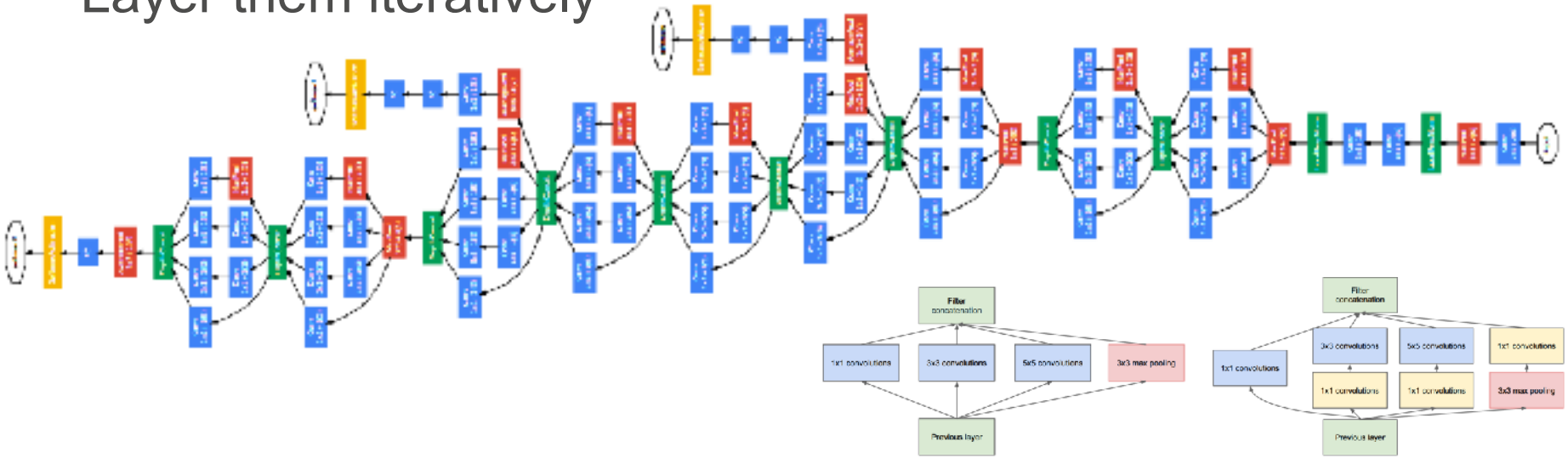




Szegedy et al. arxiv.org/pdf/1409.4842v1.pdf

Fancy structures

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

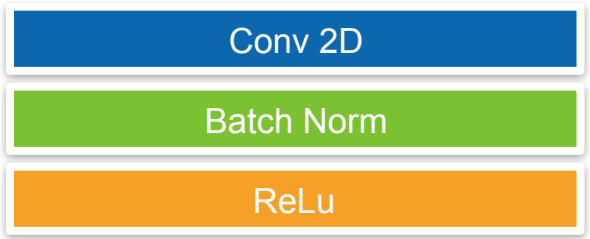


Szegedy et al. arxiv.org/pdf/1409.4842v1.pdf

Networks of networks - Sequential Composition

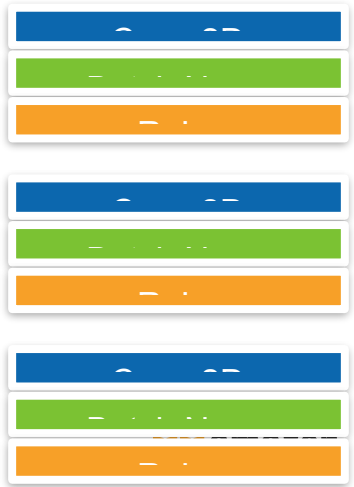
- **Define Single Block**

```
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

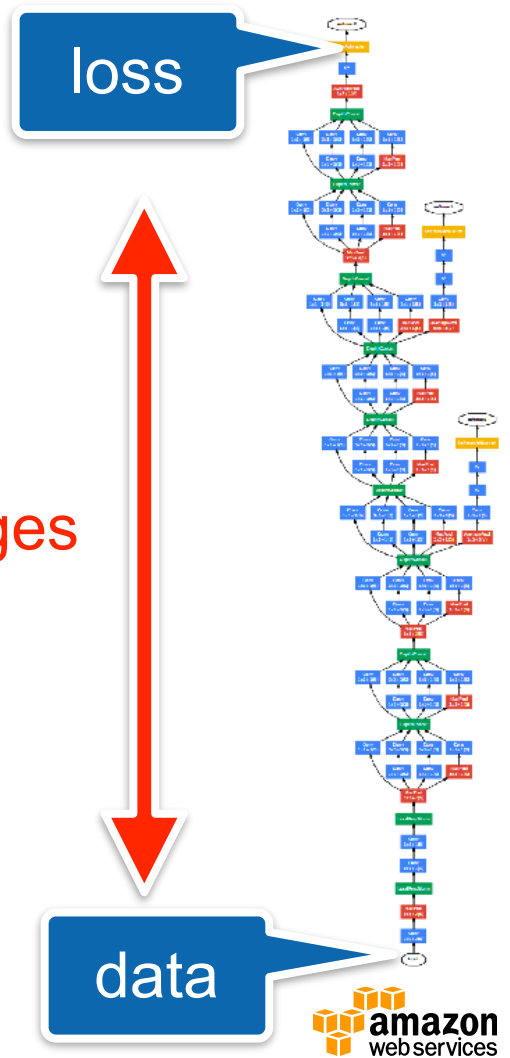
- **Parallel composition of branches**

```
def 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

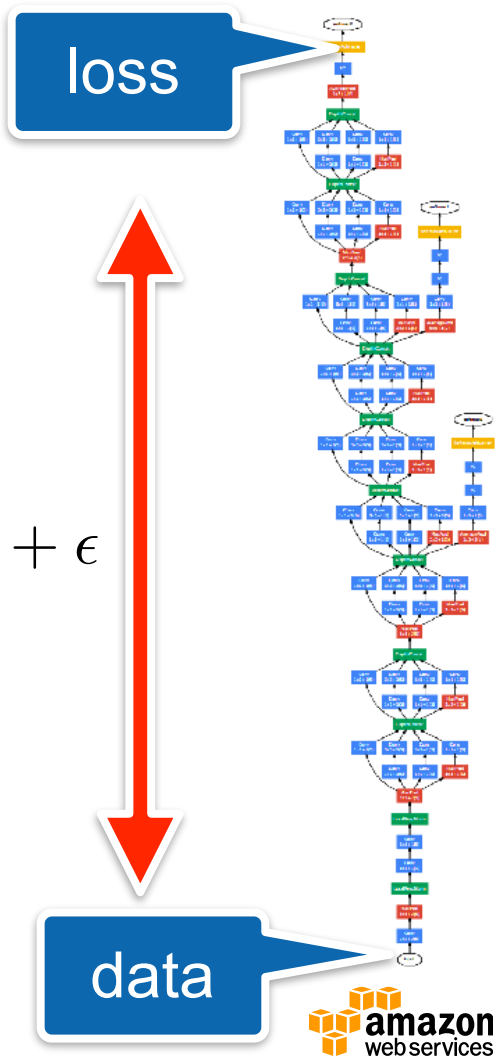
$$\mu_B = \frac{1}{|B|}\sum_{i \in B} x_i \text{ and } \sigma_B^2 = \frac{1}{|B|}\sum_{i \in B}(x_i - \mu_B)^2 + \epsilon$$

and adjust it separately

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

mean

variance



Model Zoo

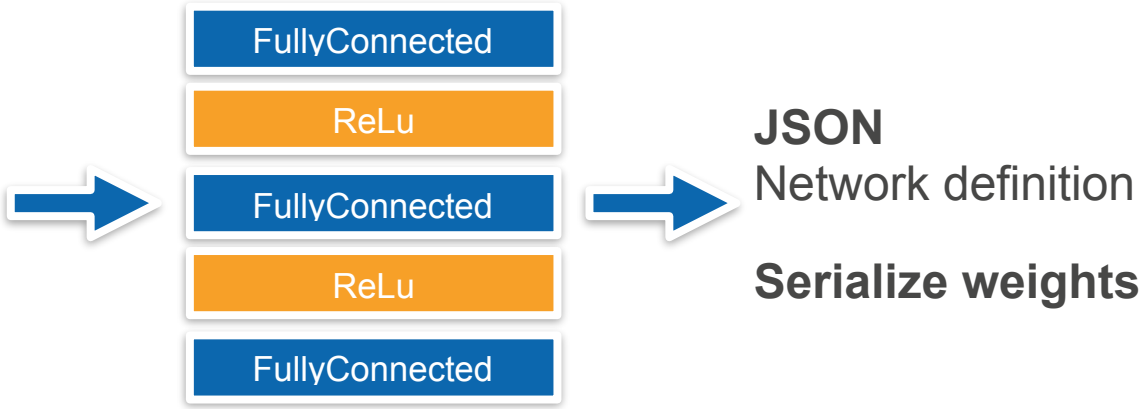
- Lots more networks at mxnet.gluon.model_zoo.vision (with all the model parameters you want)
  - AlexNet (because, why not)
  - SqueezeNet
  - Inception
  - VGG
  - DenseNet
  - ResNet

… 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)
```



Links & Notebooks

http://gluon.mxnet.io/P04-C03-deep-cnns-alexnet.html
http://gluon.mxnet.io/P04-C04-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. 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

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

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

- **Stochastic Gradient Descent (SGD)**

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

- **SGD with Momentum and Clipping**

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

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

- **Improved conditioning, momentum, precision …**

Momentum

- Average over recent gradients
  - Helps with local minima
  - Flat (noisy) gradients

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

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

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

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

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

https://blogs.princeton.edu/imabandit/2013/04/01/acceleratedgradientdescent/
https://www.cs.cmu.edu/~ggordon/10725-F12/slides/09-acceleration.pdf

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**

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

time window

Duchi, Hazan, Singer, 2010
http://www.magicbroom.info/Papers/DuchiHaSi10.pdf

Capacity control

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

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

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

prevents parameters from diverging

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

Dropout

- **Avoid parameter sensitivity**
  (small changes in value shouldn't change result)
- **Distributed representation**
  (information carried by more than 1 dimension)
- **Randomized sparsification**

$$y_{ti} = \xi_{ti} y_{ti} \text{ where } \begin{cases} \Pr(\xi_{ti} = \pi^{-1}) & = \pi \\ \Pr(\xi_{ti} = 0) & = 1 - \pi \end{cases}$$

- Same trick works for matrix W, too.
  DropConnect gives slightly better performance.

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

Dropout

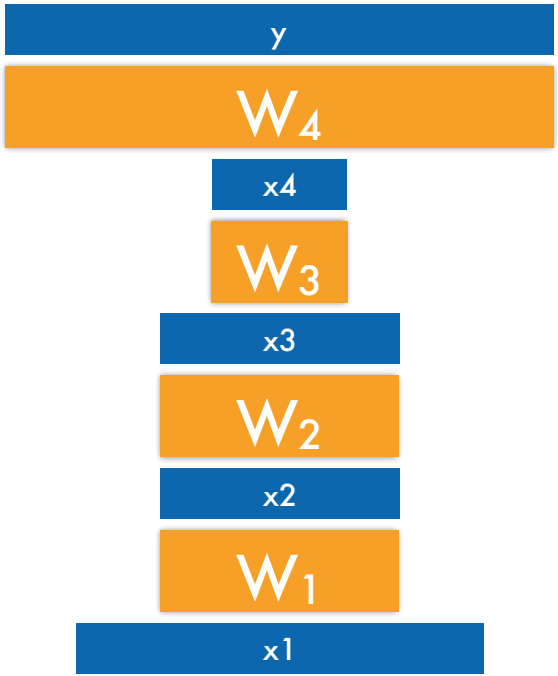
- Without dropout

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

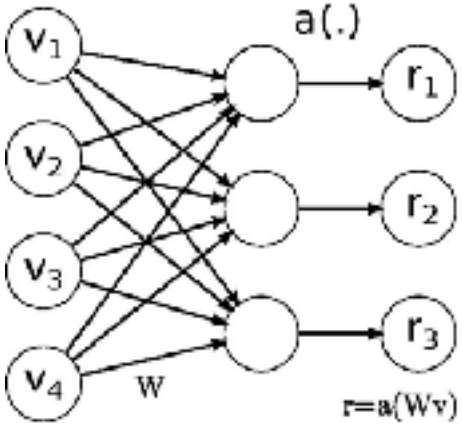
- With dropout

$$z_{ij} = \xi_{ij} x_{ij}$$
$$y_i = W z_i$$
$$x_{i+1} = \sigma(y_i)$$

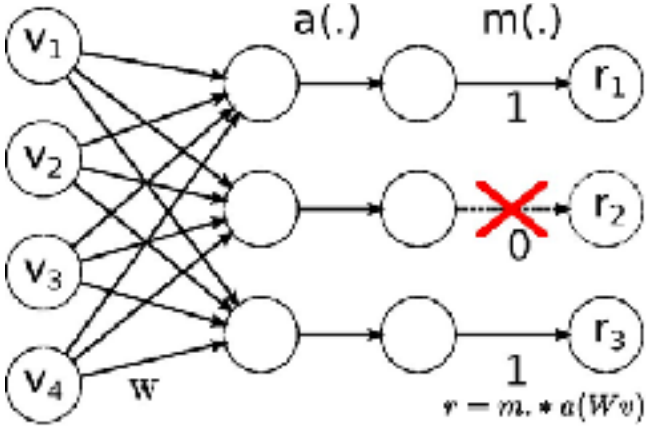
  You **must** draw a new mask for every step.
- Update via backprop



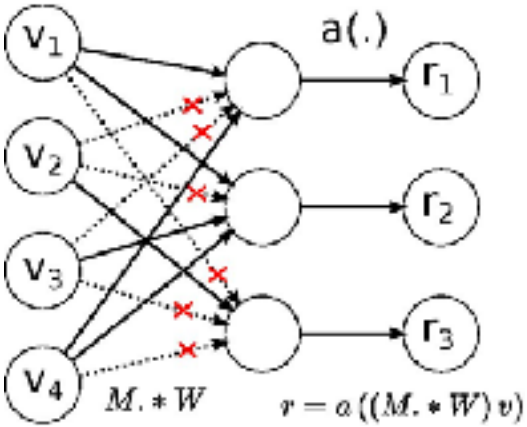
Dropout & DropConnect



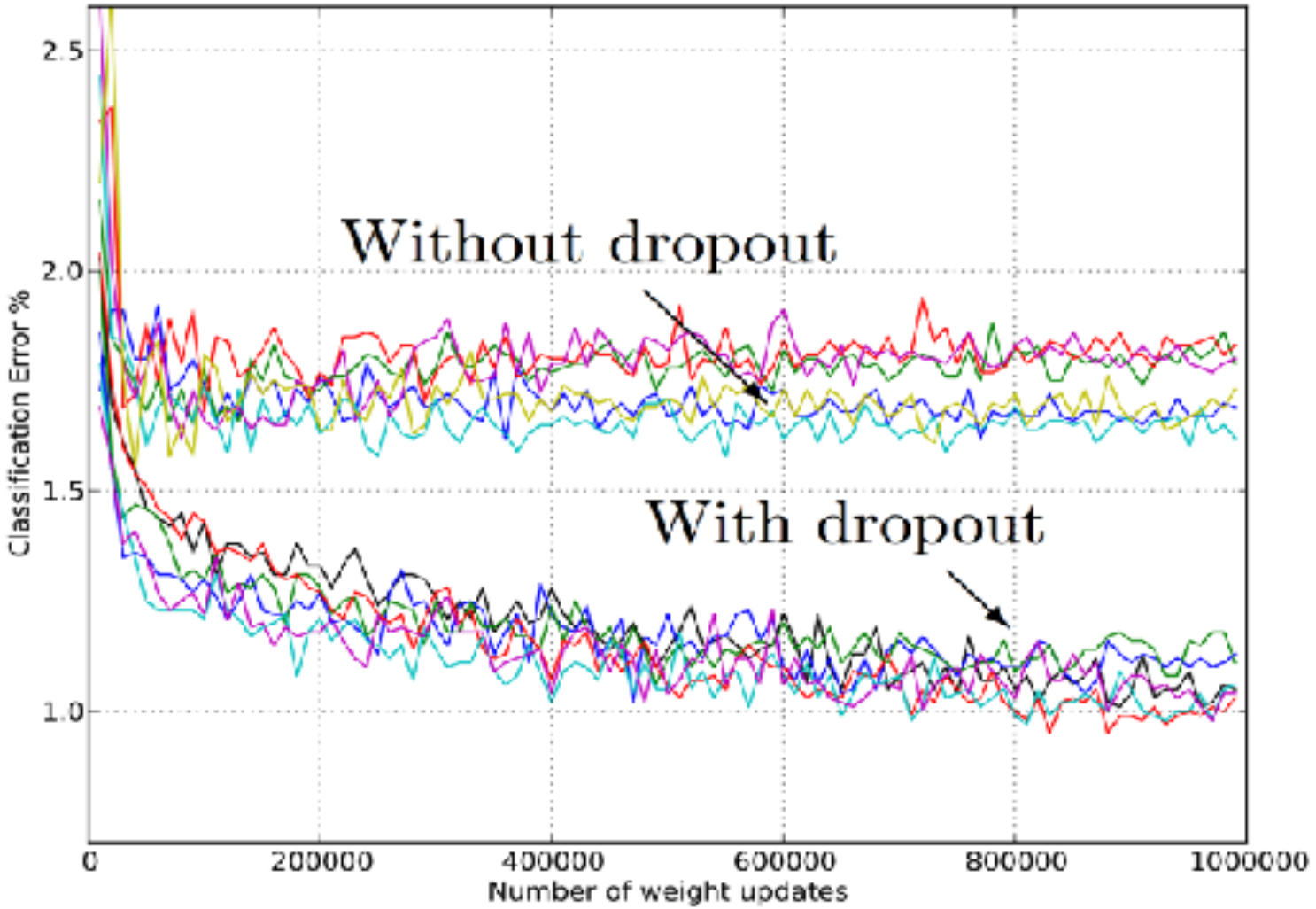
Regular



Dropout



DropConnect



Links & Notebooks

http://gluon.mxnet.io/P03-C03-mlp-dropout-scratch.html
http://gluon.mxnet.io/P03-C04-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. Loss Functions
    2.3. Convolutional Neural Networks
    2.4. Optimization & Capacity Control

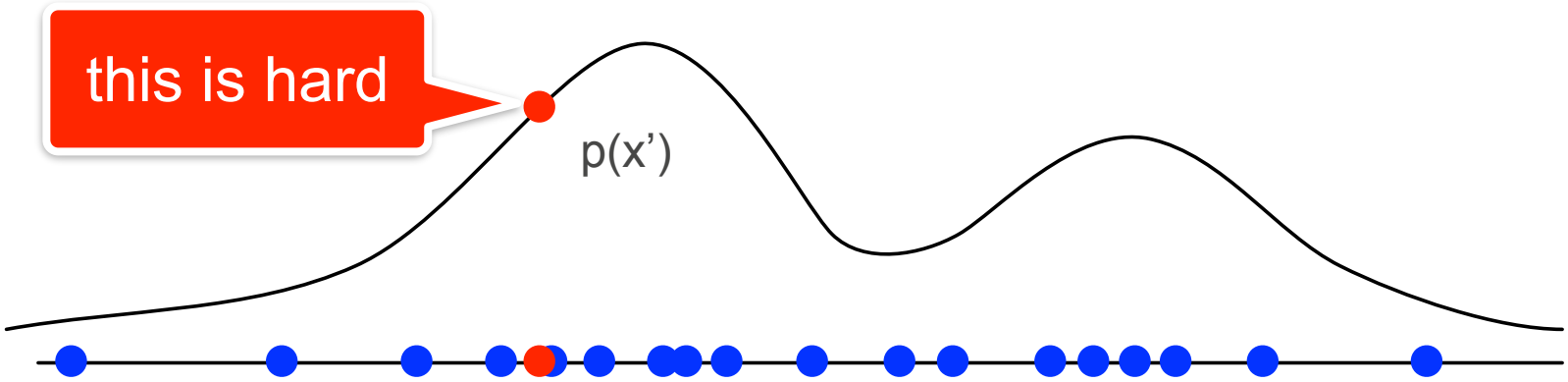
3. Advanced Topics
    3.1. Generative Adversarial Networks
    3.2. LSTMs
    3.3. TreeLSTMs
4. Scaling Learning
    4.1. Infrastructure on AWS
    4.2. Parallel Optimization Algorithms
    4.3. Parallel and Distributed Training

Generative Adversarial Networks Primer

- **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)

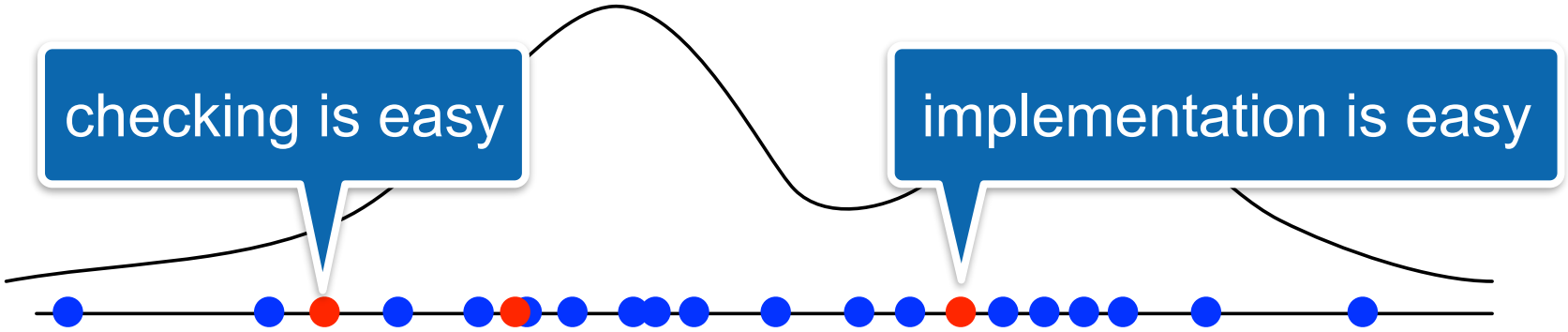
Generative Adversarial Networks Primer

- 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)



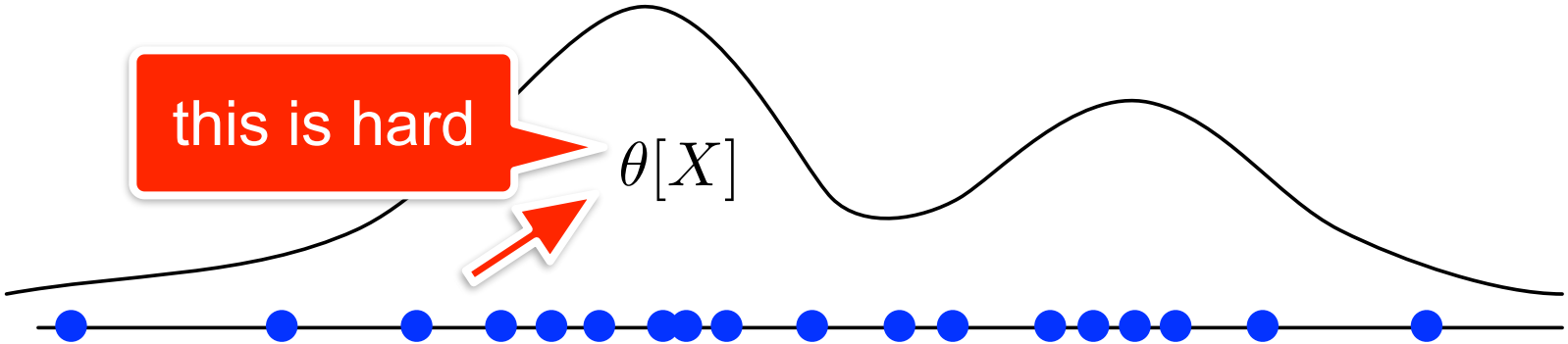
Generative Adversarial Networks Primer

- 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)



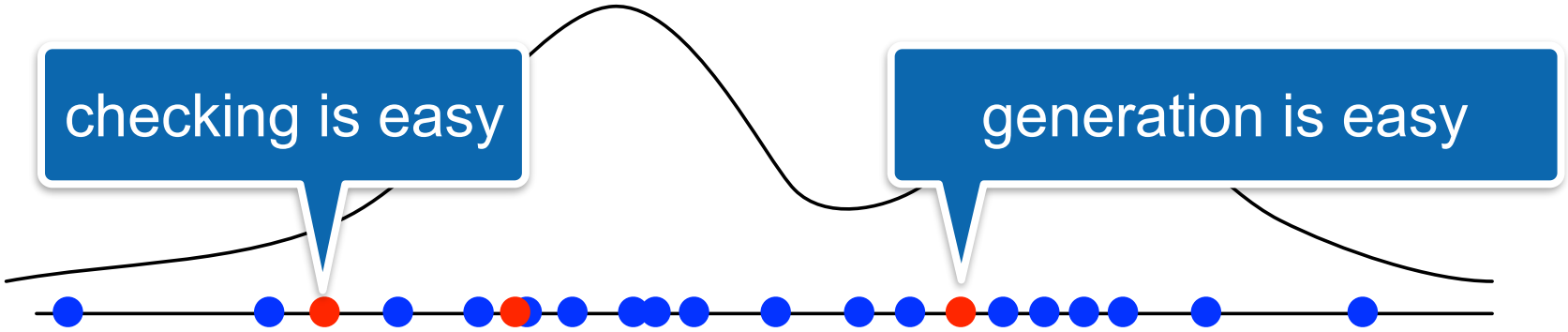
Generative Adversarial Networks Primer

- 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)



Key Idea - Focus on generating data

- **Generator network** x' = g(z) that produces 'fake' data
  - Start with some random z
  - Transform it via neural network
- **Discriminator network** d(x) distinguishing x from fake x'



Objective Functions

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

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

real data

fake data

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

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

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

Links & Notebooks

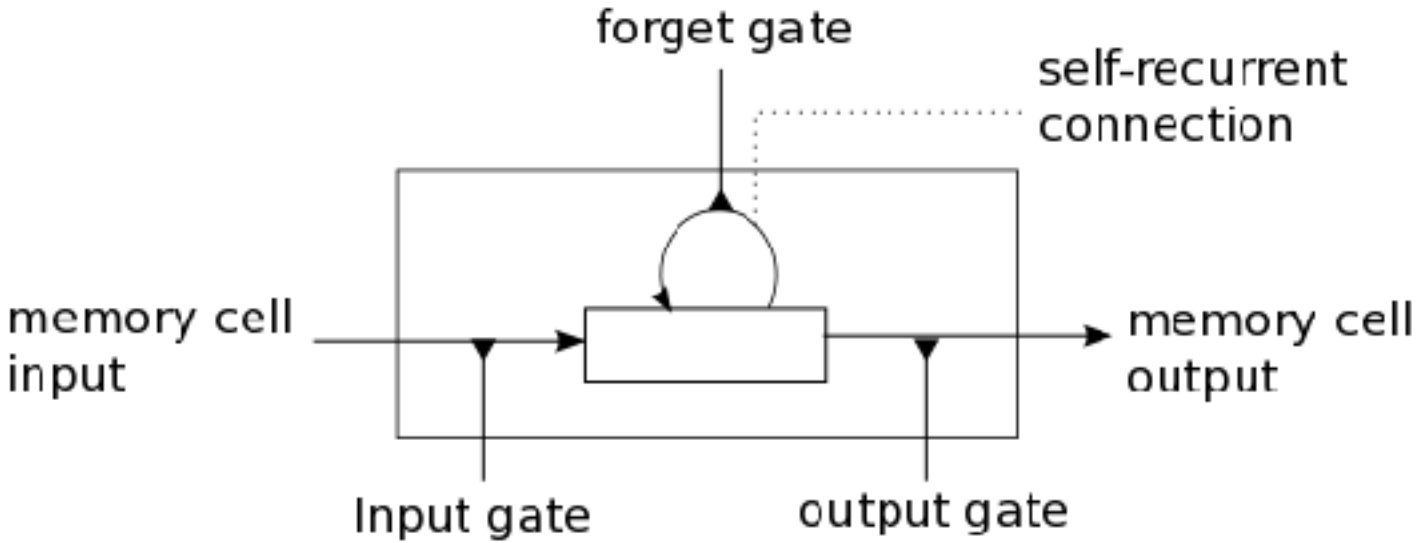
http://gluon.mxnet.io/P10-C01-gan-intro.html

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. 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

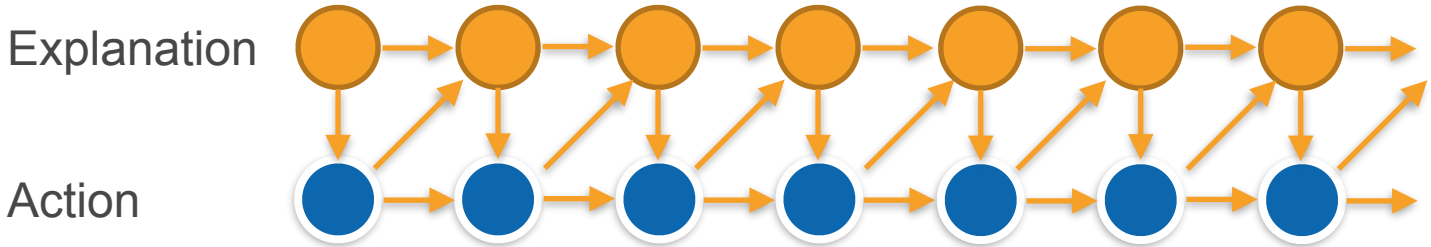
Sequence Models



Hochreiter & Schmidhuber, 1997

Latent Variable Models

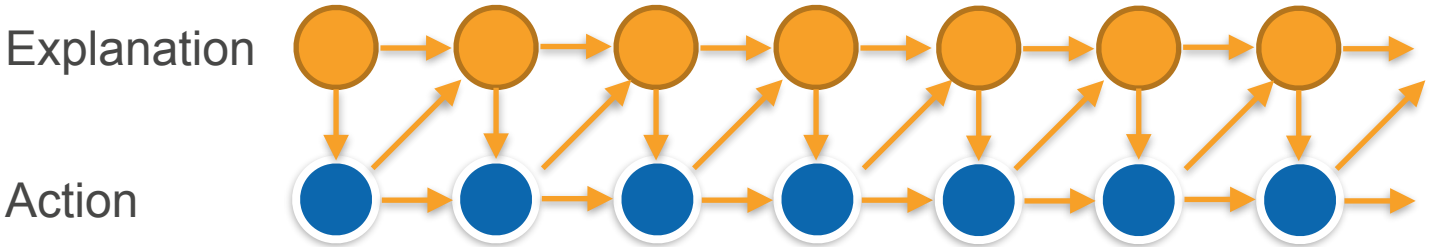
- **Temporal sequence of observations**
  Purchases, likes, app use, e-mails, ad clicks, queries, ratings
- **Latent state to explain behavior**
  - Clusters (navigational, informational queries in search)
  - Topics (interest distributions for users over time)
  - Kalman Filter (trajectory and location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



Latent Variable Models

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

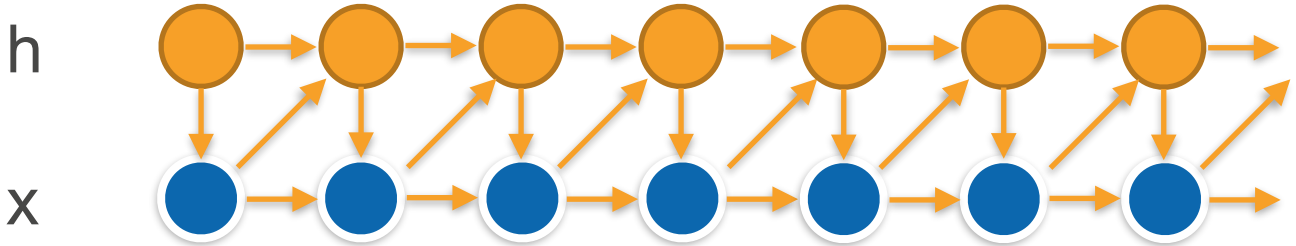
Are the parametric models really true?



Latent Variable Models

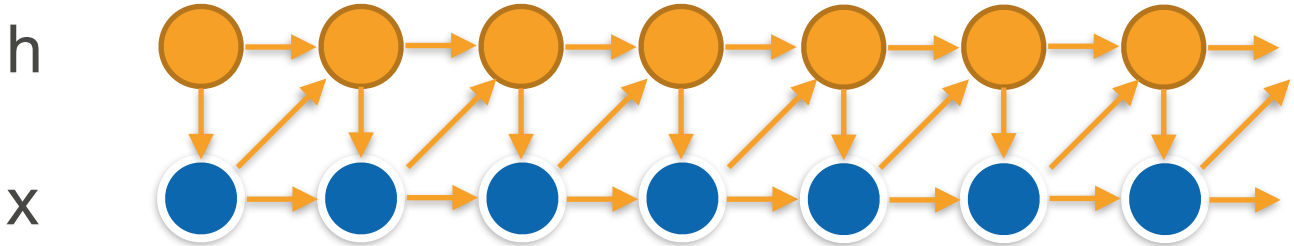
- **Temporal sequence of observations**
  Purchases, likes, app use, e-mails, ad clicks, queries, ratings
- **Latent state to explain behavior**
  - Nonparametric model / spectral
  - Use data to determine shape
  - Sidestep approximate inference

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



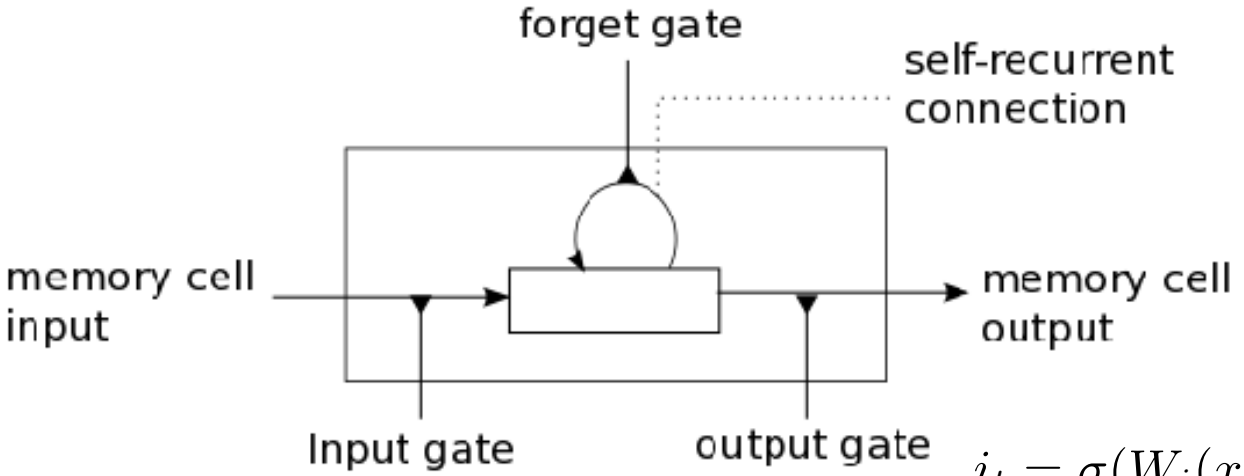
Latent Variable Models

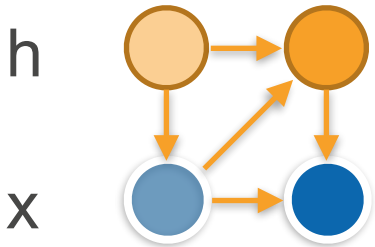
- **Temporal sequence of observations**
  Purchases, likes, app use, e-mails, ad clicks, queries, ratings
- **Latent state to explain behavior**
  - Plain deep network = RNN
  - Deep network with attention = LSTM / GRU …
    (learn when to update state, how to read out)



Long Short Term Memory

Hochreiter & Schmidhuber, 1997

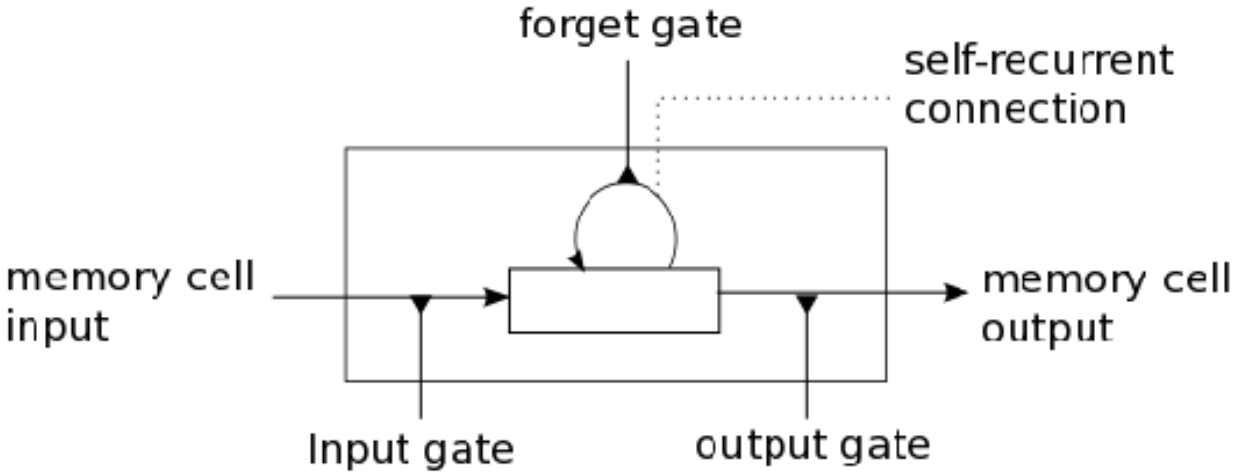


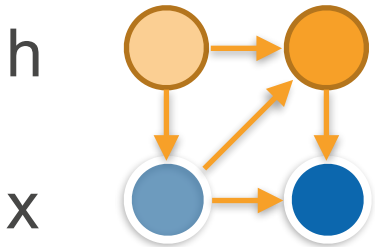


$$
\begin{aligned}
i_t &= \sigma(W_i(x_t, h_t) + b_i) \\
f_t &= \sigma(W_f(x_t, h_t) + b_f) \\
z_{t+1} &= f_t \cdot z_t + i_t \cdot \tanh(W_z(x_t, h_t) + b_z) \\
o_t &= \sigma(W_o(x_t, h_t, z_{t+1}) + b_o) \\
h_{t+1} &= o_t \cdot \tanh z_{t+1}
\end{aligned}
$$

Long Short Term Memory

Hochreiter & Schmidhuber, 1997





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

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

The anatomy of an RNN language model

- **Statistical Model**

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

- **Truncated Backpropagation Through Time (BPTT)**
  - Sequence is often too long for exact computation
  - Truncate and predict smaller segments (approximation)

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

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

$$(h_{i+1}, o_{i+1}) = \mathrm{LSTM}(h_i, x_i)$$

- **Decoder**
  - Softmax decoding for output, i.e

$$p(x_i|o_i) = \mathrm{softmax}(\mathrm{MLP}(o_i))$$

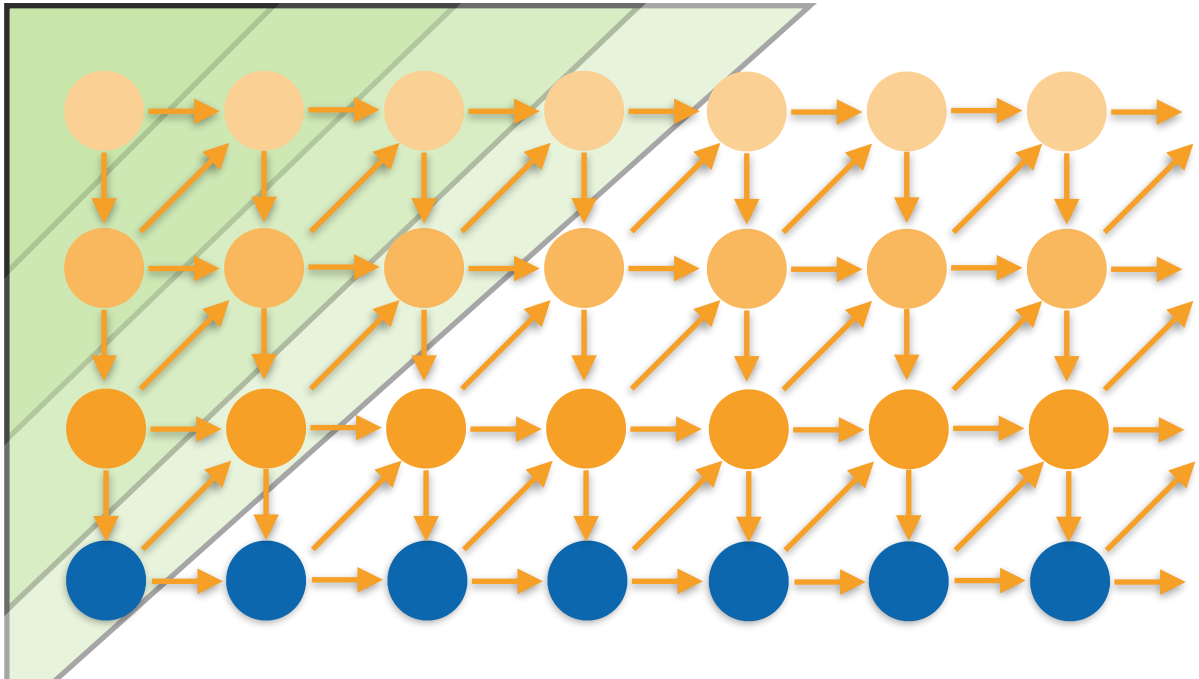
  - Better decoders possible (beam search, WFST …)

Links & Notebooks

gluon.mxnet.io/P05-C01-simple-rnn.html
gluon.mxnet.io/P05-C03-lstm-scratch.html
gluon.mxnet.io/P05-C04-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. 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

Tree LSTMs

- **LSTM**
  sequential dependence between states

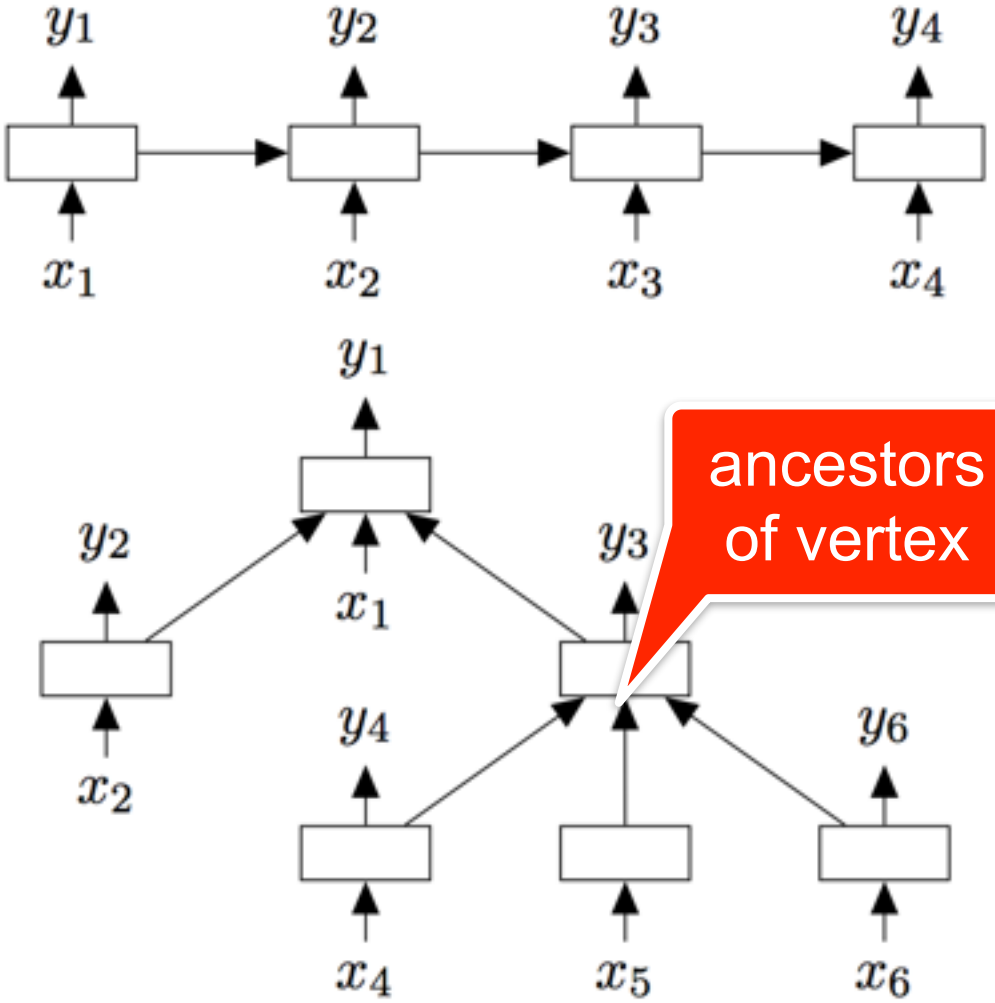
$$o_t = f(x_t, h_{t-1})$$

$$(c_t, h_t) = g(x_t, h_{t-1}, c_{t-1})$$

- **Tree LSTM**
  hierarchical dependence

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

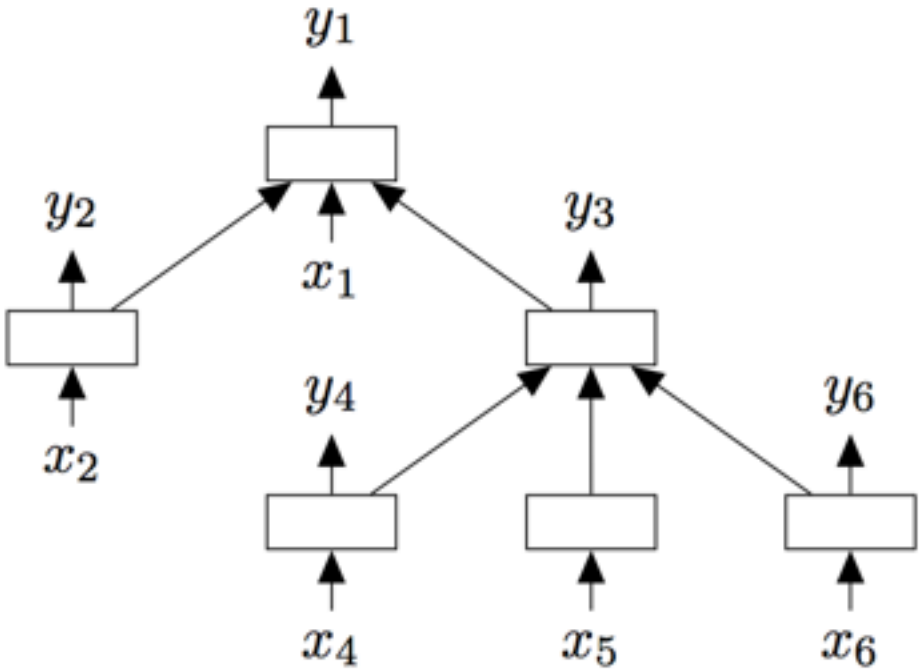
$$(c_t, h_t) = g(x_t, H_{t-1}, C_{t-1})$$



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

$$
\begin{aligned}
z &= W x_t + U h_{t-1} + b \\
[i_t, f_t, o_t, u_t] &= [\sigma(z_i), \sigma(z_f), \sigma(z_o), \tanh(z_u)] \\
c_t &= i_t \circ u_t + f_t \circ c_{t-1} \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

Tree LSTM

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

aggregate hidden state

individual forget weights

$$
\begin{aligned}
z &= W x_t + U \tilde{h} + b \\
f_{tj} &= \sigma\left(W_f x_t + U_f h_j + b_f\right) \\
[i_t, o_t, u_t] &= [\sigma(z_i), \sigma(z_o), \tanh(z_u)] \\
c_t &= i_t \circ u_t + \sum_{j \in C(t)} f_{tj} \circ c_j \\
h_t &= o_t \circ \tanh(c_t)
\end{aligned}
$$

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)

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

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

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

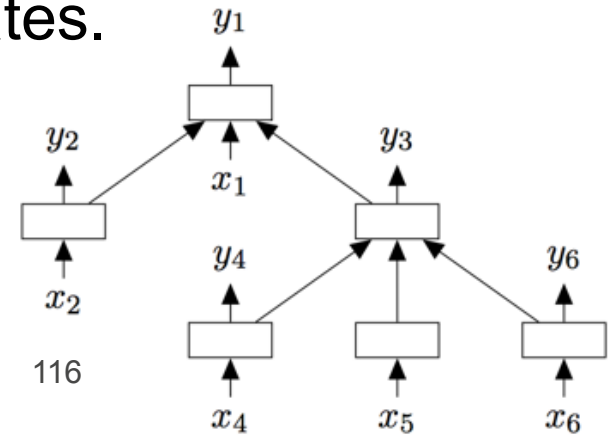
$$h_t = o_t \circ \tanh(c_t)$$

individual weight

individual forget weight

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

gluon.mxnet.io/P07-C08-tree-lstm.html
gluon.mxnet.io/P14-C05-hybridize.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.

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. 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

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		10^{7-8} web	10^{10} advertising	soc
RAM	1kB	100kB	10MB	100MB	1GB	100GB
CPU	100kF (8080)	1MF (80186)	10MF (80486)	tel Core)	GF NVIDIA	**>200TF**

old school neural nets

modern deep nets

kernel methods

A brief history of clusters

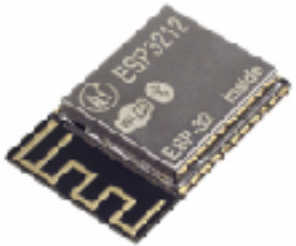
	1970	1980	1990	2000	2010	2020
Data (samples)	10^2 (e.g. Iris)	10^3	10^4 OCR	10^{7-8} web	10^{10} advertising	10^{12} social+ads
RAM	1kB	100kB	10MB	10GB	1TB	>10TB
CPU	100kF (8080)	1MF (80186)	10MF (80486)	100GF (Intel Core)	10TF NVIDIA	**>100PF**
Cluster size			10	100	32x4 GPU 1000 CPU	**128x8 GPU**
Network		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

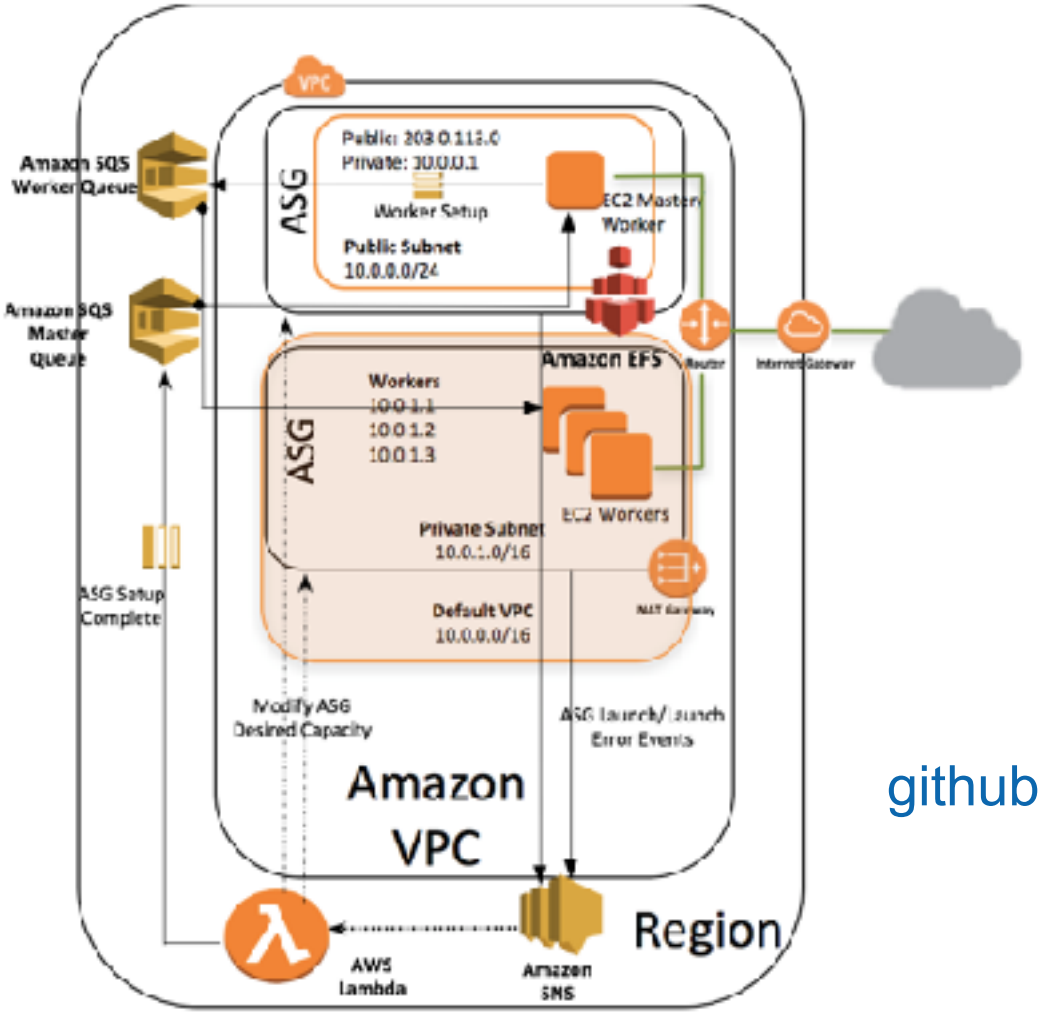
- **Compute** grows at highest rate **(1000x due to GPUs)**
- Data grows at Kryder's law (100x)
- Memory grows more slowly (20x)
- Network grows slowly **between computers** (10x)
- Network grows rapidly **within** (100x - NVLink/QPI)

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

amazon web services

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 f . AWS CloudFormation places this instance within the VPC and connects instance has both public IP addresses and DNS.
- An Amazon EFS file storage system configured in General Purpose perf
- 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. 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

Optimization Basics (Last Time)

- **Optimization Problem**

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

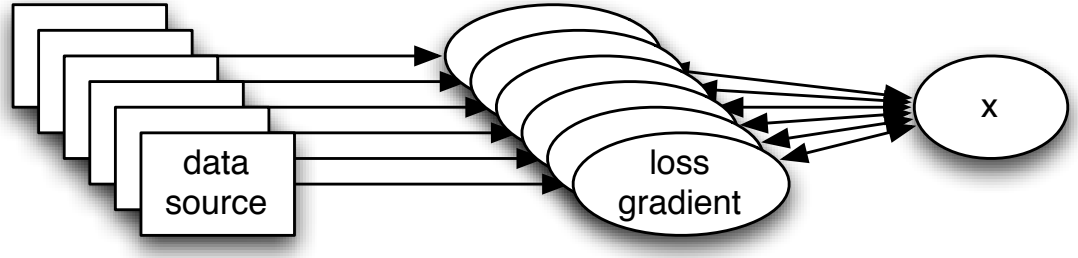
- **Stochastic Gradient Descent**
  - **Gradient on Minibatch**

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

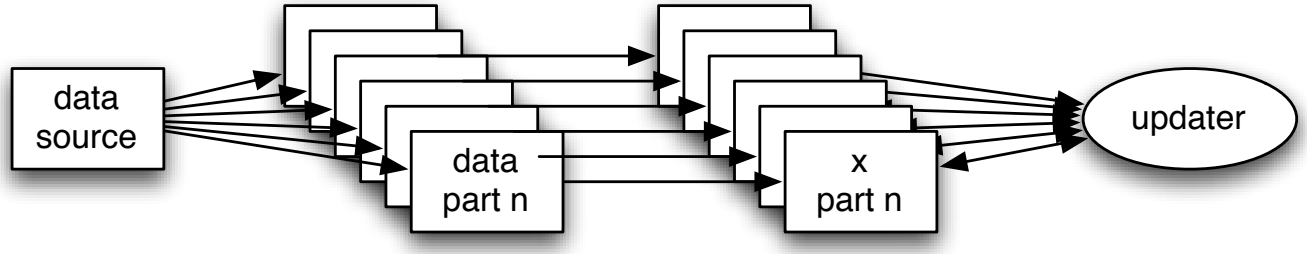
  - **More GPUs, more machines = Larger Minibatch Synchronization - data, model, gradients**

Parallelization

- **Data Parallel (easier & more efficient)**
  Distribute data over multiple GPUs / CPUs / machines

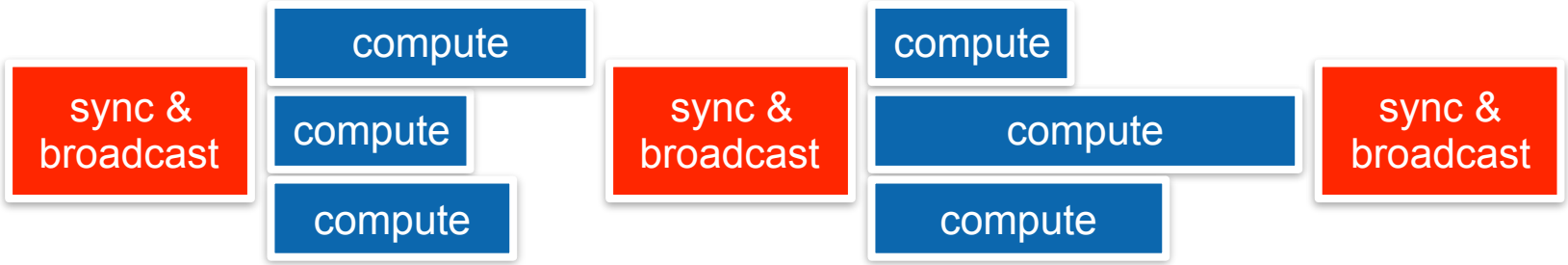


- **Model Parallel (only for huge models)**
  Distribute parts of the computation over multiple GPUs / CPUs

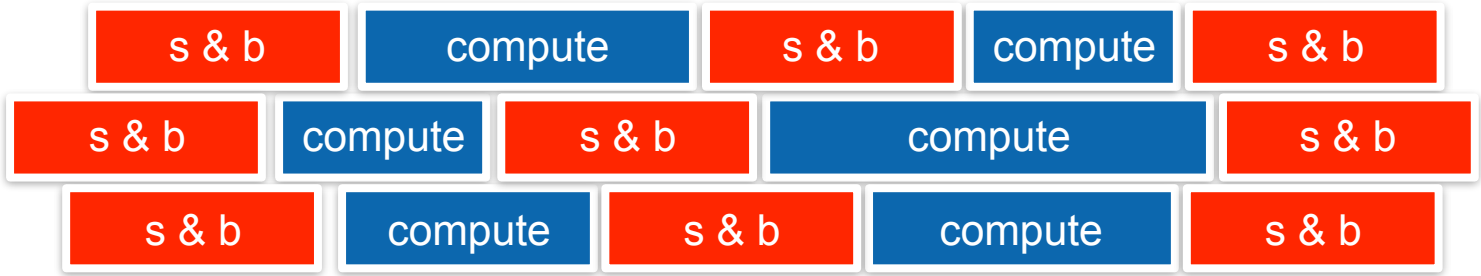


Parallelization

- **Synchronous - easy for homogeneous jobs**



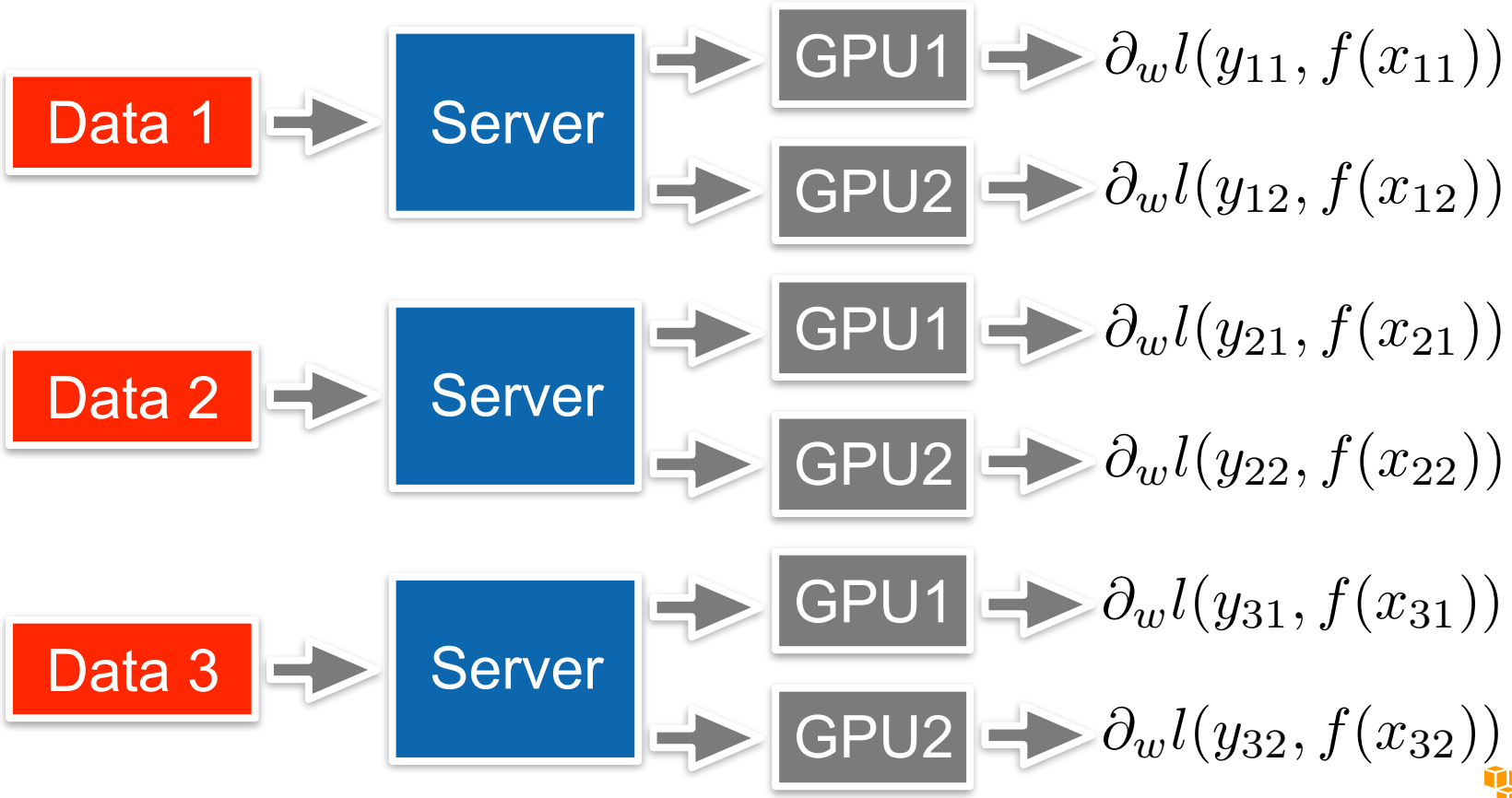
- **Asynchronous - efficient for high time dispersion**



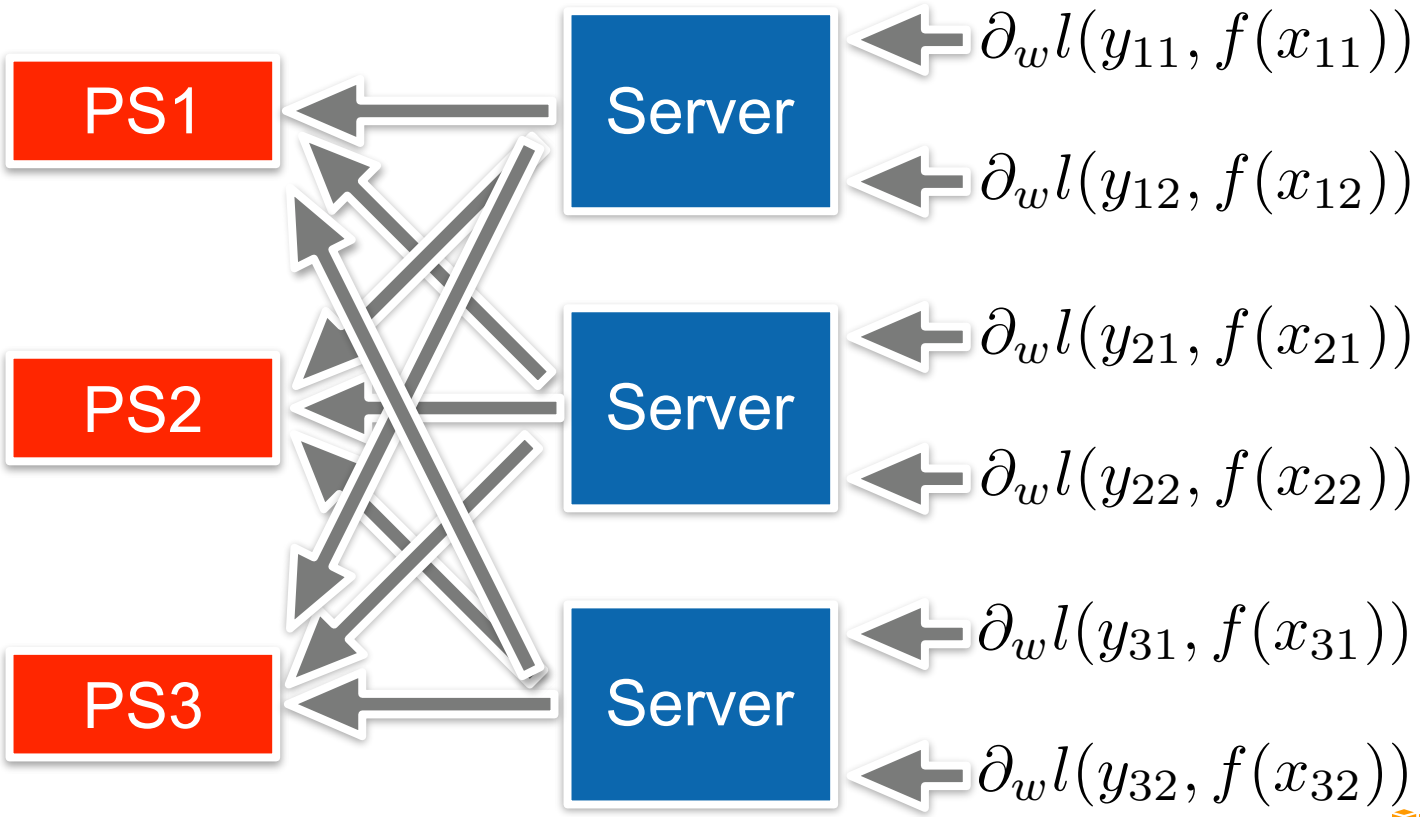
Parallelization - Pragmatic Strategy

- **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

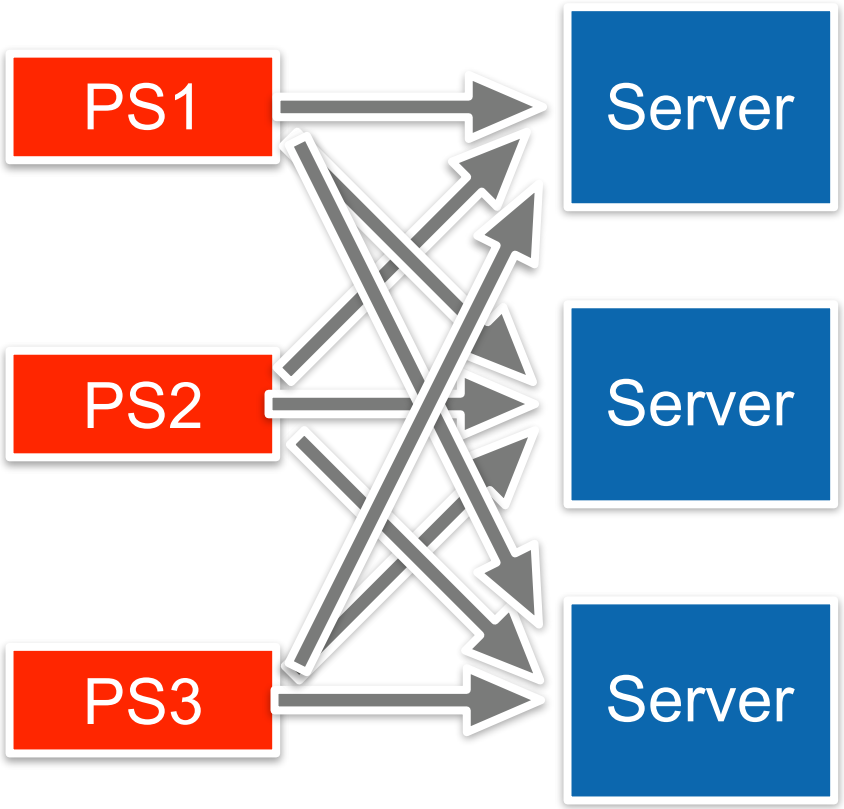
Parallelization - Pragmatic Strategy



Parallelization - Pragmatic Strategy

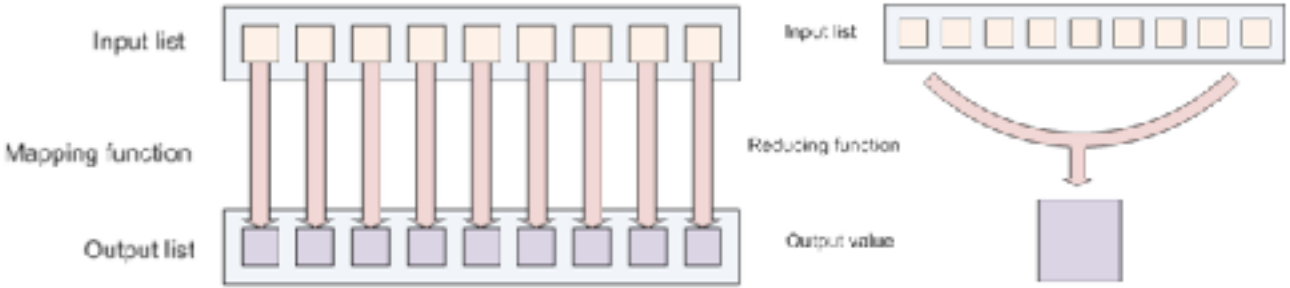


Parallelization - Pragmatic Strategy



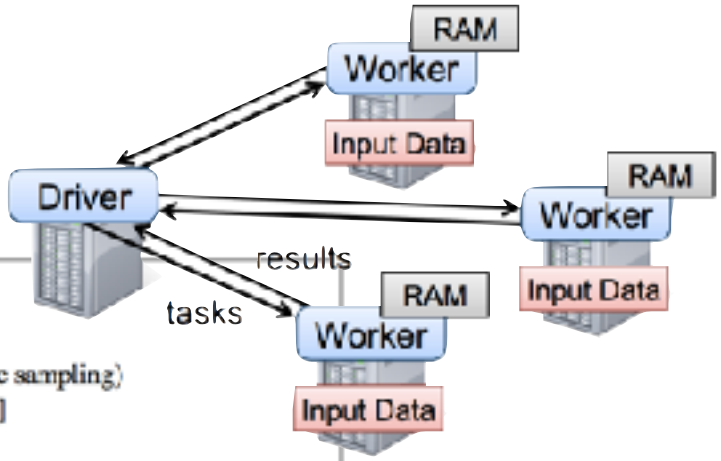
Why (not) MapReduce?

- **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)

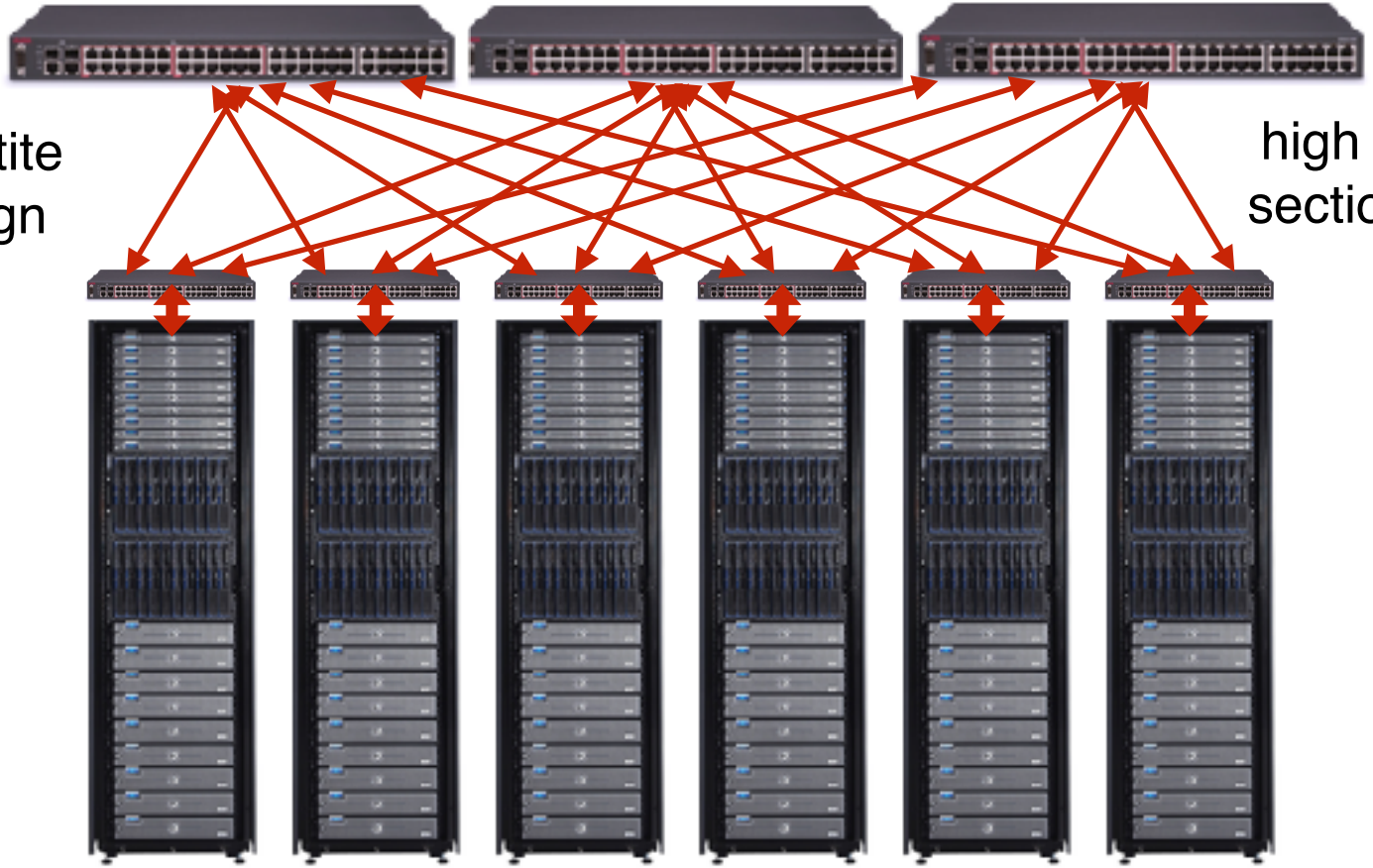
- Data is transformed by processing
- Store intermediate data using lineage
- Driver controls work



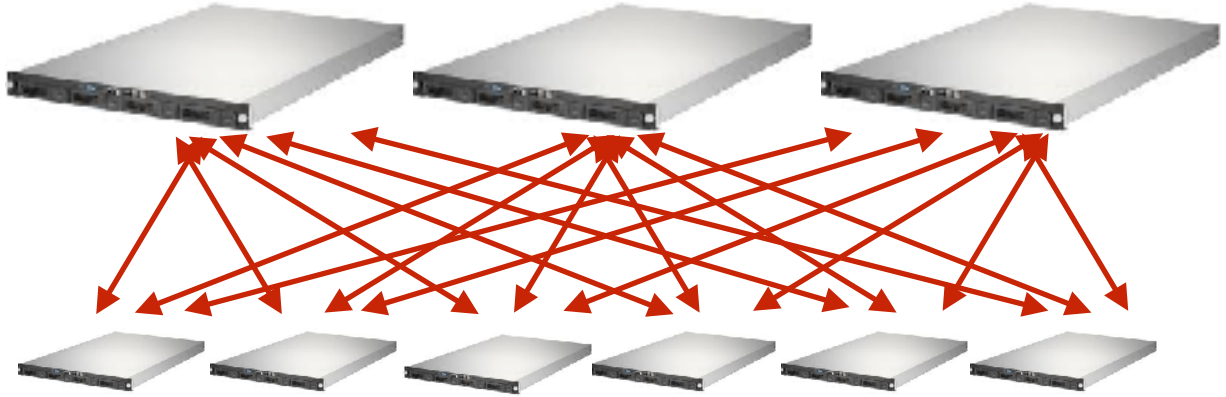
Transformations	*map*(*f* : T ⇒ U)	:	RDD[T] ⇒ RDD[U]
	filter(*f* : T ⇒ Bool)	:	RDD[T] ⇒ RDD[T]
	flatMap(*f* : T ⇒ Seq[U])	:	RDD[T] ⇒ RDD[U]
	sample(*fraction* : Float)	:	RDD[T] ⇒ RDD[T] (Deterministic sampling)
	groupByKey()	:	RDD[(K, V)] ⇒ RDD[(K, Seq[V])]
	reduceByKey(*f* : (V, V) ⇒ V)	:	RDD[(K, V)] ⇒ RDD[(K, V)]
	union()	:	(RDD[T], RDD[T]) ⇒ RDD[T]
	join()	:	(RDD[(K, V)], RDD[(K, W)]) ⇒ RDD[(K, (V, W))]
	cogroup()	:	
	crossProduct()	:	
	mapValues(*f* : V ⇒ W)	:	
	sort(*c* : Comparator[K])	:	
	partitionBy(*p* : Partitioner[K])	:	
Actions	*count*()	:	
	collect()	:	
	reduce(*f* : (T, T) ⇒ T)	:	RDD[T] ⇒ T
	lookup(*k* : K)	:	RDD[(K, V)] ⇒ Seq[V] (On hash/range partitioned RDDs)
	save(*path* : String)	:	Outputs RDD to a storage system, e.g., HDFS

great for ETL
bad for many low latency updates

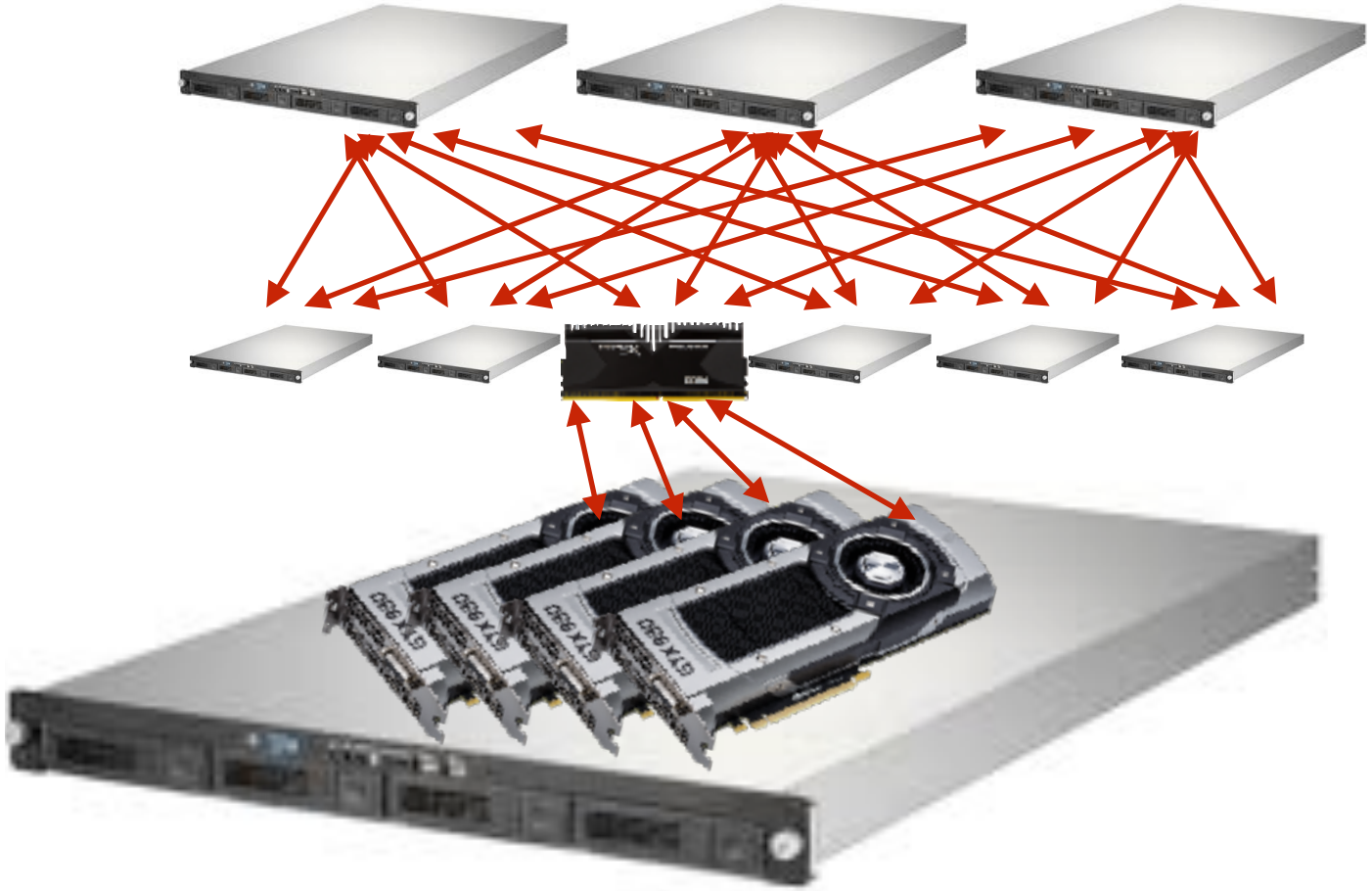
An idea from network design (Clos Networks)



Parameter Server aka (key,value) store

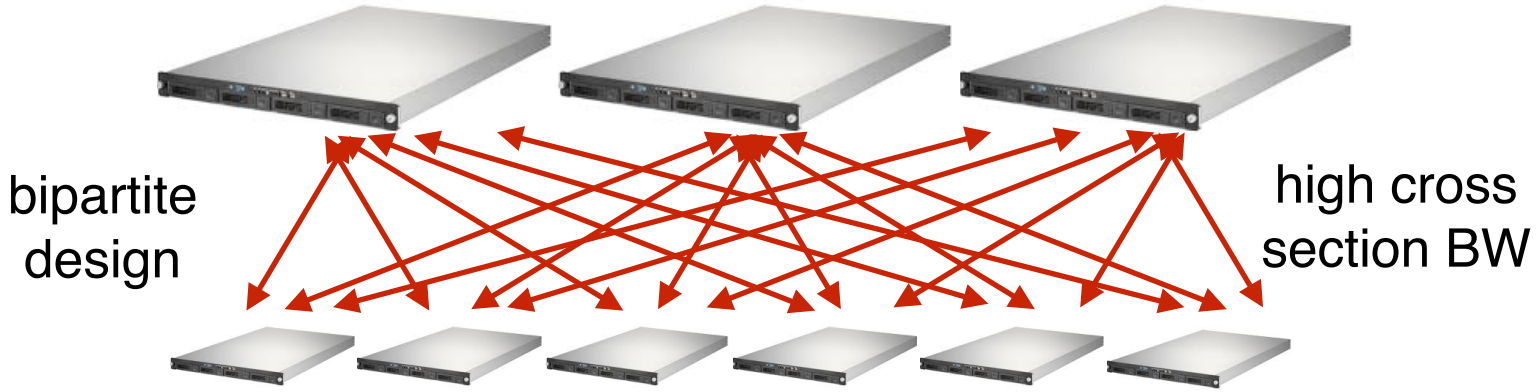


Parameter Server aka (key, value) store



amazon web services

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 & Narayanamurthy, 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)

amazon web services

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

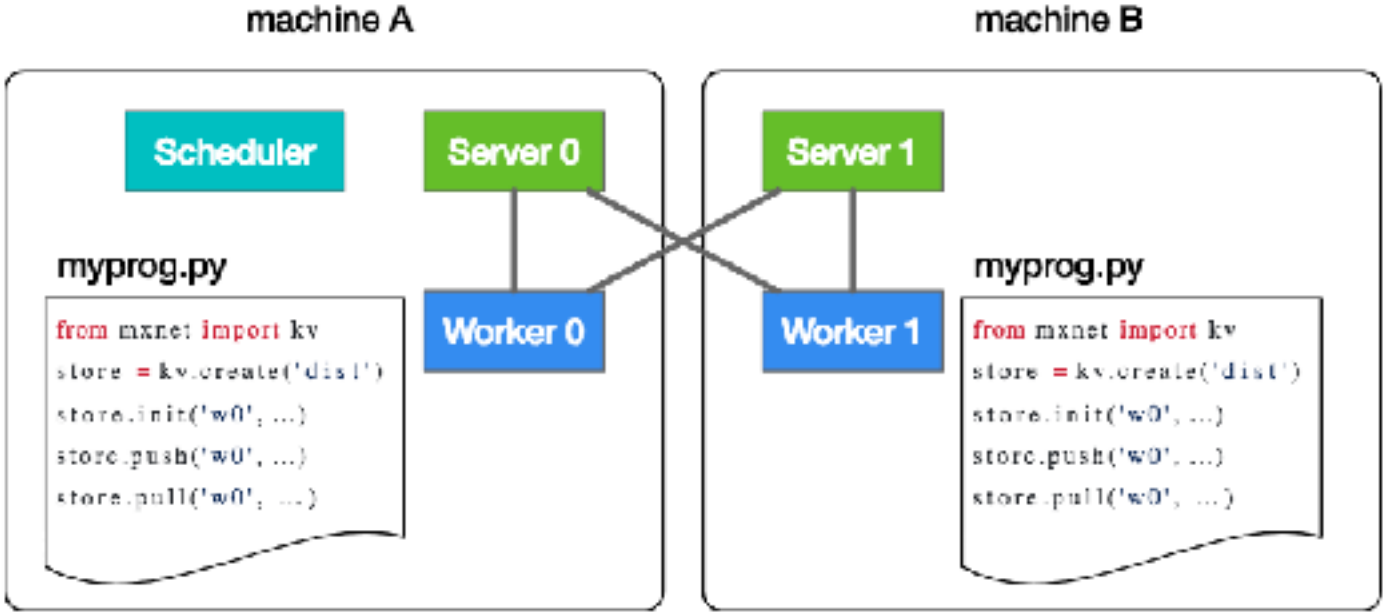
1. Getting Started
    1.1. Installation
    1.2. Deep Learning 101
    1.3. NDArray
2. Neural Networks in MXNet
    2.1. Linear Models & MLPs
    2.2. Loss Functions
    2.3. Convolutional Neural Networks
    2.4. Optimization & Capacity Control
3. Advanced Topics
    3.1. Generative Adversarial Networks
    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

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

amazon web services