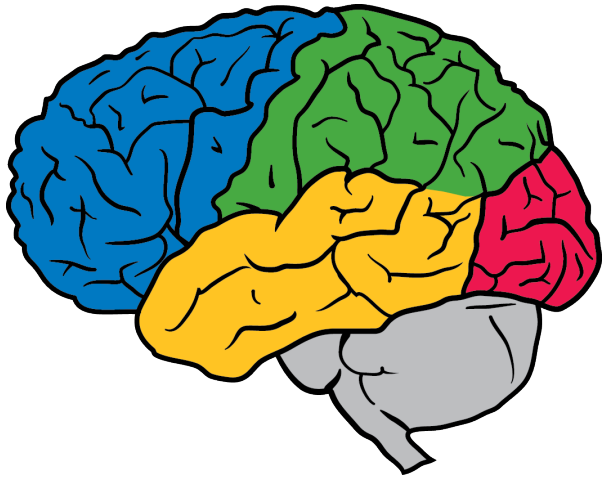
Large Scale Deep Learning

Vincent Vanhoucke

Quick Introduction

Tech Lead on the Google Brain team.

Reach me at: vanhoucke@google.com



Objectives

Give you a **practical understanding** of neural network training.

Emphasize what matters **at scale**, when models and data get large.

Dive into some of the more important **classes of models**.

Talk about some of the most exciting lines of **research in the field**.

Point out along the way the many dead-ends as well!

Lecture Agenda

Wednesday 2nd 13:40-15:10

Introduction to neural networks.
The fundamentals of model training.

Wednesday 2nd 15:30-17:00

Topics on model training: Regularization and Parallelism.
Notable models: Convolutional Networks, LSTMs, and Embeddings.

Thursday 3rd 10:30-12:00

Deep dive into applications.
Hot topics in deep learning research.

Session I

The promise (or wishful dream) of Deep Learning

Universal Machine Learning

Speech
Text
Search Queries
Images
Videos
Labels
Entities
Words
Audio
Features

Simple
Reconfigurable
High Capacity
Trainable end-to-end
Building Blocks

Speech
Text
Search Queries
Images
Videos
Labels
Entities
Words
Audio
Features

The promise (or wishful dream) of Deep Learning

Common representations across domains.

Replacing smarts with **data**.

Would merely be an interesting academic exercise…

…**if it didn't work so well!**

Recent Kaggle (http://kaggle.com) ML Competitions

Plankton Identification

Molecular Activity Prediction

Galaxy classification

Higgs Particle Detection

Note which other techniques often win competitions:

- Gradient Boosting
- Random Forests

If your ML class doesn't cover those (they rarely do), ask for a refund.

In Research and Industry

Speech Recognition

Speech Recognition with Deep Recurrent Neural Networks
Alex Graves, Abdel-rahman Mohamed, Geoffrey Hinton

Convolutional, Long Short-Term Memory, Fully Connected Deep Neural Networks
Tara N. Sainath, Oriol Vinyals, Andrew Senior, Hasim Sak

Object Recognition and Detection

Going Deeper with Convolutions
Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich

Scalable Object Detection using Deep Neural Networks
Dumitru Erhan, Christian Szegedy, Alexander Toshev, Dragomir Anguelov

Google

In Research and Industry

Machine Translation

Sequence to Sequence Learning with Neural Networks
Ilya Sutskever, Oriol Vinyals, Quoc V. Le
Neural Machine Translation by Jointly Learning to Align and Translate
Dzmitry Bahdanau, Kyunghyun Cho, Yoshua Bengio

Parsing

Grammar as a Foreign Language
Oriol Vinyals, Lukasz Kaiser, Terry Koo, Slav Petrov, Ilya Sutskever, Geoffrey Hinton

Language Modeling

One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling
Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, Tony Robinson

Neural Networks … without the Neuro-Babble

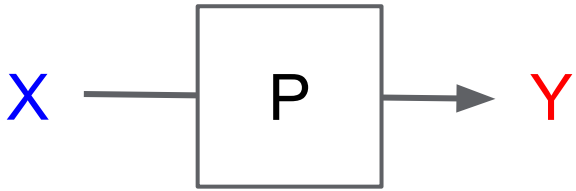
Imagine you want to build a:

Tractable

Highly **Non-linear**

Parametric

function that you can use as a predictor.



How To Build a Tractable, Non-Linear, Parametric Function

Step 1: Start with a linear function: $Y = AX$

Linear functions are nice!

- Very **efficient** to compute (BLAS). GPUs were designed for them!
- Very **stable** numerically.
- Well behaved and numerically **efficient derivatives**.
- Lots of free parameters: $O(N^2)$ for N inputs.

If you do optimization (and your name is not Steve Boyd),
you really want to optimize linear functions.

How To Build a Tractable, Non-Linear, Parametric Function

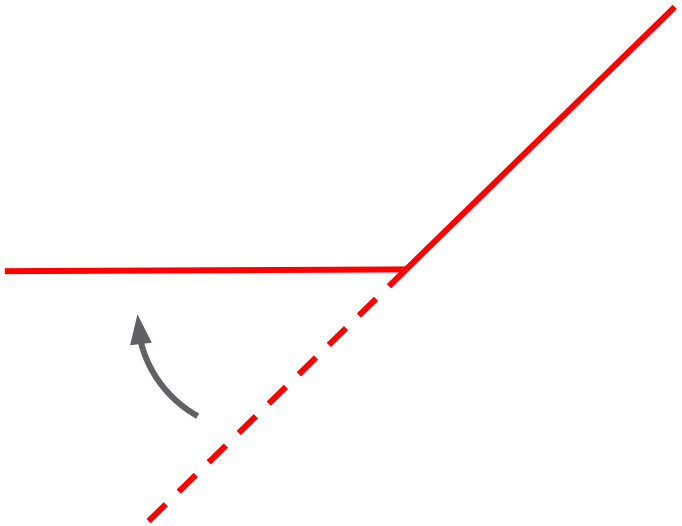
Step 2: Add a non-linearity.

Meet the Rectified Linear Unit (ReLU)

The simplest non-linearity possible:

max(0, X)

Well behaved derivative: X > 0 ? 1 : 0



How To Build a Tractable, Non-Linear, Parametric Function

Step 3: Repeat!

Very efficient representation:

Parameters are all in linear functions, yet the stack is very non-linear.

Empirically, deeper models require many fewer parameters than ‘shallow’ models of same representational power.

Where's my neuron?

'This is how the brain works' - G. Hinton

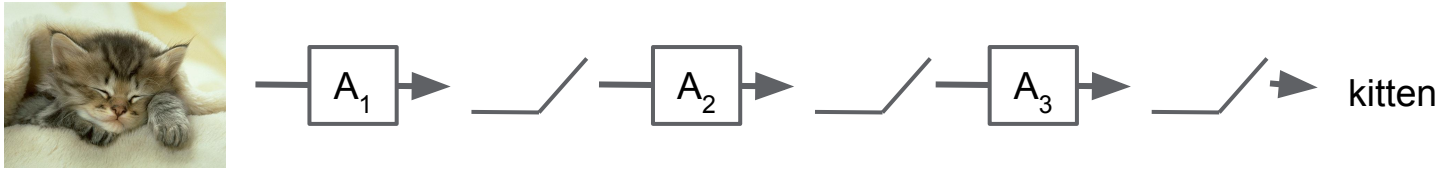
You can paint a neuro-inspired picture of neural networks / deep learning, but you don't have to, and it comes with decades of 'baggage'.

You can just think in terms of parametrizing a large non-linear function in a way that makes sense computationally, and ignore the neuro-talk.

Your pick!

Orders of Magnitude

State of the Art Object Recognition Models:



15-25 layers
10-200M parameters
1-5B multiply-adds / image

Training Models

1. The Maths
2. The Stats
3. The Hacks
4. The Computer Science

The Maths

A Neural Network in Equations

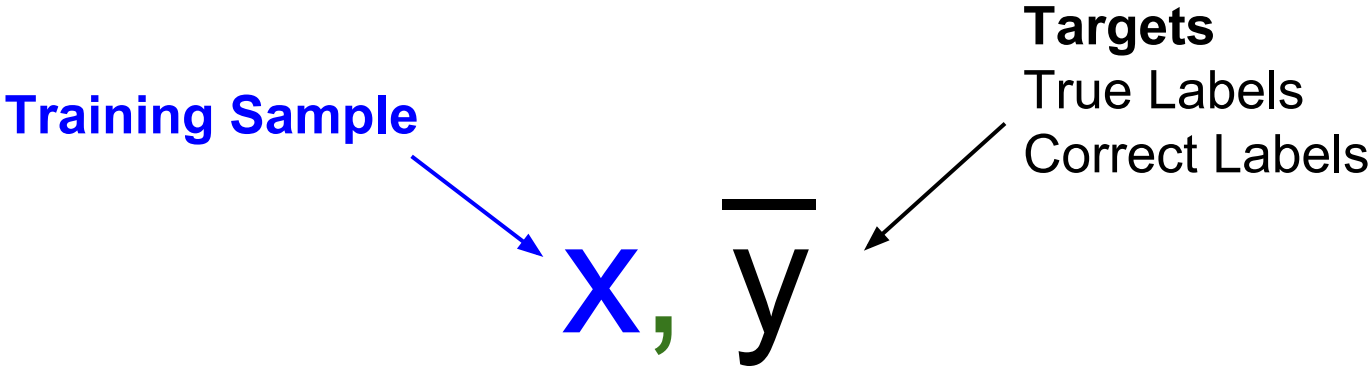
Outputs
Labels
Predictions
Cat / No Cat
Phonemes
Next Word
…

$$y = nn(x, w)$$

Inputs
Images
Spectrograms
Features
Words
…

Weights
Parameters

Training Data



Objective: $y=nn(x,w) \approx \bar{y}$

The Loss: 'How Close are We?'

$$y = nn(x, w)$$

Sum over *all* the training data.

$$\sum L(\overline{y}, y)$$

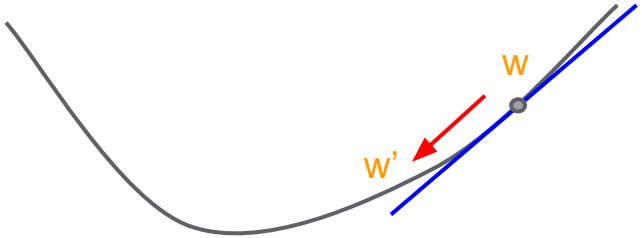
$$L_2 \text{ loss: } \sum |\overline{y} - y|^2$$

$$\text{Cross-entropy loss: } -\sum \overline{y} \log y$$

Training == Minimizing The Loss w.r.t. Weights

$$\text{argmin}_w \; L(w) = L(\bar{y}, \; nn(x, w))$$

Minimize using Gradient Descent:



Learning Rate

$$w' = w - \alpha \, \partial_w L(w)$$

Gradient
Derivative
Delta

The Loss is a Very Complicated Function of the Weights

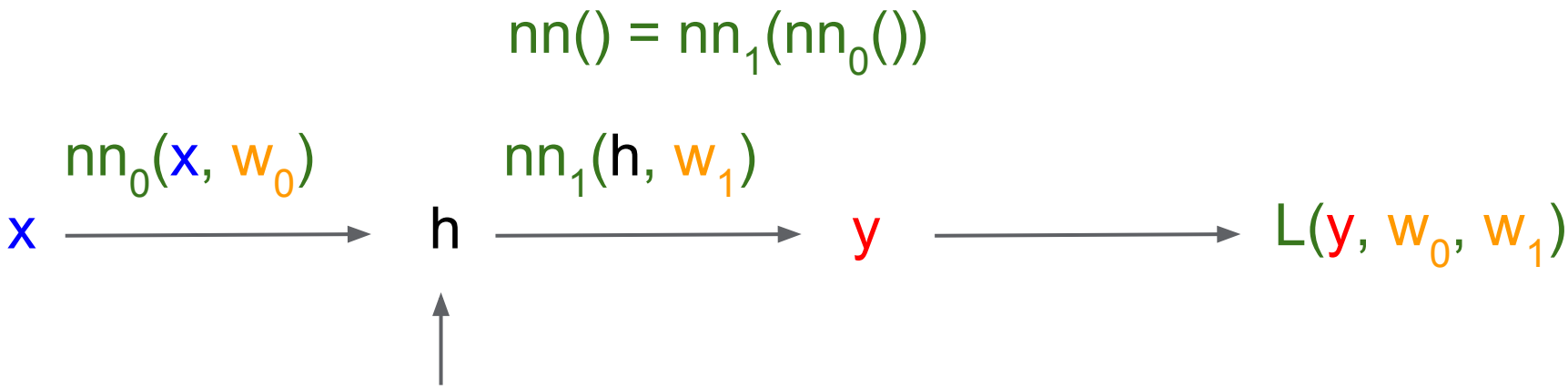
$$L(\bar{y}, nn(x,w))$$

1. nn() is a very non-linear, non-convex function!
2. Depends on all the Inputs and Targets in the training set!

Two main tricks to simplify the problem:

1. Back-Propagation
2. Stochastic Gradient Descent

Back-Propagation: Factoring of the Neural Net



Activations
Hidden States

Remember the ‘Chain Rule’ from High School:

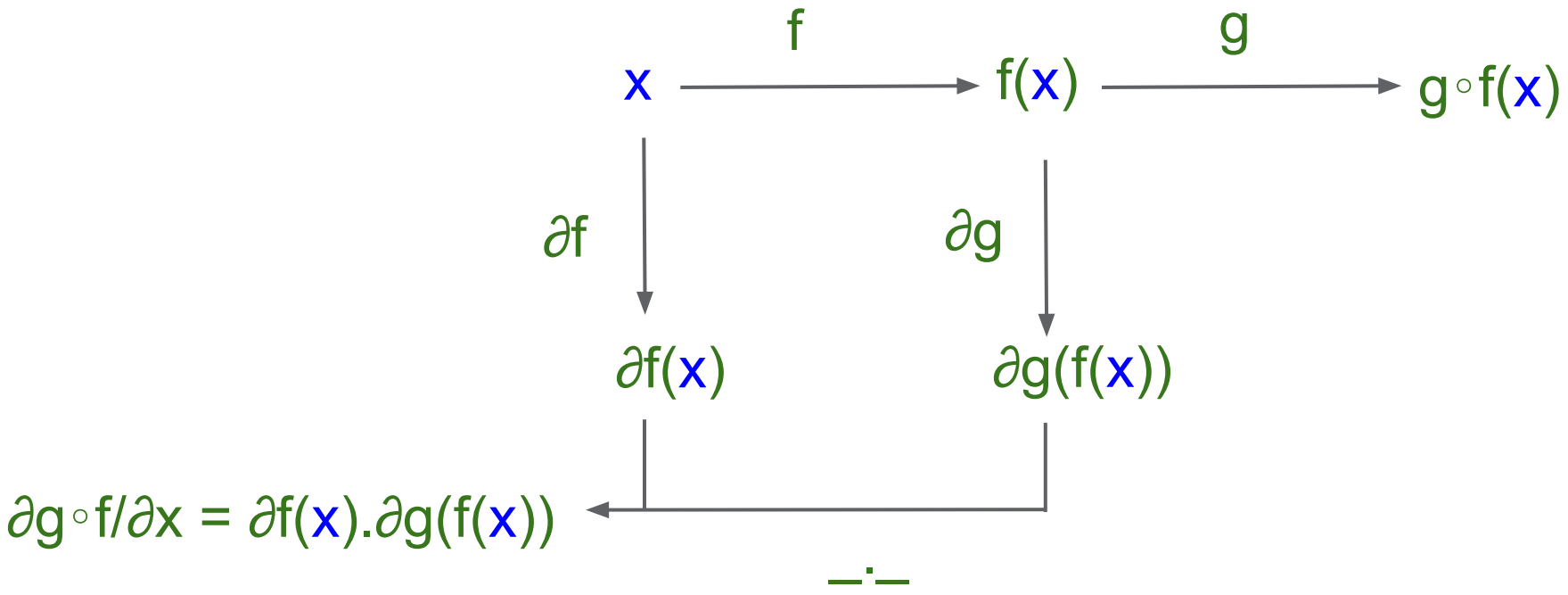
$$g(f(x))' = g'(f(x)).f'(x)$$

$$\frac{\partial g \circ f}{\partial x} = \frac{\partial g}{\partial f} \cdot \frac{\partial f}{\partial x}$$

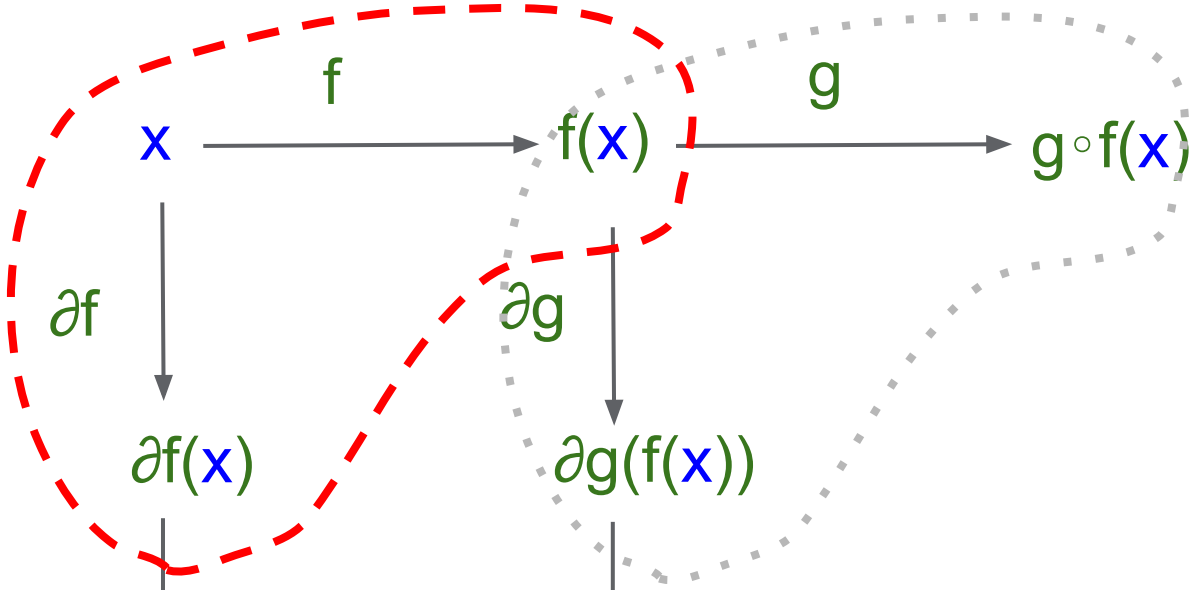
Chaining:

$$\frac{\partial j \circ i \circ h \circ g \circ f}{\partial x} = \frac{\partial j}{\partial i}\frac{\partial i}{\partial h}\frac{\partial h}{\partial g}\frac{\partial g}{\partial f}\frac{\partial f}{\partial x}$$

Graphical View of the Chain Rule



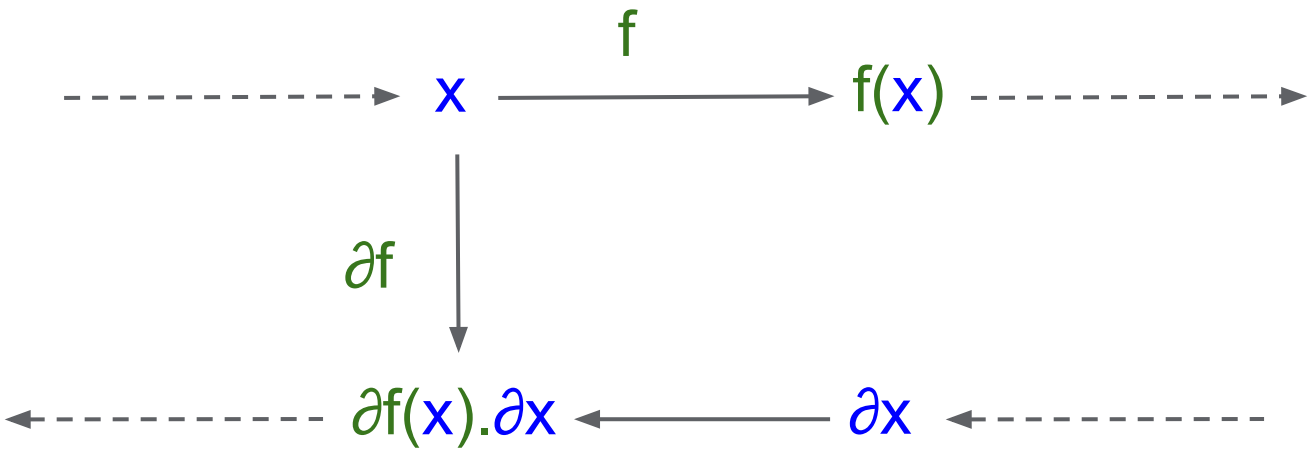
Graphical View of the Chain Rule



$$\partial g \circ f / \partial x = \partial f(x).\partial g(f(x))$$

—·—

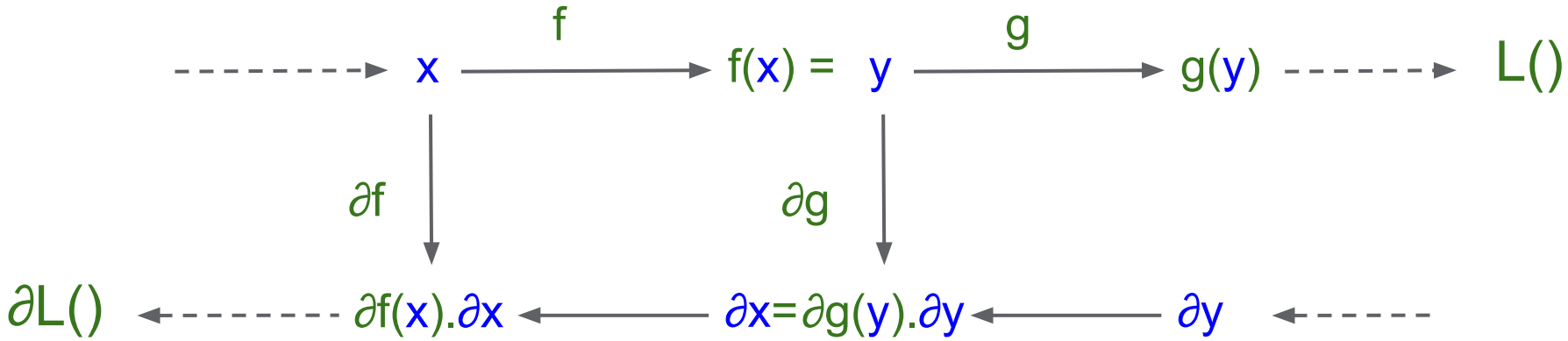
Back-Propagation using the Chain Rule



You can compute the gradient with respect to any quantity by:

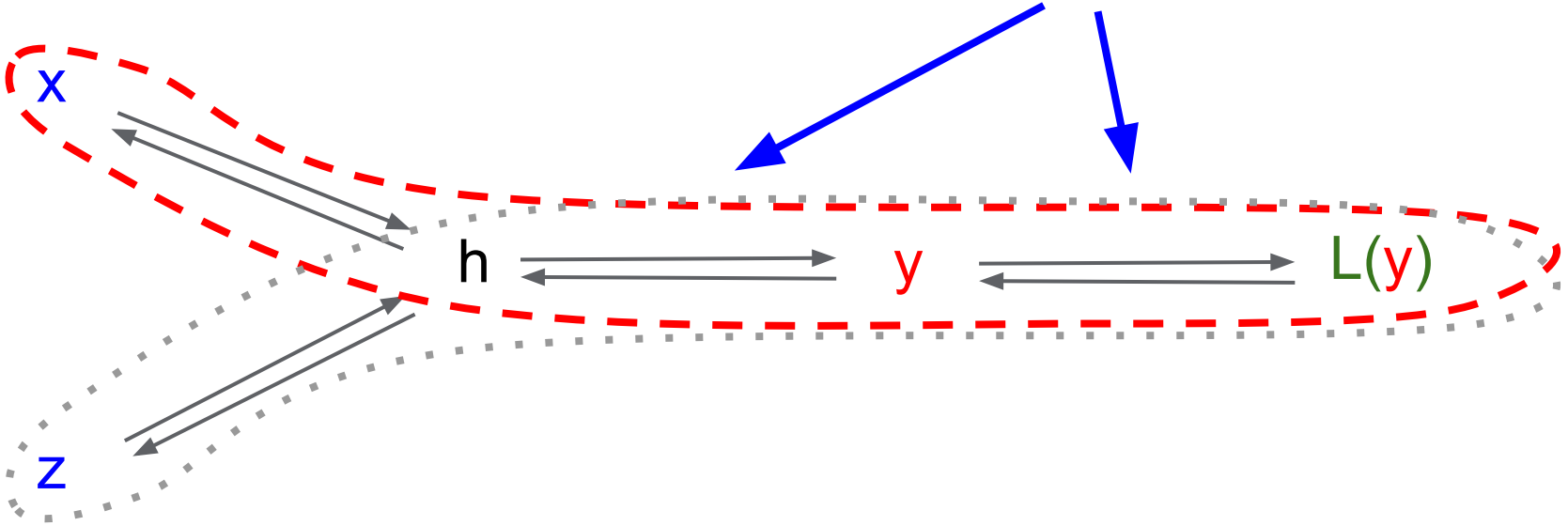
- Taking the gradient ∂x sent back from up the chain from you.
- Multiplying it by your local gradient ∂f(x) with respect to that quantity.

Back-Propagation using the Chain Rule



More Back-Propagation ‘Magic’

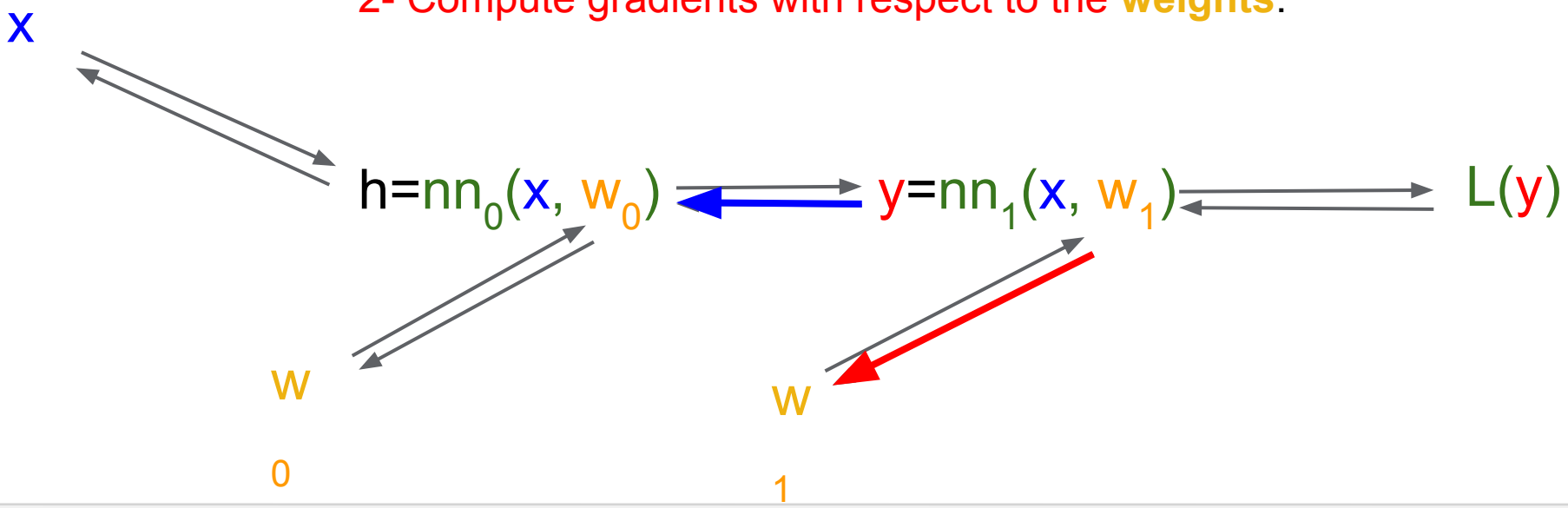
Computation along a directed graph can be **shared**:



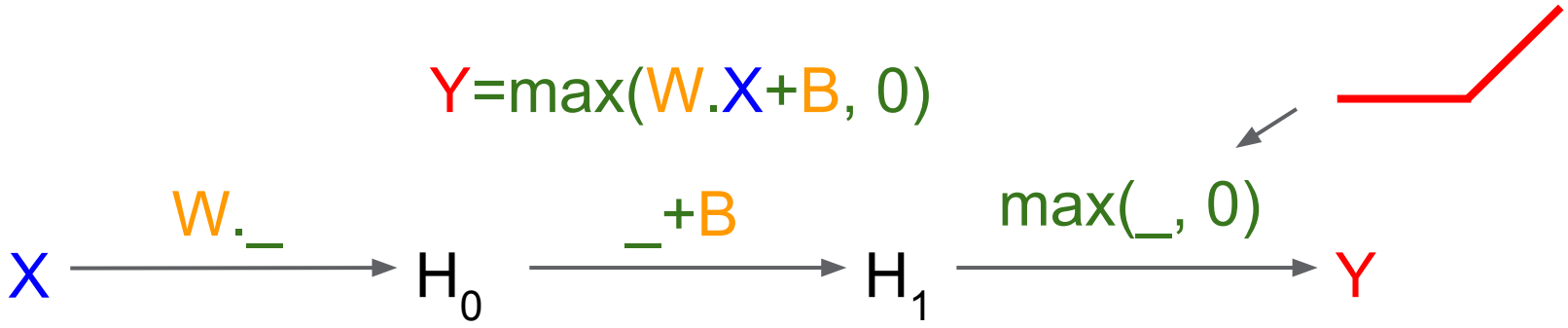
Sharing Gradient Computation via Back-Propagation

1- Compute gradients with respect to the **inputs**.

2- Compute gradients with respect to the **weights**.



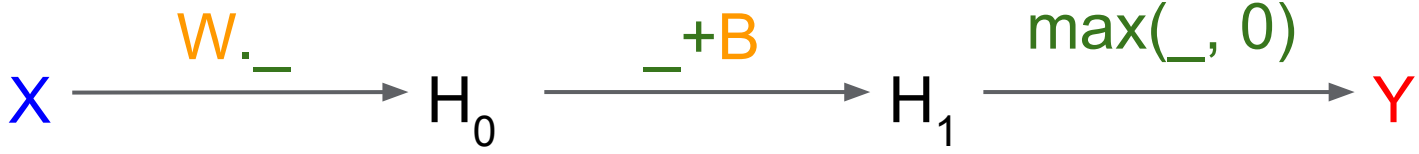
Example: 1-layer Neural Network



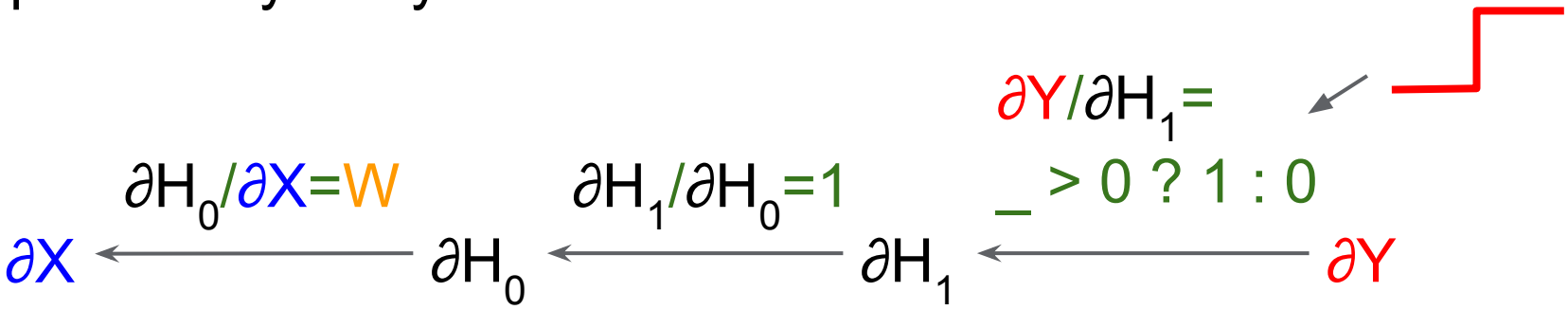
X and Y are matrices of dimension: # nodes X # examples.

Example: 1-layer Neural Network

$$Y=\max(W.X+B, 0)$$

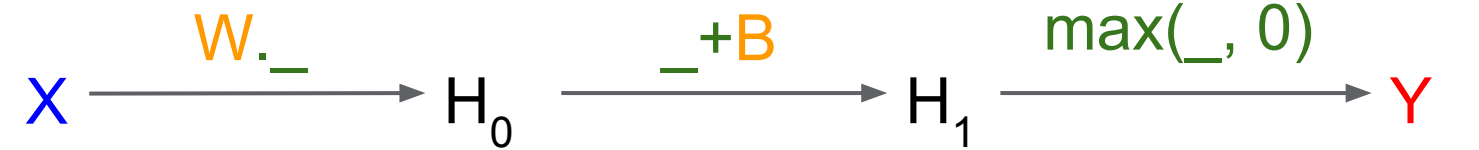


Compute analytically the Gradients:



Example: 1-layer Neural Network

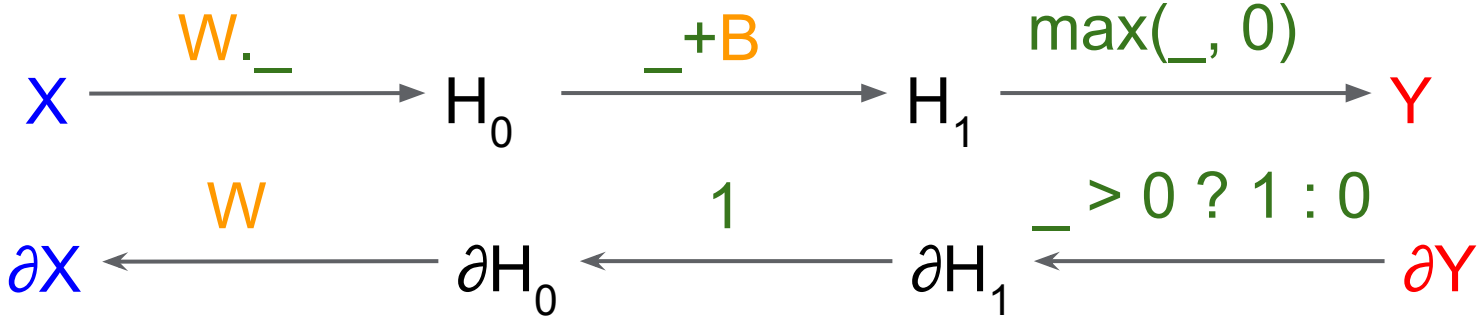
$$Y = \max(W.X + B, 0)$$



$$\partial X = W^T.\partial H_0 \qquad \partial H_0 = \partial H_1 \qquad \partial H_1 = H_1 > 0 \ ? \ \partial Y : 0$$

← Back-Propagate the Gradients: $\partial X = (\partial H_0 / \partial X).\partial H_0$, etc… ←

Example: 1-layer Neural Network



Compute the parameter Gradients:

$$H_0 = W.X \quad \rightarrow \quad \partial W = \partial H_0 . X^\top \qquad W \leftarrow W - \alpha \, \partial W$$

$$H_1 = H_0 + B \quad \rightarrow \quad \partial B = \partial H_1 . 1 \qquad B \leftarrow B - \alpha \, \partial B$$

What Happens at The Top of the Chain?

The Loss is a Very Complicated Function of the Weights

$$L(\bar{y}, nn(x,w))$$

1. ~~nn() is a very non-linear, non-convex function!~~
2. Depends on all the Inputs and Targets in the training set!

Two main tricks to simplify the problem:

1. ~~Back-Propagation~~
2. Stochastic Gradient Descent

Google

The Stats

Stochastic Gradient Descent

Loss: $\sum L(\overline{y}, y)$

- Instead of computing the true loss on all the data, we compute an estimate on a *very* small subset of the data (1 - 1024 examples).
- Terrible estimate. But we can afford to do it lots of times.
- And we have tricks to smoothe it
- If efficiency was linear in batch size, we would use batch = 1.

Stochastic Gradient Descent (SGD) Summarized

For batch in training set:

- For layer in network:
  - Compute (and store) output Activation H
- Compute Loss and Loss Gradient L, ∂Y

Forward pass

- For layer in network backwards:
  - Compute Gradient w.r.t. input Activation ∂H
  - Compute Gradient w.r.t. Weights ∂W

Backward pass

Update Weights: W ← W - α ∂W

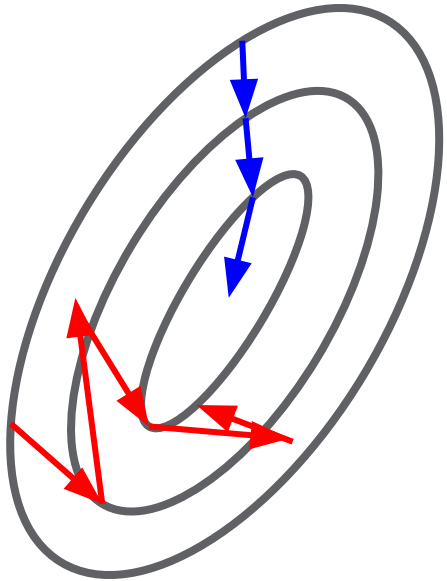
The Hacks

Stochastic Gradient Descent is a Terrible Optimizer

SGD is very **noisy**, and turns the gradient descent into a random walk over the loss.

But it's very **cheap**, and cheap is all we can afford at scale.

The hacks that follow all have one objective: **reducing the noise** in the gradient estimates while remaining very cheap to compute.



Primer on Getting Stochastic Gradient Descent to Work

Two strategies

1. Momentum + learning rate decay:
   Works best if you manage to get it to run.
2. AdaGrad:
   Works more often, not always gets you the best result.

Two more tricks:

1. Parameter averaging.
2. Gradient clipping.

Momentum

$$g' = \mu\, g + \partial_w L(w)$$
$$w' = w - \alpha\, g'$$

$$\mu = 0.9$$

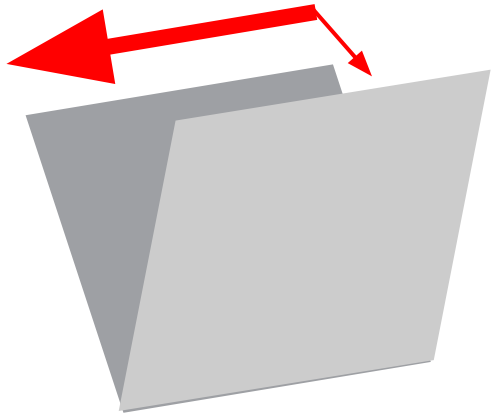
Learning Rate Decay

Think of the loss function as a fractal landscape:

Some structures / dimensions have widely differing scales.

Annealing the learning rate allows you to explore all scales:

$$\alpha = \alpha_0 e^{-\beta t}$$



AdaGrad

AdaGrad is a method for applying learning rate decay **adaptively**, **per-parameter**.

Very useful when one wants to do little hyperparameter tuning.

Adaptive subgradient methods for online learning and stochastic optimization.
John Duchi, Elad Hazan, and Yoram Singer. JMLR 2011

AdaGrad in Equations

Keep a history of the norm of the gradients:

$$n \leftarrow n + \partial_w L^2$$

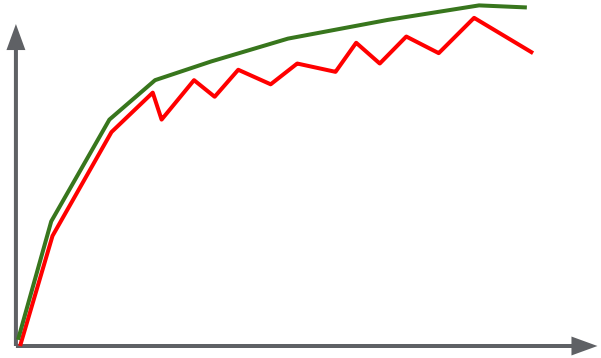
Use it to discount the learning rate:

$$w \leftarrow w - \frac{\alpha}{\sqrt{n}} \partial_w L$$

Parameter Averaging

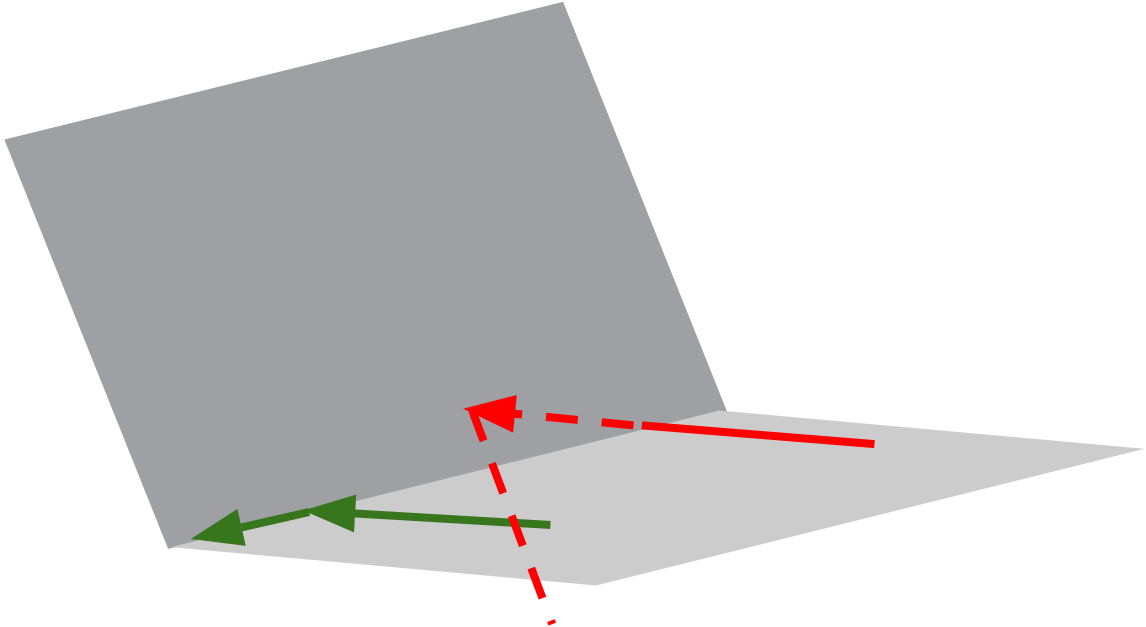
Keep a running average of your parameters over time.

Only use the averaged parameters at test time, not for training!



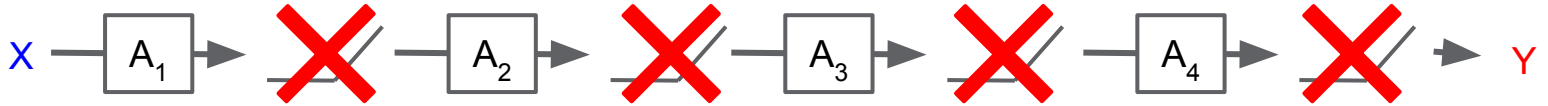
Gradient Clipping

Threshold gradient norms to protect against wall bouncing:



Weight Initialization

You want to keep your activations in a stable numerical regime: O(1.0).



$$|Y| \sim |A_4||A_3||A_2||A_1||X|$$

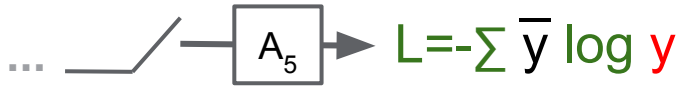
Initialize **weights** using $\mathcal{N}(0, \sigma)$ such that:

output activations ~ **input** activations.

Initialize **biases** to be **positive**: start in the linear regime of the ReLU.

Weight Initialization just before the Loss

Your loss is typically very dependent on the **scale** of the top activations:



Norm $|Y|$ of the activations ⇔ **peakiness** ($T°C$) of the probability distribution

Peakiness ⇔ magnitude of the gradients that are sent back:
big peaks == big errors == big gradients.

Weight Initialization just before the Loss

High Temperature:
soft distribution, classifier not certain, small gradients.

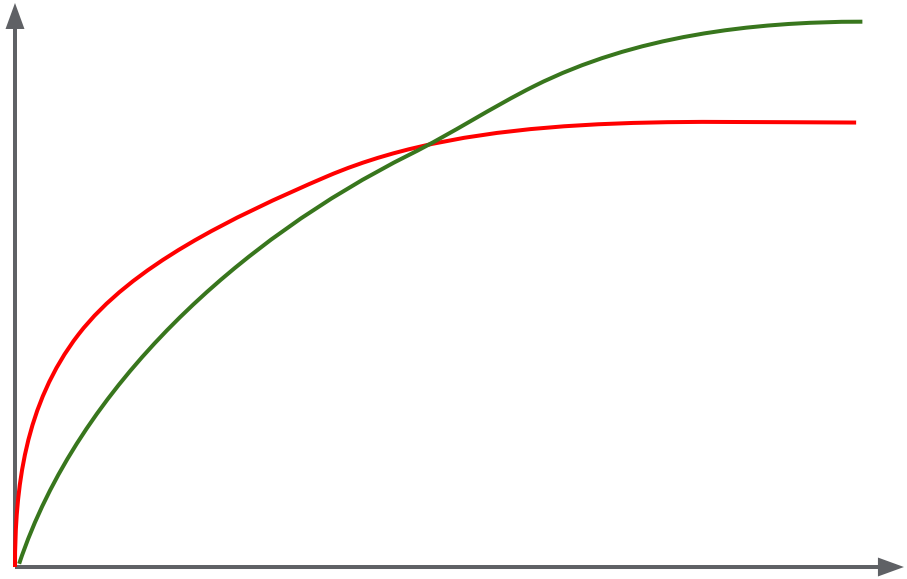
Low Temperature:
peaky distribution, classifier very (over?)confident, big gradients.

Key: Start with a very high temperature, **small weights** in your last layer.

They will anneal to peakier distribution as the classifier gets more confident.

First, lower your learning rate

Faster training ≠ Better training



Detour: Second Order (Quasi-Newton) Methods.

- A way to dramatically improve gradient descent efficiency per step:

$$W' = W - \alpha H^{-1} \partial W$$

- H is the Hessian (basically the second derivative) of the Loss.
- H^{-1} requires very *large batches* O(training set) to be estimated well.
- Huge literature on approximate 2nd order methods: L-BFGS, Conjugate Gradient, Hessian-free approximations.
- I have *never* seen them work better than SGD in practice.

The Computer Science

Parallelizing Stochastic Gradient Descent

For batch in training set:

- For layer in network:
  - Compute (and store) output Activation H
- Compute Loss and Loss Gradient L, ∂Y

Forward pass

- For layer in network backwards:
  - Compute Gradient w.r.t. input Activation ∂H
  - Compute Gradient w.r.t. Weights ∂W

Backward pass

Update Weights: W ← W - α ∂W

Oh Noes!

Serial Algorithm

For batch in training set:
    For layer in network:
        Compute (and store) output Activation H
    Compute Loss and Loss Gradien
    For layer in network backwards:
        Compute Gradient w.r.t. input
        Compute Gradient w.r.t. Weights ∂W
    Update Weights: W ← W - α ∂W

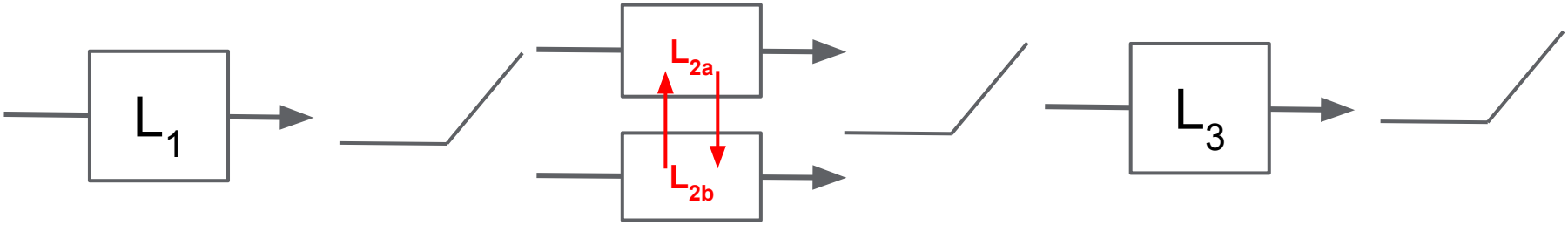
Lots of parameters in contention

Multiple Levels of Parallelism

Distribute the model, keep the data local: **Model Parallelism**

Distribute the data, keep the model local: **Data Parallelism**

Model Parallelism



Cut up layers, distribute them onto multiple cores / devices / machines.

Each cut adds several edges to your graph!

Unless you have shared memory, this means a lot more **memory** and **data transfer**.

Model Parallelism

On a single core: Instruction parallelism (SIMD, SIMT). Pretty much free.

Across cores: thread parallelism. Almost free, unless across sockets, in which case inter-socket bandwidth matters (QPI on Intel).

Across devices: for GPUs, often limited by PCIe bandwidth.

Across machines: limited by network bandwidth / latency.

Model Parallelism

Two key ideas when sizing a distributed system:

1- **Data reuse:** compute is limited by how much data can fit at any time on the lowest level cache (e.g. L1 cache on CPU). Try to maximally reuse the data in cache, or get more cache (i.e. more machines!).

2- **Overlap computation and data transfer:** in most systems compute and data transfer can happen completely in parallel. Hide the increase in data transfer latency by overlapping computation with it.

Synchronous Data Parallelism

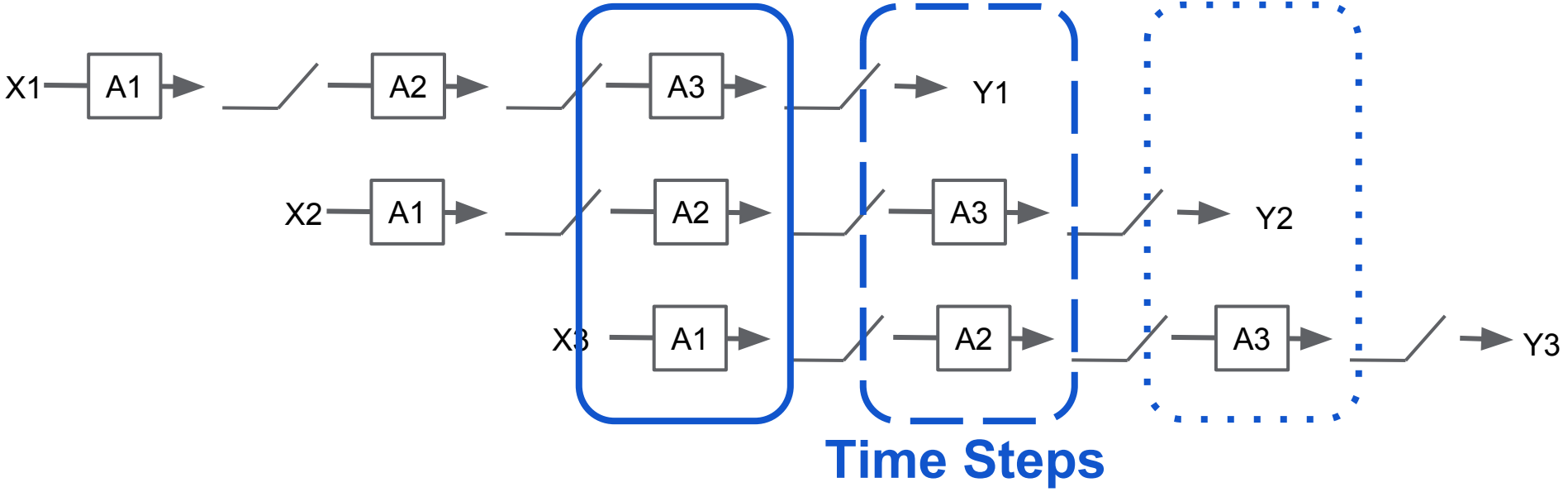
Run K batches in parallel and aggregate.

It's by far the **most popular** way to parallelize SGD today.

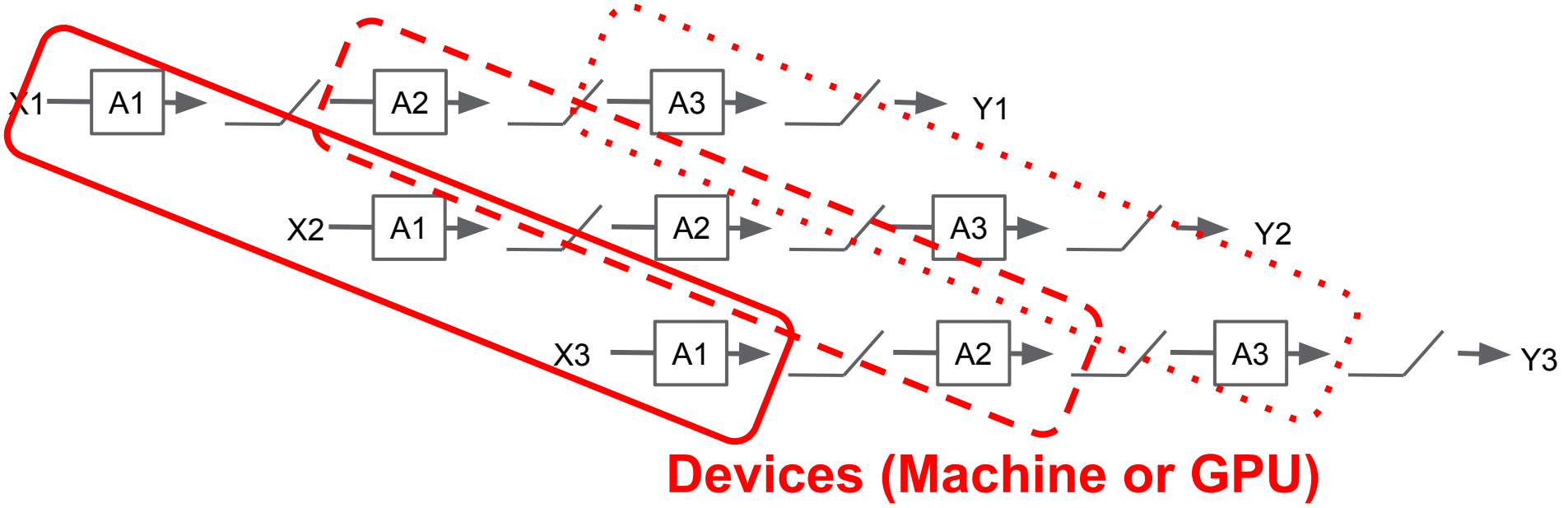
Limits:

- Per-example efficiency of gradient descent diminishes as the batch size increases.
- Cutting a batch smaller yield diminishing returns as matrix multiplies become less efficient.
- Cost of synchronization grows with K: need to wait for stragglers.

Asynchronous Data Parallelism - Pipelining



Asynchronous Data Parallelism - Pipelining



Asynchronous Data Parallelism - Pipelining

Pipelining changes the gradient updates:

$$W_{t+1} \leftarrow W_t - \alpha \, \partial W_{t-k}$$

where k is the depth of the pipeline (# of layers or less).
Stale gradients from k steps ago are less efficient per step.
Often means the learning rate needs to be reduced.

Limited by **depth of pipeline** and **balancing compute** between layers.

Fully Asynchronous Data Parallelism

Run N training loops in parallel.
Share the weights between training loops.

$$W_{t+1} \leftarrow W_t - \alpha \, \partial W_{t-k}$$

k is now O(N), potentially very large.
k is effectively unbounded if one training loop is slower than the others.

Equivalent to running N batches in parallel, but forgetting to wait for the workers to be done to aggregate the partial sums.

Fully Asynchronous Data Parallelism

Has some nice theoretical guarantees:

> **Hogwild! A Lock-Free Approach to Parallelizing Stochastic Gradient Descent.** Recht et al, NIPS'11

Works well up to ~50 replicas of the model.

Strongly diminishing returns per replica.

Great if you want speed and don't mind spending resources to get it.

Requires some care in implementation.

Data and Model Parallelism Tradeoffs

Model Parallelism means you need to exchange activations between workers:

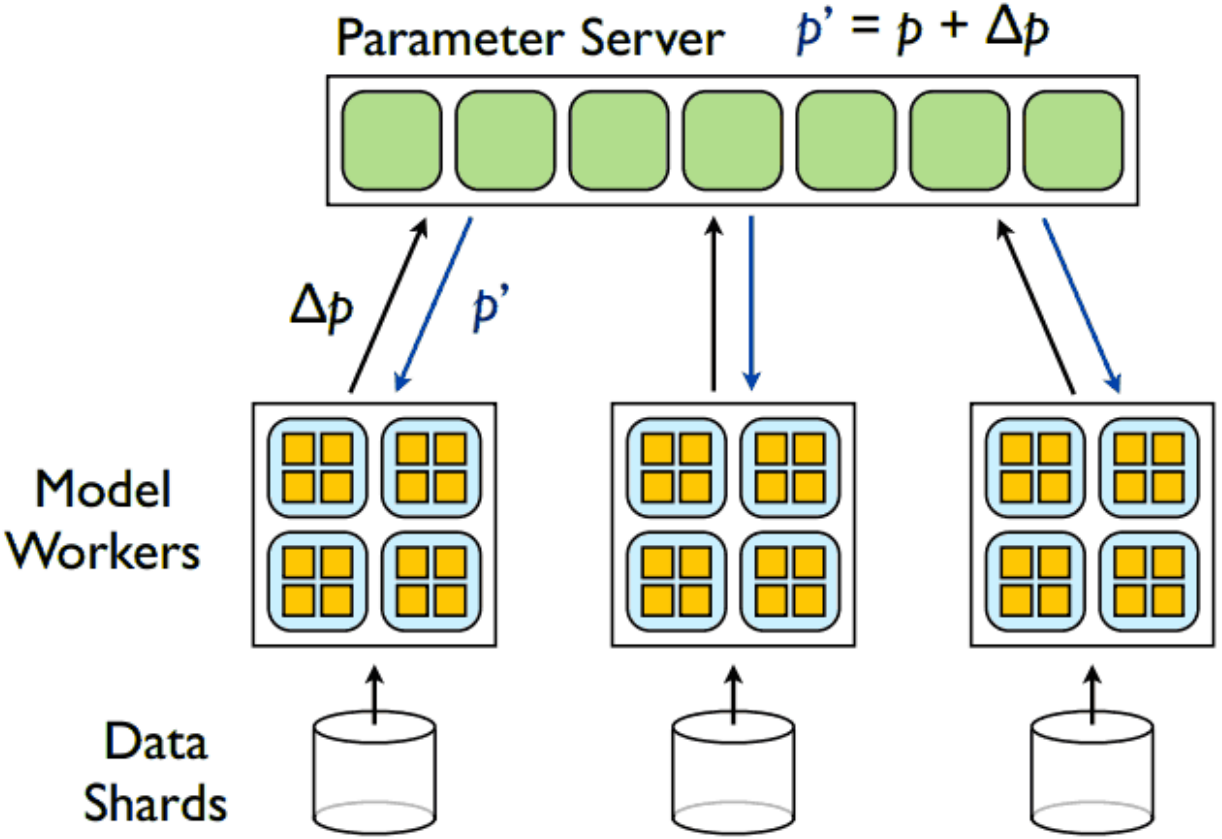
O(batch size X # network edges) values sent at every step.

Data Parallelism means you need to exchange parameters between workers:

O(# weights) values sent at every step.

DistBelief

Joint **Data Parallelism** and **Model Parallelism**.



[Large Scale Distributed Deep Networks]
Jeff Dean et al., NIPS'12

Next up!

We'll talk about the thorny problem of **regularization**.

And discuss the various **models** you might encounter in the wild.

See you soon!

Session II

Regularization

Regularization

There is an optimal size for any machine learning system:

- Too small (Underfitting): not enough parameters to express the complexity of the data.
- Too big (Overfitting): too many degrees of freedom. The model will attempt to explain every little detail of the training data and will fail to generalize.

Regularization

Problem: training a model that is just the right size impossible:

- No idea a priori what the ‘right size’ is.
- Training a model that just fits is very hard from an optimization standpoint:

‘fitting into skinny jeans’ problem.

Solution: train a model that is **way too big**, but nudge the parameters towards a more parsimonious representation.

Regularization Techniques

L_2 Regularization, a.k.a. **Weight Decay**:

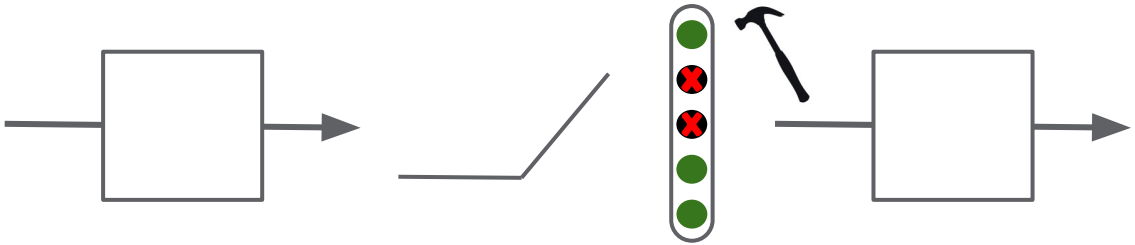
- Penalize large weights.
- Achieved using a new term to the Loss:

$$L(\overline{y}, y) + \varepsilon|w|^2$$

- ε is a new hyperparameter (global or per-layer).

Regularization Techniques: **Dropout**

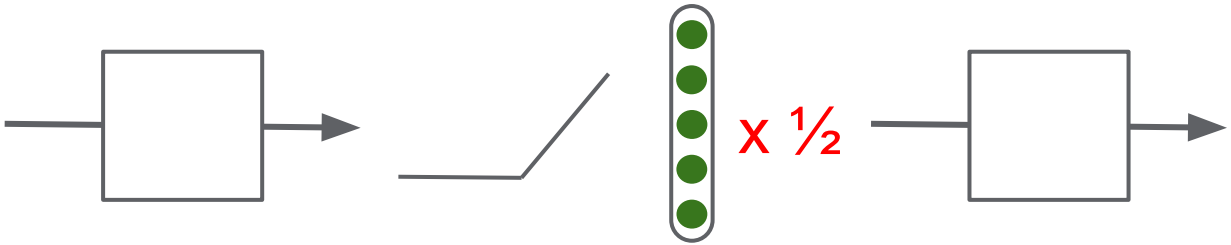
- Set 50%+ of the activations to zero randomly.



- Force other parts of the network to learn redundant representations.
- Sounds crazy, but works great.
- Complementary to L_2.

Regularization Techniques: **Dropout**

- At test time, don't drop anything, but multiply the activations by ½:



- Can be combined to great effect with the **max()** non-linearity.
  See **Maxout Networks**, Goodfellow & al.

Prototypical Models

DNNs
Embeddings
CNNs

RNNs
Generative Models

The ‘Simple’ Deep Neural Net

Try Logistic Regression, Random Forests or Gradient Boosting first!

Debug your data / problem setup on **simple models**, a **small dataset**, then scale up.

For a new problem, in the absence of any particular structure, a 2-3 layer, 128-1024 nodes / layer model is a reasonable starting point:

Models for Text

Embeddings

Handling discrete, categorical input, in particular **words**!

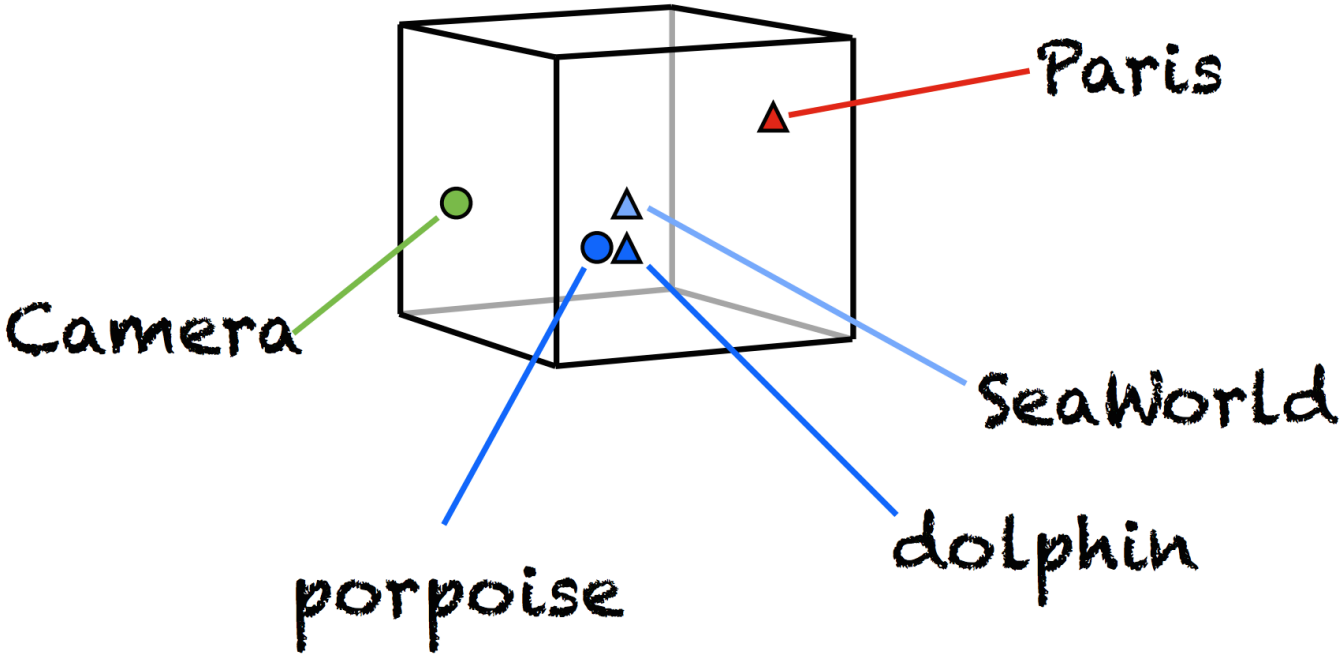
Usual representation: 1-hot encoding of position on the dictionary

[0 0 0 0 0 **1** 0 0 0 0 0 0 0]

Problem: mapping that vector to a dense layer in a neural network means a very large matrix whose columns (often called **embeddings**) are only exercised (hence trained) when that word is seen.

Rare words can mean very poor embeddings.

What Do We Want in a Good Embedding?



Similar Words Occur in Similar Contexts

Instead of training embeddings on the supervised task at hand, train them first to represent **semantic similarity** using unsupervised training on a large text corpus.

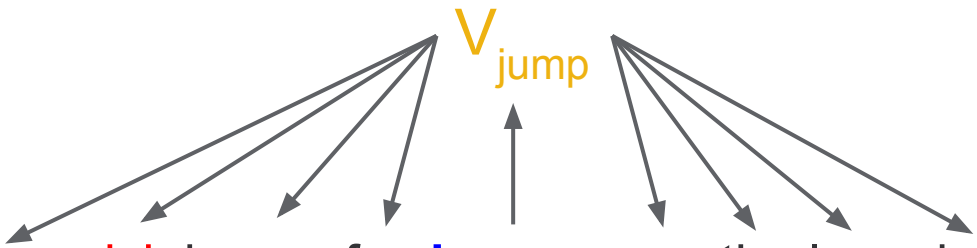
A couple of common approaches:

- Continuous skip-gram (**word2vec)**: predict surrounding words given a word.
- Continuous Bag of Words (**CBOW**): predict a word given surrounding words.

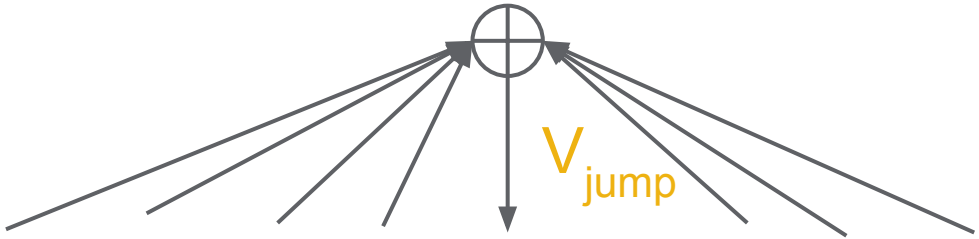
Efficient Estimation of Word Representations in Vector Space, ICLR'13.

Distributed Representations of Words and Phrases and their Compositionality, NIPS'13.

Example of word2vec and CBOW



word2vec: “The quick brown fox **jumps** over the lazy dog”



CBOW: “The quick brown fox **jumps** over the lazy dog”

Word Embeddings

Training a **very simple model** on **lots of text** mitigates the rare word problem.

The spaces learned have very good **syntactic** and **semantic** clustering.

They also have interesting local ‘algebraic structure’:

$$V_{king} - V_{man} + V_{woman} \sim V_{queen}$$

Interesting applications to **zero-shot** learning.

Sentence, Query, Paragraph, Document Embeddings

What if your ‘dictionary’ is extremely large or infinite?

Finding good embeddings for large bodies of text is a **very active area of research**. (rel: topic modeling, paraphrasing, document understanding)

Some simple approaches:

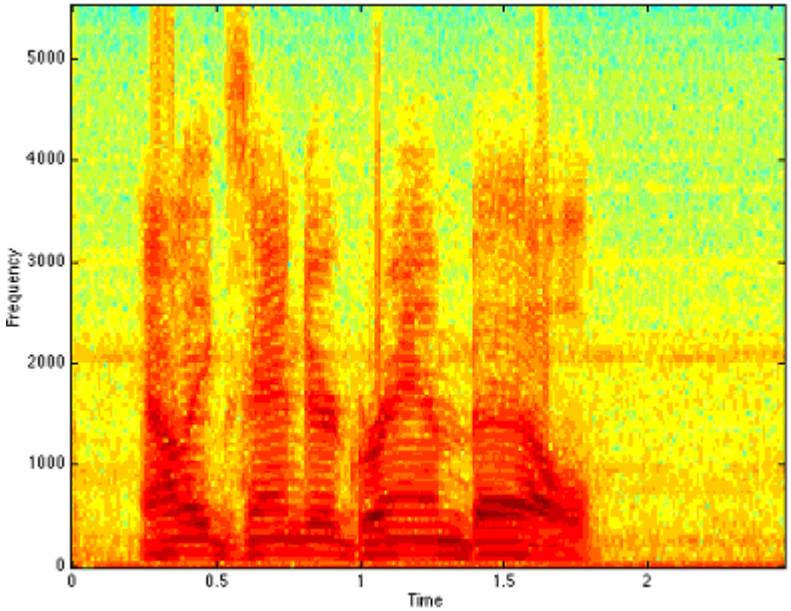
- For short sequences, **pooling** over word embeddings is the first recourse.
- For long sequences, use your favorite brand of **topic model**, or run a **recurrent model** over the sequence and use a hidden layer of the network as an embedding.

Model for Perception

Google

Common Statistical Invariants

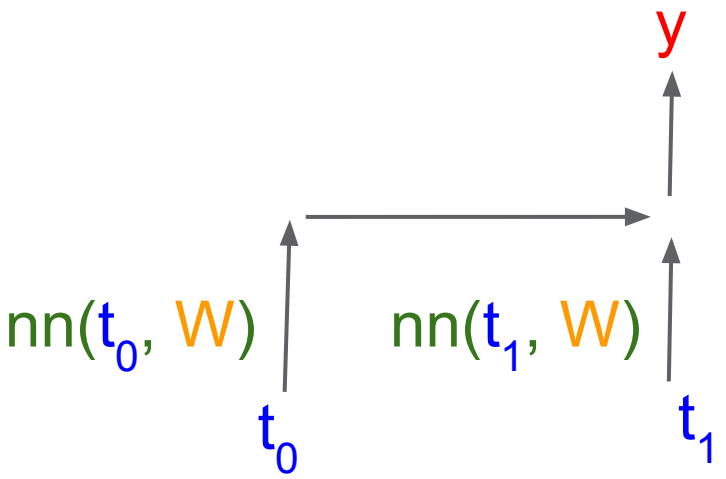
Across Time



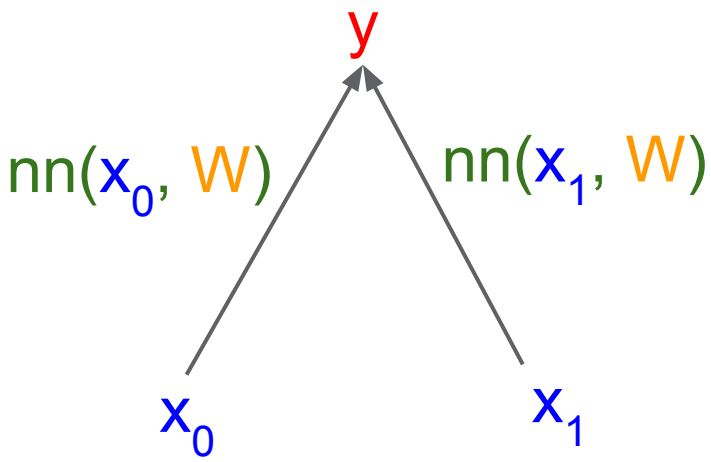
Across Space

Expressing Invariants: Weight Tying

Recurrent Neural Network



Convolutional Network



Good news: Backprop ‘just works’: simply add up all the gradients.

Models for Images

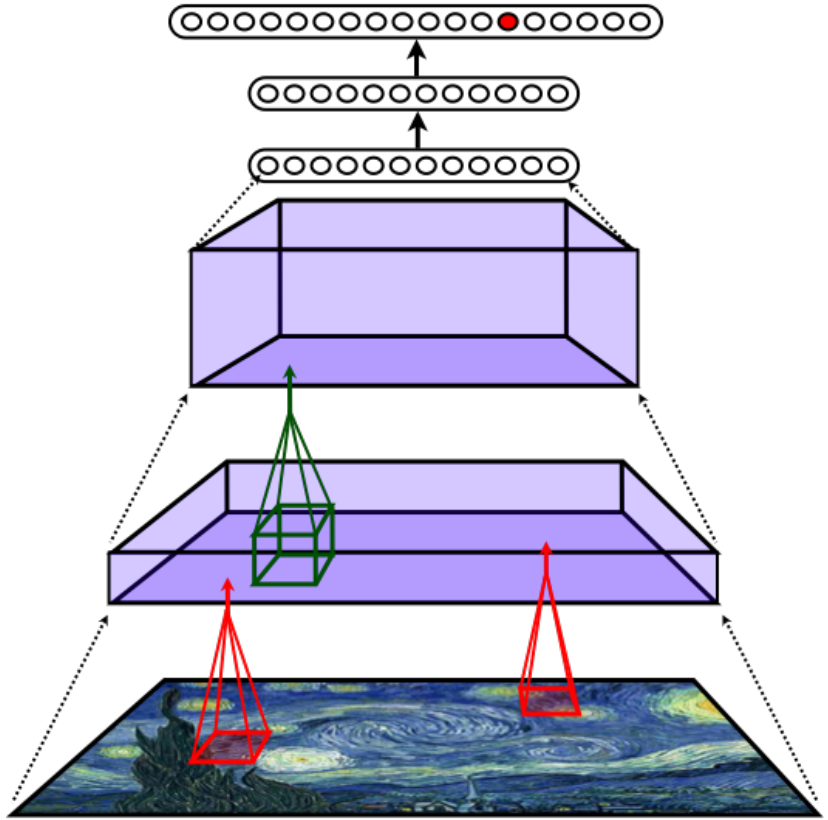
Convolutional Networks

Spatially tied deep neural networks.

State-of-the-art in visual recognition and detection / localization tasks.

One new challenge: images are large and highly redundant.

Need to introduce new types of nonlinearities which aggregate / decimate their inputs.

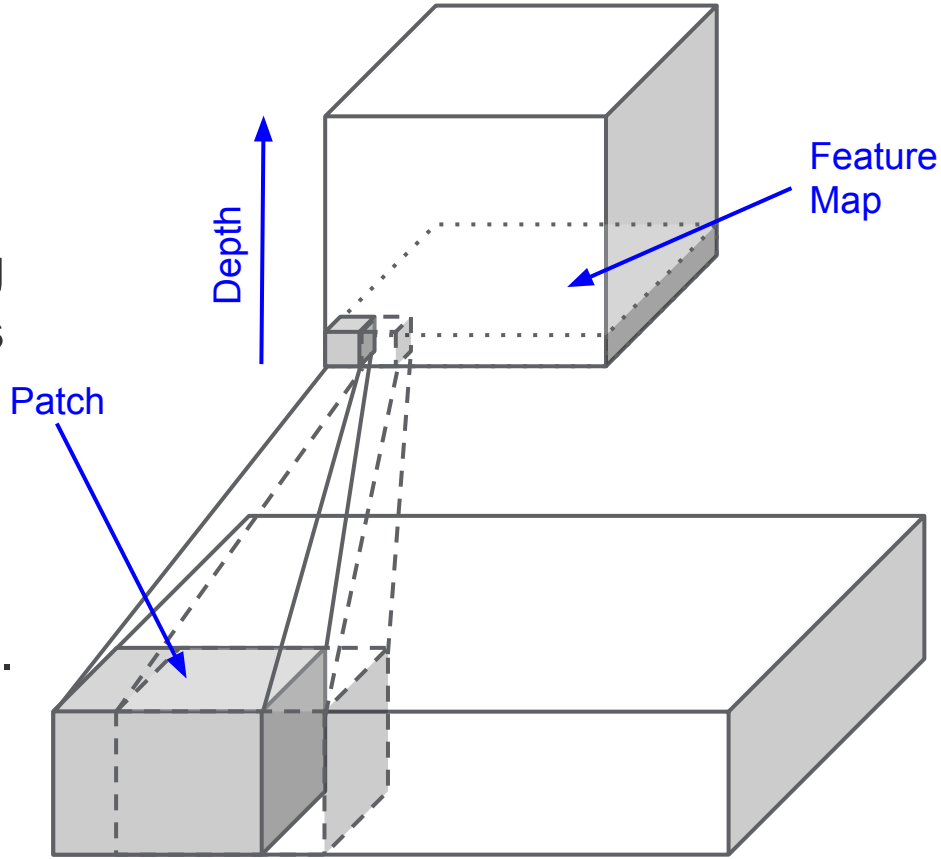


Convolutions

Lots of jargon: matrix multiplies applied over **patches** as a sliding window, producing **feature maps** of a certain **depth**.

Express spatial invariance by sharing weights across spatial dimensions, but not across depth.

Lots of implementation details related to stride and padding.

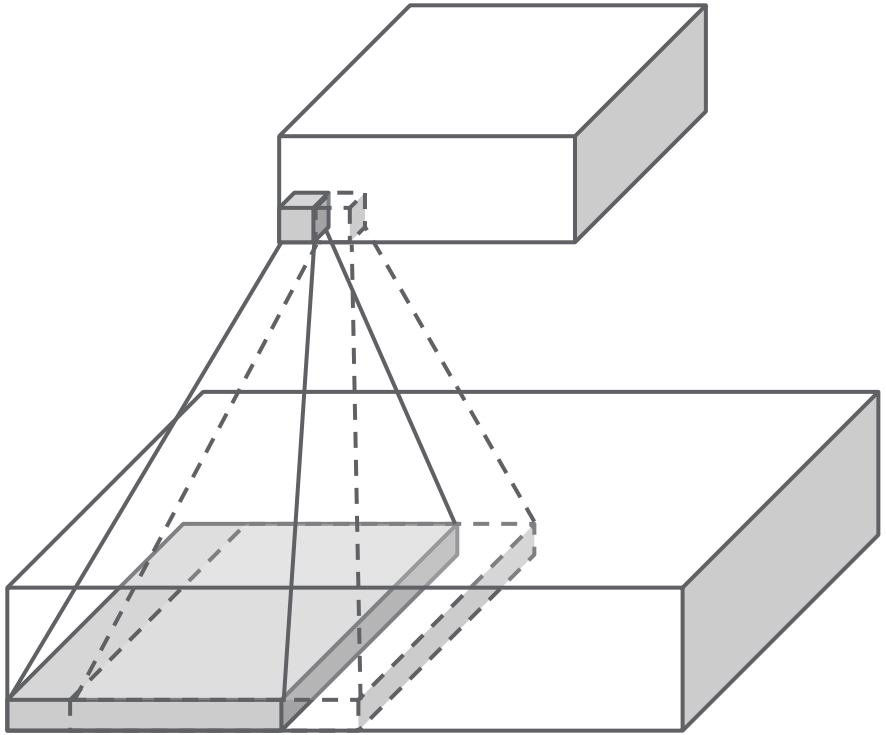


Non-linearities

Convolutional networks use the same types of pointwise non-linearities (ReLU).

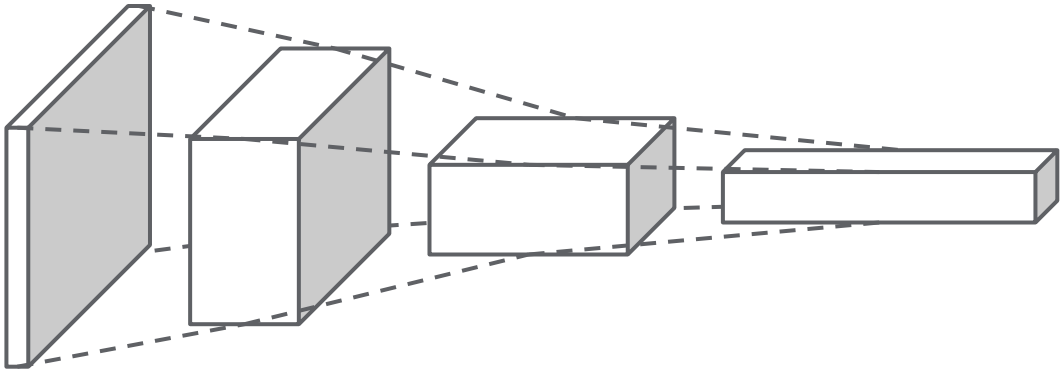
In addition, **spatial pooling** is often reduced to downsample the feature maps:

- max
- average
- L_2



Convolutional Networks

Stacking convolutions and pooling in a way that reduces the spatial extent while increasing the ‘depth’ of the representation proportionally is a good strategy to build a good convolutional network:

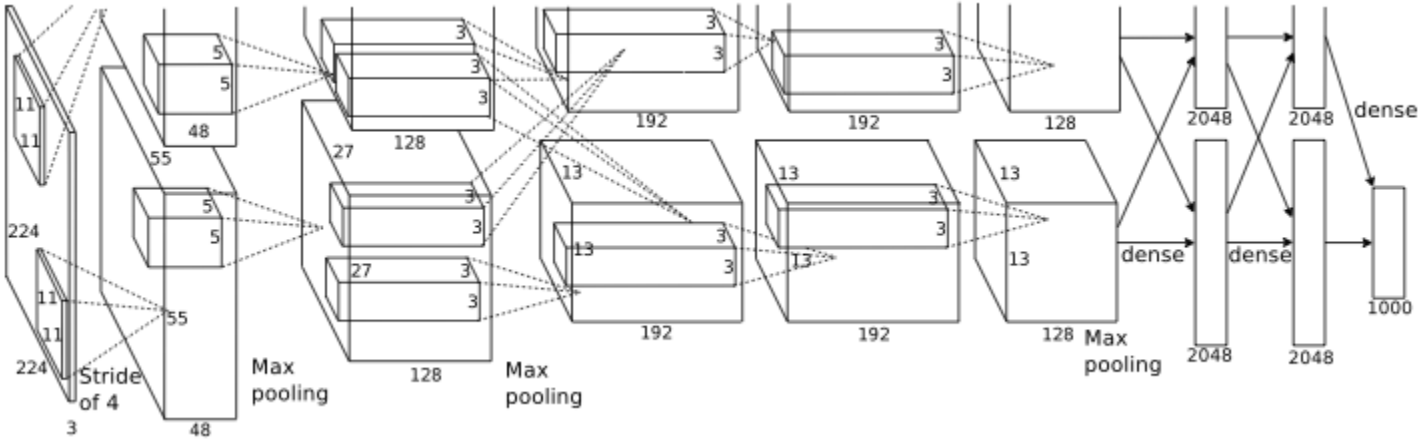


256x256 RGB → 128x128x16 → 64x64x64 → 32x32x256 …

Google

Convolutional Networks

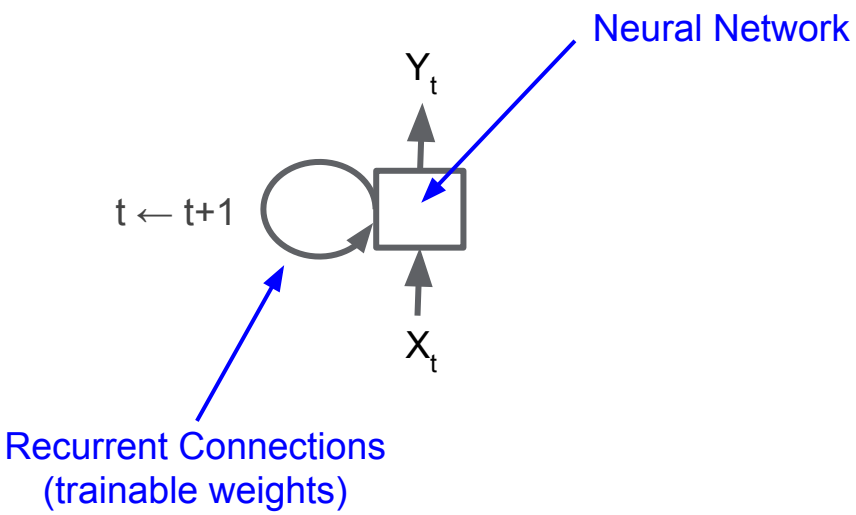
Example of AlexNet, winning ImageNet challenge entry in 2012:



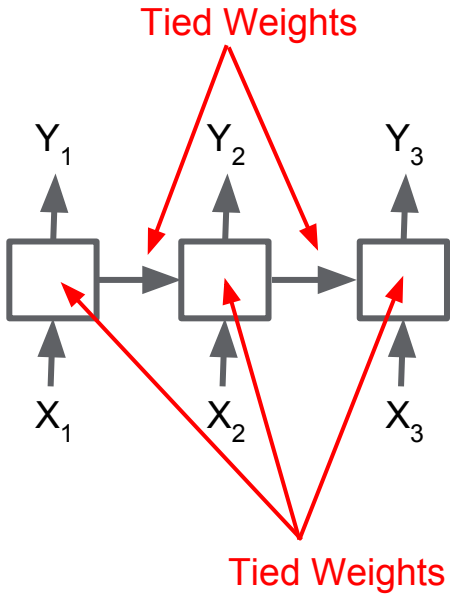
Models for Time Series

Recurrent Neural Networks

Compact View



Unrolled View



Recurrent Neural Networks

Can be implemented via explicit unrolling or dynamically by keeping state across invocations, or a combination of both.

Unrolling is conceptually simpler, but imposes a fixed sequence length.

RNNs only have one problem: **they mostly don't work!**

Very difficult to train for more than a few timesteps: numerically unstable gradients (vanishing / exploding).

Thankfully, **LSTMs**...

LSTMs: Long Short-Term Memory Networks

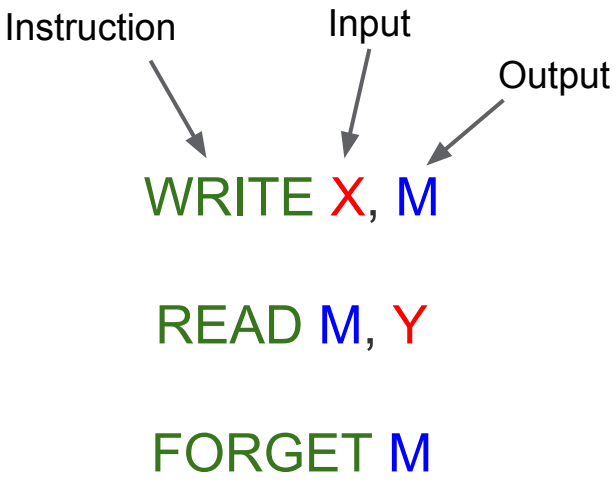
Took a long time to be recognized as ‘RNNs done right’:

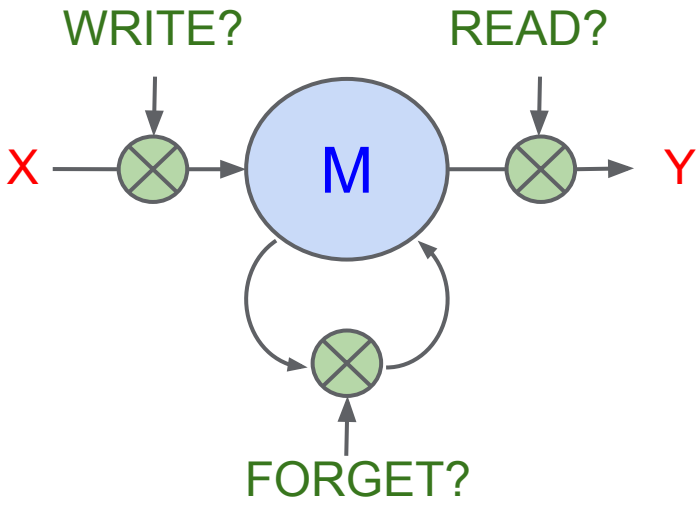
- Terrible name :)
- Look like a horribly over-engineered solution to the problem.

But:

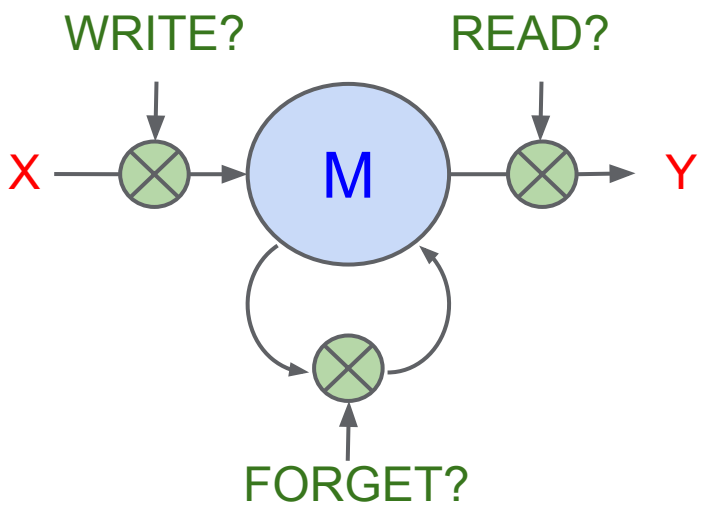
- Very effective at modeling long-term dependencies.
- Very sound theoretical and practical justifications.
- A central inspiration behind lots of recent work on using deep learning to learn complex programs:
  Memory Networks, *Neural Turing Machines*.

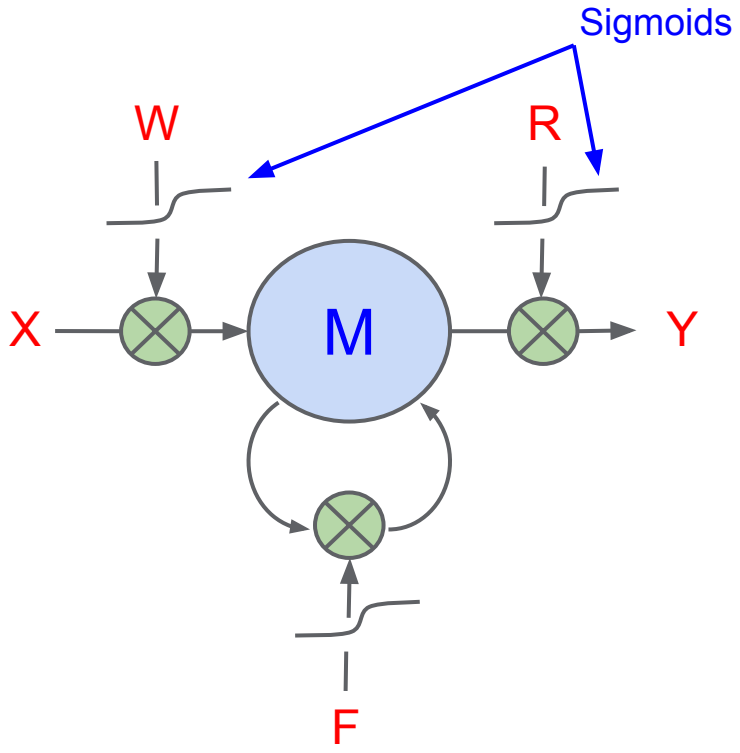
A Simple Model of Memory





Key Idea: Make Your Program Differentiable





LSTM Cells as replacement for Recurrent Connections

Recurrent connections in a RNN can be replaced by a set of LSTM cells that map inputs X, R, W, F to output Y.

R, W, and F are ‘control’ connections that affect the state of the memory through a **sigmoidal [0, 1] multiplicative gate**.

Gating behavior makes it possible for the memory cell to **retain information longer** and **discard it quickly**, while keeping the whole machine **continuous and differentiable**.

This translates into much better stability in training and modeling of much longer-range interactions compared to a RNN.

Unsupervised Learning

Generative Models and Unsupervised Learning

Amount of **unlabeled** data >> Amount of **labeled** data

Unsupervised / generative learning was once hoped to be a central appeal of deep learning.

Deep models learned to detect cats in YouTube videos without supervision! Surely they can learn anything?

Building High-level Features Using Large Scale Unsupervised Learning

Quoc V. Le et al., ICML'12

Unsupervised Learning

Likely the **biggest disappointment** in deep learning so far :(

Only real success is language models and **word embeddings**, although these leverage context as a supervised signal.

For any large task, even modest amounts of supervised data typically outperform unsupervised models.

Whither Unsupervised Learning?

Two trends to blame:

- **Dropout** made it possible to learn much bigger models without overfitting.
- **Transfer Learning** works amazingly well in practice:

It is often better to initialize your model from a **supervised model trained on a different task**, than to use unlabeled data matching your task.

Unsupervised Learning: New Approaches!

Good news: research on the topic has picked up recently.

General themes:

Variational Auto-Encoders

Adversarial Learning

It remains to be seen whether they can scale.

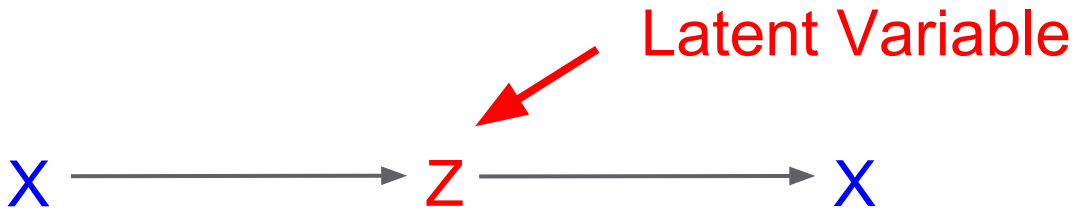
Generative Models

General idea: the **data** is the **label**: $\overline{y} = x$

Problem: There are many ways to map X to X in degenerate or trivial ways!

What would an ‘interesting’ mapping look like?

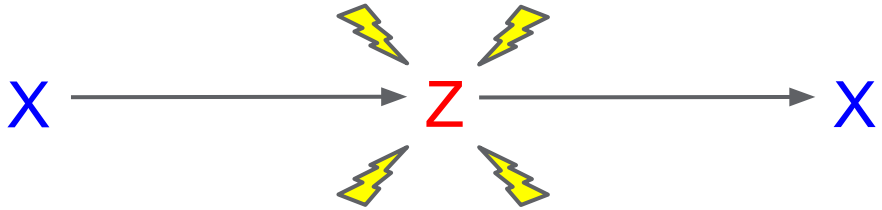
Generative Models



An interesting mapping could be:

1- One that **compresses** the data very well: Z << X
2- One that causes **semantically similar** X to have nearby Z.
3- One that has a very **simple distribution** (Gaussian, Binomial)
Makes it possible to generate sample X's from Z's.

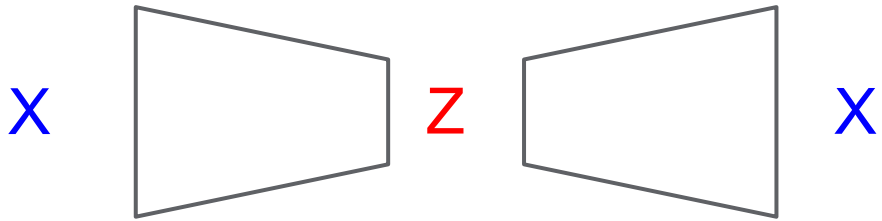
Making Generative Models Non-Trivial



Make it hard for the model to do do its job, by introducing **bottlenecks, regularizers**, **noise**, **stochasticity** or **adversarial training**.

Making Generative Models Non-Trivial

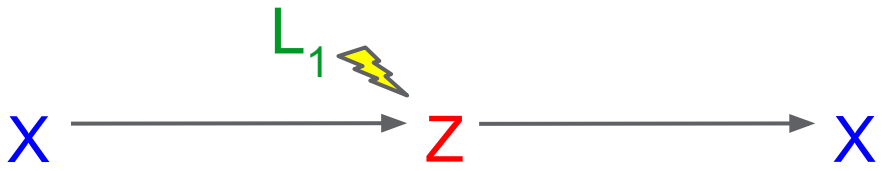
Bottlenecks:



Very common idea that's been explored at length (and reinvented) in many fields: e.g. SVD, PCA, LSA, LDA.

Making Generative Models Non-Trivial

Regularizers: Sparse Autoencoders

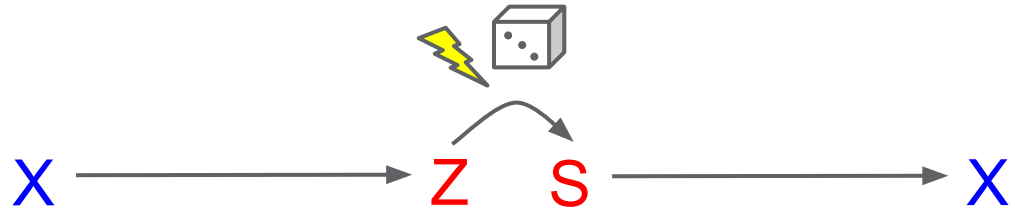


Noise: Denoising Autoencoders

The ‘ancestor’ of **dropout**.

Add **noise** to the input, force the autoencoder to reconstruct the clean signal.
Dropout is one such noise source.

Stochasticity and Generative Models

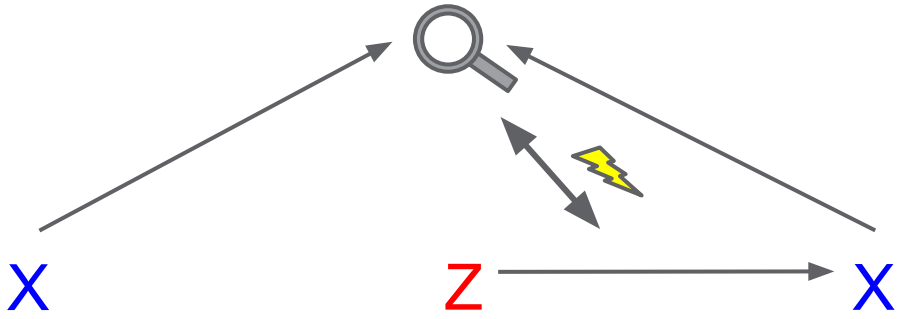


Consider Z as the parameters of a Gaussian. Sample S from it.

Very active area of research, spurred by the concept of
Variational Auto-Encoders

Auto-encoding Variational Bayes. D.P. Kingma & M. Welling, ICLR' 14.

Adversarial Training



Train a network  to try and distinguish between the real and generated X. Pit it against the “generator”, and make them compete!

Generative Adversarial Networks, Goodfellow et al, NIPS’14

Detour: Undirected Models

Model p(X, Z) instead of p(Z|X) and p(X|Z)

Boltzmann Machines and **Deep Belief Networks**.

Losing popularity. Very hard to train.

Batch Normalization

Better and Faster Way to Train Convolutional Networks

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

Sergey Ioffe, Christian Szegedy, ICML'15

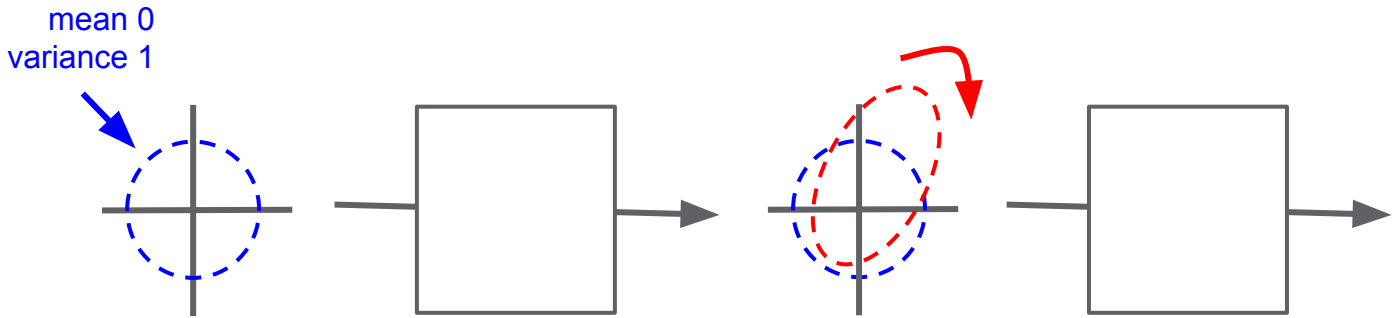
10x speedup in training, 2% improvement in performance on ImageNet!

Beautifully simple method attacking the core of what makes deep networks difficult to train.

The Covariate Shift Problem

SGD is not scale-free: most efficient on whitened data.

This is true for the inputs, but also for every layer up the stack:



Problem: the distribution of activations changes over time!

The Covariate Shift Problem

SGD needs to do two things for each layer:

1) Update its parameters to improve the objective.
2) Track the distributions of its **inputs**.

Can we eliminate or at least control 2)?

Solutions

Idea #1: Whiten the activations at each layer.
Problem: very expensive, high-dimensional covariance matrix.

Idea #2: Ok, let's just subtract the mean, and divide by the variance.
Problem: leads to degenerate gradients!

Idea #3: Let's use a noisy, local estimate of the mean and variance, e. g. one computed per mini-batch.
Problem: still strictly less powerful representationally: all filters in the layer are constrained to the same dynamic range.

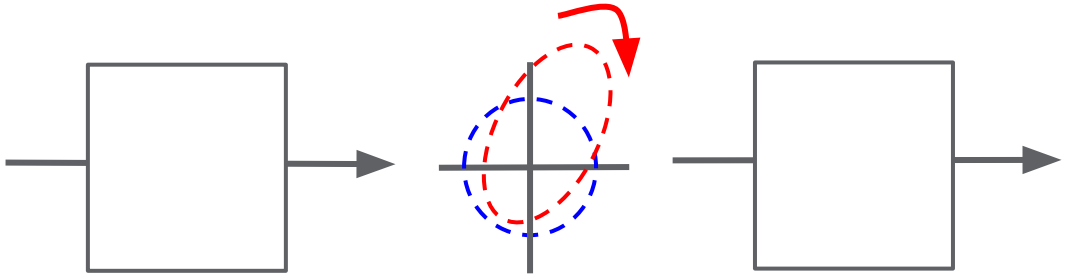
Solutions

Idea #4: Add a learned affine transform per activation to rescale the inputs.
Doesn't that defeat the purpose? No! Tightly bounds the rate of change of the input distribution: a few linear weights instead of many, many nonlinear factors.
Problem: What happens at test time, when there is no such thing as a mini-batch to normalize over?

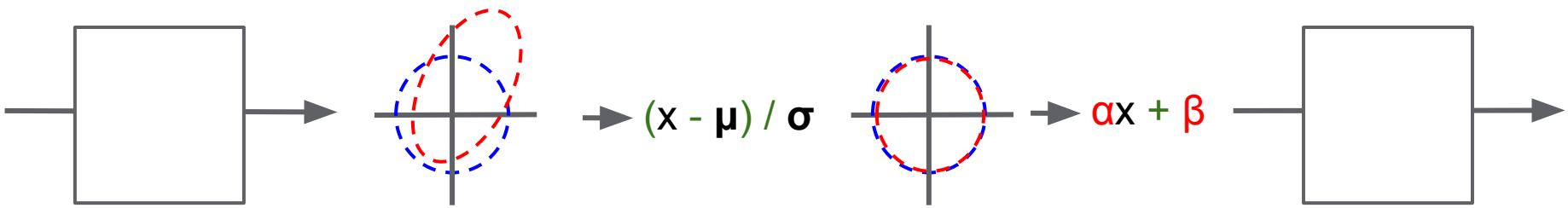
Idea #5: Replace the mini-batch mean and variance by the global mean and variance over the training set, at test time only.
Problem: That sounds really crazy…

Batch Normalization

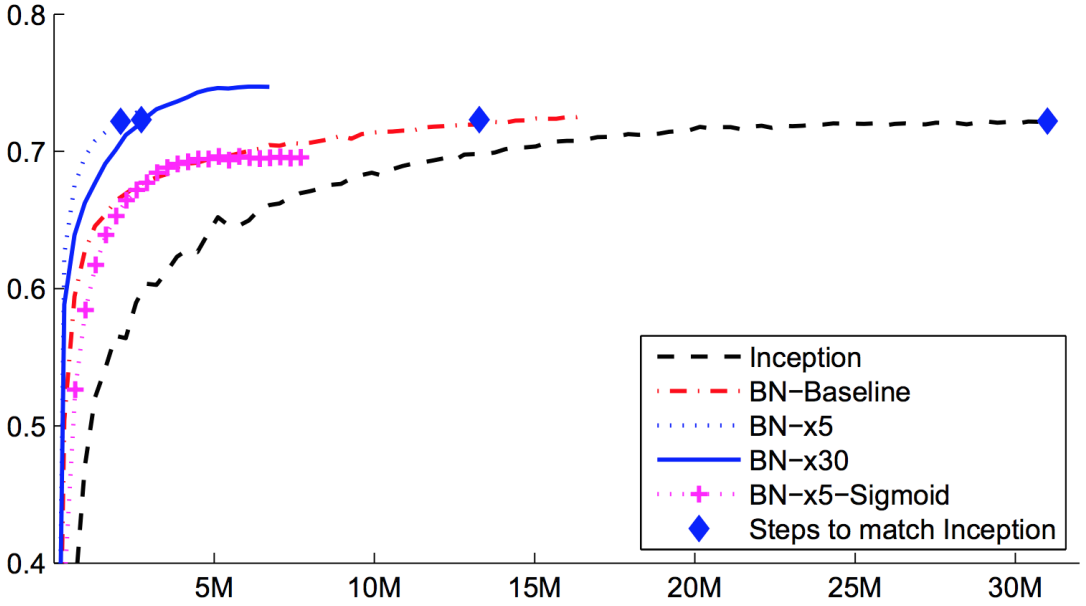
Before:



After:



Results



Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift - Sergey Ioffe, Christian Szegedy

Next up!

Object Recognition

Speech Recognition

Machine Translation

Multimodal Learning

Neural Turing Machines

Reinforcement Learning

Robots!

Art!

A close up of a child holding a stuffed animal

Session III

Hot Topics In Deep Learning

Speech Recognition
Object Recognition
Machine Translation
Image Captioning
Memory and Computation

Hot Topics in Deep Learning: Speech Recognition

Lead the Deep Learning revival by a few years:

Deep Belief Networks for phone recognition
Abdel-rahman Mohamed, George Dahl, and Geoffrey Hinton, **NIPS'09**

Very large improvements in acoustic modeling performance.

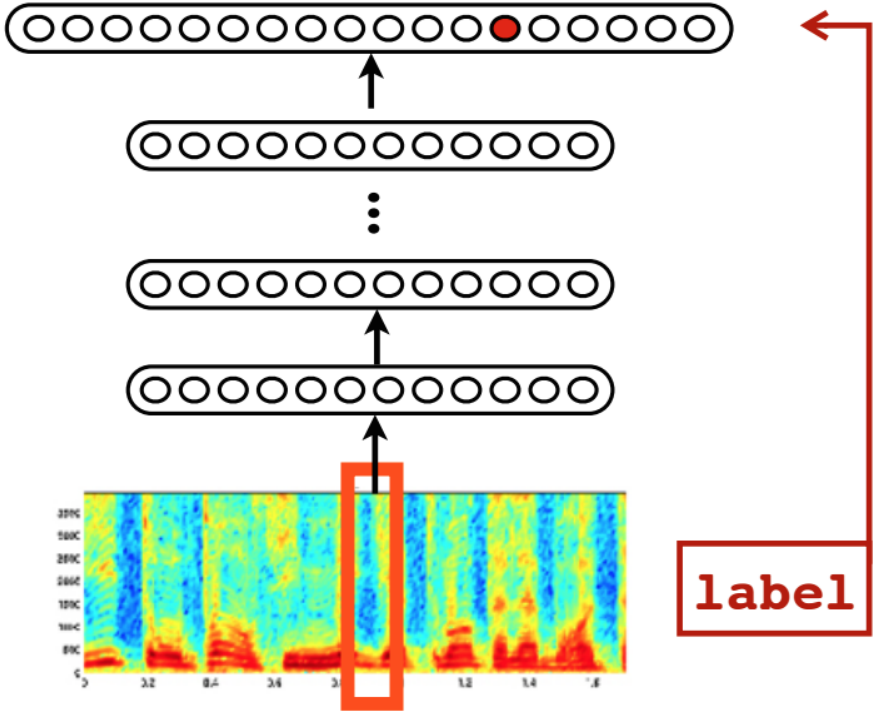
A turning point: speech recognition went from "it mostly doesn't work" to "it mostly works" in the public's perception.

In The Beginning

Model speech frame-by-frame, independently.

Fully-connected networks.

Deep Neural Networks for Acoustic Modeling in Speech Recognition
Hinton et al. IEEE Signal Processing Magazine, 2012



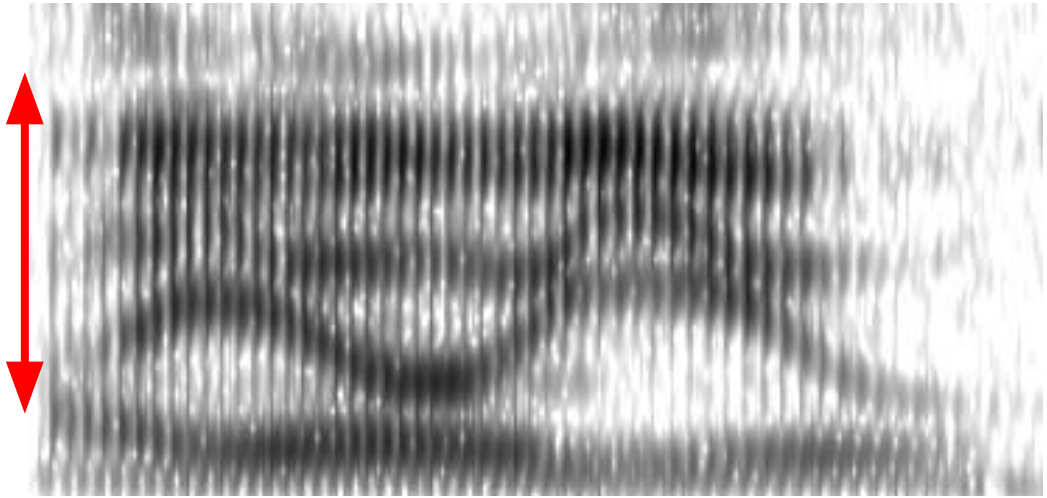
Speech is very structured.

Vertical shifts of the voiced segments are essentially pitch variations.

Irrelevant to non-tonal languages, and surprisingly weak cues for tonal languages.

Model translation invariance?

“I owe you”



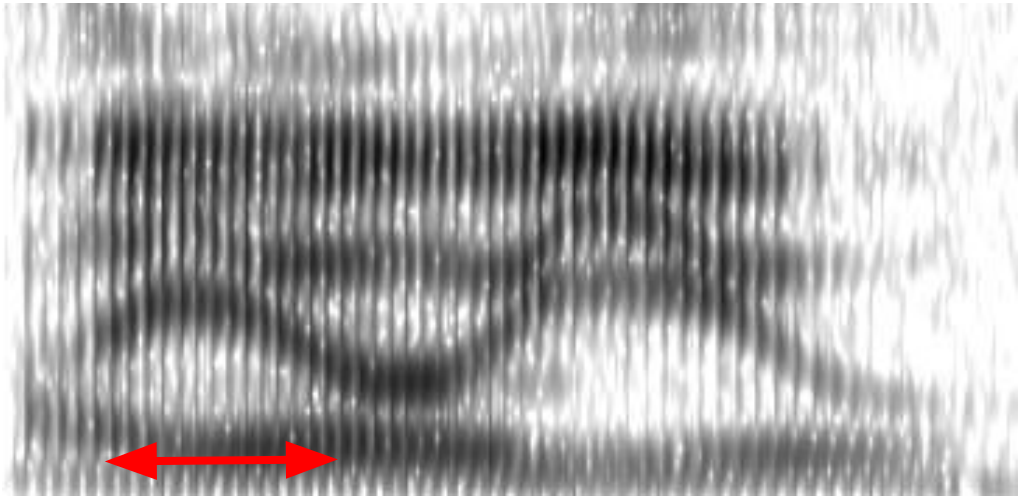
Speech is very structured.

Horizontal dilation is a change of speaking rate.

Very badly modeled by conventional Hidden Markov Models.

Model time dilation invariance?

“I owe you”



Google

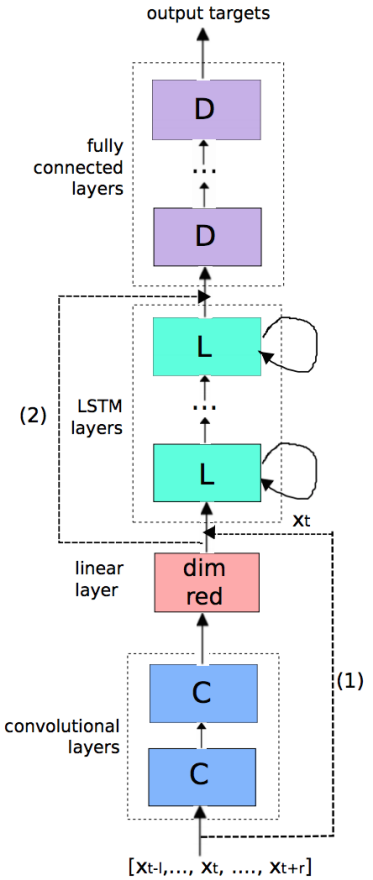
CLDNNs

Model frequency invariance using 1D convolutions.

Model time dynamics using an LSTM.

Use fully connected layers on top to add depth.

Convolutional, Long Short-Term Memory, Fully Connected Deep Neural Networks
Sainath et al. ICASSP'15



Trend: LSTMs end-to-end!

Train recurrent models that also incorporate **Lexical** and **Language Modeling**.

Fast and Accurate Recurrent Neural Network Acoustic Models for Speech Recognition, H. Sak et al.

Deep Speech: Scaling up end-to-end speech recognition, A. Hannun et al.

Listen, Attend and Spell, W. Chan et al.

Hot Topics In Deep Learning

Speech Recognition
Object Recognition
Machine Translation
Image Captioning
Memory and Computation

Hot Topics in Deep Learning: Object Recognition

Bread-and-butter task for Computer Vision

Hotly contested ImageNet ILSVRC challenge:

- First breakthrough for deep learning in 2012 (Krizhevsky et al): brought top-5 error to **16%** where the state-of-the-art was **26%**.
- Progress since brought error down to **5%**.
- Trained human performance is **3** to **5%**.
  (Humans make different kind of mistakes)

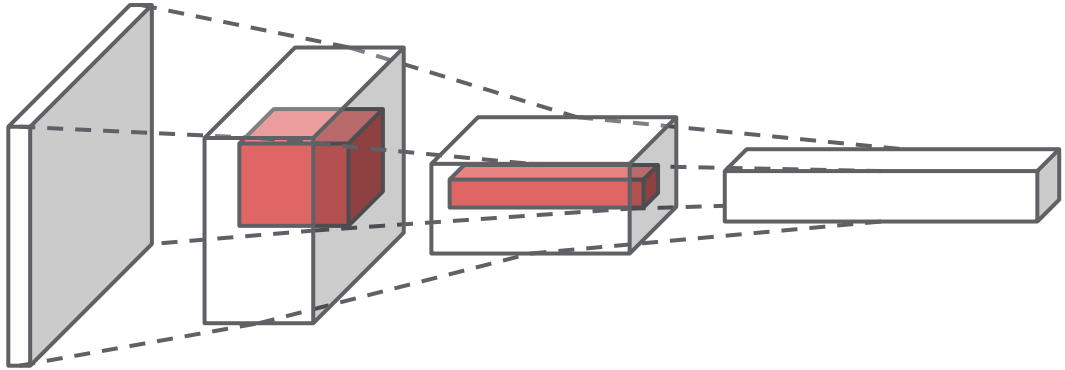
Most improvements via larger, deeper models. Except...

The Inception Architecture

Convolutions are not flexible at allocating their parameters:

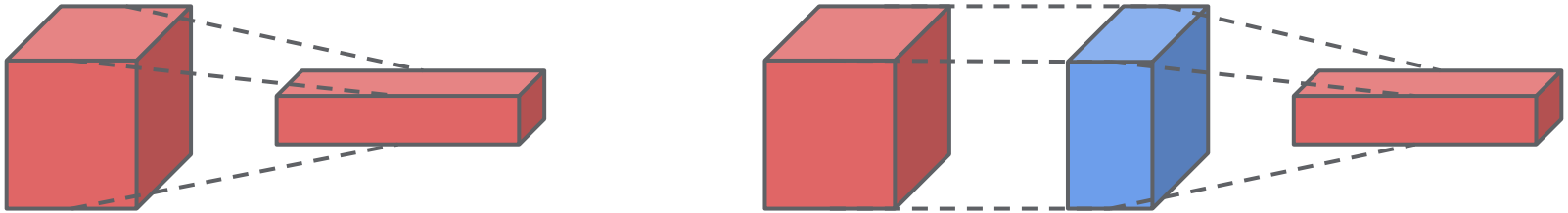
- Every filter looks at the entire depth of the input.
- Every filter has the same spatial extent. (patch size)

This puts tight constraints of the geometry and computational cost.



The Inception Architecture

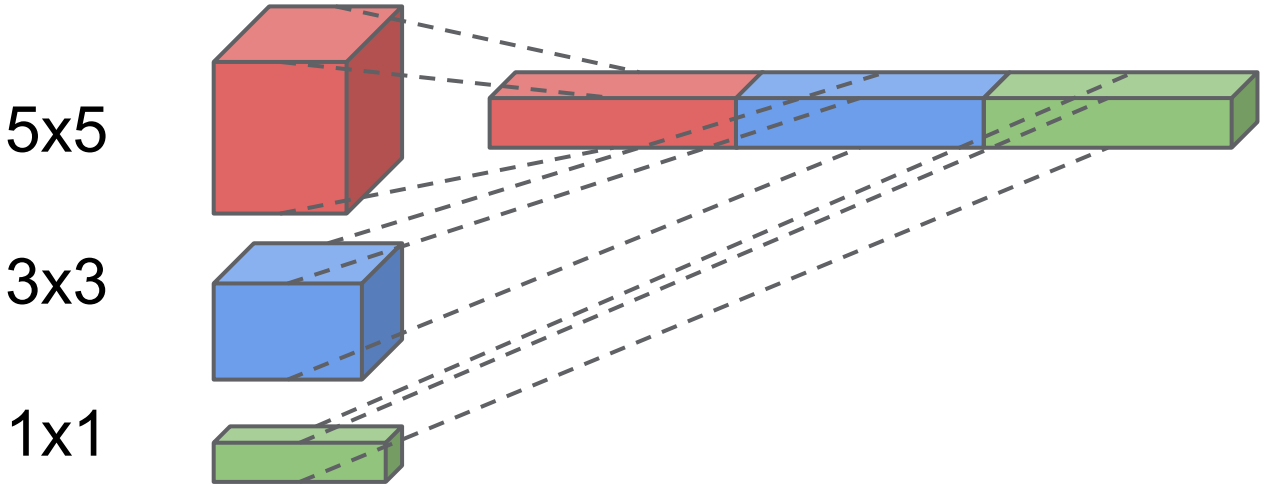
Concept #1: Have different convolutions look at different subsets of the inputs via **projection layers**:



Projection layers are 1x1 convolutions. Very efficient to implement because equivalent to a single matrix multiply. Few parameters.

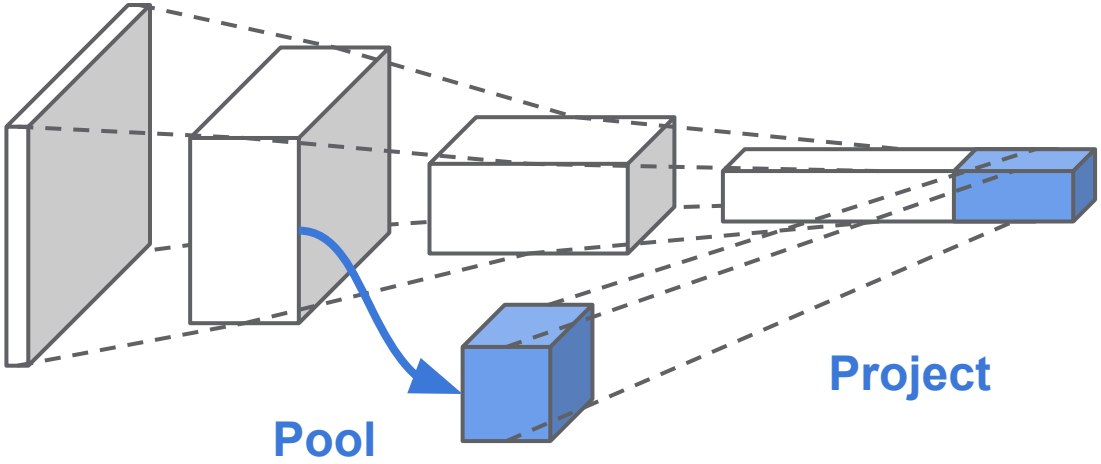
The Inception Architecture

Concept #2: Look at each feature map using a variety of filter sizes, not just one, and concatenate them.



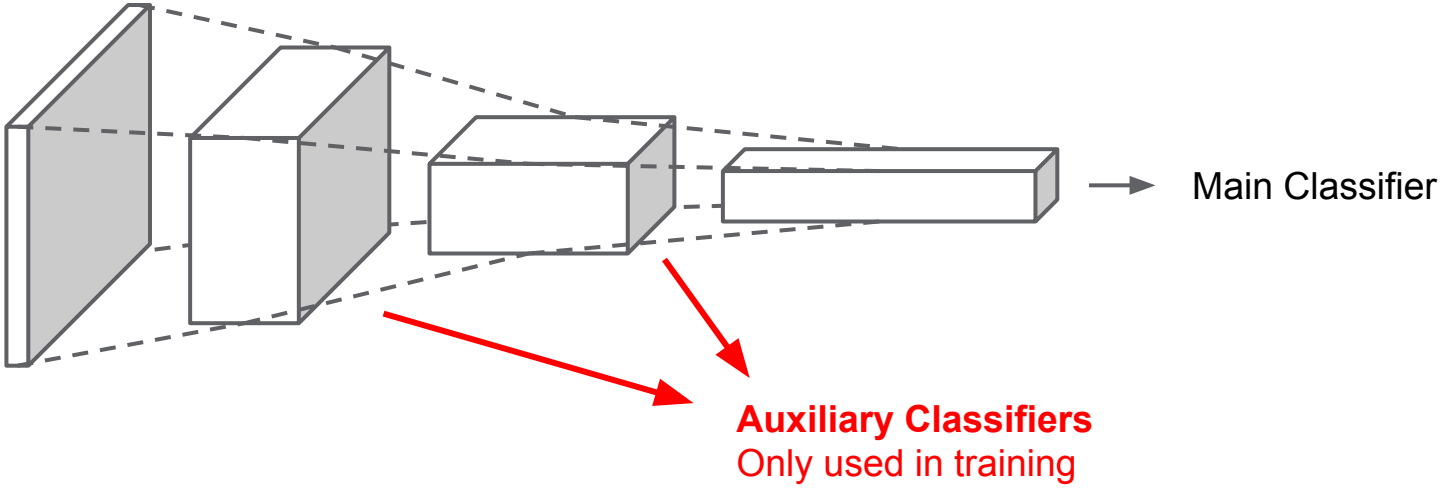
The Inception Architecture

Concept #3: Provide each layer with a low-dimensional pooled view of the previous layer. Similar to often used ‘skip connections’.



The Inception Architecture

Concept #4: Help training along by providing side objectives. Tiny classifiers added at various levels of the convolution tower:



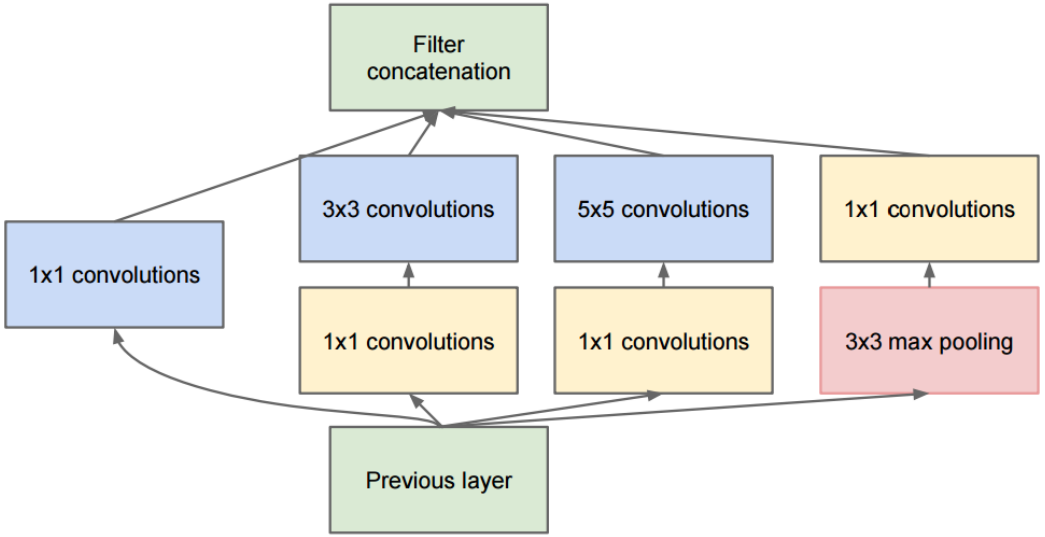
The Inception Architecture

Putting it all together:

Each 1x1 acts as a bottleneck

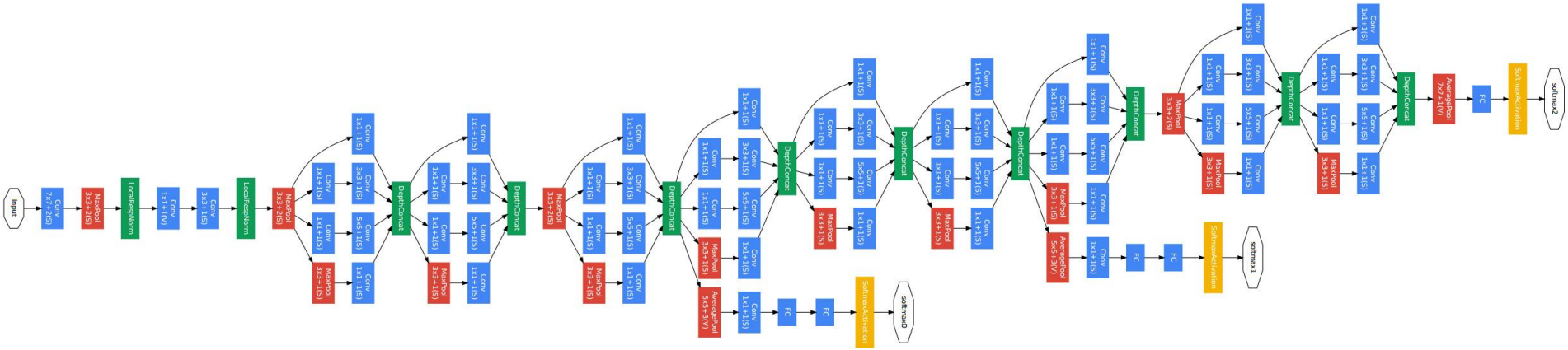
Controls the number of parameters per layers

Lots of (too many?) knobs



Google

The Inception Architecture



Going Deeper with Convolutions

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich

The Inception Architecture

ImageNet challenge:

Only **11M** parameters!

Compare to **130M** parameters for 2nd place VGG.

11MB (8 bit fixed-point): model fits easily in a mobile app

Team	Year	Place	Error (top-5)	Uses external data
SuperVision	2012	1st	16.4%	no
SuperVision	2012	1st	15.3%	Imagenet 22k
Clarifai	2013	1st	11.7%	no
Clarifai	2013	1st	11.2%	Imagenet 22k
MSRA	2014	3rd	7.35%	no
VGG	2014	2nd	7.32%	no
GoogLeNet	2014	1st	6.67%	no

Hot Topics In Deep Learning

Speech Recognition
Object Recognition
Machine Translation
Image Captioning
Memory and Computation

Hot Topics in Deep Learning: Machine Translation

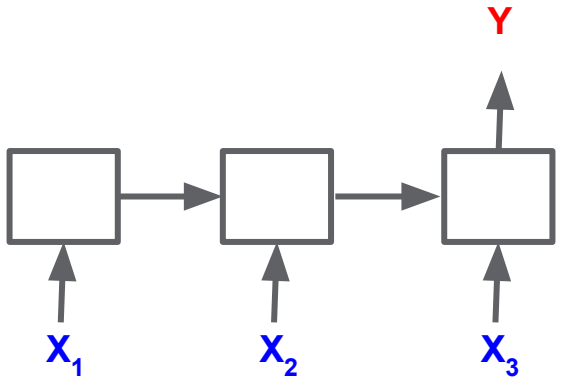
Machine Translation typically involves multiple steps of processing:

- Reordering of words into a consistent, canonical order.
- Mapping words / phrases to candidates in the target language.
- Scoring candidates using a language model.

Can we devise a system that optimizes all these steps jointly?

LSTM = Trainable Sequence-to-Vector Mapping

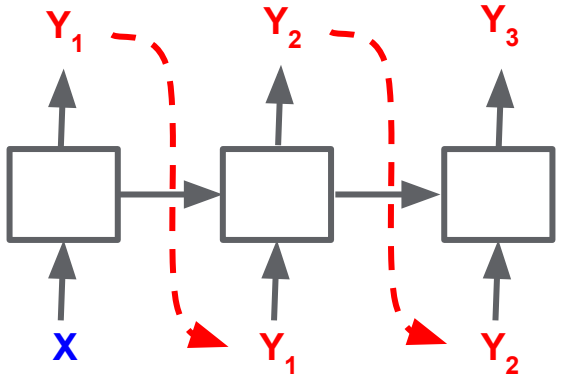
$X_1, X_2, X_3 \rightarrow Y$



Can we express the opposite operation and map a vector to a sequence?

LSTM = **Bidirectional** Trainable Sequence-to-Vector Mapping

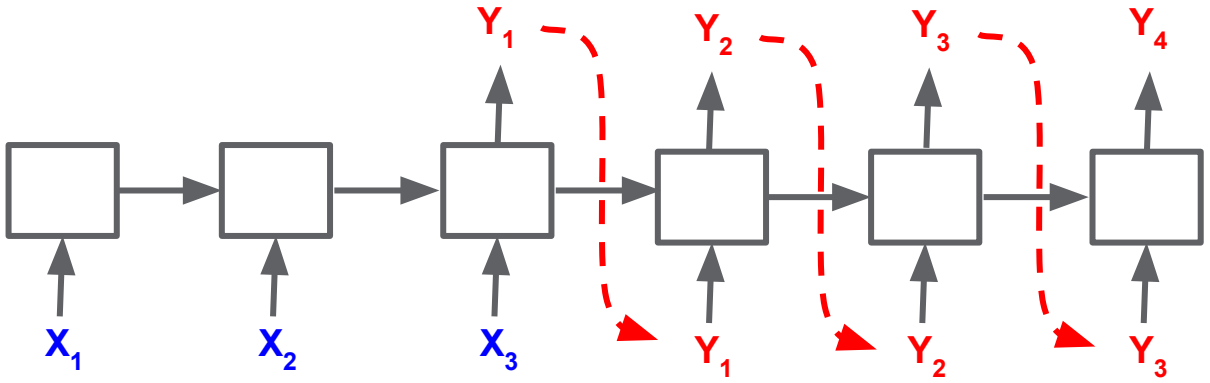
$X \rightarrow Y_1, Y_2, Y_3$



Yes! Very simple idea, but profound implications.

Mapping Sequences to Sequences

$X_1, X_2, X_3 \rightarrow Y_1, Y_2, Y_3, Y_4$



Fully trainable. Agnostic to input and output sequence length.

Sequence-to-Sequence problems

Machine Translation:

Sequence to Sequence Learning with Neural Networks
Sutskever et al., NIPS'14

Parsing:

Grammar as a Foreign Language
Vinyals et al., ICLR'15

Speech Recognition? Text-to-Speech? Filtering? Event detection?

Lots of Open Issues!

Best traditional MT systems leverage monolingual data:

On Using Monolingual Corpora in Neural Machine Translation

Caglar Gulcehre et al., arXiv, 2015

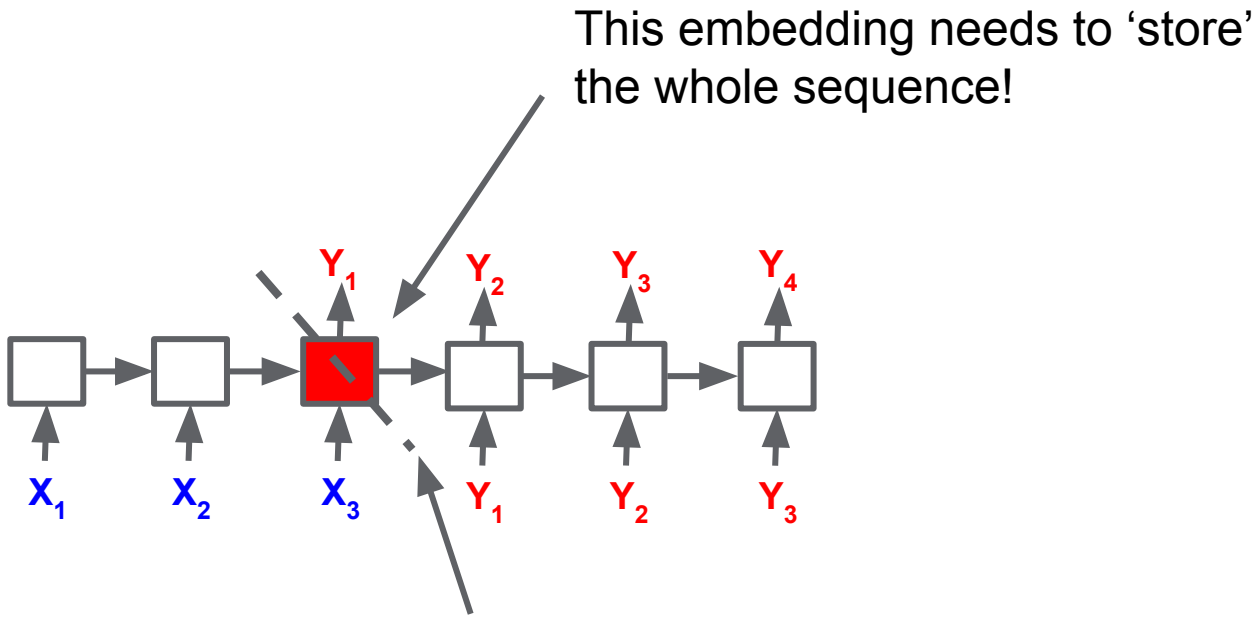
Out-of-vocabulary words:

Addressing the Rare Word Problem in Neural Machine Translation

Thang Luong et al., ACL'15

Biggest issue of all: scaling!

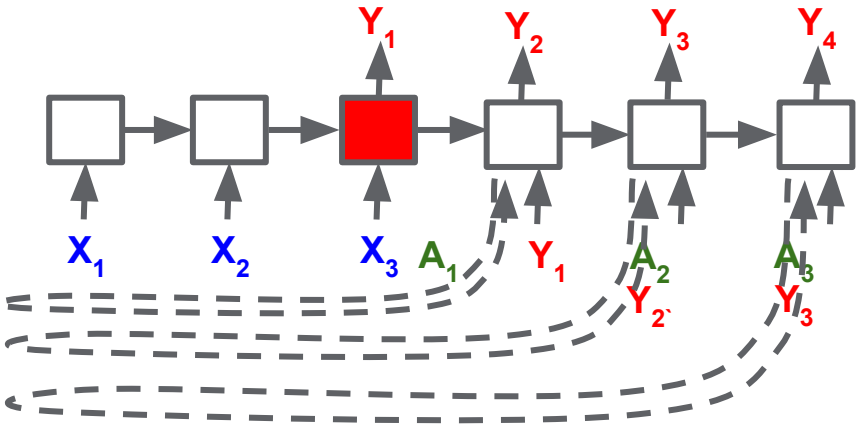
One Scaling Issue: the Embedding Bottleneck



One Approach to Scaling Neural Translation: Attention Models

Differentiable Attention:
During decoding, look back at the input sequence and derive 'attentional' embeddings A_1, A_2, A_3

Main idea: if X_2 translates to Y_2, the model can make A_2 look like X_2.



Neural Machine Translation by Jointly Learning to Align and Translate
Dzmitry Bahdanau et al., ICLR'15

Hot Topics in Deep Learning: Machine Translation

Still lots of scalability issues with these models.

Modern speech recognition and machine translation systems use **one to two orders** of magnitude more data than can be fed to a sequence-to-sequence model in practice.

Having a ‘universal’, trainable bidirectional sequence-to-vector mapper opens up interesting new avenues.

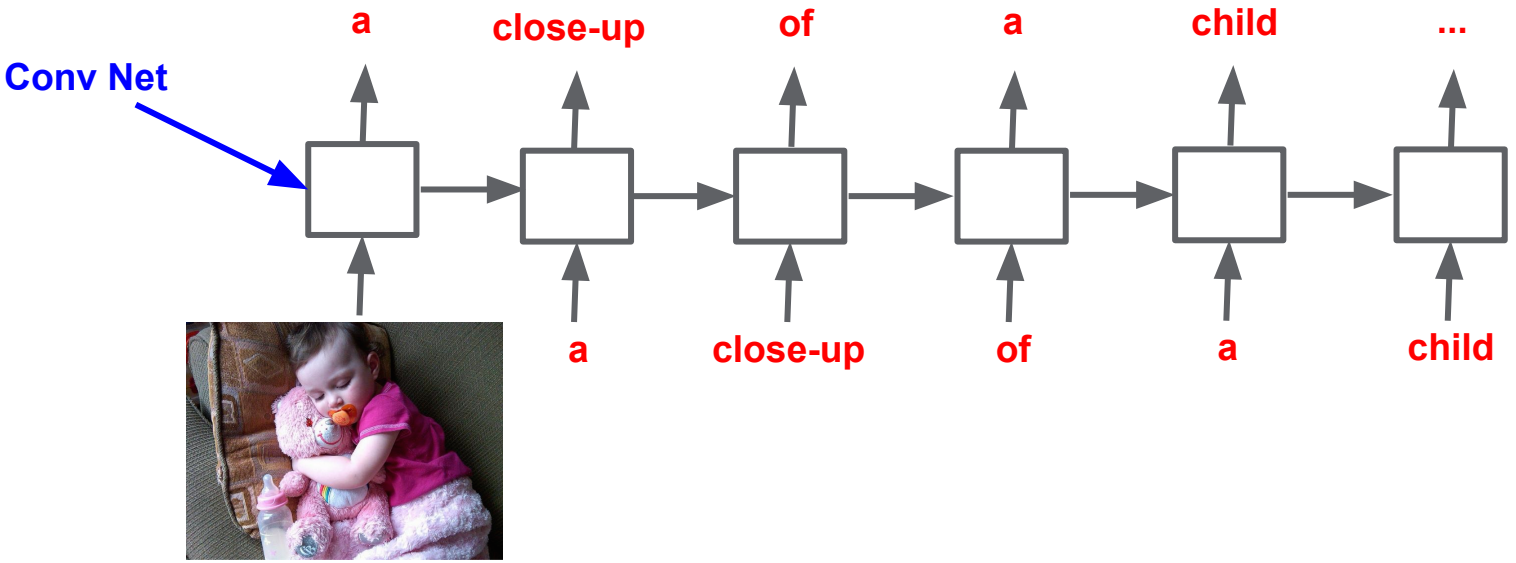
Geoff Hinton calls these **Thought Vectors.**

Hot Topics In Deep Learning

Speech Recognition
Object Recognition
Machine Translation
Image Captioning
Memory and Computation

Hot Topics in Deep Learning: Image Captioning

Image captioning is just another translation problem:
Map an image to a **Thought Vector**, and decode it back into text.



Google

MSCOCO Challenge: http://mscoco.org

Overview | Download | Evaluate | Leaderboard | Challenges

Table-C5 | Table-C40 | Table-human

Last update: June 8, 2015. Visit CodaLab for the latest results.

	M1	M2	M3	M4	M5
Human[5]	0.638	0.675	4.836	3.428	0.352
Google[4]	0.273	0.317	4.107	2.742	0.233
MSR[8]	0.268	0.322	4.137	2.662	0.234
Montreal/Toronto[10]	0.262	0.272	3.932	2.832	0.197
MSR Captivator[9]	0.250	0.301	4.149	2.565	0.233
Berkeley LRCN[2]	0.246	0.268	3.924	2.786	0.204

Hot Topics In Deep Learning

Speech Recognition
Object Recognition
Machine Translation
Image Captioning
Memory and Computation

Hot Topics in Deep Learning: Memory and Computation

These models can learn **facts** and **compute** complex relationships from data. How close are we to build a **fully trainable computer**?

Two important lines of inquiry:

- Incorporating memory.
- Learning programs.

Memory

LSTMs have the equivalent of CPU registers:
directly addressable memory cells.

Can we provide them with **RAM**? **Hard Drives**?

RAM: indirect addressing. Content-based addressing is the main idea behind the differentiable attention model.

Also:

Memory Networks
Jason Weston et al. ICLR'15

Memory

Fitting deep networks with Hard Drives: **knowledge bases**.

Currently, these models are closed systems: they have to be taught everything. Can we teach them to **retreive facts instead of teaching them facts**? Could my neural translation model learn to search the web for unknown words? Could it simply look things up in a translation table instead of having the translation table be fed during training?

Main issues: combing through databases and knowledge sources is not easy to express as a **differentiable process**.

Computation

Expressing **generic algorithms** as differentiable processes that can be **backpropagated through** to learn computation strategies is a huge problem.

LSTMs are able to express a narrow set of computations: Load, Store, Erase. Can we generalize this?

Neural Turing Machines
Alex Graves et al., arXiv, 2014

Reinforcement Learning

Deep models need to be fully (or very close to) differentiable to be trainable.

Reinforcement learning opens up the class of possible models to include non-differentiable representations.

The cost is that these models don’t scale well with the size of the space to be explored...yet.

Human-level Control Through Deep Reinforcement Learning
Volodymyr Mnih et al., Nature 518, 2015

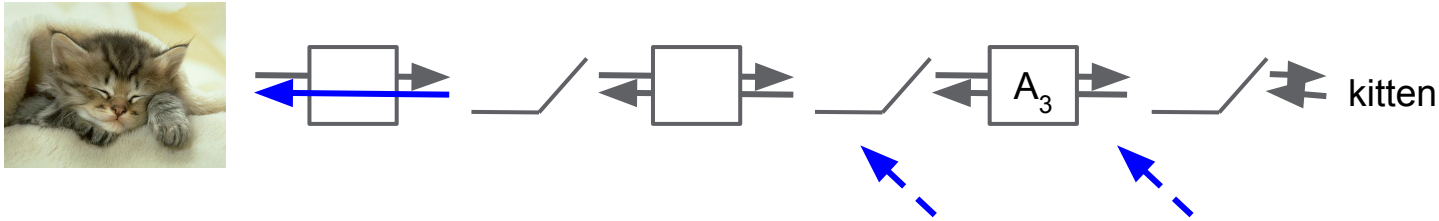
Hot Topics in Deep Learning: Robots!

End-to-end learning from example.

Bypasses much of traditional robotics approaches: localization, registration, motion planning.

End-to-End Training of Deep Visuomotor Policies
Sergey Levine, Chelsea Finn et al., arXiv, 2015

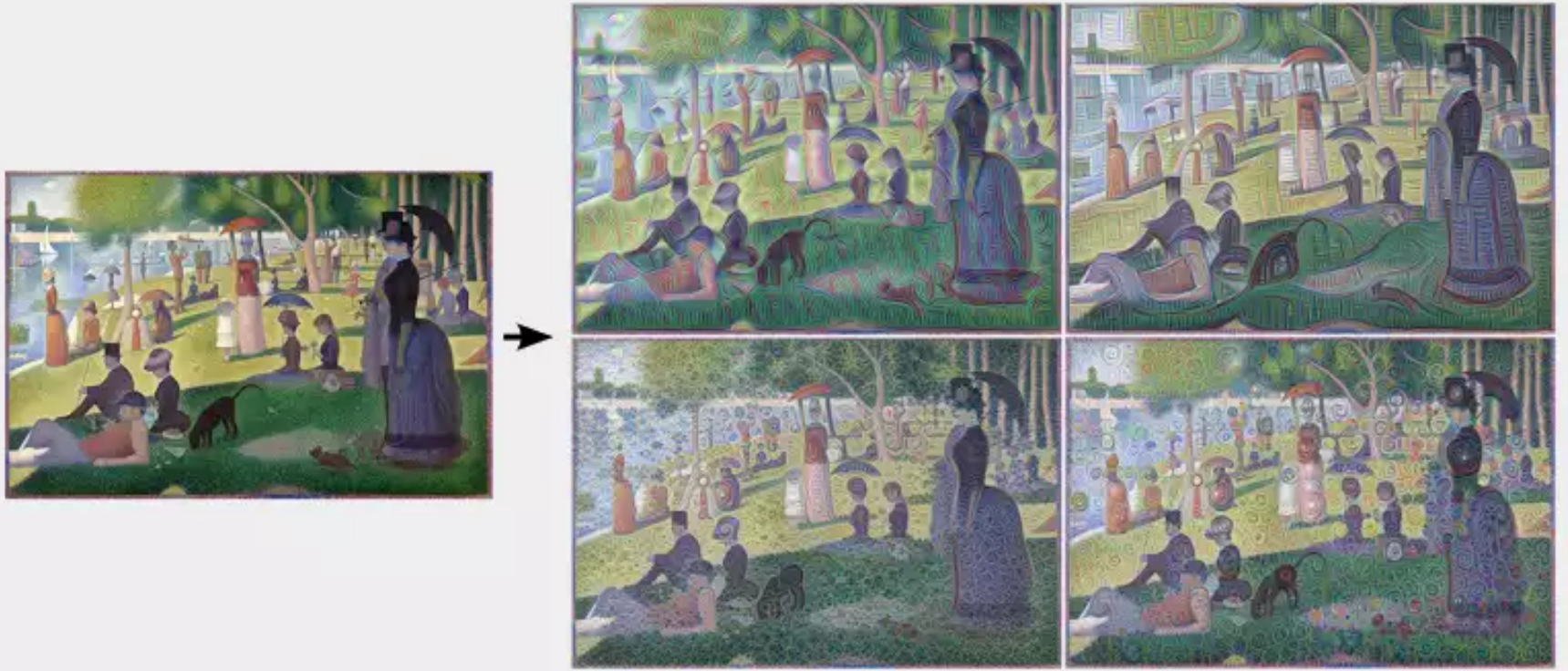
Hot Topics in Deep Learning: Art! (sort of…)



Interesting things happen when you **reinforce/bias** a network's beliefs and propagate the outcome **back to the input space**.

As seen on social media under the terms "inceptionism" or "deepsee".

Different Layers -> Different Filters



Google

ImageNet: Dogs, Birds!

Courtesy:
Alexander Mordvintsev

Google

Factoring Style and Content!

A Neural Algorithm of Artistic Style
L.A. Gatys, A.S. Ecker, M. Bethge

Parting Thoughts

Deep Learning is a rich field at the confluence of **machine learning** and **computing infrastructure** research.

Most direct perception tasks (audio and visual recognition) are on a **predictable** improvement path. It's time to focus instead on the difficult, **"A.I. complete"** problems.

Lots to explore: as we approach **human-level perception,** the dream of **general artificial intelligence** is looking a lot less implausible!

Questions:

vanhoucke@google.com