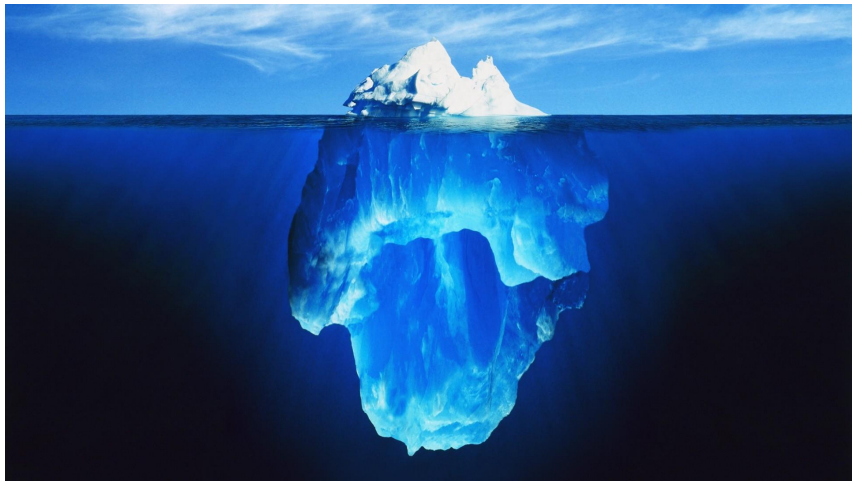
Deep Learning: A Shallow Tutorial

André Martins

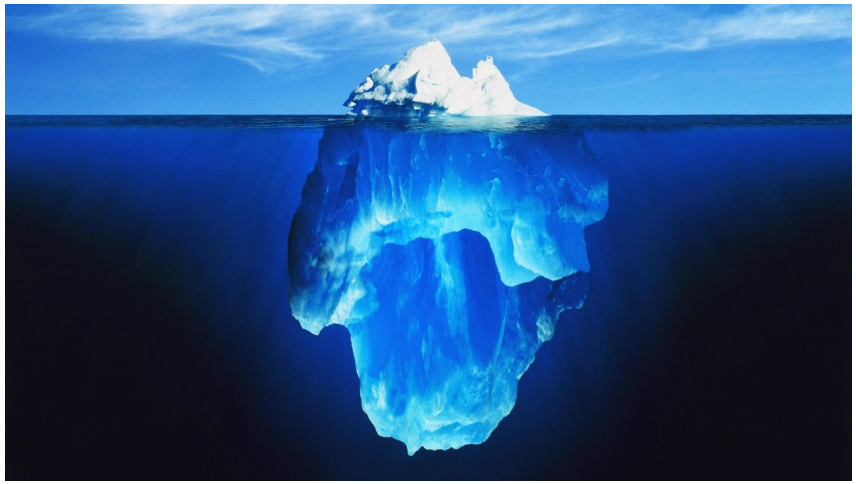
NLPL Winter School, Skeicampen, 29–31/01/18

What is Deep Learning?



- Neural networks?
- Neural networks with many hidden layers?
- Anything beyond shallow (linear) models for statistical learning?
- Anything that learns representations?
- A form of learning that is really intense and profound?

What is Deep Learning?



- Neural networks?
- Neural networks with many hidden layers?
- Anything beyond shallow (linear) models for statistical learning?
- Anything that learns representations?
- A form of learning that is really intense and profound?

Why Did Deep Learning Become Mainstream?

Lots of recent breakthroughs:

- Object recognition
- Speech and language processing
- Self-driving cars
- Machine translation
- Solving games (Atari, Go)

No signs of slowing down...

Microsoft's Deep Learning Project Outperforms Humans In Image Recognition

Michael Thomsen, CONTRIBUTOR

I write about tech, video games, science and culture. **FULL BIO**

Opinions expressed by Forbes Contributors are their own.

Microsoft's new breakthrough: AI that's as good as humans at listening... on the phone

Microsoft's new speech-recognition record means professional transcribers could be among the first to lose their jobs to artificial intelligence.

By Liam Tung | October 19, 2016 -- 10:10 GMT (11:10 BST) | Topic: Innovation

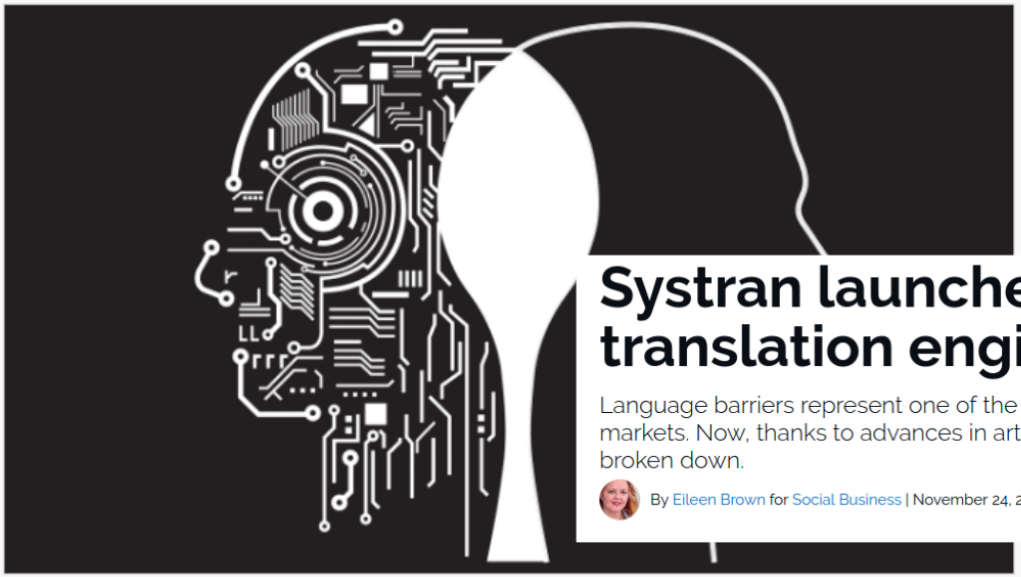
The Great A.I. Awakening

How Google used artificial intelligence to transform Google Translate, one of its more popular services — and how machine learning is poised to reinvent computing itself.

BY GIDEON LEWIS-KRAUS DEC. 14, 2016

Google unleashes deep learning tech on language with Neural Machine Translation

*Posted Sep 27, 2016 by **Devin Coldewey**, Contributor*



Systran launches neural machine translation engine in 30 languages

Language barriers represent one of the biggest challenges to develop business strategies among global markets. Now, thanks to advances in artificial intelligence and machine translation, these barriers are being broken down.

By Eileen Brown for Social Business | November 24, 2016 -- 13:49 GMT (13:49 GMT) | Topic: Artificial Intelligence

Artificial Intelligence | robotics | car | entrepreneurship | Emerging-Technologies

Drive.ai uses deep learning to teach self-driving cars – and to give them a voice

Posted Aug 30, 2016 by Darrell Etherington (@etherington)

AlphaGo Beats Go Human Champ: Godfather Of Deep Learning Tells Us Do Not Be Afraid Of AI

21 March 2016, 10:16 am EDT **By Aaron Mamiit Tech Times**

Last week, Google's artificial intelligence program

Last week, Google's artificial intelligence program AlphaGo dominated its match with South Korean world Go champion Lee Sedol, winning with a 4-1 score.

The achievement stunned artificial intelligence experts, who previously thought that Google's computer program would need at least 10 more years before developing enough to be able to beat a human world champion.

Why Now?

Why does deep learning work now, but not 20 years ago?

Many of the core ideas were there, after all.

But now we have:

- more data
- more computing power
- better software engineering
- a few algorithmic innovations (many layers, ReLUs, better initialization and learning rates, dropout, LSTMs, convolutional nets)

"But It's Non-Convex"

Why does gradient-based optimization work at all in neural nets despite the non-convexity?

One possible, partial answer:

- there are generally many hidden units
- there are many ways a neural net can approximately implement the desired input-output relationship
- we only need to find one

Outline

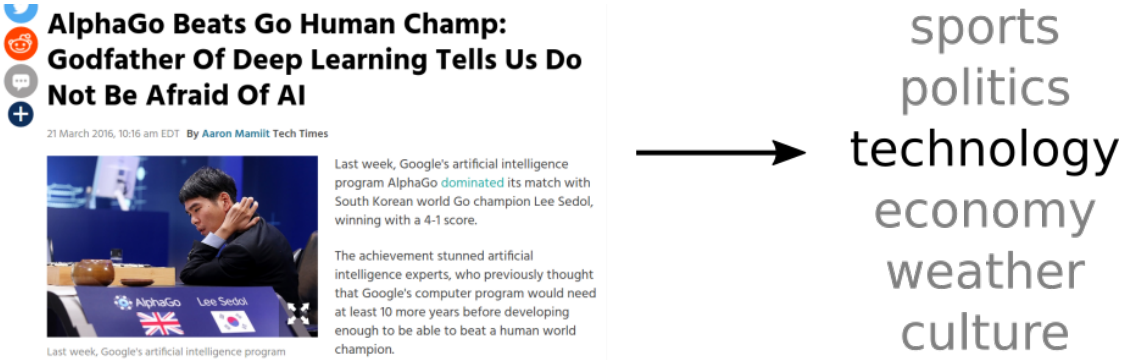
Shallow Learning

Before talking about deep learning, let us talk about shallow learning

Roadmap:

- Classification/regression
- Feature representation
- Linear classifiers
- Perceptron's mistake bound and linear separability

Notation



- Input $x \in \mathcal{X}$ (a news article, an image, ...)
- Output $y \in \mathcal{Y}$ (fake/not fake, a topic, an image segmentation)
- **Goal:** learn a classifier $\varphi : \mathcal{X} \to \mathcal{Y}$ that generalizes to arbitrary inputs
- **Supervised learning:** learn φ from labeled data

$$\{(x_n, y_n)\}_{n=1}^{N} \subseteq \mathcal{X} \times \mathcal{Y}$$

Classification/Regression

- **Regression:** $\mathcal{Y} = \mathbb{R}$
- **Multivariate regression:** $\mathcal{Y} = \mathbb{R}^K$
- **Binary classification:** $\mathcal{Y} = \{\pm 1\}$
- **Multi-class classification:** $\mathcal{Y} = \{1, 2, \ldots, K\}$
- **Structured classification:** $\mathcal{Y}$ exponentially large and structured (e.g., machine translation, caption generation)

Sometimes reductions are convenient:

- one-vs-all for reducing multi-class to binary
- greedy search to reduce structured classification to multi-class

Other times it's better to tackle the problem in its native form

Feature Representation

Feature engineering is an important step in "shallow" learning:

- Bag-of-words features for text, also lemmas, parts-of-speech, ...
- SIFT features and wavelet representations in computer vision
- Other categorical, Boolean, and continuous features

Feature Representation

Feature engineering is an important step in "shallow" learning:

- Bag-of-words features for text, also lemmas, parts-of-speech, ...
- SIFT features and wavelet representations in computer vision
- Other categorical, Boolean, and continuous features

Typical approach: define a feature map $\boldsymbol{\psi} : \mathcal{X} \to \mathbb{R}^D$

For multi-class/structured classification, a joint feature map $\boldsymbol{\phi} : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^D$ is more convenient

For example, letting $\boldsymbol{e}_y := (0, \ldots, 0, 1, 0, \ldots, 0)$ be the indicator vector of a class:

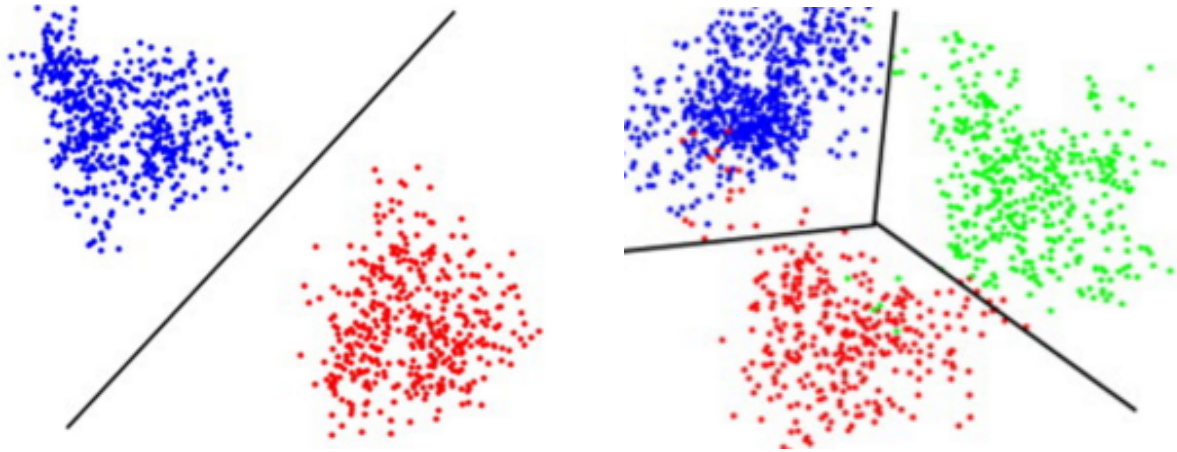
$$\begin{aligned} \boldsymbol{\phi}(x, y) &= \boldsymbol{\phi}(x) \otimes \boldsymbol{e}_y \\ &= (\mathbf{0}, \ldots, \mathbf{0}, \boldsymbol{\psi}(x), \mathbf{0}, \ldots, \mathbf{0}). \end{aligned}$$

Linear Classifiers

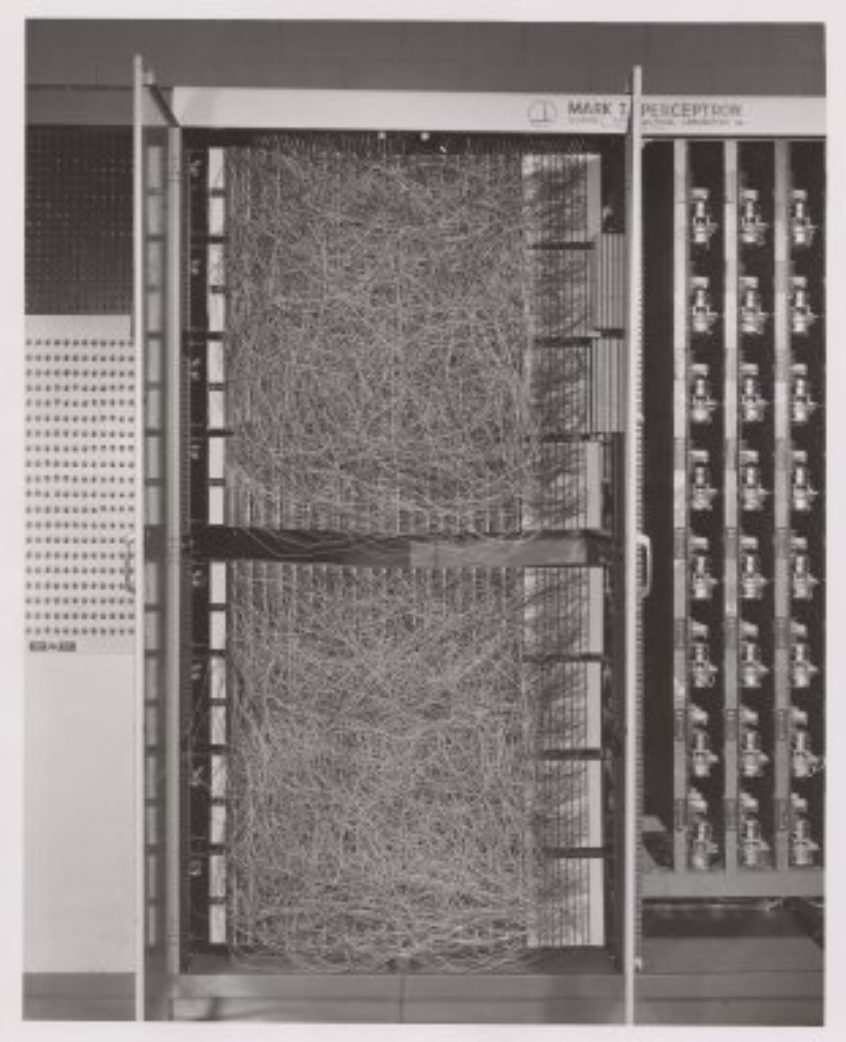
- Parametrized by a weight vector $\boldsymbol{w} \in \mathbb{R}^D$ (one weight per feature)
- Define a score for each class as a linear function of the parameters
- At test time, predict the class $\widehat{y}$ which maximizes this score:

$$\widehat{y} = \varphi(x) = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w} \cdot \boldsymbol{\phi}(x, y)$$

- **Examples:** perceptron, naïve Bayes, logistic regression, support vector machines



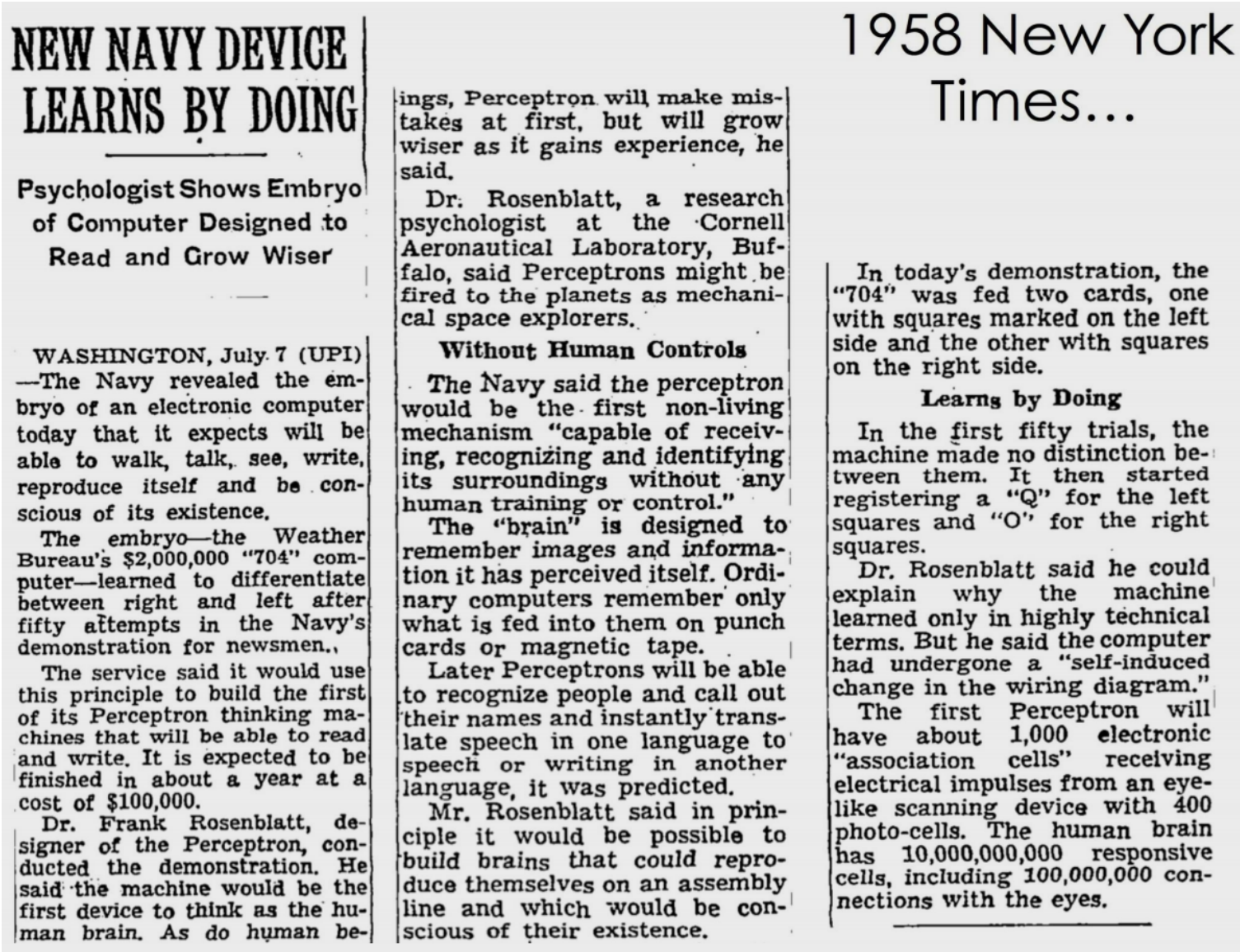
Perceptron (Rosenblatt, 1958)



(Extracted from Wikipedia)

- Invented in 1957 at the Cornell Aeronautical Laboratory by Frank Rosenblatt
- Implemented in custom-built hardware as the "Mark 1 perceptron," designed for image recognition
- 400 photocells, randomly connected to the "neurons." Weights were encoded in potentiometers
- Weight updates during learning were performed by electric motors.

Perceptron in the News...



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

1958 New York Times...

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

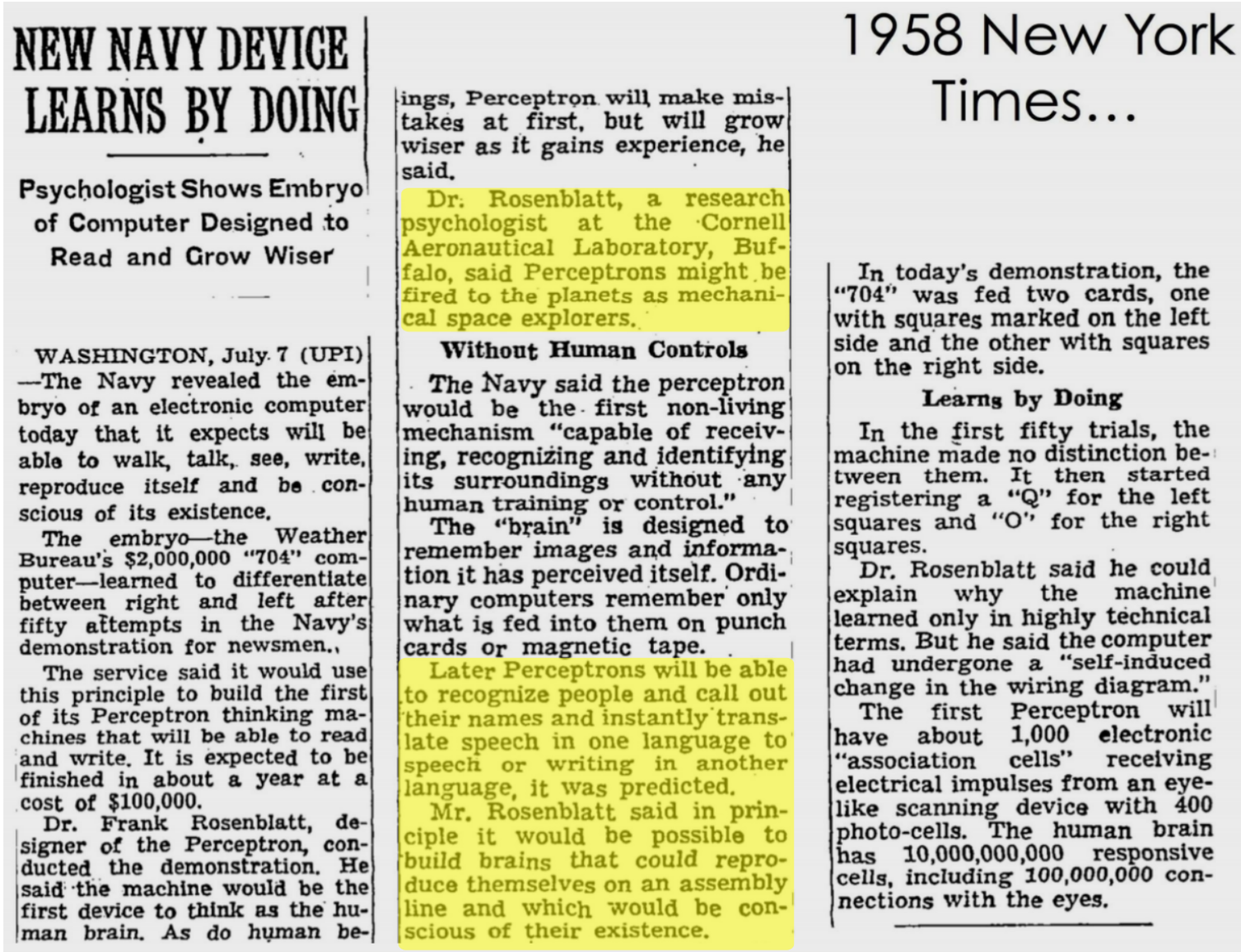
Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

Perceptron in the News...



NEW NAVY DEVICE LEARNS BY DOING

Psychologist Shows Embryo of Computer Designed to Read and Grow Wiser

WASHINGTON, July 7 (UPI) —The Navy revealed the embryo of an electronic computer today that it expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence.

The embryo—the Weather Bureau's $2,000,000 "704" computer—learned to differentiate between right and left after fifty attempts in the Navy's demonstration for newsmen.

The service said it would use this principle to build the first of its Perceptron thinking machines that will be able to read and write. It is expected to be finished in about a year at a cost of $100,000.

Dr. Frank Rosenblatt, designer of the Perceptron, conducted the demonstration. He said the machine would be the first device to think as the human brain. As do human beings, Perceptron will make mistakes at first, but will grow wiser as it gains experience, he said.

Dr. Rosenblatt, a research psychologist at the Cornell Aeronautical Laboratory, Buffalo, said Perceptrons might be fired to the planets as mechanical space explorers.

Without Human Controls

The Navy said the perceptron would be the first non-living mechanism "capable of receiving, recognizing and identifying its surroundings without any human training or control."

The "brain" is designed to remember images and information it has perceived itself. Ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons will be able to recognize people and call out their names and instantly translate speech in one language to speech or writing in another language, it was predicted.

Mr. Rosenblatt said in principle it would be possible to build brains that could reproduce themselves on an assembly line and which would be conscious of their existence.

1958 New York Times...

In today's demonstration, the "704" was fed two cards, one with squares marked on the left side and the other with squares on the right side.

Learns by Doing

In the first fifty trials, the machine made no distinction between them. It then started registering a "Q" for the left squares and "O" for the right squares.

Dr. Rosenblatt said he could explain why the machine learned only in highly technical terms. But he said the computer had undergone a "self-induced change in the wiring diagram."

The first Perceptron will have about 1,000 electronic "association cells" receiving electrical impulses from an eye-like scanning device with 400 photo-cells. The human brain has 10,000,000,000 responsive cells, including 100,000,000 connections with the eyes.

Perceptron Algorithm

input: labeled data $\{(x_i, y_i)\}_{i=1}^{N} \subseteq \mathcal{X} \times \mathcal{Y}$
initialize $\boldsymbol{w}^{(0)} = \mathbf{0}$
initialize $k = 0$ (number of mistakes)
repeat
  get new training example x_i, y_i
  predict $\widehat{y}_i = \arg\max_{y \in \mathcal{Y}} \boldsymbol{w}^{(k)} \cdot \phi(x_i, y)$
  if $\widehat{y}_i \neq y_i$ **then**
    update $\boldsymbol{w}^{(k+1)} = \boldsymbol{w}^{(k)} + \phi(x_i, y_i) - \phi(x_i, \widehat{y}_i)$
    increment k
  end if
until maximum number of epochs
output: model weights $\boldsymbol{w}$

Perceptron's Mistake Bound

A couple definitions:

- the training data is linearly separable with margin $\gamma > 0$ iff there is a weight vector $\boldsymbol{u}$ with $\|\boldsymbol{u}\| = 1$ such that

$$\boldsymbol{u} \cdot \phi(x_i, y_i) \geq \boldsymbol{u} \cdot \phi(x_i, y_i') + \gamma, \quad \forall i \in [N], \forall y_i' \neq y_i.$$

- radius of the data: $R = \max_{i \in [N], y_i' \neq y_i} \|\phi(x_i, y_i) - \phi(x_i, y_i')\|$.

Perceptron's Mistake Bound

A couple definitions:

- the training data is linearly separable with margin $\gamma > 0$ iff there is a weight vector $\boldsymbol{u}$ with $\|\boldsymbol{u}\| = 1$ such that

$$\boldsymbol{u} \cdot \phi(x_i, y_i) \geq \boldsymbol{u} \cdot \phi(x_i, y_i') + \gamma, \quad \forall i \in [N], \forall y_i' \neq y_i.$$

- radius of the data: $R = \max_{i \in [N], y_i' \neq y_i} \|\phi(x_i, y_i) - \phi(x_i, y_i')\|$.

Then we have the following bound of the number of mistakes:

Theorem (Novikoff (1962))

The perceptron algorithm is guaranteed to find a separating hyperplane after at most $\frac{R^2}{\gamma^2}$ *mistakes.*

One-Slide Proof

- **Lower bound on $\|\boldsymbol{w}^{(k+1)}\|$:**

$$\begin{aligned}
\boldsymbol{u} \cdot \boldsymbol{w}^{(k+1)} &= \boldsymbol{u} \cdot \boldsymbol{w}^{(k)} + \boldsymbol{u} \cdot (\phi(x_i, y_i) - \phi(x_i, \widehat{y}_i)) \\
&\geq \boldsymbol{u} \cdot \boldsymbol{w}^{(k)} + \delta \\
&\geq k\delta.
\end{aligned}$$

Hence $\|\boldsymbol{w}^{(k+1)}\| = \|\boldsymbol{u}\| \cdot \|\boldsymbol{w}^{(k+1)}\| \geq \boldsymbol{u} \cdot \boldsymbol{w}^{(k+1)} \geq k\delta$ (from CSI).

One-Slide Proof

- **Lower bound on $\|\boldsymbol{w}^{(k+1)}\|$:**

$$\begin{aligned}\boldsymbol{u}\cdot\boldsymbol{w}^{(k+1)} &= \boldsymbol{u}\cdot\boldsymbol{w}^{(k)} + \boldsymbol{u}\cdot(\phi(x_i,y_i)-\phi(x_i,\widehat{y}_i)) \\ &\geq \boldsymbol{u}\cdot\boldsymbol{w}^{(k)} + \delta \\ &\geq k\delta.\end{aligned}$$

  Hence $\|\boldsymbol{w}^{(k+1)}\| = \|\boldsymbol{u}\|\cdot\|\boldsymbol{w}^{(k+1)}\| \geq \boldsymbol{u}\cdot\boldsymbol{w}^{(k+1)} \geq k\delta$ (from CSI).

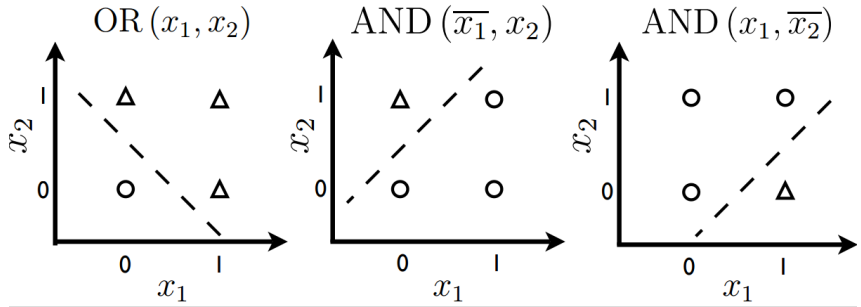
- **Upper bound on $\|\boldsymbol{w}^{(k+1)}\|$:**

$$\begin{aligned}\|\boldsymbol{w}^{(k+1)}\|^2 &= \|\boldsymbol{w}^{(k)}\|^2 + \|\phi(x_i,y_i)-\phi(x_i,\widehat{y}_i)\|^2 \\ &\quad +2\boldsymbol{w}^{(k)}\cdot(\phi(x_i,y_i)-\phi(x_i,\widehat{y}_i)) \\ &\leq \|\boldsymbol{w}^{(k)}\|^2 + R^2 \\ &\leq kR^2.\end{aligned}$$

Equating both sides, we get $(k\delta)^2 \leq kR^2 \;\Rightarrow\; k \leq R^2/\delta^2$ (QED).

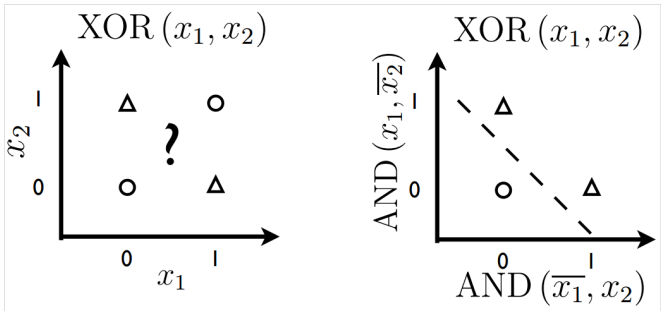
What a Simple Perceptron Can and Can't Do

- Can solve linearly separable problems (OR, AND)



(Image credit: Hugo Larochelle)

- Can't solve non-linearly separable problems (XOR)—unless input is transformed into a better representation



(Image credit: Hugo Larochelle)

- This was observed by Minsky and Papert (1969) and motivated strong criticisms

Other Linear Classifiers

Logistic regression.

- Define $p_{\boldsymbol{w}}(y \mid x) \propto \exp(\boldsymbol{w} \cdot \phi(x, y))$ and maximize the conditional log-likelihood of the training data

Support vector machines.

- Similar to perceptron, but attempts to find the model $\boldsymbol{w}$ that maximizes the separation margin δ
- Adds slack variables for non-separable data, penalizing violations

Both lead to convex optimization problems $\Rightarrow$ no issues with local minima/initialization

Both assume the features are well-engineered such that the data is nearly linearly separable

What If Data Are Not Linearly Separable?

What If Data Are Not Linearly Separable?

Engineer better features (often works!)

What If Data Are Not Linearly Separable?

Engineer better features (often works!)

Kernel methods:

- works implicitly in a high-dimensional feature space
- ... but still need to choose/design a good kernel
- model capacity confined to positive-definite kernels

What If Data Are Not Linearly Separable?

Engineer better features (often works!)

Kernel methods:

- works implicitly in a high-dimensional feature space
- ... but still need to choose/design a good kernel
- model capacity confined to positive-definite kernels

Neural networks (next)

- ... or how I stopped worrying and learned to love non-convexity
- instead of engineering features/kernels, engineer the model architecture

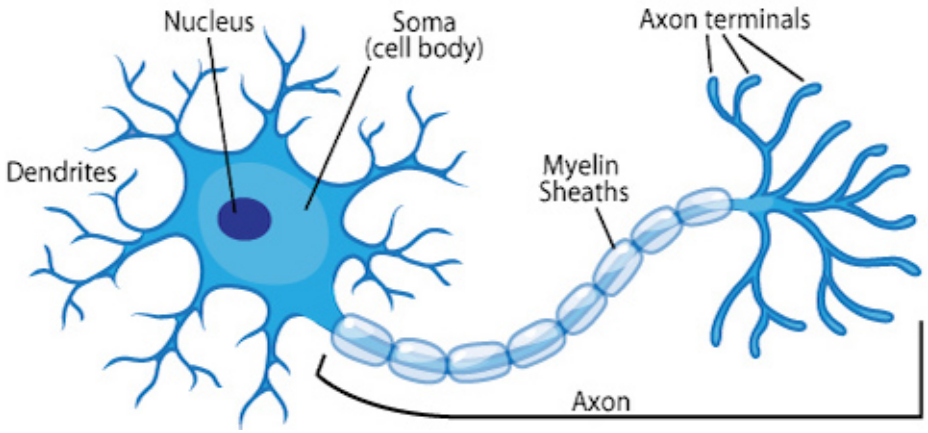
Outline

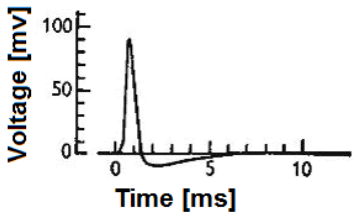
Neural Networks

Roadmap:

- Biological and artificial neuron
- Activation functions
- Multi-layer neural networks
- Softmax and sparsemax
- Universal approximation theorem

Biological Neuron

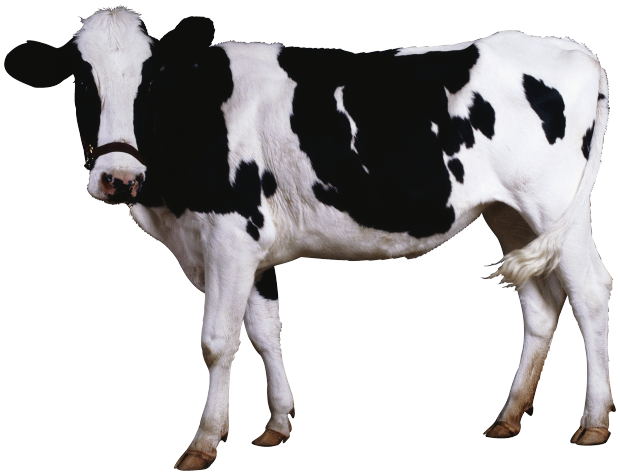




- Three main parts: the main body (soma), dendrites and an axon
- The neuron receives input signals from dendrites, and then outputs its own signals through the axon
- Axons in turn connect to the dendrites of other neurons, using special connections called synapses
- Generate sharp electrical potentials across their cell membrane (spikes), a major signaling unit of the nervous system

Word of Caution

- Artificial neurons are inspired by biological neurons in nervous systems, but...

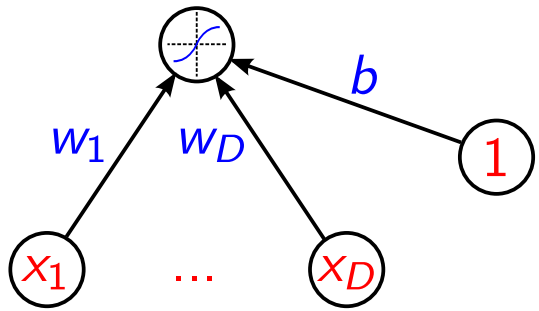


Artificial Neuron (McCulloch and Pitts, 1943; Rosenblatt, 1958)

- **Pre-activation** (input activation):

$$z(\boldsymbol{x}) = \boldsymbol{w} \cdot \boldsymbol{x} + b = \sum_{i=1}^{D} w_i x_i + b,$$

where $\boldsymbol{w}$ are the connection weights and b is a bias term.



- **Activation:**

$$h(\boldsymbol{x}) = g(z(\boldsymbol{x})) = g(\boldsymbol{w} \cdot \boldsymbol{x} + b),$$

where $g : \mathbb{R} \rightarrow \mathbb{R}$ is the activation function.

Activation Function

Typical choices:

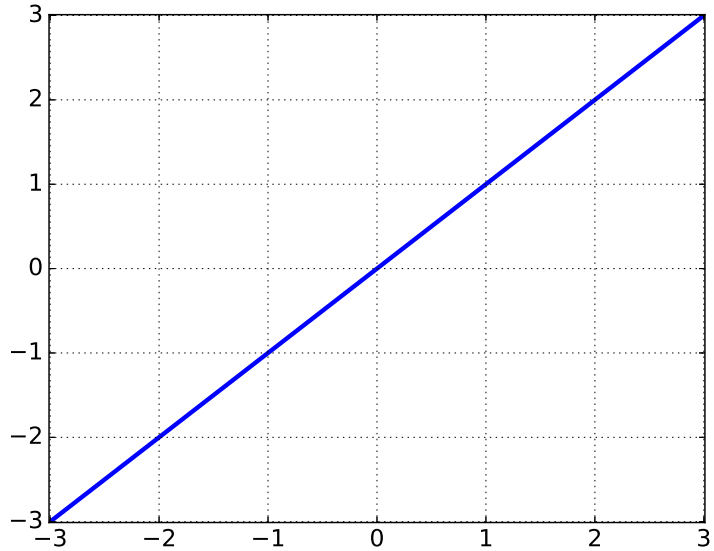
- Linear
- Sigmoid (logistic function)
- Hyperbolic Tangent
- Rectified Linear

Later:

- Softmax
- Sparsemax
- Max-pooling

Linear Activation

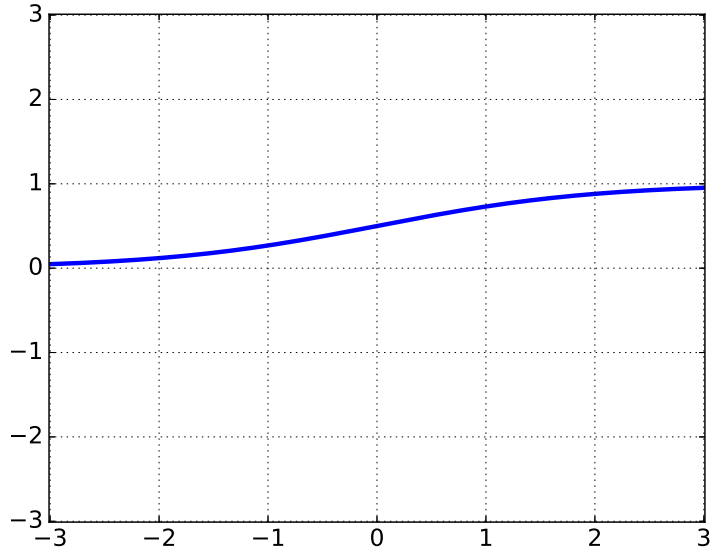
$$g(z) = z$$



- No "squashing" of the input
- Composing layers of linear units is equivalent to a single layer of linear units, so no expressive power added when going multi-layer (more later)
- Still useful to linear-project the input to a lower dimension

Sigmoid Activation

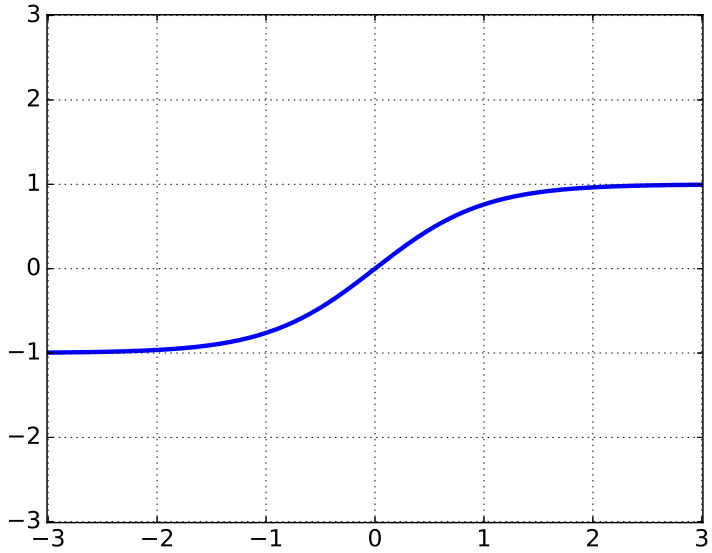
$$g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$



- "Squashes" the neuron pre-activation between 0 and 1
- The output can be interpreted as a probability
- Positive, bounded, strictly increasing
- Logistic regression corresponds to a network with a single sigmoid unit
- Combining layers of sigmoid units will increase expressiveness (more later)

Hyperbolic Tangent Activation

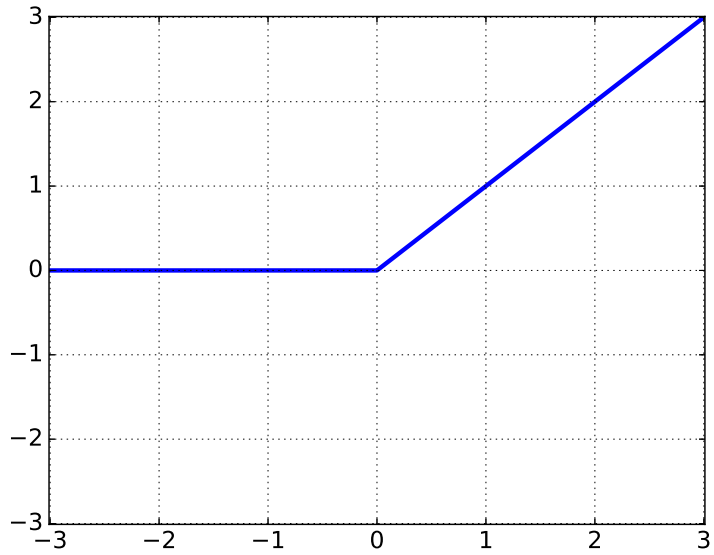
$$g(z) = \textbf{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



- "Squashes" the neuron pre-activation between -1 and 1
- Related to the sigmoid via $\sigma(z) = \frac{1+\textbf{tanh}(z/2)}{2}$
- Can be positive or negative, bounded, strictly increasing
- Combining layers of tanh units will increase expressiveness (more later)

Rectified Linear Unit Activation (Glorot et al., 2011)

$$g(z) = \textbf{relu}(z) = \max\{0, z\}$$



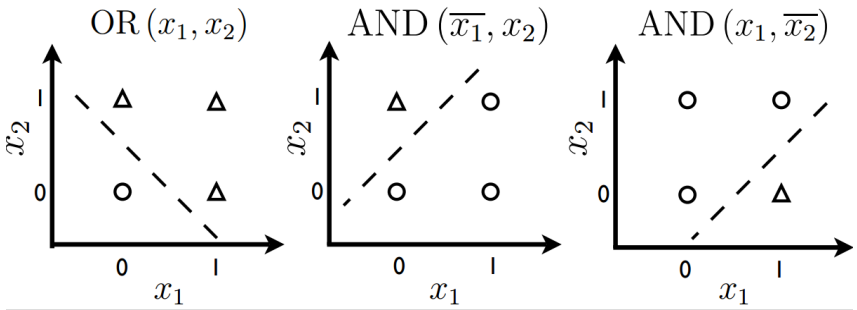
- Less prone to vanishing gradients (more later), and historically the first activation that allowed training deep nets without unsupervised pre-training (Glorot et al., 2011)
- Non-negative, increasing, but not upper bounded
- Not differentiable at 0
- Leads to neurons with sparse activities (biologically more plausible)

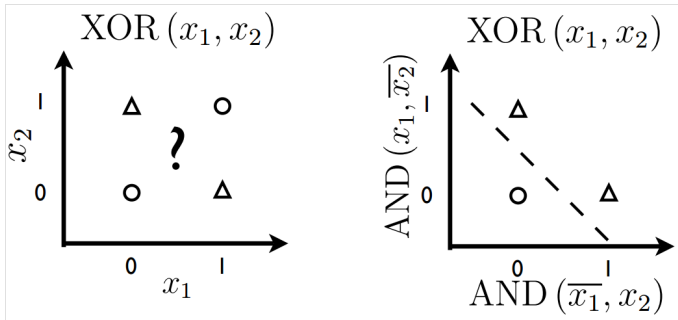
Capacity of Single Neuron (Linear Classifier)

- With a single sigmoid activated neuron we can do logistic regression:

$$p(y = 1|\boldsymbol{x}) = \sigma(\boldsymbol{w} \cdot \boldsymbol{x} + b).$$

- Can solve linearly separable problems (OR, AND)
- Can't solve non-linearly separable problems (XOR)—unless input is transformed into a better representation





(Slide credit: Hugo Larochelle)

Multi-Layer Neural Network

- **Key idea:** add intermediate layers of artificial neurons before the final output layer
- Each of these hidden units computes some representation of the input and propagates forward that representation
- This increases the expressive power of the network, yielding more complex, non-linear, classifiers
- Similar role as latent variables in probabilistic models, but no need for a probability semantics
- Also called feed-forward neural network

Single Hidden Layer

Assume D inputs ($\boldsymbol{x} \in \mathbb{R}^D$) and K hidden units ($\boldsymbol{h} \in \mathbb{R}^K$)

- **Hidden layer pre-activation:**

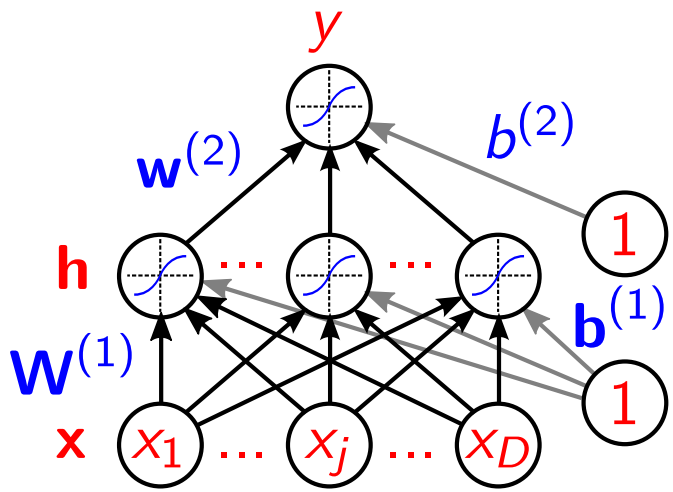
$$\boldsymbol{z}(\boldsymbol{x}) = \mathbf{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)},$$

with $\mathbf{W}^{(1)} \in \mathbb{R}^{K \times D}$ and $\boldsymbol{b}^{(1)} \in \mathbb{R}^K$.

- **Hidden layer activation:**

$$\boldsymbol{h}(\boldsymbol{z}) = \boldsymbol{g}(\boldsymbol{z}(\boldsymbol{x})),$$

where $\boldsymbol{g} : \mathbb{R}^K \to \mathbb{R}^K$ is applied vectorwise.



- **Output layer activation:**

$$f(\boldsymbol{x}) = o(\boldsymbol{w}^{(2)}\boldsymbol{h} + b^{(2)}),$$

where $\boldsymbol{w}^{(2)} \in \mathbb{R}^K$ and $o : \mathbb{R} \to \mathbb{R}$ if the output activation function.

Detour: Multiple Classes

For multi-class classification, we need multiple output units (one per class)

Each output estimates the conditional probability $p(y = c \mid \boldsymbol{x})$

Predicted class is the one with highest estimated probability

We'll see two activation functions suitable for this:

- Softmax activation
- Sparsemax activation

Softmax Activation

Let $\Delta^{C-1} \subseteq \mathbb{R}^C$ be the probability simplex

The typical activation function for multi-class classification is **softmax** : $\mathbb{R}^C \to \Delta^{C-1}$:

$$\boxed{\boldsymbol{o}(\boldsymbol{z}) = \textbf{softmax}(z) = \left[\frac{\exp(z_1)}{\sum_c \exp(z_c)}, \ldots, \frac{\exp(z_C)}{\sum_c \exp(z_c)}\right]}$$

- Strictly positive, sums to 1
- Resulting distribution has full support: $\textbf{softmax}(\boldsymbol{z}) > \mathbf{0}, \forall \boldsymbol{z}$
- A disadvantage if a *sparse* probability distribution is desired
- Common workaround: threshold and truncate

Sparsemax Activation (Martins and Astudillo, 2016)

A sparse-friendly alternative is **sparsemax** : $\mathbb{R}^C \to \Delta^{C-1}$, defined as:

$$\boxed{\boldsymbol{o}(\boldsymbol{z}) = \textbf{sparsemax}(\boldsymbol{z}) := \underset{\boldsymbol{p} \in \Delta^{C-1}}{\arg\min} \|\boldsymbol{p} - \boldsymbol{z}\|^2.}$$

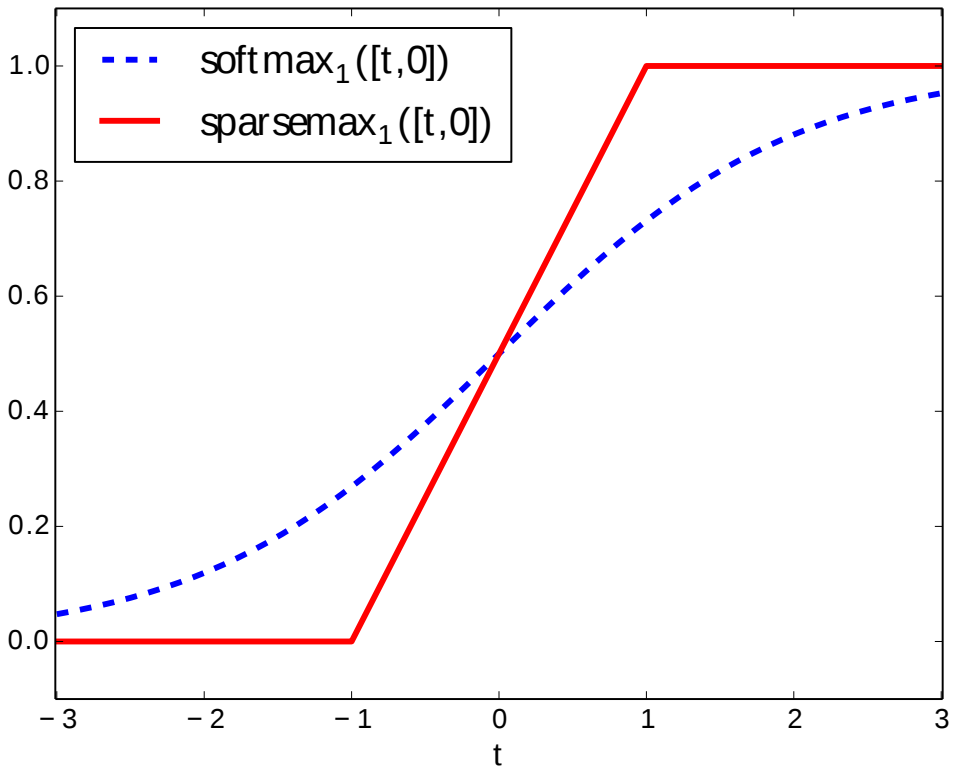
- In words: Euclidean projection of $\boldsymbol{z}$ onto the probability simplex
- Likely to hit the boundary of the simplex, in which case **sparsemax**($\boldsymbol{z}$) becomes sparse (hence the name)
- Retains many of the properties of softmax (e.g. differentiability), having in addition the ability of producing sparse distributions
- Projecting onto the simplex amounts to a soft-thresholding operation
- Efficient forward/backward propagation (more later)

Two Dimensions

- Parametrize $\boldsymbol{z} = (t, 0)$
- The 2D **softmax** is the logistic (sigmoid) function:

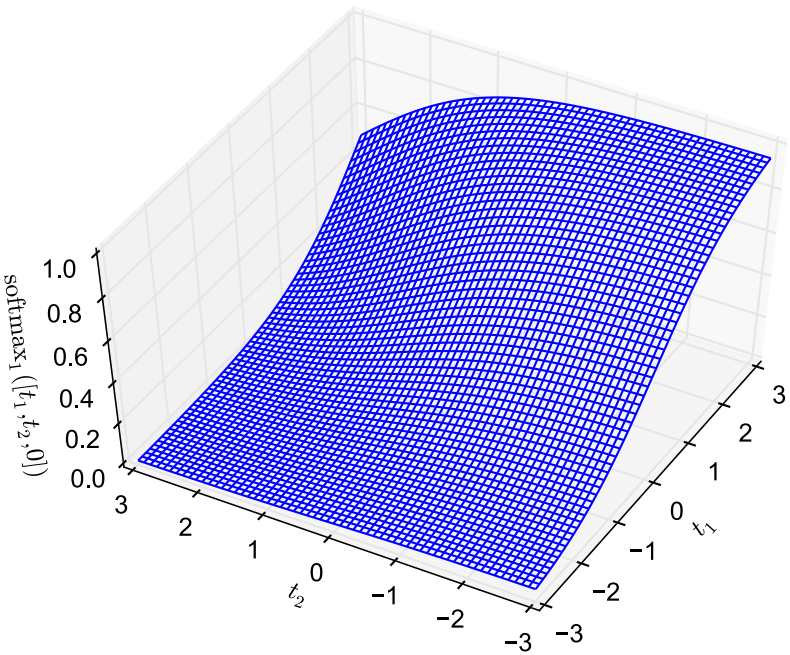
$$\textbf{softmax}_1(\boldsymbol{z}) = (1 + \exp(-t))^{-1}$$

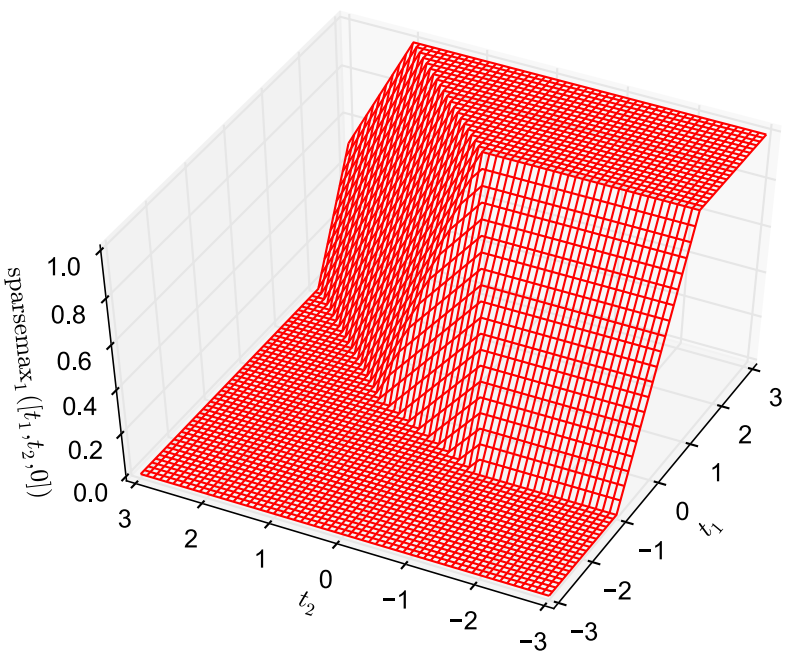
- The 2D **sparsemax** is the "hard" version of the sigmoid:



Three Dimensions

- Parameterize $\boldsymbol{z} = (t_1, t_2, 0)$ and plot $\textbf{softmax}_1(\boldsymbol{z})$ and $\textbf{sparsemax}_1(\boldsymbol{z})$ as a function of t_1 and t_2
- **sparsemax** is piecewise linear, but asymptotically similar to **softmax**





Multiple Hidden Layers

Now assume $L \geq 1$ hidden layers:

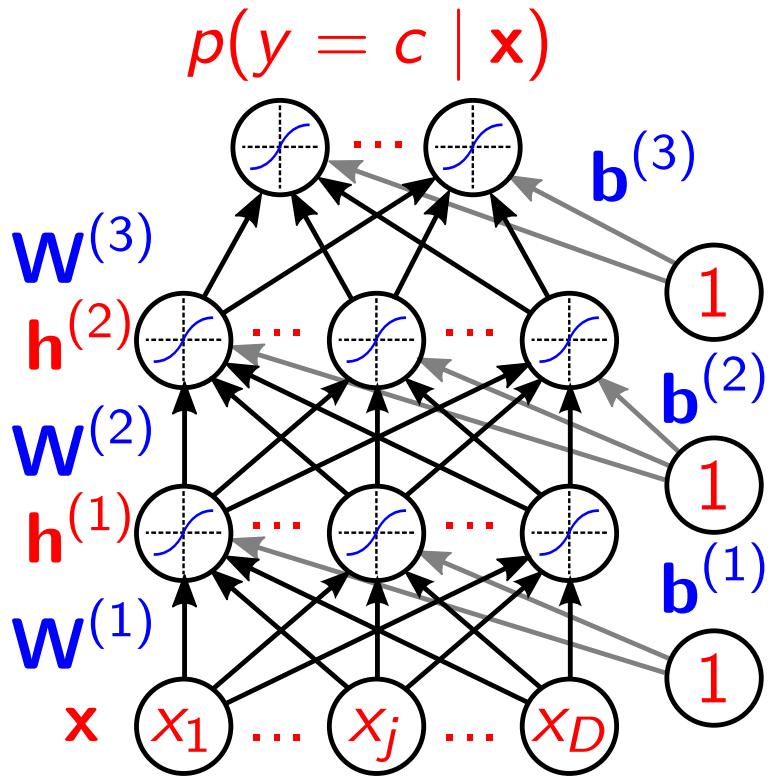
- **Hidden layer pre-activation** (define $\boldsymbol{h}^{(0)} = \boldsymbol{x}$ for convenience):

$$\boldsymbol{z}^{(\ell)}(\boldsymbol{x}) = \mathbf{W}^{(\ell)} \boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(1)},$$

  with $\mathbf{W}^{(\ell)} \in \mathbb{R}^{K_\ell \times K_{\ell-1}}$ and $\boldsymbol{b}^{(\ell)} \in \mathbb{R}^{K_\ell}$.

- **Hidden layer activation:**

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})).$$



- **Output layer activation:**

$$\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{o}(\boldsymbol{z}^{(L+1)}(\boldsymbol{x})) = \boldsymbol{o}(\mathbf{W}^{(L+1)} \boldsymbol{h}^{(L)} + \boldsymbol{b}^{(L+1)}).$$

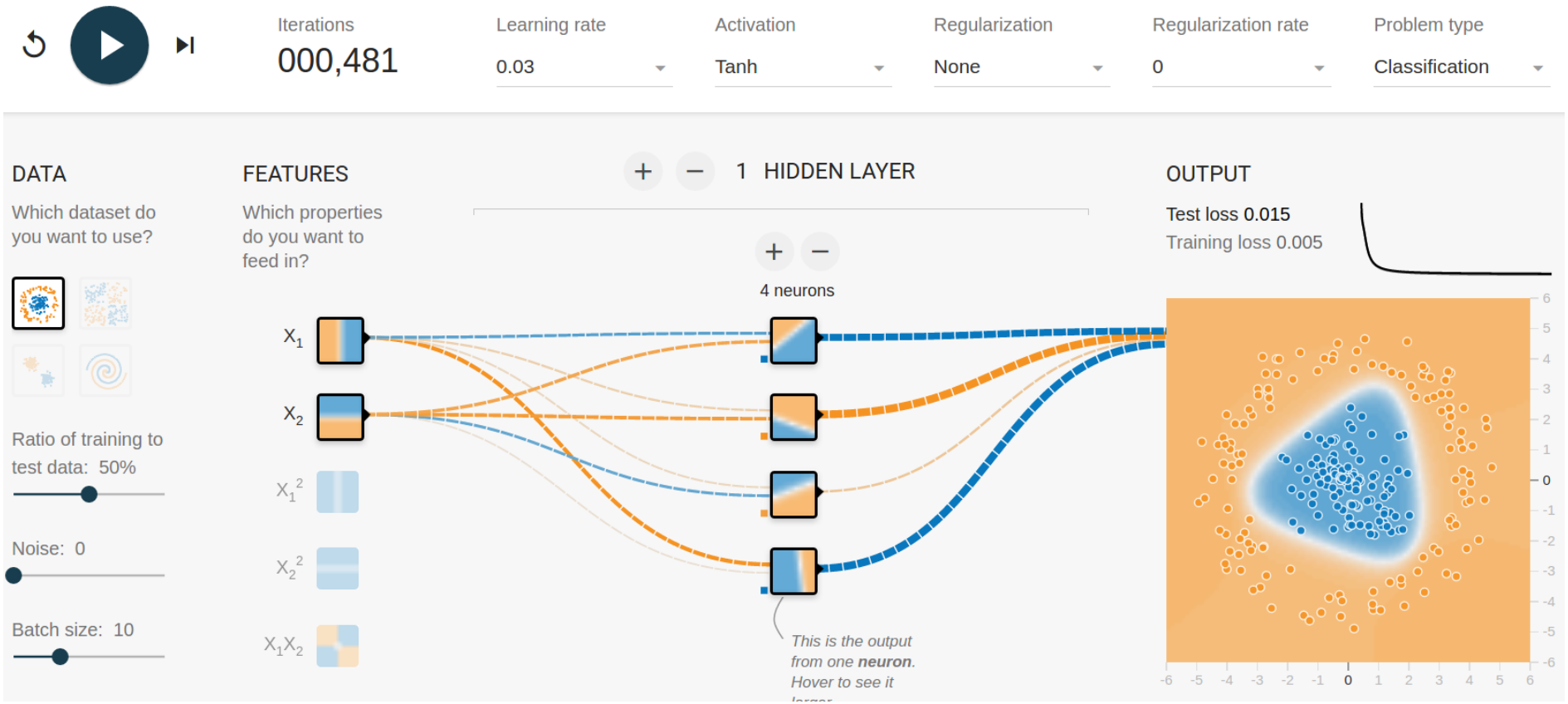
Universal Approximation Theorem

Theorem (Hornik et al. (1989))

A neural network with a single hidden layer and a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units.

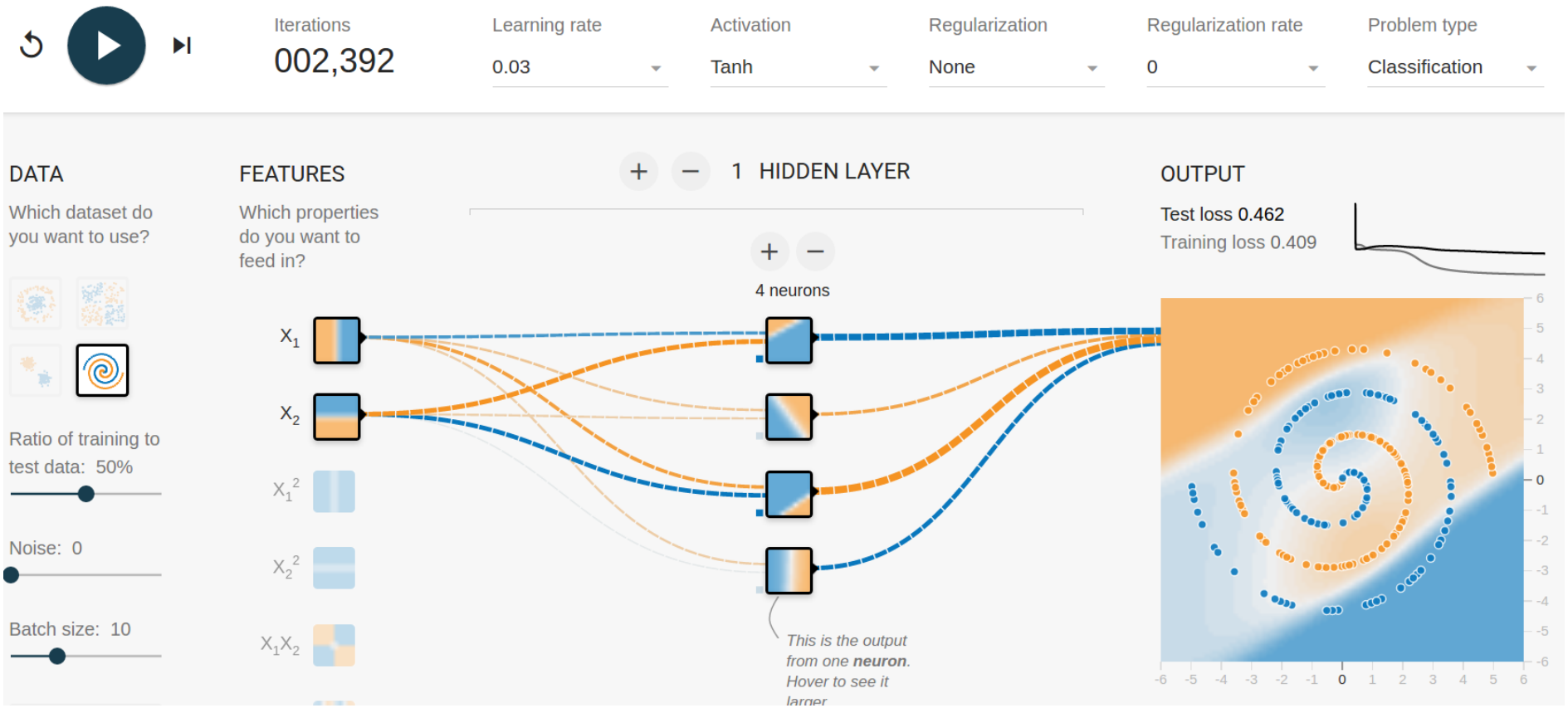
- First proved for the sigmoid case by Cybenko (1989), then to **tanh** and many other activation functions by Hornik et al. (1989)
- **Note:** may need exponentially many hidden units
- Deeper networks (more hidden layers) can provide more compact approximations

"Simple" Target Function, One Hidden Layer



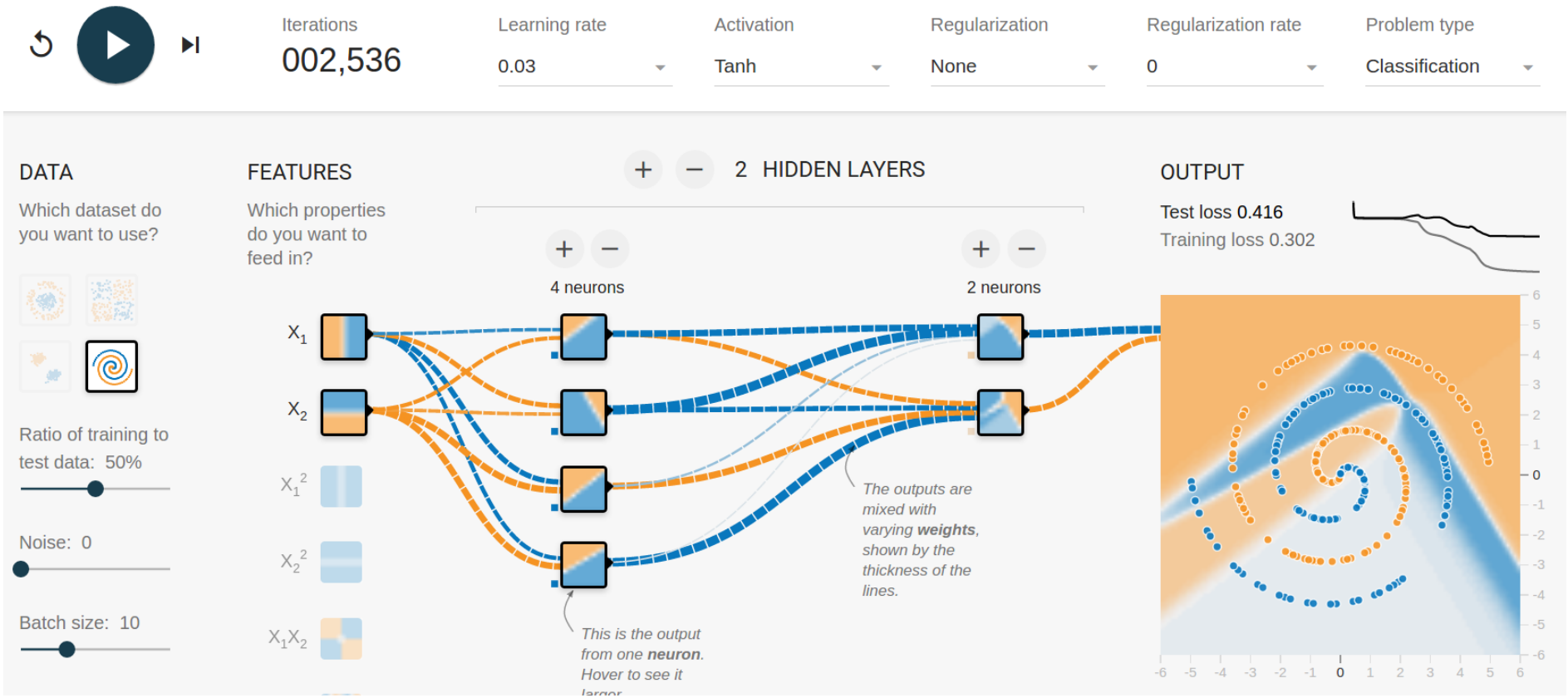
(http://playground.tensorflow.org)

Complex Target Function, One Hidden Layer



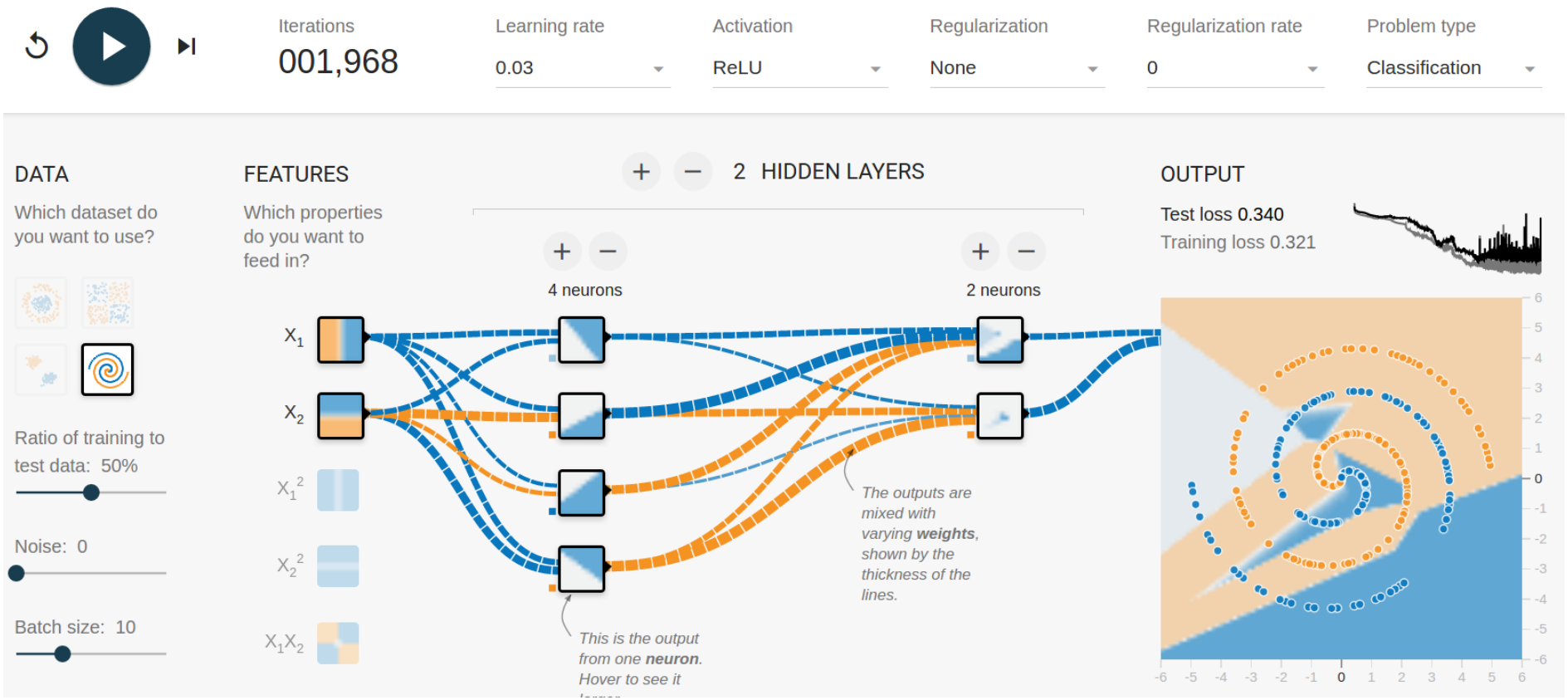
(http://playground.tensorflow.org)

Complex Target Function, Two Hidden Layers



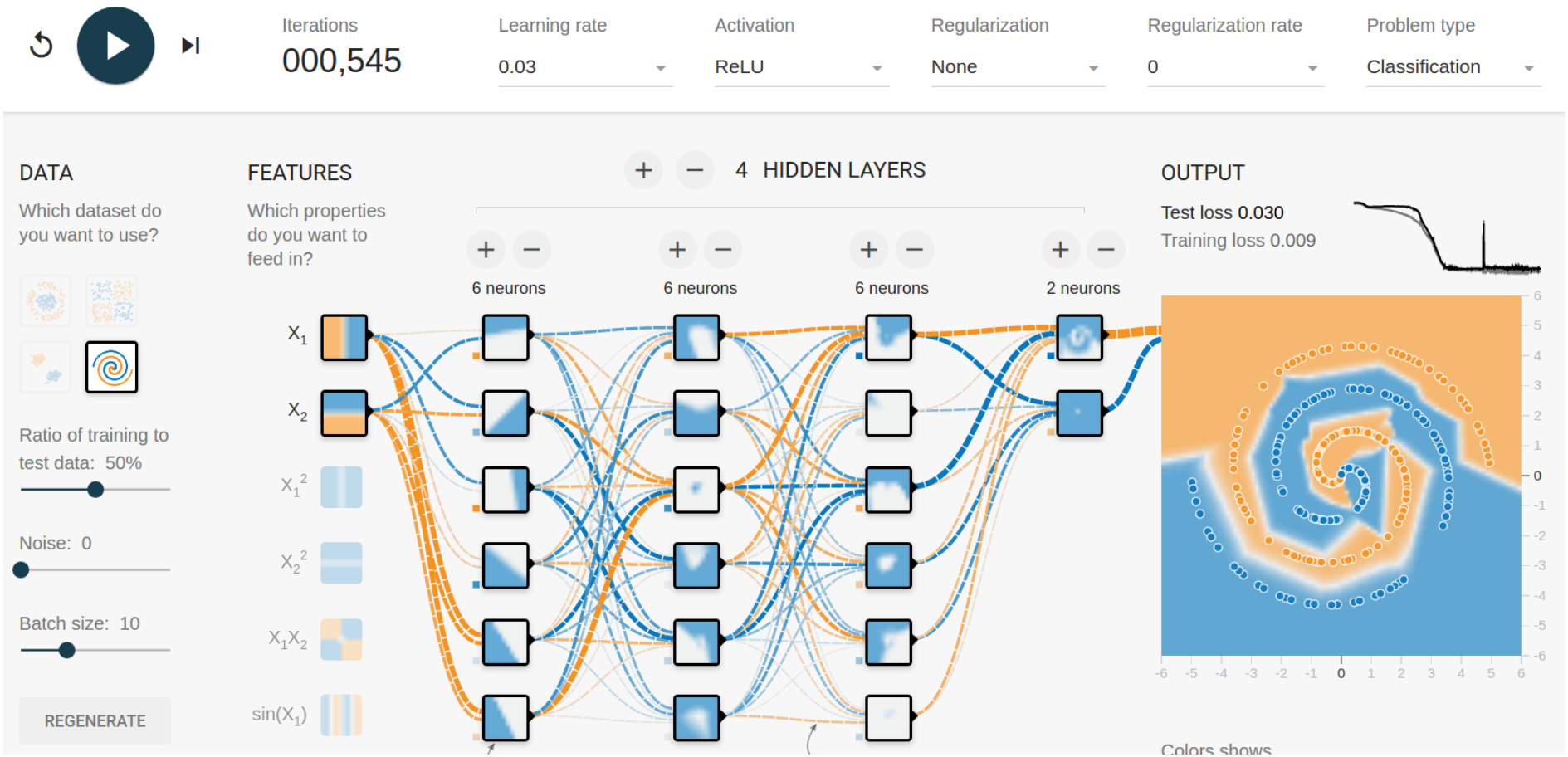
(http://playground.tensorflow.org)

Complex Target Function, Two Hidden Layers, ReLU



(http://playground.tensorflow.org)

Complex Target Function, Four Hidden Layers, ReLU



(http://playground.tensorflow.org)

Capacity of Neural Networks

Neural networks are excellent function approximators!

The universal approximation theorem is a nice result, but:

- We need a learning algorithm that finds the necessary parameter values
- ... and if we want to generalize, we need to control overfitting

Coffee Break

Outline

Training Neural Networks

Roadmap:

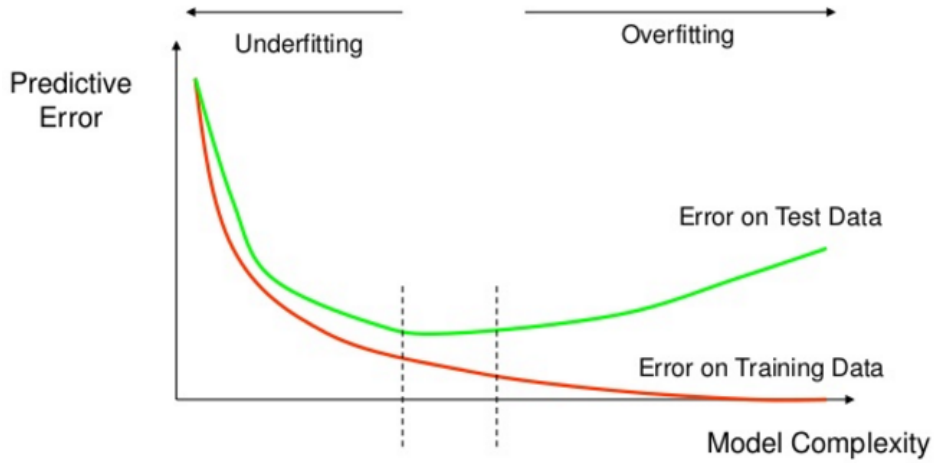
- Empirical risk minimization
- Stochastic gradient descent
- Gradient backpropagation
- Computation graph
- Regularization
- Initialization and other tricks of the trade

Empirical Risk Minimization

Goal: choose parameters $\boldsymbol{\theta} := \{(\mathbf{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ that minimize the following objective function:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{n=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$$

- $\Omega(\boldsymbol{\theta})$ is a regularizer
- $L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$ is a loss function
- λ is a regularization constant (an hyperparameter that needs to be tuned)



Stochastic Gradient Descent

- Batch gradient requires a full pass over the data before updating the weights—too slow!
- Stochastic gradient descent (SGD) approximates $\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta})$ by a "noisy gradient" based on a single example (a random $i \in [N]$):

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\boldsymbol{\theta}) \approx \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) := \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i).$$

The weights $\boldsymbol{\theta} = \{(\mathbf{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ are then updated as:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta})$$

We need:

- The loss function $L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$;
- The regularizer $\Omega(\boldsymbol{\theta})$ and its gradient;
- A procedure for computing the gradients $\nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$.

Loss Function

Should match as much as possible the metric we want to optimize at test time

Should be well-behaved (continuous, maybe smooth) to be amenable to optimization (this rules out the 0/1 loss)

Some examples:

- Squared loss for regression
- Negative log-likelihood (cross-entropy) for multi-class classification
- Sparsemax loss for multi-class and multi-label classification

Squared Loss

- The common choice for regression/reconstruction problems
- The neural network estimates $\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) \approx \boldsymbol{y}$
- We minimize the mean squared error:

$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),\boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}) - \boldsymbol{y}\|^2$$

- Loss gradient:

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta},y))}{\partial f_c(\boldsymbol{x};\boldsymbol{\theta})} = f_c(\boldsymbol{x};\boldsymbol{\theta}) - y_c$$

Negative Log-Likelihood (Cross-Entropy)

- The common choice for a softmax output layer
- The neural network estimates $f_c(\boldsymbol{x}; \boldsymbol{\theta}) \approx p(y = c \mid \boldsymbol{x})$
- We minimize the negative log-likelihood (also called cross-entropy):

$$\begin{aligned} L(\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta}), y) &= -\sum_c 1_{(y=c)} \log f_c(\boldsymbol{x}; \boldsymbol{\theta}) \\ &= -\log f_y(\boldsymbol{x}; \boldsymbol{\theta}) \\ &= -\log \textbf{softmax}(\boldsymbol{z}(\boldsymbol{x})), \end{aligned}$$

  where $\boldsymbol{z}$ is the output pre-activation.
- Loss gradient at output pre-activation:

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x}; \boldsymbol{\theta}, y))}{\partial z_c} = -(1_{y=c} - \textbf{softmax}_c(\boldsymbol{z}(\boldsymbol{x})))$$

Sparsemax Loss (Martins and Astudillo, 2016)

- The natural choice for a sparsemax output layer
- The neural network estimates $f_c(\boldsymbol{x};\boldsymbol{\theta}) \approx p(y = c \mid \boldsymbol{x})$ as a sparse distribution

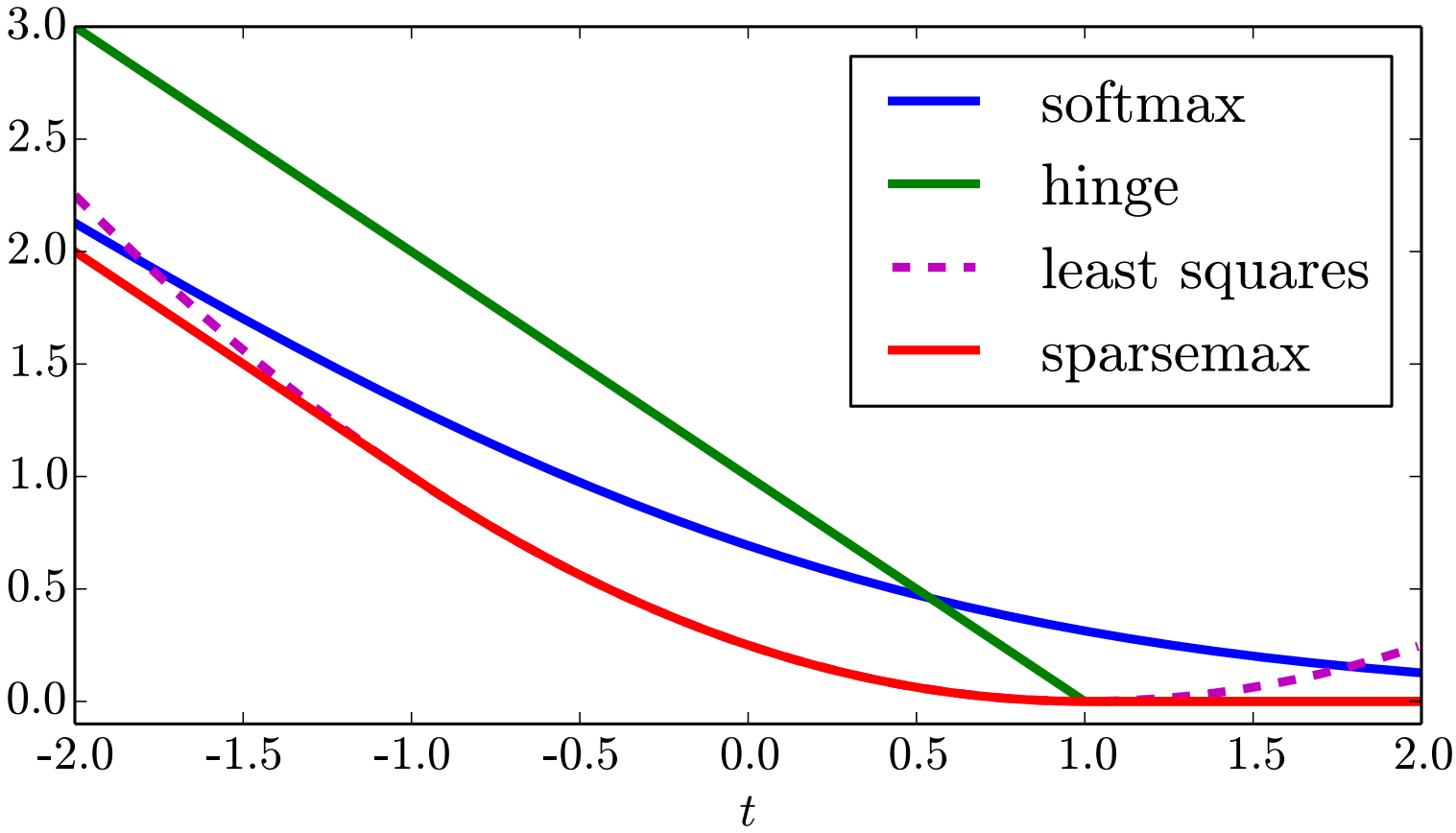
$$L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -z_c + \frac{1}{2}\sum_{j \in S(\boldsymbol{z})}(z_j^2 - \tau^2(\boldsymbol{z})) + \frac{1}{2},$$

  where $\boldsymbol{z}$ is the output pre-activation, $S(\boldsymbol{z})$ is the support of $p(y \mid \boldsymbol{x})$ and $\tau^2 : \mathbb{R}^K \rightarrow \mathbb{R}$ is the square of the threshold function (see Martins and Astudillo (2016) for details).
- Loss gradient at output pre-activation:

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}, y))}{\partial z_c} = -(1_{y=c} - \textbf{sparsemax}_c(\boldsymbol{z}(\boldsymbol{x})))$$

Classification Losses in Two Dimensions

- Let the correct label be $y = 1$ and define $t = z_1 - z_2$:



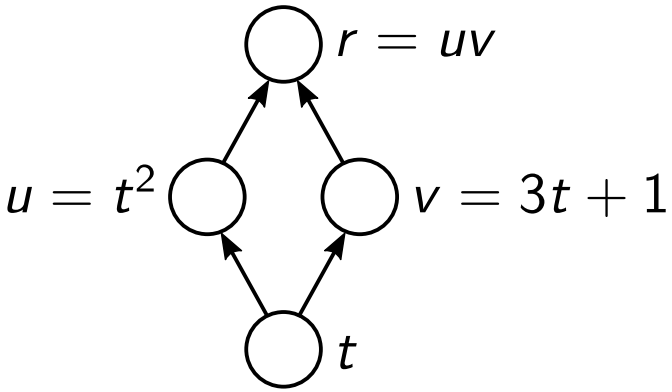
Gradient Computation

- Recall that we need to compute

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}_i(\boldsymbol{\theta}) := \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta}) + \nabla_{\boldsymbol{\theta}} L(f(\boldsymbol{x}_i; \boldsymbol{\theta}), y_i)$$

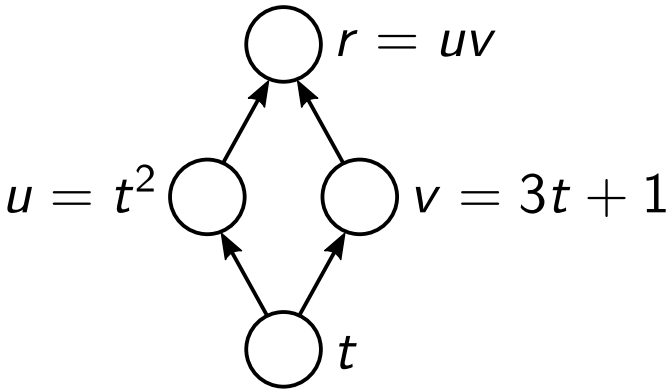
  for $\boldsymbol{\theta} = \{(\mathbf{W}^{(\ell)}, \boldsymbol{b}^{(\ell)})\}_{\ell=1}^{L+1}$ (the weights and biases at all layers)
- This will be done with the gradient backpropagation algorithm
- **Key idea:** use the chain rule for derivatives!

Recap: Chain Rule



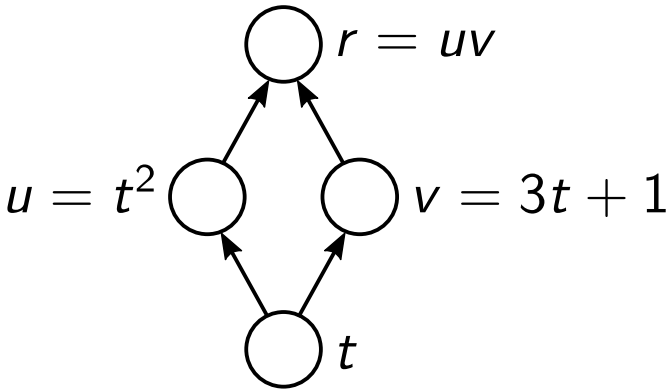
$$\frac{\partial r(t)}{\partial t} = ?$$

Recap: Chain Rule



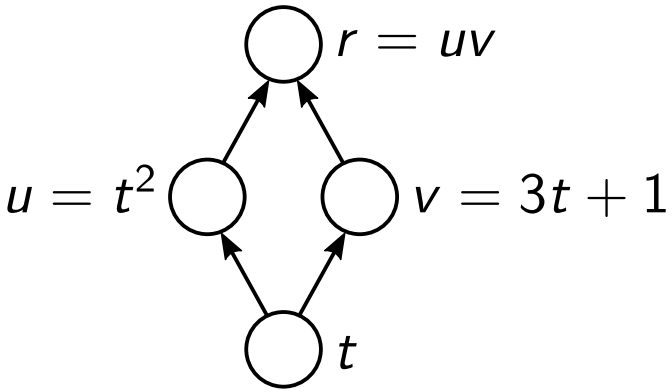
$$\frac{\partial r(t)}{\partial t} = \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t}$$

Recap: Chain Rule



$$\begin{aligned}\frac{\partial r(t)}{\partial t} &= \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t} \\ &= 2tv + 3u \\ &= 2t(3t+1) + 3t^2 = 9t^2 + 2t.\end{aligned}$$

Recap: Chain Rule



$$\begin{aligned}\frac{\partial r(t)}{\partial t} &= \frac{\partial r(u)}{\partial u}\frac{\partial u(t)}{\partial t} + \frac{\partial r(v)}{\partial v}\frac{\partial v(t)}{\partial t} \\ &= 2tv + 3u \\ &= 2t(3t+1) + 3t^2 = 9t^2 + 2t.\end{aligned}$$

- If a function $r(t)$ can be written as a function of intermediate results $q_i(t)$, then we have:

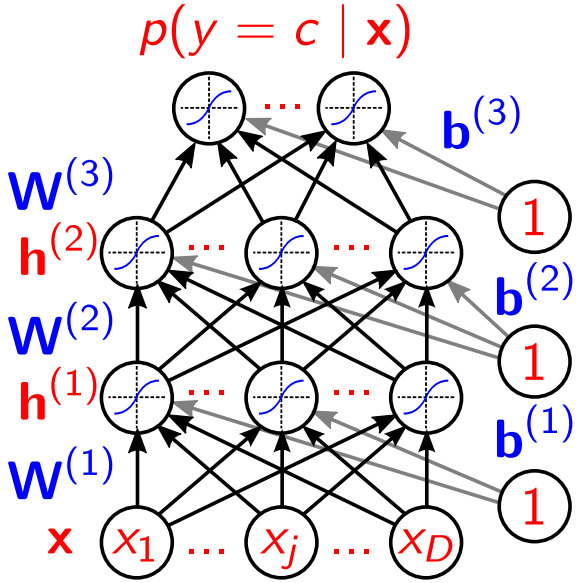
$$\frac{\partial r(t)}{\partial t} = \sum_i \frac{\partial r(t)}{\partial q_i(t)}\frac{\partial q_i(t)}{\partial t}$$

- We can invoke it by setting t to a output unit in a layer; $q_i(t)$ to the pre-activation in the layer above; and $r(t)$ to the loss function.

Hidden Layer Gradient

(Recap: $\boldsymbol{z}^{(\ell+1)} = \mathbf{W}^{(\ell+1)}\boldsymbol{h}^{(\ell)} + \boldsymbol{b}^{(\ell+1)}$)

$$\begin{aligned}\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial h_j^{(\ell)}} &= \sum_i \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell+1)}} \frac{\partial z_i^{(\ell+1)}}{\partial h_j^{(\ell)}} \\ &= \sum_i \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)}{\partial z_i^{(\ell+1)}} \boldsymbol{W}_{i,j}^{(\ell+1)}\end{aligned}$$

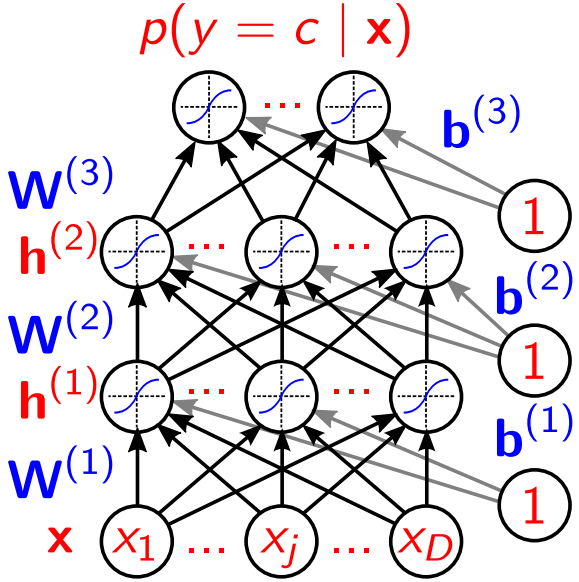


Hence $\nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y) = \boldsymbol{W}^{(\ell+1)^\top} \nabla_{\boldsymbol{z}^{(\ell+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}),y)$.

Hidden Layer Gradient (Before Activation)

(Recap: $h_j^{(\ell)} = g(z_j^{(\ell)})$, where $g : \mathbb{R} \to \mathbb{R}$ is the activation function)

$$\begin{aligned}\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z_j^{(\ell)}} &= \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} \frac{\partial h_j^{(\ell)}}{\partial z_j^{(\ell)}} \\ &= \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial h_j^{(\ell)}} g'(z_j^{(\ell)})\end{aligned}$$



Hence $\nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$.

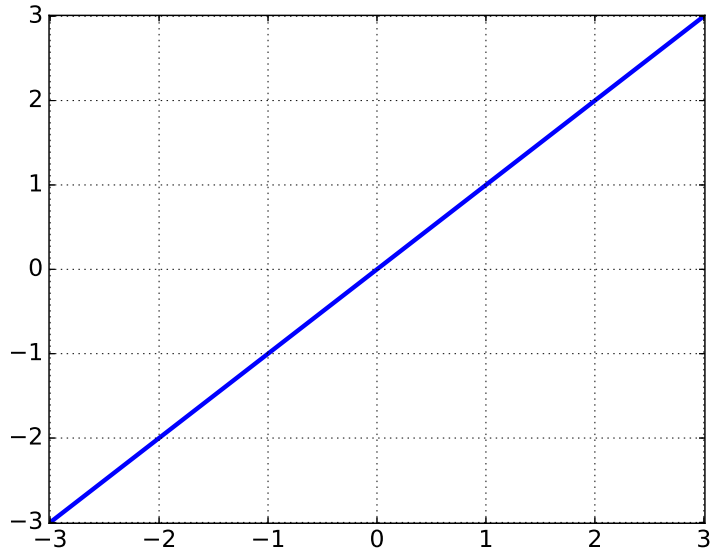
How to compute the derivative of the activation function $\boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$?

Linear Activation

$$\boxed{g(z) = z}$$

Derivative:

$$\boxed{g'(z) = 1}$$

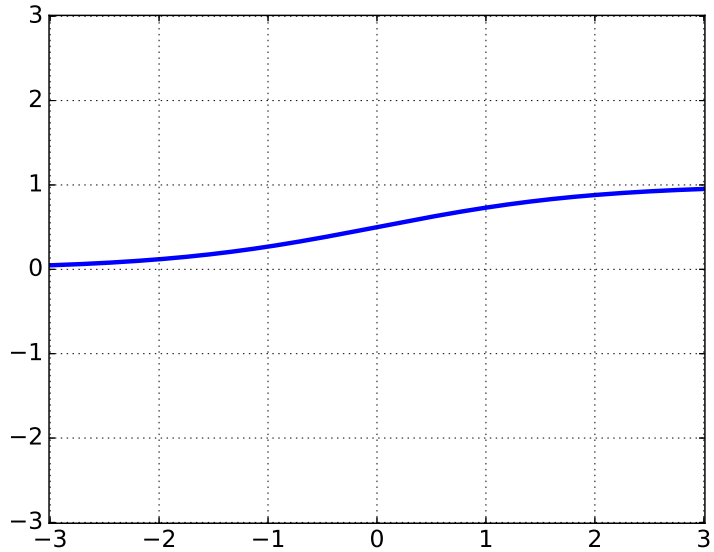


Sigmoid Activation

$$\boxed{g(z) = \sigma(z) = \frac{1}{1 + e^{-z}}}$$

Derivative:

$$\boxed{g'(z) = g(z)(1 - g(z))}$$

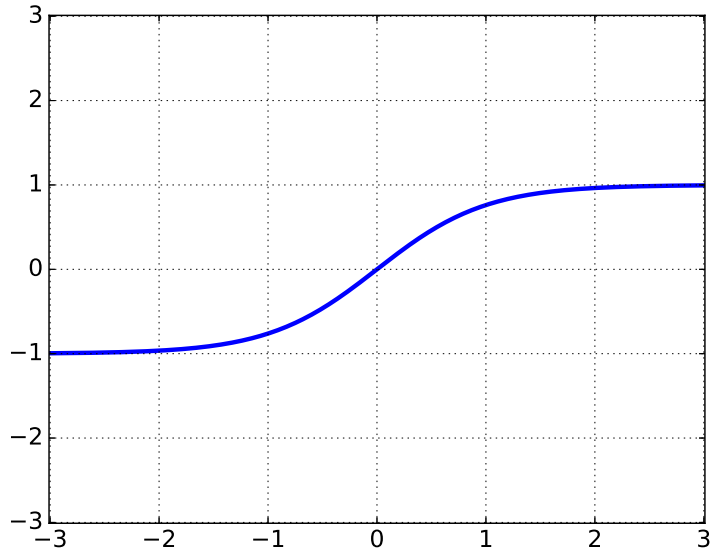


Hyperbolic Tangent Activation

$$g(z) = \textbf{tanh}(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Derivative:

$$g'(z) = 1 - g(z)^2$$

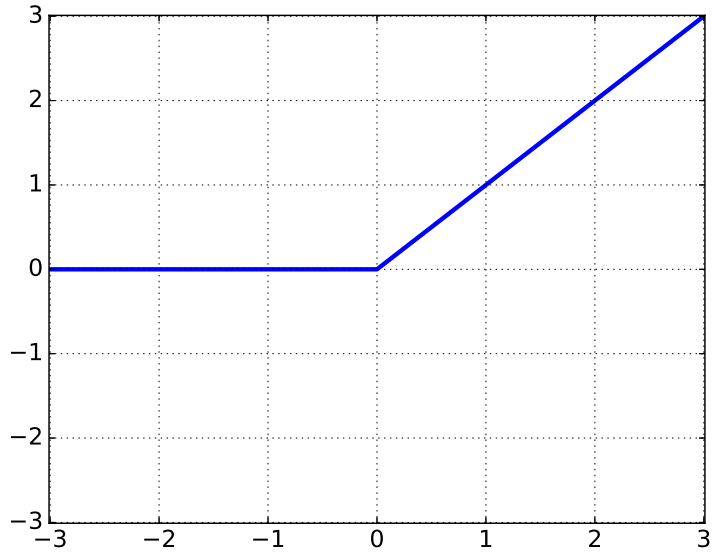


Rectified Linear Unit Activation (Glorot et al., 2011)

$$\boxed{g(z) = \textbf{relu}(z) = \max\{0, z\}}$$

Derivative (except for $z = 0$):

$$\boxed{g'(z) = 1_{z>0}}$$

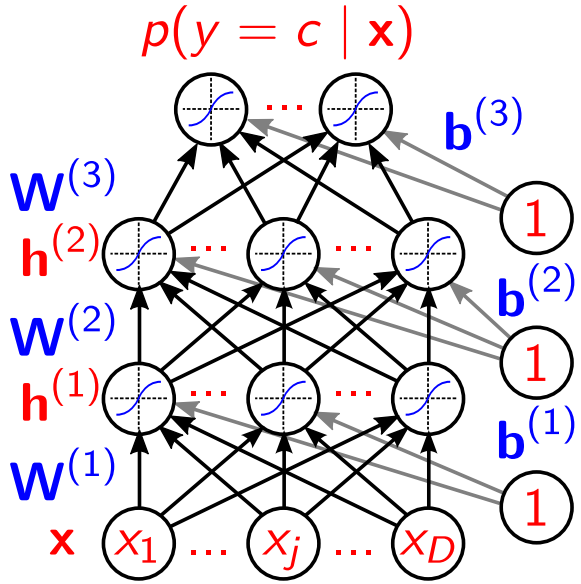


Parameter Gradient

(Recap: $\boldsymbol{z}^{(\ell)} = \mathbf{W}^{(\ell)} \boldsymbol{h}^{(\ell-1)} + \boldsymbol{b}^{(\ell)}$)

$$\frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial \mathbf{W}^{(\ell)}_{i,j}} = \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z^{(\ell)}_i} \frac{\partial z^{(\ell)}_i}{\partial \mathbf{W}^{(\ell)}_{i,j}}$$

$$= \frac{\partial L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)}{\partial z^{(\ell)}_i} h^{(\ell-1)}_j$$



Hence $\nabla_{\mathbf{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \boldsymbol{h}^{(\ell-1)\top}$

Similarly, $\nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$

Backpropagation

Compute output gradient (before activation):

$$\nabla_{\boldsymbol{z}^{(L+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -(\mathbf{1}_y - \boldsymbol{f}(\boldsymbol{x}))$$

for ℓ from $L+1$ to 1 **do**

Compute gradients of hidden layer parameters:

$$\begin{aligned} \nabla_{\mathbf{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) &= \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)\ \boldsymbol{h}^{(\ell-1)\top} \\ \nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) &= \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \end{aligned}$$

Compute gradient of hidden layer below:

$$\nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \boldsymbol{W}^{(\ell+1)\top} \nabla_{\boldsymbol{z}^{(\ell+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

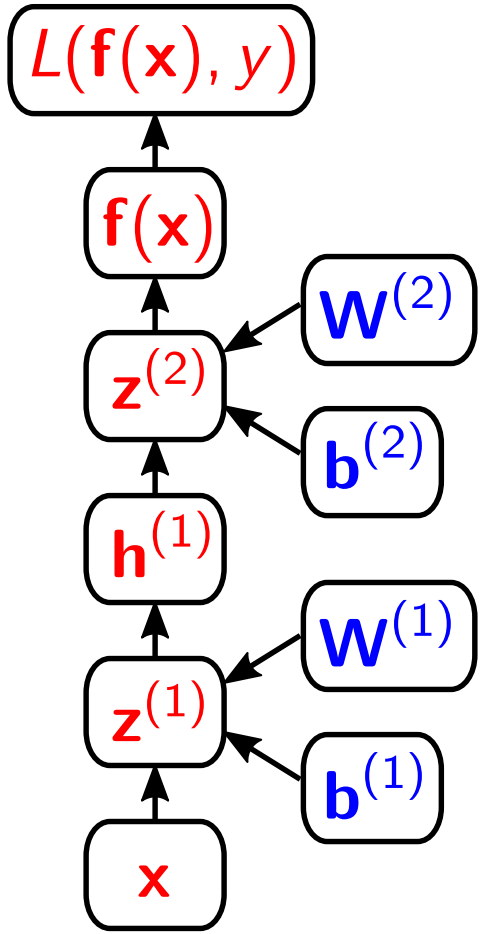
Compute gradient of hidden layer below (before activation):

$$\nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell)})$$

end for

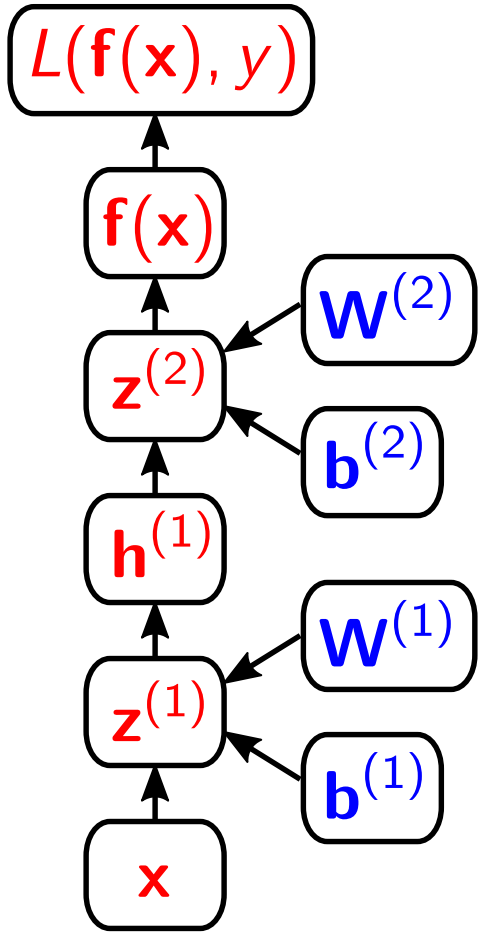
Computation Graph

- Forward propagation can be represented as a DAG
- Allows to implement forward propagation in a modular way
- Each box can be an object with a `fprop` method, that computes the value of the box given its children
- Calling the `fprop` method of each box in the right order (after a topological sort) yields forward propagation



Automatic Differentiation

- ... Also allows to implement backpropagation in a modular way
- Each box can also have a `bprop` method, that computes the loss gradient with respect to its children, given the loss gradient with respect to the output
- Can make use of cached computation done during the `fprop` method
- By calling the `bprop` method in reverse order, we get backpropagation (only need to reach the parameters)



Many Software Toolkits for Neural Networks

- Theano
- Tensorflow
- Torch
- Keras
- Caffe
- DyNet
- ...

All implement automatic differentiation.

Some Theano Code (Logistic Regression)

```python
import numpy
import theano
import theano.tensor as T
rng = numpy.random

N = 400                                   # training sample size
feats = 784                               # number of input variables

# generate a dataset: D = (input_values, target_class)
D = (rng.randn(N, feats), rng.randint(size=N, low=0, high=2))
training_steps = 10000

# Declare Theano symbolic variables
x = T.dmatrix("x")
y = T.dvector("y")

# initialize the weight vector w randomly
#
# this and the following bias variable b
# are shared so they keep their values
# between training iterations (updates)
w = theano.shared(rng.randn(feats), name="w")

# initialize the bias term
b = theano.shared(0., name="b")

print("Initial model:")
print(w.get_value())
print(b.get_value())
```

```python
# Construct Theano expression graph
p_1 = 1 / (1 + T.exp(-T.dot(x, w) - b))   # Probability that target = 1
prediction = p_1 > 0.5                    # The prediction thresholded
xent = -y * T.log(p_1) - (1-y) * T.log(1-p_1) # Cross-entropy loss function
cost = xent.mean() + 0.01 * (w ** 2).sum()# The cost to minimize
gw, gb = T.grad(cost, [w, b])             # Compute the gradient of the cost
                                          # w.r.t weight vector w and
                                          # bias term b
                                          # (we shall return to this in a
                                          # following section of this tutorial)

# Compile
train = theano.function(
          inputs=[x,y],
          outputs=[prediction, xent],
          updates=((w, w - 0.1 * gw), (b, b - 0.1 * gb)))
predict = theano.function(inputs=[x], outputs=prediction)

# Train
for i in range(training_steps):
    pred, err = train(D[0], D[1])

print("Final model:")
print(w.get_value())
print(b.get_value())
print("target values for D:")
print(D[1])
print("prediction on D:")
print(predict(D[0]))
```

Some Code in Tensorflow (Linear Regression)

```
import tensorflow as tf
import numpy as np

# Create 100 phony x, y data points in NumPy, y = x * 0.1 + 0.3
x_data = np.random.rand(100).astype(np.float32)
y_data = x_data * 0.1 + 0.3

# Try to find values for W and b that compute y_data = W * x_data + b
# (We know that W should be 0.1 and b 0.3, but TensorFlow will
# figure that out for us.)
W = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = W * x_data + b

# Minimize the mean squared errors.
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

# Before starting, initialize the variables.  We will 'run' this first.
init = tf.global_variables_initializer()

# Launch the graph.
sess = tf.Session()
sess.run(init)

# Fit the line.
for step in range(201):
    sess.run(train)
    if step % 20 == 0:
        print(step, sess.run(W), sess.run(b))

# Learns best fit is W: [0.1], b: [0.3]
```

Some Code in Keras (Multi-Layer Perceptron)

Multilayer Perceptron (MLP) for multi-class softmax classification:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
# here, 20-dimensional vectors.
model.add(Dense(64, input_dim=20, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(64, init='uniform'))
model.add(Activation('tanh'))
model.add(Dropout(0.5))
model.add(Dense(10, init='uniform'))
model.add(Activation('softmax'))

sgd = SGD(lr=0.1, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(X_train, y_train,
          nb_epoch=20,
          batch_size=16)
score = model.evaluate(X_test, y_test, batch_size=16)
```

Regularization

Recall that we're minimizing the following objective function:

$$\mathcal{L}(\boldsymbol{\theta}) := \lambda\Omega(\boldsymbol{\theta}) + \frac{1}{N}\sum_{n=1}^{N} L(\boldsymbol{f}(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)$$

It remains to define the regularizer and its gradient

We'll talk about:

- ℓ_2 regularization
- ℓ_1 regularization
- dropout regularization

ℓ_2 Regularization

- Gaussian prior on the weights
- **Note:** only the weights are regularized (not the biases)

$$\Omega(\boldsymbol{\theta}) = \frac{1}{2}\sum_{\ell} \|\mathbf{W}^{(\ell)}\|^2$$

- Gradient is:

$$\nabla_{\mathbf{W}^{(\ell)}}\Omega(\boldsymbol{\theta}) = \mathbf{W}^{(\ell)}$$

- This has the effect of a weight decay:

$$\begin{aligned}
\mathbf{W}^{(\ell)} &\leftarrow \mathbf{W}^{(\ell)} - \eta\nabla_{\mathbf{W}^{(\ell)}}\mathcal{L}_i(\boldsymbol{\theta}) \\
&= \mathbf{W}^{(\ell)} - \eta(\lambda\nabla_{\mathbf{W}^{(\ell)}}\Omega(\boldsymbol{\theta}) + \nabla_{\mathbf{W}^{(\ell)}}L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)) \\
&= (1-\eta\lambda)\mathbf{W}^{(\ell)} - \eta\nabla_{\mathbf{W}^{(\ell)}}L(f(\boldsymbol{x}_i;\boldsymbol{\theta}), y_i)
\end{aligned}$$

ℓ_1 Regularization

- Laplacian prior on the weights
- **Note:** only the weights are regularized (not the biases)

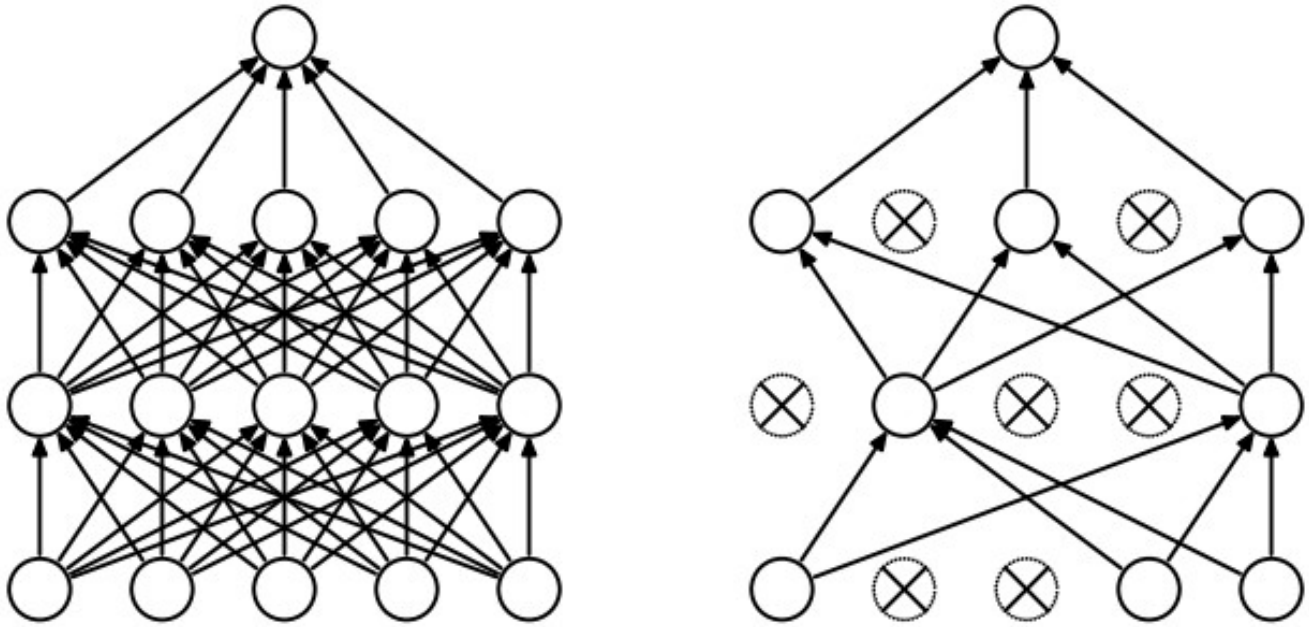
$$\Omega(\boldsymbol{\theta}) = \sum_{\ell} \|\mathbf{W}^{(\ell)}\|_1$$

- Gradient is:

$$\nabla_{\mathbf{W}^{(\ell)}} \Omega(\boldsymbol{\theta}) = \mathbf{sign}(\mathbf{W}^{(\ell)})$$

- Promotes sparsity of the weights

Dropout Regularization (Srivastava et al., 2014)



(a) Standard Neural Net

(b) After applying dropout.

Idea: During training, remove some hidden units stochastically

Dropout Regularization (Srivastava et al., 2014)

- Each hidden unit's output is set to 0 with probability p (e.g. $p = 0.5$)
- This prevents hidden units to co-adapt to other units, forcing them to be more generally useful
- At test time, keep all units, but multiply their outputs by $1 - p$
- Shown to be a form of adaptive regularization (Wager et al., 2013)
- Note: many software packages implement another variant, inverted dropout, where at training time the output of the units that were not dropped is divided by $1 - p$ and requires no change at test time

Implementation of Dropout

- This is usually implemented using random binary masks
- The hidden layer activations become (for $\ell = 1, \ldots, L$):

$$\boldsymbol{h}^{(\ell)}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{z}^{(\ell)}(\boldsymbol{x})) \odot \boldsymbol{m}^{(\ell)}$$

- Beats regular backpropagation on many datasets (Hinton et al., 2012)
- Other variants, e.g. DropConnect (Wan et al., 2013), Stochastic Pooling (Zeiler and Fergus, 2013)

Backpropagation with Dropout

Compute output gradient (before activation):

$$\nabla_{\boldsymbol{z}^{(L+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = -(\mathbf{1}_y - \boldsymbol{f}(\boldsymbol{x}))$$

for ℓ from $L+1$ to 1 **do**

Compute gradients of hidden layer parameters:

$$\nabla_{\mathbf{W}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \underbrace{\boldsymbol{h}^{(\ell-1)\top}}_{\text{includes } \boldsymbol{m}^{(\ell-1)}}$$

$$\nabla_{\boldsymbol{b}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Compute gradient of hidden layer below:

$$\nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \boldsymbol{W}^{(\ell+1)\top} \nabla_{\boldsymbol{z}^{(\ell+1)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Compute gradient of hidden layer below (before activation):

$$\nabla_{\boldsymbol{z}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) = \nabla_{\boldsymbol{h}^{(\ell)}} L(\boldsymbol{f}(\boldsymbol{x};\boldsymbol{\theta}), y) \odot \boldsymbol{g}'(\boldsymbol{z}^{(\ell)}) \odot \boldsymbol{m}^{(\ell-1)}$$

end for

Initialization

Initialize all biases to zero

For weights:

- Cannot initialize to zero with **tanh** activation (the gradients would also be zero and we would reach a saddle point)
- Cannot initialize the weights to the same value (need to break the symmetry)
- Random initialization (Gaussian, uniform), sampling around 0 to break symmetry
- For ReLU activations, the mean should be a small positive number
- Variance cannot be too high, otherwise all neuron activations will be saturated

"Glorot Initialization"

- Recipe from Glorot and Bengio (2010):

$$\mathbf{W}_{i,j}^{(\ell)} \sim U[-t, t], \text{ with } t = \frac{\sqrt{6}}{\sqrt{K^{(\ell)} + K^{(\ell-1)}}}$$

- Works well in practice with **tanh** and sigmoid activations

Training, Validation, and Test Sets

Split datasets in training, validation, and test partitions.

- Training set serves to train the model
- Validation set serves to tune hyperparameters (learning rate, number of hidden units, regularization coefficient, dropout probability, best epoch, etc.)
- Test set serves to estimate the generalization performance

Hyperparameter Tuning: Grid Search, Random Search

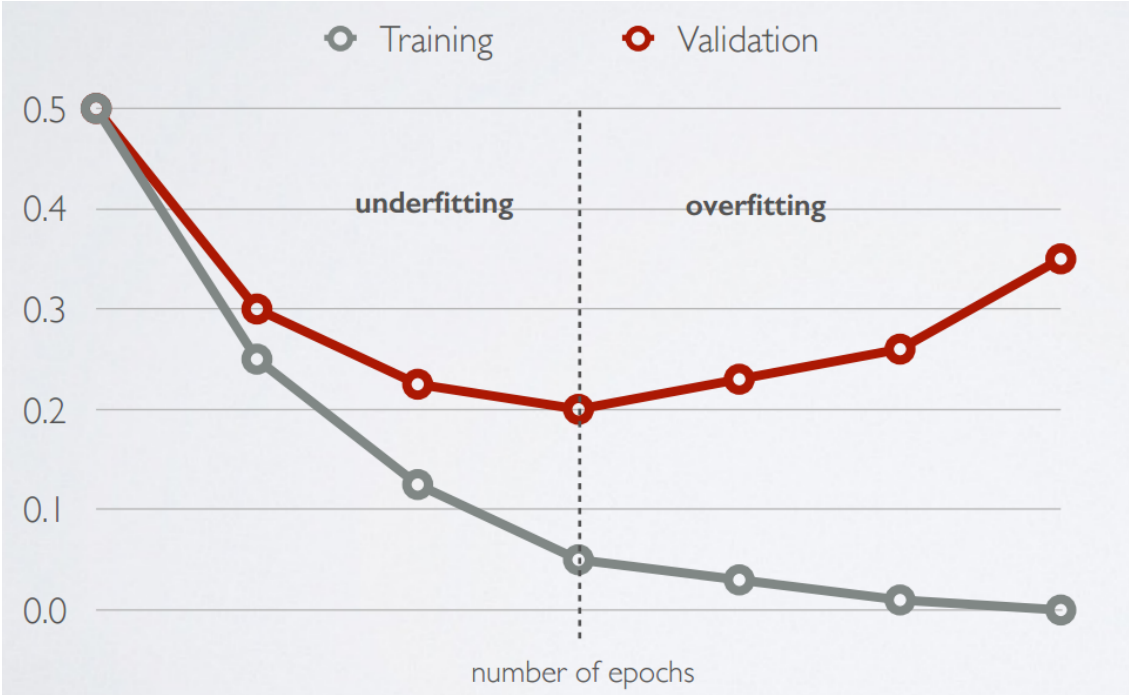
Search for the best configuration of the hyperparameters:

- Grid search: specify a set of values we want to test for each hyperparameter, and try all configurations of these values
- Random search: specify a distribution over the values of each hyper-parameter (e.g. uniform in some range) and sample independently each hyper-parameter to get configurations
- Bayesian optimization and learning to learn (Snoek et al., 2012)

We can always go back and fine-tune the grid/distributions if necessary

Early Stopping

- To select the number of epochs, stop training when validation error increases (with some look ahead)
- One common strategy (with SGD) is to halve the learning rate for every epoch where the validation error increases



(Image credit: Hugo Larochelle)

Tricks of the Trade

- Normalization of the data
- Decaying the learning rate
- Mini-batches
- Adaptive learning rates
- Gradient checking
- Debugging on a small dataset

Normalization of the Data

- For each input dimension: subtract the training set mean and divide by the training set standard deviation
- This makes each input dimension have zero mean, unit variance
- This can speed up training (in number of epochs)
- Doesn't work for sparse inputs (destroys sparsity)

Decaying the Learning Rate

In SGD, as we get closer to a local minimum, it makes sense to take smaller update steps (to avoid diverging)

- Start with a large learning rate (say 0.1)
- Keep it fixed while validation error keeps improving
- Divide by 2 and go back to the previous step

Mini-Batches

- Instead of updating after a single example, can aggregate a mini-batch of examples (e.g. 50–200 examples) and compute the averaged gradient for the entire mini-batch
- Less noisy than vanilla SGD
- Can leverage matrix-matrix computations (or tensor computations)
- Large computational speed-ups in GPUs (since computation is trivially parallelizable accross the mini-batch and we can exhaust the GPU memory)

Adaptive Learning Rates

Instead of using the same step size for all parameters, have one learning rate per parameter

- Adagrad (Duchi et al., 2011): learning rates are scaled by the square root of the cumulative sum of squared gradients

$$\eta^{(t)} = \eta^{(t-1)} + (\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y))^2, \qquad \bar{\nabla}_{\boldsymbol{\theta}}^{(t)} = \frac{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y)}{\sqrt{\eta^{(t)} + \epsilon}}$$

- RMSprop (Tieleman and Hinton, 2012): instead of cumulative sum, use exponential moving average

$$\eta^{(t)} = \beta\eta^{(t-1)} + (1-\beta)(\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y))^2, \qquad \bar{\nabla}_{\boldsymbol{\theta}}^{(t)} = \frac{\nabla_{\boldsymbol{\theta}} L(\boldsymbol{f}(\boldsymbol{x}), y)}{\sqrt{\eta^{(t)} + \epsilon}}$$

- Adam (Kingma and Ba, 2014): combine RMSProp with momentum

Gradient Checking

- If the training loss is not decreasing even with a very small learning rate, there's likely a bug in the gradient computation
- To debug your implementation of `fprop`/`bprop`, compute the "numeric gradient," a finite difference approximation of the true gradient:

$$\frac{\partial f(x)}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon}$$

Debugging on a Small Dataset

- Extract a small subset of your training set (e.g. 50 examples)
- Monitor your training loss in this set
- You should be able to overfit in this small training set
- If not, see if some units are saturated from the very first iterations (if they are, reduce the initialization variance or properly normalize your inputs)
- If the training error is bouncing up and down, decrease the learning rate

Outline

Representation Learning

One of the greatest features of neural networks is their ability to learn representations

Deeper neural networks learn coarse-to-fine representation layers

Roadmap:

- Hierarchical compositionality
- Distributed representations
- Unsupervised pre-training
- Auto-encoders
- Word embeddings

Hierarchical Compositionality

Vision:

- pixels $\rightarrow$ edge $\rightarrow$ texton $\rightarrow$ motif $\rightarrow$ part $\rightarrow$ object $\rightarrow$ scene

Speech:

- audio sample $\rightarrow$ spectral band $\rightarrow$ formant $\rightarrow$ motif $\rightarrow$ phone $\rightarrow$ word

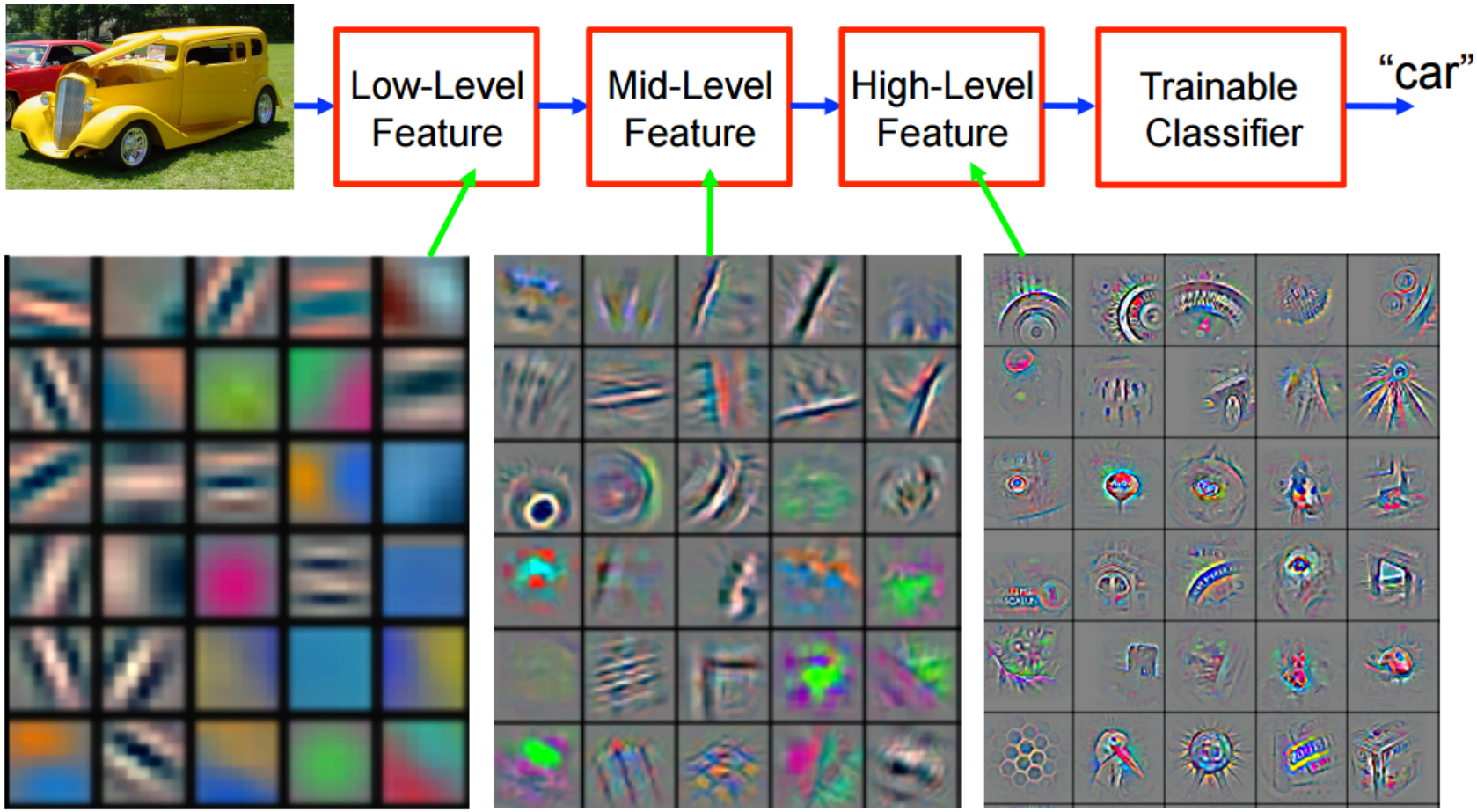
Text:

- character $\rightarrow$ word $\rightarrow$ phrase $\rightarrow$ sentence $\rightarrow$ story

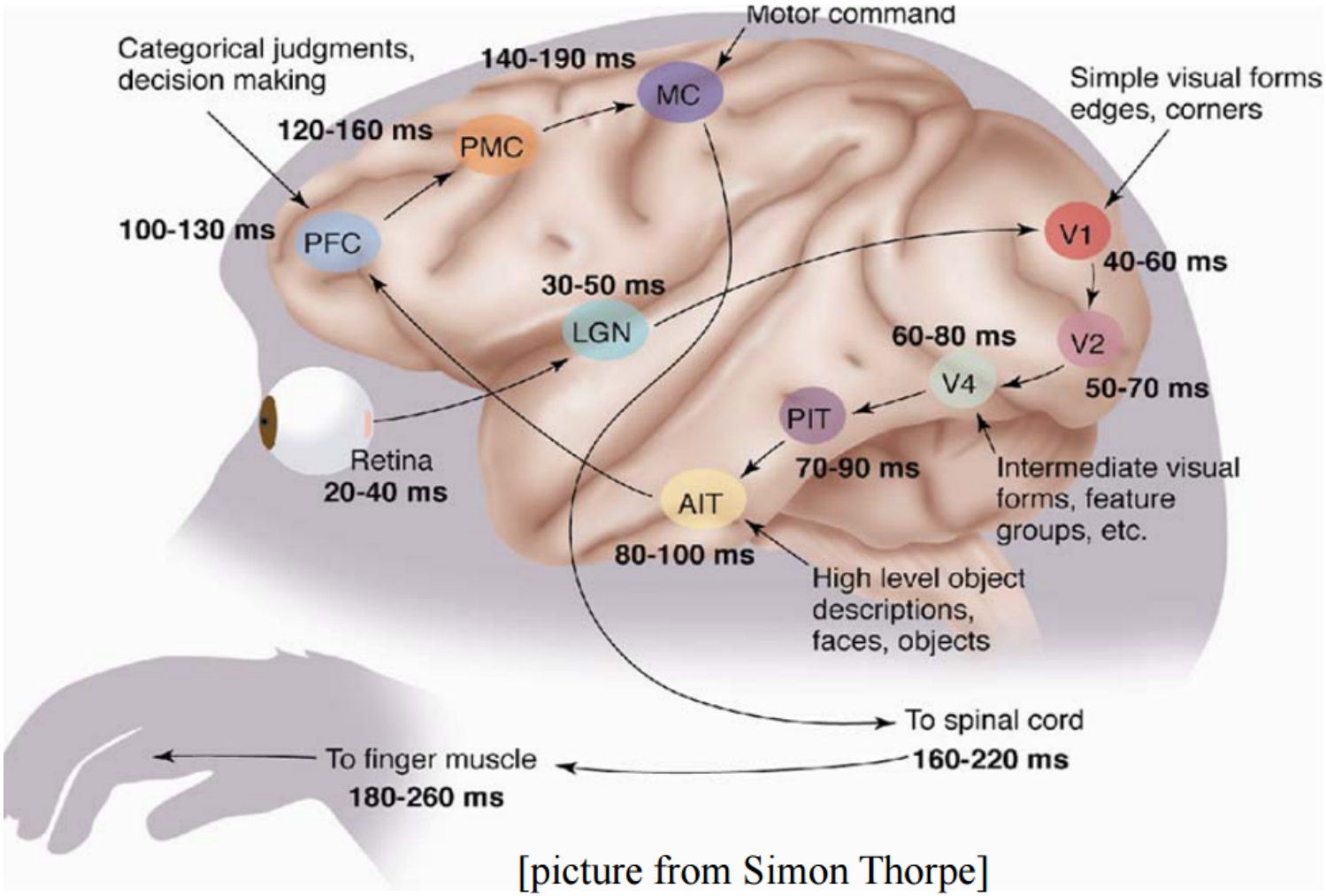
(Slide inspired by Marc'Aurelio Ranzato and Yann LeCun)

Hierarchical Compositionality

Feature visualization of convolutional net trained on ImageNet from Zeiler and Fergus (2013):



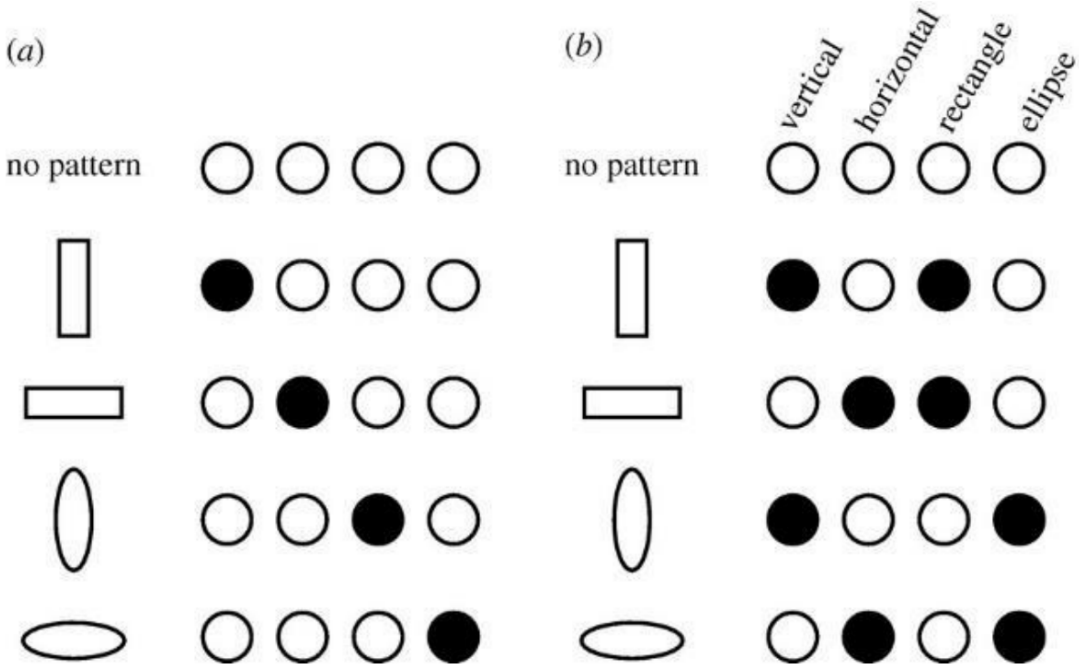
The Mammalian Visual Cortex is Hierarchical



(Slide inspired by Marc'Aurelio Ranzato and Yann LeCun)

Distributed Representations (Hinton, 1984)

- Compare one-hot representations (one dimension per object) with distributed representations (one dimension per property):

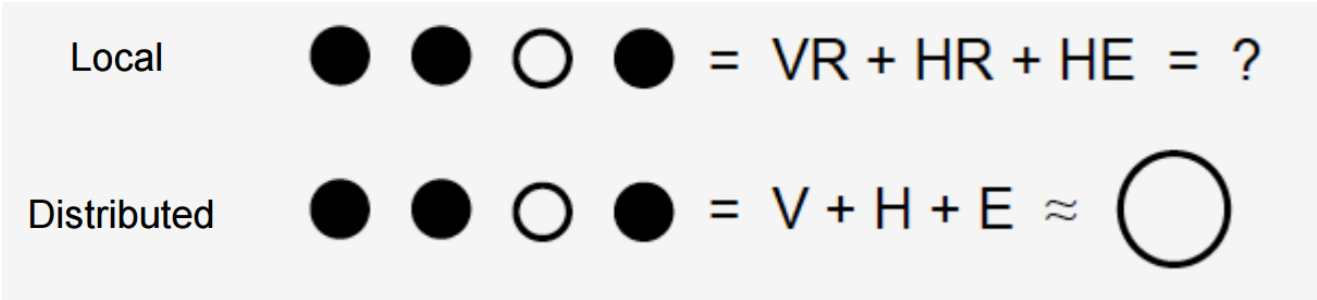


(Slide inspired by Moontae Lee and Dhruv Batra)

- **Key idea:** no single neuron "encodes" everything; groups of neurons (e.g. in the same hidden layer) work together

The Power of Distributed Representations

- Distributed representations are more powerful, as they can generalize to unseen objects in a meaningful way:



(Slide inspired by Moontae Lee and Dhruv Batra)

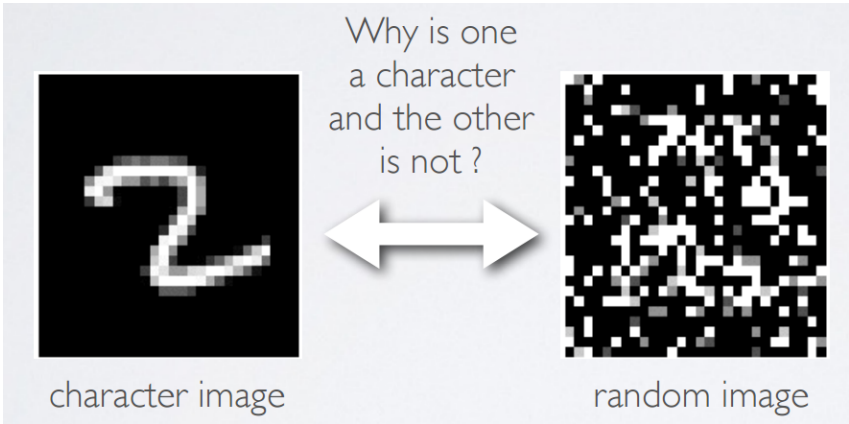
Unsupervised Pre-Training (Erhan et al., 2010)

Training deep networks (with many hidden layers) can be challenging

This has been a major difficulty with neural networks for a long time

Solution: initialize hidden layers using unsupervised learning:

- Force network to represent latent structure of input distribution
- Encourage hidden layers to encode that structure
- **Examples:** auto-encoders, restricted Boltzmann machines



(Image credit: Hugo Larochelle)

Auto-Encoders

An auto-encoder is a feed-forward neural network trained to reproduce its input at the output layer

Encoder:

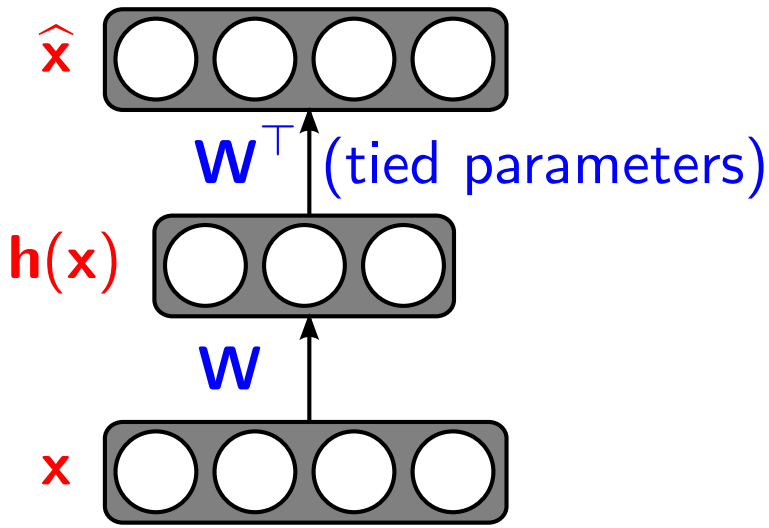
$$\boldsymbol{h}(x) = \boldsymbol{g}(\mathbf{W}\boldsymbol{x} + \boldsymbol{b})$$

Decoder:

$$\widehat{\boldsymbol{x}} = \mathbf{W}^\top \boldsymbol{h}(\boldsymbol{x}) + \boldsymbol{c}$$

Loss function (for real-valued inputs):

$$L(\widehat{\boldsymbol{x}}; \boldsymbol{x}) = \frac{1}{2}\|\widehat{\boldsymbol{x}} - \boldsymbol{x}\|^2$$



Generalizes PCA (recovered if activation function $\boldsymbol{g}$ is linear)!

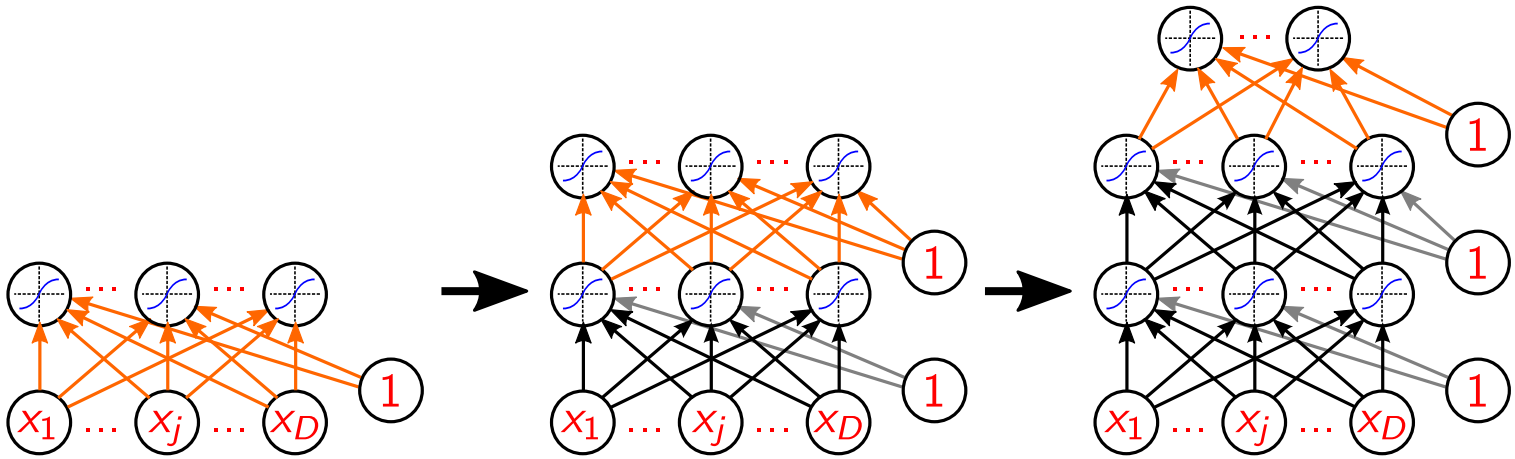
Some Variants of Auto-Encoders

- **Sparse auto-encoders:** use many hidden units, and add a ℓ_1 regularization term to encourage sparse representations of the input
- **Denoising auto-encoders:** regularize by adding noise to the input; the goal is to learn a smooth representation function that allows to output the denoised input (inspired by image denoising)
- **Stacked auto-encoders:** stack several auto-encoders on top of each other
- **Variational auto-encoders:** a generative probabilistic model that minimizes a variational bound (more later)

Unsupervised Pre-Training (Erhan et al., 2010)

A greedy, layer-wise procedure:

- train one layer at a time, from first to last, with unsupervised criterion (e.g. an auto-encoder)
- fix the parameters of previous hidden layers
- previous layers viewed as feature extraction



Pre-training initializes the parameters in a region such that the near local optima overfit less the data.

Fine-Tuning

Once all layers are pre-trained:

- add output layer
- train the whole network using supervised learning

Supervised learning is performed as in a regular feed-forward network:

- forward propagation, backpropagation and update
- all parameters are "tuned" for the supervised task at hand
- representation is adjusted to be more discriminative

Word Representations (Embeddings)

Distributional similarity based representations: represent a word by means of its neighbors

- *"You shall know a word by the company it keeps"* (J. R. Firth, 1957)
- One of the most successful ideas of modern statistical NLP

How do we obtain lower dimensional vector representations of words?

- **Method 1:** Factorization of a co-occurrence word/context matrix (latent semantic analysis, etc.)
- **Method 2:** Directly learn low-dimensional vectors by training a network to *predict* the context of a given word (word2vec)

Word2vec (Mikolov et al., 2013) follows previous ideas of Bengio et al. (2003) and Collobert et al. (2011), but in a simpler and faster model

Word Vectors

Word2vec comes with two variants:

- **Skip-gram** (our focus): predict surrounding context words in a window of length m of every word
- **Continuous bag-of-words** (CBOW): predict the central word from the context

Objective function of skip-gram: maximize the log probability of any context word given the current center word:

$$J(\boldsymbol{\theta}) = \frac{1}{T} \sum_{t=1}^{T} \sum_{-m \leq j \leq m,\ j \neq 0} \log p(w_{t+j} \mid w_t)$$

- Define $p(w_{t+j} = o \mid w_t = c) \propto \exp(\boldsymbol{u}_o \cdot \boldsymbol{v}_c)$ (a log-linear model)
- Every word has two vectors!

Negative Sampling

- With large vocabularies this objective function is not scalable and would train too slowly (requires a softmax over the entire vocabulary)
- **Workaround:** negative sampling: train binary logistic regressions for a true pair (center word and word in its context window) and a couple of random pairs (the center word with a random word)

$$J_t(\boldsymbol{\theta}) = \log \sigma(\boldsymbol{u}_o \cdot \boldsymbol{v}_c) + \sum_{i=1}^{k} \log \sigma(-\boldsymbol{u}_{j_i} \cdot \boldsymbol{v}_c), \qquad j_i \sim P(w)$$

Linear Relationships (Mikolov et al., 2013)

- These representations are very good at encoding dimensions of similarity!
- Word analogies can be solved quite well just by doing vector subtraction in the embedding space
- Syntactically:

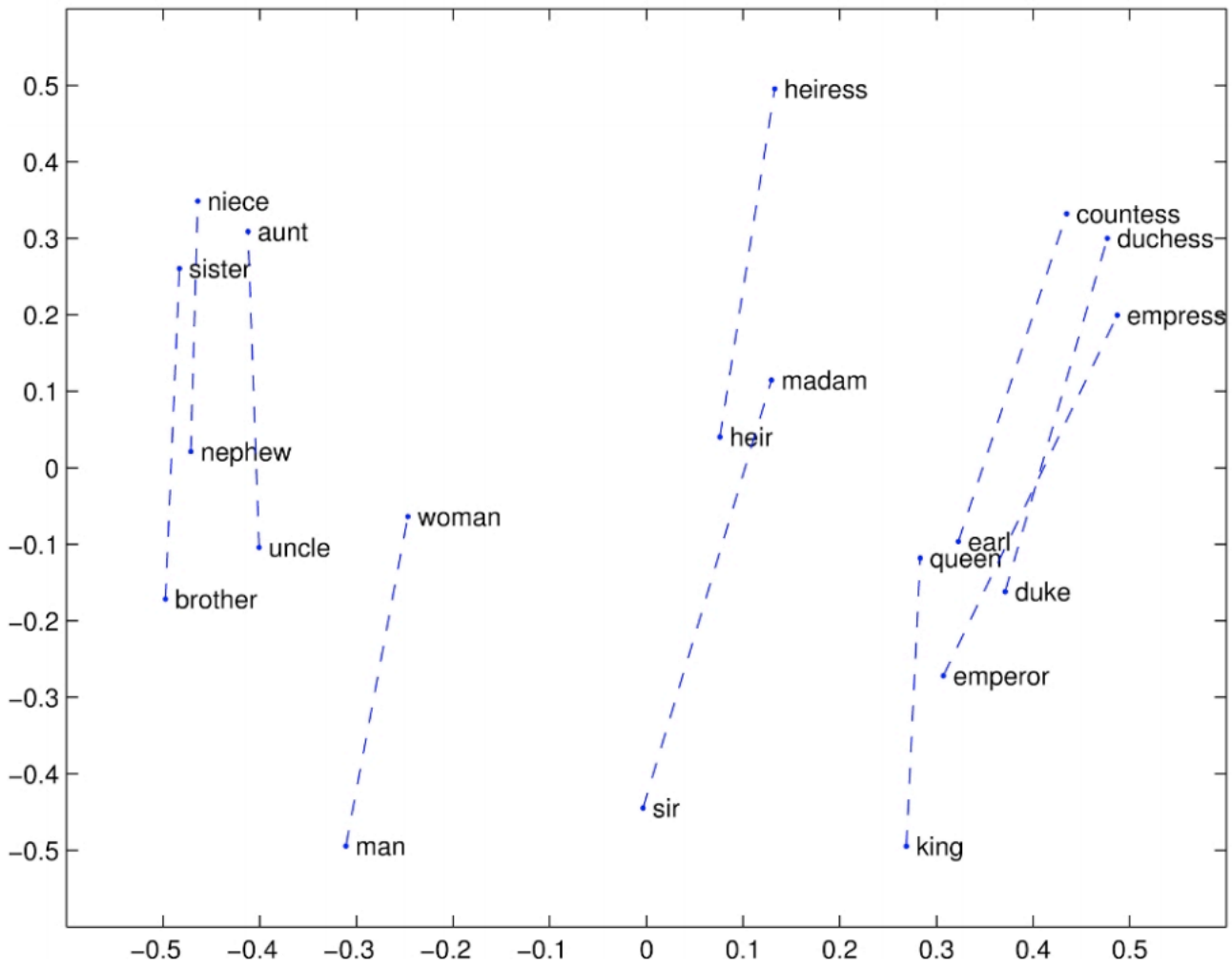
$$\boldsymbol{x}_{\text{apple}} - \boldsymbol{x}_{\text{apples}} \approx \boldsymbol{x}_{\text{car}} - \boldsymbol{x}_{\text{cars}} \approx \boldsymbol{x}_{\text{family}} - \boldsymbol{x}_{\text{families}}$$

- Semantically:

$$\begin{aligned} \boldsymbol{x}_{\text{shirt}} - \boldsymbol{x}_{\text{clothing}} &\approx \boldsymbol{x}_{\text{chair}} - \boldsymbol{x}_{\text{furniture}} \\ \boldsymbol{x}_{\text{king}} - \boldsymbol{x}_{\text{man}} &\approx \boldsymbol{x}_{\text{queen}} - \boldsymbol{x}_{\text{woman}} \end{aligned}$$

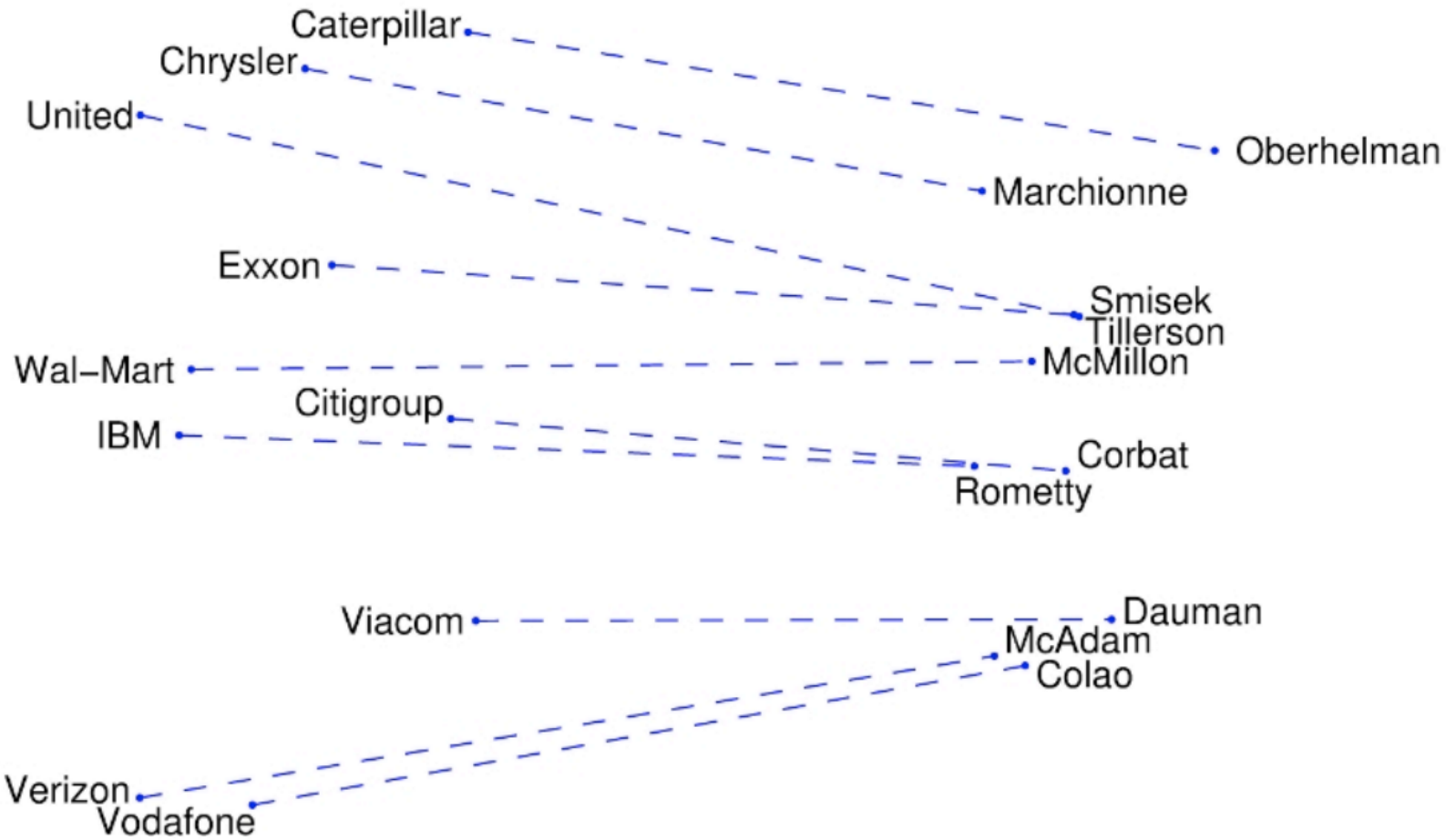
(Slide credit to Richard Socher)

Word Analogies (Mikolov et al., 2013)



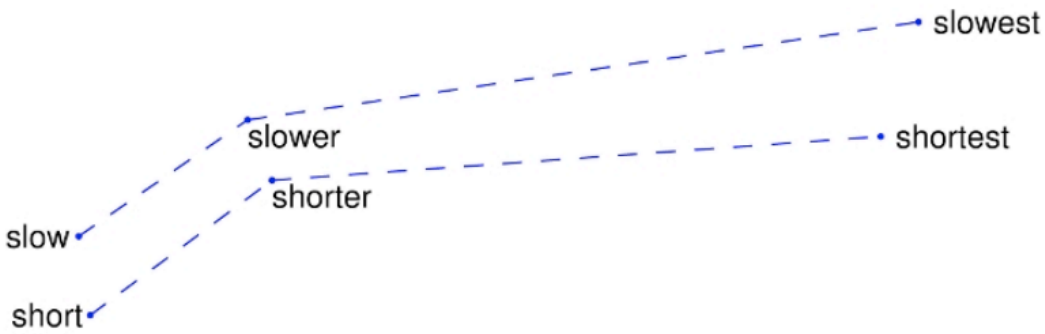
(Slide credit to Richard Socher)

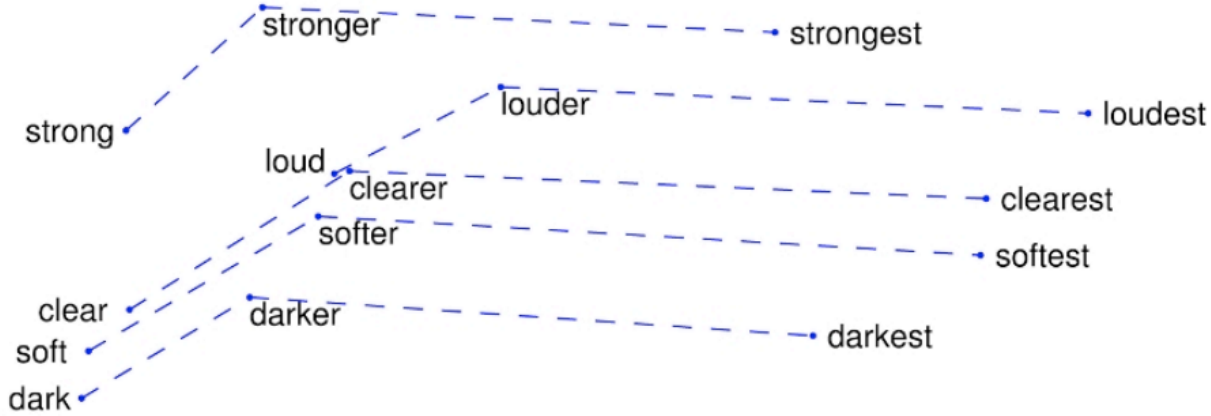
GloVe Visualizations: Company → CEO



(Slide credit to Richard Socher)

GloVe Visualizations: Superlatives





(Slide credit to Richard Socher)

Lunch Break

Outline

Convolutional Neural Networks

Convolutional Neural Networks are neural networks with specialized connectivity structure

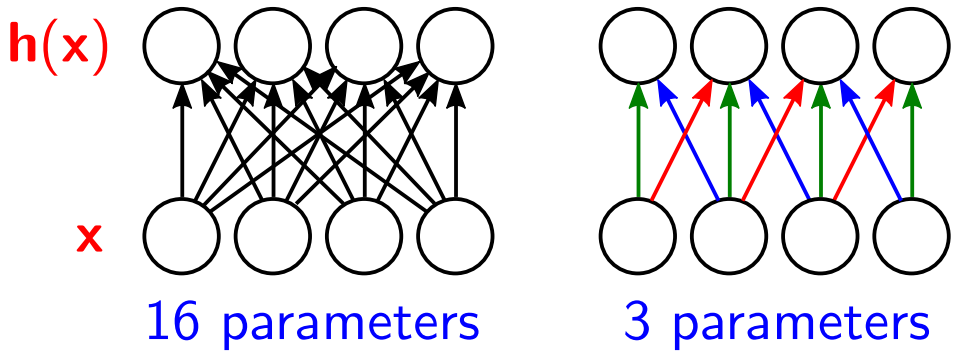
Roadmap:

- Parameter Tying
- 2D Convolutional Nets for Object Recognition
- Pooling
- ImageNet, AlexNet, GoogLeNet
- 1D Convolutional Nets in NLP
- Visualization and Google DeepDream

Convolutions and Parameter Tying

(Convolution: $h[t] = (x * w)[t] = \sum_{a=-\infty}^{\infty} x[a]w[t-a]$)

Basic idea: sparse connectivity + parameter tying



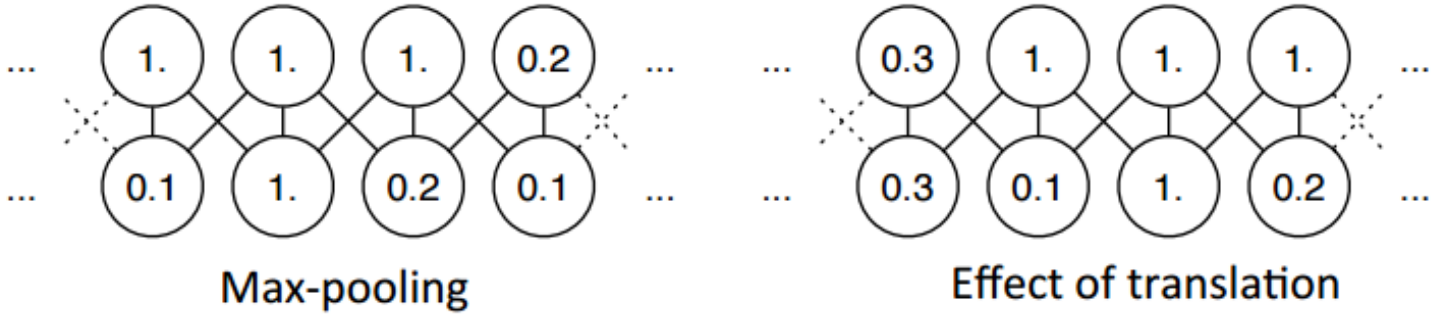
Leads to translation equivariance

Why do we want to tie (share) parameters?

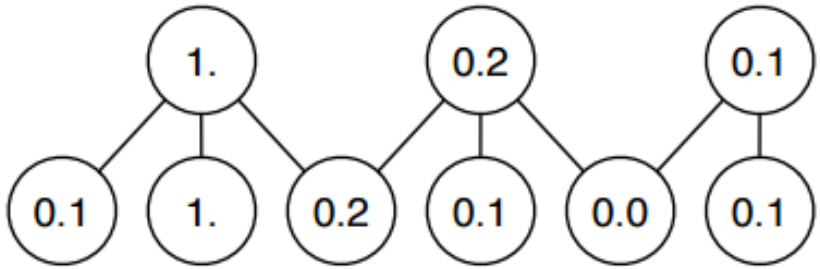
- Reduce the number of parameters to be learned
- Deal with arbitrary long, variable-length, sequences

Pooling Layers

- Aggregate to achieve local invariance:



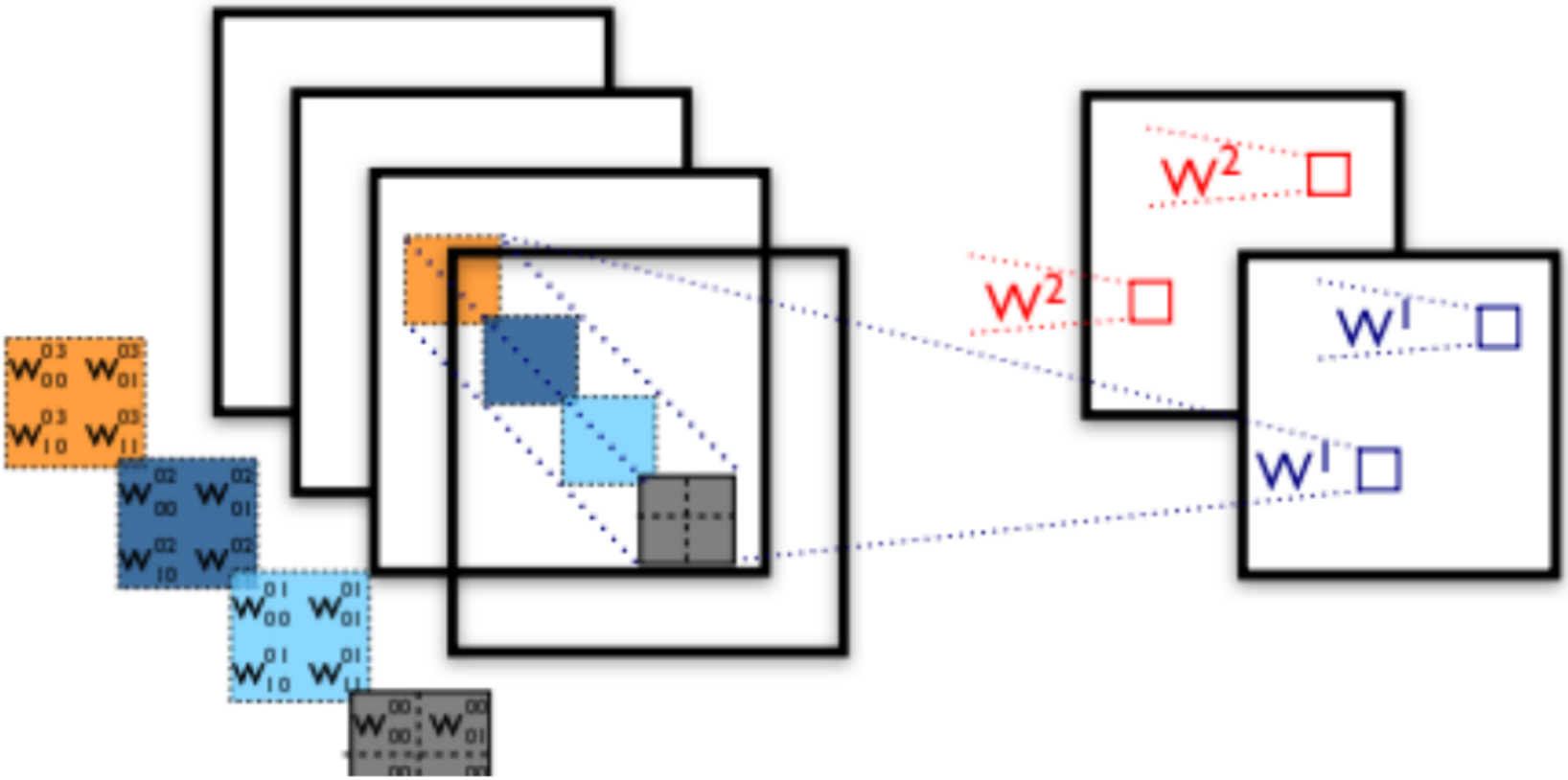
- Subsampling to reduce temporal/spacial scale and computation:



(Slide credit to Yoshua Bengio)

Multiple Convolutions: Feature Maps

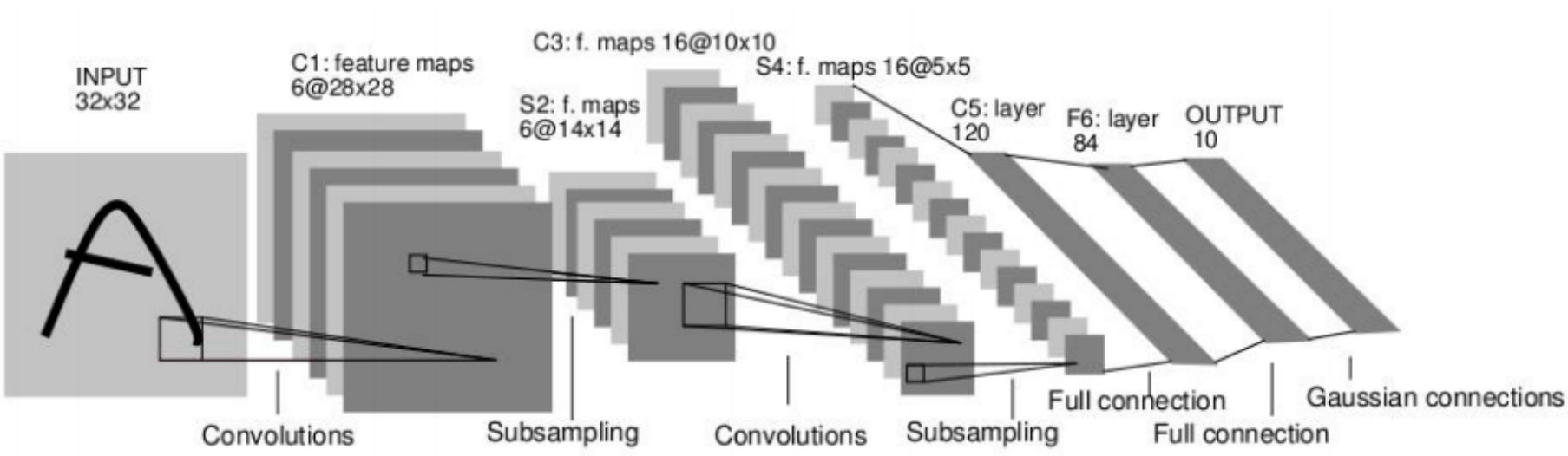
- Different filter weights for each channel, but keeping spatial invariance:



(Slide credit to Yoshua Bengio)

2D Convolutional Nets (LeCun et al., 1989)

- Inspired by "Neocognitron" (Fukushima, 1980)
- 2D Convolutions: the same filter (e.g. 3x3) is applied to each location of the image
- The filter weights are learned (as tied parameters)
- Multiple filters
- Alternates convolutional and pooling layers



ConvNet Successes

Handwritten text/digits:

- MNIST (0.35% error (Ciresan et al., 2011b))
- Arabic and Chinese (Ciresan et al., 2011a)

Simpler recognition benchmarks:

- CIFAR-10 (9.3% error (Wan et al., 2013))
- Traffic signs: 0.56% error vs 1.16% for humans (Cireşan et al., 2011)

But less good at more complex datasets, e.g. Caltech-101/256 (few training examples)

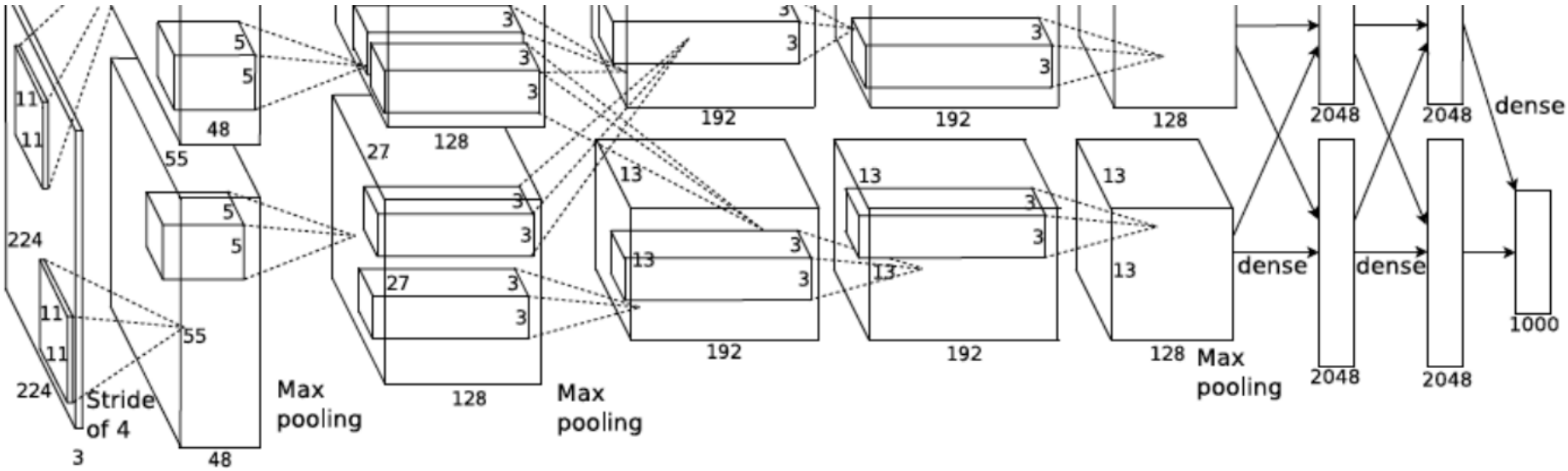
ImageNet Dataset

- 14 million labeled images, 20k classes
- Images gathered from Internet
- Human labels via Amazon Turk

(Slide credit to Rob Fergus)

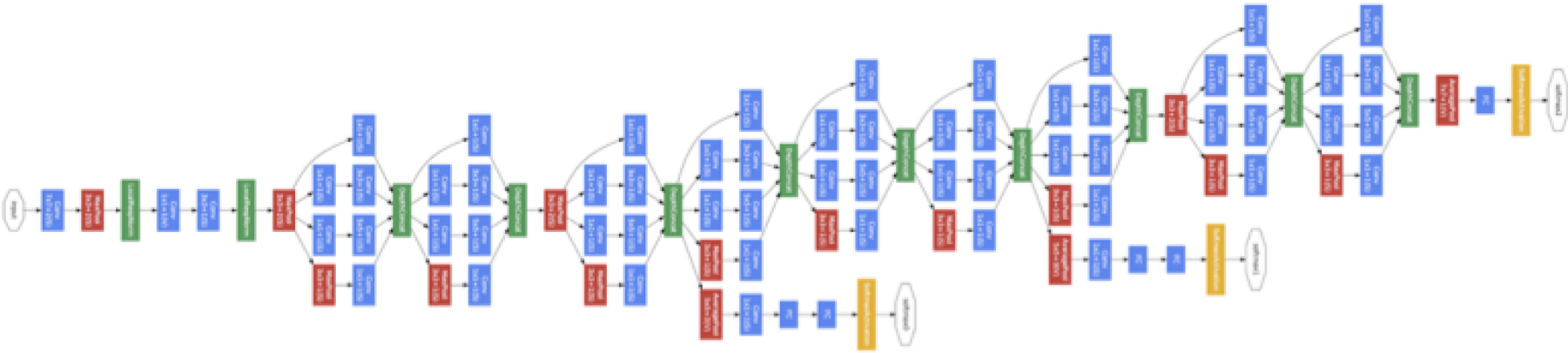
AlexNet (Krizhevsky et al., 2012)

- 54M parameters; 8 layers (5 conv, 3 fully-connected)
- Trained on 1.4M ImageNet images
- Trained on 2 GPUs for a week (50x speed-up over CPU)
- Dropout regularization
- Test error: 16.4% (second best team was 26.2%)



GoogLeNet (Szegedy et al., 2015)

- GoogLeNet inception module: very deep convolutional network, fewer (5M) parameters



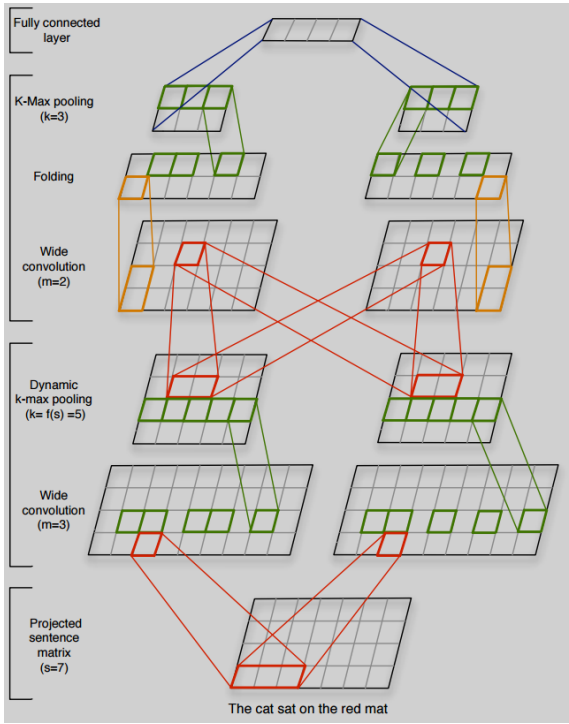
Convolution
Pooling
Softmax
Other

Convolutional Nets in NLP

- 1D convolutions
- Filters are applied to local windows around each word
- For word embeddings $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_L$, the filter response for word i is:

$$\boldsymbol{h}_i = \boldsymbol{g}(\mathbf{W}[\boldsymbol{x}_{i-h} \oplus \ldots \oplus \boldsymbol{x}_i \oplus \ldots \boldsymbol{x}_{i+h}] + \boldsymbol{b}),$$

  where $\oplus$ denotes vector concatenation and $\mathbf{W}$ are shared parameters
- Can pad left and right with special symbols if necessary



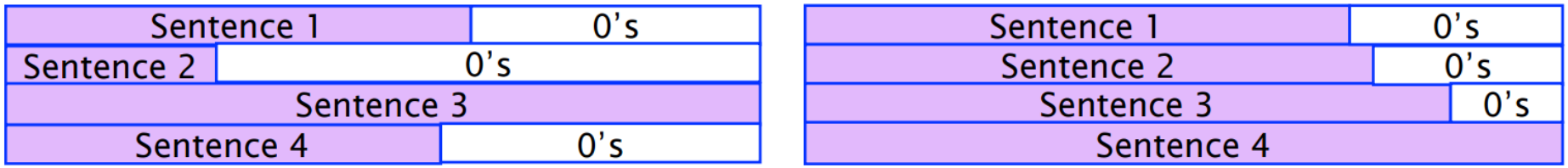
Kalchbrenner et al. (2014)

Mini-Batching, Padding, and Masking

Mini-batching is necessary to speed up training (e.g. in GPUs)

But how to cope with different input sizes (e.g. different sentence lengths)?

Solution: Minimize waste by sorting by sentence length before forming mini-batches, then padding:



(Image credit: Thang Luong, Kyunghyun Cho, Chris Manning)

What Representations Are We Learning?

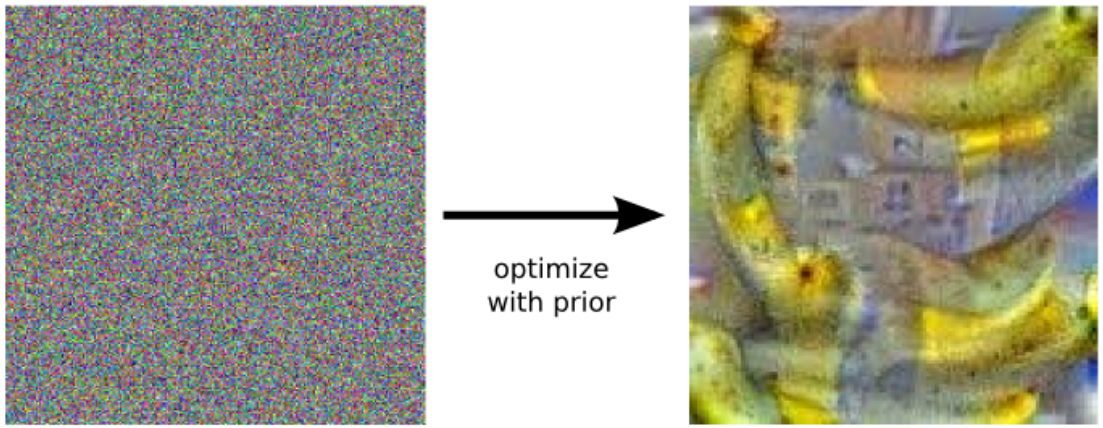
Which neurons fire for recognizing a particular object?

What parts of the network are activated?

To understand this, we need a way of visualizing what's happening inside the network.

Visualization

- **Idea:** Optimize input to maximize particular output
- Depends on the initialization
- Google DeepDream, maximizing "banana" output:



(from `https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html`)

- Can also specify a particular layer and tune the input to maximize the layer's activations—useful to see what kind of features each layer is representing
- Specifying a higher layer produces more complex representations...

Google DeepDream

(from https://research.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html)

Outline

Recurrent Neural Networks

Lots of interesting data is sequential in nature: words in sentences, DNA, stock market returns

How do we represent an arbitrarily long history?

Roadmap:

- Feedforward vs recurrent
- Backpropagation through time
- Vanishing/exploding gradient problem
- Long short-term memories and gated recurrent units
- Bidirectional RNNs

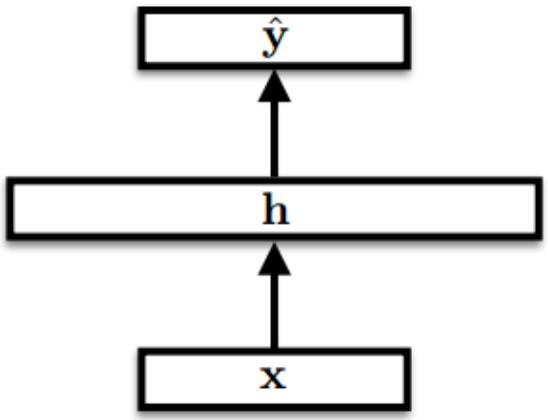
Feed-forward vs Recurrent Networks

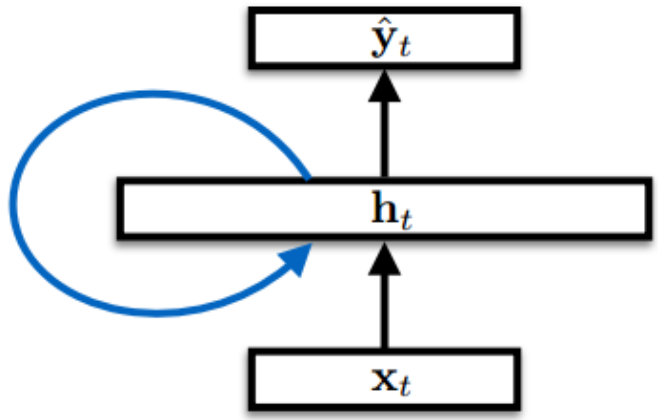
- Feed-forward neural networks:

$$\begin{aligned} \boldsymbol{h} &= \boldsymbol{g}(\mathbf{V}\boldsymbol{x} + \boldsymbol{c}) \\ \widehat{\boldsymbol{y}} &= \mathbf{W}\boldsymbol{h} + \boldsymbol{b} \end{aligned}$$

- Recurrent neural networks (Elman, 1990):

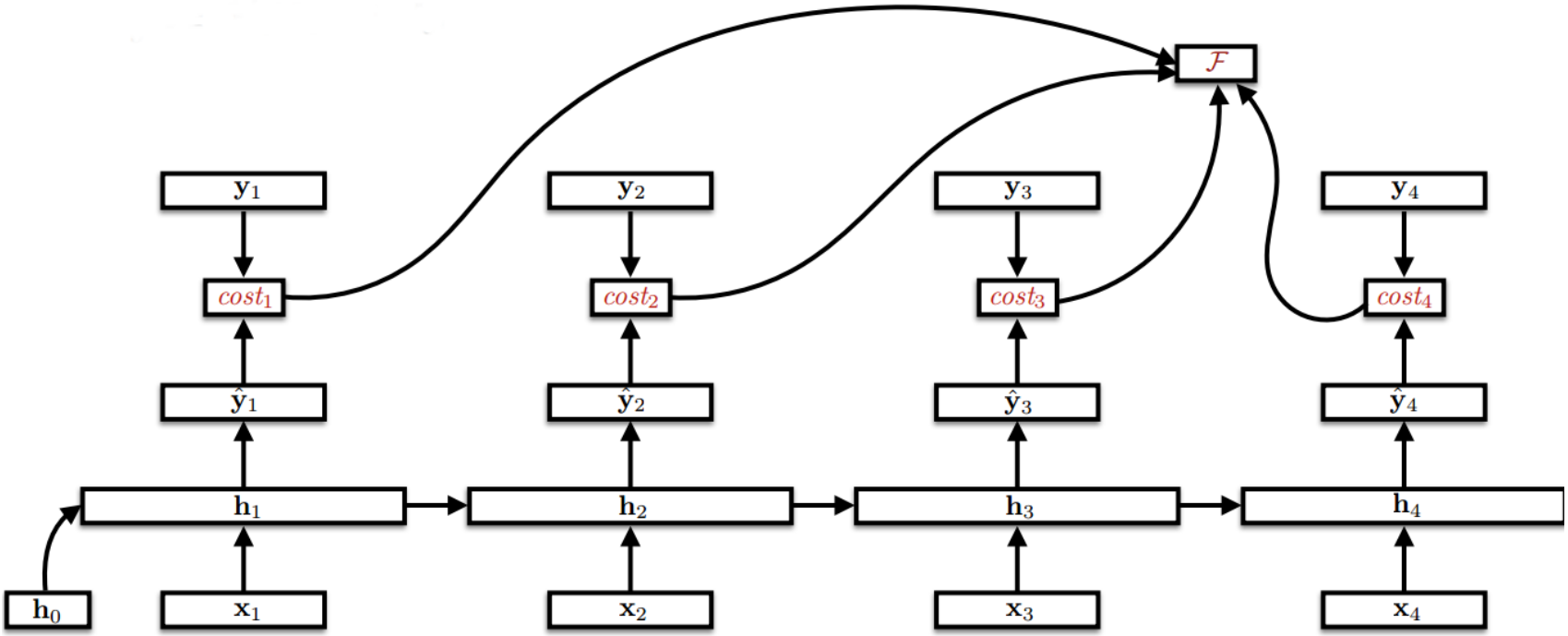
$$\begin{aligned} \boldsymbol{h}_t &= \boldsymbol{g}(\mathbf{V}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1} + \boldsymbol{c}) \\ \widehat{\boldsymbol{y}}_t &= \mathbf{W}\boldsymbol{h}_t + \boldsymbol{b} \end{aligned}$$





(Slide credit: Chris Dyer)

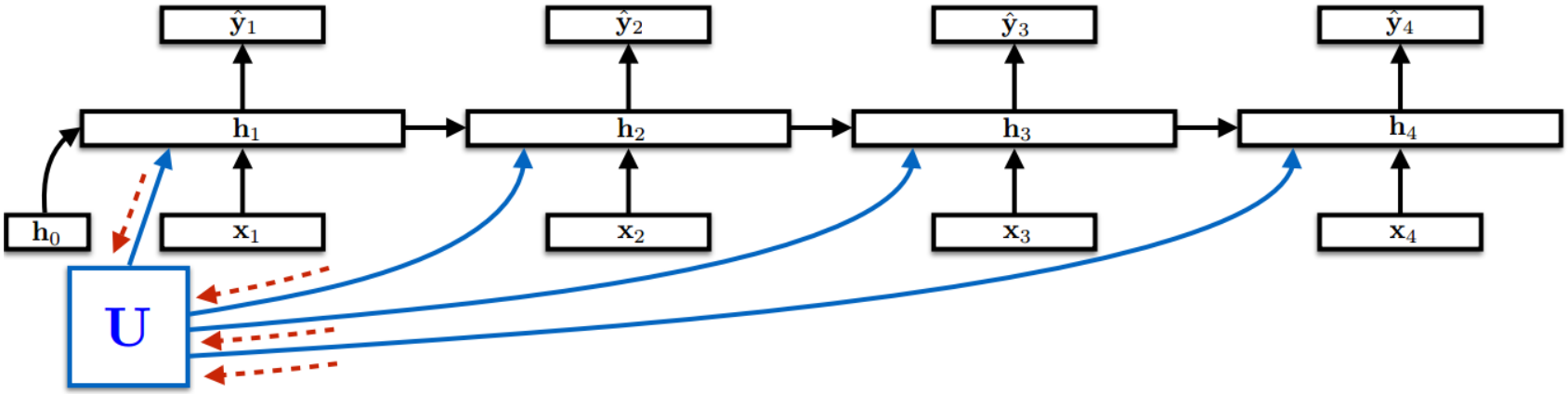
How do We Train the RNN Parameters?



(Slide credit: Chris Dyer)

- The unrolled graph is a well-formed (DAG) computation graph—we can run the gradient backpropagation algorithm as usual
- Parameters tied accross "time", derivatives aggregated across time steps
- This instantiation is called backpropagation through time (BPTT)

Parameter Tying



(Slide credit: Chris Dyer)

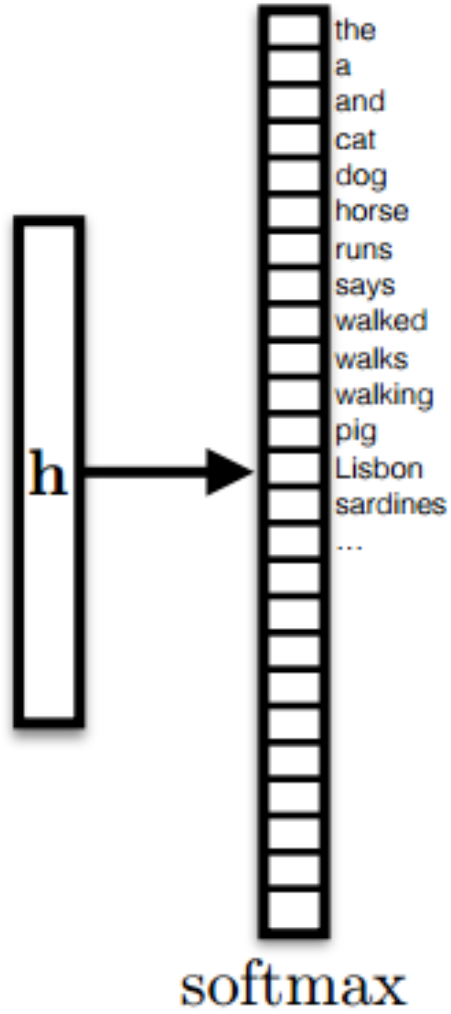
$$\frac{\partial \mathcal{F}}{\partial \mathbf{U}} = \sum_{t=1}^{4} \frac{\partial \boldsymbol{h}_t}{\partial \mathbf{U}} \frac{\partial \mathcal{F}}{\partial \boldsymbol{h}_t}$$

- Same idea as when learning the filters in convolutional neural networks

Example: Language Modeling

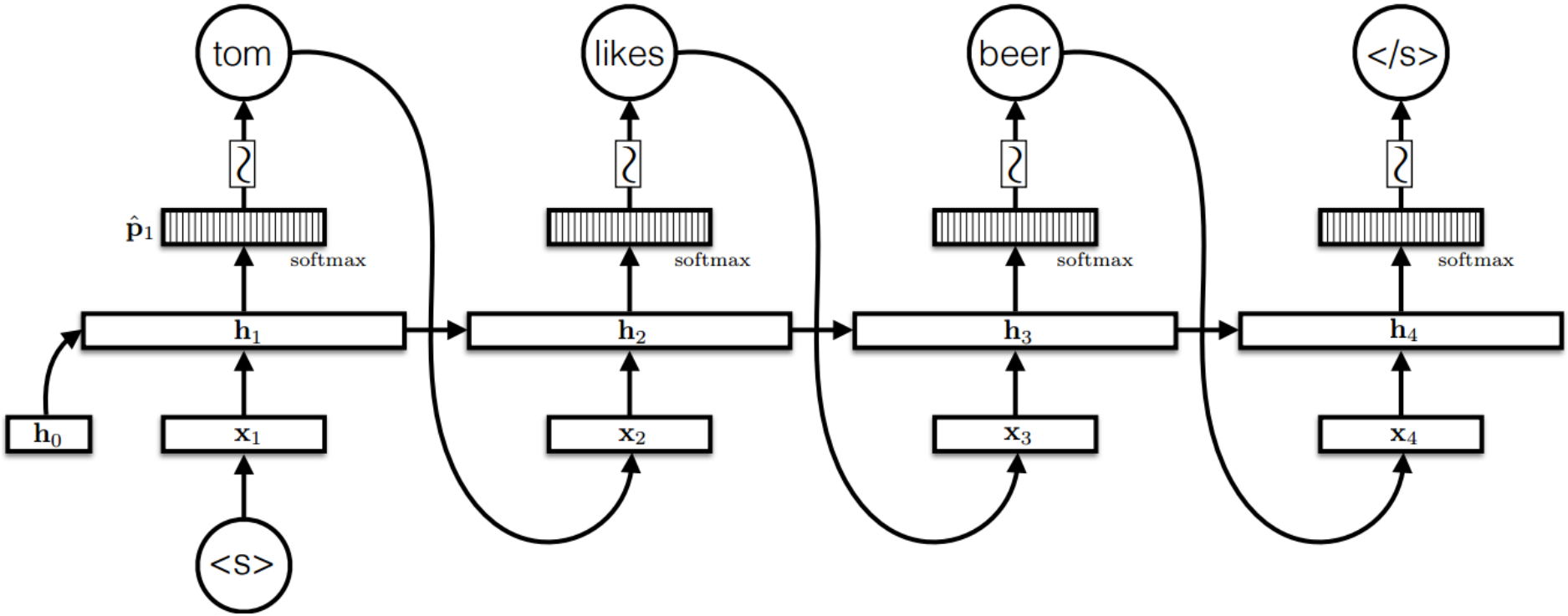
- Assume we want to generate text, and y_t is a word in the vocabulary
- Typically, large vocabulary size: $|V| = 100,000$

$$
\begin{aligned}
\boldsymbol{z} &= \mathbf{W}\boldsymbol{h} + \boldsymbol{b} \\
p(y_t = i) &= \frac{\exp(z_i)}{\sum_j \exp(z_j)} \\
&= \mathbf{softmax}_i(\boldsymbol{z})
\end{aligned}
$$



(Slide credit: Chris Dyer)

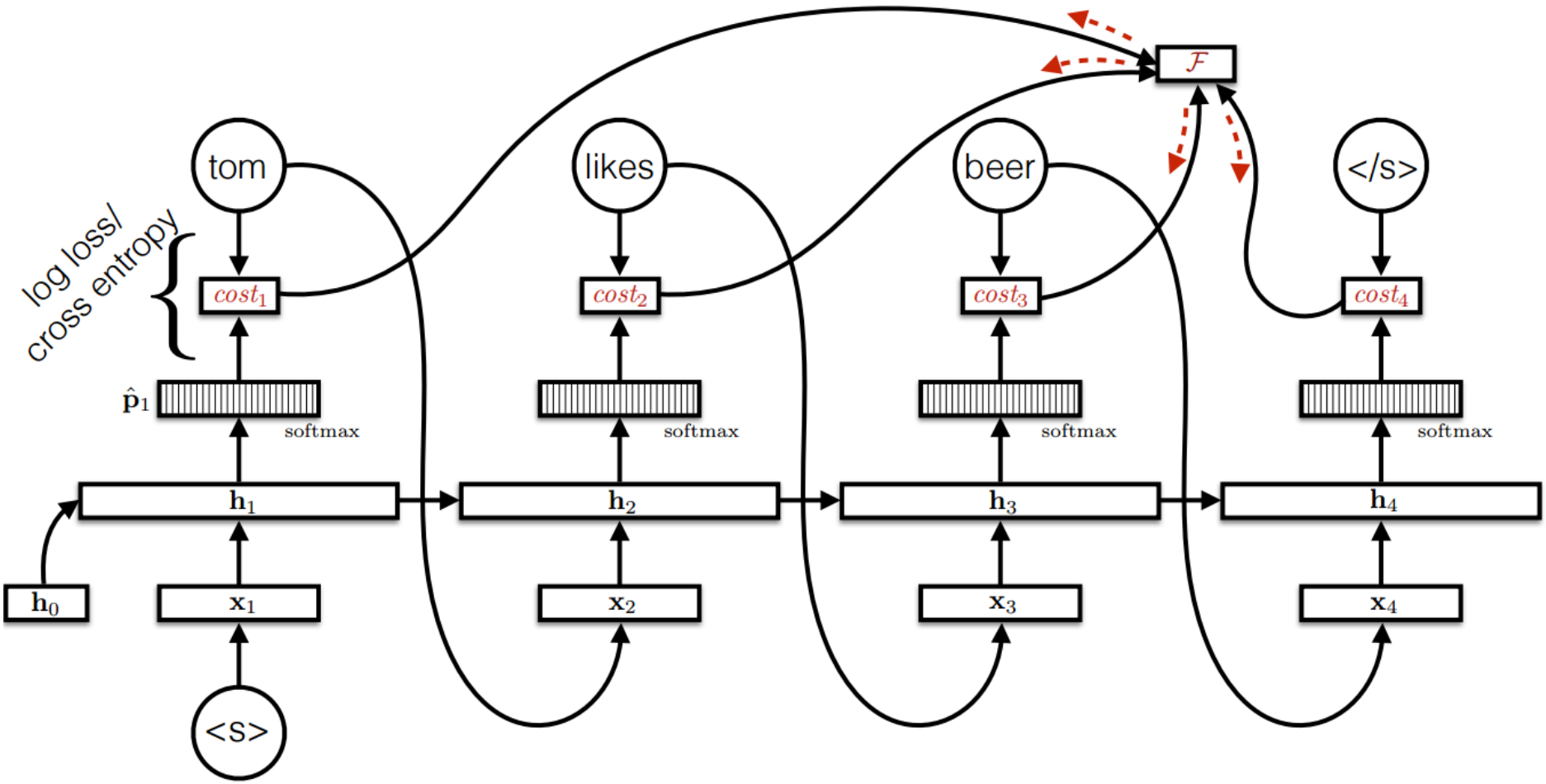
Example: Language Modeling



(Slide credit: Chris Dyer)

$$
\begin{aligned}
p(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_L) &= p(\boldsymbol{x}_1) \times p(\boldsymbol{x}_2 \mid \boldsymbol{x}_1) \times \ldots \times p(\boldsymbol{x}_L \mid \boldsymbol{x}_1, \ldots, \boldsymbol{x}_{L-1}) \\
&= \mathbf{softmax}(\mathbf{W}\boldsymbol{h}_1 + \boldsymbol{b}) \times \mathbf{softmax}(\mathbf{W}\boldsymbol{h}_2 + \boldsymbol{b}) \times \ldots \\
&\quad \times \mathbf{softmax}(\mathbf{W}\boldsymbol{h}_L + \boldsymbol{b})
\end{aligned}
$$

Language Modeling Training



(Slide credit: Chris Dyer)

Language Modeling Training

Unlike Markov (*n*-gram) models, RNNs never forget!

- However we will see they might have trouble learning to use their memories (more soon...)

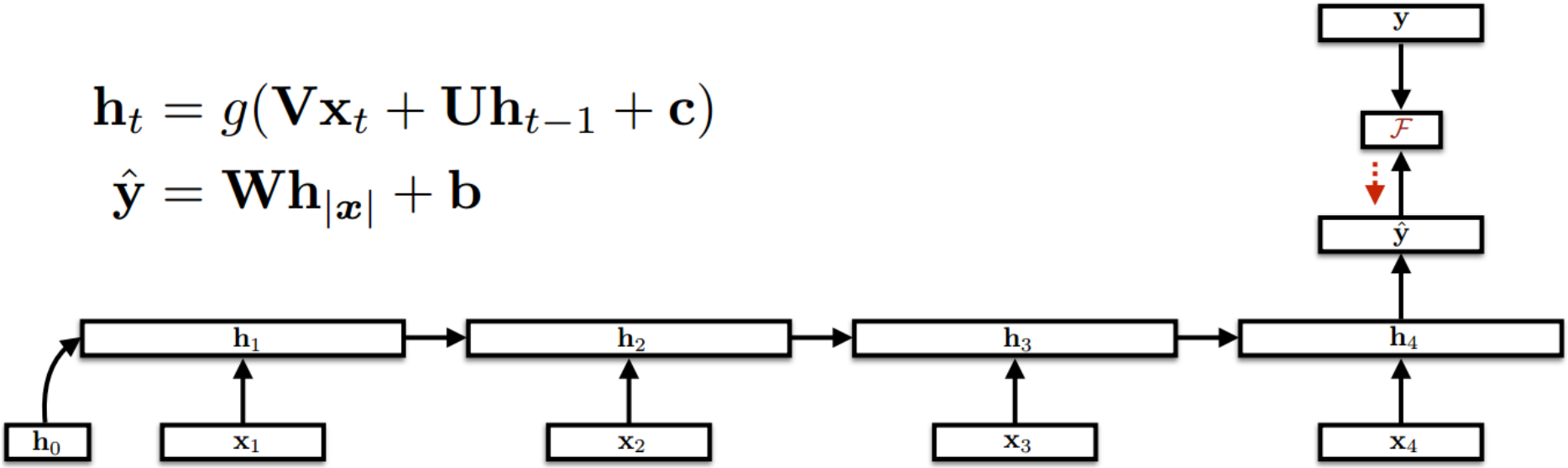
Algorithms:

- Sample a sequence from the probability distribution defined by the RNN
- Train the RNN to minimize cross entropy (aka MLE)
- **What about:** what is the most probable sequence?

Backpropagation Through Time

What happens to the gradients as we go back in time?

$$\mathbf{h}_t = g(\mathbf{V}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{c})$$
$$\hat{\mathbf{y}} = \mathbf{W}\mathbf{h}_{|\boldsymbol{x}|} + \mathbf{b}$$



(Slide credit: Chris Dyer)

Backpropagation Through Time

What happens to the gradients as we go back in time?

$$\frac{\partial \mathcal{F}}{\partial \boldsymbol{h}_1} = \underbrace{\frac{\partial \boldsymbol{h}_2}{\partial \boldsymbol{h}_1}\frac{\partial \boldsymbol{h}_3}{\partial \boldsymbol{h}_2}\frac{\partial \boldsymbol{h}_4}{\partial \boldsymbol{h}_3}}_{\prod_{t=2}^{4}\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}}}\frac{\partial \widehat{\boldsymbol{y}}}{\partial \boldsymbol{h}_4}\frac{\partial \mathcal{F}}{\partial \widehat{\boldsymbol{y}}}$$

where

$$\prod_t \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} = \prod_t \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{z}_t}\frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{h}_{t-1}} = \prod_t \text{Diag}(\boldsymbol{g}'(\boldsymbol{z}_t))\mathbf{U}$$

Three cases:

- largest eigenvalue of $\mathbf{U}$ exactly 1: gradient propagation is stable
- largest eigenvalue of $\mathbf{U} < 1$: gradient vanishes (exponential decay)
- largest eigenvalue of $\mathbf{U} > 1$: gradient explodes (exponential growth)

Vanishing and Exploding Gradients

Exploding gradients can be dealt with by gradient clipping (truncating the gradient if it exceeds some magnitude)

Vanishing gradients are more frequent and harder to deal with

- In practice: long-range dependencies are difficult to learn

Solutions:

- Better optimizers (second order methods)
- Normalization to keep the gradient norms stable across time
- Clever initialization so that you at least start with good spectra (e.g., start with random orthonormal matrices)
- Alternative parameterizations: LSTMs and GRUs

Gradient Clipping

- **Norm clipping:**

$$\tilde{\nabla} \leftrightarrow \begin{cases} \frac{c}{\|\nabla\|}\nabla & \text{if } \|\nabla\| \geq c \\ \nabla & \text{otherwise.} \end{cases}$$

- **Elementwise clipping:**

$$\tilde{\nabla}_i \leftrightarrow \min\{c, |\nabla_i|\} \times \textbf{sign}(\nabla_i),\ \forall i$$

Alternative RNNs

I'll next describe:

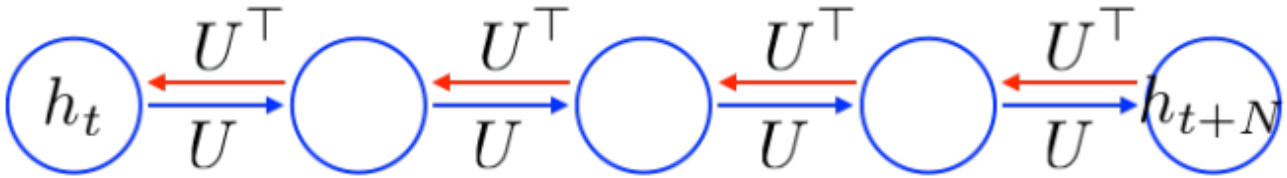
- Gated recurrent units (GRUs; Cho et al. (2014))
- Long short-term memories (LSTMs; Hochreiter and Schmidhuber (1997))

Intuition: instead of multiplying across time (which leads to exponential growth), we want the error to be approximately constant

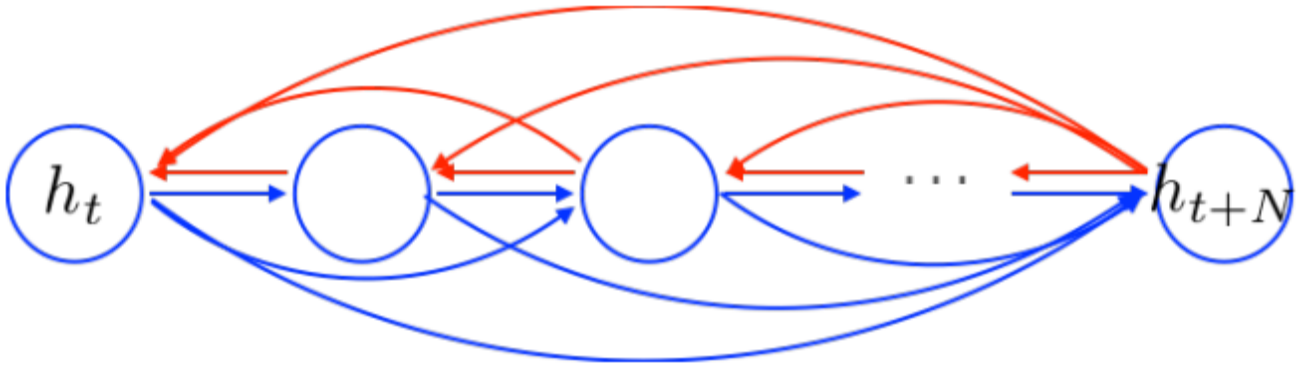
They solve the vanishing gradient problem, but still have exploding gradients (gradient clipping still helps)

Gated Recurrent Units (Cho et al., 2014)

- Recall the problem: the error must backpropagate through all the intermediate nodes:



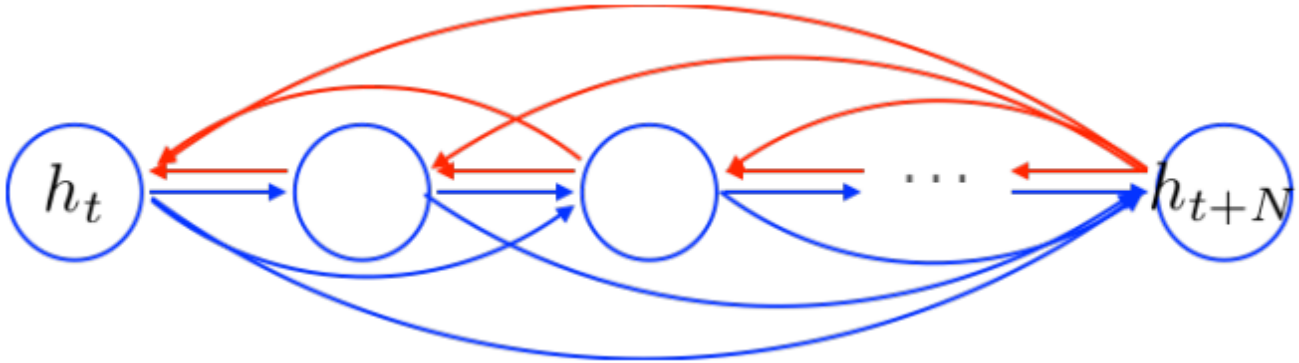
- **Idea:** Maybe we can create some kind of shortcut connections:



(Image credit: Thang Luong, Kyunghyun Cho, Chris Manning)

- Create adaptive shortcuts controlled by special gates

Gated Recurrent Units (Cho et al., 2014)



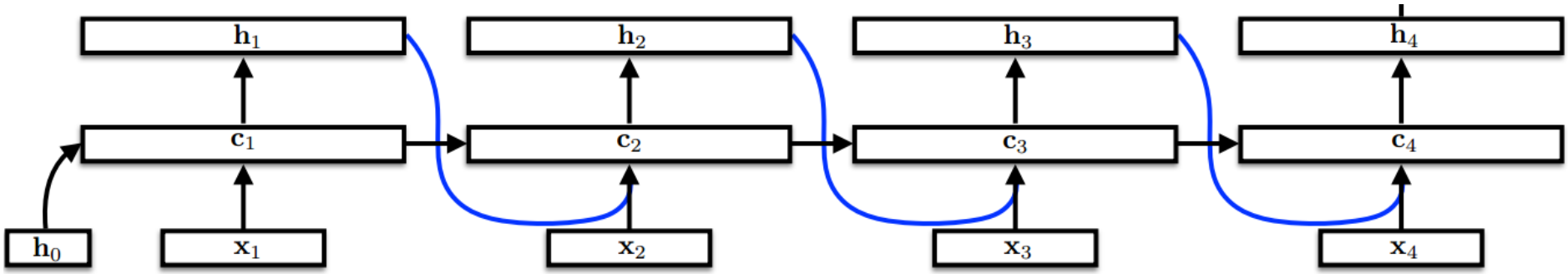
(Image credit: Thang Luong, Kyunghyun Cho, Chris Manning)

$$\boxed{\boldsymbol{h}_t = \boldsymbol{u}_t \odot \tilde{\boldsymbol{h}}_t + (1 - \boldsymbol{u}_t) \odot \boldsymbol{h}_{t-1}}$$

- **Candidate update:** $\tilde{\boldsymbol{h}}_t = \boldsymbol{g}(\mathbf{V}\boldsymbol{x}_t + \mathbf{U}(\boldsymbol{r}_t \odot \boldsymbol{h}_{t-1}) + \boldsymbol{b})$
- **Reset gate:** $\boldsymbol{r}_t = \sigma(\mathbf{V}_r\boldsymbol{x}_t + \mathbf{U}_r\boldsymbol{h}_{t-1} + \boldsymbol{b}_r)$
- **Update gate:** $\boldsymbol{u}_t = \sigma(\mathbf{V}_u\boldsymbol{x}_t + \mathbf{U}_u\boldsymbol{h}_{t-1} + \boldsymbol{b}_u)$

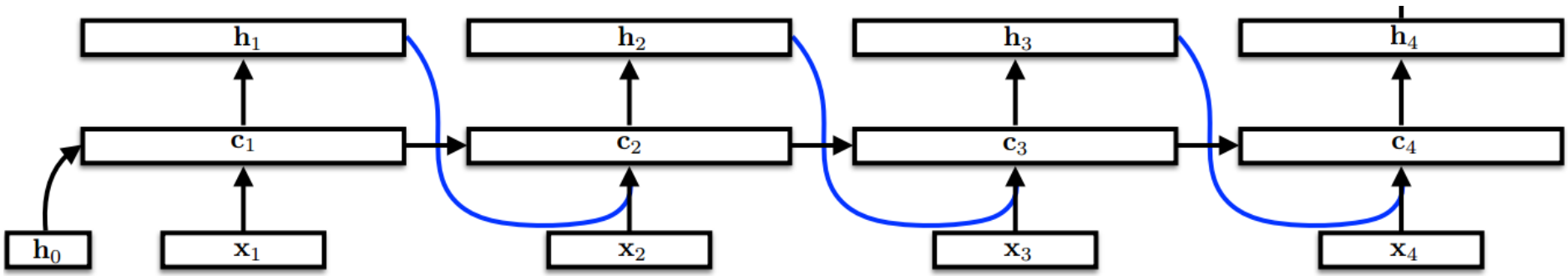
Long Short-Term Memories (Hochreiter and Schmidhuber, 1997)

- **Key idea:** use memory cells c_t
- To avoid the multiplicative effect, flow information *additively* through these cells
- Control the flow with special input, forget, and output gates



(Image credit: Chris Dyer)

Long Short-Term Memories

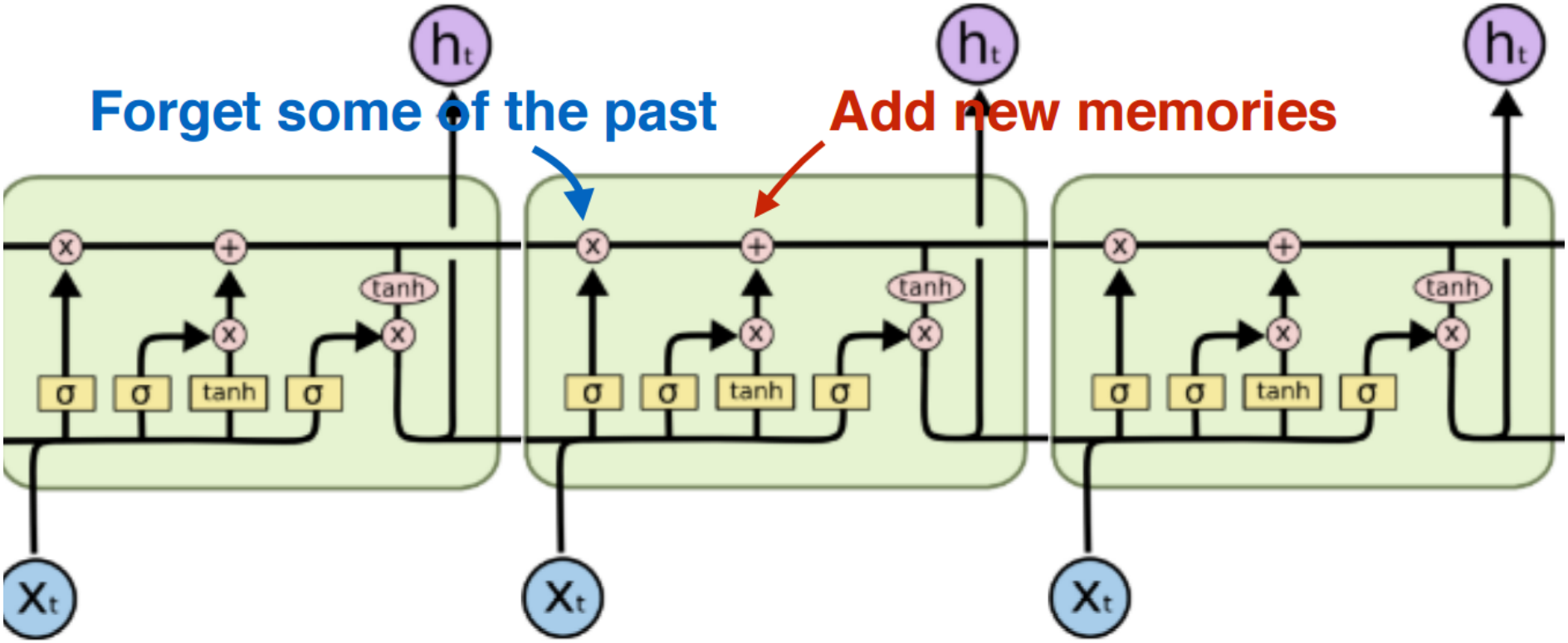


(Image credit: Chris Dyer)

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot (\mathbf{V}\boldsymbol{x}_t + \mathbf{U}\boldsymbol{h}_{t-1} + \boldsymbol{b}), \qquad \boldsymbol{h}_t = \boldsymbol{o}_t \odot \boldsymbol{g}(\boldsymbol{c}_t)$$

- **Forget gate:** $\boldsymbol{f}_t = \sigma(\mathbf{V}_f \boldsymbol{x}_t + \mathbf{U}_f \boldsymbol{h}_{t-1} + \boldsymbol{b}_f)$
- **Input gate:** $\boldsymbol{i}_t = \sigma(\mathbf{V}_i \boldsymbol{x}_t + \mathbf{U}_i \boldsymbol{h}_{t-1} + \boldsymbol{b}_i)$
- **Output gate:** $\boldsymbol{o}_t = \sigma(\mathbf{V}_o \boldsymbol{x}_t + \mathbf{U}_o \boldsymbol{h}_{t-1} + \boldsymbol{b}_o)$

Long Short-Term Memories



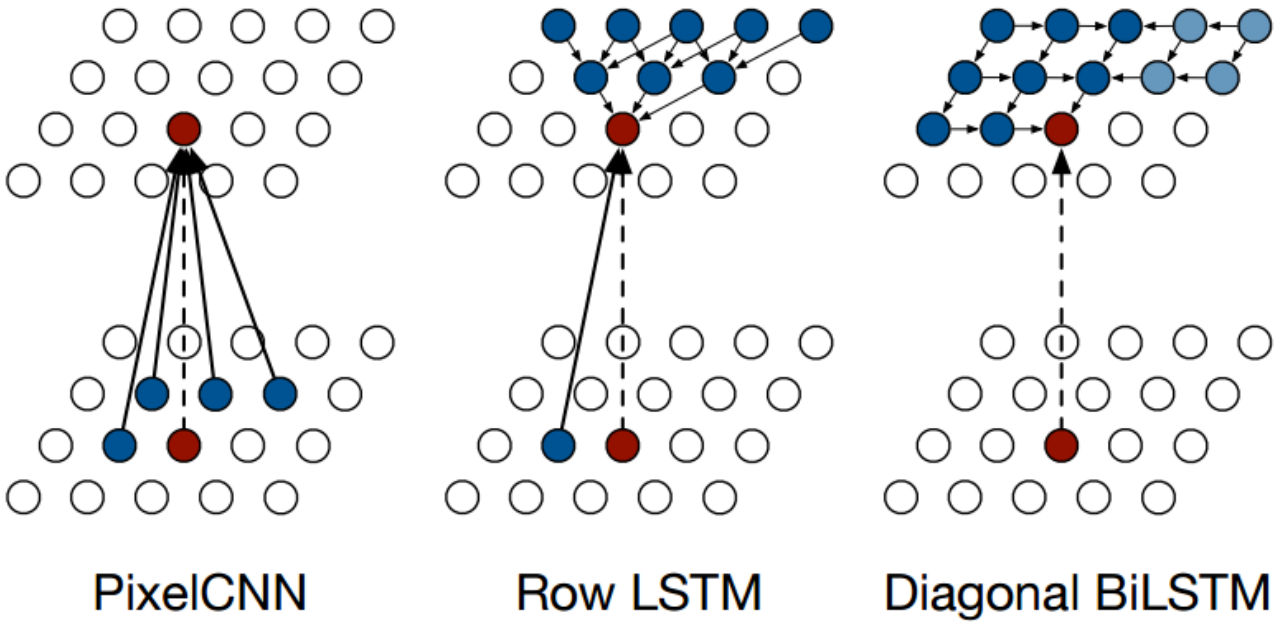
(Slide credit: Christopher Olah)

Summary

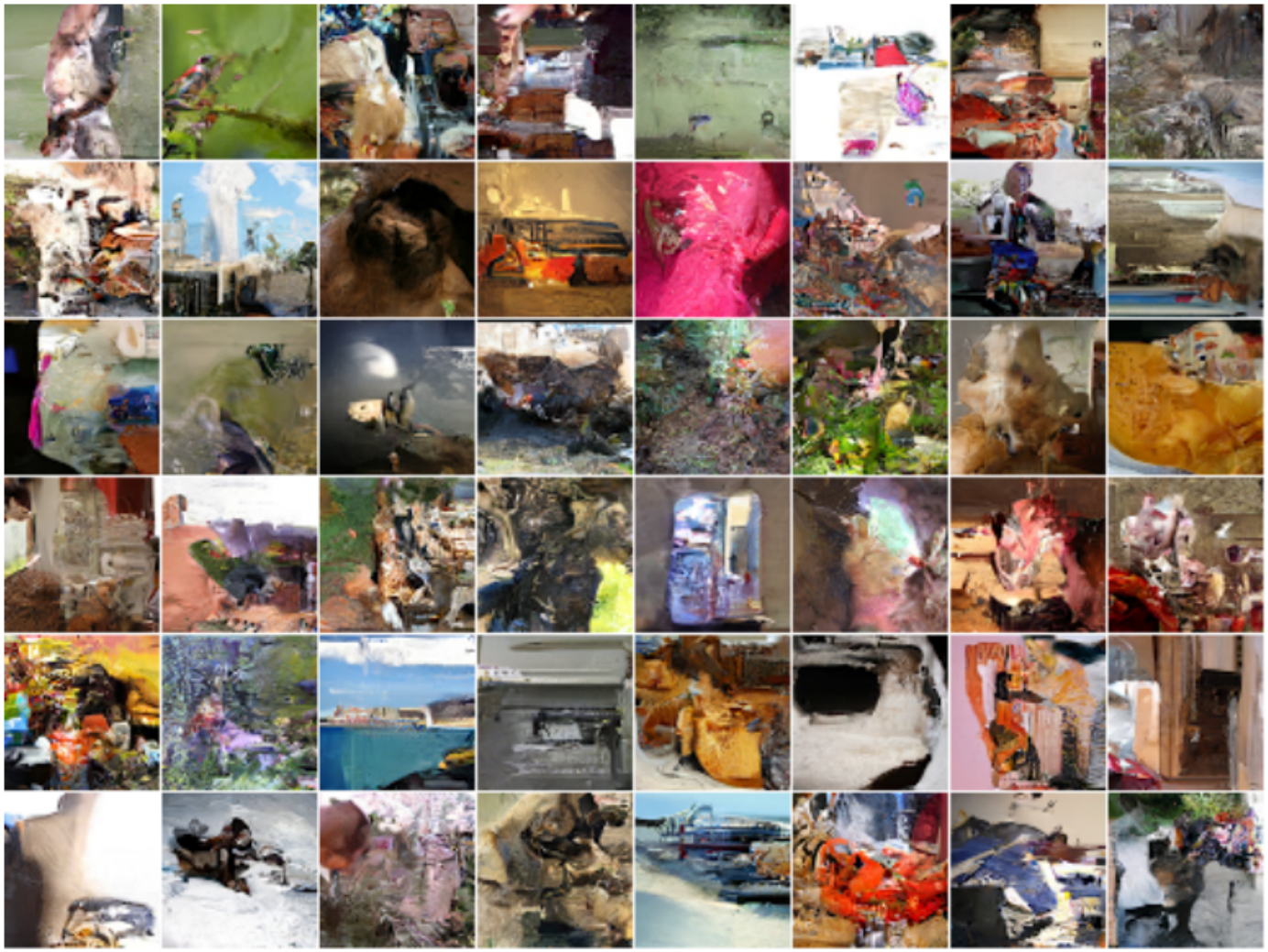
- Better gradient propagation is possible if we use additive rather than multiplicative/highly non-linear recurrent dynamics
- Recurrent architectures are an active area of research (but LSTMs are hard to beat)
- Other variants of LSTMs exist which tie/simplify some of the gates
- Extensions exist for *non-sequential* structured inputs/outputs (e.g. trees): recursive neural networks (Socher et al., 2011), PixelRNN (Oord et al., 2016)

RNNs for Generating Images

- Input-to-state and state-to-state mappings for PixelCNN an two PixelRNN models (Oord et al., 2016):



RNNs for Generating Images



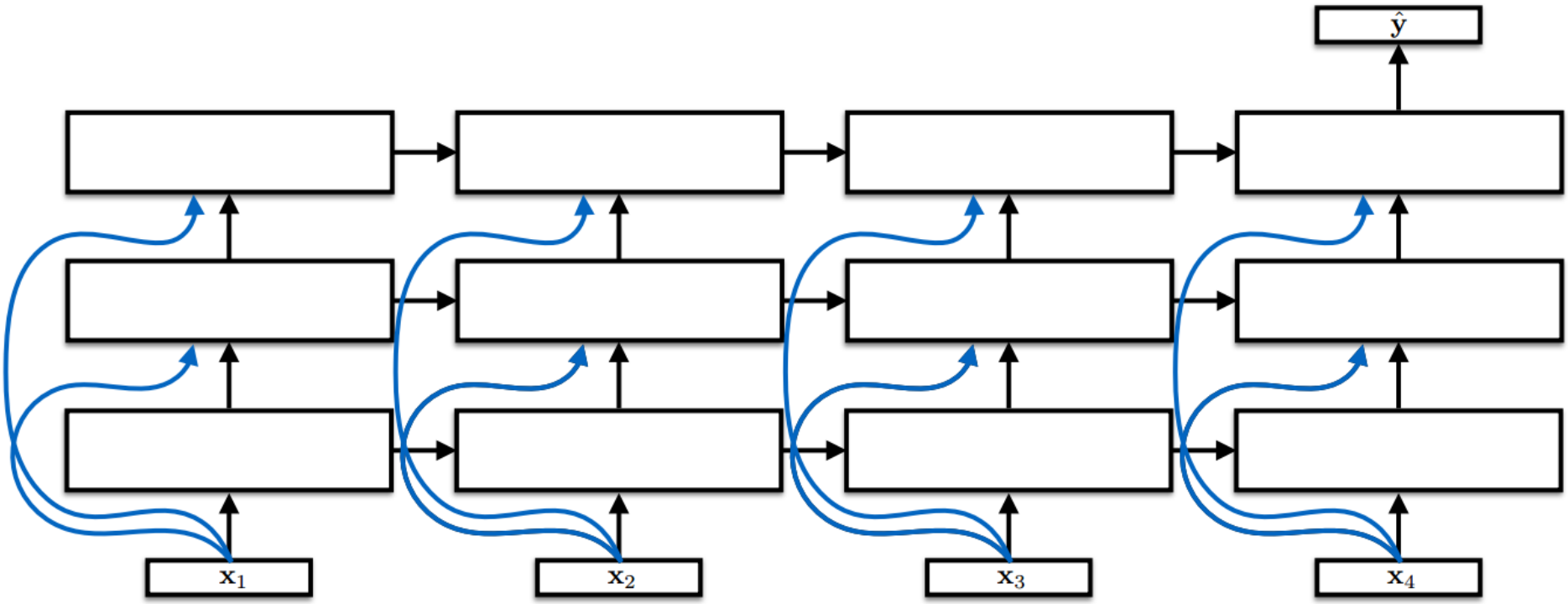
(Oord et al., 2016)

More Tricks of the Trade

- Depth
- Dropout
- Implementation Tricks

Deep RNNs/LSTMs/GRUs

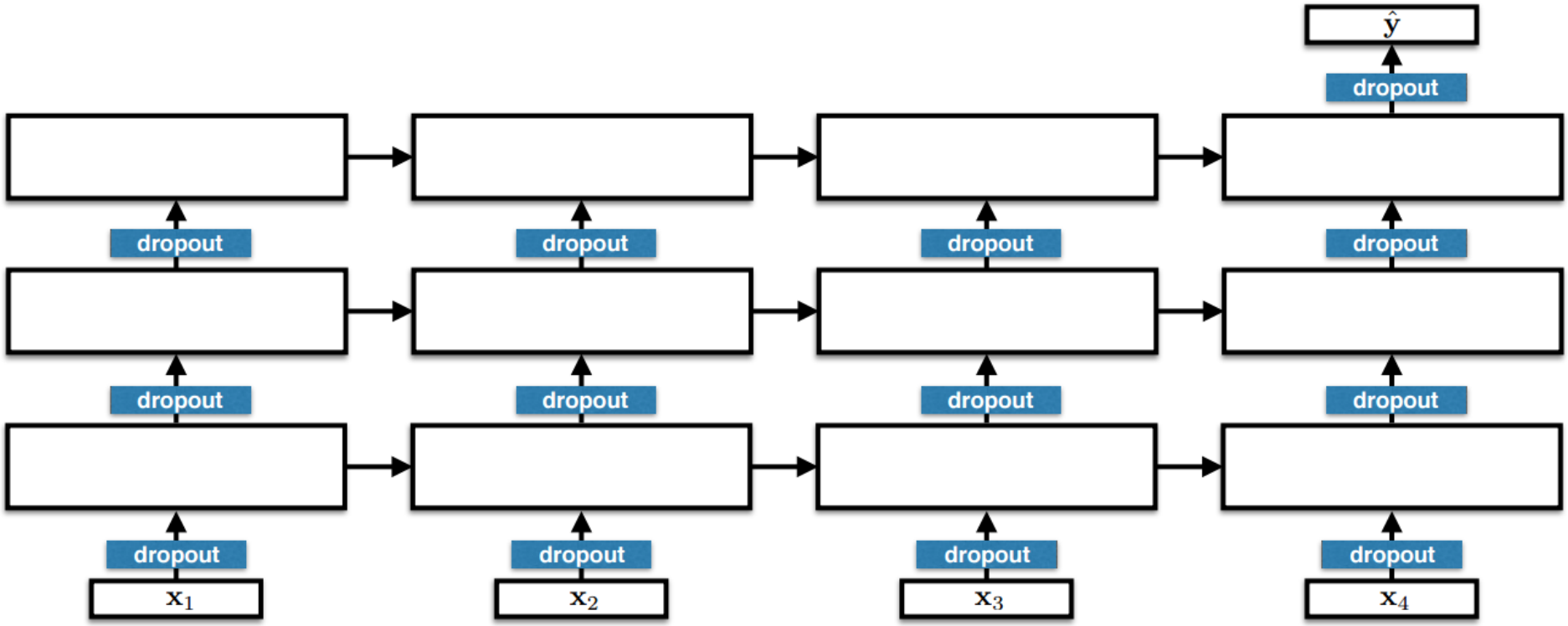
- Depth in recurrent layers helps in practice (2–8 layers seem to be standard)
- Input connections may or may not be used



(Slide credit: Chris Dyer)

Dropout in Deep RNNs/LSTMs/GRUs

- Apply dropout between layers, but not on the recurrent connections
- ... Or use the same mask for all recurrent connections (Gal and Ghahramani, 2015)



(Slide credit: Chris Dyer)

Implementation Tricks

For speed:

- Use diagonal matrices instead of full matrices (esp. for gates)
- Concatenate parameter matrices for all gates and do a single matrix-vector multiplication
- Use optimized implementations (from NVIDIA)
- Use GRUs or reduced-gate variant of LSTMs

For learning speed and performance:

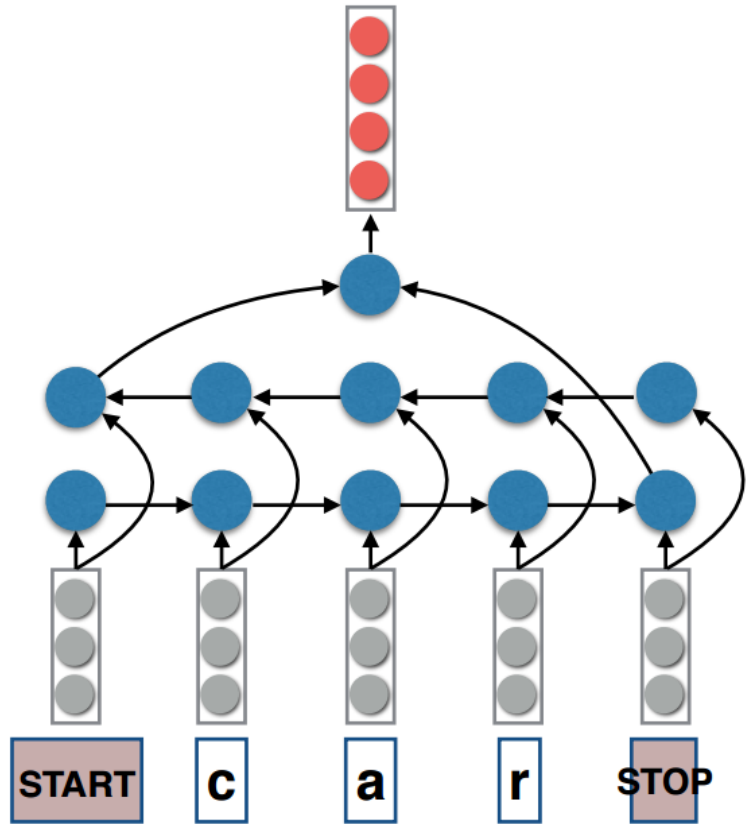
- Initialize so that the bias on the forget gate is large (intuitively: at the beginning of training, the signal from the past is unreliable)
- Use random orthogonal matrices to initialize the square matrices

Mini-Batching

- RNNs, LSTMs, GRUs all consist of lots of elementwise operations (addition, multiplication, nonlinearities), and lots of matrix-vector products
- Mini-batching: convert many matrix-vector products into a single matrix-matrix multiplication
- Batch across instances, not across time
- The challenge with working with mini batches of sequences is... sequences are of different lengths (we've seen this when talking about convolutional nets)
- This usually means you bucket training instances based on similar lengths, and pad with zeros
- Be careful when padding not to back propagate a non-zero value!

Bidirectional RNNs

- We can read a sequence from left to right to obtain a representation
- Or we can read it from right to left
- Or we can read it from both and combine the representations
- More later...



(Slide credit: Chris Dyer)

Coffee Break

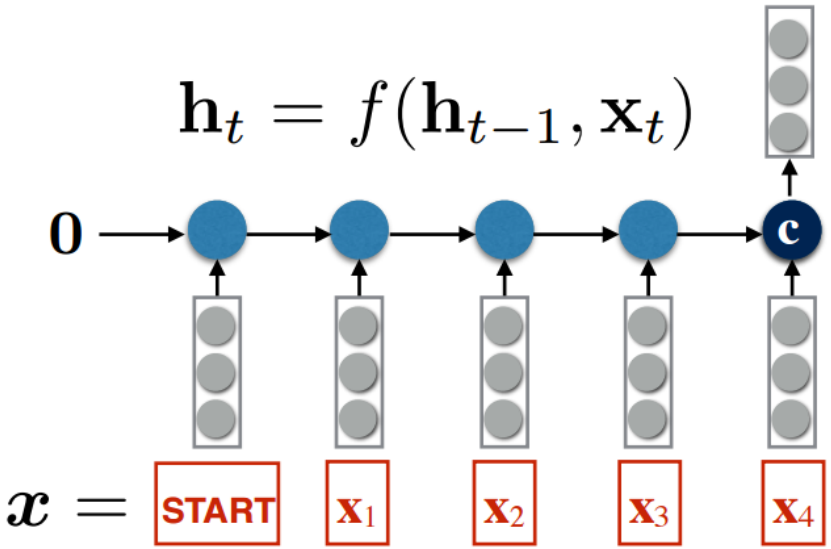
Outline

Sequence-to-Sequence Learning (Cho et al., 2014; Sutskever et al., 2014)

Roadmap:

- Sequence vector representation
- Encoder-decoder architecture
- Sequence matrix representation
- Attention mechanisms
- Encoder-decoder with attention
- Applications: machine translation, caption generation

Encode a Sequence as a Vector



(Slide credit: Chris Dyer)

What is a vector representation of a sequence $\boldsymbol{x}$?

$$\boldsymbol{c} = \mathbf{RNN}(\boldsymbol{x})$$

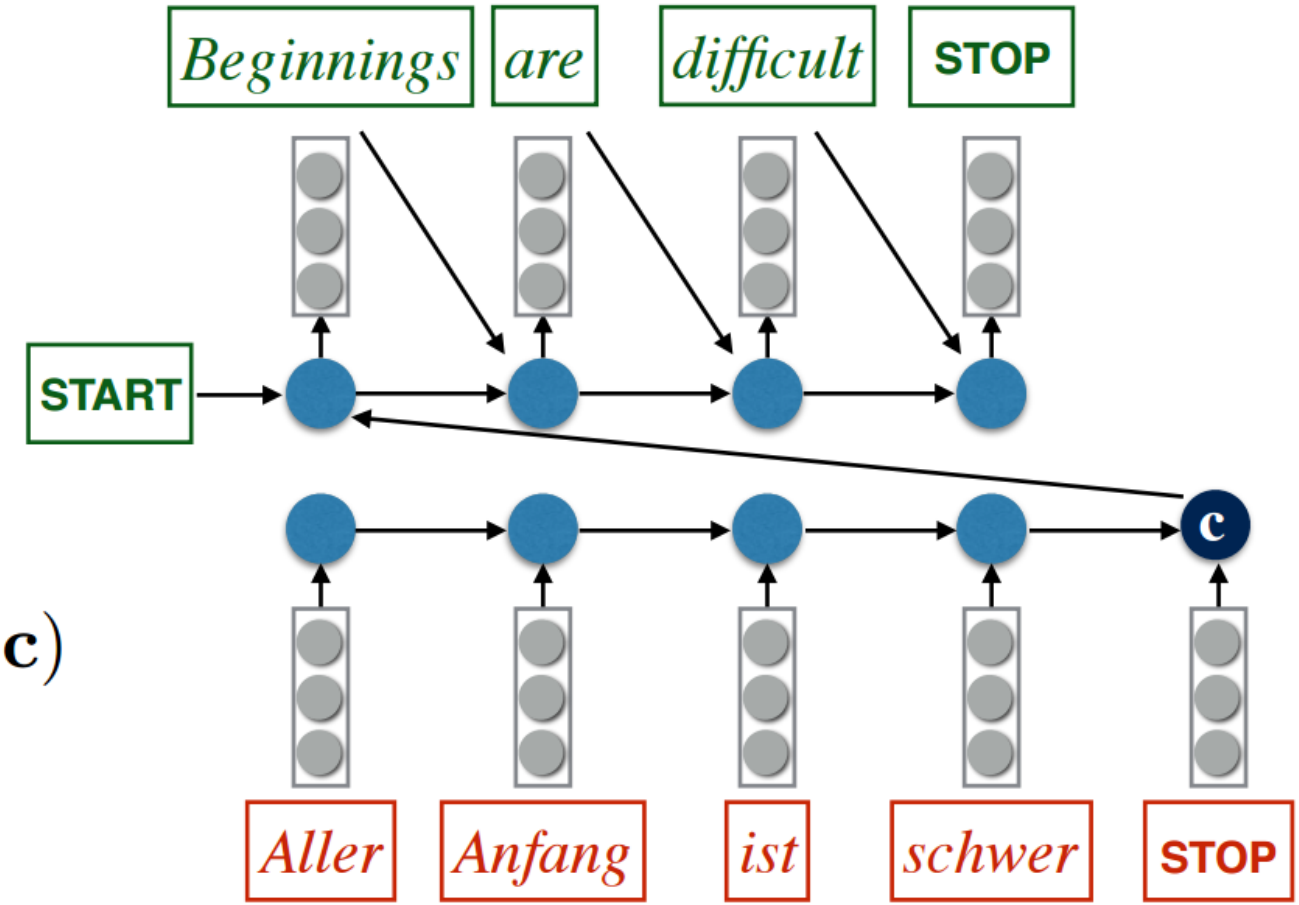
What is the probability of a sequence $\boldsymbol{y} \mid \boldsymbol{x}$?

$$\boldsymbol{y} \mid \boldsymbol{x} \sim \mathbf{RNNLM}(\boldsymbol{c})$$

Encoder-Decoder Architecture

$$\mathbf{c} = \text{RNN}(\boldsymbol{x})$$

$$\boldsymbol{y} \mid \mathbf{c} \sim \text{RNNLM}(\mathbf{c})$$



(Slide credit: Chris Dyer)

Some Additional Tricks

From Sutskever et al. (2014):

- Deep LSTMs
- Reversing the source sentence

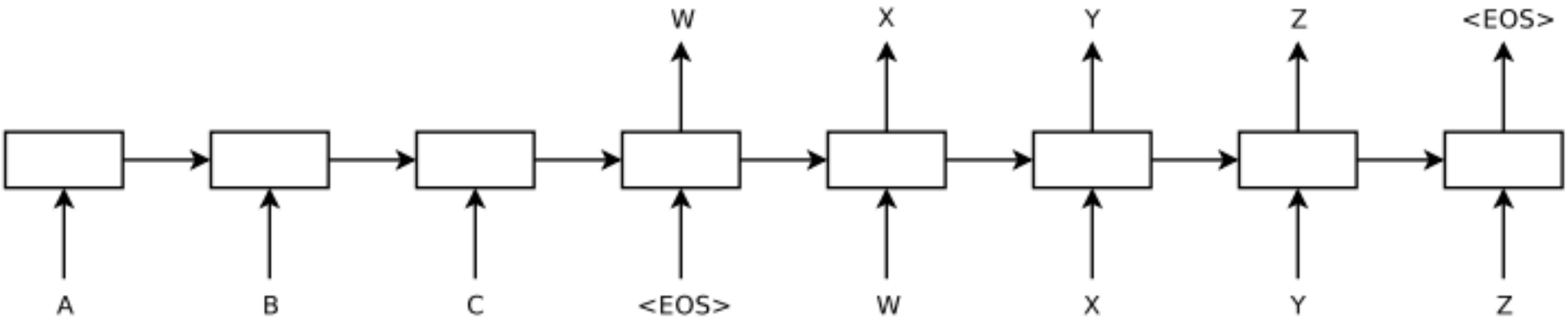
Method	test BLEU score (ntst14)
Bahdanau et al. [2]	28.45
Baseline System [29]	33.30
Single forward LSTM, beam size 12	26.17
Single reversed LSTM, beam size 12	30.59
Ensemble of 5 reversed LSTMs, beam size 1	33.00
Ensemble of 2 reversed LSTMs, beam size 12	33.27
Ensemble of 5 reversed LSTMs, beam size 2	34.50
Ensemble of 5 reversed LSTMs, beam size 12	**34.81**

At run time:

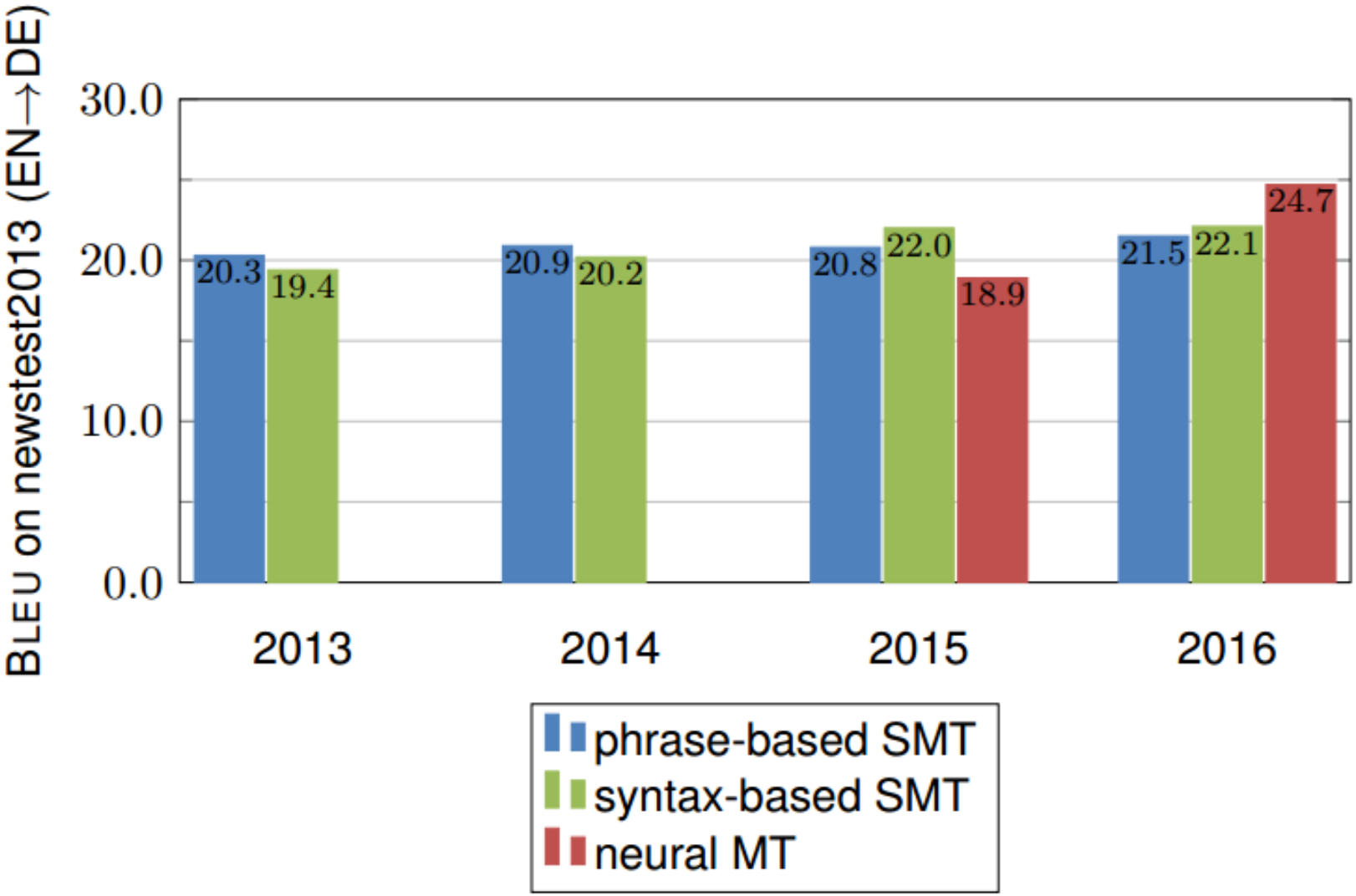
- Beam search
- Ensembling: combine N independently trained models and obtaining a "consensus" (always helps!)

End-to-End Neural Machine Translation

- Previous statistical machine translation models were complicated pipelines (word alignments $\rightarrow$ phrase table extraction $\rightarrow$ language model $\rightarrow$ decoding a phrase lattice)
- As an alternative, can do end-to-end NMT using a simple encoder-decoder
- Doesn't quite work yet, but gets close to top performance



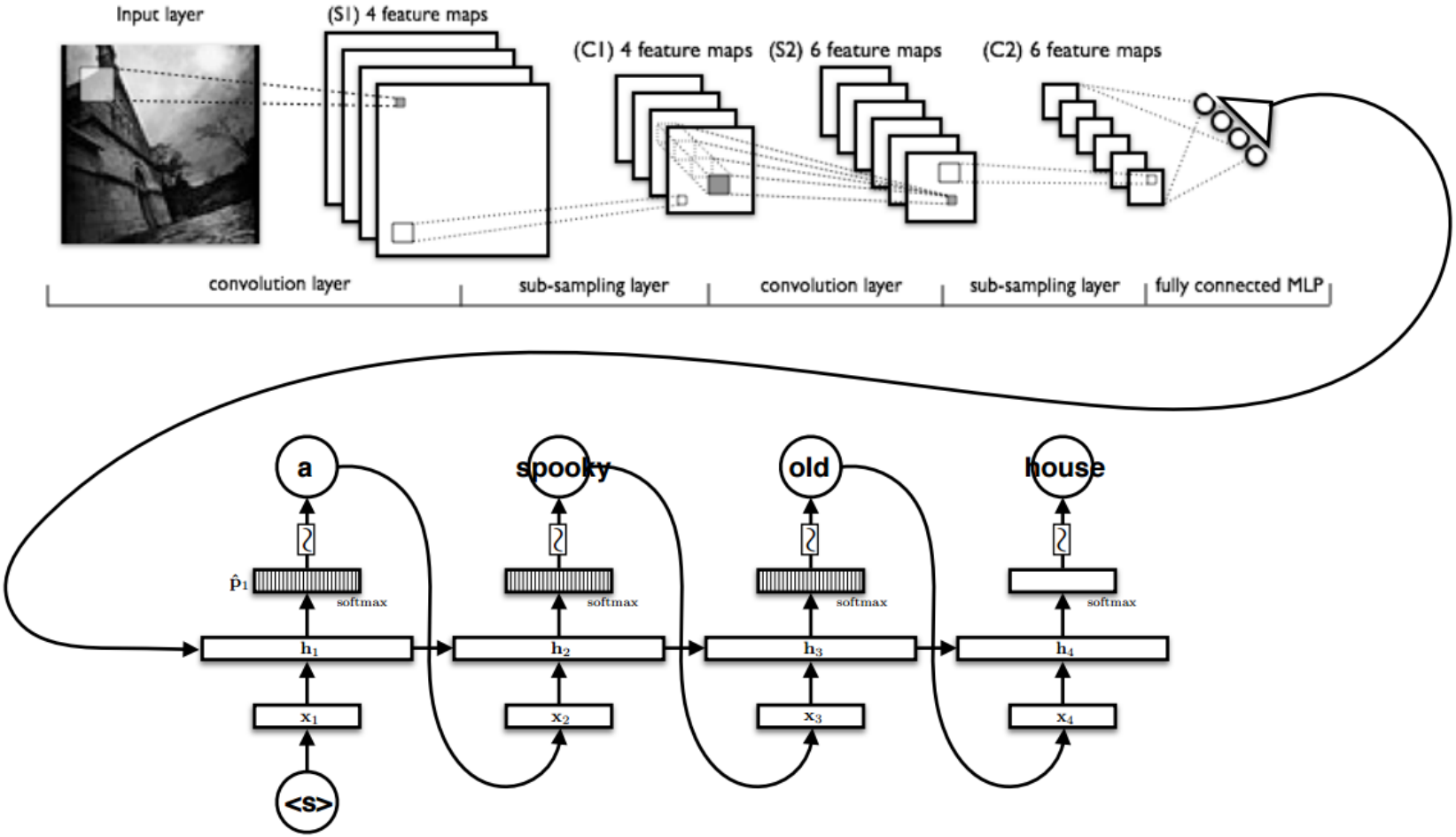
Progress in Machine Translation



Slide credit: Rico Sennrich

Encode Everything as a Vector

Works for image inputs too!



(Slide credit: Chris Dyer)

Limitations

A possible conceptual problem:

- Sentences have unbounded lengths
- Vectors have finite capacity

"You can't cram the meaning of a whole %&$# sentence into a single $&# vector!" (Ray Mooney)

A possible practical problem:

- Distance between "translations" and their sources are distant—can LSTMs learn this?

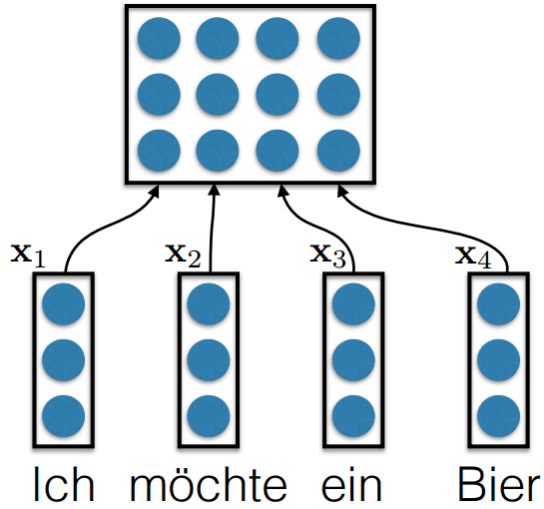
Encode Sentences as Matrices, Not Vectors

Problem with the fixed-size vector model

- Sentences are of different sizes but vectors are of the same size
- **Solution:** use matrices instead
- Fixed number of rows, but number of columns depends on the number of words
- Then, before generating each word in the decoder, use an attention mechanism to condition on the relevant source words only

How to Encode a Sentence as a Matrix?

- Define the sentence words' vectors as the columns (probably not very effective, since the word vectors carry no contextual information)

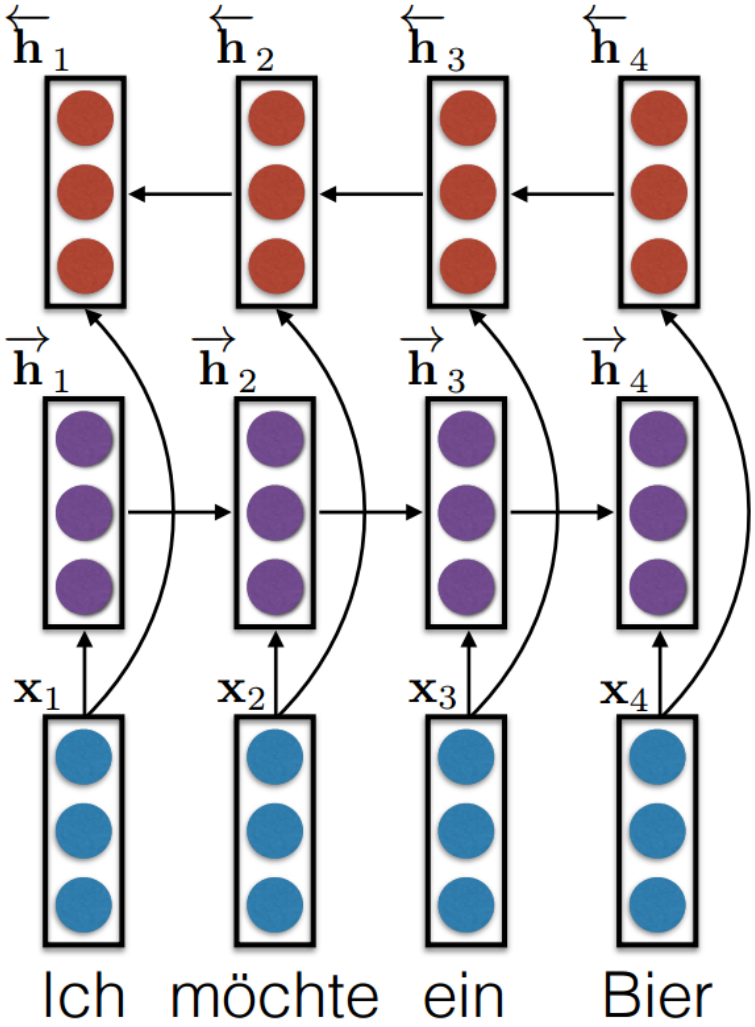


(Image credit: Chris Dyer)

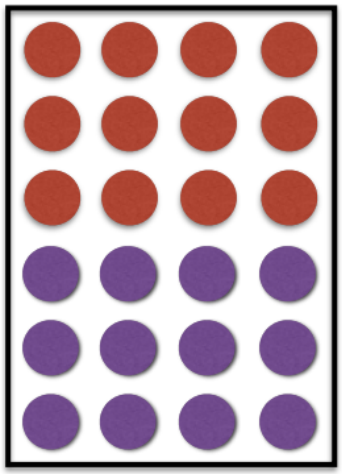
- Convolutional neural networks (Kalchbrenner et al., 2014): can capture context
- **Typical choice:** Bidirectional LSTMs (Bahdanau et al., 2015)

Bidirectional LSTM Decoder

$$\mathbf{f}_i = [\overleftarrow{\mathbf{h}}_i; \overrightarrow{\mathbf{h}}_i]$$



$\mathbf{F} \in \mathbb{R}^{2n \times |\boldsymbol{f}|}$



Ich möchte ein Bier

(Slide credit: Chris Dyer)

Generation from Matrices

We now have a matrix **F** representing the input. How to generate from it?

Answer: use attention (Bahdanau et al., 2015)

Generate the output sentence word by word using an RNN

At each output position t, the RNN receives two inputs:

- a fixed-size vector embedding of the previous output symbol y_{t-1}
- a fixed-size vector encoding a "view" of the input matrix **F**, via a weighted sum of its columns (i.e., words): $\mathbf{F}\boldsymbol{a}_t$

The weighting of the input columns at each time-step ($\boldsymbol{a}_t$) is called the attention distribution

Attention Mechanism (Bahdanau et al., 2015)

Let $\boldsymbol{s}_1, \boldsymbol{s}_2, \ldots$ be the states produced by the decoder RNN

When predicting the tth target word:

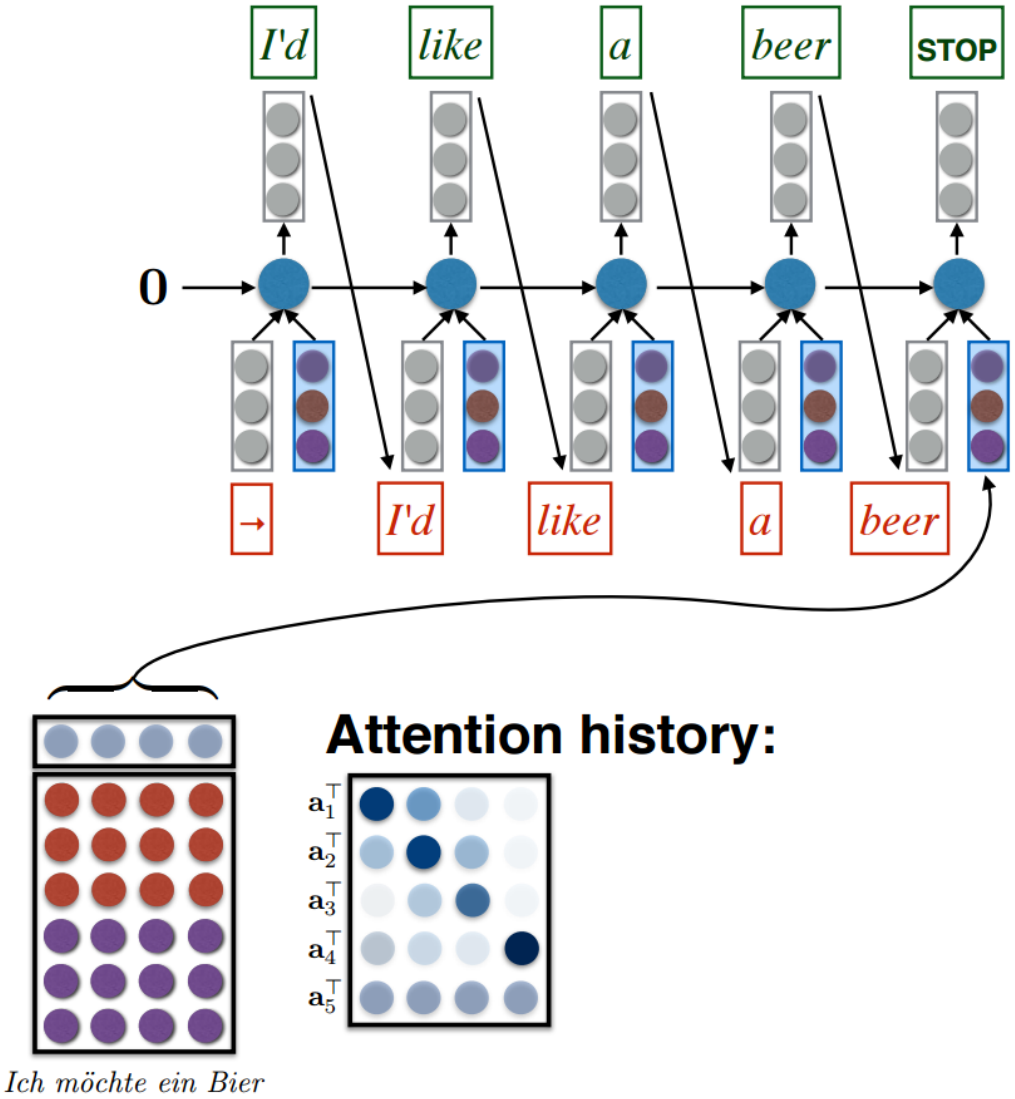
1. Compute "similarity" with each of the source words:

$$z_{t,i} = \boldsymbol{v} \cdot \boldsymbol{g}(\mathbf{W}\boldsymbol{h}_i + \mathbf{U}\boldsymbol{s}_{t-1} + \boldsymbol{b}), \quad \forall i \in [L]$$

   where $\boldsymbol{h}_i$ is the ith column of $\mathbf{F}$ (representation of the ith source word), and $\boldsymbol{v}$, $\mathbf{W}$, $\mathbf{U}$, $\boldsymbol{b}$ are parameters of the model

2. Form vector $\boldsymbol{z}_t = (z_{t,1}, \ldots, z_{t,i}, \ldots, z_{t,L})$ and compute attention $\boldsymbol{a}_t = \mathbf{softmax}(\boldsymbol{z}_t)$
3. Use attention to compute input conditioning state $\boldsymbol{c}_t = \mathbf{F}\boldsymbol{a}_t$
4. Update RNN state $\boldsymbol{s}_t$ based on $\boldsymbol{s}_{t-1}, y_{t-1}, \boldsymbol{c}_t$
5. Predict $y_t \sim p(y_t \mid \boldsymbol{s}_t)$

Encoder-Decoder with Attention



(Slide credit: Chris Dyer)

Putting It All Together

obtain input matrix $\mathbf{F}$ with a bidirectional LSTM
$t = 0$, $y_0 = \text{START}$ (the start symbol)
$\boldsymbol{s}_0 = \boldsymbol{w}$ (learned initial state)
repeat
  $t = t + 1$
  $\boldsymbol{e}_t = \boldsymbol{v} \cdot \boldsymbol{g}(\mathbf{W}\mathbf{F} + \mathbf{U}\boldsymbol{s}_{t-1} + \boldsymbol{b})$
  compute attention $\boldsymbol{a}_t = \textbf{softmax}(\boldsymbol{e}_t)$
  compute input conditioning state $\boldsymbol{c}_t = \mathbf{F}\boldsymbol{a}_t$
  $\boldsymbol{s}_t = \textbf{RNN}(\boldsymbol{s}_{t-1}, [\mathbf{E}(y_{t-1}), \boldsymbol{c}_t])$
  $y_t | y_{<t}, \boldsymbol{x} \sim \textbf{softmax}(\mathbf{P}\boldsymbol{s}_t + \boldsymbol{b})$
until $y_t \neq \text{STOP}$

Attention Mechanisms

Attention is closely related to "pooling" operations in convnets (and other architectures)

- Attention in MT plays a similar role as alignment, but leads to "soft" alignment instead of "hard" alignment
- Bahdanau et al. (2015)'s model has no bias in favor of diagonals, short jumps, fertility, etc.
- Some recent work adds some "structural" biases (Luong et al., 2015; Cohn et al., 2016)

Attention Mechanisms

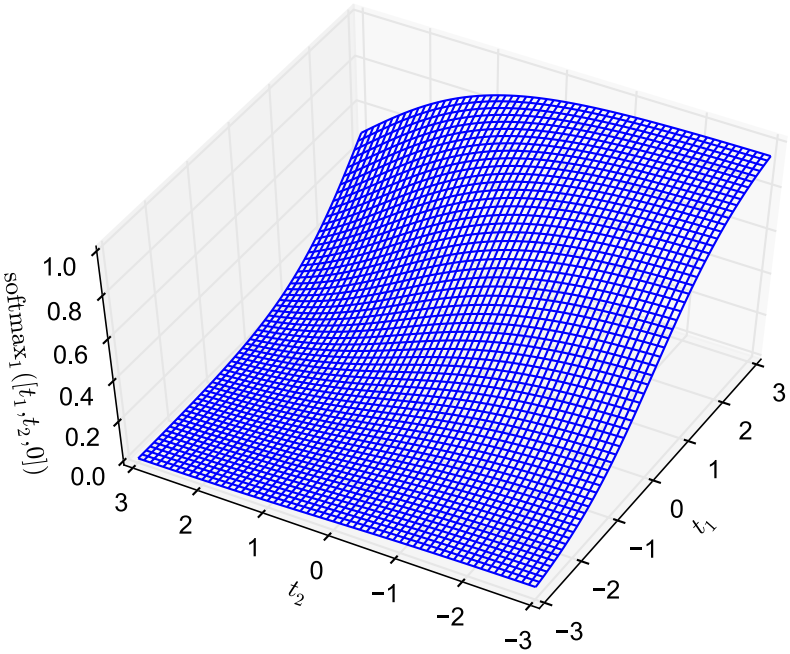
Attention is used in other problems, sometimes under different names:

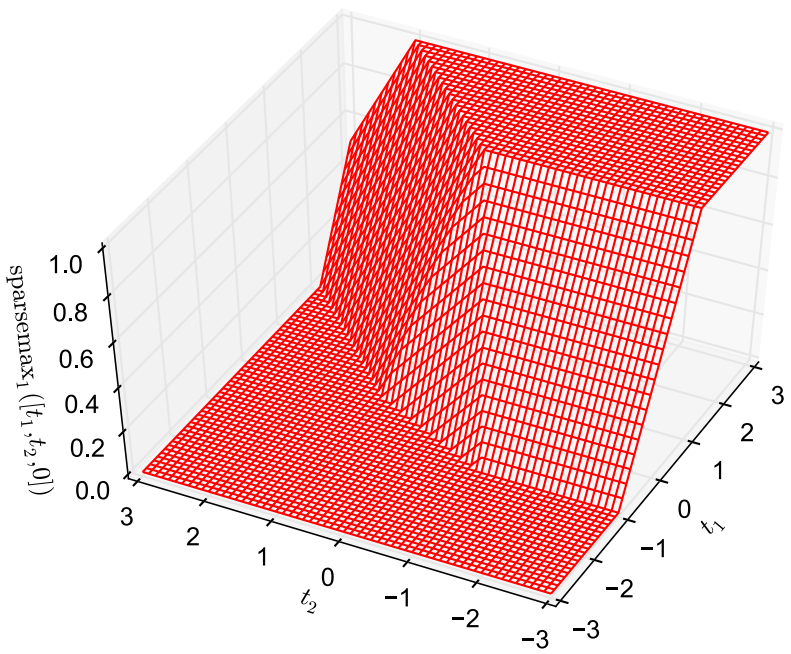
- image caption generation (Xu et al., 2015)
- speech recognition (Chorowski et al., 2015)
- memory networks for reading comprehension (Sukhbaatar et al., 2015; Hermann et al., 2015)
- neural Turing machines and other "differentiable computers" (Graves et al., 2014; Grefenstette et al., 2015)

Also: Sparse attention via **sparsemax** (Martins and Astudillo, 2016)

Sparse Attention (Martins and Astudillo, 2016)

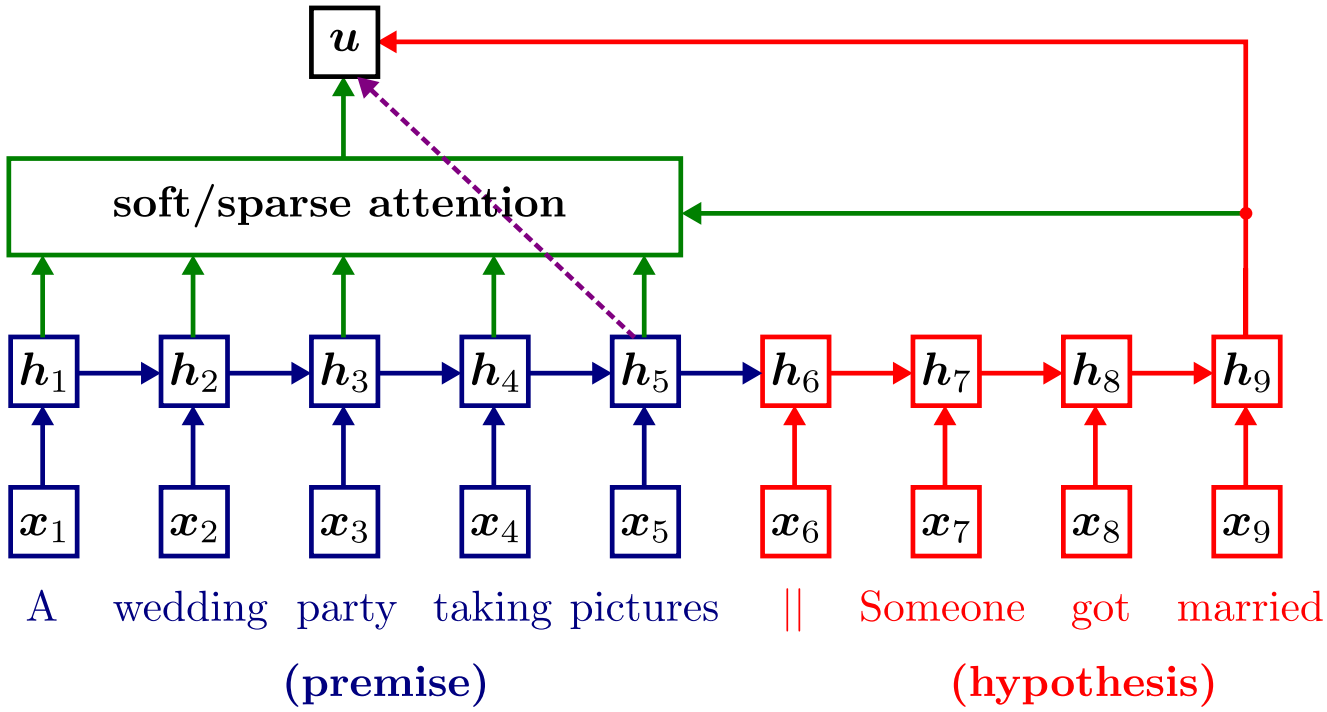
Recall the similarities/differences between **softmax** and **sparsemax**:





Example: Sparse Attention for Natural Language Inference

- SNLI corpus (Bowman et al., 2015): 570K sentence pairs (a premise and an hypothesis), labeled as **entailment**, **contradiction**, or **neutral**
- We used an attention-based architecture as Rocktäschel et al. (2015)



Example: Sparse Attention for Natural Language Inference

- *In blue*, the premise words selected by the sparse attention mechanism
- In red, the hypothesis
- Only a few words are selected, which are key for the system's decision
- The sparsemax activation yields a compact and more interpretable selection, which can be particularly useful in long sentences

---

A boy *rides on* a *camel* in a crowded area while talking on his cellphone.
—— A boy is riding an animal. [entailment]

A young girl wearing *a pink coat* plays with a *yellow* toy golf club.
—— A girl is wearing a blue jacket. [contradiction]

Two black dogs are *frolicking* around the *grass together*.
—— Two dogs swim in the lake. [contradiction]

A man wearing a yellow striped shirt *laughs* while *seated next* to another *man* who is wearing a light blue shirt and *clasping* his hands together.
—— Two mimes sit in complete silence. [contradiction]

Example: Machine Translation

Some positive examples where NMT has impressive performance:

Source	When asked about this, an official of the American administration replied: "The United States is not conducting electronic surveillance aimed at offices of the World Bank and IMF in Washington."	
PBMT	Interrogé à ce sujet, un responsable de l'administration américaine a répondu : "Les Etats-Unis n'est pas effectuer une surveillance électronique destiné aux bureaux de la Banque mondiale et du FMI à Washington".	3.0
GNMT	Interrogé à ce sujet, un fonctionnaire de l'administration américaine a répondu: "Les États-Unis n'effectuent pas de surveillance électronique à l'intention des bureaux de la Banque mondiale et du FMI à Washington".	6.0
Human	Interrogé sur le sujet, un responsable de l'administration américaine a répondu: "les Etats-Unis ne mènent pas de surveillance électronique visant les sièges de la Banque mondiale et du FMI à Washington".	6.0
Source	Martin told CNN that he asked Daley whether his then-boss knew about the potential shuffle.	
PBMT	Martin a déclaré à CNN qu'il a demandé Daley si son patron de l'époque connaissaient le potentiel remaniement ministériel.	2.0
GNMT	Martin a dit à CNN qu'il avait demandé à Daley si son patron d'alors était au courant du remaniement potentiel.	6.0
Human	Martin a dit sur CNN qu'il avait demandé à Daley si son patron d'alors était au courant du remaniement éventuel.	5.0

(From Wu et al. (2016))

Example: Machine Translation

... But also some negative examples:

- Dropping source words (lack of attention)
- Repeated source words (too much attention)

Source:	1922 in Wien geboren, studierte Mang während und nach dem Zweiten Weltkrieg Architektur an der Technischen Hochschule in Wien bei Friedrich Lehmann.
Human:	Born in Vienna in 1922, Meng studied architecture at the Technical University in Vienna under Friedrich Lehmann *during and after the second World War.*
NMT:	*Born in Vienna in 1922, Mang studied architecture at the Technical College in Vienna with Friedrich Lehmann.
Source:	Es ist schon komisch, wie dies immer wieder zu dieser Jahreszeit auftaucht.
Human:	It's funny how this always comes up at *this time* of year.
NMT:	*It's funny how **this time** to come back to **this time** of year.

Example: Machine Translation

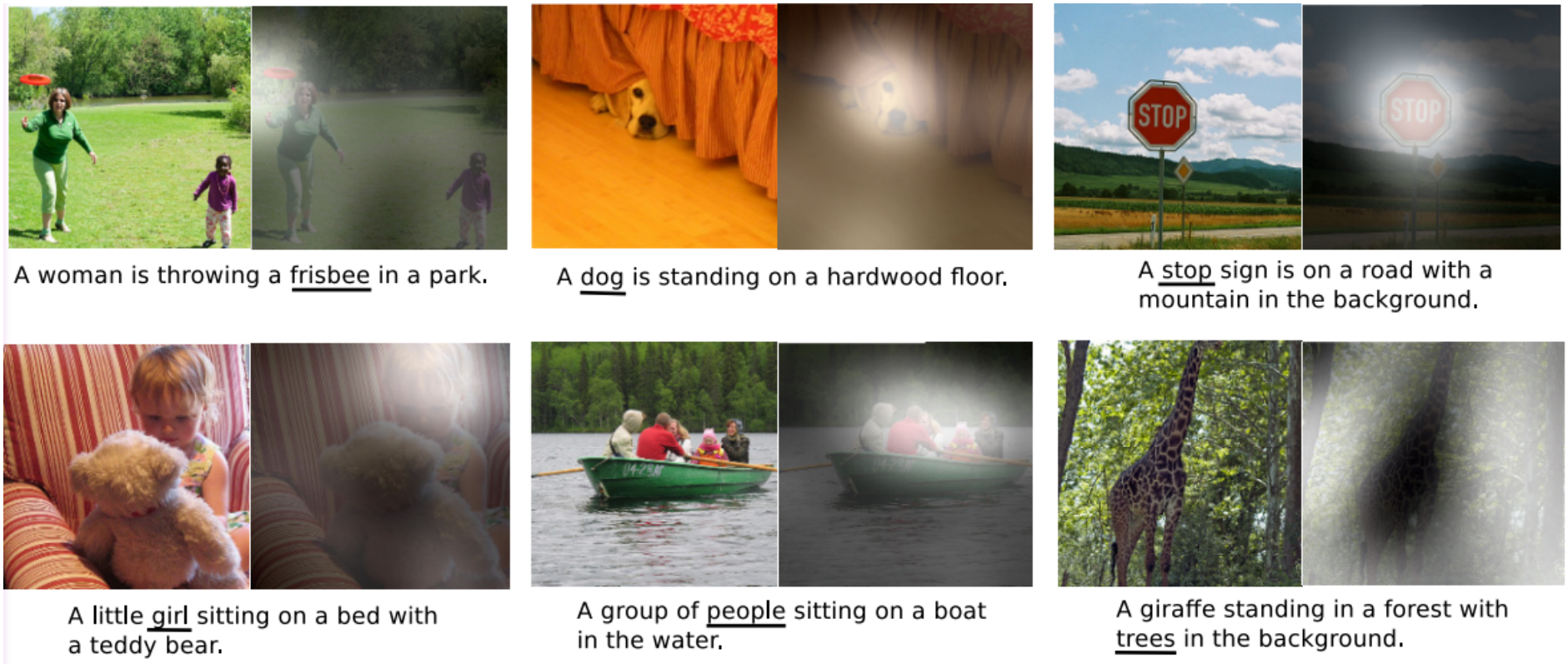
... And an example where neural MT failed miserably:

Source	A two - out walk to right fielder J . D . Martinez brought up Victor Martinez , who singled up the middle for the first run of the game .
Reference	Dva odchody pro pravého polaře J . D . Martineze vynesly Victora Martineze , který jako první oběhl všechny mety .
online-B	Dva - out chůze doprava Fielder J . D . Martinez vychován Victor Martinez , který vybral do středu za prvním spuštění hry .
uedin-nmt	Ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne , ne ,

(Credit: Barry Haddow)

Example: Caption Generation

Attention over images:



(Slide credit to Yoshua Bengio)

A More Extreme Example...

(Slide credit to Dhruv Batra)

Outline

Generative Modeling

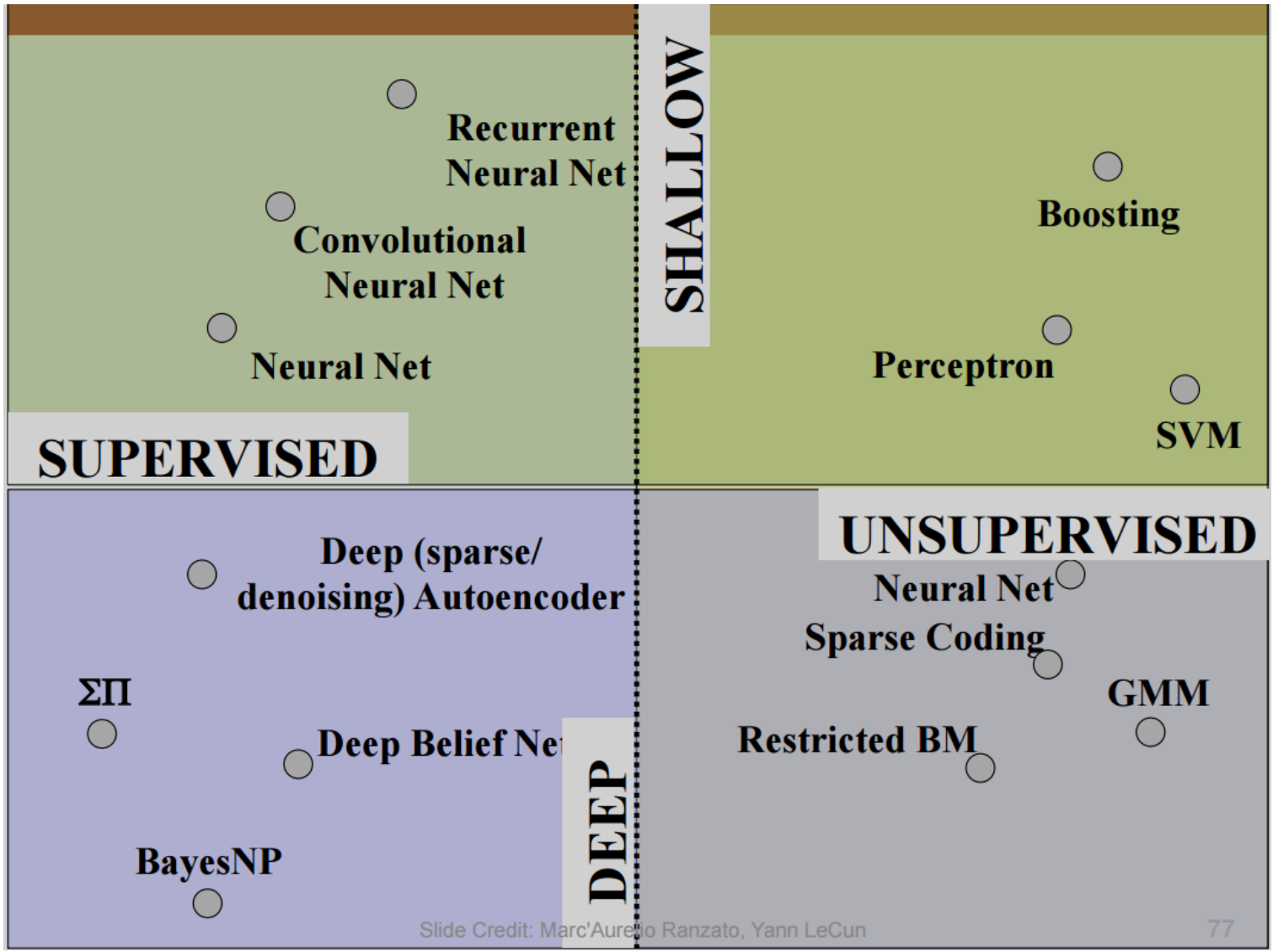
Modeling complex high-dimensional data is an open problem

Deep generative models are currently making progress on this

Roadmap:

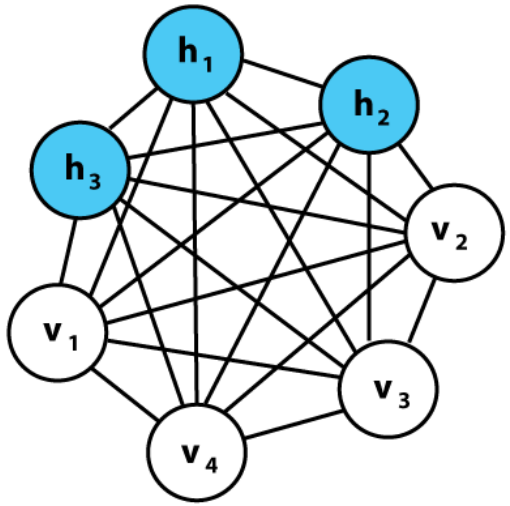
- Boltzmann machines
- Restricted Boltzmann machines
- Deep belief networks
- Variational auto-encoders
- Generative adversarial networks

The Big Picture



Boltzmann Machine

- Energy-based model to learn arbitrary probability distributions over binary vectors (Ackley et al., 1985)
- Defined over a binary random vector $\boldsymbol{x} = (\boldsymbol{v}, \boldsymbol{h}) \in \{0, 1\}^{N \times M}$:



$$p(\boldsymbol{v}, \boldsymbol{h}) = \frac{\exp(-E(\boldsymbol{v}, \boldsymbol{h}))}{Z}$$

- Some variables are observed ($\boldsymbol{v}$), others are latent ($\boldsymbol{h}$)
- Energy function:

$$E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{v}^\top \mathbf{R} \boldsymbol{v} - \boldsymbol{v}^\top \mathbf{W} \boldsymbol{h} - \boldsymbol{h}^\top \mathbf{S} \boldsymbol{h} - \boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h}$$

Boltzmann Machine

The Boltzmann machine is a universal approximator of probability mass functions over discrete variables (Le Roux and Bengio, 2008)

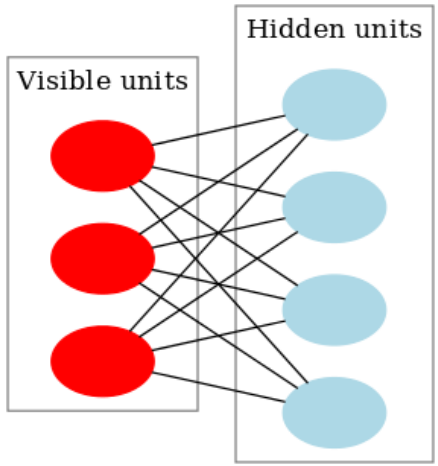
In general:

- Sampling is hard
- Inference is hard
- Learning is hard

All Boltzmann machines have an intractable partition function Z, so for learning the gradient must be approximated:

- contrastive divergence
- pseudo-likelihood
- noise-contrastive estimation
- annealed importance sampling

Restricted Boltzmann Machines



- Also called harmonium (Smolensky, 1986)
- RBMs are undirected probabilistic graphical models containing a layer of observable variables and a single layer of latent variables
- A bipartite graph, without intra-layer connections
- Energy becomes: $E(\boldsymbol{v}, \boldsymbol{h}) = -\boldsymbol{v}^\top \boldsymbol{W}\boldsymbol{h} - \boldsymbol{b}^\top \boldsymbol{v} - \boldsymbol{c}^\top \boldsymbol{h}$

Restricted Boltzmann Machines

The partition function Z is still intractable

... however, the conditional distributions $p(\boldsymbol{h} \mid \boldsymbol{v})$ and $p(\boldsymbol{v} \mid \boldsymbol{h})$ are tractable!

- easy to compute!
- easy to sample!
- can do MCMC with Gibbs sampling

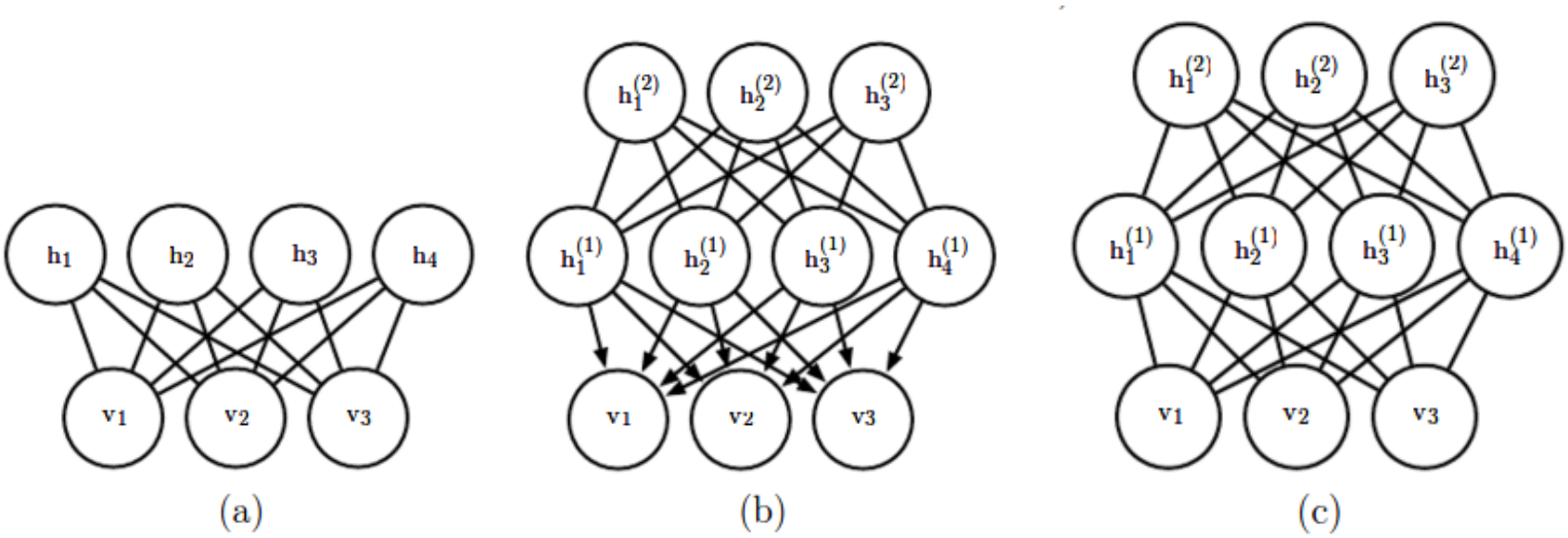
Why are the conditionals tractable?

- Because without intra-layer connections $h_1, \ldots, h_N$ are conditionally independent given $\boldsymbol{v}$, hence $p(\boldsymbol{h} \mid \boldsymbol{v})$ factors (and similarly for $p(\boldsymbol{v} \mid \boldsymbol{h})$)

RBMs are relatively straightforward to train (by approximating Z)

RBMs may be stacked (one on top of the other) to form deeper models (e.g. DBNs, DBMs)

Some RBM's Friends



(Image from Goodfellow et al. (2016))

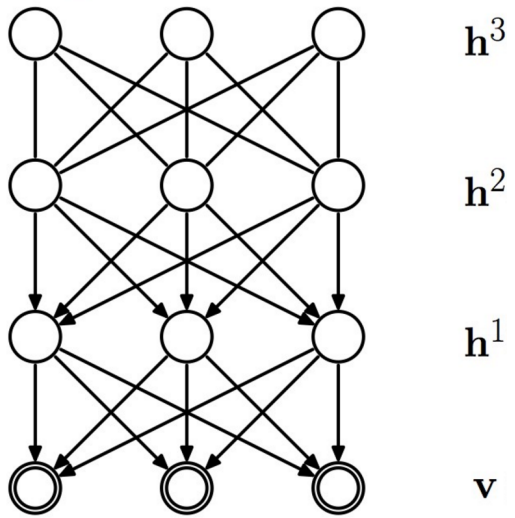
(a) Restricted Boltzmann machine (RBM)

(b) Deep belief network (DBN): hybrid directed/undirected GM with multiple latent layers

(c) Deep Boltzmann machine (DBM): undirected GM with several layers of latent variables

Deep Belief Networks (Hinton et al., 2006)

- Began the deep learning renaissance!
- Before DBNs: deep models were considered too difficult to optimize
- Today, DBNs mostly fell out of favor
- **Idea:** several layers of latent variables, again no intra-layer connections



- The connections between the top two layers are undirected:

$$p(\boldsymbol{h}^{(\ell)}, \boldsymbol{h}^{(\ell-1)}) \propto \exp(-\boldsymbol{h}^{(\ell-1)^\top} \mathbf{W}^{(\ell)} \boldsymbol{h}^{(\ell)} - (\boldsymbol{b}^{(\ell-1)})^\top \boldsymbol{h}^{(\ell-1)} - \boldsymbol{b}^{(\ell)^\top} \boldsymbol{h}^{(\ell)})$$

- The connections between all other layers are directed:

$$\begin{aligned} p(h_i^k = 1 \mid \boldsymbol{h}^{(k+1)}) &= \sigma(b_i^{(k)} + \mathbf{W}_{:,i}^{(k+1)^\top} \boldsymbol{h}^{(k+1)}) \\ p(v_i = 1 \mid \boldsymbol{h}^{(1)}) &= \sigma(b_i^{(0)} + \mathbf{W}_{:,i}^{(1)^\top} \boldsymbol{h}^{(1)}) \end{aligned}$$

Deep Belief Networks

Inference in a deep belief network is intractable:

- "explaining away" effect within each directed layer
- interaction between the two hidden layers with undirected connections

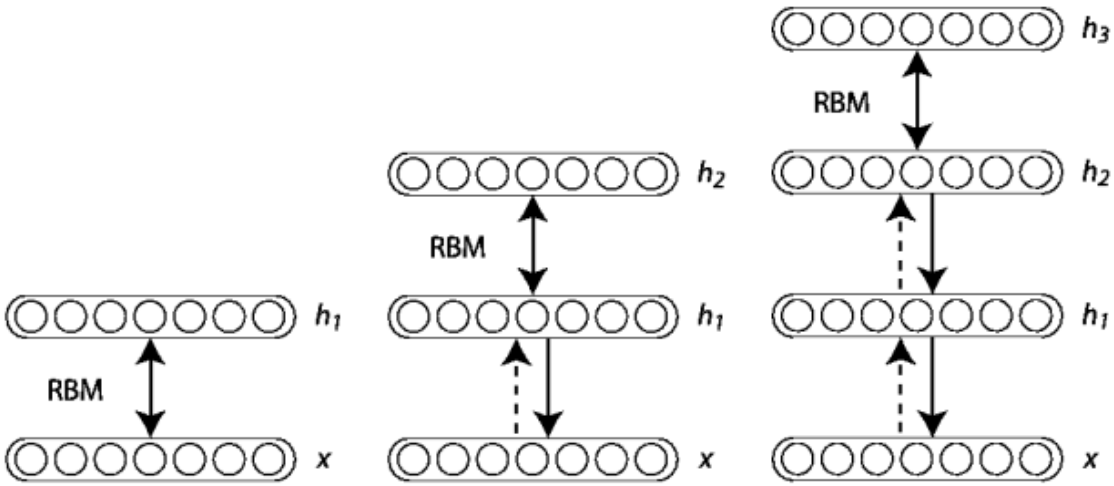
Evaluating or maximizing the standard evidence lower bound on the log-likelihood is also intractable

How to train a DBN?

Deep Belief Networks

How to train a DBN?

- Layerwise training: begin by training a RBM for the first layer; then train a second RBM to model the distribution defined by sampling the hidden units of the first RBM, etc.



Most interest in DBNs arose from their ability to improve classification:

- take DBN's weights and define a MLP (discriminative fine-tuning)

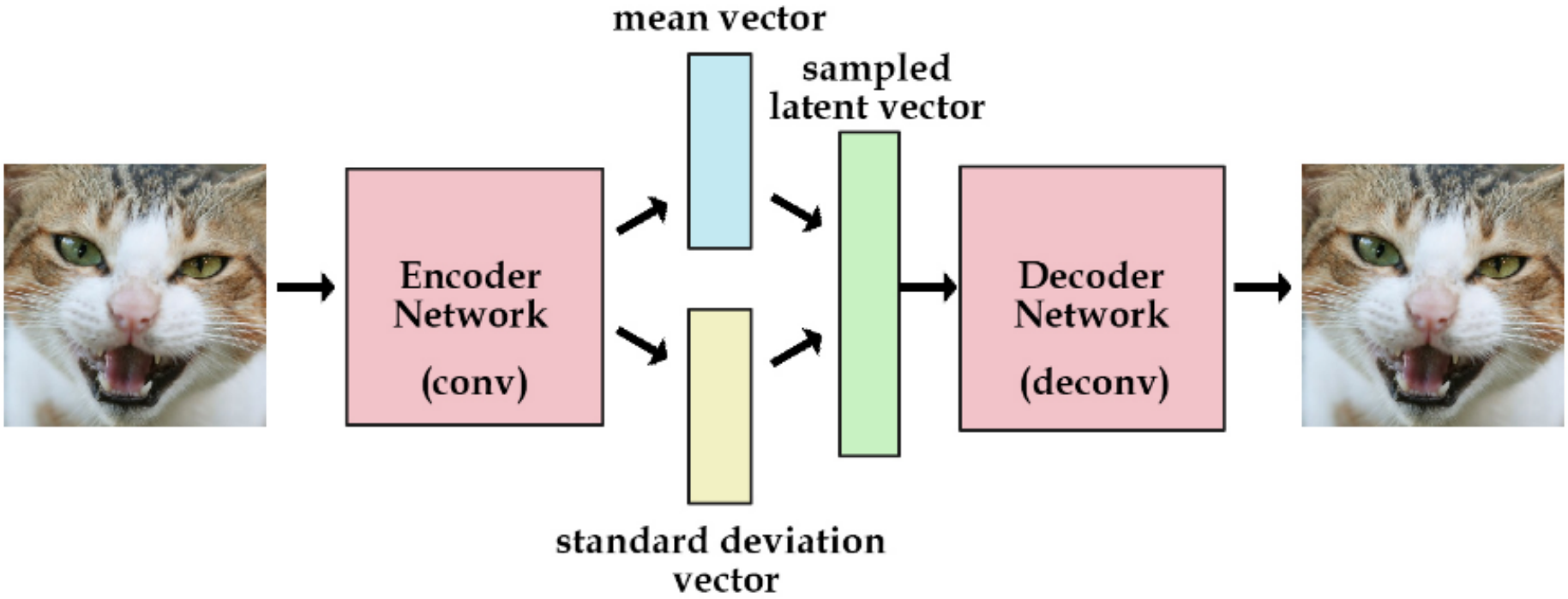
Other Deep Generative Models

- Deep Boltzmann Machines
- Gaussian-Bernoulli RBMs
- Convolutional Boltzmann Machines
- Sigmoid Belief Nets
- Variational Auto-Encoders
- Generative Adversarial Networks
- Convolutional Generative Networks
- Auto-Regressive Networks
- Generative Stochastic Networks

Other Deep Generative Models

- Deep Boltzmann Machines
- Gaussian-Bernoulli RBMs
- Convolutional Boltzmann Machines
- Sigmoid Belief Nets
- Variational Auto-Encoders
- Generative Adversarial Networks
- Convolutional Generative Networks
- Auto-Regressive Networks
- Generative Stochastic Networks

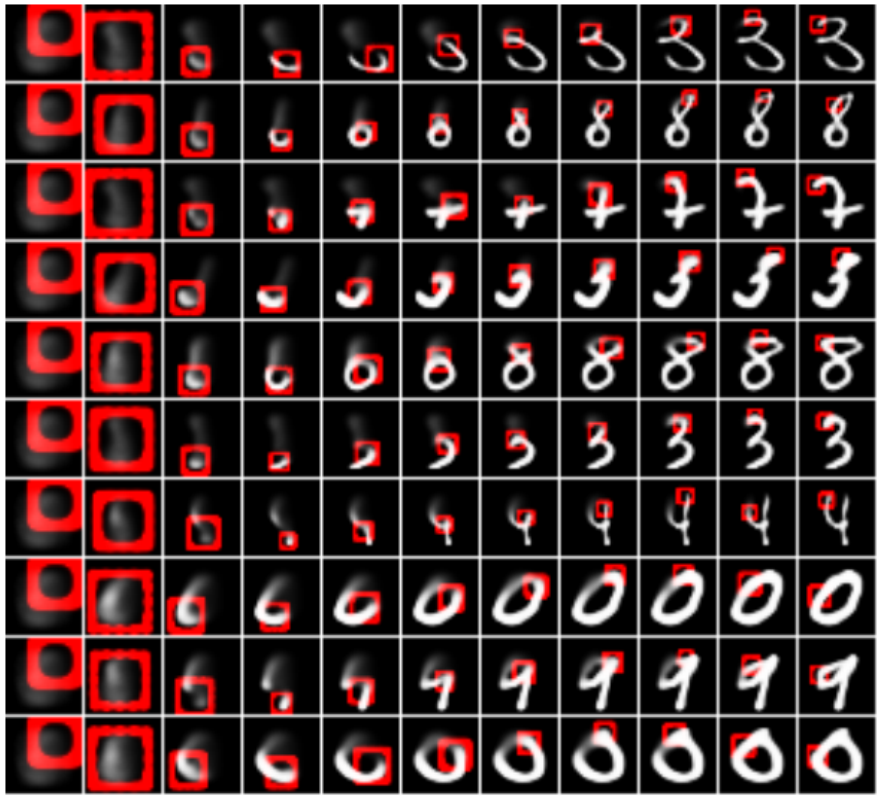
Variational Auto-Encoders (Kingma and Ba, 2014)



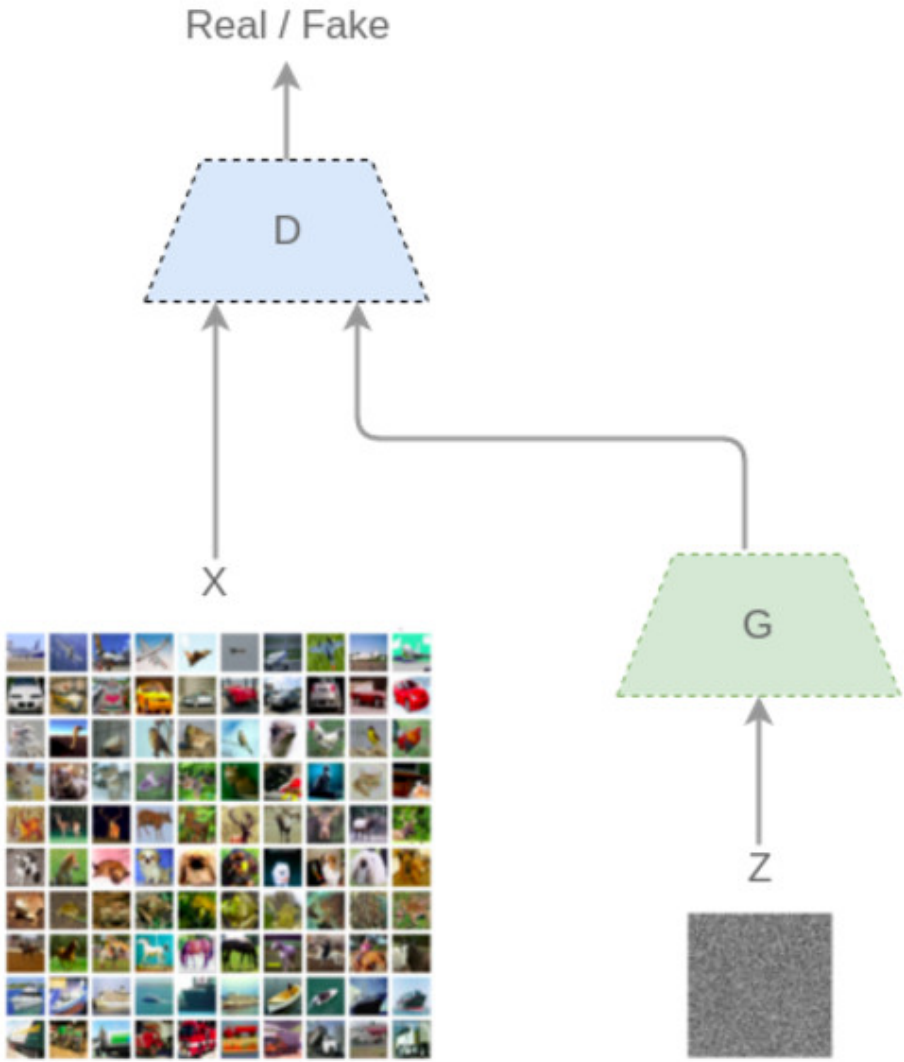
http://kvfrans.com/variational-autoencoders-explained

- Decoder computes $p(\boldsymbol{z})$ and $p_{\theta}(\boldsymbol{x} \mid \boldsymbol{z})$
- Encoder computes $q_{\varphi}(\boldsymbol{z} \mid \boldsymbol{x}) = \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}_z(\boldsymbol{x}), \boldsymbol{\sigma}_z(\boldsymbol{x}))$ with variational parameters φ
- Loss function: lower variational bound of the log-likelihood

DRAW: Deep Recurrent Attentive Writer (Gregor et al., 2015)



Generative Adversarial Networks (GANs) (Goodfellow et al., 2014)



Images Generated by GANs

(https://tryolabs.com/blog/2016/12/06/major-advancements-deep-learning-2016/)

Conclusions

Deep learning achieved a lot of recent breakthroughs

It's getting mainstream and will have a increasingly impact in our lives

Part of the success is that neural networks learn good representations, are excellent function approximators, and now we have enough data to make them generalize well (and enough computation power to train them)

Many architectures for different input/output formats: feedforward, convolutional, recurrent

Most successes are for supervised learning—unsupervised is pretty much unsolved

Things we didn't talk about:

- Neural Turing machines and differentiable computers
- Deep reinforcement learning (lots of exciting recent results)
- ...

Thank you!

Questions?

References I

Ackley, D., Hinton, G., and Sejnowski, T. (1985). A learning algorithm for Boltzmann machines. *Cognitive science*, 9(1):147–169.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural Machine Translation by Jointly Learning to Align and Translate. In *International Conference on Learning Representations*.

Bengio, Y., Ducharme, R., Vincent, P., and Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb):1137–1155.

Bowman, S. R., Angeli, G., Potts, C., and Manning, C. D. (2015). A Large Annotated Corpus for Learning Natural Language Inference. In *Proc. of Empirical Methods in Natural Language Processing*.

Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning Phrase Representations Using RNN Encoder-Decoder for Statistical Machine Translation. In *Proc. of Empirical Methods in Natural Language Processing*.

Chorowski, J. K., Bahdanau, D., Serdyuk, D., Cho, K., and Bengio, Y. (2015). Attention-based Models for Speech Recognition. In *Advances in Neural Information Processing Systems*, pages 577–585.

Cireşan, D., Meier, U., Masci, J., and Schmidhuber, J. (2011). A committee of neural networks for traffic sign classification. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1918–1921. IEEE.

Ciresan, D. C., Meier, U., Gambardella, L. M., and Schmidhuber, J. (2011a). Convolutional neural network committees for handwritten character classification. In *2011 International Conference on Document Analysis and Recognition*, pages 1135–1139. IEEE.

Ciresan, D. C., Meier, U., Masci, J., Maria Gambardella, L., and Schmidhuber, J. (2011b). Flexible, high performance convolutional neural networks for image classification. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 1237.

Cohn, T., Hoang, C. D. V., Vymolova, E., Yao, K., Dyer, C., and Haffari, G. (2016). Incorporating structural alignment biases into an attentional neural translation model. *arXiv preprint arXiv:1601.01085*.

Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., and Kuksa, P. (2011). Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537.

Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314.

References II

Duchi, J., Hazan, E., and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159.

Elman, J. L. (1990). Finding structure in time. *Cognitive science*, 14(2):179–211.

Erhan, D., Bengio, Y., Courville, A., Manzagol, P.-A., Vincent, P., and Bengio, S. (2010). Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660.

Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202.

Gal, Y. and Ghahramani, Z. (2015). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. *arXiv preprint arXiv:1506.02142*.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, volume 9, pages 249–256.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep Sparse Rectifier Neural Networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep Learning. Book in preparation for MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural Turing Machines. *arXiv preprint arXiv:1410.5401*.

Grefenstette, E., Hermann, K. M., Suleyman, M., and Blunsom, P. (2015). Learning to Transduce with Unbounded Memory. In *Advances in Neural Information Processing Systems*, pages 1819–1827.

Gregor, K., Danihelka, I., Graves, A., Rezende, D. J., and Wierstra, D. (2015). Draw: A recurrent neural network for image generation. In *Proc. of the International Conference on Machine Learning*.

Hermann, K. M., Kocisky, T., Grefenstette, E., Espeholt, L., Kay, W., Suleyman, M., and Blunsom, P. (2015). Teaching Machines to Read and Comprehend. In *Advances in Neural Information Processing Systems*, pages 1684–1692.

Hinton, G. E. (1984). Distributed representations.

Hinton, G. E., Osindero, S., and Teh, Y.-W. (2006). A fast learning algorithm for deep belief nets. *Neural computation*, 18(7):1527–1554.

References III

Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366.

Kalchbrenner, N., Grefenstette, E., and Blunsom, P. (2014). A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*.

Kingma, D. and Ba, J. (2014). Adam: A Method for Stochastic Optimization. In *Proc. of International Conference on Learning Representations*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Le Roux, N. and Bengio, Y. (2008). Representational power of restricted boltzmann machines and deep belief networks. *Neural computation*, 20(6):1631–1649.

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551.

Luong, M.-T., Pham, H., and Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*.

Martins, A. F. T. and Astudillo, R. (2016). From Softmax to Sparsemax: A Sparse Model of Attention and Multi-Label Classification. In *Proc. of the International Conference on Machine Learning*.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., and Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119.

Minsky, M. and Papert, S. (1969). Perceptrons.

Novikoff, A. B. (1962). On convergence proofs for perceptrons. In *Symposium on the Mathematical Theory of Automata*.

Oord, A. v. d., Kalchbrenner, N., and Kavukcuoglu, K. (2016). Pixel Recurrent Neural Networks. In *Proc. of the International Conference on Machine Learning*.

References IV

Rocktäschel, T., Grefenstette, E., Hermann, K. M., Kočiskỳ, T., and Blunsom, P. (2015). Reasoning about Entailment with Neural Attention. *arXiv preprint arXiv:1509.06664*.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386.

Smolensky, P. (1986). Information processing in dynamical systems: Foundations of harmony theory. Technical report, DTIC Document.

Snoek, J., Larochelle, H., and Adams, R. P. (2012). Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, pages 2951–2959.

Socher, R., Lin, C. C., Manning, C., and Ng, A. Y. (2011). Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 129–136.

Srivastava, N., Hinton, G. E., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958.

Sukhbaatar, S., Szlam, A., Weston, J., and Fergus, R. (2015). End-to-End Memory Networks. In *Advances in Neural Information Processing Systems*, pages 2431–2439.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems*, pages 3104–3112.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9.

Tieleman, T. and Hinton, G. (2012). Rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4(2).

Wager, S., Wang, S., and Liang, P. S. (2013). Dropout training as adaptive regularization. In *Advances in neural information processing systems*, pages 351–359.

Wan, L., Zeiler, M., Zhang, S., Cun, Y. L., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *Proc. of the International Conference on Machine Learning*, pages 1058–1066.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

Xu, K., Ba, J., Kiros, R., Courville, A., Salakhutdinov, R., Zemel, R., and Bengio, Y. (2015). Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *International Conference on Machine Learning*.

Zeiler, M. D. and Fergus, R. (2013). Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*.