15-381/781

Fall 2016

Deep Learning

Instructors: Ariel Procaccia and Emma Brunskill

Slides courtesy of Zico Kolter

Outline

Introduction

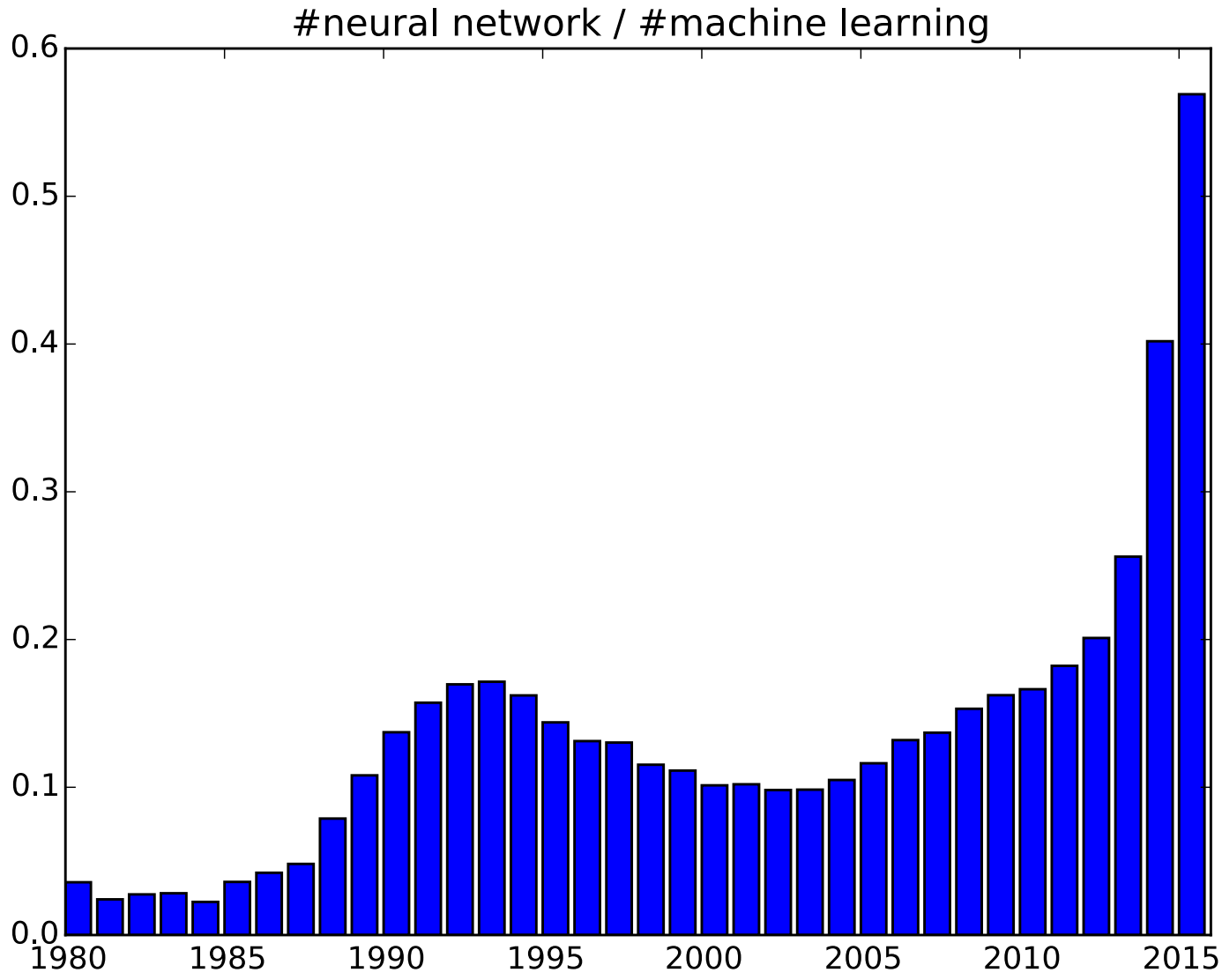
Machine learning with neural networks

Training neural networks

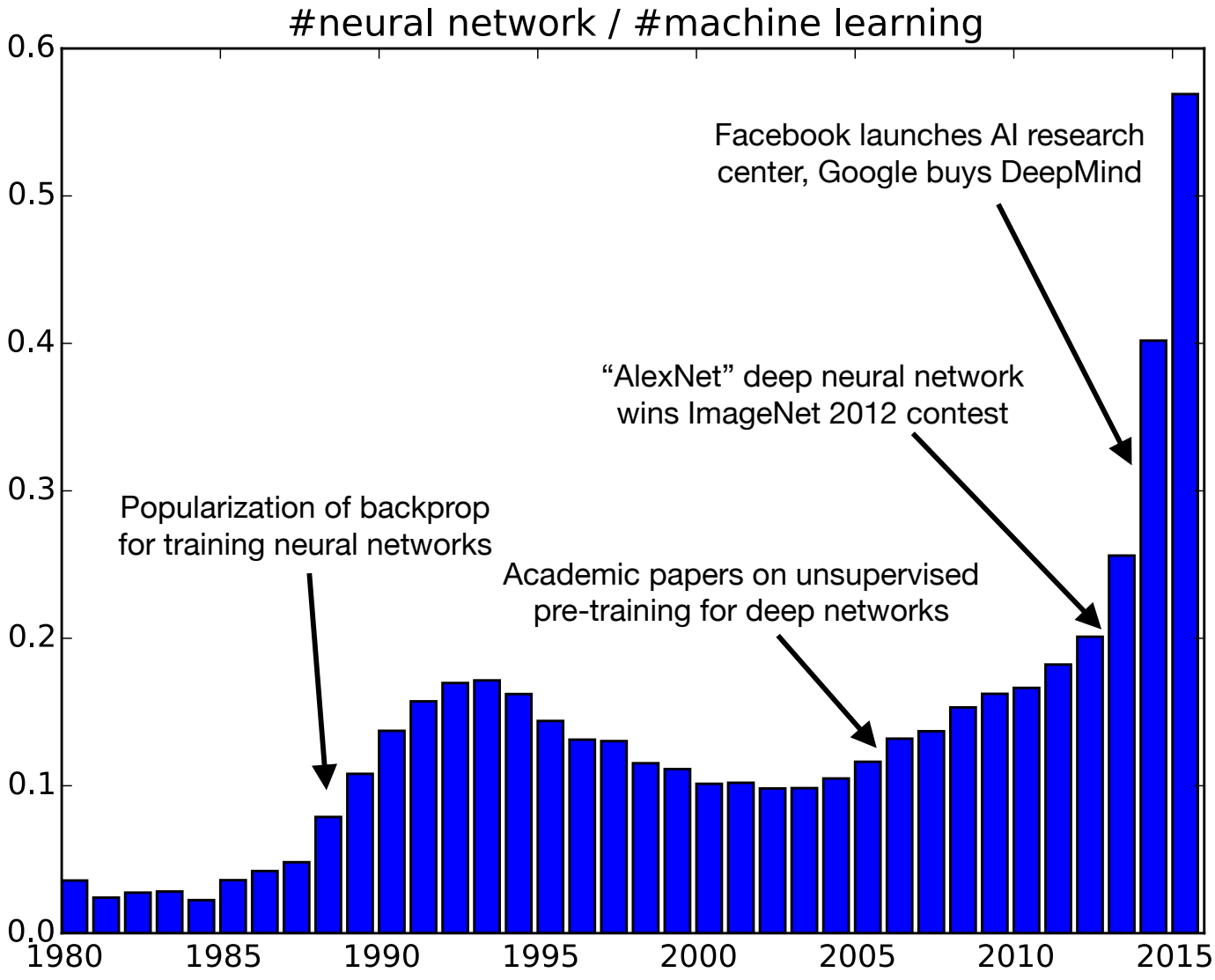
Convolutional neural networks

Recurrent neural networks

Deep reinforcement learning

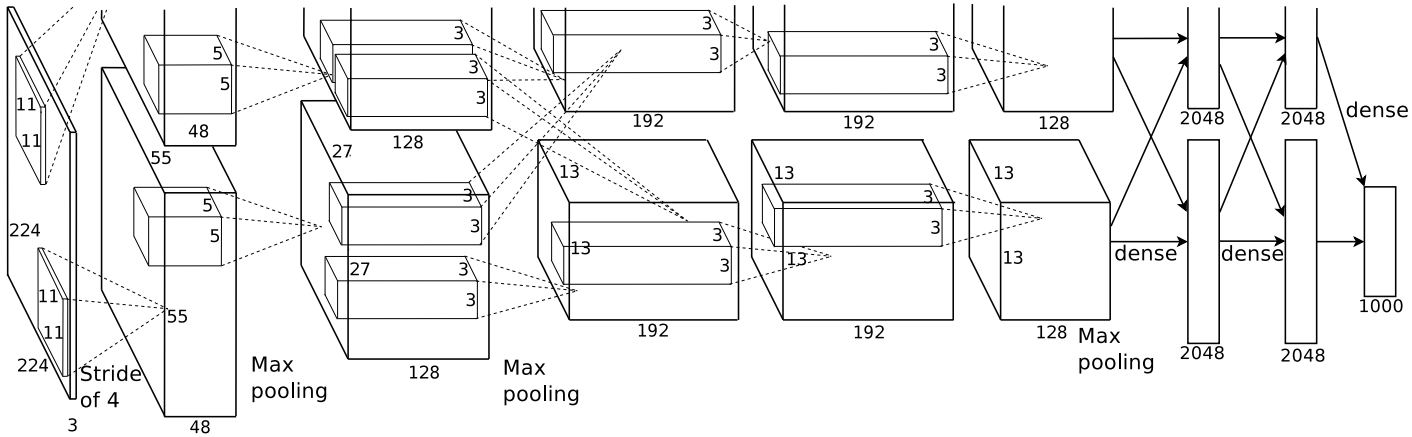


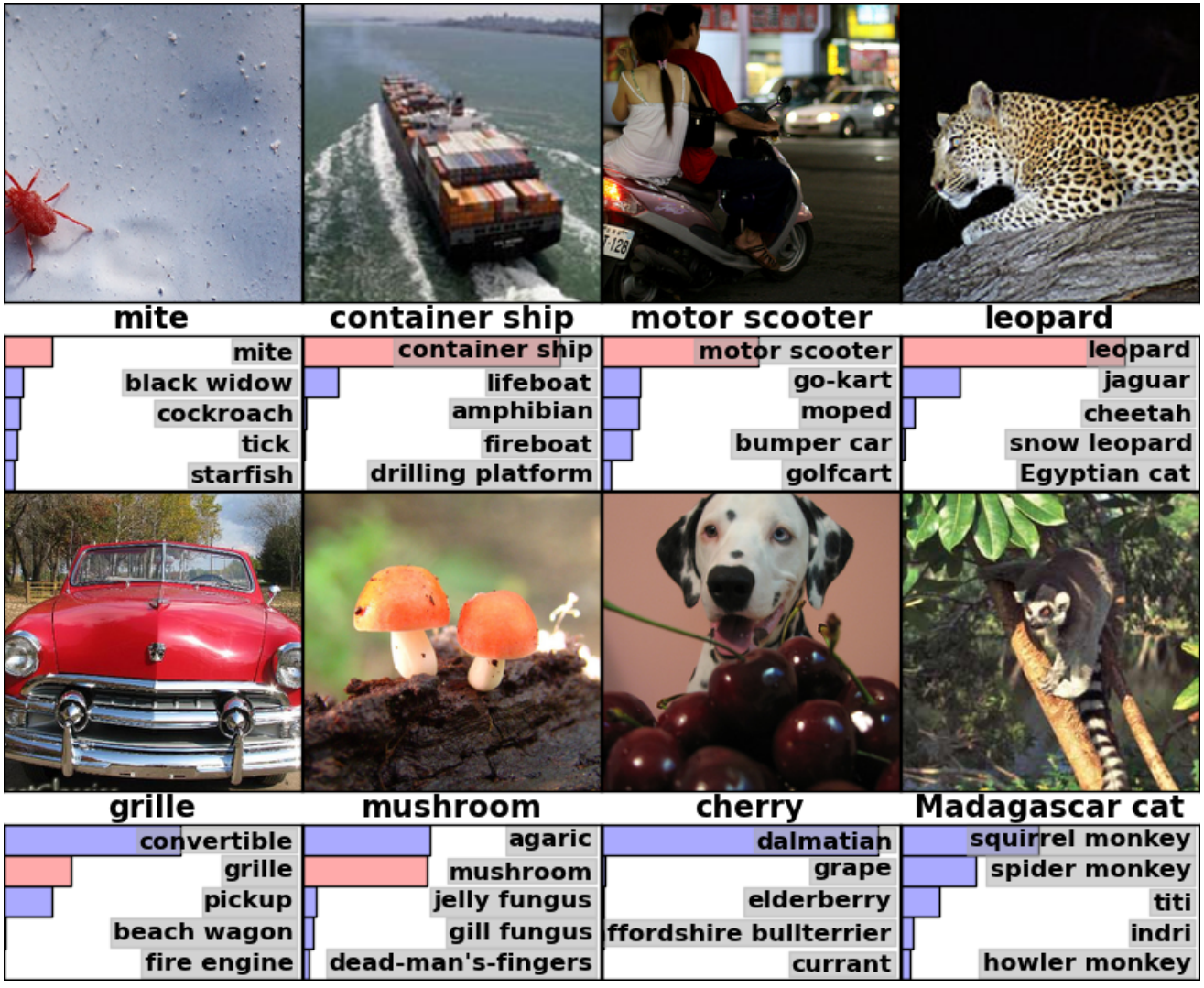
Google scholar counts of papers containing “neural network” divided by count of papers containing “machine learning”



A non-exhaustive list of some of the important events that impacted this trend

“AlexNet” (Krizhevsky et al., 2012), winning entry of ImageNet 2012 competition with a Top-5 error rate of 15.3% (next best system with highly engineered features based upon SIFT got 26.1% error)

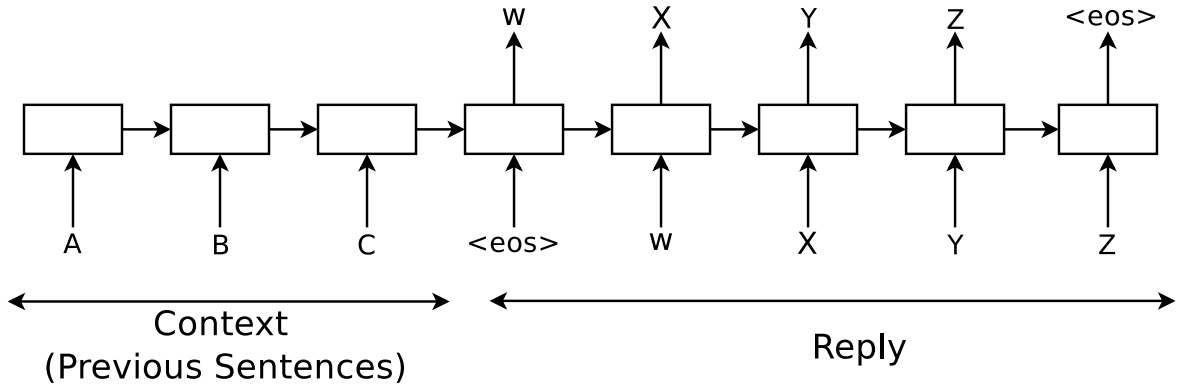




Some classification results from AlexNet

Google Deep Dream software: adjust input images (by gradient descent) to strengthen the activations that are present in an image

Question answering network (Vinyals and Le, 2015), using sequence to sequence learning method (Sutskever et al., 2014)

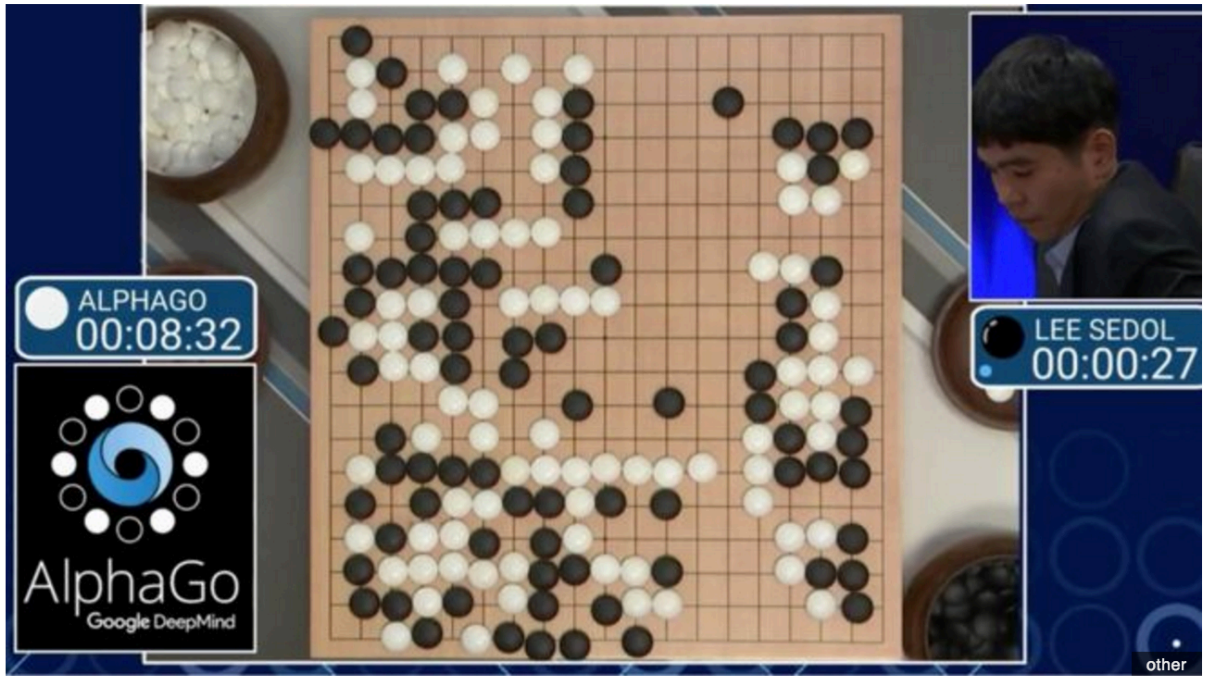


Machine: *hi*
Human: *hello*
Machine: *could you please let me know what are the operating systems you are using ?*
Human: *i am using linux*
Machine: *ok it was in the ssl vpn on which you stated it was broken*
Machine: *please check if you are logged in to corp network*
Machine: *what is the error that you are running please*
Human: *i am seeing an error related to vpn*
Machine: *what is the error message that you are getting when connecting to vpn using network connect ?*
Human: *connection refused or something like that*
Machine: *may i know the version of network connect you connect ?*
Human: *i am not sure i know that*
Machine: *is the network connect prompting for any user certificate*
Machine: *?*
Human: *yes*
Machine: *are you at home or at work at the moment ?*
Machine: *office ?*
Human: *i am at home*
Machine: *try this*
Machine: *goto <URL> page and enter your name and password*
Human: *done*
Machine: *try logging in with <NAME> and check if its working fine now*
Human: *yes , now it works !*
Machine: *great*
Machine: *anything else that i can help*
Human: *no , that was very useful*
Machine: *you are other than <NAME> :)*
Machine: *have a nice night*

AlphaGo (Silver et al., 2016) beats Lee Sedol in 5 game competition

Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol

12 March 2016 | Technology



A computer program has beaten a master Go player 3-0 in a best-of-five competition, in what is seen as a landmark moment for artificial intelligence.

Google's AlphaGo program was playing against Lee Se-dol in Seoul, in South Korea.

Outline

Introduction

Machine learning with neural networks

Training neural networks

Convolutional neural networks

Recurrent neural networks

Deep reinforcement learning

Supervised learning setup

Input features $x^{(i)} \in \mathbb{R}^n$

Outputs $y^{(i)} \in \mathcal{Y}$ (e.g. $\mathbb{R}$, $\{-1, +1\}$, $\{1, \ldots, p\}$)

Model parameters $\theta \in \mathbb{R}^k$

Hypothesis function $h_\theta : \mathbb{R}^n \to \mathbb{R}$

Loss function $\ell : \mathbb{R} \times \mathcal{Y} \to \mathbb{R}_+$

Machine learning optimization problem

$$\underset{\theta}{\text{minimize}} \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

We mainly considered the linear hypothesis class

$$h_\theta(x^{(i)}) = \theta^T \phi(x^{(i)})$$

for some set of non-linear features $\phi : \mathbb{R}^n \to \mathbb{R}^k$

(Note: previously, we just directly included the non-linear features in $x^{(i)}$, but here we separate them for clarity)

Example

$$x^{(i)} = [\text{temperature for day } i]$$

$$\phi(x^{(i)}) = \begin{bmatrix} 1 \\ x^{(i)} \\ {x^{(i)}}^2 \\ \vdots \end{bmatrix}$$

Challenges with linear models

Linear models crucially depend on choosing “good” features

Some “standard” choices: polynomial features, radial basis functions, random features (surprisingly effective)

But, many specialized domains required highly engineered special features

- E.g., computer vision tasks used Haar features, SIFT features, every 10 years or so someone would engineer a new set of features

Key question: can we come up with an algorithm that will automatically *learn* the features themselves?

Feature learning, take one

Instead of a simple linear classifier, let's consider a two-stage hypothesis class where one linear function creates the features, another models the classifier

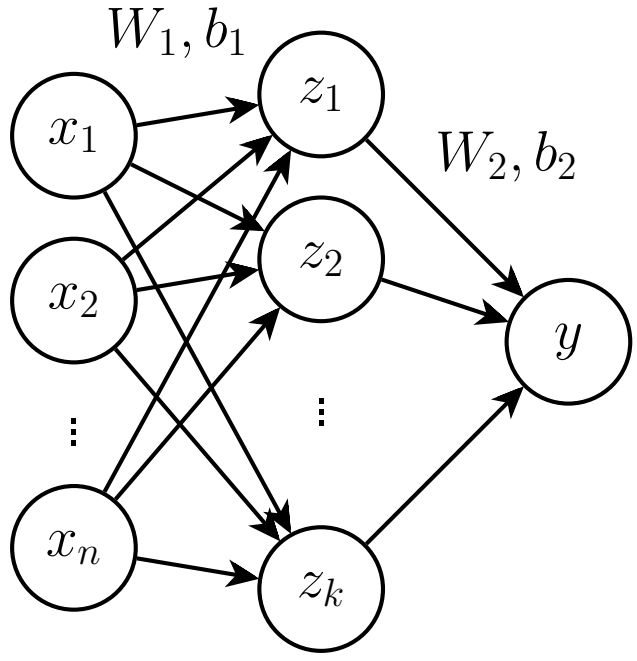
$$h_\theta(x) = W_2\phi(x) + b_2 = W_2(W_1 x + b_1) + b_2$$

where

$$\theta = \{W_1 \in \mathbb{R}^{n\times k}, b_1 \in \mathbb{R}^k, W_2 \in \mathbb{R}^{1\times k}, b_2 \in \mathbb{R}\}$$

Note that in this notation, we're explicitly separating the parameters on the "constant feature" into the b terms

Graphical depiction of the above function



But there is a problem:

$$h_\theta(x) = W_2(W_1 x + b_1) + b_2 = \tilde{W} x + \tilde{b} \tag{1}$$

in other words, we are still just using a normal linear classifier (the apparent added complexity is not giving us any additional representational power)

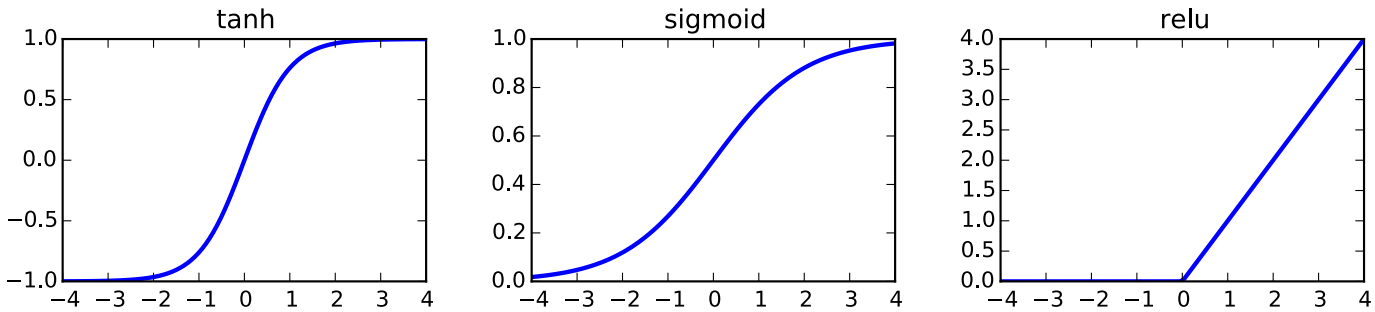
Neural networks

Neural networks are a simple extension of this idea, where we additionally apply a *non-linear* function after each linear transformation

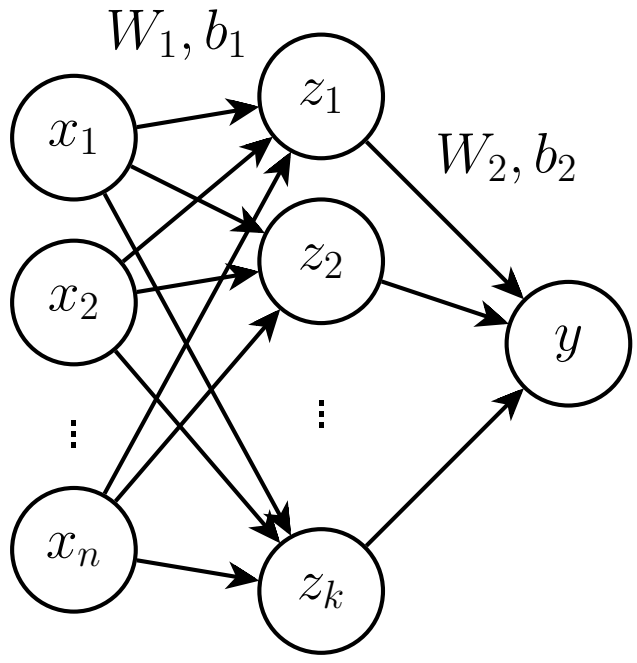
$$h_\theta(x) = f_2(\, W_2 f_1(\, W_1 x + b_1) + b_2)$$

where $f_1, f_2 : \mathbb{R} \to \mathbb{R}$ are some non-linear functions (applied elementwise to vectors)

Common choices for f_i are hyperbolic tangent $\tanh(x) = (e^{2x} - 1)/(e^{2x} + 1)$, sigmoid $\sigma(x) = 1/(1 + e^{-x})$, or rectified linear unit $f(x) = \max\{0, x\}$



We draw these the same as before (non-linear functions are virtually always implied in the neural network setting)



Middle layer z is referred to as the *hidden layer* or *activations*

These are the learned features, nothing in the data that prescribes what values these should take, left up to the algorithm to decide

Properties of neural networks

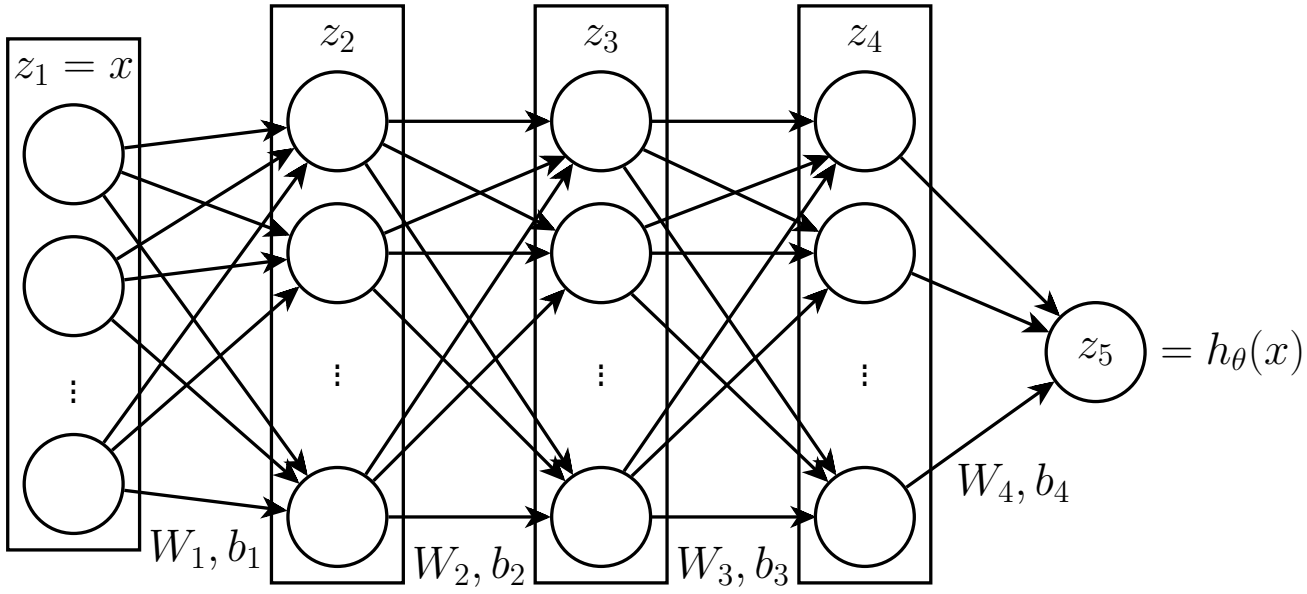
It turns out that a neural network with a single hidden layer (and a suitably large number of hidden units) is a *universal function approximator*, can approximate *any* function over the input arguments (but this is actually not very useful in practice, c.f. polynomials fitting any sets of points for high enough degree)

The hypothesis class h_θ is not a convex function of the parameters $\theta = \{W_i, b_i\}$, so we must resort to non-convex optimization methods

Architectural choices (how many layers, how are they connected), become important free parameters, more on this later

Deep learning

"Deep" neural networks refer to networks with multiple hidden layers



Mathematically, a k-layer network has the hypothesis function

$$z_{i+1} = f_i(W_i z_i + b_i), \quad i = 1, \ldots, k-1, \; z_1 = x$$
$$h_\theta(x) = z_k$$

where z_i terms now indicate *vectors*, not entries into a vector

Why use deep networks?

Motivation from circuits: many functions can be represented more compactly using deep networks than one-hidden layer networks (e.g. parity function would require (2^n) hidden units in 3-layer network, $O(n)$ units in $O(\log n)$-layer network)

Motivation from neurobiology: brain appears to use multiple levels of interconnected neurons to process information (but careful, neurons in brain are not just non-linear functions)

In practice: works better for many domains

Outline

Introduction

Machine learning with neural networks

Training neural networks

Convolutional neural networks

Recurrent neural networks

Deep reinforcement learning

Optimizing neural network parameters

How do we optimize the parameters for the machine learning loss minimization problem with a neural network

$$\underset{\theta}{\text{minimize}} \quad \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}), y^{(i)})$$

now that this problem is non-convex?

Just do exactly what we did before: initialize with random weights and run stochastic gradient descent

Now have the possibility of local optima, and function can be harder to optimize, but we won't worry about all that because the resulting models still often perform better than linear models

Stochastic gradient descent for neural networks

Recall that stochastic gradient descent computes gradients with respect to loss on each example, updating parameters as it goes

function SGD($\{(x^{(i)}, y^{(i)})\}, h_\theta, \ell, \alpha$)
  Initialize: $W_j, b_j \leftarrow$ Random, $j = 1, \ldots, k$
  Repeat until convergence:
    For $i = 1, \ldots, m$:
      Compute $\nabla_{W_j, b_j} \ell(h_\theta(x^{(i)}), y^{(i)}),\ j = 1, \ldots, k-1$
      Take gradient steps in all directions:

$$W_j \leftarrow W_j - \alpha \nabla_{W_j} \ell(h_\theta(x^{(i)}), y^{(i)}), \quad j = 1, \ldots, k$$

$$b_j \leftarrow b_j - \alpha \nabla_{b_j} \ell(h_\theta(x^{(i)}), y^{(i)}), \quad j = 1, \ldots, k$$

  return $\{W_j, b_j\}$

So how do we compute the gradients $\nabla_{W_j, b_j} \ell(h_\theta(x^{(i)}), y^{(i)})$, this is a complex function of the parameters

Backpropagation

Backpropagation is a method for computing all the necessary gradients using one “forward pass” (just computing all the values at layers), and one “backward pass” (computing gradients backwards in the network)

The equations sometimes look complex, but it’s just an application of the chain rule of calculus

The Jacobian

One (last!) bit of multivariate calculus will help us derive the backpropagation algorithm using purely matrix and vector operations

For a multivariate, vector-valued function $f : \mathbb{R}^n \to \mathbb{R}^m$, the *Jacobian* is a $m \times n$ matrix

$$\left(\frac{\partial f(x)}{\partial x}\right) \in \mathbb{R}^{m \times n} = \begin{bmatrix} \frac{\partial f_1(x)}{\partial x_1} & \frac{\partial f_1(x)}{\partial x_2} & \cdots & \frac{\partial f_1(x)}{\partial x_n} \\ \frac{\partial f_2(x)}{\partial x_1} & \frac{\partial f_2(x)}{\partial x_2} & \cdots & \frac{\partial f_2(x)}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m(x)}{\partial x_1} & \frac{\partial f_m(x)}{\partial x_2} & \cdots & \frac{\partial f_m(x)}{\partial x_n} \end{bmatrix}$$

For a scalar-valued function $f : \mathbb{R}^n \to \mathbb{R}$, the Jacobian is the transpose of the gradient $\frac{\partial f(x)}{\partial x}^T = \nabla_x f(x)$

We'll use a few simple properties of the Jacobian to derive the backpropagation algorithm for neural networks

Chain rule

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \frac{\partial g(x)}{\partial x}$$

Jacobian of a linear transformation, for $A \in \mathbb{R}^{m \times n}$

$$\frac{\partial Ax}{\partial x} = A$$

If f is a function applied elementwise,

$$\frac{\partial f(x)}{\partial x} = \mathrm{diag}(f'(x))$$

Derivation of backpropagation

Using the chain rule to compute derivatives:

$$\begin{aligned}\frac{\partial \ell(h_\theta(x), y)}{\partial b_1} &= \frac{\partial \ell(z_k, y)}{\partial b_1} \\ &= \frac{\partial \ell(z_k, y)}{\partial z_k} \frac{\partial z_k}{\partial b_1} \\ &= \frac{\partial \ell(z_k, y)}{\partial z_k} \frac{\partial z_k}{\partial z_{k-1}} \frac{\partial z_{k-1}}{\partial z_{k-2}} \cdots \frac{\partial z_3}{\partial z_2} \frac{\partial z_2}{\partial b_1}\end{aligned}$$

Furthermore, for any i

$$\frac{\partial z_{i+1}}{\partial z_i} = \frac{\partial f_i(W_i z_i + b_i)}{\partial W_i z_i + b_i} \frac{\partial W_i z_i + b_i}{\partial z_i} = \operatorname{diag}(f_i'(W_i z_i + b_i)) W_i$$

and

$$\frac{\partial z_{i+1}}{\partial b_i} = \frac{\partial f_i(W_i z_i + b_i)}{\partial W_i z_i + b_i} \frac{\partial W_i z_i + b_i}{\partial b_i} = \operatorname{diag}(f_i'(W_i z_i + b_i))$$

If we compute derivatives with respect to *all* b_i, and W_i just using this formula, we'd be repeating a lot of work (e.g. all the $\frac{\partial z_{i+1}}{\partial z_i}$ terms that appear in multiple derivatives)

Backpropagation caches these intermediate products, specifically defining

$$g_i^T = \frac{\partial \ell(z_k, y)}{\partial z_k} \frac{\partial z_k}{\partial z_{k-1}} \cdots \frac{\partial z_{i+1}}{\partial z_i}$$

which can be computed recursively via the relationship

$$g_k = \frac{\partial \ell(z_k, y)}{\partial z_k}$$

$$g_i = W_i^T (g_{i+1} \circ f'(W_i z_i + b_i))$$

where $\circ$ denotes elementwise multiplication of vectors

Gradients can then be computed via

$$\nabla_{b_i} \ell(h_\theta(x), y) = g_{i+1} \circ f'(W_i z_i + b_i)$$

$$\nabla_{W_i} \ell(h_\theta(x), y) = (g_{i+1} \circ f'(W_i z_i + b_i)) z_i^T$$

As mentioned, algorithmically backpropagation takes the form of one forward pass (to compute z_i terms) and one backward pass (to compute g_i terms) through the network

function Backpropagation$(x, y, \{ W_i, b_i, f_i \}_{i=1}^{k-1}, \ell)$

Initialize: $z_1 \leftarrow x$

For $i = 1, \ldots, k-1$

$z_{i+1}, z'_{i+1} \leftarrow f_i(W_i z_i + b_i),\ f'_i(W_i z_i + b_i)$

$L \leftarrow \ell(z_k, y)$

$g_k \leftarrow \frac{\partial \ell(z_k, y)}{\partial z_k}$

For $i = k-1, \ldots, 1$:

$g_i = W_i^T (g_{i+1} \circ z'_{i+1})$

$\nabla_{b_i} \leftarrow g_{i+1} \circ z'_{i+1}$

$\nabla_{W_i} \leftarrow (g_{i+1} \circ z'_{i+1}) z_i^T$

return $L, \{\nabla_{b_i}, \nabla_{W_i}\}_{i=1}^{k-1}$

Gradients can still get somewhat tedious to derive by hand, especially for the more complex models that follow

Fortunately, a lot of this work has already been done for you

Tools like Theano (http://deeplearning.net/software/theano/), Torch (http://torch.ch/), TensorFlow (http://www.tensorflow.org/) all let you specify the network structure and then automatically compute all gradients (and use GPUs to do so)

Autograd package for Python (https://github.com/HIPS/autograd) lets you compute the derivative of (almost) any arbitrary function using numpy operations using automatic backpropagation

What's changed since the 80s?

All these algorithms (and most of the extensions in later slides), were developed in the 80s or 90s

So why are these just becoming more popular in the last few years?

- More data
- Faster computers
- (Some) better optimization techniques

Unsupervised pre-training (Hinton et al., 2006): "Pre-train" the newtork have the hidden layers recreate their input, one layer at a time, in an unsupervised fashion

- This paper was partly responsible for re-igniting the interest in deep neural networks, but the general feeling now is that it doesn't help much

Dropout (Hinton et al., 2012): During training and computation of gradients, randomly set about half the hidden units to zero (a different randomly selected set for each stochastic gradient step)

- Acts like regularization, prevents the parameters for overfitting to particular examples

Different non-linear functions (Nair and Hinton, 2010): Use non-linearity $f(x) = \max\{0, x\}$ instead of $f(x) = \tanh(x)$