15-780 – Graduate Artificial Intelligence: Deep learning

J. Zico Kolter (this lecture) and Ariel Procaccia
Carnegie Mellon University
Spring 2017

Outline

Deep learning history

Machine learning with neural networks

Training neural networks

Backpropagation

Complex architectures and computation graphs

Midterm

Next Monday, 2/27 during class

4-5 short answer questions (~15-20 minutes each)

Questions will involve short proofs, derivations, or problem formulations

Will be completely closed-book

Practice midterm to be released today or tomorrow

Outline

Deep learning history

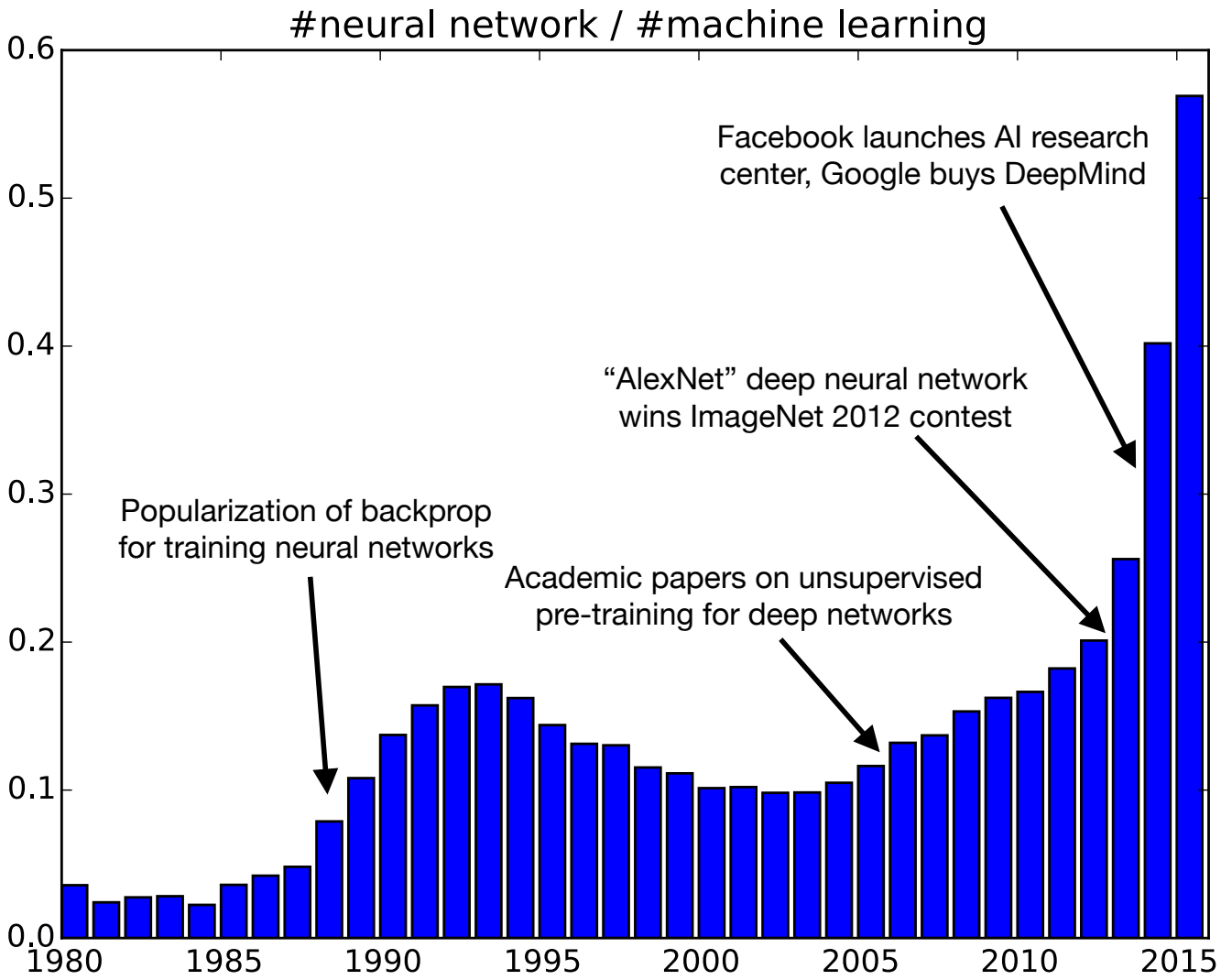
Machine learning with neural networks

Training neural networks

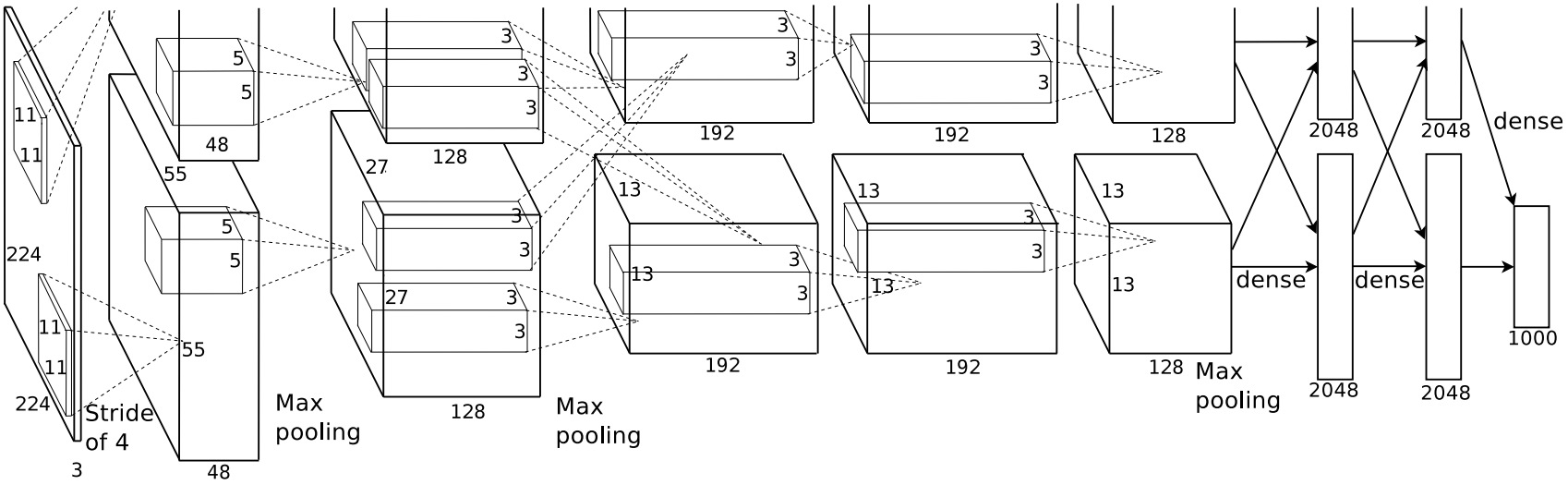
Backpropagation

Complex architectures and computation graphs

Recent history in machine learning

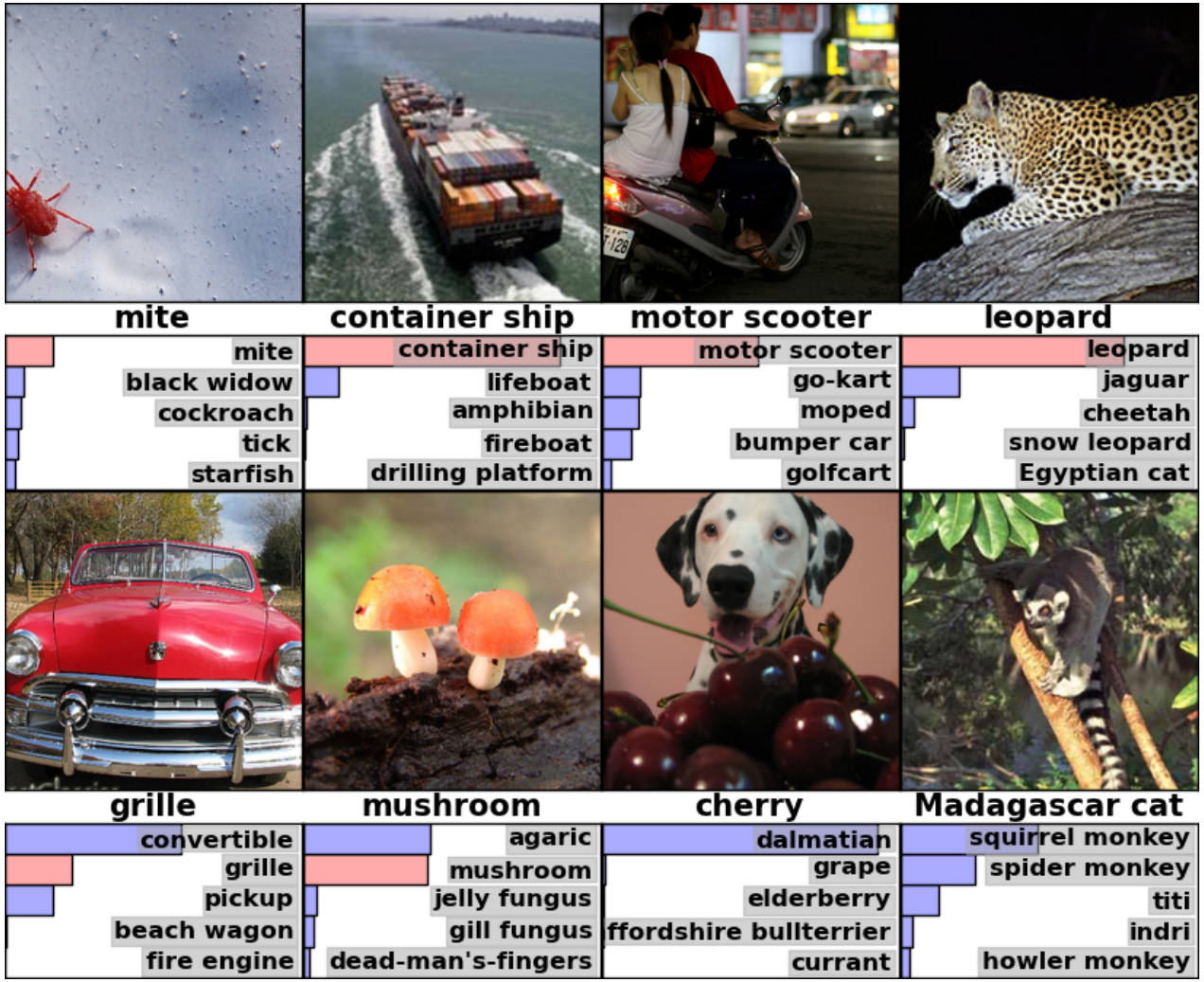


AlexNet



"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 got 26.1% error)

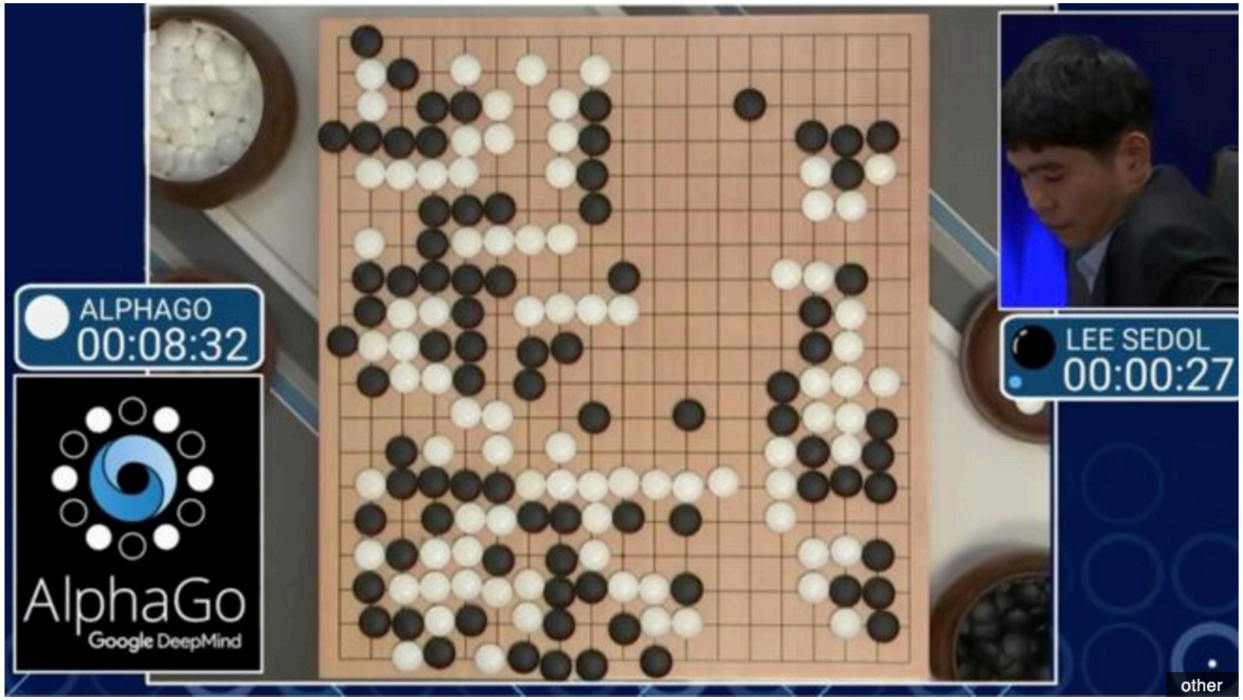
ImageNet classification



AlphaGo

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

12 March 2016 | Technology



Why now?

Why is this all happening right now?

To answer this, let's first define precisely what neural networks are in the context of machine learning

Outline

Deep learning history

Machine learning with neural networks

Training neural networks

Backpropagation

Complex architectures and computation graphs

Neural networks for machine learning

Remember that machine learning algorithms all have three components

1. Hypothesis class – the set of functions we consider
2. Loss function – measurement of how good a hypothesis is
3. Optimization – how we find a hypothesis function with low loss

The term *neural network* refers to the *first* element here: is it describing the class of hypothesis functions used within a machine learning algorithm

Specifically, neural networks refer to hypotheses consisting of a particular form of **composed non-linear functions**

Any loss function and optimization approach could be used, though some are much more common than others

Linear hypotheses and feature learning

Until now, we have (mostly) considered machine learning algorithms that linear hypothesis class

$$h_\theta(x) = \theta^T x$$

where x denotes some set of typically non-linear features (e.g., polynomials)

The performance of these algorithms depends crucially on coming up with good features

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

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 and another produces the final hypothesis

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

where

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

By convention, we're going to separate out the "constant feature" into the b terms

Poll: composed linear hypotheses

Given $x \in \mathbb{R}^n$, suppose I run two machine learning algorithms with the hypothesis functions $h, \tilde{h} : \mathbb{R}^n \to \mathbb{R}$:

$$h(x) = W_2(W_1 x + b_1) + b_2 \qquad \tilde{h}(x) = W_3 x + b_3$$
$$W_1 \in \mathbb{R}^{k \times n}, b_1 \in \mathbb{R}^k, W_2 \in \mathbb{R}^{1 \times k}, b_2 \in \mathbb{R} \qquad W_3 \in \mathbb{R}^{1 \times n}, b_2 \in \mathbb{R}$$

Suppose we use the same data, minimize the same loss function, and are (somehow) able to achieve the *global* optima of both problems (assumed to be unique), which of the following will be true:

1. h achieves lower training loss but higher validation loss than $\tilde{h}$
2. h achieves lower training loss *and* lower validation loss than $\tilde{h}$
3. $\tilde{h}$ achieves lower training loss *and* lower validation loss than h
4. They will both perform identically
5. The performance depends on the data or choice of loss function

Neural networks

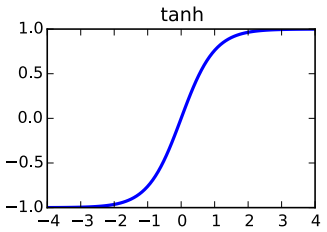
Neural networks are a simple extension of this idea, where we additionally apply a non-linear function after each linear transformation

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

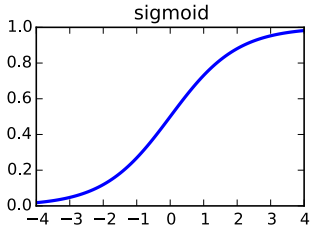
where $f_1, f_2 : \mathbb{R} \to \mathbb{R}$ are a non-linear functions (applied elementwise)

Common choices of f_i:

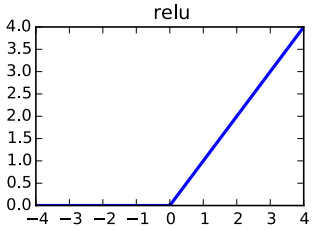
Hyperbolic tangent: $f(x) = \tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$



Sigmoid: $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$

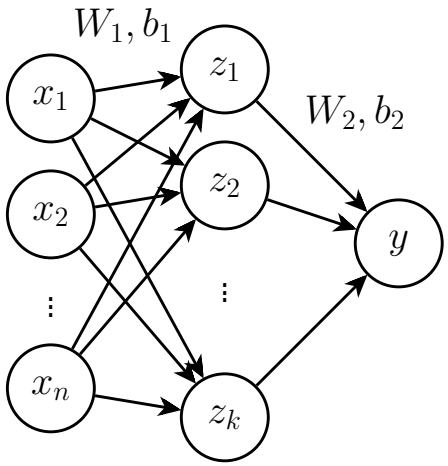


Rectified linear unit (ReLU): $f(x) = \max\{x, 0\}$



Illustrating neural networks

We draw neural networks using the same graphic as before (the non-linear function are always in 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 prescribed what values they should take, left up to algorithm to decide

Properties of neural networks

A neural network will a single hidden layers (and enough hidden units) is a *universal function approximator*, can approximate *any* function over inputs

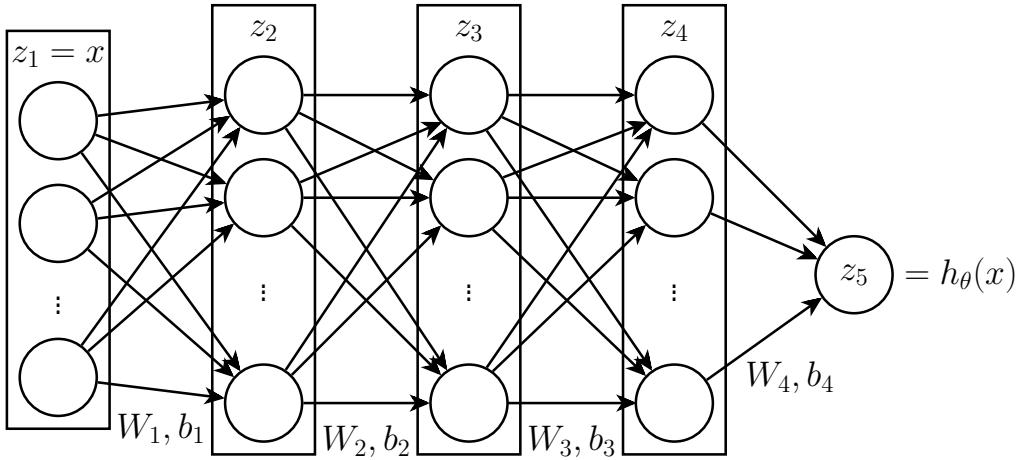
In practice, not *that* relevant (similar to how polynomials can fit any function), and the more important aspect is that they appear to work very well in practice for many domains

The hypothesis $h_\theta(x)$ is not a convex function of parameters $\theta = \{W_i, b_i\}$, so we have possibility of local optima

Architectural choices (how many layers, how they are connected, etc), become important algorithmic design choices (i.e. hyperparameters)

Deep learning

"Deep learning" refers (almost always) to machine learning using neural network models with multiple hidden layers



Hypothesis function for k-layer network

$$z_{i+1} = f_i(W_i z_i + b_i), \qquad z_1 = x, \qquad h_\theta(x) = z_k$$

(note the z_i here refers to a vector, not an entry into vector)

Why use deep networks

Motivation from circuit theory: many functions can be represented more efficiently using deep networks (e.g., parity function requires $O(2^n)$ hidden units with single hidden layer, $O(n)$ with $O(\log n)$ layers

- But not clear if deep learning really learns these types of network

Motivation from biology: brain appears to use multiple levels of interconnected neurons

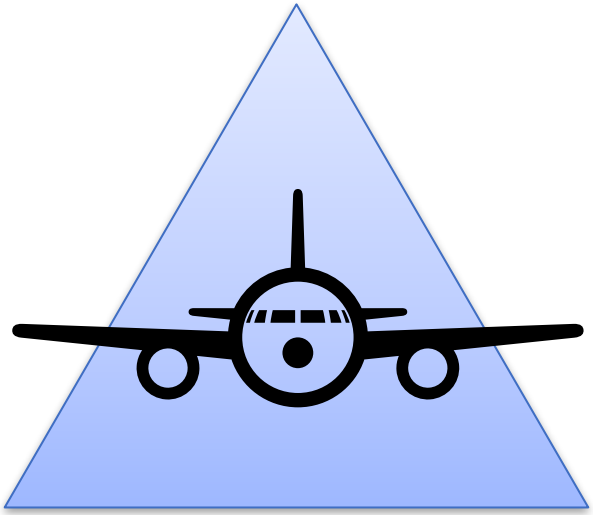
- But despite the name, the connection between neural networks and biology is extremely weak

In practice: works much better for many domains

- Hard to argue with results

Why now?

High capacity models
(i.e., large VC
dimension)



Lots of
computing power

Lots of data

Outline

Deep learning history

Machine learning with neural networks

Training neural networks

Backpropagation

Complex architectures and computation graphs

Neural networks for machine learning

How do we solve the optimization problem

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

Not a convex problem, so we don't expect to find global optimum, but we will instead be content with *local* solutions

Just use gradient descent as normal (or rather, a version called stochastic gradient descent)

Stochastic gradient descent

Key challenge for neural networks: often have very large number of samples, computing gradients can be computationally intensive.

Traditional gradient descent computes the gradient with respect to the sum over *all examples,* then adjusts the parameters in this direction

$$\theta \leftarrow \theta - \alpha \nabla_\theta \sum_{i=1}^{m} \ell(h_\theta(x^{(i)}, y^{(i)}) = \theta - \alpha \sum_{i=1}^{m} \nabla_\theta \ell(h_\theta(x^{(i)}, y^{(i)})$$

Alternative approach, *stochastic gradient descent* (SGD): adjust parameters based upon just *one* sample

$$\theta \leftarrow \theta - \alpha \nabla_\theta \ell(h_\theta(x^{(i)}), y^{(i)})$$

and then repeat these updates for all samples

Gradient descent vs. SGD

Gradient descent, repeat:

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

$$g^{(i)} \leftarrow \nabla_\theta \ell\big(h_\theta\big(x^{(i)}\big), y^{(i)}\big)$$

- Update parameters:

$$\theta \leftarrow \theta - \alpha \sum_{i=1}^{m} g^{(i)}$$

Stochastic gradient descent, repeat:

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

$$\theta \leftarrow \theta - \alpha \nabla_\theta \ell\big(h_\theta\big(x^{(i)}\big), y^{(i)}\big)$$

In practice, stochastic gradient descent uses a small collection of samples, not just one, called a *minibatch*

Outline

Deep learning history

Machine learning with neural networks

Training neural networks

Backpropagation

Complex architectures and computation graphs

Computing gradients: backpropagation

So, how do we compute the gradient $\nabla_\theta \ell(h_\theta(x^{(i)}), y^{(i)})$?

Remember θ here denotes a *set* of parameters, so this really means that we compute gradient with respect to all parameters $W_1, b_1, W_2, b_2, \ldots$

The ***backpropagation algorithm*** is an algorithm for computing *all* these gradients simultaneously, using one "forward pass" and one "backward pass" through the network

The equations look complex, but it is just an application of the (multivariate) chain rule of calculus

Digression: the Jacobian

Because I know that when we talked about the gradient, everyone really just wanted more matrix calculus…

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

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

For $f: \mathbb{R}^n \rightarrow \mathbb{R}$, $\nabla_x f(x) = \left(\frac{\partial f(x)}{\partial x}\right)^T$

Properties of Jacobian

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

1. Chain rule, for $f: \mathbb{R}^n \to \mathbb{R}^m$, $g: \mathbb{R}^k \to \mathbb{R}^n$

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

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

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

3. For a function $f(x)$ applied elementwise to a vector

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

Backpropagation

Let's consider the loss on a single example x, y, and use the chain rule to compute the Jacobian

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

where all the z_i terms are really functions of b_1, but we leave out this dependence for notational simplicity

Backpropagation, continued

We can also use the chain rule to compute intermediate terms, e.g.,

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

and

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

If we carried out the same computation for each parameter, e.g. b_i, we would repeat a lot of work; the backpropagation algorithm just "caches" certain intermediate products

Backpropagation, continued

Specific, let's consider the following term,

$$g_i^T = \frac{\partial \ell(z_k, y)}{\partial z_k} \frac{\partial z_k}{\partial z_{k-1}} \frac{\partial z_{k-1}}{\partial z_{k-2}} \dots \frac{\partial z_{i+1}}{\partial z_i}$$

Then we have the following recursive definition of g_i

$$g_k = \left(\frac{\partial \ell(z_k, y)}{\partial z_k}\right)^{\mathrm{T}} = \nabla_{z_k} \ell(z_k, y)$$

$$g_i = W_i^T \mathrm{diag}\big(f_i'(W_i z_i + b_i)\big) g_{i+1} = W_i^T \Big(g_{i+1} \circ f_i'(W_i z_i + b_i)\Big)$$

Where $\circ$ denotes elementwise vector multiplication

Backpropagation, continued

Finally, assuming we compute all the $g_k, \dots, g_1$ terms, then the gradients with respect to each W_i and b_i term can be computed as

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

Backpropagation algorithm

1. Forward pass: compute $z_1, \dots, z_k, \ell(z_k, y)$
2. Backward pass, compute $g_k, \dots, g_1$
3. Return gradients $\nabla_{b_i} \ell(h_\theta(x), y)$, $\nabla_{W_i} \ell(h_\theta(x), y)$ for all i

Poll: complexity of backpropagation

Consider the case where all z_i terms are n dimensional: what is the complexity of computing only a single g_i term using backpropagation

$$g_i^T = \frac{\partial \ell}{\partial z_k} \frac{\partial z_k}{\partial z_{k-1}} \frac{\partial z_{k-1}}{\partial z_{k-2}} \cdots \frac{\partial z_{i+1}}{\partial z_i}$$

(the complexity of multiplying two $n \times n$ matrices is $O(n^3)$ and complexity of multiplying a n-dimensional vector with an $n \times n$ matrix is $O(n^2)$)

1. $O(n^3)$
2. $O(n^2)$
3. $O\Big(n^3(k-i)\Big)$
4. $O\Big(n^2(k-i)\Big)$

Aside: gradients/Jacobians w.r.t. matrices

This final bit is only for those who are particularly curious, but there's one element that is not quite precise in the formulation above

For matrix-input function $f: \mathbb{R}^{m \times n} \rightarrow \mathbb{R}$, $\nabla_X f(X) \in \mathbb{R}^{m \times n}$ (a matrix)

But what about for $f: \mathbb{R}^{m \times n} \rightarrow \mathbb{R}^k$, what is $\frac{\partial f(X)}{\partial X}$? (we've run out of indices)

We actually had one of these in our previous setting, that I just hid, $\frac{\partial z_{i+1}}{\partial W_i}$

You can do this with tensor operations, but a slightly easier (maybe?) approach is to use *vectorization*

Aside: vectorization

Define $\mathrm{vec} : \mathbb{R}^{m \times n} \to \mathbb{R}^{mn}$ to be the vectorization operator, the operator that forms a vector from a matrix by stacking its columns together

Fact: for A, B, C such that we can form the product ABC

$$\mathrm{vec}(ABC) = (C^T \otimes A)\mathrm{vec}(B)$$

where $\otimes$ is the *Kronecker product*, for $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{p \times q}$

$$A \otimes B \in \mathbb{R}^{mp \times nq} = \begin{bmatrix} A_{11}B & \cdots & A_{1n}B \\ \vdots & \ddots & \ddots \\ A_{m1}B & \cdots & A_{mn}B \end{bmatrix}$$

We'll call the inverse operator $\mathrm{mat}: \mathbb{R}^{mn} \to \mathbb{R}^{m \times n}$

Vectorization for Jacobians

Let's use vectorization to compute:

$$\begin{aligned}\frac{\partial \ell}{\partial \text{vec}(W_i)} &= \frac{\partial \ell}{\partial z_{i+1}} \frac{\partial z_{i+1}}{\partial \text{vec}(W_i)} \\ &= g_{i+1}^T \frac{\partial f_i(W_i z_i + b_i)}{\partial W_i z_i + b} \frac{\partial W_i z_i + b_i}{\partial \text{vec}(W_i)} \\ &= g_{i+1}^T \text{diag}(f_i'(W_i z_i + b_i))(z_i^T \otimes I)\end{aligned}$$

since by vectorization we know that

$$W_i z_i = \text{vec}(W_i z_i) = (z_i^T \otimes I)\text{vec}(W_i)$$

Vectorization for Jacobians, continued

Thus, we can compute the gradient of our loss with respect to W_i

$$\begin{aligned}\nabla_{W_i}\ell(h_\theta(x),y) &= \mathrm{mat}\left(\frac{\partial \ell}{\partial \mathrm{vec} W_i}^T\right)\\ &= \mathrm{mat}\left((z_i \otimes I)\mathrm{diag}\big(f_i'(W_i z_i + b_i)\big) g_{i+1}\right)\\ &= (f'(W_i z_i + b_i) \circ g_{i+1}) z_i^T\end{aligned}$$

To be absolutely clear, you aren't expected to follow all that, but it might be a useful reference if you try to figure out how certain gradients come up in backpropagation

Outline

Deep learning history

Machine learning with neural networks

Training neural networks

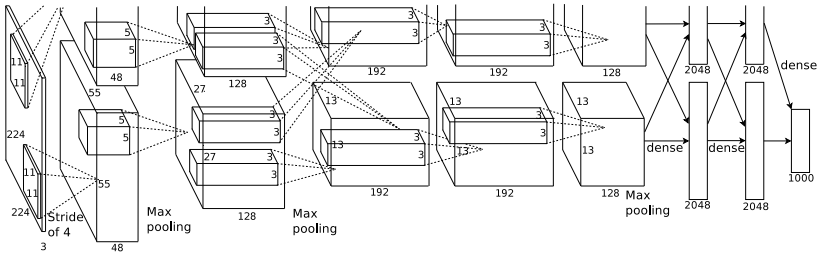
Backpropagation

Complex architectures and computation graphs

Modern deep learning architectures

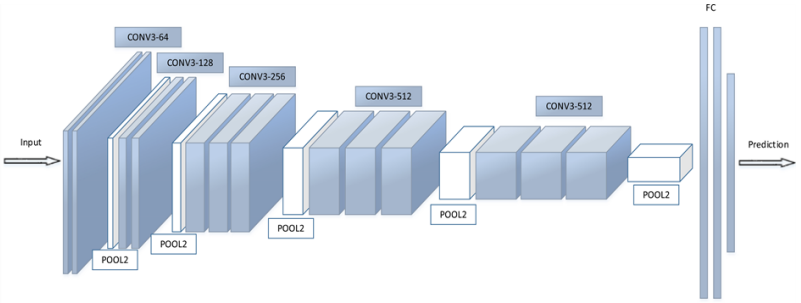
9699 citations

AlexNet (Krizhevsky et al., 2012)



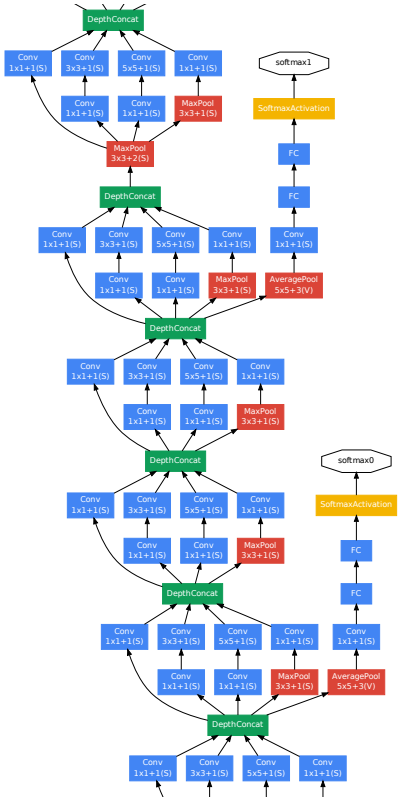
3351 citations

VGG (Simonyan and Zisserman, 2012)



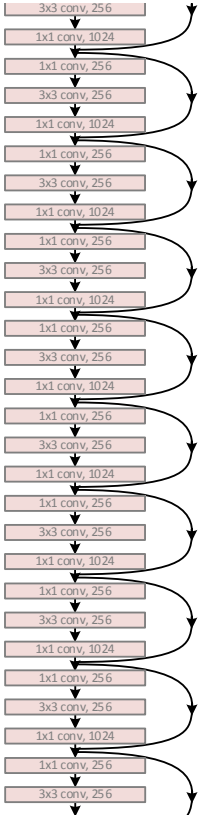
2525 citations

GoogLeNet (Szegedy et al., 2015)



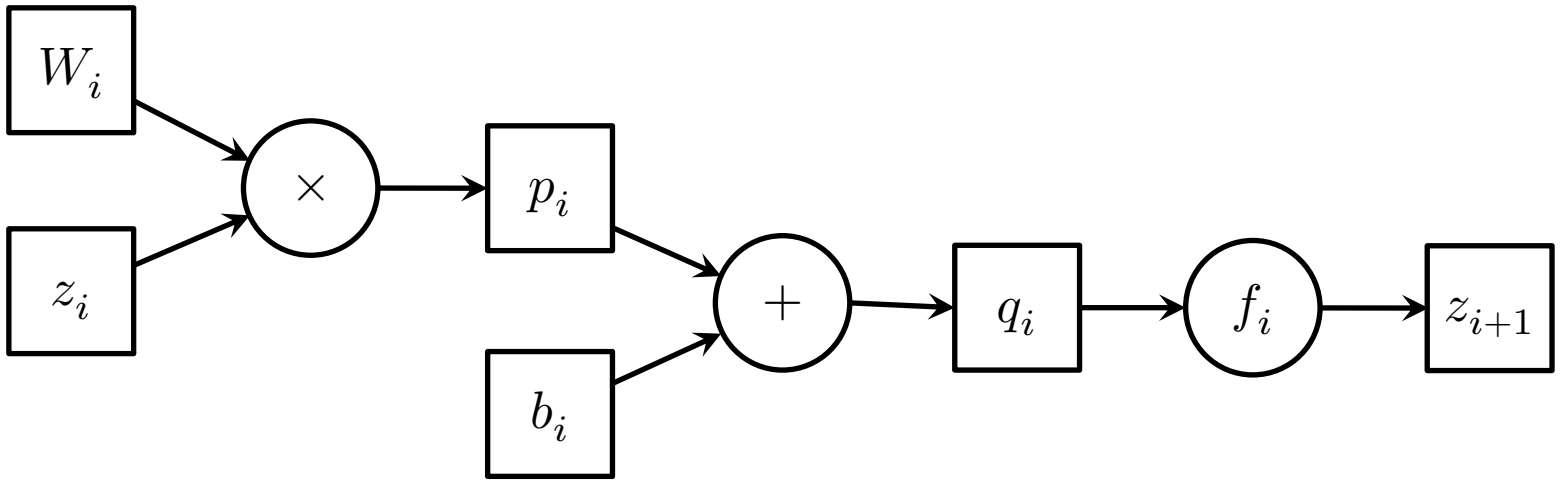
1295 citations

ResNet (He et al., 2015)



Computation graphs

To simplify backpropagation in complex networks, we can make use of a data structure called a computation graph, a directed acyclic graph (DAG) over variables (square nodes) and operators (circle nodes)



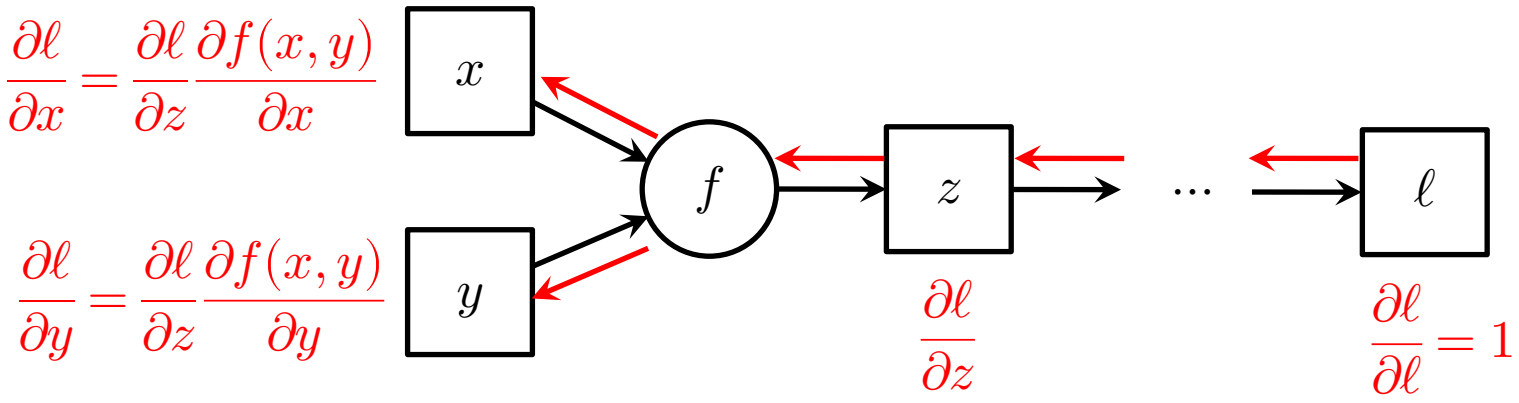
Common to omit intermediate variables that have no real significance in computation (p_i, q_i)

Forward and backward passes

In forward pass, simply compute each node in the graph after all it's parents have been computed

Eventually this terminates at some *scalar* value ℓ (our final function)

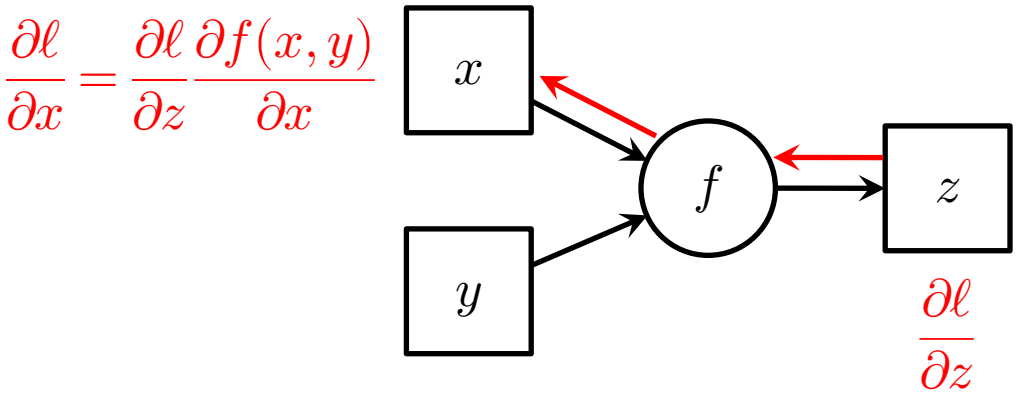
Backward pass computes Jacobians with respect to ℓ via the same graph where edge direction is reversed



Backward pass

To compute the backward pass, the implementation of our function f needs to be able to compute:

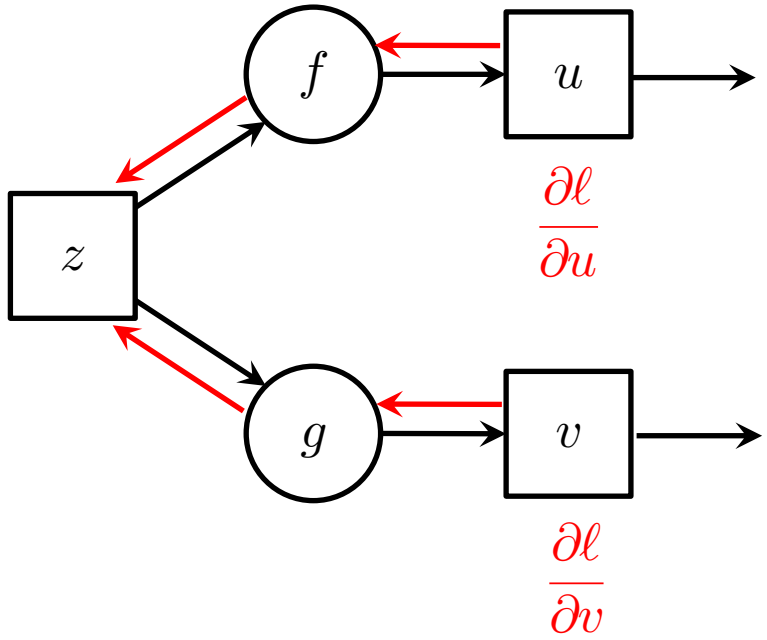
1. The forward evaluation of the function $z = f(x, y)$ (obviously)
2. The *product* $\frac{\partial \ell}{\partial z}\frac{\partial f(x,y)}{\partial x}$ for both its inputs x, y



Variables passed to multiple functions

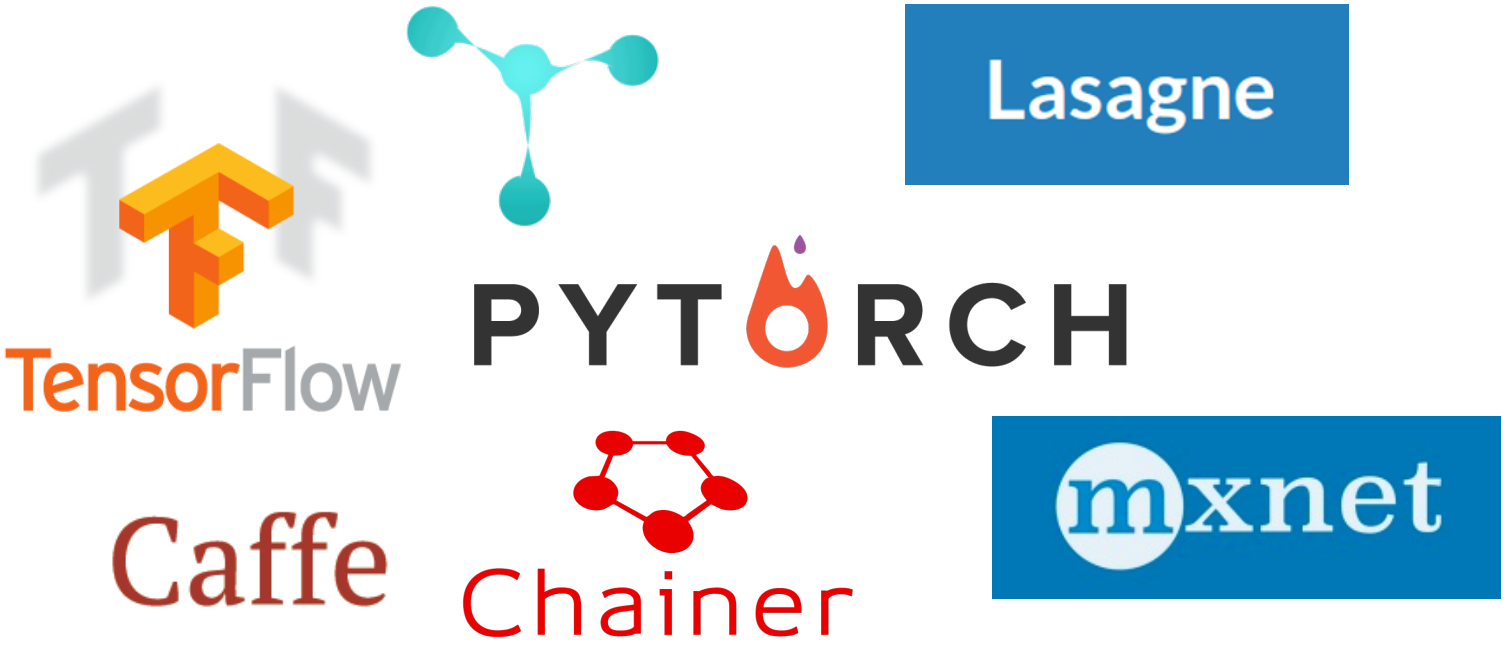
If we pass a variable z to *multiple* functions, then we simply add all the incoming Jacobians

$$\frac{\partial \ell}{\partial z} = \frac{\partial \ell}{\partial u}\frac{\partial f(z)}{\partial z} + \frac{\partial \ell}{\partial v}\frac{\partial g(z)}{\partial z}$$



Deep learning frameworks

Every modern deep learning framework uses this method for computing gradients: they (either explicitly or implicitly) *preserve* the computation graph during the forward pass, and then use the reversed graph for the backward pass



Deep learning frameworks

The frameworks implement the forward and backward elements of many common functions, so you can only specify the forward computation and get the backward pass “for free” (coding wise)

```
# NOTE: this is an illustration, not how you use Tensorflow in practice
import tensorflow as tf
x = tf.Variable(tf.zeros([784,1]))
y = tf.Variable(tf.zeros([10,1]))

W1 = tf.Variable(tf.zeros([100,784]))
b1 = tf.Variable(tf.zeros([100,1]))
W2 = tf.Variable(tf.zeros([10,100]))
b2 = tf.Variable(tf.zeros([10,1]))

z1 = tf.nn.relu(tf.matmul(W1,x) + b1)
z2 = tf.matmul(W2,z1) + b2
l = tf.nn.softmax_cross_entropy_with_logits(logits=z2, labels=y)

tf.gradients(l, [W1,b1,W2,b2])
```