# Deep Linear Neural Networks

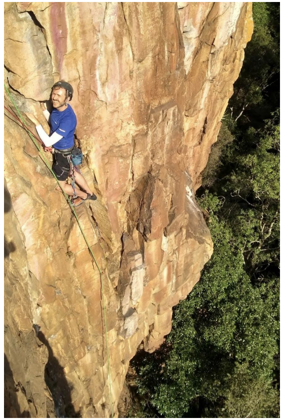
By Andrew Saxe & Saeed Salehi

neuromatch academy

# Who is Andrew?

- Interested in the theory of deep learning and applications to neuroscience and psychology.

- Avid but bad rock climber
- Avid but bad singer/guitar player
- So thrilled to be learning and studying with you



# Credits

A huge thank you to:

- **Saeed Salehi** for crafting the tutorials
- **Konrad Kording** for slides
- **Vladimir Haltakov**, **Spiros Chavlis**, **Polina Turishcheva**, **Anoop Kulkarni**, and **Khalid Almubarak** for content, comments & production

# **Welcome** to Deep Linear Networks Day

We'll use the simplest possible networks to understand:

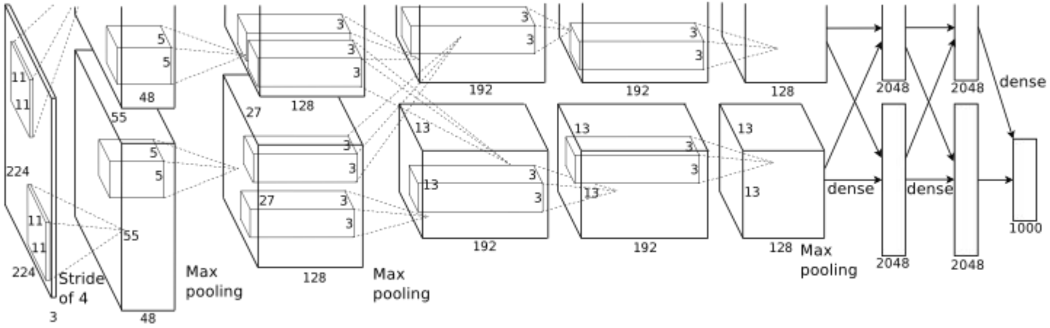
- The basics of gradient descent (Tutorial 1)
- The effect of depth on training dynamics (Tutorial 2)
- The internal representations that deep networks learn (Tutorial 3)

# Gradient Descent and AutoGrad

neuromatch academy

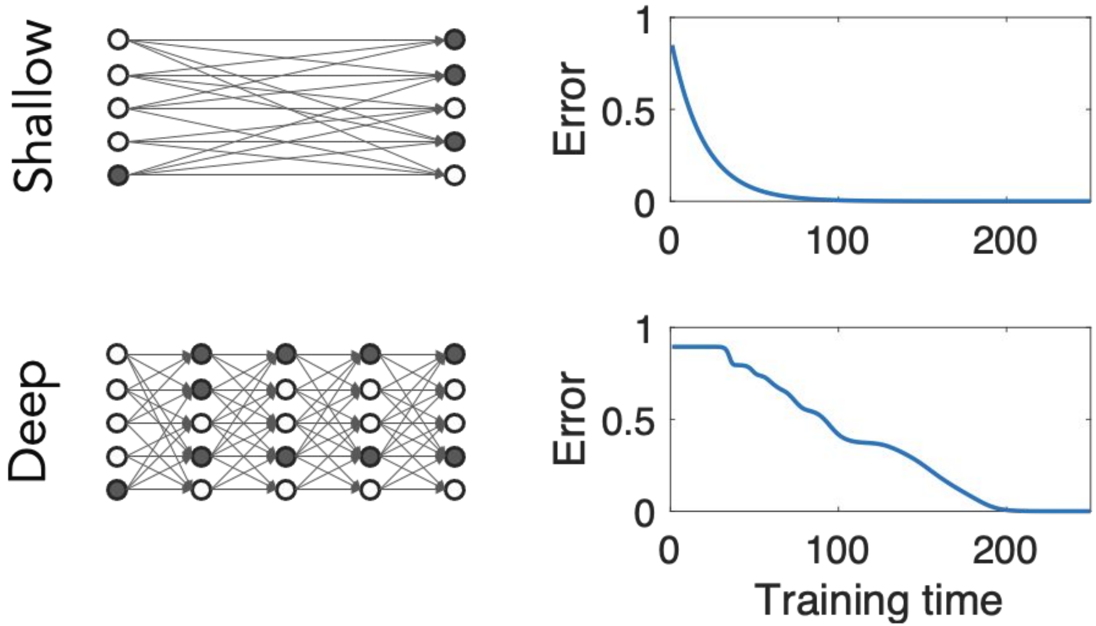
# Gradient descent

How can we change parameters to make the overall system work better?

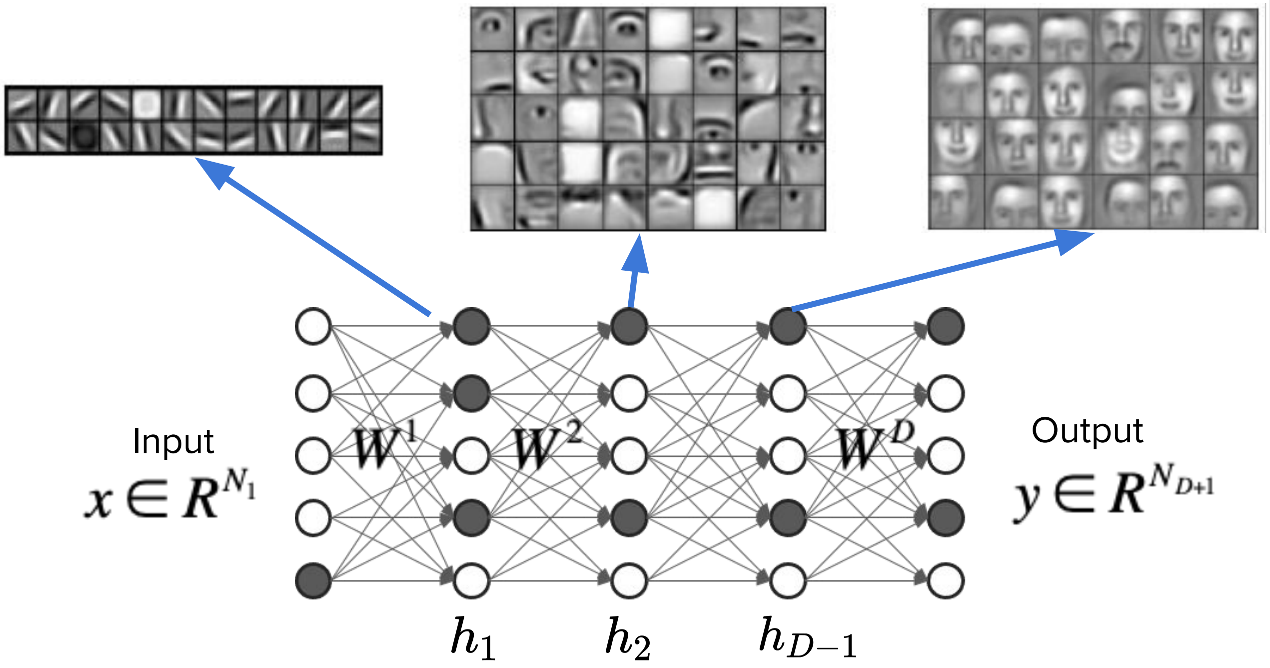



Output:
Cat
Target:
Dog

# The effect of depth



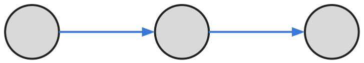
# Representation learning



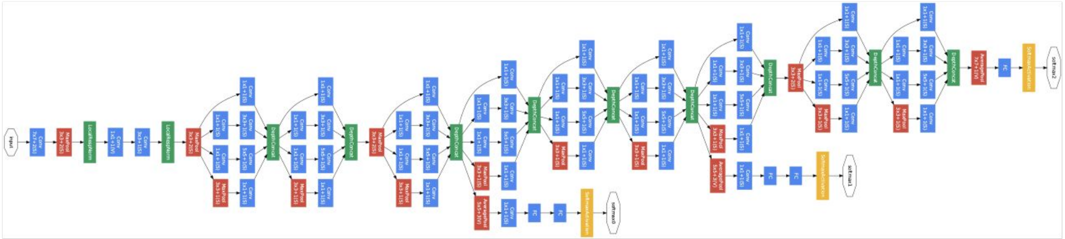

Lee et al., 2009

# Simple models

**Today**



**The rest of your career**



Szegedy et al., CVPR 2015

# Deep learning: key components

The designer specifies:

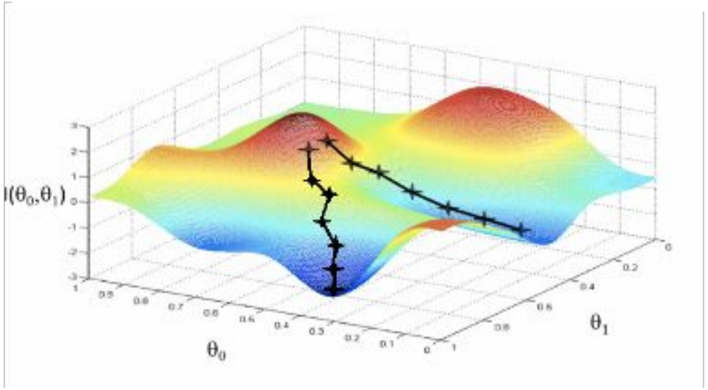
1. Objective function
2. Learning rule
3. Architecture
4. Initialisation
5. Environment

# Deep learning: key components

The designer specifies:

1. **Objective function**
2. Learning rule
3. Architecture
4. Initialisation
5. Environment

What is the goal of the computation?



# Deep learning: key components

The designer specifies:

1. Objective function
2. **Learning rule**
3. Architecture
4. Initialisation
5. Environment

How will weights change to improve the objective function?

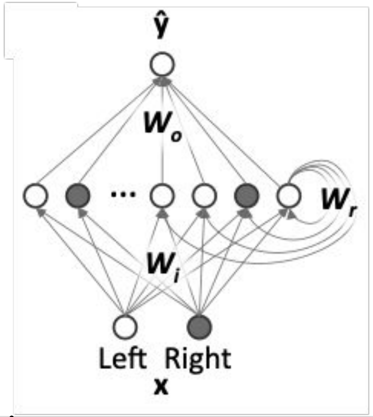
$$\Delta W = -\eta \frac{\partial L}{\partial W}$$

# Deep learning: key components

The designer specifies:

1. Objective function
2. Learning rule
3. **Architecture**
4. Initialisation
5. Environment

What are the components and connectivity?



# Deep learning: key components

The designer specifies:

1. Objective function
2. Learning rule
3. Architecture
4. **Initialisation**
5. Environment

What are the initial weight values?

$$W(0) \sim N(0, \sigma^2)$$

# Deep learning: key components

The designer specifies:

1. Objective function
2. Learning rule
3. Architecture
4. Initialisation
5. **Environment**

What is the data provided during learning?

IMAGENET

# Example

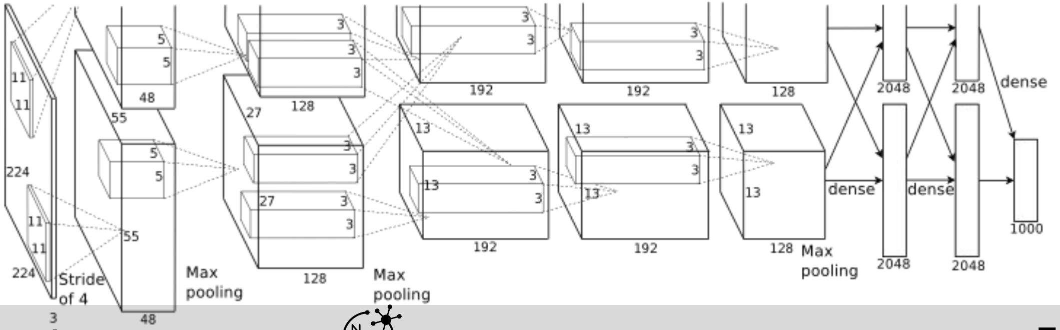
Objective function: Cross entropy loss

Learning rule: Gradient descent with momentum

Architecture: Deep convolutional ReLU network

Initialisation: He et al. (Scaled Gaussian)

Environment: ImageNet dataset




Output:
Cat
Target:
Dog

# Learning as optimization

An input-output function (an ANN): $y = f_w(x)$

A loss function: $L = \ell(y, data)$

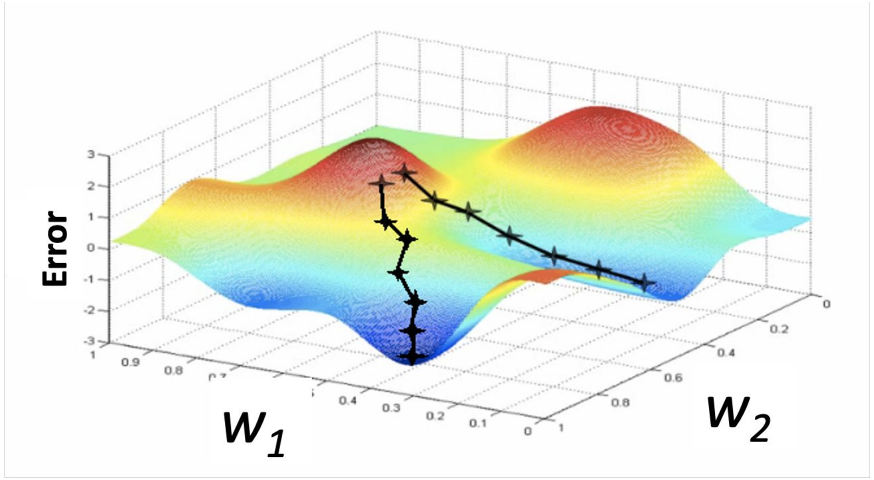
Optimization problem: $w^* = \mathrm{argmin}_w \ell(f_w(x), data)$

# The workhorse algorithm: Gradient descent

So important we will understand it at different levels:

- Conceptually
- By taking derivatives by hand (just once!)
- Through automatic differentiation in PyTorch

# Gradient descent




http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

Minimize function by taking many small steps, each pointing downhill

# Gradients

"how much would loss change if I changed a parameter just a tiny bit"

$$\nabla L(w) = \left[ \frac{\partial L}{\partial w_1} \frac{\partial L}{\partial w_2} \cdots \frac{\partial L}{\partial w_N} \right] \Bigg|_w$$

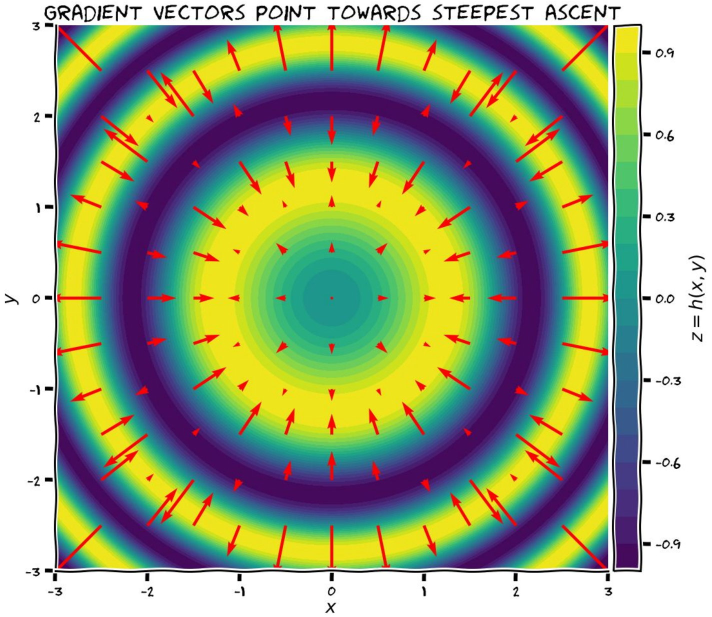
# Gradients

Why are gradients useful?

Let's start by investigating what directions the gradient points in

**Derive the gradient by hand!**

# Gradient




The gradient points in the direction of steepest ascent.

# Gradient descent

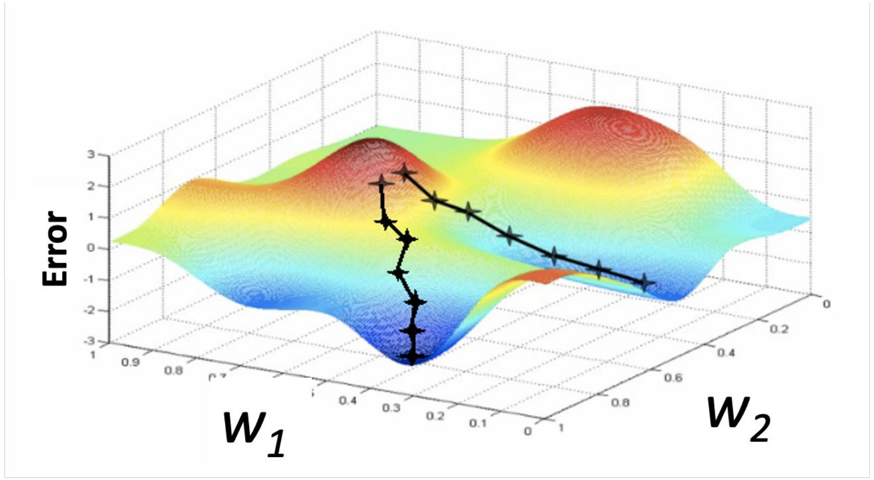
"Make small change in weights that most rapidly improves task performance"

$\Updownarrow$

Change each weight in proportion to the negative gradient of the loss

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla L(\mathbf{w}^{(t)})$$

# Gradient descent




http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png

Initialize: $\mathbf{w}^{(0)} = \mathbf{w}_0$

For $t=0$ to $T$:

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \eta \nabla L(\mathbf{w}^{(t)})$$

# Why gradient descent?

There are an infinite set of learning approaches that make us better

However, GD is the one that most rapidly reduces loss (for infinitesimal steps)

# The core computation: Calculating the gradient

Let's try a slightly more complicated example

Because it is so fundamental, you should do it at least once

Basic tools: partial derivatives; chain rule

**Derive the gradient by hand (again)!**

# Gradients via the computational graph

Deriving gradients ad hoc is hard and it's easy to make mistakes

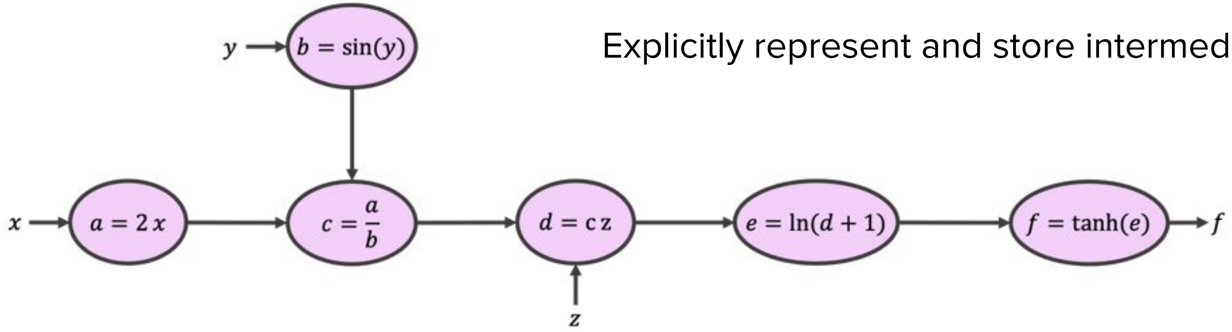
How can we simplify and systematize our approach?

# Computational Graph (forward)

$$f(x, y, z) = \tanh\left(\ln\left[1 + z\frac{2x}{sin(y)}\right]\right)$$

# Computational Graph (forward)

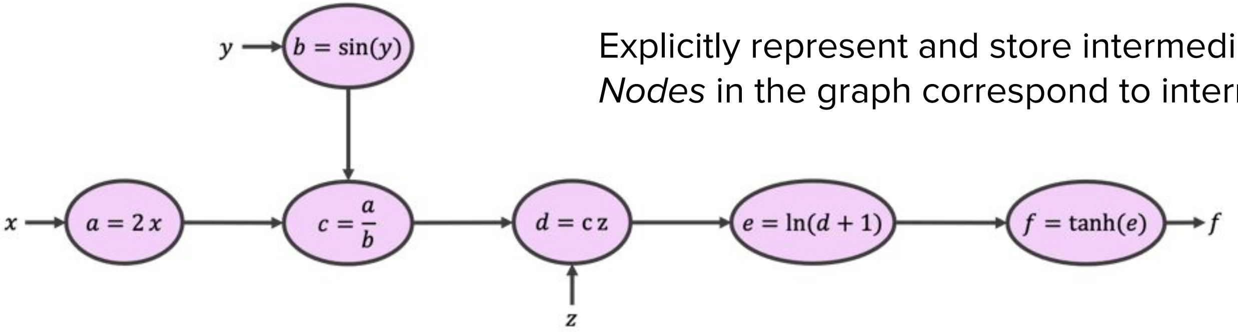
$$f(x,y,z) = \tanh\left(\ln\left[1 + z\frac{2x}{sin(y)}\right]\right)$$




Explicitly represent and store intermediate variables *a,b,c,d,e*

# Computational Graph (forward)

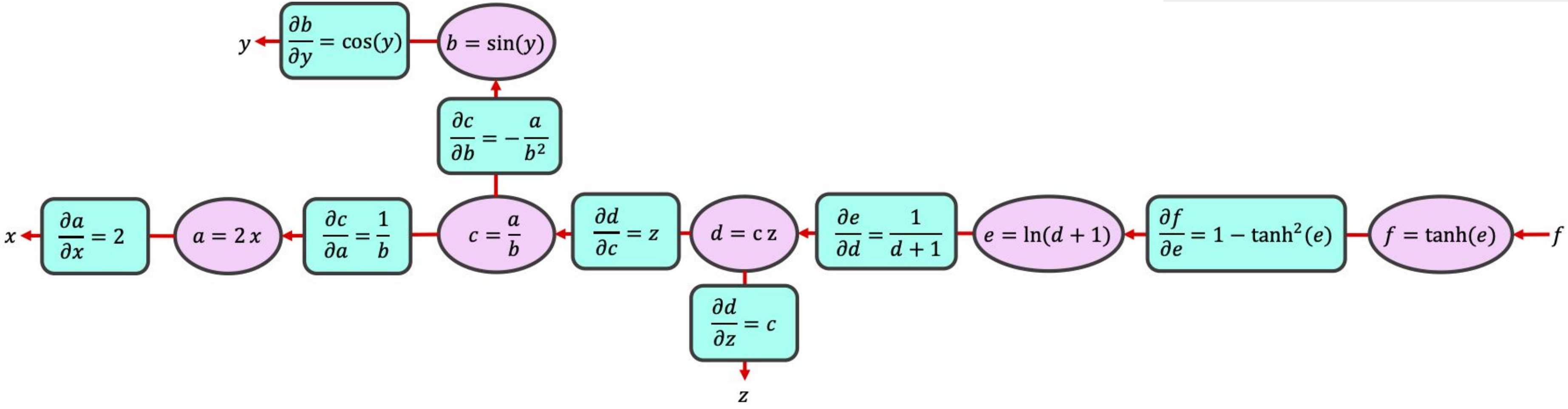
$$f(x,y,z) = \tanh\left(\ln\left[1 + z\frac{2x}{sin(y)}\right]\right)$$




Explicitly represent and store intermediate variables *a,b,c,d,e*.
*Nodes* in the graph correspond to intermediate variables.

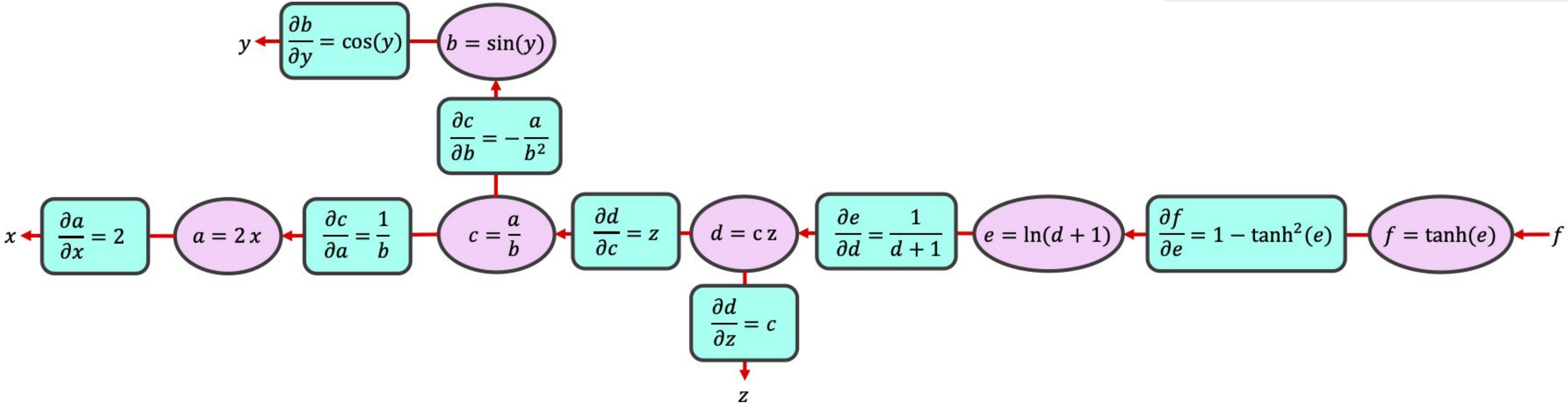
# Computational Graph (backward)

Starting from the top, pass backward. Each *edge* stores partial derivative of the head of the edge with respect to the tail.



# Computational Graph (backward)

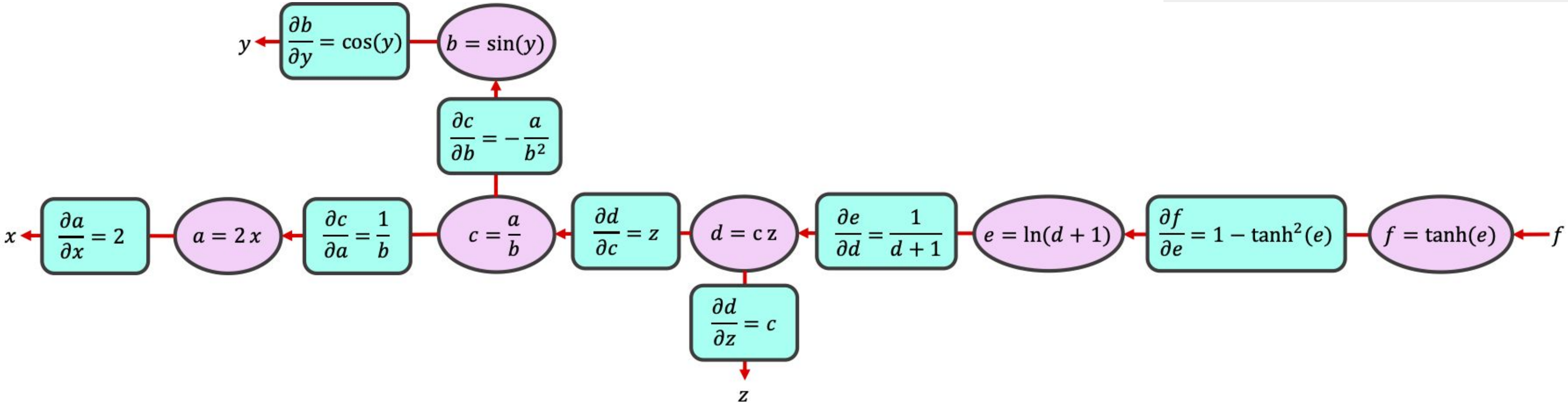
Starting from the top, pass backward. Each *edge* stores partial derivative of the head of the edge with respect to the tail.




Conveniently, the partial derivatives can often be expressed using the intermediate variables calculated in the forward pass (*a,b,c,d,e)*.

# Computational Graph (gradients)

Gradients can then be easily computed using the chain rule.



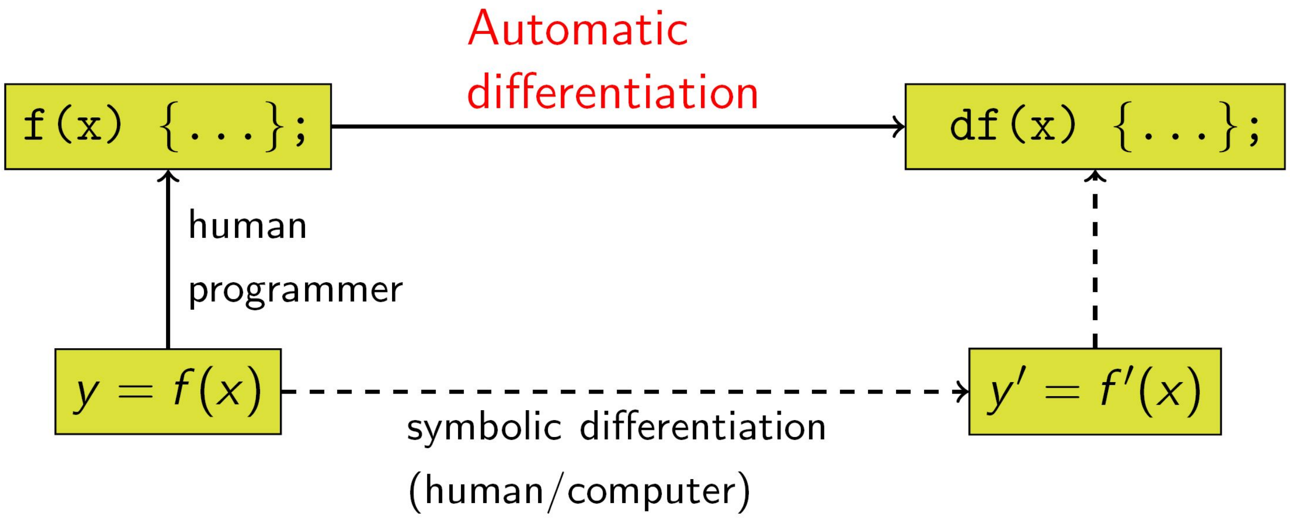

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial a}\frac{\partial a}{\partial x} = (1 - \tanh^2(e)) \cdot \frac{1}{d+1} \cdot z \cdot \frac{1}{b} \cdot 2$$

# Computational Graph (gradients)

Let's try it: compute $\dfrac{\partial f}{\partial y}$

**Derive the gradient using the graph**

# The magic of automatic differentiation




wikipedia

# Building the computational graph

A data structure for storing intermediate values and partial derivatives needed to compute gradients.

- Node *v* represents variable
  - Stores value
  - Gradient
  - The function that created the node
- Directed edge from *v* to *u* represents the partial derivative of *u* w.r.t. *v*
- To compute the gradient $\partial L/\partial v$, find the unique path from *L* to *v* and multiply the edge weights, where L is the overall loss.

# Building the computational graph

When we perform operations on PyTorch Tensors, PyTorch does not simply calculate the output

Instead, each operation is added to the computational graph

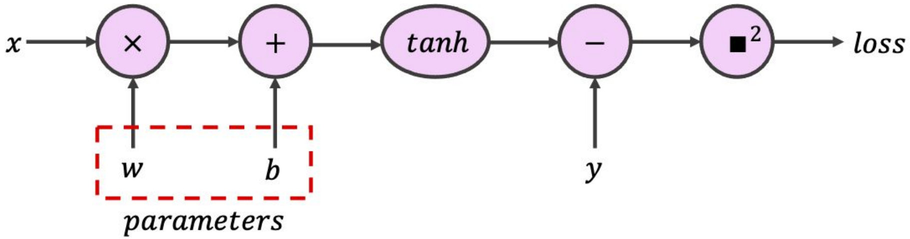
PyTorch can then do a forward and backward pass through the graph, storing necessary intermediate variables, and yield any gradients we need

# Building the computational graph

Often only some parameters are trainable and require gradients.

We indicate tensors that require gradients by setting `requires_grad=True`

$$(y - \tanh(wx + b))^2$$



# Building the computational graph

PyTorch can keep adding to the graph as your code winds through functions and classes

In essence, you write code to compute the loss *L;* AutoGrad does the rest

$$(y - \tanh(wx + b))^2$$

**Compute the gradient with respect to *w* and *b* the easy way!**

# Putting it together: a simple ANN

PyTorch can differentiate through fairly arbitrary functions

But neural networks often make use of simple building blocks

Let's look at some ways that PyTorch makes building and training ANNs particularly simple

# Aligning concepts and code

An input-output function (an ANN): $y = f_w(x)$

nn.Module, nn.Sequential
nn.Linear, nn.ReLU

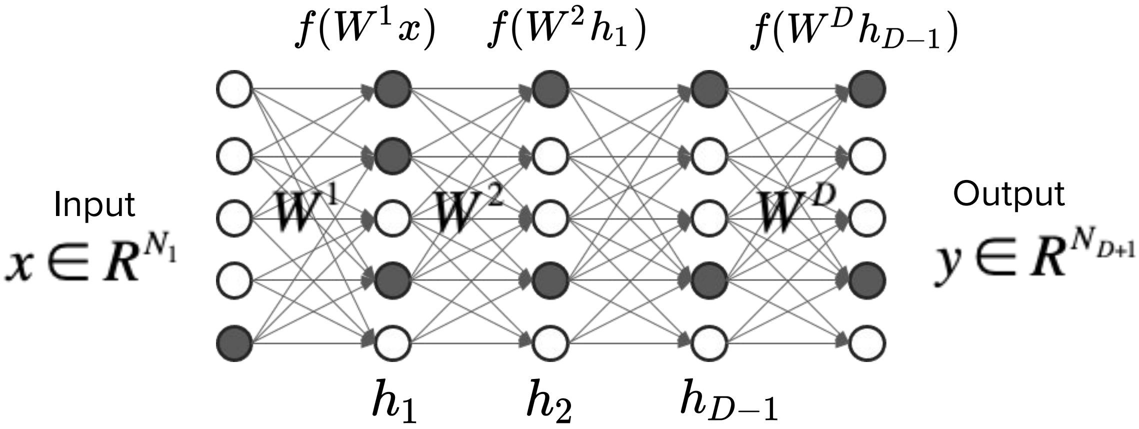
A loss function: $L = \ell(y, data)$

nn.MSELoss, nn.CrossEntropyLoss

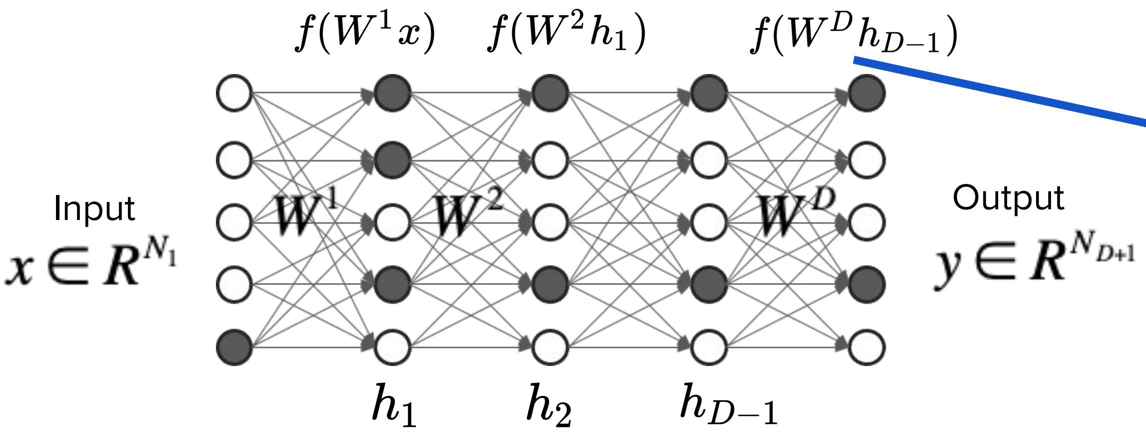
Optimization problem: $w^* = \mathrm{argmin}_w \ell(f_w(x), data)$

torch.optim.SGD, torch.optim.ADAM

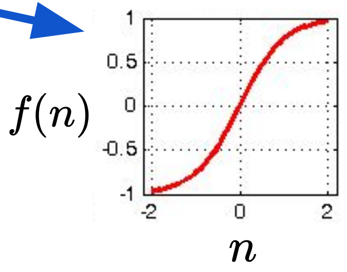
# Deep Network



# Deep Network




Neural nonlinearity



# The canonical train loop in PyTorch

```
for i in range(n_epochs):
    optimizer.zero_grad() # Reset all gradients to zero
    prediction = neural_net(inputs) # Forward pass
    loss = loss_function(prediction, targets) # Compute the loss
    loss.backward() # Backward pass to build the graph and compute the gradients
    optimizer.step() # Update weights using gradient
```

# The canonical train loop in PyTorch

Let's use these tools!

Build a network using the nn.Module and nn components

Compute predictions for some input data

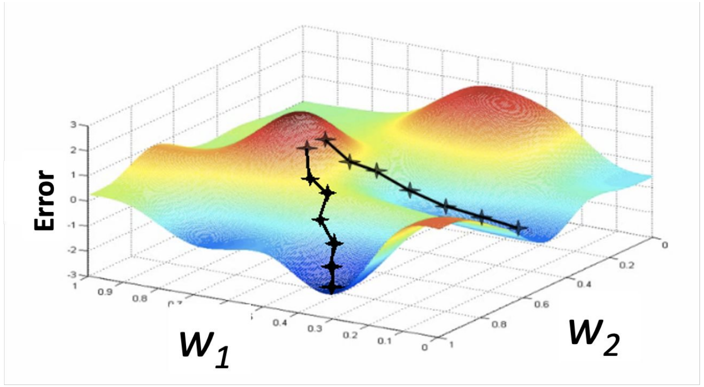
Calculate the loss

Calculate derivatives with AutoGrad

Run a step of the optimizer

**Train a neural network!**

# Wrap up: gradient descent




http://blog.datumbox.com/wp-content/uploads/2013/10/gradient-descent.png