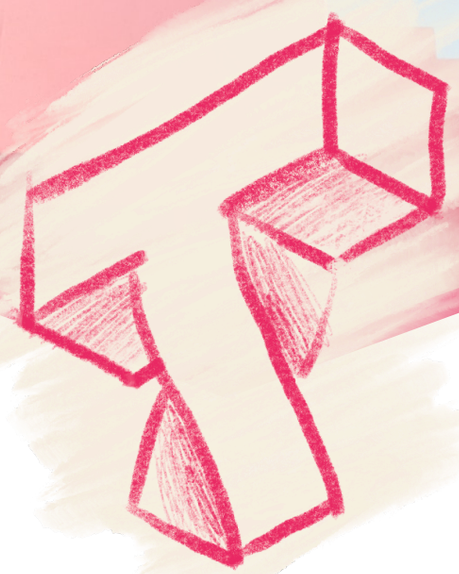
Backpropagation and Neural Networks

Kimberly Wilber

November 25



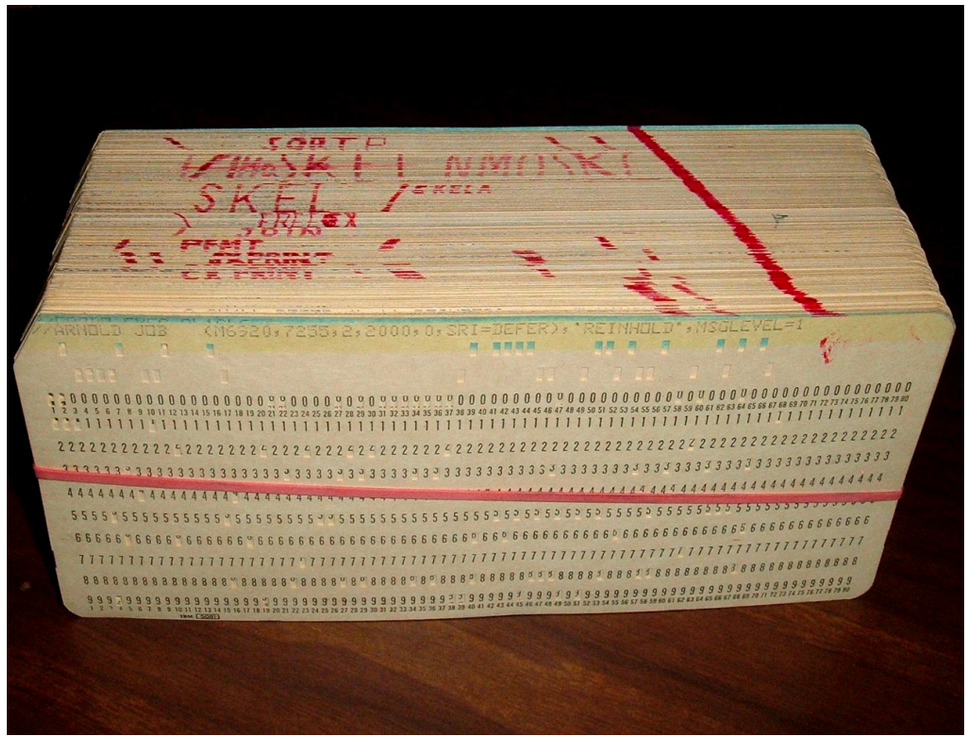
What is “deep learning”?

- Learning a model that **approximates arbitrary functions** using **hierarchical representations**…

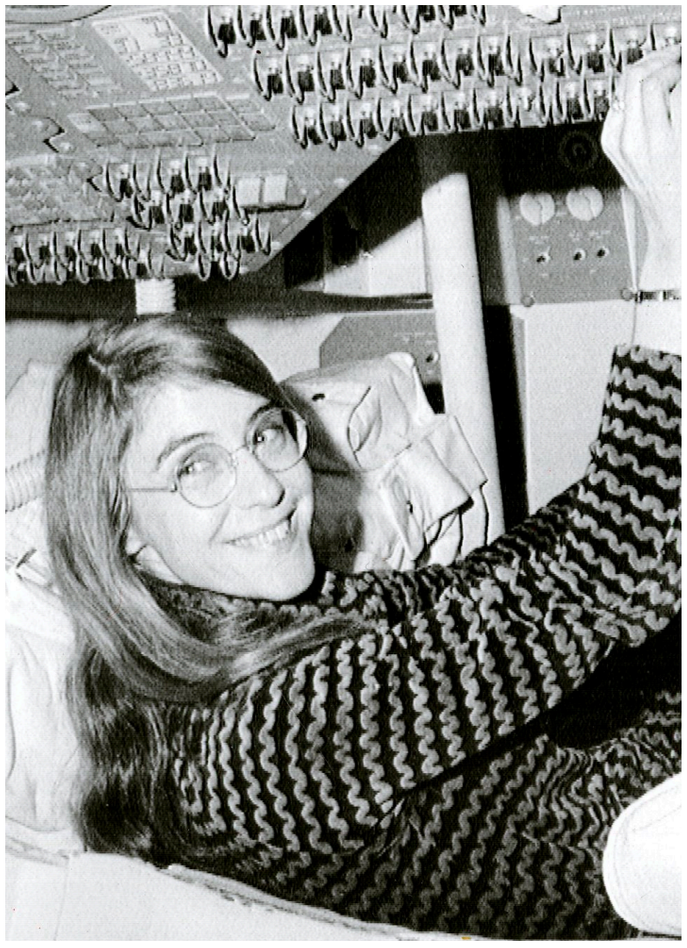
- …that are generally expressed with **explicit computation graphs** of reusable layers (building blocks)…

- …all trained with **backpropogation** and **stochastic gradient descent.**

Classic: Hand-tuned programming



Deck of punch cards for the IBM 360.
(Arnold Reinhold, 2006)



Margaret Hamilton, lead engineer of Apollo & Skylab control systems. (NASA, 1969)

✨Neural Networks✨

A short history of Neural Networks

1957: Perceptron (Frank Rosenblatt, Cornell): one layer neural network. Huge amounts of hype.

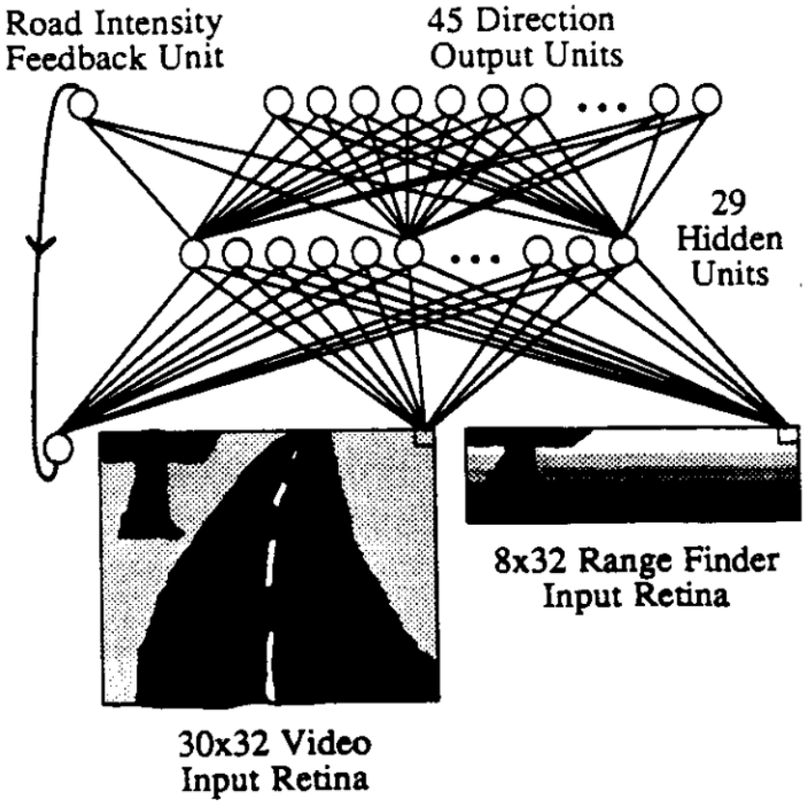
> “ The Navy last week demonstrated the embryo of an electronic computer named the Perceptron which, when completed in about a year, is expected to be the first non-living mechanism able to "perceive, recognize and identify its surroundings without human training or control."

1959: first neural network to solve a real world problem, i.e., eliminates echoes on phone lines (Widrow & Hoff)

1988: Backpropagation (Rumelhart, Hinton, Williams): learning a multi-layered network

A short history of NNs

1989: ALVINN: autonomous driving car using NN (CMU)



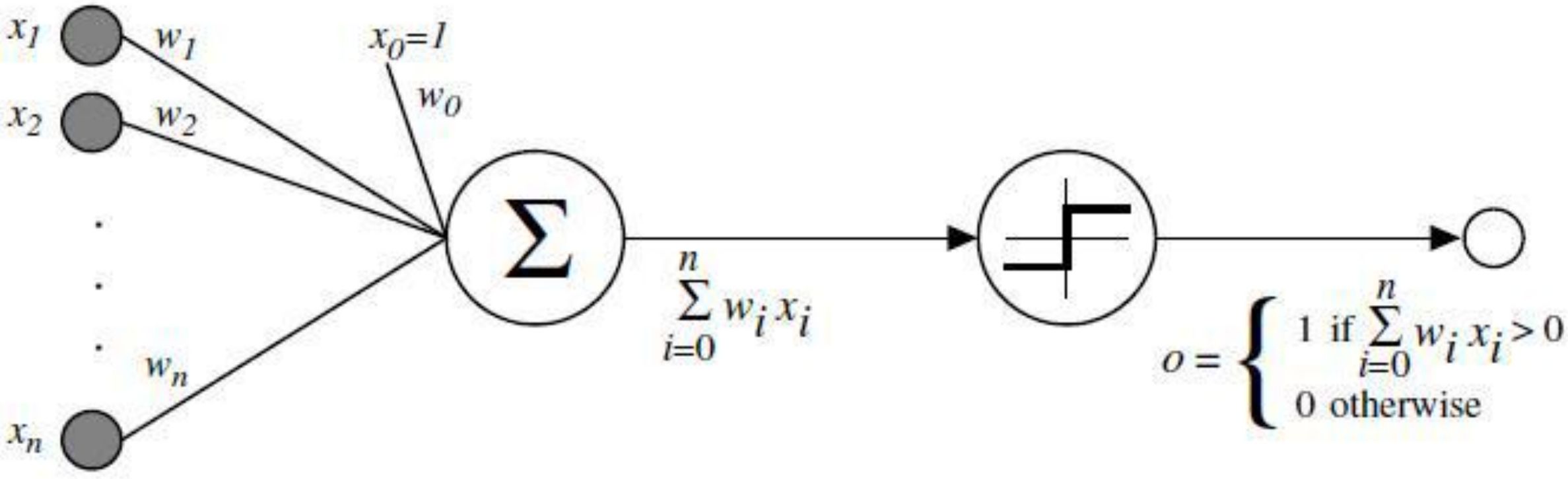
1989: (LeCun) Successful application to recognize handwritten ZIP codes on mail using a “deep” network

2006 - 2012: Advances made it possible to train the darn things on modern hardware (GPUs)

2010s: Near-human capabilities for image recognition, speech recognition, and language translation

Perceptron

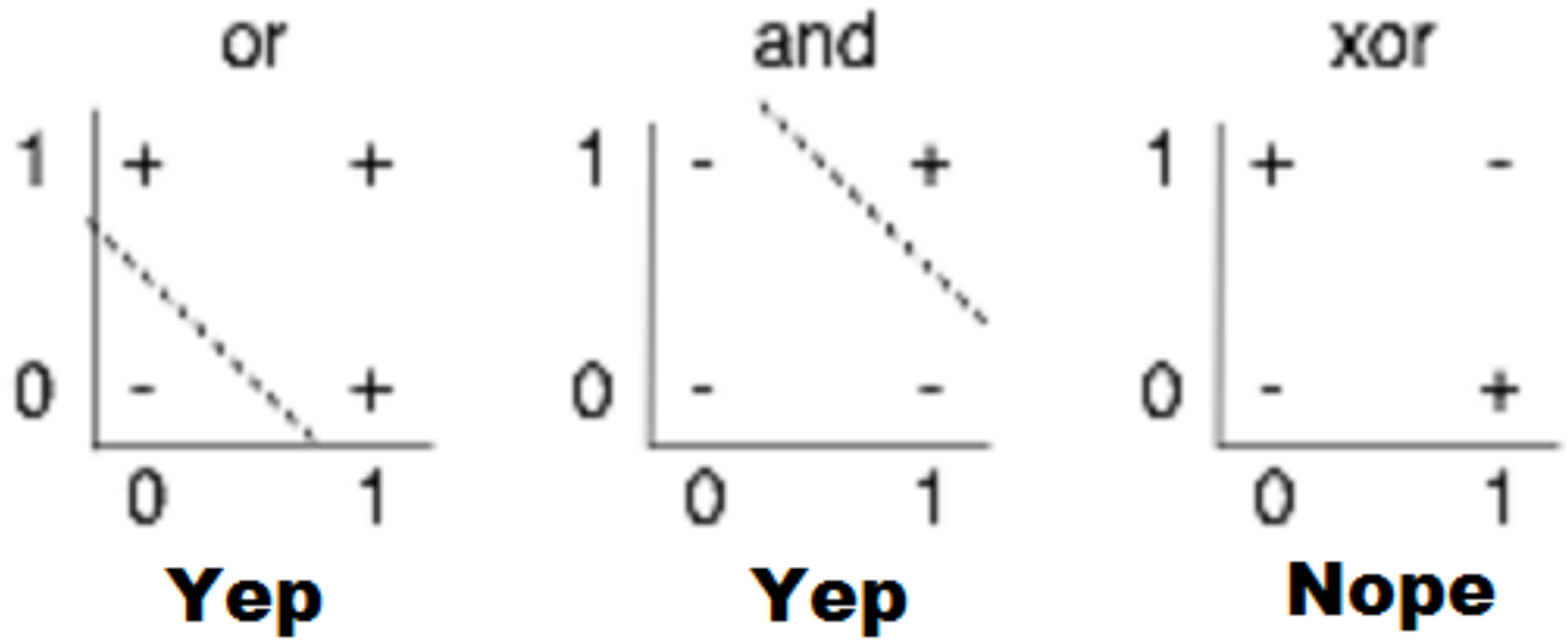
Invented by Frank Rosenblatt (1957): simplified mathematical model of how the neurons in our brains operate



From: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-deep-learning/

Perceptron

Could implement AND, OR, but not XOR. Marvin Minsky and Seymour Papert wrote a book about this problem in 1969, which killed progress in the entire field for a while.

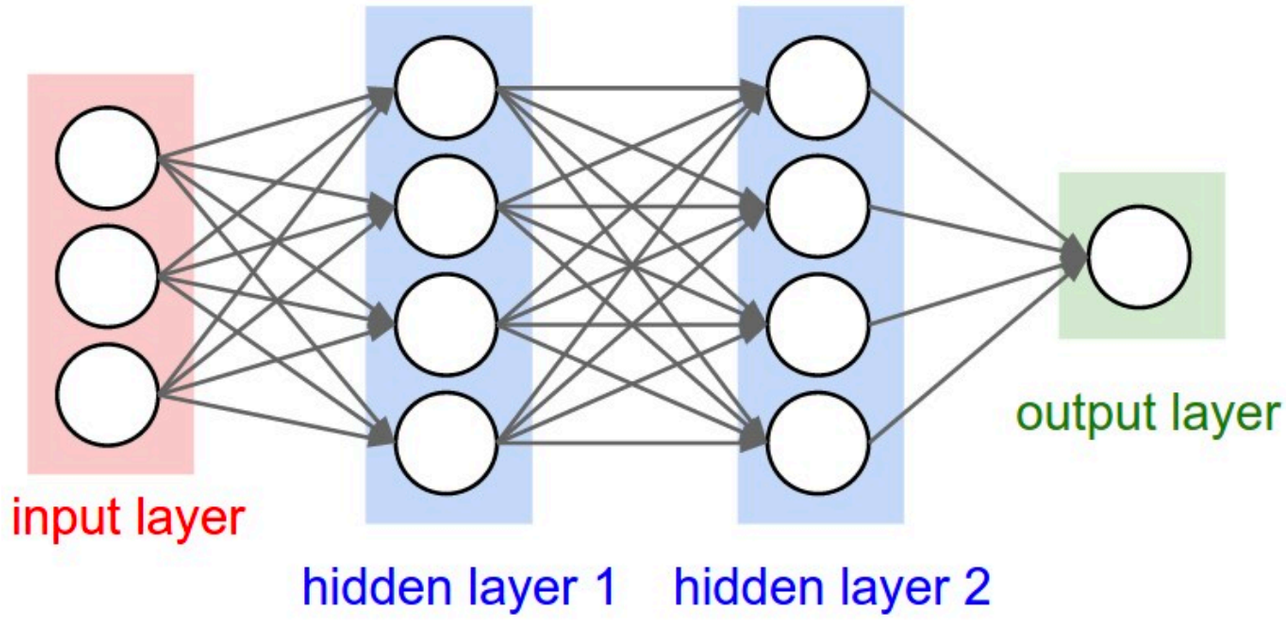


From: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-deep-learning/

Hidden layers

Hidden layers can find features within the data and allow following layers to operate on those features

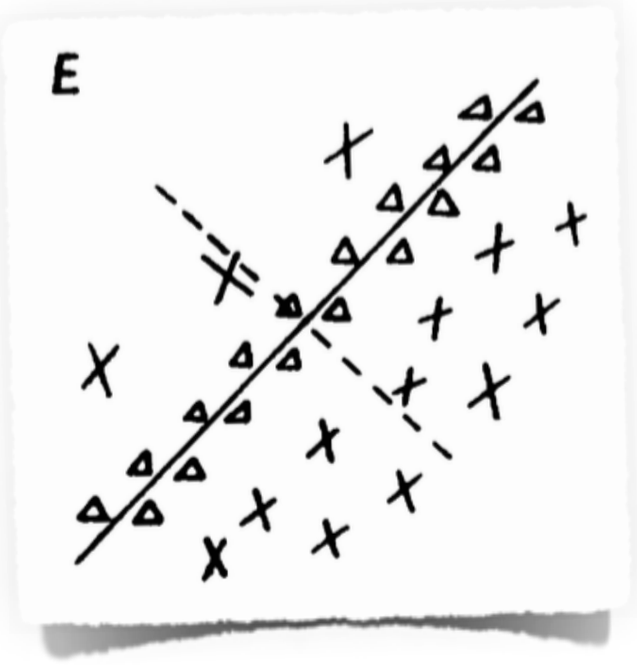
- Can implement XOR



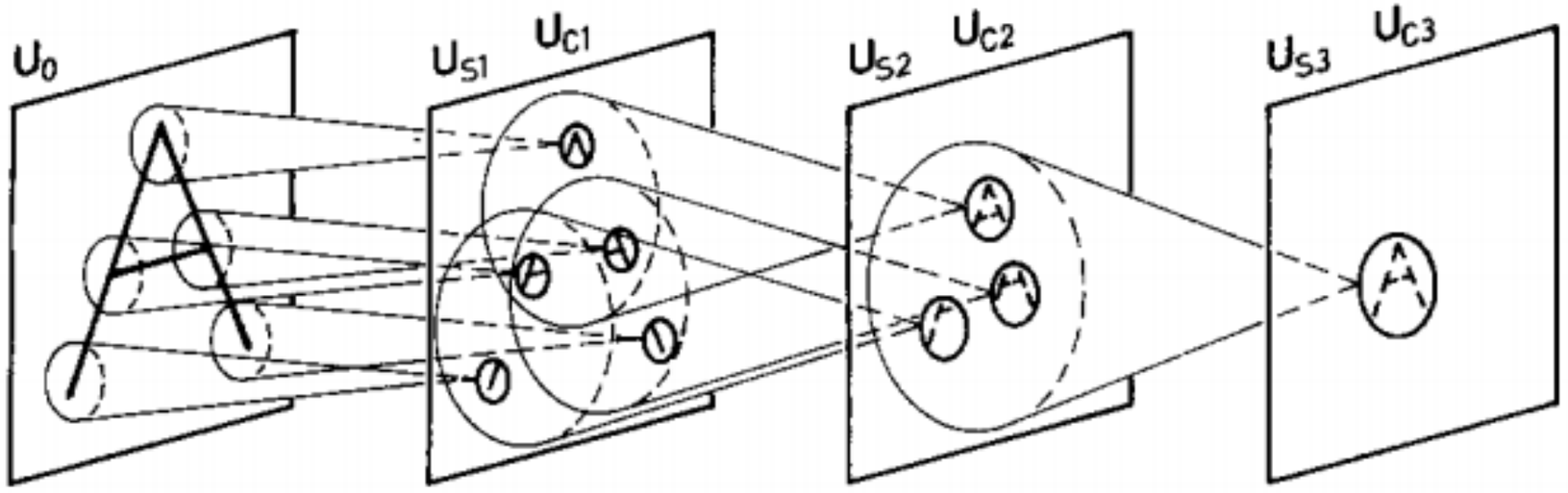
From: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-deep-learning/

Why hidden layers? Influenced by psychology

- Hubel & Wiesel, 1962 investigated the V1 area (visual cortex) in monkeys. They hypothesized that **neurons were organized in layers**, with “simple cells” connected to “higher-level cells”



Hubel & Wiesel, 1962
Receptive field

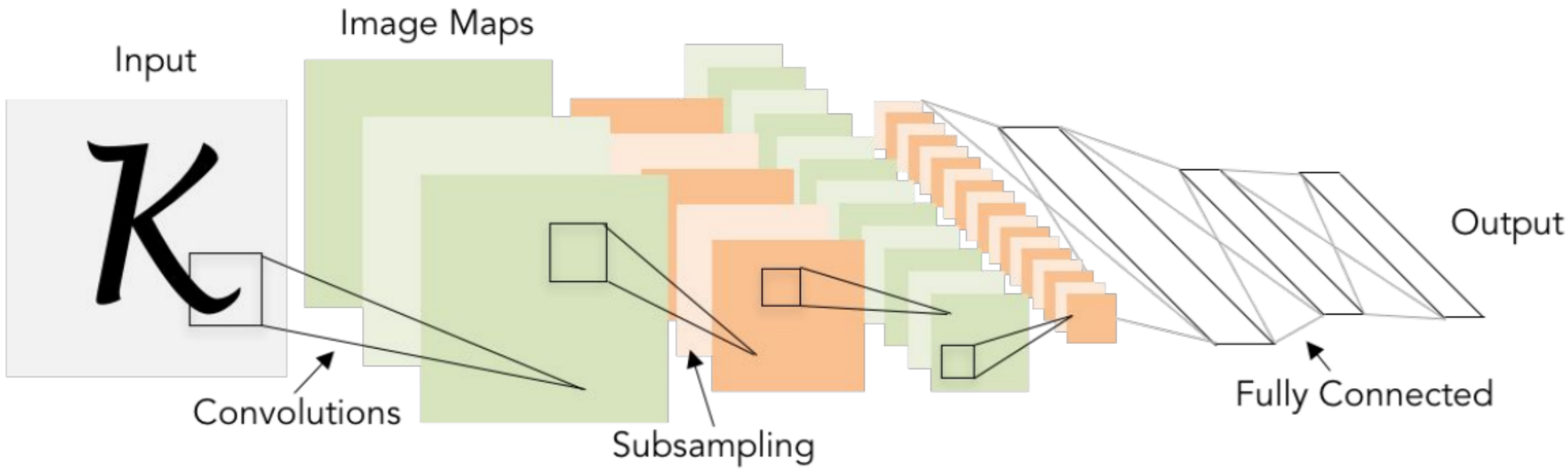


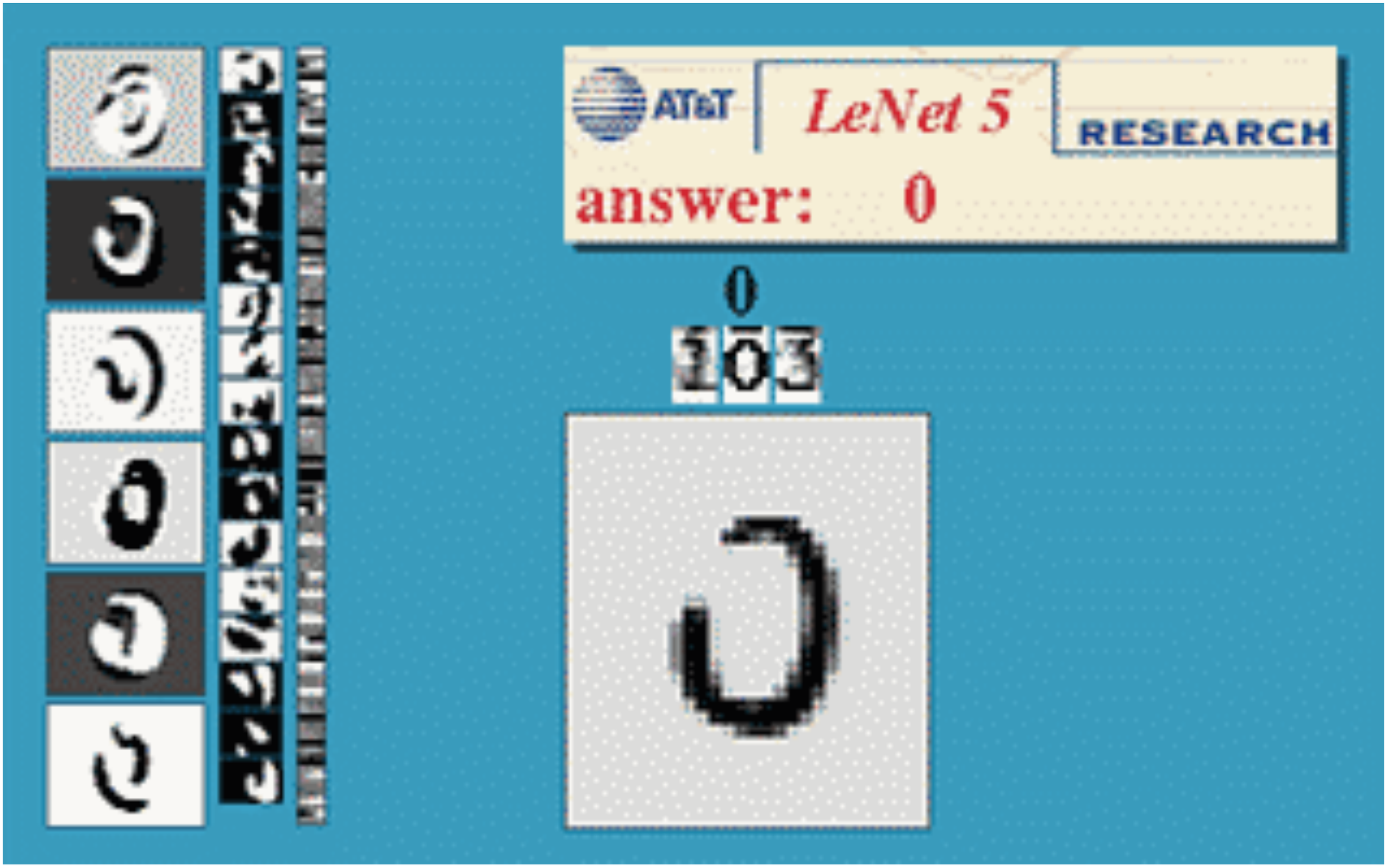
Fukushima, 1980. The Neocognitron, one of the first feedforward neural networks.

1989: LeCunn and friends' digit recognizer

- Yann LeCunn's "LeNet" built off of these ideas to build a classifier for handwritten digits

- Applied to bank checks and postal zip codes. The system was deployed in ATMs and used to process 10-20% of checks in the US.




AT&T
LeNet 5
RESEARCH
answer: 0
0
103

4627

These days, NNs are popular…

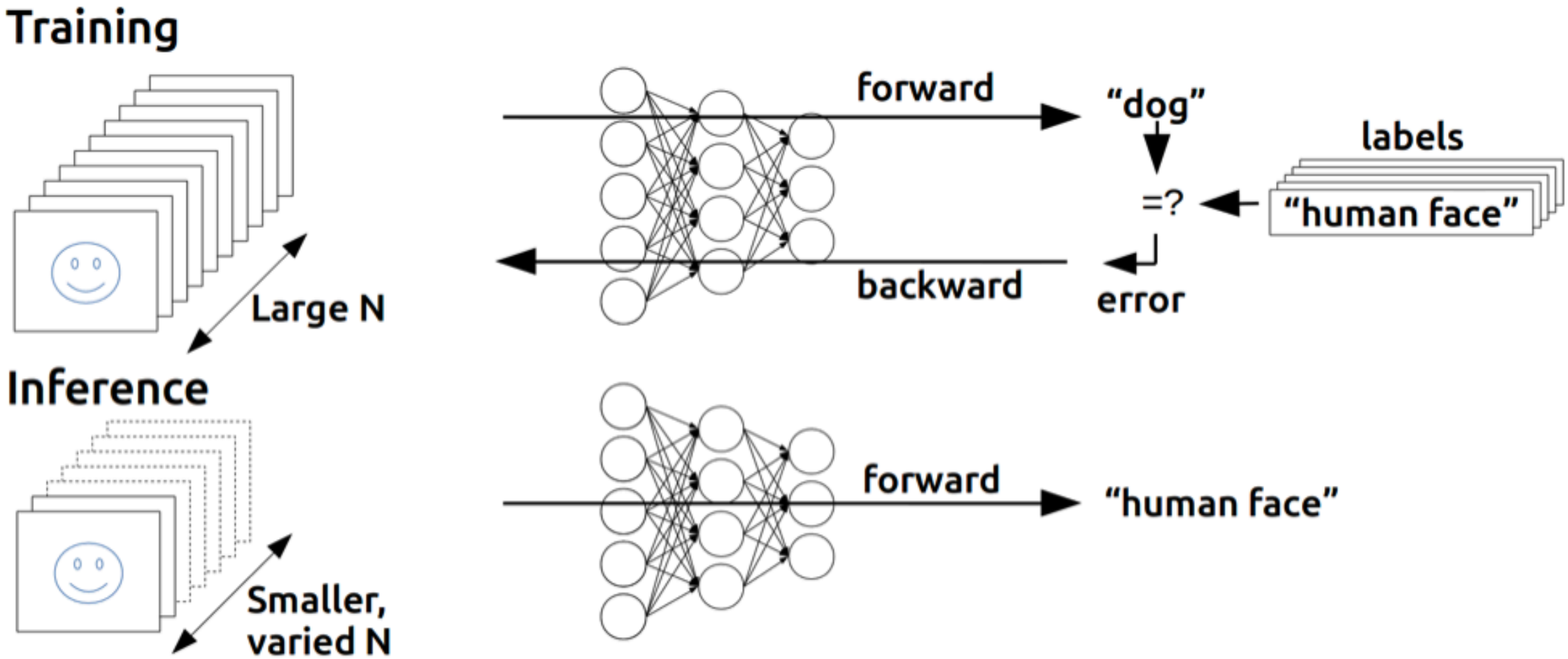
Deep learning already claiming big successes

Team	Year	Place	Error (top-5)
XRCE (pre-neural-net explosion)	2011	1st	25.8%
Supervision (AlexNet)	2012	1st	16.4%
Clarifai	2013	1st	11.7%
GoogLeNet (Inception)	2014	1st	6.66%
Andrej Karpathy (human)	2014	N/A	5.1%
BN-Inception (Arxiv)	2015	N/A	4.9%
Inception-v3 (Arxiv)	2015	N/A	3.46%

Imagenet challenge classification task

From: http://www.wsdm-conference.org/2016/slides/WSDM2016-Jeff-Dean.pdf

Learning: Backpropagation



From: http://www.andreykurenkov.com/writing/ai/a-brief-history-of-neural-nets-deep-learning/

Outline: What is deep learning?

- Learning a model that **approximates arbitrary functions** using **hierarchical representations**…
- …that are generally expressed with **explicit computation graphs** of reusable layers (building blocks)…
- …all trained with **backpropogation** and **stochastic gradient descent.**

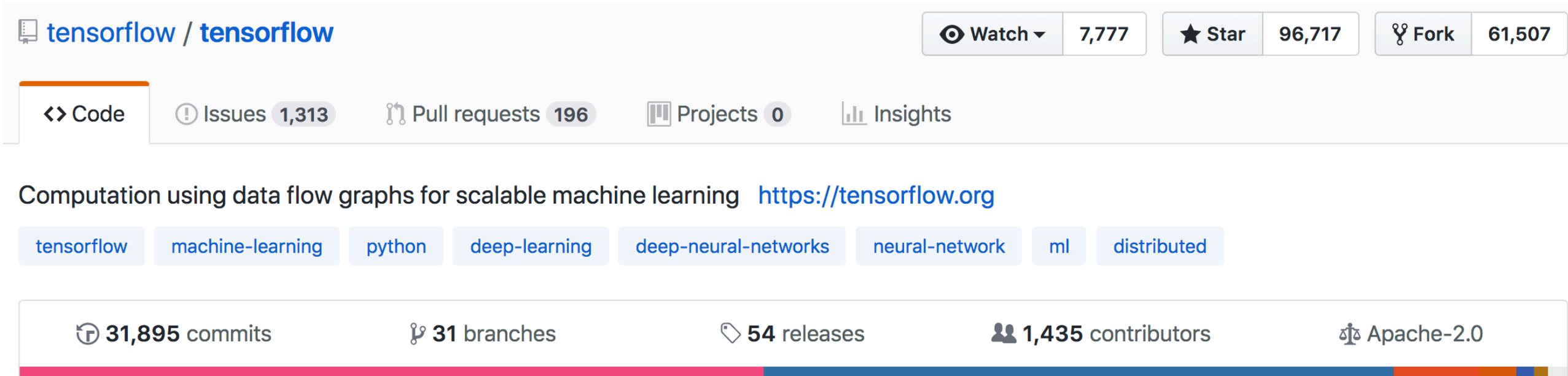
What is TensorFlow?

Open source library for numerical computation using **data flow graphs**

Developed by Google Brain Team to conduct machine learning research

- Based on DisBelief used internally at Google since 2011

"TensorFlow is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms"



What is TensorFlow?

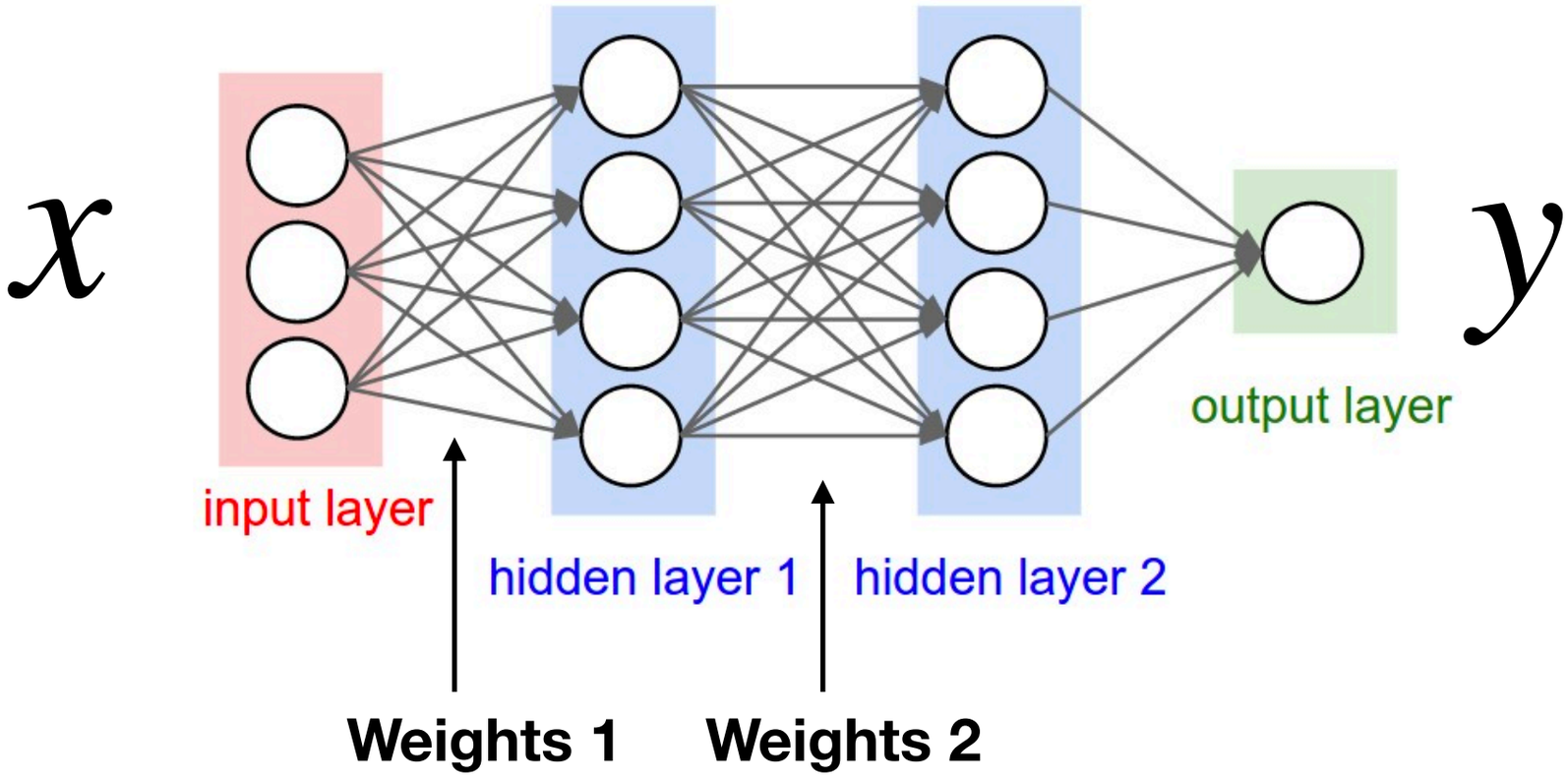
Key idea: express a numeric computation as a graph

Graph nodes are operations with any number of inputs and outputs

Graph edges are tensors which flow between nodes

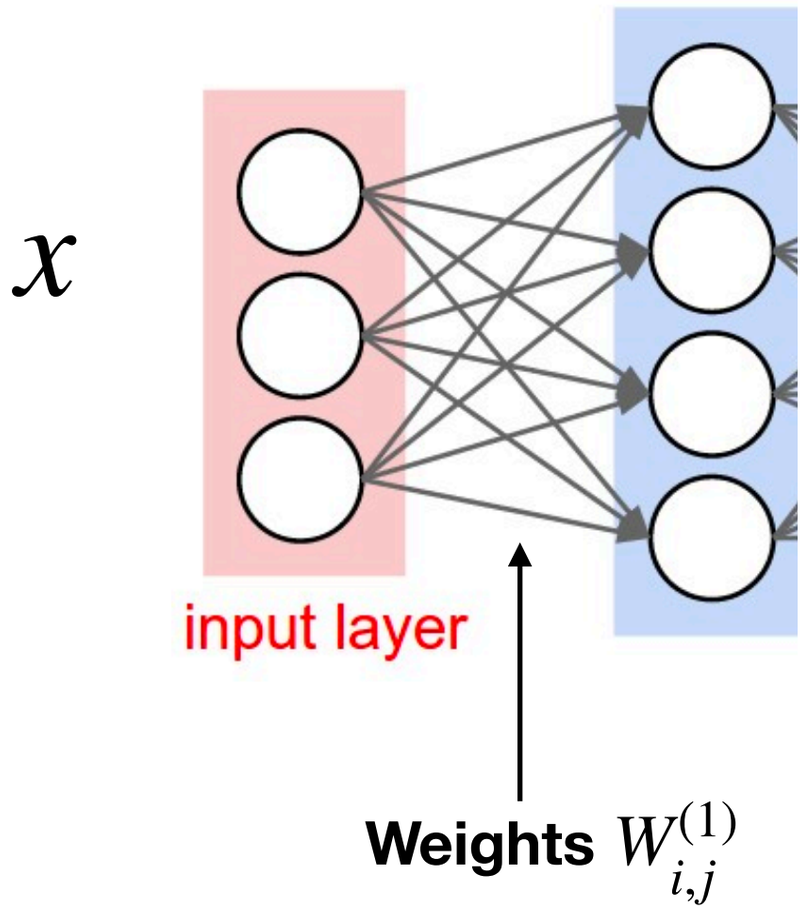
Programming model

Typically we don't express each neuron individually!
Instead, we use building blocks that use **matrix multiplication** to group operations together!



Programming model

Typically we don't express each neuron individually!
Instead, we use building blocks that use **matrix multiplication** to group operations together!

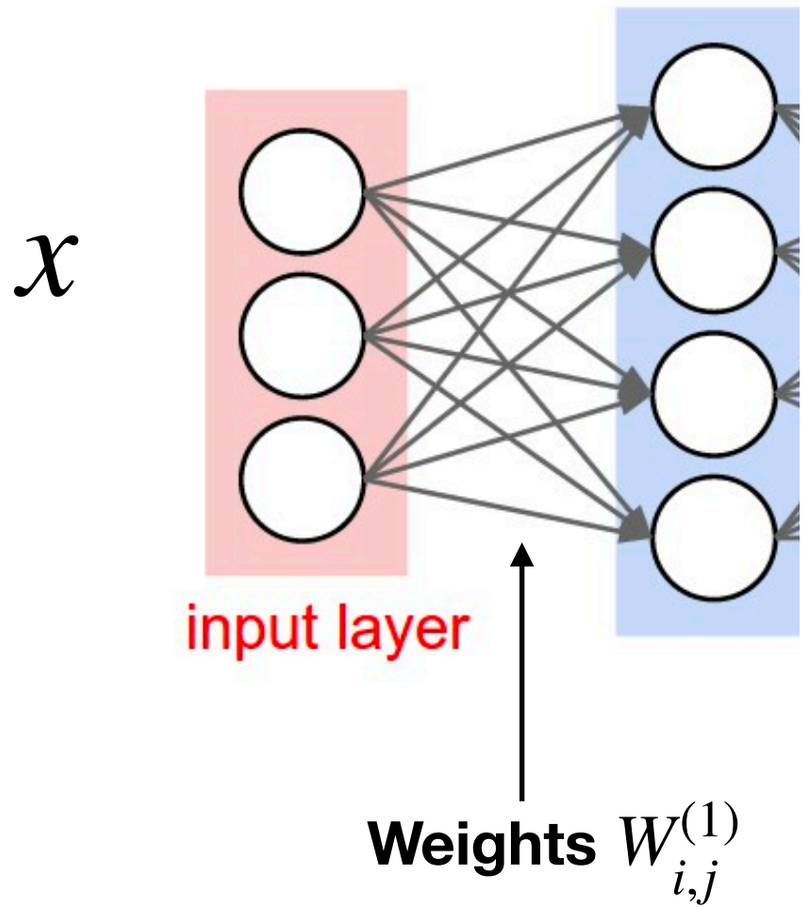


$$x$$

$$y_i^{(1)} = \sum_j x_j \cdot W_{i,j}^{(1)}$$

Programming model

Typically we don't express each neuron individually!
Instead, we use building blocks that use **matrix multiplication** to group operations together!



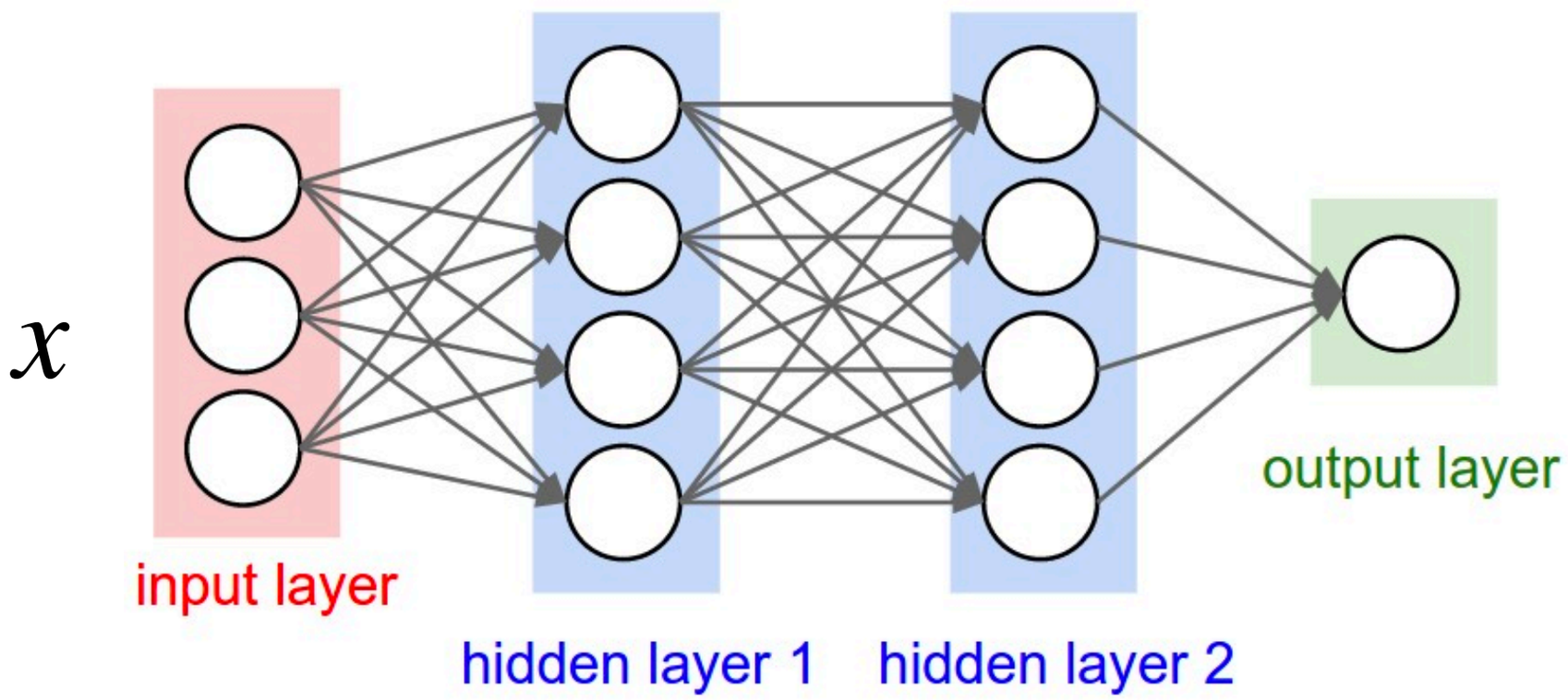
$$y_i^{(1)} = \sum_j x_j \cdot W_{i,j}^{(1)}$$

$$\mathbf{y}^{(1)} = W^{(1)}\mathbf{x}$$

(Matrix multiplication)

Programming model

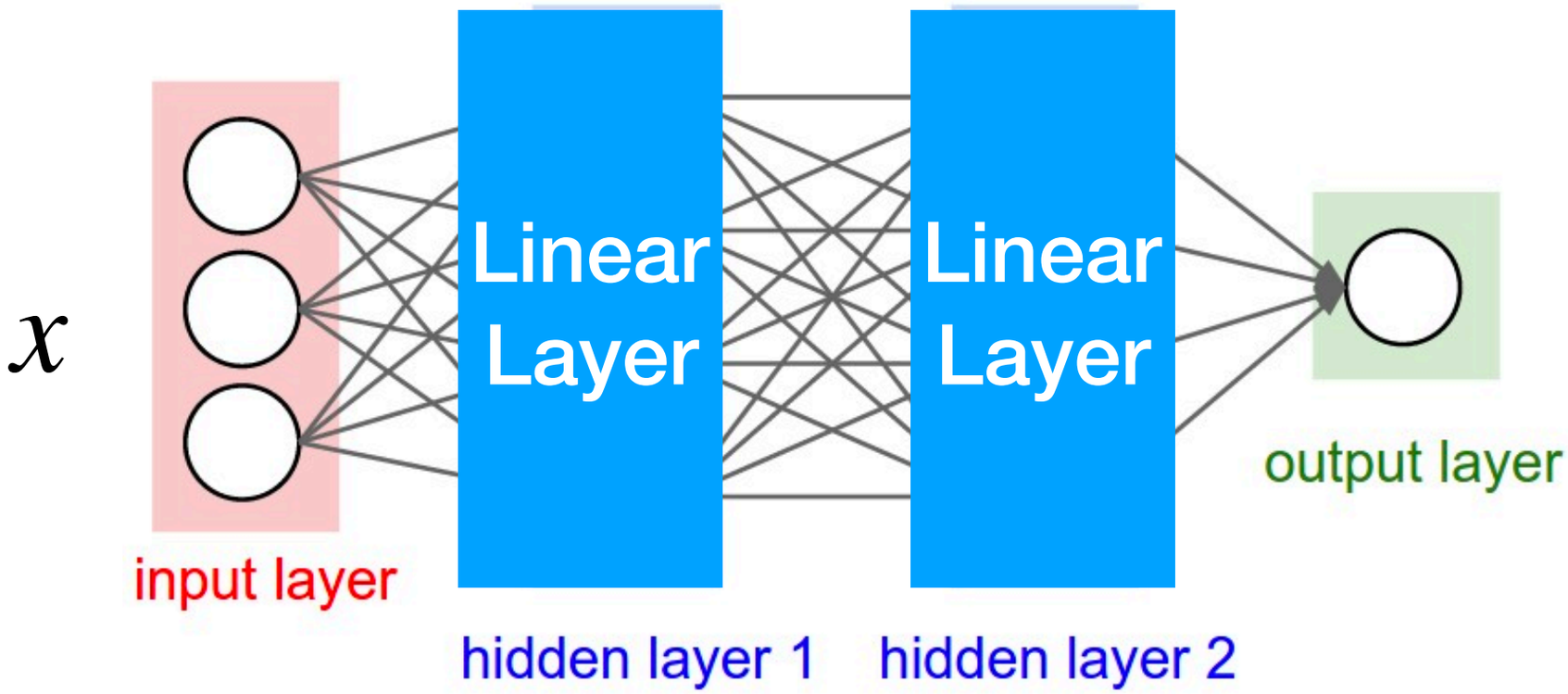
Typically we don't express each neuron individually!
Instead, we use building blocks that use **matrix multiplication** to group operations together!



$$\mathbf{y}^{(1)} = W^{(1)}\mathbf{x} \qquad \mathbf{y}^{(2)} = W^{(2)}\mathbf{y}^{(1)}$$

Programming model

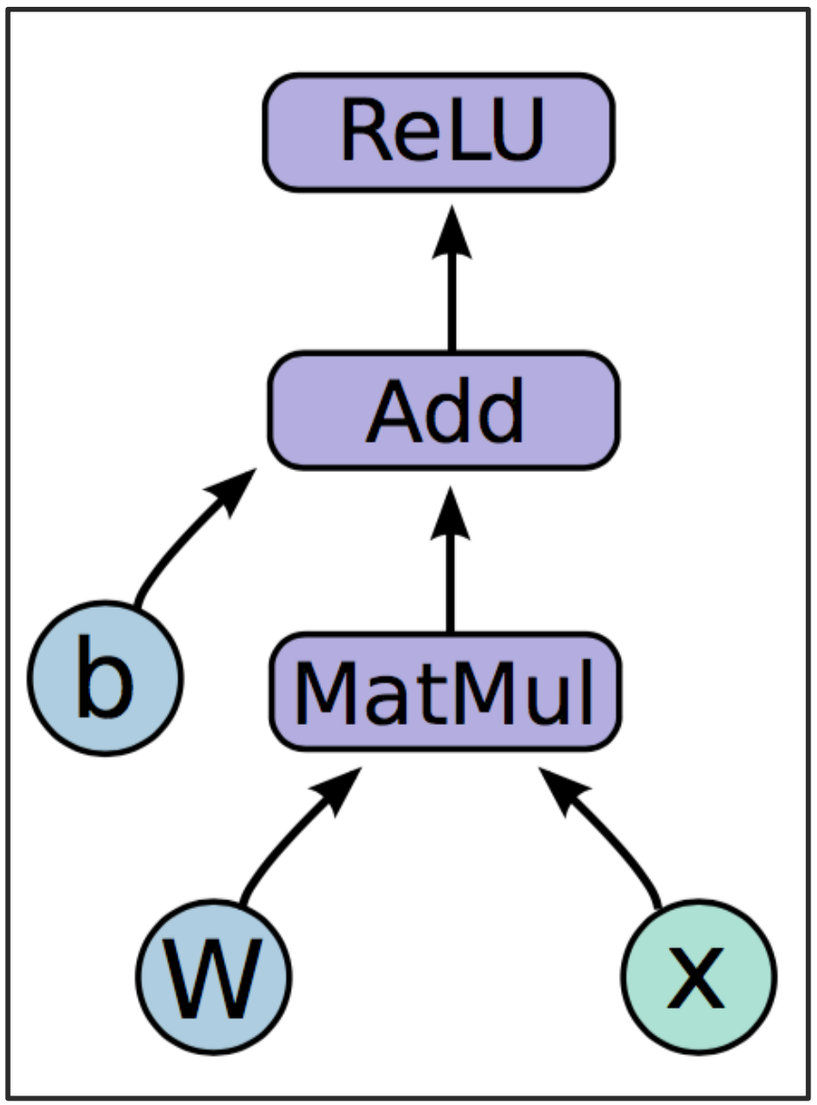
Typically we don't express each neuron individually!
Instead, we use building blocks that use **matrix multiplication** to group operations together!



$$\mathbf{y}^{(1)} = W^{(1)}\mathbf{x} \qquad \mathbf{y}^{(2)} = W^{(2)}\mathbf{y}^{(1)}$$

Typical layer “building block”

$$h = ReLU(Wx + b)$$

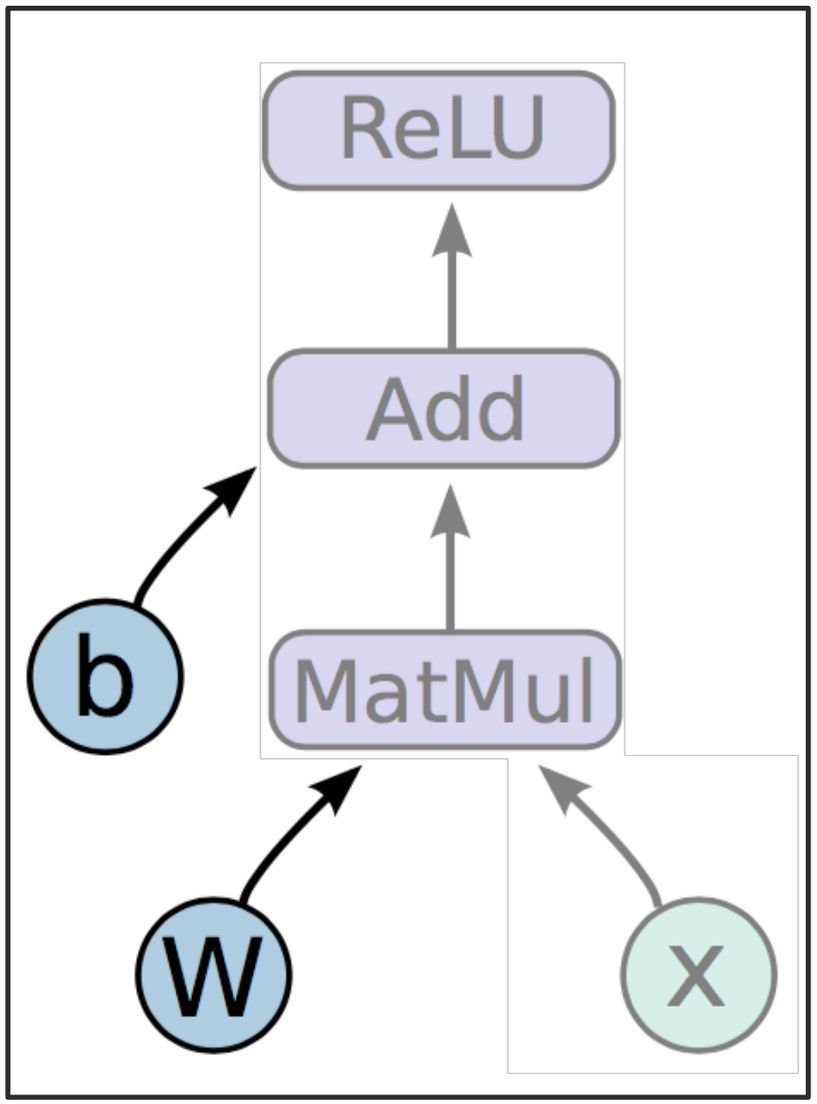


Typical layer “building block”

$$h = ReLU(Wx + b)$$

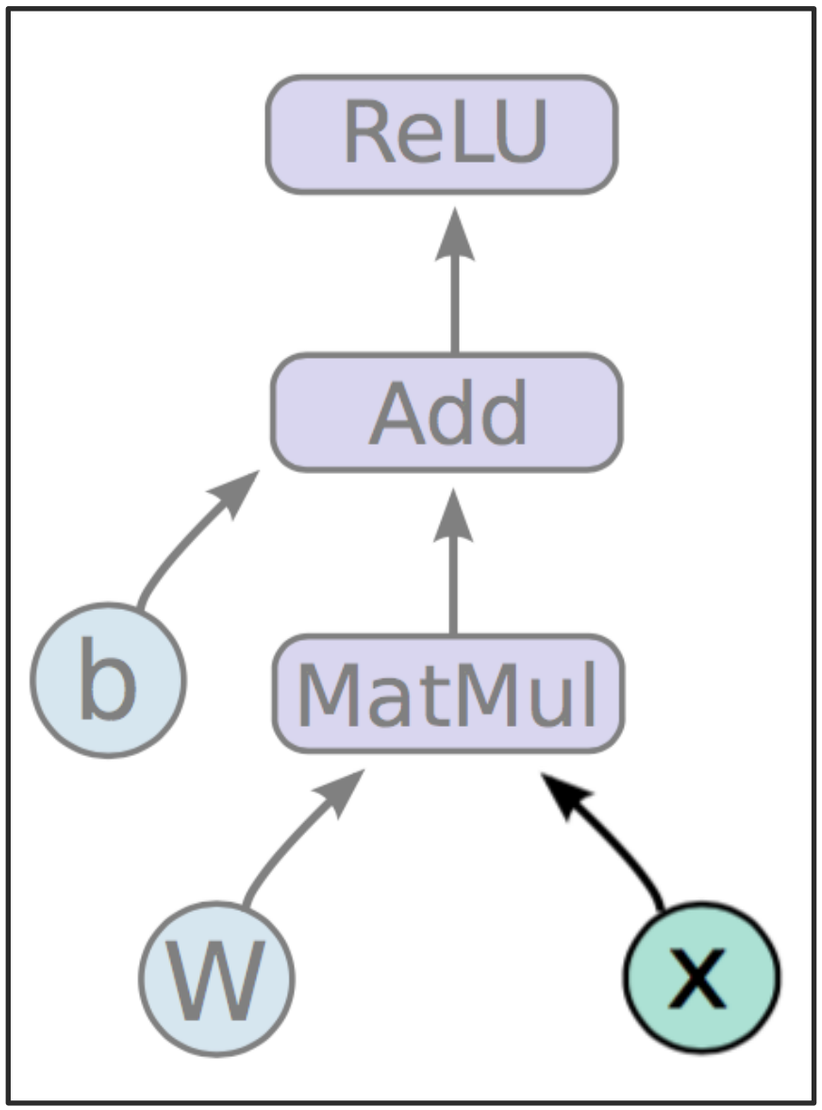
Variables are stateful nodes which output their current value. State is retained across multiple executions of a graph

(mostly parameters)



Typical layer “building block”

$$h = ReLU(Wx + b)$$



Placeholders are nodes whose value is fed in at execution time

(inputs, labels, …)

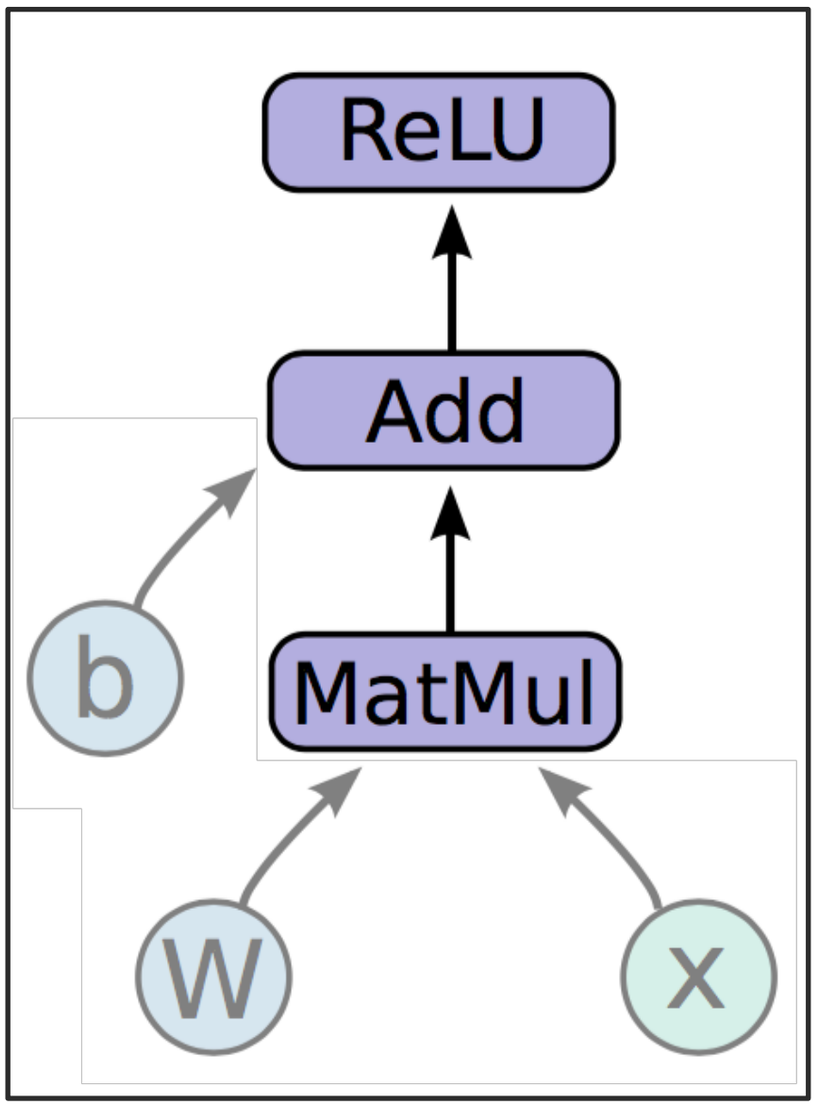
Typical layer “building block”

Mathematical operations:

MatMul: Multiply two matrices

Add: Add elementwise

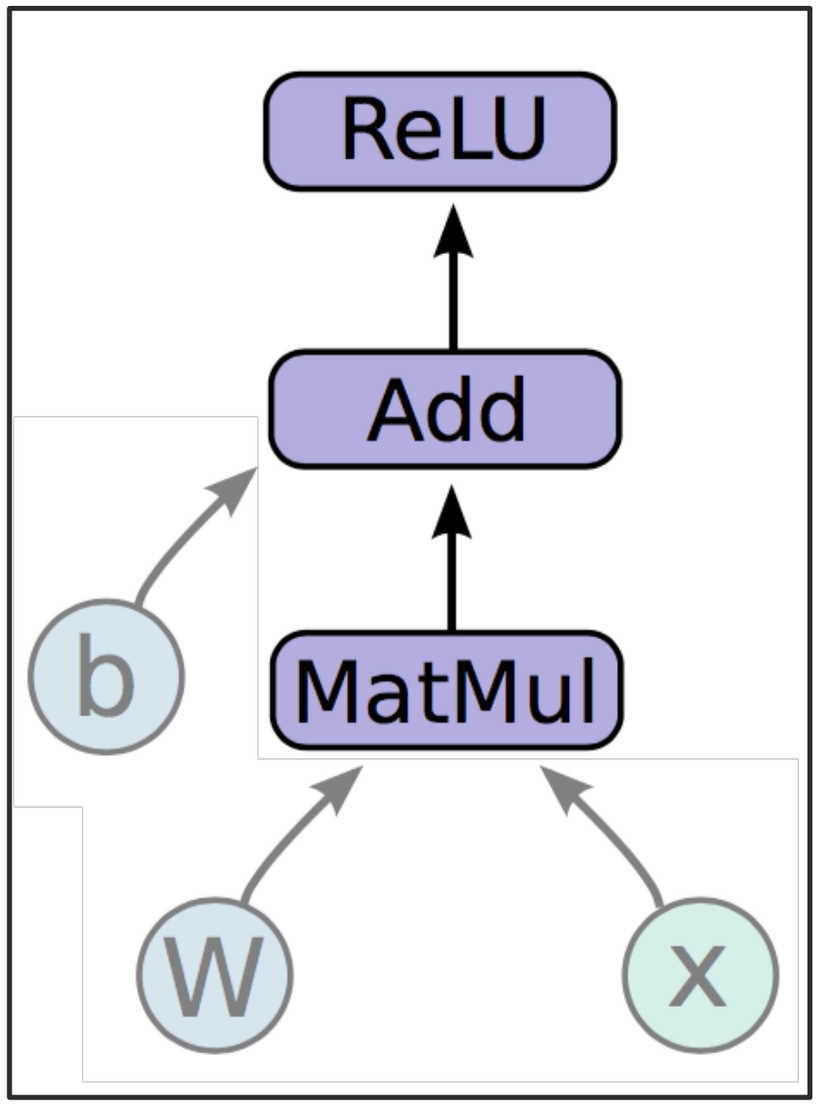
ReLU: Activate with elementwise rectified linear function



$$ReLu(x) = \begin{cases} 0, & x <= 0 \\ x, & x > 0 \end{cases}$$

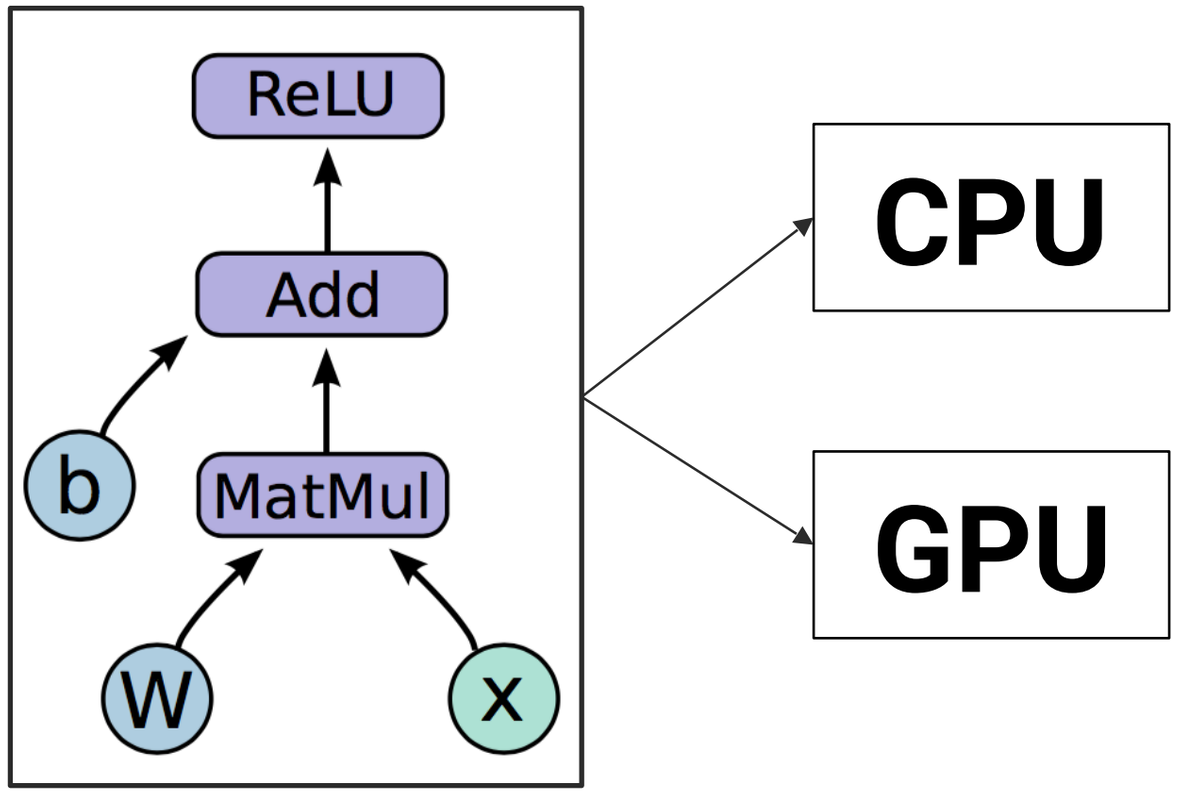
Example code (just get a feel)

```
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(
    tf.random_uniform((784, 100), -1, 1))

x = tf.placeholder(tf.float32, (1, 784))

h = tf.nn.relu(tf.matmul(x, W) + b)
```



Running the graph

Deploy graph with a session: a binding to a particular execution context (e.g. CPU, GPU)



Outline: What is deep learning?

- Learning a model that **approximates arbitrary functions** using **hierarchical representations**…

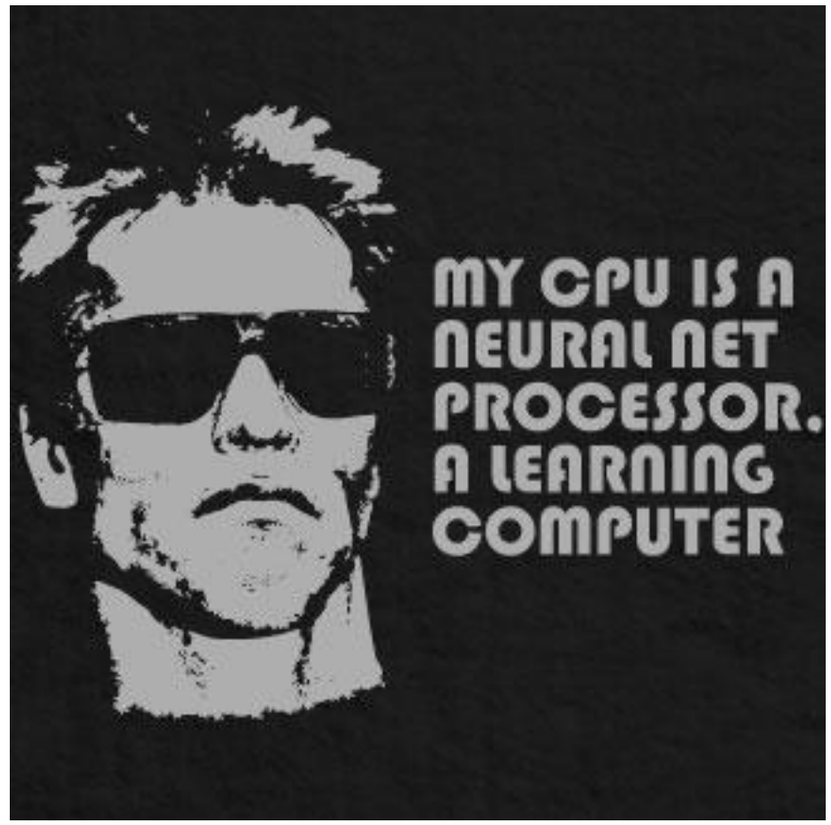
- …that are generally expressed with **explicit computation graphs** of reusable layers (building blocks)…

- …all trained with **backpropogation** and **stochastic gradient descent.**

“Learned" representations

- Models can be written as functions of their inputs, x, and their weights, θ. Their outputs, y, are task-specific.

$$f(x, \theta) \rightarrow y$$



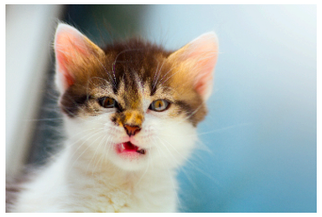
- Example: For Imagenet classification,
  - x is an input image (150k dimensions);
  - θ is all trainable parameters (typically 23,000,000 dimensions);
  - y is a list of possible classes to detect (1000 dimensions)

- Goal: Use **optimization** techniques to pick the best θ given a training set X, Y

“Learned" representations

- Consider a large **dataset** of **input/ label** pairs, $\{(x_1, y_1), \cdots, (x_N, y_N)\}$.

x y

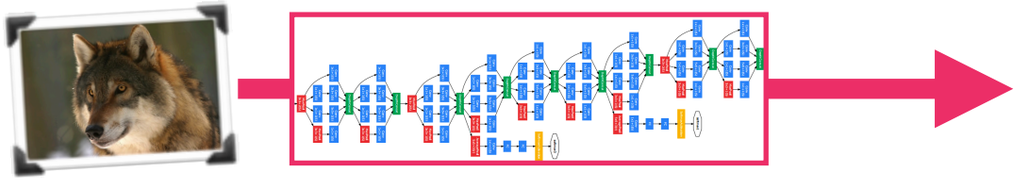


#23, cat

- We have a **model**, $f(x, \theta) \rightarrow \hat{y}$

#115, timber wolf

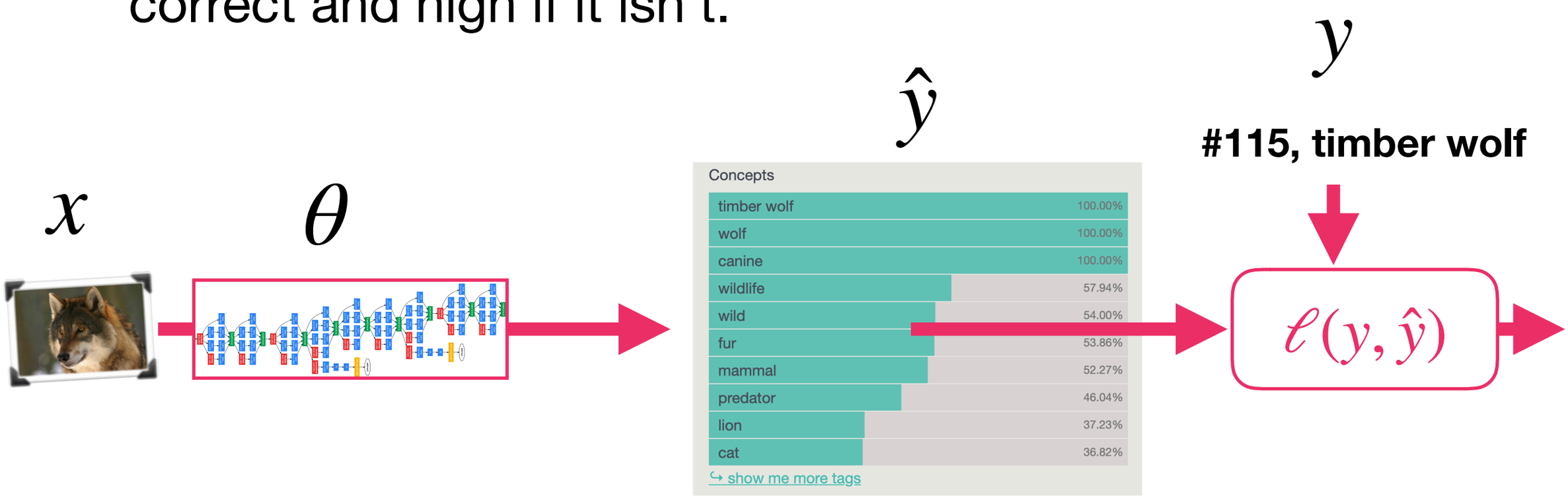
x θ $\hat{y}$



“Learned" representations

- A **loss function**, ℓ, compares the estimated $\hat{y}$ with the true y, outputting a **loss** value.

- This value should be low if the model is correct and high if it isn’t.



“Learned" representations

- A few common losses that people like to use:
  - **Classification** problems typically use the **cross-entropy loss**: taking the negative log-likelihood of the softmax-normalized model output. This loss is nice because it minimizes the KL-divergence between the true and predicted score distributions.

$$\ell_{CE}(y, \hat{y}) = -\log\left(\frac{e^{\hat{y}_y}}{\sum_j e^{\hat{y}_j}}\right)$$

  - **Regression** problems typically use **L2** loss, encouraging the output of the network and the expected value to have a low Euclidean distance.

$$\ell_{L_2}(y, \hat{y}) \quad = \quad \|y - \hat{y}\|^2 \quad = \quad \sum_i (y_i - \hat{y}_i)^2$$

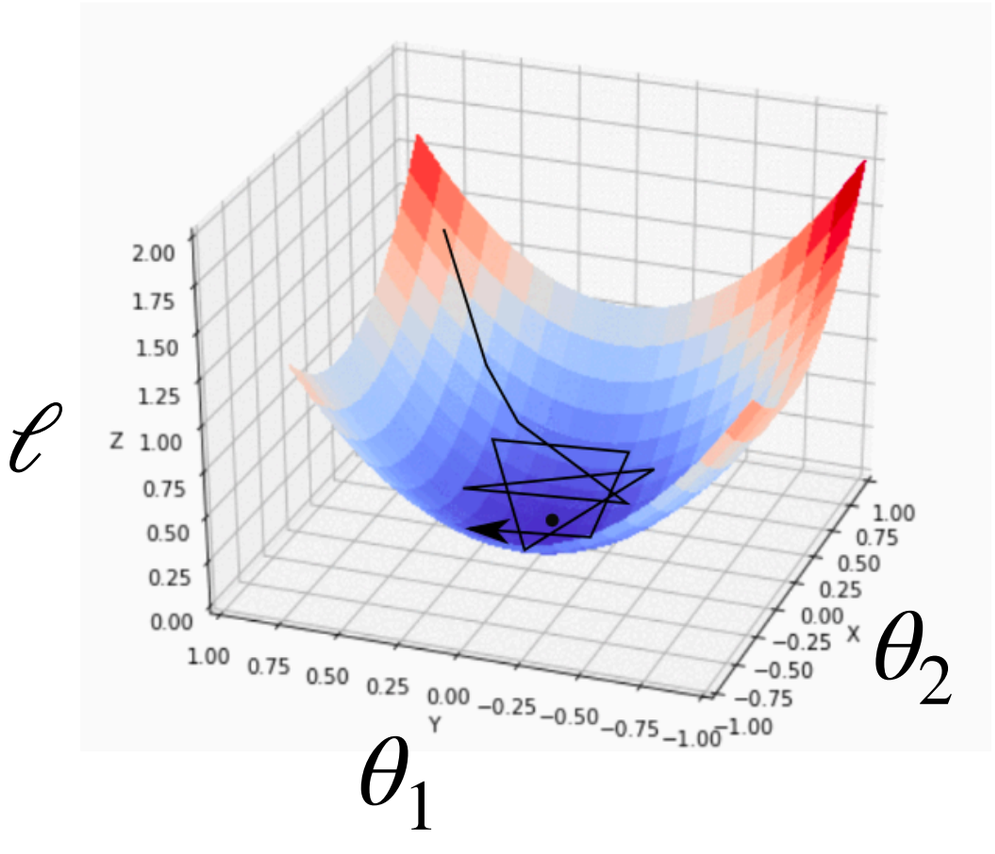
“Learned" representations

Overall goal: Given a dataset, and given a model architecture parameterized by θ, find the θ that **minimizes the training loss** over that dataset.

Find $\theta_{best} = \arg\min_{\theta} \ell(y, f(x, \theta))$

Loss surfaces

- Loss surfaces can be tricky! It’s often not clear how to optimize them.



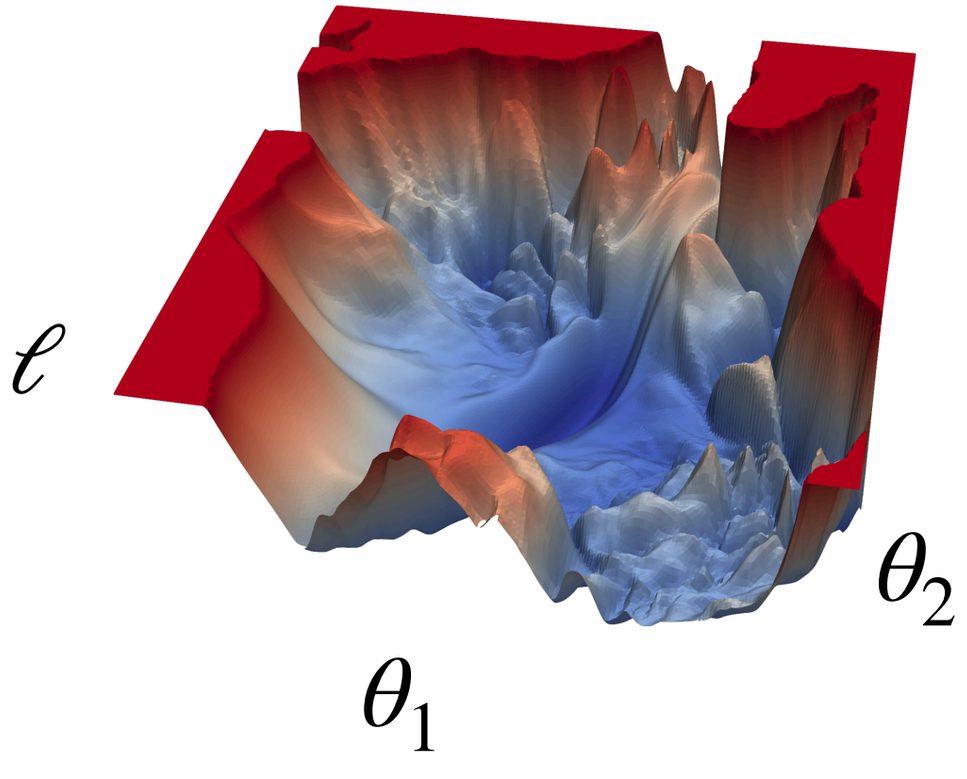
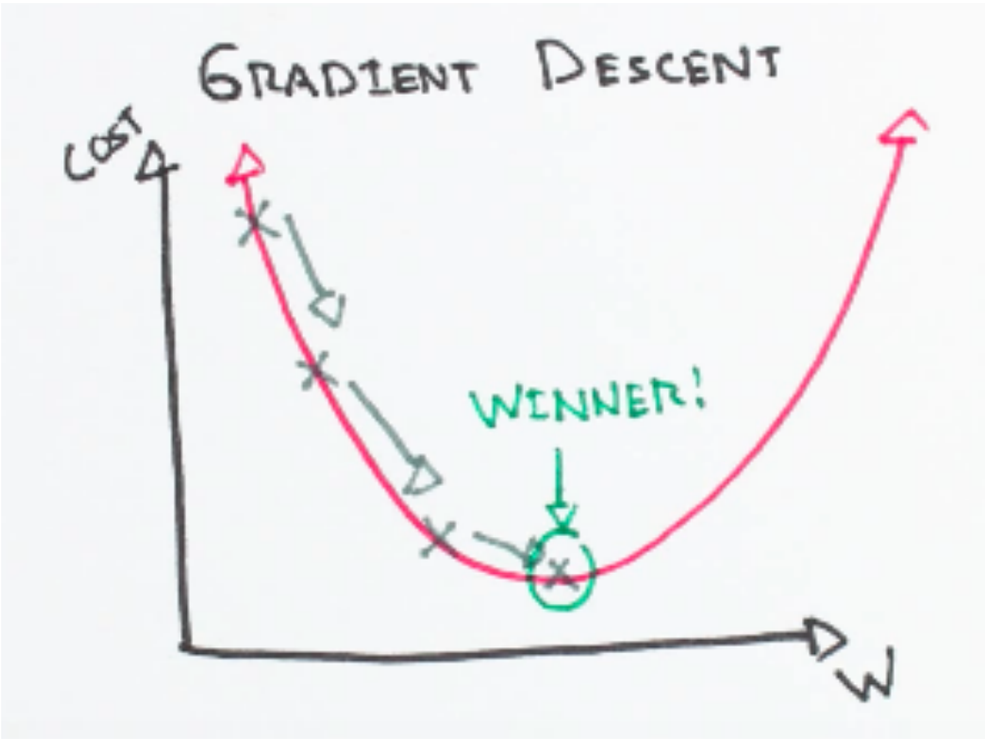


Image credit:
https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/
https://www.cs.umd.edu/~tomg/projects/landscapes/

Gradient descent!

- We typically use some techniques from calculus to find the minimum θ:

  1. Start from an initial guess, θ_1
  2. **Forward**: Sample (x, y) from dataset, compute $\ell(f(x, \theta_0), y)$
  3. **Backward**: Compute one step:
  $$\theta_t = \theta_{t-1} - \lambda \frac{\partial \ell}{\partial \theta_{t-1}}$$
  4. Repeat until convergence

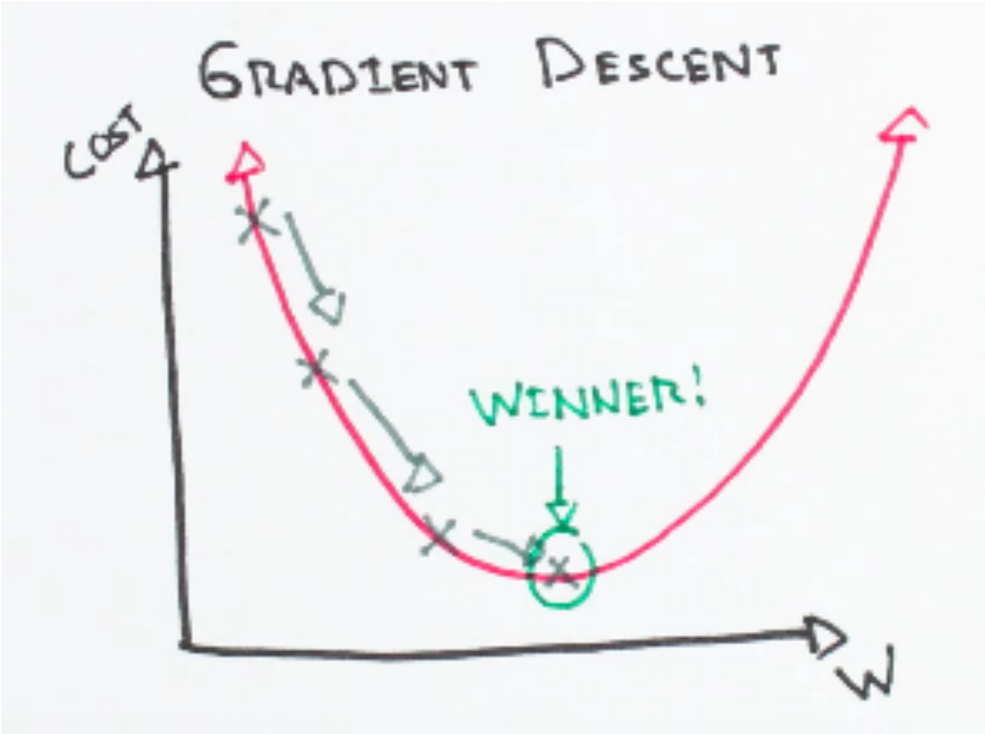


Backpropagation

- But how do we compute $\frac{\partial \ell}{\partial \theta}$? The answer: **backpropagation**! Usually using "reverse-mode accumulating automatic differentiation"

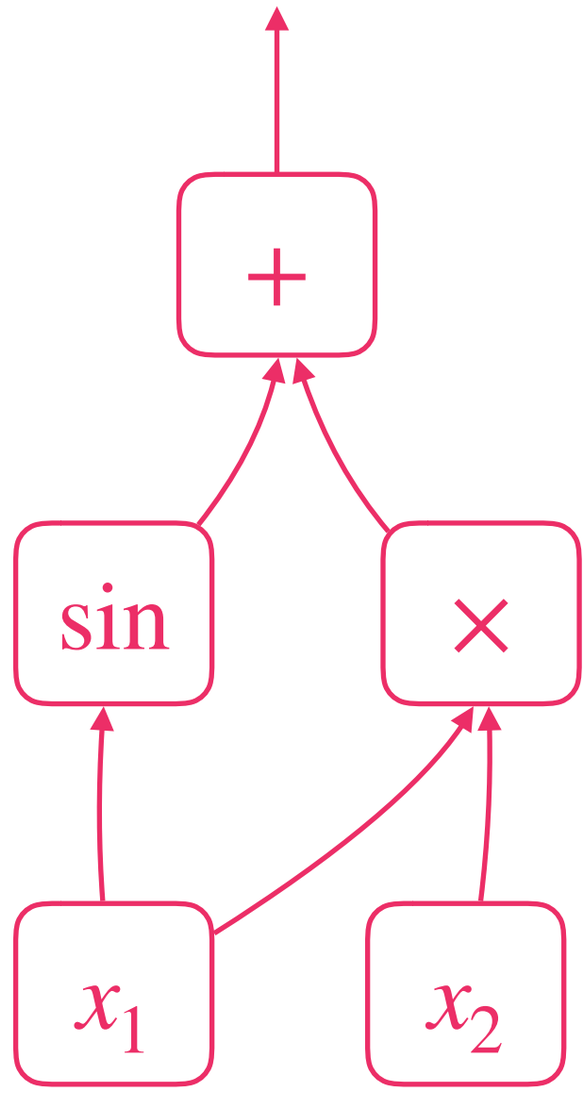
- Deep learning frameworks take care of this for you; no need to write complex expressions.

- …But, since this is an algorithms class, let's look at the mechanics, which are quite interesting!



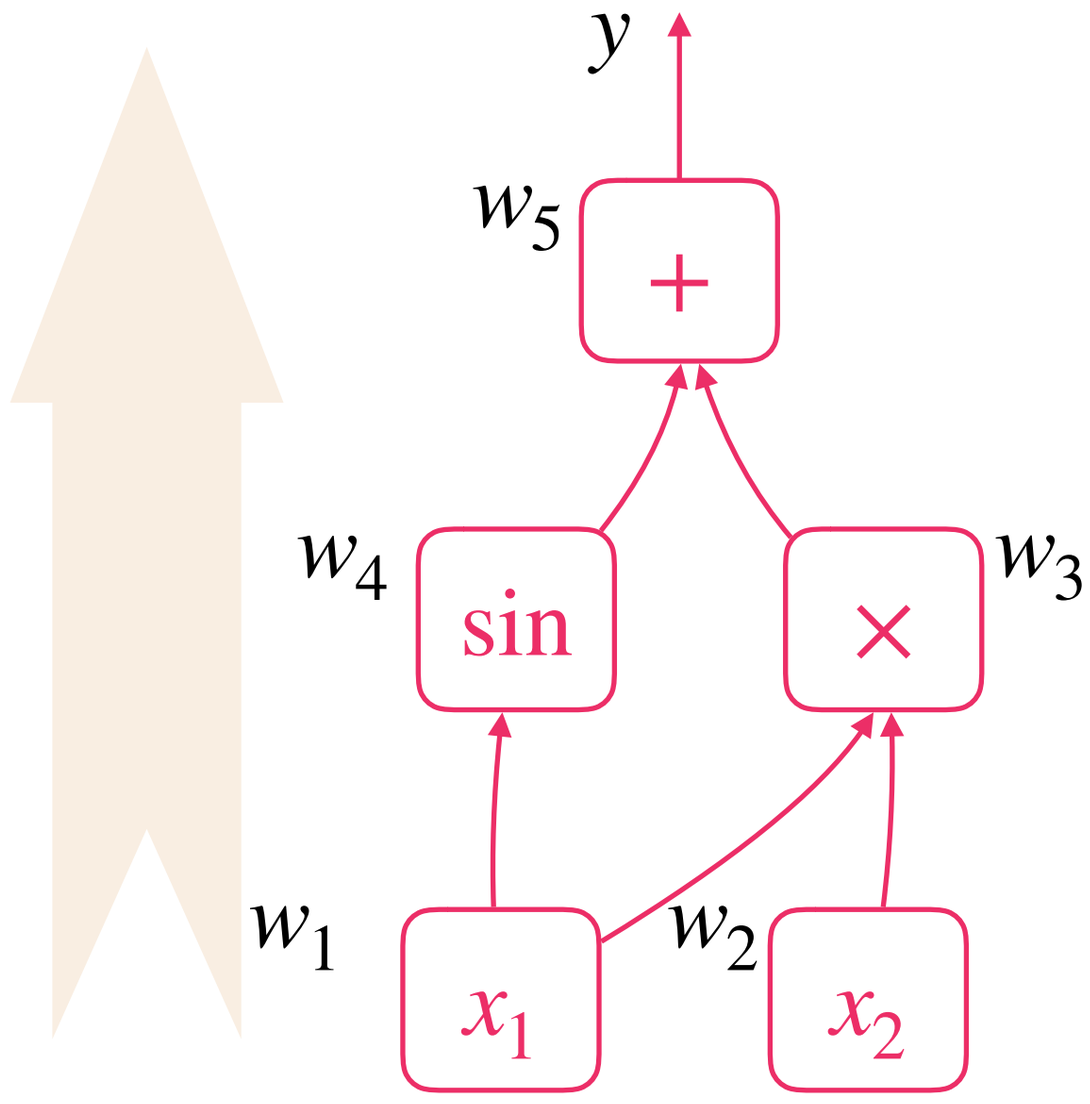
Reverse-accumulation automatic differentiation

$$f(x_1, x_2) = \sin x_1 + x_1 x_2$$



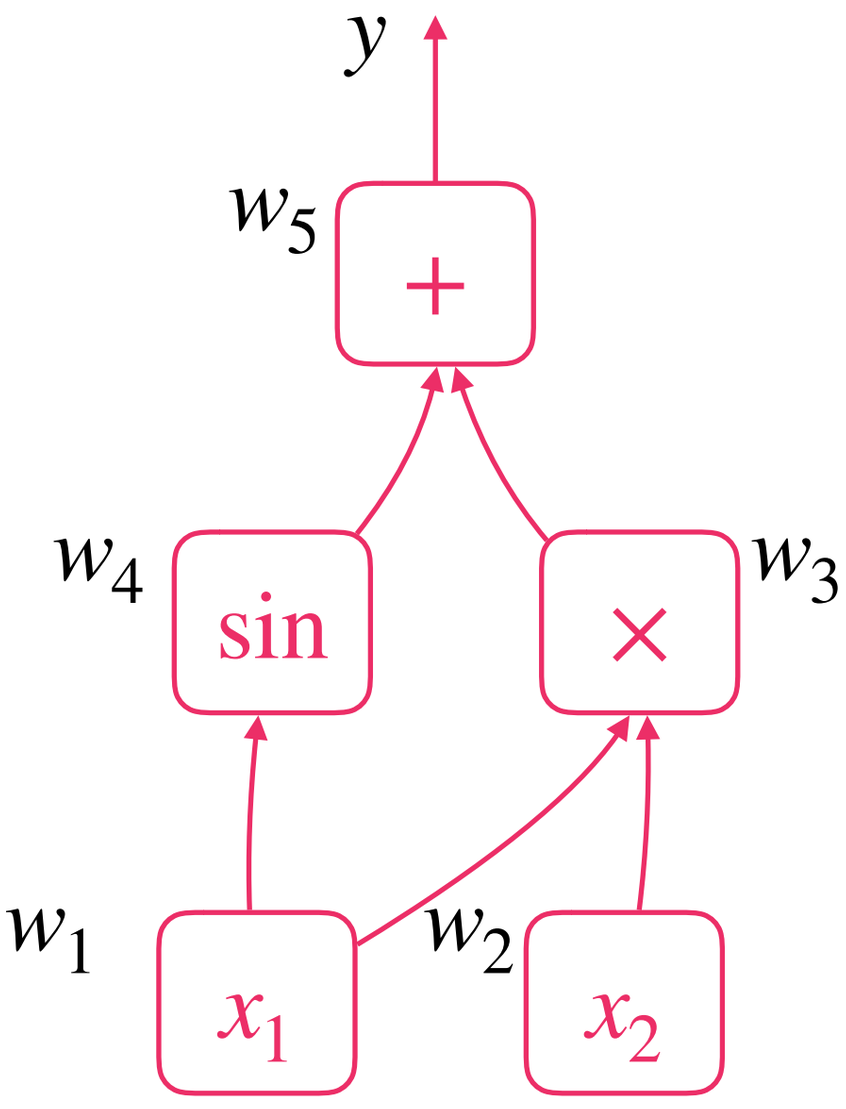
Reverse-accumulation automatic differentiation

$$
\begin{aligned}
w_1 &= x_1 \\
w_2 &= x_2 \\
w_3 &= w_1 w_2 \\
w_4 &= \sin w_1 \\
w_5 &= w_4 + w_3 \\
f(x_1, x_2) &= w_5 \\
f(x_1, x_2) &= \sin x_1 + x_1 x_2
\end{aligned}
$$



Reverse-accumulation automatic differentiation

$$\frac{\partial y}{\partial x} = ?$$



Reverse-accumulation automatic differentiation

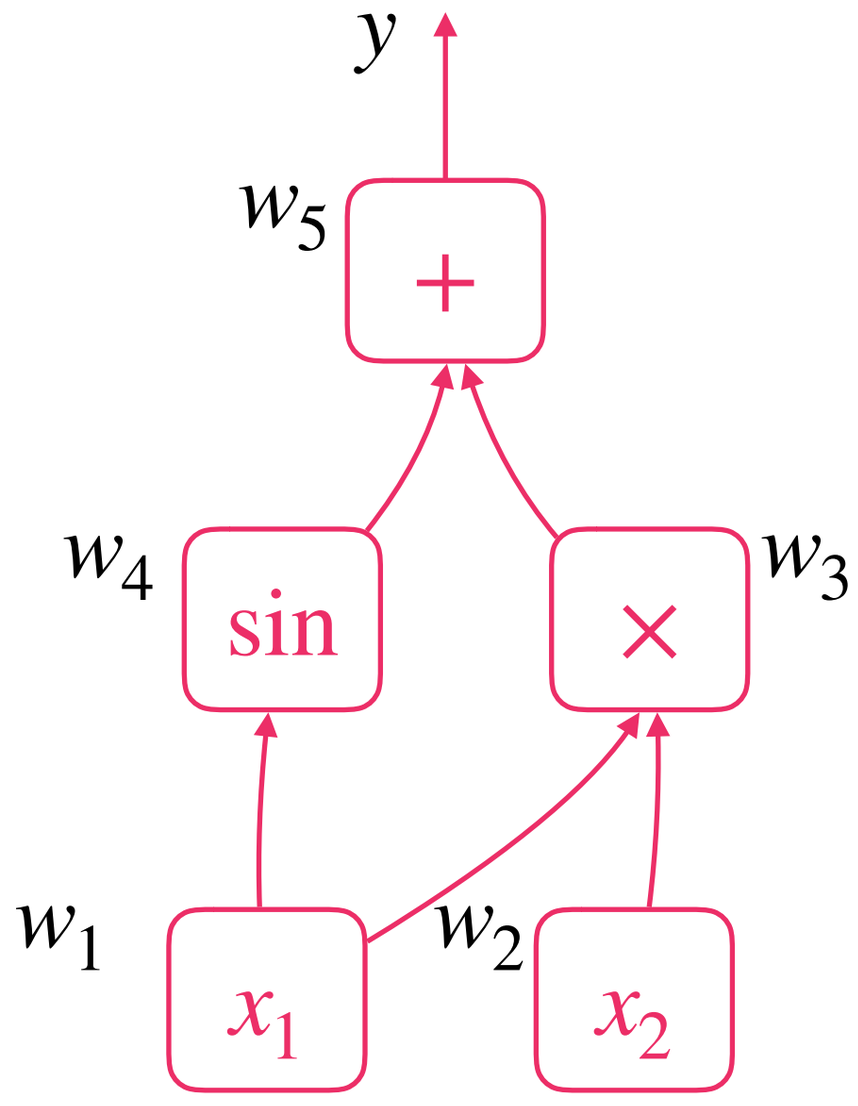
Use the chain rule!

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial x}$$

$$= \frac{\partial y}{\partial u}\frac{\partial u}{\partial v}\frac{\partial v}{\partial x}$$

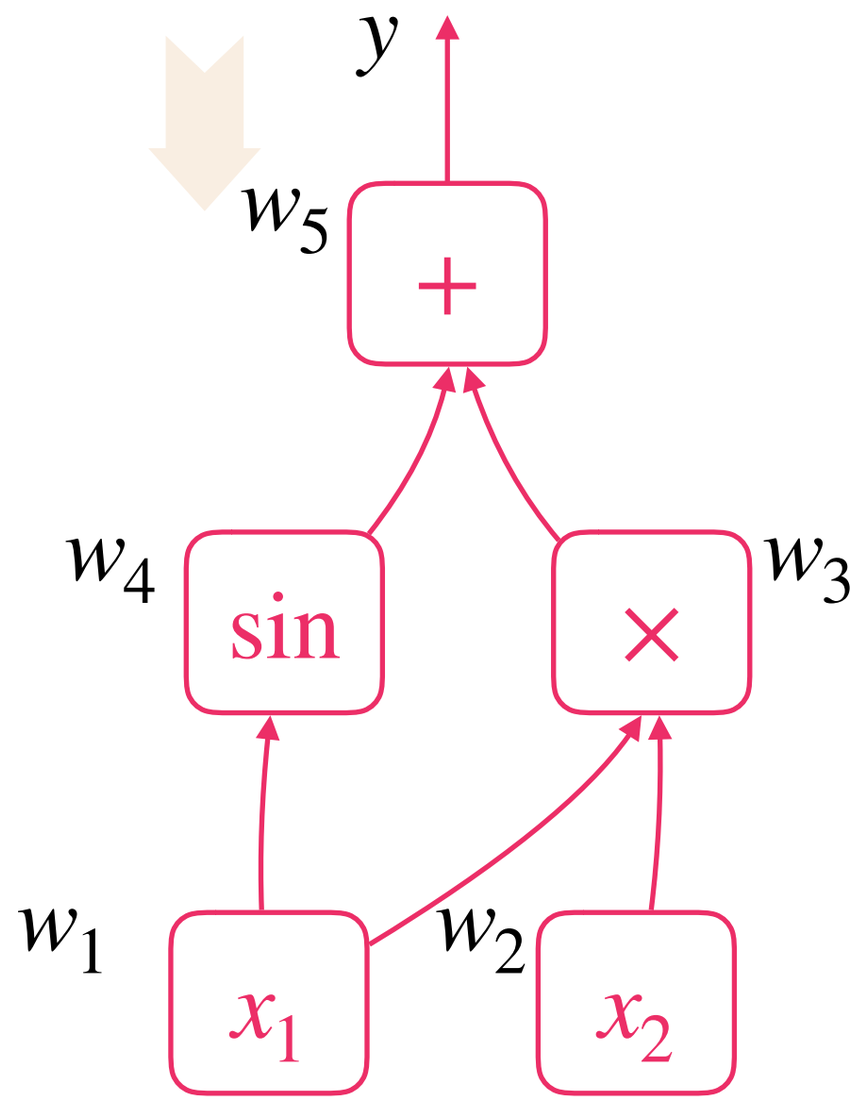
$$\vdots$$

In automatic differentiation, we start from the **output** and work **backwards** through each subexpression…



Reverse-accumulation automatic differentiation

$$\frac{\partial y}{\partial w_5} = 1$$

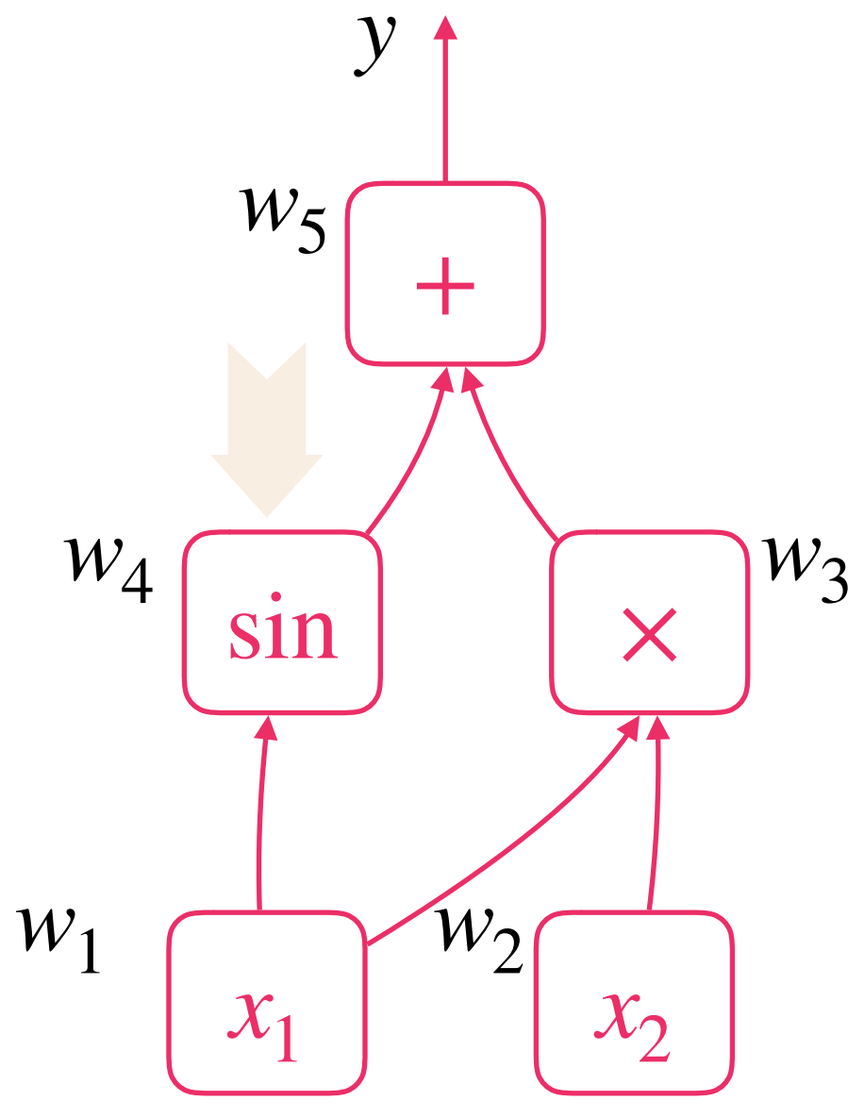


Reverse-accumulation automatic differentiation

$$\frac{\partial y}{\partial w_4} = \frac{\partial y}{\partial w_5}\frac{\partial w_5}{\partial w_4}$$

$$= \frac{\partial y}{\partial w_5}\frac{\partial (w_4 + w_3)}{\partial w_4}$$

$$= \frac{\partial y}{\partial w_5} \cdot 1$$

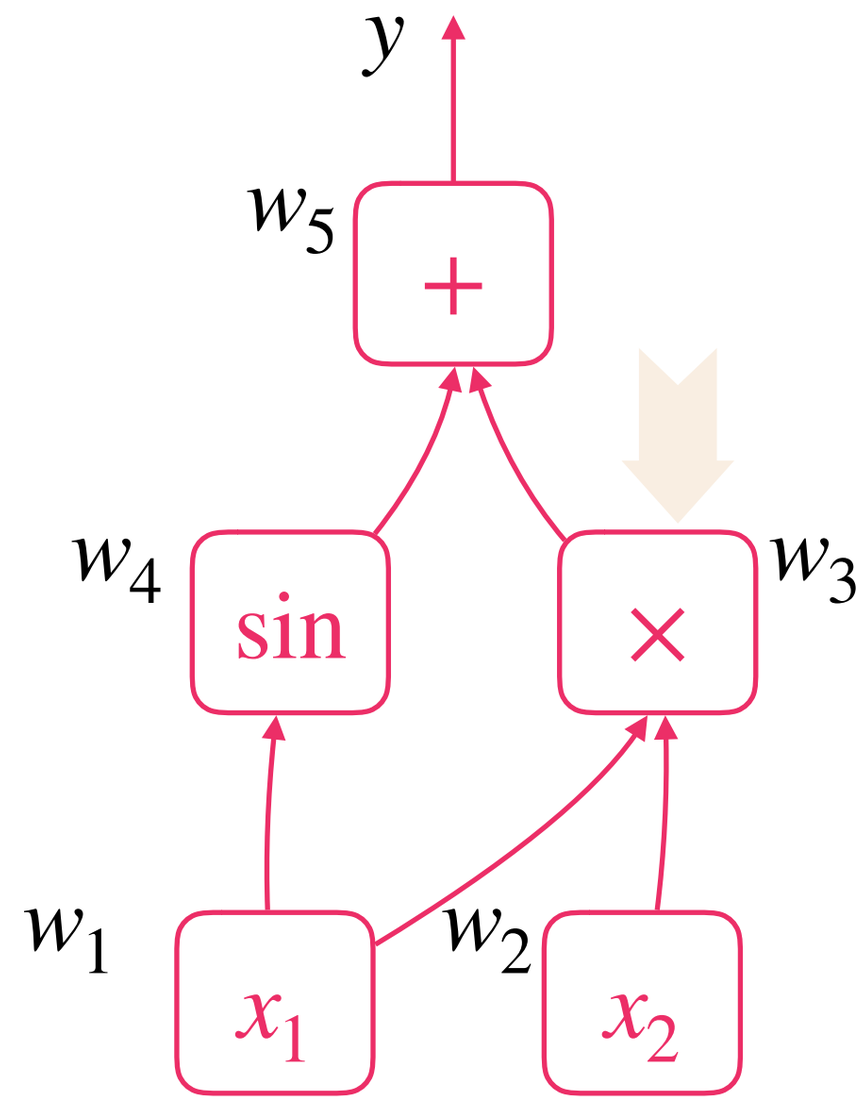


Reverse-accumulation automatic differentiation

$$\frac{\partial y}{\partial w_3} = \frac{\partial y}{\partial w_5}\frac{\partial w_5}{\partial w_3}$$

$$= \frac{\partial y}{\partial w_5}\frac{\partial (w_4 + w_3)}{\partial w_3}$$

$$= \frac{\partial y}{\partial w_5} \cdot 1$$

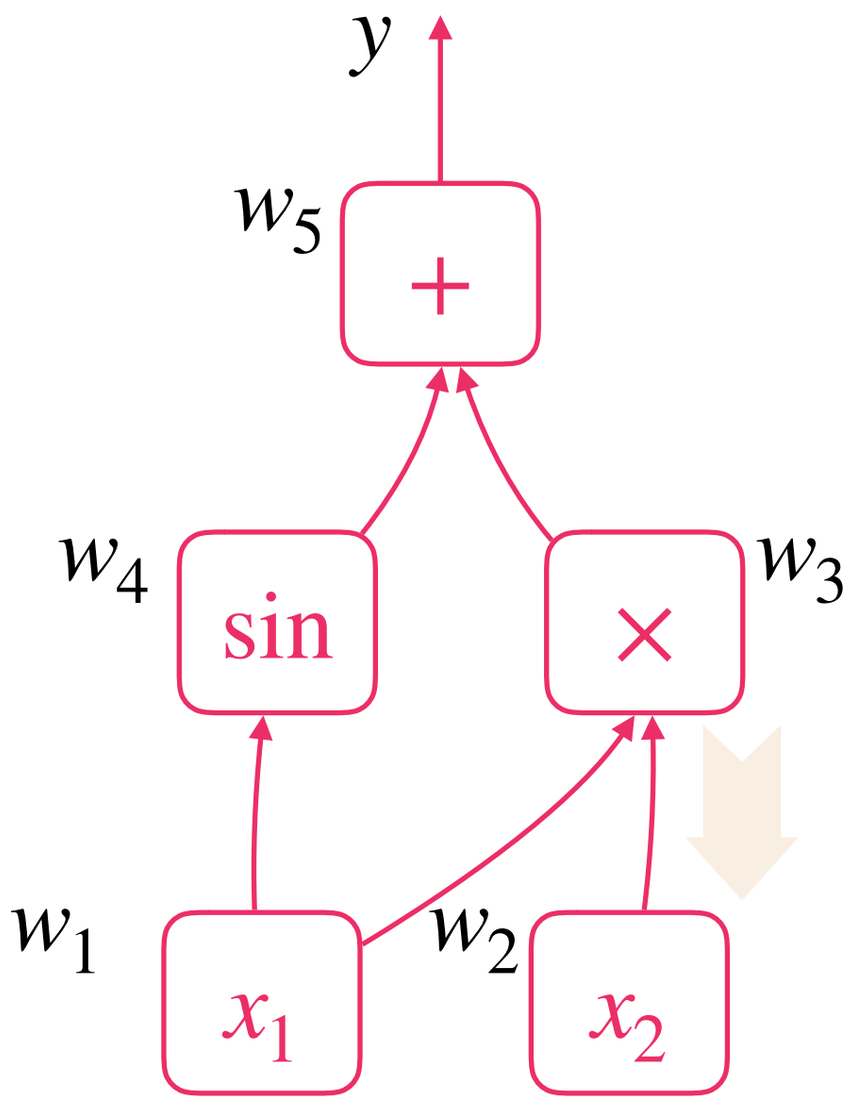


Reverse-accumulation automatic differentiation

$$\frac{\partial y}{\partial w_2} = \frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}$$

$$= \frac{\partial y}{\partial w_3}\frac{\partial (w_1 w_2)}{\partial w_2}$$

$$= \frac{\partial y}{\partial w_3} \cdot w_1$$

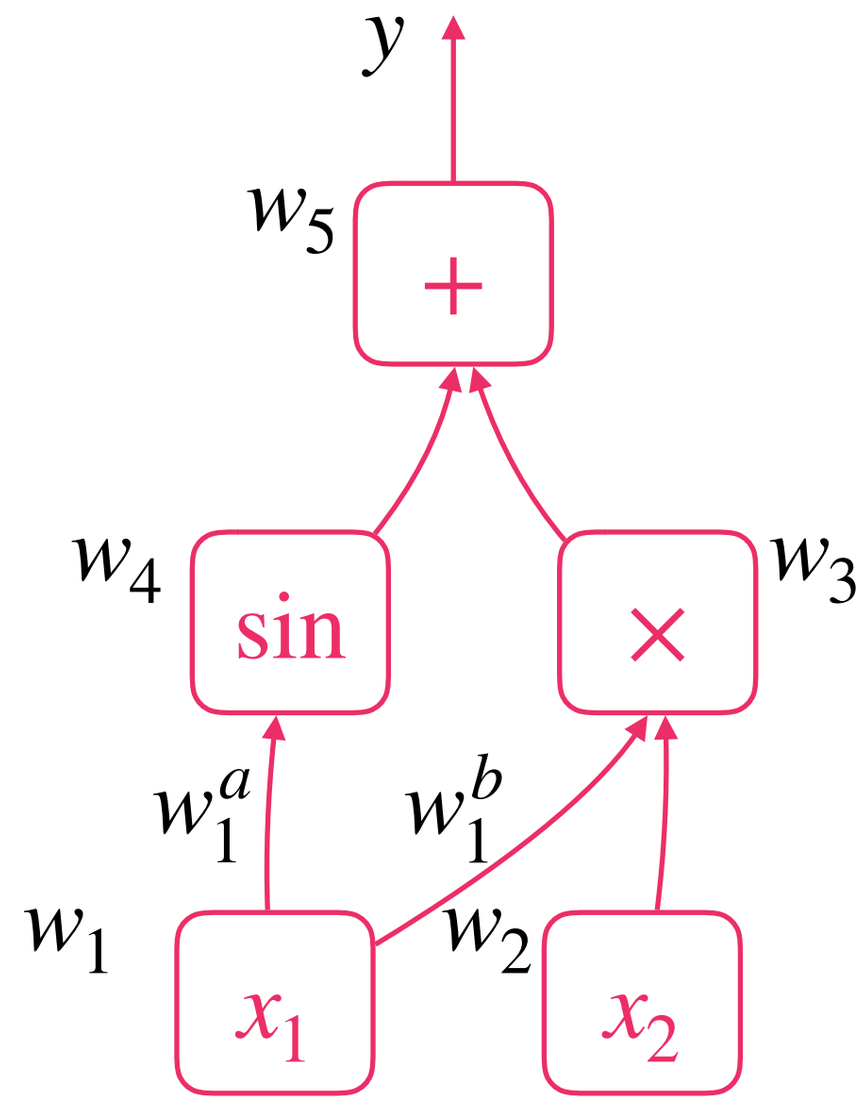


Reverse-accumulation automatic differentiation

$$\frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial w_1^a} + \frac{\partial y}{\partial w_1^b}$$

The **multivariate chain rule** comes into play here, because w_1 is used in two subexpressions, so the total gradient on w_1 must be their sum.

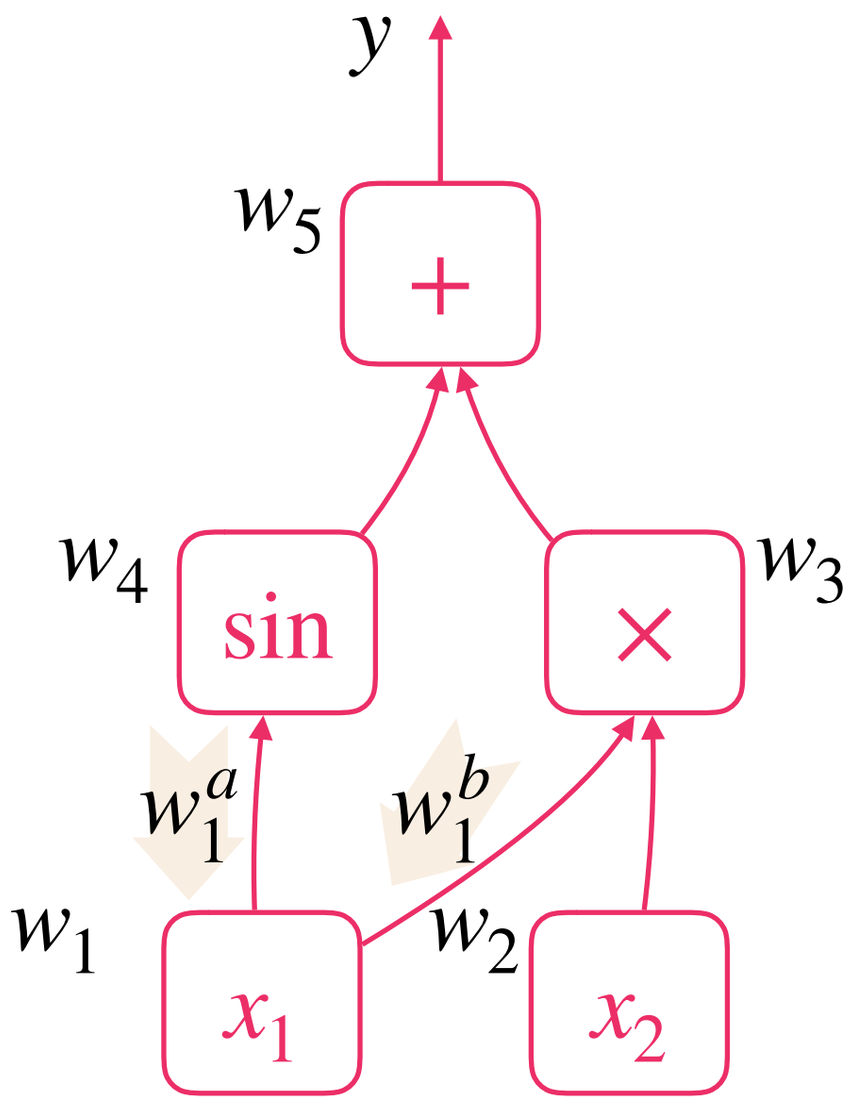
(These are *partial* derivatives!)



Reverse-accumulation automatic differentiation

$$\frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial w_1^a} + \frac{\partial y}{\partial w_1^b}$$

$$= \frac{\partial y}{\partial w_4} \cos w_1 + \frac{\partial y}{\partial w_3} w_2$$

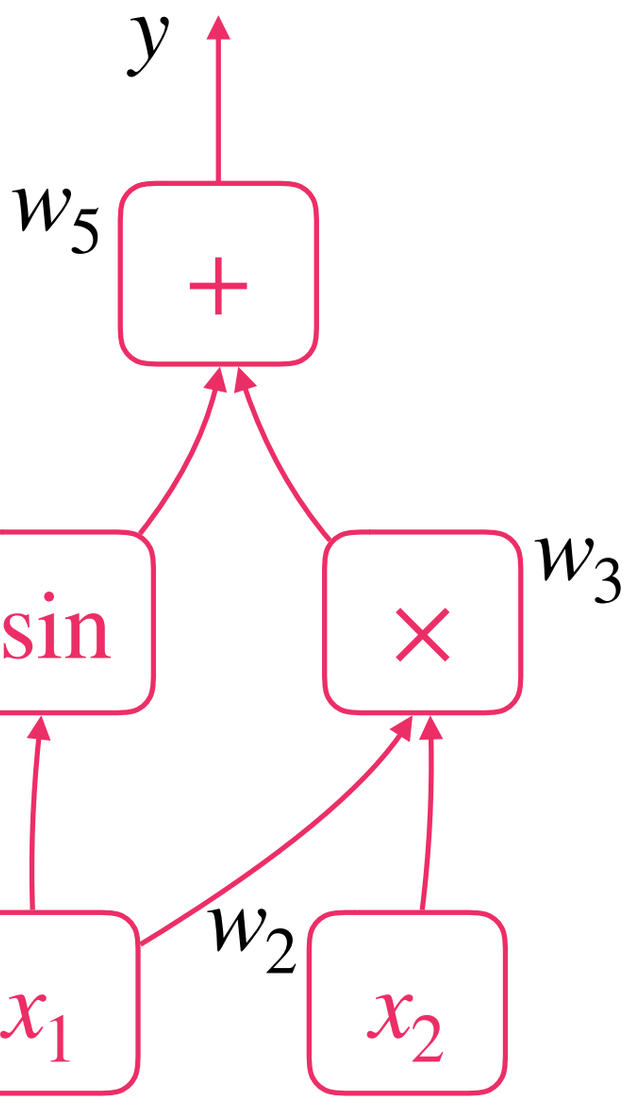


Reverse-accumulation automatic differentiation

Final result:

$$\frac{\partial y}{\partial x_2} = x_1$$

$$\frac{\partial y}{\partial x_1} = x_2 + \cos x_1$$



Reverse-accumulation automatic differentiation

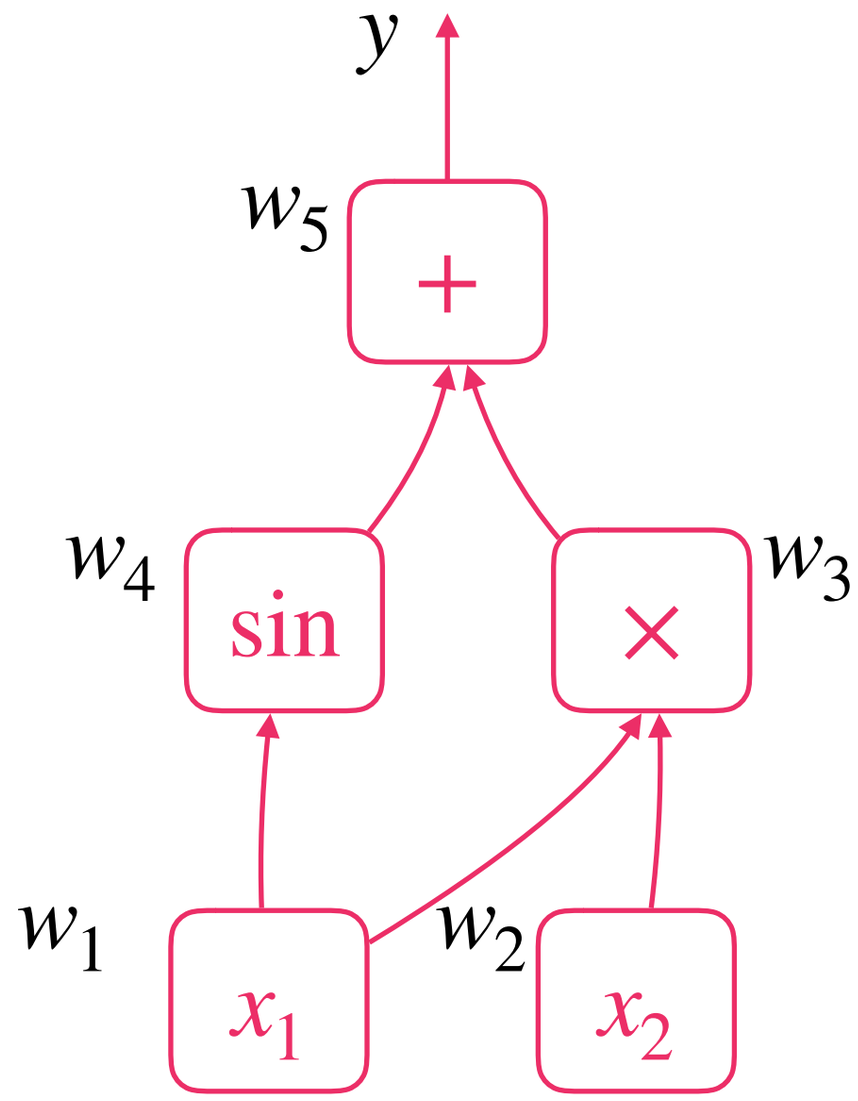
Automatic differentiation is efficient!

For each layer, we only need to

1. Accumulate the gradients of its output layers,

2. Compute the gradient at this layer,

3. Multiply the results together



Reverse-accumulation automatic differentiation

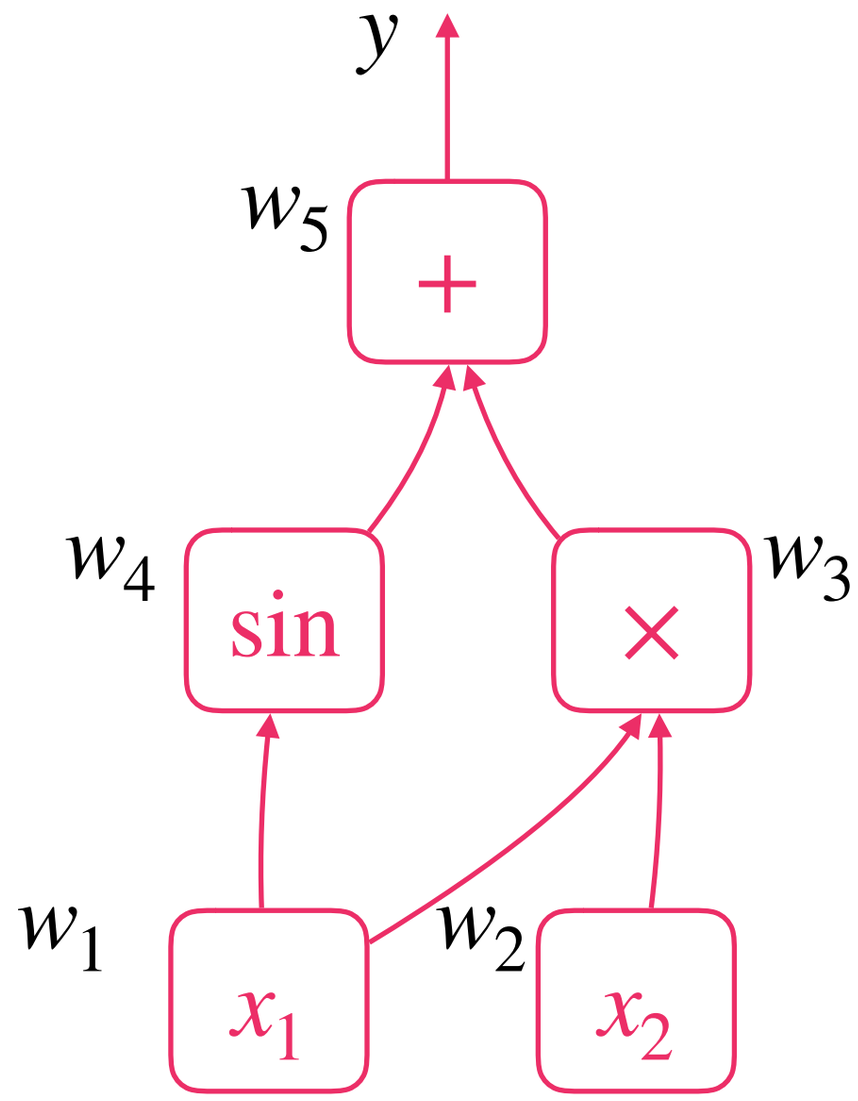
Automatic differentiation is efficient!

For each layer, we only need to

1. Accumulate the gradients of its output layers,

2. Compute the gradient at this layer,

3. Multiply the results together

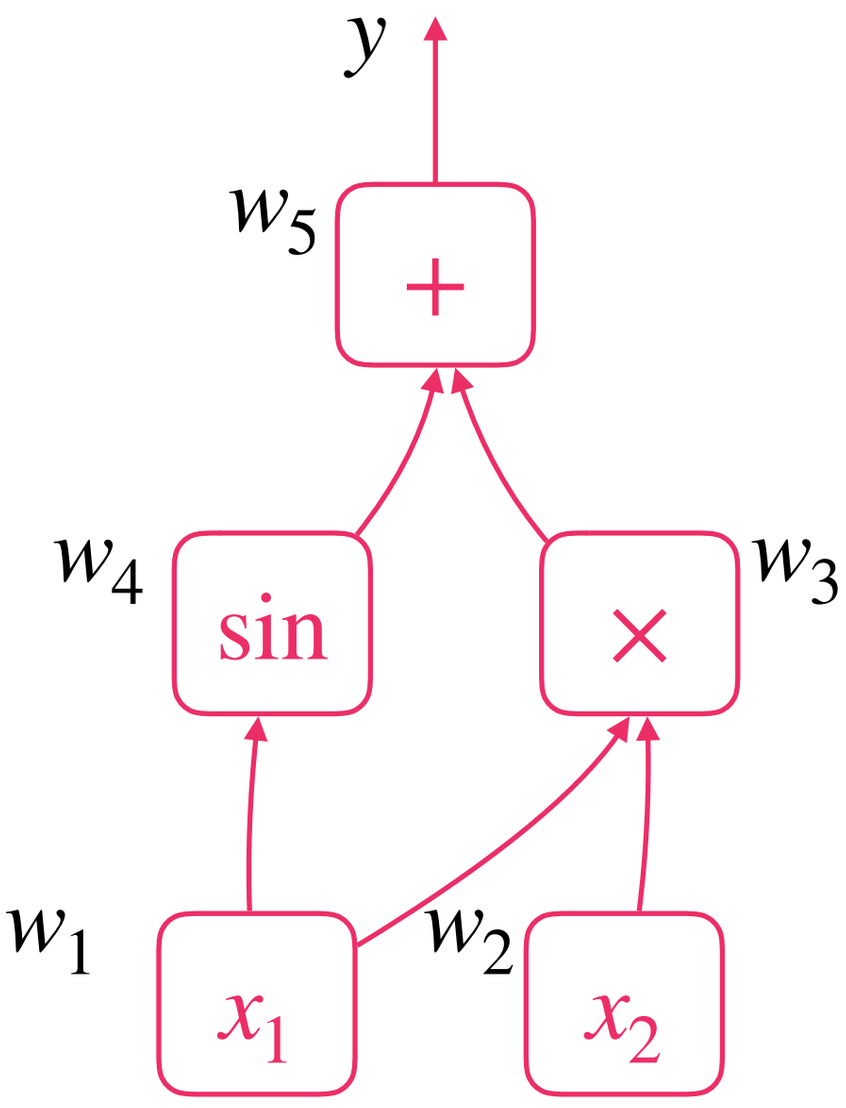


Reverse-accumulation automatic differentiation

Automatic differentiation is efficient!

For each layer, we only need to

1. Accumulate the gradients of its output layers,
2. Compute the gradient at this layer,
3. Multiply the results together



Layer writers only need to worry about this bit!

In practice, all of that boils down to…

- `train_step = \`
  `tf.train.GradientDescentOptimizer(lr) \`
  `.minimize(cross_entropy)`
  which adds optimization operation to computation graph

- TensorFlow graph nodes have attached gradient operations

- Gradient with respect to parameters computed with backpropagation … automatically!

Implementation in the framework: just add more nodes!

Implementations in practice:

- In this approach backpropagation never accesses any numerical values
- Instead it just adds nodes to the graph that describe how to compute derivatives
- A graph evaluation engine will then do the actual computation
- Approach taken by Theano and TensorFlow

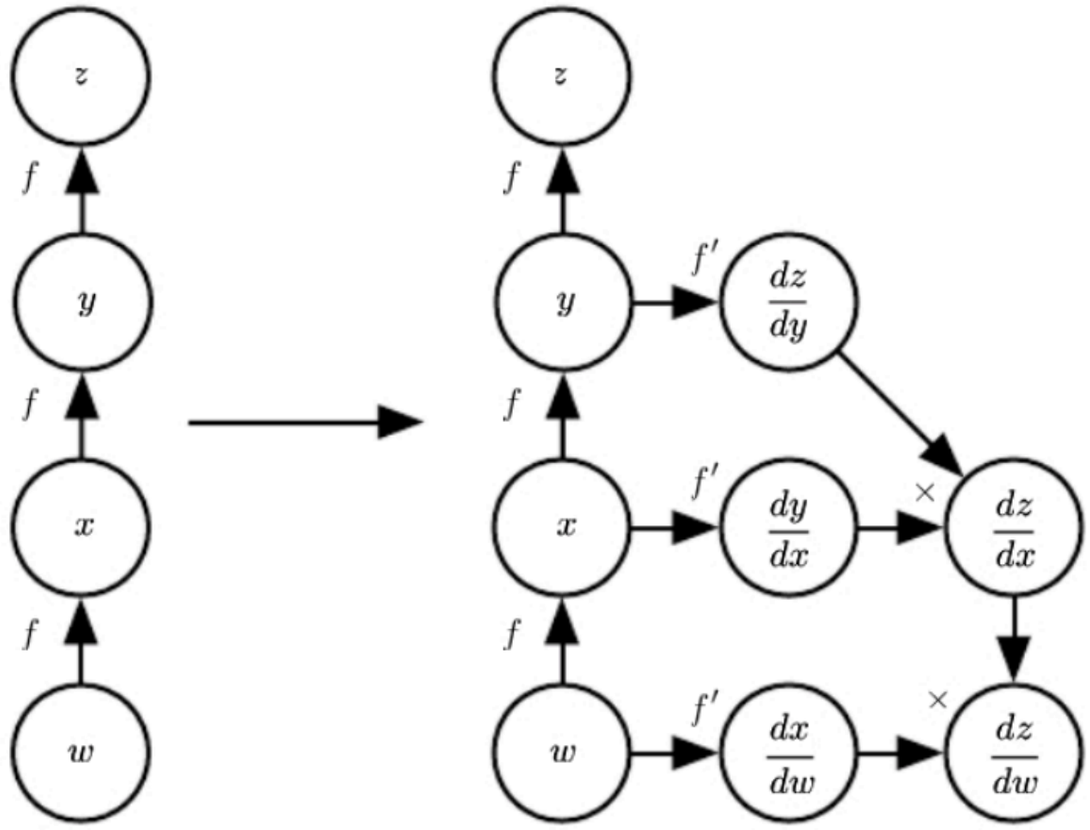


Figure: Goodfellow *et al.*

Slide credit: Shubhendu Trivedi & Risi Kondor https://ttic.uchicago.edu/~shubhendu/Pages/Files/Lecture4_flat.pdf

Questions + discussion!!

Many slides are the work of Ali Ghodsi and Ion Stoica, https://ucbrise.github.io/cs262a-spring2018/notes/24-TensorFlow-Clipper.pdf