جامعة قطر
QATAR UNIVERSITY

CMPS 460 – Spring 2022

# MACHINE LEARNING

**Tamer Elsayed**

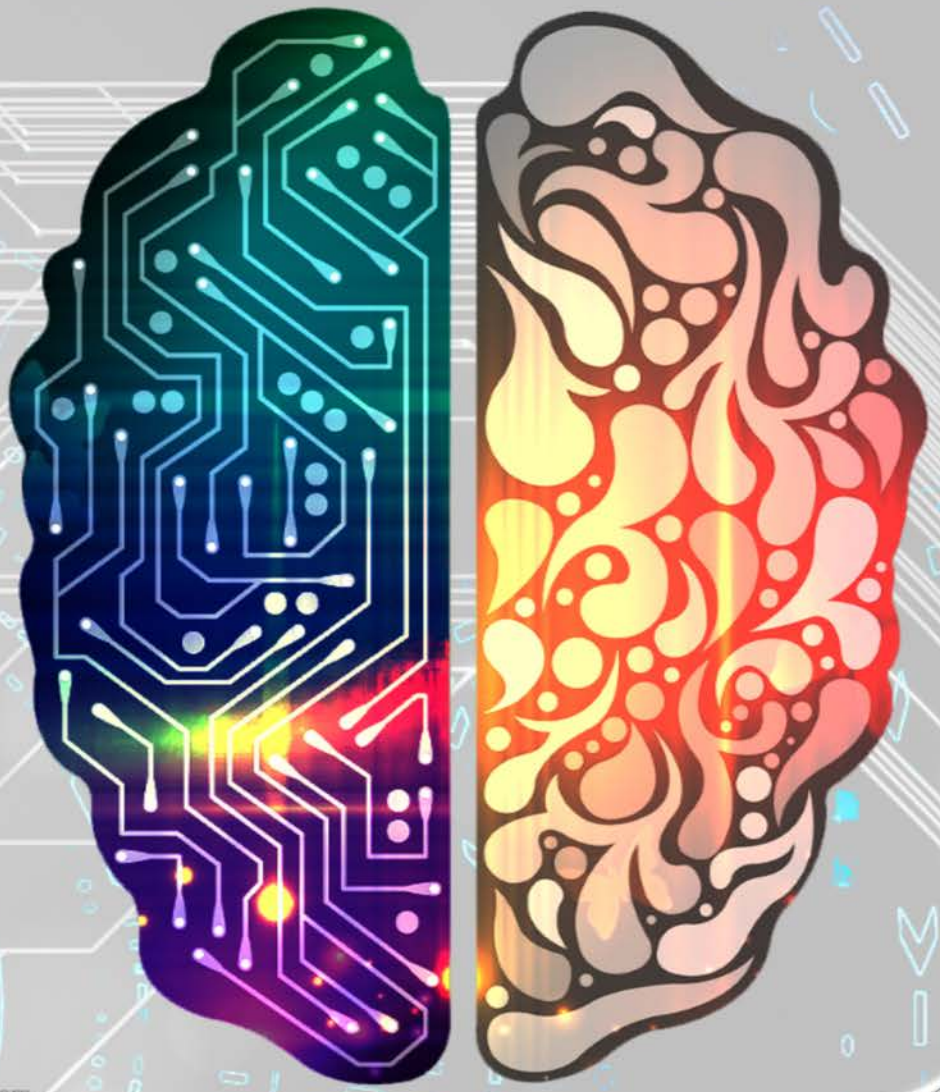


Image hosted by: WittySparks.com | Image source: Pixabay.com

9.a

## Neural Networks I

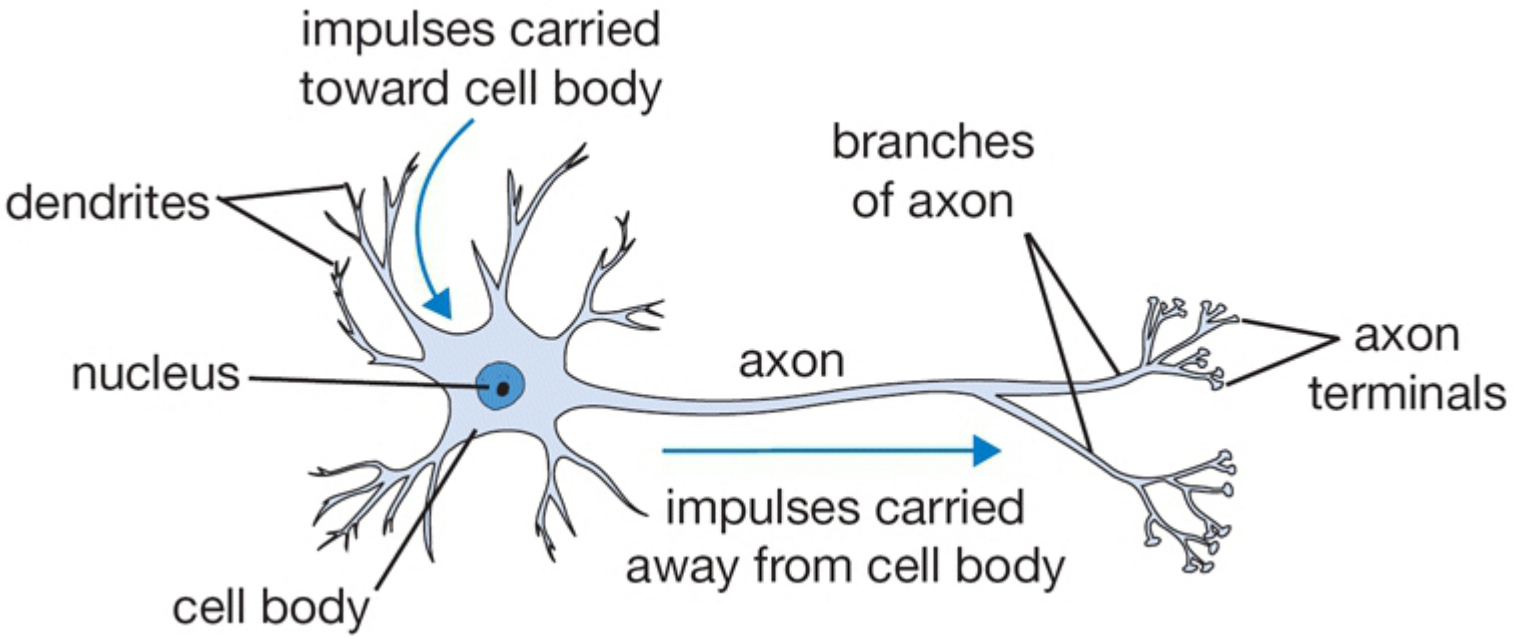
Sec 10.1, 10.5 + Handouts

# Roadmap …

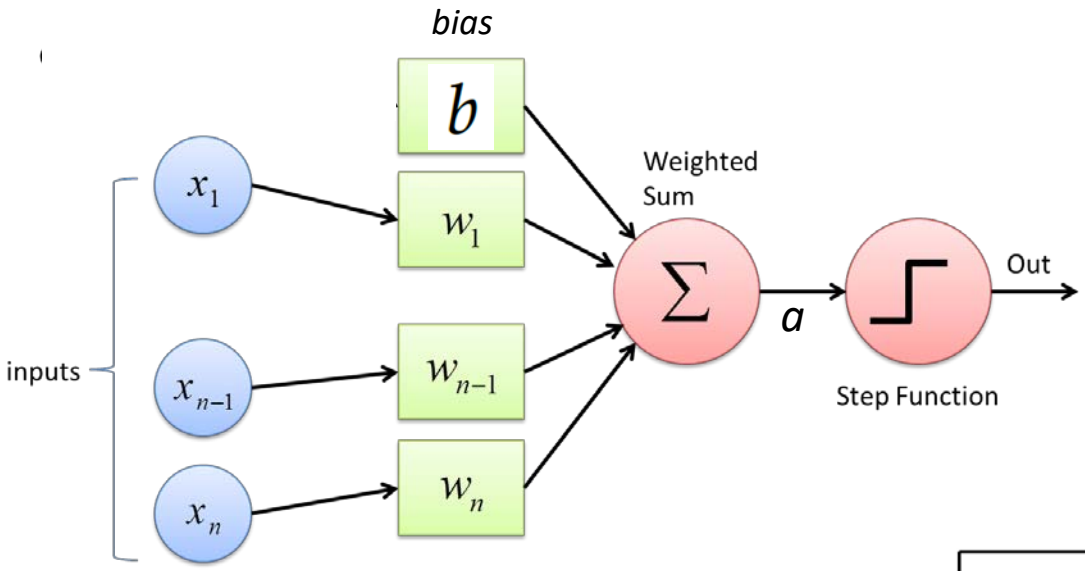
- What are Neural Networks?
- How to make a prediction given an input?
- Are neural networks really powerful?
- What is Backpropagation?

# Recall: Perceptron

# Biological Inspiration: A Neuron …

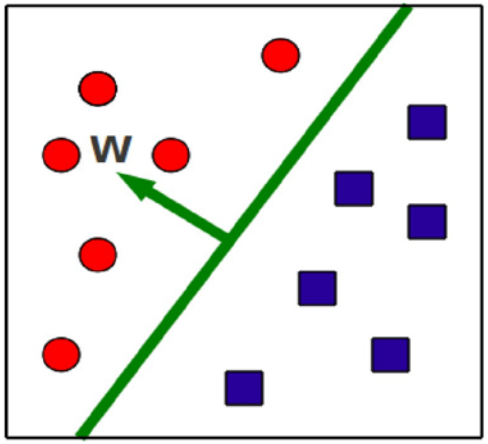


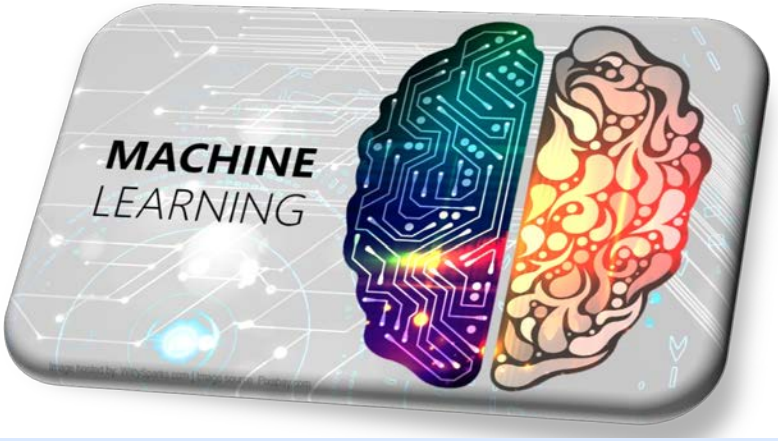
# Perceptron as a Neuron




$$a = \left[\sum_{d=1}^{D} w_d x_d\right] + b$$

$$\hat{y} = \boldsymbol{sign}(\boldsymbol{a})$$

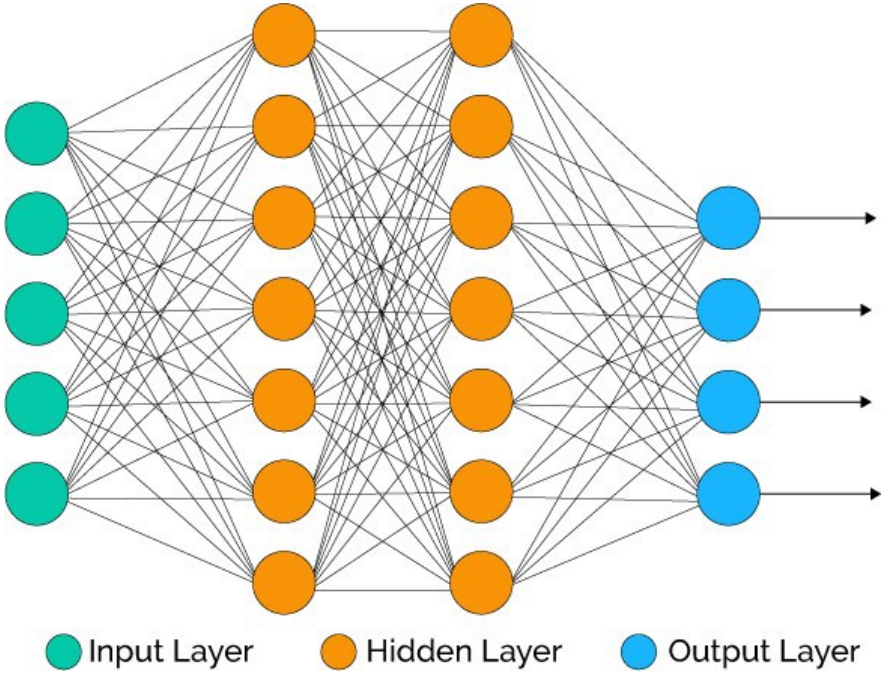





# Neural Networks

# Neural Networks

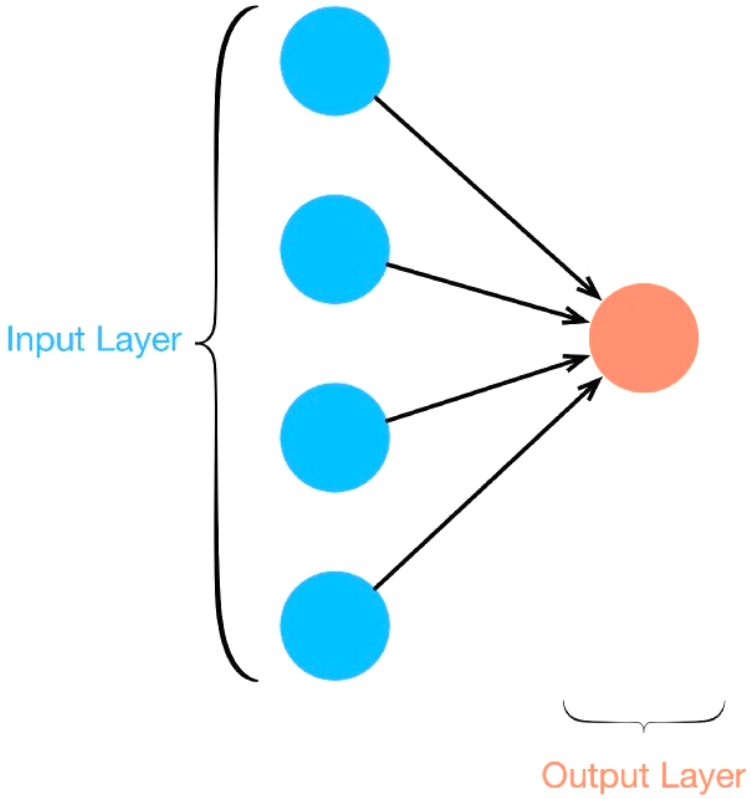
- Combination of multiple perceptrons
  - Multilayer Perceptron (MLP)
- Outputs of some neurons become inputs to other neurons.
- Often organized into distinct layers of neurons.



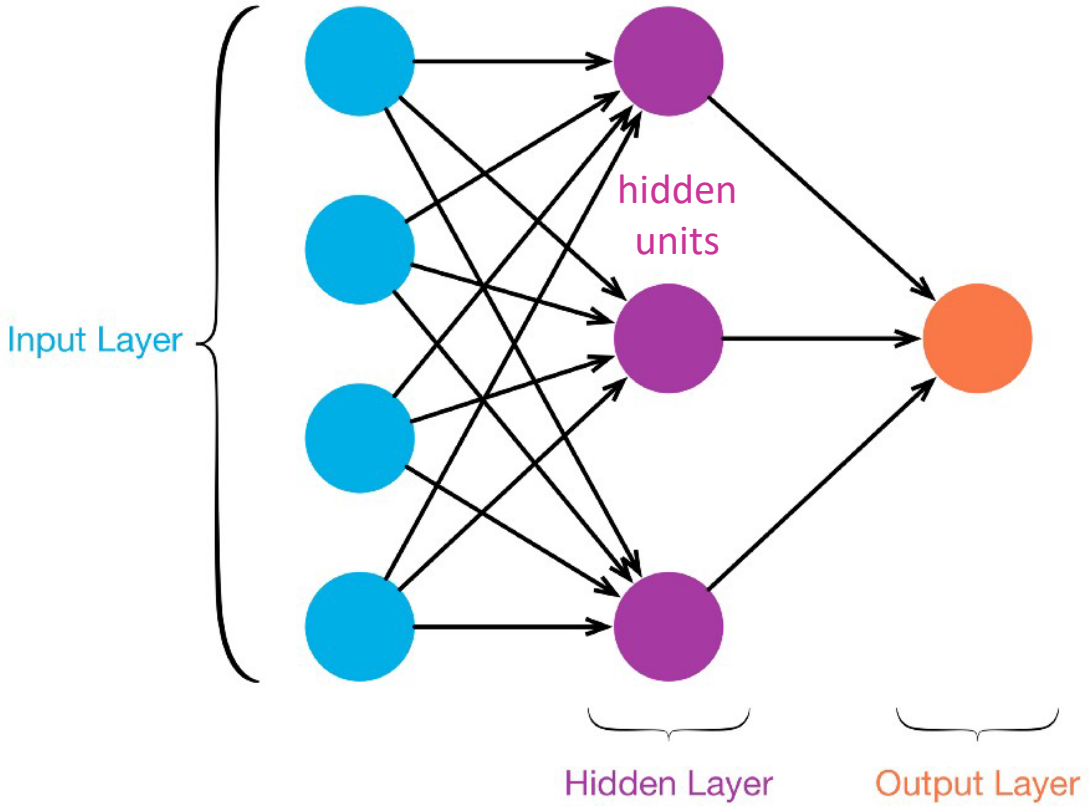

***Why would we want to do that?***

Discover more complex decision boundaries!

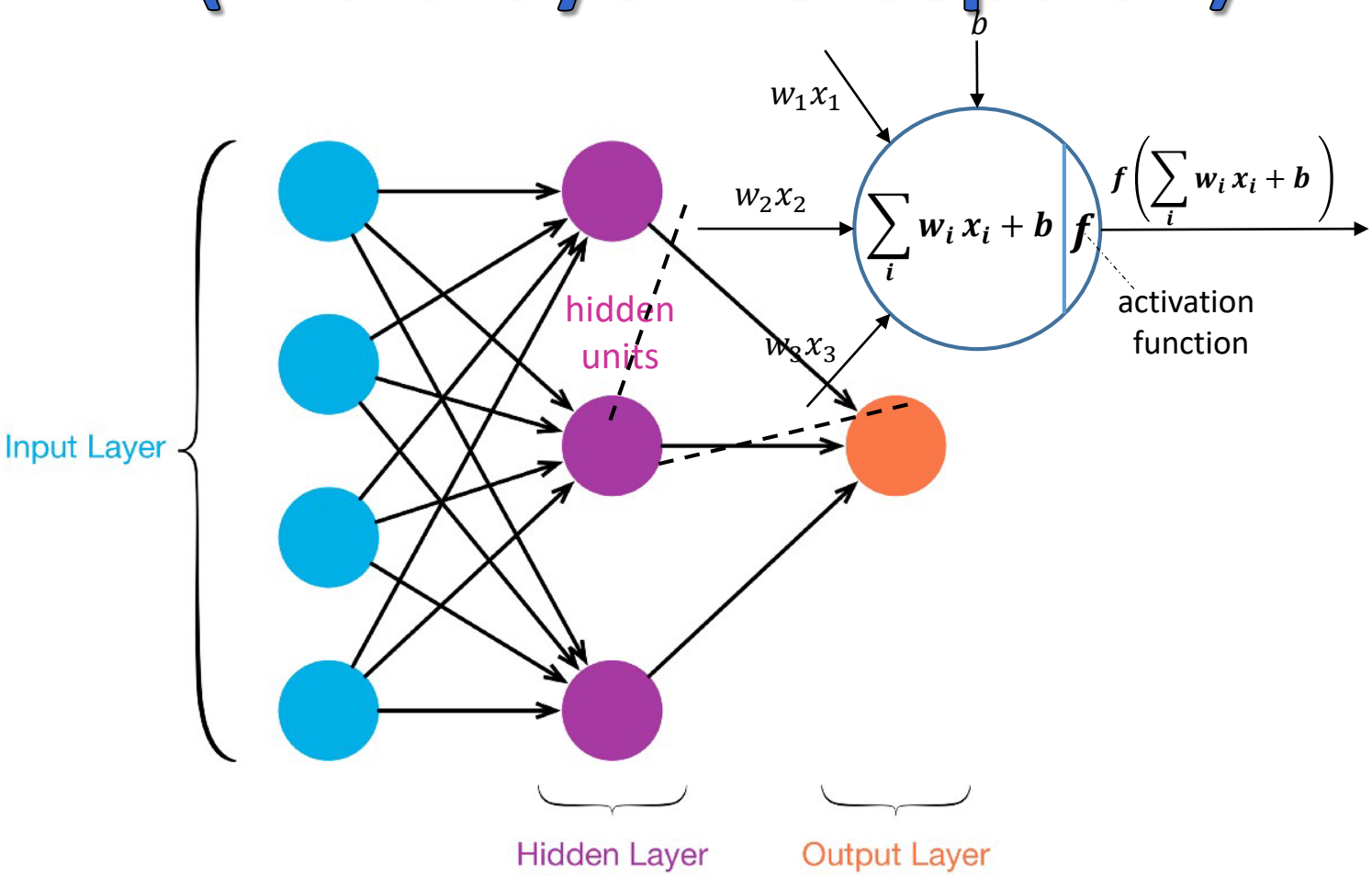
# 1-Layer NN (Perceptron)



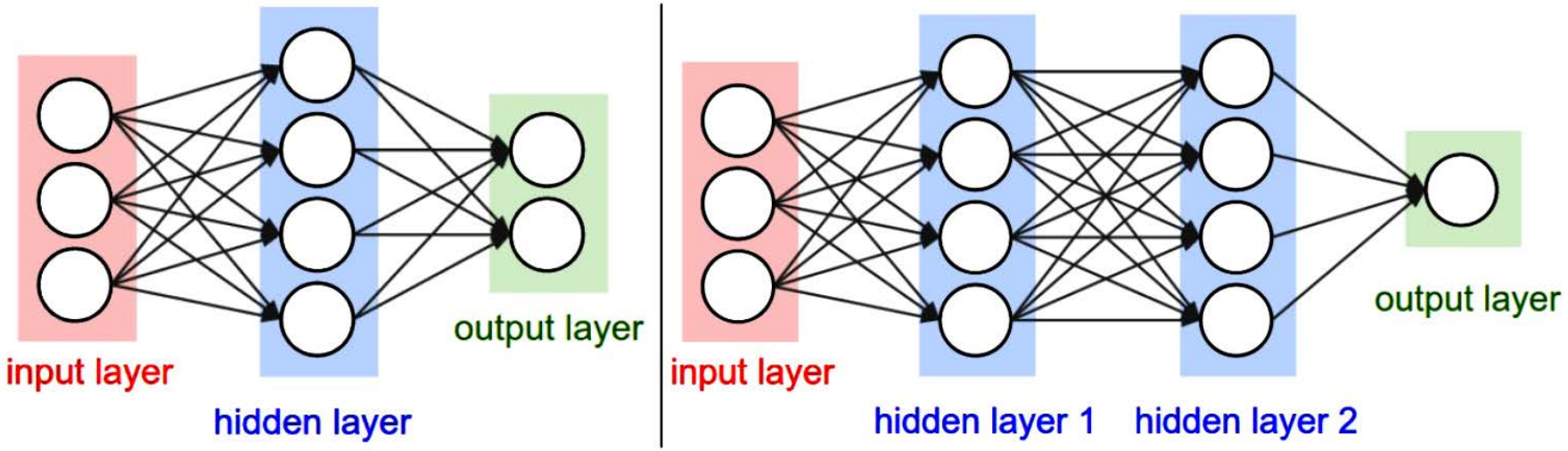
# 2-Layer NN (Multilayer Perceptron)



# 2-Layer NN (Multilayer Perceptron)



# Layer-wise organization




- Most common layer type: **fully-connected**
- Neurons within a single layer share no connections
- Output layer neurons most commonly do not have an activation function
  - usually represent class scores
- NN Size: number of neurons or number of parameters.

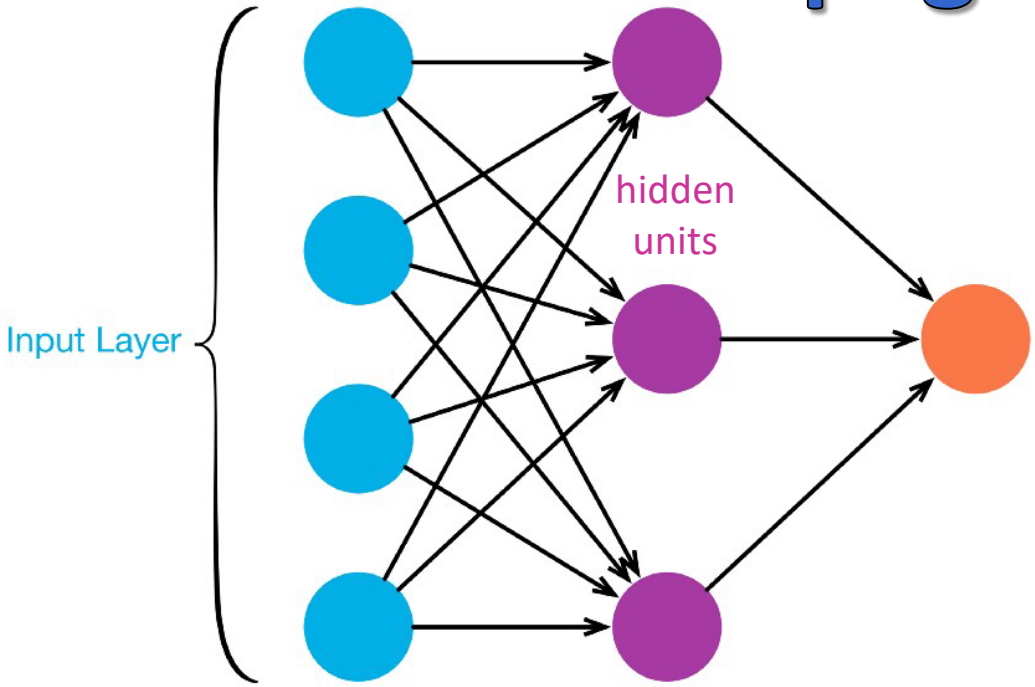
# Activation functions

(aka link functions)

- Activation functions are non-linear functions
  - e.g., sign/step function, as in the perceptron
  - e.g., hyperbolic tangent (tanh) and other sigmoid functions that approximate sign but are differentiable

*What happens if the hidden units use the identity function as an activation function?*

# Prediction: Forward Propagation




Matrix of hidden layer parameters

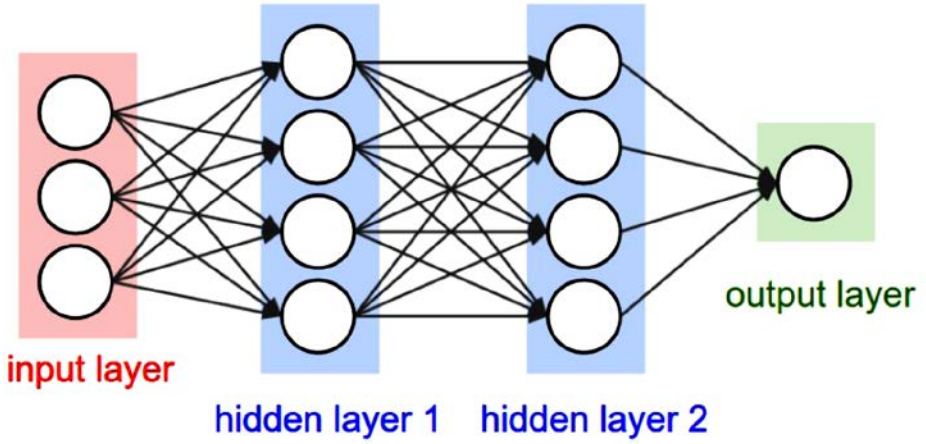
Vector of output layer parameters

**Algorithm 24** TwoLayerNetworkPredict($\mathbf{W}$, $v$, $\hat{x}$)

1: **for** $i = 1$ **to** *number of hidden units* **do**
2: $h_i \leftarrow \tanh(w_i \cdot \hat{x})$ // compute activation of hidden unit $i$
3: **end for**
4: **return** $v \cdot h$ // compute output unit

# In practice …

- Structure makes it very simple and efficient to use matrix vector operations
  - interwoven with activation function.




```
# forward-pass of a 3-layer neural network:
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

- W1, W2, W3, b1, b2, b3 are the learnable parameters
- x could hold an entire batch of training data

# Representational Power

# Representational Power

***What is the representational power of this family of functions?***

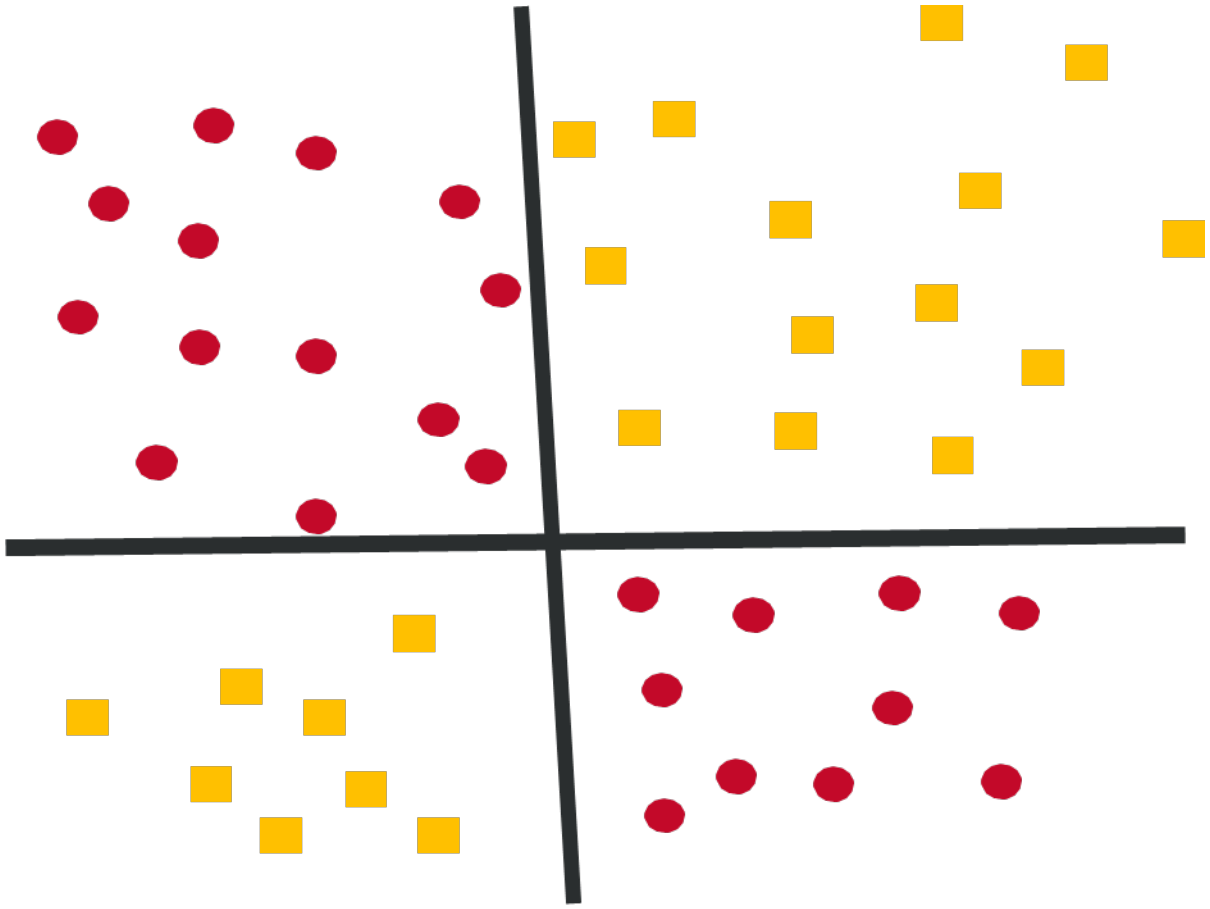
***Are there functions that cannot be modeled with a Neural Network?***

# Two-Layer NNs are Universal Function Approximators!

# Theorem

Let $F$ be a continuous function on a bounded subset of D- dimensional space. Then there exists a ***two-layer neural network*** $\hat{F}$ with a finite number of hidden units that approximates $F$ arbitrarily well. Namely, for all $x$ in the domain of $F$, $\left|F(x) - \hat{F}(x)\right| < \epsilon$

# Can Solve XOR Problem!



***if two-layer networks are so great, why do we care about deeper networks?***

# Breadth versus Depth

- There are functions that require a huge number of hidden units with shallow network, but can be done in a small number of units if deep.
- Ex: $$parity(\boldsymbol{x}) = \sum_{d} x_d \mod 2$$
$$= \begin{cases} 1 & \text{if the number of 1s in } \boldsymbol{x} \text{ is odd} \\ 0 & \text{if the number of 1s in } \boldsymbol{x} \text{ is even} \end{cases}$$
- "Constant-depth circuits are less powerful than deep circuits."
- **Number of parameters:** a deep model could potentially require exponentially fewer examples to train than a shallow model!

***if deep is potentially so much better, why doesn't everyone use deep networks?***

# Why not always deep?

1. You need to choose how many layers, and the width of all those layers!
   - More hyper-parameters to tune!
2. A problem with training with back-propagation

*more later!*

- Finding good ways to train deep networks is an active research area!

# In practice: go deep but not very deep!

- In practice, it is often the case that 3-layer neural networks will outperform 2-layer nets, but going even deeper (4, 5, 6-layer) rarely helps much more.

- This is in stark contrast to Convolutional Networks!
  - images contain hierarchical structure (e.g. faces are made up of eyes, which are made up of edges, etc.), so several layers of processing make intuitive sense.

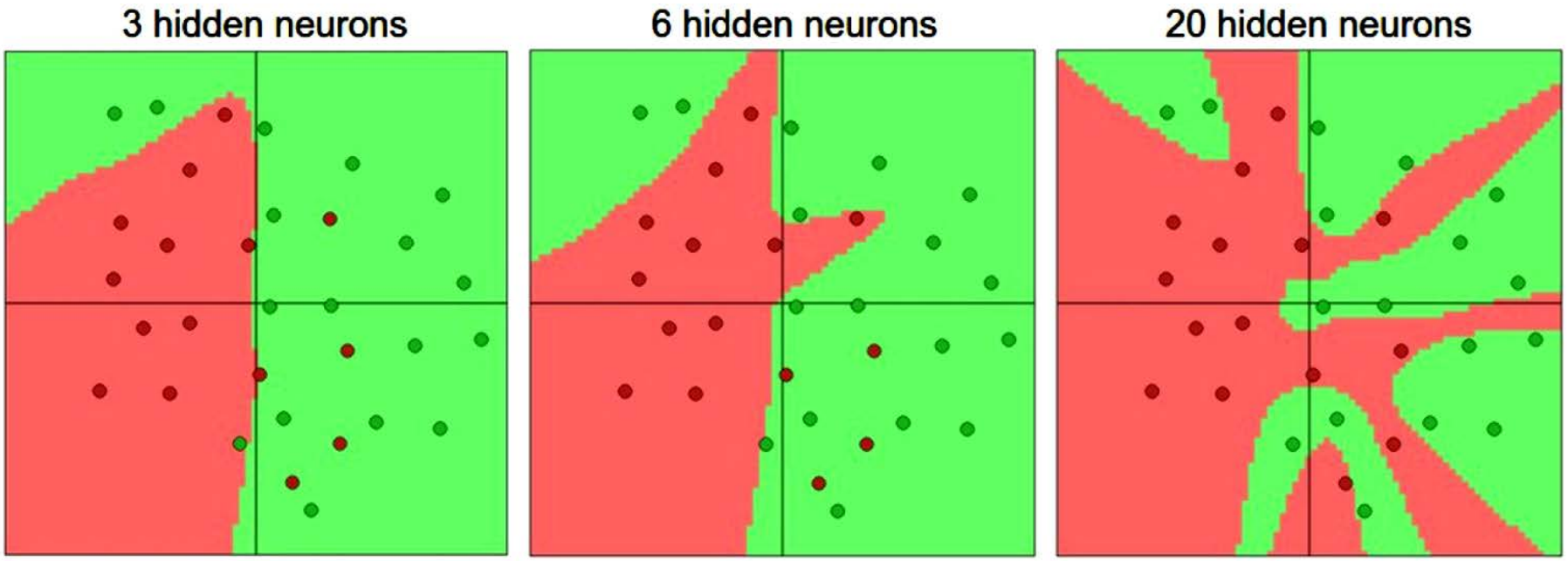
- Topic of much recent research too!

# Setting number of layers and their sizes

- Should we use no hidden layers? One hidden layer? Two hidden layers?
- How large should each layer be?

# Capacity

- As we increase size and number of layers, capacity of the network increases.




***smaller neural networks preferred?***

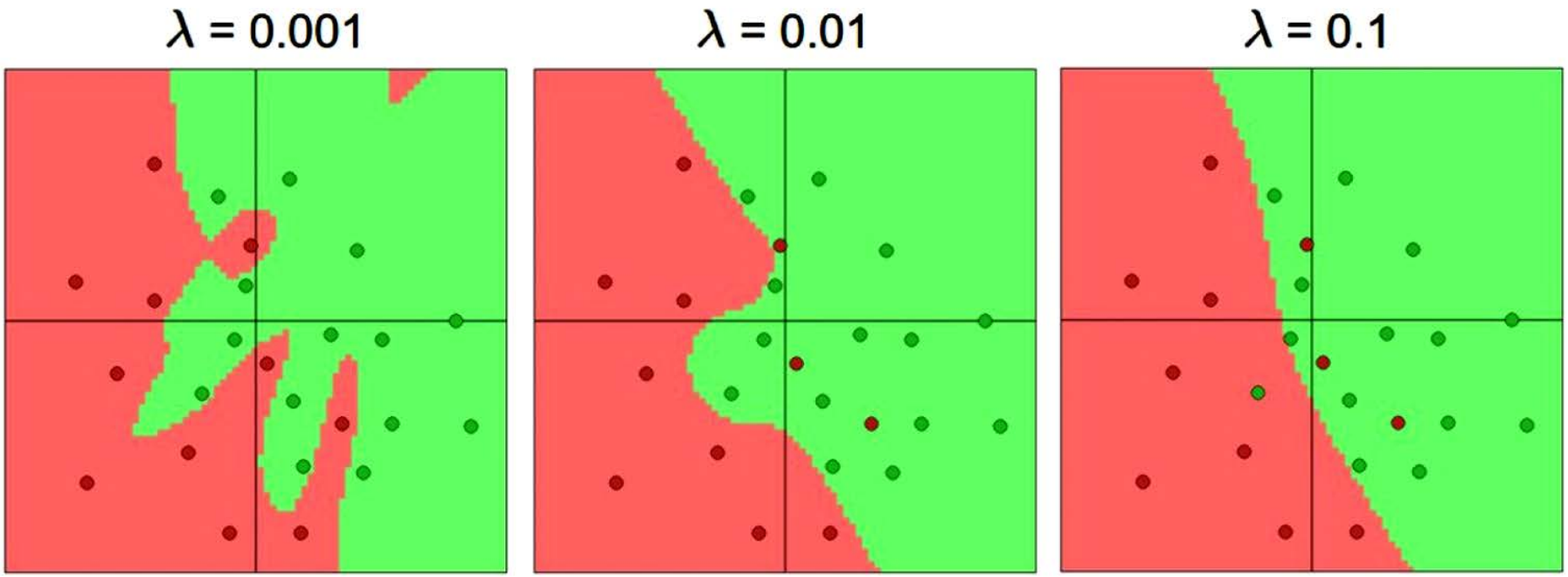
# Smaller NNs preferred?

- The answer is: No!

- Many other preferred ways to prevent overfitting in Neural Networks
  - will discuss later.

- In practice, it is always better to use these methods to control overfitting instead of the number of neurons.

# Ex: Regularization w/20 hidden neurons




*You should not be using smaller networks to avoid overfitting. Instead, use as bigger networks (as budget allows) + regularization.*

# Backpropagation

# Setup

- Given some function $f(x)$ where $x$ is a vector of inputs and we are interested in **computing the gradient of $f$ at $x$ (i.e. $\nabla f(x)$)**

- $f$ will correspond to the loss function ($L$)
- The inputs $x$ will consist of: the training data and the neural network weights.

- In practice, we usually only compute the gradient for the parameters (e.g. $W, b$) so that we can use it to perform a parameter update.

# Interpretation of Gradient

$$f(x,y) = xy \quad \rightarrow \quad \frac{\partial f}{\partial x} = y \quad \frac{\partial f}{\partial y} = x$$

Ex: $x = 4, y = -3$

- Indicate rate of change of a function with respect to that variable:

$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- The derivative on each variable tells the sensitivity of the whole expression on its value.

# Interpretation of Gradient

$$f(x,y) = x + y \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1 \quad \frac{\partial f}{\partial y} = 1$$

$$f(x,y) = \max(x,y) \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1(x \geq y) \quad \frac{\partial f}{\partial y} = 1(y \geq x)$$

# Composite Function: Chain Rule …

$$f(x, y, z) = (x + y)z$$

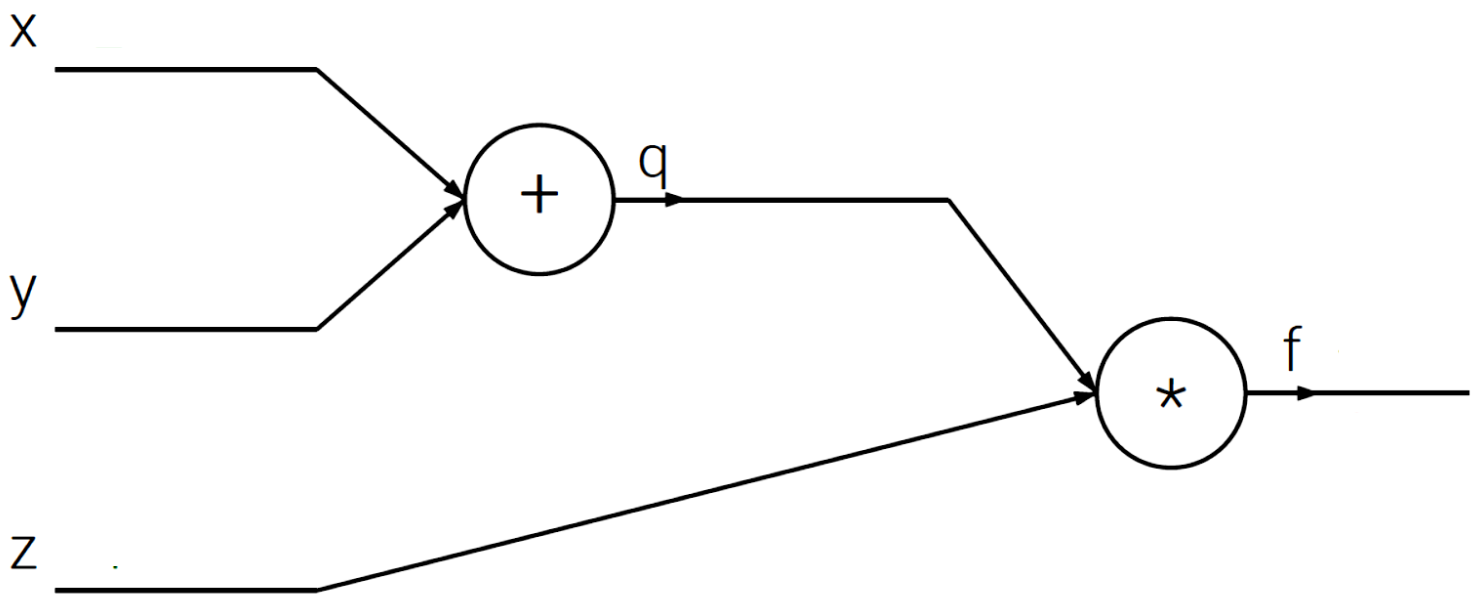
can be broken down into two expressions:

$$q = x + y$$
$$f = qz$$

- $\frac{\partial f}{\partial q} = z \quad \frac{\partial f}{\partial z} = q$
- $\frac{\partial q}{\partial x} = 1 \quad \frac{\partial q}{\partial y} = 1$
- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z \quad \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z$
- $\frac{\partial f}{\partial z} = q = x + y$

# Computational Graph

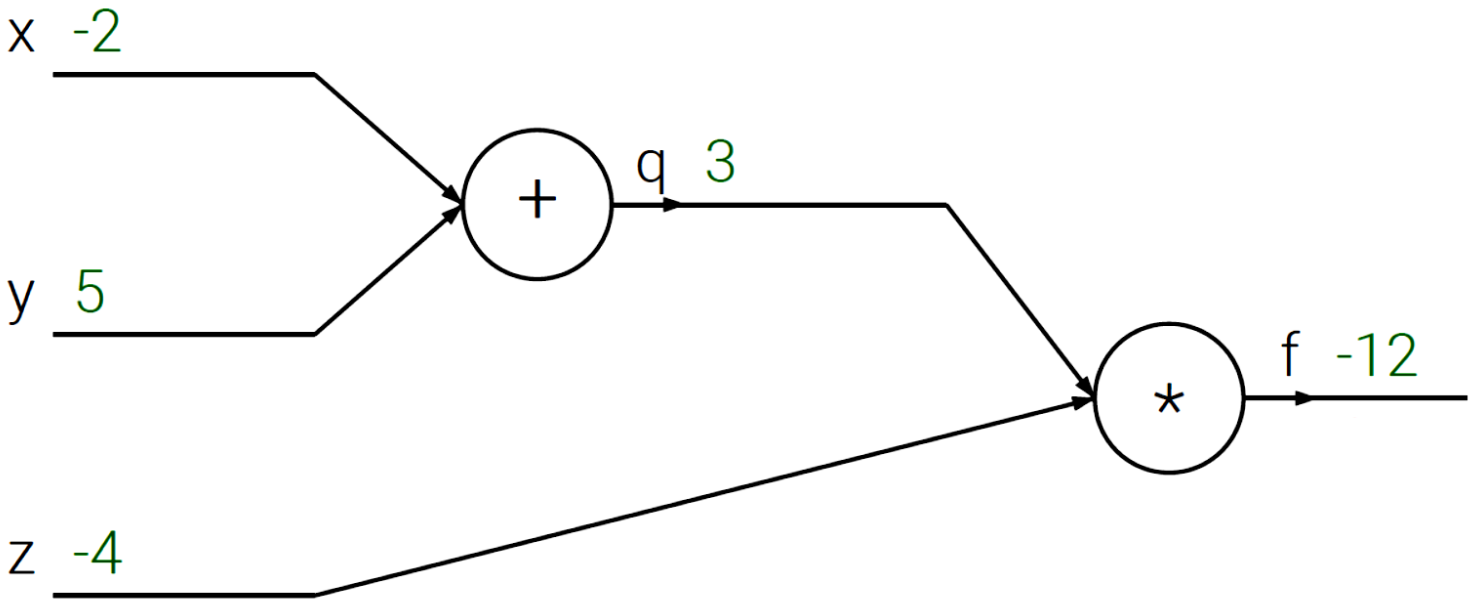
$$f(x, y, z) = (x + y)z = qz$$



# Computational Graph: Forward Pass

$$f(x, y, z) = (x + y)z = qz$$

1. Compute **output values**
2. Compute **local derivatives**: $\frac{\partial f}{\partial q} = z$ $\frac{\partial f}{\partial z} = q$ $\frac{\partial q}{\partial x} = 1$ $\frac{\partial q}{\partial y} = 1$

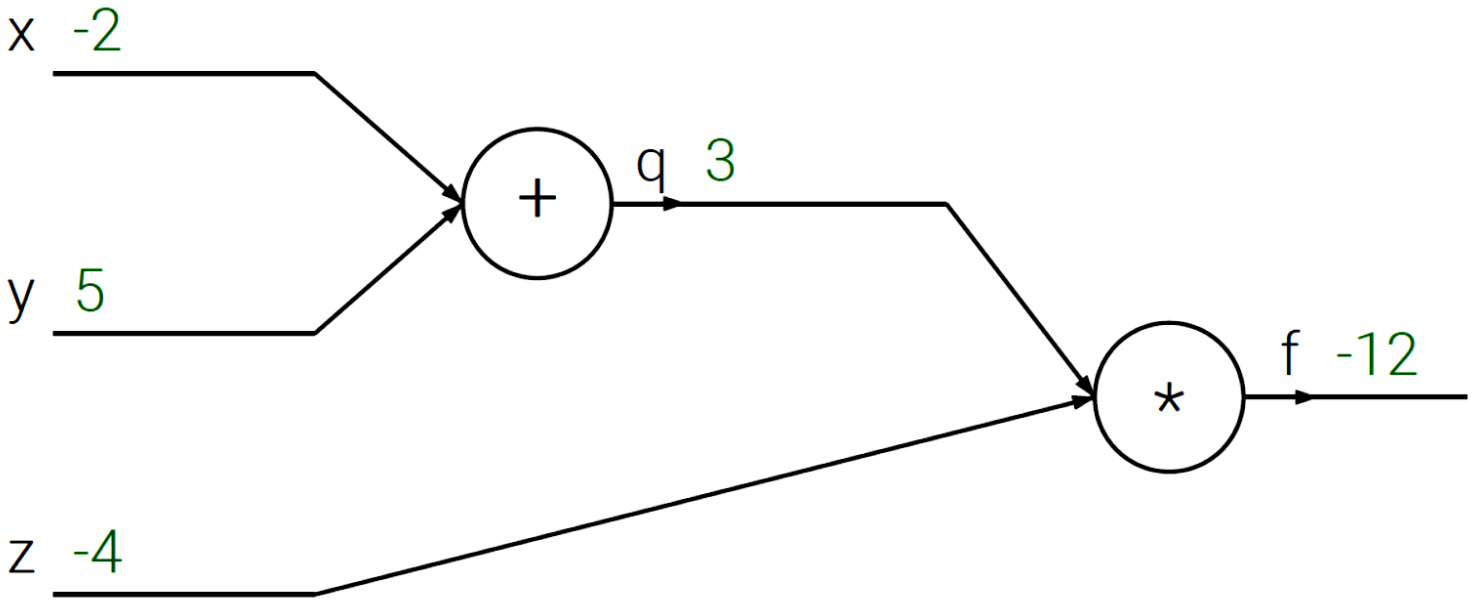


# Backpropagation!

$$f(x, y, z) = (x + y)z = qz$$

2. **Local derivatives**: $\frac{\partial f}{\partial q} = z$ $\frac{\partial f}{\partial z} = q$ $\frac{\partial q}{\partial x} = 1$ $\frac{\partial q}{\partial y} = 1$

3. **Chain Rule**: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$ $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$

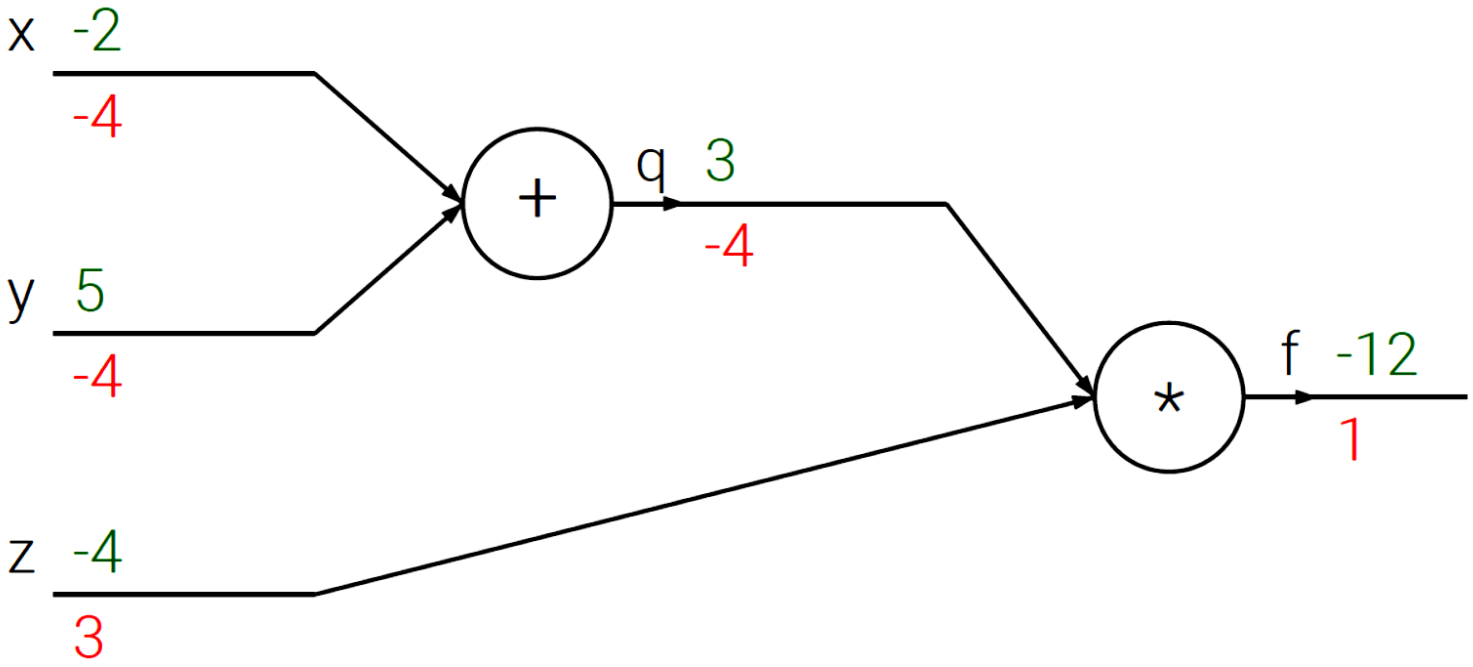


# Backpropagation!

$$f(x, y, z) = (x + y)z = qz$$

2. **Local derivatives**: $\frac{\partial f}{\partial q} = z$ $\frac{\partial f}{\partial z} = q$ $\frac{\partial q}{\partial x} = 1$ $\frac{\partial q}{\partial y} = 1$

3. **Chain Rule**: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$ $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$



# Intuitive understanding of backpropagation

**Forward pass:**

Every gate gets some inputs and can right away compute two things:

1. its output value
2. the local gradient of its inputs w.r.t. its output value.

Gates can do this completely independently!

**Backpropagation:**

1. Gate learns the gradient of its output.
2. Chain rule says that the gate takes that gradient and multiply it into every gradient it normally computes for its inputs.

# Intuitive understanding of backpropagation

- Backpropagation can thus be thought of as *gates communicating to each other* (through the gradient signal) *whether they want their outputs to increase or decrease* (and how strongly), so as to make the final output value higher.

- Any kind of differentiable function can act as a gate.
- We can group multiple gates into a single gate, or decompose a function into multiple gates whenever it is convenient.

# Hands-on Exercise: Backpropagation

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

- $w_0 = 2, x_0 = -1, w_1 = -3, x_1 = -2, w_2 = -3$

# Hands-on Exercise: Backpropagation

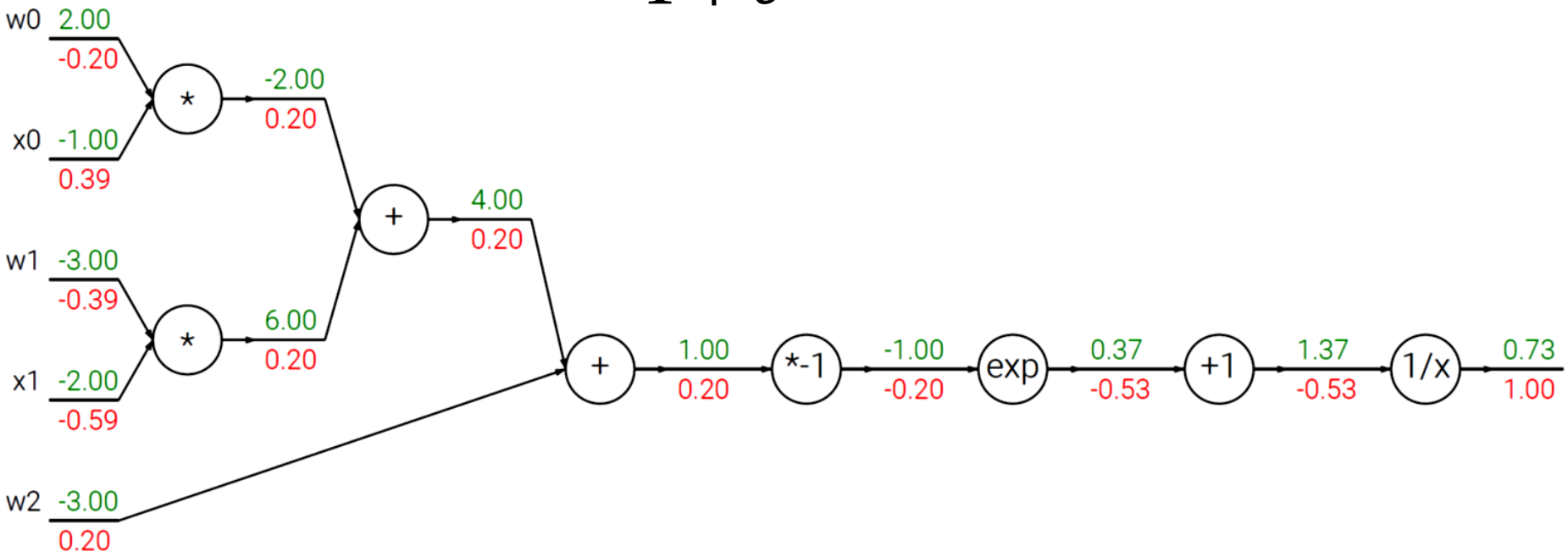
$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$

Needed derivatives

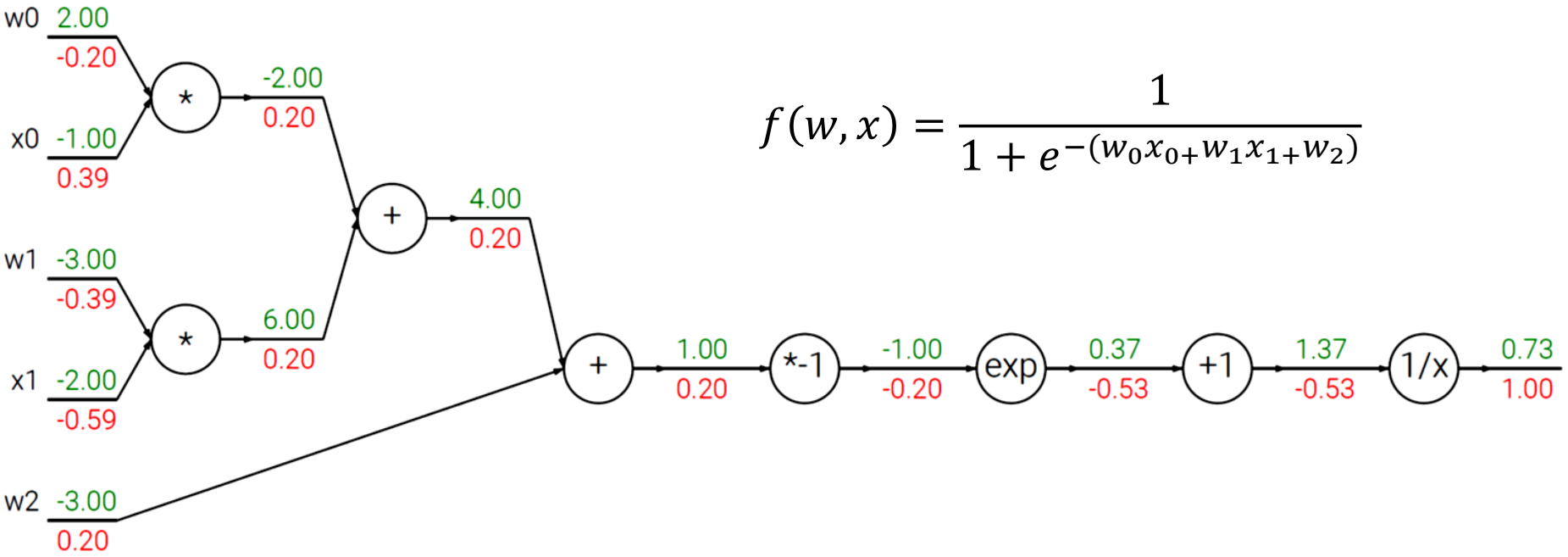
- $f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{\partial f}{\partial x} = \frac{-1}{x^2}$
- $f(x) = c + x \quad \rightarrow \quad \frac{\partial f}{\partial x} = 1$
- $f(x) = e^x \quad \rightarrow \quad \frac{\partial f}{\partial x} = e^x$
- $f(x) = ax \quad \rightarrow \quad \frac{\partial f}{\partial x} = a$

# Backpropagation!

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_{0+} w_1 x_{1+} w_2)}}$$



# Chaining gradients




$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_{0+} w_1 x_{1+} w_2)}}$$

- $\sigma(x) = \frac{1}{1+e^{-x}} \quad \rightarrow \quad \frac{\partial \sigma(x)}{\partial x} = \big(1 - \sigma(x)\big)\sigma(x)$
- It helps to be aware of which parts of the expression have easy local gradients
  - that they can be chained together!

# Another example!

$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

- Structure the code in such way that it contains multiple intermediate variables, each of which are only simple expressions for which we already know the local gradients.

# Implementation!

$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}$$

```
x = 3 # example values
y = -4

# forward pass
sigy = 1.0 / (1 + math.exp(-y)) # sigmoid in numerator   #(1)
num = x + sigy # numerator                               #(2)
sigx = 1.0 / (1 + math.exp(-x)) # sigmoid in denominator #(3)
xpy = x + y                                              #(4)
xpysqr = xpy**2                                          #(5)
den = sigx + xpysqr # denominator                        #(6)
invden = 1.0 / den                                       #(7)
f = num * invden # done!                                 #(8)
```

```
# backprop f = num * invden
dnum = invden # gradient on numerator                             #(8)
dinvden = num                                                     #(8)
# backprop invden = 1.0 / den
dden = (-1.0 / (den**2)) * dinvden                                #(7)
# backprop den = sigx + xpysqr
dsigx = (1) * dden                                                #(6)
dxpysqr = (1) * dden                                              #(6)
# backprop xpysqr = xpy**2
dxpy = (2 * xpy) * dxpysqr                                        #(5)
# backprop xpy = x + y
dx = (1) * dxpy                                                   #(4)
dy = (1) * dxpy                                                   #(4)
# backprop sigx = 1.0 / (1 + math.exp(-x))
dx += ((1 - sigx) * sigx) * dsigx # Notice += !! See notes below  #(3)
# backprop num = x + sigy
dx += (1) * dnum                                                  #(2)
dsigy = (1) * dnum                                                #(2)
# backprop sigy = 1.0 / (1 + math.exp(-y))
dy += ((1 - sigy) * sigy) * dsigy                                 #(1)
# done! phew
```

- Cache forward pass variables.
- Gradients add up at forks.

# Patterns in backward flow

- 3 most commonly used gates in NN (add, mul, max), all have very simple interpretations during backpropagation.
- **Add gate:** takes the gradient on its output and distributes it equally to all of its inputs.
- **Max gate:** routes the gradient to the higher input.
- **Multiply gate:** takes input activations, swaps them and multiplies by its gradient.