ECE 20875

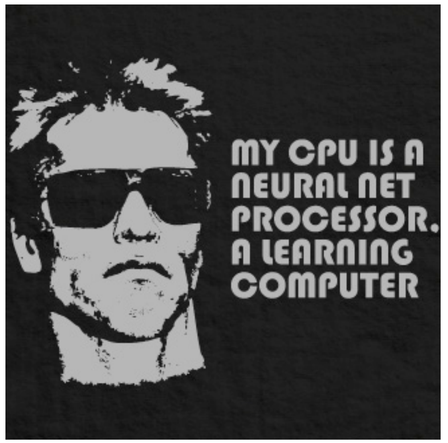
Python for Data Science

Chris Brinton and Qiang Qiu

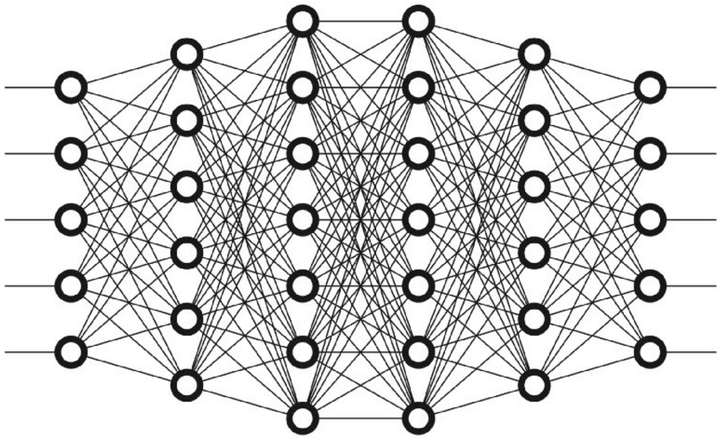
(Adapted from material developed by Profs. Milind Kulkarni, Stanley Chan, Chris Brinton, David Inouye)

introduction to neural networks

neural networks

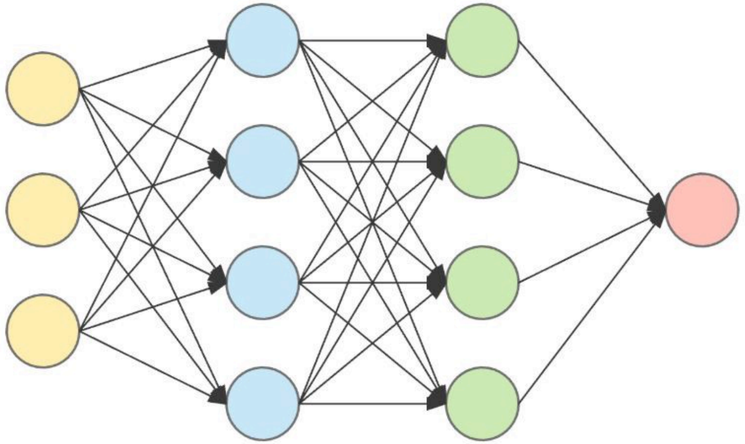


- Show up everywhere (including in pop culture)
  - Machine translation
  - Image recognition
  - Video generation
  - …
- Form the basis of the **deep learning** field
- Too many use cases for us to cover in this class
  - We will focus on neural networks used as **classifiers**



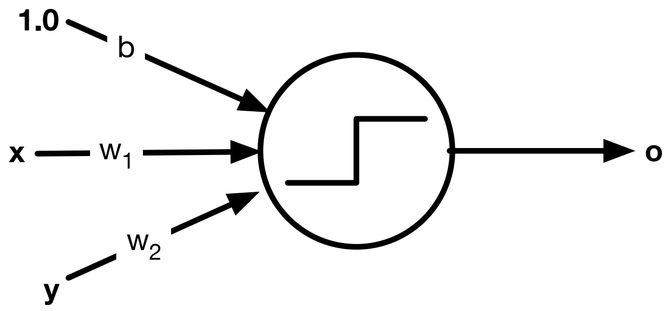
neurons

- The fundamental building blocks of neural networks are called **neurons**
  - Each has an **activation function**, modeled loosely after neurons in the brain, which “activate” when given enough stimulus
  - The human brain is estimated to have more than 10 billion neurons, to give you an idea
- Can view a neuron graphically as a “node” with inputs, and weights
  - The input to the activation function is the dot product of the input and weights



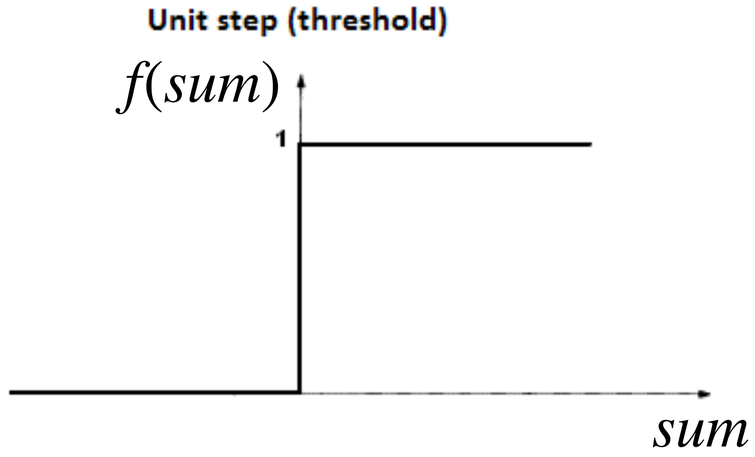
perceptrons

- A **perceptron** is the simplest form of a neuron
  - Activation function is the (Heaviside) **unit step function**: either “on” or “off”
- It uses the following **linear decision boundary**:



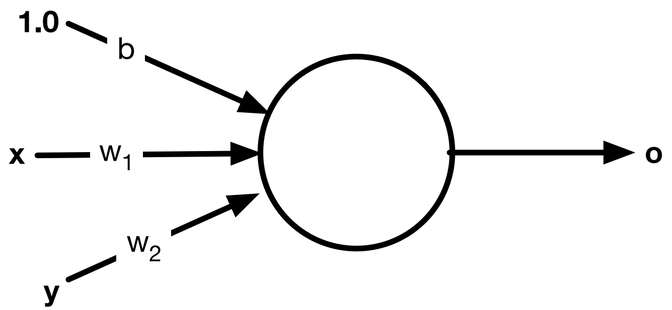
$$sum = \begin{bmatrix} b & w_1 & w_2 \end{bmatrix} \begin{bmatrix} 1.0 \\ x \\ y \end{bmatrix}$$

$$o = f(sum) = \begin{cases} 0, & sum \leq 0 \\ 1, & sum > 0 \end{cases}$$

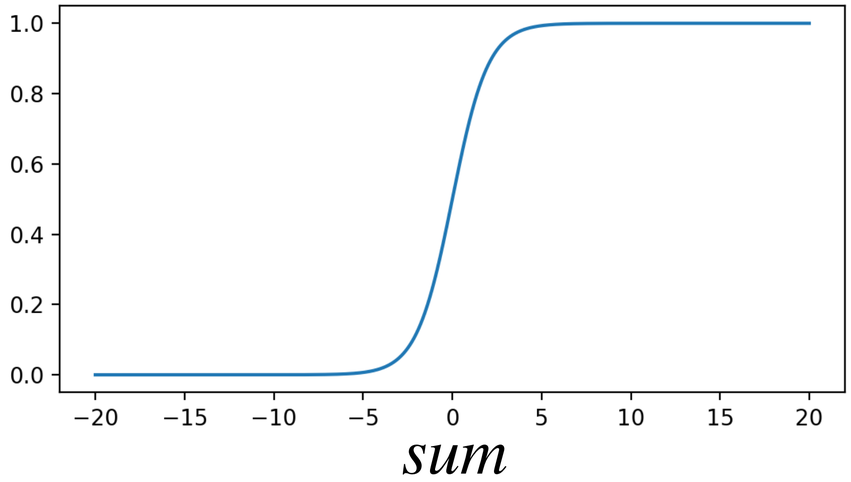


logistic regression: single layer NN

- Learning becomes a problem, because the unit step function cannot be differentiated
  - We need to somehow “smoothen” the transition at $sum = 0$
- One common activation function that does this is the **sigmoidal activation**, shown to the right
  - We can readily calculate the derivative
- This is just logistic regression!
  - A neural network with a single layer and sigmoidal activation is equivalent to logistic regression



$$o = f(sum) = \frac{1}{1 + e^{-sum}}$$

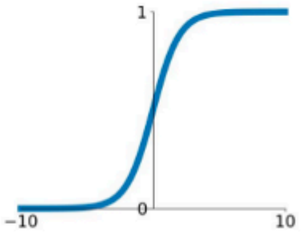


choices of activation functions

- The sigmoid function is computationally expensive, though (recall its derivative is complicated)
- There are many other activation functions we can use too. For example:
  - `tanh`: Hyperbolic tangent, has steeper derivatives than sigmoid
  - `ReLU`: Much easier to compute, but the outputs can be very large, and outputs below $x = 0$ suffer from the **vanishing gradient** problem
  - `Leaky ReLU`: Allows the output of `ReLU` below 0 to be slightly negative, which helps prevent neurons from falling into "dead states" from the vanishing gradient

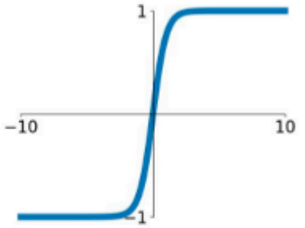
Sigmoid

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



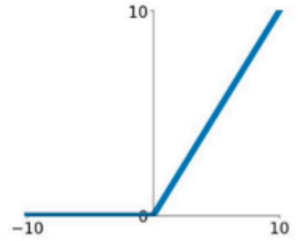
tanh

$$\tanh(x)$$



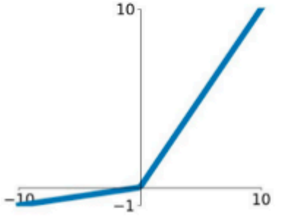
ReLU

$$\max(0, x)$$



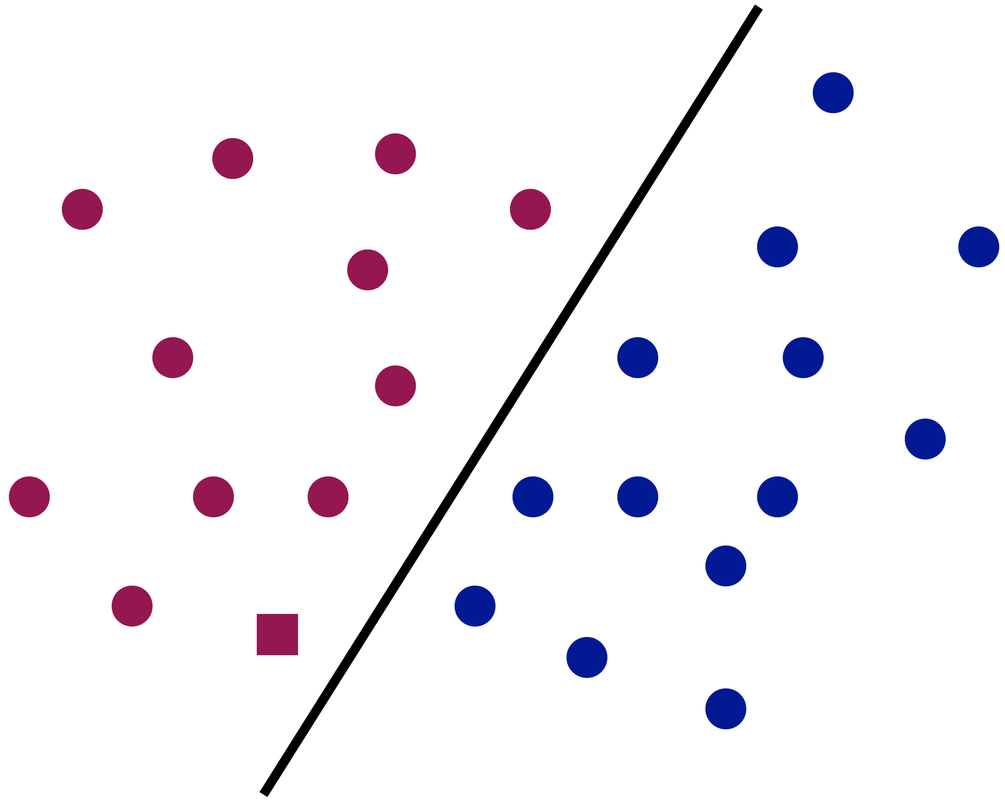
Leaky ReLU

$$\max(0.1x, x)$$



decision boundaries

- Basic classification problem for neural networks:
  - I have a set of labeled **training data**
  - Learn a **decision boundary** that separates the two classes of data
- Given a **new point**
  - Classify it using the decision boundary you learned
  - Similar to other classifiers we looked at!



creating decisions with neurons

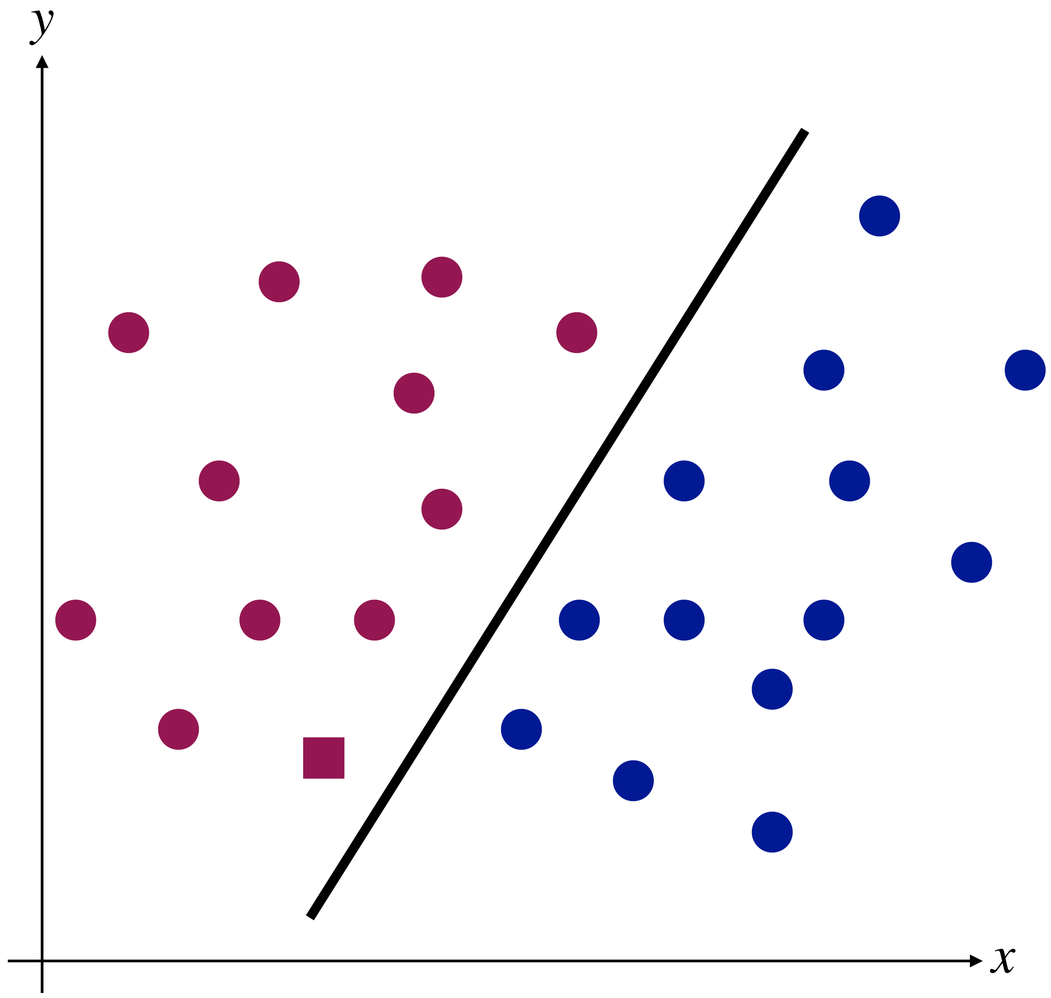
- The basic idea of neural networks is to add layers of complexity on how decision boundaries are defined

- A perceptron will induce a decision boundary that is a straight line, i.e.,

$$f(x, y) = \begin{cases} 0, & b + w_1 x + w_2 y \leq 0 \\ 1, & b + w_1 x + w_2 y > 0 \end{cases}$$

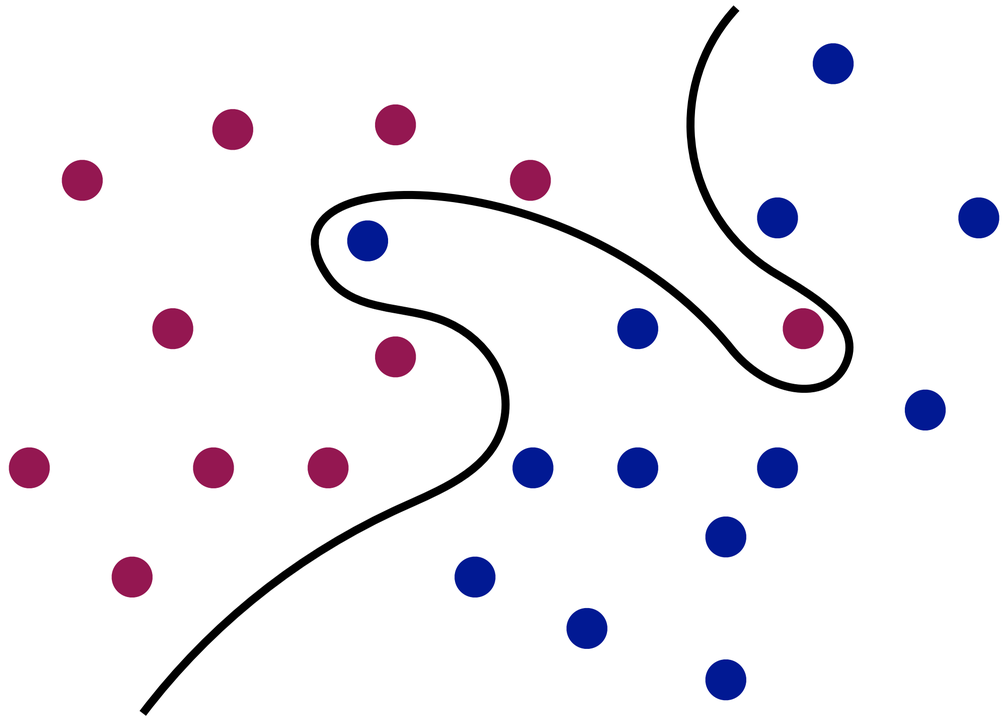
  - How do we learn the parameters w_1, w_2, and b of this model?

  - Instead of gradient descent, there is a "special" algorithm for perceptrons



non-linear decision boundaries

- The special **perceptron training algorithm** is guaranteed to converge if a **linear decision boundary** exists
- But if no linear boundary exists, the algorithm will not converge, not even to an imperfect solution
- Perceptrons cannot learn non-linear decision boundaries!
- To learn them with neural networks, we need two things:
  - Multiple layers of neurons
  - Smoother activation functions



multi-layer NN structure and intuition

(a) The building block of neural networks (a single **neuron**) is like a little logistic regression model:

1. Weighted summation of n inputs: $z = \sum_{i=1}^{n} w_i x_i$
2. Activation function: $y = f(z) = f\left(\sum_{i=1}^{n} w_i x_i \right)$

(b) We can put many of these neurons together to form a **feed-forward neural network** (or sometimes simply **deep NN** or **multilayer NN**)

1. Each neuron computes weighted summation and activation function
2. Stacking the neurons vertically forms a NN **layer**
3. Feeding the output of one layer as the input to the next layer creates a deep NN **(DNN)**

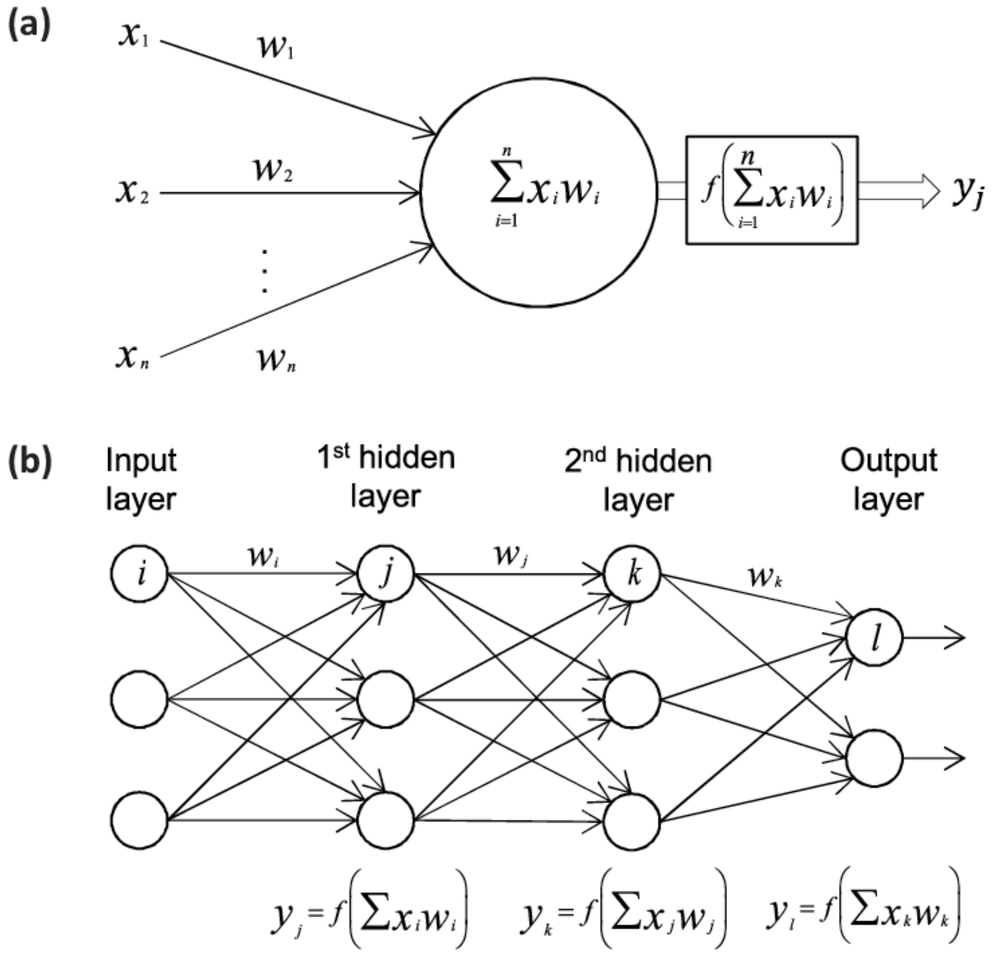


Figure from: Vieira, Sandra & Pinaya, Walter & Mechelli, Andrea. (2017). Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications. Neuroscience & Biobehavioral Reviews. 74. 10.1016/j.neubiorev.2017.01.002.

multi-layer NN mathematical form

1. Notice that the weighted summation for neuron j can be seen as a dot product:

$$z_j = \sum_{i=1}^{n} w_i x_i = \mathbf{w}_j^T \mathbf{x}$$

2. When stacking neurons vertically the layer outputs can be seen as a matrix multiplication:

$$\begin{aligned} z_1 &= \mathbf{w}_1^T \mathbf{x} \\ z_2 &= \mathbf{w}_2^T \mathbf{x} \\ &\vdots \\ z_n &= \mathbf{w}_n^T \mathbf{x} \end{aligned} \quad , \quad \text{which can be written as } \mathbf{z} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \vdots \\ \mathbf{w}_n^T \end{bmatrix} \mathbf{x} = W\mathbf{x}$$

3. Now the activation function is applied *independently* to each output:

$$\begin{aligned} y_1 &= f(z_1) \\ y_2 &= f(z_2) \\ &\vdots \\ y_n &= f(z_n) \end{aligned} \quad , \quad \text{which can be written as } \mathbf{y} = \begin{bmatrix} f(z_1) \\ f(z_2) \\ \vdots \\ f(z_n) \end{bmatrix} = f(\mathbf{z})$$

4. Thus we can write a DNN mathematically as function composition:

$$DNN(\mathbf{x}) = \mathbf{f}(W^{(3)}\,\mathbf{f}(W^{(2)}\,\overbrace{\mathbf{f}(W^{(1)}\mathbf{x})}^{\text{Layer 1}}))$$

Layer 2 (the inner $\mathbf{f}(W^{(2)}\,\mathbf{f}(W^{(1)}\mathbf{x}))$)

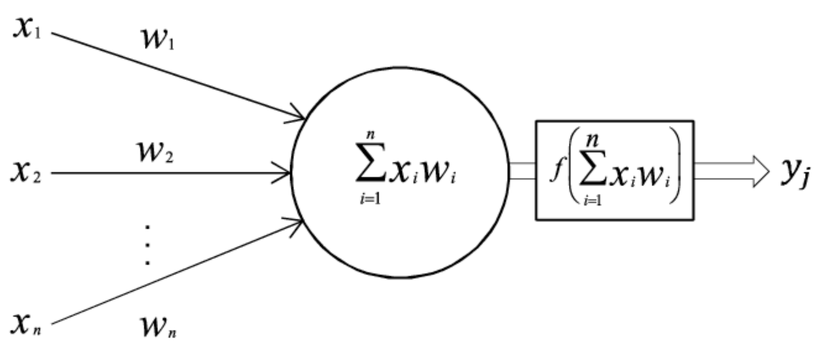
Output layer (the whole expression)

, or equivalently

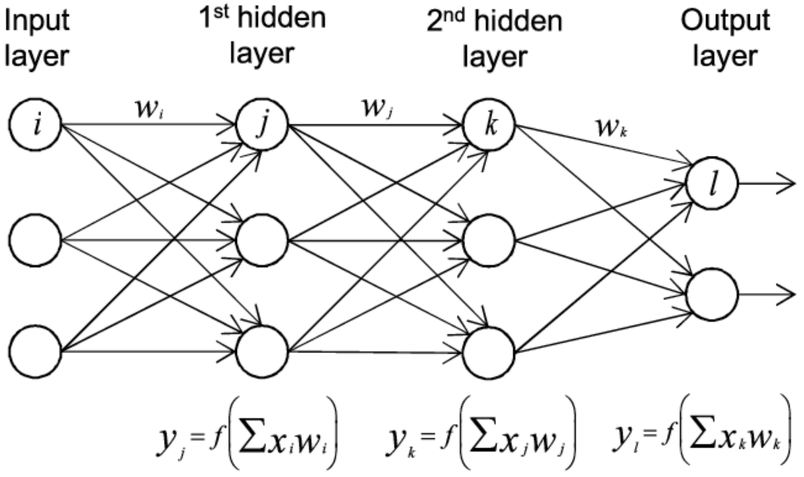
$$\begin{aligned} \mathbf{z}^{(1)} &= W^{(1)}\mathbf{x} \\ \mathbf{y}^{(1)} &= \mathbf{f}(\mathbf{z}^{(1)}) \\ \mathbf{z}^{(2)} &= W^{(2)}\mathbf{y}^{(1)} \\ \mathbf{y}^{(2)} &= \mathbf{f}(\mathbf{z}^{(2)}) \\ \mathbf{z}^{(3)} &= W^{(3)}\mathbf{y}^{(2)} \\ \mathbf{y}^{(3)} &= \mathbf{f}(\mathbf{z}^{(3)}) \end{aligned}$$

Alternating between **linear** transformation and **non-linear** activation functions

(a)



(b)



example of non-linear decision boundary

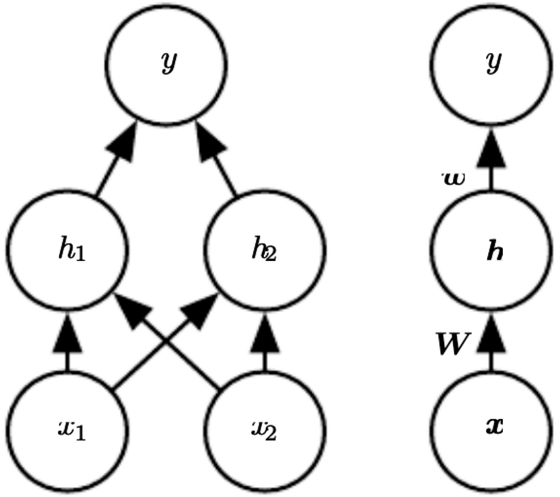
- Consider XOR classification function (i.e. “exclusive or”)
  - Outputs 1 only when exactly one of x_1 and x_2 is 1
  - Clearly not a linear decision boundary
- Can single layer NN handle this non-linear decision boundary problem?
- We will use simple two layer NN:

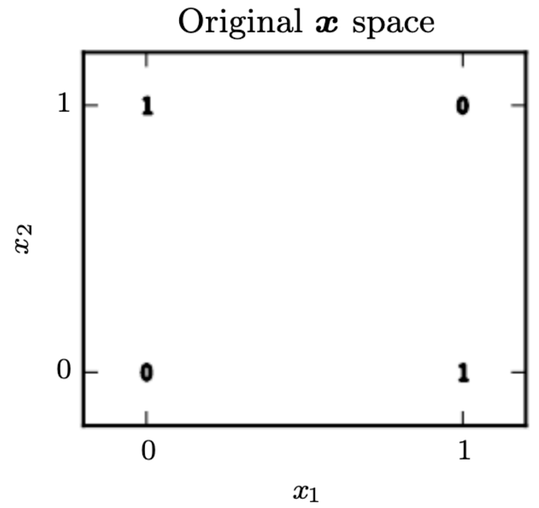
$$\mathbf{h} = \text{ReLU}(W\mathbf{x} + \mathbf{c}) = \max\{0, W\mathbf{x} + \mathbf{c}\}$$

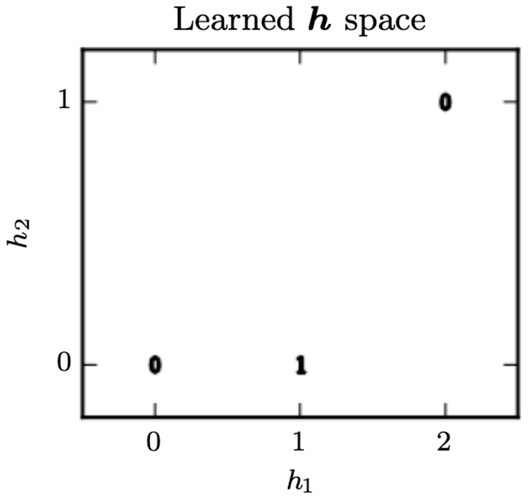
$$y = \mathbf{w}^T\mathbf{h}$$

- Solution:

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$







https://www.deeplearningbook.org/contents/mlp.html

XOR example walkthrough

- We can verify that this two-layer NN implements the XOR function:
- $x_1 = 0$ and $x_2 = 0$: $\mathbf{h} = \text{ReLU}\left(\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) = \text{ReLU}\left(\begin{bmatrix}0\\-1\end{bmatrix}\right) = \begin{bmatrix}0\\0\end{bmatrix}$,

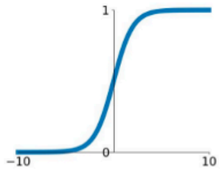
  $y = \begin{bmatrix}1 & -2\end{bmatrix}\begin{bmatrix}0\\0\end{bmatrix} = 0$
- $x_1 = 0$ and $x_2 = 1$: $\mathbf{h} = \text{ReLU}\left(\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) = \text{ReLU}\left(\begin{bmatrix}1\\0\end{bmatrix}\right) = \begin{bmatrix}1\\0\end{bmatrix}, y = \begin{bmatrix}1 & -2\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = 1$
- $x_1 = 1$ and $x_2 = 0$: $\mathbf{h} = \text{ReLU}\left(\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) = \text{ReLU}\left(\begin{bmatrix}1\\0\end{bmatrix}\right) = \begin{bmatrix}1\\0\end{bmatrix}, y = \begin{bmatrix}1 & -2\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix} = 1$
- $x_1 = 1$ and $x_2 = 1$: $\mathbf{h} = \text{ReLU}\left(\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix}\begin{bmatrix}1\\1\end{bmatrix} + \begin{bmatrix}0\\-1\end{bmatrix}\right) = \text{ReLU}\left(\begin{bmatrix}2\\1\end{bmatrix}\right) = \begin{bmatrix}2\\1\end{bmatrix}, y = \begin{bmatrix}1 & -2\end{bmatrix}\begin{bmatrix}2\\1\end{bmatrix} = 0$

architecture and parameters of NN

- **Depth**: # of layers
- **Width**: # of neurons per layer
- **Activations**: sigmoid, ReLU, tanh, etc.

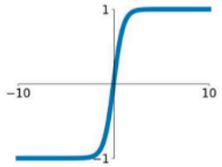
Sigmoid

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



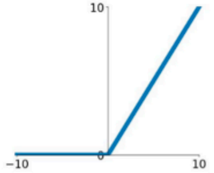
tanh

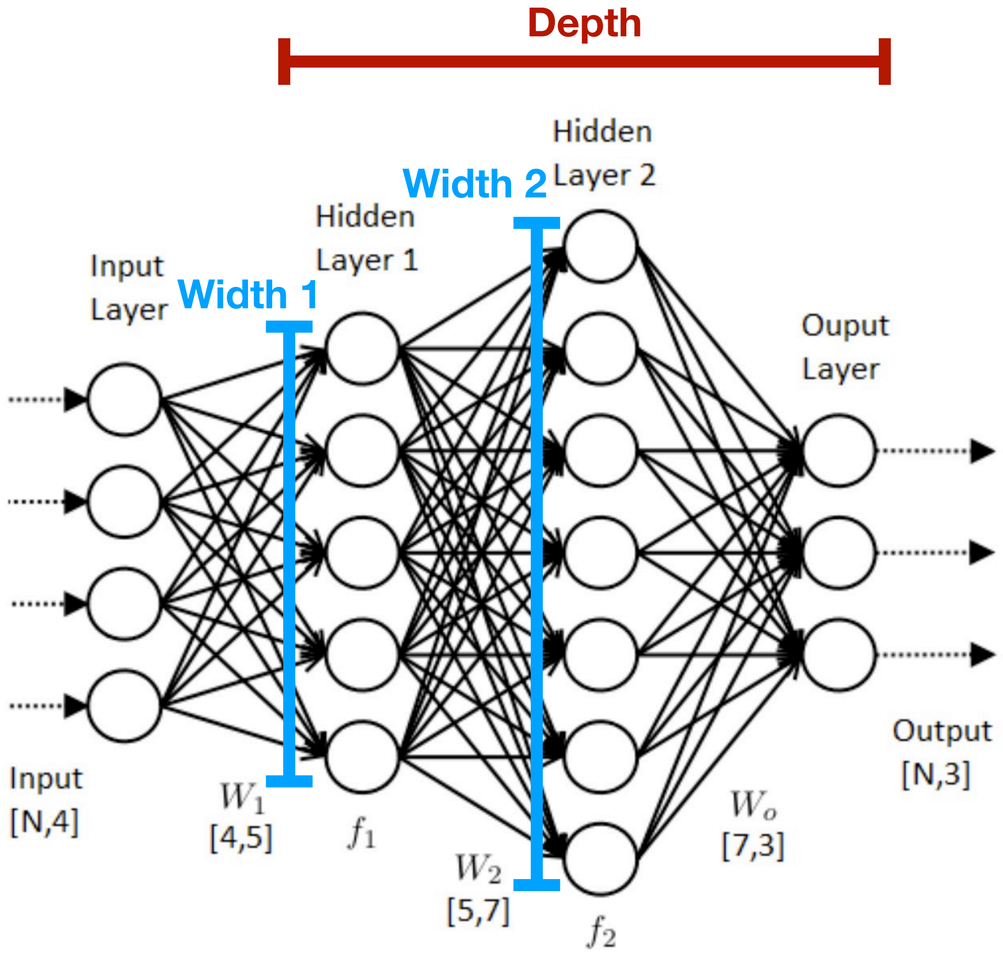
$\tanh(x)$



ReLU

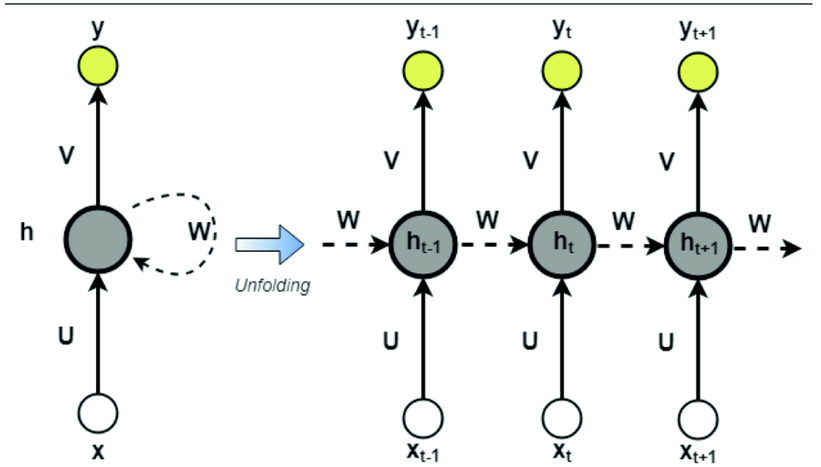
$\max(0, x)$

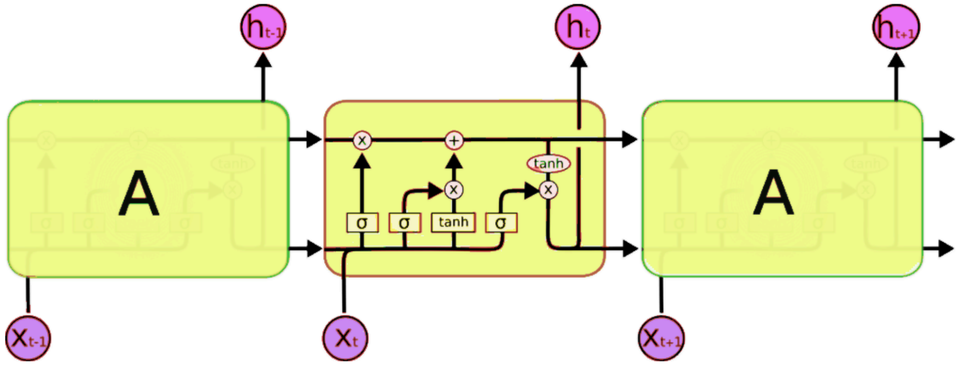




neural network architectures

- A plethora of neural network architectures have been proposed, for different applications
  - Multi-layer Perceptron (**MLP**): Cascading perceptrons
  - Recurrent Neural Networks (**RNN**): Sequential data modeling
  - Convolutional Neural Networks (**CNN**): Image recognition
  - Long Short Term Memory (**LSTM**): Memory cells with “forgetting” factors
  - Transformer (most recent), Gated Recurrent Units (**GRU**), Hopfield Networks, Boltzmann Machines, Generative Adversarial Networks (**GAN**), …





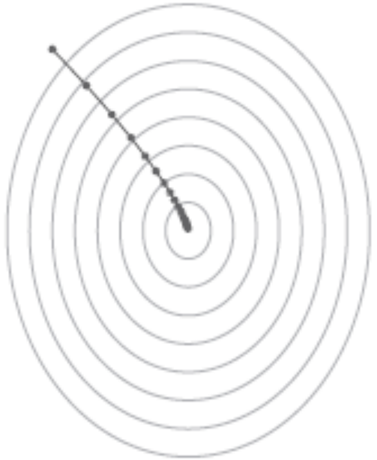
learning neural networks

- **(Batch) Gradient descent (GD)** can be computationally expensive for large datasets
  - E.g., if we have 1M images, *every update* requires computing and summing 10^6 gradients
    $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \sum_{i=1}^{n=10^6} \nabla F(x_i, y_i, \mathbf{w}^{(t)})$
  - If we add a normalizing constant of $1/n$, we can view this update as taking the expected gradient over all data samples:
    $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \frac{1}{n} \sum_{i=1}^{n} \nabla F(x_i, y_i, \mathbf{w}^{(t)}) = \mathbf{w}^{(t)} - \alpha \mathbb{E}[\nabla F(x_i, y_i, \mathbf{w}^{(t)})]$
- **Stochastic gradient descent (SGD)** massively reduces the computational complexity by only using 1 sample at each time step t, (x_t, y_t):

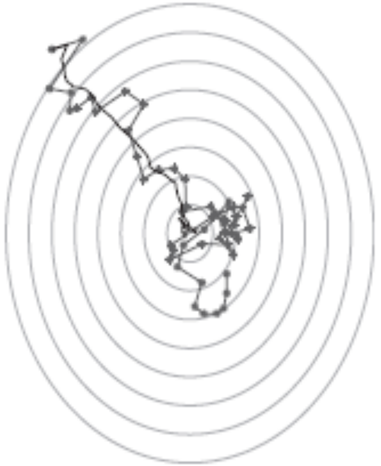
  $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \alpha \mathbb{E}[\nabla F(x_i, y_i, \mathbf{w}^{(t)})] \rightarrow \mathbf{w}^{(t)} - \alpha \nabla F(x_t, y_t, \mathbf{w}^{(t)})$

  - Note that the *variance of the steps* is much higher but the *cost is much lower*
  - Sometimes called **amortized learning** because it amortizes (spreads out) the computational cost across many iterations
  - **Mini-batch gradient descent** is actually used in practice, where often 64, 128 or 256 samples are used in each batch (bridging between SGD and GD)

Gradient descent



Stochastic Gradient Descent



SGD for a sigmoidal neuron

- Letting y_i be the label of datapoint i, $\mathbf{w} = (w_1, w_2, \dots)$ be the vector of weights, and $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots)$ be the datapoint vector, define the error $E(\mathbf{x}_i)$ of the output of a specific input:

$$E(\mathbf{x}_i) = \frac{1}{2}\left(y_i - f(sum)\right)^2 = \frac{1}{2}\left(y_i - f(\mathbf{w}^T\mathbf{x}_i)\right)^2$$

- For SGD, we only need the partial derivative for one specific input

$$\frac{\partial E(\mathbf{x}_i)}{\partial w_j} = \frac{\partial E(\mathbf{x}_i)}{\partial f(sum)} \cdot \frac{\partial f(sum)}{\partial sum} \cdot \frac{\partial sum}{\partial w_j} = -\underbrace{(y_i - f(sum)) \cdot f(sum)(1 - f(sum))}_{\text{Denote as } \delta_0 \text{ since same for every } w_j} \cdot x_{ij}$$

  - Remember that $\frac{\partial f(x)}{\partial x} = f(x)(1 - f(x))$ when f is a sigmoid

SGD for a sigmoidal neuron

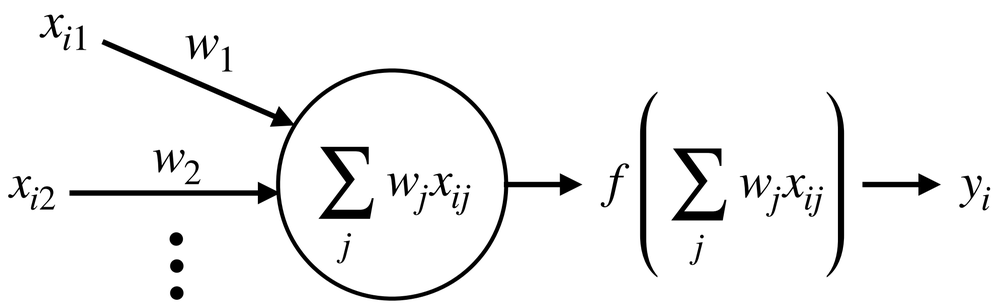
- From previous slide:

$$\frac{\partial E(\mathbf{x}_i)}{\partial w_j} = \frac{\partial E(\mathbf{x}_i)}{\partial f(sum)} \cdot \frac{\partial f(sum)}{\partial sum} \cdot \frac{\partial sum}{\partial w_j} = -\underbrace{(y_i - f(sum)) \cdot f(sum)(1 - f(sum))}_{\text{Denote as } \delta_0 \text{ since same for every } w_j} \cdot x_{ij}$$

- Thus, our SGD update rule becomes:

$$w_j^{(t+1)} = w_j^{(t)} + \alpha \cdot \delta_0 \cdot x_{ij}$$

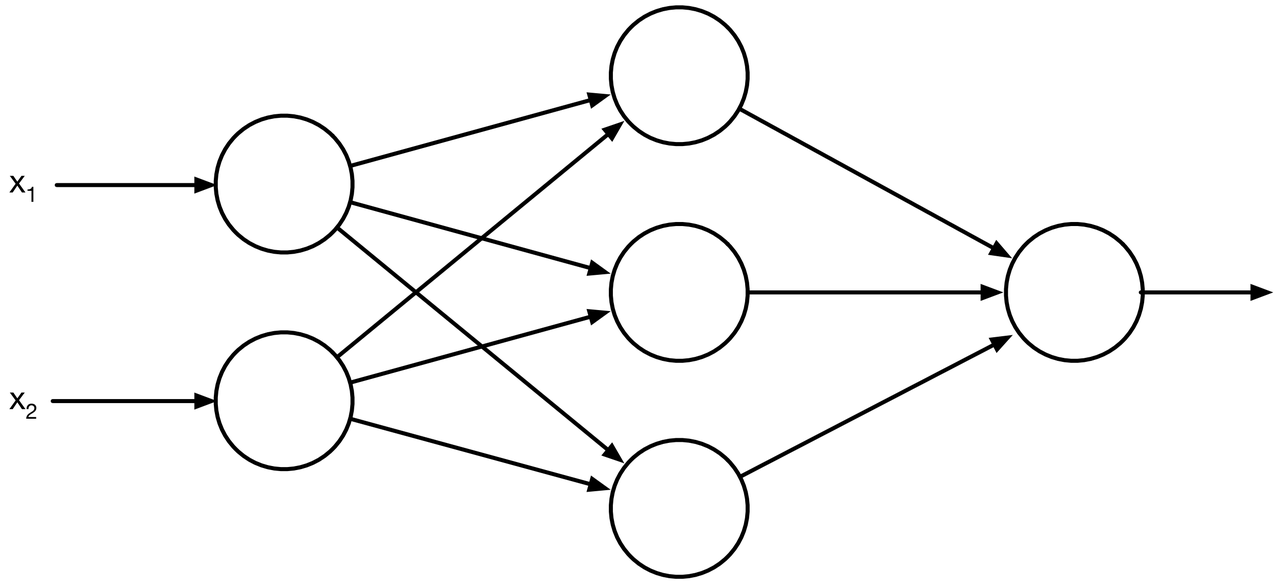
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \alpha \cdot \delta_0 \cdot \mathbf{x}_i$$



  - Importantly, δ_0 is reused for every w_j, so we only have to compute it once for each t

learning complex separators

- Let’s build up to more complex models by cascading neurons
- Learning the weights of the edges to the output neuron is easy — same as learning for a single neuron
- But what about the weights on the inputs to the hidden layer?



updating the deltas

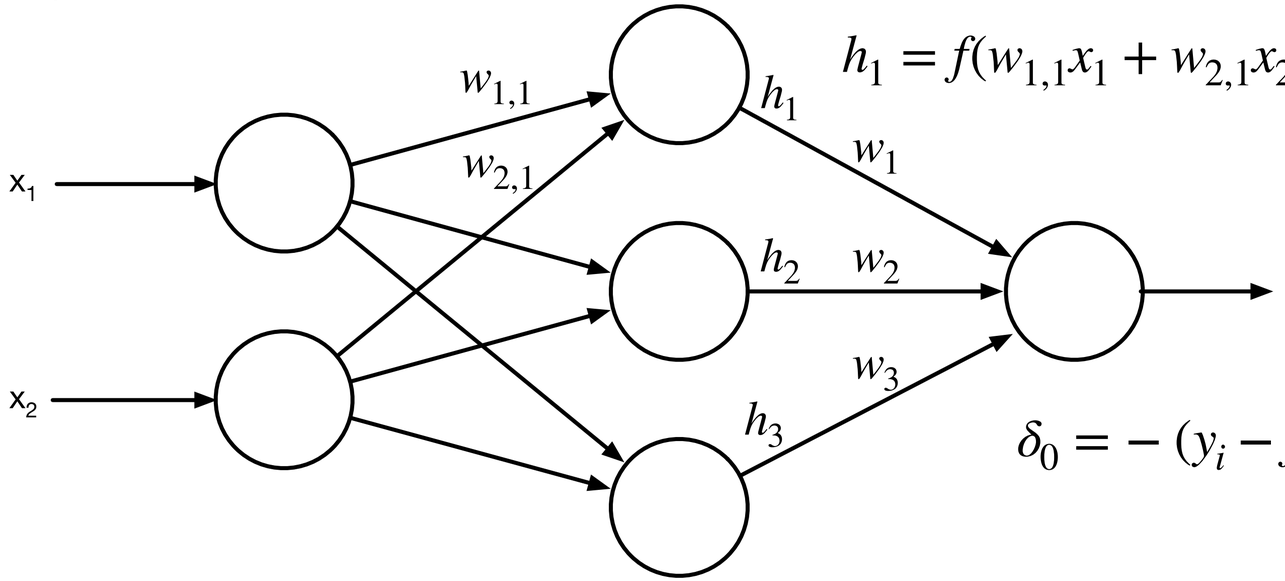
- Consider the network below. The output of this hidden layer is a vector $\mathbf{h}$. We can write the error of the network as:

$$E(\mathbf{w}) = \frac{1}{2}(y_i - f(w_1 h_1 + w_2 h_2 + w_3 h_3))^2 = \frac{1}{2}(y_i - f(w_1 f(w_{1,1} x_1 + w_{2,1} x_2) + w_2 h_2 + w_3 h_3))^2$$

- The change in output error with respect to $w_{1,1}$ is:

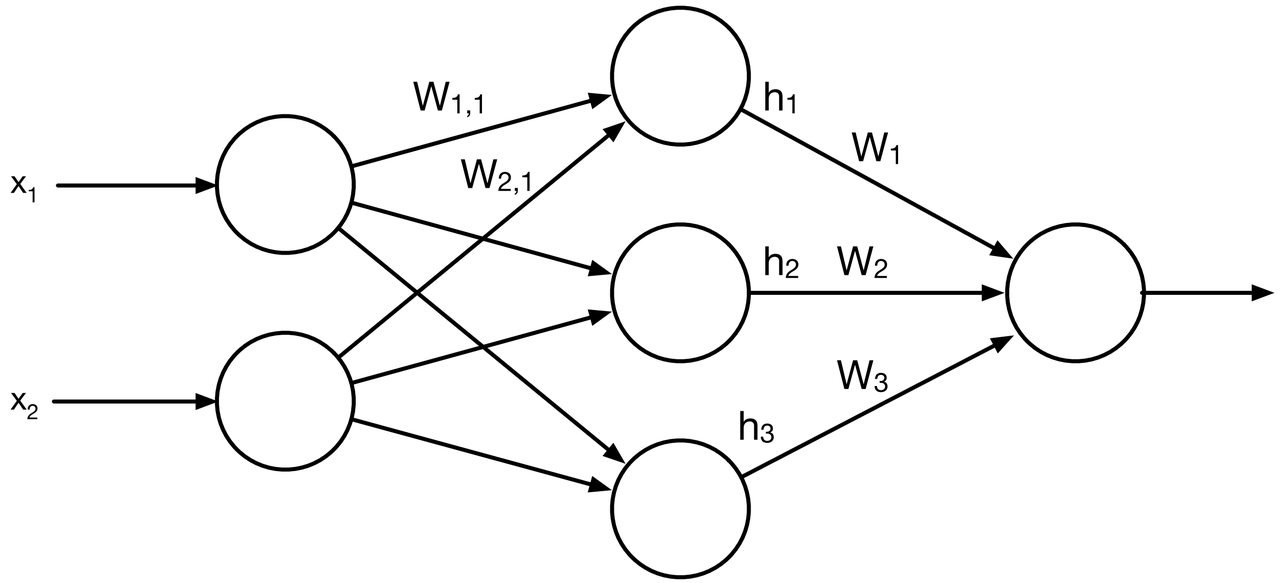
$$\frac{\partial E}{\partial w_{1,1}} = \frac{\partial E}{\partial h_1} \cdot \frac{\partial h_1}{\partial sum_{h_1}} \cdot \frac{\partial\, sum_{h_1}}{\partial w_{1,1}} = -\delta_0 w_1 \cdot f'(sum_{h_1}) \cdot x_1 = -\delta_{h_1} \cdot x_1$$

this term has the same form as what we derived for a single neuron previously



essence of backpropagation

- Computing the gradient for each neuron gives us the delta ($\delta_0, \delta_{h_1}, \ldots$) for the “upstream” neurons, so we can keep pushing error back
- This gives us the essence of **backpropagation** for training neural networks
  - **Forward pass**: Compute outputs of each neuron
  - **Backward pass**: Push errors (deltas, $\delta_0, \delta_{h_1}, \ldots$) weighted by edges to compute how the weights change.
  - Update: Apply stochastic gradient descent to each weight. Repeat.



implementing neural networks

- sklearn now has a built in MLP module:

```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes=(13,13,13),max_iter=500)
```

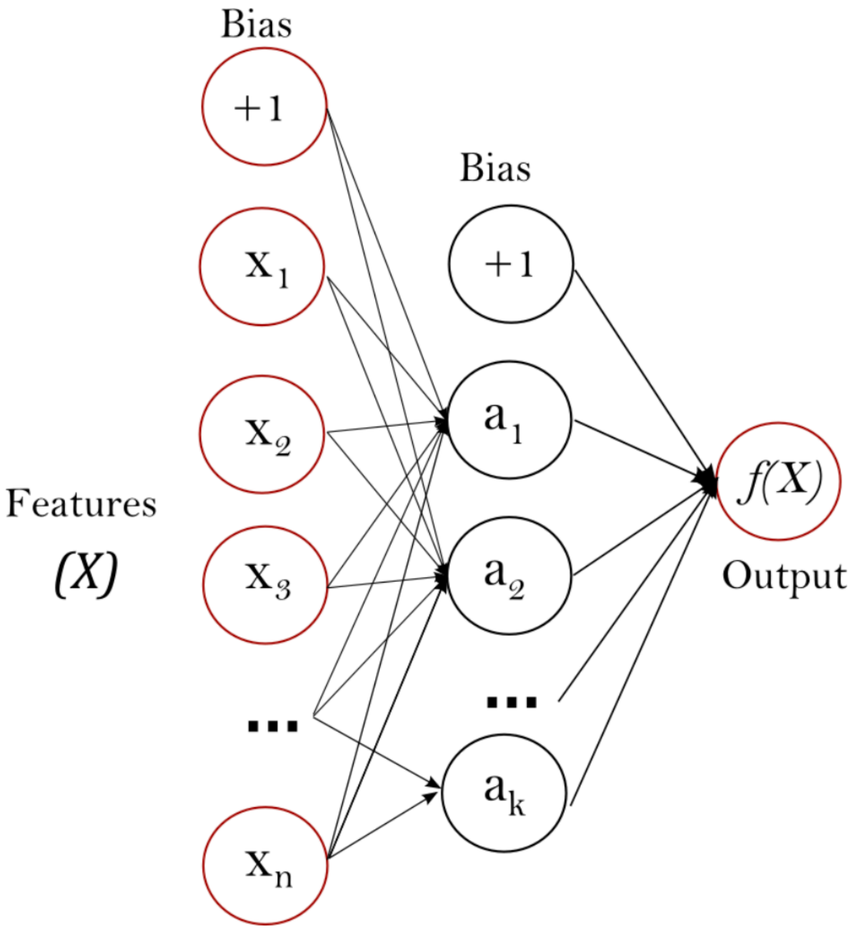
- For more complex neural networks, we typically leverage other machine learning libraries/platforms:
  - pytorch (https://pytorch.org/)
  - tensorflow (https://www.tensorflow.org/)
  - Both have Python interfaces

TensorFlow

PyTorch

Multi-layer Perceptron (MLP)

- Generally speaking, learns a function $f(\cdot) : \mathbb{R}^n \rightarrow \mathbb{R}^o$ by training on a dataset of input datapoints
  - Each input datapoint's dimension is n, output dimension is o
  - ML**P** is a misnomer, because it uses anything *except* a perceptron activation
- Architecture of an MLP:
  - Input layer, where each of the n input features is represented as a neuron
  - l hidden layers, where each layer performs a linear transformation followed by a non-linear activation
  - Output which has one non-linear activation for each of the o output dimensions



- Single hidden layer ($l = 1$)
- Single output ($o = 1$)

MLP in Python

- sklearn has a built in MLP (multi-layer perception) module

```
from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes=(13,13,13), max_iter=500)

mlp.fit(train_X, train_y)

print(mlp.coefs_[i])  # weight matrix corresponding to layer i (i=0,…,3)

print(mlp.intercept_[i])  # bias vector for neurons in layer i+1 (i=0,…,2)

results = mlp.predict(test_X)
```

- Check out https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html

deep learning training

- With deep learning, we have non-linear (and non-convex) error functions
- Therefore SGD is not guaranteed to converge to the global optimum solution
- A lot of research is devoted to …
  - Speeding up backpropagation, with methods like the Adam optimizer, or by distributing training across many nodes
  - Finding conditions for global solutions in neural networks