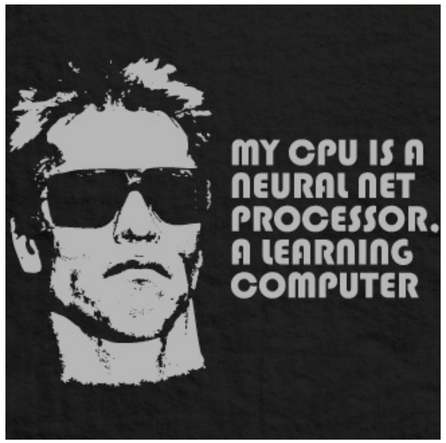
ECE 20875

Python for Data Science

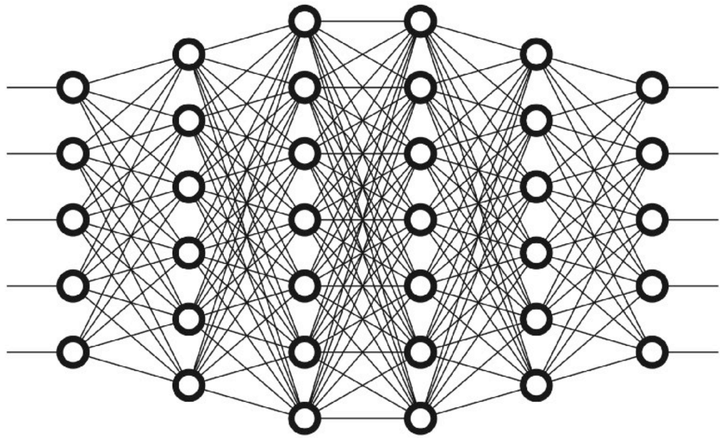
Milind Kulkarni and Chris Brinton

introduction to neural networks

neural networks

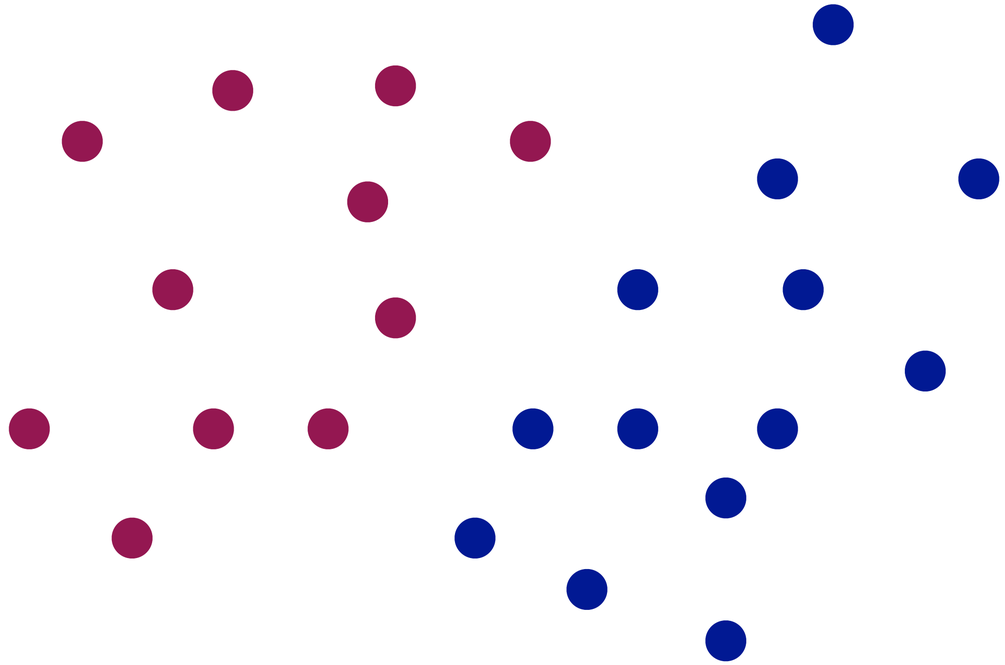


- Show up everywhere (including in pop culture)
  - Machine translation
  - Image recognition
  - Video generation
  - …
- Form the basis of the **deep learning** field
- Too many use cases for us to cover in this class
  - We will focus on neural networks used as **classifiers**



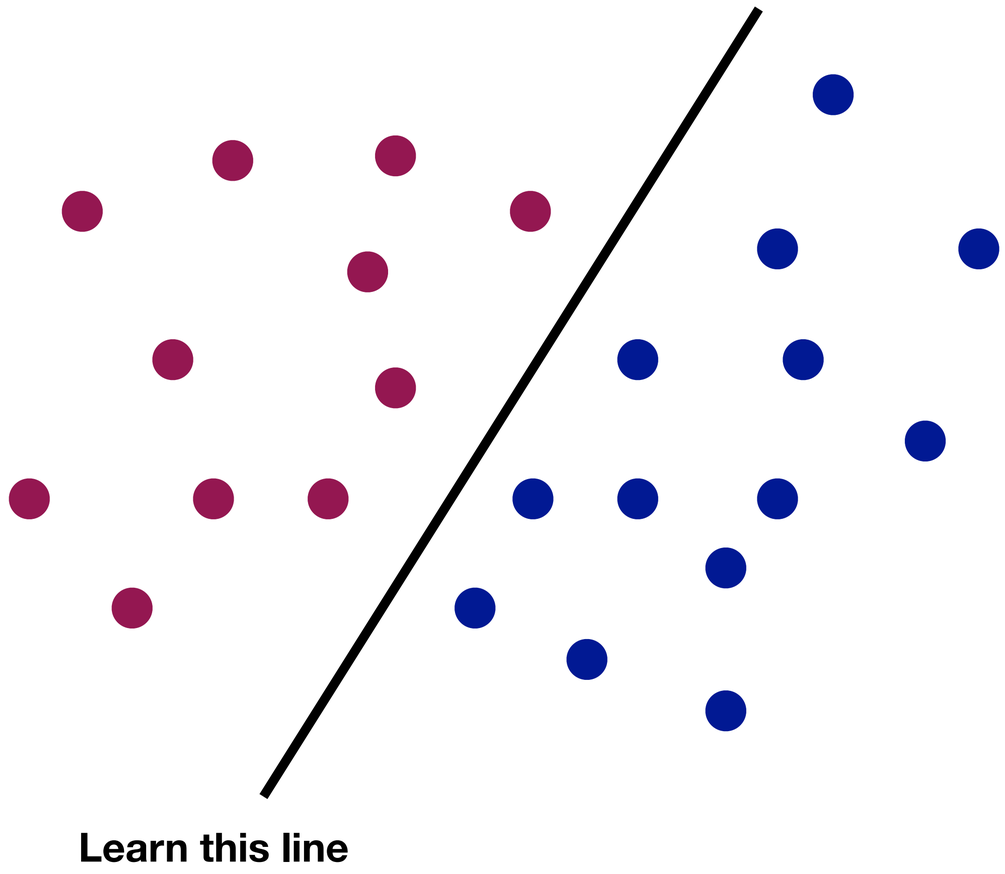
neural networks

- 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



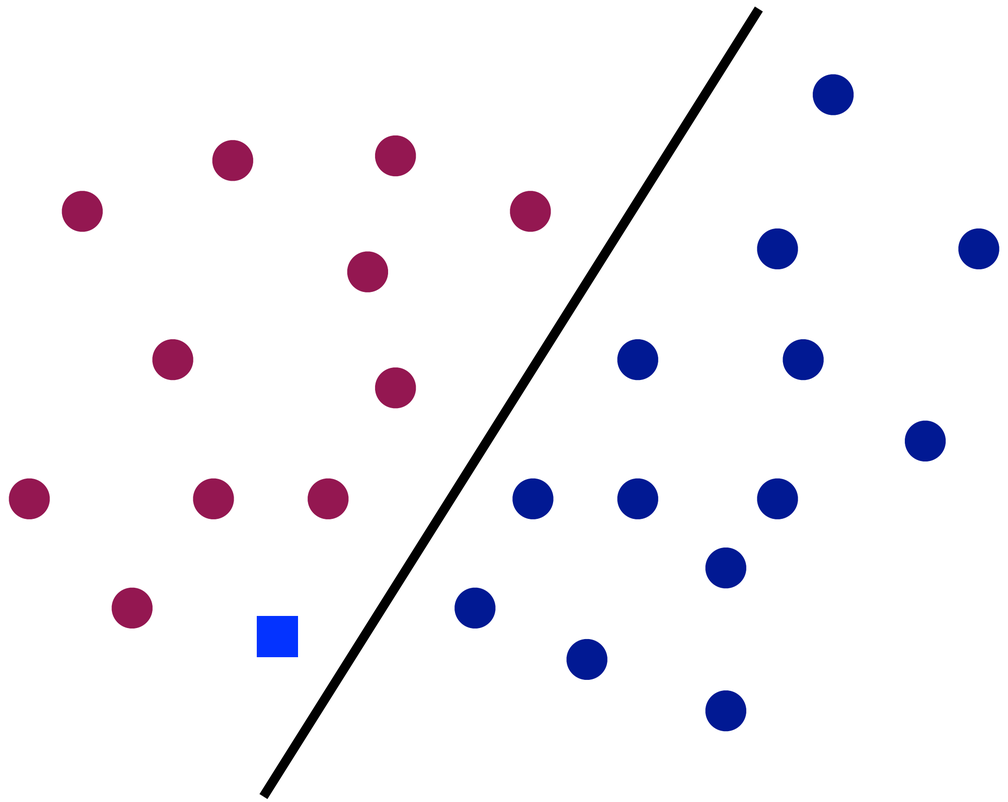
neural networks

- 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



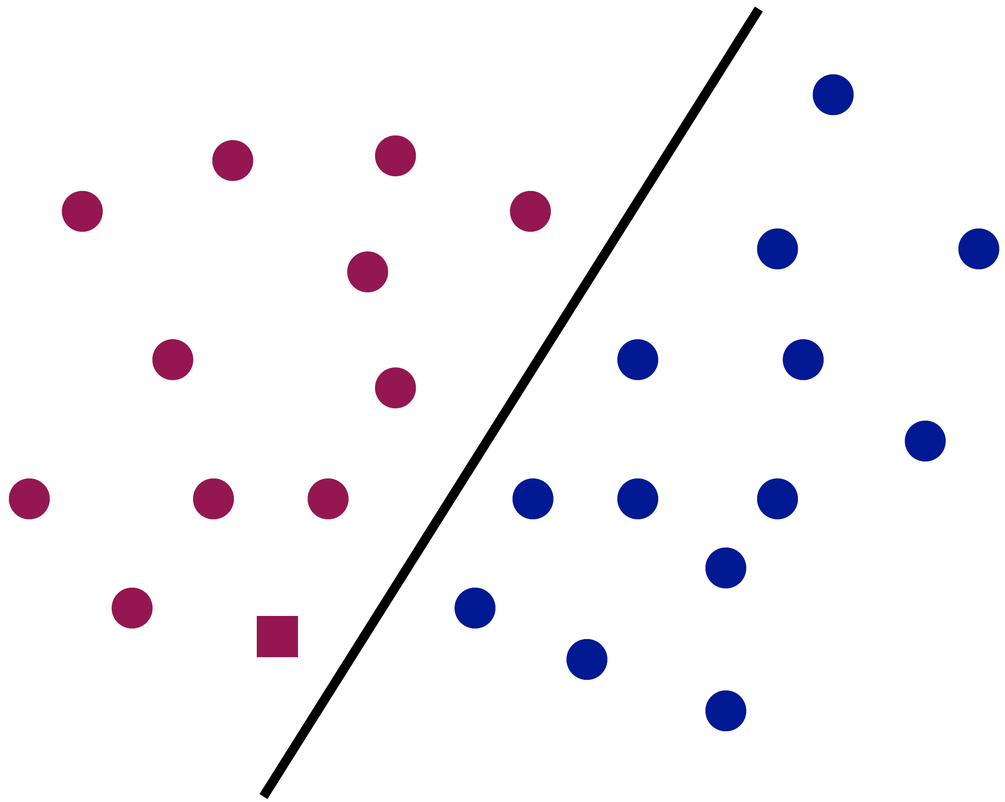
neural networks

- 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!



neural networks

- 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!



what is this model?

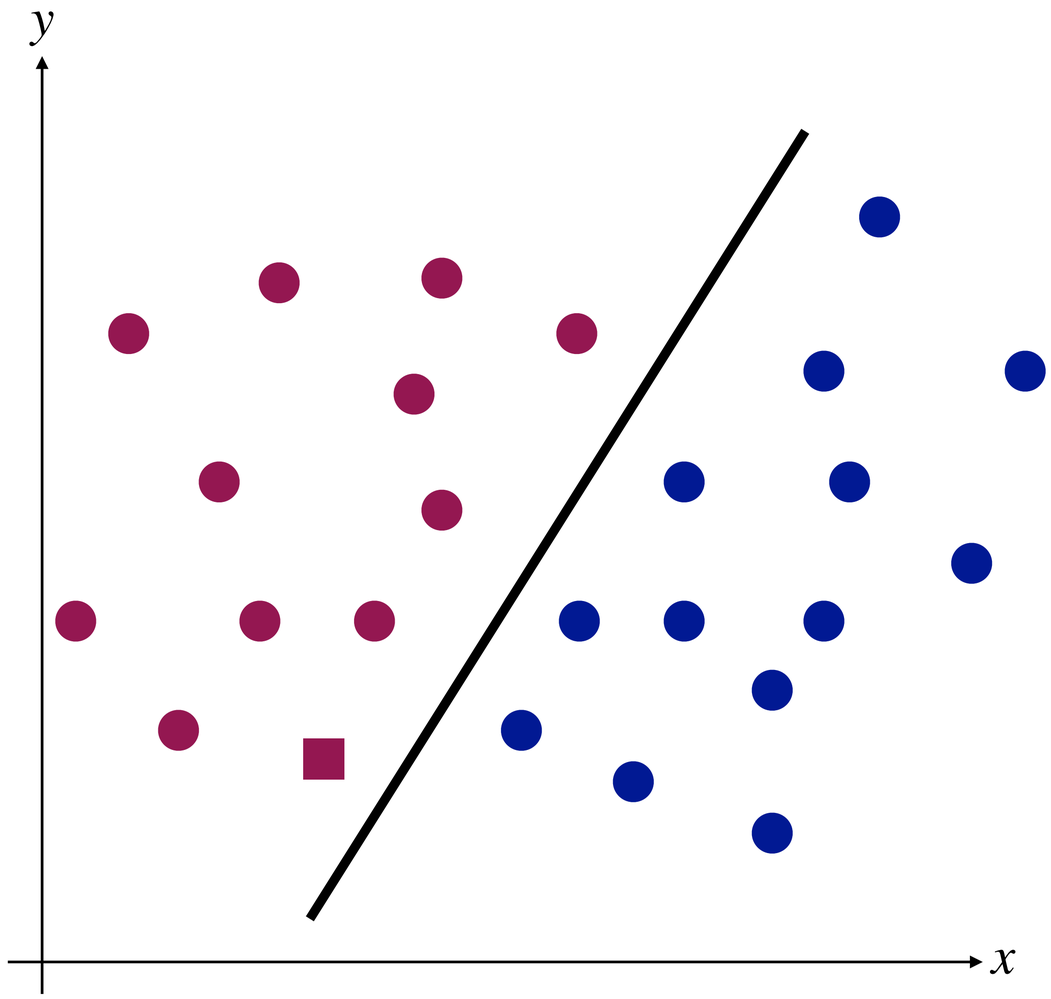
- The basic idea of neural networks is to add layers of complexity on how decision boundaries are defined

- First, suppose the decision boundary is just a *straight line*, i.e.,

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

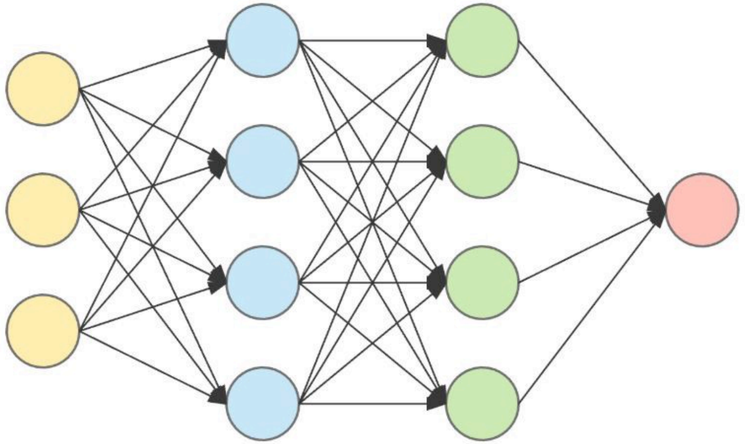
  - How do we learn the parameters w_1, w_2, and b of this model?

  - Let's find out by translating this into a neural network model



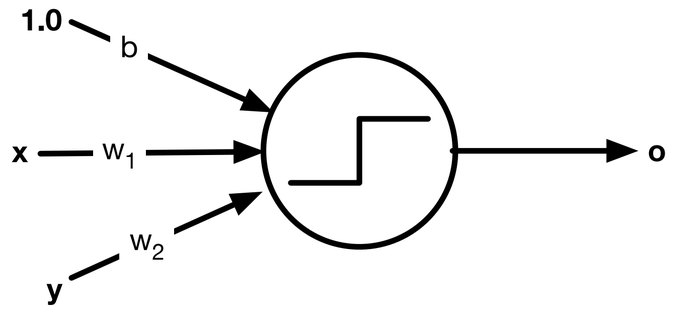
neurons

- 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



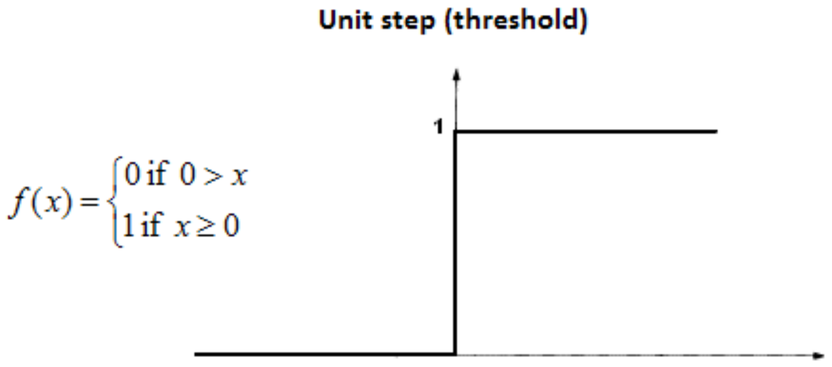
perceptrons

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



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

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



perceptron training algorithm

- So how do we learn the weights? One possible procedure:
- Randomly initialize weights
- For each training input:
  - Run the perceptron on the datapoint to get a "predicted" output
  - If the predicted output matches the real output (from the labels), leave the weights alone
  - If it doesn't match, move the weights in the direction of the input if the label is 1, and away from the input if the label is 0
- Repeat the previous step until convergence

input $i : (x_i, y_i)$

actual output: o_i

predicted output: p_i

error $\delta_i = o_i - p_i$

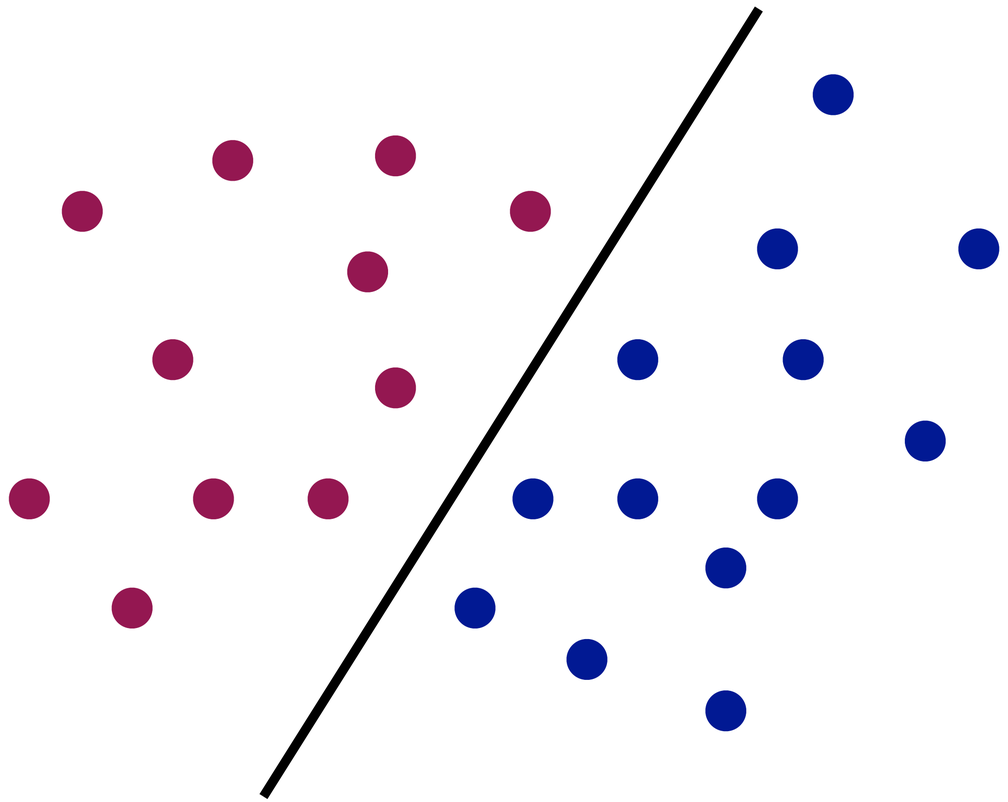
$$b^{(t+1)} = b^{(t)} + \delta_i \cdot 1.0$$

$$w_1^{(t+1)} = w_1^{(t)} + \delta_i \cdot x_i$$

$$w_2^{(t+1)} = w_2^{(t)} + \delta_i \cdot y_i$$

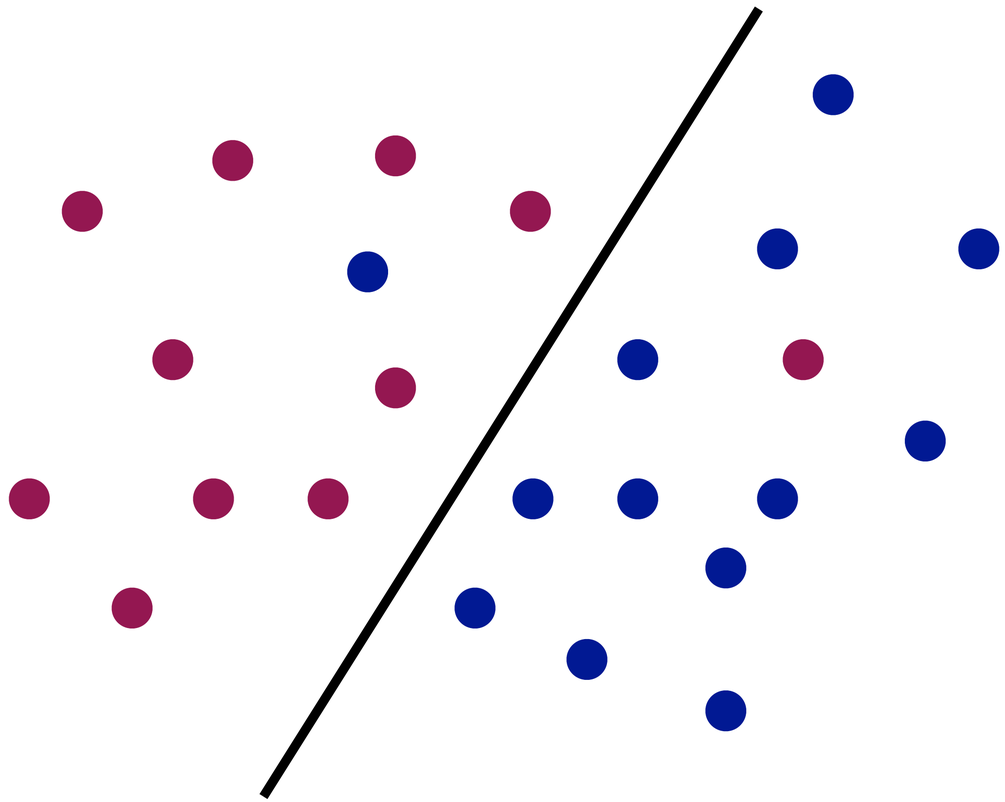
perceptron training algorithm

- Does this algorithm always work?
- Guaranteed to converge if a **linear decision boundary** exists
- But if no decision boundary exists, the algorithm will not converge, not even to an imperfect solution
  - Perceptrons cannot learn non-linear decision boundaries
- So what can?



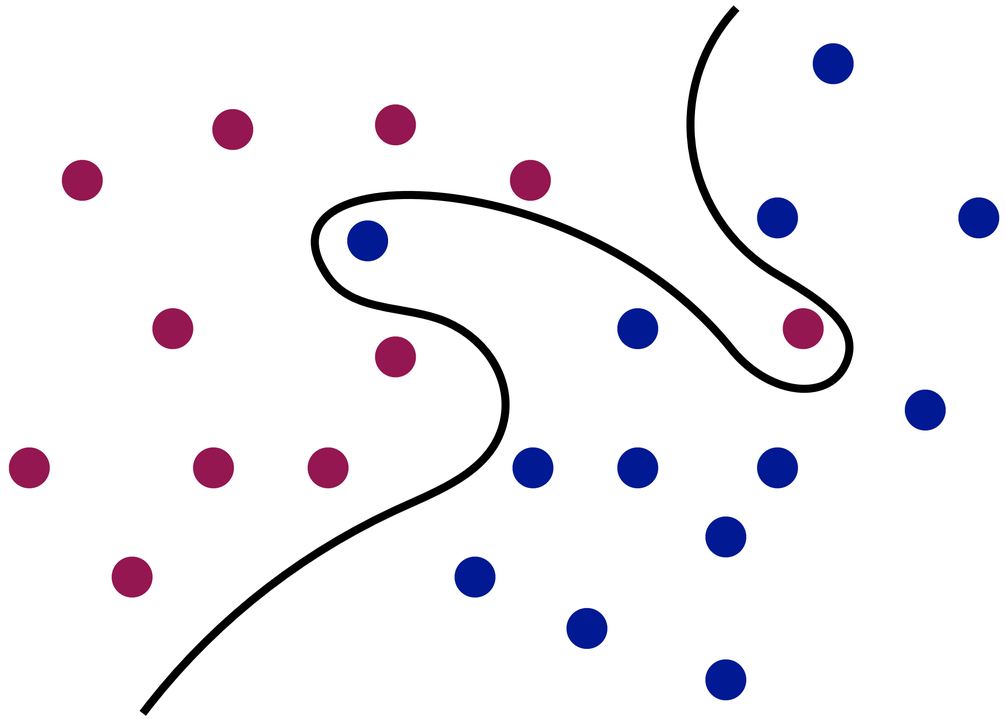
perceptron training algorithm

- Does this algorithm always work?
- Guaranteed to converge if a **linear decision boundary** exists
- But if no decision boundary exists, the algorithm will not converge, not even to an imperfect solution
  - Perceptrons cannot learn non-linear decision boundaries
- So what can?



perceptron training algorithm

- Does this algorithm always work?
- Guaranteed to converge if a **linear decision boundary** exists
- But if no decision boundary exists, the algorithm will not converge, not even to an imperfect solution
  - Perceptrons cannot learn non-linear decision boundaries
- So what can?



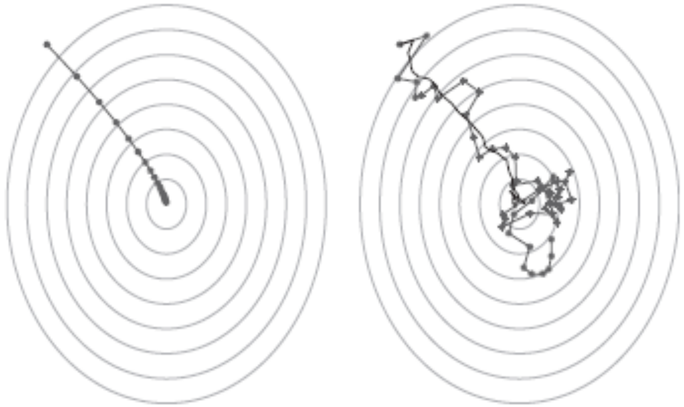
stochastic gradient descent

- Training a perceptron is a slight variant of **stochastic gradient descent (SGD)**
  - Iteratively move in the direction of the gradient until convergence
- We looked at **batch gradient descent** (BGD) previously:

$$w^{(t+1)} = w^{(t)} - \alpha \sum_i \nabla F(x_i, y_i, w^{(t)})$$

  - Consider each datapoint i in each iteration
- With SGD, we only train using the gradient from one input (or a batch of inputs) at a time, rather than across the entire input set

$$w^{(t+1)} = w^{(t)} - \alpha \nabla F(x_t, y_t, w^{(t)})$$

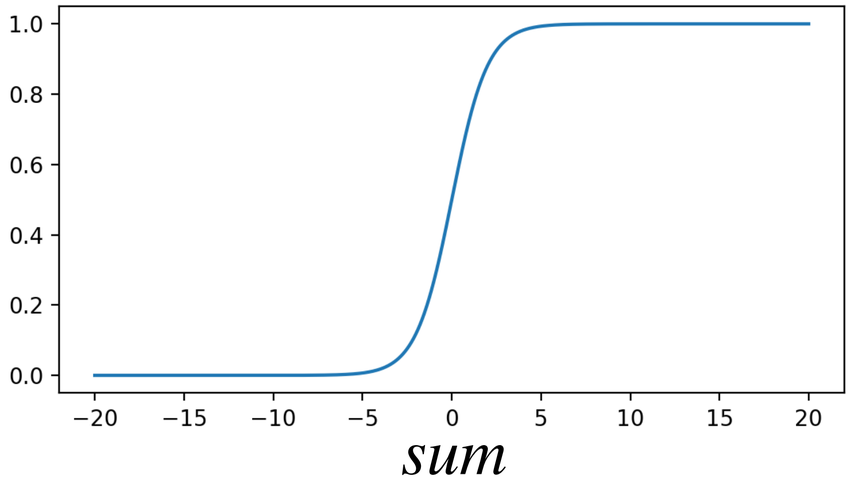


Batch Stochastic

getting to gradient descent

- The perceptron activation function is not differentiable
- Instead of using a step activation, let us use a **sigmoid** as we did in logistic regression
  - There are other choices of activation functions, too: **ReLU**, **ArcTan**, **TanH**, **Softmax**, etc.
  - Remember: function is applied to the weighted sum of the inputs to the neuron
- Recall that for a sigmoid: $f'(x) = f(x)(1 - f(x))$

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



training using gradient descent

- Letting $l(x_i)$ 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:

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

- Want to find out how much to adjust a given weight based on the gradient for that input, so need to find gradients with respect to each weight:

$$\frac{\partial E(\mathbf{x}_i)}{\partial \mathbf{w}_j} = \frac{\partial E(\mathbf{x}_i)}{\partial f(sum)} \cdot \frac{\partial f(sum)}{\partial sum} \cdot \frac{\partial sum}{\partial w_j} = -\left(l(\mathbf{x}_i) - f(sum)\right) \cdot f'(sum) \cdot x_{ij}$$

  - We know how to evaluate $f'(sum)$ for a sigmoid

training using gradient descent

- Learn like this:

$$w_j^{(t+1)} = w_j^{(t)} + \alpha \cdot (l(\mathbf{x}_i) - f(\mathbf{w}^T\mathbf{x}_i)) \cdot f'(\mathbf{w}^T\mathbf{x}_i) \cdot x_{ij}$$

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \alpha \cdot (l(\mathbf{x}_i) - f(\mathbf{w}^T\mathbf{x}_i)) \cdot f'(\mathbf{w}^T\mathbf{x}_i)) \cdot \mathbf{x}_i$$

- Which we will want to write as:

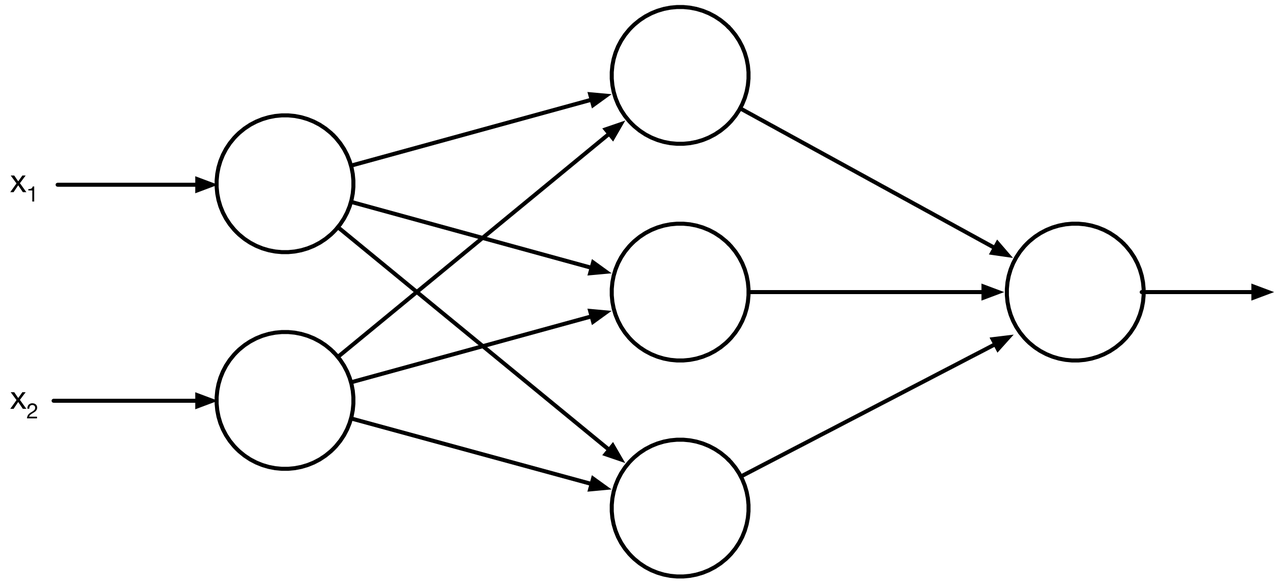
$$\delta_0 = (l(\vec{x}) - f(\vec{w} \cdot \vec{x})) \cdot f'(\vec{w} \cdot \vec{x})$$

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

- This is a case of learning a non-linear decision boundary with gradient descent

learning complex separators

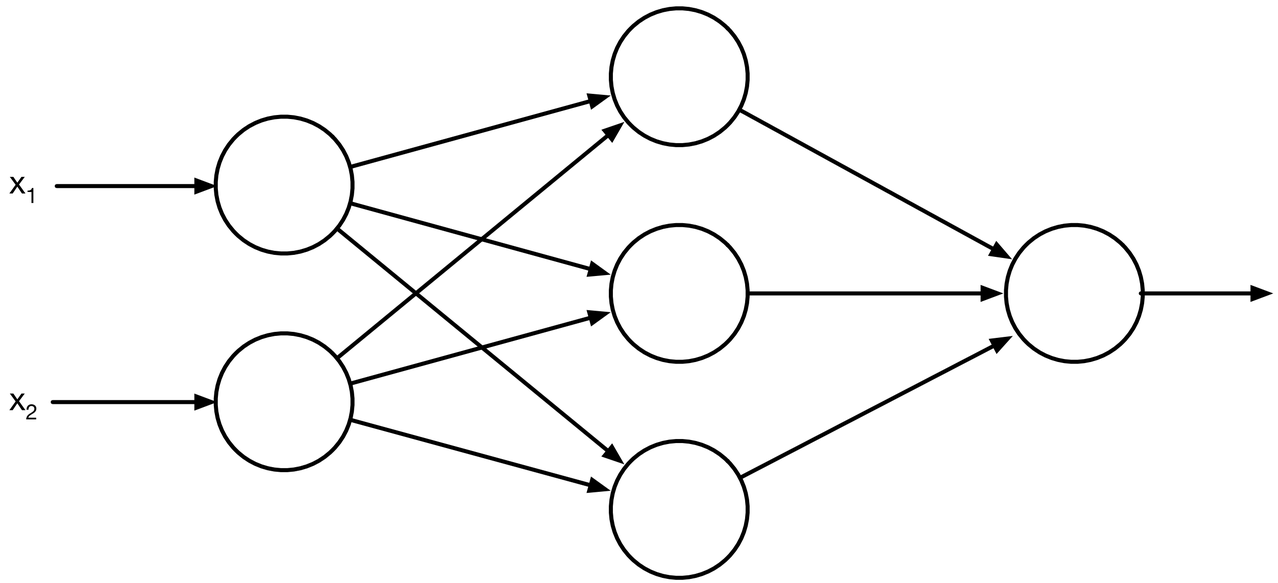
- So far we have only used one neuron
- We can learn more complex decision boundaries by building more complex networks, i.e., cascading neurons
- “Running” the classifier is the same as before
  - Pass weighted sum of inputs through activation function, that output becomes the input to the next neuron
  - This is the **inference** stage
- The **training** stage of learning the weights is a little trickier, but not by much!



learning complex separators

- 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?

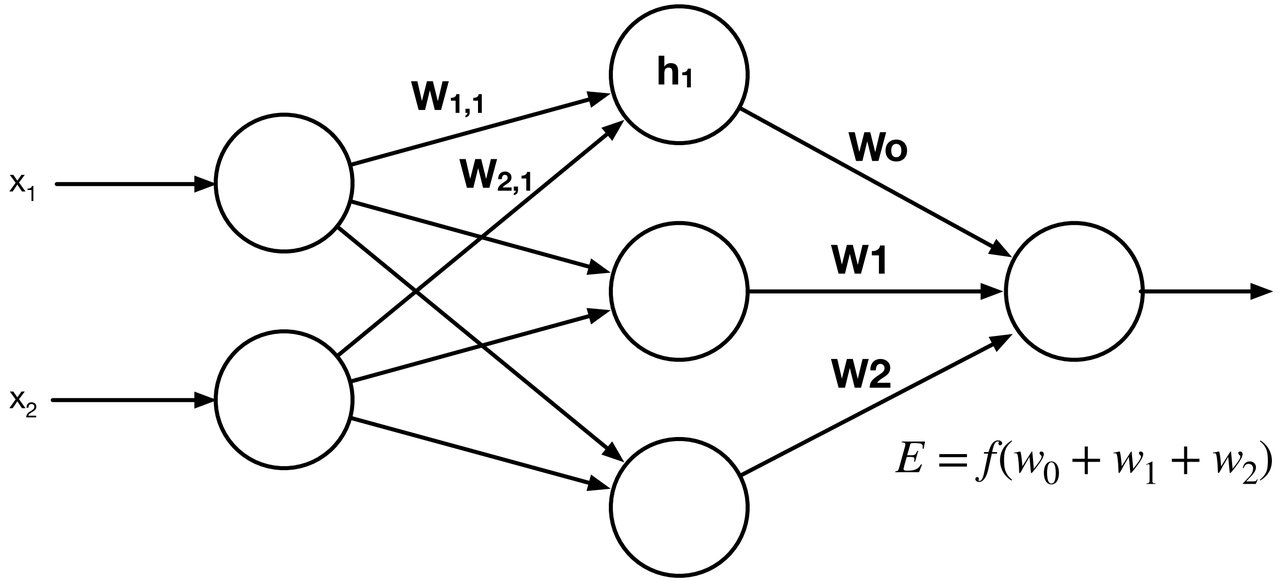


updating the deltas

- Consider hidden neuron h_1, which has output weight w_0 and input weight $w_{1,1}$ from x_1 and $w_{2,1}$ from x_2

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

$$\frac{\partial E}{\partial w_{1,1}} = \frac{\partial E}{\partial w_0} \cdot \frac{\partial w_0}{\partial sum\ h_1} \cdot \frac{\partial\ sumh_1}{\partial w_{1,1}} = \delta_0 \cdot w_o \cdot f'(sum_{h_1}) \cdot x_1 = \delta_{h_1} \cdot x_1$$



$$E = f(w_0 + w_1 + w_2) = f(f(w_{1,1}, w_{2,1}) + w_1 + w_2)$$

essence of backpropagation

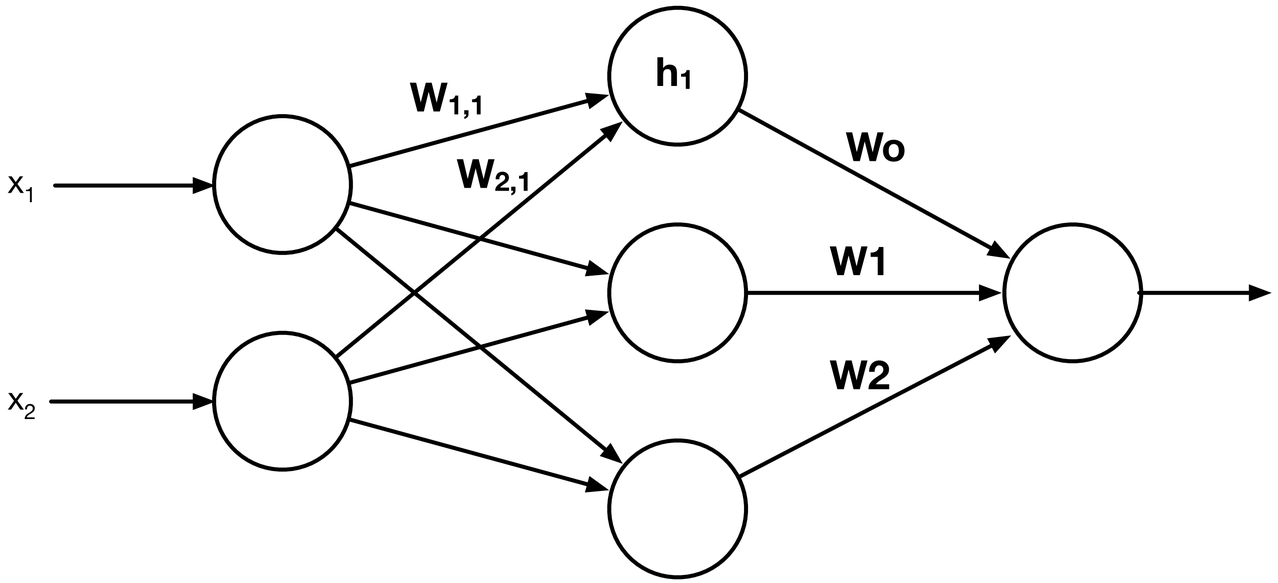
- Computing the gradient for each neuron gives us the delta for the “upstream” neurons, so we can keep pushing error back

- This gives us the essence of **backpropagation** for training neural networks

  - Forward pass: Compute outputs of each neuron

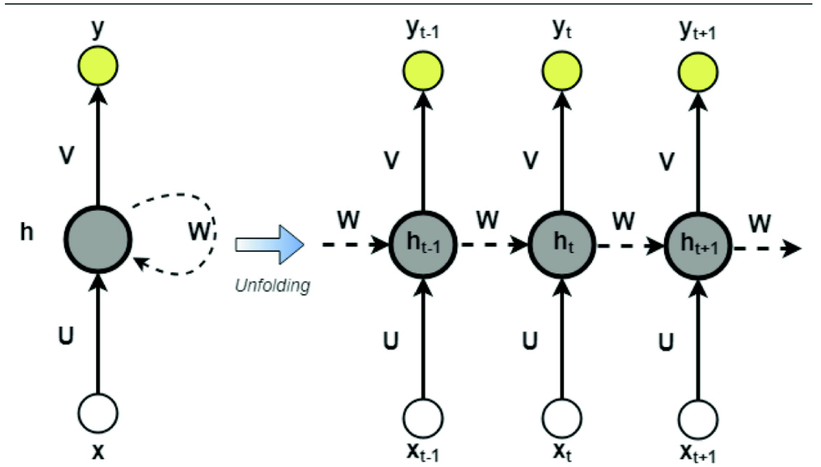
  - Backward pass: Push errors (deltas) weighted by edges to compute how the weights change.

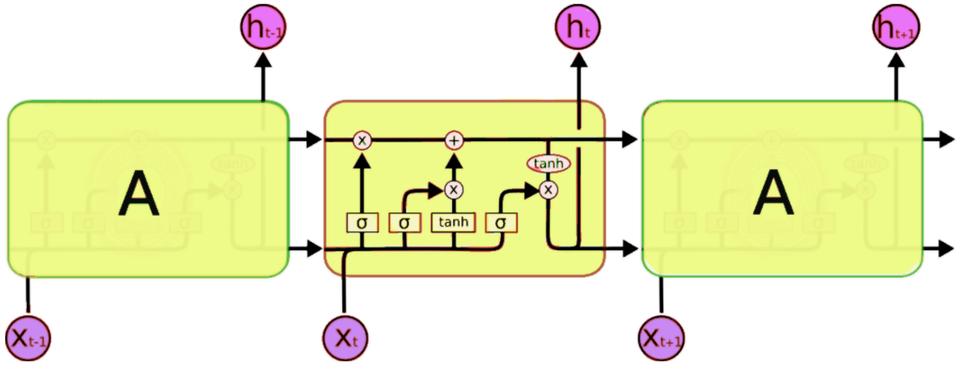
  - Update: Apply stochastic gradient descent to each weight. Repeat.



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
  - Gated Recurrent Units (**GRU**), Hopfield Networks, Boltzmann Machines, Generative Adversarial Networks (**GAN**), …





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

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
  - Finding conditions for global solutions in neural networks