CS5489
Lecture 9.1: Neural Networks and Deep Learning: Part I

Kede Ma

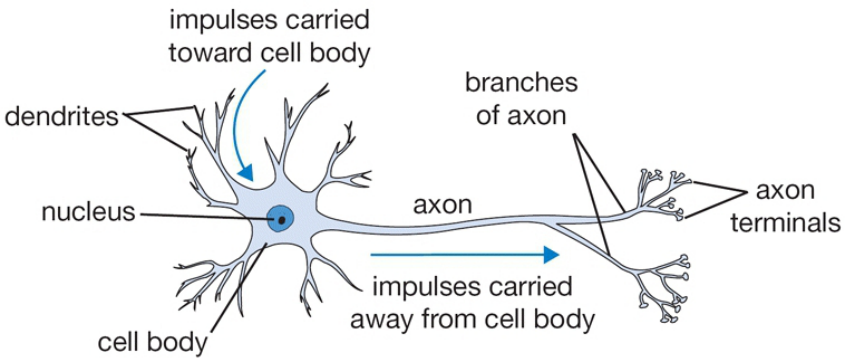
City University of Hong Kong (Dongguan)

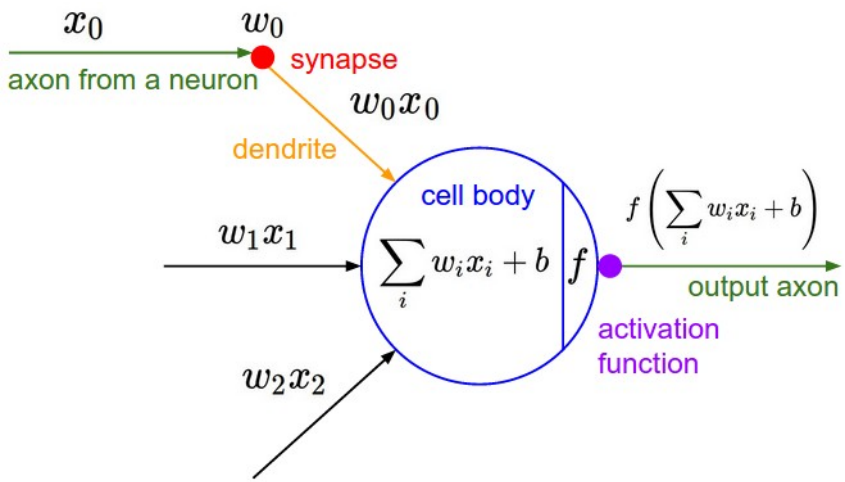
Slide template by courtesy of Benjamin M. Marlin

Outline

Original Idea

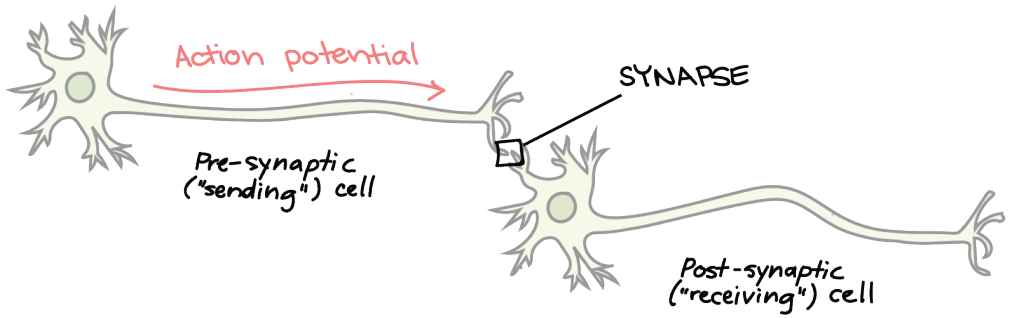
- Perceptron
  - Warren McCulloch and Walter Pitts (1943), Rosenblatt (1957)
- Simulate a neuron in the brain
  1. Take binary inputs (input from nearby neurons)
  2. Multiply by weights (synapses)
  3. Sum and threshold to get binary output (output axon)
- Train weights from data





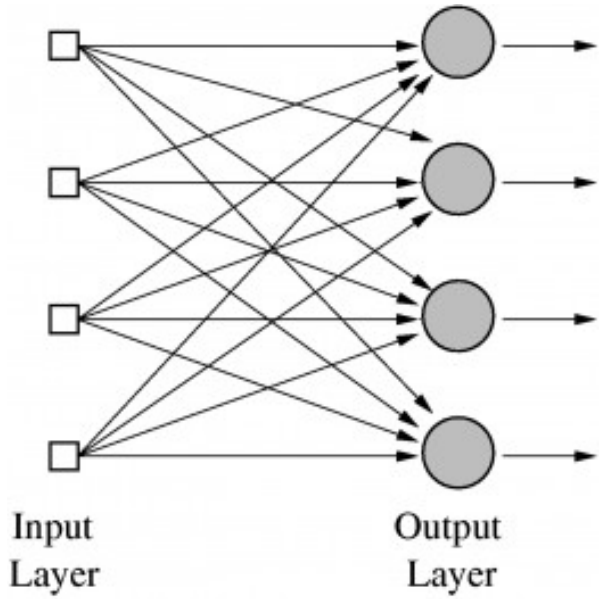
Synapses

- Junctions at which neurons communicate with one another
- Most synapses are chemical (i.e., chemical messengers); other synapses are electrical (i.e., ions flow directly between cells)
- At a chemical synapse, an action potential triggers the presynaptic neuron to release neurotransmitters. These molecules bind to receptors on the postsynaptic cell and make it more or less likely to fire an action potential



What about Multiple Outputs?

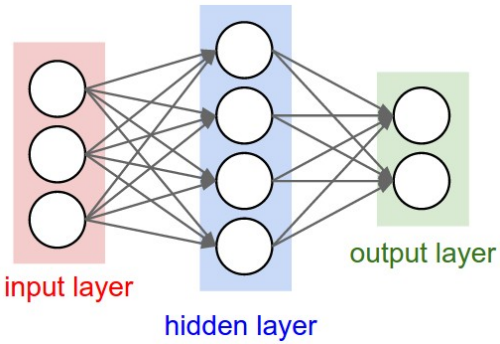
- Multiple outputs handled by using multiple perceptrons

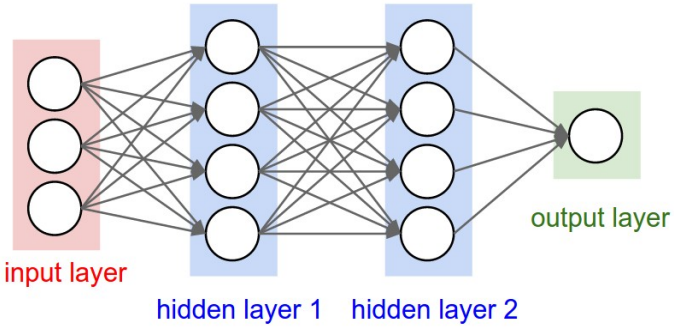


- **Problem**:
  - Linear classifier, can't solve harder problems

Multi-Layer Perceptron

- Add hidden layers between input and output neurons
  - Each layer extracts some features from previous layers
  - Can represent complex non-linear functions
  - Train weights using backpropagation algorithm (1970-80s)
  - Now called a neural network (NN)





- **Problem**:
  - Difficult to train
  - Sensitive to initialization
  - Computationally expensive (at that time)

Decline in the 1990s

- Because of those problems, NN became less popular in the 1990s
- Support vector machines (SVM) had good accuracy
  - Easy to use - only one global optimum
  - Learning is not sensitive to initialization
  - Theory about performance guarantees
- Only a few groups continued to work on neural networks during this time period (LeCun, Bengio, Hinton, and Schmidhuber)
- Highly recommend reading: “Deep Learning: Our Miraculous Year 1990-1991” by Jürgen Schmidhuber

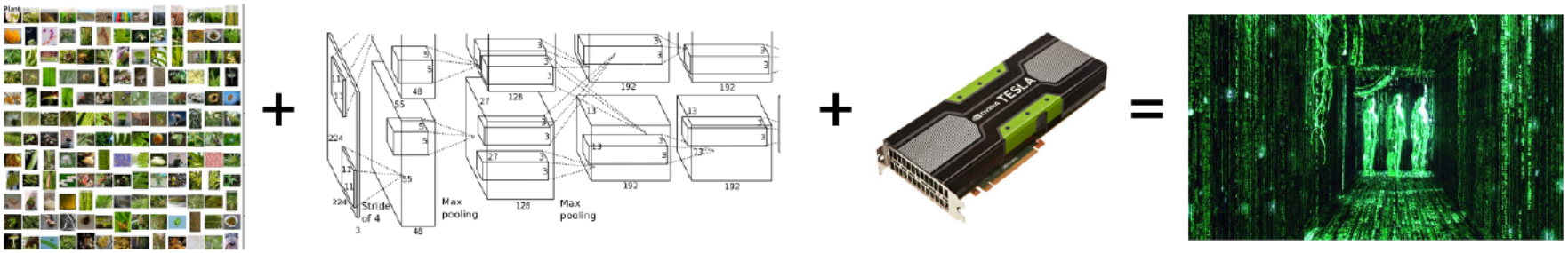
Deep Learning

- NN resurged in the 2000s, due to a number of factors:
  - Improvements in network architecture
    - Developed nodes that are easier to train
  - Better training algorithms
    - Better stochastic optimizers for large-scale nonlinear problems
    - Better ways to prevent overfitting
    - Better initialization methods
  - Infrastructure for faster computing
    - Massively parallel GPUs
    - Distributed computing
  - More labeled data
    - From Internet
    - Crowd-sourcing for labeling data (Amazon Mechanical Turk)

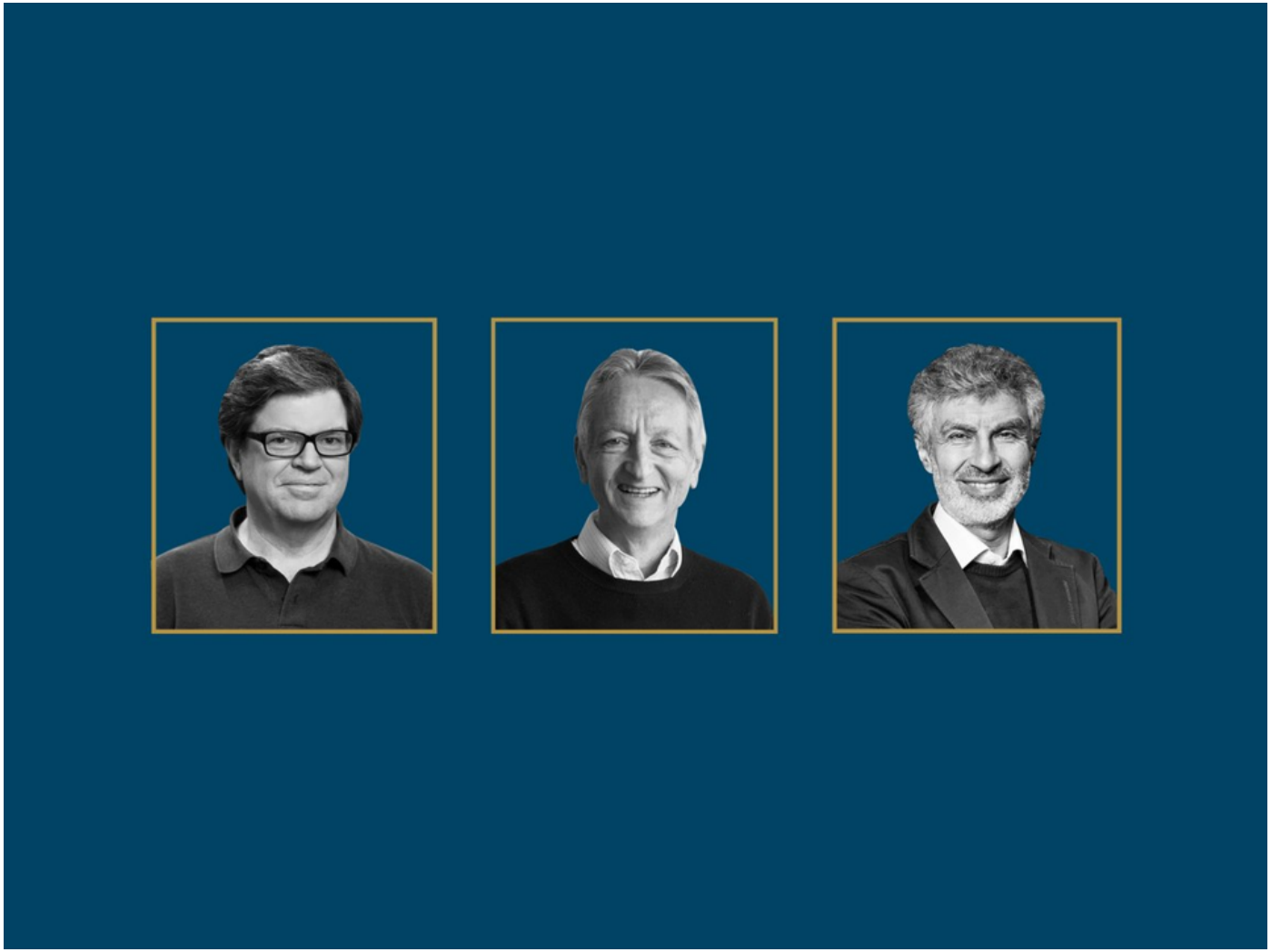
Deep Learning

- We can train NN with more and more layers –> Deep Learning

The Deep Learning "Computer Vision Recipe"



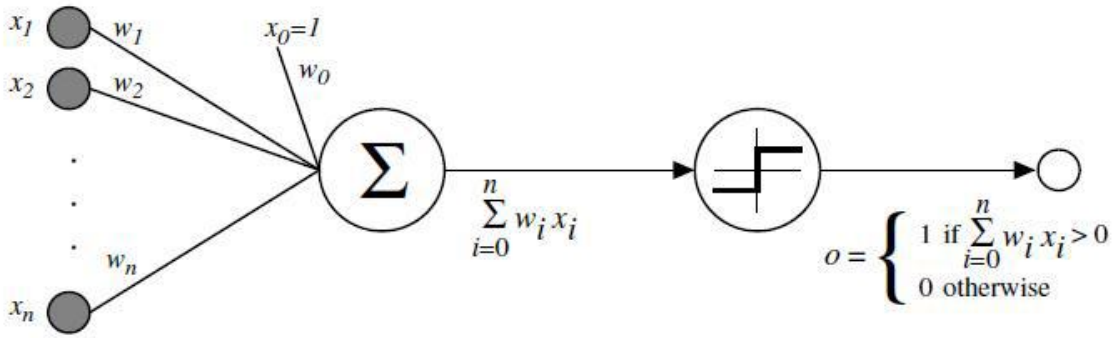
Big Names in Deep Learning



Outline

McCulloch and Pitts Neuron (1943)

- Model a single neuron
  - Input $\mathbf{x} \in \mathbb{R}^N$ is an N-dim vector
  - Apply a weight vector to the inputs
  - Sum and threshold to get the output
- Formally,
  - $y = f(\mathbf{w}^T\mathbf{x}) = f(\sum_{j=0}^{N} w_j x_j)$
  - $\mathbf{w}$ is the weight vector
  - $f(a)$ is the activation function, e.g., $f(a) = \begin{cases} 1, & a > 0 \\ 0, & \text{otherwise} \end{cases}$



The Perceptron (1950)

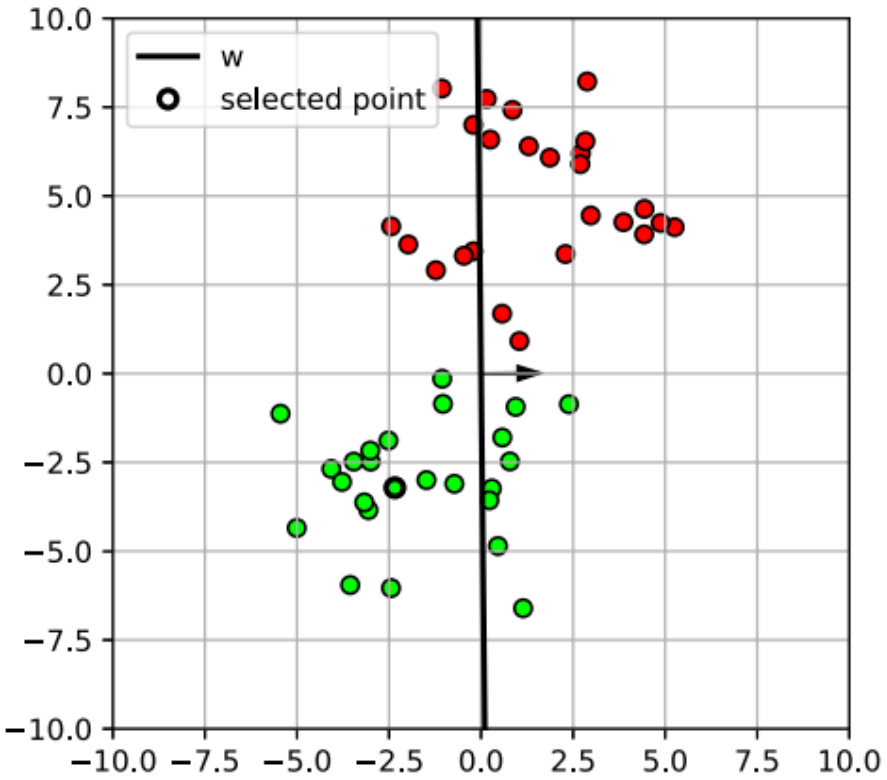
- The Perceptron is a simple algorithm for adapting the weights in a McCulloch/Pitts neuron
  - It was developed in the 1950s by Rosenblatt at Cornell
- Perceptron training criteria:
  - Train the perceptron on data $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{M}$
  - Only look at the points that are misclassified
    - Loss is based on how badly points are misclassified
    - $\ell(\mathbf{w}) = \sum_{i=1}^{M} \begin{cases} -y^{(i)}\mathbf{w}^T\mathbf{x}^{(i)}, & \mathbf{x}^{(i)} \text{ is misclassified} \\ 0, & \text{otherwise} \end{cases}$
  - Minimize the loss: $\mathbf{w}^\star = \text{argmin}_{\mathbf{w}}\, \ell(\mathbf{w})$

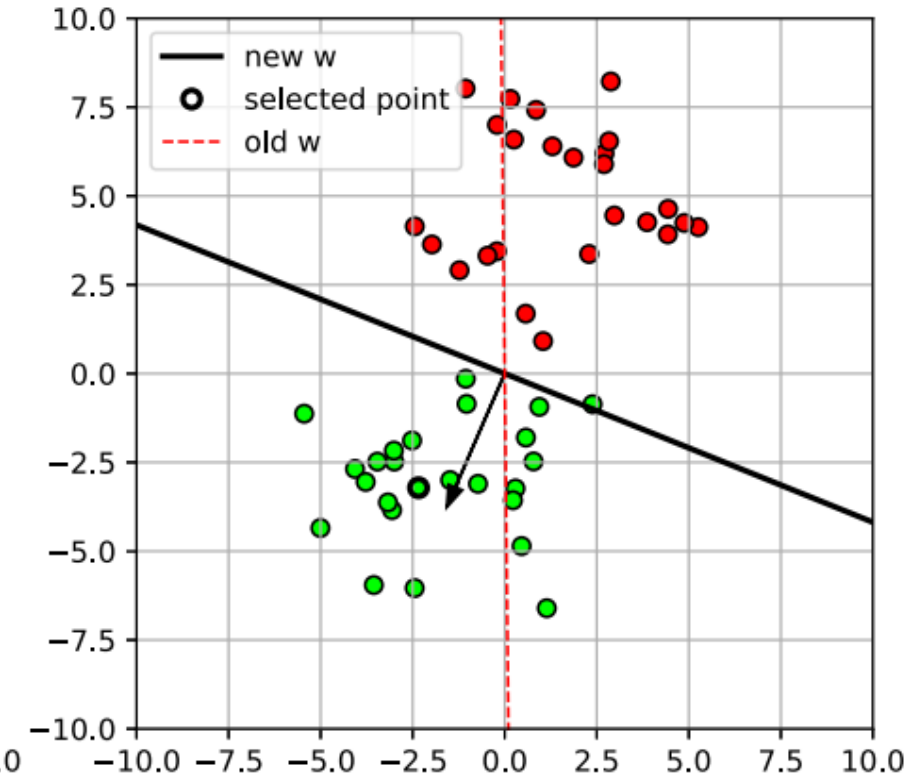
The Perceptron (1950)

- Slow computers back then... only look at one data point at a time
  - I.e., stochastic gradient descent (SGD)
- **Procedure**:
  1. Start with all-zero weight vector $\mathbf{w}^{(0)}$, and initialize t to 0
  2. For all $(\mathbf{x}^{(i)}, y^{(i)}) \in \mathcal{D}$, compute the activation $a^{(i)} = (\mathbf{x}^{(i)})^T \mathbf{w}^{(t)}$
  3. If $y^{(i)} a^{(i)} < 0$, then $\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \alpha y^{(i)} \mathbf{x}^{(i)}$ and $t \leftarrow t + 1$
  4. Repeat Steps 1-3 until no more points are misclassified
- **Notes**:
  - α is the update step (i.e., learning rate) for SGD
    - The effect of the update step is to rotate $\mathbf{w}$ towards/away the misclassified point $\mathbf{x}^{(i)}$
  - If $y^{(i)} = 1$ and $\hat{y}^{(i)} = \text{sign}(a^{(i)}) = -1$, the activation is initially negative and will be increased
  - If $y^{(i)} = -1$ and $\hat{y}^{(i)} = \text{sign}(a^{(i)}) = 1$, the activation is initially positive and will be decreased

Example

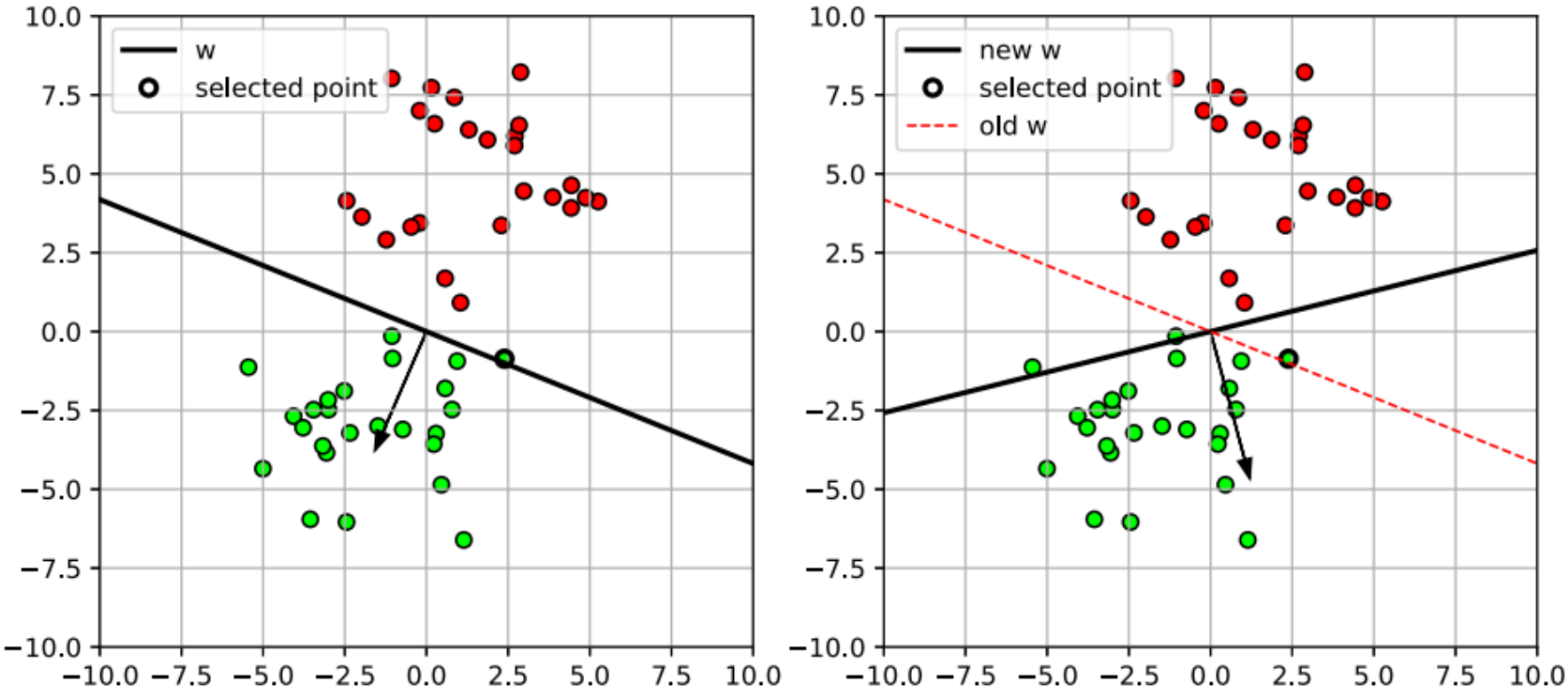
- Iteration 1
  - **w** rotates towards the misclassified point (bold circle)





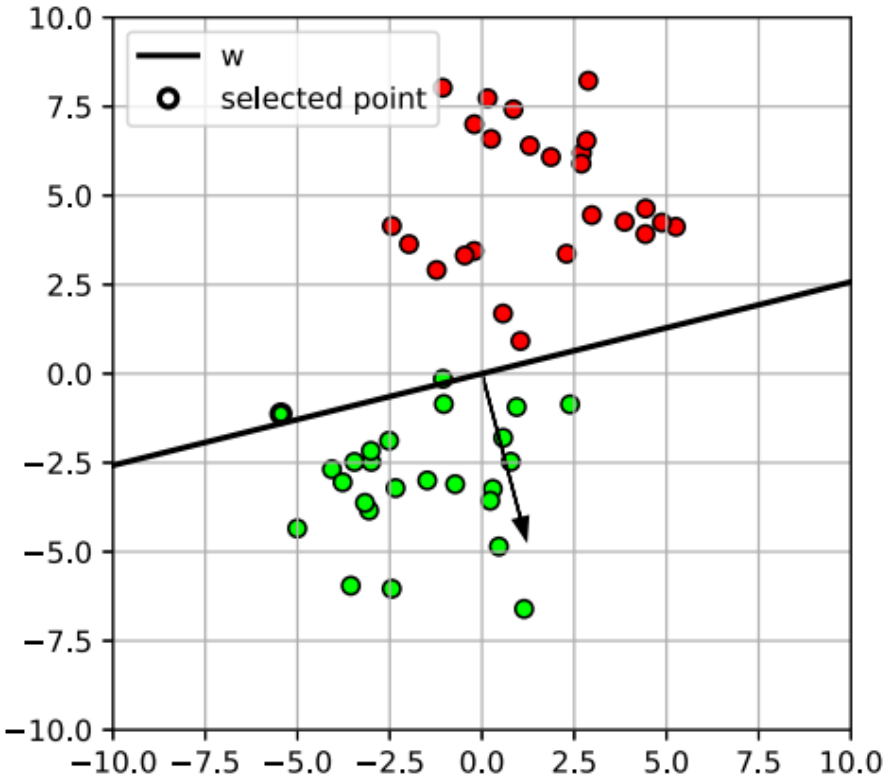
Example

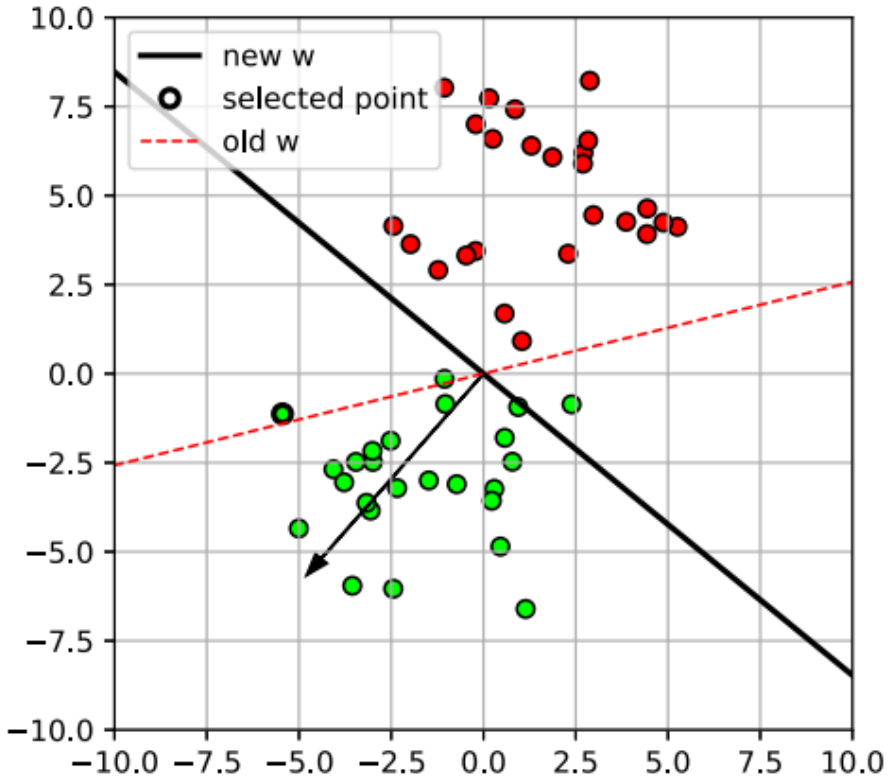
- Iteration 2



Example

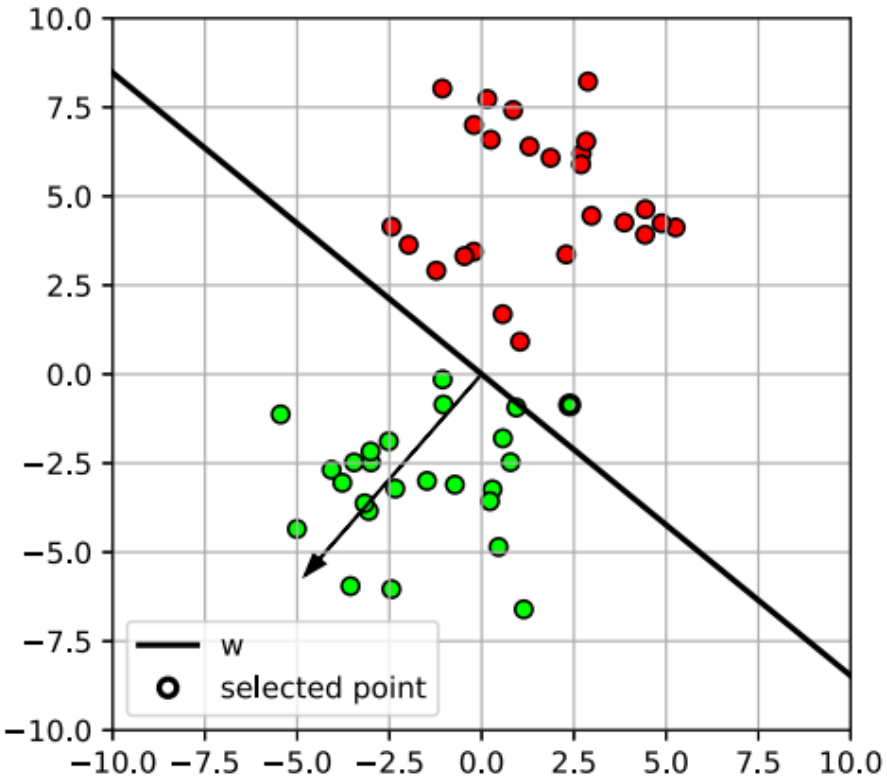
- Iteration 3

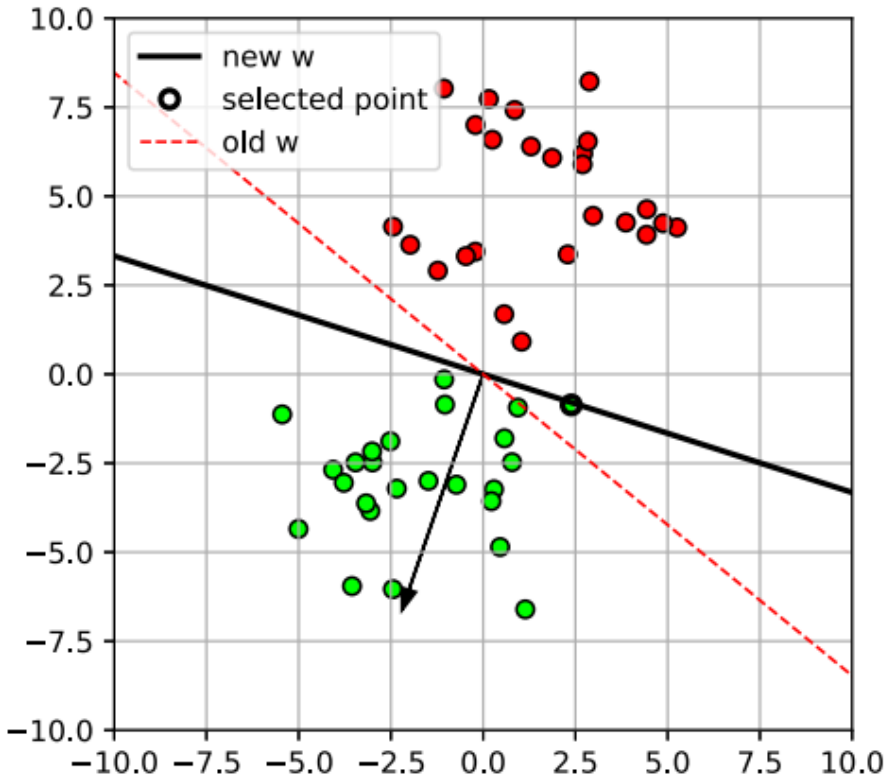




Example

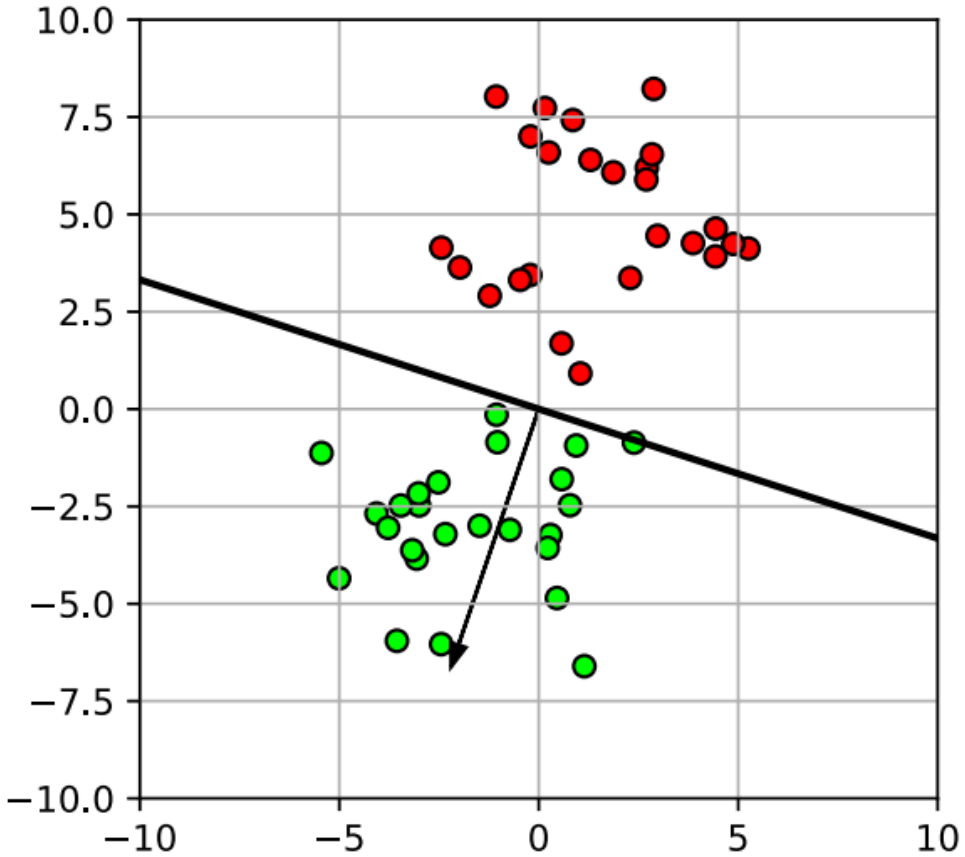
- Iteration 4
  - No more errors





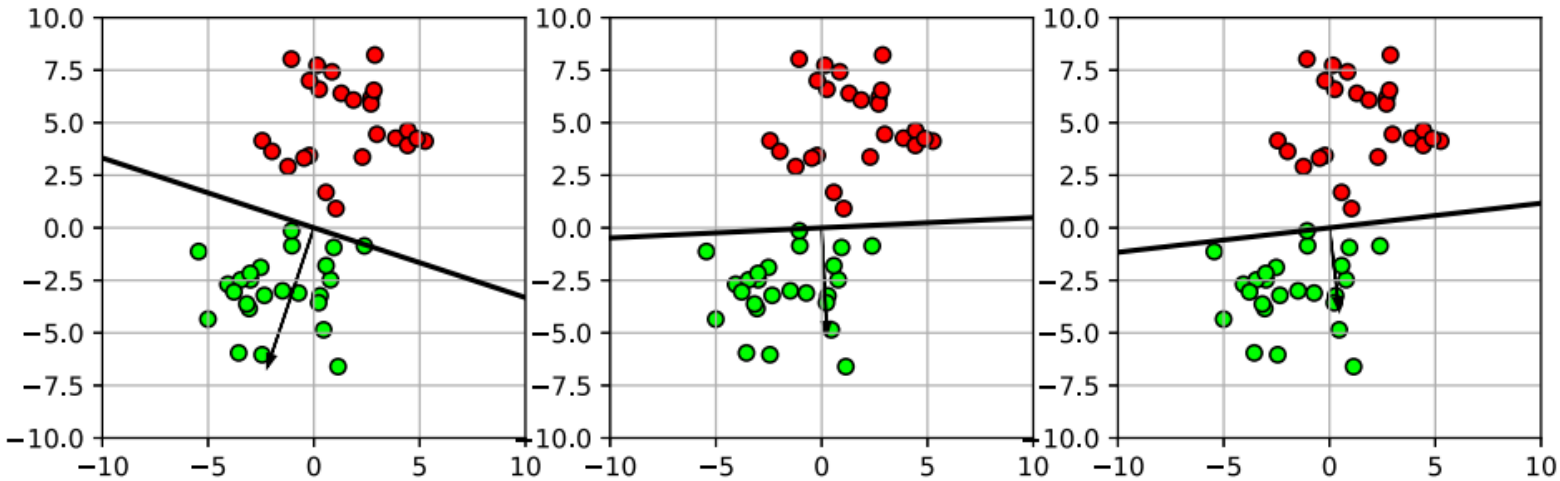
Example

- Final Classifier



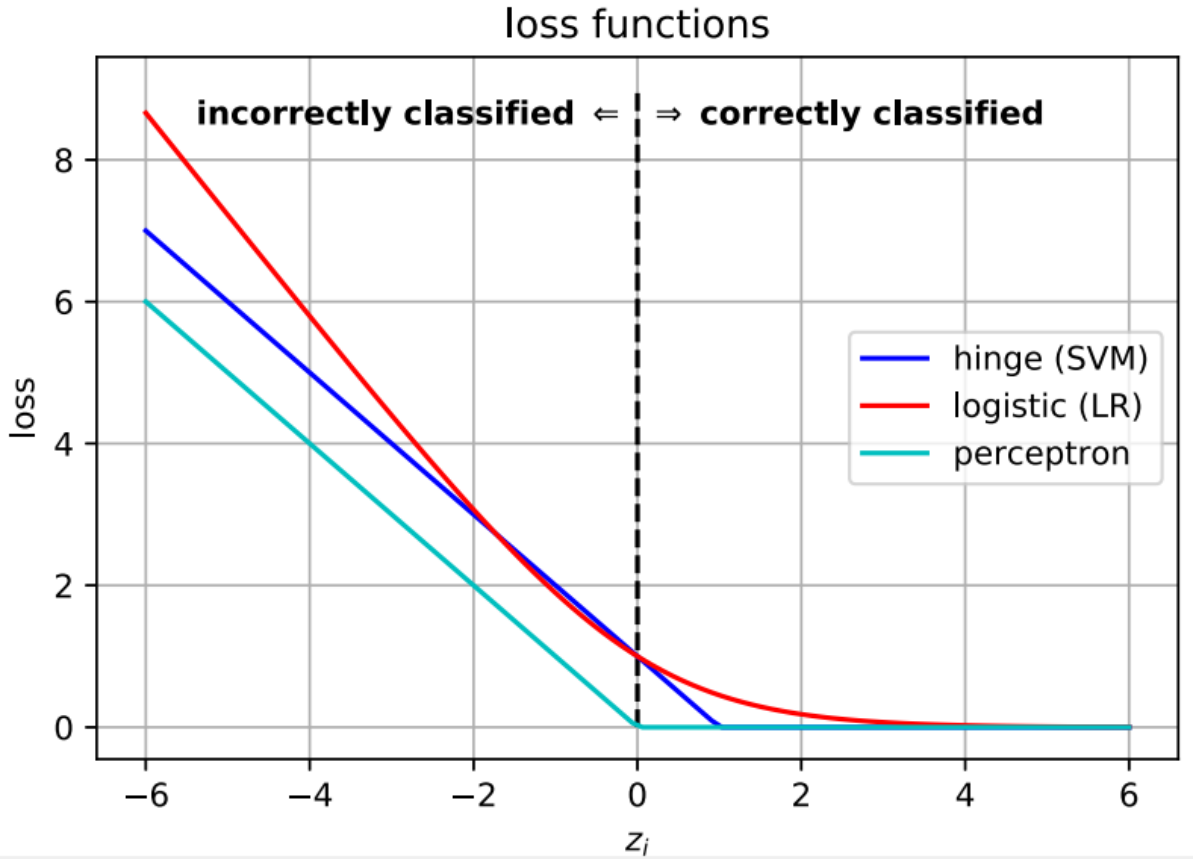
Perceptron Algorithm

- Fails to converge if data is not linearly separable
- Rosenblatt proved that the algorithm will converge if the data is linearly separable
  - The number of iterations is inversely proportional to the separation (margin) between classes
  - This was one of the first machine learning results!
- Different initializations can yield different weights



Perceptron Loss Function

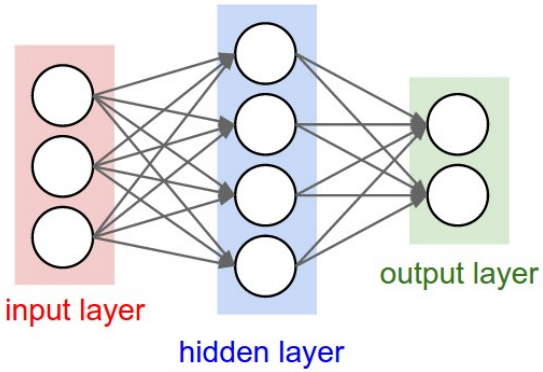
- Define $z^{(i)} = y^{(i)} \mathbf{w}^T \mathbf{x}^{(i)}$
- The loss function is $\ell(z^{(i)}) = \max(0, -z^{(i)})$

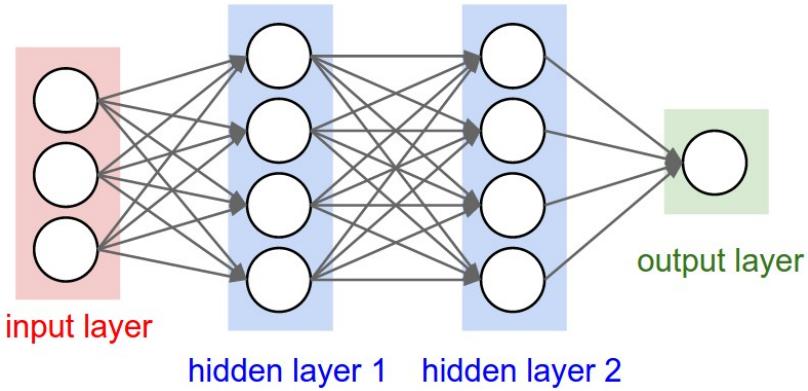


Outline

Multi-Layer Perceptron

- Add hidden layers between the inputs and outputs
  - Each hidden node is a Perceptron (with its own set of weights)
    - Its inputs are the outputs from previous layer
    - Extracts a feature pattern from the previous layer
  - Can model more complex functions





Multi-Layer Perceptron

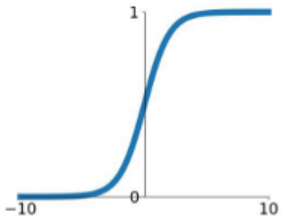
- Formally, for one layer:
  - $\mathbf{h} = f(\mathbf{W}^T \mathbf{x})$
    - Weight matrix $\mathbf{W}$ - one column for each output node
    - Input $\mathbf{x}$ - from previous layer
    - Output $\mathbf{h}$ - to next layer
    - $f(\cdot)$ is the activation function - applied to each dimension to get output

Activation Functions

- There are different types of activation functions

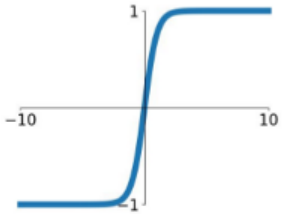
Sigmoid

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



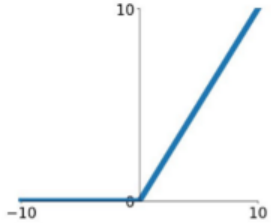
tanh

$$\tanh(x)$$



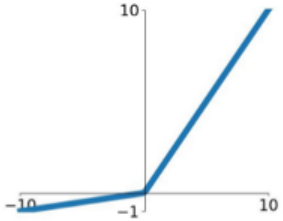
ReLU

$$\max(0, x)$$



Leaky ReLU

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

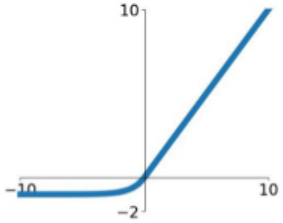


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Sigmoid or Logistic Activation Function

- The sigmoid function is defined as

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

- Sigmoid function translates the input ranged in $[-\infty; +\infty]$ to the range in $(0, 1)$
- A more generalized sigmoid function that is used for multiclass classification called softmax function
- Problems with sigmoid function
  - The $\exp(\cdot)$ function is computationally expensive
  - It has the problem of vanishing gradients

Tanh Activation Function

- The tanh function is defined as

$$\tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

- It is bound to the range $(-1, 1)$
- The gradient is stronger (*i.e.*, steeper) for tanh than sigmoid
- Like sigmoid, tanh also has a vanishing gradient problem
- Optimization is easier for tanh, hence in practice it is always preferred over sigmoid function

ReLU Activation Function

- The ReLU (Rectified Linear Units) function is defined as

$$\text{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{if } x < 0 \end{cases}$$

- ReLU is identity for positive values, and zero for negative values
- It is traditionally known as **half-wave rectification** in signal processing
- Benefits of ReLU:
  - Cheap to compute and easy to optimize
  - It converges faster
  - No vanishing gradient problem
  - Can output a true zero value, leading to representational sparsity
- Problems with ReLU:
  - If one neuron gets negative it is unlikely for it to recover. This is called “dying ReLU” problem

Variants of ReLU

- Leaky ReLU is defined as

$$\text{LReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ ax, & \text{if } x < 0 \end{cases}$$

- Leaky ReLU attempts to fix the "dying ReLU" problem. Instead of the function being zero when $x < 0$, a leaky ReLU gives a small slope

- a is a parameter constrained to be positive. It can either be pre-determined or learned from data

Variants of ReLU

- The ELU (Exponential Linear Unit) is defined as

$$\text{ELU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ a(e^x - 1), & \text{if } x < 0 \end{cases}$$

- It follows the same rule for $x \geq 0$ as ReLU, and increases exponentially for $x < 0$
- ELU tries to make the mean activations closer to zero which speeds up training (by adjusting a)
- Empirically, ELU leads to higher performance

Maxout Activation

- The maxout activation function is defined as

$$\text{Maxout}(\mathbf{x}; \mathbf{w}_1, \mathbf{w}_2) = \max\{\mathbf{w}_1^T\mathbf{x}, \mathbf{w}_2^T\mathbf{x}\}$$

- It is piecewise linear
- The maxout activation is a generalization of ReLU and leaky ReLU. It is a learnable activation function
- The maxout neuron, therefore, enjoys all the benefits of ReLU (linear regime of operation, no saturation) and does not have its drawbacks (dying ReLU)
- However, it increases the total number of parameters for each neuron and hence, a higher total number of parameters need to be trained

Other Emerging Activation Functions

- Gaussian Error Linear Unit (GELU):

$$\text{GELU}(x) = x\Phi(x)$$

  - Φ is the standard Normal cumulative distribution function (CDF)
- Softplus:

$$\text{Softplus}(x) = \log(1 + \exp(x))$$

  - As a continuous and differentiable approximation to ReLU
- Swish:

$$\text{Swish}(x) = x \cdot \text{sigmoid}(\beta x) = \frac{x}{1 + \exp(-\beta x)}$$

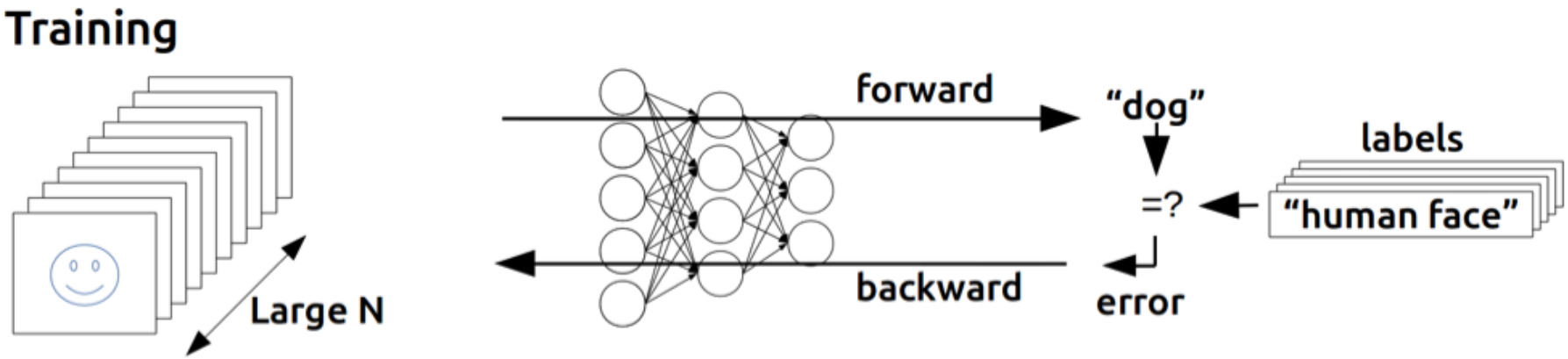
  - β is either constant or a trainable parameter

Training an MLP

- For classification, we use the cross-entropy function as the loss
  - $\ell = -\sum_{j=1}^{C} y_j \log \hat{y}_j$
    - y_j is 1 for the true class, and 0 otherwise
    - $\hat{y}_j$ is the softmax output for the j-th class
- Use (stochastic) gradient descent as the optimization tool
  - $w_{ij} \leftarrow w_{ij} - \alpha \frac{d\ell}{dw_{ij}}$
    - Layer i, node j
  - α is the learning rate, which controls convergence rate
    - too small –> converges very slowly
    - too large –> possibly doesn't converge

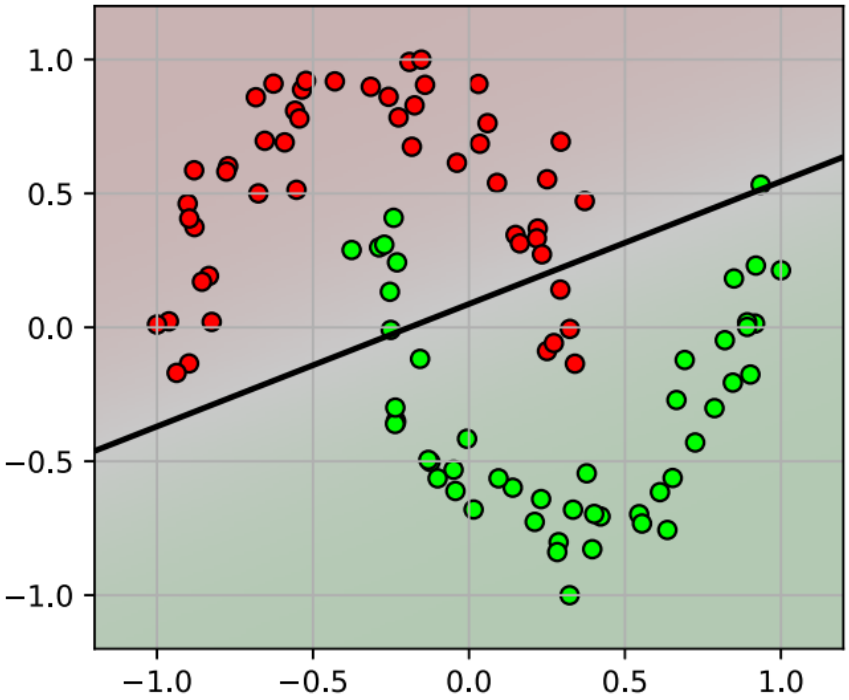
Backpropagation (Backward Propagation)

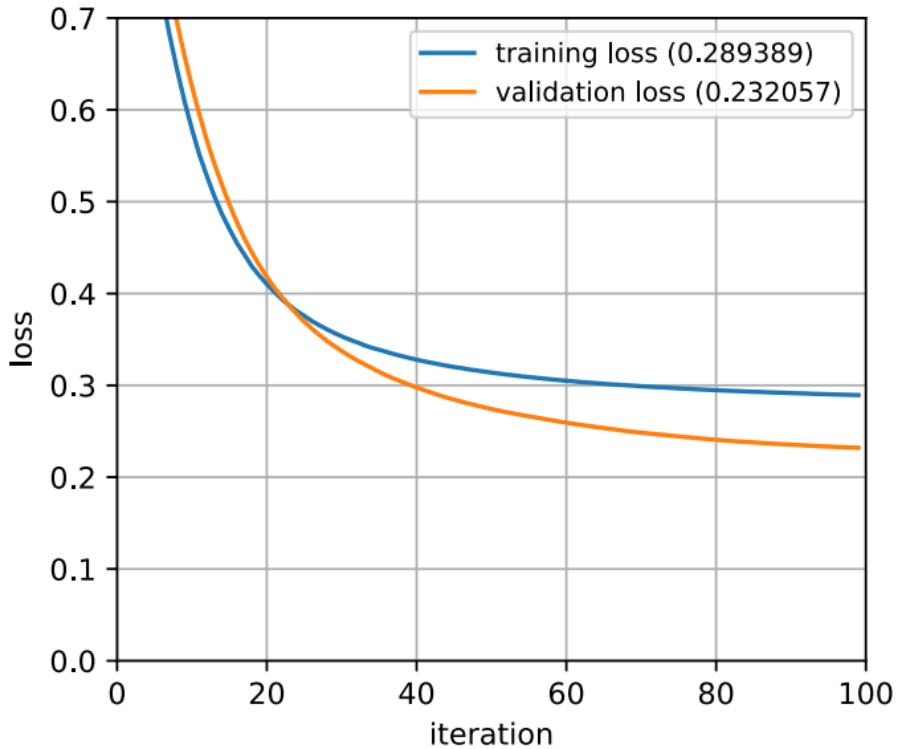
- Do a forward pass to calculate the prediction
- Do a backward pass to update weights that were responsible for an error



Example

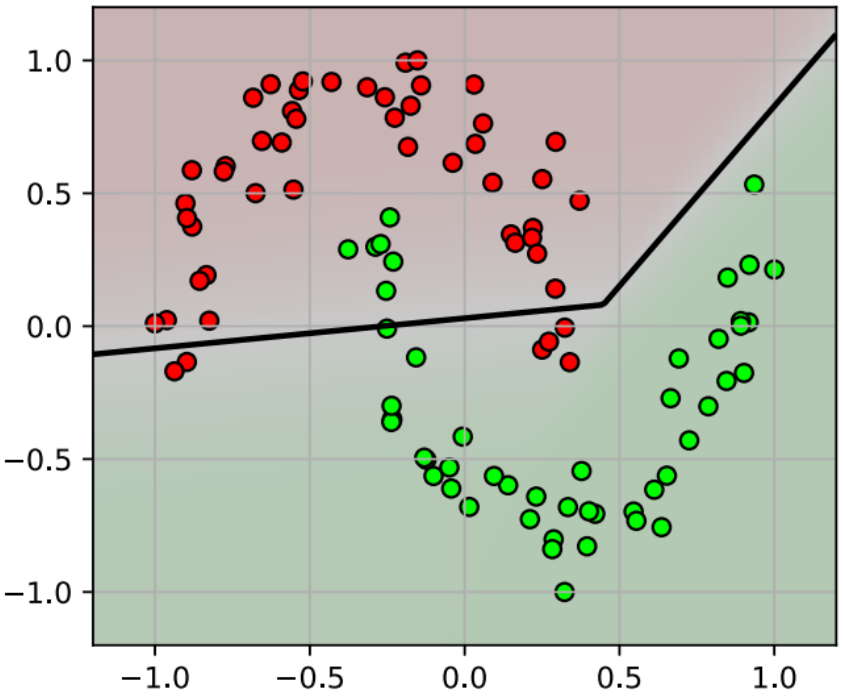
- Train 1 NN with just one output layer
  - This is the same as logistic regression
- Training and validation losses have converged

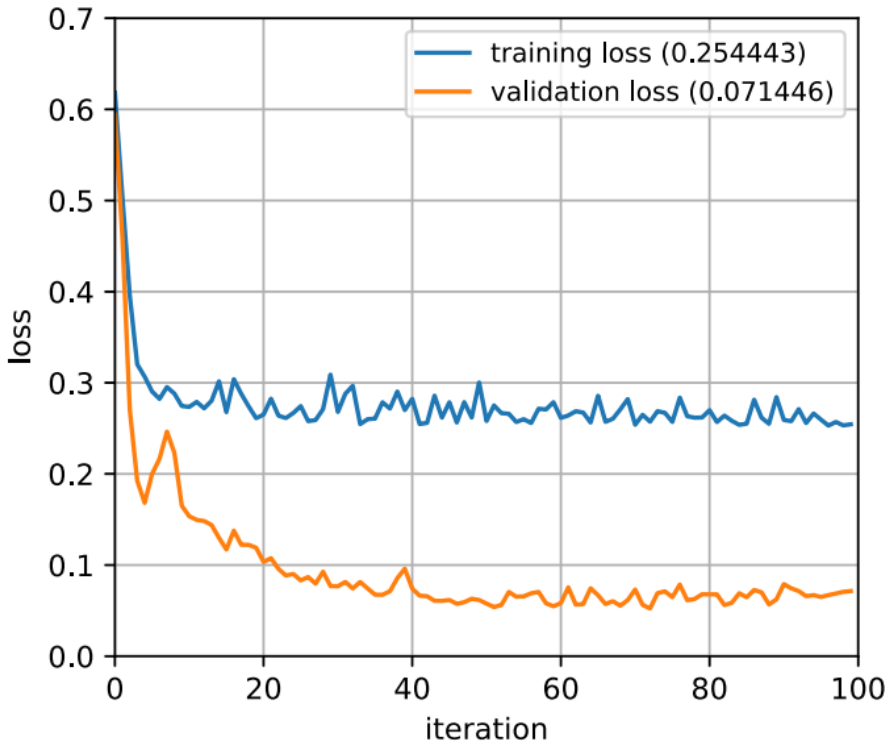




Example

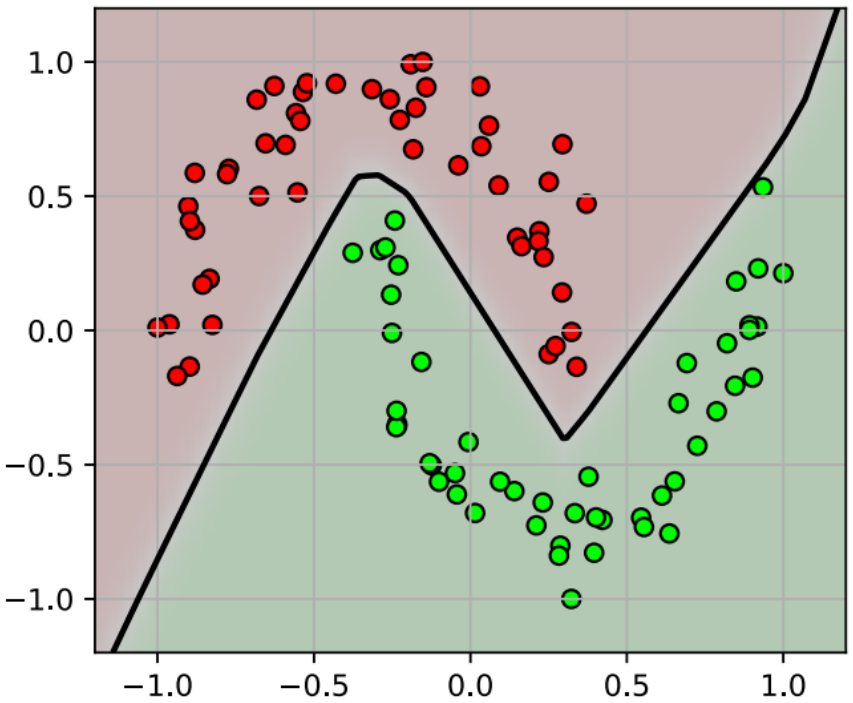
- Add one 1 hidden layer with 2 ReLU nodes
  - Can carve out part of the red class

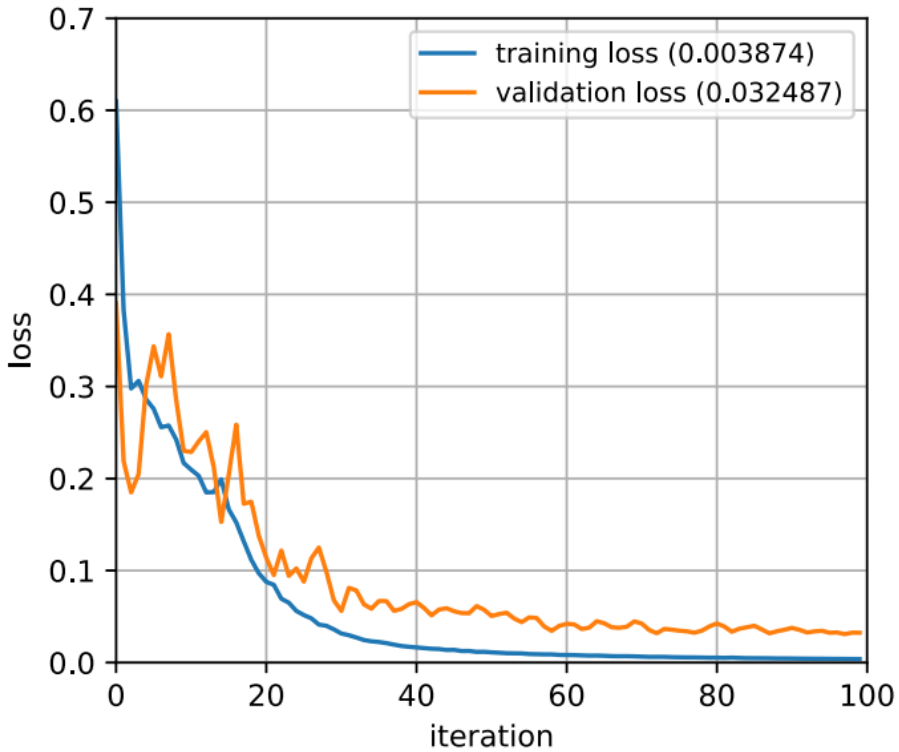




Example

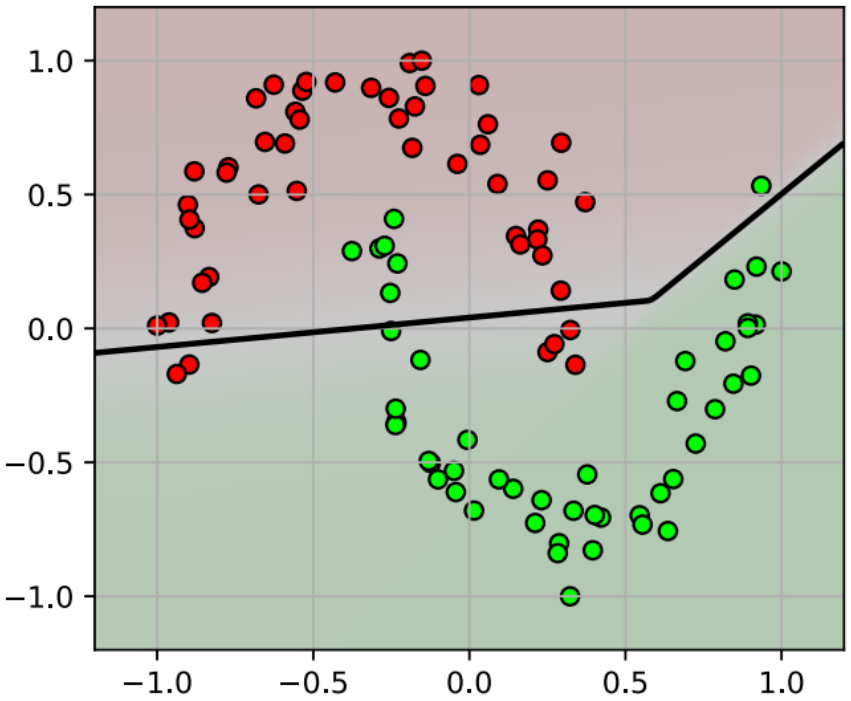
- Let's try more nodes
  - 1 hidden layer with 20 hidden nodes
  - With enough nodes, we can get a perfect classifier

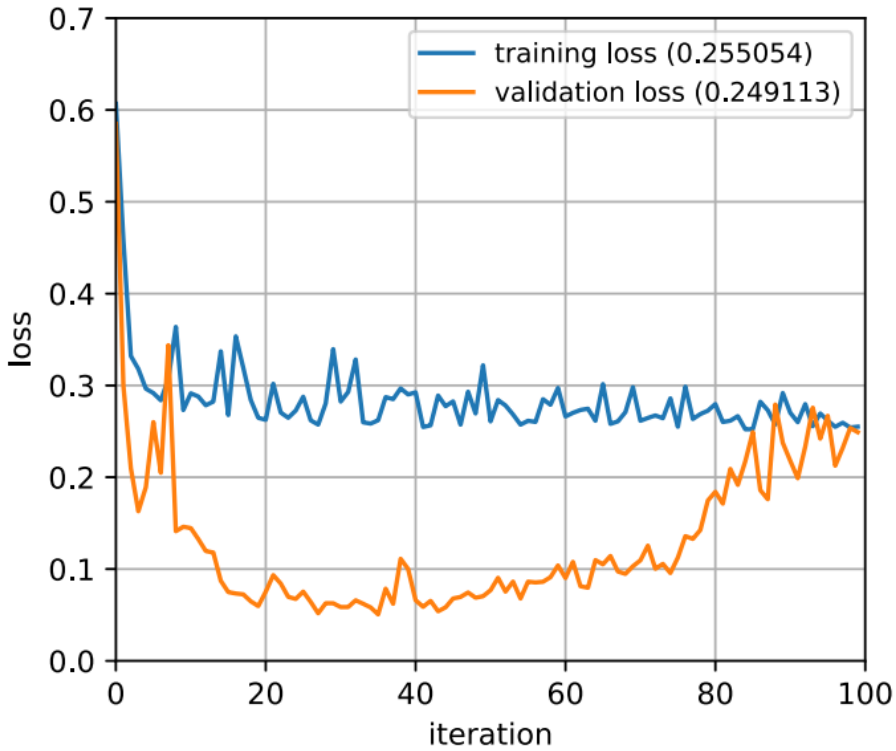




Overfitting

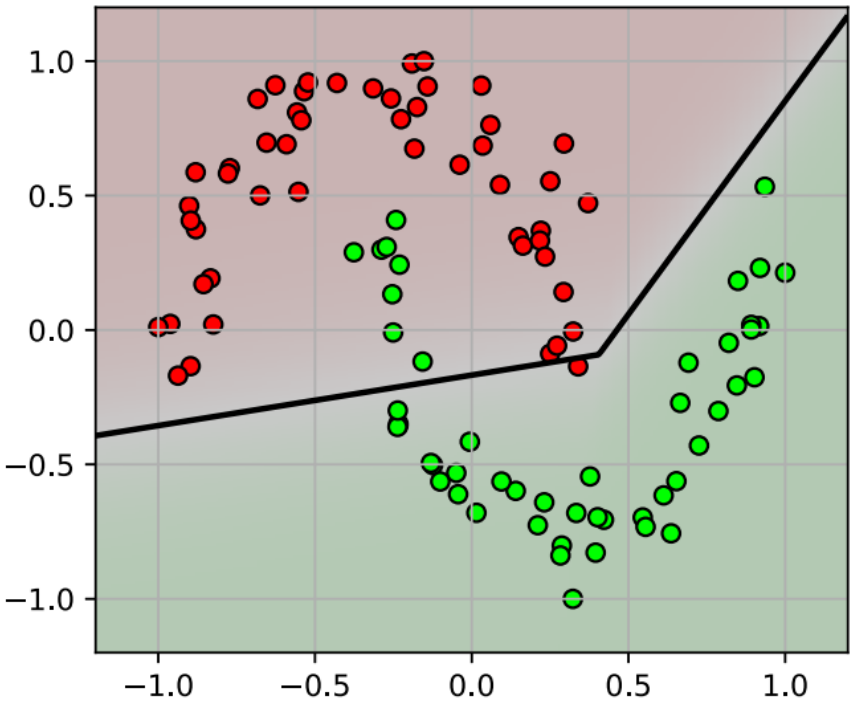
- Continuous training will sometimes lead to overfitting
  - The training loss decreases, but the validation loss increases

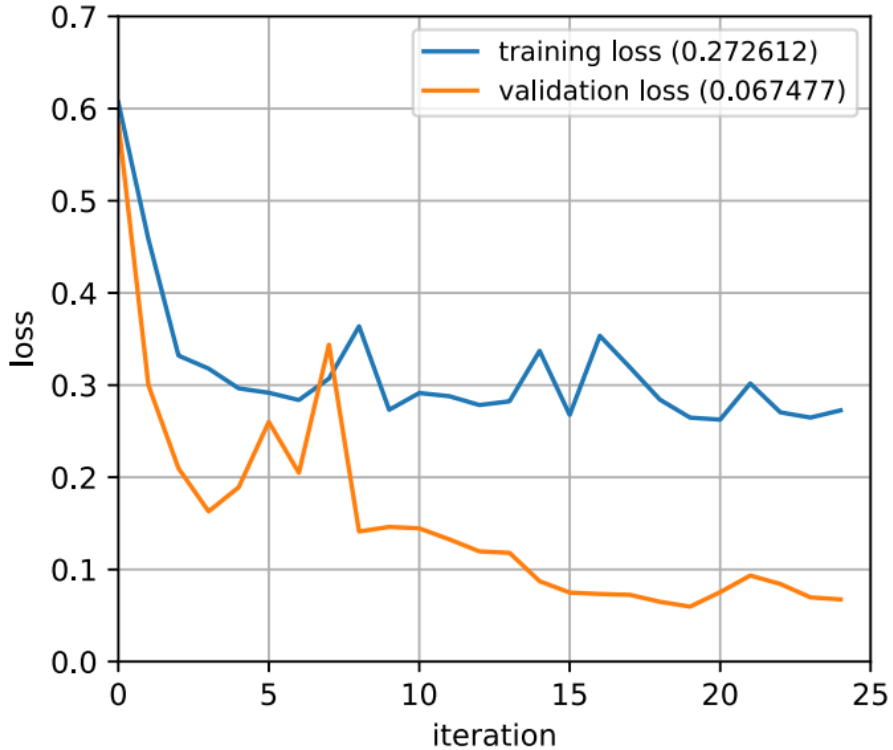




Early Stopping

- Training can stop when the validation loss is stable for a number of iterations
  - Stable means change below a threshold
  - This is to prevent overfitting the training data
  - We can limit the number of iterations





Universal Approximation Theorem

- It can be shown that a sigmoid network with one hidden layer (of infinite nodes) is a universal function approximator
- In terms of classification, this means neural networks with one hidden layer (of unbounded size) can represent any decision boundary and thus have infinite capacity
- It was also shown that deep networks can be more efficient at representing certain types of functions than shallow networks
- It is not the specific choice of the activation function, but rather the multi-layer feedforward architecture itself which gives neural networks the potential of being universal approximators
- It does not touch upon the algorithmic learnability of those parameters