# Classifiers We’ve Covered So Far

<table>
<thead>
<tr>
<th>Classifier</th>
<th>decision boundary?</th>
<th>difficult part of learning?</th>
</tr>
</thead>
<tbody>
<tr>
<td>decision trees</td>
<td>piecewise-axis-aligned</td>
<td>greedy split decisions</td>
</tr>
<tr>
<td>$K$-nearest neighbors</td>
<td>possibly very complex</td>
<td>indexing training data</td>
</tr>
<tr>
<td>perceptron</td>
<td>linear</td>
<td>iterative optimization method required</td>
</tr>
<tr>
<td>logistic regression</td>
<td>linear</td>
<td>iterative optimization method required</td>
</tr>
<tr>
<td>naïve Bayes</td>
<td>linear (see A4)</td>
<td>none</td>
</tr>
</tbody>
</table>
Classifiers We’ve Covered So Far

<table>
<thead>
<tr>
<th></th>
<th>decision boundary?</th>
<th>difficult part of learning?</th>
</tr>
</thead>
<tbody>
<tr>
<td>decision trees</td>
<td>piecewise-axis-aligned</td>
<td>greedy split decisions</td>
</tr>
<tr>
<td>$K$-nearest neighbors</td>
<td>possibly very complex</td>
<td>indexing training data</td>
</tr>
<tr>
<td>perceptron</td>
<td>linear</td>
<td>iterative optimization method required</td>
</tr>
<tr>
<td>logistic regression</td>
<td>linear</td>
<td>iterative optimization method required</td>
</tr>
<tr>
<td>naïve Bayes</td>
<td>linear (see A4)</td>
<td>none</td>
</tr>
</tbody>
</table>

The next methods we’ll cover permit **nonlinear** decision boundaries.
Inspiration from Neurons

Image from Wikimedia Commons.

Input signals come in through dendrites, output signal passes out through the axon.
Neuron-Inspired Classifiers


Activation

Fire, or not?

Output

Bias parameter
Neuron-Inspired Classifiers


Input nodes: \( x[1], x[2], x[3], \ldots, x[d] \)

Weights: \( w[1], w[2], w[3], \ldots, w[d] \)

Bias: \( b \)

Output: \( \hat{y} \)
Neuron-Inspired Classifiers

The diagram illustrates the process of a neuron-inspired classifier. The input $x_n$ is multiplied by the weights $w$ and added to the bias $b$ to produce the pre-activation value $\sum$. This value is then passed through an "activation" function (not explicitly shown here) to produce the classifier output $\hat{y}$. The correct output $y_n$ is compared to the classifier output to compute the loss $L_n$. The weights and bias are adjusted to minimize this loss in the training process.

Mathematically, this can be represented as:

$$\hat{y} = \sum \text{activation}(x_n w + b)$$

where $\sum$ is the summation operation.

The goal is to minimize the loss $L_n$ to improve the classifier's performance.

---

Note: The diagram and equations are simplified for clarity and do not include all the details of a neural network.
Neuron-Inspired Classifiers

Hyperbolic tangent function, \( \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \).

Generalization: apply elementwise to a vector, so that \( \tanh : \mathbb{R}^k \to (-1, 1)^k \).
Neuron-Inspired Classifiers

The diagram illustrates a neuron-inspired classifier for pattern classification. The process begins with an input vector $\mathbf{x}_n$ that is multiplied by weights $\mathbf{w}_1$ and then summed with bias $b_1$, resulting in the output of the first layer. This output is then passed through a tanh activation function, leading to a sum with another bias $b_2$ and weight $\mathbf{w}_2$, followed by another tanh activation function. The final output $\hat{y}$ is compared to the correct output $y_n$ to compute the loss $L_n$.

The diagram uses nodes to represent the input, weights, biases, and activation functions, with edges showing the flow of data. The notation for the process is as follows:

- $\mathbf{x}_n$: Input vector
- $\mathbf{w}_1$: Weights for the first layer
- $\mathbf{w}_2$: Weights for the second layer
- $b_1$: Bias for the first layer
- $b_2$: Bias for the second layer
- $\mathbf{v}_1$: Weights for the second layer
- $\mathbf{v}_2$: Weights for the second layer
- $\tanh$: Activation function
- $\times$: Multiplication
- $\Sigma$: Summation
- $y_n$: Correct output
- $L_n$: Loss
- $\hat{y}$: Classifier output
Neuron-Inspired Classifiers

\[ x_n W \times \sum b \rightarrow \text{tanh} \rightarrow \sum v \rightarrow \text{activation} \rightarrow y \rightarrow L_n \rightarrow \text{loss} \]

input, weights, "hidden units", "activation", classifier output, "f"
Two-Layer Neural Network

\[ f(x) = \text{sign}\left( \sum_{h=1}^{H} v_h \cdot \tanh(w_h \cdot x + b_h) \right) \]

\[ = \text{sign}(v \cdot \tanh(Wx + b)) \]

▶ Two-layer networks allow decision boundaries that are nonlinear.

▶ It's fairly easy to show that "XOR" can be simulated (recall conjunction features from the "practical issues" lecture on 10/18).

▶ Theoretical result: any continuous function on a bounded region in \( \mathbb{R}^d \) can be approximated arbitrarily well, with a finite number of hidden units.

▶ The number of hidden units affects how complicated your decision boundary can be and how easily you will overfit.
Two-Layer Neural Network

\[ f(x) = \text{sign} \left( \sum_{h=1}^{H} v_h \cdot \tanh (w_h \cdot x + b_h) \right) \]

\[ = \text{sign} (v \cdot \tanh (Wx + b)) \]

- Two-layer networks allow decision boundaries that are nonlinear.
Two-Layer Neural Network

\[ f(x) = \text{sign} \left( \sum_{h=1}^{H} v_h \cdot \tanh (w_h \cdot x + b_h) \right) \]

\[ = \text{sign} (v \cdot \tanh (Wx + b)) \]

- Two-layer networks allow decision boundaries that are nonlinear.
- It’s fairly easy to show that “XOR” can be simulated (recall conjunction features from the “practical issues” lecture on 10/18).
Two-Layer Neural Network

\[ f(x) = \text{sign} \left( \sum_{h=1}^{H} v_h \cdot \tanh (w_h \cdot x + b_h) \right) \]

\[ = \text{sign} (v \cdot \tanh (Wx + b)) \]

- Two-layer networks allow decision boundaries that are nonlinear.
- It’s fairly easy to show that “XOR” can be simulated (recall conjunction features from the “practical issues” lecture on 10/18).
- Theoretical result: any continuous function on a bounded region in \( \mathbb{R}^d \) can be approximated arbitrarily well, with a finite number of hidden units.
Two-Layer Neural Network

\[
f(x) = \text{sign} \left( \sum_{h=1}^{H} v_h \cdot \tanh (w_h \cdot x + b_h) \right)
\]

\[
= \text{sign} (v \cdot \tanh (Wx + b))
\]

- Two-layer networks allow decision boundaries that are nonlinear.
- It’s fairly easy to show that “XOR” can be simulated (recall conjunction features from the “practical issues” lecture on 10/18).
- Theoretical result: any continuous function on a bounded region in \( \mathbb{R}^d \) can be approximated arbitrarily well, with a finite number of hidden units.
- The number of hidden units affects how complicated your decision boundary can be and how easily you will overfit.
Learning with a Two-Layer Network

Parameters: \( W \in \mathbb{R}^{H \times d}, b \in \mathbb{R}^{H}, \) and \( v \in \mathbb{R}^{H} \)
Learning with a Two-Layer Network

Parameters: $\mathbf{W} \in \mathbb{R}^{H \times d}$, $\mathbf{b} \in \mathbb{R}^H$, and $\mathbf{v} \in \mathbb{R}^H$

- If we choose a differentiable loss, then the whole function will be differentiable with respect to all parameters.

- Because of the squashing function, which is not convex, the overall learning problem is not convex.

- What does (stochastic) (sub)gradient descent do with non-convex functions?

- To calculate gradients, we need to use the chain rule from calculus.

- Special name for (S)GD with chain rule invocations: backpropagation.
Learning with a Two-Layer Network

Parameters: $\mathbf{W} \in \mathbb{R}^{H \times d}$, $\mathbf{b} \in \mathbb{R}^{H}$, and $\mathbf{v} \in \mathbb{R}^{H}$

- If we choose a differentiable loss, then the whole function will be differentiable with respect to all parameters.
- Because of the squashing function, which is not convex, the overall learning problem is not convex.
Learning with a Two-Layer Network

Parameters: $\mathbf{W} \in \mathbb{R}^{H \times d}$, $\mathbf{b} \in \mathbb{R}^{H}$, and $\mathbf{v} \in \mathbb{R}^{H}$

- If we choose a differentiable loss, then the whole function will be differentiable with respect to all parameters.
- Because of the squashing function, which is not convex, the overall learning problem is not convex.
- What does (stochastic) (sub)gradient descent do with non-convex functions?
Learning with a Two-Layer Network

Parameters: $\mathbf{W} \in \mathbb{R}^{H \times d}$, $\mathbf{b} \in \mathbb{R}^H$, and $\mathbf{v} \in \mathbb{R}^H$

- If we choose a differentiable loss, then the whole function will be differentiable with respect to all parameters.
- Because of the squashing function, which is not convex, the overall learning problem is not convex.
- What does (stochastic) (sub)gradient descent do with non-convex functions? It finds a \textit{local} minimum.
- To calculate gradients, we need to use the chain rule from calculus.
Learning with a Two-Layer Network

Parameters: $\mathbf{W} \in \mathbb{R}^{H \times d}$, $\mathbf{b} \in \mathbb{R}^H$, and $\mathbf{v} \in \mathbb{R}^H$

- If we choose a differentiable loss, then the whole function will be differentiable with respect to all parameters.
- Because of the squashing function, which is not convex, the overall learning problem is not convex.
- What does (stochastic) (sub)gradient descent do with non-convex functions? It finds a *local* minimum.
- To calculate gradients, we need to use the chain rule from calculus.
- Special name for (S)GD with chain rule invocations: **backpropagation**.