Artificial Neural Networks

TIN0255/09P9M80/D82 - Aprendizagem Profunda

2025.1 - Prof. Pedro Moura

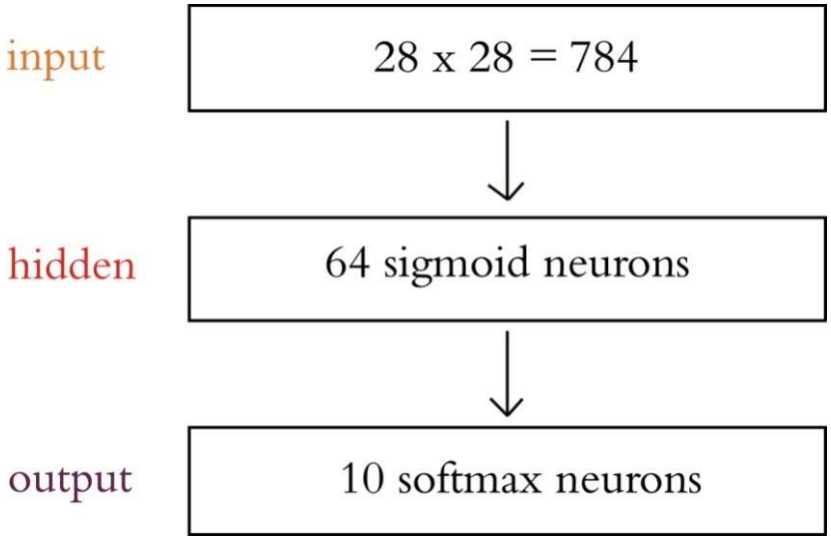
Artificial Neural Networks

- We saw the intricacies of artificial neurons.
- We are now going to explore the natural extension of that: how individual neural units are linked together to form **artificial neural networks.**
- These artificial neural networks include **deep learning models**.

The Input Layer

- In the *Shallow Net in Keras* we have seen, we crafted an artificial neural network with the following layers:
  - An input layer consisting of 784 neurons
  - A hidden layer composed of 64 sigmoid neurons
  - An output layer consisting of 10 softmax neurons

The Input Layer



A rough schematic of the shallow artificial-neural-network architecture we're whipping up in this chapter.

The Input Layer

- Neurons in the input layer don't perform any calculations.
- They are simply placeholders for input data.
- This placeholding is essential because the use of artificial neural networks involves performing computations on matrices that have predefined dimensions.
- **At least one of these predefined dimensions in the network architecture corresponds directly to the shape of the input data.**

The Input Layer

- One of the major activities when modeling and crafting a neural network is to choose the **input's representation**.
- This depends on the nature of the problem and data.
- In our classes, we are going to see some forms of **representation**.

Dense Layers

- There are many kinds of hidden layers.
- The most general is the **dense layer** (also called a **fully-connected layer**).
- Dense layers are found in many deep learning architectures, including the majority of the models we will see in our classes.

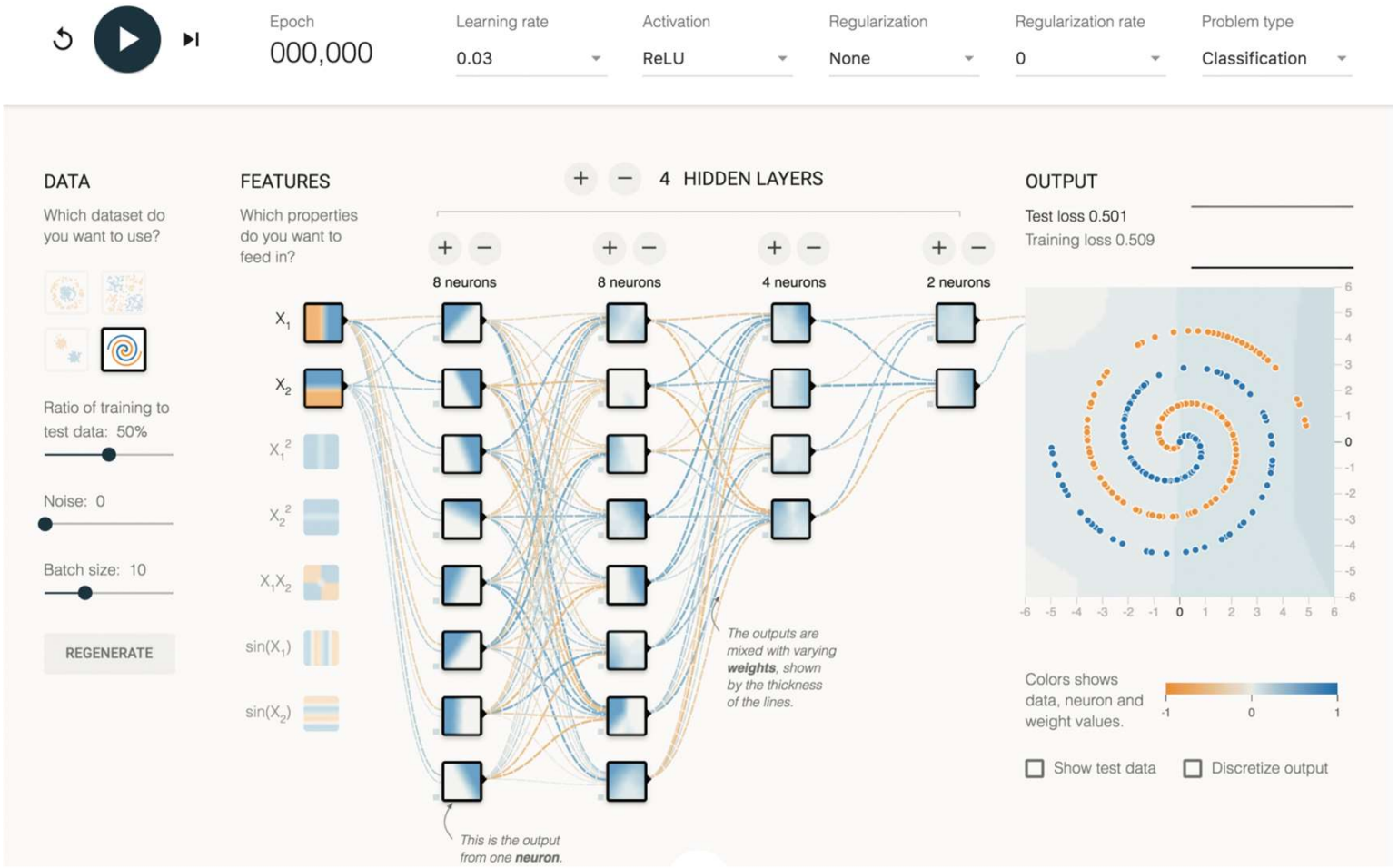
Dense Layers

- Their definition is uncomplicated:
- Each of the neurons in a given dense layer receive information from every one of the neurons in the preceding layer of the network.
- **A dense layer is fully connected to the layer before it!**

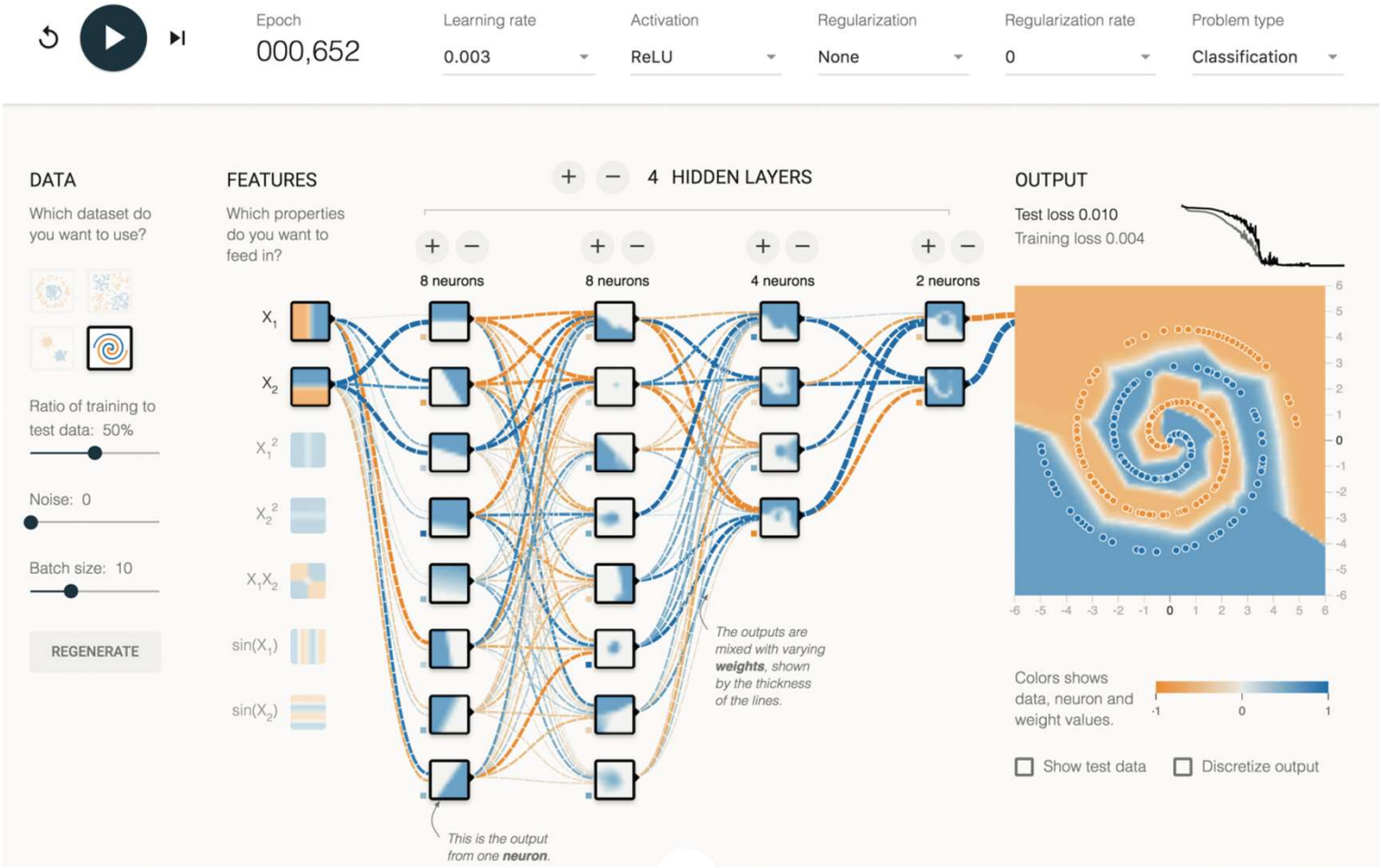
Dense Layers

- While they might not be as specialized nor as efficient as the other flavors of hidden layers we will dig into, dense layers are broadly useful.
- Because they can **nonlinearly recombine** the information provided by the preceding layer of the network.
- Let's revisit the TensorFlow Playground demo.
- Let's explore this configuration!

Dense Layers



Dense Layers



Dense Layers

- An **input layer with two neurons.**
  - One for vertical position of a given dot and the other for storing the dot's horizontal position.
- A **hidden layer composed of eight ReLU neurons**.
  - This is a dense layer because each of the eight neurons in it is connected to both input-layer neurons.
- Another **hidden layer composed of eight ReLU neurons**.
  - Note how the neurons in this layer are nonlinearly recombining the straight-edge features provided by the neurons in previous layer to produce more-elaborate features like curves and circles.

Dense Layers

- A third **dense hidden layer**, this one **consisting of four ReLU neurons** for a total of 32 (= 4 x 8) connecting inputs.
  - This layer nonlinearly recombines the features from the previous hidden layer to learn more-complex features that begin to look directly relevant to the binary (orange versus blue).
- A fourth and final **dense hidden layer**. With its two ReLU neurons, it receives a total of 8 (= 2 x 4) inputs from the previous layer.
- An **output layer made up of a single sigmoid neuron.**
  - Sigmoid is the typical choice of neuron for a **binary classification** problem like this one.
- In summary, **every layer** within networks provided by the TensorFlow Playground is a dense layer. → It can then be called **dense network** (or **Multilayer Perceptron**).

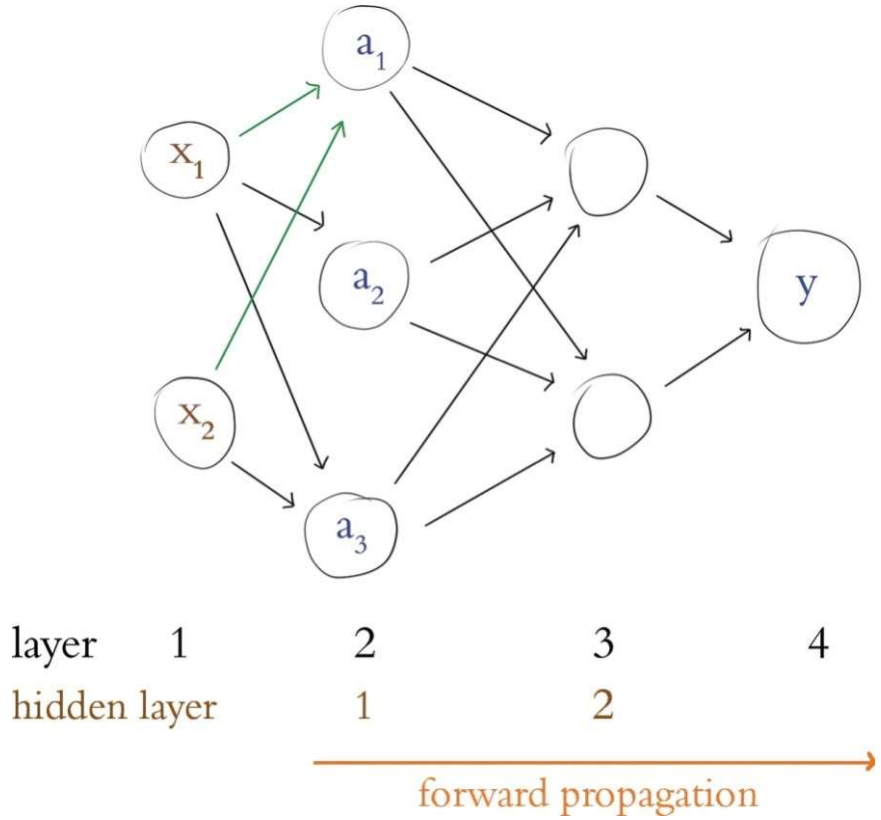
A Hot Dog-Detecting Dense Network

- Let's further strengthen our comprehension of dense networks.
- Our hot dog classifier is no longer a single neuron; in this chapter; it is now a **dense network** of artificial neurons.
- We have reduced the number of input neurons down to two for simplicity.
  - The first input neuron, x_1, represents the volume of ketchup (in, say, milliliters) → We are no longer restricted to binary inputs only.
  - The second input neuron, x_2, represents milliliters of mustard.
- We have two dense hidden layers.
  - The first hidden layer has three ReLU neurons.
  - The second hidden layer has two ReLU neurons.

A Hot Dog-Detecting Dense Network

- The output neuron is denoted by y in the network.
- This is a binary classification problem, hence it should be a **sigmoid neuron**.
- As in our perceptron examples:
  - **y** = 1 corresponds to the presence of a hot dog
  - **y** = 0 corresponds to the presence of some other object

A Hot Dog-Detecting Dense Network



A dense network of artificial neurons, highlighting the inputs to the neuron labeled $\mathbf{a_1}$.

Forward Propagation Through the First Hidden Layer

- Let's break down the **most importante equation**:

$$z = w \cdot x + b$$

$$z = (w_1 x_1 + w_2 x_2) + b$$

- In turn, with the **z** value for the neuron labeled a_1, we can calculate the activation **a** it outputs, using the fact that a_1 is a ReLU neuron.

$$a = max(0, z)$$

Forward Propagation Through the First Hidden Layer

- To make this computation of the output of neuron a_1 tangible:
  - x_1 is 4.0 mL of ketchup for a given object presented to the network.
  - x_2 is 3.0 mL of mustard for that same object.
  - $w_1 = -0.5$
  - $w_2 = 1.5$
  - $b = -0.9$

Forward Propagation Through the First Hidden Layer

- To calculate **z**, let's use the last equation and fill in these values:

$$
\begin{aligned}
z &= w \cdot x + b \\
&= w_1 x_1 + w_1 x_2 + b \\
&= -0.5 \times 4.0 + 1.5 \times 3.0 - 0.9 \\
&= -2 + 4.5 - 0.9 \\
&= 1.6
\end{aligned}
$$

Forward Propagation Through the First Hidden Layer

- Finally, let's compute **a**, which is the activation ouput of a ReLU neuron labeled $\mathbf{a_1}$.

$$\begin{aligned} a &= max(0, z) \\ &= max(0, 1.6) \\ &= 1.6 \end{aligned}$$

- Executing the calculations through an artificial neural network from the input layer (**x** values) through to the output layer (**y**) is called **forward propagation**.

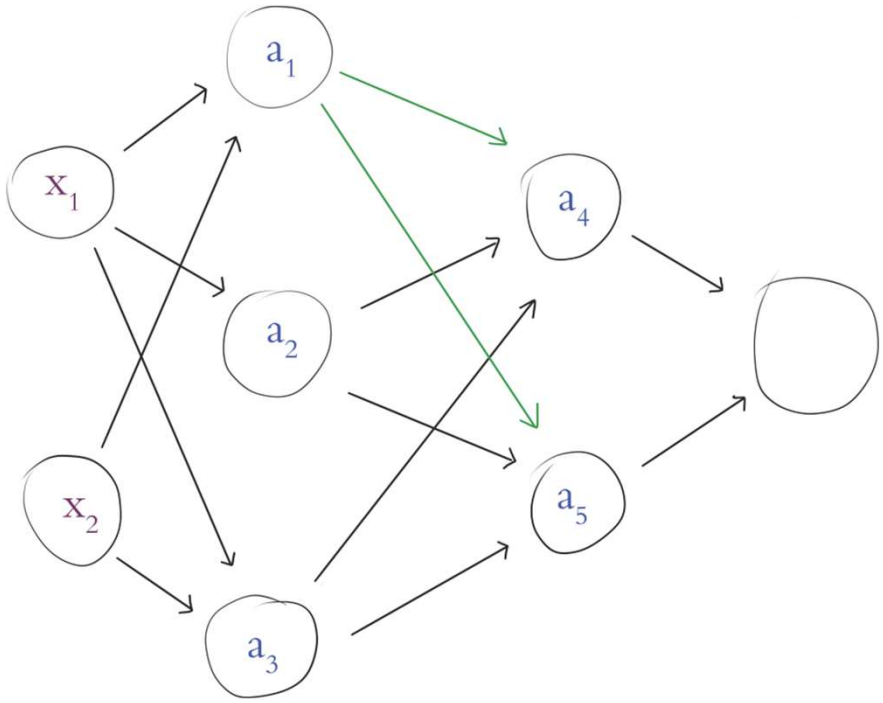
Forward Propagation Through the First Hidden Layer

- We saw the process for **forward propagating** through a single neuron in the first hidden layer of our network.
- To propagate through the remaining neurons of the first hidden layer ($\mathbf{a_2}$ and $\mathbf{a_3}$) we would follow the same process.
- The inputs $\mathbf{x_1}$ and $\mathbf{x_2}$ are identical for all three neurons, but each neuron in the first hidden layer will ouput a different activation $\mathbf{a}$ → $\mathbf{w_1}$, $\mathbf{w_2}$ and $\mathbf{b}$ parameters vary for each of the neurons in the layer.

Forward Propagation Through Subsequent Layers

- The process of **forward propagating** through the remaining layers is essentially the same as propagating through the first hidden layer.
- Let's work through an example.
- We will assume that we already have calculated the activation **a** for each of the neurons in the first hidden layer.
- The activation of the neuron $\mathbf{a}_1$, which we calculated to be 1.6, is one of the three inputs into the neurons labeled $\mathbf{a}_4$ and $\mathbf{a}_5$.

Forward Propagation Through Subsequent Layers



Our hot dog-detecting network, now highlighting the activation output of neuron a_1, which is provided as an input to both neuron a_4 and neuron a_5.

Forward Propagation Through Subsequent Layers

- To calculate the **forward propagation** through the second hidden layer, let's compute **a** for the neuron labeled a_4.

- We employ again the equation **w . x + b**.

$$
\begin{aligned}
a &= max(0, z) \\
&= max(0, (w \cdot x + b)) \\
&= max(0, (w_1 x_1 + w_2 x_2 + w_3 x_3 + b))
\end{aligned}
$$

Forward Propagation Through Subsequent Layers

- It is essentially similar to the calculation for the first hidden layer.
- The only twist is that the layer's inputs (i.e., **x**) do not come from outside the network → they are provided by the first hidden layer.
- In the last equation:
  - $\mathbf{x}_1$ is the value **a** = 1.6 for the neuron $\mathbf{a}_1$
  - $\mathbf{x}_2$ is the activation output from the neuron $\mathbf{a}_2$
  - $\mathbf{x}_3$ is likewise the activation ouput from the neuron $\mathbf{a}_3$

Forward Propagation Through Subsequent Layers

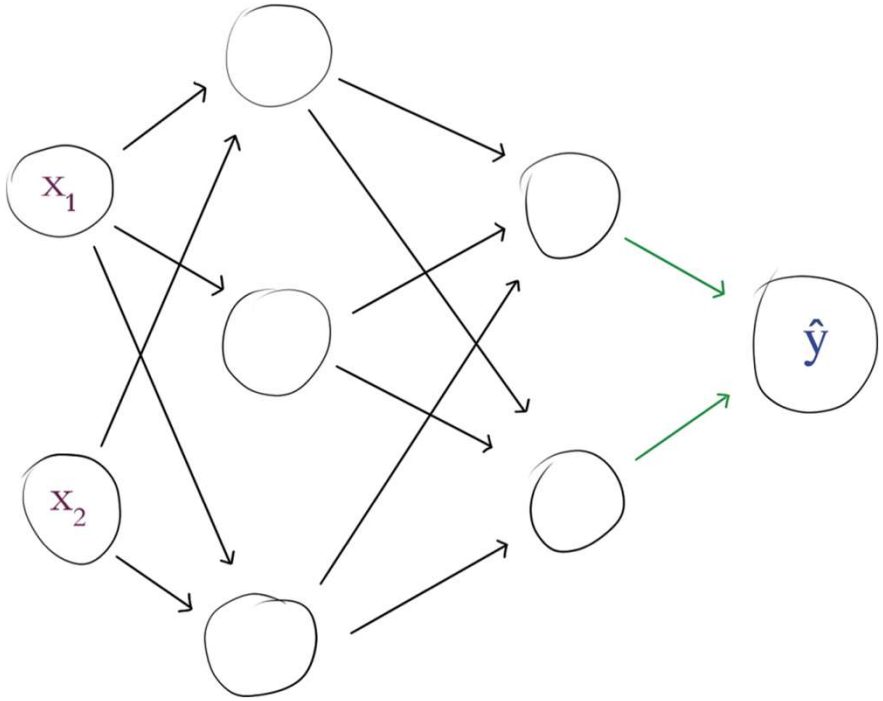
- Neuron labeled a_4 is able to nonlinearly recombine the information provided by the three neurons from first hidden layer.

- Neuron a_5 also non-linearly recombines this information, but in its own way:
  - The parameters w_1, w_2, w_3 and **b** for this neuron lead it to ouput a unique activation.

Forward Propagation Through Output Layer

- Let's round the process of by propagating through the output layer.
- Our single output neuron receives its inputs from the neurons labeled a_4 and a_5.
- Let's calculate **z** for this output neuron.
- The formula is identical to the previous ones for other neurons, but we have to take into account the **sigmoid function**.
  - These used values were contrived.

Forward Propagation Through Output Layer



Our hot dog-detecting network, with the activations providing input to the output neuron $\hat{y}$ highlighted.

Forward Propagation Through Output Layer

$$
\begin{aligned}
z &= \boldsymbol{w} \cdot \boldsymbol{x} + \boldsymbol{b} \\
&= \boldsymbol{w}_1 \boldsymbol{x}_1 + \boldsymbol{w}_2 \boldsymbol{x}_2 + \boldsymbol{b} \\
&= 1.0 \times 2.5 + 0.5 \times 2.0 - 5.5 \\
&= 3.5 - 5.5 \\
&= -2.0
\end{aligned}
$$

Forward Propagation Through Output Layer

- As the output neuron is **sigmoid**, to compute its activation **a** we pass its **z** value through the sigmoid function:
  - We can use the *Sidmoig Function* notebook for this purpose.

$$
\begin{aligned}
a &= \sigma(z) \\
&= \frac{1}{1 + e^{-z}} \\
&= \frac{1}{1 + e^{-(-2.0)}} \\
&\approx 0.1192
\end{aligned}
$$

Forward Propagation Through Output Layer

- The activation **a** computed by the sigmoid neuron in the output layer is a special case: it is the **final output** of our entire hot dog-detecting network.

- Because it is so special, we assign it a distinctive designation: **ŷ**.

- The value represented by **ŷ** is the network's guess as to whether a given object is a hot dog or not a hot dog (**probabilistically**).

- Given the inputs **x** and **x** that we fed into the network, it estimates that there is an **11.92%** chance that an object with those particular condiment measures **is a hot dog**.

Forward Propagation Through Output Layer

- If the object presented to the network was indeed a hot dog ($\mathbf{y} = 1$), then this $\hat{\mathbf{y}}$ of 0.1192 was pretty far off the mark.
- On the other hand, if the object was truly not a hot dog ($\mathbf{y} = 0$), then the $\hat{\mathbf{y}}$ is quite good.
- We will formalize the evaluation of $\hat{\mathbf{y}}$ predictions later.

The Softmax Layer

- As we have seen, the **sigmoid neuron** suits us well as an output neuron if we're building a network to distinguish two classes: blue dot vs orange dot, or a hot dog vs non hot dog.

- In many other circumstances, however, you have more than two classes to distinguish between.

- For example: the MNIST dataset consists of 10 numerical digits.

The Softmax Layer

- When concerned with a multiclass problem, the solution is to use a **softmax layer** as the output layer of our network.
- Softmax is in fact the activation function that we specified for the output layer in our *Shallow Net* in Keras Jupyter notebook.
- Let's see an example that builds upon out binary hot dog classifier.
- The schematic is the same - right down to its volumes-of-ketchup-and-mustard inputs - except that instead of having a single output neuron, we now have **three**.

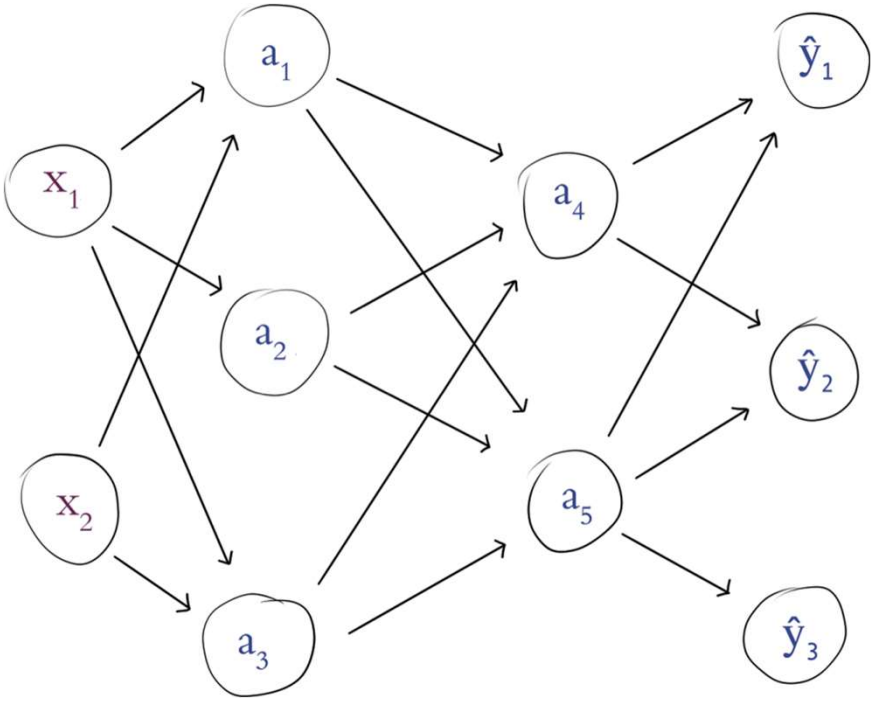
The Softmax Layer

- This **multiclass output layer is still dense**, so each of the three neurons receives information from both of the neurons in the final hidden layer.

- Let's say that now:
  - y_1 represents hot dogs.
  - y_2 is for burgers.
  - y_3 is for pizza.

- There are no alternatives to hot dogs, burgers, or pizza.
  - All objects presented to the network belong to one of these three classes of fast food, and **one** of the classes only.

The Softmax Layer



Our food-detecting network, now with three softmax neurons in the output layer.

The Softmax Layer

- Because the sigmoid function applies solely to binary problems, the output neurons in the example take advantage of the softmax activation function.
- Let's use code from our *Softmax Demo* notebook to understand how this activation function operates.
- Let's say that we presented a slice of pizza to the network.
- This pizza slice has negligible amounts of ketchup and mustard on it, and so x_1 and x_2 are near-0 values.
- Provided these inputs, we use forward propagation to pass information through the network **toward the output layer**.

The Softmax Layer

- Based on the information that the three neurons receive from the **final hidden layer**, they individually use our old friend **w . x + b**.

- We then calculate three unique (and contrived, in this case) **z** values:
  - For the neuron labeled $\hat{y}_1$, which represents hot dogs, **z** comes out to -1.0.
  - For the neuron labeled $\hat{y}_2$, which represents burgers, **z** is 1.0.
  - For the pizza neuron $\hat{y}_3$, **z** comes out to 5.0.

- These values indicate that the network estimates that the object presented to it is most likely to be **pizza** and least likely to be a **hot dog**.

The Softmax Layer

- Expressed as **z**, however, it it isn't straightforward to intuit **how much** more likely the network predicts to be pizza relative to the other two classes.
- This is where the **softmax function** comes in.
- Applying the softmax function to this list involves a three-step process.
  - 1) The first step is to **calculate the exponential** of each of the **z** values.
  - 2) The second step is to **sum up our exponentials**.
  - 3) The third step provides proportions for each of our three classes relative to the sum of all the classes.

The Softmax Layer

- **Softmax** transforms values into probabilities:

$$\hat{y}_i = \sigma(z)_i = e^{z_i} / \Sigma\, e^{z_j}$$

- Generalization of **Sigmoid** activation function.

The Softmax Layer

- In our code, we created a list named **z** to store our tree **z** values:

- `z = [-1.0, 1.0, 5.0]`

- We then apply the first step:
  - `exp(z[0])` comes out to `0.3679` for hot dog.
  - `exp(z[1])` gives us `2.718` for burger.
  - `exp(z[2])` gives us the much, much larger (exponentially so!) `148.4` for pizza.

The Softmax Layer

- We now carry out our second step (to sum up our exponentials):

- `total = exp(z[0]) + exp(z[1]) + exp(z[2])`

- With this `total` variable we can execute the third and final step:
  - `exp(z[0])/total` outputs a $\hat{y}_1$ value of **0.002428** → there's a ~0.2% chance that the object presented to it is a **hot dog**.
  - `exp(z[1])/total` outputs a $\hat{y}_2$ value of **0.01794** → ~1.8% chance that it's a **burger**.
  - `exp(z[2])/total` outputs a $\hat{y}_3$ value of **0.9796** → ~98.0% chance that the object is **pizza**.

The Softmax Layer

- Now the etimology of the **softmax** name should be discernible.
- The function returns **z** with the highest value (the **max**), but it does so **softly**.
- It doesn't not indicate that there's a 100% chance it's pizza and a 0% chance it's either of the other two fast food classes (that would be a **hard max function**).
- The network hedges its bets, to an extent, and provides a likelihood that the object is each of the three classes.
- This leaves us to make the decision about how much **confidence** we would require to accept **a neural network's guess**.

The Softmax Layer

- Note: **the use of the softmax function with a single neuron is a special case of softmax that is mathematically equivalent to using a sigmoid neuron.**

The Softmax Layer

- The denominator of the softmax function a sum of the form: $\Sigma_j e^{x_j}$.
- When x_j take a wide range of values -> exponentes e^{x_j} take on na even wider range of values.
  - **Some tiny and some very large.**
- Adding very large and very small floating point numbers → **numeric inaccuracy** issues in fixed-width floating point representations (such as 32-bit or 64-bit)

The Softmax Layer

- Fortunately, there is a trick to avoid this problem.
- We observe that:

$$\mu(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} = \frac{e^{x_i - c}}{\sum_j e^{x_j - c}}$$

for any constant *c.*

The Softmax Layer

- We pick $c = \max_j x_j$, so that $x_j - c \leq 0$ for all j.
- This ensures that $e^{(x_j - c)}$ is always between 0 and 1 -> leads to a more accurate calculation.
- **Most deep learning frameworks will take care to compute so will take care to compute softmax in this manner.**

Exercise

- **Exercise** (13.2.5 from “Mining of Massive Datasets”): Show that, for any vector [v_1, v_2, ..., v_k],

$$\sum_{i=1}^{k} \mu\left(v_i\right) = 1$$

where μ represents the softmax function.

Revisiting Our Shallow Network

- Let's now revisit our shallow network!

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 64)                50240
_________________________________________________________________
dense_2 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
_________________________________________________________________
```

A summary of the model object from our *Shallow Net in Keras* Jupyter notebook.

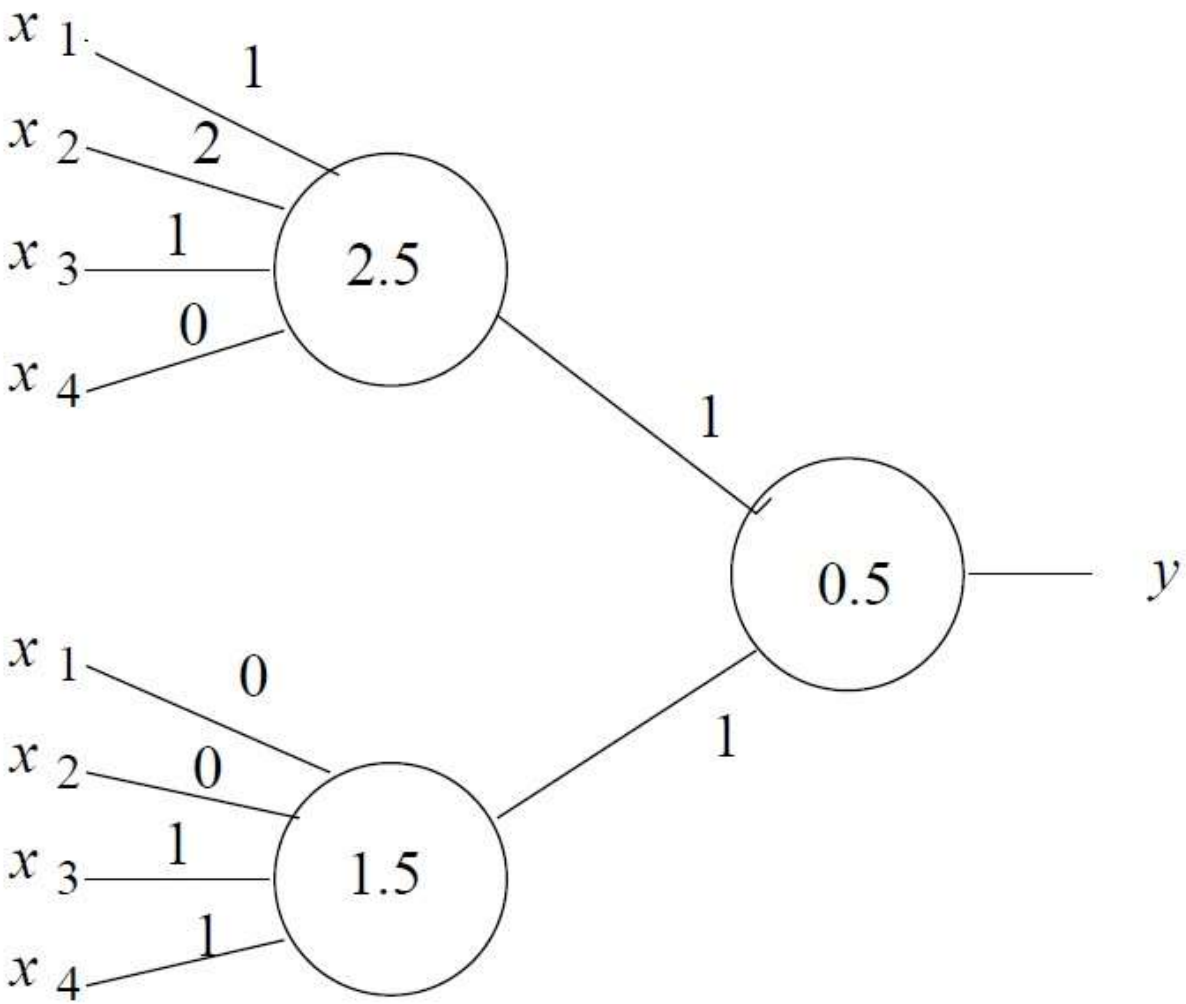
Example of Neural Network

- We want to design a neural net that only selects **"good" bit-vectors**, which are those that have **two consecutive 1's**.
- Since we want to deal with only tiny example instances, we shall assume bit-vectors have length four.
- Our training examples will thus have the form ($[x_1, x_2, x_3, x_4]$, y), where each of the x's are bits, 0 or 1.
- There are 16 possible training examples, and we shall assume we are given some subset of these as our training set.

Example of Neural Network

- Eight of the possible bit vectors are good – they do have consecutive 1's, and there are also eight "bad" examples.
  - **0111 and 1100 are good.**
  - **1001 and 0100 are bad.**
- How we **might design this net from training examples** we will see later in our classes.
  - But this net serves as an example of what we would like to achieve.

Example of Neural Network



Example of Neural Network

- The net has **three layers:**
  - Input layer consists of the bit vector: [x_1, x_2, x_3, x_4].
  - Hidden layer contains two nodes.
  - Output layer with a single node that produces the output y.

- **Each node is a perceptron** (exactly as we have seen)**.**

Example of Neural Network

- In the hidden layer, the first node is characterized by weight vector [w_1, w_2, w_3, w_4] = [1, 2, 1, 0] and threshold 2.5.

- Each input x is either 0 or 1 → the only way to reach a sum $\sum_{i=1}^{4} x_i w_i$ as high as 2.5 is if x_2 = 1 and at least one of x_1 and x_3 is also 1.
  - The output of this node is 1 if and only if the input is one of 1100, 1101, 1110, 1111, 0110, or 0111.

- It recognizes those bit-vectors that either begin with two 1's or have two 1's in the middle.

- The second node, with weights [0, 0, 1, 1] and threshold 1.5 gives output 1 whenever x_3 = x_4 = 1, and not otherwise.
  - It recognizes the inputs 0011 and 1011, as well as some other good inputs that are also recognized by the first node.

Example of Neural Network

- The output layer, with a single node, has weights [1, 1] and threshold 0.5.

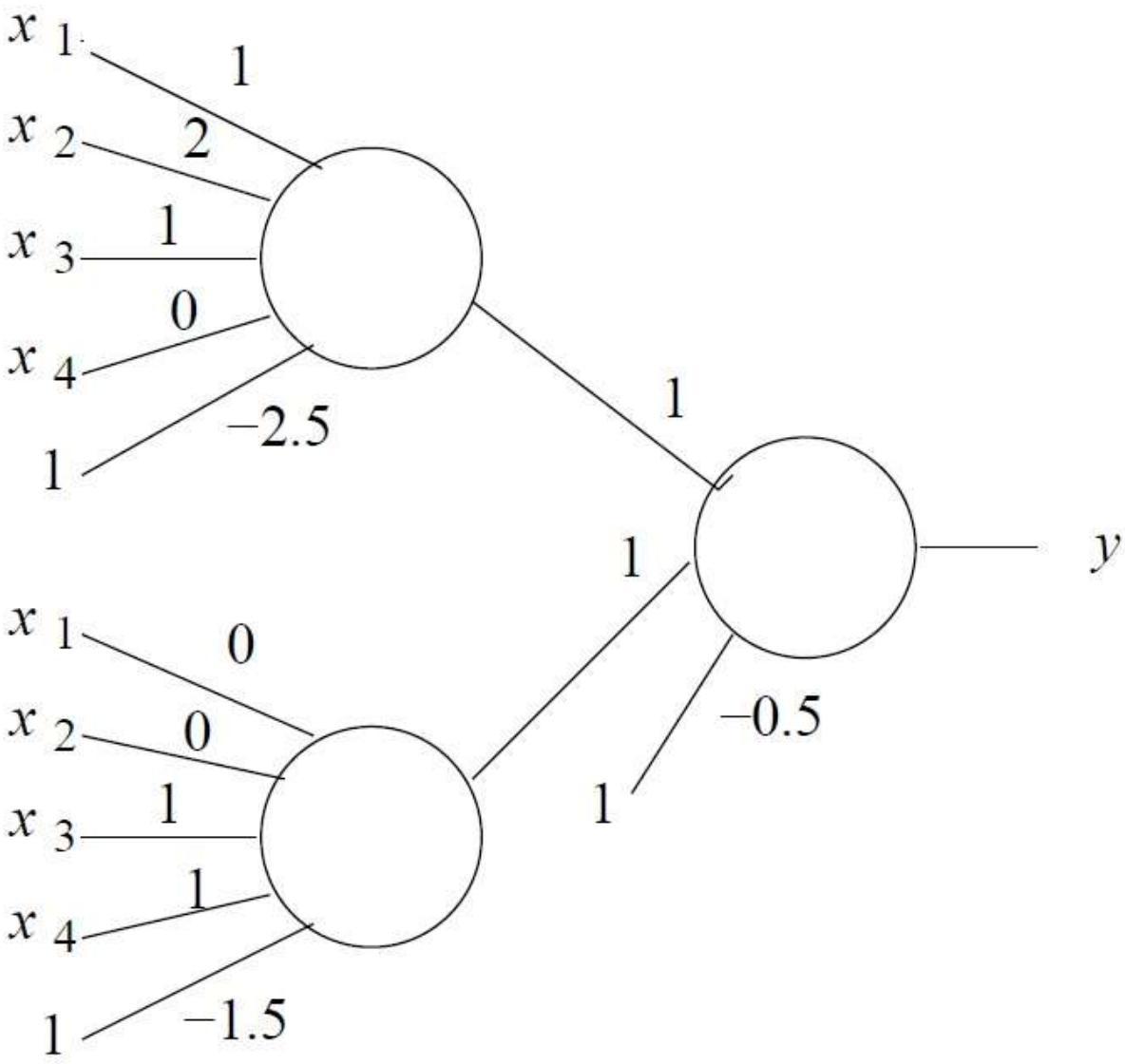
- It thus behaves as an **"OR-gate"**.
  - It gives output $y = 1$ whenever either or both of the nodes in the hidden layer have output 1.
  - But gives output $y = 0$ if both of the hidden-layer nodes give output 0.

- Thus, the neural net gives output 1 for all the good inputs but none of the bad inputs.

Example of Neural Network

- It is useful in many situations to assume that nodes have a **threshold of 0**.
- We can always convert a perceptron with a nonzero threshold *t* to one with a 0 threshold if we add an **additional input**.
  - **That input always has value 1 and a weight equal to −*t*.**
- We can convert our neural network that selects “good” bit vectors to the following.

Example of Neural Network



Exercise

- **Exercise** (13.1.1 from “Mining of Massive Datasets”): Prove that the problem of classifying “good” bit-vectors, those that have two consecutive 1’s, cannot be solved by a perceptron, i.e., the good and bad points are not linearly separable.

Exercise

- **Exercise** (13.1.3 from “Mining of Massive Datasets”): Design a neural net that functions as an **exclusive-or gate**; that is, it takes two inputs and gives output 1 if exactly one of the inputs is 1, otherwise it gives output 0. *Hint*: remember that both weights and thresholds can be negative.

Exercise

- **Exercise** (13.1.4 from “Mining of Massive Datasets”**):** Prove that there is no single perceptron that behaves like an exclusive-or gate.

Exercise

- **Exercise** (13.1.5 from “Mining of Massive Datasets”): Design a neural net to compute the **exclusive-or** of three inputs; that is, output 1 if an odd number of the three inputs is 1 and output 0 if an even number of the inputs is 1.

Building a Neural Network

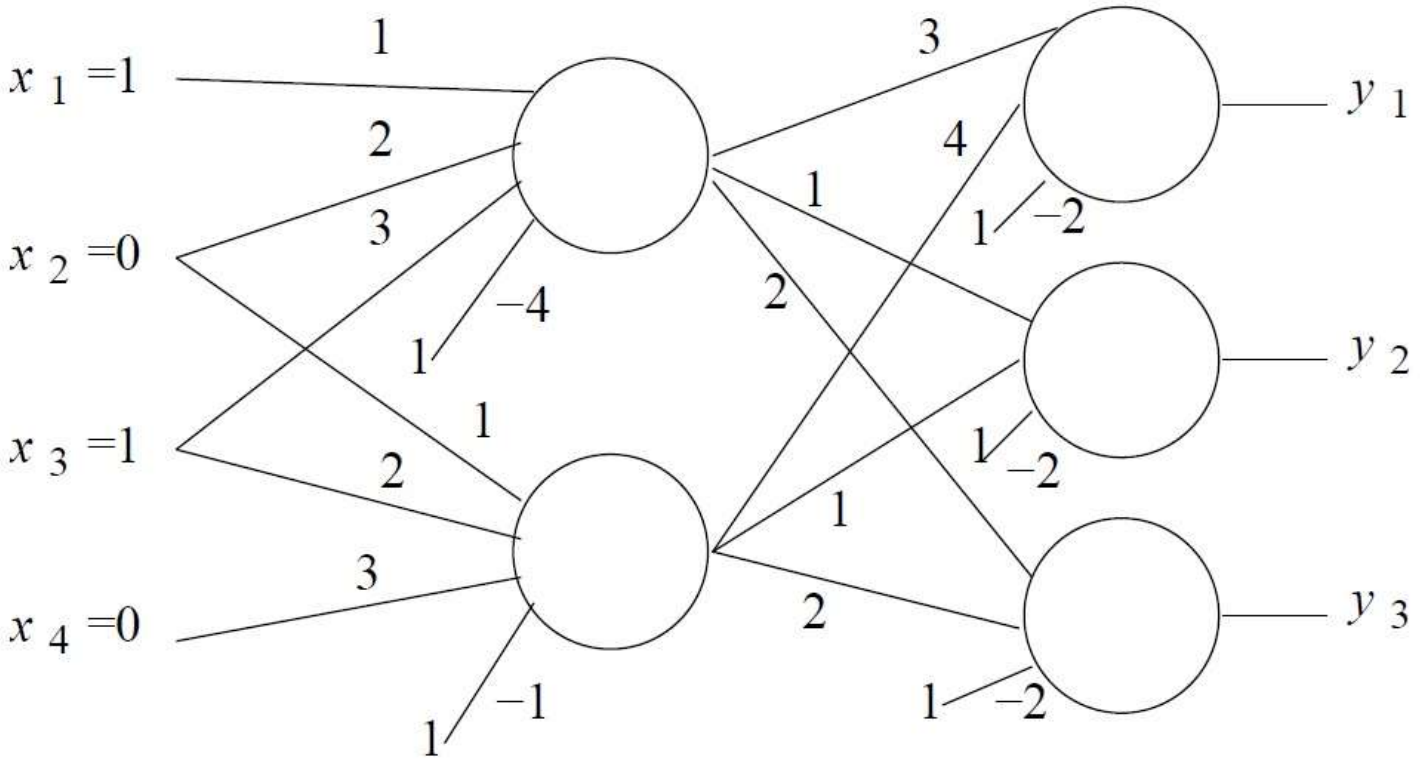
- Building a neural net to solve a given problem is **partially art** and **partially science**.
- Before we can begin to train a neural network, we need to make a number of design decisions:
  - How many hidden layers should we use?
  - How many nodes will there be in each of the hidden layers?
  - In what manner will we interconnect the outputs of one layer to the inputs of the next layer?

Building a Neural Network

- There are other decisions that need to be made when we train the neural net:
  - What cost function should we minimize to express what weights are best?
  - What algorithm do we use to exploit the training examples in order to optimize the weights?

Exercise

- **Exercise** (13.2.5 from "Mining of Massive Datasets"): The figure below is a neural net with particular values shown for all the weights and inputs. Suppose that we use the sigmoid function to compute outputs of nodes at the first layer, and we use softmax to compute outputs of the nodes in the output layer.



Exercise

- (a) Compute the values of the outputs for each of the five nodes.
- (b) Assuming each of the weights and each of the x_i is a Variable, express the output of the first (top) output node in terms of the weights and the x_i.
- (c) Find the derivative of your function from part (b) with respect to the weight on the first (top) input to the first (top) node in the first layer.

The Secrets to the Power of Function Composition

- Biological metaphor sounds like an exciting way to justify the computational power of a neural network.
- However, it does not provide a **complete picture** of the settings in which neural networks perform well
- Most basic level → ANN is a computational graph that performs compositions of simpler functions to provide a more complex function.
- Much of the power of ANNs → **repeated composition of multiple nonlinear functions has significant expressive power.**

The Secrets to the Power of Function Composition

- The work of Hornik, Stinchcombe and White shows that the **single composition of a large number of squashing functions can approximate almost any function.**
  - K. Hornik, M. Stinchcombe, and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks and Networks*, 2(5), pp. 359-366, 1989.

- This approach will require an **extremely large number of units (i.e., parameters) of the network**.

- This increases the capacity of the network, which causes **overfitting unless the data set is extremely large**.

The Secrets to the Power of Function Composition

- Much of the power of deep learning → **repeated composition of certain types of functions increases the representation power of the network, and therefore, reduces the parameter space required for learning.**

- **Not all functions are equally good at achieving this goal.**

- Nonlinear squashing functions used are not arbitrarily chosen.
  - They are carefully designed because of certain properties.

The Secrets to the Power of Function Composition

- Imagine a situation in which the identity activation function is used in each layer → only linear functions are computed.
- **The resulting neural network is no stronger than a single-layer linear network.**
- **Theorem:** *A multi-layer network that uses only the identity activation function in all its layers reduces to a single-layer network performing a linear regression.*
- **Proof:** on the board.

The Secrets to the Power of Function Composition

- This result is for the case of regression modeling with numeric target variables.
- **A similar result holds true for binary target variables.**
- **Special case:** all layers use identity activation and the final layer uses a single output with sign activation for prediction.
  - **This multilayer network reduces to the perceptron.**

The Secrets to the Power of Function Composition

- **Lemma:** *Consider a multilayer network in which all hidden layers use identity activation and the single output node uses the perceptron criterion as the loss function and the sign activation for prediction. This neural network reduces to the single-layer perceptron.*

- **Proof:** The proof of this result is almost identical to the theorem we saw previously.

The Secrets to the Power of Function Composition

- The universal approximation result of neural networks posits that a linear combinatoin of sigmoid units **in a single layer can be used to approximate any function well**.
  - This is valid for most other reasonable squashing functions.

- This linear combination can be perfomed by a single output node.

- **Therefore, a two-layer network is sufficient as long as the number of hidden units is large enough.**

The Secrets to the Power of Function Composition

- Activation functions must have basic non-linearity in order to model turns and twists in an arbitrary function.

- All 1-dimensional functions can be approximated as a sum of scaled/translated step functions.
  - Most of the activation functions we saw look awfully like step functions (e.g., sigmoid).

- **This basic idea is the essence of the universal approximation theorem in neural networks.**

The Secrets to the Power of Function Composition

- However, the number of base functions required to reach a high level of approximation can be extremely large!

- Potentially increasing the data-centric requirements to an **unmanageable level.**
  - One possible solution → **trade breadth for depth.**

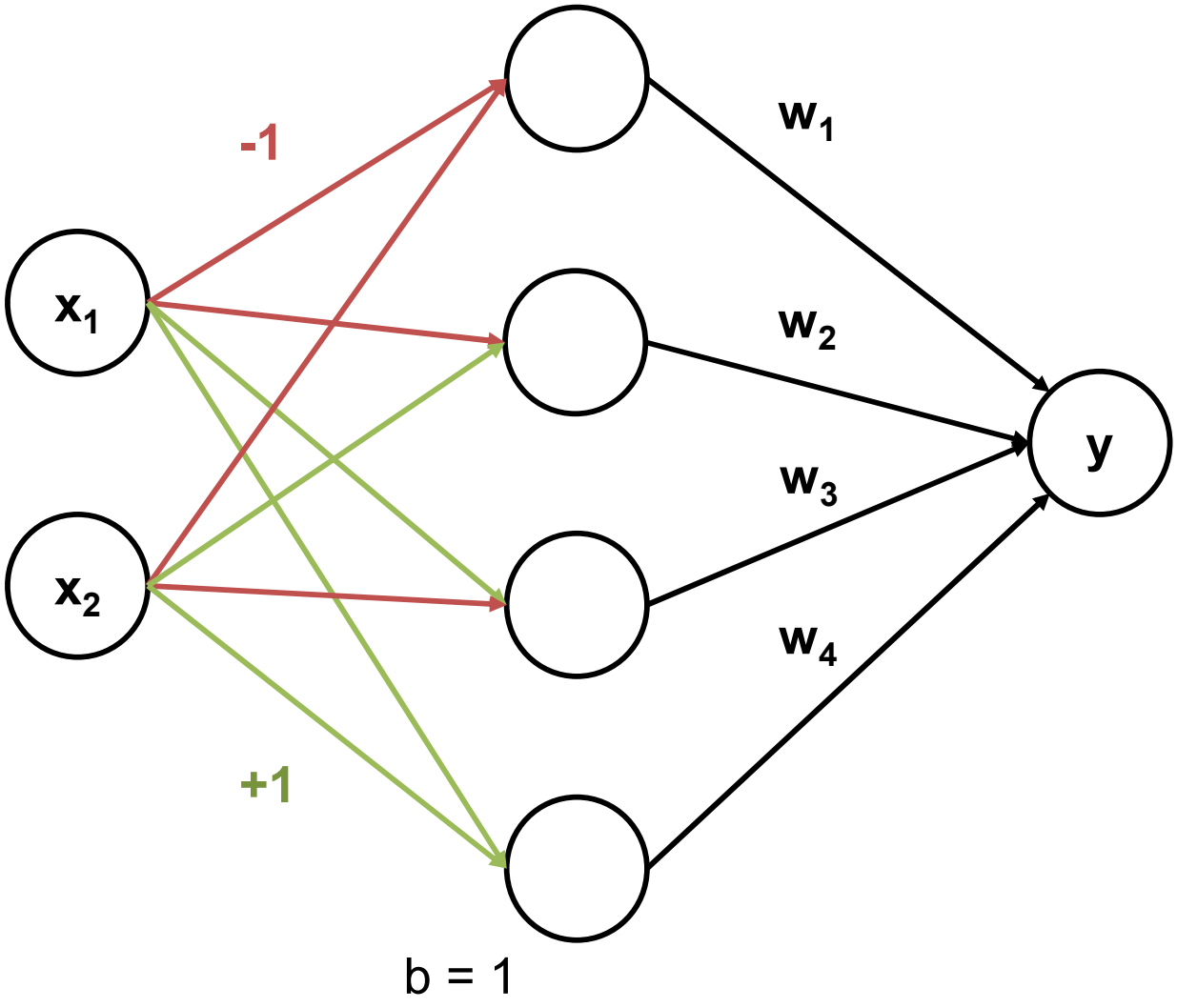
- **For this reason, shallow networks face the persistent problem of overfiting.**

- The universal approximation theorem asserts the ability to well-approximate the function implict in the training data, **but makes no guarantee about whether the function can be generalized to unseen test data.**

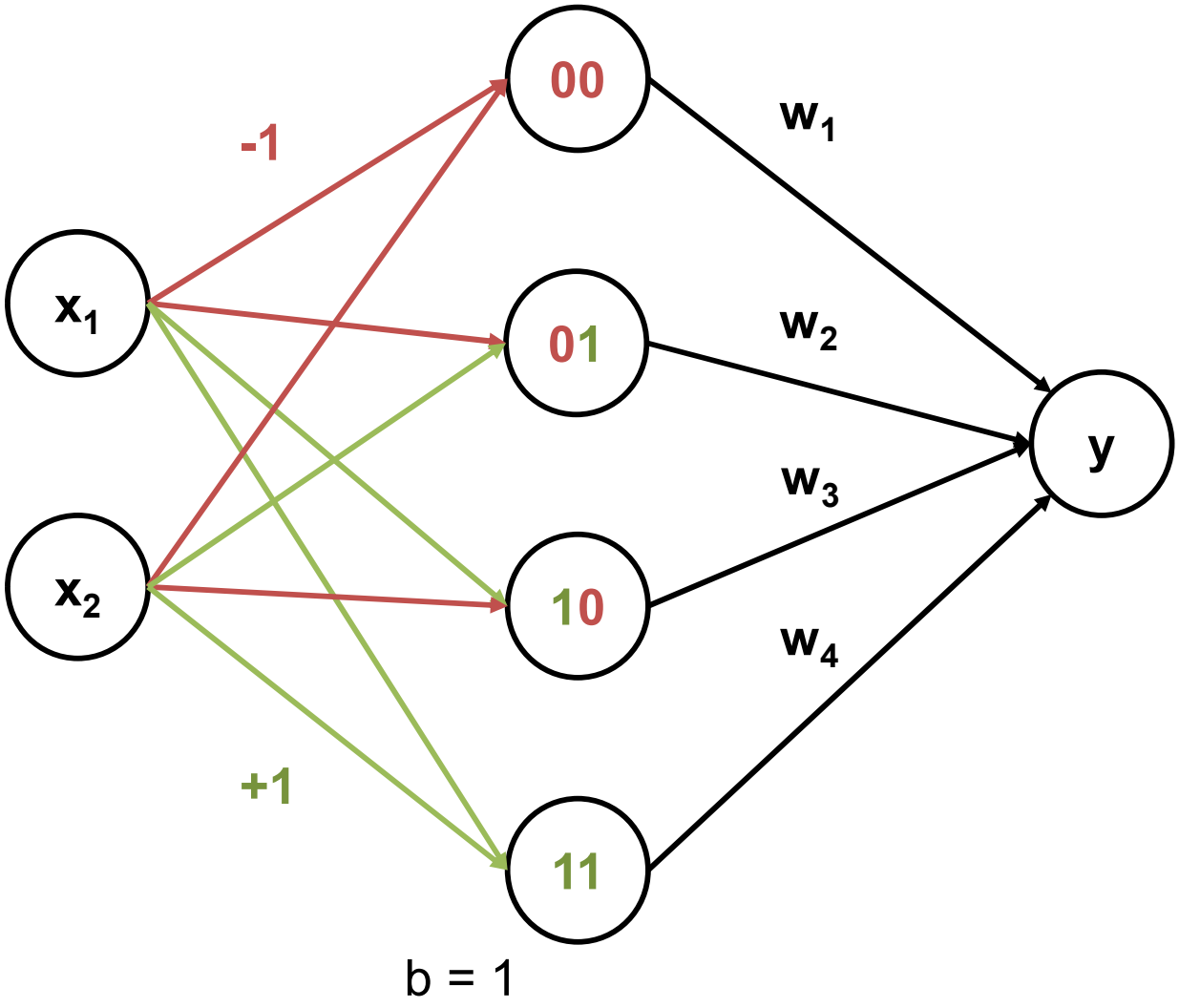
Representation Power of Neural Networks

- ANN with 1 hidden layer can (thus **shallow**) can represent:
  - Any bounded continuous function (to arbitrary ε)
    - Universal Approximation Theorem (Cybenko, 1989)
  - Any Boolean function (exactly)

Representation Power of Neural Networks

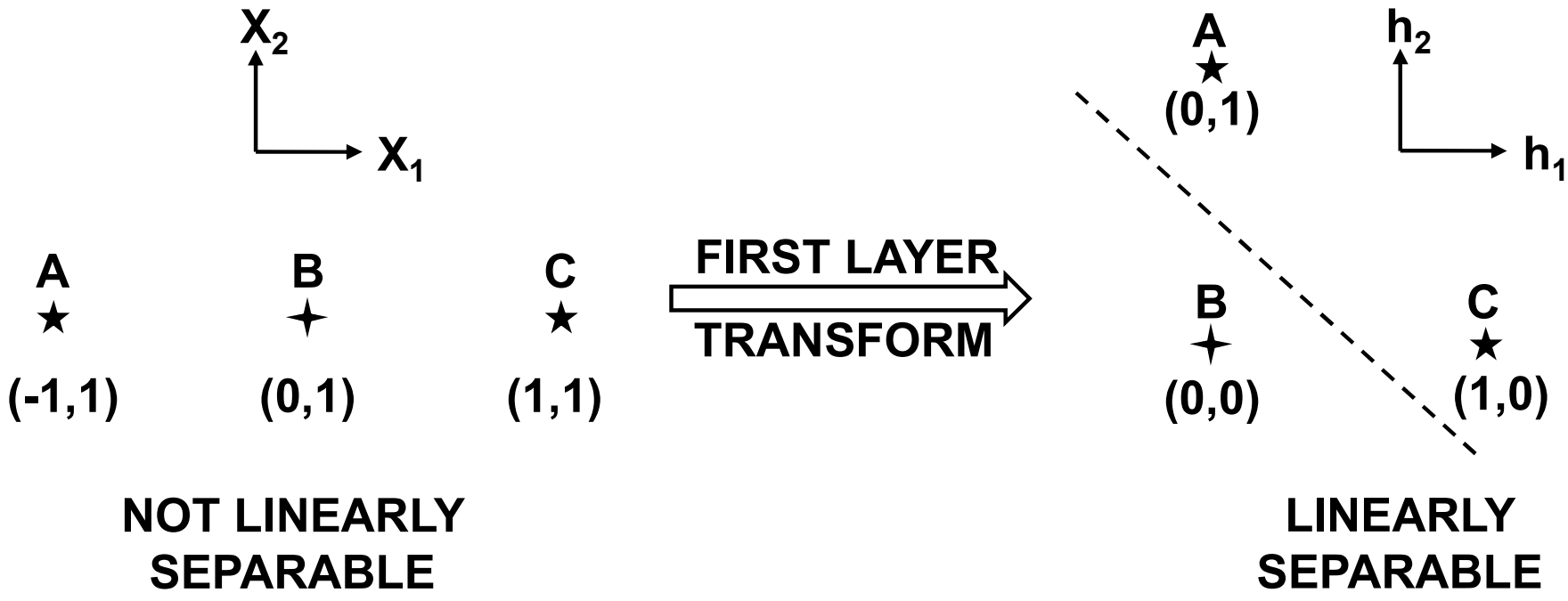


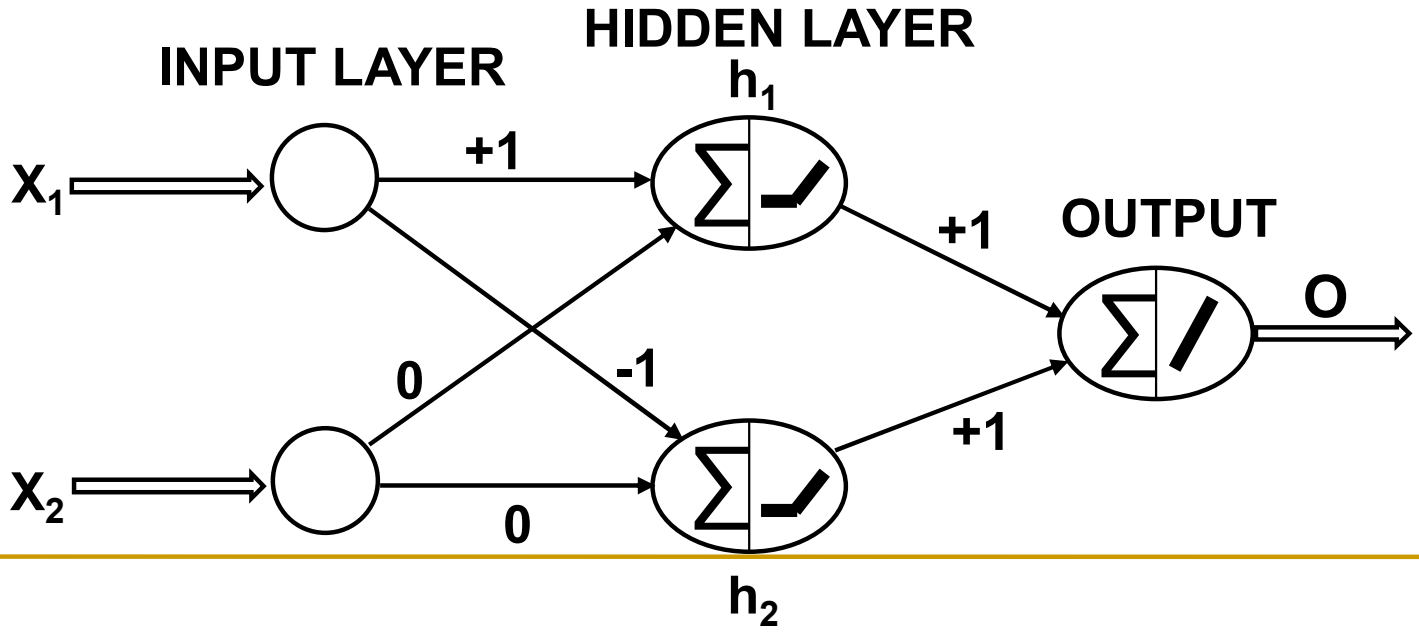
Representation Power of Neural Networks



The Importance of Nonlinear Activation

- Let's see an example which contains a two-class data set.
- A and C → class denoted by '*'.
  - Coordinates (1,1) and (-1,1), respectively.
- B → class denoted by '+'.
  - Coordinates (0,1).
- **A neural network with only linear activations will never be able to classify the training data perfectly because the points are not linearly separable.**


X2
X1
A
(-1,1)
B
(0,1)
C
(1,1)
FIRST LAYER
TRANSFORM
A
(0,1)
h2
h1
B
(0,0)
C
(1,0)
NOT LINEARLY SEPARABLE
LINEARLY SEPARABLE


INPUT LAYER
HIDDEN LAYER
h1
X1
X2
+1
0
-1
0
+1
+1
OUTPUT
O
h2

Example of Classifying Inseparable Data

- Consider now a situation in which the hidden units have ReLU activation, and they **learn the two new features** h_1 and h_2:

$$h_1 = \max\{x_1, 0\}$$
$$h_2 = \max\{-x_1, 0\}$$

- These goals can be achieved by:
  - Appropriate **weights** from the input to hidden layer.
  - Applying a ReLU activation function (threshold negative values to 0).

- We can also check a plot of the data in terms of h_1 and h_2 in the same figure.

Example of Classifying Inseparable Data

- Coordinates in the 2-dimensional hidden layer are: {(1,0), (0,1), (0,0)}.
- They are now **separable** in terms of the **new hidden representation**.
- Task of the first layer → **representation learning**.
  - To enable the solution of the problem with a linear classifier.
  - Now a single linear output layer is able to classify perfectly.
- Key point → nonlinear ReLU function is crucial in ensuring this linear separability.
- **Activations functions enable nonlinear mappings of the data, so that the embedded points can become linearly separable.**

Example of Classifying Inseparable Data

- If both weights from hidden to output layer are set to 1 with a linear activation function, the output O is:
  - $O = h_1 + h_2$
- This separates the two classes because it always takes on the value of 1 for the two points '*' and takes on 0 for the point '+'.
- **Much of the power → hidden in the use of non-linear activation functions.**

Example of Classifying Inseparable Data

- Activation functions enable nonlinear transformations of the data → **become increasingly powerful with multiple layers**.
- Sequence of nonlinear activations imposes a specific type of structure on the learned model.
  - **Power increases with depth of the sequence (number of layers).**

Example of Classifying Inseparable Data

- There are many alternative choices of weights that can make the hidden representation linearly separable.
  - The weights will need to **learned** in a data-driven manner.

- Learned weights may be different than the ones we used.

- Data set is not linearly separable in the original space → **Perceptron would fail!**
  - There is no choice of weights at which one could hope to classify this data perfectly.

- **Activation functions enable nonlinear transformations of the data.**
  - It becomes increasingly powerful with multiple layers.

Reducing Parameter Requirements with Depth

- Reducing the depth of the network is **not without its disadvantages.**
- **Deeper networks are often harder to train.**
  - All types of unstable behavior such as the vanishing and exploding gradiente problems.
- **Deeper networks are also unstable to parameter choice.**
- These issues are addressed with **careful design of functions** computed within nodes.
  - As well as use of **pretraining procedures** to improve performance.

Reducing Parameter Requirements with Depth

- There are many alternative choices of weights that can make the hidden representation linearly separable.
  - The weights will need to **learned** in a data-driven manner.
- Learned weights may be different than the ones we used.
- Data set is not linearly separable in the original space → **Perceptron would fail!**
  - There is no choice of weights at which one could hope to classify this data perfectly.
- **Activation functions enable nonlinear transformations of the data.**
  - It becomes increasingly powerful with multiple layers.

Exercise

- **Exercise** (1.10.1 from “Neural Networks and Deep Learning: A Textbook”): Consider the case of the XOR function in which the two points {(0,0), (1,1)} belong to one class, and the other two points {(1,0), (0,1)} belong to the other class. Show how you can use the ReLU activation function to separate the two classes in a manner similar to the previous example.

Exercise

- **Exercise** (1.10.9 from "Neural Networks and Deep Learning: A Textbook"): Consider a network with two inputs x_1 and x_2. It has two hidden layers, each of which contain two units.
- Assume that the weights in each layer are set so that **top units in each layer applies sigmoid activation** to the sum of its inputs and **the bottom unit in each layer applies tanh activation** to the sum of its inputs.
- Finally, **the single output node applies ReLU activation** to the sum of its two inputs. Write the output of this neural network in **closed form** as a function of x_1 and x_2. This exercise should give you an idea of the complexity of functions computed by neural networks.

Exercise

- **Exercise** (1.10.10 from “Neural Networks and Deep Learning: A Textbook”): Compute the partial derivative of the closed form computed in the previous exercise with respect to x_1. Is it practical to compute derivatives for gradient descent in neural networks by using closed-form expressions (as in traditional machine learning)?

Referências

1. Capítulo 7 do livro *"Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence"* de Jon Krohn, Grant Beyleveld e Aglaé Bassens.

2. Capítulo 13 do livro *"Mining of Massive Datasets"* de Jure Leskovec, Anand Rajaraman e Jeffrey Ullman.

3. Capítulo 1 do livro "*Neural Networks and Deep Learning: A Textbook"* de Charu C. Aggarwal. 2ª ed.