COMP 761: Lecture 38 – Neural Networks III

David Rolnick

December 2, 2020

Problem

Describe a gradient descent approach to learning the weights of a perceptron.

(Please don't post your ideas in the chat just yet, we'll discuss the problem soon in class.)

Course Announcements

Course Announcements

- Last class tomorrow (Thursday) at 4 pm Montreal.

Course Announcements

- Last class tomorrow (Thursday) at 4 pm Montreal.
- Office hours as usual Thurs 10:30 am (Vincent) and Fri 10 am (David).

Review: Perceptrons

Review: Perceptrons

- A *perceptron* is a function $f(x) = w \cdot x$ for $x, w \in \mathbb{R}^n$.

Review: Perceptrons

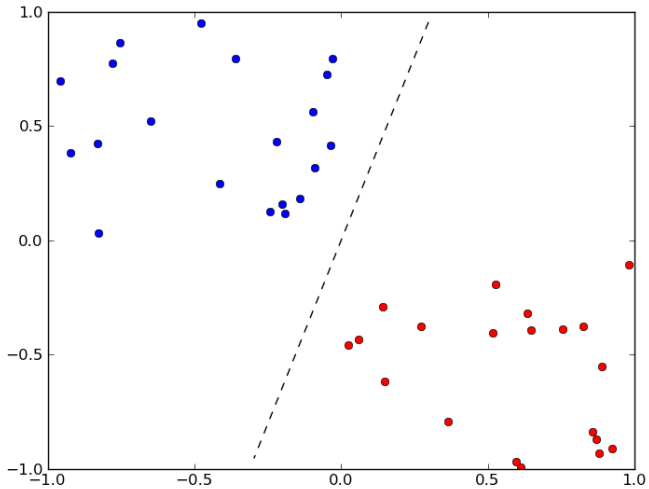
- A *perceptron* is a function $f(x) = w \cdot x$ for $x, w \in \mathbb{R}^n$.
- Given a dataset of points $x^1, x^2, \ldots, x^k$ labeled as ± 1, we can choose the weight vector w so that $f(x^i) > 0$ if x^i is labeled $+1$ and $f(x^i) < 0$ if x^i is labeled -1.

Review: Perceptrons

- A *perceptron* is a function $f(x) = w \cdot x$ for $x, w \in \mathbb{R}^n$.
- Given a dataset of points $x^1, x^2, \ldots, x^k$ labeled as ± 1, we can choose the weight vector w so that $f(x^i) > 0$ if x^i is labeled $+1$ and $f(x^i) < 0$ if x^i is labeled -1.
- We can do this via an iterative algorithm where we change w to fix errors in the classification.

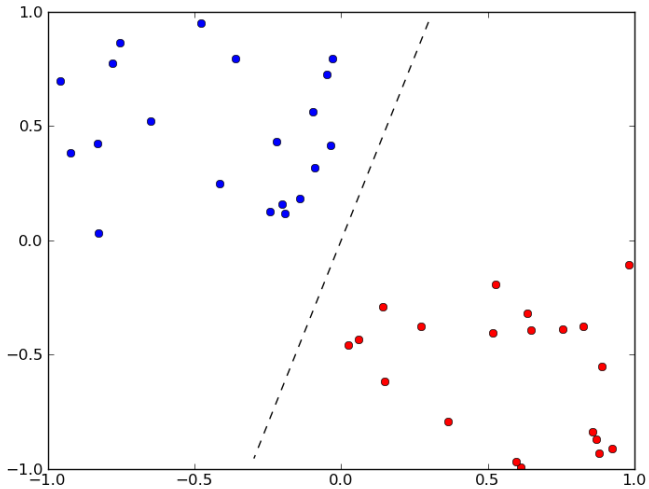
Review: Perceptrons

- A *perceptron* is a function $f(x) = w \cdot x$ for $x, w \in \mathbb{R}^n$.
- Given a dataset of points $x^1, x^2, \ldots, x^k$ labeled as ± 1, we can choose the weight vector w so that $f(x^i) > 0$ if x^i is labeled $+1$ and $f(x^i) < 0$ if x^i is labeled -1.
- We can do this via an iterative algorithm where we change w to fix errors in the classification.
- It only works if the data points are actually linearly separable.



Review: Perceptrons

- A *perceptron* is a function $f(x) = w \cdot x$ for $x, w \in \mathbb{R}^n$.
- Given a dataset of points $x^1, x^2, \ldots, x^k$ labeled as ± 1, we can choose the weight vector w so that $f(x^i) > 0$ if x^i is labeled $+1$ and $f(x^i) < 0$ if x^i is labeled -1.
- We can do this via an iterative algorithm where we change w to fix errors in the classification.
- It only works if the data points are actually linearly separable.



- What if the points are not linearly separable?

Neural networks

Neural networks

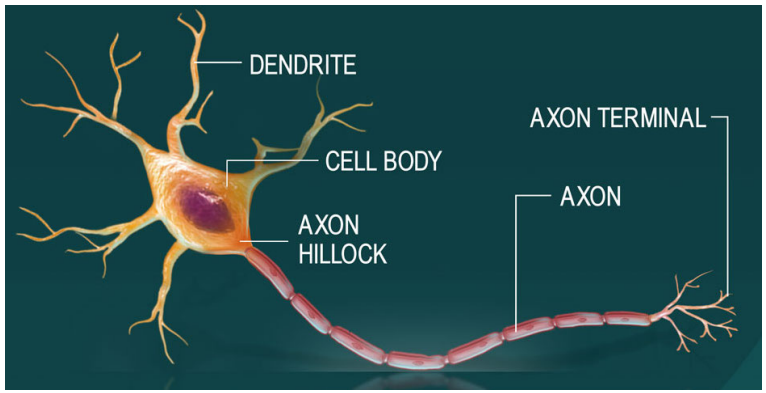
- (Artificial) neural networks are computational models loosely inspired by the brain.

Neural networks

- (Artificial) neural networks are computational models loosely inspired by the brain.
- In the brain, there are many simple computational units (neurons = nerve cells) that perform complex computations by interacting.

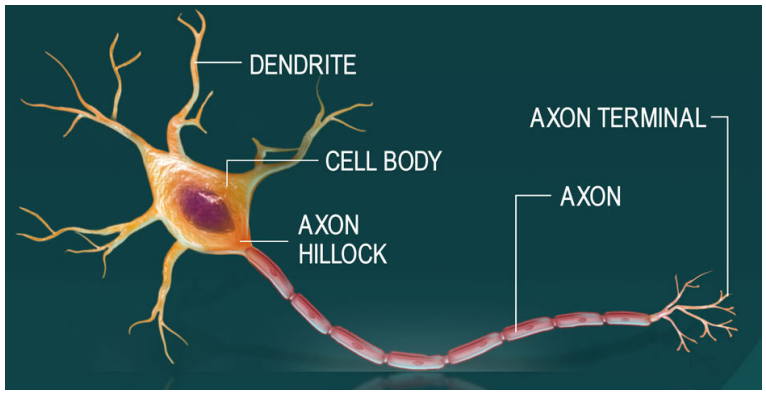
Neural networks

- (Artificial) neural networks are computational models loosely inspired by the brain.
- In the brain, there are many simple computational units (neurons = nerve cells) that perform complex computations by interacting.
- Each biological neuron has:
  - *Dendrites* that receive electrical signals from other neurons.
  - A *cell body* that integrates those signals.
  - An *axon* that, depending on the inputs, allows the neuron to *fire* (send an electrical output to other neurons).



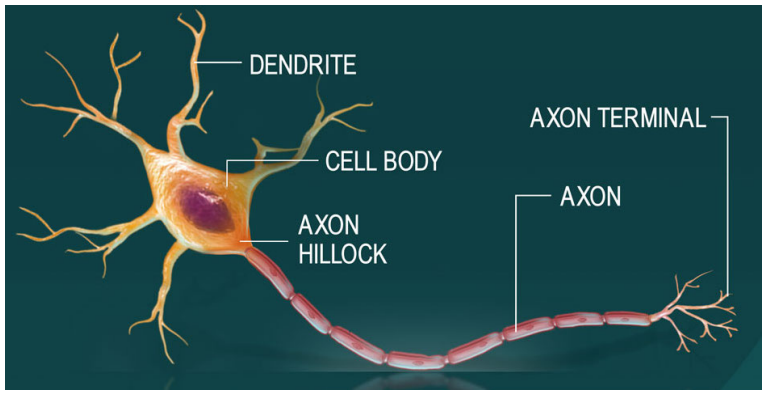
Neural networks

- (Artificial) neural networks are computational models loosely inspired by the brain.
- In the brain, there are many simple computational units (neurons = nerve cells) that perform complex computations by interacting.
- Each biological neuron has:
  - *Dendrites* that receive electrical signals from other neurons.
  - A *cell body* that integrates those signals.
  - An *axon* that, depending on the inputs, allows the neuron to *fire* (send an electrical output to other neurons).



Neural networks

- (Artificial) neural networks are computational models loosely inspired by the brain.
- In the brain, there are many simple computational units (neurons = nerve cells) that perform complex computations by interacting.
- Each biological neuron has:
  - *Dendrites* that receive electrical signals from other neurons.
  - A *cell body* that integrates those signals.
  - An *axon* that, depending on the inputs, allows the neuron to *fire* (send an electrical output to other neurons).

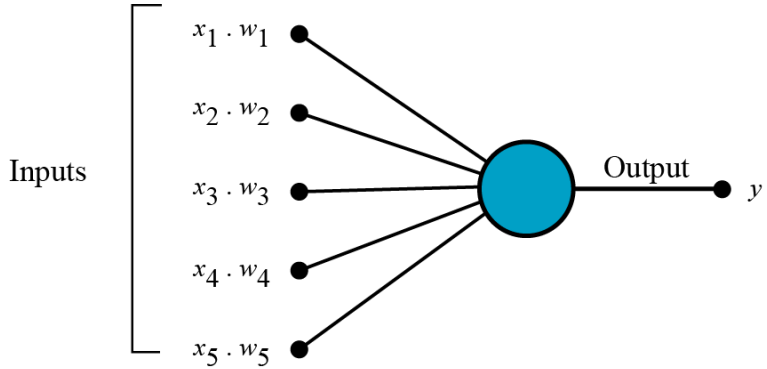


- Whole brain = many neurons that fire together in patterns that in turn lead to other firing patterns.

Perceptrons as a model for biological neurons

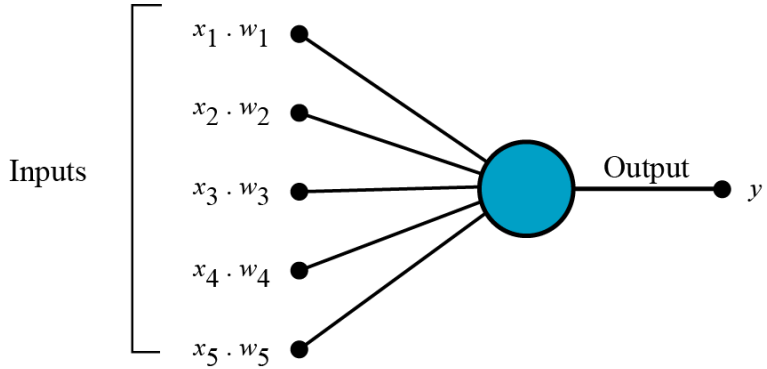
Perceptrons as a model for biological neurons

- Perceptrons = simple model for biological neurons: If a linear combination of inputs is large enough, then the neuron fires, otherwise it doesn't.



Perceptrons as a model for biological neurons

- Perceptrons = simple model for biological neurons: If a linear combination of inputs is large enough, then the neuron fires, otherwise it doesn't.

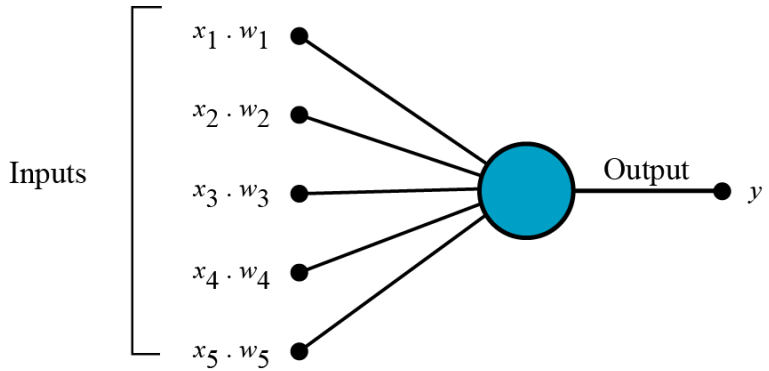


- More generally, we can use a perceptron with a *threshold* or *bias* b:

$$y(x) = 1 \text{ if } w \cdot x - b > 0, \text{ otherwise } 0.$$

Perceptrons as a model for biological neurons

- Perceptrons = simple model for biological neurons: If a linear combination of inputs is large enough, then the neuron fires, otherwise it doesn't.



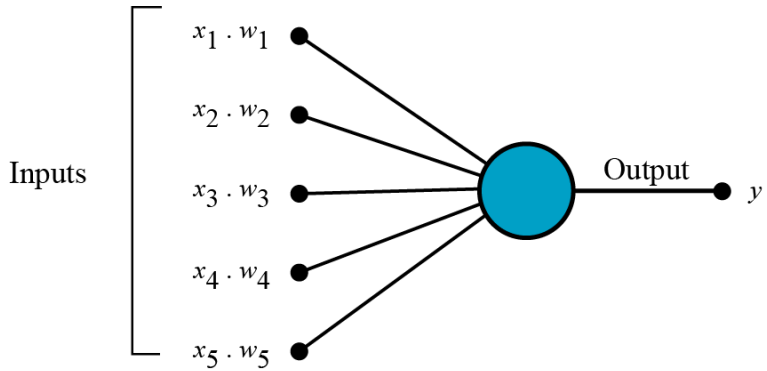
- More generally, we can use a perceptron with a *threshold* or *bias* b:

$$y(x) = 1 \text{ if } w \cdot x - b > 0, \text{ otherwise } 0.$$

- This is a good simple model of how a biological neuron actually works.

Perceptrons as a model for biological neurons

- Perceptrons = simple model for biological neurons: If a linear combination of inputs is large enough, then the neuron fires, otherwise it doesn't.



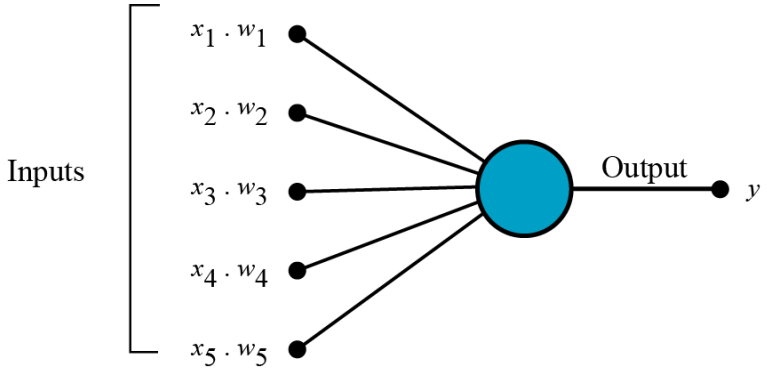
- More generally, we can use a perceptron with a *threshold* or *bias* b:

$$y(x) = 1 \text{ if } w \cdot x - b > 0, \text{ otherwise } 0.$$

- This is a good simple model of how a biological neuron actually works.
- If a linear combination of inputs is bigger than threshold b, then it fires.

Perceptrons as a model for biological neurons

- Perceptrons = simple model for biological neurons: If a linear combination of inputs is large enough, then the neuron fires, otherwise it doesn't.



- More generally, we can use a perceptron with a *threshold* or *bias* b:

$$y(x) = 1 \text{ if } w \cdot x - b > 0, \text{ otherwise } 0.$$

- This is a good simple model of how a biological neuron actually works.
- If a linear combination of inputs is bigger than threshold b, then it fires.
- (Biologically, this works with voltage-gated ion channels – if the input electrical signal is big enough, then the neuron lets in charged ions and sends an electrical signal of its own.)

Perceptrons as a model for biological neurons

- Perceptrons = simple model for biological neurons: If a linear combination of inputs is large enough, then the neuron fires, otherwise it doesn't.
- More generally, we can use a perceptron with a *threshold* or *bias* b:

$$y(x) = 1 \text{ if } w \cdot x - b > 0, \text{ otherwise } 0.$$

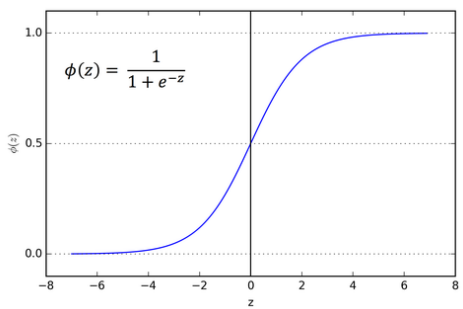
- This is a good simple model of how a biological neuron actually works.
- If a linear combination of inputs is bigger than threshold b, then it fires.
- (Biologically, this works with voltage-gated ion channels – if the input electrical signal is big enough, then the neuron lets in charged ions and sends an electrical signal of its own.)
- Can generalize to $y(x) = \sigma(w \cdot x - b)$, where σ is a nonlinear function (since neurons aren't quite as exact as on and off).

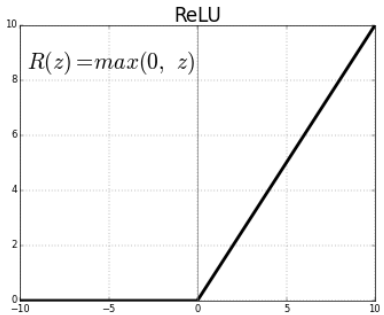
Perceptrons as a model for biological neurons

- Perceptrons = simple model for biological neurons: If a linear combination of inputs is large enough, then the neuron fires, otherwise it doesn't.
- More generally, we can use a perceptron with a *threshold* or *bias* b:

$$y(x) = 1 \text{ if } w \cdot x - b > 0, \text{ otherwise } 0.$$

- This is a good simple model of how a biological neuron actually works.
- If a linear combination of inputs is bigger than threshold b, then it fires.
- (Biologically, this works with voltage-gated ion channels – if the input electrical signal is big enough, then the neuron lets in charged ions and sends an electrical signal of its own.)
- Can generalize to $y(x) = \sigma(w \cdot x - b)$, where σ is a nonlinear function (since neurons aren't quite as exact as on and off).
- Two common choices $\sigma(z)$ are sigmoid $1/(1 + e^{-z})$ and ReLU $\max(0, z)$:

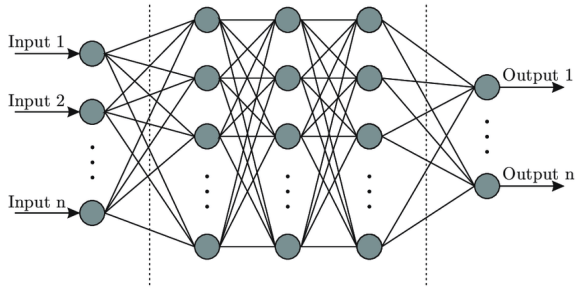




Multilayer perceptrons

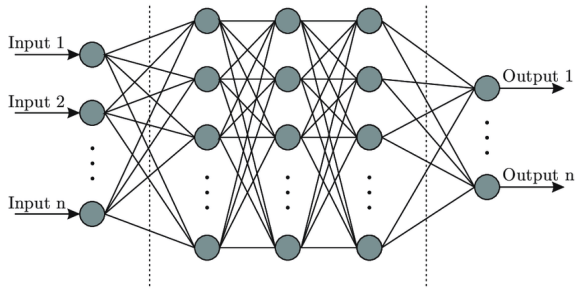
Multilayer perceptrons

- Inspired by the brain, we can combine many perceptrons according to a weighted graph.



Multilayer perceptrons

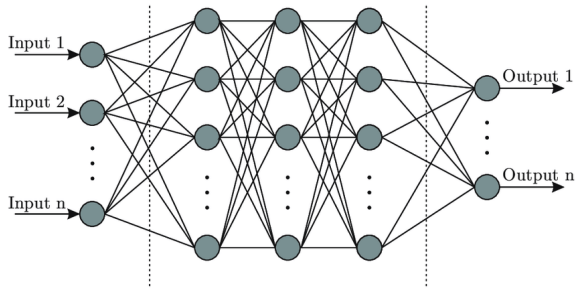
- Inspired by the brain, we can combine many perceptrons according to a weighted graph.



- We define a *multilayer perceptron* (= *fully connected network*) to be a particular kind of function $f(x)$ from inputs $x \in \mathbb{R}^{n_{in}}$ to outputs $y \in \mathbb{R}^{n_{out}}$.

Multilayer perceptrons

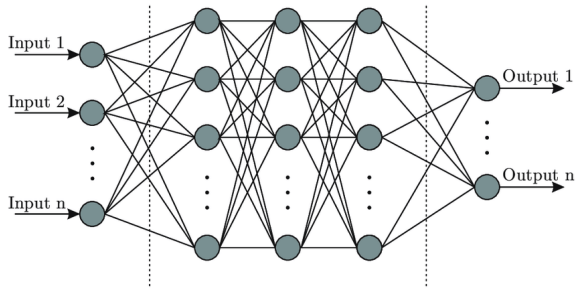
- Inspired by the brain, we can combine many perceptrons according to a weighted graph.



- We define a *multilayer perceptron* (= *fully connected network*) to be a particular kind of function $f(x)$ from inputs $x \in \mathbb{R}^{n_{in}}$ to outputs $y \in \mathbb{R}^{n_{out}}$.
- Specifically, it computes its output according to various intermediates, called *hidden layers*.

Multilayer perceptrons

- Inspired by the brain, we can combine many perceptrons according to a weighted graph.



- We define a *multilayer perceptron* (= *fully connected network*) to be a particular kind of function $f(x)$ from inputs $x \in \mathbb{R}^{n_{in}}$ to outputs $y \in \mathbb{R}^{n_{out}}$.
- Specifically, it computes its output according to various intermediates, called *hidden layers*.
- Given a particular nonlinear function $\sigma(\cdot)$, we define:

$$h^1 = \sigma(W^1 x - b^1) \in \mathbb{R}^{n_1}, \quad \text{with } W^1 \in \mathbb{R}^{n_1 \times n_{in}}, b^1 \in \mathbb{R}^{n_1}$$

$$h^2 = \sigma(W^2 h^1 - b^2) \in \mathbb{R}^{n_2}, \quad \text{with } W^2 \in \mathbb{R}^{n_2 \times n_1}, b^2 \in \mathbb{R}^{n_2}$$

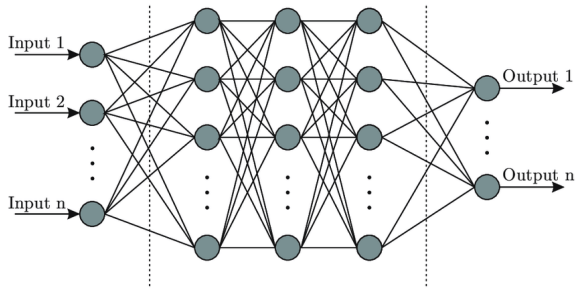
$$\dots$$

$$h^d = \sigma(W^d h^{d-1} - b^d) \in \mathbb{R}^{n_d}, \quad \text{with } W^d \in \mathbb{R}^{n_d \times n_{d-1}}, b^d \in \mathbb{R}^{n_d}$$

$$y = \sigma(W^{d+1} h^d - b^{d+1}) \in \mathbb{R}^{n_{d+1}}, \quad \text{with } W^{d+1} \in \mathbb{R}^{n_{out} \times n_d}, b^{d+1} \in \mathbb{R}^{n_{out}}$$

Multilayer perceptrons

- Inspired by the brain, we can combine many perceptrons according to a weighted graph.

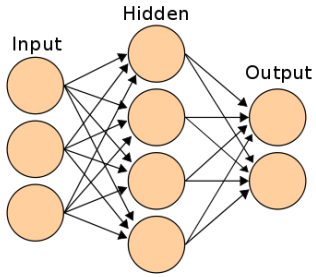


- We define a *multilayer perceptron* (= *fully connected network*) to be a particular kind of function $f(x)$ from inputs $x \in \mathbb{R}^{n_{in}}$ to outputs $y \in \mathbb{R}^{n_{out}}$.
- Specifically, it computes its output according to various intermediates, called *hidden layers*.
- Given a particular nonlinear function $\sigma(\cdot)$, we define:

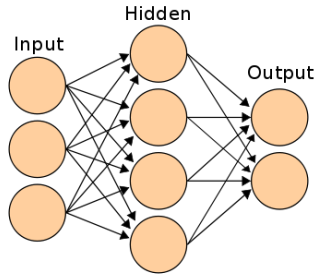
$$h^1 = \sigma(W^1 x - b^1) \in \mathbb{R}^{n_1}, \quad \text{with } W^1 \in \mathbb{R}^{n_1 \times n_{in}}, b^1 \in \mathbb{R}^{n_1}$$
$$h^2 = \sigma(W^2 h^1 - b^2) \in \mathbb{R}^{n_2}, \quad \text{with } W^2 \in \mathbb{R}^{n_2 \times n_1}, b^2 \in \mathbb{R}^{n_2}$$
$$\dots$$
$$h^d = \sigma(W^d h^{d-1} - b^d) \in \mathbb{R}^{n_d}, \quad \text{with } W^d \in \mathbb{R}^{n_d \times n_{d-1}}, b^d \in \mathbb{R}^{n_d}$$
$$y = \sigma(W^{d+1} h^d - b^{d+1}) \in \mathbb{R}^{n_{d+1}}, \quad \text{with } W^{d+1} \in \mathbb{R}^{n_{out} \times n_d}, b^{d+1} \in \mathbb{R}^{n_{out}}$$

- So each layer is a vector of perceptrons applied to the previous layer.

Example



Example

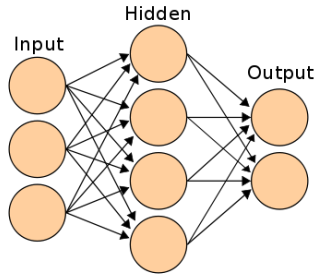


- Let's compute the output of this multilayer perceptron at the point $x = [1, 0, 0]$ with

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

$$b^1 = [1, 1, 1, 1], \quad b^2 = [-1, -1], \quad \sigma(z) = \text{ReLU}(z) = \max(0, z)$$

Example



- Let's compute the output of this multilayer perceptron at the point $x = [1, 0, 0]$ with

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

$$b^1 = [1, 1, 1, 1], \quad b^2 = [-1, -1], \quad \sigma(z) = \text{ReLU}(z) = \max(0, z)$$

$$\begin{aligned} h^1 &= \sigma(W^1 x - b^1) = \text{ReLU}([2, 0, 0, 0] - [1, 1, 1, 1]) \\ &= \text{ReLU}([1, -1, -1, -1]) = [1, 0, 0, 0] \\ y &= \sigma(W^2 h^1 - b^2) = \text{ReLU}([-1, 0] - [-1, -1]) \\ &= \text{ReLU}([0, 1]) = [0, 1]. \end{aligned}$$

Using neural networks

Using neural networks

- We often use neural networks to approximate functions of observed data.

Using neural networks

- We often use neural networks to approximate functions of observed data.
- For example, in classification problems we have $x^1, x^2, \ldots, x^k \in \mathbb{R}^n$ and labels $y^1, y^2, \ldots, y^k$ in some set S (such as “dog”, “cat”, “book”, etc).

Using neural networks

- We often use neural networks to approximate functions of observed data.
- For example, in classification problems we have $x^1, x^2, \ldots, x^k \in \mathbb{R}^n$ and labels $y^1, y^2, \ldots, y^k$ in some set S (such as “dog”, “cat”, “book”, etc).
- We want to learn a function f such that $y^i \approx f(x^i)$.

Using neural networks

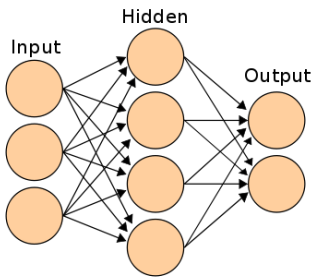
- We often use neural networks to approximate functions of observed data.
- For example, in classification problems we have $x^1, x^2, \ldots, x^k \in \mathbb{R}^n$ and labels $y^1, y^2, \ldots, y^k$ in some set S (such as “dog”, “cat”, “book”, etc).
- We want to learn a function f such that $y^i \approx f(x^i)$.
- One way to do this is to write down a neural network that computes such a function f.

Using neural networks

- We often use neural networks to approximate functions of observed data.
- For example, in classification problems we have $x^1, x^2, \ldots, x^k \in \mathbb{R}^n$ and labels $y^1, y^2, \ldots, y^k$ in some set S (such as "dog", "cat", "book", etc).
- We want to learn a function f such that $y^i \approx f(x^i)$.
- One way to do this is to write down a neural network that computes such a function f.
- Typically, we fix the *structure* (= *architecture*) and activation function of the neural network and change the weights and biases to fit the data.

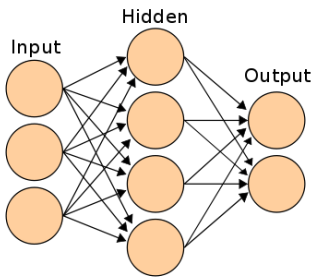
Using neural networks

- We often use neural networks to approximate functions of observed data.
- For example, in classification problems we have $x^1, x^2, \ldots, x^k \in \mathbb{R}^n$ and labels $y^1, y^2, \ldots, y^k$ in some set S (such as “dog”, “cat”, “book”, etc).
- We want to learn a function f such that $y^i \approx f(x^i)$.
- One way to do this is to write down a neural network that computes such a function f.
- Typically, we fix the *structure* (= *architecture*) and activation function of the neural network and change the weights and biases to fit the data.
- Example: we might fix the activation function σ as ReLU and fix the structure as one hidden layer with size 4:



Using neural networks

- We often use neural networks to approximate functions of observed data.
- For example, in classification problems we have $x^1, x^2, \ldots, x^k \in \mathbb{R}^n$ and labels $y^1, y^2, \ldots, y^k$ in some set S (such as "dog", "cat", "book", etc).
- We want to learn a function f such that $y^i \approx f(x^i)$.
- One way to do this is to write down a neural network that computes such a function f.
- Typically, we fix the *structure* (= *architecture*) and activation function of the neural network and change the weights and biases to fit the data.
- Example: we might fix the activation function σ as ReLU and fix the structure as one hidden layer with size 4:



- We change the weight matrices W^1 and W^2 and the bias vectors b^1 and b^2 to create a function that best replicates the patterns observed in data.

Training neural networks

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).
- For example, in our network with 1 hidden layer, increasing b^2 will increase $f_\theta(x^i)$ for every i.

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).
- For example, in our network with 1 hidden layer, increasing b^2 will increase $f_\theta(x^i)$ for every i.
- Changing individual entries of W^1 changes f_θ in more subtle ways.

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).
- For example, in our network with 1 hidden layer, increasing b^2 will increase $f_\theta(x^i)$ for every i.
- Changing individual entries of W^1 changes f_θ in more subtle ways.
- How do we pick θ? The perceptron rule won't work here unfortunately.

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).
- For example, in our network with 1 hidden layer, increasing b^2 will increase $f_\theta(x^i)$ for every i.
- Changing individual entries of W^1 changes f_θ in more subtle ways.
- How do we pick θ? The perceptron rule won't work here unfortunately.
- We can find a good set of parameters θ by using gradient descent.

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).
- For example, in our network with 1 hidden layer, increasing b^2 will increase $f_\theta(x^i)$ for every i.
- Changing individual entries of W^1 changes f_θ in more subtle ways.
- How do we pick θ? The perceptron rule won't work here unfortunately.
- We can find a good set of parameters θ by using gradient descent.
- What is the thing we want to minimize?

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).
- For example, in our network with 1 hidden layer, increasing b^2 will increase $f_\theta(x^i)$ for every i.
- Changing individual entries of W^1 changes f_θ in more subtle ways.
- How do we pick θ? The perceptron rule won't work here unfortunately.
- We can find a good set of parameters θ by using gradient descent.
- We want to minimize the difference between $f_\theta(x^i)$ and y^i for each i.

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).
- For example, in our network with 1 hidden layer, increasing b^2 will increase $f_\theta(x^i)$ for every i.
- Changing individual entries of W^1 changes f_θ in more subtle ways.
- How do we pick θ? The perceptron rule won't work here unfortunately.
- We can find a good set of parameters θ by using gradient descent.
- We want to minimize the difference between $f_\theta(x^i)$ and y^i for each i.
- We pick a *loss function* $\ell(y', y)$ that is 0 if $y' = y$, e.g. $\ell(y', y) = ||y' - y||^2$.

Training neural networks

- We can write θ as a big vector with all the parameters we are learning - e.g. $W^1, W^2, \ldots, b^1, b^2, \ldots$.
- Then, the function $f(x)$ that the neural network is computing will depend on θ - we can write $f_\theta(x)$.
- We want to change θ so that $f_\theta(x^i) \approx y^i$ for each datapoint (x^i, y^i).
- For example, in our network with 1 hidden layer, increasing b^2 will increase $f_\theta(x^i)$ for every i.
- Changing individual entries of W^1 changes f_θ in more subtle ways.
- How do we pick θ? The perceptron rule won't work here unfortunately.
- We can find a good set of parameters θ by using gradient descent.
- We want to minimize the difference between $f_\theta(x^i)$ and y^i for each i.
- We pick a *loss function* $\ell(y', y)$ that is 0 if $y' = y$, e.g. $\ell(y', y) = ||y' - y||^2$.
- Then, we want to minimize $\ell(f_\theta(x^i), y^i)$.

Training neural networks

Training neural networks

- We want to minimize $\ell(f_\theta(x^i), y^i)$, where $f_\theta(x^i)$ is the function computed by a neural network with parameters θ.

Training neural networks

- We want to minimize $\ell(f_\theta(x^i), y^i)$, where $f_\theta(x^i)$ is the function computed by a neural network with parameters θ.
- If we are using gradient descent with θ, we need to calculate the gradient with respect to θ.

Training neural networks

- We want to minimize $\ell(f_\theta(x^i), y^i)$, where $f_\theta(x^i)$ is the function computed by a neural network with parameters θ.
- If we are using gradient descent with θ, we need to calculate the gradient with respect to θ.
- I.e., for each individual parameter θ_j, we need the partial derivative

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

Training neural networks

- We want to minimize $\ell(f_\theta(x^i), y^i)$, where $f_\theta(x^i)$ is the function computed by a neural network with parameters θ.
- If we are using gradient descent with θ, we need to calculate the gradient with respect to θ.
- I.e., for each individual parameter θ_j, we need the partial derivative

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

- Let's do this for a perceptron: $f_\theta(x) = \sigma(w \cdot x - b)$ for $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$.

Training neural networks

- We want to minimize $\ell(f_\theta(x^i), y^i)$, where $f_\theta(x^i)$ is the function computed by a neural network with parameters θ.
- If we are using gradient descent with θ, we need to calculate the gradient with respect to θ.
- I.e., for each individual parameter θ_j, we need the partial derivative

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

- Let's do this for a perceptron: $f_\theta(x) = \sigma(w \cdot x - b)$ for $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$.
- Then, the parameters θ_j are (i) the entries of w and (ii) the bias b.

Training neural networks

- We want to minimize $\ell(f_\theta(x^i), y^i)$, where $f_\theta(x^i)$ is the function computed by a neural network with parameters θ.
- If we are using gradient descent with θ, we need to calculate the gradient with respect to θ.
- I.e., for each individual parameter θ_j, we need the partial derivative

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

- Let's do this for a perceptron: $f_\theta(x) = \sigma(w \cdot x - b)$ for $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$.
- Then, the parameters θ_j are (i) the entries of w and (ii) the bias b.
- Let's suppose that $\ell(f_\theta(x^i), y^i) = (f_\theta(x^i) - y^i)^2$. Then, by the chain rule:

Training neural networks

- We want to minimize $\ell(f_\theta(x^i), y^i)$, where $f_\theta(x^i)$ is the function computed by a neural network with parameters θ.
- If we are using gradient descent with θ, we need to calculate the gradient with respect to θ.
- I.e., for each individual parameter θ_j, we need the partial derivative

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

- Let's do this for a perceptron: $f_\theta(x) = \sigma(w \cdot x - b)$ for $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$.
- Then, the parameters θ_j are (i) the entries of w and (ii) the bias b.
- Let's suppose that $\ell(f_\theta(x^i), y^i) = (f_\theta(x^i) - y^i)^2$. Then, by the chain rule:

$$\begin{aligned}
\frac{\partial}{\partial b} \ell(f_\theta(x^i), y^i) &= \frac{\partial}{\partial b} (\sigma(w \cdot x^i - b) - y^i)^2 \\
&= 2(\sigma(w \cdot x^i - b) - y^i) \times \sigma'(w \cdot x^i - b) \times \frac{\partial}{\partial b}(w \cdot x^i - b) \\
&= -2(\sigma(w \cdot x^i - b) - y^i)\sigma'(w \cdot x^i - b).
\end{aligned}$$

Training neural networks

- We want to minimize $\ell(f_\theta(x^i), y^i)$, where $f_\theta(x^i)$ is the function computed by a neural network with parameters θ.
- If we are using gradient descent with θ, we need to calculate the gradient with respect to θ.
- I.e., for each individual parameter θ_j, we need the partial derivative

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

- Let's do this for a perceptron: $f_\theta(x) = \sigma(w \cdot x - b)$ for $w \in \mathbb{R}^n$ and $b \in \mathbb{R}$.
- Then, the parameters θ_j are (i) the entries of w and (ii) the bias b.
- Let's suppose that $\ell(f_\theta(x^i), y^i) = (f_\theta(x^i) - y^i)^2$. Then, by the chain rule:

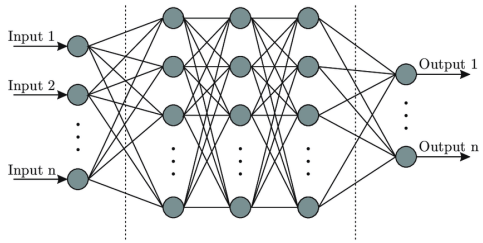
$$\begin{aligned}
\frac{\partial}{\partial w_j} \ell(f_\theta(x^i), y^i) &= \frac{\partial}{\partial w_j} (\sigma(w \cdot x^i - b) - y^i)^2 \\
&= 2(\sigma(w \cdot x^i - b) - y^i) \times \sigma'(w \cdot x^i - b) \times \frac{\partial}{\partial w_j}(w \cdot x^i - b) \\
&= 2(\sigma(w \cdot x^i - b) - y^i) \times \sigma'(w \cdot x^i - b) \times x^i_j.
\end{aligned}$$

Backpropagation

Backpropagation

- For a general neural network, we can similarly use the chain rule to compute the gradients with respect to each parameter of the network:

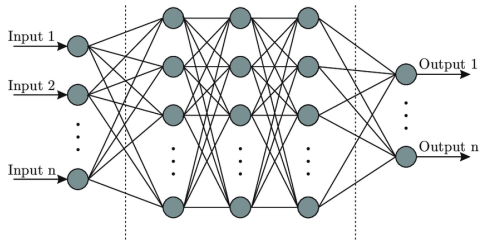
$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$



Backpropagation

- For a general neural network, we can similarly use the chain rule to compute the gradients with respect to each parameter of the network:

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

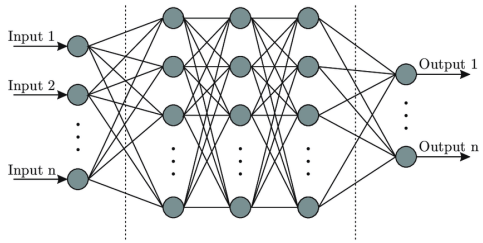


- There is an especially fast way to do this by *backpropagation* – i.e. working out the gradients with respect to the parameters in the last layer, then the next to last layer, and so on.

Backpropagation

- For a general neural network, we can similarly use the chain rule to compute the gradients with respect to each parameter of the network:

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

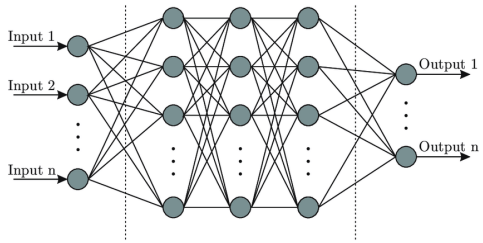


- There is an especially fast way to do this by *backpropagation* – i.e. working out the gradients with respect to the parameters in the last layer, then the next to last layer, and so on.
- (This is a form of dynamic programming!)

Backpropagation

- For a general neural network, we can similarly use the chain rule to compute the gradients with respect to each parameter of the network:

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

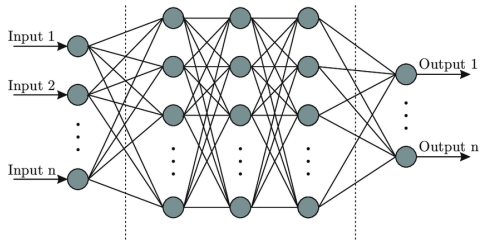


- There is an especially fast way to do this by *backpropagation* – i.e. working out the gradients with respect to the parameters in the last layer, then the next to last layer, and so on.
- (This is a form of dynamic programming!)
- Overall learning algorithm:

Backpropagation

- For a general neural network, we can similarly use the chain rule to compute the gradients with respect to each parameter of the network:

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

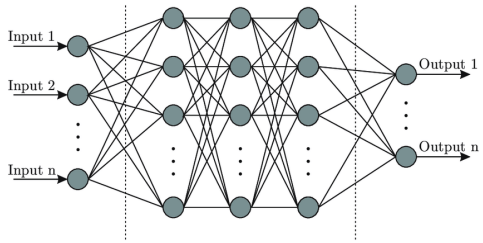


- There is an especially fast way to do this by *backpropagation* – i.e. working out the gradients with respect to the parameters in the last layer, then the next to last layer, and so on.
- (This is a form of dynamic programming!)
- Overall learning algorithm:
  - Pick a structure / activation functions for the network.

Backpropagation

- For a general neural network, we can similarly use the chain rule to compute the gradients with respect to each parameter of the network:

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

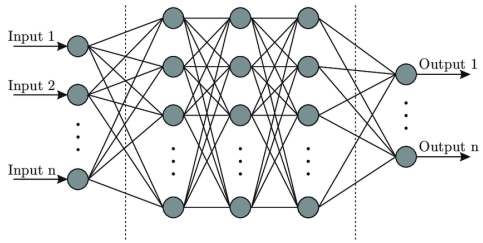


- There is an especially fast way to do this by *backpropagation* – i.e. working out the gradients with respect to the parameters in the last layer, then the next to last layer, and so on.
- (This is a form of dynamic programming!)
- Overall learning algorithm:
  - Pick a structure / activation functions for the network.
  - Pick some starting parameters θ – typically random numbers.

Backpropagation

- For a general neural network, we can similarly use the chain rule to compute the gradients with respect to each parameter of the network:

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$

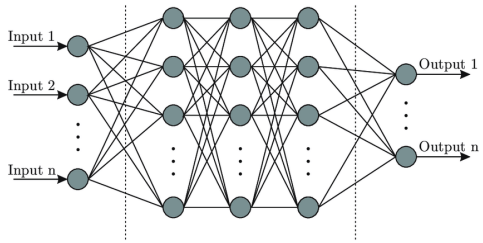


- There is an especially fast way to do this by *backpropagation* – i.e. working out the gradients with respect to the parameters in the last layer, then the next to last layer, and so on.
- (This is a form of dynamic programming!)
- Overall learning algorithm:
  - Pick a structure / activation functions for the network.
  - Pick some starting parameters θ – typically random numbers.
  - Update θ by gradient descent until $f_\theta(x^i) \approx y^i$ for all datapoints (x^i, y^i).

Backpropagation

- For a general neural network, we can similarly use the chain rule to compute the gradients with respect to each parameter of the network:

$$\frac{\partial}{\partial \theta_j} \ell(f_\theta(x^i), y^i).$$



- There is an especially fast way to do this by *backpropagation* – i.e. working out the gradients with respect to the parameters in the last layer, then the next to last layer, and so on.
- (This is a form of dynamic programming!)
- Overall learning algorithm:
  - Pick a structure / activation functions for the network.
  - Pick some starting parameters θ – typically random numbers.
  - Update θ by gradient descent until $f_\theta(x^i) \approx y^i$ for all datapoints (x^i, y^i).
- (Even for a perceptron, gradient descent can be better than the perceptron learning rule if the data is not perfectly linearly separable.)

So why does deep learning work?

So why does deep learning work?

- We saw last class that gradient descent works well with convex functions since all local minima are global minima.

So why does deep learning work?

- We saw last class that gradient descent works well with convex functions since all local minima are global minima.
- Here we are performing gradient descent on θ to minimize $\ell(f_\theta(x^i), y^i)$. Is this a convex function of θ?

So why does deep learning work?

- We saw last class that gradient descent works well with convex functions since all local minima are global minima.
- Here we are performing gradient descent on θ to minimize $\ell(f_\theta(x^i), y^i)$. Is this a convex function of θ?
- No! Neural network optimization is almost never convex.

So why does deep learning work?

- We saw last class that gradient descent works well with convex functions since all local minima are global minima.
- Here we are performing gradient descent on θ to minimize $\ell(f_\theta(x^i), y^i)$. Is this a convex function of θ?
- No! Neural network optimization is almost never convex.
- Two reasons why this isn't a huge problem: (i) Usually close to a global minimum, (ii) few local minima that aren't global minima (even though the function isn't convex).

So why does deep learning work?

- We saw last class that gradient descent works well with convex functions since all local minima are global minima.
- Here we are performing gradient descent on θ to minimize $\ell(f_\theta(x^i), y^i)$. Is this a convex function of θ?
- No! Neural network optimization is almost never convex.
- Two reasons why this isn't a huge problem: (i) Usually close to a global minimum, (ii) few local minima that aren't global minima (even though the function isn't convex).
- Intuition for (i): Many global minima because so many parameters – *often more parameters than datapoints*. So probably a global min nearby.

So why does deep learning work?

- We saw last class that gradient descent works well with convex functions since all local minima are global minima.
- Here we are performing gradient descent on θ to minimize $\ell(f_\theta(x^i), y^i)$. Is this a convex function of θ?
- No! Neural network optimization is almost never convex.
- Two reasons why this isn't a huge problem: (i) Usually close to a global minimum, (ii) few local minima that aren't global minima (even though the function isn't convex).
- Intuition for (i): Many global minima because so many parameters – *often more parameters than datapoints*. So probably a global min nearby.
- Intuition for (ii): Given so many parameters, generally there is some parameter we can change to decrease the loss, unless we are truly at a global minimum.

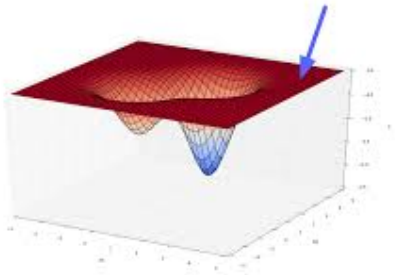
So why does deep learning work?

- We saw last class that gradient descent works well with convex functions since all local minima are global minima.
- Here we are performing gradient descent on θ to minimize $\ell(f_\theta(x^i), y^i)$. Is this a convex function of θ?
- No! Neural network optimization is almost never convex.
- Two reasons why this isn't a huge problem: (i) Usually close to a global minimum, (ii) few local minima that aren't global minima (even though the function isn't convex).
- Intuition for (i): Many global minima because so many parameters – *often more parameters than datapoints*. So probably a global min nearby.
- Intuition for (ii): Given so many parameters, generally there is some parameter we can change to decrease the loss, unless we are truly at a global minimum.
- However, we still don't really know why deep learning works so well!

So why does deep learning work?

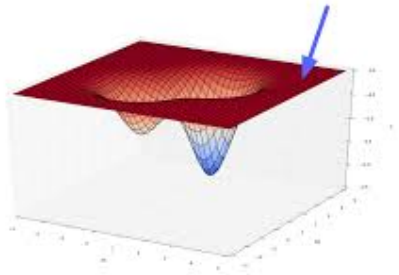
So why does deep learning work?

- The problem of flat regions in gradient descent is very common in deep learning.

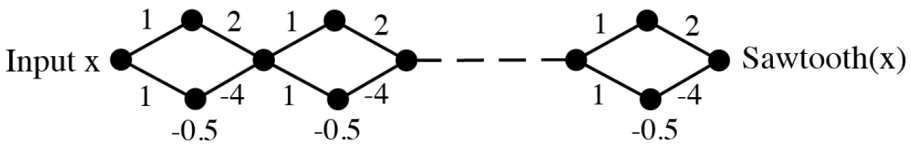


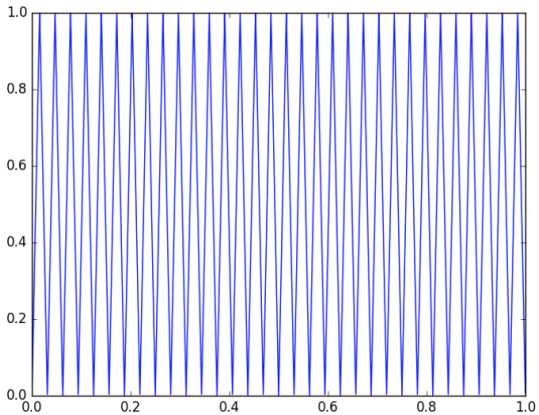
So why does deep learning work?

- The problem of flat regions in gradient descent is very common in deep learning.



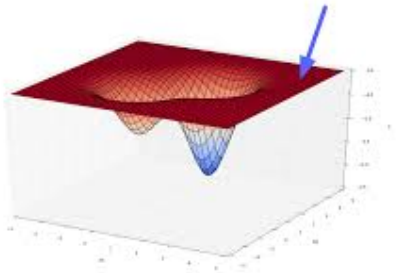
- For example, this neural network expresses the sawtooth function:



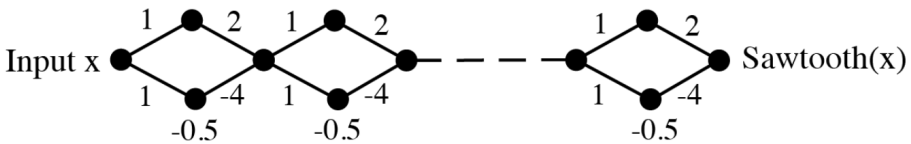


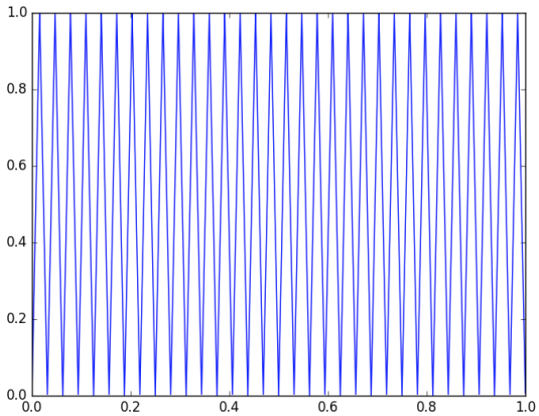
So why does deep learning work?

- The problem of flat regions in gradient descent is very common in deep learning.



- For example, this neural network expresses the sawtooth function:





- If you were to set the weights perfectly, it would work, but unless you are close already the gradient is very low.

Activation functions

Activation functions

- Which activation function should we pick?

Activation functions

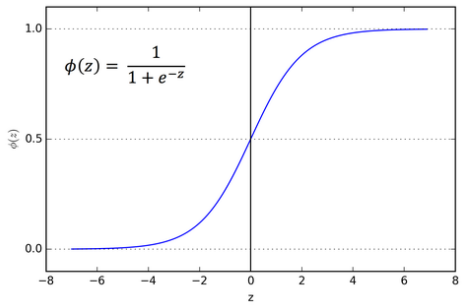
- Which activation function should we pick?
- Remember in the gradient computation we got this: $\sigma'(w \cdot x^i - b)$.

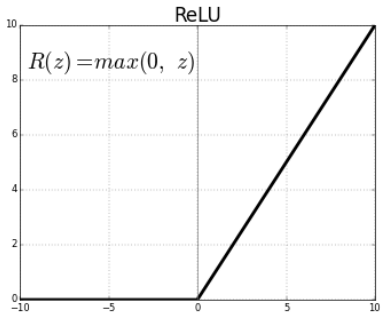
Activation functions

- Which activation function should we pick?
- Remember in the gradient computation we got this: $\sigma'(w \cdot x^i - b)$.
- If σ' is close to 0 then all our gradients vanish.

Activation functions

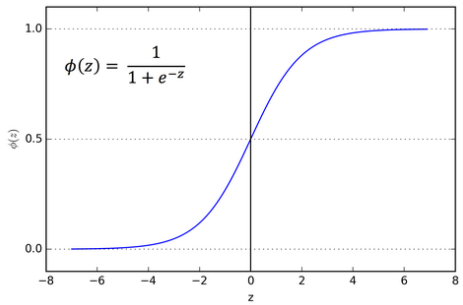
- Which activation function should we pick?
- Remember in the gradient computation we got this: $\sigma'(w \cdot x^i - b)$.
- If σ' is close to 0 then all our gradients vanish.
- This is one reason people often use ReLU $\max(0, z)$ instead of sigmoid $1/(1 + e^{-z})$:

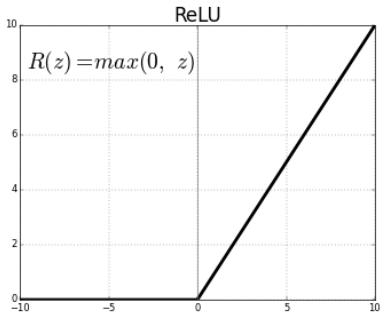




Activation functions

- Which activation function should we pick?
- Remember in the gradient computation we got this: $\sigma'(w \cdot x^i - b)$.
- If σ' is close to 0 then all our gradients vanish.
- This is one reason people often use ReLU $\max(0, z)$ instead of sigmoid $1/(1 + e^{-z})$:

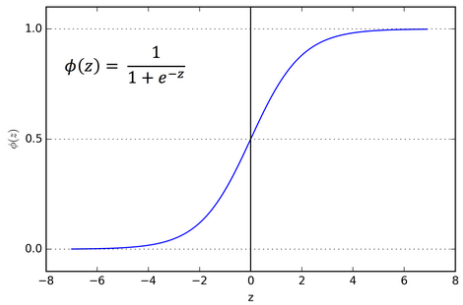


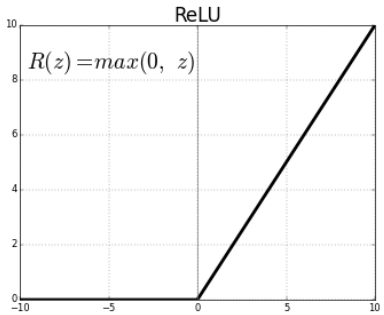


- Derivative of sigmoid is almost always close to 0, so there is very little signal in learning.

Activation functions

- Which activation function should we pick?
- Remember in the gradient computation we got this: $\sigma'(w \cdot x^i - b)$.
- If σ' is close to 0 then all our gradients vanish.
- This is one reason people often use ReLU $\max(0, z)$ instead of sigmoid $1/(1 + e^{-z})$:

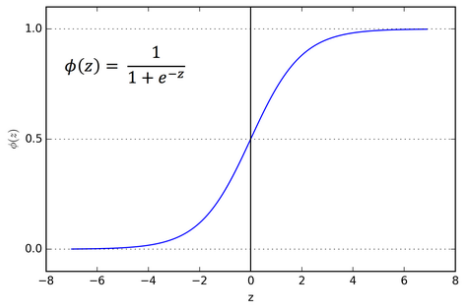


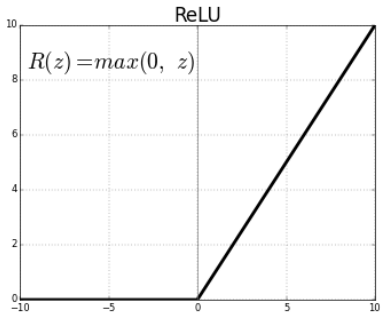


- Derivative of sigmoid is almost always close to 0, so there is very little signal in learning.
- The derivative of ReLU is 0 for negative inputs, but typically that occurs for only about half the neurons.

Activation functions

- Which activation function should we pick?
- Remember in the gradient computation we got this: $\sigma'(w \cdot x^i - b)$.
- If σ' is close to 0 then all our gradients vanish.
- This is one reason people often use ReLU $\max(0, z)$ instead of sigmoid $1/(1 + e^{-z})$:





- Derivative of sigmoid is almost always close to 0, so there is very little signal in learning.
- The derivative of ReLU is 0 for negative inputs, but typically that occurs for only about half the neurons.
- Typically each ReLU will be 0 for some inputs and > 0 for others, which means about half of the weights to be learning at a particular datapoint.