Machine learning from scratch

Lecture 9: Conclusion (and some deep learning)

Alexis Zubiolo
alexis.zubiolo@gmail.com

Data Science Team Lead @ Adcash

March 30, 2017

Course outline

This is our last lecture.

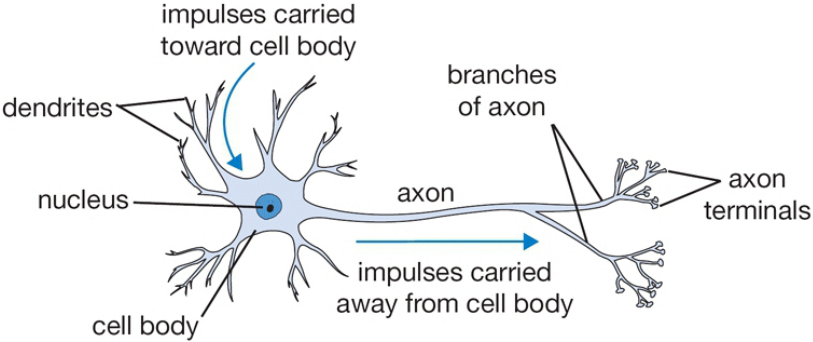
Course outline

This is our last lecture.

Today we'll conclude, and talk a bit about **neural networks**.

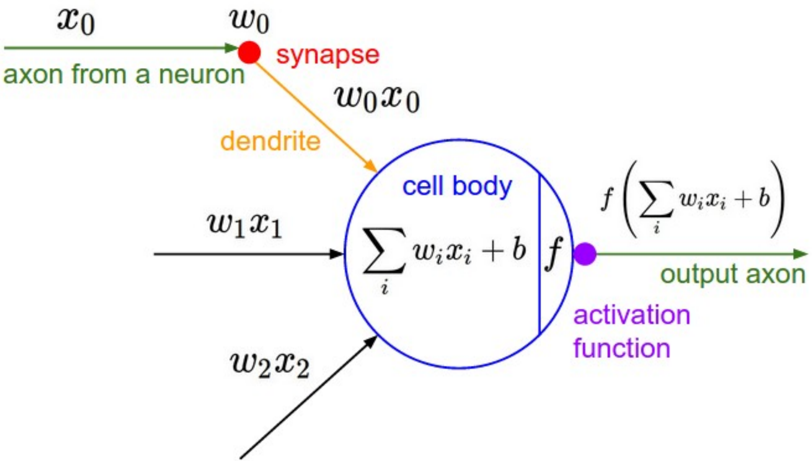
Neural networks

Neuron


impulses carried
toward cell body
dendrites
branches
of axon
axon
nucleus
axon
terminals
impulses carried
away from cell body
cell body

Neuron (mathematical model)

Neural networks are trying to model the logic:



Single neuron case

The neuron we've described performs the same task as the linear ML models we've seen previously:

- Take an input $\boldsymbol{x}$
- Compute the output $\theta^T \boldsymbol{x}$
- Compare the output to the actual label y

Single neuron case

The neuron we've described performs the same task as the linear ML models we've seen previously:

- Take an input $\boldsymbol{x}$
- Compute the output $\theta^T \boldsymbol{x}$
- Compare the output to the actual label y

A neural network will consist in **stacking layers** of neurons in order to build more complex models.

Stacking neuron layers

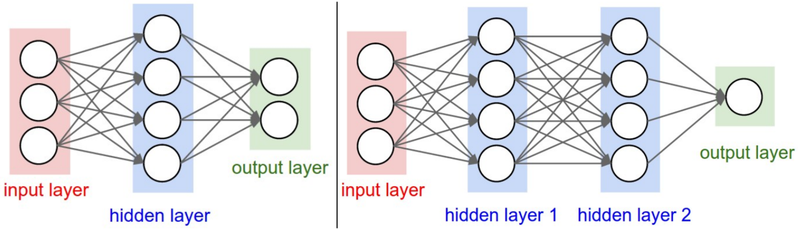
There are 3 types of **neuron layers**:

- **Input layer** (which corresponds to $\boldsymbol{x}$)
- **Output layer** (which correspond to $\hat{y}$)
- **Hidden layers** (layers between the input and output layers)

Stacking neuron layers

There are 3 types of **neuron layers**:

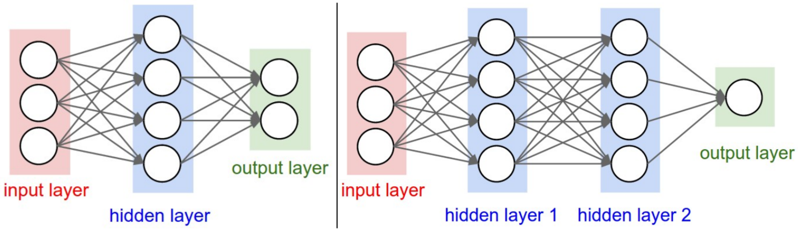
- **Input layer** (which corresponds to $\boldsymbol{x}$)
- **Output layer** (which correspond to $\hat{y}$)
- **Hidden layers** (layers between the input and output layers)



Stacking neuron layers

There are 3 types of **neuron layers**:

- **Input layer** (which corresponds to $\boldsymbol{x}$)
- **Output layer** (which correspond to $\hat{y}$)
- **Hidden layers** (layers between the input and output layers)



The key point is that we introduce **non-linearity between layers**. This non-linearity is often referred to as the **activation function** (more on this later).

Neural network motivation

As we've seen several times, linear models do not necessarily fit the data properly. **Example**:

- `AND`, `OR` and `NOT` logic operators are linearly separable
- `XOR` is not linearly separable

Neural network motivation

As we've seen several times, linear models do not necessarily fit the data properly. **Example**:

- `AND`, `OR` and `NOT` logic operators are linearly separable
- `XOR` is not linearly separable

However: If we transform the inputs with `AND` and `NOT` operators to generate new inputs, then `XOR` becomes linearly separable.

Theoretical guarantees

There is an interesting theoretical result that is in favor of neural networks modeling called the **universal approximation theorem** that states (in a nutshell):

Theoretical guarantees

There is an interesting theoretical result that is in favor of neural networks modeling called the **universal approximation theorem** that states (in a nutshell):

We can approximate any function by a neural network with a neural network with one hidden layer.

Theoretical guarantees

There is an interesting theoretical result that is in favor of neural networks modeling called the **universal approximation theorem** that states (in a nutshell):

We can approximate any function by a neural network with a neural network with one hidden layer.

This tells a lot about how powerful neural networks are. However, this is **just a theoretical result**. We could think that in practice, we just need to use a neural network with one hidden layer with an arbitrary number of neurons (or units), but this does not give good results. Depending on the application, it can be more efficient to **stack multiple hidden layers**.

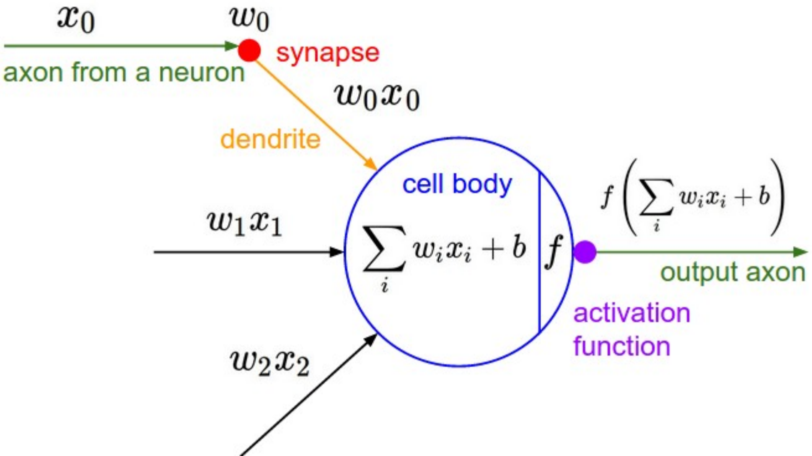
Activation functions

There one issue with stacking neuron layers: The composition of linear function is still linear, so we don't gain any generalization when doing it.

Activation functions

There one issue with stacking neuron layers: The composition of linear function is still linear, so we don't gain any generalization when doing it.

The solution for this is to **introduce non-linearity between layers** (f), this is called the **activation function**.



Activation functions (examples)

The most commonly used activation functions are:

- The **rectified linear unit (ReLU)**:

$$f(x) = \max(0, x)$$

Activation functions (examples)

The most commonly used activation functions are:

- The **rectified linear unit (ReLU)**:

$$f(x) = \max(0, x)$$

- The **logistic sigmoid**:

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

Activation functions (examples)

The most commonly used activation functions are:

- The **rectified linear unit (ReLU)**:

$$f(x) = \max(0, x)$$

- The **logistic sigmoid**:

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

- The **hyperbolic tangent**:

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

Activation functions (examples)

The most commonly used activation functions are:

- The **rectified linear unit (ReLU)**:

$$f(x) = \max(0, x)$$

- The **logistic sigmoid**:

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

- The **hyperbolic tangent**:

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

With the activation functions, we break the linearity between layers, which leads to richer function representations.

Application: Convolutional neural networks

Neural networks got a lot of attention thanks to their results in image processing/computer vision.

Application: Convolutional neural networks

Neural networks got a lot of attention thanks to their results in image processing/computer vision.

The architecture is a bit different in this case. In images, there is **a notion of neighborhood** that justifies the use of **convolutional neural networks** (CNN).

Application: Convolutional neural networks

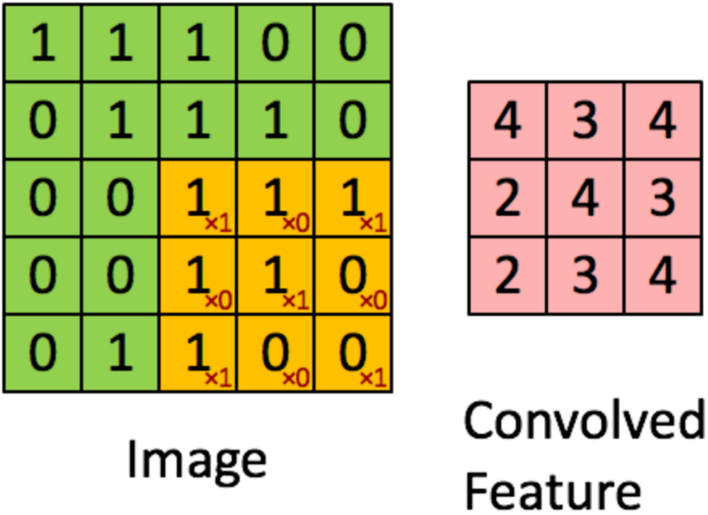
Neural networks got a lot of attention thanks to their results in image processing/computer vision.

The architecture is a bit different in this case. In images, there is **a notion of neighborhood** that justifies the use of **convolutional neural networks** (CNN).

The neural networks we've mentioned so far are **fully-connected**, which means that between 2 consecutive layers, all the units are connected. For CNNs, many of the connections are missing.

Application: Convolutional neural networks

A convolution looks like that:

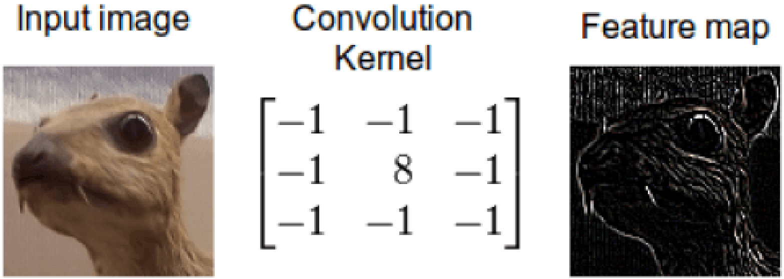


Application: Convolutional neural networks

Convolutions help detect some patterns (e.g. edges) in images:

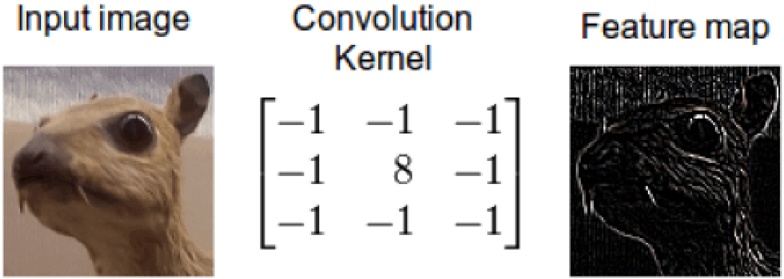
Application: Convolutional neural networks

Convolutions help detect some patterns (e.g. edges) in images:



Application: Convolutional neural networks

Convolutions help detect some patterns (e.g. edges) in images:



The advantage of CNNs is that they can **learn the convolutions kernels** and **find patterns**. We often say that they **learn features** (as well as learning the model!) for this reason.

Application: Neural auto-encoder

Auto-encoders are useful for compressing information. They usually have:

- An input and an output layer (of course)
- A **hidden layer with a low number of units**
- (optional) Other hidden layers to make the model more complex

Training neural networks

So far, we've seen how neural nets are defined. How do we train them?

Training neural networks

So far, we've seen how neural nets are defined. How do we train them?

In the end, a neural network takes an input $\boldsymbol{x}$ and returns an output $\hat{y}$ that we hope is close enough to y. Hence, we can define a loss function (such as the squared loss)

$$\ell(y, \hat{y}) = (y - \hat{y})^2 .$$

Training neural networks

So far, we've seen how neural nets are defined. How do we train them?

In the end, a neural network takes an input $\boldsymbol{x}$ and returns an output $\hat{y}$ that we hope is close enough to y. Hence, we can define a loss function (such as the squared loss)

$$\ell(y, \hat{y}) = (y - \hat{y})^2 .$$

The logic is exactly the same as with linear models:

- We define a way to model the output $\hat{y}$
- We chose a loss function ℓ
- We optimize the parameters so that $\hat{y} \approx y$

Training neural networks

So far, we've seen how neural nets are defined. How do we train them?

In the end, a neural network takes an input $\boldsymbol{x}$ and returns an output $\hat{y}$ that we hope is close enough to y. Hence, we can define a loss function (such as the squared loss)

$$\ell(y, \hat{y}) = (y - \hat{y})^2 \,.$$

The logic is exactly the same as with linear models:

- We define a way to model the output $\hat{y}$
- We chose a loss function ℓ
- We optimize the parameters so that $\hat{y} \approx y$

The only difference is that a neural network is a more sophisticated way to model the data.

Training neural networks

To train the model, we can use gradient descent algorithm (as usual).

Training neural networks

To train the model, we can use gradient descent algorithm (as usual).

Also, neural networks are **prone to overfitting** (a lot!), so regularizing them is important. As with other linear models, using an ℓ_2-norm penalty works well in practice.

Computing the gradient

Due to stacking layers and having coefficients at each layer, the gradient is a bit harder to compute.

Computing the gradient

Due to stacking layers and having coefficients at each layer, the gradient is a bit harder to compute.

The method used to compute the gradient and update the weights is the **backpropagation**. It is based on the chain rule:

$$\frac{\partial f}{\partial x} = \frac{df}{dq}\frac{dq}{dx}$$

when we want to compute the derivative of $f(q(x))$.

Conclusions

Neural networks are very similar to the simpler models we've seen throughout this course. They just rely on a more sophisticated way to model $\hat{y}$.

Conclusions

Neural networks are very similar to the simpler models we've seen throughout this course. They just rely on a more sophisticated way to model $\hat{y}$.

ML algorithm (almost) systematically follow **the same logic**:

- Get some data points $(\boldsymbol{x}, y)$
- Define a way to model the prediction (e.g. $\theta^T \boldsymbol{x}$)
- Define a loss function ℓ
- Optimize the loss over the whole training set

Thank you! Questions?

P̌S: I've updated a few of the notebooks on the GitHub pages. I'll do some more polishing by the end of the week.