ELL 881: Fundamentals of Deep Learning

Lec 03a: Deep Feedforward Networks

Vineet Kumar

August 8, 2018

Table of Contents

Table of Contents

Deep Feedforward Networks

- Also known as **Feedforward Neural Networks** or **Multilayer perceptrons** (MLP)
- In this class, we will refer deep feed forward networks as MLP

Deep Feedforward Networks

- Also known as **Feedforward Neural Networks** or **Multilayer perceptrons** (MLP)
- In this class, we will refer deep feed forward networks as MLP
- MLP defines mapping $y = f(\boldsymbol{x}; \boldsymbol{\theta})$

Deep Feedforward Networks

- Also known as **Feedforward Neural Networks** or **Multilayer perceptrons** (MLP)
- In this class, we will refer deep feed forward networks as MLP
- MLP defines mapping $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- f assigns a category y to an input $\boldsymbol{x}$

Deep Feedforward Networks

- Also known as **Feedforward Neural Networks** or **Multilayer perceptrons** (MLP)
- In this class, we will refer deep feed forward networks as MLP
- MLP defines mapping $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- f assigns a category y to an input $\boldsymbol{x}$
- Goal of MLP is to approximate some true function $f^*(\boldsymbol{x})$

Deep Feedforward Networks

- Also known as **Feedforward Neural Networks** or **Multilayer perceptrons** (MLP)
- In this class, we will refer deep feed forward networks as MLP
- MLP defines mapping $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- f assigns a category y to an input $\boldsymbol{x}$
- Goal of MLP is to approximate some true function $f^*(\boldsymbol{x})$
- This is achieved by estimating the parameters $\boldsymbol{\theta}$

Layers as Composition of Functions

- DFN is represented by composing together many functions

Layers as Composition of Functions

- DFN is represented by composing together many functions
- For example, DFN could be constructed by connecting three functions $f^{(1)}, f^{(2)}$ and $f^{(3)}$ in a chain such that:

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

Layers as Composition of Functions

- DFN is represented by composing together many functions
- For example, DFN could be constructed by connecting three functions $f^{(1)}, f^{(2)}$ and $f^{(3)}$ in a chain such that:

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

- We call $f^{(1)}$ the first layer, $f^{(2)}$ the second layer ...
- The final layer is called the **output layer**

Multiple Layers

- MLP can be thought of as a *black box*
- MLP makes a prediction $f(\boldsymbol{x})$ for an input $\boldsymbol{x}$

Multiple Layers

- MLP can be thought of as a *black box*
- MLP makes a prediction $f(\boldsymbol{x})$ for an input $\boldsymbol{x}$
- MLP is trained by *guiding* the predictions as close as possible to the ground truth category y

Multiple Layers

- MLP can be thought of as a *black box*
- MLP makes a prediction $f(\boldsymbol{x})$ for an input $\boldsymbol{x}$
- MLP is trained by *guiding* the predictions as close as possible to the ground truth category y
- As we only observe the final layer, all intermediate layers are called hidden

Multiple Layers

- MLP can be thought of as a *black box*
- MLP makes a prediction $f(\boldsymbol{x})$ for an input $\boldsymbol{x}$
- MLP is trained by *guiding* the predictions as close as possible to the ground truth category y
- As we only observe the final layer, all intermediate layers are called hidden
- Final layer is called the output layer

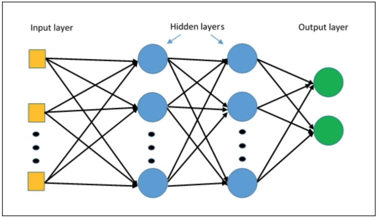

Figure 1: Multi Layer Perceptron
Image: Getting started with Tensorflow, Safari Books, Giancarlo Zaccone

Example code for MLP

Let us see multiple layers in action, via some Tensorflow code!
Notebook: mlp_example_eager.ipynb

Table of Contents

Limitations of Linear Models: Learning XOR

- XOR function is an operation on two binary values x_1 and x_2, which returns 1 only when one of the values is 1.

Limitations of Linear Models: Learning XOR

- XOR function is an operation on two binary values x_1 and x_2, which returns 1 only when one of the values is 1.
- We want to learn XOR function $y = f^*(\boldsymbol{x})$

Limitations of Linear Models: Learning XOR

- XOR function is an operation on two binary values x_1 and x_2, which returns 1 only when one of the values is 1.
- We want to learn XOR function $y = f^*(\boldsymbol{x})$
- Let us try to learn a MLP $y = f(\boldsymbol{x}; \boldsymbol{\theta})$

Limitations of Linear Models: Learning XOR

- XOR function is an operation on two binary values x_1 and x_2, which returns 1 only when one of the values is 1.
- We want to learn XOR function $y = f^*(\boldsymbol{x})$
- Let us try to learn a MLP $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- We only care about $X = \{[0,0]^T, [0,1]^T, [1,0]^T, [1,1]^T\}$

Code: Linear XOR Model

Notebook: xor_eager_keras.ipynb

Why does a linear XOR Model fail?

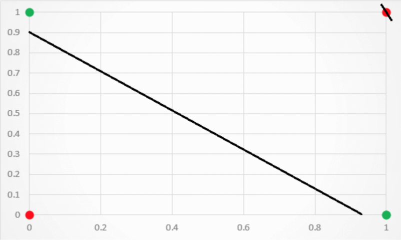

Figure 2: The data points are not separable via a linear function
Image Courtesy:
https://medium.com/@jayeshbahire/the-xor-problem-in-neural-networks-50006411840b

- We cannot find a line which separates $label = 1$ from $label = 0$

Why does a linear XOR Model fail?

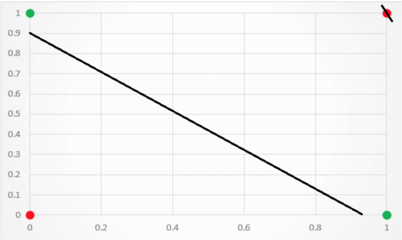

Figure 2: The data points are not separable via a linear function
Image Courtesy: https://medium.com/@jayeshbahire/the-xor-problem-in-neural-networks-50006411840b

- We cannot find a line which separates $label = 1$ from $label = 0$
- Thus, the solution is to add a non linear layer

XOR Model: Adding multiple layers

- We define a two layer network: one *hidden layer* and one *output* layer

XOR Model: Adding multiple layers

- We define a two layer network: one *hidden layer* and one *output* layer
- First layer $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; W, c)$
- Second layer $y = f^{(2)}(\boldsymbol{h}; w, b)$

XOR Model: Adding multiple layers

- We define a two layer network: one *hidden layer* and one *output* layer
- First layer $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; W, c)$
- Second layer $y = f^{(2)}(\boldsymbol{h}; w, b)$
- If both $f^{(1)}$ and $f^{(2)}$ are linear, we effectively have only one linear layer! Why?

XOR Model: Adding multiple layers

- We define a two layer network: one *hidden layer* and one *output* layer
- First layer $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; W, c)$
- Second layer $y = f^{(2)}(\boldsymbol{h}; w, b)$
- If both $f^{(1)}$ and $f^{(2)}$ are linear, we effectively have only one linear layer! Why?
- $f^{(1)}(x) = W^T\boldsymbol{x}$; $f^{(2)}(h) = h^T\boldsymbol{w}$; Thus, $f(\boldsymbol{x}) = w^T W^T \boldsymbol{x}$

XOR Model: Adding multiple layers

Notebook: xor_eager_keras_ml.ipynb

XOR Model: Adding multiple layers

Notebook: xor_eager_keras_ml.ipynb
Add Non-linear Layer

XOR Model: Adding Non-Linear Layer

- We apply a nonlinear function g, such that

$$\boldsymbol{h} = g(W^T \boldsymbol{x} + \boldsymbol{c})$$

- Note, that g is applied elementwise

XOR Model: Adding Non-Linear Layer

- We apply a nonlinear function g, such that

$$\boldsymbol{h} = g(W^T \boldsymbol{x} + \boldsymbol{c})$$

- Note, that g is applied elementwise
- The default recommendation is to use Rectified Linear Unit **ReLU**.
- $g(z) = max\{0, z\}$

XOR Model: Adding Non-Linear Layer

- We apply a nonlinear function g, such that

$$\boldsymbol{h} = g(W^T \boldsymbol{x} + \boldsymbol{c})$$

- Note, that g is applied elementwise
- The default recommendation is to use Rectified Linear Unit **ReLU**.
- $g(z) = max\{0, z\}$
- Note that ReLU is a piecewise linear function, and still has easy to compute derivatives.
- We will talk more about ReLU in next section.

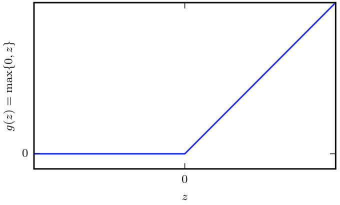

Figure 3: ReLU
Image Courtesy:
http://www.deeplearningbook.org/slides/06_mlp.pdf

XOR Model: Adding Non-Linear Layer

Notebook: xor_eager_keras_ml_relu.ipynb

Table of Contents

Non differentiable Hidden Unit

- Design of hidden units is still an **active research area**

Non differentiable Hidden Unit

- Design of hidden units is still an **active research area**
- ReLU is an excellent default choice

Non differentiable Hidden Unit

- Design of hidden units is still an **active research area**
- ReLU is an excellent default choice
- ReLU $g(z) = max\{0, z\}$ is not differentiable at all points. How can we still use it for learning?

Non differentiable Hidden Unit

- Design of hidden units is still an **active research area**
- ReLU is an excellent default choice
- ReLU $g(z) = max\{0, z\}$ is not differentiable at all points. How can we still use it for learning?
- Neural Network training does not usually reach a local minimum, and it is okay to not have a gradient defined at 0.

Non differentiable Hidden Unit

- Design of hidden units is still an **active research area**
- ReLU is an excellent default choice
- ReLU $g(z) = max\{0, z\}$ is not differentiable at all points. How can we still use it for learning?
- Neural Network training does not usually reach a local minimum, and it is okay to not have a gradient defined at 0.
- Key point to remember: ReLU is not differentiable at 0 but it can be used as its left and right derivative are defined.

Hidden Unit

- Most hidden units, first do an affine transformation of the input

$$z = W^T \boldsymbol{x} + \boldsymbol{b}$$

Hidden Unit

- Most hidden units, first do an affine transformation of the input
$$z = W^T \boldsymbol{x} + \boldsymbol{b}$$
- They later apply an element wise *nonlinear* function $g(z)$ such as ReLU

Hidden Unit

- Most hidden units, first do an affine transformation of the input
$$z = W^T \boldsymbol{x} + \boldsymbol{b}$$
- They later apply an element wise *nonlinear* function $g(z)$ such as ReLU
- Hidden units usually only differ in choice of the function g

More on ReLU

- ReLU is similar to a linear unit
- Gradients are large and consistent even for small values of input!
- This makes learning easier

More on ReLU

- ReLU is similar to a linear unit
- Gradients are large and consistent even for small values of input!
- This makes learning easier
- It is important to initialize the biases with small constant values such as 0.1
- This allows ReLU units activate initially, and allow them to pass gradients through!

Logistic Sigmoid and Hyperbolic Tangent

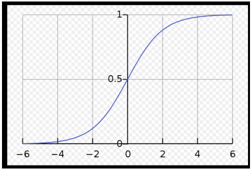

Figure 4: Sigmoid
Image Courtesy:
https://en.wikipedia.org/wiki/Logistic_function

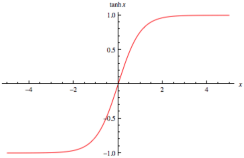

Figure 5: Tanh
Image Courtesy:
http://mathworld.wolfram.com/HyperbolicTangent.html

- Unlike ReLU, sigmoid suffers from *saturation*

Logistic Sigmoid and Hyperbolic Tangent

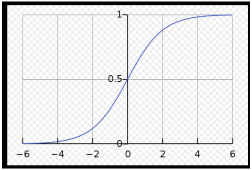

Figure 4: Sigmoid
Image Courtesy:
https://en.wikipedia.org/wiki/Logistic_function

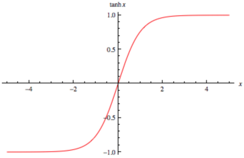

Figure 5: Tanh
Image Courtesy:
http://mathworld.wolfram.com/HyperbolicTangent.html

- Unlike ReLU, sigmoid suffers from *saturation*
- When z is very positive σ saturates to a high value
- When z is very negative σ saturates to a low value
- Sigmoid is only strongly sensitive to the input near zero.
- Thus, use of sigmoids for MLP is discouraged
- A better alternative is to use *tanh*, which behaves like a linear function, when activations are small.

Table of Contents

Architecture Design Choices

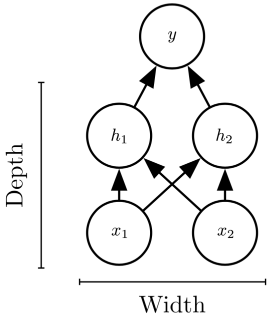

Figure 6: Architecture Design Choices
Image Courtesy: http://www.deeplearningbook.org/slides/06_mlp.pdf

Universal Approximation Properties & Depth

- **Universal Approximation Theorem** states that one hidden layer (such as ReLU or sigmoid) is enough to represent an approximation of any function

Universal Approximation Properties & Depth

- **Universal Approximation Theorem** states that one hidden layer (such as ReLU or sigmoid) is enough to represent an approximation of any function
- However, this theorem only talks about representing a function, and not learning it!

Universal Approximation Properties & Depth

- **Universal Approximation Theorem** states that one hidden layer (such as ReLU or sigmoid) is enough to represent an approximation of any function
- However, this theorem only talks about representing a function, and not learning it!
- MLP only provide a guarantee that there exists some MLP which can estimate a function

Universal Approximation Properties & Depth

- **Universal Approximation Theorem** states that one hidden layer (such as ReLU or sigmoid) is enough to represent an approximation of any function
- However, this theorem only talks about representing a function, and not learning it!
- MLP only provide a guarantee that there exists some MLP which can estimate a function
- In practise, using deeper models can reduce the number of units required to learn a function.

Universal Approximation Properties & Depth

- **Universal Approximation Theorem** states that one hidden layer (such as ReLU or sigmoid) is enough to represent an approximation of any function
- However, this theorem only talks about representing a function, and not learning it!
- MLP only provide a guarantee that there exists some MLP which can estimate a function
- In practise, using deeper models can reduce the number of units required to learn a function.
- Another reason to select deeper network is to define a model in terms of composition of simpler functions.