Intro to Neural Networks

Joël Marbet
CEMFI

Working Group on Econometric Modelling
European Central Bank

September 17, 2021

Introduction

Introduction

- Machine learning (ML) has become incredibly popular. Crucial factors:
  - Advances in computational power of personal computers
  - Increased availability of large datasets
- These novel techniques have a wide variety of applications
  - Computer vision, speech recognition, data mining, and many more
  - Example: AlphaGo is able to beat the best human Go players
- Much hype but barriers to entry not as high as they may seem
- Main topic of this course are neural networks (NN)
  - At the core of many cutting-edge ML models
  - Easy to understand with the mathematical background of economists
  - Many potential (but unexplored) applications in economics

What Are We Going to Do?

- Our main goal is to implement a neural network from scratch
  - Allows us to get a deeper understanding of theory
  - Hopefully, makes it easier to connect to your research
- We will also try to have a very brief look at
  - Some classical results on universal function approximation and the curse of dimensionality
  - Recent applications in economics
- To get you started, we will also briefly discuss freely available machine learning libraries

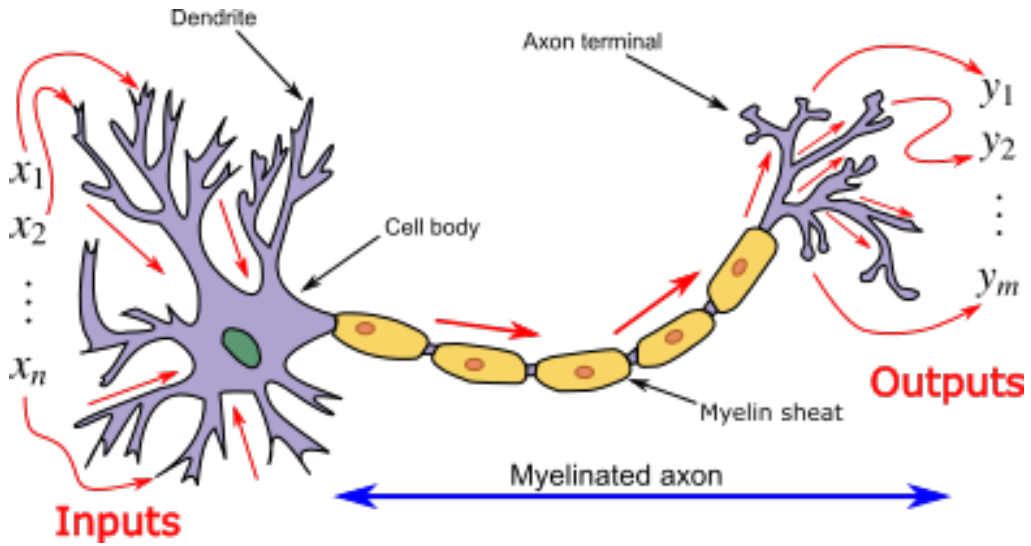
How Are We Going to Do It?

- Implementation in Julia: Why not Matlab, Python, R, ...?
  - High-level + High-performance
  - Geared towards scientific computing
  - Solves two-language problem: All code in Julia, no need for fast low-level C/C++ code
- A very short guide to get you started with Julia is provided
- Since Python is the dominant ML language, an example for TensorFlow/Keras is provided nevertheless

Appendix: Install Julia

An Overview of the Basics

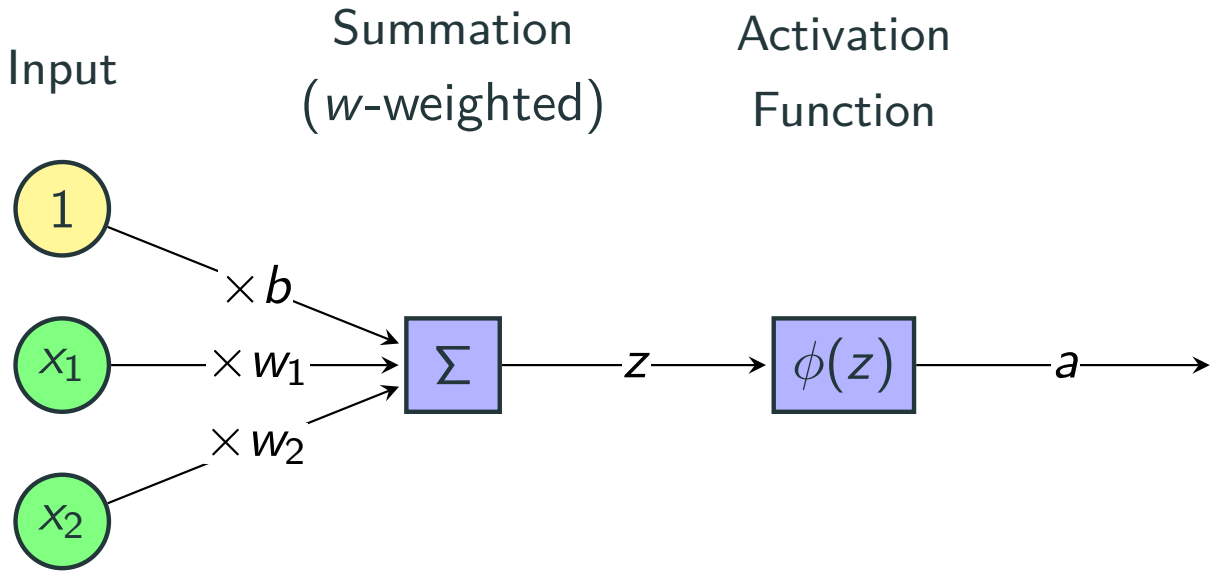
Origins of the Term "Neural Network"



A biological neuron (Source: Wikipedia)

- Origins in attempts to find mathematical representations of information processing in biological systems (Bishop, 2006)
- Nowadays, the biological interpretation is not very important for research
- But the interpretation can be useful when starting to learn about NN

An Artificial Neuron I



- Artifical neurons are the basic building blocks of neural networks
- N inputs denoted $x = (x_1, x_2, \ldots, x_N)'$ and a single output denoted a
- Inputs are linearly combined into z using weights w_i and bias b

$$z = b + \sum_{i=1}^{N} w_i x_i = \sum_{i=0}^{N} w_i x_i$$

where we defined an additional input $x_0 = 1$ and $w_0 = b$

An Artificial Neuron II

- Linear combination z is then transformed using an activation function

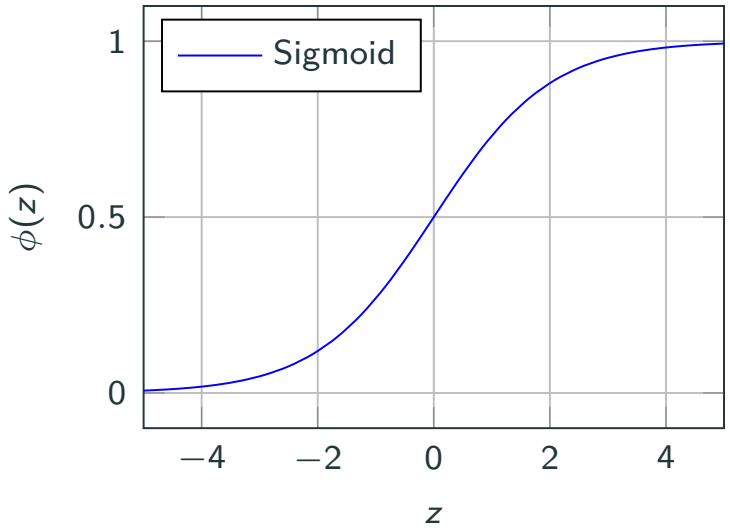
$$a = \phi(z) = \phi\left(\sum_{i=0}^{N} w_i x_i\right)$$

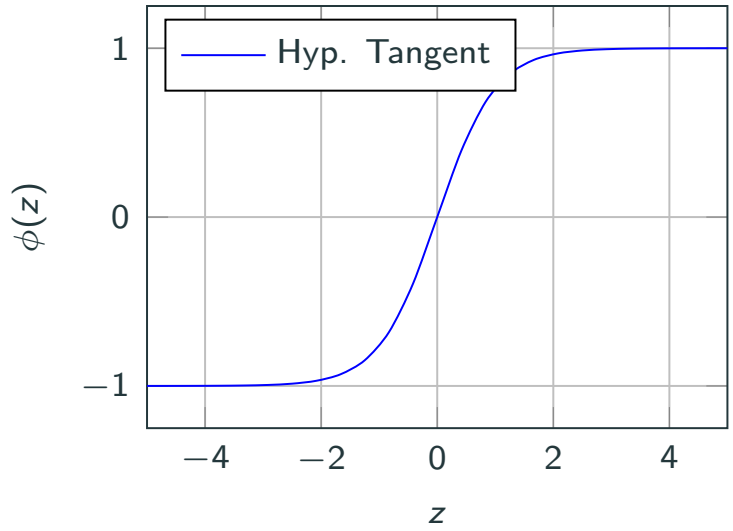
- Activation function introduces non-linearity into the NN and allows the NN to learn highly non-linear functions
- Particular choice of activation function depends on the application

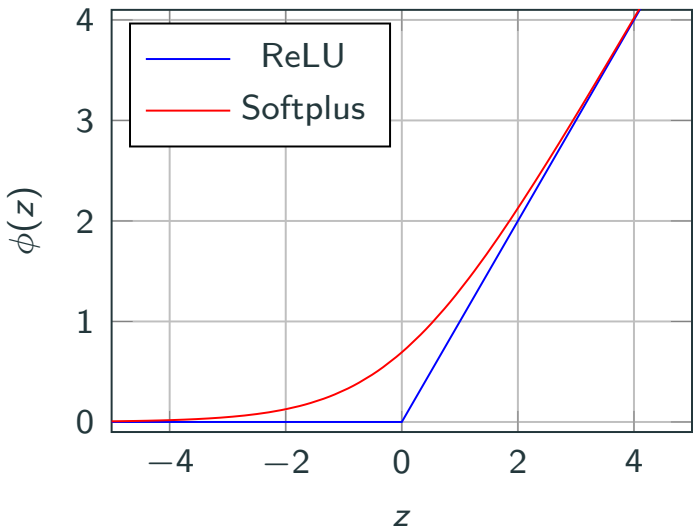
Activation Functions

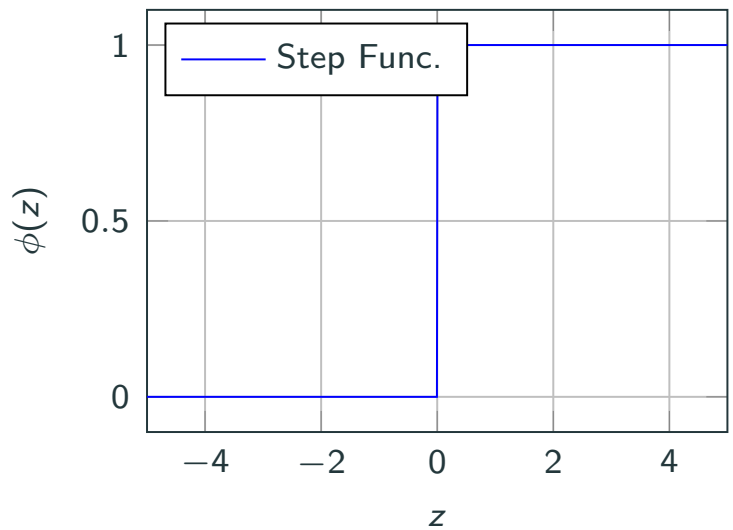
- Common activation functions include
  - Sigmoid: $\phi(z) = \frac{1}{1+e^{-z}}$
  - Hyperbolic tangent: $\phi(z) = tanh(z)$
  - Rectified linear unit (ReLU): $\phi(z) = \max(0, z)$
  - Softplus: $\phi(z) = \log(1 + e^z)$
- ReLU has become a popular in deep neural networks in recent years
- For approximations of smooth functions, softplus can be a good alternative to ReLU

Examples of Activation Functions









Building a Neural Network from Artificial Neurons

- A single-layer neural network is a linear combination of M artificial neurons

$$a_j = \phi(z_j) = \phi\left(b_j^1 + \sum_{i=1}^{N} w_{ji}^1 x_i\right)$$

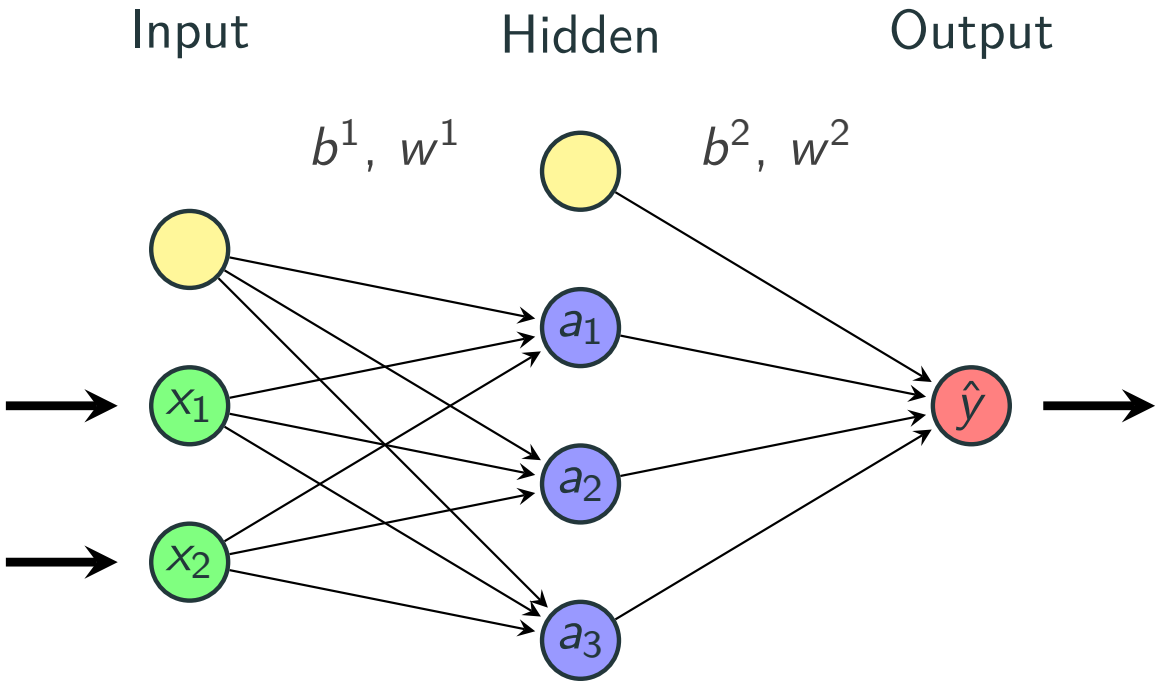
with the output defined as

$$g(x; w) = b^2 + \sum_{j=1}^{M} w_j^2 a_j$$

- This neural network has N inputs, a single hidden layer with M nodes/neurons, and a single output
- M is also called the width of the neural network

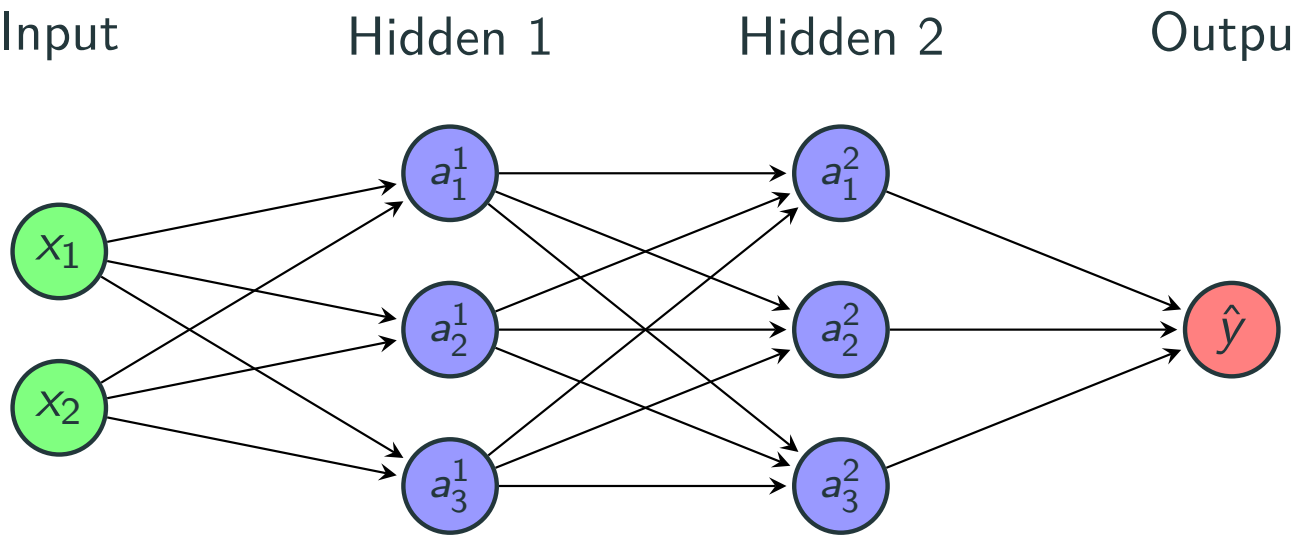
Relation to Linear Regression

A Graphical Representation of Neural Networks



- Common to represent NN as directed graphs (here $N = 2$ and $M = 3$)
- We will only consider neural networks
  - that are feedforward (i.e. their graphs are acyclical),
  - with dense layers (i.e. each layer is fully-connected to the previous), and
  - without connections that skip layers
- However, for most applications these restrictions are not very limiting

Deep Neural Networks



- By increasing the number of hidden layers you get deep neural networks
- The number of hidden layers is also referred to as the depth of the NN
- A deep neural network can learn potentially more complicated things
- For simple function approximation, a single hidden layer is sufficient

Deep Neural Network Example

Why Is This Useful?

- Suppose we want to approximate an unknown function

$$y = f(x)$$

  where $y = (y_1, y_2, \ldots, y_K)'$ and $x = (x_1, x_2, \ldots, x_N)'$ are vectors

- $f(x)$ could stand for many different functions in economics (e.g. a value function, a policy function, a conditional expectation, a classifier, ...)
- It turns out that neural networks are universal approximators and break the curse of dimensionality

Universal approximation theorem (Hornik, Stinchcombe, and White, 1989)

A neural network with at least one hidden layer can approximate any Borel measurable function mapping finite-dimensional spaces to any desired degree of accuracy.

Relation to Other Approximation Methods I

- Recall the expression for our basic neural network

$$y \cong g(x; w) = b^2 + \sum_{j=1}^{M} w_j^2 \phi \left(b_j^1 + \sum_{i=1}^{N} w_{ji}^1 x_i \right)$$

  where for notational simplicity we assumed that y is scalar

- This looks similar to a projection

$$y \cong \tilde{g}(x; w) = w_0 + \sum_{j=1}^{M} w_j \phi_j(x)$$

  where ϕ_j is, for example, a Chebyshev polynomial

Relation to Other Approximation Methods II

Breaking the curse of dimensionality (Barron, 1993)

A one-layer NN achieves integrated square errors of order $O(1/M)$, where M is the number of nodes. In comparison, for series approximations, the integrated square error is of order $O(1/(M^{2/N}))$ where N is the dimensions of the function to be approximated.

- Crucial difference between NN and traditional series approximations
  - In series approximation coefficients should increase exponentially with dimensions to preserve a given precision (this is not the case for NN)
  - NN require much more samples to train, given a number of parameters, than traditional series approximation
- Large data requirements may make it infeasible to properly train the NN in some applications (e.g. GDP forecasting)
- However, in some cases (e.g. macro-modelling) we can generate data

Examples of Recent Applications in Economics

- Applications in macro-modelling
  - Non-linear solution of heterogeneous agent models with agg. uncertainty

    Fernández-Villaverde, Hurtado, and Nuño (2020)
  - Non-linear solution of HANK model with ZLB and aggregate uncertainty

    Fernández-Villaverde, Marbet, Nuño, and Rachedi (2021)
  - Deep-learning to solve economic models

    Maliar, Maliar and Winant (2021)
- Applications in econometrics
  - Estimation of structural models with the help of neural networks

    Kaji, Pouliot, and Manresa (2020)
  - Deep-learning model to detect emotions in voices during press conferences after FOMC meetings

    Gorodnichenko, Pham, and Talavera (2021)

How to Train Your Neural Network

How to Train Your Neural Network

- We have not yet discussed how to determine the weights and biases
- The weights and biases w are selected to minimize a loss function

$$E(w; X, Y) = \frac{1}{N} \sum_{n=1}^{N} E_n(w; x_n, y_n)$$

where N refers to the number of input-output pairs that we use for training and $E_n(w; x_j, y_j)$ refers to the loss of an individual pair n

$$E_n(w; x_n, y_n) = \frac{1}{2} \left\| g\left(x_n; w\right) - y_n \right\|^2$$

- For notational simplicity, I will write $E(w)$ and $E_n(w)$ in the following or in some cases even omit argument w

Gradient Descent I

1. Initialize weights by drawing from a proposal distribution (e.g. Gaussian)

$$w^{(0)} \sim N(0, I)$$

2. Compute the gradient of the loss function with respect to the weights

$$\nabla E(w^{(i)}) = \frac{1}{N} \sum_{n=1}^{N} \nabla E_n \left(w^{(i)} \right)$$

3. Update the weights by making a small step into the direction of the negative gradient

$$w^{(i+1)} = w^{(i)} - \eta \nabla E \left(w^{(i)} \right)$$

   where $\eta > 0$ is the learning rate

4. Repeat step 2 and 3 until a terminal condition (e.g. fixed number of iterations) is reached

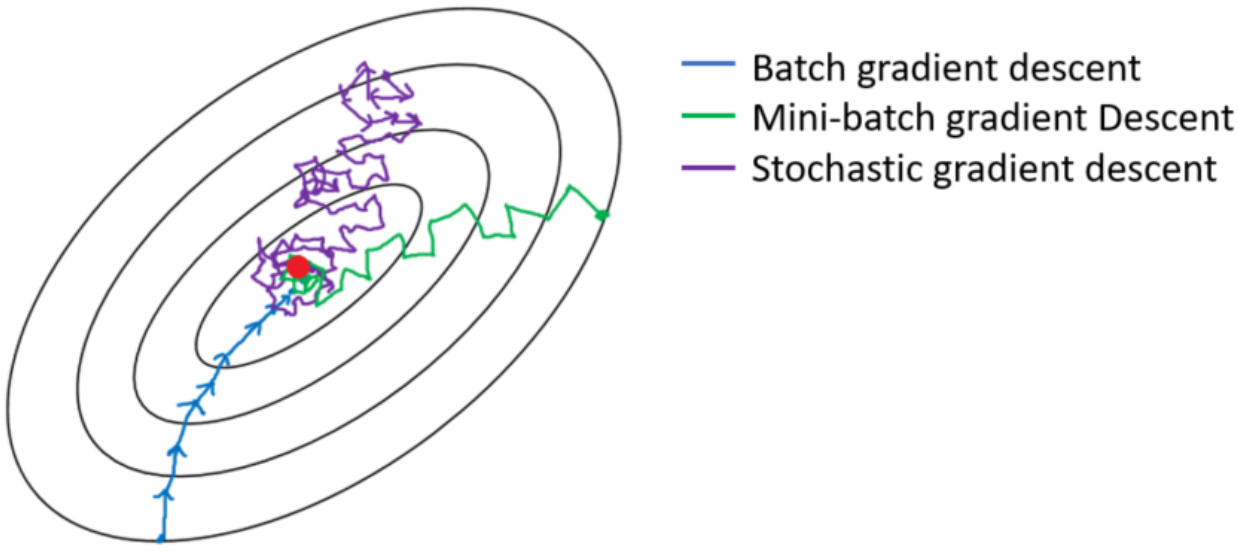
Gradient Descent II

- This algorithm is also called batch gradient descent
- In practice, slight variations have proven to be more reliable
  - Stochastic gradient descent: Use only a single observation to compute the gradient and update the weights for each observation

$$w^{(i+1)} = w^{(i)} - \eta \nabla E_n \left(w^{(i)} \right)$$

  - Minibatch gradient descent: Use a small batch of observations (e.g. 32) to compute the gradient and update the weights for each minibatch
- Less likely to get stuck in shallow local minimum of the loss function
- Currently, minibatch gradient descent is probably most commonly used

Gradient Descent III



Gradient Descent Comparison (Source: Analytics Vidhya, medium.com)

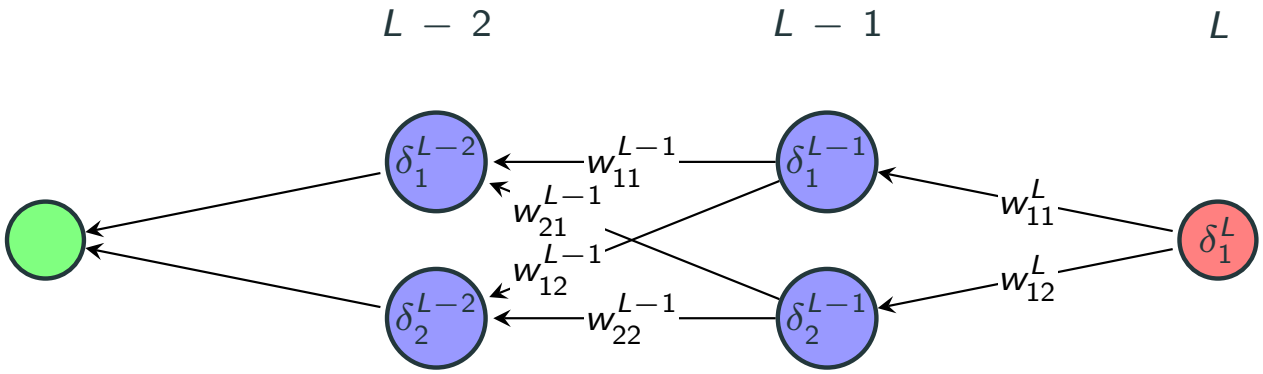
Backpropagation Algorithm I

- Gradient is evaluated using the backpropagation algorithm (Rumelhart et al., 1986)
- Recall how the layers in the neural network were built from neurons

$$a_j^l = \phi\left(z_j^l\right) \quad \text{with} \quad z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l \,,$$

  where w_{ji}^l is associated with the connection from neuron i in layer $l-1$ to neuron j in layer l
- Our goal is to compute $\frac{\partial E_n}{\partial w_{ji}^l}$ for all weights in our network
- We will compute these derivatives starting from the output layer (which has index L) backwards through the network

Backpropagation Algorithm II



- It is convenient to define

$$\delta_j^l = \frac{\partial E_n}{\partial z_j^l}$$

which we call the error associated with neuron j in layer l

- Given this definition, note that for $l < L$ we have

$$\delta_j^l = \frac{\partial E_n}{\partial z_j^l} = \sum_k \frac{\partial E_n}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \phi'(z_j^l) \sum_k w_{kj} \delta_k^{l+1}$$

which gives us an expression that relates the errors from layer $l+1$ with the error at layer l

Backpropagation Algorithm III

- With a single hidden layer, we would get the following
- We can compute for the output layer

$$\frac{\partial E_n}{\partial w_{ji}^L} = \frac{\partial E_n}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{ji}^L} = (a_j^L - y_{nj})a_i^{L-1} = \delta_j^L a_i^{L-1}$$

- For the hidden layer, we have

$$\frac{\partial E_n}{\partial w_{ji}^{L-1}} = \frac{\partial E_n}{\partial z_j^{L-1}}\frac{\partial z_j^{L-1}}{\partial w_{ji}^{L-1}} = \delta_j^{L-1} x_{ni}$$

where we can compute δ_j^{L-1} using the relation from the previous slide

Backpropagation Algorithm IV

- All of this can be more conveniently written in matrix notation

$$\begin{aligned}
\delta^L &= \nabla_a E_n = a^L - y_n \\
\delta^l &= \left(\left(w^{l+1} \right)' \delta^{l+1} \right) \odot \phi' \left(z^l \right) \\
\frac{\partial E_n}{\partial b^l} &= \delta^l \\
\frac{\partial E_n}{\partial w^l} &= \delta^l \left(a^{l-1} \right)'
\end{aligned}$$

  where $\delta^l = (\delta_1^l, \delta_2^l, \ldots)'$, $a^l = (a_1^l, a_2^l, \ldots)'$, $z^l = (z_1^l, z_2^l, \ldots)'$, $b^l = (b_1^l, b_2^l, \ldots)'$ and w^l is an appropriately defined matrix

- The gradient for E is then simply the sum of E_n over all observations

Practical Considerations

- From a practical perspective, there are many more things to consider
- Often times it's beneficial to do some (or all) of the following
  - Input/output normalization (e.g. to have unit variance and mean zero) can improve performance of the NN
  - Check for overfitting by splitting the dataset into a training dataset and a validation dataset
  - Use regularization to avoid overfitting (e.g. add term to loss function that penalizes large weights)
  - Adjust the learning rate during training

Implementing a Neural Network from Scratch

Implementing a Neural Network from Scratch

- We have now seen enough to implement our own neural network
- The main components will be presented in the slides
  1. Object that holds the main settings
  2. Function to evaluate the neural network (feedforward)
  3. Function to compute the gradient (using backpropagation)
  4. Function to train the network (using gradient descent)
- The remaining parts which "glue" the whole code together will be briefly discussed when we have a look at the code in VSCode

Defining Default Neural Network Settings

```julia
@with_kw mutable struct NeuralNetwork

    # Number of nodes of input, output and hidden layer
    nInputs::Int64 = 1
    nHidden::Int64 = 5
    nOutputs::Int64 = 1

    # Weights (Initialized by drawing from standard normal distribution)
    w1::Array{Float64,2} = randn(nHidden, nInputs)
    w2::Array{Float64,2} = randn(nOutputs, nHidden)

    # Biases (Initialized to zero)
    b1::Array{Float64,1} = zeros(nHidden)
    b2::Array{Float64,1} = zeros(nOutputs)

    # Regularization parameter
    λ::Float64 = 0.0

    # Learning rate during gradient descent
    learningSpeed::Float64 = 0.01

    # Activation function
    activationFunction::Symbol = :sigmoid # Supported: :sigmoid, :softplus, :relu

end
```

```julia
# To use this, you would type in REPL (or your function)
NN = NeuralNetwork()

# Or adjusting some of the default settings
NN = NeuralNetwork(nInputs = 4, λ = 0.001)
```

Evaluating the Neural Network

```julia
function feedforward(NN::NeuralNetwork, inputs)

    # NN contains all required information regarding weights, biases and activation functions

    # Compute the output of the hidden neurons
    hidden = activation.(Ref(NN), NN.w1 * inputs .+ NN.b1)  # Note: the dot-notation means that
                                                            # operations or functions are applied
                                                            # elementwise (this is also called
                                                            # broadcasting)

    # Compute the output of the neural network
    outputs = NN.w2 * hidden .+ NN.b2

    return outputs

end
```

```julia
# Usage example
NN = NeuralNetwork(nInputs = 2) # Initialize NN with 2 inputs
                                # (and default setting for the remaining settings)
inputs = [1.3, 2.4] # Vector of inputs
output = feedforward(NN, inputs) # Vector of outputs
```

activation(NN, x)

Defining a Struct for the Gradient

```
struct NeuralNetworkGradient
    w1::Array{Float64,2}
    w2::Array{Float64,2}
    b1::Array{Float64,1}
    b2::Array{Float64,1}
end

function NeuralNetworkGradient(NN::NeuralNetwork)

    NeuralNetworkGradient(
        zeros(size(NN.w1)),
        zeros(size(NN.w2)),
        zeros(size(NN.b1)),
        zeros(size(NN.b2))
    )

end
```

```
# Usage example
NN = NeuralNetwork()
NNGradient = NeuralNetworkGradient(NN)  # Automatically, creates matrices for the gradient
                                        # in the correct dimensions

# Note the gradient is allocated in advance to reduce matrix allocations during training
```

Computing the Gradient (for One Observation)

```julia
function computeGradientSimplified!(NN::NeuralNetwork, NNGradient, inputs, outputs)

    # NNGradient is a struct of matrices holding the gradients for weights and biases

    # Feed forward
    z_Lm1 = NN.w1 * inputs .+ NN.b1          # Summation hidden layer
    a_Lm1 = activation.(Ref(NN), z_Lm1)      # Activation hidden layer
    z_L = NN.w2 * a_Lm1 .+ NN.b2             # Summation output layer
    a_L = z_L                                # Activation output layer (linear)

    # Backpropagation
    δ_L = (a_L .- outputs)
    δ_Lm1 = (NN.w2' * δ_L) .* activationPrime.(Ref(NN), z_Lm1)

    # Compute gradient
    NNGradient.w2 .= δ_L * a_Lm1'
    NNGradient.w1 .= δ_Lm1 * inputs'
    NNGradient.b2 .= δ_L
    NNGradient.b1 .= δ_Lm1

    # Add regularization term
    @. NNGradient.w1 = 2 * (NNGradient.w1 + NN.λ * NN.w1)
    @. NNGradient.w2 = 2 * (NNGradient.w2 + NN.λ * NN.w2)
    @. NNGradient.b1 = 2 * NNGradient.b1
    @. NNGradient.b2 = 2 * NNGradient.b2
    # Note @. applies a dot to each operation and function call on a line
    # (i.e. makes everything into elementwise operations)

    nothing

end
```

Training the Neural Network

```julia
function trainNeuralNetwork!(NN::NeuralNetwork, NNGradient, inputs, outputs)

    for ii in 1:length(inputs)

        # Compute the gradient
        computeGradient!(NN, NNGradient, inputs[ii], outputs[ii]) # More efficient but harder to understand
        # computeGradientSimplified!(NN, NNGradient, inputs[ii], outputs[ii]) # Slower but easier to understand

        # Determine learning rate (more advanced techniques possible)
        learn = NN.learningSpeed

        # Update the weights of the neural network
        @. NN.w1 = NN.w1 - learn * NNGradient.w1
        @. NN.w2 = NN.w2 - learn * NNGradient.w2
        @. NN.b1 = NN.b1 - learn * NNGradient.b1
        @. NN.b2 = NN.b2 - learn * NNGradient.b2

    end

    nothing

end
```

```julia
# The function trains the network for one epoch (i.e. it goes over the data set once)
# To train for multiple epochs, we would use
for ii in 1:epochs
    trainNeuralNetwork!(NN, NNGradient, inputs, outputs)
end
```

That's it! Almost...

```
# Initializing the neural network
NN = NeuralNetwork()
NNGradient = NeuralNetworkGradient(NN)

# Training
epochs = 1000
for ii in 1:epochs
    trainNeuralNetwork!(NN, NNGradient, inputs, outputs)
end

# Evaluating
prediction = feedforward(NN, [1.0])
```

- For some `inputs` and `outputs`, the code above (together with the functions from before) is all you need to train and evaluate your network
- However, the code that I provide does some additional things
  - It normalizes inputs and outputs, and
  - It plots the some results
- Let's have a look...

Machine Learning Libraries

Machine Learning Libraries

- Many good, free libraries available
- Julia Packages
  - Flux.jl: Machine learning library fully written in Julia
- Python Packages
  - TensorFlow: Machine learning library developed by Google
  - PyTorch: Another popular machine learning library
  - Keras: Programming interface for several deep learning libraries (incl. TensorFlow)
- Matlab also has a Deep Learning Toolbox (but it's not free)
- There is also a Tensorflow wrapper for Julia. However, it is not actively updated anymore

A Basic Example in Flux.jl

```julia
using Flux

# Define the neural network
model = Chain(
    Dense(1, 5, σ), # Input-Hidden (sigmoid activation)
    Dense(5, 1)     # Hidden-Output (linear activation)
    )

# Define loss function and weights
loss(x, y) = Flux.Losses.mse(model(x), y)
ps = Flux.params(model)

# Train the neural network
epochs = 1000
opt = Descent(0.01) # learning rate

for ii in 1:epochs
    Flux.train!(loss, ps, zip(inputs, outputs), opt)
end

# Evaluate
prediction = model([1.0])
```

- You need to have some training data with inputs/outputs
- A full example to approximate a function is provided in the code folder

A Basic Example in TensorFlow/Keras (written in Python)

```
1  from tensorflow.keras.models import Sequential
2  from tensorflow.keras.layers import Dense
3  from tensorflow.keras import optimizers
4
5  # Define the neural network
6  model = Sequential()
7  model.add(Dense(5, input_dim = 1, activation = 'sigmoid'))
8  model.add(Dense(1))
9
10 # Train the neural network
11 model.compile(loss = 'mean_squared_error', optimizer = optimizers.SGD(learning_rate = 0.01))
12 history = model.fit(inputs, outputs, epochs = 1000, batch_size = 1, verbose = 0)
13
14 # Evaluate
15 prediction = model.predict([1.0])
```

- Again, you need to have some training data with inputs/outputs
- A full example to approximate a function is provided in the code folder

Which One Should You Use?

- Mainly a matter of personal preference and your own needs
- Currently, I use my own implementation for the following reasons
  1. Understanding: Get a better understanding of what is going on
  2. Simple NN: Up until now I did not need deep neural networks
  3. Speed: It is faster since it uses analytical derivatives (this might become less relevant if you train deep neural networks on GPU and/or computation is parallelized)
  4. Integration with Julia: Only need a single language to solve my model
- However, it will quickly get complicated to extend the neural network
- Therefore, I highly suggest you familiarize yourself with one of the machine learning libraries

Resources for Further Learning

Resources for Further Learning

- Useful references for neural networks and machine learning
  - Goodfellow et al. (2016), "Deep Learning" (`https://www.deeplearningbook.org`)
  - Bishop (2006), "Pattern Recognition And Machine Learning"
  - Nielsen (2019), "Neural Networks and Deep Learning" (`http://neuralnetworksanddeeplearning.com/`)
- Some useful references for Julia
  - TechyTok! (Good tutorial): `https://techytok.com/from-zero-to-julia/`
  - QuantEcon (Another good tutorial with economic applications): `https://julia.quantecon.org/`
  - Julia Documentation: `https://docs.julialang.org/`
  - Flux.jl: `https://fluxml.ai/Flux.jl`

Thank you!

Appendix: Perceptron

- Perceptrons were developed in the 1950s and consist of only one neuron
- They use a step function as activation

$$\phi(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise}\,, \end{cases}$$

- Perceptrons can be used for basic classification
- Note that the step function is usually not used in neural networks
  - Derivative is not defined at $z = 0$ and zero everywhere else
  - Thus, it cannot be used for the back-propagation algorithm, which is used for determining the network weights

Back

Appendix: Relation to Linear Regression

- Suppose we use a linear activation function, e.g. $\phi(x) = x$
- Then, the neural network collapses to a linear regression

$$y \cong g(x; w) = \tilde{w}_0 + \sum_{i=1}^{N} \tilde{w}_i x_i$$

  with appropriately defined regression coefficients $\tilde{w}$
- NN is a linear combination of M generalized linear models of x

Back

Appendix: Deep Neural Network Example

- The first hidden layer consists of M_1 artificial neurons with inputs $x_1, x_2, \ldots, x_N$

$$a_j^1 = \phi\left(b_j^1 + \sum_{i=1}^{N} w_{ji}^1 x_i\right)$$

- The second hidden layer consists of M_2 artificial neurons with inputs $a_1^1, a_2^1, \ldots, a_{M_1}^1$

$$a_k^2 = \phi\left(b_k^2 + \sum_{j=1}^{M_1} w_{kj}^2 a_j^1\right)$$

- After Q hidden layers the output defined as

$$y \cong g(x; w) = b^{Q+1} + \sum_{j=1}^{M_Q} w_j^{Q+1} a_j^Q$$

- Note that activation functions do not need to be the same everywhere

Back

Appendix: Activation Function and its Derivative

```
function activation(NN::NeuralNetwork, x)

    if NN.activationFunction == :softplus
        return log(1+exp(x))
    elseif NN.activationFunction == :relu
        return (x < 0.0) ? 0.0 : x
    elseif NN.activationFunction == :sigmoid
        return 1/(1+exp(-x))
    end

end
```

```
function activationPrime(NN::NeuralNetwork, x)

    if NN.activationFunction == :softplus
        return 1/(1+exp(-x))
    elseif NN.activationFunction == :relu
        return (x < 0.0) ? 0.0 : 1
    elseif NN.activationFunction == :sigmoid
        return (1/(1+exp(-x))) * (1 - 1/(1+exp(-x)))
    end

end
```

Back

Appendix: Julia Installation I

- You can install Julia (`https://julialang.org/downloads/`) and use any text editor you like
- However, VSCode with the Julia extension offers many IDE features that improve the usability a lot. For example, with VSCode you get
  - a debugger,
  - a profiler,
  - a list of variables in the workspace (similar to Matlab),
  - git-repository integration,
  - easy access to the Julia documentation,
  - many more features that can be added using VSCode extensions

Back

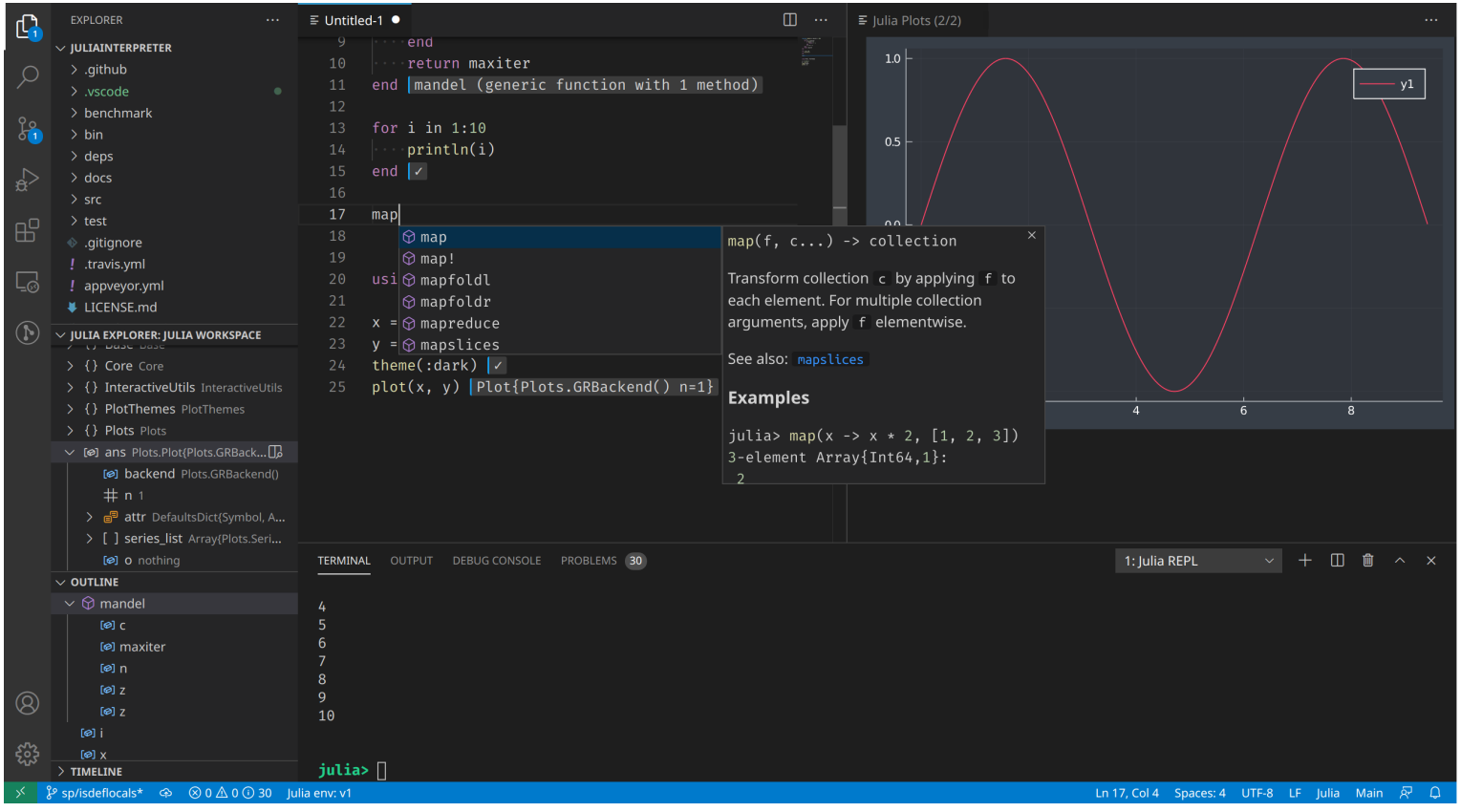
Appendix: Julia Installation II

- Installation of the recommended setup from scratch
  1. Install Julia: `https://julialang.org/downloads/`
  2. Install VSCode: `https://code.visualstudio.com`
  3. Install Julia for VSCode: Go to View in VSCode, then click on "Extensions" and type "julia" in the search box and hit enter. Install the `julia` extension.
  4. Required Julia packages can then be installed by running `RequiredPackages.jl`

Back

Appendix: Julia Installation III

Once everything is installed VSCode should look similar to this



Back