Chapter 10

> Introduction to Artificial Neural Networks with Keras

👉 GitHub Page: LINK
Google Drive: LINK

- Lot of human inventions came from nature
  - Planes from birds
  - Velcro from burdock plants
  - Machine Learning from brains
- Deep Learning is seen in
  - Google images
  - Siri
  - AlphaGo
  - YouTube Recommendation

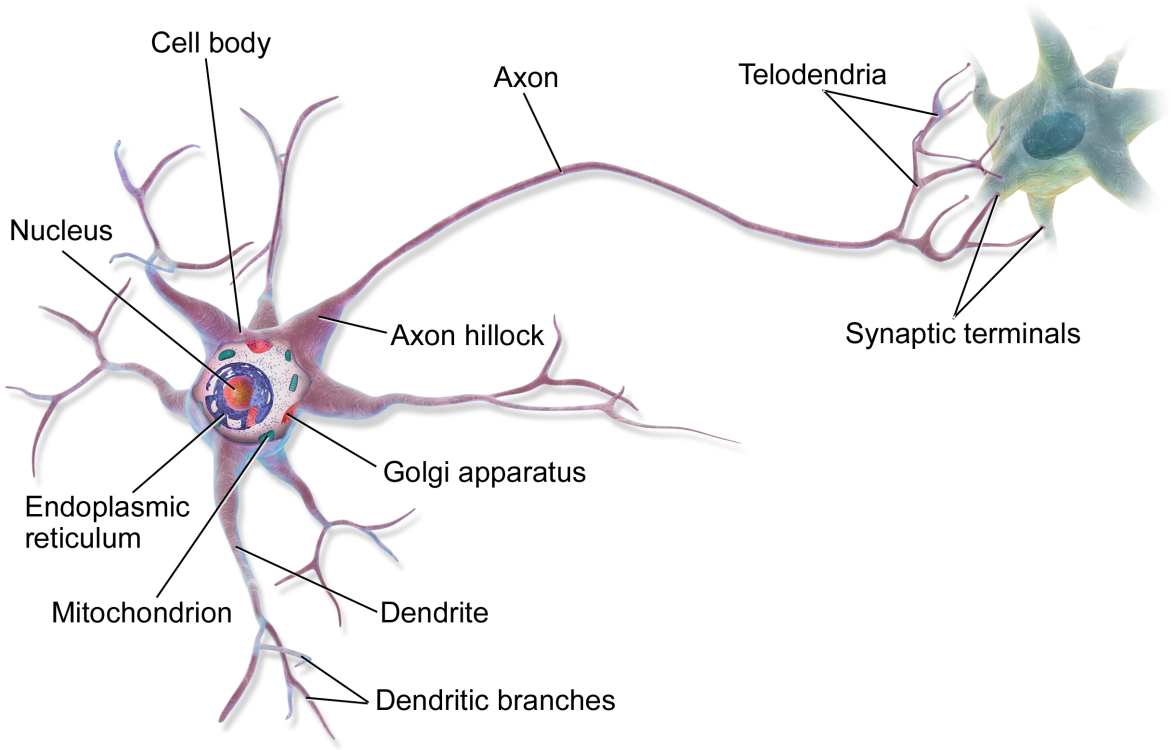
From Biological to Artificial Neurons

- Idea was first introduced in 1943
- ANNs usually outperform other ML techniques on large and complex problems
- Due to the increase of computing power since 1990s, neural network training takes less time
  - Due to Moore's Law
- Theoretical limitations of ANNs have turned out to be benign in practice
  - Likely to stuck in local optima but turns out to be very rare

- ANNs have been funded based on it's regularly news in headlines which brings attention and funding

Biological Neurons

- Usual looking cell found in animal brains



- composed of a body cell containing the nucleus and most of the cell's complex components
- branches are called dendrite
- long extended branches are called Axon
- Near the end of the Axon is called telodendria
    - On the top of the branches are minuscule structures called synaptic terminals which are connected to the dendrites or cell bodies of other neurons
- Biological neurons produce impulses called action potentials which sends electrical signals around
- Thus individual biological neurons seem to behave in a rather simple way
- but they are organized in a vast network of billions

- one neurons typically connect to thousands of neurons

Logical computations with Neurons

- A simple artificial neurons has a binary input and one binary output
- artificial neurons activates its output when more than a certain number of its inputs are activated

The Perceptron

- The perceptron is one of the simplest ANN architectures
- It is based on a threshold logical unit (TLU) or linear threshold unit (LTU)
- each the inputs and output are numbers instead of binary
- each input connection is associated with a weight
- TLU computes a weighted sum of its inputs
- Then applies a step function to that sum and outputs the result
- A single TLU can be used for simple linear binary classification
  - If the result exceeds a threshold then it output positive
  - If the result is below a threshold then it output negative
- A perceptron is a single layer of TLUs, with each of the TLU connected to all the inputs
- each layer of neurons connected to each other is called fully connected layers or a Dense layer
- the input of the perceptron is called input layer
- A extra bias feature is added called bias neuron which outputs 1 all the time

> 👉 Equation and Graph refer back to book page 286

- The networks trains by having the fed one training instance at a time and for each instance it makes its predictions
- For every output neurons that is wrong it reinforces the connection weights from inputs that would contributed to the correct prediction

$$w_{i,j}^{nextstep} = w_{i,j} + n(y_j - \hat{y}_j)x_i$$

- $w_{i,j}$ is the connection weights between the i^{th} input neuron and the j^{th} output neuron
- x_i is the i^{th} input value of the current training instance
- $\hat{y}_j$ is the output of the j^{th} output neuron for the current training instance
- n is the learning rate

👉 Scikit-Learn code refer back to page 287

The Multilayer Perceptron and Backpropagation

- MLP is composed of one input layer
- One or more of TLUs is called hidden layer
- And the final layer of TLUs are called the output layer
- The one close to output are usually called upper layer
- Every layer except the output layer includes a bias neuron and is fully connected to the next layer

👉 Refer back to page 289 for diagram

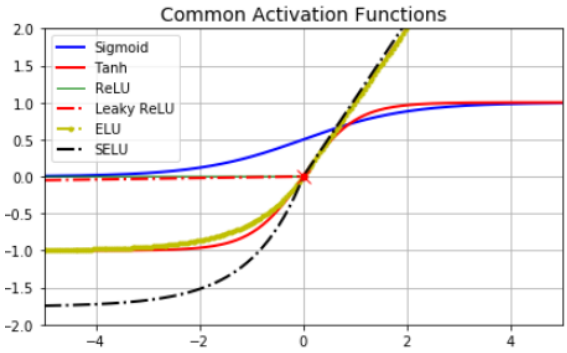
- When ANN contains a deep stack of hidden layers, it's called deep neural network (DNNs)
- Training MLNs were difficult until the backpropagation training algorithm was discovered by David Rumelhart
- The backpropagation algorithm was able to compute the gradient of the network's error with regards to every single parameter
- it can find out how each connection weight and each bias terms, should be tweaked in order to reduce the error
- Details for backpropagation
    - it handles mini-batches at a time and goes through training multiple times, each called an epoch

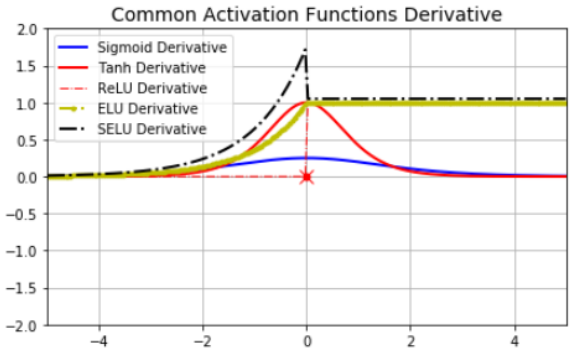
    - the results are passed forward from layer to layer, this is called forward pass
    - the algorithm measure the network's output error with loss function
    - then it computes how much each output connection contributed to the error with chain rule
    - The algorithm measure how much these error contribution came from each connection in the layer then it works back to the input layer hence the name backpropagation
    - The algorithm performs a gradient descent step to tweak the connection weights in the network using the error gradients it just computed
- A shorter version of the summary:
    - for each of the training instances
        - makes a prediction and measure the error
        - goes through each layer in reverse to measure the error contribution from each connection
        - tweaks the connection weights to reduce the error
- It's important to initialize all the hidden layer weights randomly since if it's all zeros then all the layers would be the same
- In order for this to work properly, the authors changed MLP's architecture

$$o(z) = 1/(1 + exp(-z))$$

- this was essential because the step function contains only flat segments, so there is no gradient to work with
- backpropagation algorithm works well with not only logistic activation
    - hyperbolic tangent function: $tanh(z) = 2o(2z) - 1$
        - Just like logistic function this activation function is:
            - S shaped
            - Continuous
            - differentiable

    - output ranges from -1 to 1
    - centered around 0 at the start of the training to speed up convergence
- Rectified Linear Unit function: $ReLU(z) = max(0, z)$
    - continuous but not differentiable at z=0
    - derivative is 0 fir z < 0
    - advantage of fast to compute so it has become the default
    - Does not have maximum output value





- Why do we need activation functions?
    - The reason is if you chain linear transformation, all you get is linear transformation
    - If you don't have nonlinearity between layers then a deep stack of layers is equivalent to a single layer and you cannot solve complex problems like that

Regression MLPs

- MLPs can be used for regression tasks
- if you want to predict a single value, then you just need a single output neuron
- you need one more output neuron per output dimension
    - predicting images need 2 neurons because the image is 2d
    - Then you need width and the height of the object
    - So in the end you would need 4 neurons

- In general building MLP regression, you don't want to use any activation functions
- they are free range values
- To guarantee that the output will always be positive, you can use ReLU on the output layer
- or alternatively you can use the softplus activation function which is a smooth variant of ReLU
- If you want the output to stay in a range of values , then use logistic function or hyperbolic tangent
- loss function during training is typically mean squared error

Classification MLPs

- MLP can also be used to classification tasks
  - For binary classification problems you need a single output neuron using the logistic activation function
- MLPs can also handle multilabel binary classification tasks
  - It could predict weather the incoming email is spam or not
- If there's multiple classes as output
  - Then you should you softmax activation function
  - This is called multilayer classification

Implementing MLPs with Keras

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/5ca60eb9-f7f5-45ad-9e72-ead980433174/fashion_mnist.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/96160184-e937-4b80-95d9-b528ee816226/fashion_mnist.py

Building Regression MLP Using the Sequential API

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ad62705e-1c3a-4a8c-b478-c48fd6433653/regression_MLP.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/20368676-732c-4287-8ff2-4d3fa7035876/regression_mlp.py

- Using Scikit-Learn fetch_California_housing

Building Complex Models Using the Functional API

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e3755339-3242-4b1b-978b-c9f4b53330aa/regression_MLP.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/0ef94b30-6769-451e-aa1d-fbedcca898f7/regression_mlp.py

Using the Sub-classing API to Build Dynamic Models

- Both sequential API and the functional API are declarative
  - You start by declaring which layers you want to use and how they should be connected
  - And then you start feeding the model

- Simply subclass the Model class, create the layer you need in the constructer and use them to perform the computations you want in call

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/95f4cd80-23af-4503-b4e3-9f37e4c279ae/subclassing_API.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a63e5afc-4d4e-4c58-ac4a-657a2a44a285/subclassing_api.py

Saving and Restoring a Model

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/000d0148-9c24-437f-b863-bafe09f29df0/save_and_restore.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/9a936c4a-767f-4fc9-b561-db0b093832d2/save_and_restore.py

Using Callbacks

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/1910027f-2d70-4ce5-97f6-aa50fee68683/callbacks.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/e4b92c79-5661-419d-bf3d-e93bb3cbf807/callbacks.py

Using TensorBoard for Visualization

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/001ab9a2-0e97-4d33-82dc-49bd8fbfb3f5/tensorboard.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2e887dd5-d649-4779-badd-5df304e4f374/tensorboard.ipynb

Fine-Tuning Neural Network Hyperparameters

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/1c6016a4-ffde-4042-a8a5-43ff559839e1/hyperparameter.py

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/d81f6324-c071-4a44-967f-62e4a14fb5f1/hyperparameter.ipynb

- The last step of finding the right hyperparameter may take hours depending on the hardware
- It will produce:
  - Optimal learning rate
  - optimal hidden layers and number of neurons in it
  - best score
- Here are some python libraries to optimize hyperparameters
  - Hyperopt
  - Hyperopt, kopt, or Talos

- Keras Tuner
- Sikit-Optimizer (skopt)
- Spearmint
- Hyperband
- Sklearn-Deap

👉 For more details visit Page 323

Number of Hidden Layers

- Can begin with one single hidden layer and get reasonable results
  - one layer could theoretically model even the most complex functions (provided enough neurons)
- many problems can start with one or two hidden layers and the network would be fine
  - Could achieve around 97 % accuracy with one layer using mnist
  - Could achieve around 98 % accuracy with two layers using mnist
- for more complex problems could start ramping up the layer until it overfits

Numbers of Neurons per Hidden Layer

- Number of neurons for input and output is determined by:
  - for mnist library:
    - 28*28=784 therefore it requires 784 input neurons
    - 10 output neurons
- For hidden layers it's common to size them to form a pyramid with fewer and fewer neurons at each layer
- Could start increasing the neurons gradually until it overfits

Learning Rate, Batch Size, and Other Hyperparameters

- **Learning Rate**
  - arguably the most important hyperparameter
  - The optimal learning rate is about half of the maximum learning rate
  - To find the optimal learning rate
    - start with a very low learning rate like: 10^{-5}
    - And gradually increase it up to a very large value like 10
  - If you plot the loss as a function of the learning rate using a log scale for learning rate
    - you should see it dropping at first but after a while the learning rate will be too large so the loss shoots back up
  - The optimal learning rate will be a bit lower than the point at which the loss starts to climb
  - typically around 10 times larger than turning point
- **Optimizer**
  - Choosing a better optimizer than plain old Mini-batch Gradient Descent is also quite important (See more in chapter 11)
- **Batch Size**
  - significant impact on the model's performance
  - The point of using batches is so GPUs can process them efficiently
  - Some large batch size cannot fit inside the VRAM
  - Larger batch size might lead to instabilities especially in the start of the training and resulting model may not generalize as well
  - Try and use batch size between 2-32
  - However 8192 batches was showed possible
  - If the training is unstable try and reduce the batch size
- **Activation function**

- RELU is generally good for hidden layers
- output layers depends on the task

- **Number of iterations**
  - generally the number of iteration does not need to be tweaked just stop early instead

Exercises

1. TensorFlow PlayGround

   a. Finds a good solution quickly. The more layers there is the more complex the image could be. (Image A)

   b. It finds the solution faster but the boundaries are linear due to the structure of ReLU

   c. Training varies a lot and sometimes gets stuck in local minima

   d. With Only 2 neurons it underfits

   e. It Never gets stuck on local minima and it's very consistent because of the size of the network

   f. Vanishing Gradient Issue

   g. Tested with all types of data shapes

Image A

Image B

Image C

Image D

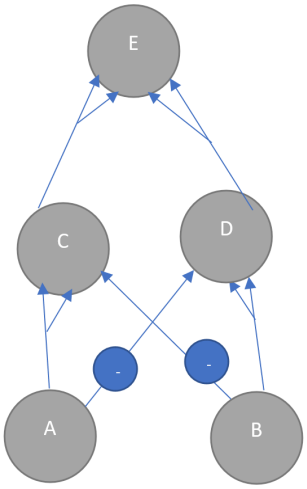
Image E

Image F

Best Result with spiral Dataset

Other Shaped data

2.



3. Logistic regression is preferable because it is able to classify that's not linear, classical perceptron is only able to perform simple linear tasks. If the activation function is changed to logistic or softmax for multiple neurons and use gradient decent to train the model then it will be equivalent to logistic regression

4. Logistic activation is a key ingredient because it's derivative is always non-zero so gradient descent can always go down a slope

5. ReLU, Tahn, Sigmoid

- ReLU is flat from until zero and shoots up
- Tahn has a value between -1 to 1 and it's non linear
- Sigmoid has a value between 0-1 and it's non linear

6.

- The Input shape of matrix X is 10 * m (batch size)
- Shape of hidden layer W_h is 10*50 and bias vector for b_h is 50
- Shape of output layer's weight vector is W_o is 3*50 it's bias vector of b_o is 3
- The shape of the network's matrix Y is 3 * m (batch size)

- $Y^* = ReLU(ReLU(XW_h + b_h)W_o + b_o)$

7. You would need two neurons as output layer if you want to classify email spams, the activation that should be used is sigmoid. If it is tackling MNIST, it requires 10 output neuron and the activation function should be softmax. For house prediction you would need one output neuron and no activation function

8. Backpropagation is a technique for machine learning. It calculates the loss and it goes backwards from the output layer and change the weight and bias for each layer to achieve a certain output value. Reverse-mode autodiff is just a technique to compute gradient efficiently but backpropagation refers to the entire process of training a network.

9. Hyperparameters you can tune are:

- Learning rate
- Optimizer
- Batch size
- Activation function
- epochs
- Number of layers
- Number of neurons in a layer

If the model overfits, try and reduce the number of hidden layers and number of neurons in the layers.

10.

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/918738f2-ed9f-40a3-a0a5-51dc7b157b29/mnist.ipynb

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a351a988-95bc-4e1f-8d6c-dab719dae30f/mnist.py