Understanding Basics of Deep Learning by solving XOR problem

Brief history of AI

Lalit Pal Follow

Apr 14 · 12 min read

Artificial Intelligence aims to mimic human intelligence using various mathematical and logical tools. Initial AI systems were rule based systems. These system were able to learn formal mathematical rules to solve problem and were deemed intelligent systems. But these system were not performing well in solving problems which doesn't have formal rules and as humans we were able to tackle them with ease e.g. identifying objects, understanding spoken words etc. To solve this problem, active research started in mimicking human mind and in 1958 once such popular learning network called "Perceptron" was proposed by Frank Rosenblatt. Perceptrons got a lot of attention at that time and later on many variations and extensions of perceptrons appeared with time. But, not everyone believed in the potential of Perceptrons, there were people who believed that true AI is rule based and perceptron is not a rule based. Minsky and Papert did an analysis of Perceptron and conluded that perceptrons only separated linearly separable classes. Their paper gave birth to the Exclusive-OR(X-OR) problem.

X-OR problem

XOR problem is a classical problem in the domain of AI which was one of the reason for winter of AI during 70s. To understand it, we must understand how Perceptron works. Perceptron is based on the simplification on neuron architechture as proposed by McCulloch–Pitts, termed as McCulloch–Pitts neuron. Not going into much details, here we will discuss the neuron function in simpler language. It has two inputs and one output and the neuron has a predefined threshold, if the sum of inputs exceed threshold then output is active else it is inactive[Ref. Image 1]

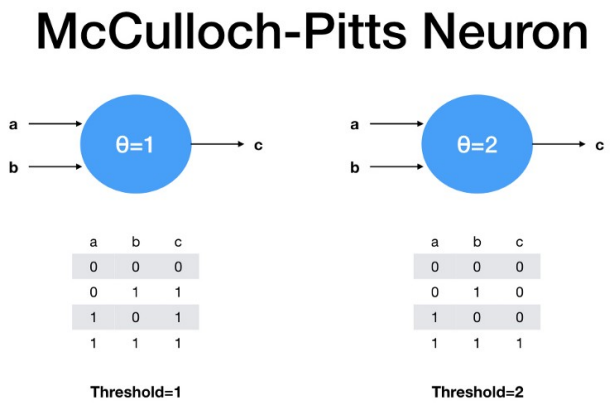


Image 1: McCulloch–Pitts Neurorn

Minsky and Papert used this simplification of Perceptron to prove that it is incapable of learning very simple functions. Learning by perceptron in a 2-D space is shown in image 2. They chose Exclusive-OR as one of the example and proved that Perceptron doesn't have ability to learn X-OR. As described in image 3, X-OR is not separable in 2-D. So, perceptron can't propose a separating plane to correctly classify the input points. This incapability of perceptron to not been able to handle X-OR along with some other factors led to an AI winter in which less work was done in neural networks. Later many approaches appeared which are extension of basic perceptron and are capable of solving X-OR.

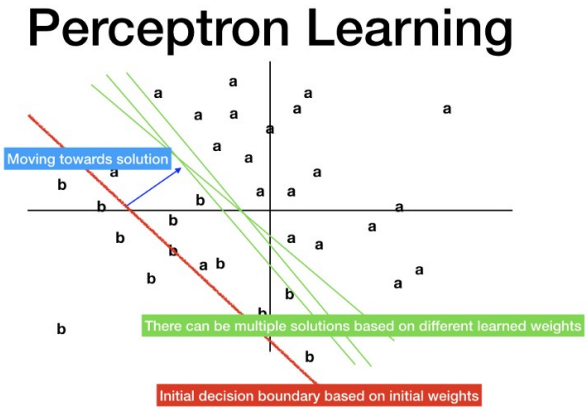


Image 2: Perceptron Learning

Image 3: Inclusive-OR vs Exclusive-OR

Brief introduction of Deep Learning

Deep Learning is one such extension of basic Perceptron model, in which we create stack of neurons and arrange them in multiple layers.Initial models with single hidden layers were termed multi layer perceptrons and are considered shallow networks. Deep networks have multiple layers and in recent works have shown capability to efficiently solve problems like object identification, speech recognition, language translation and many more. While neural networks were inspired by human mind, the Goal in Deep Learning is not to copy human mind, but to use mathematical tools to create models which perform well in solving problems like image recognition, speech/dialogue, language translation, art generation etc.

Solving X-OR with multi layer perceptron[MLP] in Keras

Multi layer perceptron are the networks having stack of neurons and multiple layers. A basic neuron in modern architectures looks like image 4:

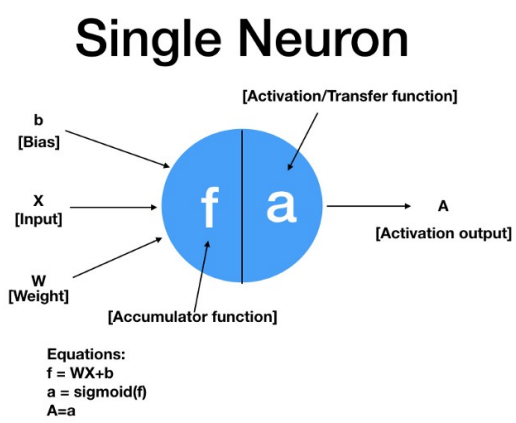


Image 4: Single Neuron

Each neuron is fed with an input along with associated weight and bias. A neuron has two functions:

1) Accumulator function: It essentially is the weighted sum of input along with a bias added to it.
2) Activation function: Activation functions are non-linear function. And as the name suggests is a function to decide whether output of a node will be actively participating in the overall output of the model or not. ReLu is the most popular activation function used now a days.

Now i will describe a process of solving X-OR with the help of MLP with one hidden layer. So, our model will have an input layer, one hidden layer and an output layer. Out model will look something like image 5:

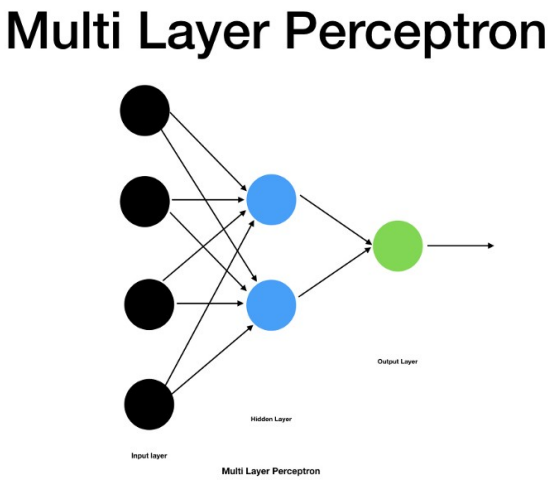


Image 5: Multi Layer Perceptron with one hidden layer

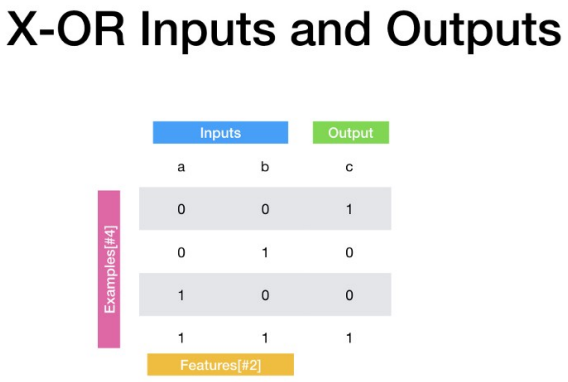
Input Data

As explained earlier, Deep learning models use mathematical tools to process input data. So, we need are input layer to represent data in form of numbers. In many applications we get data in other forms like input images, strings etc. We need to find methods to represent them as numbers e.g. for images we can use RGB values of each pixel of image, for text strings we can map each word to a predefined dictionary. In the input data we need to focus on two major aspects:

1. Number of features: Input given to a learning model may have only single feature which impacts the output e.g. we are given a collection of green and red balls and we want our model to segregate them input separate classes. Here, we need only one feature for this task i.e. color of the ball. But, in most cases output depends on multiple features of input e.g. face recognition or object identification in a color image considers RGB values associated with each pixel.
2. Number of examples: For each problem we will have to feed our network multiple input examples so that it can generalize over problem space. e.g. if we wish to develop a model which identifies cats, we would require thousands of cat images in different environments, postures, images of different cat breeds.

The input is arranged as a matrix where rows represent examples and column represent features. So, if we have say **m** examples and **n** features then we will have an **m x n** matrix as input.

In our X-OR example, we have four examples and two features so our input is a **4 x 2** matrix[Ref. image 6].



In Keras we defines our input and expected output with following lines of code:

x = np.array([[0,0],[0,1],[1,0],[1,1]])

y = np.array([1,0,0,1])

Output

Based on the problem at hand we expect different kinds of output e.g. for cat recognition task we expect system to output Yes or No[1 or 0] for cat or not cat respectively. Such problems are said to be two class classification problem. We have only a single output for one example. Then we can have multi class classification problems, in which input is a distribution over multiple classes e.g. say we have balls of 4 different colors and model is supposed to put a new ball given as input into one of the 4 classes. Some advanced tasks like language translation, text summary generation have complex output space which we will not consider in this article. The activation function in output layer is selected based on the output space. For a binary classification task sigmoid activations is correct choice while for multi class classification softmax is the most populary choice.

In our X-OR problem, output is either 0 or 1 for each input sample. So, it is a two class or binary classification problem. We will use binary cross entropy along with sigmoid activation function at output layer. [Ref image 6].

In Keras we defines our output layer as follows:

model.add(Dense(units=1,activation="sigmoid"))

Hidden units:

A deep learning network can have multiple hidden units. The purpose of hidden units is the learn some hidden feature or representation of input data which eventually helps in solving the problem at hand. For example, in case of cat recognition hidden layers may first find the edges, second hidden layer may identify body parts and then third hidden layer may make prediction whether it is a cat or not.

As explained, we are using MLP with only one hidden layer. It is a shallow network and our expectation is that he hidden layer will transform the input of X-OR from a 2-D plane to another form where we can find a separating plane matching our expectation for X-OR output. We will use ReLu activation function in our hidden layer to transform the input data. One such transformation is as shown in image 7[our model may predict a different transformation]:

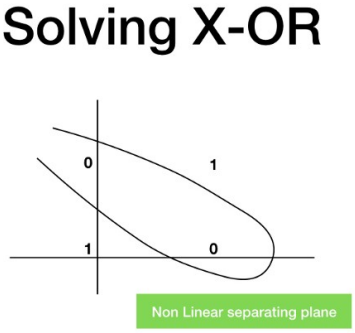


Image 7: A hyperplane boundary to solve X-OR

Following code line implements our intended hidden unit in Keras:

model.add(Dense(units=2,activation="relu",input_dim=2))

Hidden layer has 2 units and uses ReLu as activation. The input to hidden unit is 4 examples each having 2 features. ie a 4x2 matrix. Hence the dimensions of associated weight matrix would be 2x2.

Weights and Biases

All input and hidden layers in neural networks have associated weights and biases. These weights and biases are the values which moves the solution boundary in solutions space to correctly classify the inputs[ref. image 4]. Weights are generally randomly initialized and biases are all set to zero.

For, X-OR values of **initial weights and biases** are as follows[set randomly by Keras implementation during my trial, your system may assign different random values]. We can get weight value in keras using *model.get_weights()* function.

Hidden Layer weights: *[array([[-0.36376578, -0.9752173], [-1.1924702 , 1.1387202]], dtype=float32)*

Hidden Layer bias: *array([0., 0.], dtype=float32)*

Output Layer weights: *array([[0.56955063], [0.27998888]], dtype=float32)*

Output layer bias: *array([0.], dtype=float32)]*

Learning strategy

For learning to happen, we need to train our model with sample input/output pairs, such learning is called supervised learning. Supervised learning approach has given amazing result in deep learning when applied to diverse tasks like face recognition, object

identification, NLP tasks. Most of the practically applied deep learning models in tasks such as robotics, automotive etc are based on supervised learning approach only. Other approaches are unsupervised learning and reinforcement learning. We will stick with supervised approach only.

We are also using supervised learning approach to solve X-OR using neural network.

Loss function and cost function

For the system to generalize over input space and to make it capable of predicting accurately for new use cases, we require to train the model with available inputs. During training, we predict the output of model for different inputs and compare the predicted output with actual output in our training set. The difference in actual and predicted output is termed as loss over that input. The summation of losses across all inputs is termed as cost function. Selection of a loss and cost functions depends on the kind of output we are targeting. For classification we use cross entropy cost function. In Keras we have *binary cross entropy* cost funtion for binary classification and *categorical cross entropy* function for multi class classification.

We compile our model in Keras as follows:

model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['accuracy'])

Back propagation

The goal of training is to minimize the cost function. This is achieved using back propagation algorithm. Back propagation algorithm is a milestone in neural networks, in summary back propagation allows the gradients to back propagate through the network and then these are used to adjust weights and biases to move the solution space towards the direction of reducing cost function. Here is wikipedia link to read more about back propagation algorithm:
https://en.wikipedia.org/wiki/Backpropagation

Training in keras is started with following line:

model.fit(x,y,epochs=1000,batch_size=4)

We are running 1000 iterations to fit the model to given data. Batch size is 4 i.e. full data set as our data set is very small. In practice, we use very large data sets and then defining batch size becomes important to apply stochastic gradient descent[sgd].

Complete Keras code to solve XOR

```
import numpy as np
from keras.layers import Dense
from keras.models import Sequential
```

```
model = Sequential()

model.add(Dense(units=2,activation='relu',input_dim=2))
model.add(Dense(units=1,activation='sigmoid'))

model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['
accuracy'])

print(model.summary())
print(model.get_weights())

x = np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])
y = np.array([1.,0.,0.,1.])

model.fit(x,y,epochs=1000,batch_size=4)

print(model.get_weights())

print(model.predict(x,batch_size=4))
```

Outputs of model after training

Hidden layer weights: array([[1.3936518, -1.2166872],
[-0.8251805, 1.2149163]], dtype=float32)

Hidden layer bias: array([-5.8428086e-02, -1.0609365e-05],
dtype=float32),

Output layer weights: array([[-0.9934472],
[-1.7539338]], dtype=float32)

Output layer bias: array([0.9178055], dtype=float32)]

Prediction for x = [[0,0],[0,1],[1,0],[1,1]]

[[0.7145948]
[0.2291603]
[0.39923137]
[0.6013527]]

values <0.5 mapped to 0 and values >0.5 mapped to 1. Hence, our model has successfully solved the X-OR problem.

Improving the performance of Neural Network:

We will start discussion of performance improvement with respect to following components:

1. Input
2. Output
3. Activation function
4. Weights initialization

5. Loss function
6. Optimizer function

Input

Input in our XOR example is:

x = np.array([[0.,0.],[0.,1.],[1.,0.],[1.,1.]])

In deep learning the optimization strategy applied at input level is Normalization. You can refer following video understand the concept of Normalization: https://www.youtube.com/watch?v=FDCfw-YqWTE

Input in case of XOR is simple. Both the features lie in same range, so It is not required to normalize this input.

In some practical cases e.g. when collecting product reviews online for various parameters and if the parameters are optional fields we may get some missing input values. In such case, we can use various approaches like setting the missing value to most occurring value of the parameter or set it to mean of the values. One interesting approach could be to use neural network in reverse to fill missing parameter values.

Output

Output in our XOR example is:

y = np.array([1.,0.,0.,1.])

It is again very simple data and is also complete.

But, Similar to the case of input parameters, for many practical problems the output data available with us may have missing values to some given inputs. And it could be dealt with the same approaches described above.

Activation function

Activation used in our present model are “relu” for hidden layer and “sigmoid” for output layer. The choice appears good for solving this problem and can also reach to a solution easily.

```
model.add(Dense(units=2,activation='relu',input_dim=2))
model.add(Dense(units=1,activation='sigmoid'))
```

But, with multiple retries with this choice of activation function, i observed that sometimes relu activation can cause a well known problem of **dying ReLu. This occurs when ReLu units are repeatedly receiving negative values as input and as a result the output is always 0. As the gradient of 0 will also be 0, it halts the learning process of network. For more details about dying ReLu, you can**

refer to following article https://medium.com/tinymind/a-practical-guide-to-relu-b83ca804f1f7

Their are various variants of ReLu to handle the problem of dying ReLu, so i replaced “relu” with one of it’s variants called “LeakyReLu” to solve it. It can be done in keras as follows:

```
from keras.layers import LeakyReLU
act = LeakyReLU(alpha = 0.3)
```

and modify hidden layer as follows:

```
model.add(Dense(units=2,activation=act,input_dim=2))
```

This enhances the training performance of the model and convergence is faster with LeakyReLU in this case.

Weight Initialization

Weight initialization is an important aspect of a neural network architecture.

One simple approach is to set all weights to 0 initially, but in this case network will behave like a linear model as the gradient of loss w.r.t. all weights will be same in each layer respectively. It will make network symmetric and thus the neural network looses it’s advantages of being able to map non linearity and behaves much like a linear model.

So, weight are initialised to random values. There are various schemes for random initialization of weights. In Keras, dense layers by default uses “glorot_uniform” random initializer, it is also called Xavier normal initializer.

For more information on weight initializers, you can check out followin keras documentation regarding initialisers https://keras.io/initializers/

In our code, we have used this default initialiser only which works pretty well for us.

Loss function

Selecting a correct loss function is very important, while selecting loss function following points should be considered

1. Measuring the loss i.e. the distance between actual and predicted value effectively
2. Differentiability for using Gradient Descent

Selection of a loss function usually depends on the problem at hand. Following is some examples of loss functions corresponding to specific class of problems

1. Classification->Cross entropy

2. Linear regression->Mean Squared Error

Keras provides binary_crossentropy and categorical_crossentropy loss functions repectively for binary and multi class classification. Checkout all keras supported loss functions at https://keras.io/losses/

As our XOR problem is a binary classification problem, we are using binary_crossentropy loss.

Optimizer function

Optimisers basically are the functions which uses loss calculated by loss functions and updates weight parameters using back propagation to minimize the loss over various iteration. The goal is to move towards the global minima of loss function. Gradient descent is the oldest of the optimisation strategy used in neural networks. Many of it's variants and advanced optimisation functions now are available, some of the most popular once are

1. Stochastic Gradient Descent[sgd]
2. Adagrad
3. Adamax
4. RMSprop
5. Adam

SGD works well for shallow networks and for our XOR example we can use sgd. Others are more advanced optimizers e.g. RMSprop works well in Recurrent Neural Networks. The selection of suitable optimization strategy is a matter of experience, personal liking and comparison. Keras by default uses "adam" optimizer, so we have also used the same in our solution of XOR and it works well for us.

As, out example for this post is a rather simple problem, we don't have to do much changes in our original model except going for LeakyReLU instead of ReLU function. For, many of the practical problems we can directly refer to industry standards or common practices to achieve good results.

P.S. I have started blogging only recently and would love to hear feedback from the community to improve myself.