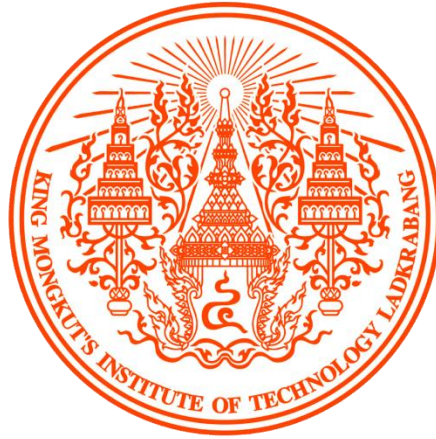


KING MONGKUT'S INSTITUTE OF TECHNOLOGY LATKRABANG
SCHOOL OF ENGINEERING GROUP OF
ROBOTICS & AI



01416305-ARTIFICIAL INTELLIGENCE TECHNOLOGY

Building a Neural Network From The Ground Up for Binary Classification

INSTRUCTED BY: Dr. Bee-ing Sae-ang

Chanet Ruangrit 65011278

Fadil Wannamart 65011297

Kidtipod Juntacheevakul 65011332

1. Introduction

Our primary motivation for choosing to build a neural network from scratch lies in gaining a comprehensive understanding of the fundamental mechanics behind said neural network, bypassing the use of pre-built deep learning frameworks and libraries like TensorFlow or PyTorch.

By choosing not to rely on pre-built framework and libraries, we're taking on the challenge of facing the complexities of neural network design directly. This means we're making careful decisions about how to structure our network, what functions to use for activation and loss, and how to optimize our model. Each decision is guided by a deeper understanding of how neural networks actually work.

2. Problem Statement

While pre-built libraries offer convenience in building neural networks, they can obscure the underlying complexities of this technology. To gain a deeper understanding of how neural networks function, we propose a hands-on approach where we develop a neural network from scratch. This project will involve:

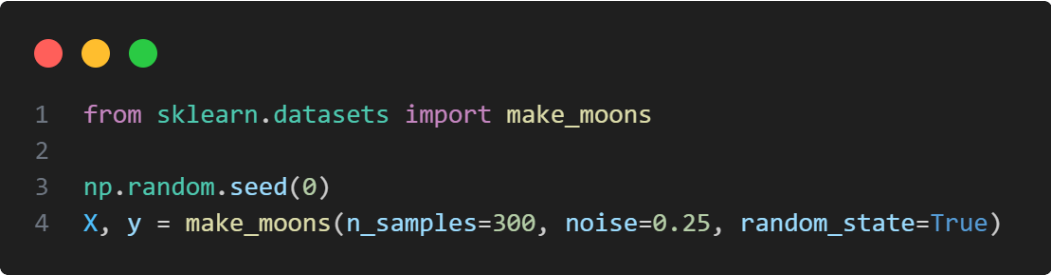
- Designing the network architecture (number of layers, neurons per layer)
- Selecting appropriate activation functions for processing information within the network
- Choosing a suitable loss function to measure the network's performance
- Implementing an optimization algorithm to improve the network's ability to learn from data

By confronting these challenges directly, we aim to achieve a more comprehensive grasp of neural network design principles and how each hyperparameter affects the performance of the model.

3. Methodology

3.1 Dataset Generation

A randomly generated set of 300 circles is used as the basis for classification. Our objective is to construct a neural network capable of differentiating between 2 classes of circles (labeled 0 and 1).



```
1 from sklearn.datasets import make_moons
2
3 np.random.seed(0)
4 X, y = make_moons(n_samples=300, noise=0.25, random_state=True)
```

3.2 Object-oriented programming (OOP)

Structurally, object-oriented programming principles are applied through the definition of classes and their methods.

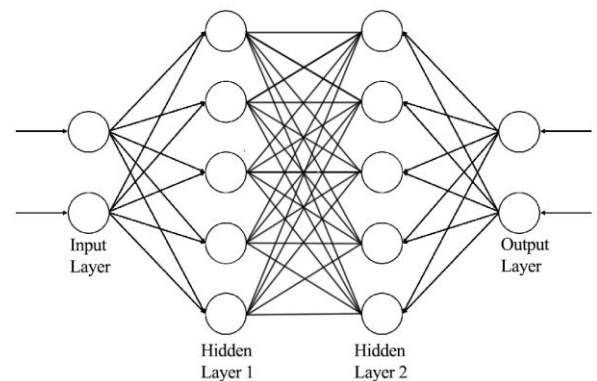
Encapsulation: Each class encapsulates related functionality within itself. For example, the Model class encapsulates methods for initializing the model, making predictions, calculating loss, and training the model. Methods within each class operate on the class's attributes and perform specific tasks related to that class's responsibility and each class can be called for use seamlessly.

```
1 class Model:
2     def __init__(self, layers_dim):
3         .
4         .
5         .
6     def calculate_loss(self, X, y):
7         .
8         .
9         .
10    def predict(self, X):
11        .
12        .
13        .
14    def train(self, X, y, num_passes, epsilon, reg_lambda, print_loss):
15        .
16        .
17        .
```

3.3 Forward propagation

In a neural network, the input is linearly transformed using a weight matrix and bias term, and then passed through a non-linear activation function. The output from the activation function becomes the input for the next layer. This process repeats for all hidden layers until the output layer is reached, where the final output is produced.

For the sake of calculation, let's assume there are 2 input values, x and y coordinates of each circle, with randomized weights and the target outputs are set to red (0) and purple (1), where W (weight) and b (bias) are randomly generated. The Tanh Function is used as the activation function (a),



Our neural network architecture

$$\text{Node: output} = a(W \cdot x^T + b)$$

$$\text{Tanh Function: } \sigma(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

The first equation which is linear in nature $a(w \cdot x^T + b)$ is activated through the Tanh Function to convert it into a non-linear equation.

Calculating the output of the Neural Network starting from h1, where:

$$h1 = x - coordinate * w1 + y - coordinate * w2 + b1$$
$$out_{h1} = \tanh(h1) = \frac{e^{h1} - e^{-h1}}{e^{h1} + e^{-h1}}$$

To calculate for O_1 , out_{h1} becomes input for the next layer.

$$net_{h1} = out_{h1} * w5 + out_{h2} * w6 + b2 \dots out_{hi} * wj + bk$$

$$O_1 = \tanh(net_{h1}) = \frac{e^{net_{h1}} - e^{-net_{h1}}}{e^{net_{h1}} + e^{-net_{h1}}}$$

The SoftMax activation function translates the total value from net node calculation to probability value between 0 and 1 with respect to other values, which makes the sum of probability of a row equal to 1. In our case it is employed during the loss calculation and prediction process to calculate the probability of each output.

$$SoftMax : \sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}$$

, where

σ = SoftMax

\vec{z} = Input vector

e^{z_i} = Standard exponential function for input vector

k = Number of classes in the multi-class classifier

e^{z_j} = Standard exponential function for output vector

3.4 Backpropagation

Backpropagation starts by comparing the error values from the output of the Neural Network with the expected target. Once the error values are obtained, they are then propagated back to those involved, namely the weights. The weights that have higher values are adjusted more, while those with lower values are adjusted less. This process aims to identify the contributors responsible for causing the output values to deviate from the target.

After obtaining the output values from the calculated nodes, we then input them into the loss function to determine how much error our model has. In this case, where we are classifying between two outputs, we choose binary cross-entropy loss function, where

$$CE = Error_{total} = -\frac{1}{N} \sum_{i=1}^N [y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i)]$$

y_i = true label for the i^{th} data point

p_i = Predicted probability for the true label y_i from the Softmax function

```

1 def loss(self, X, y):
2     num_examples = X.shape[0]
3     probs = self.predict(X)
4     correct_logprobs = -np.log(probs[range(num_examples), y])
5     data_loss = np.sum(correct_logprobs)
6     return 1./num_examples * data_loss

```

We can calculate the partial derivative of the loss function with respect to each weight and bias in the network in a chain rule sequence. This is done to determine who is responsible for which error. Each weight is continuously adjusted during the process training. This optimization process is referred to as gradient descent, where

$$\text{gradient: } \frac{\partial Error_{total}}{\partial w_i}$$

By taking steps in the opposite direction of the gradient, we can iteratively update the weights to minimize error which, in this case, is the loss function. This process is known as optimization.

By applying the chain rule, we know that:

$$\frac{\partial Error_{total}}{\partial w_i} = \frac{\partial Error_{total}}{\partial outO_1} * \frac{\partial outO_1}{\partial netO_1} * \frac{\partial netO_1}{\partial w_i}$$

This determines how much the total error changes with respect to the output.

Calculating the first term:

$$\frac{\partial Error_{total}}{\partial outO_1} = \frac{\partial \left[-\frac{1}{N} \sum_{i=1}^N [y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i)] \right]}{\partial outO_1}$$

Calculating the second term:

$$\frac{\partial outO_1}{\partial netO_1} = \frac{\partial \frac{e^{net_{o1}} - e^{-net_{o1}}}{e^{net_{o1}} + e^{-net_{o1}}}}{\partial netO_1}$$

Calculating the last term:

$$\frac{\partial netO_1}{\partial w_i} = \frac{\partial out_{h1} * w5 + out_{h2} * w6 + b2 \dots out_{hi} * wj + bk}{\partial w_i}$$

3.5 Gradient Descent

The process of gradient descent in mathematical form:

$$w_i^{new} = w_i - \alpha * \frac{\partial Error_{total}}{\partial w_i}$$

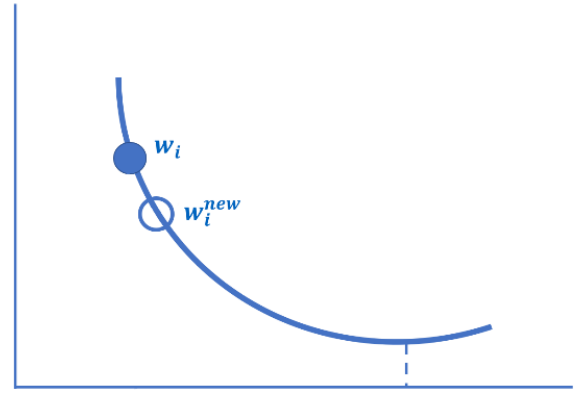
,where

w_{new} = position of the next iteration

w_i = position of the current position

α = Learning rate

$\frac{\partial Error_{total}}{\partial w_i}$ = Gradient



By subtracting the weights with the optimizer, we can iteratively update the weights to minimize the loss and improve the performance of the neural network.

3.6 Regularization

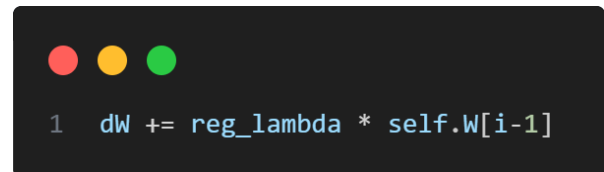
A penalty term is added to the *loss function* that is proportional to the squared magnitude of the weights. This penalty encourages the weights to remain small and thereby prevents overfitting by discouraging large weight values.

$$J_{regularized} = -\frac{1}{N} \sum_{i=1}^N [y_i * \log(p_i) + (1 - y_i) * \log(1 - p_i)] + \lambda(w_{i-1})$$

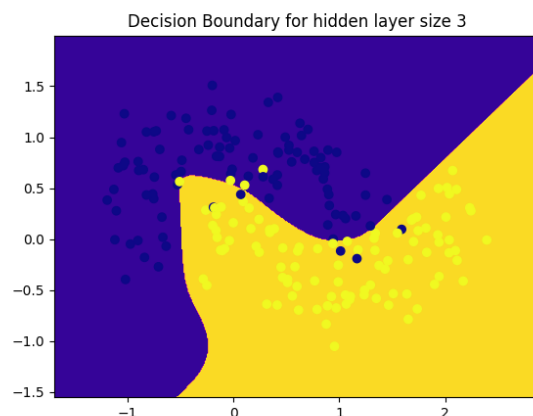
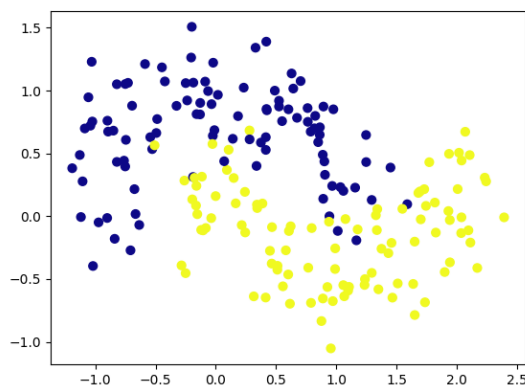
, where

λ = Regularization parameter

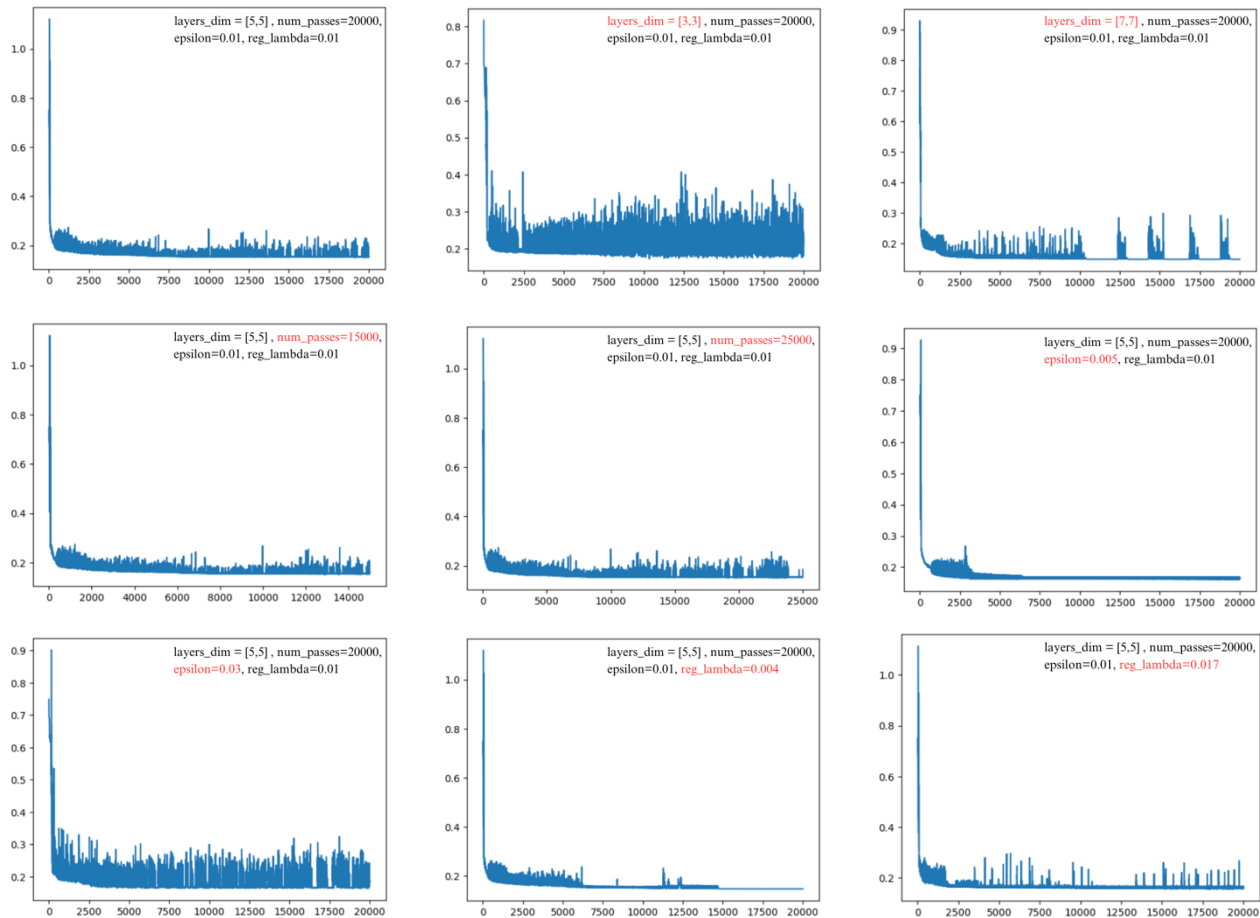
w_{i-1} = weight of current layer



4. Result and Conclusion



By building the neural network from the ground up, we gained insights into how each hyperparameter influences the model's performance. This enabled us to fine-tune the model and achieve satisfactory levels of performance.



- ***layers_dim (neurons):*** The number of neurons in the hidden layers significantly affects the model's capacity to learn intricate patterns. More neurons allow for potentially capturing complex decision boundaries, but they may also lead to overfitting. Overfitting may appear as a decrease in the loss function on the training data while performing poorly on new data.
- ***num_passes (epoch):*** The number of training iterations (epochs) dictates the duration of training on the dataset. More epochs enable the model to grasp patterns better, potentially reducing the loss function. However, excessive epochs may result in overfitting, where the loss continues to decrease on the training data but fails to adapt to unseen data.
- ***epsilon (learning rate):*** This setting controls how much the model adjusts its internal settings based on errors. A higher learning rate can lead to faster initial improvement but may cause the model to miss the optimal settings. A lower learning rate is more stable but may take longer to reach a satisfactory level of performance.
- ***reg_lambda (regularization parameter):*** Regularization reduces the chance of overfitting by penalizing large weights in the model. A higher regularization parameter appends a penalty term to the loss function discouraging overfitting and promoting better generalization. Nonetheless, excessive regularization may hinder the model from capturing essential complex patterns, leading to increased loss.

Appendix

