What is Deep Learning, Really? (Series #1)

Table of Content

Introduction

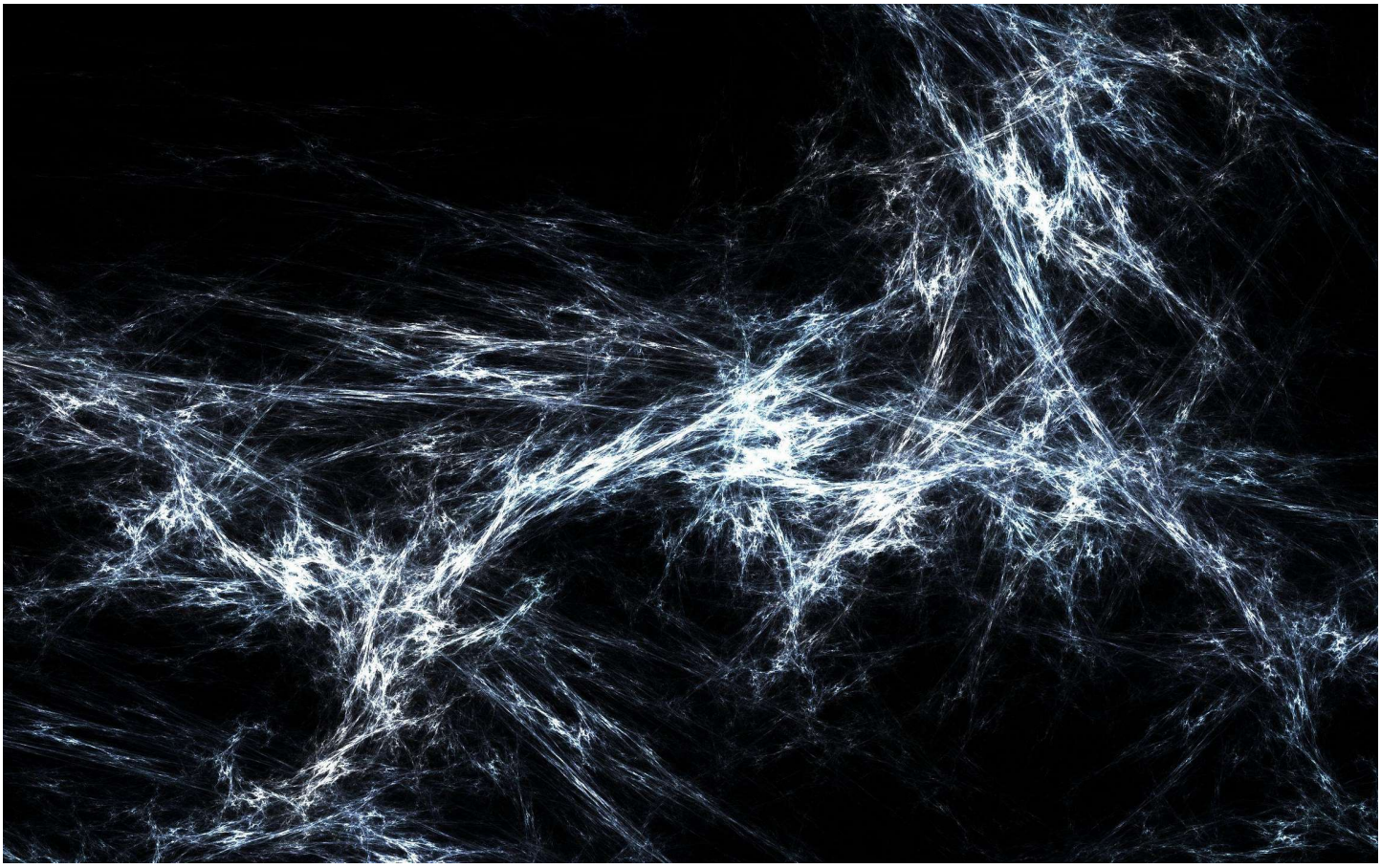
If you're here, you're in for the full journey—from exploring the mathematical machinery that powers it all to grasping the nuts and bolts of how neural networks learn via backpropagation and gradient descent.

In this blog, we make no assumptions about your coding skills or math prowess. We break down the core elements of deep neural networks—weights, biases, optimizers, learning rate and activation functions—using mathematical expressions and building simple FNN with optimization attribute.

We'll uncover the inner workings of learning in neural networks with backpropagation and gradient descent, ensuring you understand not just the what, but the why.

Then, we'll dive into the art of hyperparameter tuning, showing you how to make key choices, such as selecting the right learning rate and loss function, to optimize your neural network using data from Kaggle.

Throughout this blog, math isn't a barrier; it's your ally in unlocking the secrets of deep neural networks.



Get ready to enter a world where equations reveal the intelligence behind the systems, where numbers tell the story of learning, and where deep neural networks become a realm of mathematical fascination.

This blog is your ticket to understanding the inner workings of neural networks like never before.

Feed-Forward Neural Networks

Feed-Forward Neural Networks (FNNs) primarily serve as information transporters, facilitating the flow of data through various optimization processes to ensure that the outputs converge with actual parameters.

For every neuron, the input is multiplied by a weight, which the forward pass algorithm initializes randomly (see: LeCun or Xavier/Glorot Initialization). This weight signifies the importance of each neuron within the network. The concept of 'importance' is quite significant, and we will delve into its further details in the forthcoming chapters.

Similarly, the algorithm also introduces a bias term into the product of the input value and weight for each neuron. This bias is initialized at the outset of training and evolves throughout the training process, exactly as the weight.

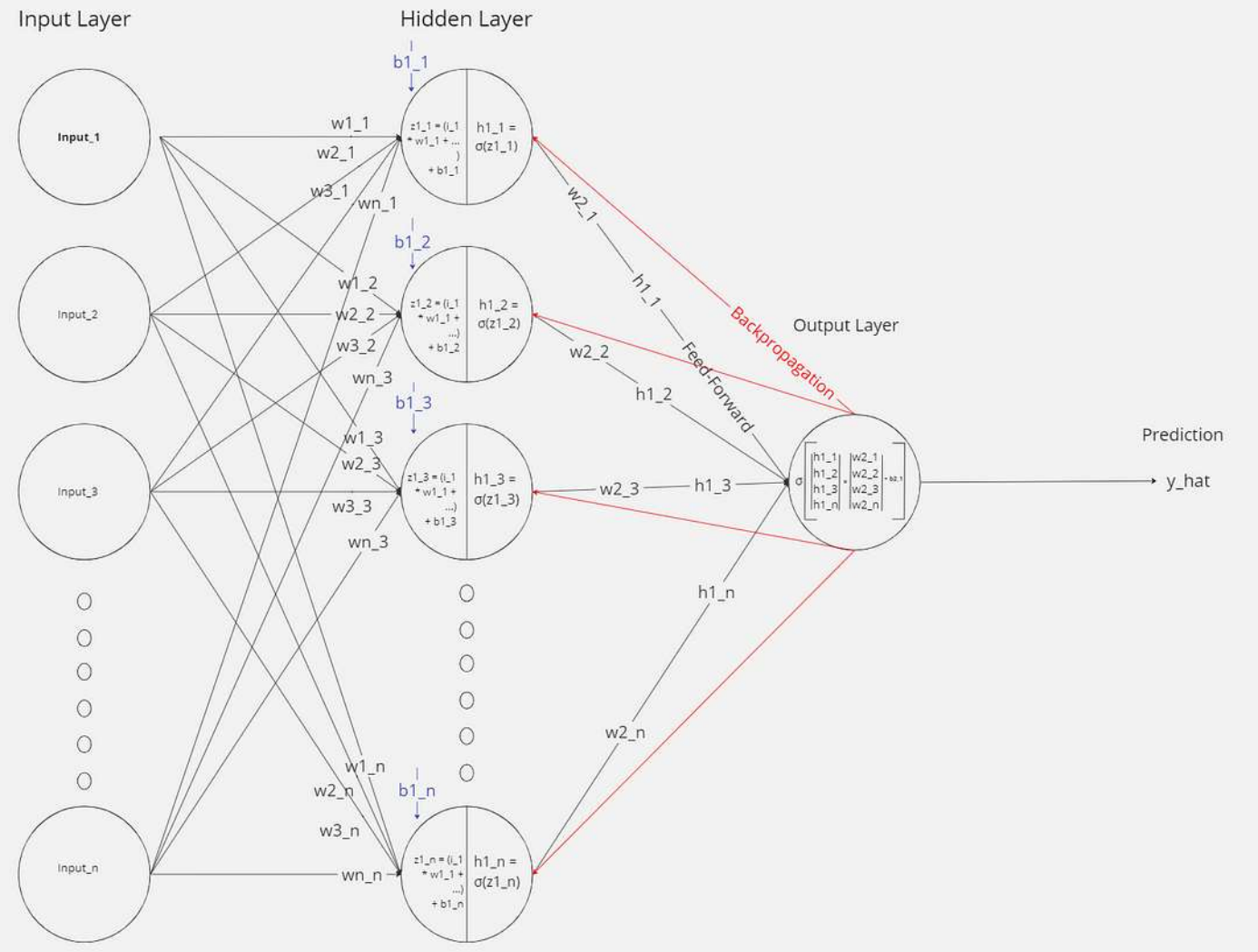
Moreover, during the forward pass, the weighted sum of hidden inputs undergoes processing to capture intricate non-linear connections between features and target variables. Activation functions play a key role here, being applied to the input of each hidden layer neuron to establish these complex non-linear relationships.

This process guides the model to assign greater 'importance' to specific neurons, while rendering less competent neurons less influential. It also significantly accelerates the training time.

However, there are instances when training fails to yield the expected results and leads to the vanishing of some crucial neurons, a phenomenon known as the 'vanishing gradient problem.' These neurons, which could have otherwise made significant contributions to model improvement, become non-trainable. Nevertheless, we will provide an in-depth explanation of this phenomenon in the forthcoming chapters.

Subsequently, after the activations are calculated, a similar mathematical journey begins for the output of hidden layers, culminating in becoming inputs for the selected loss functions.

Then, the loss function measures the extent to which the output predictions deviate from the actual parameters.



Feed-Forward Neural Network (Image by Author)

Quintessentially, the fundamental mathematical expression of a simple FNN with a loss function applied is as follows;

$$\bar{Z}^l = (\bar{W}^l * \bar{h}^{l-1}) + \bar{b}^l$$
$$\bar{h}^l = \sigma(\bar{Z}^l)$$
$$\hat{\bar{Y}} = (\bar{h} * \bar{W}^{l+1}) + \bar{b}^{l+1}$$
$$\text{MSE} = \frac{1}{n} * \sum_{i=1}^{n} (\hat{\bar{Y}} - \bar{Y}_{true})^2$$

Where;

$\bar{W}^l$: Weight matrix of the current layer
$\bar{h}^{l-1}$: Output vector of the previous layer
$\bar{b}_1^l$: Bias vector of the current layer
$\bar{Z}^l$: Input vector of the current layer
$\bar{h}^l$: Output vector of the current layer
σ : Activation Function
$\bar{W}^{l+1}$: Weight matrix of the next layer
$\bar{b}_1^{l+1}$: Bias vector of the next layer
$\hat{\bar{Y}}$: Prediction Vector
n : Number of Predictions
MSE : Loss Function; Mean Squared Error

If we express the element-wise representation of the aforementioned FNN, it would resemble the following:

$$z_1^i = [w_{11}^i * h_1^{i-1} + w_{21}^i * h_1^{i-1} + w_{31}^i * h_1^{i-1} + \ldots + w_{n1}^i * h_1^{i-1}] + b_1^i$$
$$z_2^i = [w_{12}^i * h_2^{i-1} + w_{22}^i * h_2^{i-1} + w_{32}^i * h_2^{i-1} + \ldots + w_{n2}^i * h_2^{i-1}] + b_2^i$$
$$z_3^i = [w_{13}^i * h_3^{i-1} + w_{23}^i * h_3^{i-1} + w_{23}^i * h_3^{i-1} + \ldots + w_{n3}^i * h_3^{i-1}] + b_3^i$$
$$\ldots$$
$$z_n^i = [w_{1n}^i * h_n^{i-1} + w_{2n}^i * h_n^{i-1} + w_{3n}^i * h_n^{i-1} + \ldots + w_{nn}^i * h_n^{i-1}] + b_n^i$$

$$h_1^i = \sigma(z_1^i)$$
$$h_2^i = \sigma(z_2^i)$$
$$h_3^i = \sigma(z_3^i)$$
$$\ldots$$
$$h_n^i = \sigma(z_n^i)$$

$$Y_1^i = (h_1^i * w_1^{i+1}) + b_1^{i+1}$$
$$Y_2^i = (h_2^i * w_1^{i+1}) + b_2^{i+1}$$
$$Y_3^i = (h_2^i * w_1^{i+1}) + b_3^{i+1}$$
$$\ldots$$
$$Y_n^i = (h_2^i * w_1^{i+1}) + b_n^{i+1}$$

At this point, you might intuitively realize that these variables can also be represented using column vectors and matrices. Instead of employing an element-wise representation for each input and output, let's rewrite the entire formula using matrices and vectors.

More importantly, this is where computational efficiency comes into play. By employing dot products across matrices and vectors, we can efficiently calculate inputs and outputs, update weights and biases, and compare loss function outputs.

$$\begin{bmatrix} z_1^1 \\ z_2^1 \\ z_3^1 \\ \vdots \\ z_n^1 \end{bmatrix} = \begin{bmatrix} [w_{11}^1 & w_{21}^1 & w_{31}^1 & \dots & w_{n1}^1] \\ [w_{12}^1 & w_{22}^1 & w_{32}^1 & \dots & w_{n2}^1] \\ [w_{13}^1 & w_{23}^1 & w_{33}^1 & \dots & w_{n3}^1] \\ \vdots & & & & \\ [w_{1n}^1 & w_{2n}^1 & w_{3n}^1 & \dots & w_{nn}^1] \end{bmatrix} \cdot \begin{bmatrix} h_1^0 \\ h_2^0 \\ h_3^0 \\ \vdots \\ h_n^0 \end{bmatrix} + \begin{bmatrix} b_1^1 \\ b_2^1 \\ b_3^1 \\ \vdots \\ b_n^1 \end{bmatrix}$$

$$\begin{bmatrix} h_1^1 \\ h_2^1 \\ h_3^1 \\ \vdots \\ h_n^1 \end{bmatrix} = \begin{bmatrix} \sigma(z_1^1) \\ \sigma(z_2^1) \\ \sigma(z_3^1) \\ \vdots \\ \sigma(z_n^1) \end{bmatrix}$$

$$\begin{bmatrix} Y_1^1 \\ Y_2^1 \\ Y_3^1 \\ \vdots \\ Y_n^1 \end{bmatrix} = \begin{bmatrix} w_{11}^2 & w_{21}^2 & w_{31}^2 & \dots & w_{n1}^2 \\ w_{12}^2 & w_{22}^2 & w_{32}^2 & \dots & w_{n2}^2 \\ w_{13}^2 & w_{23}^2 & w_{33}^2 & \dots & w_{n3}^2 \\ \vdots & & & & \\ w_{1n}^2 & w_{2n}^2 & w_{3n}^2 & \dots & w_{nn}^2 \end{bmatrix} \cdot \begin{bmatrix} h_1^1 \\ h_2^1 \\ h_3^1 \\ \vdots \\ h_n^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \\ \vdots \\ b_n^2 \end{bmatrix}$$

Now it is an appropriate time to introduce ourselves with how this algorithm works with Python code. First, we create a class instance, called “FeedForwardNeuralNetwork”, fnn in short, and it takes a couple of parameters, nothing you haven’t seen yet.

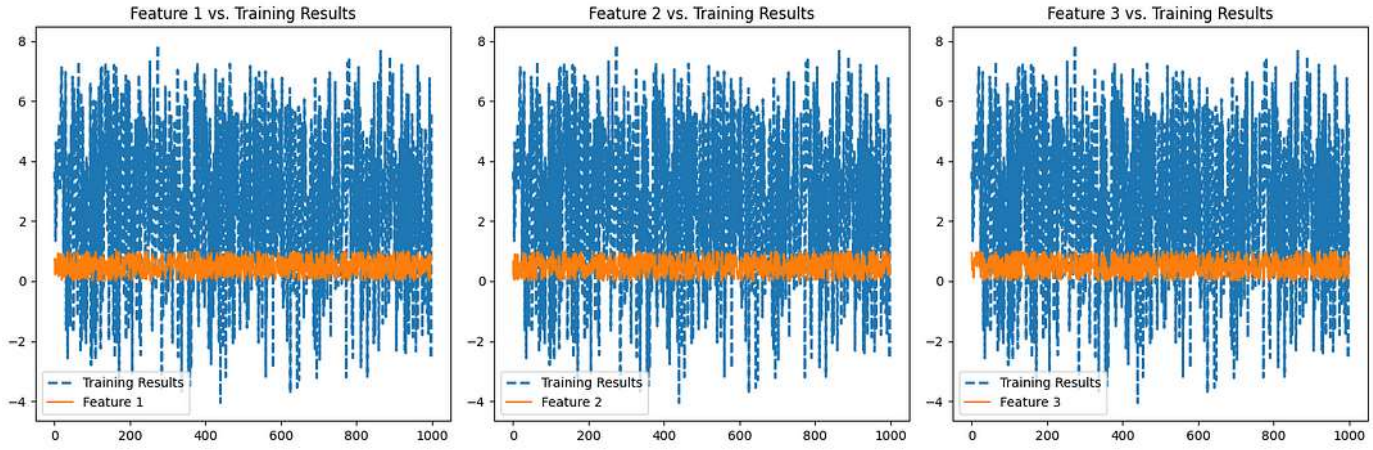
- num_features
- num_hidden_nuerons
- num_output_neurons

This algorithm begins by randomly initializing weights and biases, drawing from a normal distribution with a mean of zero and a variance of 1.

Next, it delves into dot product calculations to compute hidden layer inputs and outputs, followed by another round of dot products to determine the final output inputs.

I’ve also incorporated activation functions. We’ll explore the reasons for their use shortly and examine how different activations impact training.

```python
class FeedForwardNeuralNetwork:
    def __init__(self, num_features, num_hidden_neurons, 
        self.num_features = num_features
        self.num_hidden_neurons = num_hidden_neurons
        self.num_output_neurons = num_output_neurons
        self.hidden_weights, self.hidden_bias, self.outpu

    def initialize_weights(self):
        # Hidden Unit Weight & Bias Initialization
        hidden_weights = np.random.randn(self.num_feature
        hidden_bias = np.random.randn(1, self.num_hidden_

        # Output Layer Weight & Bias Initialization
        output_weights = np.random.randn(self.num_hidden_
        output_bias = np.random.randn(1, self.num_output_

        return hidden_weights, hidden_bias, output_weight

    @staticmethod
    def sigmoid(x):
        return 1 / (1 + np.exp(-x))

    @staticmethod
    def relu(x):
        return np.maximum(0, x)

    @staticmethod
    def tanh(x):
        return np.tanh(x)

    @staticmethod
    def rmse(y_true, y_pred):
        squared_error = np.square(np.subtract(y_true, y_p
        mean_squared_error = np.mean(squared_error)
        rmse = np.sqrt(mean_squared_error)
        return rmse

    @staticmethod
    def mse(y_true, y_pred):
        squared_error = np.square(np.subtract(y_true, y_p
        mean_squared_error = np.mean(squared_error)
        return mean_squared_error

    @staticmethod
    def mae(y_true, y_pred):
        absolute_error = np.abs(np.subtract(y_true, y_pre
        mean_absolute_error = np.mean(absolute_error)
        return mean_absolute_error

    def forward_pass(self, input_data, activation_functio
        # Check if dimensions are compatible
        if input_data.shape[1] != self.hidden_weights.sha
            raise ValueError("Input data and weight dimen

        # Compute the input to the hidden layer
        hidden_input = np.dot(input_data, self.hidden_wei

        # Apply the activation function to the hidden lay
        hidden_output = activation_function(hidden_input)

        # Compute the input to the output layer
        output_input = np.dot(hidden_output, self.output_

        # Calculate the error using the provided loss fun
        error = loss_function(input_data, output_input)
```



As you can observe, without optimization—meaning, without employing the backpropagation algorithm—the training only takes place once and essentially amounts to an awful random guessing.

Additionally, take note that the loss function produces an RMSE score of 3.35. This score is crucial, as the optimization process will try to minimize it. In other words, the algorithm will adjust hyperparameters, such as weights and biases, to minimize the RMSE score as much as possible.

Now, you might be wondering, 'How are these weights and biases initialized?'

. . .

Weights & Biases

At the outset of training, weights and biases are initialized randomly, employing various distribution techniques.

These include normal distribution, uniform distribution, or well-known initialization formulas such as Xavier, LeCun, or He initialization techniques. These values essentially serve as synthetic coefficients that assist the training in converging toward the actual inputs.

Now, let's delve into the mathematics behind these initialization techniques.

> *LeCun Initialization*

LeCun initialization is designed for activation functions like the hyperbolic tangent (tanh) and aims to combat the vanishing gradient problem. It scales the weights based on the number of input units to ensure that the weights are neither too large nor too small, making the training process more stable.

> *LeCun (Uniform Distribution)*

In the uniform distribution version of LeCun initialization, you calculate the "limit" value as:

$$limit = \sqrt{\frac{1}{fan_{in}}}$$

The term "limit" refers to a value that defines the range within which the initial weights will be randomly sampled. The limit specifies the

maximum absolute value of the weights when sampled from the uniform distribution.

When initializing weights, we sample each weight from a uniform distribution within the range [−*limit*, *limit*].

The choice of the "limit" value is crucial because it affects the range of initial weights and can impact the training behavior of the neural network. Different weight initialization methods use different formulas to calculate this "limit" to achieve desirable properties, such as avoiding vanishing or exploding gradients and enabling efficient training.

The specific formula for calculating the "limit" may vary depending on the weight initialization technique being used.

Additionally, the term "fan_in" represents the number of input units in the layer. The reason for using 1 in the numerator is a choice to scale the weights to a reasonable range.

It is important to take note that the selection of limit is solely dependent on the initialization technique deployed. Variations in -+ limit range utterly changes the training, as specified earlier.

LeCun (Gaussian Distribution)

In the normal distribution version, we calculate the standard deviation (σ) as:

$$\sigma = \sqrt{\frac{1}{fan_{in}}}$$

Again, *fan_in* represents the number of input units in the layer. This factor ensures that the weights are initialized with a specific standard deviation in the normal distribution.

Xavier/Glorot Initialization

Xavier initialization, also known as Glorot initialization, is designed to address the problem of gradients vanishing or exploding during training for activation functions like sigmoid and hyperbolic tangent (tanh)

Xavier/Glorot (Uniform Distribution)

In the uniform distribution version of Xavier initialization, you calculate the "limit" value as:

$$\text{limit} = \sqrt{\frac{6}{fan_{in} + fan_{out}}}$$

Here, *fan_in* represents the number of input units, and *fan_out* represents the number of output units in the layer. The factor 6 in the numerator is chosen to ensure that the weights have a broader initial range, which helps mitigate vanishing/exploding gradients.

Xavier/Glorot (Gaussian Distribution)

In the normal distribution version, we calculate the standard deviation (σ) as:

$$\sigma = \sqrt{\frac{2}{fan_{in} + fan_{out}}}$$

Again, *fan_in* represents the number of input units, and *fan_out* represents the number of output units in the layer.

> *He Initialization*

He initialization is designed for activation functions like ReLU (Rectified Linear Unit)

> *He (Uniform Distribution)*

In the uniform distribution version of He initialization, you calculate the "limit" value as:

$$\text{limit} = \sqrt{\frac{6}{fan_{in}}}$$

The factor 6 in the numerator is chosen similarly to Xavier initialization to ensure that the weights have an appropriate initial range to prevent vanishing/exploding gradients.

> *He (Gaussian Distribution)*

In the normal distribution version, we calculate the standard deviation (σ) as:

$$\sigma = \sqrt{\frac{2}{fan_{in}}}$$

In summary, the choice of scaling factors like 6 in Xavier and He initialization methods is somewhat empirical. These values have been found to work well in practice to achieve better convergence and training performance by ensuring that weights are initialized within a certain appropriate range.

Different activation functions and network architectures may benefit from different scaling factors, and these values have been derived through experimentation to provide good starting points for weight initialization.

The choice of the most efficient weight initialization technique, including the appropriate scaling factor, often depends on the specific activation function you're using. Here's a general guideline for commonly used activation functions:

Sigmoid

- Xavier/Glorot initialization is typically efficient with sigmoid activation functions. Use the scaling factor associated with this initialization method for both uniform and normal distributions.

Hyperbolic Tangent (tanh)

- Xavier/Glorot initialization is also efficient with tanh activation functions. Like with sigmoid, use the scaling factor associated with this initialization method for both uniform and normal distributions.

Rectified Linear Unit (ReLU)

- He initialization is commonly used with ReLU activation functions. Use the scaling factor associated with He initialization for both uniform and normal distributions.

Leaky Rectified Linear Unit (Leaky ReLU)

- He initialization, which is designed for ReLU, can also be effective with Leaky ReLU. Use the scaling factor associated with He initialization for both uniform and normal distributions.

It is important to note that these guidelines are not strict rules, and you can experiment with different initialization methods to find the one that works best for your specific neural network and problem.

"But how do we update and optimize these weights and biases?", you might be thinking.

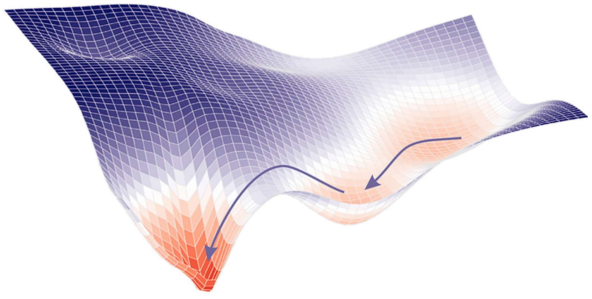
Back-Propagation

The answer lies in an algorithm known as backpropagation. This algorithm essentially reverses the processes of the forward pass, optimizing the loss function in a manner that guides the output of each node to converge towards the actual values.

Here, weights and biases serve as the cornerstone of this process. This 'fine-tuning' is achieved by computing gradients for each weight and bias, revealing how much they contributed to the model's errors in previous training steps.

This algorithm calculates the gradient of each weight and bias starting from the latest layer to the very first. It does so by multiplying the gradient of the cost function with respect to the weight or bias, by a synthetic coefficient known as the 'learning rate,' which is also referred to as the 'step size.' This resulting magnitude is then subtracted from the current weight or bias.

The direction and magnitude of this subtraction depend on the slope of the weight or bias as initialized during the forward-pass. When the hyperparameter, whether it's a weight or bias, is a small value, its gradient is likely to be negatively sloped. This causes the updated weight to increase until it reaches a point where it no longer increases. Conversely, if the slope is positive, the updated weight or bias value will continue to decrease until it reaches the smallest possible value.



Gradient Descent On A Multivariate Plane

This process is repeated throughout the training until the pre-determined number of epochs is reached. We will be introduced to our new term 'Epoch' in the forthcoming chapter. But for now, let's delve into the mathematics behind it.

Gradient Descent

Our cost function—meaning the function that we are going to minimize, can be expressed as the changes in weight and bias with respect to the changes in the performance metric, which is the selected loss function.

$$Loss = \frac{1}{n} \cdot \sum_{1}^{n} L(Y_i, \bar{Y}_i)^2$$
$$w^b = w^a - \eta \cdot \frac{\partial Loss}{\partial W^a}$$
$$b^b = b^a - \eta \cdot \frac{\partial Loss}{\partial b^a}$$

Where;

η : Learning rate
L : A loss function
w^a : Current weight value
b^a : Current bias value
w^b : Updated weight value
b^b : Updated bias value
Y_i : Actual values
$\bar{Y}_i$: Predicted values

This is where the chain rule comes into the play. The chain rule is a fundamental concept in calculus that allows us to calculate the derivative of a composite function.

In the context of backpropagation, it enables us to break down the computation of gradients for each layer by propagating gradients backward through the network. This is crucial because it allows us to understand how a change in a particular weight or bias in one layer affects the overall error in the output, providing the information needed to update these parameters effectively during training.

The chain rule is essentially the mathematical tool that enables the reverse pass of information, making backpropagation possible.

Here, let's combine everything we've covered thus far and initialize some weights and biases, using different techniques to truly grasp how choosing appropriate initialization technique affects the loss function. Then, we proceed with the backpropagation, applying gradient descent.

Let's consider a single neuron with poor initialization:

- Initial weight (W): 0.1
- Initial bias (b): 0.1
- Learning rate (η): 0.1
- Target output (y_true): 1

> *Here is the outline of the following computations*

1. Compute the neuron's weighted inputs
2. Compute the neuron's output using the sigmoid activation function
3. Calculate the loss using mean squared error
4. Compute the gradient of the loss with respect to the weight using the chain rule
5. Calculate the gradient of the loss with respect to the bias, and update
6. Update weight and bias using gradient descent

Scenario 1. Poor Weight and Bias Initialization

$$
\begin{aligned}
z &= 1 \cdot 0.1 + 0.1 \approx 0.2 \\
h_1 &= \frac{1}{1+e^{-0.2}} \approx 0.5498.. \\
\text{Loss} &= \frac{1}{2}(1 - 0.5498) \approx 0.1237 \\
\frac{\partial Loss}{\partial h_1} &= -(y_{true} - h_1) \approx -0.4502 \\
\frac{\partial h_1}{\partial z} &= h_1 \cdot (1 - h_1) \approx 0.2476 \\
\frac{\partial z}{\partial W} &= x = 1 \\
\frac{\partial Loss}{\partial W} &= -0.4502 \cdot 0.2476 \cdot 1 \approx -0.11146 \\
\frac{\partial Loss}{\partial b} &= \frac{\partial Loss}{\partial h_1} \cdot \frac{\partial h_1}{\partial z} \cdot \frac{\partial z}{\partial b} = -0.4502 \cdot 0.2476 \cdot 1 \approx -0.11146
\end{aligned}
$$

Let's use gradient descent to update weight and bias

$$
\begin{aligned}
W_{new} &= W - h_1 \cdot \frac{\partial Loss}{\partial W} = 0.1 \cdot -0.1 \cdot (-0.11146) = 0.001115 \\
b_{new} &= b - h_1 \cdot \frac{\partial Loss}{\partial b} = 0.1 \cdot -0.1 \cdot (-0.11146) = 0.001115
\end{aligned}
$$

Weight and bias initialization using Uniform Distribution

Now, let's initialize the same weight and bias using a better approach.

Scenario: Xavier/Glorot Initialization

$$
\begin{aligned}
z &= 1 \cdot W + 0.1 \approx 0.6 \\
h_1 &= \frac{1}{1+e^{-0.6}} \approx 0.6457 \\
\text{Loss} &= \frac{1}{2}(1 - 0.6457)^2 \approx 0.0978 \\
\frac{\partial Loss}{\partial h_1} &= -(y_{\text{true}} - h_1) \approx -0.3543 \\
\frac{\partial h_1}{\partial z} &= h_1 \cdot (1 - h_1) \approx 0.2282 \\
\frac{\partial z}{\partial W} &= 1 \\
\frac{\partial Loss}{\partial W} &= -0.3543 \cdot 0.2282 \cdot 1 \approx -0.0811 \\
\frac{\partial Loss}{\partial b} &= \frac{\partial Loss}{\partial h_1} \cdot \frac{\partial h_1}{\partial z} \cdot \frac{\partial z}{\partial b} = -0.3543 \cdot 0.2282 \cdot 1 \approx -0.0811
\end{aligned}
$$

$$
\begin{aligned}
W_{\text{new}} &= W - \alpha \cdot \frac{\partial Loss}{\partial W} = W - 0.1 \cdot (-0.0811) = W + 0.00811 \\
b_{\text{new}} &= b - \alpha \cdot \frac{\partial Loss}{\partial b} = 0.1 - 0.1 \cdot (-0.0811) = 0.10811
\end{aligned}
$$

Weight and bias initialization using Xavier Initialization (Normal Distribution) Formula

> *Poor Initialization (Uniform Distribution)*

- Initial Weight (*W*): 0.1
- Initial Bias (*b*): 0.1
- Loss: Approximately 0.1237
- Weight Update: *W*_new ≈ 0.01115
- Bias Update: *b*_new ≈ 0.01115

> *Xavier/Glorot Initialization (Normal Distribution)*

- Initial Weight (W): Approximately 0.5
- Initial Bias (b): 0.1
- Loss: Approximately 0.0978
- Weight Update: W_new ≈ 0.5081
- Bias Update: b_new ≈ 0.1081

> *Comparison between Scenarios*

Xavier/Glorot Initialization provides a more suitable initial weight value (approximately 0.5), which is closer to an ideal range, while poor initialization starts with a significantly smaller weight (0.1).

Also Xavier Initialization results in a lower initial loss value (approximately 0.0978) compared to poor initialization (approximately 0.1237).

When applying gradient descent to update weights and biases, Xavier/Glorot Initialization yields weight and bias updates that are better suited for efficient learning and convergence. For example, W_new for Xavier initialization is approximately 0.5081, which is a larger and more suitable update compared to the small update (W_new ≈ 0.01115) in the poor initialization scenario—meaning that, weight and bias values from Xavier Initialization are less likely to experience vanishing/exploiting gradient issue.

In summary, Xavier/Glorot Initialization produces initializations and updates that are closer to ideal values, contributing to faster convergence and more efficient learning compared to poor initialization.

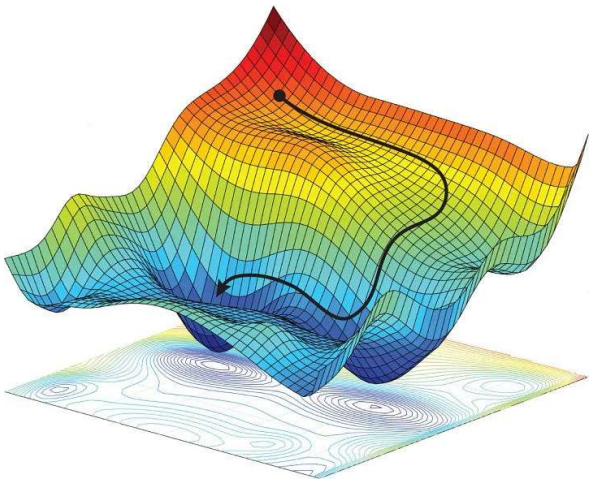
Learning Rate

Now that we have mentioned about “faster convergence”, you might intuitively think of the training duration. In fact, you are correct, however the underlying reason is much more complicated than the time elapsed.

Finding a good step size, or learning rate, is an empirical endeavor. At its core, neural network training often relies on gradient descent algorithms.

These optimization techniques work by iteratively adjusting the model’s parameters (weights and biases) to minimize the cost or loss function. The learning rate dictates how large or small these adjustments are.

A too-large learning rate can result in overshooting the minimum of the loss function, causing divergence and preventing convergence. Conversely, an overly small learning rate can lead to painfully slow convergence, as the steps taken are tiny.



The choice of learning rate is intimately linked to the training duration. Inefficient learning rates can lead to training processes that drag on for an eternity, especially for deep and complex neural networks.

In practice, neural network practitioners often begin their experiments with common learning rate values such as 1e-2 (0.01), 1e-3 (0.001), or 1e-4 (0.0001). These values have proven effective in a wide range of scenarios and can serve as good starting points for experimentation.

However, it's important to note that there is no one-size-fits-all learning rate. The ideal rate can vary depending on the specific problem, dataset, and network architecture. Therefore, experimentation and fine-tuning are crucial.

In essence, the learning rate is a delicate balancing act. Finding the right value requires experimentation and fine-tuning. It's an empirical journey where model performance, training duration, and prediction capability are at stake.

Optimizers

We've explored the critical role of the learning rate in neural network training, let's delve deeper into the world of optimization techniques.

Just as the learning rate impacts the efficiency of gradient descent, various optimizers play a crucial role in shaping how our models learn and adapt during the training process.

Optimizers, in the context of neural networks, are algorithms or methods that guide the process of adjusting the model's parameters (weights and biases) during training. The primary objective is to minimize the cost or loss function, ultimately leading to a well-trained and accurate model.

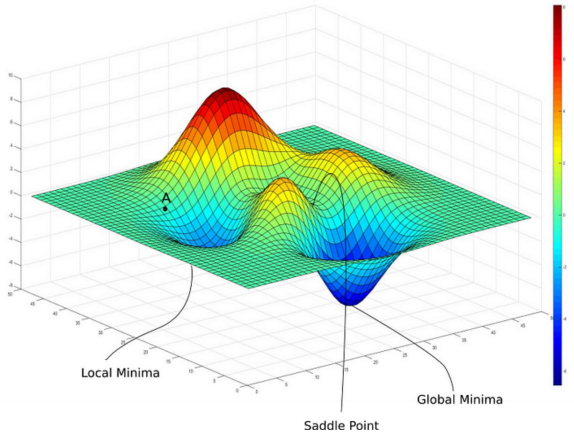
"Why would we be in need of using optimizers?" you might wonder.

Decay and momentum are crucial components in optimization algorithms, particularly in fine-tuning their performance. Learning rate decay ensures that the learning rate decreases over time, helping optimization to settle into a minimum effectively and prevent overshooting.

On the other hand, momentum introduces a dynamic element to parameter updates by considering the accumulation of gradients

from previous iterations, enabling faster convergence and overcoming optimization challenges like plateaus.

These two elements, decay and momentum, play pivotal roles in ensuring efficient and stable optimization, making them essential considerations in the design of effective learning algorithms.



Take this 3D multivariate function as an example. Let's consider the loss function, the point A on the graph, is converging toward the local minima. However, as you might guess we have a better point all the way down at global minima.

Here, momentum and decay comes into the play and helps learning rate to navigate the system to find its lowest point possible.

First, loss function would be increasing to the point at the plateau, called "Saddle Point", this is where we receive increasing training/validation losses and find ourselves thinking "Maybe the model is overfitting and I should use the early stopping callback", however, this is where the momentum is pushing the model toward the global minima by providing some thruster. Now, it is an appropriate time to get to know those optimizers.

Stochastic Gradient Descent(SGD)

Update Rule

$$\theta_{t+1} = \theta_t - \eta \nabla J(\theta_t)$$

Where;

θ_t : Model parameters at time step t

η : Learning rate

$\nabla J(\theta_t)$: Gradient of the loss function J with respect to the parameters θ_t.

Momentum

$$v_{t+1} = \beta v_t + (1-\beta)\nabla J(\theta_t)$$
$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

Where;

β : momentum term, typically close to 1

v_t : momentum vector

The learning rate (η) in SGD is a pivotal hyperparameter. It defines the size of steps taken during parameter updates. A well-chosen learning rate is essential for efficient training. If the learning rate is too large, the optimizer might overshoot the minimum, causing divergence. Conversely, if it's too small, the optimizer might converge very slowly. Therefore, finding the right learning rate is a crucial part of using SGD effectively.

The gradient ($\nabla J(\theta t)$) in SGD represents the direction and magnitude of the steepest ascent of the loss function with respect to the model's parameters (θt). It provides the information needed to determine how the parameters should be updated to reduce the loss. SGD

computes and uses this gradient to adjust the model's weights and biases in each training iteration.

SGD is the foundation upon which many other optimization algorithms are built. Its simplicity and effectiveness make it a popular choice in various machine learning tasks. However, the careful tuning of the learning rate is often required to strike the right balance between fast convergence and stability during training

Adagrad (Adaptive Gradient Algorithm)

Update Rule

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \nabla J(\theta_t)$$

Where;

G_t : Sum of squared past gradients

ϵ : Small constant to avoid division by zero

Adagrad maintains a sum of the squared past gradients (*G_t*). It adapts the learning rate individually for each parameter based on the historical gradient information. This adaptivity allows Adagrad to perform well when dealing with sparse data.

RMSprop (Root Mean Square Propagation)

Update Rules

$$G_t = \beta G_{t-1} + (1 - \beta)(\nabla J(\theta_t) \odot \nabla J(\theta_t))$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{G_t + \epsilon}} \odot \nabla J(\theta_t)$$

Where;

G_t : Moving average of squared past gradients

ϵ : Small constant to avoid division by zero

RMSprop, similar to Adagrad, computes a moving average of squared past gradients (*G_t*). However, it introduces an exponentially weighted moving average, which helps prevent the learning rate from becoming too small during training. RMSprop is effective in handling non-stationary or noisy environments.

Adam (Adaptive Moment Estimation)

Update Rules

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla J(\theta_t)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\nabla J(\theta_t) \odot \nabla J(\theta_t))$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{\hat{v}_t + \epsilon}} \odot \hat{m}_t$$

Where;

β_1 : Exponential decay rate for the first moment estimates

β_2 : Exponential decay rate for the second moment estimates

m_t : First moment (mean) of the gradients

v_t : Second moment (uncentered variance) of the gradients

ϵ : Small constant to avoid division by zero

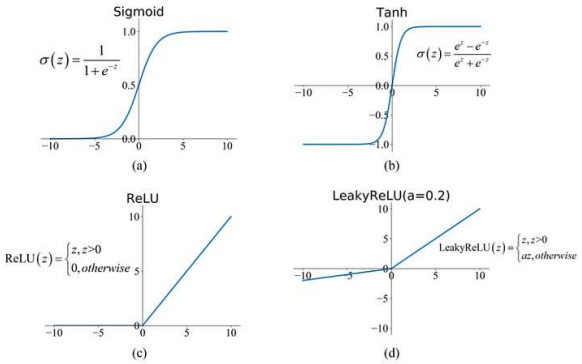
Adam, short for Adaptive Moment Estimation, maintains two moving averages (*m_t* and *v_t*) for the first moment (mean) and second moment (uncentered variance) of the gradients, respectively. These moving averages are used to adaptively adjust the learning rates for each parameter. Adam combines the benefits of momentum and adaptivity.

We've explored optimization techniques and their significance well enough. Now, let's shift our focus to another critical aspect: activation functions.

Activation Functions

Activation functions introduce non-linearity to the network, enabling it to learn complex relationships in data. Without non-linearity, a neural network would essentially reduce to a linear model, severely limiting its capacity to model intricate patterns and representations.

They transform the input signal, allowing neural networks to learn and represent features at different levels of abstraction. This ability to capture hierarchical features is vital for tasks like image recognition, natural language processing, and more.



Now that we know why we use activation functions, let's see their use in action in different tasks and data.

Sigmoid

Sigmoid is historically used in the output layer for binary classification problems, where the network's output needs to be in the range [0, 1].

It squashes the output into a bounded range, which can be useful for interpreting probabilities. However, it is prone to vanishing gradients, especially in deep networks, which can slow down training.

Hyperbolic Tangent Function (tanh)

Similar to the sigmoid function, but with an output range between -1 and 1. It is used in scenarios where the data distribution is centered around zero, which helps in quicker convergence during training compared to the sigmoid function. However, still susceptible to vanishing gradients.

Rectified Linear Unit (ReLU)

ReLu is the default choice for many hidden layers in deep neural networks and is efficient to compute and alleviates the vanishing gradient problem for positive values.

It often leads to faster training and better convergence. However, as other activation functions it can suffer from the "dying ReLU" problem, where neurons can get stuck during training and never activate.

Last but not least,

Leaky Rectified Linear Unit (Leaky ReLU)

Leaky ReLU addresses the "dying ReLU" problem by allowing a small gradient for negative values and prevents neurons from being completely inactive during training and can lead to better generalization. However, Leaky ReLu is not always necessary for all tasks and can introduce a new hyperparameter to fine-tune (the leak factor).

To conclude, Choosing the right activation function is often an empirical process, involving experimentation with different functions to determine which one works best for a particular task and dataset. Additionally, architectural choices, like using skip connections or batch normalization, can also influence the choice of activation functions.

Loss Functions

Choosing the right loss function is a critical decision when designing and training machine learning models. The selection of a loss function should align with the specific task and characteristics of the data. Using the wrong loss function can lead to suboptimal model performance. It's essential to thoroughly understand the problem at hand, the desired model output (e.g., regression, binary classification, multi-class classification), and any specific requirements (e.g., handling outliers or class imbalances) when making this choice.

The correct choice of a loss function can have a significant impact on the model's capacity to learn from the data and generalize effectively, ultimately determining the success of the machine learning application.

To emphasize the importance of selecting the appropriate loss function, let's consider a dataset where the target labels are imbalanced, whether they are binary [0,1], sentiment labels [positive, neutral, negative], or involve multiple classes.

The intuitive approach might be to choose accuracy as the loss function. The model would then optimize its hyperparameters to minimize the divergence based on accuracy. However, when dealing with imbalanced data, this approach is akin to navigating blindfolded.

In such cases, we have two options: either apply data augmentation methods like SMOTE to balance the data or, more importantly, select the appropriate loss function to accurately measure our losses. In this scenario, the precision-recall score becomes a more suitable choice than accuracy.

Now, let's explore some of the commonly used loss functions for different machine learning tasks and types of data.

Mean Squared Error (MSE)

Typically used for regression tasks where the goal is to predict continuous values.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Binary Cross-Entropy (Log Loss)

Suitable for binary classification problems where the output is either 0 or 1.

$$\text{Binary Cross-Entropy} = -\frac{1}{N}\sum_{i=1}^{N}[y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

Categorical Cross-Entropy (Softmax Cross-Entropy)

Commonly used for multi-class classification tasks where there are more than two classes.

$$\text{Categorical Cross-Entropy} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{C} y_{ij} \log(\hat{y}_{ij})$$

Huber Loss

Combines properties of MSE and MAE, useful when dealing with outliers in regression tasks.

$$\text{Huber Loss} = \frac{1}{N}\sum_{i=1}^{N} \begin{cases} \frac{1}{2}(y_i - \hat{y}_i)^2 & \text{if } |y_i - \hat{y}_i| \leq \delta \\ \delta(|y_i - \hat{y}_i| - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

Now, the last step before proceeding to building a model with actual data, the very heart of building any model—Hyperparameters...

Hyperparameters

To name some of the hyperparameters we've seen thus far;

- Optimizer
- Learning Rate
- Number of layers
- Number of Neurons in each layer
- Weight and Bias Initialization techniques ,

Now, let's see a couple of more of them that we are going to use in the following chapter.

Epoch

- An epoch is one complete pass through the entire training dataset during neural network training.
- Use it when you want to define how many times the entire dataset should be used for training. The number of epochs is often a hyperparameter you can tune.

Batch Size

- Batch size is the number of training examples used in one iteration (forward and backward pass) to update the model's parameters.
- Use it to control the trade-off between computational efficiency and model convergence. Smaller batch sizes may generalize better but take longer to train, while larger batch sizes speed up training but might lead to overfitting.

Batch Normalization

- Batch normalization is a technique used to stabilize and accelerate training by normalizing the inputs to a layer in mini-batches.

- Use it to address the vanishing gradient problem, speed up training, and reduce sensitivity to weight initialization.

Layer Normalization

- Layer normalization is similar to batch normalization but normalizes the inputs across the entire layer for each training example.
- Use it when batch normalization isn't suitable, such as in recurrent neural networks (RNNs) where batch sizes can be small or inconsistent.
- Use layer normalization especially when dealing with LSTMs to address time-series/dependent modelling. Batch normalization layer carries past information to, the mean, to following information and causes auto-correlation within data to arise in time series.

Dropout

- Dropout is a regularization technique that randomly deactivates a fraction of neurons during each training iteration.
- Use it to reduce overfitting and improve the model's generalization ability, especially in deep networks.

Residual Connection (ResNet)

- Residual connections are used in deep neural networks to skip one or more layers in a network.
- Use them to mitigate the vanishing gradient problem and enable training of very deep networks. They are particularly valuable in tasks requiring deep architectures, such as image classification.

Each of these techniques serves specific purposes in training and regularizing neural networks, and their effectiveness depends on the problem, the architecture, and the data. The choice of which to use and when depends on the specific challenges and goals of your machine learning task.

Now, we are closing this chapter to open up another one. We are building a deep neural network to train a model for a regression task. Let's dive in

• • •

Developing Artificial Neural Network For A Regression Task

Become One With Data

We will be using an insurance dataset from the Medical Cost Personal database available on Github. The dataset contains 1.33k rows and 7 features, 3 of which are categorical, while the remaining consist of numerical variables.

For all Machine Learning problems, it is crucial to begin by getting to know your data. This step might involve preprocessing for various reasons, such as handling missing values, addressing outliers, correcting data types, converting text data into categorical variables, and dealing with scaling issues that can arise from having both large

and small numeric values in the same dataset, among other potential challenges.

For these reasons, we should start by familiarizing ourselves with our insurance cost dataset.

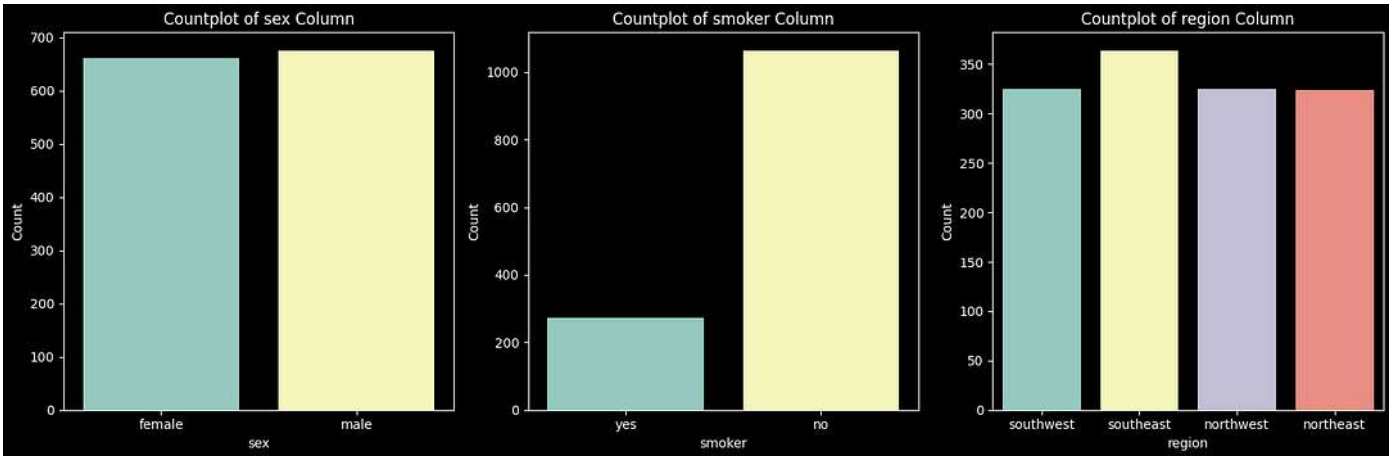
	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

Medical Insurance Cost Dataset

Here, I've presented a summary of our data. As you can see, these are crucial features that can significantly impact the cost related to current or potential medical needs that may arise in the future.

For this reason, insurance companies collect, analyze, and forecast similar personal data from their clients to assess their financial risks, allocate budgets more effectively, and, in some cases, make decisions about high-risk clients. High-risk clients may pose a greater financial burden to insurance companies due to their potentially higher health-related risks.

Now, let's dive into the data and see what insights it holds.



Count plot of Categorical Variables

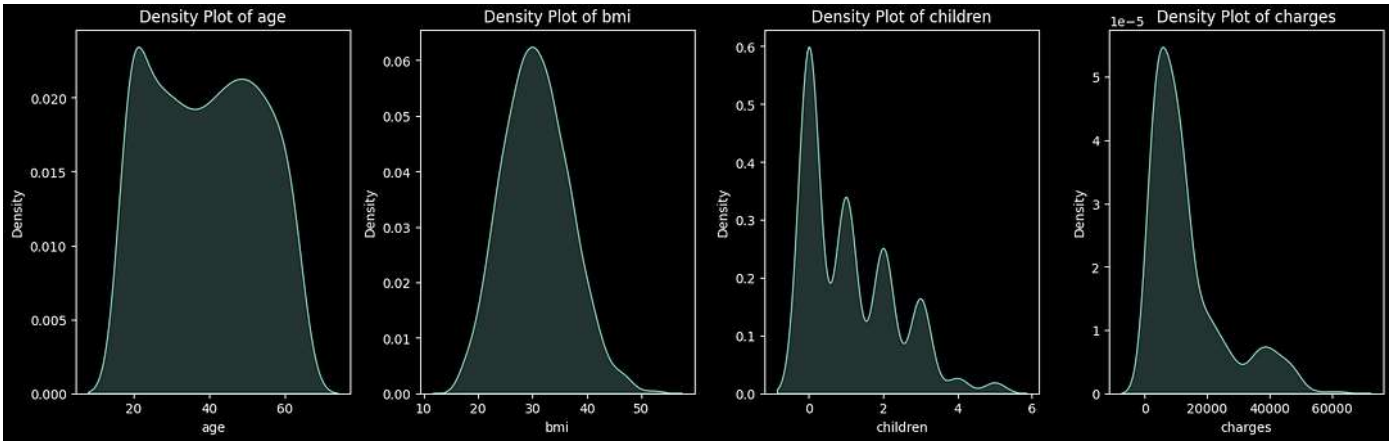
As mentioned earlier, we have three categorical variables:

1. Sex
2. Smoking Information (binary; yes or no)
3. Region

You may have noticed that the "smoker" column is highly imbalanced, with the majority consisting of non-smokers. In contrast, the remaining categorical variables appear to be evenly distributed.

Nonetheless, we will need to apply categorical transformation to convert this data into binary values. This transformation will enable the model to better capture non-linear relationships within the data.

These non-linearities might involve scenarios where an individual has never smoked, is male, has a higher BMI, yet may be charged less for insurance. The reason for this could be related to factors such as their age being lower than 30, exceeding a certain age threshold, or having a certain number of children. Our model will help answer questions like these.



Kernel Density Estimations of Numerical Variables

The numerical variables in our dataset are evidently not normally distributed and are scaled at different levels. For instance, we have the “charges” column with values ranging from 0 to 60k+, and the “age” column with values ranging from 18 to 60+.

To prepare these columns as better inputs for our model and enable it to capture non-linearities effectively, we’ll employ several data preprocessing methods.

First and foremost, we’ll divide our dataset into three sets: validation, training, and test. We set aside the validation set to avoid data leakage during hyperparameter tuning. This separation is crucial because our model will learn from the training data and make predictions based on it, while we want to assess its performance on unseen data from the test set.

Next, we’ll utilize the `make_column_transformer` package from scikit-learn to transform both categorical and numerical columns into suitable inputs for the model. To achieve this, we'll use `OneHotEncoder` to convert categorical variables into binary values (dummy variables) and `MinMaxScaler` to scale all numeric values to a consistent range.

It’s worth noting that we’ll apply this preprocessing to all our datasets, including the validation, training, and test sets. This ensures that the data used for training and evaluation is consistently processed and allows us to make fair comparisons and reliable predictions.

Preprocessing

```python
def divide_to_sets(X: np.array, y: np.array, val_cut: floa
    """
    Divide the dataset into training, testing, and validat

    Parameters:
    X (numpy.array): Feature values of the dataset.
    y (numpy.array): Target values of the dataset.
    val_cut (float): Proportion of data to be used for val
    test_cut (float): Proportion of data to be used for te

    Returns:
    tuple: A tuple containing X_train, X_test, y_train, y_

    """

    validation_size = int(len(X) * val_cut)
    X_val = X[-validation_size:]
    y_val = y[-validation_size:]
    X_train, X_test, y_train, y_test = train_test_split(
        X[:-validation_size],
        y[:-validation_size],
        test_size=test_cut,
        shuffle=True,
        random_state=42
    )

    return X_train, X_test, y_train, y_test, X_val, y_val

# Instantiate the scaler
col_transform = make_column_transformer(
    (MinMaxScaler(), ["age","bmi","children"]),
    (OneHotEncoder(handle_unknown="ignore"), ["sex","smoke
)

# Separate feature and target
X = insurance.drop(["charges"], axis=1)
y = insurance["charges"]

# Divide data as training, testing and validation sets
X_train, X_test, y_train, y_test, X_val, y_val = divide_to_

```

The preprocessing phase has increased the number of features in our dataset by separating unique values. For example, it now distinguishes between being a smoker or not, living in different regions like region_x or region_y, and being male or female, treating each as distinct features. Consequently, we now have 11 features, which is four more columns than we had at the beginning.

Modelling

Let's briefly recap what we've accomplished so far. We began by gaining a deep understanding of our dataset. Following that, we divided it into three distinct parts for modeling purposes. Then, we undertook the critical task of preprocessing the data to create improved feature inputs.

Now, it's time to set up some callbacks to monitor how our model progresses with the specified hyperparameters. Additionally, we'll save each best model along with its corresponding loss metric and prediction score.

We'll also establish a callback to dynamically reduce the learning rate if our model fails to show improvement for 5 consecutive epochs. Furthermore, we'll initialize early stopping to halt training if the model doesn't make progress in 10 iterations.

Lastly, the CSVLogger will be configured to record training and testing scores/losses throughout each unique training session. This will provide us with valuable insights into model performance over time.

Our next step involves conducting hyperparameter tuning using the `product` module from the `itertools` library. This will allow us to systematically experiment with various hyperparameter combinations to identify the best-performing ones.

As a consideration for the loss functions, we will be using root mean squared error, since we are building a model for regression tasks and RMSE is one of the most common performance metrics (loss function) for regression tasks. Also, we will use ReLu as our choice of activation function.

The model architecture will have various layers such as, `Dense`, `BatchNormalization`, `Dropout`. We add batch normalization, to regularize our data and omit out some untrainable nodes, and dropout to accelerate training and again, preventing overfitting by randomizing the inputs to the next layer. Usually, we first attach batch normalization, and then dropout layer before connecting them to any other fully-connected layer, however, it is up to you to carry out different experiments.

Lastly, the following hyperparameters are experimented within the loop:

- epoch: [64, 128, 256, 512]
- batch_size: [8, 16, 32, 64, 128, 256]
- learning_rate: [1e-1, 1e-2, 1e-3, 1e-5]
- optimizers: [SGD, Adam]

Get ready, we're about to unleash the power of this machine!

```
tf.random.set_seed(42)
# Save Best epoch based on the smallest RMSE score
cb_checkpoint = tf.keras.callbacks.ModelCheckpoint(
    filepath="new_checkpoints/model-{epoch:02d}-{root_mea
    monitor="val_loss",
    mode="min",
    save_best_only=True,
    verbose=1
)

# Drop the learning rate if rmse score does not improve
cb_reducelr = tf.keras.callbacks.ReduceLROnPlateau(
    monitor="val_root_mean_squared_error",
    mode="min",
    factor=0.1,
    patience=5,
    verbose=1,
    min_lr=1e-7
)

# Perform early stopping if model does not improve after
cb_earlystop = tf.keras.callbacks.EarlyStopping(
    monitor="val_root_mean_squared_error",
    mode="min",
    min_delta=0.001,
    patience=10,
    verbose=1,
)

# Save training loss, val_loss and performance metrics as
cb_csvlogger = tf.keras.callbacks.CSVLogger(
    filename="training_log.csv",
    separator=",",
    append=False
)

def evaluate_regression(y_true, y_pred):

    mae = tf.metrics.mean_absolute_error(y_true, tf.squee
    mse = tf.metrics.mean_squared_error(y_true, tf.squeez
    rmse = np.sqrt(mse)
    metrics_df = ({"mae": mae.numpy(),"mse": mse.numpy(),
                   "rmse": rmse}, index=[0])
    return metrics_df

def build_model(loss_metric, optimization_algorithm, eval

  tf.random.set_seed(42)
  model= tf.keras.Sequential([
      tf.keras.layers.BatchNormalization(),
      tf.keras.layers.Dense(100, activation="relu"),
      tf.keras.layers.BatchNormalization(),
      tf.keras.layers.Dropout(0.5),
      tf.keras.layers.Dense(100, activation="relu"),
      tf.keras.layers.BatchNormalization(),
      tf.keras.layers.Dropout(0.5),
      tf.keras.layers.Dense(100, activation="relu"),
      tf.keras.layers.BatchNormalization(),
      tf.keras.layers.Dense(1, activation="linear")
  ])

  model.compile(loss=loss_metric,
                optimizer=optimization_algorithm,
                metrics=eval_metric)

  return model
```

Evaluation

	optimizer	epoch	batch_size	learning_rate	rmse
73	<class 'keras.optimizers.adam.Adam'>	64	32	0.01000	5204.733887
63	<class 'keras.optimizers.sgd.SGD'>	512	64	0.00001	5243.323730
124	<class 'keras.optimizers.adam.Adam'>	512	64	0.10000	5257.530273
121	<class 'keras.optimizers.adam.Adam'>	512	32	0.01000	5274.824219
76	<class 'keras.optimizers.adam.Adam'>	64	64	0.10000	5285.414062

Hyperparameter optimization results, sorted by RMSE score in ascending order.

After 192 iterations and about 1.5 hours of training, the model concluded the below hyperparameters as the best one in terms of the lowest RMSE score of 5204.

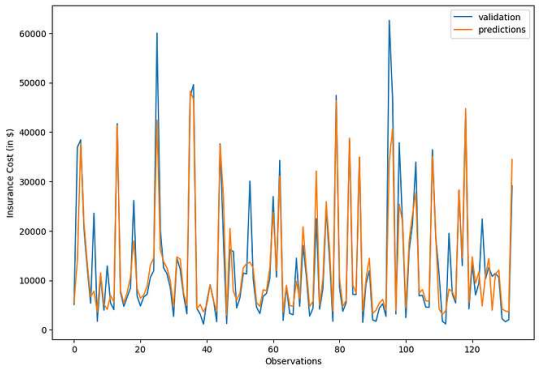
- optimizer : Adam
- epochs: 64
- batch_size: 32
- learning_rate: 1e-2
- number of hidden layers: 3
- number of neurons: 100 (in each layer)

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 batch_normalization_774 (Ba  (None, 11)               44
 tchNormalization)

 dense_774 (Dense)           (None, 100)               1200

 batch_normalization_775 (Ba  (None, 100)              400
 tchNormalization)

 dropout_312 (Dropout)       (None, 100)               0

 dense_775 (Dense)           (None, 100)               10100

 batch_normalization_776 (Ba  (None, 100)              400
 tchNormalization)

 dropout_313 (Dropout)       (None, 100)               0

 dense_776 (Dense)           (None, 100)               10100

 batch_normalization_777 (Ba  (None, 100)              400
 tchNormalization)

 dense_777 (Dense)           (None, 1)                 101

=================================================================
Total params: 22,745
Trainable params: 22,123
Non-trainable params: 622
_________________________________________________________________
```

Model Architecture Summary

The table above displays the count of trainable and non-trainable parameters within our trained model. Trainable parameters encompass weights and biases that can be updated during backpropagation, whereas non-trainable parameters represent the opposite—they are essentially redundant, signifying that these neurons do not contribute to training.

Now, let's move on to the exciting part. It's time to utilize the validation set to assess the predictive performance of our model concerning data that the model has never encountered before!



Predictions with unseen validation set

Et voilà!

Together, we've accomplished a great deal by training a model from scratch that makes fairly accurate predictions at times, albeit with occasional inaccuracies. There's always room for further improvement and fine-tuning!

• • •

Conclusion

Thank you for coming this far and accompanying me throughout this journey. Together, we have learned so much. We have seen the mathematical genius behind feedforward neural networks, how neural networks learn, adjust, update themselves, and actually come closer to the actual parameters.

I always appreciate valuable and constructive criticism; therefore, feel free to elaborate!

If you enjoyed this blog, I appreciate your likes and comments. If you are interested in seeing the rest of this modeling process and the code, I'm sharing a GitHub repo where you will find what you need..

• • •

References

- https://wandb.ai/site/articles/fundamentals-of-neural-networks
- https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/
- https://towardsdatascience.com/deriving-the-backpropagation-equations-from-scratch-part-1-343b300c585a
- https://brilliant.org/wiki/backpropagation/#:~:text=%E2%88%82ajk%E2%80%8B,1%20%CE%B4%20l%20k%20%2B%201%20.
- EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks
- REVISITING SMALL BATCH TRAINING FOR DEEP NEURAL NETWORKS
- http://karpathy.github.io/2019/04/25/recipe/

- Weight Initialization Techniques in Neural Networks by Saurabh Yadav
- A Gentle Introduction To Weight Initialization for Neural Networks by Saurav Maheshkar
-