Machine Learning and Neuronal Networks

Machine learning tutorial

Author:

Cristina Baron Suarez cb24ack@herts.ac.uk

Repository Github

March 26, 2025

Contents

1 Introduction

The purpose of this tutorial is to explain Artificial Neuronal Networks (ANNs), which predict a value y from an input x. We will illustrate these ideas with the Titanic dataset, predicting whether a passenger survives (1) or dies (0) based on their `class`, `sex`, and `age`. Furthermore, we will use 70% of the data for training and 30% for testing to measure prediction accuracy.

2 Data & Architecture

Before training, we scale and encode our data so that no single feature dominates the learning process.

- `pclass`: Normalize using the Z-score Normalization (mean = 0, standard deviation = 1), since the data follow a normal distribution.

$$\text{pclass}_{\text{scaled}} = \frac{\text{pclass} - \mu}{\sigma}$$

$$\mu = \frac{1}{N}\sum_{i=1}^{N} \text{pclass}_i$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\text{pclass}_i - \mu)^2}$$

  Where:

  - $\text{pclass}_{\text{scaled}}$ is the standardized pclass.
  - μ is the mean class of all passengers.
  - σ is the standard deviation of the class.
  - N is the total number of passengers.

  After applying this normalization:

  - Negative values represent classes lower than the mean.
  - Positive values represent classes higher than the mean.
  - A value of 0 represents the exact mean of the original classes.

- `sex`: Represent 'male' and 'female' as numbers (1 and 0, accordingly).
- `age`: Normalize using the Z-score Normalization, as before.

After scaling, we build our neuronal network with three inputs (`pclass`, `sex` and `age`) and one output ($\widehat{\texttt{survived}}$ - which is an estimate of 'survived'). Furthermore, we will use one hidden layer with ten neurons.

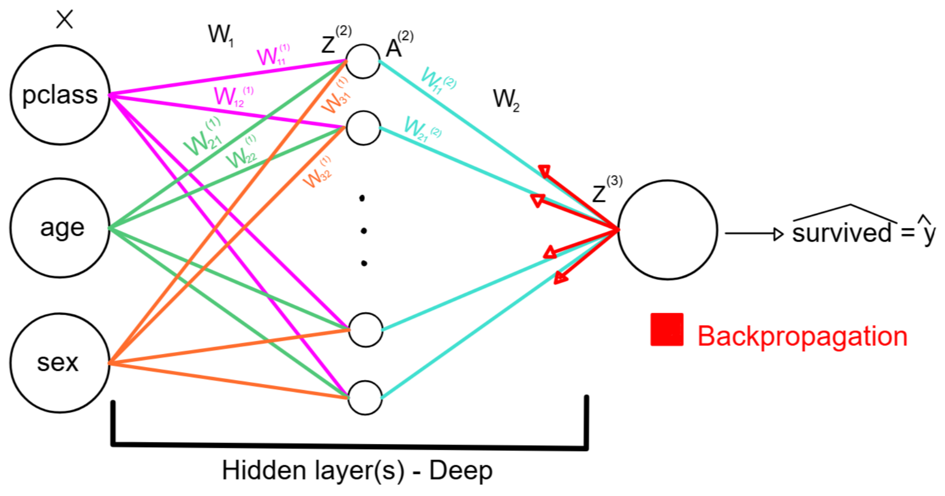

Figure 1: Artificial Neuronal Network

Neurons are represented by circles, while the lines represent synapses. The mission of synapses is to take a value from their input, multiply it by a specific weight, and output the result.

In contrast, neurons add to the output of other synapses and apply an activation function. All of this will be explained in more detail in section 3.

Why do we multiply each feature by a different weight?

Each weight w_i determines how much influence its corresponding feature has on the final prediction of the probability. For our example, each input feature (`pclass`, `sex`, and `age`) has a different impact on survival probability. The weights allow the model to learn how important each feature is in making predictions.

- If the weight for `sex` is high, it means that being female is a strong predictor of survival.
- If the weight for `pclass` is negative, it indicates that passengers in lower classes have a lower probability of survival.
- If the weight for `age` is negative, it suggests that older people are less likely to survive.

3 Forward Propagation

How information moves from the input to the output:

1. Each input is multiplied by a weight, which increases or decreases the importance of that input.
2. All these weighted inputs are added together, plus a *bias*[1].
3. We apply functions to these sums to help the network handle more complicated patterns. For the hidden layer, we will use **ReLU**. For the final output, we will use the **sigmoid** activation function. These will be explained below.

[1]This is a constant term that allows the model to make better predictions. It allows the activation function to shift to the left or right, helping the model make better predictions when the inputs are zero or small.

We will use matrices to process multiple inputs simultaneously.

$$Z^{(2)} = X \cdot W^{(1)} + b_1 \tag{1}$$

Each neuron in the hidden layer calculates a linear combination of the input values.

In the first iteration, we set the weight values, W, and the bias, b, randomly.

Then, each neuron in the hidden layer ($Z^{(2)}$) passes its output through the ReLU activation function (f).

$$ReLU(x) = max(0, x) \tag{2}$$

This function sets any negative input to 0 and leaves positive inputs unchanged. In other words, it only pays attention if the signal is strong enough.

$$A^{(2)} = ReLU(Z^{(2)}) = max(0, Z^{(2)})$$

The resulting matrix, $A^{(2)}$, is the one that will be propagated to the output, $\hat{y}$. Again, this is done by multiplying $A^{(2)}$ by our second-layer weights ($W^{(2)}$) and adding a bias (b_2), but this time applying the sigmoid activation function, which transforms the output into a value between 0 and 1, which represents the probability of belonging to a certain category (in this case, surviving).

$$Z^{(3)} = A^{(2)} W^{(2)} + b_2$$

$$\hat{y} = f(Z^{(3)}) = \frac{1}{1 + e^{-Z^{(3)}}} \tag{3}$$

At this point, we will have our official survival estimate ($\hat{y}$). If the result is greater than 0.5, the network predicts "survive" (1). Otherwise, it guesses "dies" (0).

Linear relationship between Z, A and W

Think about the relationship between Z, A and W as a cooking recipe:

- W is an ingredient (e.g. the amount of flour)
- A is the recipe or proportion (e.g. 2 cups of flour per cake)
- Z is the final result (e.g. the amount of flour that was used)

If W increases (more flour), Z increases too. If A is bigger (more flour is needed per cake), the result increases faster as well.

- W is the weight that indicates the importance of an input.
- A is the activation, that determines how much the synapses contribute.
- Z is the combined result of that weight in the neuron.

4 Gradient Descent

After the network produces an output (e.g. predicting a 70% chance of survival for a certain passenger), we compare it to the correct answer (the e.g. the passenger did not survive). The difference between the prediction and the true result tells us how wrong our network was.

To fix these problems, we use the gradient descent method. It resembles a hiker trying to find the lowest point in a valley:

- **Identify Error**: If the hiker is high up on the mountain (big error), they need to move down.
- **Step in the Right Direction**: The hiker (network) adjusts its steps (weights) so that it moves closer to the bottom (lower error).

With each pass over the training examples, the network takes small steps to reduce its errors, so the predictions gradually get better.

The bias value and the optimal weight values that best fit the data are learned during the training process. In order to know how wrong our model is, we calculate the error by applying the loss function, and we try to minimize it:

$$J = \sum \frac{1}{2}(y - \hat{y})^2 \tag{4}$$

We will change the weights in order to minimize the cost. For that, we need to understand the rate of change of J with respect to W - also known as the derivative. Since we are considering one weight at a time, we need to get the partial derivative.

- If $\frac{\partial J}{\partial W}$ is positive, the cost function is going uphill.
- If $\frac{\partial J}{\partial W}$ is negative, the cost function is going downhill.

To improve our predictions, now that we know in which direction the cost decreases, we can iteratively take steps downhill updating the weights and stop whenever the cost stops getting smaller.

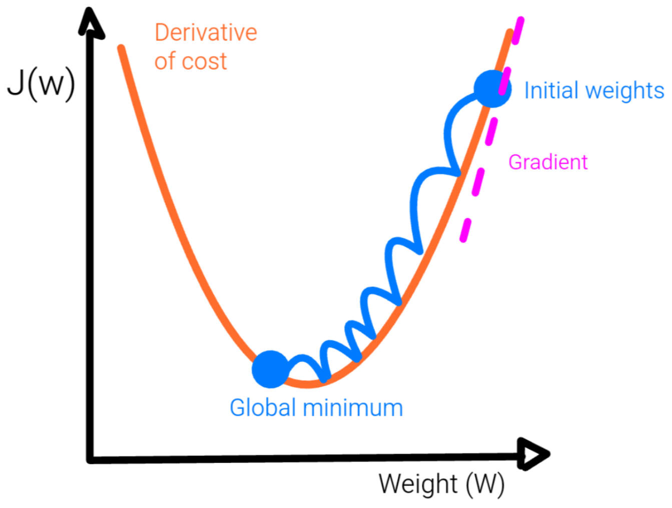

Figure 2: Gradient Descendt

5 Backpropagation

To make these corrections, the network determines which weights contributed the most to the error. We start at the final output, seeing how far off the prediction was. Then, we move layer by layer backward, deciding how each weight in the hidden layers affected that mistake. Lastly, we adjust the weights. The weights that are less responsable change less compared to the weights that are more responsible.

This cycle of passing data forward to get a prediction, checking the error, and passing the error back to update all the connections, repeats over and over until the network's predictions become as accurate as possible.

Since our weights are spread across two matrices, $W^{(1)}$ and $W^{(2)}$, we will compute $\frac{\partial J}{\partial W}$ independently.

In order to know how each weight contributed to the error, we use the `chain rule`. This derivative tells us how changing a specific weight will increase or decrease the error.

- If a weight has a large positive derivative, increasing that weight makes the error worse.
- If a weight has a large negative derivative, the error decreases when we adjust that weight downward.

Since the output of each neuron in one layer serves as the input for the next, the partial derivatives are successively multiplied through the chain rule as they propagate backward. At each step, the network adjusts the error information based on the local weights and activation functions, continuously passing these derivative calculations back to the previous layer.

Backpropagate from the third to the second layer

We start at the output layer:

$$\frac{\partial J}{\partial W^{(2)}} = (y - \hat{y})$$

As we mentioned before, we now apply the `chain rule` [2] to solve the derivative. y is a constant, but $\hat{y}$ changes with respect to $W^{(2)}$:

$$\frac{\partial J}{\partial W^{(2)}} = -(y - \hat{y})(\frac{\partial \hat{y}}{\partial W^{(2)}})$$

Equation 3 tells us that $\hat{y}$ is our activation function of $z^{(3)}$ so we need to apply the chain rule again:

$$\frac{\partial J}{\partial W^{(2)}} = -(y - \hat{y}) f'(z^{(3)}) \frac{\partial z^{(3)}}{\partial W^{(2)}}$$

- To find the rate of change of $\hat{y}$ with respect to $z^{(3)}$ we need to differenciate the sigmoid activation function (equation 3).

$$f'(z) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

[2] If $y = f(u)$ and $u = g(x)$, then: $\frac{dy}{dx} = \frac{dy}{du} \cdot \frac{du}{dx}$.

- $\frac{\partial Z^{(3)}}{\partial W^{(2)}}$ represents the rate of change of Z (formula 3), our third layer activity, with respect to $W^{(2)}$ (the weights in the second layer).
  - Focusing on a single synapse, we see that there is a linear relationship between W and Z, where A is the slope, as explained before. This means that, for each synapse, $\frac{\partial Z}{\partial W^{(2)}}$ is the activation A in that synapse.
  - Since we backpropagate the error to each weight, by multiplying the activity on each synapse, the weights that contribute more to the overall error will have larger propagations, which means that they will have larger $\frac{\partial J}{\partial W^{(2)}}$ values and will change more when gradient descent is performed.

To sum up, we can compute the gradient for updating the weights, W, of a layer (in this case, the second one):

$$\begin{aligned}\frac{\partial J}{\partial W^{(2)}} &= (a^{(2)})^T \delta^{(3)} \\ \delta^{(3)} &= -(y-\hat{y})f'(z^{(3)})\end{aligned} \tag{5}$$

The gradient with respect to each example pulls our gradient descent algorithm in a certain direction. After adding them all, we get the direction in which we have to go to minimize the error.

Backpropagate from the second to the first layer

The logic is the same as before (but using the ReLU activation function instead), and there is still a linear relationship along each synapses, but now we want to know the rate of change of $z^{(3)}$ with respect to $a^{(2)}$. Hence, the slope is equal to the weight value of that synapse.

$$\begin{aligned}\frac{\partial J}{\partial W^{(1)}} &= X^T \delta^{(2)} \\ \delta^{(2)} &= \delta^{(3)}(W^{(2)})^T ReLU'(z^{(2)})\end{aligned} \tag{6}$$

$$\text{ReLU}'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{if } x \leq 0 \end{cases}$$

At $x = 0$, the derivative is undefined so it can be treated as 0 by convention.

6 Optimization Algorithm

Optimization is about finding the best settings or inputs that minimize or reduce the output of a function to its lowest possible value. (e.g. trying to find the lowest point in a valley).

Gradient Descent would be like walking down a hill and adjusting your steps to reach the bottom more efficiently. It can avoid the following problems:

- Not getting stuck at a local minimum.
- Not moving so slowly that we never reach the global minimum.
- Not moving so quickly that we bounce out of the global minimum.

Adam is a technique that automatically adjusts how big each “step” should be when updating the weights. It uses two techniques, that will be explained later, to do this: Momentum and RMSprop. By combining them, Adam helps to find the best settings faster and more effectively than just using the basic gradient descent method.

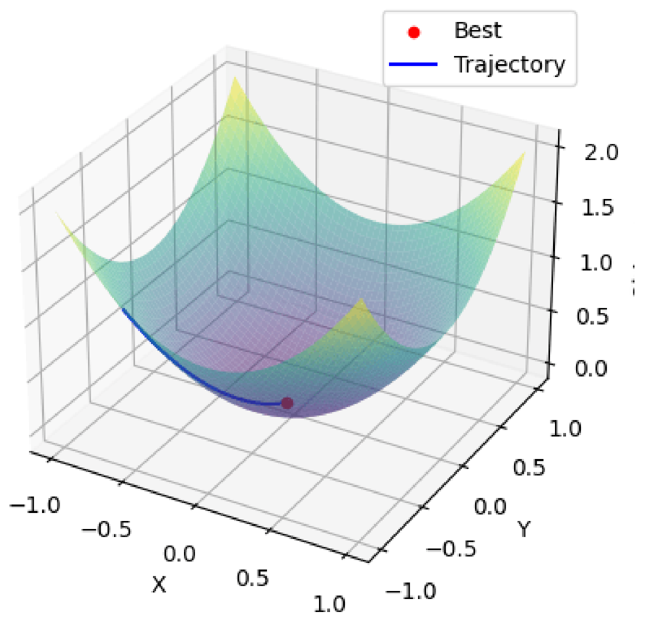

Figure 3: Adam optimization

- If the prediction was very high (e.g. 0.7 instead of 0), the weights see themselves reduced in that direction.
- If the prediction was very low (e.g. 0.2 instead of 1), the weights are adjusted in the opposite direction.

Adam improves this process by dynamically adjusting the learning rate, making sure that the changes are small as we approach the minimum error.

Momentum

This keeps track of the direction you have been moving in. If you have been consistently moving in one direction, momentum will help you keep going that way, making your descent faster. Since it builds on what happened in previous steps, even if we hit a ’small bump’, the algorithm will not make drastic direction changes, it follows the overall trend. It will keep moving smoothly toward better answers.

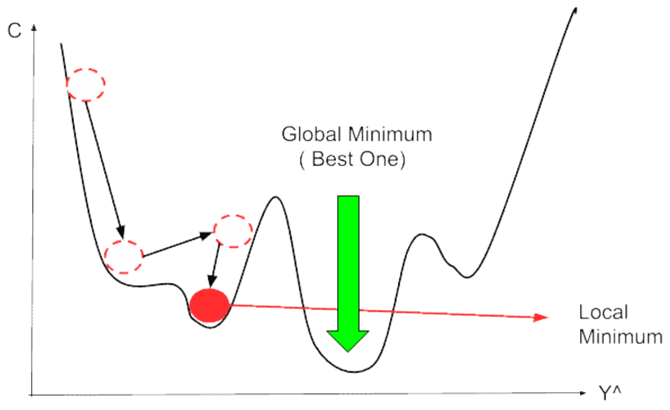

Figure 4: Global minimum

$$m_t = \beta_1 m_{t-1} + (1 + \beta_1)(\frac{\partial J}{\partial W_t}) \tag{7}$$

Where:

- β_1 the momentum parameter and it is usually around 0.9
- m_t starts at zero.

RMSprop *(Root Mean Square Propagation)*

This adjusts your steps based on recent changes. If the slope has been very steep, it makes your steps smaller to prevent you from overshooting the lowest point. In other words, it dynamically adjusts the learning rate to improve the stability and speed of the training.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)(\frac{\partial J}{\partial W}_t)^2 \tag{8}$$

Where:

- v_t is the exponentially weighted average of the squared gradients. It starts at zero.
- β_2 is usually around 0.999

Adam Optimizer

Since m_t and v_t start at zero, they tend to be biased toward zero, especially during the initial steps. To correct this, Adam computes the bias-corrected estimates:

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

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

Finally, we update the weights:

$$w_{t+1} = w_t - \frac{\hat{m_t}}{\sqrt{\hat{v_t} + \epsilon}}\alpha$$

Where:

- α is the learning rate. 0.001 by default.
- ϵ is a small positive constant, like 10^{-8}, used to avoid division by zero.

7 Conclusion

After several iterations passing data forward, checking the difference between predictions and reality, passing that difference backward to adjust connections, and using an optimization strategy (Adam) to make learning smoother, the neural network will gradually learn the best way to combine its input features in order to predict who survives the Titanic sinking.

8 References

References

[1] Youtube, *Neural Networks Demystified [Part 1: Data and Architecture]*. https://youtu.be/bxe2T-V8XRs?si=tztJxfPkmvdJUSvA

[2] Youtube, *Neural Networks Demystified [Part 2: Forward Propagation]*. https://youtu.be/UJwK6jAStmg?si=G-uCjtmB1rblumRu

[3] Youtube, *Neural Networks Demystified [Part 3: Gradient Descent]*. https://youtu.be/5u0jaA3qAGk?si=l2rvJpv_nSJu5YPe

[4] Youtube, *Neural Networks Demystified [Part 4: Backpropagation]*. https://youtu.be/GlcnxUlrtek?si=IzgfujG0s4z-MqUV

[5] Youtube, *Neural Networks Demystified [Part 6: Training]*. https://youtu.be/9KM9Td6RVgQ?si=37av7BWHAd3zGNv9

[6] GeeksforGeeks, *What is Adam Optimizer?*. https://www.geeksforgeeks.org/adam-optimizer/

[7] Guru99, *Back Propagation in Neural Network: Machine Learning Algorithm*. https://www.guru99.com/backpropogation-neural-network.html

[8] DeepChecks, *Rectified Linear Unit (ReLU)*. https://www.deepchecks.com/glossary/rectified-linear-unit-relu/

[9] GeeksforGeeks, *How to Implement Adam Gradient Descent from Scratch using Python?*. https://www.geeksforgeeks.org/how-to-implement-adam-gradient-descent-from-scratch-using-python/