Mathematical Aspect of Deep Learning

ACHRAF KHABAR

July 12, 2023

1 Feed Forward Neural Network

A neural network can be represented as a function $f_\theta : x \longrightarrow y$. Its purpose is to convert the input signal $x \in \mathbb{R}^n$ to the output signal $y \in \mathbb{R}^m$, where $\theta \in \mathbb{R}^p$ represents the parameters that determine its behavior. For example, a simple example of f_θ would be $y = f_\theta(x) = \theta \cdot x$.

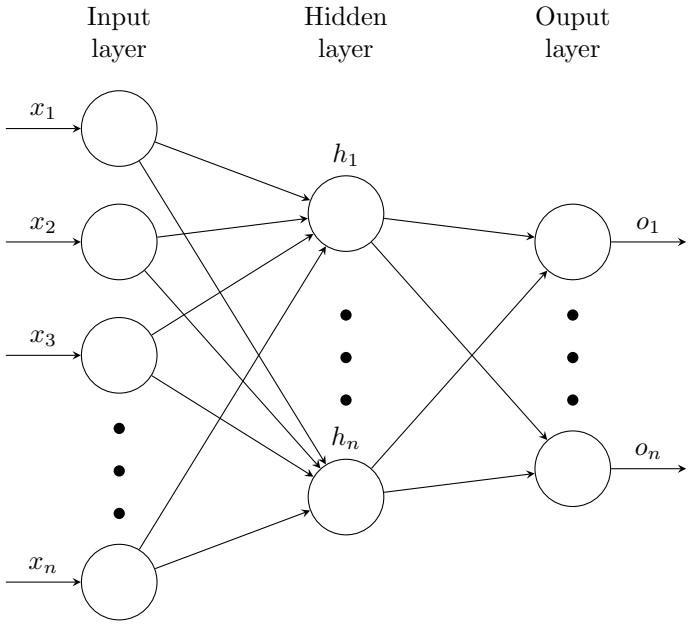

figure 1 : Feed Forward Neural Network Architecture

2 Expressing a Neural Network in Vector Form

If we assume that the dimensionality of the input is $x \in \mathbb{R}^n$ and the first layer has p_1 units. the unit has $w \in \mathbb{R}^n$ weights associated with it. That is, the weights associated with the first layer are a matrix of the form $W_1 \in \mathbb{R}^{n \times p_1}$ and each p_1 unit also has a bias term associated with it .

The first layer produces an output $o_1 \in \mathbb{R}^{p_1}$ where $o_i = f(\sum_{k=1}^{n} x_k.w_k + b_i)$ .Note that the index k corresponds to each of the **inputs/weights** (going from $1 \cdots n$), and the index i corresponds to the units in the first layer (going from $1 \cdots p_1$).

Let's now look at the output of first layer in a vectorized notation. By vectorized notation, we simply mean linear algebraic operations, such as vector matrix multiplications and computation of the activation function on a vector producing a vector (rather than scalar to scalar). The output of the first layer can be represented as $f(x.w_1 + b_1)$.

Here, we are treating the input $x \in \mathbb{R}^n$ to be of dimensionality $1 \times n$, the weight matrix w_1 to be of dimensionality $n \times p_1$, and the bias term to be a vector of dimensionality $1 \times p_1$. Notice, then, that $x \times w_1 + b$ produces a vector of dimensionality $1 \times p_1$, and the function f simply transforms each element of the vector to produce $o_1 \in \mathbb{R}^{p_1}$.

A similar process follows for the second layer that goes from $o_1 \in \mathbb{R}^{p_1}$ to $o_2 \in \mathbb{R}^{p_2}$. This can be written in vectorized form as $f(o_1 \cdot w_2 + b_2)$ .We can also write the entire computation up to layer 2 in vectorized form as $f(f(x \cdot w_1 + b_1)w_2 + b_2)$.

3 Evaluating the Output of a Neural Network

For a single data point, we can compute the output of a neural network, which we denote as $\widehat{y}$. Now we need to compute how good the prediction of our neural network $\widehat{y}$ is as compared to y. Here comes the notion of a loss function. A loss function measures the disagreement between $\widehat{y}$ and y, which we denote by l.

4 Cost Functions

4.1 Binary Cross-Entropy

Let's consider a simple example to understand the concept of binary cross entropy and also get a fundamental intuition of maximum likelihood.

We have some data consisting of $D = \{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$, where $x \in \mathbb{R}^n$ and $y \in \{0, 1\}$ which is the target of interest also known as the criterion variable. Without dimenstartions the final value of $P(D|\theta)$:

$$P(\theta) = -\sum_{i=1}^{n} y_i \cdot log\, f(x_i, \theta) \cdot log\, (1 - f(x_i, \theta)) \tag{1}$$

is the recommended loss function for binary classification. This loss function should typically be used when the neural network is designed to predict the probability of the outcome. In such cases, the output layer has a single unit with a suitable sigmoid as the activation function.

4.2 Cross-Entropy

Consider deriving the cross-entropy loss function to be used in the context of multiclassification, Let's assume that $y \in \{0, 1, \cdots, k\}$, where $\{0, 1, \cdots, k\}$, are the classes. We also denote $n_1, n_k, \cdots n_k$ to be the observed counts of each of the k classe. Observe that $\sum_{i=1}^{k} n_i = n$. without any demonstrations we have $P(D|\theta)$:

$$P(\theta) = -\sum_{i=1}^{n} y_i \cdot log\, f(x_i, \theta) \tag{2}$$

Is the recommended loss function for multiclassification. This losfunction shouldld typically be used with the neural network designed to predict the probability of the outcomes of each of the classes. In such cases, the output layer has softmax units (one for each class).

4.3 Squared Error

Let us now discuss deriving the squared error to be used in the context of regression using maximum likelihood. Let us assume that $y \in \mathbb{R}$. Unlike the previous cases, where we assumed that we had a model that predicted a probability, we will assume that we have a model that predicts the value of y . To apply the maximum likelihood idea, we assume that the difference between the actual y and the predicted $\widehat{y}$ has a Gaussian distribution with a zero mean and a variance of σ^2 . Then, it can be shown that minimizing :

$$\sum_{i=1}^{n} (y - \hat{y})^2 \tag{3}$$

should be used for regression problems. The output layer in this case will have a single unit.

5 Types of Activation Functions

5.1 Linear Unit

The linear unit is simplest unit that transforms the input as $y = w \cdot x + b$. As the name indicates, the unit does not have a non-linear behavior and is typically used to generate the mean of a conditional **Gaussian distribution**.

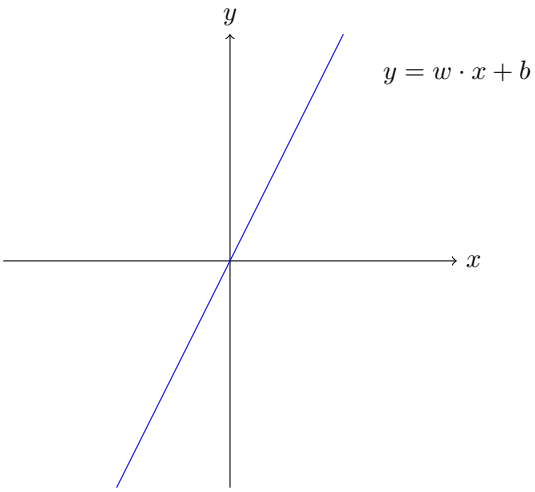

figure 2 : Linear unit ploting

5.2 Sigmoid Activation

The sigmoid activation transforms the input as follows:

$$y = \frac{1}{1 + e^{-(w \cdot x + b)}} \tag{4}$$

Sigmoid units can be used in the output layer in conjunction with binary cross-entropy for binary classification problems. The output of this unit can model a Bernoulli distribution over the output y conditioned over x.

5.3 Softmax Activation

The softmax layer is typically used only within the output layer for multiclassification tasks in conjunction with the cross-entropy loss function. Typically, the units of the previous layer model an unnormalized score of how likely the input is to belong to a particular class. The softmax layer normalized this so that the output represents the probability for every class.

5.4 Rectified Linear Unit

A rectified linear unit (ReLU) used in conjunction with a linear transformation transforms the input as :

$$f(x) = max(0,\ w \cdot x + b) \tag{5}$$

The underlying activation function is $f(x) = max(0, x)$. Recently, the ReLU is more commonly used as a hidden unit. Results show that ReLUs lead to large and consistent gradients, which helps gradient-based learning.

There are a few disadvantages with ReLU, however. When inputs approach near zero, the gradient of the function becomes zero and thus gets stuck within the training steps with no progress in the training. This is commonly known as the dying ReLU problem.

5.5 Hyperbolic Tangent

The Hyperbolic Tangent unit transforms the input (used in conjunction with a linear transformation) as follows:

$$y = \tanh(w \cdot x + b) \tag{6}$$

The underlying activation function is given by :

$$f(x) = \tanh(x) \tag{7}$$

6 Backpropagation

We investigate the loss function, which measures the discrepancy between predicted and actual outcomes. First, the weights of the network are initialized randomly. For the network to learn, the next logical step is to adjust the weights so that the divergence is minimal (ideally zero). This step is implemented with the backpropagation algorithm, which uses the chain rule to calculate the slope of the loss with respect to the weights.

In the forward pass, the network computes the prediction for a given input sample, and the loss function measures the disagreement between the actual target value and network's prediction. Backpropagation computes the gradient of the loss with respect to the weights and biases and thus provides us with a fair overall picture of how a small change in the weight impacts the overall loss. We would then need to update the weights iteratively and with small increments (in the opposite direction of the gradient) to reach the local minima. This process is called the gradient descent—i.e., reducing the loss function to reach the minimum. The network therefore learns (iterative and incremental updates on weights) the patterns that can correctly predict for a given input sample with the least disagreement.

There are several variants to update the weights in gradient descent for neural networks. The next section explores a few of them. In the next chapter, we will take a brief look at automatic differentiation that enables the idea of backpropagation programmatically.

7 Gradient Descent Variants

7.1 Batch Gradient Descent

The original gradient descent is referred to as the batch gradient descent (BGD) technique. The name is derived from the amount of data used to compute the gradient—in this case, the entire batch. The BGD technique essentially leverages the entire dataset available to compute the gradient of the cost function with respect to the parameters (weights). This results in inherently slow and, in most cases, a non-viable option, as we might run out of memory to load the entire batch. In most common scenarios, we would mostly tend to avoid the BGD approach, sparring small datasets (which is a rare phenomenon in deep learning).

7.2 Stochastic Gradient Descent

To overcome the issues from BGD, we have stochastic gradient descent (SGD). With SGD, we compute the gradient and update the weights for each sample in the dataset. This process results in far less use of memory in the deep learning hardware and achieves results faster. However, the updates are far more frequent than desired. With more frequent updates to the weights, the cost function fluctuates heavily.

SGD, however, results in bigger problems when the goal is to converge the updates towards the exact minima. Given the far more frequent updates, the possibility of overshooting an update is very high. To overcome these tradeoffs, we might need to slowly reduce the learning rate over a period of time in order to help the network converge to local or global minima.

7.3 Mini-Batch Gradient Descent

Mini-batch gradient descent (MBGD) combines the best of SGD and BGD. Instead of using the entire dataset (batch) or just a single sample from the dataset to compute the gradient of the cost function with respect to the parameters, MBGD leverages a smaller batch, which is greater than 1 but smaller than the entire dataset. Common batch sizes are 16/32/64/... 1024, etc. A number in the range of powers of 2 is recommended (but not necessary), as it suits best from a computation perspective .

With MBGD, the updates are less frequent than SGD but more frequent than BGD, and leverage a small batch instead of individual samples or the entire dataset. In this way, the variance reduces to a greater extent and we achieve a better trade-off on the speed.

7.4 Gradient Descent with Momentum

The problems we discussed earlier between SGD and BGD are fairly smoothed using MBGD. However, even with the use of MBGD, the direction of the update still fluctuates (though less than with SGD but more than with MGD). Gradient descent with momentum leverages the past gradients to calculate an exponentially weighted average of the gradients to further smoothen the parameter updates.

7.5 RMSprop

RMSprop is an unpublished optimization algorithm proposed by Geoffry Hinton in lecture 6 of the online course "**Neural Networks for Machine Learning**" on Coursera. At the core, RMSprop computes the moving average of the squared gradients for each weight and divides the gradient by the square root of the mean square. This complex process should help in decoding the name **root-mean-square prop**. Leveraging exponential average here helps in giving recent updates more preferences than less recent ones. For each weight w, we have :

$$v_i = \beta \cdot v_{t-1} + (1 - \beta) \cdot g_i^2 \tag{8}$$

and

$$\Delta w_t = -\frac{\eta}{\sqrt{v_t + \epsilon}} \cdot g_t \tag{9}$$

To update the weight

$$w_{t+1} = w_t + \Delta w_t \tag{10}$$

where η is a hyperparameter that defines the initial learning rate, and g_t is the gradient at time t for a parameter/weight w in θ. We add ϵ to the denominator to avoid divide by zero situations.

7.6 Adam

A simplified name for adaptive moment estimation, Adam is the most popular choice recently for optimizers in deep learning. In a simple way, Adam combines the best of RMSprop and stochastic gradient descent with momentum. From RMSprop, it borrows the idea of using squared gradients to scale the learning rate, and it takes the idea of moving averages of the gradient instead of directly using the gradient when compared to SGD with momentum.

for each weight w :

$$v_i = \beta_1 \cdot v_{t-1} + (1 - \beta_1) \cdot g_t \tag{11}$$

and

$$S_t = \beta_2 \cdot s_{t-1} + (1 - \beta_2) \cdot g_t^2 \tag{12}$$

which then is used to compute

$$\Delta w_t = -\eta \cdot \frac{v_t}{\sqrt{S_t + \epsilon}} \cdot g_t^2 \tag{13}$$

And, finally, the weight is updated as

$$w_{t+1} = w_t + \Delta w_t \tag{14}$$

8 Performance Metrics

8.1 Classification Metrics

The model development process typically starts by formulating a clear problem definition. This basically involves defining the input and the output of the model and the impact (usefulness) such a model can deliver. An example of such a problem definition is the categorization of product images into product categories—the input to such a model being product images and the output being product categories. Such a model might aid the automated categorization of products in an ecommerce or online marketplace setting.

Having defined the problem definition, the next task is to define the performance metrics. The key purpose of performance metrics is to tell us how well our model is doing. A simple metric of performance may be accuracy (or, equivalently, the error), which simply measures the disagreements between the expected output and the output produced by the model. Accuracy, however, can be a poor measure of performance. The two main reasons are class imbalance and unequal misclassification costs. Let's look at the class imbalance problem with an example. As a sub problem of the problem in our previous example of product classification, consider the case of distinguishing between mobile phones and their accessories. The number of examples for classes of mobile phones is a lot smaller that the classes of mobile phone accessories. If, for example, 95of the examples are mobile phone accessories and 5an accuracy of 95class. Thus, accuracy is a poor choice of a metric in this example.

Let's now understand the problem of unequal misclassification costs, again by considering an example related to the problem of product classification. Consider the error associated with categorizing food products that are allergen-free (not containing the eight top allergens— namely, milk, eggs, fish, crustacean shellfish, tree nuts, peanuts, wheat, and soybean) versus the rest (non-allergen-free). From a buyer's point of view, as well as a business point of view, the error associated with categorizing a non-allergen-free product as an allergen-free product is significantly more as compared to categorizing an allergen-free product as a non-allergen-free product. Accuracy does not capture this and hence would be a poor choice in this case.

Precision and recall are often visualized using a PR curve, which plots precision on the Y axis and recall on the X axis (see Figure 5-2). Different values of precision and recall can be obtained by varying the decision threshold on the score or the probability the model produces—for instance, 0 implying class A, and 1 implying class B, with a higher value on one side indicating a particular class. This curve can be used to trade off precision for recall by varying the threshold.

The *F-score* defined as $\frac{2 \cdot p \cdot r}{p+r}$, where p denotes precision and r denotes recall, can be used to summerize the PR curve.

8.2 Regression Metrics

Performance metrics for regression are fairly straightforward when compared to metrics for classification. The most common metric that can be universally applied to most use cases is the mean squared error (MSE). Depending on the use case, a few other metrics could be used for more favorable outcomes. Consider the problem of predicting the monthly sales for a given store, where store sales could range from $5,000$ to $50,000$ across months.

8.3 Mean Squared Error

Mathematically, we can define MSE as :

$$MSE = \frac{1}{n} \sum_{i=0}^{n} (y_i - \hat{y}_i)^2 \tag{15}$$

and

$$RMSE = \sqrt{\frac{\sum_{i=0}^{n} (y_i - \hat{y}_i)^2}{n}} \tag{16}$$

8.4 Mean Absolute Error

The mean absolute error (MAE) computes the mean of the absolute difference between predictions and target. The outcome, which is always positive, is a much more interpretable performance metric than MSE for regression use cases. The lower the MAE for a model the better the performance :

$$MAE = \frac{1}{n}\sum_{i=0}^{n}|y_i - \hat{y_i}| \tag{17}$$

8.5 Mean Absolute Percentage Error

The mean absolute percentage error (MAPE) is the percentage equivalent of the MAE. Given its relative nature, it is by far the most interpretable performance metric for regression. The lower the MAPE for a model, the better the performance for the model :

$$MAPE = \frac{1}{n}\sum_{i=0}^{n}\frac{|y_i - \hat{y_i}|}{y_i} \tag{18}$$

9 Model Capacity

We have data of the form $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$, and our task is to generate a computational procedure that implements the function $f : x \longrightarrow y$. We measure performance over this task as the root mean squared error (RMSE) over unseen data, as follows:

$$E(f, D, U) = \left(\frac{\sum_{(x_i,y_i)\in U}(y_i - f(x_i))^2}{|U|}\right)^{\frac{1}{2}} \tag{19}$$

Given a dataset of the form $D = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, where $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$, , we use the least squares model, which takes the form $y = \beta \cdot x$, where β is a vector such that $\|X \cdot \beta - y\|_2^2$ is minimized. Here, X is a matrix where each row is an x . The value of β can be derived using the closed form :

$$\beta = (X^T \cdot X)^{-1} \cdot X^T \cdot y \tag{20}$$

We can transform x to be a vector of values $[x_0, x_1, x_2]$. That is, if $x = 2$, it will be transformed to $[1, 2, 4]$. Following this transformation, we can generate a least squares model β using the preceding formula. Under the hood, we are approximating the given data with a second order polynomial ($degree = 2$) equation, and the least squares algorithm is simply curve fitting or generating the coefficients for each of $[x_0, x_1, x_2]$.

10 Regularizing the Model

From the previous example, is easy to see that while fitting models, a central problem is to get the capacity of the model exactly right so that one neither overfits nor underfits the data. Regularization can be simply seen as any modification to the model (or its training process) that intends to improve the error on the unseen data (at the cost of the error on the training data) by systematically limiting the capacity of the model. This of process systematically limiting or regulating the capacity of the model is guided by a portion of the labelled data that is not used of training. This data is commonly referred to as the validation set.

In our running example, a regularized version of least squares takes the form $y = \beta \cdot x$, where β is a vector such that $\|X \cdot \beta - y\|_2^2 + \lambda \cdot \|\beta\|_2^2$ is minimized, and λ is a user-defined parameter that controls the complexity. Here, by introducing the term $\lambda \|\beta\|_2^2$ we are penalizing models with extra capacity. To see why this is the case, consider fitting a least squares model using a polynomial of degree 10, but the values in the vector β have 8 zeros and 2 non-zeros. As opposed to this, consider the case where all values in the vector β are non-zeros. For all practical purposes, the former model is a model with degree = 2 and a lower value of $\lambda \cdot \|\beta\|_2^2$.