Vanishing Gradient and Exploding Gradient

1.Fay Elhassan 2. Vongai Mitchell 3. Bahati Kilongo 4. Solafa Fadlallah

April 19, 2023

Abstract

Vanishing and Exploding gradients are common problems that occur during the training of deep neural networks, particularly those with many layers. The issue arises when gradients become too small or too large, making it difficult for the network to converge to an optimal solution. In this report, we provide an overview of these issues, their impact on the process of model training, and common solutions used to resolve them.

1 Introduction

Neural network models are trained by the optimization algorithm of gradient descent. The input training data helps these models learn, and the loss function gauges how accurate the prediction performance is for each iteration when parameters get updated. As training goes on, the goal is to reduce the loss function/prediction error by adjusting the parameters attractively. Specifically, the gradient descent algorithm has a forward step and a backward step, which lets it do this. During backward propagation, two issues could happen: the derivative term gets extremely small, i.e., approaches zero vs. this term gets extremely large and overflows. These issues are referred to as the Vanishing and Exploding Gradients, respectively.

In 2 we covered the neural network theory by using a simple neural network as an illustration, A special accent is made on forward propagation which is the key concept for understanding vanishing and exploding gradients covered in 3 with their causes, identification and its impact on the training process of the model. Section 4.2 presents the different techniques used to resolve the vanishing and exploding gradient issues, by making a particular accent in 4.3 on different Architectures that are designed to deal with Vanishing and Exploding Gradient.

2 Neural Network

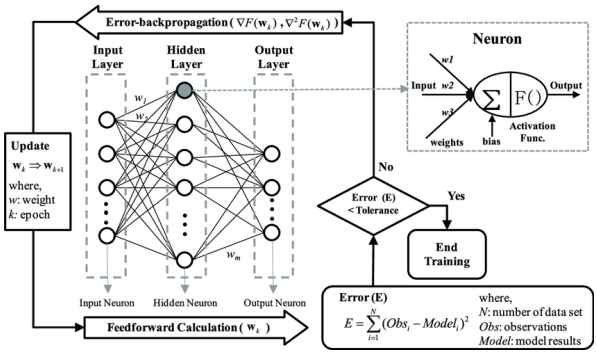

Figure 1: Simple Neural Network: Research Gate

"A neural network is a type of machine learning model that consists of three main layers: the input layer, hidden layer(s), and output layer. It uses activation functions to process input data and transform it into predictions. During training, the model starts with randomly initialized weights and biases and then uses the gradient descent optimization algorithm along with the backpropagation algorithm to update these parameters in order to minimize the error between the predicted output and the actual output. The gradients of the loss function with respect to the weights and biases are computed during backpropagation, and the weights and biases are updated in the direction of the negative gradient using the learning rate hyperparameter."

Forward Propagation

In forward propagation, input vectors/data move forward through the network using a formula to compute each neuron in the next layer. The formula consists of input/output, activation function f, weight W and bias b:

$$A^{(l)} = f(W^{(l)}A^{(l-1)} + b^{(l)})$$

where:

- $A^{(l)}$ is the output of the neuron in layer l,
- f is the activation function applied element-wise to the output of the neuron,
- $W^{(l)}$ is the weight matrix for layer l,
- $A^{(l-1)}$ is the output of the previous layer (input to layer l),
- $b^{(l)}$ is the bias vector for layer l.

We then compute the loss function which is simply the sum of the errors committed at the predicted output for the different rows and is given by the following formula (for logistic regression):

$$Loss = \frac{1}{m}\sum_{i=1}^{m} L(y^{(i)}, \hat{y}^{(i)})$$

where:

- $Loss$ is the loss function,
- m is the number of training examples,
- $y^{(i)}$ is the true label for the ith training example,
- $\hat{y}^{(i)}$ is the predicted output for the ith training example.

Back Propagation

It consists of computing gradients of the cost function w.r.t the different parameters; then applying a gradient descent algorithm e.g. SGD to update the parameters such that we minimize our cost function:

$$W_{new} = W_{old} - \alpha(\nabla L_W)$$

where, α represents the learning rate.

Back Propagation Challenges

Two major problems arise in backpropagation, particularly for larger or deeper networks and these are vanishing gradient and exploding gradient.

3 Vanishing and Exploding Gradient

Since backpropagation utilises the chain rule to compute gradients from output later towards the input layer, gradients frequently become SMALLER and approach zero eventually leaving weights or lower layers nearly unchanged and thus gradient descent algorithm never converges to the optimal solution. This is called the problem of *vanishing gradient*.

In other cases, BIG gradients can accumulate as the backpropagation algorithm progresses and this leads to the point where large parameter updates are observed. This in turn causes gradient descent to diverge instead of converging. The parameters can become so large that they overflow and return NaN values that cannot be updated anymore. This is called the problem of *exploding gradient*.

3.1 Identification of vanishing and exploding gradient

The vanishing gradient is identified by:

- Parameters of higher layers change significantly whilst those of lower layers do not change much (or might not change at all).
- In some cases, model weights may become zero during training.
- The model learns slowly and training might even stagnate at an early stage after only a few iterations.
- Model performance is poor.

While exploding gradient is identified by:

- Parameters experience exponential growth.

- In some cases, model weights may become NaN very quickly during training.
- The model loss can become NaN and the model experiences avalanche learning.

3.2 Causes of vanishing/exploding gradient

The derivatives of certain activation functions, like sigmoid, are close to zero; this could cause the vanishing gradient issue. In other cases, initial weights assigned to the neural networks can generate small/large losses. very small/large gradient values can accumulate to the point where small/large parameter updates are observed, causing a vanishing/exploding gradient.

4 Solution to vanishing and exploding issues

Some architectures and techniques deal with both vanishing and exploding gradient issues, other techniques deal only with vanishing gradient issues and others deal only with exploding gradient issues.

4.1 Solutions to Vanishing Gradient Challenges

How to solve vanishing gradient issues:

- Change activation function:
  Certain activation functions, such as ReLU and its variants, can help alleviate the vanishing gradient problem.
- Reducing model complexity:
  Reduce for example layers or neurons by using the dropout technique this technique randomly drops out neurons during training, which can help prevent overfitting and improve gradient flow.
- Weight initialization:
  Use for example Xavier's initialization which works better for layers with the sigmoid activation function, or Heuristic initialization which works better with the Relu activation function.
- Utilizing a better optimizer with a well-tuned learning rate
- batch normalization:
  This technique normalizes the activations of the previous layer.

4.2 Solutions to Exploding gradient Issues

How to solve exploding gradient issues:

- Proper weight initialization techniques such as Xavier's initialization.
- L2 norm regularization:
  Like a regularization method, it will control the complexity of the method by reducing the impact of features that are not useful for the prediction.
- Gradient clipping:
  This technique limits the magnitude of the gradients.

4.3 Architectures that are designed to deal with Vanishing and Exploding Gradient

4.3.1 Residual Neural Networks (ResNets)

ResNets introduce a direct connection which allows information to bypass or skip one or more layers in the networks. This connection is called a 'skip connection' and is the core of residual blocks. Vanishing and exploding gradients can be avoided as the gradients can flow directly from the output layer to the input layer without being multiplied by some of the intermediate layers and gradients are effectively constrained by the skip connections to be within a certain range.

$$y_l = x_l + F(x_l, W_l) \quad \text{(Residual connection)}$$
$$x_{l+1} = f(y_l) \quad \text{(Activation function)}$$

where:

- x_l is the input to the l-th layer of the ResNet.
- y_l is the output of the skip connection, which is the sum of the input x_l and the residual function $F(x_l, W_l)$.

- $F(x_l, W_l)$ is the residual function, which is typically implemented as a convolutional or fully connected layer with weights W_l.
- $f(\cdot)$ is the activation function, such as ReLU (Rectified Linear Unit) or any other activation function used in the network.
- x_{l+1} is the output of the l-th layer after passing through the activation function.

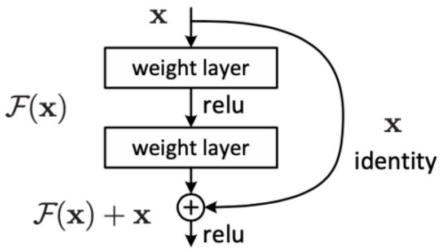

Figure 2: ResNet: Deep Residual Learning

Recurrent Neural Networks (RNN)

Recurrent Neural Networks (RNNs) are a type of neural network architecture that are designed to handle sequential data by allowing information to persist across time steps. They do this by maintaining a "hidden state" that is updated at each time step and passed on to the next time step. This allows the network to capture temporal dependencies in the data.

One of the problems with training deep RNNs, however, is the vanishing or exploding gradient problem. The gradients can become very small or very large as they propagate through time, making it difficult to train the network effectively. To address this, various modifications to the basic RNN architecture have been proposed, such as the Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures, which are better able to handle long-term dependencies and mitigate the vanishing/exploding gradient problem.

4.3.2 Long-Short Term Memory (LSTM)

In traditional RNN, due to the same weights being used across all time steps, as the signal is repeatedly multiplied by these weights, the gradient can eventually vanish or explode.

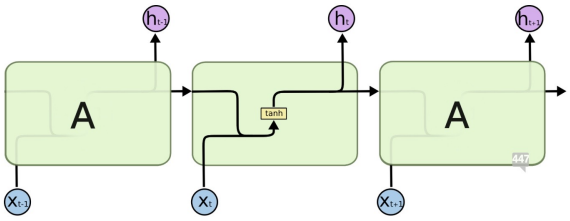

Figure 3: Repeating module in standard RNN:ResearchGate

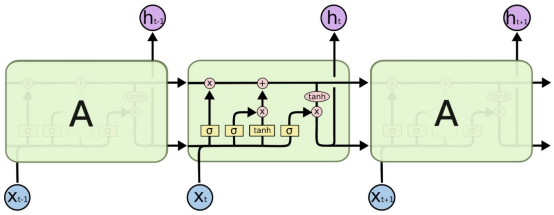

Figure 4: Repeating module in LSTM Research Gate

LSTM is a special type of RNN that can be used in modelling long-term dependencies of sequences. LSTM introduces gating mechanisms that control the flow of information between time steps by selectively discarding or storing information thereby allowing gradient signal to flow more easily through time. LSTM allows the model to effectively learn long-term dependencies in data and thus reducing the impact of the vanishing gradient problem. Plain LSTM has an input gate: that controls the storage of inputs; a forget gate: that controls the forgetting of the former cell state and an output gate: that controls the

cell output from the current cell state

$$
\begin{aligned}
i_t &= \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i) \quad \text{(Input gate)} \\
f_t &= \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f) \quad \text{(Forget gate)} \\
o_t &= \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o) \quad \text{(Output gate)} \\
g_t &= \tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g) \quad \text{(Cell input)} \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot g_t \quad \text{(Cell state)} \\
h_t &= o_t \cdot \tanh(c_t) \quad \text{(Hidden state)}
\end{aligned}
$$

where:

- σ is the sigmoid activation function
- tanh is the hyperbolic tangent activation function
- W_{xi}, W_{hi}, b_i are the weight matrix and biases for the input gate
- W_{xf}, W_{hf}, b_f are the weight matrix and biases for the forget gate
- W_{xo}, W_{ho}, b_o are the weight matrix and biases for the output gate
- W_{xg}, W_{hg}, b_g are the weight matrix and biases for the cell input
- x_t is the input at time step t

4.3.3 Gated Recurrent Unit (GRU)

Like the LSTM, it also utilizes gating mechanisms to control the flow of information. However, it makes use of fewer parameters than LSTM, making it less computationally expensive to train. GRU uses two gates: a reset gate to determine how much of previous data should be discarded and an update gate to determine how much of current input should be utilised or incorporated into the current hidden layer. Because they are selectively storing information, it makes them effective at modelling long-term dependencies in sequential data and thus can be able to address the issue of the vanishing gradient problem.

$$
\begin{aligned}
r_t &= \sigma(W_r \cdot [h_{t-1}, x_t] + b_r) \\
z_t &= \sigma(W_z \cdot [h_{t-1}, x_t] + b_z) \\
\tilde{h}_t &= \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h) \\
h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t
\end{aligned}
$$

where:

- r_t is the reset gate at time step t, which determines how much of the previous hidden state h_{t-1} should be discarded.
- z_t is the update gate at time step t, which determines how much of the current input x_t should be incorporated into the current hidden state h_t.
- $\tilde{h}t$ is the new memory cell at time step t, which is calculated using the reset gate r_t, the previous hidden state $ht - 1$, and the current input x_t.
- h_t is the hidden state at time step t, which is calculated by combining the current hidden state $(1 - z_t) \odot h_{t-1}$ and the new memory cell $z_t \odot \tilde{h}_t$.

5 Python Code

The code aims to showcase the vanishing and exploding gradient phenomena and how to address these challenges. Click here to visit the notebook.

6 Conclusion

In this report, we dive into the phenomena of vanishing and exploding gradient that occurs during the model training process which causes the neural network to have unstable behaviour. To overcome this issue, several solution approaches have been introduced including the implementation of residual neural networks and long-short term memory.

References

[Glorot and Bengio, 2010] Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In Teh, Y. W. and Titterington, M., editors, *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, volume 9 of *Proceedings of Machine Learning Research*, pages 249–256, Chia Laguna Resort, Sardinia, Italy. PMLR.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition.

[Kerem SARI, 2022] Kerem SARI, İsmail Hakkı İPEK, I. S. (2022). Tradenic: Trading-specific smartnic design for low latency and high throughput algorithmic trading.

[Kumar, 2017] Kumar, S. K. (2017). On weight initialization in deep neural networks.

[Tatsunami and Taki, 2023] Tatsunami, Y. and Taki, M. (2023). Sequencer: Deep lstm for image classification.

[Web1, 2023] Web1 (Accessed April 2023). Vanishing/exploding. Webots,https://www.comet.com/site/vanishing-exploding-gradients-in-deep-neural-networks/?cn-reloaded=1.

[Web2, 2023] Web2 (Accessed April 2023). Neural network. Webots,https://www.researchgate.net/figure/Schematic-diagram-of-backpropagation-training-algorithm-and-typical-neuron-model_fig2_275721804.