Deep Learning from Scratch

JR Jahed

mrahman13@qub.ac.uk | jrjahed100@gmail.com

1 Introduction

With each passing day, Artificial Intelligence is becoming more and more prominent in our everyday lives. With the meteoric rise of AI in recent years as one of the hottest topics globally, the number of people interested in learning it is skyrocketing. Deep learning, being the most powerful subdomain of AI, must be mastered by anyone interested in building a career in this field.

Highly optimised and sophisticated libraries such as PyTorch and TensorFlow exist, allowing users to build and train neural networks without knowing their underlying structures. However, anyone interested in advanced research or asserting full control over training, testing, and inference should grasp the fundamental concepts of these models and all the intricacies involved. Otherwise, they will be left with a superficial understanding, potentially causing trouble when issues arise.

The best way to develop a complete understanding of these models is by implementing them from scratch, without relying on any library. This approach allows valuable insights into what happens inside the model, layer by layer, step by step — insights that theoretical learning alone can never fully provide.

I learned Convolutional Neural Networks (CNNs) in January 2023 and trained multiple models on various datasets using TensorFlow in the first few months of that year. Afterward, my interests shifted towards history, geopolitics, and related areas. I read 10 books, thousands of news articles across 9-10 news outlets, thousands of Wikipedia articles, and watched hundreds of YouTube videos. For more than a year, I was completely out of touch not only with AI but also with computer programming in general. However, when I resumed programming in September 2024, it didn't take long for me to regain my previous level of proficiency. After finishing the first semester at Queen's, I decided to learn PyTorch and implement CNN using C++ and Python during the Christmas vacation in December. Initially, I thought I had a solid grasp of CNN concepts. However, I realised how mistaken I was when I started implementing them in December 2024.

In this project, I aimed to implement some of the most fundamental deep learning models — Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), and the Attention architecture — to deepen my understanding. Initially, I planned to implement only CNNs. However, completing CNN in five days encouraged me to expand the project to include other fundamental architectures as well. I then decided to prepare a report to elucidate each step and discuss the challenges encountered — some of which gave me several sleepless nights, an experience I deeply enjoy. Throughout this process, I extensively used ChatGPT. Interestingly, ChatGPT itself made multiple errors, which I had to fix myself. I used the free version throughout the implementation of CNN and RNN, and initially for the Transformer as well. When I became completely stuck with the vanishing gradient problem in the Transformer implementation, ChatGPT couldn't identify the issue. Thinking the paid version might help, I subscribed to ChatGPT Plus. But unfortunately, or fortunately — because it forced me to think harder and eventually solve the issue on my own — more capable model that comes with ChatGPT plus couldn't detect the problem either.

2 Background and Motivation

In March 2020, when the entire world was in lockdown due to Covid-19, I started computer programming. Although I had enrolled in a Computer Science degree in January 2017, I hadn't done much programming at that time because I aspired to become a professional cricket player. However, when that didn't materialise, I resumed my studies, only to find that my friends and even juniors had advanced significantly in programming. Initially, my friend was my backbone, guiding me through every step. He provided valuable resources and helped me learn basic data structures and algorithms. Within a few months, however, I reached the same level as my peers, and in some cases, even surpassed them.

In my early programming days, I used to google things as silly as "how to order a C++ map by value?" Eventually, after understanding the concept of a binary search tree (BST) and learning that the underlying data structure of a C++ map is a BST, I realised that a C++ map is inherently ordered by keys and cannot be ordered by values. After gaining substantial knowledge of such fundamental concepts, certain questions arose in my mind. For example, what happens when we insert [1, 2, 3, 4, 5] into a C++ set or map in the given order? All numbers following 1 would become right children of the previous ones, essentially forming an array. This structure becomes highly inefficient, defeating the purpose of a BST, which is efficiency. At that time, ChatGPT didn't exist, and Google was my primary resource. I discovered that balanced BSTs were developed to address this issue. I then learned about AVL and Red-Black trees, implementing them in both C++ and Java in early 2021. Interestingly, my university peers, who had 3-4 years of experience in competitive programming and were specialists or experts on platforms like Codeforces, probably never considered such questions. They used C++ sets and maps without realising the underlying data structure was a balanced BST (specifically, a Red-Black tree), not just a simple BST. They knew it was a BST but never questioned what would happen if a sorted array were inserted into it.

I included this extensive background because, after speaking with professors and PhD students at Queen's, I realised they were surprised that I took the trouble to implement these concepts. They preferred to use existing libraries and find online resources whenever necessary.

Additionally, I discussed binary search trees with some of my classmates at Queen's, including two who completed their undergraduate degrees there, one from Trinity College Dublin, two from India, and one from Pakistan. Some of these classmates are among the top performers in our class, consistently ranking within the top 3-5. Surprisingly, almost all lacked sufficient knowledge of BSTs, let alone balanced BSTs. They weren't curious about the underlying structures of common programming language features such as sets, maps, treesets, or treemaps, and some didn't even remember what they'd learned during their undergraduate studies. This made me realise that their foundational understanding of computer science was weaker than I expected, despite some having studied at prestigious institutions like Queen's or Trinity.

After these interactions, I understood that my curiosity has no boundaries, and I possess an uncanny ability to ask foundational "why" questions, such as "Why does it work like that?" or "Why doesn't it work differently?" This curiosity compels me to dive deeper and leaves me restless until I find satisfactory answers. Recently, I became slightly concerned because no such questions were arising in my mind regarding AI. This was unusual because, historically, whenever I delved deeper into any subject—be it calculus, physics, or programming—questions naturally arose. However, after a few days, one question emerged regarding the dropout technique:

How does dropout work?

Do we randomly deactivate specific neurons once at the start of training, making them inactive throughout all training batches and epochs? Or do we randomly select neurons for deactivation at every batch and/or epoch?

Initially busy with other tasks, I saved the question to explore later. I asked one of my professors, but he had never considered this detail. I refrained from asking ChatGPT until thoroughly reading the original dropout paper [1] (which I'd skimmed previously but never completely read). I assumed that neurons are randomly selected for deactivation in each epoch or batch because, if neurons were permanently deactivated throughout training, including them would be pointless.

The primary motivation behind this project is to solidify my understanding of deep learning's fundamental concepts, preparing me for advanced research, which I'm eager to undertake in the future. AI is currently one of the most dynamic research fields, and foundational concepts might become outdated, replaced by emerging theories. Some may wonder why we should implement models that might become anachronistic in the future. A notable example is the recurrent neural network (RNN), once central to natural language processing, which significantly declined in importance after the introduction of the attention mechanism and transformer architecture. However, implementing these concepts helps us gain essential insights into deep learning in general. The expertise acquired is easily transferable and greatly aids in understanding new theories as they arise.

3 Convolutional Neural Network

I first learned Convolutional Neural Networks (CNNs) in January 2023, trained models using TensorFlow, deployed one to the cloud, and developed what I later realised was a superficial understanding when I attempted implementation from scratch in December 2024. Back in 2023, I had no knowledge of backpropagation or its internal workings and struggled to understand certain aspects of the forward pass. For example, I asked ChatGPT:

> *model = Sequential([*
> *Input(shape=(28, 28, 3)),*
> *Conv2D(1, (3, 3)),*
> *])*
> *Consider this model. if convolutional operation is performed on 3 color channels, shouldn't the number of activation map be 3. Don't we get 3 distinct feature map after performing convolutional operation 3 times on 3 color channel. Then why only 1 activation map comes out of convolutional layer.*

Yet, I could easily train models without fully understanding these internal processes.

My chat with ChatGPT regarding the implementation of CNN can be found on Google Drive

https://drive.google.com/file/d/1w7rWWJZArWKC4kmjgELlcj9vacIqO7AT

3.1 Issues Encountered

1. **Weight Tensor Dimensions in Convolutional Layer:** Initially, I thought the weight tensor should be a 3D array shaped (output_channels, kernel_height, kernel_width). Later, I realised there should be a convolutional kernel for every input channel in each output channel. Therefore, the weight tensor should be a 4D array shaped (output_channels, input_channels, kernel_height, kernel_width).

```
class Conv2d {
public:
    vector<vector<vector<double>>> kernels;  // Filters, each of size 3x3
    vector<double> biases;  // Bias for each filter
    int neurons;  // Number of filters
    int kernel_size = 3;  // Kernel size (3x3)

    Conv2d(int _neurons) : neurons(_neurons) {
        kernels = vector<vector<vector<double>>>(neurons,
        vector<vector<double>>(kernel_size, vector<double>(kernel_size)));
        biases = vector<double>(neurons, .1);

        for(auto& matrix: kernels) {
            for(auto& row: matrix) {
                for(auto& val: row) {
                    val = dist(rng);
                }
            }
        }
    }
}
```

2. **Array Indexing:** I made a critical mistake in array indexing in the backward function of the Dense class, which was extremely difficult to identify and resolve.

```
    Dense class
    vector<double> backward(const vector<double>& dL_dout,
    const vector<double>& input, double learning_rate) {

        vector<double> dL_din(input.size(), 0.0);
        vector<double> dL_dweights(weights.size(), 0.0);
        vector<double> dL_dbiases(biases.size(), 0.0);

        // Gradients for weights and biases
        for(int i = 0; i < neurons; i++) {
            dL_dbiases = dL_dout[i];
            for(int j = 0; j < input.size(); j++) {
                dL_dweights[j] += dL_dout[i] * input[j];
                dL_din[j] += weights[i] * dL_dout[i];
            }
        }
        // Update weights and biases
        for(int i = 0; i < neurons; i++) {
            biases[i] -= learning_rate * dL_dbiases[i];
            for(int j = 0; j < input.size(); j++) {
                weights[i] -= learning_rate * dL_dweights[j];
            }
        }
        return dL_din;
    }

    Sequential class
    void backward(const vector<double>& dL_dout, double learning_rate) {
        vector<double> current_dL_dout = dL_dout;
        for(int i = dense_layers.size() - 1; i >= 0; i--) {
            const vector<double>& input = (i == 0) ? flatten(conv_outputs.back())
                        : dense_outputs[i - 1];
            current_dL_dout = dense_layers[i].backward(current_dL_dout,
                            input, learning_rate);
        }
    }
```

```
Conv2d conv1(2);
model.addConvLayer(conv1);
Dense dense1(32, "relu");
model.addDenseLayer(dense1);
Dense dense2(3, "softmax");
model.addDenseLayer(dense2);
```

Consider this model. During the first iteration in line 26 we're trying to find the gradients of the weights of last dense layer which has 3 neurons. This layer has 32 inputs. Isn't it incorrect to use j in line 10 to access dL_dweights[j] as dL_dweights stores the values for the last dense layer, which has 3 neurons, but j can be up to 31? Shouldn't it be dL_dweights[i]?

3. **Model Predicting One Class:** Even after correcting the issues, the model was predicting the same class for all inputs. At that point, I thought I had reached my limit. Then, I decided to try Glorot initialization, which resolved the issue and allowed the model to correctly identify input images.

3.2 Mistakes of ChatGPT

1. It failed to detect the incorrect dimensions of the weight tensor in the Conv2D class.
2. It couldn't solve the indexing issue; I had to resolve this myself.

4 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are well-suited for processing sequential data, where the input at a given time step depends on previous inputs. After computing the output, known as the hidden state, for a particular time step, this hidden state is used in the computation of the next time step. In this way, information is carried forward through the sequence.

I had to work with the Eigen library for the first time while implementing RNNs in C++. However, the C++ implementation suffered from gradient explosion, which I did not attempt to fix because I had already started working on the Transformer implementation. I might revisit and try to address this issue in the future if time permits.

Google Drive link to access the chat with ChatGPT while implementing RNN.

https://drive.google.com/file/d/1PfsBpNzq7PcNVmFOoxgZ1M0S6XZH_SVo

4.1 Issues Encountered

1. **Dimension Expansion and Reduction:** Expanding dimension of the tensors and dropping the dimension again multiple times for matrix multiplication was very confusing and highly error-prone.
2. **Reshaping Tensors:** Preserving the correct dimensions, especially for batch size and hidden size during tensor reshaping, was particularly challenging.

4.2 Mistakes of ChatGPT

1. ChatGPT gave me the following line which is a grave mistake because it doesn't preserve the dimension along which batch size and hidden size is stored.

   h = h.reshape(batch_size, self.hidden_size, 1)

   The shape of h before reshaping is (hidden_size, batch_size) and it should be (batch_size, hidden_size, 1) after reshaping.

Suppose,

max_sequence_length = 3
hidden_size = 4
batch_size = 2

```
output_for_time_step_1 = np.array([[111, 112],
                                  [121, 122],
                                  [131, 132],
                                  [141, 142],])

output_for_time_step_2 = np.array([[211, 212],
                                  [221, 222],
                                  [231, 232],
                                  [241, 242],])

output_for_time_step_3 = np.array([[311, 312],
                                  [321, 322],
                                  [331, 332],
                                  [341, 342],])
```

Here, first digit corresponds to the time step, second digit to neuron, and third to batch instance. The rows correspond to the output of a specific neuron for different instances of the batch, and therefore, the third digits differ along the row. The columns correspond to the output of different neurons for a specific batch instance, and therefore, the second digits differ along the column.

We get the following after reshaping:

```
output_for_time_step_1_reshaped = [[111 112 121 122]
                                   [131 132 141 142]]

output_for_time_step_2_reshaped = [[211 212 221 222]
                                   [231 232 241 242]]

output_for_time_step_3_reshaped = [[311 312 321 322]
                                   [331 332 341 342]]
```

This is wrong because neither the rows nor the columns correspond to the output of a particular neuron or a batch instance. Instead, we should transpose the matrices to get the correct output, such as the following:

```
output_for_time_step_1_transposed = [[111 121 131 141]
                                     [112 122 132 142]]

output_for_time_step_2_transposed = [[211 221 231 241]
                                     [212 222 232 242]]
```

```
output_for_time_step_3_transposed = [[311 321 331 341]
                                     [312 322 332 342]]
```

5 Transformer

Since its inception in 2017 by Vaswani et al.[2], Transformer has become one of the most valuable architectures of machine learning. Originally developed for machine translation, it is now adapted for various tasks including text classification, sentiment analysis, text summarisation, and more. Large Language Models (LLMs) such as ChatGPT, BERT, DeepSeek etc. are based on this architecture. Its usage extends well beyond NLP, with applications in image and speech recognition, audio generation and reinforcement learning.

Implementations of Transformers vary depending on specific tasks. Following the original paper, "Attention is All You Need," which focused on machine translation, my implementation aimed to build a model capable of mapping one sequence to another.

Google Drive link to the chat with ChatGPT while implementing Transformer.

https://drive.google.com/file/d/1xbDsGyZbDak8zwwl6fPTgebaPto5G3mI

5.1 Questions That Arose While Implementing

The following are the questions that arose in my mind while implementing transformer.

1. What do we use transformer for? It seems it's a general architecture that can tailored for specific needs. We can use it for translation from one language to another. We can build LLMs based on this architecture and train them using enormous amounts of data to generate responses to prompts. How correct is my understanding? Is there any other usage of transformer?

2. If it's indeed a general architecture that is tailored for specific tasks, then the implementations of transformer for different tasks are supposed to be different. For example, if we are building a transformer model for machine translation, it should be able to map the sentences of one language to the sentences of another. If we are building an LLM, it should generate appropriate responses based on users' input. If the task is classification, it doesn't have to generate anything and only predicting the class would suffice. Therefore, the implementation seems to vary based on tasks.

3. Since I'm trying to replicate the original architecture, which was developed for machine translation, my goal in this project is to develop a transformer model that can map one sequence to another.

   As it's a translation model, there will be two different inputs — the first one is from one language and the second one is from another.

   - How do we insert the input to the model
   - What is the shape of the input? Is it (batch size, sequence length)?

4. How does positional encoding work? It's a fixed matrix of shape (position, d_model). My understanding is that adding it to the embedding matrix injects information about the relative positions of the words regardless of what the words are. For example, "I love chocolate" and "Universe is ginormous" are two distinct sentences of equal length that have different meanings and share no common word. Embedding is concerned with the words themselves. However, positional encoding only cares about the fact that "love" comes after "I", "is" comes after "Universe" and so on. When we add embedding with positional encoding, we get an output that captures the words and their relative positions. Am I right?

5. I think the first dimension of the positional encoding matrix is flexible. In Tensorflow documentation, they used (2048, d_model). Does that mean the model can capture the positions of the words in a sequence of length 2048 at most?

6. Consider the following statement from the paper.

   > *The encoder contains self-attention layers. In a self-attention layer all of the keys, values and queries come from the same place, in this case, the output of the previous layer in the encoder. Each position in the encoder can attend to all positions in the previous layer of the encoder.*

   Where does the input to the first encoder layer come from? It's supposed to be the output of positional embedding which is of shape (batch_size, max_sequence_length, d_model). How do we get the query, key, and value?

7. As d_k and d_v are equal, what can be the possible reason behind using two variable names? Can't we just use d to refer to that value?

8. The output of global self attention goes to dense layer as input. There is also a residual connection which means it goes to add layer that adds this output to the output of the dense layer. The output of attention layer affects the final output in two ways — once by going through the dense layer and again by being added to the output of the dense layer. Therefore, during backpropagation we should add the gradient in add layer with the gradient returned by dense layer to get the gradient of the output of attention layer. Shouldn't we?

   This is true for other layers as well.

5.2 Mistakes of ChatGPT

1. Updating the weights immediately after computing gradients in MultiHeadAttention and LayerNormalisation.

```
class MultiHeadAttention:
    def backward(self, gradient, learning_rate):
        """
        @Params
        gradient: (batch_size, max_sequence_length, d_model)

        @Returns
        gradient: (batch_size, max_sequence_length, d_model)
        """

        gradient_weights = np.dot(self.concatenated.reshape(
        -1, self.d_model).T, gradient.reshape(-1, self.d_model))
        self.W -= learning_rate * gradient_weights

        # Gradient with respect to the concatenated output
        gradient_concatenated = np.dot(gradient, self.W.T)

        # Split gradient into each head's gradient
        gradient_heads = np.split(gradient_concatenated, self.n_heads,
        axis=-1)  # shape (batch_size, max_sequence_length, head_dim)

        gradient_query = 0
        gradient_key = 0
        gradient_value = 0

        for i, head in enumerate(self.attention_heads):
            gradient_q, gradient_k, gradient_v = head.backward(gradient_heads[i], learning_rate)
            gradient_query += gradient_q
            gradient_key += gradient_k
            gradient_value += gradient_v

        return gradient_query, gradient_key, gradient_value

class LayerNormalisation:
    def backward(self, gradient, learning_rate):
        """
        @Params
        gradient: (batch_size, max_sequence_length, d_model)
        @Returns
        gradient: (batch_size, max_sequence_length, d_model)
        """

        gradient_g = np.sum(gradient * self.normalised, axis=(0, 1))  # shape: (d_model,)
        gradient_b = np.sum(gradient, axis=(0, 1))  # shape: (d_model,)

        self.g -= learning_rate * gradient_g
        self.b -= learning_rate * gradient_b

        gradient_normalised = gradient
        * self.g  # shape: (batch_size, max_sequence_length, d_model)

        # Following the formula for the gradient of layer norm:
        # d_x = (1/sigma) * (d_normalized - mean(d_normalized) - normalized
        * mean(d_normalized * normalized))

        mean_gradient_normalised = np.mean(gradient_normalised, axis=-1,
        keepdims=True)  # shape: (batch_size, max_sequence_length, 1)

        mean_gradient_normalised_normalised = np.mean(gradient_normalised * self.normalised,
        axis=-1, keepdims=True)  # shape (batch_size, max_sequence_length, 1)

        gradient_input = (1.0 / self.standard_deviation) * (gradient_normalised -
        mean_gradient_normalised - self.normalised * mean_gradient_normalised_normalised)

        return gradient_input
```

"Consider these two backward functions. We are updating weights immediately after calculating the gradients of the weights and we are using that updated weights to calculate other values. Isn't it a mistake? We are calculating the gradients for the previous forward pass during which the current states of the weights were used for calculations, not the updated weights. Shouldn't we update the weights after doing all the calculations in the backward pass?"

2. It couldn't detect the mistake in the backward function of the Decoder class. I was passing gradient_output that the backward function receives from final output layer to all the subsequent decoder

layers.

```
def backward(self, gradient_output, learning_rate):
    """
    @Params
    gradient_output: (batch_size, max_sequence_length, d_model)

    @Returns
    gradient_context: (batch_size, max_sequence_length, d_model)
    """

    # We need to pass this to encoder. It's the summation of the gradients of context
    of all the cross attention layers
    gradient_context = 0

    for layer in reversed(self.decoder_layers):
        gradient_x, gradient_context_ = layer.backward(gradient_output, learning_rate)
        gradient_context += gradient_context_

    self.positional_embedding.backward(gradient_output, learning_rate)
    return gradient_context
```

> "I was just passing the initial gradient_output — that the Decoder class receives from the final output layer — to all the decoder layers. It was a mistake. Then, I realised that we should pass gradient of the output of the previous decoder layer, which is the input of the current layer we're processing."

3. It failed to detect the mistake in the gradients of residual connections. I was not handling the gradients of residual connections of attention layers.

```
def backward(self, gradient_output, learning_rate):
    """
    @Params
    gradient_output: (batch_size, max_sequence_length, d_model)
    @Returns
    gradient_x: (batch_size, max_sequence_length, d_model)
    """

    gradient = self.layer_normalisation.backward(gradient_output, learning_rate)

    """
    I'm not taking care of the gradients of the residual connections of attention layers
    and this might be causing vanishing gradients when there are more than one layer
    """

    # As MultiHeadAttention receives 3 inputs (query, key, value),
    it returns 3 gradients during backpropagation
    gradient_query, gradient_key, gradient_value = self.mha.backward(gradient, learning_rate)

    # Sum the three gradients
    return gradient_query + gradient_key + gradient_value
```

6 Conclusion

This project allowed me to deeply explore the internal mechanisms of fundamental deep learning algorithms. When I began implementing CNN in December 2024, I didn't expect to complete it relatively quickly and move on to implement other architectures as well. There were times when I felt overwhelmed, thought it was beyond my capability and considered giving up. However, the realisation that abandoning the project after significant progress made no sense, coupled with having ample free time due to finishing courseworks well before deadlines, kept me motivated.

Thanks to OpenAI for creating a tool like ChatGPT which played a critical role in this project. I'm not sure if I could do this without ChatGPT. Even if I could, it would have taken weeks, if not months, but certainly not days. Despite the mistakes it made, its support was invaluable.

References

[1] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: a simple way to prevent neural networks from overfitting,” *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[2] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” *Advances in neural information processing systems*, vol. 30, 2017.