In the previous two chapters, we learned extensively about the various convolutional and recurrent network architectures available, along with their implementations in PyTorch. In this chapter, we will take a look at some other deep learning model architectures that have proven to be successful on various machine learning tasks and are neither purely convolutional nor recurrent in nature. We will continue from where we left off in both *Chapter 3*, *Deep CNN Architectures*, and *Chapter 4, Deep Recurrent Model Architectures*.

First, we will explore transformers, which, as we learnt toward the end of *Chapter 4, Deep Recurrent Model Architectures*, have outperformed recurrent architectures on various sequential tasks. Then, we will pick up from the **EfficientNets** discussion at the end of *Chapter 3, Deep CNN Architectures*, and explore the idea of generating randomly wired neural networks, also known as **RandWireNNs.**

With this chapter, we aim to conclude our discussion of different kinds of neural network architectures in this book. After completing this chapter, you will have a detailed understanding of transformers and how to apply these powerful models to sequential tasks using PyTorch. Furthermore, by building your own RandWireNN model, you will have hands-on experience of performing a neural architecture search in PyTorch. This chapter is broken down into the following topics:

- Building a transformer model for language modeling
- Developing a RandWireNN model from scratch

Building a transformer model for language modeling

In this section, we will explore what transformers are and build one using PyTorch for the task of language modeling. We will also learn how to use some of its successors, such as **BERT** and **GPT**, via PyTorch's pretrained model repository. Before we start building a transformer model, let's quickly recap what language modeling is.

Reviewing language modeling

Language modeling is the task of figuring out the probability of the occurrence of a word or a sequence of words that should follow a given sequence of words. For example, if we are given *French is a beautiful* _____ as our sequence of words, what is the probability that the next word will be *language* or *word*, and so on? These probabilities are computed by modeling the language using various probabilistic and statistical techniques. The idea is to observe a text corpus and learn the grammar by learning which words occur together and which words never occur together. This way, a language model establishes probabilistic rules around the occurrence of different words or sequences, given various different sequences.

Recurrent models have been a popular way of learning a language model. However, as with many sequence-related tasks, transformers have outperformed recurrent networks on this task as well. We will implement a transformer-based language model for the English language by training it on the Wikipedia text corpus.

Now, let's start training a transformer for language modeling. During this exercise, we will demonstrate only the most important parts of the code. The full code can be accessed at our github repository [5.1].

We will delve deeper into the various components of the transformer architecture in-between the exercise.

For this exercise, we will need to import a few dependencies. One of the important `import` statements is listed here:

```
from torch.nn import TransformerEncoder, TransformerEncoderLayer
```

Besides importing the regular `torch` dependencies, we must import some modules specific to the transformer model; these are provided directly under the torch library. We'll also import `torchtext` in order to download a text dataset directly from the available datasets under `torchtext.datasets`.

In the next section, we will define the transformer model architecture and look at the details of the model's components.

Understanding the transformer model architecture

This is perhaps the most important step of this exercise. Here, we define the architecture of the transformer model.

First, let's briefly discuss the model architecture and then look at the PyTorch code for defining the model. The following diagram shows the model architecture:

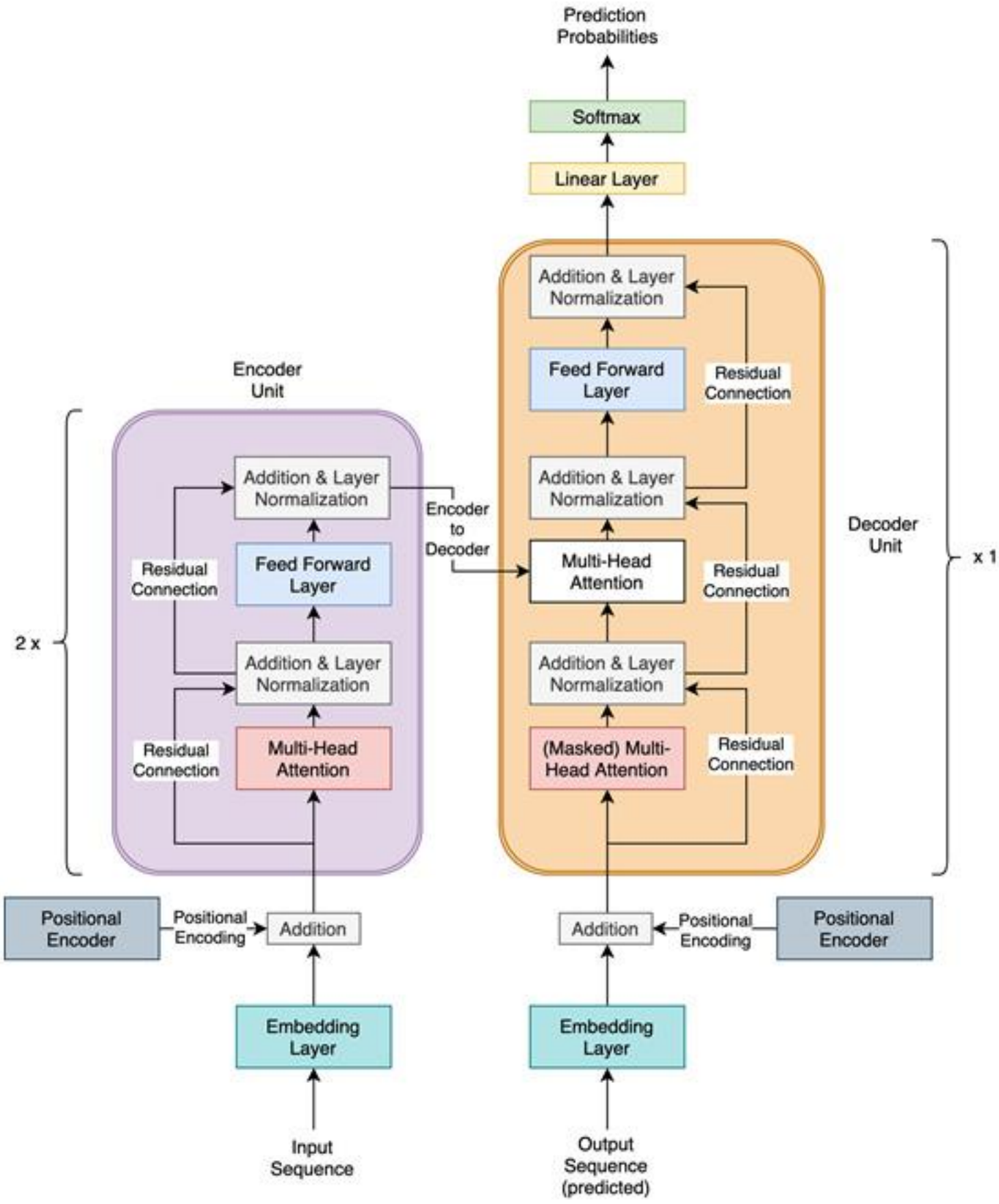

Figure 5.1 – Transformer model architecture

The first thing to notice is that this is essentially an encoder-decoder based architecture, with the **Encoder Unit** on the left (in purple) and the **Decoder Unit** (in orange) on the right. The encoder and decoder units can be tiled multiple times for even deeper architectures. In our example, we have two cascaded encoder units and a single decoder unit. This encoder-decoder setup essentially means that the encoder takes a sequence as input and generates as many embeddings as there are words in the input sequence (that is, one embedding per word). These embeddings are then fed to the decoder, along with the predictions made thus far by the model.

Let's walk through the various layers in this model:

Embedding Layer: This layer is simply meant to perform the traditional task of converting each input word of the sequence into a vector of numbers; that is, an embedding. As always, here, we use the `torch.nn.Embedding` module to code this layer.

Positional Encoder: Note that transformers do not have any recurrent layers in their architecture, yet they outperform recurrent networks on sequential tasks. How? Using a neat trick known as *positional encoding*, the model is provided the sense of sequentiality or sequential-order in the data. Basically, vectors that follow a particular sequential pattern are added to the input word embeddings.

These vectors are generated in a way that enables the model to understand that the second word comes after the first word and so on. The vectors are generated using the `sinusoidal` and `cosinusoidal` functions to represent a systematic periodicity and distance between subsequent words, respectively. The implementation of this layer for our exercise is as follows:

Copy Explain

```
class PosEnc(nn.Module):
    def __init__(self, d_m, dropout=0.2, size_limit=5000):
        # d_m is same as the dimension of the embeddings
        pos = torch.arange(     size_limit, dtype=torch.float).unsqueeze(1)
        divider = torch.exp(torch.arange(0, d_m, 2).float() * (-math.log(10000.0) /
d_m))
        # divider is the list of radians, multiplied by position indices of words, and
fed to the sinusoidal and cosinusoidal function.
        p_enc[:, 0, 0::2] = torch.sin(pos * divider)
        p_enc[:, 0, 1::2] = torch.cos(pos * divider)
    def forward(self, x):
        return self.dropout(x + self.p_enc[:x.size(0)     ])
```

As you can see, the `sinusoidal` and `cosinusoidal` functions are used alternatively to give the sequential pattern. There are many ways to implement positional encoding though. Without a positional encoding layer, the model will be clueless about the order of the words.

Multi-Head Attention: Before we look at the multi-head attention layer, let's first understand what a **self-attention layer** is. We covered the concept of attention in *Chapter 4*, *Deep Recurrent Model Architectures*, with respect to recurrent networks. Here, as the name suggests, the attention mechanism is applied to self; that is, each word of the sequence. Each word embedding of the sequence goes through the self-attention layer and produces an individual output that is exactly the same length as the word embedding. The following diagram describes the process of this in detail:

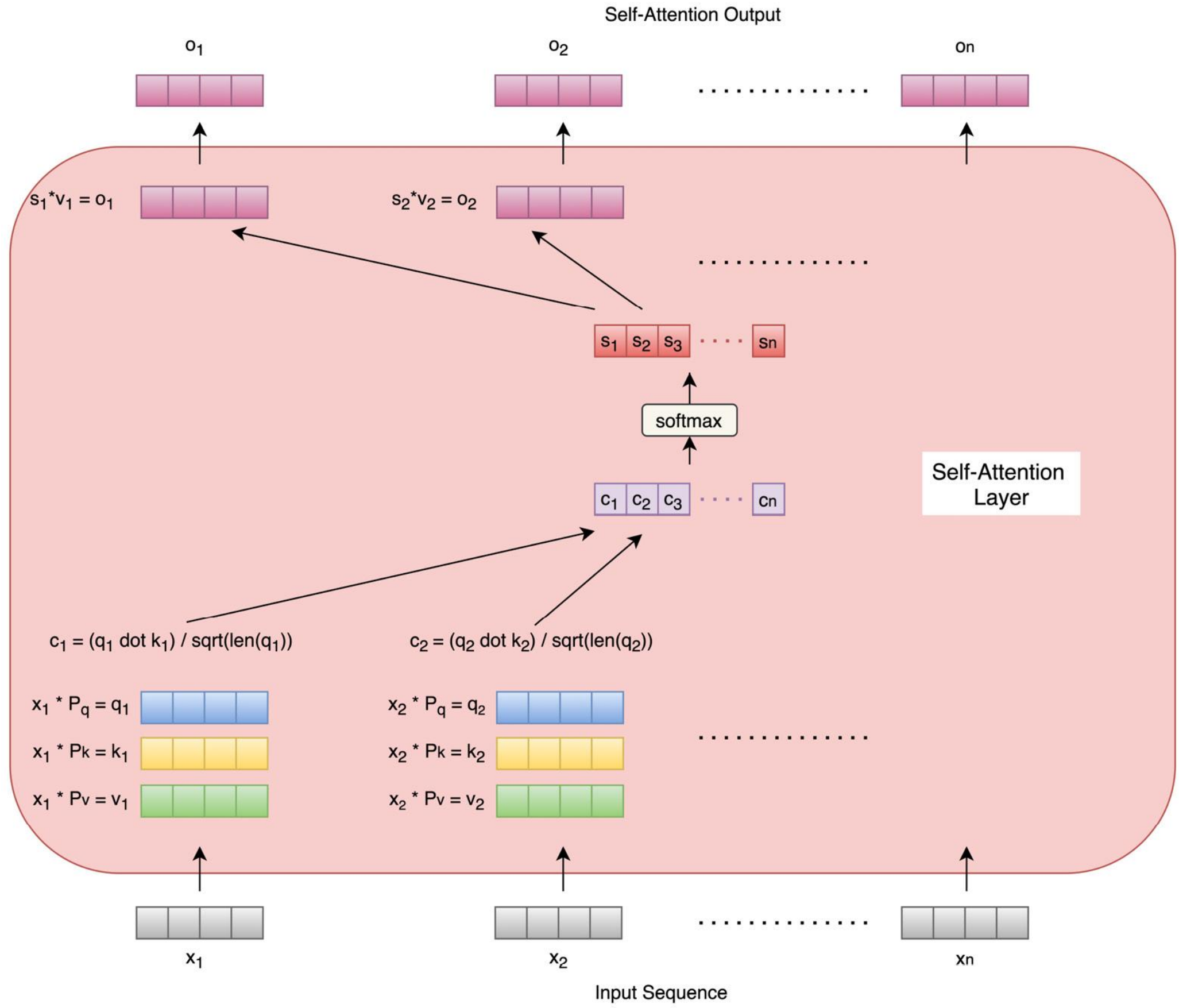

Figure 5.2 – Self-attention layer

As we can see, for each word, three vectors are generated through three learnable parameter matrices (**Pq**, **Pk**, and **Pv**). The three vectors are query, key, and value vectors. The query and key vectors are dot-multiplied to produce a number for each word. These numbers are normalized by dividing the square root of the key vector length for each word. The resultant numbers for all words are then Softmaxed at the same time to produce probabilities that are finally multiplied by the respective value vectors for each word. This results in one output vector for each word of the sequence, with the lengths of the output vector and the input word embedding being the same.

A multi-head attention layer is an extension of the self-attention layer where multiple self-attention modules compute outputs for each word. These individual outputs are concatenated and matrix-multiplied with yet another parameter matrix (**Pm**) to generate the final output vector, whose length is equal to the input embedding vector's. The following diagram shows the multi-head attention layer, along with two self-attention units that we will be using in this exercise:

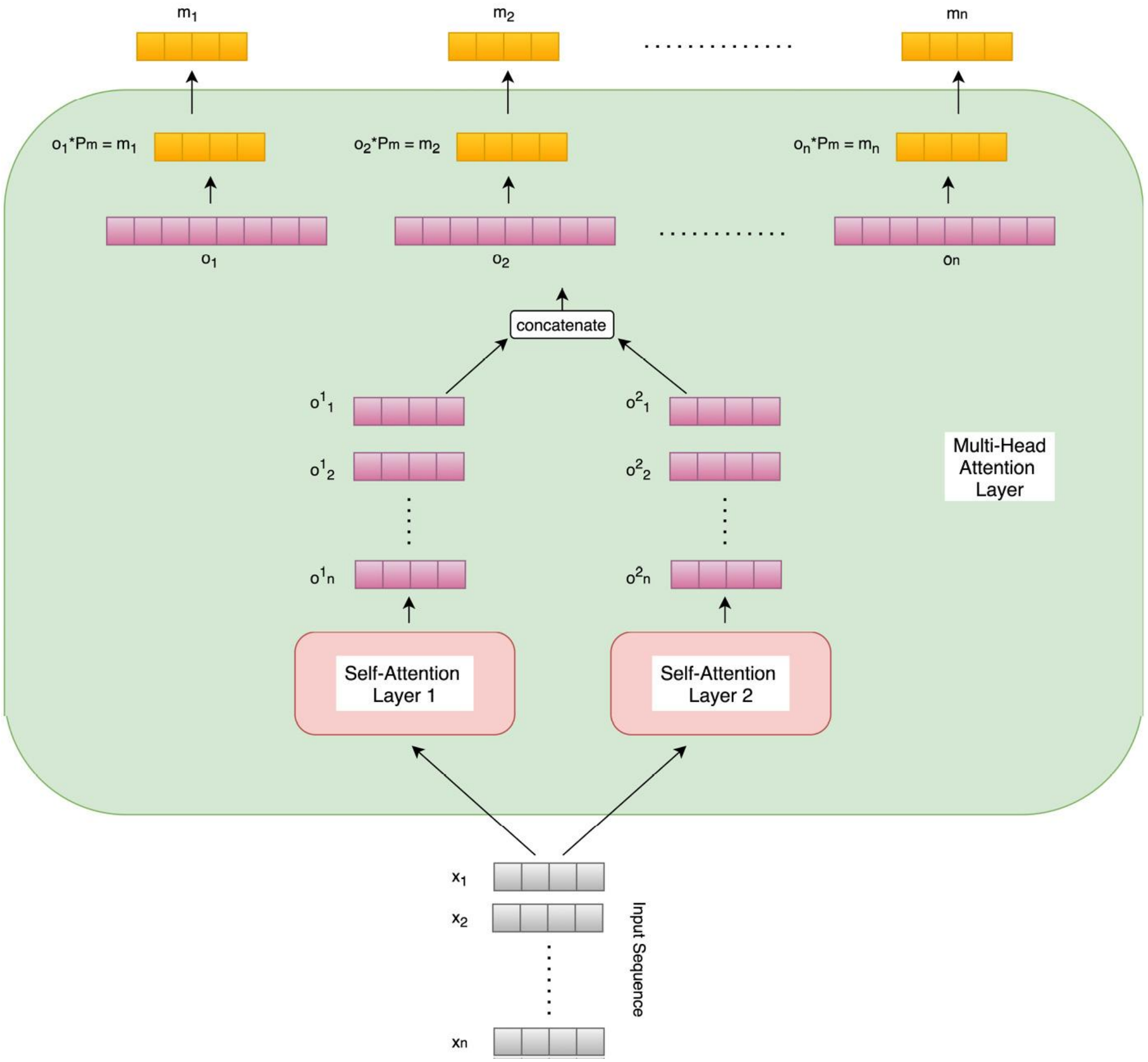

Figure 5.3 – Multi-head attention layer with two self-attention units

Having multiple self-attention heads helps different heads focus on different aspects of the sequence word, similar to how different feature maps learn different patterns in a convolutional neural network. Due to this, the multi-head attention layer performs better than an individual self-attention layer and will be used in our exercise.

Also, note that the masked multi-head attention layer in the decoder unit works in exactly the same way as a multi-head attention layer, except for the added masking; that is, given time step *t* of processing the sequence, all words from *t+1* to *n* (length of the sequence) are masked/hidden.

During training, the decoder is provided with two types of inputs. On one hand, it receives query and key vectors from the final encoder as inputs to its (unmasked) multi-head attention layer, where these query and key vectors are matrix

transformations of the final encoder output. On the other hand, the decoder receives its own predictions from previous time steps as sequential input to its masked multi-head attention layer.

Addition and Layer Normalization: We discussed the concept of a residual connection in *Chapter 3*, *Deep CNN Architectures*, while discussing ResNets. In *Figure 5.1*, we can see that there are residual connections across the addition and layer normalization layers. In each instance, a residual connection is established by directly adding the input word embedding vector to the output vector of the multi-head attention layer. This helps with easier gradient flow throughout the network and avoiding problems with exploding and vanishing gradients. Also, it helps with efficiently learning identity functions across layers.

Furthermore, layer normalization is used as a normalization trick. Here, we normalize each feature independently so that all the features have a uniform mean and standard deviation. Please note that these additions and normalizations are applied individually to each word vector of the sequence at each stage of the network.

Feedforward Layer: Within both the encoder and decoder units, the normalized residual output vectors for all the words of the sequence are passed through a common feedforward layer. Due to there being a common set of parameters across words, this layer helps with learning broader patterns across the sequence.

Linear and Softmax Layer: So far, each layer is outputting a sequence of vectors, one per word. For our task of language modeling, we need a single final output. The linear layer transforms the sequence of vectors into a single vector whose size is equal to the length of our word vocabulary. The **Softmax** layer converts this output into a vector of probabilities summing to 1. These probabilities are the probabilities that the respective words (in the vocabulary) occur as the next words in the sequence.

Now that we have elaborated on the various elements of a transformer model, let's look at the PyTorch code for instantiating the model.

Defining a transformer model in PyTorch

Using the architecture details described in the previous section, we will now write the necessary PyTorch code for defining a transformer model, as follows:

Copy Explain

```
class Transformer(nn.Module):
    def __init__(self, num_token, num_inputs, num_heads, num_hidden, num_layers,
dropout=0.3):
        self.position_enc = PosEnc(num_inputs, dropout)
        layers_enc = TransformerEncoderLayer(num_inputs, num_heads, num_hidden,
dropout)
        self.enc_transformer = TransformerEncoder(layers_enc, num_layers)
        self.enc = nn.Embedding(num_token, num_inputs)
        self.num_inputs = num_inputs
        self.dec = nn.Linear(num_inputs, num_token)
```

As we can see, in the `__init__` method of the class, thanks to PyTorch's `TransformerEncoder` and `TransformerEncoderLayer` functions, we do not need to implement these ourselves. For our language modeling task, we just need a single output for the input sequence of words. Due to this, the decoder is just a linear layer that transforms the sequence of vectors from an encoder into a single output vector. A position encoder is also initialized using the definition that we discussed earlier.

In the `forward` method, the input is positionally encoded and then passed through the encoder, followed by the decoder:

Copy Explain

```
    def forward(self, source):
        source = self.enc(source) * math.sqrt(self.num_inputs)
        source = self.position_enc(source)
        op = self.enc_transformer(source, self.mask_source)
        op = self.dec(op)
        return op
```

Now that we have defined the transformer model architecture, we shall load the text corpus to train it on.

Loading and processing the dataset

In this section, we will discuss the steps related to loading a text dataset for our task and making it usable for the model training routine. Let's get started:

For this exercise, we will be using texts from Wikipedia, all of which are available as the `WikiText-2` dataset.

Dataset Citation
https://blog.einstein.ai/the-wikitext-long-term-dependency-language-modeling-dataset/.

We'll use the functionality of `torchtext` to download the training dataset (available under `torchtext` datasets) and tokenize its vocabulary:

Copy Explain

```
tr_iter = WikiText2(split='train')
tkzer = get_tokenizer('basic_english')
vocabulary = build_vocab_from_iterator(map(tkzer, tr_iter), specials=['<unk>'])
vocabulary.set_default_index(vocabulary['<unk>'])
```

1. We will then use the vocabulary to convert raw text into tensors for the training, validation and testing datasets:

Copy Explain

```
def process_data(raw_text):
    numericalised_text = [torch.tensor(vocabulary(tkzer(text)), dtype=torch.long) for
text in raw_text]
    return torch.cat(tuple(filter(lambda t: t.numel() > 0, numericalised_text)))
tr_iter, val_iter, te_iter = WikiText2()
training_text = process_data(tr_iter)
validation_text = process_data(val_iter)
testing_text = process_data(te_iter)
```

1. We'll also define the batch sizes for training and evaluation and declare a batch generation function, as shown here:

Copy Explain

```
def gen_batches(text_dataset, batch_size):
    num_batches = text_dataset.size(0) // batch_size
    text_dataset = text_dataset[:num_batches * batch_size]
    text_dataset = text_dataset.view(batch_size, num_batches).t().contiguous()
    return text_dataset.to(device)
training_batch_size = 32
evaluation_batch_size = 16
training_data = gen_batches(training_text, training_batch_size)
```

1. Next, we must define the maximum sequence length and write a function that will generate input sequences and output targets for each batch, accordingly:

```
max_seq_len = 64
def return_batch(src, k):
    sequence_length = min(max_seq_len, len(src) - 1 - k)
    sequence_data = src[k:k+sequence_length]
    sequence_label = src[k+1:k+1+sequence_length].reshape(-1)
    return sequence_data, sequence_label
```

Having defined the model and prepared the training data, we will now train the transformer model.

Training the transformer model

In this section, we will define the necessary hyperparameters for model training, define the model training and evaluation routines, and finally execute the training loop. Let's get started:

In this step, we define all the model hyperparameters and instantiate our transformer model. The following code is self-explanatory:

```
num_tokens = len(vocabulary) # vocabulary size
embedding_size = 256 # dimension of embedding layer
num_hidden_params = 256 # transformer encoder's hidden (feed forward) layer dimension
num_layers = 2 # num of transformer encoder layers within transformer encoder
num_heads = 2 # num of heads in (multi head) attention models
dropout = 0.25 # value (fraction) of dropout
loss_func = nn.CrossEntropyLoss()
lrate = 4.0 # learning rate
optim_module = torch.optim.SGD(transformer_model.parameters(), lr=lrate)
sched_module = torch.optim.lr_scheduler.StepLR(optim_module, 1.0, gamma=0.88)
transformer_model = Transformer(num_tokens, embedding_size, num_heads,
num_hidden_params, num_layers, dropout).to(device)
```

Before starting the model training and evaluation loop, we need to define the training and evaluation routines:

Copy Explain

```
def train_model():     )
    for b, i in enumerate(range(0, training_data.size(0) - 1, max_seq_len)):
        train_data_batch, train_label_batch = return_batch(training_data, i)
        sequence_length = train_data_batch.size(0)
        if sequence_length != max_seq_len:  # only on last batch
            mask_source = mask_source[:sequence_length, :sequence_length]

        op = transformer_model(train_data_batch, mask_source)
        loss_curr = loss_func(op.view(-1, num_tokens), train_label_batch)
       optim_module.zero_grad()
        loss_curr.backward()
torch.nn.utils.clip_grad_norm_(transformer_model.parameters(), 0.6)
        optim_module.step()
        loss_total += loss_curr.item()
def eval_model(eval_model_obj, eval_data_source):
...
```

Finally, we must run the model training loop. For demonstration purposes, we are training the model for 5 epochs, but you are encouraged to run it for longer in order to get better performance:

Copy Explain

```
min_validation_loss = float("inf")
eps = 5
best_model_so_far = None
for ep in range(1, eps + 1):
    ep_time_start = time.time()
    train_model()
    validation_loss = eval_model(transformer_model, validation_data)
    if validation_loss < min_validation_loss:
        min_validation_loss = validation_loss
        best_model_so_far = transformer_model
```

This should result in the following output:

```
epoch 1, 100/1000 batches, training loss 8.77, training perplexity 6460.73
epoch 1, 200/1000 batches, training loss 7.30, training perplexity 1480.28
epoch 1, 300/1000 batches, training loss 6.88, training perplexity 969.18
epoch 1, 400/1000 batches, training loss 6.64, training perplexity 764.52
epoch 1, 500/1000 batches, training loss 6.54, training perplexity 689.84
epoch 1, 600/1000 batches, training loss 6.38, training perplexity 590.40
epoch 1, 700/1000 batches, training loss 6.33, training perplexity 559.04
epoch 1, 800/1000 batches, training loss 6.20, training perplexity 493.46
epoch 1, 900/1000 batches, training loss 6.17, training perplexity 478.56
epoch 1, 1000/1000 batches, training loss 6.16, training perplexity 471.33

epoch 1, validation loss 5.89, validation perplexity 362.39

epoch 2, 100/1000 batches, training loss 6.05, training perplexity 424.82
epoch 2, 200/1000 batches, training loss 5.98, training perplexity 396.19
```



```
epoch 5, 700/1000 batches, training loss 5.26, training perplexity 193.29
epoch 5, 800/1000 batches, training loss 5.13, training perplexity 169.04
epoch 5, 900/1000 batches, training loss 5.19, training perplexity 178.59
epoch 5, 1000/1000 batches, training loss 5.27, training perplexity 193.60

epoch 5, validation loss 5.32, validation perplexity 204.29
```

Figure 5.4 – Transformer training logs

Besides the cross-entropy loss, the perplexity is also reported. **Perplexity** is a popularly used metric in natural language processing to indicate how well a **probability distribution** (a language model, in our case) fits or predicts a sample. The lower the perplexity, the better the model is at predicting the sample. Mathematically, perplexity is just the exponential of the cross-entropy loss. Intuitively, this metric is used to indicate how perplexed or confused the model is while making predictions.

Once the model has been trained, we can conclude this exercise by evaluating the model's performance on the test set:

Copy Explain

```
testing_loss = eval_model(best_model_so_far, testing_data)
print(f"testing loss {testing_loss:.2f}, testing perplexity
{math.exp(testing_loss):.2f}")
```

This should result in the following output:

```
testing loss 5.23, testing perplexity 187.45
```

Figure 5.5 – Transformer evaluation results

In this exercise, we built a transformer model using PyTorch for the task of language modeling. We explored the transformer architecture in detail and how it is implemented in PyTorch. We used the `WikiText-2` dataset and `torchtext` functionalities to load and process the dataset. We then trained the transformer model for `5` epochs and evaluated it on a separate test set. This shall provide us with all the information we need to get started on working with transformers.

Besides the original transformer model, which was devised in 2017, a number of successors have since been developed over the years, especially around the field of language modeling, such as the following:

Bidirectional Encoder Representations from Transformers (BERT), 2018

Generative Pretrained Transformer (GPT), 2018

GPT-2, 2019

Conditional Transformer Language Model (CTRL), 2019

Transformer-XL, 2019

Distilled BERT (DistilBERT), 2019

Robustly optimized BERT pretraining Approach (RoBERTa), 2019

GPT-3, 2020

While we will not cover these models in detail in this chapter, you can nonetheless get started with using these models with PyTorch thanks to the `transformers` library, developed by HuggingFace 5.2 .We will explore HuggingFace in detail in Chapter 19. The transformers library provides pre-trained transformer family models for various tasks, such as language modeling, text classification, translation, question-answering, and so on.

Besides the models themselves, it also provides tokenizers for the respective models. For example, if we wanted to use a pre-trained BERT model for language modeling, we would need to write the following code once we have installed the `transformers` library:

```
import torch
from transformers import BertForMaskedLM, BertTokenizer
bert_model = BertForMaskedLM.from_pretrained('bert-base-uncased')
token_gen = BertTokenizer.from_pretrained('bert-base-uncased')
ip_sequence = token_gen("I love PyTorch !", return_tensors="pt")["input_ids"]
op = bert_model(ip_sequence, labels=ip_sequence)
total_loss, raw_preds = op[:2]
```

As we can see, it takes just a couple of lines to get started with a BERT-based language model. This demonstrates the power of the PyTorch ecosystem. You are encouraged to explore this with more complex variants, such as *Distilled BERT* or *RoBERTa*, using the `transformers` library. For more details, please refer to their GitHub page, which was mentioned previously.

This concludes our exploration of transformers. We did this by both building one from scratch as well as by reusing pre-trained models. The invention of transformers in the natural language processing space has been paralleled with the ImageNet moment in the field of computer vision, so this is going to be an active area of research. PyTorch will have a crucial role to play in the research and deployment of these types of models.

In the next and final section of this chapter, we will resume the neural architecture search discussions we provided at the end of *Chapter 3, Deep CNN Architectures*, where we briefly discussed the idea of generating optimal network architectures. We will explore a type of model where we do not decide what the model architecture will look like, and instead run a network generator that will find an optimal architecture for the given task. The resultant network is called a **randomly wired neural network** (**RandWireNN**) and we will develop one from scratch using PyTorch.

Developing a RandWireNN model from scratch

We discussed EfficientNets in *Chapter 3, Deep CNN Architectures*, where we explored the idea of finding the best model architecture instead of specifying it manually. RandWireNNs, or randomly wired neural networks, as the name suggests, are built on

a similar concept. In this section, we will study and build our own RandWireNN model using PyTorch.

Understanding RandWireNNs

First, a random graph generation algorithm is used to generate a random graph with a predefined number of nodes. This graph is converted into a neural network by a few definitions being imposed on it, such as the following:

Directed: The graph is restricted to be a directed graph, and the direction of edge is considered to be the direction of data flow in the equivalent neural network.

Aggregation: Multiple incoming edges to a node (or neuron) are aggregated by weighted sum, where the weights are learnable.

Transformation: Inside each node of this graph, a standard operation is applied: ReLU followed by 3x3 separable convolution (that is, a regular 3x3 convolution followed by a 1x1 pointwise convolution), followed by batch normalization. This operation is also referred to as a **ReLU-Conv-BN triplet.**

Distribution: Lastly, multiple outgoing edges from each neuron carry a copy of the aforementioned triplet operation.

One final piece in the puzzle is to add a single input node (source) and a single output node (sink) to this graph in order to fully transform the random graph into a neural network. Once the graph is realized as a neural network, it can be trained for various machine learning tasks.

In the **ReLU-Conv-BN triplet unit**, the output number of channels/features are the same as the input number of channels/features for repeatability reasons. However, depending on the type of task at hand, you can stage several of these graphs with an increasing number of channels downstream (and decreasing spatial size of the data/images). Finally, these staged graphs can be connected to each other by connecting the sink of one to the source of the other in a sequential manner.

Next, in the form of an exercise, we will build a RandWireNN model from scratch using PyTorch.

Developing RandWireNNs using PyTorch

We will now develop a RandWireNN model for an image classification task. This will be performed on the CIFAR-10 dataset. We will start from an empty model, generate a random graph, transform it into a neural network, train it for the given task on the given dataset, evaluate the trained model, and finally explore the resulting model that was generated. In this exercise, we will only show the important parts of the code for demonstration purposes. In order to access the full code, visit the book's github repo [5.3] .

Defining a training routine and loading data

In the first sub-section of this exercise, we will define the training function that will be called by our model training loop and define our dataset loader, which will provide us with batches of data for training. Let's get started:

First, we need to import some libraries. Some of the new libraries that will be used in this exercise are as follows:

Copy Explain

```
from torchviz import make_dot
import networkx as nx
```

Next, we must define the training routine, which takes in a trained model that can produce prediction probabilities given an RGB input image:

Copy Explain

```
def train(model, train_dataloader, optim, loss_func, epoch_num, lrate):
    for training_data, training_label in train_dataloader:
        pred_raw = model(training_data)
        curr_loss = loss_func(pred_raw, training_label)
        training_loss += curr_loss.data
    return training_loss / data_size, training_accuracy / data_size
```

Next, we define the dataset loader. We will use the `CIFAR-10` dataset for this image classification task, which is a well-known database of 60,000 32x32 RGB images labeled across 10 different classes containing 6,000 images per class. We will use the `torchvision.datasets` module to directly load the data from the torch dataset repository.

Dataset Citation

Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

The code is as follows:

Copy Explain

```
def load_dataset(batch_size):
    train_dataloader = torch.utils.data.DataLoader(
        datasets.CIFAR10('dataset', transform=transform_train_dataset, train=True,
download=True),
        batch_size=batch_size,  shuffle=True)
    return train_dataloader, test_dataloader
train_dataloader, test_dataloader = load_dataset(batch_size)
```

This should give us the following output:

```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to dataset/cifar-10-python.tar.gz
                                                                  170500096/? [03:10<00:00, 889623.81it/s]
Extracting dataset/cifar-10-python.tar.gz to dataset
```

Figure 5.6 – RandWireNN data loading

We will now move on to designing the neural network model. For this, we will need to design the randomly wired graph.

Defining the randomly wired graph

In this section, we will define a graph generator in order to generate a random graph that will be later used as a neural network. Let's get started:

As shown in the following code, we must define the random graph generator class:

Copy Explain

```
class RndGraph(object):
    def __init__(self, num_nodes, graph_probability, nearest_neighbour_k=4,
num_edges_attach=5):
    def make_graph_obj(self):
        graph_obj = nx.random_graphs.connected_watts_strogatz_graph(self.num_nodes,
self.nearest_neighbour_k,self.graph_probability)
        return graph_obj
```

In this exercise, we'll be using a well-known random graph model – the **Watts Strogatz (WS)** model. This is one of the three models that was experimented on in the original research paper about RandWireNNs. In this model, there are two parameters:

The number of neighbors for each node (which should be strictly even), *K*

A rewiring probability, *P*

First, all the *N* nodes of the graph are organized in a ring fashion and each node is connected to *K/2* nodes to its left and *K/2* to its right. Then, we traverse each node clockwise *K/2* times. At the *m*th traversal (*0<m<K/2*), the edge between the current node and its *m*th neighbor to the right is *rewired* with a probability, *P*.

Here, rewiring means that the edge is replaced by another edge between the current node and another node different from itself, as well as the *m*th neighbor. In the preceding code, the `make_graph_obj` method of our random graph generator class instantiates the WS graph model using the `networkx` library.

In the preceding code, the make_graph_obj method of our random graph generator class instantiates the WS graph model using the networkx library.

Furthermore, we add a `get_graph_config` method to return the list of nodes and edges in the graph. This will come in handy while we're transforming the abstract graph into a neural network. We will also define some graph saving and loading

methods for caching the generated graph both for reproducibility and efficiency reasons:

```
    def get_graph_config(self, graph_obj):
        return node_list, incoming_edges
    def save_graph(self, graph_obj, path_to_write):
        nx.write_yaml(graph_obj, "./cached_graph_obj/" + path_to_write)
    def load_graph(self, path_to_read):
        return nx.read_yaml("./cached_graph_obj/" + path_to_read)
```

Next, we will work on creating the actual neural network model.

Defining RandWireNN model modules

Now that we have the random graph generator, we need to transform it into a neural network. But before that, we will design some neural modules to facilitate that transformation. Let's get started:

Starting from the lowest level of the neural network, first, we will define a separable 2D convolutional layer, as follows:

```
class SepConv2d(nn.Module):
    def __init__(self, input_ch, output_ch, kernel_length=3, dilation_size=1,
padding_size=1, stride_length=1, bias_flag=True):
        super(SepConv2d, self).__init__()
        self.conv_layer = nn.Conv2d(input_ch, input_ch, kernel_length, stride_length,
padding_size, dilation_size, bias=bias_flag, groups=input_ch)
        self.pointwise_layer = nn.Conv2d(input_ch, output_ch, kernel_size=1, stride=1,
padding=0, dilation=1, groups=1, bias=bias_flag)
    def forward(self, x):
        return self.pointwise_layer(self.conv_layer(x))
```

The separable convolutional layer is a cascade of a regular 3x3 2D convolutional layer followed by a pointwise 1x1 2D convolutional layer.

Having defined the separable 2D convolutional layer, we can now define the ReLU-Conv-BN triplet unit:

Copy Explain

```
class UnitLayer(nn.Module):
    def __init__(self, input_ch, output_ch, stride_length=1):
        self.unit_layer = nn.Sequential(
            nn.ReLU(),
            SepConv2d(input_ch, output_ch,
stride_length=stride_length),nn.BatchNorm2d(output_ch),nn.Dropout(self.dropout)
        )
    def forward(self, x):
        return self.unit_layer(x)
```

As we mentioned earlier, the triplet unit is a cascade of a ReLU layer, followed by a separable 2D convolutional layer, followed by a batch normalization layer. We must also add a dropout layer for regularization.

With the triplet unit in place, we can now define a node in the graph with all of the `aggregation`, `transformation`, and `distribution` functionalities we need, as discussed at the beginning of this exercise:

Copy Explain

```
class GraphNode(nn.Module):
    def __init__(self, input_degree, input_ch, output_ch, stride_length=1):
        self.unit_layer = UnitLayer(input_ch, output_ch, stride_length=stride_length)
    def forward(self, *ip):
        if len(self.input_degree) > 1:
            op = (ip[0] * torch.sigmoid(self.params[0]))
            for idx in range(1, len(ip)):
                op += (ip[idx] * torch.sigmoid(self.params[idx]))
            return self.unit_layer(op)
        else:
            return self.unit_layer(ip[0])
```

In the `forward` method, we can see that if the number of incoming edges to the node is more than `1`, then a weighted average is calculated and these weights are learnable parameters of this node. The triplet unit is applied to the weighted average and the transformed (ReLU-Conv-BN-ed) output is returned.

We can now consolidate all of our graph and graph node definitions in order to define a randomly wired graph class, as shown here:

```
class RandWireGraph(nn.Module):
    def __init__(self, num_nodes, graph_prob, input_ch, output_ch, train_mode,
graph_name):
        # get graph nodes and in edges
        rnd_graph_node = RndGraph(self.num_nodes, self.graph_prob)
        if self.train_mode is True:
            rnd_graph = rnd_graph_node.make_graph_obj()
            self.node_list, self.incoming_edge_list =
rnd_graph_node.get_graph_config(rnd_graph)
        else:
        # define source Node
        self.list_of_modules = nn.ModuleList([GraphNode(self.incoming_edge_list[0],
self.input_ch, self.output_ch,
stride_length=2)])
        # define the sink Node
self.list_of_modules.extend([GraphNode(self.incoming_edge_list[n], self.output_ch,
self.output_ch)
                                     for n in self.node_list if n > 0])
```

In the `__init__` method of this class, first, an abstract random graph is generated. Its list of nodes and edges are derived. Using the `GraphNode` class, each abstract node of this abstract random graph is encapsulated as a neuron of the desired neural network. Finally, a source or input node and a sink or an output node are added to the network to make the neural network ready for the image classification task.

The `forward` method is also unconventional, as shown here:

```
    def forward(self, x):
        # source vertex
        op = self.list_of_modules[0].forward(x)
        mem_dict[0] = op
        # the rest of the vertices
        for n in range(1, len(self.node_list) - 1):
            if len(self.incoming_edge_list[n]) > 1:
                op = self.list_of_modules[n].forward(*[mem_dict[incoming_vtx]
                                                       for incoming_vtx in
self.incoming_edge_list[n]])
            mem_dict[n] = op
        for incoming_vtx in range(1, len(self.incoming_edge_list[self.num_nodes + 1])):
            op += mem_dict[self.incoming_edge_list[self.num_nodes + 1][incoming_vtx]]
        return op / len(self.incoming_edge_list[self.num_nodes + 1])
```

First, a forward pass is run for the source neuron, and then a series of forward passes are run for the subsequent neurons based on the `list_of_nodes` for the graph. The individual forward passes are executed using `list_of_modules`. Finally, the forward pass through the sink neuron gives us the output of this graph.

Next, we will use these defined modules and the randomly wired graph class to build the actual RandWireNN model class.

Transforming a random graph into a neural network

In the previous step, we defined one randomly wired graph. However, as we mentioned at the beginning of this exercise, a randomly wired neural network consists of several staged randomly wired graphs. The rationale behind that is to have a different (increasing) number of channels/features as we progress from the input neuron to the output neuron in an image classification task. This would be impossible with just one randomly wired graph because the number of channels is constant through one such graph, by design. Let's get started:

In this step, we define the ultimate randomly wired neural network. This will have three randomly wired graphs cascaded next to each other. Each graph will have double the number of channels compared to the previous graph to help us align with the general practice of increasing the number of channels (while downsampling spatially) in an image classification task:

Copy Explain

```
class RandWireNNModel(nn.Module):
    def __init__(self, num_nodes, graph_prob, input_ch, output_ch, train_mode):
        self.conv_layer_1 = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=self.output_ch, kernel_size=3,
padding=1),
            nn.BatchNorm2d(self.output_ch) )
        self.conv_layer_2 = …
        self.conv_layer_3 = …
        self.conv_layer_4 = …
        self.classifier_layer = nn.Sequential(
            nn.Conv2d(in_channels=self.input_ch*8, out_channels=1280, kernel_size=1),
nn.BatchNorm2d(1280))
        self.output_layer = nn.Sequential(nn.Dropout(self.dropout), nn.Linear(1280,
self.class_num))
```

The `__init__` method starts with a regular 3x3 convolutional layer, followed by three staged randomly wired graphs with channels that double in terms of numbers. This is followed by a fully connected layer that flattens the convolutional output from the last neuron of the last randomly wired graph into a vector that's `1280` in size.

Finally, another fully connected layer produces a 10-sized vector containing the probabilities for the 10 classes, as follows:

Copy Explain

```
    def forward(self, x):
        x = self.conv_layer_1(x)
        x = self.conv_layer_2(x)
        x = self.conv_layer_3(x)
        x = self.conv_layer_4(x)
        x = self.classifier_layer(x)
        # global average pooling
        _, _, h, w = x.size()
        x = F.avg_pool2d(x, kernel_size=[h, w])
        x = torch.squeeze(x)
        x = self.output_layer(x)
        return x
```

The `forward` method is quite self-explanatory, besides the global average pooling that is applied right after the first fully connected layer. This helps reduce dimensionality and the number of parameters in the network.

At this stage, we have successfully defined the RandWireNN model, loaded the datasets, and defined the model training routine. Now, we are all set to run the model training loop.

Training the RandWireNN model

In this section, we will set the model's hyperparameters and train the RandWireNN model. Let's get started:

We have defined all the building blocks for our exercise. Now, it is time to execute it. First, let's declare the necessary hyperparameters:

Copy Explain

```
num_epochs = 5
graph_probability = 0.7
node_channel_count = 64
num_nodes = 16
lrate = 0.1
batch_size = 64
train_mode = True
```

Having declared the hyperparameters, we instantiate the RandWireNN model, along with the optimizer and loss function:

Copy Explain

```
rand_wire_model = RandWireNNModel(num_nodes, graph_probability, node_channel_count,
node_channel_count, train_mode).to(device)
optim_module = optim.SGD(rand_wire_model.parameters(), lr=lrate, weight_decay=1e-4,
momentum=0.8)
loss_func = nn.CrossEntropyLoss().to(device)
```

Finally, we begin training the model. We're training the model for 5 epochs here for demonstration purposes, but you are encouraged to train for longer to see the boost in performance:

Copy Explain

```
for ep in range(1, num_epochs + 1):
    epochs.append(ep)
    training_loss, training_accuracy = train(rand_wire_model, train_dataloader,
optim_module, loss_func, ep, lrate)
    test_accuracy = accuracy(rand_wire_model, test_dataloader)
    test_accuracies.append(test_accuracy)
    training_losses.append(training_loss)
    training_accuracies.append(training_accuracy)
    if best_test_accuracy < test_accuracy:
        torch.save(model_state, './model_checkpoint/' + model_filename + 'ckpt.t7')
    print("model train time: ", time.time() - start_time)
```

This should result in the following output:

```
epoch 1, loss: 1.9863920211791992, accuracy: 25.0
epoch 1, loss: 1.762235641479492, accuracy: 31.25
epoch 1, loss: 1.6300958395004272, accuracy: 35.9375
epoch 1, loss: 1.685456395149231, accuracy: 37.5
epoch 1, loss: 1.506748080253601, accuracy: 50.0
epoch 1, loss: 1.3709843158721924, accuracy: 56.25
epoch 1, loss: 1.7547672986984253, accuracy: 29.6875
test acc: 43.81%, best test acc: 0.00%
model train time:  1922.020732164383
epoch 2, loss: 1.5338982343673706, accuracy: 42.1875
epoch 2, loss: 1.330839633941650 4, accuracy: 53.125
epoch 2, loss: 1.485781192779541, accuracy: 56.25
epoch 2, loss: 1.611755132675171, accuracy: 42.1875
epoch 2, loss: 1.2486891746520996, accuracy: 53.125
epoch 2, loss: 1.55242121219635, accuracy: 46.875
epoch 2, loss: 1.3306803703308105, accuracy: 57.8125
test acc: 48.21%, best test acc: 43.81%
model train time:  3945.885367870331
epoch 3, loss: 1.351543664932251, accuracy: 45.3125
                                  ⋮
epoch 5, loss: 1.042513132095337, accuracy: 62.5
test acc: 68.60%, best test acc: 63.73%
```

Figure 5.7 – RandWireNN training logs

It is evident from these logs that the model is progressively learning as the epochs progress. The performance on the validation set seems to be consistently increasing, which indicates model generalizability.

With that, we have created a model with no particular architecture in mind that can reasonably perform the task of image classification on the CIFAR-10 dataset.

Evaluating and visualizing the RandWireNN model

Finally, we will look at this model's test set performance before briefly exploring the model architecture visually. Let's get started:

Once the model has been trained, we can evaluate it on the test set:

Copy Explain

```
rand_wire_nn_model.load_state_dict(model_checkpoint['model'])
for test_data, test_label in test_dataloader:
    success += pred.eq(test_label.data).sum()
    print(f"test accuracy: {float(success) * 100. / len(test_dataloader.dataset)} %")
```

This should result in the following output:

```
best model accuracy: 67.73%, last epoch: 4
```

Figure 5.8 – RandWireNN evaluation results

The best performing model was found at the fourth epoch, with over 67% accuracy. Although the model is not perfect yet, we can train it for more epochs to achieve better performance. Also, a random model for this task would perform at an accuracy of 10% (because of 10 equally likely classes), so an accuracy of 67.73% is still promising, especially given the fact that we are using a randomly generated neural network architecture.

To conclude this exercise, let's look at the model architecture that was learned. The original image is too large to be displayed here. You can find the full image at our github repository both in .svg format [5.4] and in .pdf format [5.5] . In the following figure, we have vertically stacked three parts - the input section, a mid section and the output section, of the original neural network:

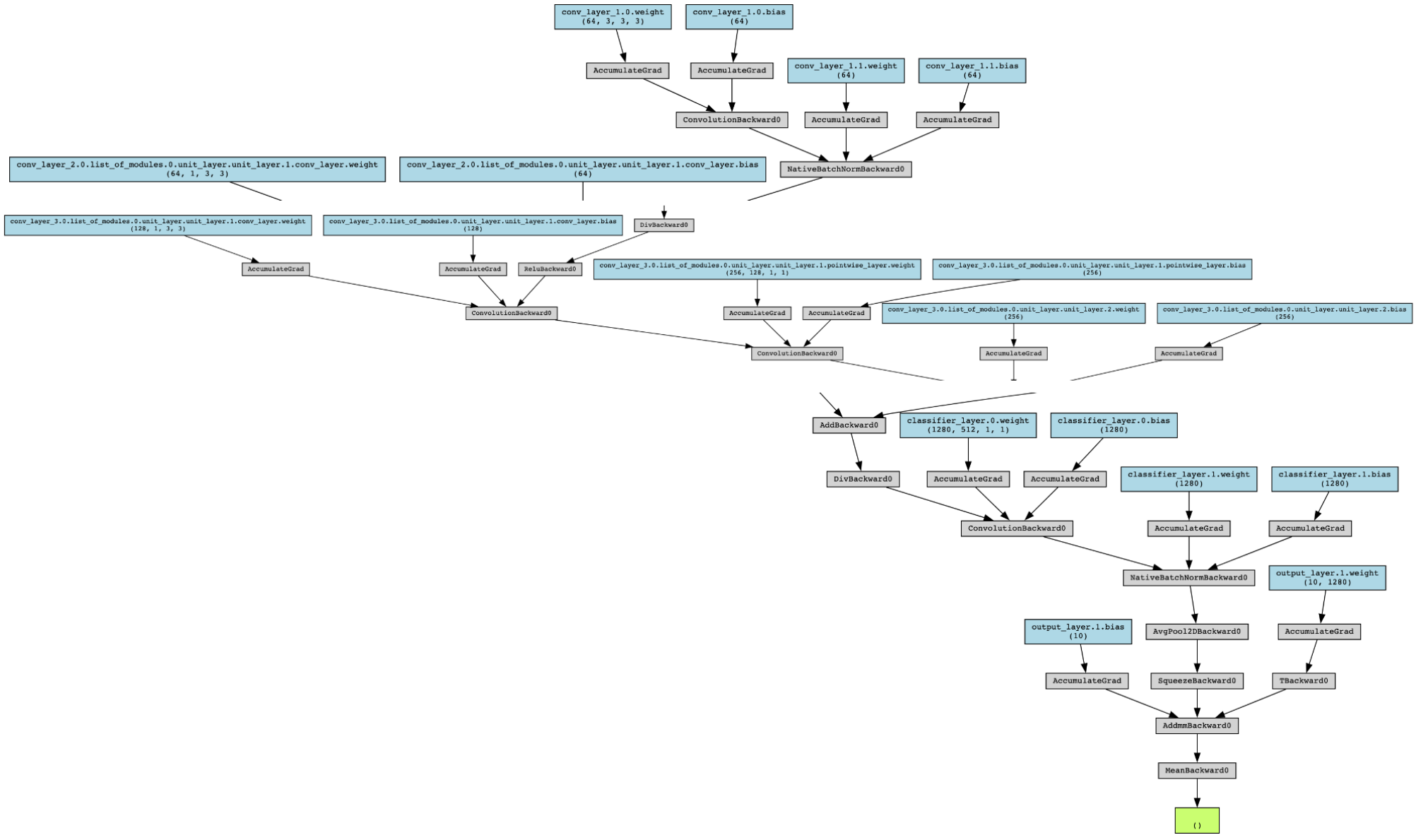

Figure 5.9 – RandWireNN architecture

From this graph, we can observe the following key points:

At the top, we can see the beginning of this neural network, which consists of a 64-channel 3x3 2D convolutional layer, followed by a 64-channel 1x1 pointwise 2D convolutional layer.

In the middle section, we can see the transition between the third- and fourth-stage random graphs, where we can see the sink neuron, `conv_layer_3`, of the stage 3 random graph followed by the source neuron `conv_layer_4`, of the stage 4 random graph.

Lastly, the lowermost section of the graph shows the final output layers – the sink neuron (a 512-channel separable 2D convolutional layer) of the stage 4 random graph, followed by a fully connected flattening layer, resulting in a 1,280-size feature vector, followed by a fully connected softmax layer that produces the 10 class probabilities.

Hence, we have built, trained, tested, and visualized a neural network model for image classification without specifying any particular model architecture. We did specify some overarching constraints over the structure, such as the penultimate feature vector length (`1280`), the number of channels in the separable 2D convolutional layers (`64`), the number of stages in the RandWireNN model (`4`), the definition of each neuron (ReLU-Conv-BN triplet), and so on.

However, we didn't specify what the structure of this neural network architecture should look like. We used a random graph generator to do this for us, which opens up an almost infinite number of possibilities in terms of finding optimal neural network architectures.

Neural architecture search is an ongoing and promising area of research in the field of deep learning. Largely, this fits in well with the field of training custom machine learning models for specific tasks, referred to as AutoML.

AutoML stands for **automated machine learning** as it does away with the necessity of having to manually load datasets, predefine a particular neural network model architecture to solve a given task, and manually deploy models into production systems. In *Chapter 16, PyTorch and AutoML*, we will discuss AutoML in detail and learn how to build such systems with PyTorch.

Summary

In this chapter, we looked at two distinct hybrid types of neural networks. First, we looked at the transformer model – the attention-only-based models with no recurrent connections that have outperformed all recurrent models on multiple sequential tasks. We ran through an exercise where we built, trained, and evaluated a transformer model on a language modeling task with the WikiText-2 dataset using PyTorch. In the second and final section of this chapter, we took up from where we left off in *Chapter 3, Deep CNN Architectures*, where we discussed the idea of optimizing for model architectures rather than optimizing for just the model parameters while fixing the architecture. We explored one of the approaches to do that – using randomly wired neural networks (RandWireNNs) – where we generated random graphs, assigned meanings to the nodes and edges of these graphs, and interconnected these graphs to form a neural network.

In the next chapter, we will switch gears and move away from model architectures and look at some interesting PyTorch applications. We will learn how to generate music and text through generative deep learning models using PyTorch.

Previous Chapter

Next Chapter