11

Training and Evaluation of Advanced ML Algorithms – GPT and Autoencoders

Classical **machine learning** (**ML**) and **neural networks** (**NNs**) are very good for classical problems – prediction, classification, and recognition. As we learned in the previous chapter, training them requires a moderate amount of data, and we train them for specific tasks. However, breakthroughs in ML and **artificial intelligence** (**AI**) in the late 2010s and the beginning of 2020s were about completely different types of models – **deep learning** (**DL**), **Generative Pre-Trained Transformers** (**GPTs**), and **generative AI** (**GenAI**).

GenAI models provide two advantages – they can generate new data and they can provide us with an internal representation of the data that captures the context of the data and, to some extent, its semantics. In the previous chapters, we saw how we can use existing models for inference and generating simple pieces of text.

In this chapter, we explore how GenAI models work based on GPT and Bidirectional Encoder Representations from Transformers (BERT) models. These models are designed to generate new data based on the patterns that they were trained on. We also look at the concept of autoencoders (AEs), where we train an AE to generate new images based on previously trained data.

In this chapter, we're going to cover the following main topics:

- From classical ML models to GenAI
- The theory behind GenAI models – AEs and transformers
- Training and evaluation of a **Robustly Optimized BERT Approach** (**RoBERTa**) model
- Training and evaluation of an AE
- Developing safety cages to prevent models from breaking the entire system

From classical ML to GenAI

Classical AI, also known as symbolic AI or rule-based AI, emerged as one of the earliest schools of thought in the field. It is rooted in the concept of explicitly encoding knowledge and using logical rules to manipulate symbols and derive intelligent behavior. Classical AI systems are designed to follow predefined rules and algorithms, enabling them to solve well-defined problems with precision and determinism. We delve into the underlying principles of classical AI, exploring its reliance on rule-based systems, expert systems, and logical reasoning.

In contrast, GenAI represents a paradigm shift in AI development, capitalizing on the power of ML and NNs to create intelligent systems that can generate new content, recognize patterns, and make informed decisions. Rather than relying on explicit rules and handcrafted knowledge, GenAI leverages data-driven approaches to learn from vast amounts of information and infer patterns and relationships. We examine the core concepts of GenAI, including DL, NNs, and probabilistic models, to unravel its ability to create original content and foster creative problem-solving.

One of the examples of a GenAI model is the GPT-3 model. GPT-3 is a state-of-the-art language model developed by OpenAI. It is based on the transformer architecture. GPT-3 is trained using a technique called **unsupervised learning** (**UL**), which enables it to generate coherent and contextually relevant text.

The theory behind advanced models – AEs and transformers

One of the large limitations of classical ML models is the access to annotated data. Large NNs contain millions (if not billions) of parameters, which means that they require equally many labeled data points to be trained correctly. Data labeling, also known as annotation, is the most expensive activity in ML, and therefore it is the labeling process that becomes the de facto limit of ML models. In the early 2010s, the solution to that problem was to use crowdsourcing.

Crowdsourcing, which is a process of collective data collection (among other things), means that we use users of our services to label the data. A CAPTCHA is one of the most prominent examples. A CAPTCHA is used when we need to recognize images in order to log in to a service. When we introduce new images, every time a user needs to recognize these images, we can label a lot of data in a relatively short time.

There is, nevertheless, an inherent problem with that process. Well, there are a few problems, but the most prominent one is that this process works mostly with images or similar kinds of data. It is also a relatively limited process – we can only ask users to recognize an image, but not add a semantic map and not draw a bounding box over an image. We cannot ask users to assess the similarity of images or any other, bit more advanced, task.

Here enter more advanced methods – GenAI and networks such as **generative adversarial networks** (**GANs**). These networks are designed to generate data and learn which data is like the original data.

These networks are very powerful and have been used in such applications as the generation of images; for example, in the so-called "deep fakes."

AEs

One of the main components of such a model is the AE, which is designed to learn a compressed representation (encoding) of the input data and then reconstruct the original data (decoding) from this compressed representation.

The architecture of an AE (*Figure 11.1*) consists of two main components: an encoder and a decoder. The encoder takes the input data and maps it to a lower-dimensional latent space representation, often referred to as the encoding/embedding or latent representation. The decoder takes this encoded representation and reconstructs the original input data:

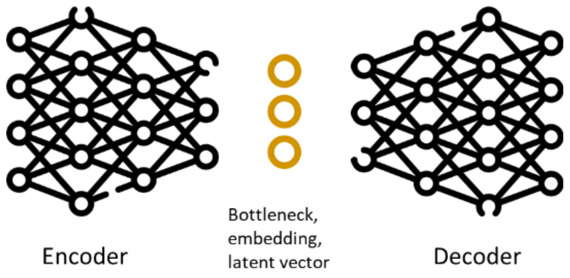

Figure 11.1 – High-level architecture of AEs

The objective of an AE is to minimize reconstruction errors, which is the difference between the input data and the output of the decoder. By doing so, the AE learns to capture the most important features of the input data in the latent representation, effectively compressing the information. The most interesting part is the latent space or encoding. This part allows this model to learn the representation of a complex data point (for example, an image) in a small vector of just a few numbers. The latent representation learned by the AE can be considered a compressed representation or a low-dimensional embedding of the input data. This compressed representation can be used for various purposes, such as data visualization, dimensionality reduction, anomaly detection, or as a starting point for other downstream tasks.

The encoder part calculates the latent vector, and the decoder part can expand it to an image. There are different types of AEs; the most interesting one is the **variational AE** (**VAE**), which encodes the parameters of a function that can generate new data rather than the representation of the data itself. In this way, it can create new data based on the distribution. In fact, it can even create completely new types of data by combining different functions.

Transformers

In **natural language processing** (**NLP**) tasks, we usually employ a bit different type of GenAI – transformers. Transformers revolutionized the field of machine translation but have been applied to many other tasks, including language understanding and text generation.

At its core, a transformer employs a self-attention mechanism that allows the model to weigh the importance of different words or tokens in a sequence when processing them. This attention mechanism enables the model to capture long-range dependencies and contextual relationships between words more effectively than traditional **recurrent NNs** (**RNNs**) or **convolutional NNs** (**CNNs**).

Transformers consist of an encoder-decoder structure. The encoder processes the input sequence, such as a sentence, and the decoder generates an output sequence, often based on the input and a target sequence. Two elements are unique for transformers:

- **Multi-head self-attention (MHSA)**: A mechanism that allows the model to attend to different positions in the input sequence simultaneously, capturing different types of dependencies. This is an extension to the RNN architecture, which was able to connect neurons in the same layer, thus capturing temporal dependencies.
- **Positional encoding**: To incorporate positional information into the model, positional encoding vectors are added to the input embeddings. These positional encodings are based on the tokens and their relative position to one another. This mechanism allows us to capture the context of a specific token and therefore to capture the basic contextual semantics of the text.

Figure 11.2 presents the high-level architecture of transformers:

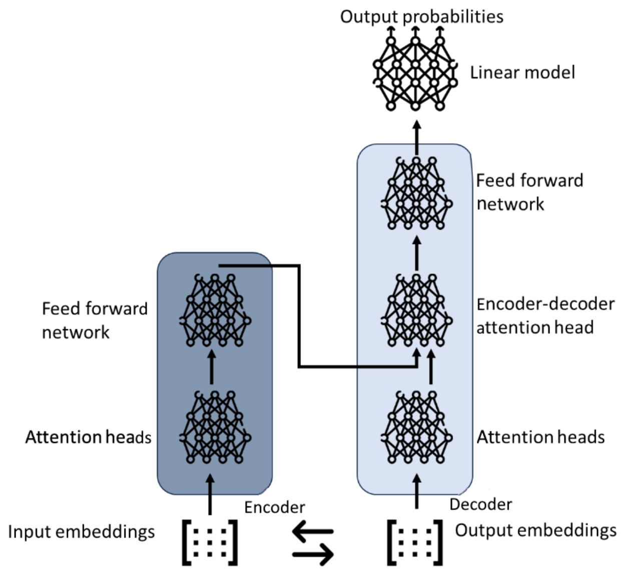

Figure 11.2 – High-level architecture of transformers

In this architecture, self-attention is a key mechanism that allows the model to weigh the importance of different words or tokens in a sequence when processing them. The self-attention mechanism is applied independently to each word in the input sequence, and it helps capture contextual relationships and dependencies between words. The term *head* refers to an independent attention mechanism that operates in parallel. Multiple self-attention heads can be used in the transformer model to capture different types of relationships (although we do not know what these relationships are).

Each self-attention head operates by computing attention scores between query representations and key representations. These attention scores indicate the importance or relevance of each word in the sequence with respect to the others. Attention scores are obtained by taking the dot product between the query and key representations, followed by applying a softmax function to normalize the scores.

The attention scores are then used to weigh the value representations. The weighted values are summed together to obtain the output representation for each word in the sequence.

The feed-forward networks in transformers serve two main purposes: feature extraction and position-wise representation. The feature extraction extracts higher-level features from the self-attention outputs, in a way that is very similar to the word-embeddings extraction that we learned previously. By applying non-linear transformations, the model can capture complex patterns and dependencies in the input sequence. The position-wise representation ensures that the model can learn different transformations for each position. It allows the model to learn complex representations of the sentences and therefore capture the more complex context of each word and sentence.

The transformer architecture is the basis for modern models such as GPT-3, which is a pre-trained generative transformer; that is, a transformer that has been pre-trained on a large mass of text. However, it is based on models such as BERT and its relatives.

Training and evaluation of a RoBERTa model

In general, the training process for GPT-3 involves exposing the model to a massive amount of text data from diverse sources, such as books, articles, websites, and more. By analyzing the patterns, relationships, and language structures within this data, the model learns to predict the likelihood of a word or phrase appearing based on the surrounding context. This learning objective is achieved through a process known as **masked language modeling** (**MLM**), where certain words are randomly masked in the input, and the model is tasked with predicting the correct word based on the context.

In this chapter, we train the RoBERTa model, which is a variation of the now-classical BERT model. Instead of using generic sources such as books and *Wikipedia* articles, we use programs. To make our training task a bit more specific, let us train a model that is capable of “understanding” code from a networking domain – WolfSSL, which is an open source implementation of the SSL protocol, used in many embedded software devices.

Once the training is complete, BERT models are capable of generating text by leveraging their learned knowledge and the context provided in a given prompt. When a user provides a prompt or a partial sentence, the model processes the input and generates a response by probabilistically predicting the most likely next word based on the context it has learned from the training data.

When it comes to GPT-3 (and similar) models, it is an extension of the BERT model. The generation process in GPT-3 involves multiple layers of attention mechanisms within the transformer architecture. These attention mechanisms allow the model to focus on relevant parts of the input text and make connections between different words and phrases, ensuring coherence and contextuality in the generated output. The model generates text by sampling or selecting the most probable next word at each step, taking into account previously generated words.

So, let us start our training process by preparing the data for training. First, we read the dataset:

```
from tokenizers import ByteLevelBPETokenizer

paths = ['source_code_wolf_ssl.txt']

print(f'Found {len(paths)} files')
print(f'First file: {paths[0]}')
```

This provides us with the raw training set. In this set, the text file contains all the source code from the WolfSSL protocol in one file. We do not have to prepare it like this, but it certainly makes the process easier as we only deal with one source file. Now, we can train the tokenizer, very similar to what we saw in the previous chapters:

```
# Initialize a tokenizer
tokenizer = ByteLevelBPETokenizer()

print('Training tokenizer...')

# Customize training
# we use a large vocabulary size, but we could also do with ca. 10_000
tokenizer.train(files=paths,
                vocab_size=52_000,
                min_frequency=2,
                special_tokens=["<s>","<pad>","</
s>","<unk>","<mask>",])
```

The first line initializes an instance of the `ByteLevelBPETokenizer` tokenizer class. This tokenizer is based on a byte-level version of the **Byte-Pair Encoding** (**BPE**) algorithm, which is a popular subword tokenization method. We discussed it in the previous chapters.

The next line prints a message indicating that the tokenizer training process is starting.

The `tokenizer.train()` function is called to train the tokenizer. The training process takes a few parameters:

- `files=paths`: This parameter specifies the input files or paths containing the text data to train the tokenizer. It expects a list of file paths.
- `vocab_size=52_000`: This parameter sets the size of the vocabulary; that is, the number of unique tokens the tokenizer will generate. In this case, the tokenizer will create a vocabulary of 52,000 tokens.
- `min_frequency=2`: This parameter specifies the minimum frequency a token must have in the training data to be included in the vocabulary. Tokens that occur less frequently than this threshold will be treated as **out-of-vocabulary** (**OOV**) tokens.
- `special_tokens=["<s>","<pad>","</s>","<unk>","<mask>"]`: This parameter defines a list of special tokens that will be added to the vocabulary. Special tokens are commonly used to represent specific meanings or special purposes. In this case, the special tokens are `<s>`, `<pad>`, `</s>`, `<unk>`, and `<mask>`. These tokens are often used in tasks such as machine translation, text generation, or language modeling.

Once the training process is completed, the tokenizer will have learned the vocabulary and will be able to encode and decode text using the trained subword units. We can now save the tokenizer using this piece of code:

```
import os

# we give this model a catchy name - wolfBERTa
# because it is a RoBERTa model trained on the WolfSSL source code
token_dir = './wolfBERTa'

if not os.path.exists(token_dir):
  os.makedirs(token_dir)

tokenizer.save_model('wolfBERTa')
```

We also test this tokenizer using the following line: `tokenizer.encode("int main(int argc, void **argv)").tokens`.

Now, let us make sure that the tokenizer is comparable with our model in the next step. To do that, we need to make sure that the output of the tokenizer never exceeds the number of tokens that the model can accept:

```
from tokenizers.processors import BertProcessing

# let's make sure that the tokenizer does not provide more tokens than
we expect
```

```
# we expect 512 tokens, because we will use the BERT model
tokenizer._tokenizer.post_processor = BertProcessing(
    ("</s>", tokenizer.token_to_id("</s>")),
    ("<s>", tokenizer.token_to_id("<s>")),
)
tokenizer.enable_truncation(max_length=512)
```

Now, we can move over to preparing the model. We do this by importing the predefined class from the HuggingFace hub:

```
import the RoBERTa configuration
from transformers import RobertaConfig

# initialize the configuration
# please note that the vocab size is the same as the one in the
tokenizer.
# if it is not, we could get exceptions that the model and the
tokenizer are not compatible
config = RobertaConfig(
    vocab_size=52_000,
    max_position_embeddings=514,
    num_attention_heads=12,
    num_hidden_layers=6,
    type_vocab_size=1,
)
```

The first line, `from transformers import RobertaConfig`, imports the `RobertaConfig` class from the `transformers` library. The `RobertaConfig` class is used to configure the RoBERTa model. Next, the code initializes the configuration of the RoBERTa model. The parameters passed to the `RobertaConfig` constructor are as follows:

- `vocab_size=52_000`: This parameter sets the size of the vocabulary used by the RoBERTa model. It should match the vocabulary size used during the tokenizer training. In this case, the tokenizer and the model both have a vocabulary size of 52,000, ensuring they are compatible.
- `max_position_embeddings=514`: This parameter sets the maximum sequence length that the RoBERTa model can handle. It defines the maximum number of tokens in a sequence that the model can process. Longer sequences may need to be truncated or split into smaller segments. Please note that the input is 514, not 512 as the output of the tokenizer. This is caused by the fact that we leave the place from the starting and ending tokens.
- `num_attention_heads=12`: This parameter sets the number of attention heads in the **multi-head attention** (**MHA**) mechanism of the RoBERTa model. Attention heads allow the model to focus on different parts of the input sequence simultaneously.

- `num_hidden_layers=6`: This parameter sets the number of hidden layers in the RoBERTa model. These layers contain the learnable parameters of the model and are responsible for processing the input data.
- `type_vocab_size=1`: This parameter sets the size of the token type vocabulary. In models such as RoBERTa, which do not use the token type (also called a segment) embeddings, this value is typically set to 1.

The configuration object config stores all these settings and will be used later when initializing the actual RoBERTa model. Having the same configuration parameters as the tokenizer ensures that the model and tokenizer are compatible and can be used together to process text data properly.

It is worth noting that this model is rather small, compared to the 175 billion parameters of GPT-3. It has (only) 85 million parameters. However, it can be trained on a laptop with a moderately powerful GPU (any NVIDIA GPU with 6 GB of VRAM will do). The model is, nevertheless, much larger than the original BERT model from 2017, which had only six attention heads and a handful of millions of parameters.

Once the model is created, we need to initiate it:

```
# Initializing a Model From Scratch
from transformers import RobertaForMaskedLM

# initialize the model
model = RobertaForMaskedLM(config=config)

# let's print the number of parameters in the model
print(model.num_parameters())

# let's print the model
print(model)
```

The last two lines print out the number of parameters in the model (a bit over 85 million) and then the model itself. The output of that model is quite large, so we do not present it here.

Now that the model is ready, we need to go back to the dataset and prepare it for training. The simplest way is to reuse the previously trained tokenizer by reading it back from the folder, but with the changed class of that tokenizer so that it fits the model:

```
from transformers import RobertaTokenizer

# initialize the tokenizer from the file
tokenizer = RobertaTokenizer.from_pretrained("./wolfBERTa", max_
length=512)
```

Once this is done, we can read the dataset:

```
from datasets import load_dataset

new_dataset = load_dataset("text", data_files='./source_code_wolf_ssl.
txt')
```

The previous code fragment reads the same dataset that we used to train the tokenizer. Now, we will use the tokenizer to transform the dataset into a set of tokens:

```
tokenized_dataset = new_dataset.map(lambda x: tokenizer(x["text"]),
num_proc=8)
```

This takes a moment, but it gives us a moment to also reflect on the fact that this code takes advantage of the so-called map-reduce algorithm, which became a golden standard for processing large files at the beginning of the 2010s when the concept of big data was very popular. It is the `map()` function that utilizes that algorithm.

Now, we need to prepare the dataset for training by creating so-called masked input. Masked input is a set of sentences where words are replaced by the mask token (`<mask>` in our case). It can look something like the example in *Figure 11.3*:

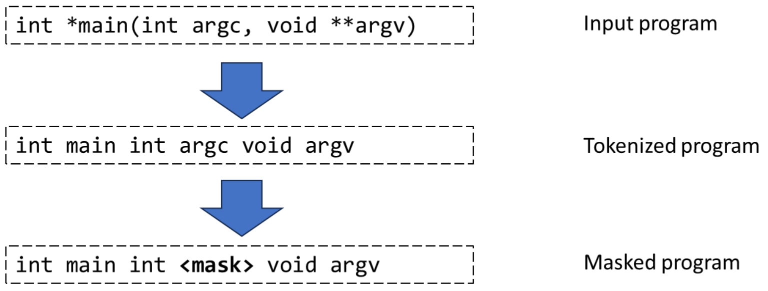
int *main(int argc, void **argv)
Input program
int main int argc void argv
Tokenized program
int main int <mask> void argv
Masked program

Figure 11.3 – Masked input for MLMs

It's easy to guess that the `<mask>` token can appear at any place and that it should appear several times in similar places in order for the model to actually learn the masked token's context. It would be very cumbersome to do it manually, and therefore, the HuggingFace library has a dedicated class for it – `DataCollatorForLanguageModeling`. The following code demonstrates how to instantiate that class and how to use its parameters:

```
from transformers import DataCollatorForLanguageModeling

data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer, mlm=True, mlm_probability=0.15
)
```

The from transformers import DataCollatorForLanguageModeling line imports the DataCollatorForLanguageModeling class, which is used for preparing data for language modeling tasks. The code initializes a DataCollatorForLanguageModeling object named data_collator. This object takes several parameters:

- tokenizer=tokenizer: This parameter specifies the tokenizer to be used for encoding and decoding the text data. It expects an instance of a tokenizer object. In this case, it appears that the tokenizer object has been previously defined and assigned to the tokenizer variable.
- mlm=True: This parameter indicates that the language modeling task is an MLM task.
- mlm_probability=0.15: This parameter sets the probability of masking a token in the input text. Each token has a 15% chance of being masked during data preparation.

The data_collator object is now ready to be used for preparing data for language modeling tasks. It takes care of tasks such as tokenization and masking of the input data to be compatible with the RoBERTa model. Now, we can instantiate another helper class – Trainer – which manages the training process of the MLM model:

```
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir="./wolfBERTa",
    overwrite_output_dir=True,
    num_train_epochs=10,
    per_device_train_batch_size=32,
    save_steps=10_000,
    save_total_limit=2,
)

trainer = Trainer(
    model=model,
    args=training_args,
    data_collator=data_collator,
    train_dataset=tokenized_dataset['train'],
)
```

The from transformers import Trainer, TrainingArguments line imports the Trainer class and the TrainingArguments class from the transformers library. Then it initializes a TrainingArguments object, training_args. This object takes several parameters to configure the training process:

- output_dir="./wolfBERTa": This parameter specifies the directory where the trained model and other training artifacts will be saved.

- `overwrite_output_dir=True`: This parameter determines whether to overwrite `output_dir` if it already exists. If set to `True`, it will overwrite the directory.
- `num_train_epochs=10`: This parameter sets the number of training epochs; that is, the number of times the training data will be iterated during training. In our example, it is enough with a few epochs only, such as 10. It takes a lot of time to train these models, so that's why we go with a small number of epochs.
- `per_device_train_batch_size=32`: This parameter sets the batch size per GPU for training. It determines how many training examples are processed together in parallel during each training step. If you do not have a lot of VRAM in your GPU, decrease this number.
- `save_steps=10_000`: This parameter specifies the number of training steps before saving a checkpoint of the model.
- `save_total_limit=2`: This parameter limits the total number of saved checkpoints. If the limit is exceeded, older checkpoints will be deleted.

After initializing the trainer arguments, the code initializes a `Trainer` object with the following arguments:

- `model=model`: This parameter specifies the model to be trained. In this case, the pre-initialized RoBERTa model from our previous steps is assigned to the model variable.
- `args=training_args`: This parameter specifies the training arguments, which we prepared in our previous steps.
- `data_collator=data_collator`: This parameter specifies the data collator to be used during training. This object was prepared previously in our code.
- `train_dataset=tokenized_dataset['train']`: This parameter specifies the training dataset. It appears that a tokenized dataset has been prepared and stored in a dictionary called `tokenized_dataset`, and the training portion of that dataset is assigned to `train_dataset`. In our case, since we did not define the train-test split, it take the entire dataset.

The `Trainer` object is now ready to be used for training the RoBERTa model using the specified training arguments, data collator, and training dataset. We do this by simply writing `trainer.train()`.

Once the model finishes training, we can save it using the following command: `trainer.save_model("./wolfBERTa")`. After that, we can use the model just as we learned in *Chapter 10*.

It takes a while to train the model; on a consumer-grade GPU such as NVIDIA 4090, it can take about one day for 10 epochs, but if we want to use a larger dataset or more epochs, it can take much longer. I do not advise executing this code on a computer without a GPU as it takes ca. 5-10 times longer than on a GPU. Hence my next best practice.

Best practice #57

Use NVIDIA **Compute Unified Device Architecture** (**CUDA**; accelerated computing) for training advanced models such as BERT, GPT-3, and AEs.

For classical ML, and even for simple NNs, a modern CPU is more than enough. The number of calculations is large, but not extreme. However, when it comes to training BERT models, AEs, and similar, we need acceleration for handling tensors (vectors) and making calculations on entire vectors at once. CUDA is NVIDIA's acceleration framework. It allows developers to utilize the power of NVIDIA GPUs to accelerate computational tasks, including training DL models. It provides a few benefits:

- **GPU parallelism**, designed to handle many parallel computations simultaneously. DL models, especially large models such as RoBERTa, consist of millions or even billions of parameters. Training these models involves performing numerous mathematical operations, such as matrix multiplications and convolutions, on these parameters. CUDA enables these computations to be parallelized across the thousands of cores present in a GPU, greatly speeding up the training process compared to a traditional CPU.
- **Optimized tensor operations for PyTorch or TensorFlow**, which are designed to work seamlessly with CUDA. These frameworks provide GPU-accelerated libraries that implement optimized tensor operations specifically designed for GPUs. Tensors are multi-dimensional arrays used to store and manipulate data in DL models. With CUDA, these tensor operations can be efficiently executed on the GPU, leveraging its high memory bandwidth and parallel processing capabilities.
- **High memory bandwidth**, which enables data to be transferred to and from the GPU memory at a much faster rate, enabling faster data processing during training. DL models often require large amounts of data to be loaded and processed in batches. CUDA allows these batches to be efficiently transferred and processed on the GPU, reducing training time.

By utilizing CUDA, DL frameworks can effectively leverage the parallel computing capabilities and optimized operations of NVIDIA GPUs, resulting in significant acceleration of the training process for large-scale models such as RoBERTa.

Training and evaluation of an AE

We mentioned AEs in *Chapter 7* when we discussed the process of feature engineering for images. AEs, however, are used to do much more than just image feature extraction. One of the major aspects of them is to be able to recreate images. This means that we can create images based on the placement of the image in the latent space.

So, let us train the AE model for a dataset that is pretty standard in ML – Fashion MNIST. We got to see what the dataset looks like in our previous chapters. We start our training by preparing the data in the following code fragment:

```
# Transforms images to a PyTorch Tensor
tensor_transform = transforms.ToTensor()

# Download the Fashion MNIST Dataset
dataset = datasets.FashionMNIST(root = "./data",
                         train = True,
                         download = True,
                         transform = tensor_transform)

# DataLoader is used to load the dataset
# for training
loader = torch.utils.data.DataLoader(dataset = dataset,
                                     batch_size = 32,
                                     shuffle = True)
```

It imports the necessary modules from the PyTorch library.

It defines a transformation called `tensor_transform` using `transforms.ToTensor()`. This transformation is used to convert images in the dataset to PyTorch tensors.

The code fragment downloads the dataset using the `datasets.FashionMNIST()` function. The `train` parameter is set to `True` to indicate that the downloaded dataset is for training purposes. The `download` parameter is set to `True` to automatically download the dataset if it is not already present in the specified directory.

Since we use the PyTorch framework with accelerated computing, we need to make sure that the image is transformed into a tensor. The `transform` parameter is set to `tensor_transform`, which is a transformer defined in the first line of the code fragment.

Then, we create a `DataLoader` object used to load the dataset in batches for training. The `dataset` parameter is set to the previously downloaded dataset. The `batch_size` parameter is set to `32`, indicating that each batch of the dataset will contain 32 images.

The `shuffle` parameter is set to `True` to shuffle the order of the samples in each epoch of training, ensuring randomization and reducing any potential bias during training.

Once we have the dataset prepared, we can create our AE, which we do like this:

```
# Creating a PyTorch class
# 28*28 ==> 9 ==> 28*28
class AE(torch.nn.Module):
    def __init__(self):
        super().__init__()

        # Building an linear encoder with Linear
        # layer followed by Relu activation function
        # 784 ==> 9
        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(28 * 28, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 36),
            torch.nn.ReLU(),
            torch.nn.Linear(36, 18),
            torch.nn.ReLU(),
            torch.nn.Linear(18, 9)
        )

        # Building an linear decoder with Linear
        # layer followed by Relu activation function
        # The Sigmoid activation function
        # outputs the value between 0 and 1
        # 9 ==> 784
        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(9, 18),
            torch.nn.ReLU(),
            torch.nn.Linear(18, 36),
            torch.nn.ReLU(),
            torch.nn.Linear(36, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 28 * 28),
            torch.nn.Sigmoid()
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded
```

First, we define a class named `AE` that inherits from the `torch.nn.Module` class, which is the base class for all NN modules in PyTorch. The `super().__init__()` line ensures that the initialization of the base class (`torch.nn.Module`) is called. Since AEs are a special kind of NN class with backpropagation learning, we can just inherit a lot of basic functionality from the library.

Then, we define the encoder part of the AE. The encoder consists of several linear (fully connected) layers with ReLU activation functions. Each `torch.nn.Linear` layer represents a linear transformation of the input data followed by an activation function. In this case, the input size is 28 * 28 (which corresponds to the dimensions of an image in the Fashion MNIST dataset), and the output size gradually decreases until it reaches 9, which is our latent vector size.

Then, we define the decoder part of the AE. The decoder is responsible for reconstructing the input data from the encoded representation. It consists of several linear layers with ReLU activation functions, followed by a final linear layer with a sigmoid activation function. The input size of the decoder is 9, which corresponds to the size of the latent vector space in the bottleneck of the encoder. The output size is 28 * 28, which matches the dimensions of the original input data.

The `forward` method defines the forward pass of the AE. It takes an `x` input and passes it through the encoder to obtain an encoded representation. Then, it passes the encoded representation through the decoder to reconstruct the input data. The reconstructed output is returned as the result. We are now ready to instantiate our AE:

```
# Model Initialization
model = AE()

# Validation using MSE Loss function
loss_function = torch.nn.MSELoss()

# Using an Adam Optimizer with lr = 0.1
optimizer = torch.optim.Adam(model.parameters(),
                             lr = 1e-1,
                             weight_decay = 1e-8)
```

In this code, we first instantiate our AE as our model. Then, we create an instance of the **Mean Squared Error** (**MSE**) loss function provided by PyTorch. MSE is a commonly used loss function for regression tasks. We need it to calculate the mean squared difference between the predicted values and the target values – which are the individual pixels in our dataset, providing a measure of how well the model is performing. *Figure 11.4* shows the role of the learning function in the process of training the AE:

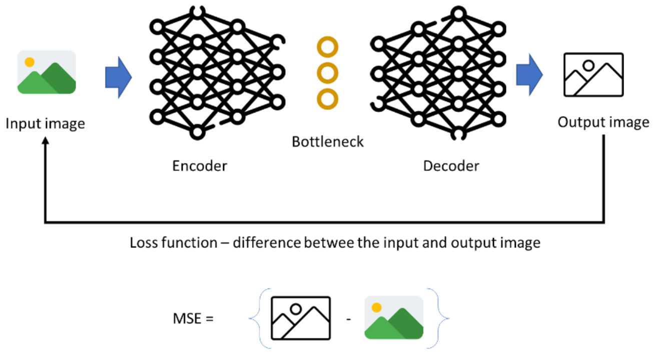
Input image
Encoder
Bottleneck
Decoder
Output image
Loss function – difference betwee the input and output image
MSE =

Figure 11.4 – Loss function (MSE) in the AE training process

Then, we initialize the optimizer used to update the model's parameters during training. In this case, the code creates an Adam optimizer, which is a popular optimization algorithm for training NNs. It takes three important arguments:

- `model.parameters()`: This specifies the parameters that will be optimized. In this case, it includes all the parameters of the model (the AE) that we created earlier.
- `lr=1e-1`: This sets the learning rate, which determines the step size at which the optimizer updates the parameters. A higher learning rate can lead to faster convergence but may risk overshooting the optimal solution, while a lower learning rate may converge more slowly but with potentially better accuracy.
- `weight_decay=1e-8`: This parameter adds a weight decay regularization term to the optimizer. Weight decay helps prevent overfitting by adding a penalty term to the loss function that discourages large weights. The `1e-8` value represents the weight decay coefficient.

With this code, we have now an instance of an AE to train. Now, we can start the process of training. We train the model for 10 epochs, but we can try more if needed:

```
epochs = 10
outputs = []
losses = []
for epoch in range(epochs):
    for (image, _) in loader:

      # Reshaping the image to (-1, 784)
      image = image.reshape(-1, 28*28)

      # Output of Autoencoder
      reconstructed = model(image)

      # Calculating the loss function
```

```
        loss = loss_function(reconstructed, image)

        # The gradients are set to zero,
        # the gradient is computed and stored.
        # .step() performs parameter update
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Storing the losses in a list for plotting
        losses.append(loss)
    outputs.append((epochs, image, reconstructed))
```

We start by iterating over the specified number of epochs for the training. Within each epoch, we iterate over the loader, which provides batches of image data and their corresponding labels. We do not use the labels, because the AE is a network to recreate images and not to learn what the images show – in that sense, it is an unsupervised model.

For each image, we reshape the input image data by flattening each image, originally in the shape of `(batch_size, 28, 28)`, into a 2D tensor of shape `(batch_size, 784)`, where each row represents a flattened image. The flattened image is created when we take each row of pixels and concatenate it to create one large vector. It is needed as the images are two-dimensional, while our tensor input needs to be of a single dimension.

Then, we obtain the reconstructed image using `reconstructed = model(image)`. Once we get the reconstructed image, we can calculate the MSE loss function and use that information to manage the next step of the learning (`optimizer.zero_grad()`). In the last line, we add this information to the list of losses per iteration so that we can create a learning diagram. We do it by using the following code fragment:

```
# Defining the Plot Style
plt.style.use('seaborn')
plt.xlabel('Iterations')
plt.ylabel('Loss')

# Convert the list to a PyTorch tensor
losses_tensor = torch.tensor(losses)

plt.plot(losses_tensor.detach().numpy()[::-1])
```

This results in a learning diagram, shown in *Figure 11.5*:

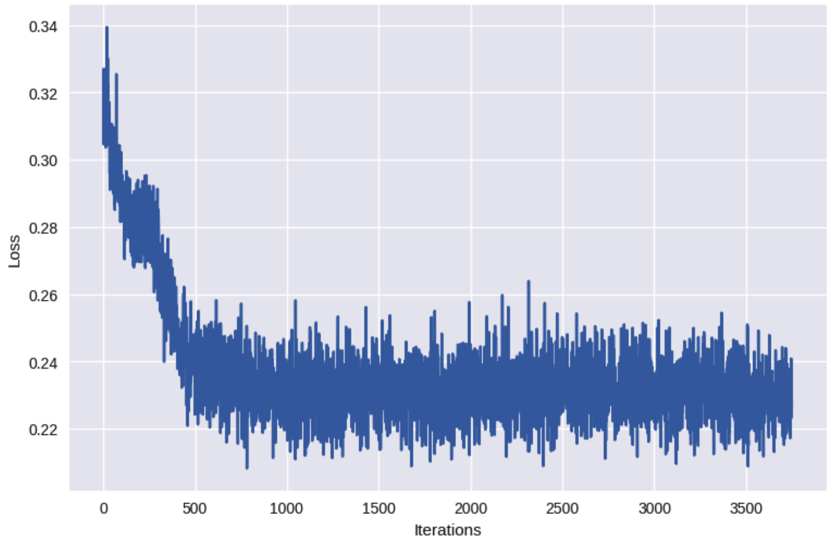

Figure 11.5 – Learning rate diagram from training our AE

The learning rate diagram shows that the AE is not really great yet and that we should train it a bit more. However, we can always check what the recreated images look like. We can do that using this code:

```
for i, item in enumerate(image):

  # Reshape the array for plotting
  item = item.reshape(-1, 28, 28)
  plt.imshow(item[0])
```

The code results in the output shown in *Figure 11.6*:

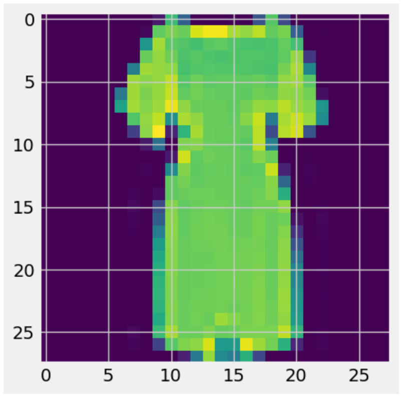

Figure 11.6 – Recreated image from our AE

Despite the learning rate, which is OK, we still can get very good results from our AEs.

Best practice #58

In addition to monitoring the loss, make sure to visualize the actual results of the generation.

Monitoring the loss function is a good way to understand when the AE stabilizes. However, just the loss function is not enough. I usually plot the actual output to understand whether the AE has been trained correctly.

Finally, we can visualize the learning process when we use this code:

```
yhat = model(image[0])

make_dot(yhat,
         params=dict(list(model.named_parameters())),
         show_attrs=True,
         show_saved=True)
```

This code visualizes the learning process of the entire network. It creates a large image, and we can only show a small excerpt of it. *Figure 11.7* shows this excerpt:

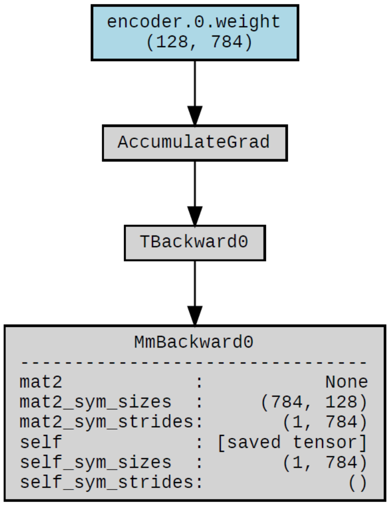

Figure 11.7 – The first three steps in training an AE, visualized as the AE architecture

We can even visualize the entire architecture in a text form by using the following code:

```
from torchsummary import summary
summary(model, (1, 28 * 28))
```

This results in the following model:

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1               [-1, 1, 128]         100,480
              ReLU-2               [-1, 1, 128]               0
            Linear-3                [-1, 1, 64]           8,256
              ReLU-4                [-1, 1, 64]               0
            Linear-5                [-1, 1, 36]           2,340
              ReLU-6                [-1, 1, 36]               0
            Linear-7                [-1, 1, 18]             666
              ReLU-8                [-1, 1, 18]               0
            Linear-9                 [-1, 1, 9]             171
           Linear-10                [-1, 1, 18]             180
             ReLU-11                [-1, 1, 18]               0
           Linear-12                [-1, 1, 36]             684
             ReLU-13                [-1, 1, 36]               0
           Linear-14                [-1, 1, 64]           2,368
             ReLU-15                [-1, 1, 64]               0
           Linear-16               [-1, 1, 128]           8,320
             ReLU-17               [-1, 1, 128]               0
           Linear-18               [-1, 1, 784]         101,136
          Sigmoid-19               [-1, 1, 784]               0
================================================================
Total params: 224,601
Trainable params: 224,601
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 0.02
Params size (MB): 0.86
Estimated Total Size (MB): 0.88
----------------------------------------------------------------
```

The bottleneck layer is in boldface, to illustrate the place where the encode and decode parts are linked to one another.

Developing safety cages to prevent models from breaking the entire system

As GenAI systems such as MLMs and AEs create new content, there is a risk that they generate content that can either break the entire software system or become unethical.

Therefore, software engineers often use the concept of a safety cage to guard the model itself from inappropriate input and output. For an MLM such as RoBERTa, this can be a simple preprocessor that checks whether the content generated is problematic. Conceptually, this is illustrated in *Figure 11.8*:

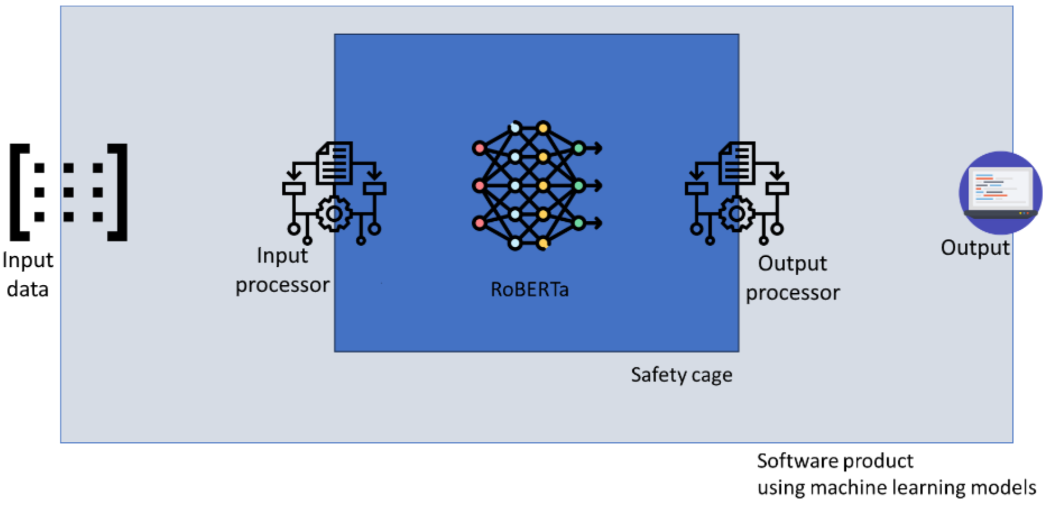

Figure 11.8 – Safety-cage concept for MLMs

In the example of the `wolfBERTa` model, this can mean that we check whether the generated code does not contain cybersecurity vulnerabilities, which can potentially allow hackers to take over our system. This means that all programs generated by the `wolfBERTa` model should be checked using tools such as SonarQube or CodeSonar to check for cybersecurity vulnerabilities, hence my next best practice.

Best practice #59

Check the output of GenAI models so that it does not break the entire system or provide unethical responses.

My recommendation to create such safety cages is to start from the requirements of the system. The first step is to understand what the system is going to do and understand which dangers and risks this task entails. The safety cage's output processor should ensure that these dangerous situations do not occur and that they are handled properly.

Once we understand how to prevent dangers, we can move over to conceptualizing how to prevent these risks on the language-model level. For example, when we train the model, we can select code that is known to be secure and does not contain security vulnerabilities. Although it does not guarantee that the model generates secure code, it certainly reduces the risk for it.

Summary

In this chapter, we learned how to train advanced models and saw that their training is not much more difficult than training classical ML models, which were described in *Chapter 10*. Even though the models that we trained are much more complex than the models in *Chapter 10*, we can use the same principles and expand this kind of activity to train even more complex models.

We focused on GenAI in the form of BERT models (fundamental GPT models) and AEs. Training these models is not very difficult, and we do not need huge computing power to train them. Our `wolfBERTa` model has ca. 80 million parameters, which seems like a lot, but the really good models, such as GPT-3, have billions of parameters – GPT-3 has 175 billion parameters, NVIDIA Turing has over 350 billion parameters, and GPT-4 is 1,000 times larger than GPT-3. The training process is the same, but we need a supercomputing architecture in order to train these models.

We have also learned that these models are only parts of larger software systems. In the next chapter, we learn how to create such a larger system.

References

- *Kratsch, W. et al., Machine learning in business process monitoring: a comparison of deep learning and classical approaches used for outcome prediction. Business & Information Systems Engineering, 2021, 63: p. 261-276.*
- *Vaswani, A. et al., Attention is all you need. Advances in neural information processing systems, 2017, 30.*
- *Aggarwal, A., M. Mittal, and G. Battineni, Generative adversarial network: An overview of theory and applications. International Journal of Information Management Data Insights, 2021. 1(1): p. 100004.*
- *Creswell, A., et al., Generative adversarial networks: An overview. IEEE signal processing magazine, 2018. 35(1): p. 53-65.*