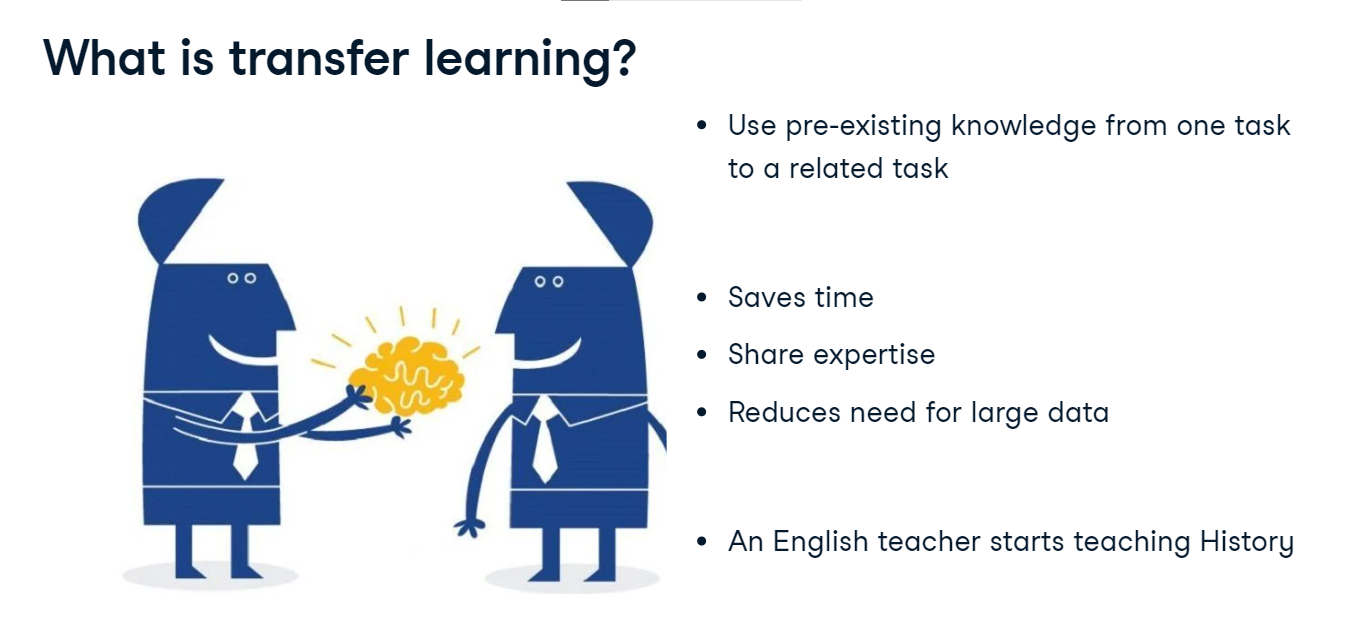
**Transfer learning for text classification**

Pre-trained models are trained on specific tasks, but we can use them for other tasks with transfer learning.

**What is transfer learning?**

Transfer learning is a powerful technique that uses pre-existing knowledge from one task to improve performance in a related task, saving time, making use of expertise from different domains, and reducing data requirements. For example, an English Literature teacher might find it easier to teach History due to the overlapping themes and narratives. We'll explore its application in text classification.

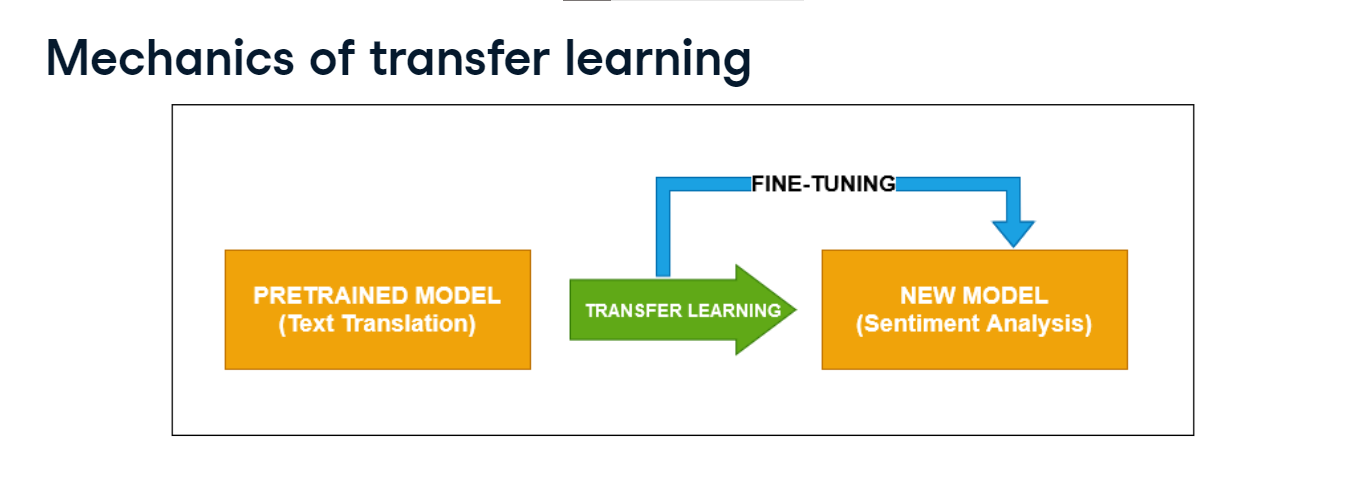


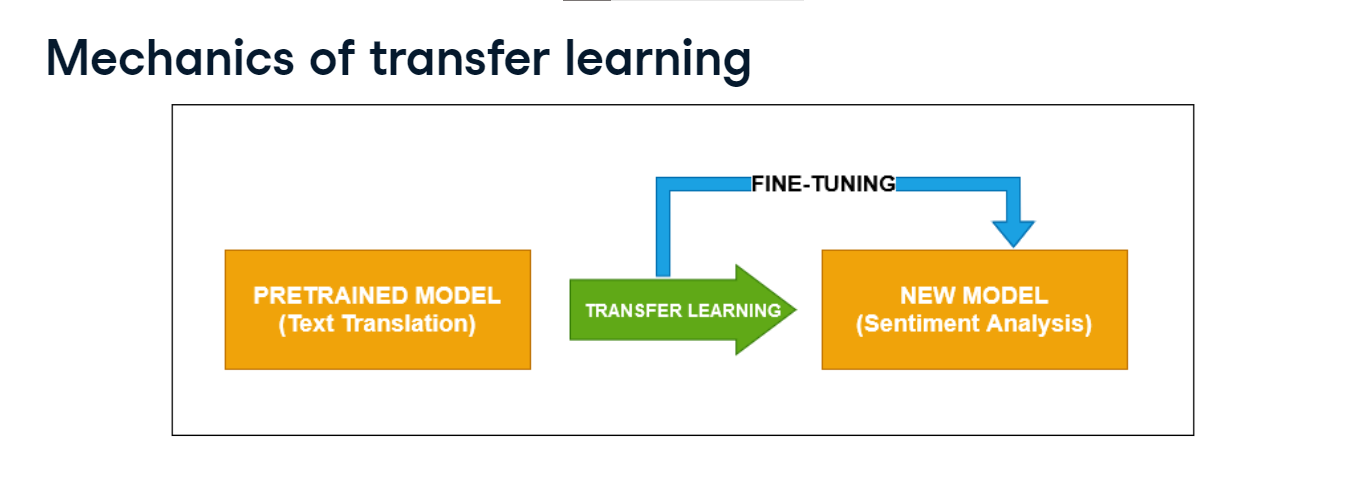
**Mechanics of transfer learning**

Transfer learning for text classification works in three main steps. We begin with a pre-trained model, which has learned patterns and features from its original task, such as text translation.

Now, instead of starting from scratch, we transfer this learned knowledge to a new, but related task like sentiment analysis.

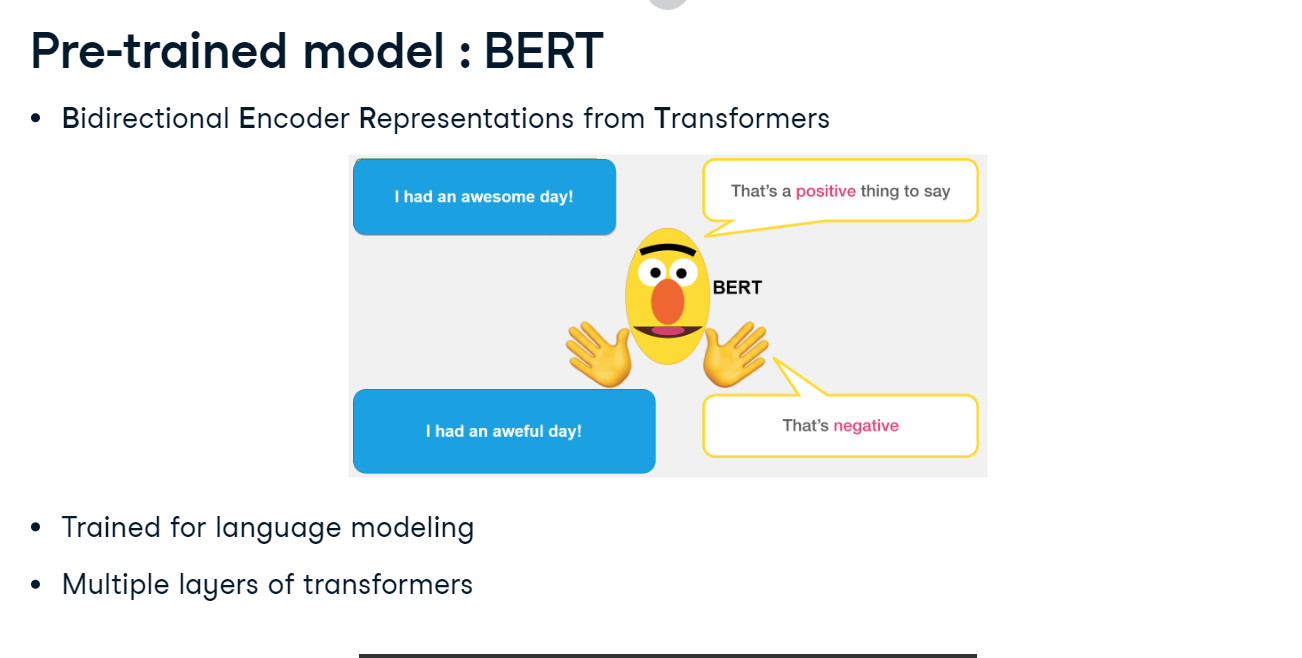
The final step is fine-tuning, where we adjust the model specifically for sentiment analysis by retraining the existing model with more details.

Finally, this creates a new model that can be used for sentiment analysis.

]

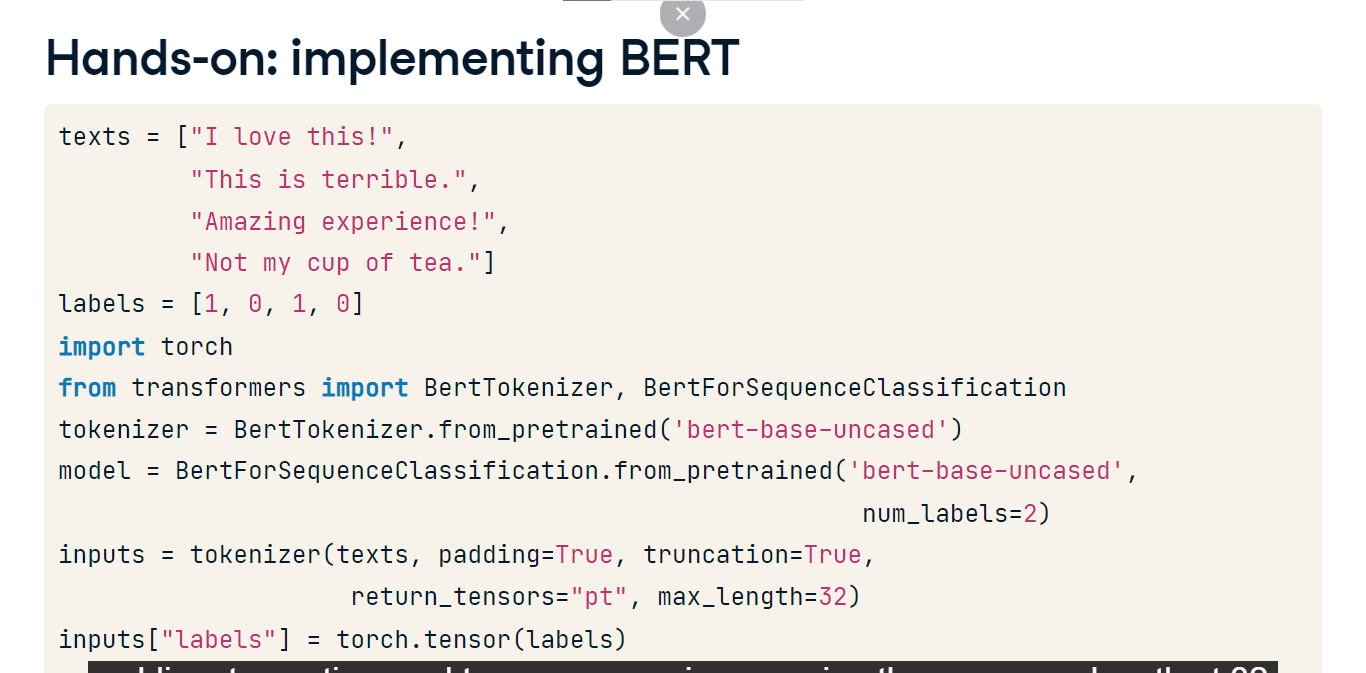
**Pre-trained model : BERT**

We have already used pretrained models in the previous chapter. Now, with the help of transfer learning we will use a language model, BERT, and tune it to work with sentiment analysis. BERT, or Bidirectional Encoder Representations from Transformers, is trained for language modeling. It contains multiple layers of transformers and is pre-trained on large texts.



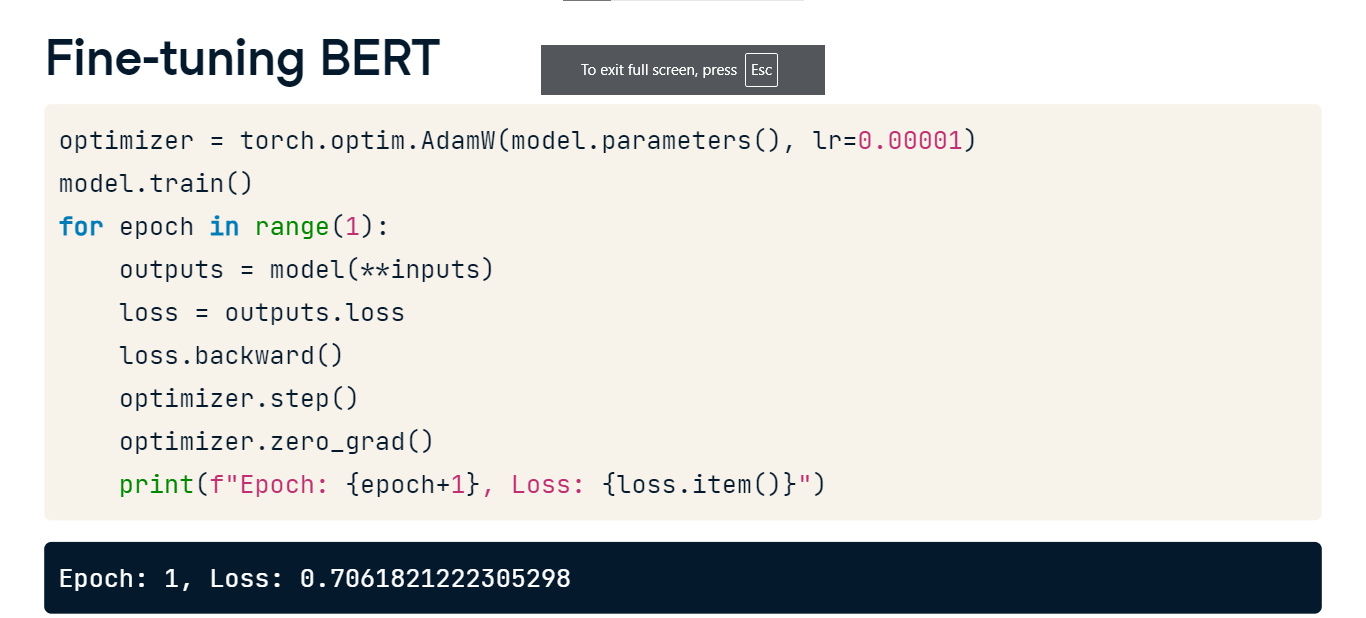
**Hands-on: implementing BERT**

We begin with texts labeled one for positive and zero for negative sentiments for training and testing. We import BertTokenizer and BertForSequenceClassification from the transformers package, the latter being apt for text classification, leveraging PyTorch. This is where transfer learning is pivotal. We initialize a BERT tokenizer and model using from\_pretrained and bert-base-uncased, designed for English texts, with num\_labels set to two for binary classification. Subsequently, our texts are prepared using the tokenizer, which handles padding, truncation, and tensor conversion, capping the sequence length at 32. Labels are assigned to the preprocessed inputs, preparing them for training and evaluation in the BERT model.



**Fine-tuning BERT**

We fine-tune using AdamW optimizer, a variant of Adam, with a learning rate of 0-point-00001 for subtle parameter adjustments. Initially, we use one epoch to verify training, increasing it as needed to reduce loss. Our model enters training mode, adopting the approach used for other text models. In each epoch, data is fed using the unpacked input\_eval variable. We then calculate the loss with the model’s loss attribute and find gradients with loss-dot-backward. Weights are adjusted with optimizer-dot-step and gradients are reset with optimizer-dot-zero\_grad to avoid past interference. Observing loss per epoch, starting at zero-dot-706, is crucial; the goal is its reduction through successive epochs.



**Evaluating on new text**

Testing the fine-tuned BERT involves tokenizing new text, ensuring it complies with the model’s constraints using return\_underscore\_tensors, truncation, padding, and max\_length. Tokenized input is dynamically inputted to the model via -asterisk-asterisk-input\_eval. Interpretation of the model’s output involves applying the softmax function on outputs\_eval-dot-logits, translating logits, or model outputs, into probabilities between zero and one, with dim=-1 ensuring application over the tensor's last dimension. The classification of the text as positive or negative corresponds to the highest probability, allowing the model to determine the sentiment accurately. Our model accurately identifies the text as positive.



**Transformers for text processing**

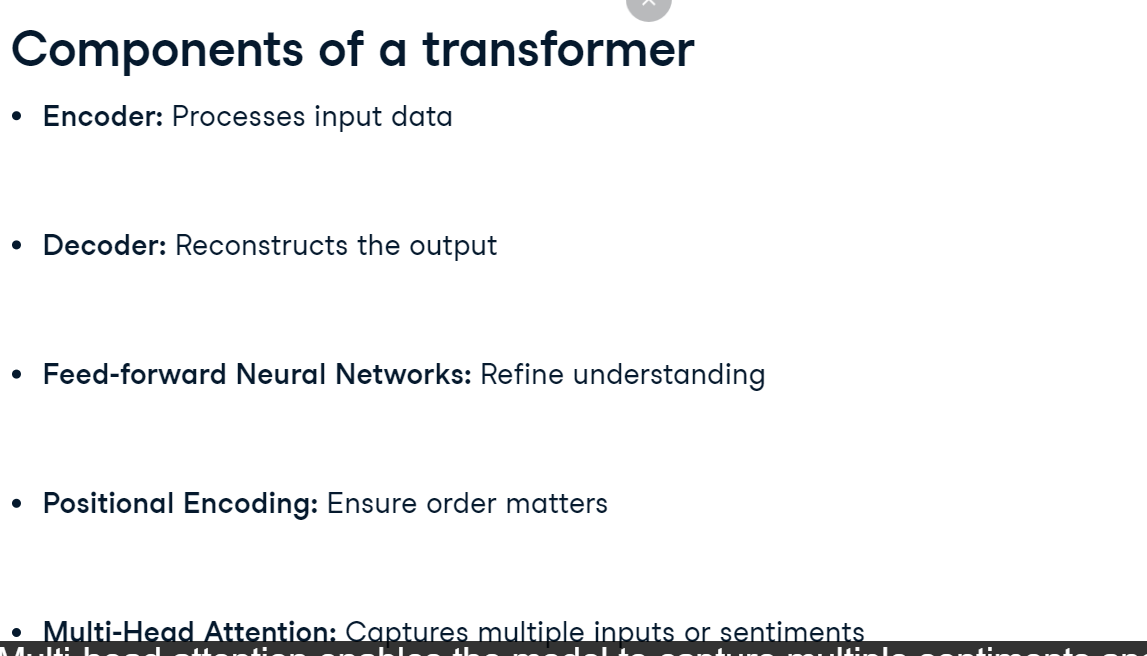
Now, let's explore how pre-trained models function.

**Why use transformers for text processing?**

Transformers, like those from Hugging Face, are the foundation of many pre-trained models and are known for speed and understanding the deep relationship between words, even if they're distant in a sentence, unlike RNNs, which trudge word by word. Transformers can also generate highly authentic human-like text. Let's take a peek inside.

**Components of a transformer**

There are several components to a Transformer. Encoder layers process input, such as analyzing a movie review's tone. Decoder layers reconstruct output, as in English-to-French translation. However, for sentiment analysis, we only need to interpret so we'll use the encoder, not the decoder. Feed-forward networks refine understanding, identifying nuances like sarcasm. Positional encoding ensures order matters - because in reviews, a don't can change everything. Multi-head attention enables the model to capture multiple sentiments and complex patterns in lengthy reviews.



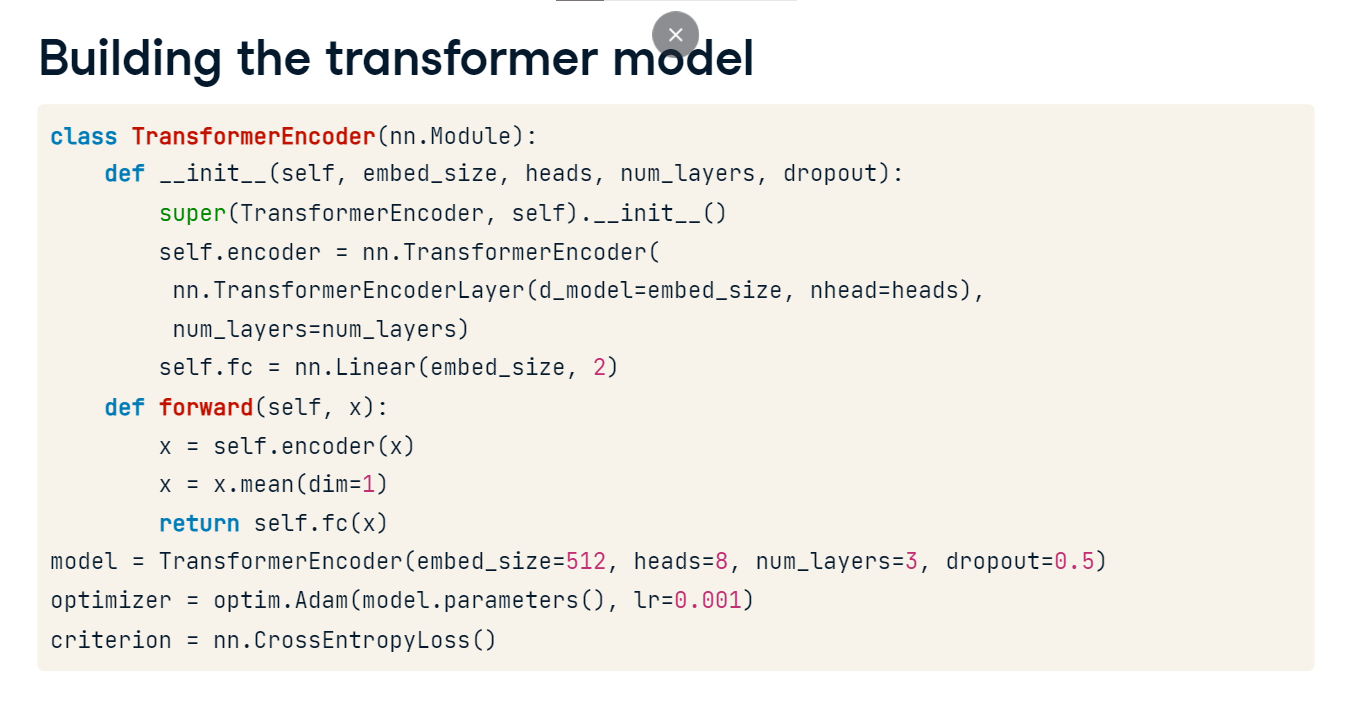
**Preparing our data: train-test split**

Let's use a transformer on text data. We'll create training and test datasets. We have four sentences with sentiment labels, one for positive, and zero for negative. The first three sentences are for training, and the last one is for testing. Typically, datasets are larger than this sample.



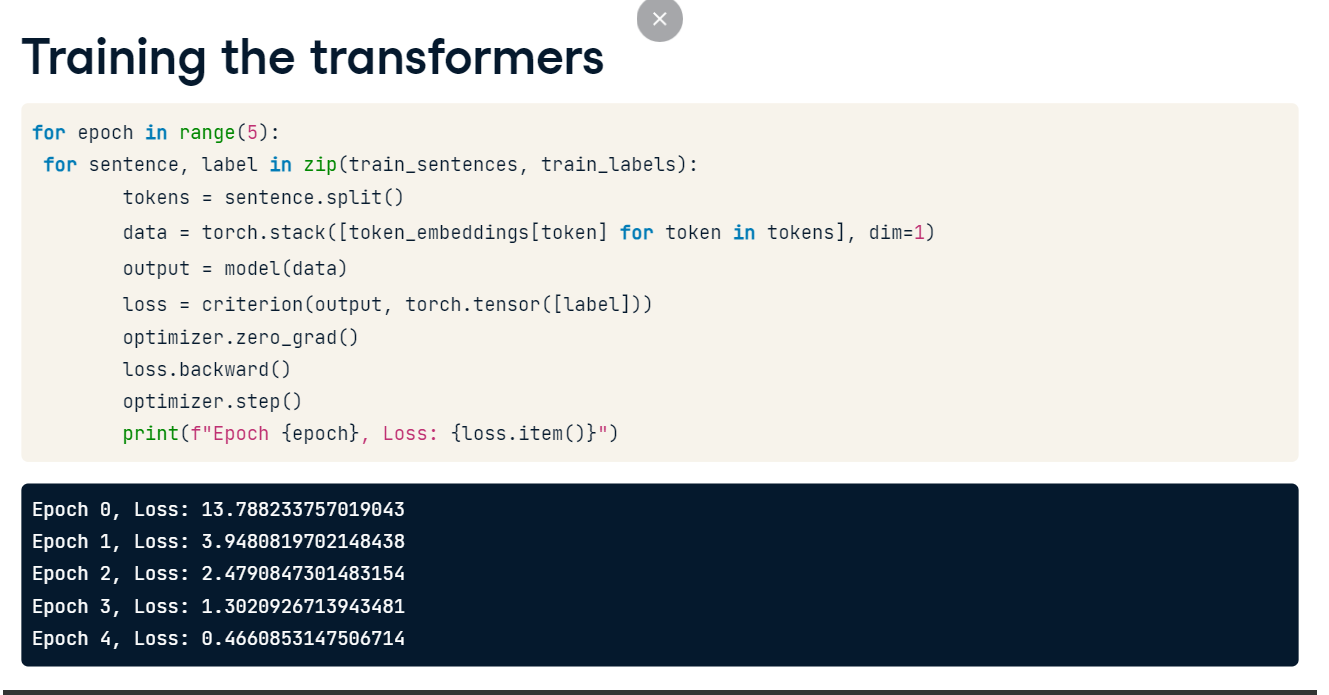
**Building the transformer model**

Let’s create a TransformerEncoder class with nn-dot-Module. This custom class wraps PyTorch’s nn-dot-TransformerEncoder, specializing it for sentiment analysis since the built-in version is generally too broad for our needs. The parameters for the init method of our TransformerEncoder class include embed\_size for embedding, heads for attention heads, num\_layers for the number of layers, and dropout rate. It leverages properties from nn-dot-Module through super. It employs nn-dot-TransformerEncoder with nn-dot-TransformerEncoderLayer, with parameters d\_model and nhead, equating to embed\_size and heads, respectively. d\_model influences the model's representational depth, and nhead determines how many word contexts the model can focus on simultaneously, impacting its contextual understanding. The class includes a linear layer, self-dot-fc, transforming input features to two classes for binary classification. During the forward method, data moves through the encoder, is averaged, and reaches self-dot-fc. Initializing the class, we set embed\_size to 512 for balanced power and efficiency, with 8 heads allowing focus on 8 word segments at once. Adjusting these affects complexity and overfitting risk. We use three num\_layers and a dropout of zero-dot-five to combat overfitting. Finally, we use Adam optimizer with a learning rate of 0-point-001 and CrossEntropyLoss for classification tasks.



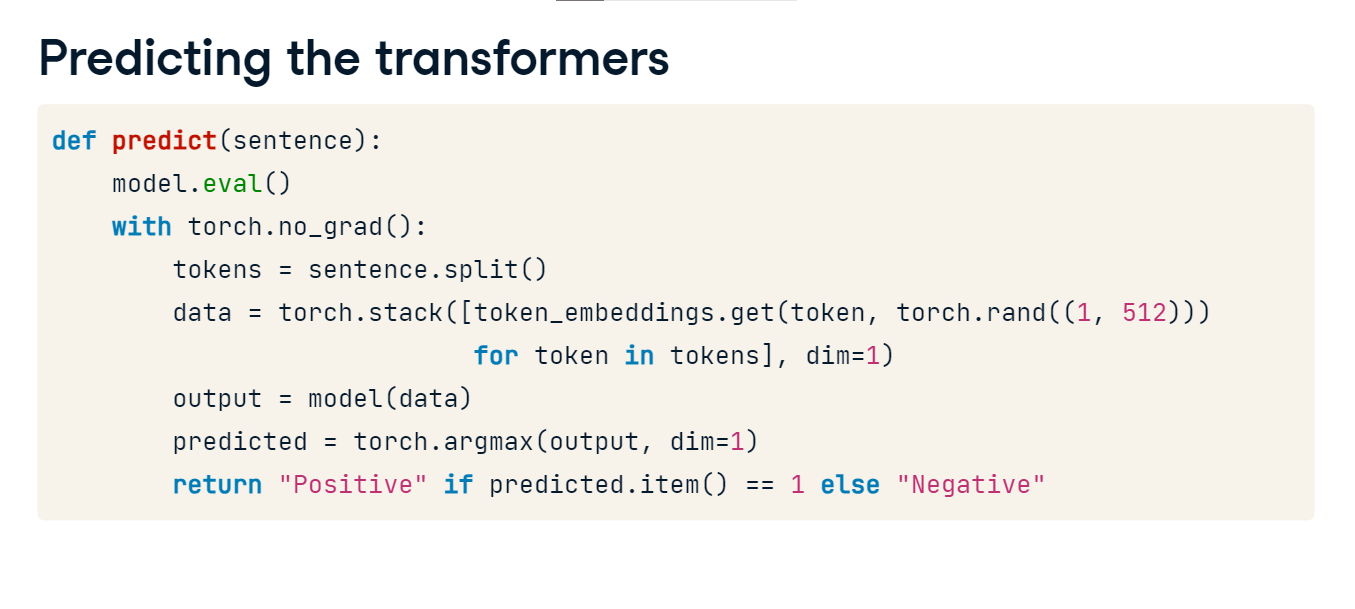
**Training the transformers**

We train for five epochs. In each epoch, sentences are tokenized into words and converted to embeddings using a pre-made token\_embeddings dictionary. These embeddings are stacked along a new dimension, dimension one, using torch-dot-stack to form a data tensor. The model processes this tensor to produce an output. The disparity between this output and the actual label constitutes the loss. After each iteration, the optimizer clears the gradient to prevent accumulation. The decreasing loss through the epochs indicates improving accuracy of the model.



**Predicting the transformers**

To predict sentiments using our trained Transformer, we define a predict function and set the model to evaluation mode. We utilize torch-dot-no\_grad to skip gradient calculations, saving memory. Within, we tokenize the input sentence and get the embeddings using the torch-dot-stack. Here, we loop through each token in the sentence and retrieve its embedding from the token\_embeddings dictionary. If the token is not in the dictionary, we generate a random tensor with shape 1 to 512 as a placeholder using torch-dot-rand. All random tensors are then stacked along dimension one, creating a 3D tensor. The tensor is passed to the model to get an output and fetch predictions. The torch-dot-argmax function identifies the predicted class, which we then translate to 'Positive' or 'Negative' based on its value.



**Predicting on new text**

Using our predict function and the sentence 'This product can be better', we determine its sentiment, which the model interprets as 'Negative'.

