

We Raised \$8M Series A to Continue Building Experiment Tracking and Model Registry That “Just Works” [Read more »](#)



blog

Search all articles

Get Newsletter



[Blog »](#)

[Natural Language Processing](#)

[ML Model Development](#)

[MLOps](#)

[Machine Learning](#)



How to Code BERT Using PyTorch – Tutorial With Examples

16 mins read Author Nilesh Barla Updated July 21st, 2022

If you are an NLP enthusiast then you might have heard about BERT. In this article, we are going to explore BERT: what it is? and how it works?, and learn how to code it using PyTorch.

In 2018, Google published a paper titled “[Pre-training of deep bidirectional transformers for language understanding](#)”. In this paper, they introduced a language model called **BERT (Bidirectional Encoder Representation with Transformers)** that achieved state-of-the-art performance in tasks like *Question-Answering, Natural Language Inference, Classification, and General language understanding evaluation or (GLUE)*.

BERT release was followed after the release of three architectures that also achieved state-of-the-art performances. These models were:

- ULM-Fit (January)
- ELMo (February),
- OpenAI GPT (June)
- BERT (October).

The OpenAI GPT and BERT use the [Transformer](#) architecture that does not use recurrent neural networks; this enabled the architecture to take into account long-term dependencies through the **self-attention mechanism** that inherently changed the way we model sequential data. It introduced an **encoder-decoder** architecture which was seen in computer vision applications such as image generation through variational autoencoder encoder.

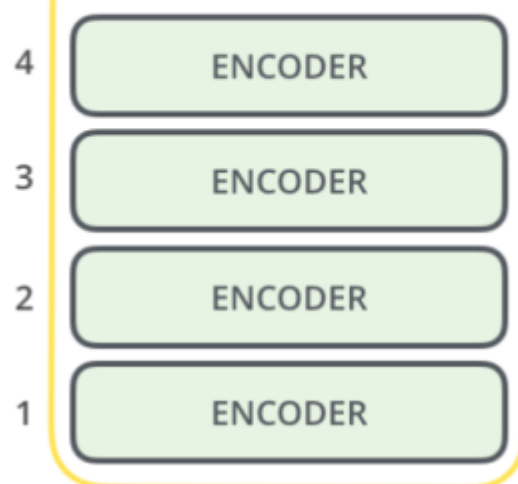
So how is BERT different from all the models that were released in 2018?

Well, to answer that question we need to understand what BERT is and how it works.

So, let's begin.

What is BERT?

BERT stands for “Bidirectional Encoder Representation with Transformers”. To put it in simple words BERT extracts patterns or representations from the data or word embeddings by passing it through an encoder. The encoder itself is a transformer architecture that is stacked together. It is a bidirectional transformer which means that during training it considers the context from both left and right of the vocabulary to extract patterns or representations.

[Source](#)

BERT uses two training paradigms: **Pre-training** and **Fine-tuning**.

During **pre-training**, the model is trained on a large dataset to extract patterns. This is generally an **unsupervised learning** task where the model is trained on an unlabelled dataset like the data from a big corpus like Wikipedia.

During **fine-tuning** the model is trained for downstream tasks like Classification, Text-Generation, Language Translation, Question-Answering, and so forth. Essentially, you can download a pre-trained model and then Transfer-learn the model on your data.

MIGHT INTEREST YOU

💡 [AI Limits: Can Deep Learning Models Like BERT Ever Understand Language?](#)

💡 [10 Things You Need to Know About BERT and the Transformer Architecture That Are Reshaping the AI Landscape](#)

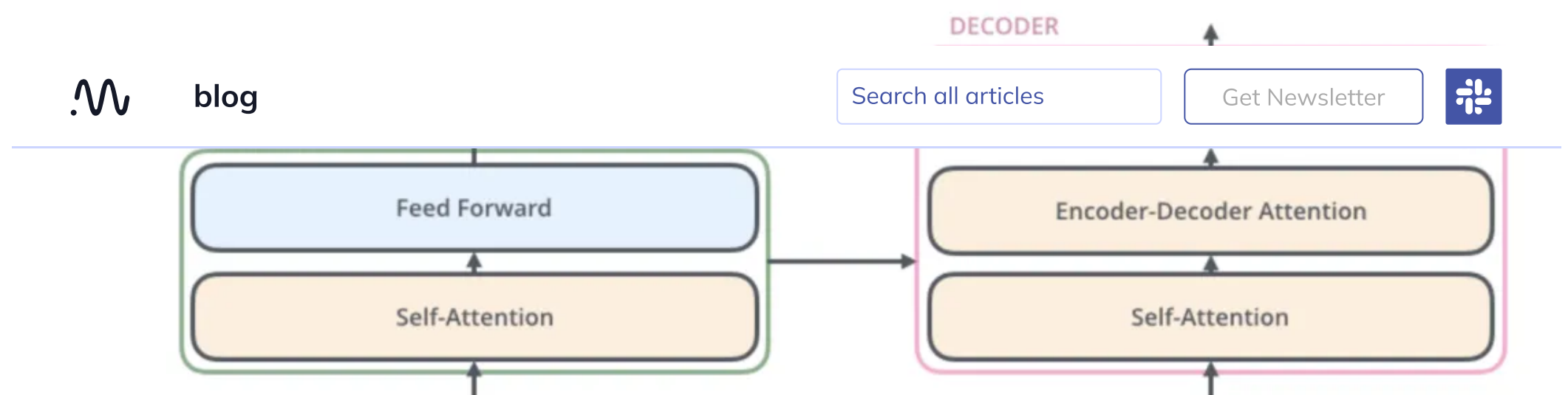
Core components of BERT

BERT borrows ideas from the previous release SOTA models. Let's elaborate on that statement.

The Transformers

BERT's main component is the transformer architecture. The transformers are made up of two components: **encoder** and **decoder**. The encoder itself contains two components: the **self-attention layer** and **feed-forward neural network**.

The self-attention layer takes an input and encodes each word into intermediate encoded representations which are then passed through the feed-forward neural network. The feed-forward network passes those representations to the decoder that itself is made up of three components: **self-attention layer**, **Encoder-Decoder Attention**, and **feed-forward neural network**.



[Source](#)

The benefit of the transformer architecture is that it helps the model to retain infinitely long sequences that were not possible from the traditional RNNs, LSTMs, and GRU. But even from the fact that it can achieve long-term dependencies it still **lacks contextual understanding**.

Jay Alamar explains transformers in-depth in his article [The Illustrated Transformer](#), worth checking out.

ELMo

BERT borrows another idea from ELMo which stands for Embeddings from Language Model. ELMo was introduced by [Peters et. al.](#) in 2017 which dealt with the idea of contextual understanding. The way ELMo works is that it uses **bidirectional** LSTM to make sense of the context. Since it considers words from both directions, it can assign different word embedding to words that are spelled similarly but have different meanings.

For instance, “You kids should **stick** together in the dark” is completely different from “Hand me that **stick**”. Even though the same word is being used in both sentences the meaning is different based on the context.

So, ELMo assigns embeddings by considering the words from both the right and left directions as compared to the models that were developed previously which took into consideration words, only from the left. These models were unidirectional like RNNs, LSTMs et cetera.

This enables ELMo to capture contextual information from the sequences but since ELMo uses LSTM it does not have long-term dependency compared to transformers.

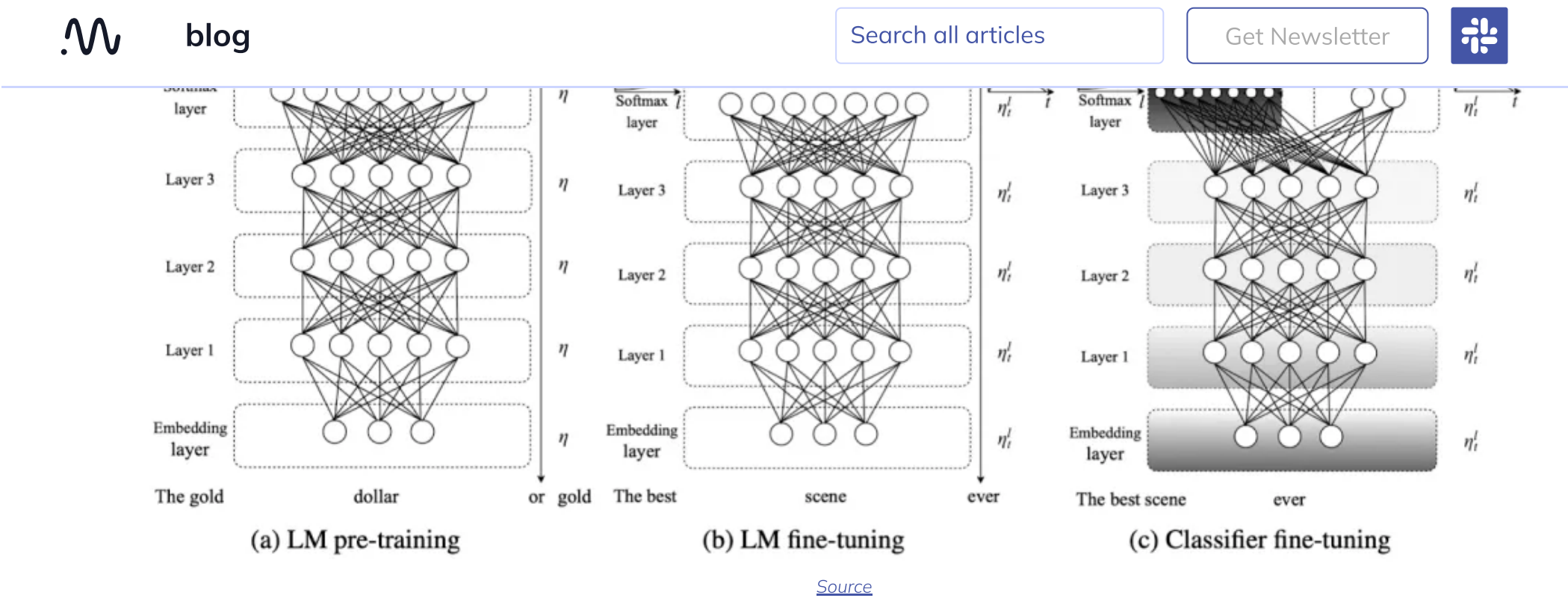
So far we have seen that BERT can access sequences in the document even if it is ‘n’ words behind the current word in the sequence because of the attention mechanism present in transformers, i.e. it can preserve long term dependencies and it can also achieve a contextual understanding of the sentence because of the bidirectional mechanism present in ELMo.

ULM-FiT

In 2018 Jeremy Howard and Sebastian Ruder released a paper called [Universal Language Model Fine-tuning or ULM-FiT](#), where they argued that transfer learning can be used in NLP just like it is used in computer vision.

Previously we were using pre-trained models for word-embeddings that only targeted the first layer of the entire model, i.e. the embedding layers, and the whole model was trained from the scratch, this was time-consuming, and not a lot of success was found in this area. However, Howard and Ruder proposed 3 methods for the classification of text:

- The first step includes training the model on a larger dataset so that the model learns representations.
- The second step included fine-tuning the model with a task-specific dataset for classification, during which they introduced two more methods: Discriminative fine-tuning and Slanted triangular learning rates (STLR). The former method tries to fine-tune or optimize the parameters for each during the transfer layer in the network while the latter controls the learning rate in each of the optimization steps.
- The third step was to fine-tune the classifier on the task-specific dataset for classification.



With the release of ULM-FiT NLP practitioners can now practice the transfer learning approach in their NLP problems. But the only problem with the ULM-FiT approach to transfer learning was that it included fine-tuning all the layers in the network which was a lot of work.

OpenAI GPT

Generative Pre-trained Transformer or GPT was introduced by OpenAI's team: Radford, Narasimhan, Salimans, and Sutskever. They presented a model that only uses decoders from the transformer instead of encoders in a unidirectional approach. As a result, it outperformed all the previous models in various tasks like:

- Classification
- Natural Language Inference
- Semantic similarity
- Question answering
- Multiple Choice.

Even though the GPT used only the decoder, it could still retain long-term dependencies. Furthermore, it reduced fine-tuning to a minimum compared to what we saw in ULM-FiT.

Below is the table that compares different models based upon pre-training, downstream tasks, and most importantly fine-tuning.

	Base model	pre-training	Downstream tasks	Downstream model	Fine-tuning
CoVe	seq2seq NMT model	supervised	feature-based	task-specific	/
ELMo	two-layer biLSTM	unsupervised	feature-based	task-specific	/
CVT	two-layer biLSTM	semi-supervised	model-based	task-specific / task-agnostic	/
ULMFIT	AWD-LSTM	unsupervised	model-based	task-agnostic	all layers; with various training tricks
GPT	Transformer decoder	unsupervised	model-based	task-agnostic	pre-trained layers + top task layer(s)
BERT	Transformer encoder	unsupervised	model-based	task-agnostic	pre-trained layers + top task layer(s)
GPT-2	Transformer decoder	unsupervised	model-based	task-agnostic	pre-trained layers + top task layer(s)

An excerpt from the GPT [paper](#) reads “*This model choice provides us with a more structured memory for handling long-term*



blog

Search all articles

Get Newsletter



enable us to fine-tune effectively with minimal changes to the architecture of the pre-trained model.”

Let’s compare all the model with BERT for the tasks they can perform:

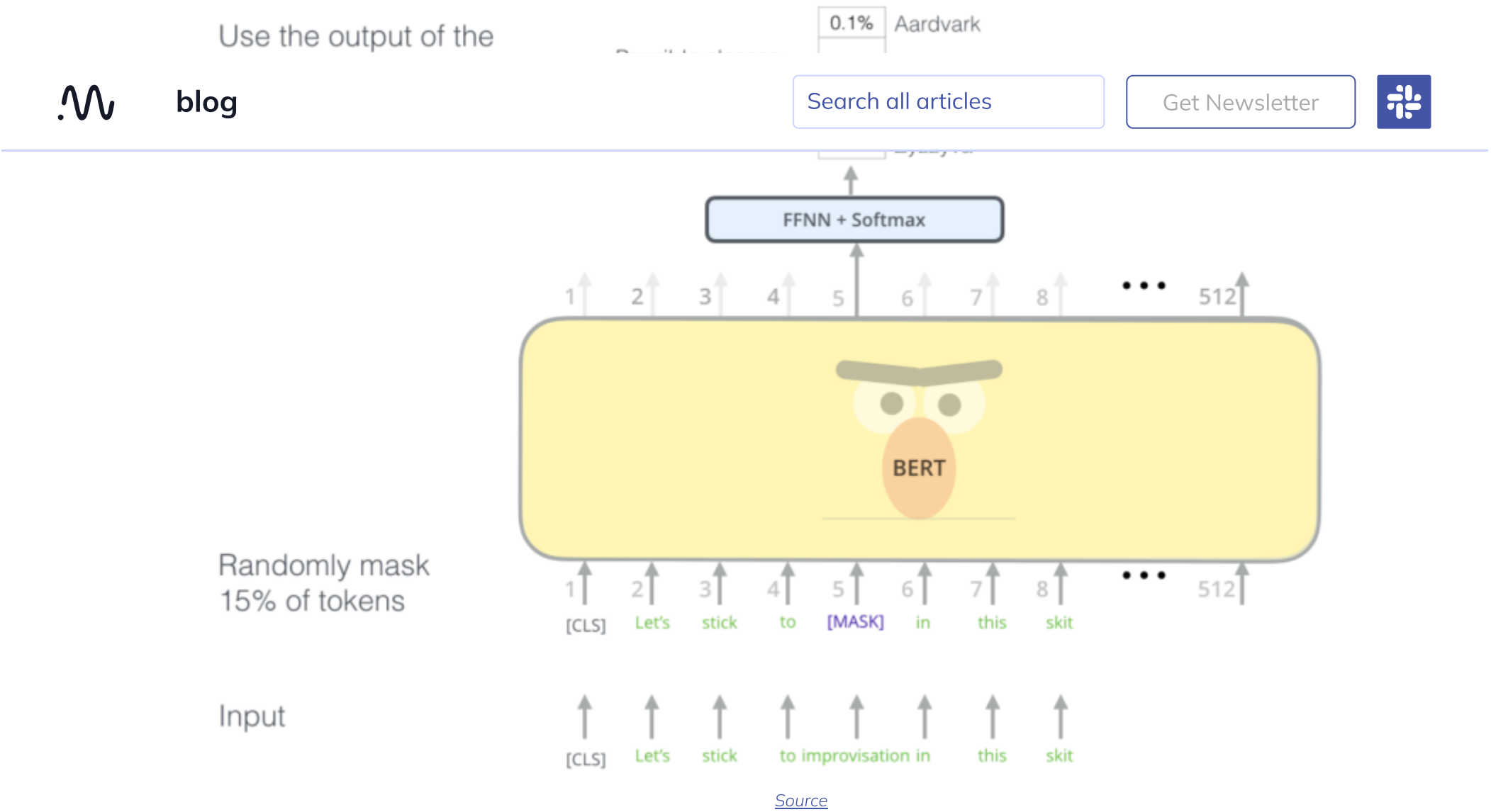
	Transformer	ELMo	ULM-FiT	OpenAI GPT	BERT
Contextual understanding	No	Yes (Weak)	Yes (Weak)	Yes (Moderate)	Yes (Strong)
Long-term dependencies	Infinite	Finite	Finite	Infinite	Infinite
Machine translation	Yes	No	No	Yes	Yes
Natural language inference	No	Yes	No	Yes	Yes
Question answering	No	Yes	No	Yes	Yes
Classification or sentiment analysis	No	Yes	Yes	Yes	Yes
Text generation	No	No	No	Yes (Poor)	No
Fill-Mask	No	No	No	No	Yes

You can check [Huggingface models](#) to check the model’s performance on every task.


Why BERT?

BERT falls into a [self-supervised](#) model. That means, it can generate inputs and labels from the raw corpus without being explicitly programmed by humans. Remember the data it is trained on is unstructured.

BERT was pre-trained with two specific tasks: Masked Language Model and Next sentence prediction. The former uses masked input like “the man [MASK] to the store” instead of “the man went to the store”. This restricts BERT to see the words next to it which allows it to learn bidirectional representations as much as possible making it much more flexible and reliable for several downstream tasks. The latter predicts whether the two sentences are contextually assigned to each other.




In the original BERT paper, it was compared with GPT on the [General Language understanding evaluation benchmark](#), and here are the results.

blog

Search all articles

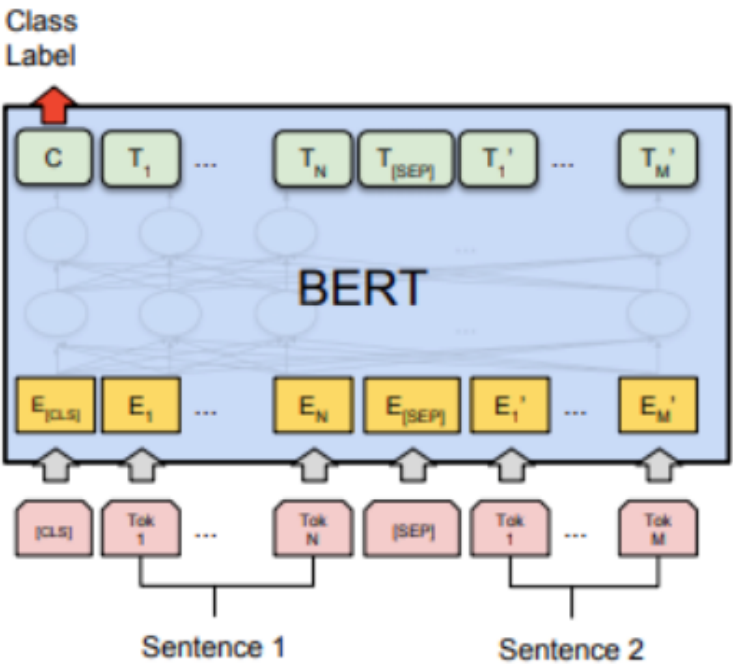
Get Newsletter



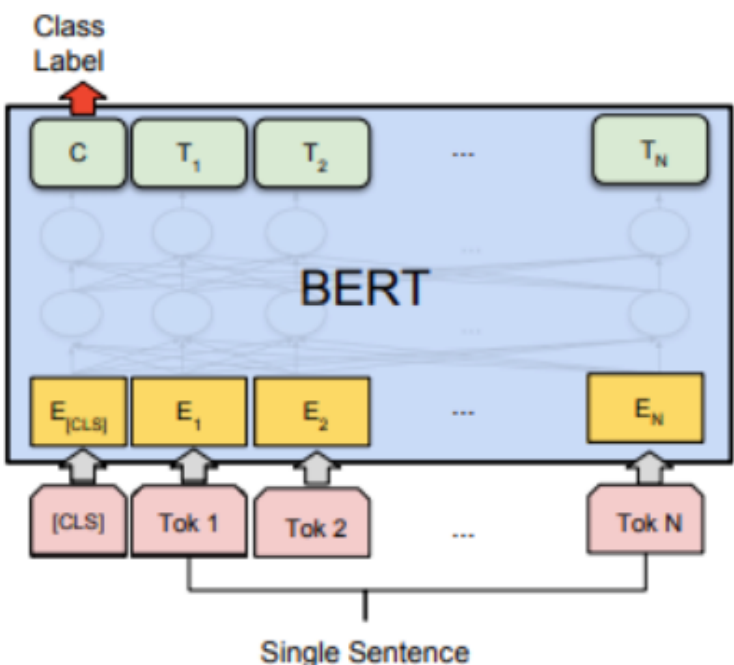
	392k	363k	108k	67k	8.5k	5.7k	3.5k	2.5k	-
Pre-OpenAI SOTA	80.6/80.1	66.1	82.3	93.2	35.0	81.0	86.0	61.7	74.0
BiLSTM+ELMo+Attn	76.4/76.1	64.8	79.8	90.4	36.0	73.3	84.9	56.8	71.0
OpenAI GPT	82.1/81.4	70.3	87.4	91.3	45.4	80.0	82.3	56.0	75.1
BERT _{BASE}	84.6/83.4	71.2	90.5	93.5	52.1	85.8	88.9	66.4	79.6
BERT _{LARGE}	86.7/85.9	72.1	92.7	94.9	60.5	86.5	89.3	70.1	82.1

Source

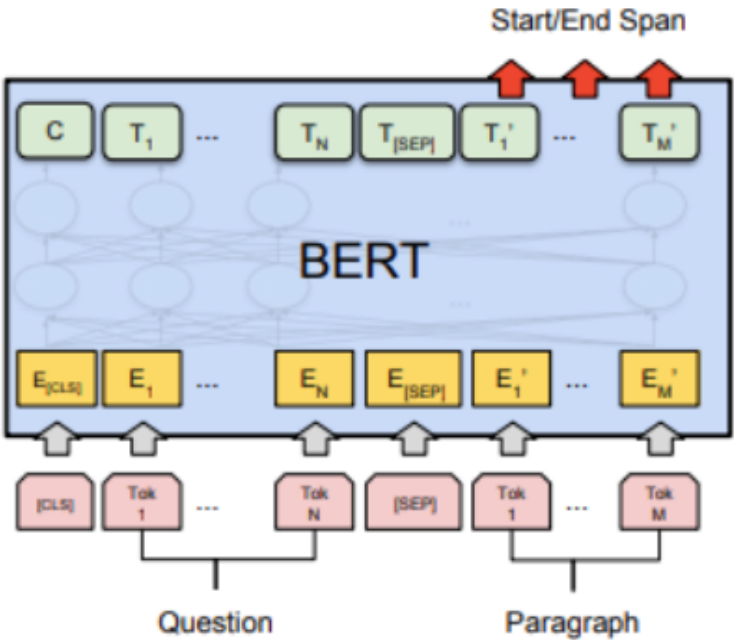
As you can see BERT outperformed GPT in all the tasks and averages 7% better than GPT.



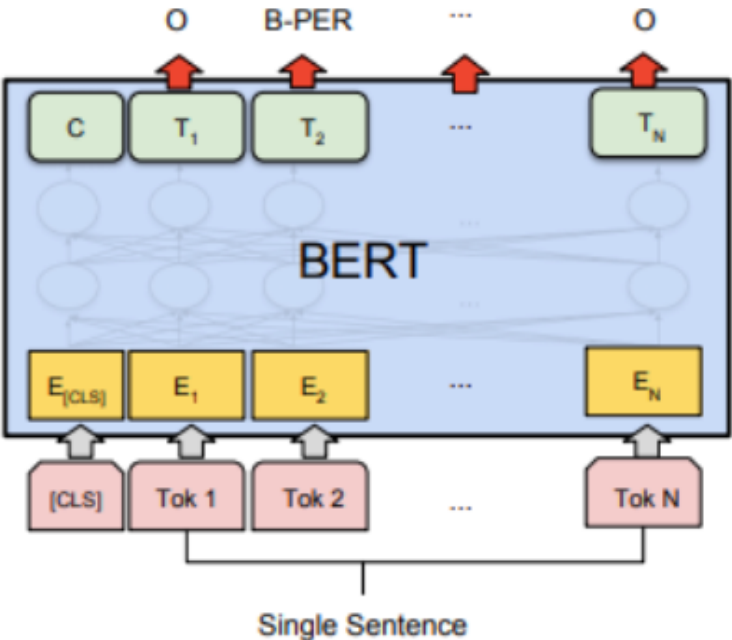
(a) Sentence Pair Classification Tasks:
MNLI, QQP, QNLI, STS-B, MRPC,
RTE, SWAG



(b) Single Sentence Classification Tasks:
SST-2, CoLA



(c) Question Answering Tasks:
SQuAD v1.1



(d) Single Sentence Tagging Tasks:
CoNLL-2003 NER

The image above shows the different tasks that BERT can be used for. | [Source](#)

Coding BERT with Pytorch

We will break the entire program into 4 sections:

1. Preprocessing

CHECK ALSO

[How to Keep Track of Experiments in PyTorch Using Neptune](#)

Preprocessing

In preprocessing we will structure the data such that the neural network can process it. We start by assigning a raw text for training.

```
text = (  
    'Hello, how are you? I am Romeo.\n'  
    'Hello, Romeo My name is Juliet. Nice to meet you.\n'  
    'Nice meet you too. How are you today?\n'  
    'Great. My baseball team won the competition.\n'  
    'Oh Congratulations, Juliet\n'  
    'Thanks you Romeo'  
)
```

Then we will clean the data by:

- Making the sentences into lower case.
- Creating vocabulary. **Vocabulary** is a list of unique words in the document.

```
sentences = re.sub("[.,!?\\-]", ' ', text.lower()).split('\n') # filter '.', ',', '?', '!'
word_list = list(set(" ".join(sentences).split()))
```

Now, in the following step, it is important to remember that BERT takes special tokens during training. Here is a table explaining the purpose of various tokens:

Token	Purpose
[CLS]	The first token is always classification
[SEP]	Separates two sentences
[END]	End the sentence.
[PAD]	Use to truncate the sentence with equal length.
[MASK]	Use to create a mask by replacing the original word

These tokens should be included in the word dictionary where each token and word in the vocabulary is assigned with an index number.



```
for i, w in enumerate(word_list):
    word_dict[w] = i + 4
number_dict = {i: w for i, w in enumerate(word_dict)}
vocab_size = len(word_dict)
```

Once that is taken care of, we need to create a function that formats the input sequences for three types of embeddings: **token embedding**, **segment embedding**, and **position embedding**.

What is token embedding?

For instance, if the sentence is “The cat is walking. The dog is barking”, then the function should create a sequence in the following manner: “[CLS] the cat is walking [SEP] the dog is barking”.

After that, we convert everything to an index from the word dictionary. So the previous sentence would look something like “[1, 5, 7, 9, 10, 2, 5, 6, 9, 11]”. Keep in mind that 1 and 2 are [CLS] and [SEP] respectively.

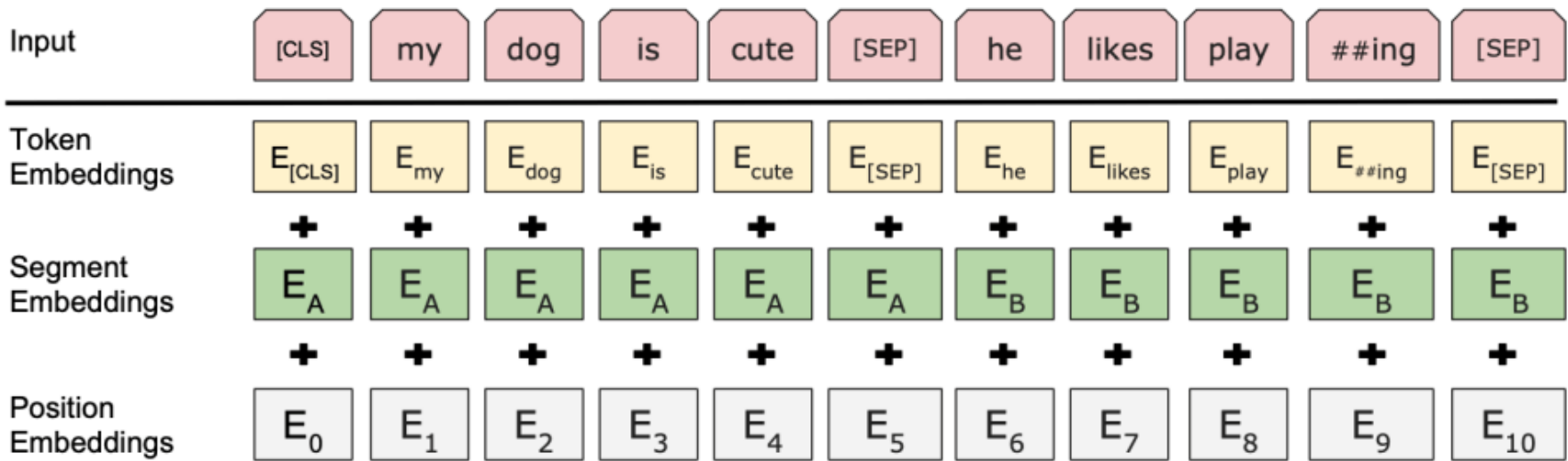
What is segment embedding?

A segment embedding separates two sentences from each other and they are generally defined as 0 and 1.

What is position embedding?

A position embedding gives position to each embedding in a sequence.

We will create a function for position embedding later.



[Source](#)

Now the next step will be to create **masking**.

As mentioned in the original paper, BERT randomly assigns masks to 15% of the sequence. But keep in mind that you don’t assign masks to the special tokens. For that, we will use conditional statements.

Once we replace 15% of the words with [MASK] tokens, we will add padding. Padding is usually done to make sure that all the sentences are of equal length. For instance, if we take the sentence :

“The cat is walking. The dog is barking at the tree”

then with padding, it will look like this:

“[CLS] The cat is walking [PAD] [PAD] [PAD]. [CLS] The dog is barking at the tree.”

The length of the first sentence is equal to the length of the second sentence.

```
def make_batch():
    batch = []
```



blog

Search all articles

Get Newsletter



```
tokens_a_index, tokens_b_index= randrange(len(sentences)),
randrange(len(sentences))

tokens_a, tokens_b= token_list[tokens_a_index], token_list[tokens_b_index]

input_ids = [word_dict['[CLS]']] + tokens_a + [word_dict['[SEP]']] + tokens_b
+ [word_dict['[SEP]']]

segment_ids = [0] * (1 + len(tokens_a) + 1) + [1] * (len(tokens_b) + 1)

# MASK LM
n_pred = min(max_pred, max(1, int(round(len(input_ids) * 0.15)))) # 15 % of
tokens in one sentence
cand_maked_pos = [i for i, token in enumerate(input_ids)
                  if token != word_dict['[CLS]'] and token !=
word_dict['[SEP]']]
shuffle(cand_maked_pos)
masked_tokens, masked_pos = [], []
for pos in cand_maked_pos[:n_pred]:
    masked_pos.append(pos)
    masked_tokens.append(input_ids[pos])
    if random() < 0.8: # 80%
        input_ids[pos] = word_dict['[MASK]'] # make mask
    elif random() < 0.5: # 10%
        index = randint(0, vocab_size - 1) # random index in vocabulary
        input_ids[pos] = word_dict[number_dict[index]] # replace

# Zero Paddings
n_pad = maxlen - len(input_ids)
input_ids.extend([0] * n_pad)
segment_ids.extend([0] * n_pad)

# Zero Padding (100% - 15%) tokens
if max_pred > n_pred:
    n_pad = max_pred - n_pred
    masked_tokens.extend([0] * n_pad)
    masked_pos.extend([0] * n_pad)

if tokens_a_index + 1 == tokens_b_index and positive < batch_size/2:
    batch.append([input_ids, segment_ids, masked_tokens, masked_pos, True]) #
IsNext
    positive += 1
elif tokens_a_index + 1 != tokens_b_index and negative < batch_size/2:
    batch.append([input_ids, segment_ids, masked_tokens, masked_pos, False])
# NotNext
    negative += 1

return batch
```

Since we are dealing with next-word prediction, we have to create a label that predicts whether the sentence has a consecutive sentence or not, i.e. IsNext or NotNext. So we assign True for every sentence that precedes the next sentence and we use a conditional statement to do that



$tokens_a_index + 1 == tokens_b_index$, i.e. second sentence in the same context, then we can set the label for this input as True.

If the above condition is not met i.e. if $tokens_a_index + 1 != tokens_b_index$ then we set the label for this input as False.

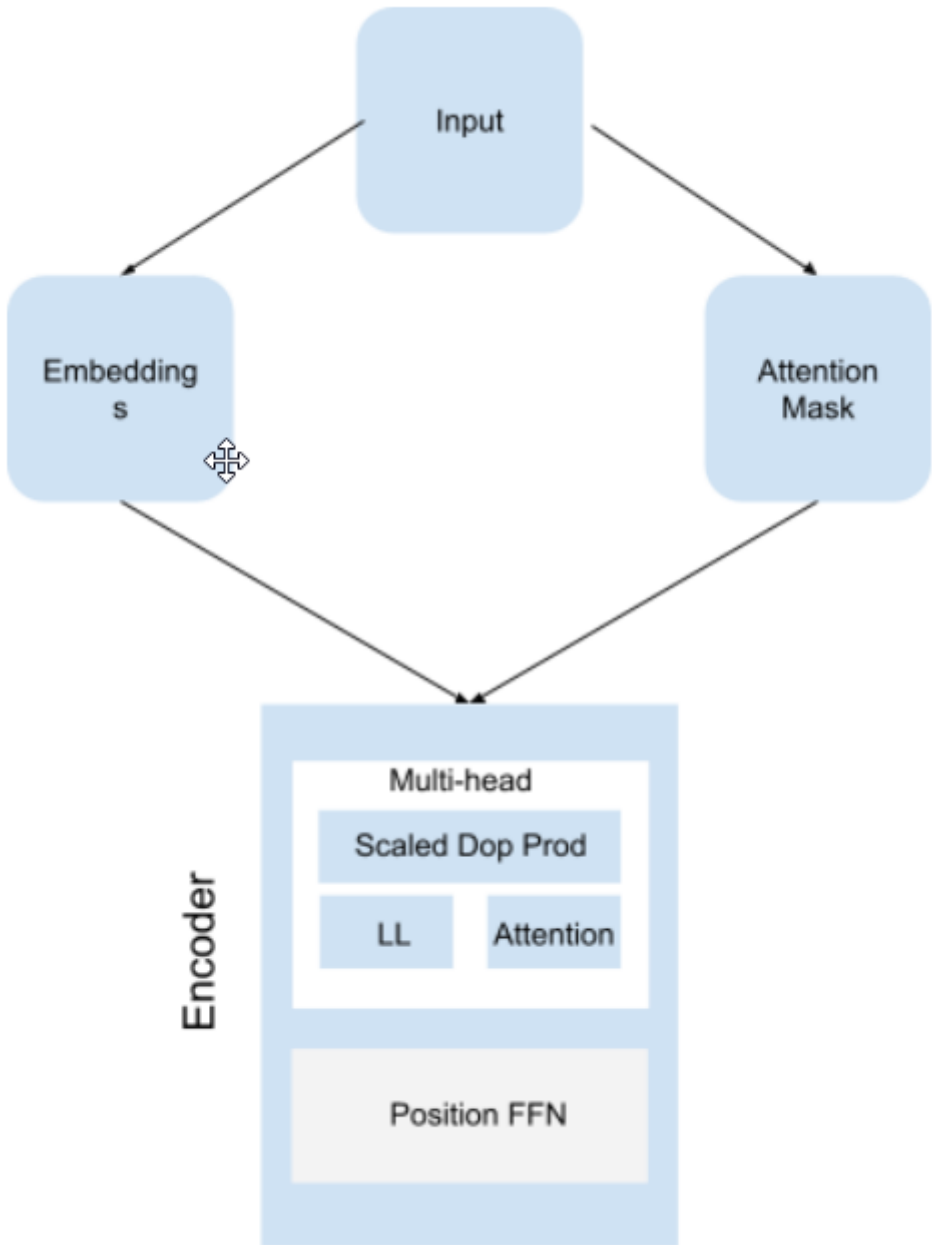
Building model

BERT is a complex model and if it is perceived slowly you lose track of the logic. So it'll only make sense to explain its component by component and their function.

BERT has the following components:

- 1. Embedding layers
- 2. Attention Mask
- 3. Encoder layer
 - 1. Multi-head attention
 - 1. Scaled dot product attention
 - 2. Position-wise feed-forward network
- 4. BERT (assembling all the components)

To make learning easier you can always refer to this diagram.



Source: Author

Embedding layer

The embedding layer also preserves different relationships between words such as: semantic, syntactic, linear, and since BERT is bidirectional it will also preserve contextual relationships as well.



- Segments and
- Position.

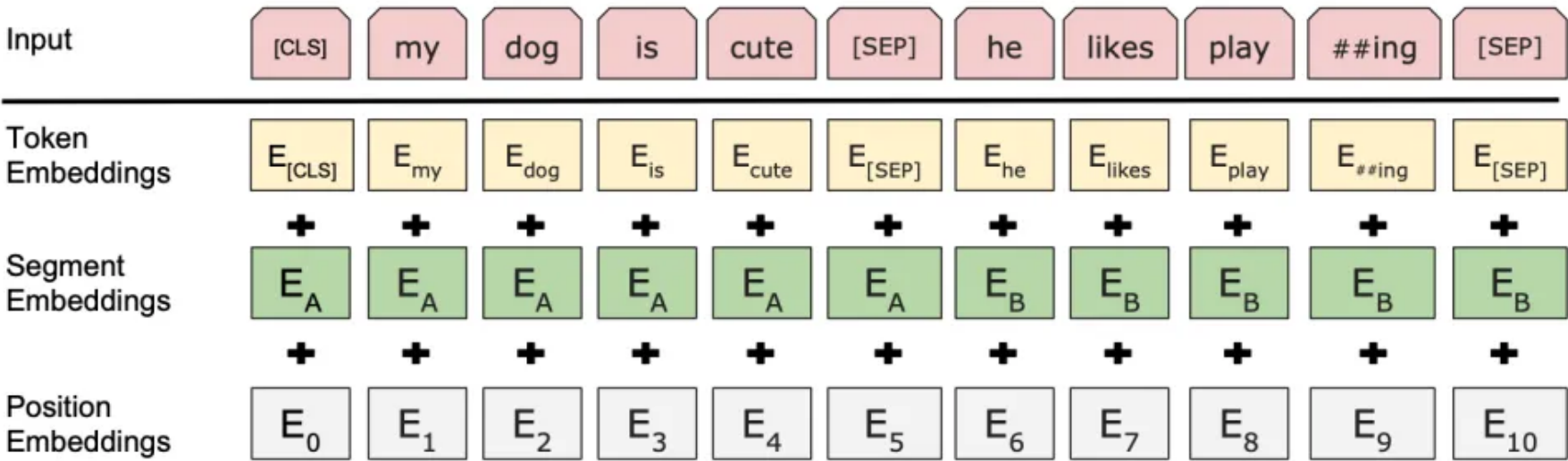
If you recall we haven't created a function that takes the input and formats it for position embedding but the formatting for token and segments are completed. So we will take the input and create a position for each word in the sequence. And it looks something like this:

```
print(torch.arange(30, dtype=torch.long).expand_as(input_ids))
```

Output:

tensor([[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]])

In the forward function, we sum up all the embeddings and normalize them.



[Source](#)



```
self.pos_embed = nn.Embedding(maxlen, d_model) # position embedding
self.seg_embed = nn.Embedding(n_segments, d_model) # segment(token type)
embedding
self.norm = nn.LayerNorm(d_model)

def forward(self, x, seg):
    seq_len = x.size(1)
    pos = torch.arange(seq_len, dtype=torch.long)
    pos = pos.unsqueeze(0).expand_as(x) # (seq_len,) -> (batch_size, seq_len)
    embedding = self.tok_embed(x) + self.pos_embed(pos) + self.seg_embed(seg)
    return self.norm(embedding)
```

READ ALSO

[Training, Visualizing, and Understanding Word Embeddings: Deep Dive Into Custom Datasets](#)

Creating attention mask

[BERT needs attention masks](#). And these should be in a proper format. The following code will help you create masks.

It will convert the [PAD] to 1 and elsewhere 0.

```
def get_attn_pad_mask(seq_q, seq_k):
    batch_size, len_q = seq_q.size()
    batch_size, len_k = seq_k.size()
    # eq(zero) is PAD token
    pad_attn_mask = seq_k.data.eq(0).unsqueeze(1) # batch_size x 1 x len_k(=len_q), one is masking
    return pad_attn_mask.expand(batch_size, len_q, len_k) # batch_size x len_q x len_k
```

```
print(get_attn_pad_mask(input_ids, input_ids)[0][0], input_ids[0])
```

```
Output:
(tensor([False, False, False, False, False, False, False, False, False, False,
         False, False, False, True, True, True, True, True, True, True,
         True, True, True, True, True, True, True, True, True, True]),
 tensor([ 1,  3, 26, 21, 14, 16, 12,  4,  2, 27,  3, 22,  2,  0,  0,  0,  0,  0,
         0,  0,  0,  0,  0,  0,  0,  0,  0,  0]))
```

- Multi-head Attention
- Position-wise feed-forward network.



blog

Search all articles

Get Newsletter



```
class EncoderLayer(nn.Module):
    def __init__(self):
        super(EncoderLayer, self).__init__()
        self.enc_self_attn = MultiHeadAttention()
        self.pos_ffn = PoswiseFeedForwardNet()

    def forward(self, enc_inputs, enc_self_attn_mask):
        enc_outputs, attn = self.enc_self_attn(enc_inputs, enc_inputs, enc_inputs,
        enc_self_attn_mask) # enc_inputs to same Q,K,V
        enc_outputs = self.pos_ffn(enc_outputs) # enc_outputs: [batch_size x len_q x
        d_model]
        return enc_outputs, attn
```

Multi-head attention

This is the first of the main components of the encoder.

The attention model takes three inputs: **Query**, **Key**, and **Value**.

I highly recommend you to read [The Illustrated Transformer](#) by Jay Alammar that explains Attention models in depth.

Multihead attention takes four inputs: **Query**, **Key**, **Value**, and **Attention mask**. The embeddings are fed as input to the Query, Key, and Value argument, and the attention mask is fed as input to the attention mask argument.

These three inputs and the attention mask are operated with a dot product operation that yields two outputs: **context vectors** and **attention**. The context vector is then passed through a linear layer and finally that yields the output.

```
class MultiHeadAttention(nn.Module):
```

```
    def __init__(self):
```

```
        self.W_K = nn.Linear(d_model, d_k * n_heads)
```

```
        self.W_V = nn.Linear(d_model, d_v * n_heads)
```

```
    def forward(self, Q, K, V, attn_mask):
```

```
        # q: [batch_size x len_q x d_model], k: [batch_size x len_k x d_model], v:
        [batch_size x len_k x d_model]
```

```
        residual, batch_size = Q, Q.size(0)
```

```
        # (B, S, D) -proj-> (B, S, D) -split-> (B, S, H, W) -trans-> (B, H, S, W)
```

```
        q_s = self.W_Q(Q).view(batch_size, -1, n_heads, d_k).transpose(1,2) # q_s:
        [batch_size x n_heads x len_q x d_k]
```

```
        k_s = self.W_K(K).view(batch_size, -1, n_heads, d_k).transpose(1,2) # k_s:
        [batch_size x n_heads x len_k x d_k]
```

```
        v_s = self.W_V(V).view(batch_size, -1, n_heads, d_v).transpose(1,2) # v_s:
        [batch_size x n_heads x len_k x d_v]
```

```
        attn_mask = attn_mask.unsqueeze(1).repeat(1, n_heads, 1, 1) # attn_mask :
        [batch_size x n_heads x len_q x len_k]
```

```
        # context: [batch_size x n_heads x len_q x d_v], attn: [batch_size x n_heads
        x len_q(=len_k) x len_k(=len_q)]
```

```
        context, attn = ScaledDotProductAttention()(q_s, k_s, v_s, attn_mask)
```

```
        context = context.transpose(1, 2).contiguous().view(batch_size, -1, n_heads *
        d_v) # context: [batch_size x len_q x n_heads * d_v]
```

```
        output = nn.Linear(n_heads * d_v, d_model)(context)
```

```
    return nn.LayerNorm(d_model)(output + residual), attn # output: [batch_size x len_q
        x d_model]
```

Now, let's explore this Scaled Dot Product attention:

- The scaled dot product attention class takes four arguments: Query, Key, Value, and Attention mask. Essentially, the first three arguments are fed with the word embeddings and the attention mask argument is fed with attention mask embeddings.
- Then it does a matrix multiplication between **query** and **key** to get scores.

Following that we use `scores.masked_fill_(attn_mask, -1e9)`. This attribute fills the element of scores with -1e9 where the attention masks are **True** while the rest of the elements get an **attention score** which is then passed through a softmax function that gives a score between 0 and 1. Finally, we perform a matrix multiplication between attention and values which gives us the context vectors.

```
class ScaledDotProductAttention(nn.Module):  
    def __init__(self):
```



blog

Search all articles

Get Newsletter



```
    def forward(self, Q, K, V, attn_mask):  
        scores = torch.matmul(Q, K.transpose(-1, -2)) / np.sqrt(d_k) # scores :  
        [batch_size x n_heads x len_q(=len_k) x len_k(=len_q)]  
        scores.masked_fill_(attn_mask, -1e9) # Fills elements of self tensor with  
        value where mask is one.  
        attn = nn.Softmax(dim=-1)(scores)  
        context = torch.matmul(attn, V)  
        return score, context, attn
```

```
emb = Embedding()  
embeds = emb(input_ids, segment_ids)  
  
attenM = get_attn_pad_mask(input_ids, input_ids)  
  
SDPA= ScaledDotProductAttention()(embeds, embeds, embeds, attenM)  
  
S, C, A = SDPA  
  
print('Masks', masks[0][0])  
print()  
print('Scores: ', S[0][0], '\n\nAttention Scores after softmax: ', A[0][0])
```

Output:

```
Masks tensor([False, False, False, False, False, False, False, False, False, False,  
              False, False, False, True, True, True, True, True, True, True,  
              True, True, True, True, True, True, True, True, True])  
  
Scores:  tensor([ 9.6000e+01,  3.1570e+01,  2.9415e+01,  3.3990e+01,  3.7752e+01,  
                 3.7363e+01,  3.1683e+01,  3.2156e+01,  3.5942e+01, -2.4670e+00,  
                -2.2461e+00, -8.1908e+00, -2.1571e+00, -1.0000e+09, -1.0000e+09,  
                -1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09,  
                -1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09,  
                -1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09, -1.0000e+09],  
        grad_fn=<SelectBackward>)  
  
Attention Scores after softmax::  tensor([1.0000e+00, 1.0440e-28, 1.2090e-29,  
    1.1732e-27, 5.0495e-26, 3.4218e-26,  
    1.1689e-28, 1.8746e-28, 8.2677e-27, 1.7236e-43, 2.1440e-43, 0.0000e+00,  
    2.3542e-43, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00,  
    0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00,  
    0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00, 0.0000e+00],  
    grad_fn=<SelectBackward>)
```


Let’s take a breath and revise what we’ve learned so far:

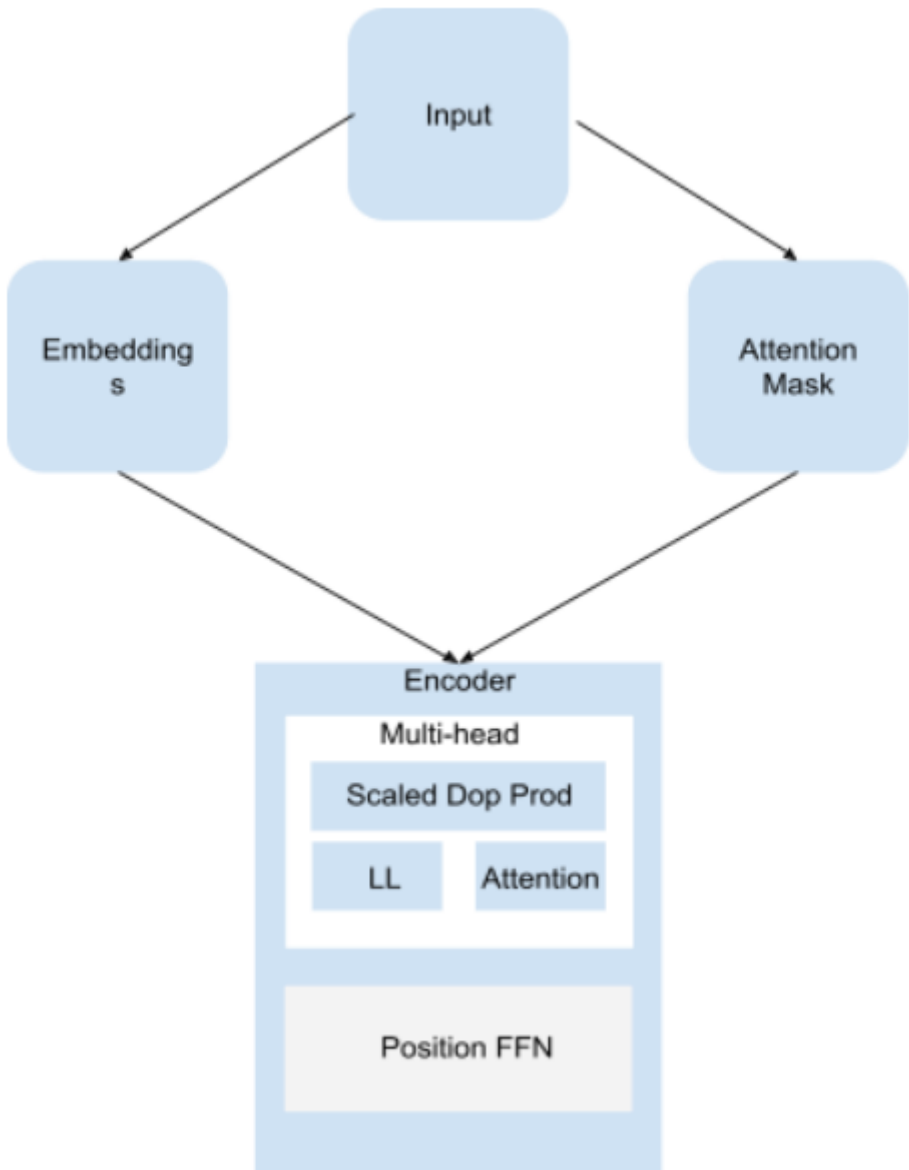
- The input goes into the embedding and as well attention function. Both of which are fed into the encoder which has a multi-



blog

Search all articles

Get Newsletter



Source: Author

Assembling all the components

Let’s continue from where we left, i.e. the output from the encoder.

The encoder yields two outputs:

- One which comes from the feed-forward layer and
- the Attention mask.

It’s key to remember that BERT does not explicitly use a decoder. Instead, it uses the output and the attention mask to get the desired result.

Although the decoder section in the transformers is replaced with a shallow network which can be used for classification as shown in the code below.

Also, BERT outputs two results: one for the **classifier** and the other for **masked**.



```
self.layers = nn.ModuleList([EncoderLayer() for _ in range(n_layers)])
self.fc = nn.Linear(d_model, d_model)
self.activ1 = nn.Tanh()
self.linear = nn.Linear(d_model, d_model)
self.activ2 = gelu
self.norm = nn.LayerNorm(d_model)
self.classifier = nn.Linear(d_model, 2)
# decoder is shared with embedding layer
embed_weight = self.embedding.tok_embed.weight
n_vocab, n_dim = embed_weight.size()
self.decoder = nn.Linear(n_dim, n_vocab, bias=False)
self.decoder.weight = embed_weight
self.decoder_bias = nn.Parameter(torch.zeros(n_vocab))

def forward(self, input_ids, segment_ids, masked_pos):
    output = self.embedding(input_ids, segment_ids)
    enc_self_attn_mask = get_attn_pad_mask(input_ids, input_ids)
    for layer in self.layers:
        output, enc_self_attn = layer(output, enc_self_attn_mask)
    # output : [batch_size, len, d_model], attn : [batch_size, n_heads, d_model,
d_model]
    # it will be decided by first token (CLS)
    h_pooled = self.activ1(self.fc(output[:, 0])) # [batch_size, d_model]
    logits_clsf = self.classifier(h_pooled) # [batch_size, 2]

    masked_pos = masked_pos[:, :, None].expand(-1, -1, output.size(-1)) #
[batch_size, max_pred, d_model]

    # get masked position from final output of transformer.
    h_masked = torch.gather(output, 1, masked_pos) # masking position
[batch_size, max_pred, d_model]
    h_masked = self.norm(self.activ2(self.linear(h_masked)))
    logits_lm = self.decoder(h_masked) + self.decoder_bias # [batch_size,
max_pred, n_vocab]

    return logits_lm, logits_clsf
```

Few things to keep in mind:

- 1. You can assign the number of encoders. In the original paper, the base model has 12.
- 2. There are two activation functions: Tanh and GELU(Gaussian Error Linear Unit).

```
def gelu(x):
    return x * 0.5 * (1.0 + torch.erf(x / math.sqrt(2.0)))
```

Loss and optimization

Neptune.ai uses cookies to ensure you get the best experience on this website. By continuing you agree to our use of cookies. [Learn more](#)

Got it!

Although the original paper calculates the probability distribution over all the vocabulary, we can use a softmax approximation. But a neat way to do it is to use ***cross-entropy loss***. It's a combination of both *softmax* and *negative log-likelihood*.

**blog**[Search all articles](#)[Get Newsletter](#)

When it comes to optimization we will be using **Adam** optimizer.

```
criterion = nn.CrossEntropyLoss()  
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

RELATED ARTICLE

[PyTorch Loss Functions: The Ultimate Guide](#)

Training

Finally, we'll start the training.

```
model = BERT()
batch = make_batch()
```



blog

Search all articles

Get Newsletter



```
for epoch in range(100):
    optimizer.zero_grad()
    logits_lm, logits_clsf = model(input_ids, segment_ids, masked_pos)
    loss_lm = criterion(logits_lm.transpose(1, 2), masked_tokens) # for masked LM
    loss_lm = (loss_lm.float()).mean()
    loss_clsf = criterion(logits_clsf, isNext) # for sentence classification
    loss = loss_lm + loss_clsf
    if (epoch + 1) % 10 == 0:
        print('Epoch:', '%04d' % (epoch + 1), 'cost =', '{:.6f}'.format(loss))
    loss.backward()
    optimizer.step()

# Predict mask tokens
input_ids, segment_ids, masked_tokens, masked_pos, isNext = map(torch.LongTensor,
zip(batch[0]))
print(text)
print([number_dict[w.item()] for w in input_ids[0] if number_dict[w.item()] !=
'[PAD]'])

logits_lm, logits_clsf = model(input_ids, segment_ids, masked_pos)
logits_lm = logits_lm.data.max(2)[1][0].data.numpy()
print('masked tokens list : ', [pos.item() for pos in masked_tokens[0] if
pos.item() != 0])
print('predict masked tokens list : ', [pos for pos in logits_lm if pos != 0])

logits_clsf = logits_clsf.data.max(1)[1].data.numpy()[0]
print('isNext : ', True if isNext else False)
print('predict isNext : ', True if logits_clsf else False)
```

Output:

```
Hello, how are you? I am Romeo.
Hello, Romeo My name is Juliet. Nice to meet you.
Nice meet you too. How are you today?
Great. My baseball team won the competition.
Oh Congratulations, Juliet
Thanks you Romeo
['[CLS]', 'nice', 'meet', 'you', 'too', 'how', 'are', 'you', 'today', '[SEP]',
'[MASK]', 'congratulations', '[MASK]', '[SEP]']
masked tokens list :  [27, 22]
predict masked tokens list :  []
isNext :  False
predict isNext :  True
```

So that was BERT coding from scratch. If you train it over a large corpus you then you can use the same model for

Neptune.ai uses cookies to ensure you get the best experience on this website. By continuing you agree to our use of cookies. [Learn more](#)

Got it!

3. Feature extractor for different tasks, or even topic modeling.

You can find the complete notebook [here](#).



blog

Search all articles

Get Newsletter



Is there a way to get a pre-trained model?

In the original paper, two models were released: BERT-base, and BERT-large. In the article, I showed how you can code BERT from scratch.

Generally, you can download the pre-trained model so that you don't have to go through these steps. The [Huggingface](#) 🤗 library offers this feature you can use the transformer library from Huggingface for PyTorch. The process remains the same.

I have a notebook where I used a pre-trained BERT from Huggingface, you can check it out [here](#).

When you use a pre-trained model, all you need to do is download the model and then call it inside a class and use a forward method to feed your inputs and masks.

For instance:

```
import transformers

class BERTClassification(nn.Module):
    def __init__(self):
        super(BERTClassification, self).__init__()
        self.bert = transformers.BertModel.from_pretrained('bert-base-cased')
        self.bert_drop = nn.Dropout(0.4)
        self.out = nn.Linear(768, 1)

    def forward(self, ids, mask, token_type_ids):
        _, pooledOut = self.bert(ids, attention_mask = mask,
                                token_type_ids=token_type_ids)
        bertOut = self.bert_drop(pooledOut)
        output = self.out(bertOut)

    return output
```

Final thoughts

BERT is a very powerful state-of-the-art NLP model. The pre-trained model is trained on a large corpus and you can fine-tune it according to your needs and based on the task on a smaller dataset. The best thing about fine-tuning is that you don't do it for 1000 epochs, it can mimic SOTA performances even in 3 to 10 epochs depending on the parameters and how well the dataset is processed.

I hope this tutorial was interesting and informative. And I hope you were able to take something out of it.

Resources

- [BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding](#)
- [The Illustrated BERT, ELMo, and co. \(How NLP Cracked Transfer Learning\)](#)
- [Deep contextualized word representations : ELMo](#)
- [Universal Language Model Fine-tuning for Text Classification](#)
- [Attention is All you Need: Transformer](#)

Neptune.ai uses cookies to ensure you get the best experience on this website. By continuing you agree to our use of cookies. [Learn more](#)

Got it!



READ NEXT

10 Things You Need to Know About BERT and the Transformer Architecture That Are Reshaping the AI Landscape

25 mins read | Author Cathal Horan | Updated May 31st, 2021

Few areas of AI are more exciting than NLP right now. In recent years language models (LM), which can perform human-like linguistic tasks, have evolved to perform better than anyone could have expected.

In fact, they're performing so well that [people are wondering](#) whether they're reaching a level of [general intelligence](#), or the evaluation metrics we use to test them just can't keep up. When technology like this comes along, whether it is electricity, the railway, the internet or the iPhone, one thing is clear – you can't ignore it. It will end up impacting every part of the modern world.

It's important to learn about technologies like this, because then you can use them to your advantage. So, let's learn!

We will cover ten things to show you where this technology came from, how it was developed, how it works, and what to expect from it in the near future. The ten things are:

1. **What is BERT and the transformer, and why do I need to understand it?** Models like BERT are already massively impacting academia and business, so we'll outline some of the ways these models are used, and clarify some of the terminology around them.
2. **What did we do before these models?** To understand these models, it's important to look at the problems in this area and understand how we tackled them before models like BERT came on the scene. This way we can understand the limits of previous models and better appreciate the motivation behind the key design aspects of the Transformer architecture, which underpins most SOTA models like BERT.
3. **NLPs "ImageNet moment; pre-trained models:** Originally, we all trained our own models, or you had to fully train a model for a specific task. One of the key milestones which enabled the rapid evolution in performance was the creation of pre-trained models which could be used "off-the-shelf" and tuned to your specific task with little effort and data, in a process known as transfer learning. Understanding this is key to seeing why these models have been, and continue to perform well in a range of NLP tasks.
4. **Understanding the Transformer:** You've probably heard of BERT and GPT-3, but what about [RoBERTa](#), [ALBERT](#), [XLNet](#), or the [LONGFORMER](#), [REFORMER](#), or [T5 Transformer](#)? The amount of new models seems overwhelming, but if you understand the Transformer architecture, you'll have a window into the internal workings of all of these models. It's the same as when you understand RDBMS technology, giving you a good handle on software like MySQL, PostgreSQL, SQL Server, or Oracle. The relational model that underpins all of the DBs is the same as the Transformer architecture that underpins our models. Understand that, and RoBERTa or XLNet becomes just the difference between using MySQL or PostgreSQL. It still takes time to learn the nuances of each model, but you have a solid foundation and you're not starting from scratch.
5. **The importance of bidirectionality:** As you're reading this, you're not strictly reading from one side to the other. You're not reading this sentence letter by letter in one direction from one side to the other. Instead, you're jumping ahead and learning context from the words and letters ahead of where you are right now. It turns out this is a critical feature of the Transformer architecture. The Transformer architecture enables models to process text in a bidirectional manner, from start to finish and from finish to start. This has been central to the limits of previous models which could only process text from start to finish.

[Continue reading ->](#)

[documenting it. Adding a metadata store to your workflow can change this.](#)



blog

Search all articles

Get Newsletter



Comprehensive Guide to Transformers

by Ahmed Hashesh

[Read more](#)

Data Augmentation in NLP: Best Practices From a Kaggle Master

by Shahul ES

[Read more](#)

How to Track Machine Learning Model Metrics in Your Projects

by Jakub Czakon

[Read more](#)



8 Creators and Core Contributors Talk About Their Model Training Libraries From PyTorch Ecosystem

by Jakub Czakon

[Read more](#)

Neptune is a metadata store for MLOps, built for research and production teams that run a lot of experiments.



Product

- Overview
- Experiment Tracking
- Model Registry
- Quickstart
- Neptune Docs
- Neptune Integrations
- Resources
- Pricing
- Roadmap
- Service Status

Legal

- Terms of service
- Privacy policy

Resources

- MLOps Live
- MLOps Blog
- ML Metadata Store
- ML Experiment Tracking
- ML Model Registry
- MLOps
- MLOps at a Reasonable Scale

Competitor Comparison

- ML Experiment Tracking Tools
- Neptune vs Weights & Biases
- Neptune vs MLflow
- Neptune vs TensorBoard
- Other Comparisons
- Best MLflow Alternatives
- Best TensorBoard Alternatives

Company

Neptune.ai uses cookies to ensure you get the best experience on this website. By continuing you agree to our use of cookies. [Learn more](#)

Got it!

[documenting it. Adding a metadata store to your workflow can change this.](#)



How to Code BERT Using PyTorch - Tutorial With Examples



blog

Search all articles

Get Newsletter

