Lecture 19

RNNs and Transformers for Sequence-to-Sequence Modeling
Lecture Topics

1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
5. Transformer Models
6. Transformer in PyTorch
Many-to-Many RNNs for Generating Text

1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
5. Transformer Models
6. Transformer in PyTorch
Different Types of Sequence Modeling Tasks

Previously, we built an (Word-level) RNN classifier

Figure based on:
The Unreasonable Effectiveness of Recurrent Neural Networks by Andrej Karpathy (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)
many-to-many

"training"

many-to-many

"generating new text"
Character RNN

Output layer

Hidden layer

Embedding layer

Inputs

E

S

T

0.7 0.2 0.1

0.2 0.6 0.2

0.2 0.25 0.55

1.13 -2.1 4.1

-5.4 -2.5 1.5

7.8 -1.1 2.3

0

1

0

0

1

0

0

1

0

1

0
At each time step
Softmax output (probability) for each possible "next letter"

For the next input, ignore the prediction but use the "correct" next letter from the dataset

many-to-many
"training"
To generate new text, now, sample from the softmax outputs and provide the letter as input to the next time step.

"many-to-many"  
"one"  
"generating new text"
To generate new text, now, sample from the softmax outputs and provide the letter as input to the next time step.

Note that this approach works with both Word- and Character-RNNs.
Advantages and Disadvantages of Character RNNs over Word RNNs

+ Character embeddings (only 24 letters plus punctuation in English language) require less memory compared to word embeddings
+ Smaller output layers for the same reason as above
  - Can create weird & nonsense words
  - Worse at capturing long-distance dependencies
Implementing Character RNNs in PyTorch

1. Sequence Generation with RNNs

2. Character RNN in PyTorch

3. RNNs with Attention

4. Attention is All We Need

5. Transformer Models

6. Transformer in PyTorch
LSTM Class


Parameters

- **input_size** – The number of expected features in the input \( x \)
- **hidden_size** – The number of features in the hidden state \( h \)
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two
  LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM
  and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights \( b_{lh} \) and \( b_{hh} \). Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (batch, seq, feature).
  Default: `False`
- **dropout** – If non-zero, introduces a Dropout layer on the outputs of each LSTM layer except the last
  layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If > 0, will use LSTM with projections of corresponding size. Default: 0

Examples:

```python
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> c0 = torch.randn(2, 3, 20)
>>> output, (hn, cn) = rnn(input, (h0, c0))
```
LSTM Class

Examples:

```python
>>> rnn = nn.LSTM(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> c0 = torch.randn(2, 3, 20)
>>> output, (hn, cn) = rnn(input, (h0, c0))
```

LSTMCell Class


<table>
<thead>
<tr>
<th>Inputs: input, (h_0, c_0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• input of shape (batch, input_size): tensor containing input features</td>
</tr>
<tr>
<td>• h_0 of shape (batch, hidden_size): tensor containing the initial hidden state for each element in the batch.</td>
</tr>
<tr>
<td>• c_0 of shape (batch, hidden_size): tensor containing the initial cell state for each element in the batch.</td>
</tr>
<tr>
<td>If (h_0, c_0) is not provided, both h_0 and c_0 default to zero.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Outputs: (h_1, c_1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• h_1 of shape (batch, hidden_size): tensor containing the next hidden state for each element in the batch</td>
</tr>
<tr>
<td>• c_1 of shape (batch, hidden_size): tensor containing the next cell state for each element in the batch</td>
</tr>
</tbody>
</table>

Examples:

```python
>>> rnn = nn.LSTMCell(10, 20) # (input_size, hidden_size)
>>> input = torch.randn(2, 3, 10) # (time_steps, batch, input_size)
>>> hx = torch.randn(3, 20) # (batch, hidden_size)
>>> cx = torch.randn(3, 20)
>>> output = []
>>> for i in range(input.size()[0]):
...     hx, cx = rnn(input[i], (hx, cx))
...     output.append(hx)
>>> output = torch.stack(output, dim=0)
```
LSTMCell Class

Example:

```python
>>> rnn = nn.LSTMCell(10, 20)  # (input_size, hidden_size)
>>> input = torch.randn(2, 3, 10)  # (time_steps, batch, input_size)
>>> hx = torch.randn(3, 20)  # (batch, hidden_size)
>>> cx = torch.randn(3, 20)
>>> output = []
>>> for i in range(input.size(0)):
...     hx, cx = rnn(input[i], (hx, cx))
...     output.append(hx)
>>> output = torch.stack(output, dim=0)
```

Translation with a Sequence to Sequence Network and Attention (English to French)

https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
Dealing Better with Long Sequences by Outfitting RNNs with an Attention Mechanism

1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
5. Transformer Models
6. Transformer in PyTorch
Many-to-Many Architecture for Language Translation

Translation with a Sequence to Sequence Network and Attention (English to French)

https://pytorch.org/tutorials/intermediate/seq2seq_translation_tutorial.html
Today is a great day

Heute ist ein großartiger Tag
If you’ve ever studied a foreign language, you’ve probably encountered a “false friend” at some point.

Translation:

Wenn Sie jemals eine Fremdsprache gelernt haben, sind Sie wahrscheinlich irgendwann auf einen „falschen Freund“ gestoßen.
Challenge in language translation: memorize whole input sentence in one hidden state

many-to-many

Challenge in language translation: memorize whole input sentence in one hidden state

many-to-many
Attention Mechanism

- Originally developed for language translation:

"... allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word ...

Figure 2: The BLEU scores of the generated translations on the test set with respect to the lengths of the sentences. The results are on the full test set which includes sentences having unknown words to the models.
Attention Mechanism

Assign attention weight to each word, to know how much "attention" the model should pay to each word (i.e., for each word, the network learns a "context")
Attention Mechanism

- Originally developed for language translation:

**Hidden state in a regular RNN (RNN #2)**

**Attention weight**

**1st input word**

**1st translated word**

**Bidirectional RNN (RNN #1)**

Figure 1: The graphical illustration of the proposed model trying to generate the \( t \)-th target word \( y_t \) given a source sentence \( (x_1, x_2, \ldots, x_T) \).
RNN Attention Mechanism

where the context vector $c_1$ is defined as
$$c_1 = \sum_{t=1}^{T} \alpha_{1,t} \ h_t$$

Added attention
(looks like a standard RNN but with context vectors as in-/output)

Bidirectional RNN

$$\hat{y}_1$$

$$S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow \ldots$$

$$c_1$$

$$\alpha_{1,1} \alpha_{1,2} \alpha_{1,T-1}$$

$$h_1$$

$$h_2$$

$$h_{T-1}$$

$$x_0 \rightarrow h_{F,1} \ h_{B,1}$$

$$x_1 \rightarrow h_{F,2} \ h_{B,2}$$

$$x_{T-1} \rightarrow h_{F,T-1} \ h_{B,T-1}$$

$$x_T$$
Computing attention weights

\[ S_{t-1} \rightarrow h_t' \rightarrow \text{Neural Net} \rightarrow e_{t,t'} \]

\[ \alpha_{t,t'} = \frac{\exp(e_{t,t'})}{\sum_{t'=1}^{T} \exp(e_{t,t'})} \]
Attention Mechanism

Computing attention weights

\[ \alpha_{t,t'} = \frac{\exp(e_{t,t'})}{\sum_{t'=1}^{T} \exp(e_{t,t'})} \]

Using Attention Without the RNN
-- Self-Attention Mechanism & Transformers

1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
   4.1. Basic Form of Self-Attention
   4.2. Self-Attention & Scaled Dot-Product Attention
   4.3. Multi-Head Attention
5. Transformer Models
   5.1. The Transformer Architecture
   5.2. Some Popular Transformer Models: BERT, GPT, and BART
6. Transformer in PyTorch
"Original" (RNN) Attention Mechanism

Where the context vector $c_1$ is defined as

$$c_1 = \sum_{t=1}^{T} \alpha_{1,t} h_t$$

And the attention weights are

$$\alpha_{t,t'} = \frac{\exp(e_{t,t'})}{\sum_{t'=1}^{T} \exp(e_{t,t'})}$$
Getting rid of the sequential parts...

- No recurrence, no convolution

- Transformers rely on the self-attention mechanism, processing the whole sequence all at once (no sequential processing like in RNNs)

- Transformers also have encoder & decoder parts. But instead of using LSTMs, they use stacked attention layers
Abstract
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

1 Introduction
Recurrent neural networks, long short-term memory [13] and gated recurrent [7] neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and

---


https://arxiv.org/abs/1706.03762

---
Since ~2018, Transformers have been growing in popularity ... and size

[Diagram showing the growth of Transformers in terms of popularity and size, with notable models like ELMo, GPT, BERT-Large, GPT-2, Transformer ELMo, XLNet, and DistilBERT, along with their respective years and size metrics.]
Self-Attention Mechanism
-- Very Basic Form

Main procedure:
1) Derive attention weights: similarity between current input and all other inputs (next slide)
2) Normalize weights via softmax (next slide)
3) Compute attention value from normalized weights and corresponding inputs (below)

Self-attention as weighted sum:

$$A_i = \sum_{j=1}^{T} a_{ij} x_j$$

output corresponding to the i-th input

weight based on similarity between current input $x_i$ and all other inputs
Self-Attention Mechanism
-- Very Basic Form

Self-attention as weighted sum:

\[ A_i = \sum_{j=0}^{T} a_{ij} x_j \]

output corresponding to the i-th input

weight based on similarity between current input \( x_i \) and all other inputs

How to compute the attention weights?

Here as simple dot product:

\[ e_{ij} = x_i^\top x_j \]

repeat this for all inputs \( j \in \{1...T\} \), then normalize

\[ a_{ij} = \frac{\exp(e_{ij})}{\sum_{j=1}^{T} \exp(e_{ij})} = \text{softmax} \left( \left[ e_{ij} \right]_{j=1,...,T} \right) \]
Self-Attention Mechanism
-- Very Basic Form

$$\omega_{i,i} = x_i^\top x_i$$

After computing these similarity-based weights for the $i$th input and all inputs in the sequence ($x_i$ to $x_T$), the "raw" weights ($\omega_{i,0}$ to $\omega_{i,i}$) are then normalized using the familiar softmax function, as follows:

$$W_{i,i} = \frac{\exp(\omega_{i,i})}{\sum_{j=0}^{T} \exp(\omega_{i,j})}$$

Notice that as a consequence of applying the softmax function, the weights will sum to 1 after this normalization, that is,$$
\sum_{j=0}^{T} W_{i,j} = 1
$$

To recap, let's summarize the three main steps behind the self-attention operation:

1. For a given input element, $x_i$, and each $j$th element in the range $[0, T]$,
   compute the dot product, $x_i^\top x_j$.
2. Obtain the weight, $W_{i,i}$, by normalizing the dot products using the softmax function.
3. Compute the output, $o_i$, as the weighted sum over the entire input sequence:
   $$o_i = \sum_{j=0}^{T} W_{i,j} x_j$$

These steps are further illustrated in the following figure:

Using Attention Without the RNN
-- Self-Attention Mechanism & Transformers

1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
   4.1. Basic Form of Self-Attention
   4.2. Self-Attention & Scaled Dot-Product Attention
   4.3. Multi-Head Attention
5. Transformer Models
   5.1. The Transformer Architecture
   5.2. Some Popular Transformer Models: BERT, GPT, and BART
6. Transformer in PyTorch
After computing these similarity-based weights for the $i$th input and all inputs in the sequence ($x_1$ to $x_T$), the "raw" weights ($\omega_{i0}$ to $\omega_{ii}$) are then normalized using the familiar softmax function, as follows:

$$W_i = \frac{\exp(\omega_{ii})}{\sum_{j=0}^{T} \exp(\omega_{ij})}$$

Notice that as a consequence of applying the softmax function, the weights will sum to 1 after this normalization, that is,

$$\sum_{T=0}^{T} W_i = 1$$

To recap, let's summarize the three main steps behind the self-attention operation:

1. For a given input element, $x_i$, and each $j$th element in the range $[0, T]$, compute the dot product, $x_i \top x_j$.
2. Obtain the weight, $W_i$, by normalizing the dot products using the softmax function.
3. Compute the output, $o_i$, as the weighted sum over the entire input sequence:

$$o_i = \sum_{T=0}^{T} W_i x_i$$

These steps are further illustrated in the following figure:

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

1 Introduction

Recurrent neural networks, long short-term memory [13] and gated recurrent [7] neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and

---


https://arxiv.org/abs/1706.03762
Self-Attention Mechanism

- Previous basic version did not involve any learnable parameters, so not very useful for learning a language model.
- We are now adding 3 trainable weight matrices that are multiplied with the input sequence embeddings ($x_i$'s)

\[
\begin{align*}
\text{query} &= W^q x_i \\
\text{key} &= W^k x_i \\
\text{value} &= W^v x_i
\end{align*}
\]
Self-Attention Mechanism

For each query, model learns which key-value input it should attend to.

As in the simplified version, this is a form of similarity or compatibility measure ("multiplicative attention")

\[
A(q_2, K, V) = \sum_{i=1}^{T} \frac{\exp(q_2 \cdot k_i^T)}{\sum_j \exp(q_2 \cdot k_j^T)} \times v_i
\]

weighted sum: values weighted by attention weight (softmax score)
Self-Attention Mechanism

\[ d_e = \text{embedding size (original transformer = 512)} \]

where \( d_q = d_k \)

In original transformer, \( d_q = d_v \) as well

\[
A(q_2, K, V) = \sum_{i=1}^{T} \frac{\exp(q_2 \cdot k_i^\top)}{\sum_j \exp(q_2 \cdot k_j^\top)} \times v_i
\]

softmax

\[ 1 \times 1 \]

\[ 1 \times d_v \]
Self-Attention Mechanism

Attention score matrix: \[ A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \end{bmatrix} \]
Self-Attention Mechanism (Scaled Dot Product Attention)

\(d_e = \) embedding size

\(T = \) input sequence size

\[\begin{align*}
x & \in \mathbb{R}^{T \times d_e} \\
Q & \in \mathbb{R}^{T \times d_q} \\
K & \in \mathbb{R}^{T \times d_k} \\
V & \in \mathbb{R}^{T \times d_v}
\end{align*}\]

\[
A(Q, K, V) = \text{softmax} \left( \frac{QK^\top}{\sqrt{d_k}} \right) V
\]
Scaled Dot-Product Attention

\[
A(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d_k}} \right) V
\]

To ensure that the dot-products between query and key don't grow too large (and softmax gradient become too small) for large \(d_k\)

Using Attention Without the RNN
-- Self-Attention Mechanism & Transformers

1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
   4.1. Basic Form of Self-Attention
   4.2. Self-Attention & Scaled Dot-Product Attention
   4.3. Multi-Head Attention
5. Transformer Models
   5.1. The Transformer Architecture
   5.2. Some Popular Transformer Models: BERT, GPT, and BART
6. Transformer in PyTorch
Self-Attention Mechanism (Scaled Dot Product Attention)

- $d_e = \text{embedding size}$
- $T = \text{input sequence size}$

$x \in \mathbb{R}^{T \times d_e}$

$Q \in \mathbb{R}^{T \times d_q}$

$K \in \mathbb{R}^{T \times d_k}$

$V \in \mathbb{R}^{T \times d_v}$

"attention matrix" $A(Q, K, V) = \text{softmax} \left( \frac{QK^\top}{\sqrt{d_k}} \right) V$

"attention-based embedding"
Multi-Head Attention

- Apply self-attention multiple times in parallel (similar to multiple kernels for channels in CNNs)

- For each head (self-attention layer), use different $W^q$, $W^k$, $W^v$, then concatenate the results, $A_{(i)}$

- 8 attention heads in the original transformer, i.e.,
  $W^q_{(1)}$, $W^k_{(1)}$, $W^v_{(1)}$, \ldots, $W^q_{(8)}$, $W^k_{(8)}$, $W^v_{(8)}$

- Allows attending to different parts in the sequence differently
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension \( d_k \), and values of dimension \( d_v \). We compute the dot products of the query with all keys, divide each by \( \frac{1}{\sqrt{d_k}} \), and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix \( Q \). The keys and values are also packed together into matrices \( K \) and \( V \). We compute the matrix of outputs as:

\[
\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V
\]

The two most commonly used attention functions are additive attention \(^2\), and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of \( \frac{1}{\sqrt{d_k}} \). Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of \( d_k \) the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of \( d_k \) \(^3\). We suspect that for large values of \( d_k \), the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients \(^4\). To counteract this effect, we scale the dot products by \( \frac{1}{\sqrt{d_k}} \).

3.2.2 Multi-Head Attention

Instead of performing a single attention function with \( d_{\text{model}} \)-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values \( h \) times with different, learned linear projections to \( d_k \), \( d_k \) and \( d_v \) dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding \( d_v \)-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

\[^4\] To illustrate why the dot products get large, assume that the components of \( q \) and \( k \) are independent random variables with mean 0 and variance 1. Then their dot product, \( q \cdot k = \sum_{i=1}^{d_k} q_i k_i \), has mean 0 and variance \( d_k \).

### 3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $p_{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

The two most commonly used attention functions are additive attention and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{p_{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{p_{d_k}}$.

### 3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

---


**Formula:**

$$A(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

**Input sequence dim. in original transformer:**

$$T \times d_e = T \times 512$$

and

$$d_v = 512/h = 64$$
3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\frac{1}{\sqrt{d_k}}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

The two most commonly used attention functions are additive attention \cite{2}, and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$ \cite{3}. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

\begin{align*}
V \cdot K &= d_v \cdot h \\
W_o \cdot V &= d_o
\end{align*}

In transformer paper:

$$d_v \times h = d_o$$

\cite{Vaswani2017}. Attention Is All You Need.

\begin{align*}
\text{Concatenated:} \\
&T \\
&T \\
&d_v \cdot h \\
&\ldots
\end{align*}
Using Attention Without the RNN
-- Self-Attention Mechanism & Transformers

1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
   4.1. Basic Form of Self-Attention
   4.2. Self-Attention & Scaled Dot-Product Attention
   4.3. Multi-Head Attention
5. Transformer Models
   5.1. The Transformer Architecture
   5.2. Some Popular Transformer Models: BERT, GPT, and BART
6. Transformer in PyTorch
Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

1 Introduction

Recurrent neural networks, long short-term memory [13] and gated recurrent [7] neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and...
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $p_{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{Q K^T}{\sqrt{d_k}}\right) V$$

The two most commonly used attention functions are additive attention, and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $1/p_{d_k}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.
3.2.1 Scaled Dot-Product Attention

We call our particular attention “Scaled Dot-Product Attention” (Figure 2). The input consists of queries and keys of dimension $d_k$, and values of dimension $d_v$. We compute the dot products of the query with all keys, divide each by $\frac{1}{\sqrt{d_k}}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix $Q$. The keys and values are also packed together into matrices $K$ and $V$. We compute the matrix of outputs as:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

The two most commonly used attention functions are additive attention \[2\], and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of $d_k$ the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of $d_k$ \[3\]. We suspect that for large values of $d_k$, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients \[4\]. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with $d_{\text{model}}$-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values $h$ times with different, learned linear projections to $d_k$, $d_k$ and $d_v$ dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding $d_v$-dimensional output values. These are concatenated and once again projected, resulting in the final values, as depicted in Figure 2.

4 To illustrate why the dot products get large, assume that the components of $q$ and $k$ are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance $d_k$.
While for small values of we call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of we found it beneficial to linearly project the queries, keys and values. The two most commonly used attention functions are additive attention \[ \text{Additive Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d}} \right) V \]
dot product attention without scaling for larger values of \( d \).

3.2.2 Multi-Head Attention

Instead of performing a single attention function with extremely small gradients, we can use multiple attention mechanisms in parallel, called multi-head attention. This allows the model to focus on different aspects of the input.

3.2.1 Scaled Dot-Product Attention

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix \( Q \) and keys and values \( K \) and \( V \). We compute the dot products of the queries with all keys, divide each by \( d \), and apply a softmax function to obtain the weights on the values.

\[
\text{Scaled Dot-Product Attention}(Q, K, V) = \text{softmax} \left( \frac{QK^T}{\sqrt{d}} \right) V
\]

Figure 1: The Transformer - model architecture.

3.2 Attention

A layer of the Transformer model contains a series of sub-layers. The first is a multi-head self-attention mechanism, which allows the model to learn relationships between different parts of the input. The second sub-layer is a position-wise fully connected feed-forward network, which processes the information in parallel. The output of each sub-layer is then combined through a residual connection, where the output of the sub-layer is added to the input of the sub-layer.

The attention mechanism computes a weighted sum of the input vectors, where the weights are determined by a compatibility function. This allows the model to focus on specific parts of the input when computing the output.

The decoder is also composed of a stack of identical layers, similar to the encoder. However, it contains an additional sub-layer to perform multi-head attention on the output of the encoder stack. This sub-layer is used to attend to the output of the encoder, allowing the decoder to incorporate information from the encoder into its own computations.

Both the encoder and decoder stacks are composed of identical layers. Each layer consists of two sub-layers, followed by layer normalization. The output embeddings are offset by one position, which helps prevent positions from attending to subsequent positions. This is achieved through the use of a masking mechanism.

The figure illustrates the structure of the Transformer model, showing how the encoder and decoder stacks are composed of identical layers. The input is processed through the encoder, and the output of the encoder is then passed to the decoder to generate the final output.
3.1 Encoder and Decoder Stacks

Encoder:
The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection \[ \text{LayerNorm}(x + \text{Sublayer}(x)) \] around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder:
The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position $i$ can depend only on the known outputs at positions less than $i$.

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Masked Multi-head attention

Mask subsequent sequence elements. I.e., only allow to attend to positions up to and including the current position. This is achieved by setting softmax values for those to $-\infty$.

Figure 1: The Transformer - model architecture.

3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers: the first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is $\text{LayerNorm}(\text{Sublayer}(x) + x)$, where $\text{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position $i$ can depend only on the known outputs at positions less than $i$.

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.
Figure 1: The Transformer - model architecture.

Figure 1: The Transformer - model architecture.

The Law will never be perfect, but its application should be just—this is what we are missing, in my opinion.

Figure 5: Many of the attention heads exhibit behaviour that seems related to the structure of the sentence. We give two such examples above, from two different heads from the encoder self-attention at layer 5 of 6. The heads clearly learned to perform different tasks.

Using Attention Without the RNN
-- Self-Attention Mechanism & Transformers

1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
   4.1. Basic Form of Self-Attention
   4.2. Self-Attention & Scaled Dot-Product Attention
   4.3. Multi-Head Attention
5. Transformer Models
   5.1. The Transformer Architecture
   5.2. Some Popular Transformer Models: BERT, GPT, and BART
6. Transformer in PyTorch
3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of \( N = 6 \) identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection \([11]\) around each of the two sub-layers, followed by layer normalization \([1] \). That is, the output of each sub-layer is \( \text{LayerNorm}(x + \text{Sublayer}(x)) \), where \( \text{Sublayer}(x) \) is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension \( d_{\text{model}} = 512 \).

Decoder: The decoder is also composed of a stack of \( N = 6 \) identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position \( i \) can depend only on the known outputs at positions less than \( i \).

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Recap


Figure 1: The Transformer - model architecture.
The Two Keys to Success Behind Transformers

1. Self-attention for encoding long-range dependencies

2. Self-supervision for leveraging large unlabeled datasets
1. Pre-training on large unlabeled datasets (self-supervised learning)

2. Training for downstream-tasks on labeled data (supervised learning)
   a) fine-tuning approach
   b) feature-based approach
5.2 Some Popular Transformer Models: BERT, GPT, and BART

- 5.2.2 GPT-v1: Generative Pre-Trained Transformer
- 5.2.3 BERT: Bidirectional Encoder Representations from Transformers
- 5.2.4 GPT-v2: Language Models are Unsupervised Multitask Learners
- 5.2.5 GPT-v3: Language Models are Few-Shot Learners
- 5.2.6 BART: Combining Bidirectional and Auto-Regressive Transformers
- 5.2.7: Closing Words -- The Recent Growth of Language Transformers
5.2 Some Popular Transformer Models: BERT, GPT, and BART

5.2.2 GPT-v1:
Generative Pre-Trained Transformer
GPT (Generative Pre-trained Transformer)

- Developed by OpenAI
- Unidirectional: trained to predict next word in a sentence

GPT (110 million parameters)

GPT-2 1.5 billion parameters)

GPT-3 (175 billion parameters)
GPT-v1 Key Concepts

- Bottleneck: Lack of labeled data
- 2-step training process ("semi-supervised")
  1. Generative pre-training (on unlabeled data); unsupervised/"self-supervised" learning
  2. Discriminative fine-tuning (on labeled data), supervised learning
- Pre-training on large BookCorpus dataset (7000 books)
- Based on decoder architecture from original Transformer ("Attention Is All You Need")
Figure 1: (left) Transformer architecture and training objectives used in this work. (right) Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

GPT-v1 Ablation Study

Table 5: Analysis of various model ablations on different tasks. Avg. score is a unweighted average of all the results. (mc= Mathews correlation, acc=Accuracy, pc=Pearson correlation)

<table>
<thead>
<tr>
<th>Method</th>
<th>Avg. Score</th>
<th>CoLA (mc)</th>
<th>SST2 (acc)</th>
<th>MRPC (F1)</th>
<th>STSB (pc)</th>
<th>QQP (F1)</th>
<th>MNLI (acc)</th>
<th>QNLI (acc)</th>
<th>RTE (acc)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transformer w/ aux LM (full)</td>
<td>74.7</td>
<td>45.4</td>
<td>91.3</td>
<td>82.3</td>
<td>82.0</td>
<td>70.3</td>
<td>81.8</td>
<td>88.1</td>
<td>56.0</td>
</tr>
<tr>
<td>Transformer w/o pre-training</td>
<td>59.9</td>
<td>18.9</td>
<td>84.0</td>
<td>79.4</td>
<td>30.9</td>
<td>65.5</td>
<td>75.7</td>
<td>71.2</td>
<td>53.8</td>
</tr>
<tr>
<td>Transformer w/o aux LM</td>
<td>75.0</td>
<td>47.9</td>
<td>92.0</td>
<td>84.9</td>
<td>83.2</td>
<td>69.8</td>
<td>81.1</td>
<td>86.9</td>
<td>54.4</td>
</tr>
<tr>
<td>LSTM w/ aux LM</td>
<td>69.1</td>
<td>30.3</td>
<td>90.5</td>
<td>83.2</td>
<td>71.8</td>
<td>68.1</td>
<td>73.7</td>
<td>81.1</td>
<td>54.6</td>
</tr>
</tbody>
</table>
5.2 Some Popular Transformer Models: BERT, GPT, and BART

5.2.3 BERT: Bidirectional Encoder Representations from Transformers
BERT (Bidirectional Encoder Representations from Transformers)


• multi-layer bidirectional transformer encoder
• architecture almost identical to original transformer & GPT, except
  • bidirectional masking (known as "Cloze" task, Taylor 1953*)
  • next sentence prediction as additional pre-training task

**BERT Inputs**

![Diagram of BERT input representation](image)

Figure 2: BERT input representation. The input embeddings are the sum of the token embeddings, the segmentation embeddings and the position embeddings.

**Token embeddings are WordPiece embeddings** with vocabulary size of 30,000

BERT Pre-Training Tasks

Pre-training datasets
- BookCorpus (800 million words)
- Wikipedia (2500 million words)

Pre-training tasks
- Masked language model ("Cloze")
- Next sentence prediction
BERT Pre-Training Task #1

Masked Language Model

Input sentence: A quick brown fox jumps over the lazy dog

80%: replace with [MASK]

"Mark" 15% of the words

10%: replace with random word (coffee)

10%: leave as is (fox) to mimick fine-tuning scenario
BERT Pre-Training Task #1

Masked Language Model

Input sentence: A quick brown fox jumps over the lazy dog

Randomly masked: A quick brown [MASK] jumps over the lazy dog

Possible classes (all words):
- ant ...
- fox ...
- zoo

0.2%
0.01%
BERT Pre-Training Task #2

Next Sentence Prediction

Balanced binary classification task (50% IsNext, 50% NotNext)

Input = [CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk [SEP]
Label = IsNext

Input = [CLS] the man [MASK] to the store [SEP] penguin [MASK] are flight ##less birds [SEP]
Label = NotNext
BERT Pre-Training & Downstream Tasks

There is mini-NSP as a running example for this section. The following question-answering example in Figure 1: Overall pre-training and fine-tuning procedures for BERT. Apart from output layers, the same architecture is used in both pre-training and fine-tuning. The same pre-trained model parameters are used to initialize models for different down-stream tasks. During fine-tuning, all parameters are fine-tuned. [CLS] is a special symbol added in front of every input example, and [SEP] is a special separator token (e.g., separating questions/answers).
Transformer Training Approach

1. **Pre-training** on large unlabeled datasets (self-supervised learning)

2. **Training for downstream-tasks** on labeled data (supervised learning)
   a) fine-tuning approach
   b) feature-based approach (nowadays also called "fine-tuning")
BERT Pre-Training & Fine-Tuning Approach

- Add classification layer
- Train end-to-end on labeled dataset for downstream task (update ALL parameters)

Figure 1: The Transformer - model architecture.

3.1 Encoder and Decoder Stacks

**Encoder:**
The encoder is composed of a stack of \( N = 6 \) identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is:

\[
\text{LayerNorm}(x + \text{Sublayer}(x))
\]

where \( \text{Sublayer}(x) \) is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension \( d_{\text{model}} = 512 \).

**Decoder:**
The decoder is also composed of a stack of \( N = 6 \) identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position \( i \) can depend only on the known outputs at positions less than \( i \).

3.2 Attention
An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.
BERT vs GPT-v1 Performance

<table>
<thead>
<tr>
<th>System</th>
<th>MNLI-(m/mm)</th>
<th>QQP</th>
<th>QNLI</th>
<th>SST-2</th>
<th>CoLA</th>
<th>STS-B</th>
<th>MRPC</th>
<th>RTE</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>392k</td>
<td>363k</td>
<td>108k</td>
<td>67k</td>
<td>8.5k</td>
<td>5.7k</td>
<td>3.5k</td>
<td>2.5k</td>
<td></td>
</tr>
<tr>
<td>Pre-OpenAI SOTA</td>
<td>80.6/80.1</td>
<td>66.1</td>
<td>82.3</td>
<td>93.2</td>
<td>35.0</td>
<td>81.0</td>
<td>86.0</td>
<td>61.7</td>
<td>74.0</td>
</tr>
<tr>
<td>BiLSTM+ELMo+Attn</td>
<td>76.4/76.1</td>
<td>64.8</td>
<td>79.8</td>
<td>90.4</td>
<td>36.0</td>
<td>73.3</td>
<td>84.9</td>
<td>56.8</td>
<td>71.0</td>
</tr>
<tr>
<td>OpenAI GPT</td>
<td>82.1/81.4</td>
<td>70.3</td>
<td>87.4</td>
<td>91.3</td>
<td>45.4</td>
<td>80.0</td>
<td>82.3</td>
<td>56.0</td>
<td>75.1</td>
</tr>
<tr>
<td>BERT&lt;sub&gt;BASE&lt;/sub&gt;</td>
<td>84.6/83.4</td>
<td>71.2</td>
<td>90.5</td>
<td>93.5</td>
<td>52.1</td>
<td>85.8</td>
<td>88.9</td>
<td>66.4</td>
<td>79.6</td>
</tr>
<tr>
<td>BERT&lt;sub&gt;LARGE&lt;/sub&gt;</td>
<td><strong>86.7/85.9</strong></td>
<td><strong>72.1</strong></td>
<td><strong>92.7</strong></td>
<td><strong>94.9</strong></td>
<td><strong>60.5</strong></td>
<td><strong>86.5</strong></td>
<td><strong>89.3</strong></td>
<td><strong>70.1</strong></td>
<td><strong>82.1</strong></td>
</tr>
</tbody>
</table>

Table 1: GLUE Test results, scored by the evaluation server (https://gluebenchmark.com/leaderboard). The number below each task denotes the number of training examples. The “Average” column is slightly different than the official GLUE score, since we exclude the problematic WNLI set. BERT and OpenAI GPT are single-model, single task. F1 scores are reported for QQP and MRPC, Spearman correlations are reported for STS-B, and accuracy scores are reported for the other tasks. We exclude entries that use BERT as one of their components.
BERT Pre-Training & Feature-based Training

- Keep BERT frozen after pre-training
- Create BERT embeddings for labeled dataset for downstream task and train new model on these embeddings (in original paper, 2-layer biLSTM on embeddings from concatenated last 4 layers performed best)

1) Download BERT pre-trained on large corpus (in self-supervised fashion)

2) Feature-based training ("fine-tuning") on target task (supervised learning)
mixed results on the downstream task impact of increasing the pre-trained bi-LM size from two to four layers and Melamud et al. (2016) mentioned in passing that increasing hidden dimension size from 200 to 600 helped, but increasing further to 1,000 did not bring further improvements. Both of these prior works used a feature-based approach — we hypothesize that when the model is fine-tuned directly on the downstream tasks and uses only a very small number of randomly initialized additional parameters, the task-specific models can benefit from the larger, more expressive pre-trained representations even when downstream task data is very small.

5.3 Feature-based Approach with BERT

All of the BERT results presented so far have used the fine-tuning approach, where a simple classification layer is added to the pre-trained model, and all parameters are jointly fine-tuned on a downstream task. However, the feature-based approach, where fixed features are extracted from the pre-trained model, has certain advantages. First, not all tasks can be easily represented by a Transformer encoder architecture, and therefore require a task-specific model architecture to be added. Second, there are major computational benefits to pre-compute an expensive representation of the training data once and then run many experiments with cheaper models on top of this representation. In this section, we compare the two approaches by applying BERT to the CoNLL-2003 Named Entity Recognition (NER) task (Tjong Kim Sang and De Meulder, 2003). In the input to BERT, we use a case-preserving WordPiece model, and we include the maximal document context provided by the data. Following standard practice, we formulate this as a tagging task but do not use a CRF.

<table>
<thead>
<tr>
<th>System</th>
<th>Dev F1</th>
<th>Test F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELMo (Peters et al., 2018a)</td>
<td>95.7</td>
<td>92.2</td>
</tr>
<tr>
<td>CVT (Clark et al., 2018)</td>
<td>-</td>
<td>92.6</td>
</tr>
<tr>
<td>CSE (Akbik et al., 2018)</td>
<td>-</td>
<td><strong>93.1</strong></td>
</tr>
<tr>
<td>Fine-tuning approach</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BERT\textsubscript{LARGE}</td>
<td>96.6</td>
<td>92.8</td>
</tr>
<tr>
<td>BERT\textsubscript{BASE}</td>
<td>96.4</td>
<td>92.4</td>
</tr>
<tr>
<td>Feature-based approach (BERT\textsubscript{BASE})</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Embeddings</td>
<td>91.0</td>
<td>-</td>
</tr>
<tr>
<td>Second-to-Last Hidden</td>
<td>95.6</td>
<td>-</td>
</tr>
<tr>
<td>Last Hidden</td>
<td>94.9</td>
<td>-</td>
</tr>
<tr>
<td>Weighted Sum Last Four Hidden</td>
<td>95.9</td>
<td>-</td>
</tr>
<tr>
<td>Concat Last Four Hidden</td>
<td>96.1</td>
<td>-</td>
</tr>
<tr>
<td>Weighted Sum All 12 Layers</td>
<td>95.5</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 7: CoNLL-2003 Named Entity Recognition results. Hyperparameters were selected using the Dev set. The reported Dev and Test scores are averaged over 5 random restarts using those hyperparameters.
5.2 Some Popular Transformer Models: BERT, GPT, and BART

5.2.4 GPT-v2: Language Models are Unsupervised Multitask Learners
GPT (Generative Pre-trained Transformer)

• Developed by OpenAI
• Unidirectional: trained to predict next word in a sentence

GPT (110 million parameters)

GPT-2 (1.5 billion parameters)

GPT-3 (175 billion parameters)
Figure 1: (left) Transformer architecture and training objectives used in this work. (right) Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

GPT-v2 Key Concepts

- Unidirectional like GPT-v1
- Compared to GPT-v1
  - Larger model (the larger the better)
  - Larger unlabeled dataset (the larger the better)
  - No fine-tuning (use zero-shot transfer instead)
GPT-v2 Architecture

• Overall, similar to GPT-v1 (which is based on original Transformer decoder)
• Some small rearranging of layer norm and residual layers
• Increase vocabulary size from 30,000 -> 50,257
• Increase context size from 512 -> 1024 tokens
• Overall, 1.5 billion instead of 110 million parameters
GPT-v2 Training Dataset

- WebText (millions of webpages)
- Emphasized dataset quality
- Based on Reddit posts with more than 3 karma
  - Get 45 million links to websites
    - After preprocessing and cleaning: 8 million documents
    - 40 Gb of text
Zero-Shot Task Transfer

In contrast to GPT-v1, no specific instruction / rearranging for specific tasks

https://huggingface.co/models?filter=zero-shot-classification

Zero Shot Topic Classification

Choose an example

Custom

Text

What is the color of grass?

Possible topics (separated by `,`)

green, red, blue, nothing, pink, purple

Allow multiple correct topics

Top Predictions

<table>
<thead>
<tr>
<th>Label</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>green</td>
<td>72.9%</td>
</tr>
<tr>
<td>red</td>
<td>1.6%</td>
</tr>
<tr>
<td>nothing</td>
<td>0.6%</td>
</tr>
<tr>
<td>blue</td>
<td>0.5%</td>
</tr>
<tr>
<td>purple</td>
<td>0.3%</td>
</tr>
</tbody>
</table>

Update: Zero-shot classification is now supported in our API and you can experiment with a number of compatible models on our Model Hub.

Recently, the NLP science community has begun to pay increasing attention to zero-shot and few-shot applications, such as in the paper from OpenAI introducing GPT-3. This demo shows how Transformers can be used for zero-shot topic classification, the task of predicting a topic that the model has not been trained on.
Figure 1. wards more flexible forms of transfer. First, word vectors these methods still require supervised training in order language modeling is usually framed as unsupervised distri-

2. Approach achieve promising, competitive, and state of the art results perform a wide range of tasks in a zero-shot setting. We demonstrate language models can perform down-stream architecture modification. We demonstrate this approach shows continue the trend of more general methods of transfer. We potential by highlighting the ability of language models to

Table 3. WebText LMs transfer well across domains and datasets, transferred (. . . , s

Language Models are Unsupervised Multitask Learners

<table>
<thead>
<tr>
<th></th>
<th>LAMBADA (PPL)</th>
<th>LAMBADA (ACC)</th>
<th>CBT-CN (ACC)</th>
<th>CBT-NE (ACC)</th>
<th>WikiText2 (PPL)</th>
<th>PTB (PPL)</th>
<th>enwik8 (BPB)</th>
<th>text8 (BPC)</th>
<th>WikiText103 (PPL)</th>
<th>1BW* (PPL)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOTA</td>
<td>99.8</td>
<td>59.23</td>
<td>85.7</td>
<td>82.3</td>
<td>39.14</td>
<td>46.54</td>
<td>0.99</td>
<td>1.08</td>
<td>18.3</td>
<td>21.8</td>
</tr>
<tr>
<td>117M</td>
<td>35.13</td>
<td>45.99</td>
<td>87.65</td>
<td>83.4</td>
<td>29.41</td>
<td>65.85</td>
<td>1.16</td>
<td>1.17</td>
<td>37.50</td>
<td>75.20</td>
</tr>
<tr>
<td>345M</td>
<td>15.60</td>
<td>55.48</td>
<td>92.35</td>
<td>87.1</td>
<td>22.76</td>
<td>47.33</td>
<td>1.01</td>
<td>1.06</td>
<td>26.37</td>
<td>55.72</td>
</tr>
<tr>
<td>762M</td>
<td>10.87</td>
<td>60.12</td>
<td>93.45</td>
<td>88.0</td>
<td>19.93</td>
<td>40.31</td>
<td>0.97</td>
<td>1.02</td>
<td>22.05</td>
<td>44.575</td>
</tr>
<tr>
<td>1542M</td>
<td>8.63</td>
<td>63.24</td>
<td>93.30</td>
<td>89.05</td>
<td>18.34</td>
<td>35.76</td>
<td>0.93</td>
<td>0.98</td>
<td>17.48</td>
<td>42.16</td>
</tr>
</tbody>
</table>

* bad 1BW performance—probably due to sentence-level reshuffling in that dataset, so larger, long-range contexts are lost
5.2 Some Popular Transformer Models: BERT, GPT, and BART

5.2.5 GPT-v3: Language Models are Few-Shot Learners
GPT (Generative Pre-trained Transformer)

- Developed by OpenAI
- Unidirectional: trained to predict next word in a sentence

GPT (110 million parameters)

GPT-2 1.5 billion parameters)

GPT-3 (175 billion parameters)
GPT-v3 Architecture

• Overall, similar to GPT-v2
• 175 billion instead 1.5 billion parameters (more layers etc.)
• Double the context size (2048 instead of 1024)
• Larger word embeddings (12.8k instead of 1.6k)
• Attention pattern from Sparse Transformer*

GPT-v3 Training Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Quantity (tokens)</th>
<th>Weight in training mix</th>
<th>Epochs elapsed when training for 300B tokens</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Crawl (filtered)</td>
<td>410 billion</td>
<td>60%</td>
<td>0.44</td>
</tr>
<tr>
<td>WebText2</td>
<td>19 billion</td>
<td>22%</td>
<td>2.9</td>
</tr>
<tr>
<td>Books1</td>
<td>12 billion</td>
<td>8%</td>
<td>1.9</td>
</tr>
<tr>
<td>Books2</td>
<td>55 billion</td>
<td>8%</td>
<td>0.43</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>3 billion</td>
<td>3%</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Table 2.2: Datasets used to train GPT-3. “Weight in training mix” refers to the fraction of examples during training that are drawn from a given dataset, which we intentionally do not make proportional to the size of the dataset. As a result, when we train for 300 billion tokens, some datasets are seen up to 3.4 times during training while other datasets are seen less than once.
**Implicit Task Learning**

(... While Learning to Predict the Next Word)

**Figure 1.1: Language model meta-learning.** During unsupervised pre-training, a language model develops a broad set of skills and pattern recognition abilities. It then uses these abilities at inference time to rapidly adapt to or recognize the desired task. We use the term “in-context learning” to describe the inner loop of this process, which occurs within the forward-pass upon each sequence. The sequences in this diagram are not intended to be representative of the data a model would see during pre-training, but are intended to show that there are sometimes repeated sub-tasks embedded within a single sequence.
Figure 2.1: Zero-shot, one-shot and few-shot, contrasted with traditional fine-tuning. The panels above show four methods for performing a task with a language model – fine-tuning is the traditional method, whereas zero-, one-, and few-shot, which we study in this work, require the model to perform the task with only forward passes at test time. We typically present the model with a few dozen examples in the few shot setting. Exact phrasings for all task descriptions, examples and prompts can be found in Appendix G.
Some of the Many Results ...

**Figure 3.3:** On TriviaQA GPT3’s performance grows smoothly with model size, suggesting that language models continue to absorb knowledge as their capacity increases. One-shot and few-shot performance make significant gains over zero-shot behavior, matching and exceeding the performance of the SOTA fine-tuned open-domain model, RAG LPP+20.
5.2 Some Popular Transformer Models: BERT, GPT, and BART

5.2.6 BART: Combining Bidirectional and Auto-Regressive Transformers
BART: Combining Bidirectional and Auto-Regressive Transformers


Facebook AI's BART combines Google's BERT and OpenAI's GPT

**BERT's** bidirectional, **autoencoder** nature is ...
+ good for downstream tasks (e.g., classification) that require info about the whole sequence
- not so good for generation tasks where generated word should only depend on previously generated words

**GPT's** unidirectional, **autoregressive** approach is ...
+ good for text generation
- not so good for tasks that require info of whole sequence, e.g., classification

BART is the best of both worlds
BART: BERT Encoder + GPT Decoder + Noise Transformations

(a) BERT: Random tokens are replaced with masks, and the document is encoded bidirectionally. Missing tokens are predicted independently, so BERT cannot easily be used for generation.

(b) GPT: Tokens are predicted auto-regressively, meaning GPT can be used for generation. However words can only condition on leftward context, so it cannot learn bidirectional interactions.

(c) BART: Inputs to the encoder need not be aligned with decoder outputs, allowing arbitrary noise transformations. Here, a document has been corrupted by replacing spans of text with mask symbols. The corrupted document (left) is encoded with a bidirectional model, and then the likelihood of the original document (right) is calculated with an autoregressive decoder. For fine-tuning, an uncorrupted document is input to both the encoder and decoder, and we use representations from the final hidden state of the decoder.
3.1 Encoder and Decoder Stacks

Encoder:
The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection \[ \text{LayerNorm}(x + \text{Sublayer}(x)) \] around each of the two sub-layers, followed by layer normalization. That is, the output of each sub-layer is $\text{LayerNorm}(x + \text{Sublayer}(x))$, where $\text{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder:
The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position $i$ can depend only on the known outputs at positions less than $i$.

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Noise Transformations in BART for Pre-Training on Unlabeled Data

Figure 2: Transformations for noising the input that we experiment with. These transformations can be composed.

Like a denoising autoencoder, it optimizes reconstruction loss
BART Performance Under Different Noise Transformations

<table>
<thead>
<tr>
<th>Model</th>
<th>SQuAD 1.1 F1</th>
<th>MNLI Acc</th>
<th>ELI5 PPL</th>
<th>XSum PPL</th>
<th>ConvAI2 PPL</th>
<th>CNN/DM PPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT Base (Devlin et al., 2019)</td>
<td>88.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Masked Language Model</td>
<td>90.0</td>
<td>83.5</td>
<td>24.77</td>
<td>7.87</td>
<td>12.59</td>
<td>7.06</td>
</tr>
<tr>
<td>Masked Seq2seq</td>
<td>87.0</td>
<td>82.1</td>
<td>23.40</td>
<td>6.80</td>
<td>11.43</td>
<td>6.19</td>
</tr>
<tr>
<td>Language Model</td>
<td>76.7</td>
<td>80.1</td>
<td><strong>21.40</strong></td>
<td>7.00</td>
<td>11.51</td>
<td>6.56</td>
</tr>
<tr>
<td>Permutated Language Model</td>
<td>89.1</td>
<td>83.7</td>
<td>24.03</td>
<td>7.69</td>
<td>12.23</td>
<td>6.96</td>
</tr>
<tr>
<td>Multitask Masked Language Model</td>
<td>89.2</td>
<td>82.4</td>
<td>23.73</td>
<td>7.50</td>
<td>12.39</td>
<td>6.74</td>
</tr>
<tr>
<td>BART Base</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w/ Token Masking</td>
<td>90.4</td>
<td>84.1</td>
<td>25.05</td>
<td>7.08</td>
<td>11.73</td>
<td>6.10</td>
</tr>
<tr>
<td>w/ Token Deletion</td>
<td>90.4</td>
<td>84.1</td>
<td>24.61</td>
<td>6.90</td>
<td>11.46</td>
<td>5.87</td>
</tr>
<tr>
<td>w/ Text Infilling</td>
<td><strong>90.8</strong></td>
<td>84.0</td>
<td>24.26</td>
<td><strong>6.61</strong></td>
<td><strong>11.05</strong></td>
<td>5.83</td>
</tr>
<tr>
<td>w/ Document Rotation</td>
<td>77.2</td>
<td>75.3</td>
<td>53.69</td>
<td>17.14</td>
<td>19.87</td>
<td>10.59</td>
</tr>
<tr>
<td>w/ Sentence Shuffling</td>
<td>85.4</td>
<td>81.5</td>
<td>41.87</td>
<td>10.93</td>
<td>16.67</td>
<td>7.89</td>
</tr>
<tr>
<td>w/ Text Infilling + Sentence Shuffling</td>
<td><strong>90.8</strong></td>
<td>83.8</td>
<td>24.17</td>
<td>6.62</td>
<td>11.12</td>
<td><strong>5.41</strong></td>
</tr>
</tbody>
</table>
Fine-Tuning on Labeled Data

(a) To use BART for classification problems, the same input is fed into the encoder and decoder, and the representation from the final output is used.

(b) For machine translation, we learn a small additional encoder that replaces the word embeddings in BART. The new encoder can use a disjoint vocabulary.

Figure 3: Fine tuning BART for classification and translation.
BART Performance for Discriminative Tasks

<table>
<thead>
<tr>
<th></th>
<th>SQuAD 1.1 EM/F1</th>
<th>SQuAD 2.0 EM/F1</th>
<th>MNLI Acc</th>
<th>SST Acc</th>
<th>QQP Acc</th>
<th>QNLI Acc</th>
<th>STS-B Acc</th>
<th>RTE Acc</th>
<th>MRPC Acc</th>
<th>CoLA Mcc</th>
</tr>
</thead>
<tbody>
<tr>
<td>BERT</td>
<td>84.1/90.9</td>
<td>79.0/81.8</td>
<td>86.6/-</td>
<td>93.2</td>
<td>91.3</td>
<td>92.3</td>
<td>90.0</td>
<td>70.4</td>
<td>88.0</td>
<td>60.6</td>
</tr>
<tr>
<td>UniLM</td>
<td>-/-</td>
<td>80.5/83.4</td>
<td>87.0/85.9</td>
<td>94.5</td>
<td>-</td>
<td>92.7</td>
<td>-</td>
<td>70.9</td>
<td>-</td>
<td>61.1</td>
</tr>
<tr>
<td>XLNet</td>
<td><strong>89.0/94.5</strong></td>
<td>86.1/88.8</td>
<td>95.6</td>
<td>91.8</td>
<td>93.9</td>
<td>91.8</td>
<td>83.8</td>
<td>89.2</td>
<td>63.6</td>
<td></td>
</tr>
<tr>
<td>RoBERTa</td>
<td>88.9/94.6</td>
<td><strong>86.5/89.4</strong></td>
<td><strong>90.2/90.2</strong></td>
<td>96.4</td>
<td>92.2</td>
<td>94.7</td>
<td><strong>92.4</strong></td>
<td>86.6</td>
<td><strong>90.9</strong></td>
<td><strong>68.0</strong></td>
</tr>
<tr>
<td>BART</td>
<td><strong>88.8/94.6</strong></td>
<td>86.1/89.2</td>
<td>89.9/90.1</td>
<td><strong>96.6</strong></td>
<td><strong>92.5</strong></td>
<td><strong>94.9</strong></td>
<td>91.2</td>
<td><strong>87.0</strong></td>
<td>90.4</td>
<td>62.8</td>
</tr>
</tbody>
</table>

Table 2: Results for large models on SQuAD and GLUE tasks. BART performs comparably to RoBERTa and XLNet, suggesting that BART’s uni-directional decoder layers do not reduce performance on discriminative tasks.
BART Performance for Generative Tasks

<table>
<thead>
<tr>
<th></th>
<th>CNN/DailyMail</th>
<th>XSum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
</tr>
<tr>
<td>Lead-3</td>
<td>40.42</td>
<td>17.62</td>
</tr>
<tr>
<td>PTGEN (See et al., 2017)</td>
<td>36.44</td>
<td>15.66</td>
</tr>
<tr>
<td>PTGEN+COV (See et al., 2017)</td>
<td>39.53</td>
<td>17.28</td>
</tr>
<tr>
<td>UniLM</td>
<td>43.33</td>
<td>20.21</td>
</tr>
<tr>
<td>BERTSUMABS (Liu &amp; Lapata, 2019)</td>
<td>41.72</td>
<td>19.39</td>
</tr>
<tr>
<td>BERTSUMEXTABS (Liu &amp; Lapata, 2019)</td>
<td>42.13</td>
<td>19.60</td>
</tr>
<tr>
<td>BART</td>
<td><strong>44.16</strong></td>
<td><strong>21.28</strong></td>
</tr>
</tbody>
</table>

Table 3: Results on two standard summarization datasets. BART outperforms previous work on summarization on two tasks and all metrics, with gains of roughly 6 points on the more abstractive dataset.

<table>
<thead>
<tr>
<th></th>
<th>Valid F1</th>
<th>Valid PPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq + Attention</td>
<td>16.02</td>
<td>35.07</td>
</tr>
<tr>
<td>Best System</td>
<td>19.09</td>
<td>17.51</td>
</tr>
<tr>
<td>BART</td>
<td><strong>20.72</strong></td>
<td><strong>11.85</strong></td>
</tr>
</tbody>
</table>

Table 4: BART outperforms previous work on conversational response generation. Perplexities are renormalized based on official tokenizer for ConvAI2.

<table>
<thead>
<tr>
<th></th>
<th>ELI5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
</tr>
<tr>
<td>Best Extractive</td>
<td>23.5</td>
</tr>
<tr>
<td>Language Model</td>
<td>27.8</td>
</tr>
<tr>
<td>Seq2Seq</td>
<td>28.3</td>
</tr>
<tr>
<td>Seq2Seq Multitask</td>
<td>28.9</td>
</tr>
<tr>
<td>BART</td>
<td><strong>30.6</strong></td>
</tr>
</tbody>
</table>

Table 5: BART achieves state-of-the-art results on the challenging ELI5 abstractive question answering dataset. Comparison models are from Fan et al. (2019).
5.2 Some Popular Transformer Models: BERT, GPT, and BART

5.2.7: Closing Words
-- The Recent Growth of Language Transformers
Transformers for Longer Sequences

Transformer-XL:
• encoder-free, decoder-only model
• trained to predict next word in sentence
• uses hidden states to remember previous (512-token) text segment

Longformer:
• instead of attention mechanism that scales quadratically, uses attention mechanism that scales linearly with sequence length
• uses extremely long text segments (thousands of tokens); similar to RoBERTa
Image Source: https://medium.com/huggingface/distilbert-8cf3380435b5
OpenAI’s text-generating system GPT-3 is now spewing out 4.5 billion words a day

Robot-generated writing looks set to be the next big thing

By James Vincent  |  Mar 29, 2021, 8:24am EDT

https://www.theverge.com/2021/3/29/22356180/openai-gpt-3-text-generation-words-day
THE COST OF TRAINING NLP MODELS
A CONCISE OVERVIEW

Or Sharir
AI21 Labs
ors@ai21.com

Barak Peleg
AI21 Labs
barakp@ai21.com

Yoav Shoham
AI21 Labs
yoavs@ai21.com

April 2020


Costs: Not for the faint hearted

• $2.5k - $50k (110 million parameter model)
• $10k - $200k (340 million parameter model)
• $80k - $1.6m (1.5 billion parameter model)
Transformers for Better Efficiency

**Reformer:**
- dot-product attention is replaced with locality-sensitive hashing (LSH) attention
- this achieves attention with $O(n \log(n))$ instead of $O(n^2)$ memory cost


**ALBERT:**
- 5x smaller size as BERT at same performance, due to compression via pruning
1. Sequence Generation with RNNs
2. Character RNN in PyTorch
3. RNNs with Attention
4. Attention is All We Need
   4.1. Basic Form of Self-Attention
   4.2. Self-Attention & Scaled Dot-Product Attention
   4.3. Multi-Head Attention
5. Transformer Models
   5.1. The Transformer Architecture
   5.2. Some Popular Transformer Models: BERT, GPT, and BART
6. **Transformer in PyTorch**