Generative AI: opdracht 3: Generating text, using LSTMs, attention and multi-headed attention

Wilfred Van Casteren
nr:837377516
Open Universiteit
Masteropleiding AI

February 18, 2026

Introduction

Text generation In this report I describe my ventures into the world of Generative AI. In this report I will describe 3 LSTM-based text generating models. The models gradually build upon each other.

LSTMs The main layer in the models will be an LSTM. So before anything else I will describe the way LSTMs work.

Attention As attention will be part of two out of three models, I will pay some attention to it as well.

Three text generation models There is a basic model using one LSTM layer. Furthermore there is a more complex model using two LSTM layers and attention. The most complex model has two LSTM layers and multi-headed attention. I will prove the models are trainable and will show some text they produced.

Optimizers While experimenting with a more complex, loss started oscillating, this inclined me to investigate the way optimizers work.

Lightning My regular training loop didn't prove to be very flexible, so I started using torch-lightning. To use this framework, I needed to refactor my training loop, which didn't work immediately.

Long Short-Term Memory networks

Long Short-Term Memory (LSTM) networks possess several key qualities that make them well suited for modeling sequential and temporal data. LSTM networks are a special case of the more generic RNN networks. These recurrent networks are chains of LSTM cells or RNN cells respectively.

RNNs

An RNN cell and its context is depicted in Figure 1

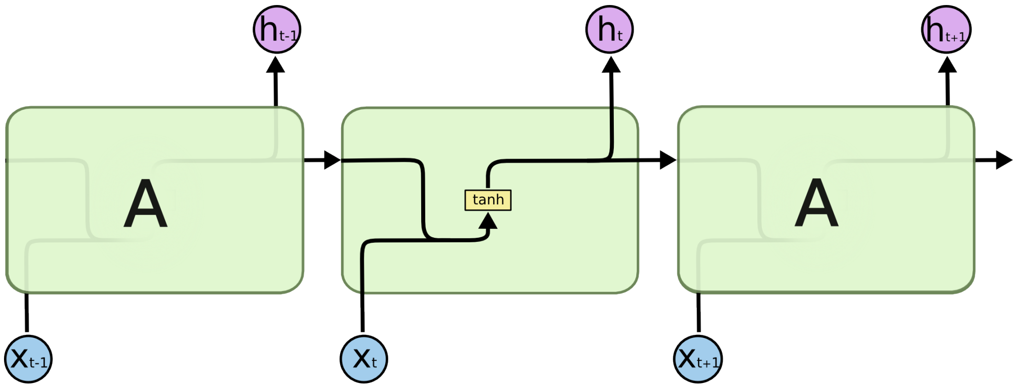


Figure 1: An RNN cell in a chain of RNN cells.

LSTMs

The more complex LSTM cell and its context is depicted in Figure 2.

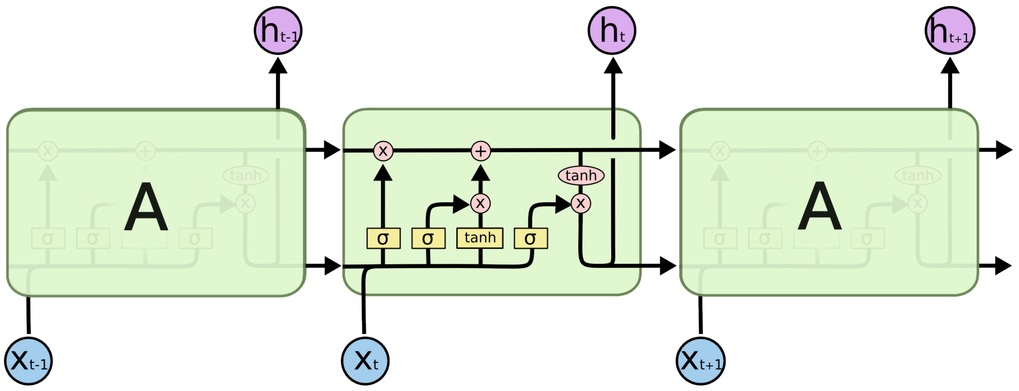


Figure 2: An LSTM cell in a chain of LSTM cells.

At each time step t, the LSTM cell receives an input vector x_t, the previous hidden state h_{t-1}, and the previous cell state c_{t-1}. It outputs an updated hidden state h_t and cell state c_t.

Long-Term Dependency Modeling LSTMs are specifically designed to capture long-range dependencies in sequences. The presence of a memory cell with additive updates allows gradients to flow across many time steps, mitigating the vanishing gradient problem.

Gated Information Control LSTMs employ gating mechanisms (forget, input, and output gates) that regulate the flow of information. These gates enable the network to selectively retain, update, or discard information based on relevance.

Mechanism

The mechanism of an LSTM cell is explained in the following part. This part on the mechanism of LSTM cells is actually based on this webpage. I did add some correct formulas. And changed the wording.

A chain of LSTM cells

An LSTM cell differs from the RNN cell's structure. Instead of having a single gate, there are four. And this makes its structure a bit more complicated, as can be seen in Figure 3.

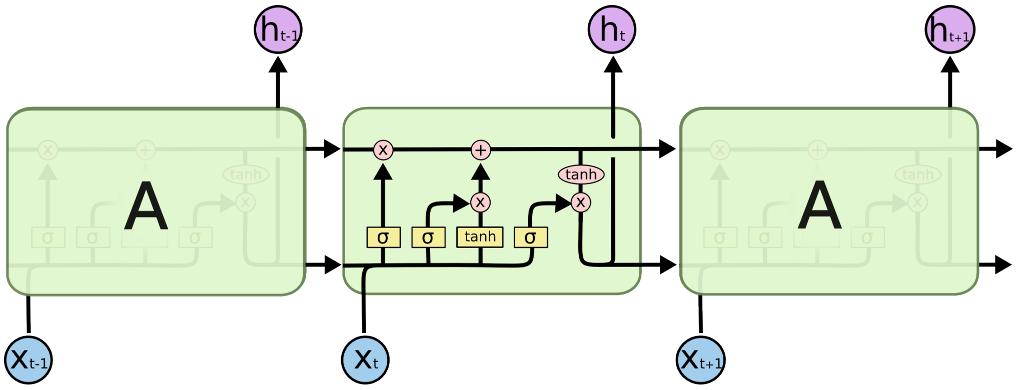


Figure 3: An LSTM cell in a chain of LSTM cells.

Main idea

The cell state C_t serves as a long-term memory channel with controlled linear self-connections, enabling gradient preservation across long sequences. The gates learn to selectively retain, update, and expose information, allowing the LSTM to model long-range temporal dependencies effectively. The separation between memory (C_t) and output (h_t) is the defining feature that distinguishes LSTMs from standard recurrent neural networks.

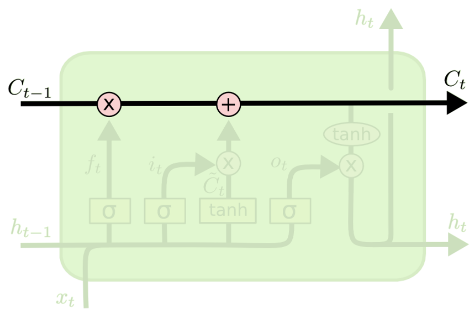


Figure 4: The cell state.

Gates

An LSTM cell does have the ability to remove information from or add information to the cell state, carefully regulated by gates. Gates are a way to optionally let information through.

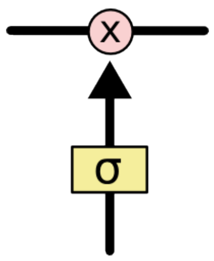


Figure 5: A gate.

Activation layers Something about activation layers used in the gates.

Sigmoid A sigmoid operation produces numbers between zero and one, describing how much of each component should be let through.

Tanh A tanh operation outputs numbers between -1 and 1.

Walk Through

What to forget?

The first step the LSTM cell needs to do is to decide what cell state information is to be forgotten. The forget vector produced by the following formula, will have the shape of C_{t-1}. The vector will contain values between 0 and 1, these values will be used to 'forget' specific information. The forget vector is calculated below:

$$f_t = \sigma(W_f x_t + U_f h_{t-1} + b_f)$$

where W_f and U_f are learned weight vectors, b_f is the bias, h_{t-1} is the hidden state from the previous time step, x_t is the current time step's input. The weight vectors aren't really visible but are part of the gate σ.

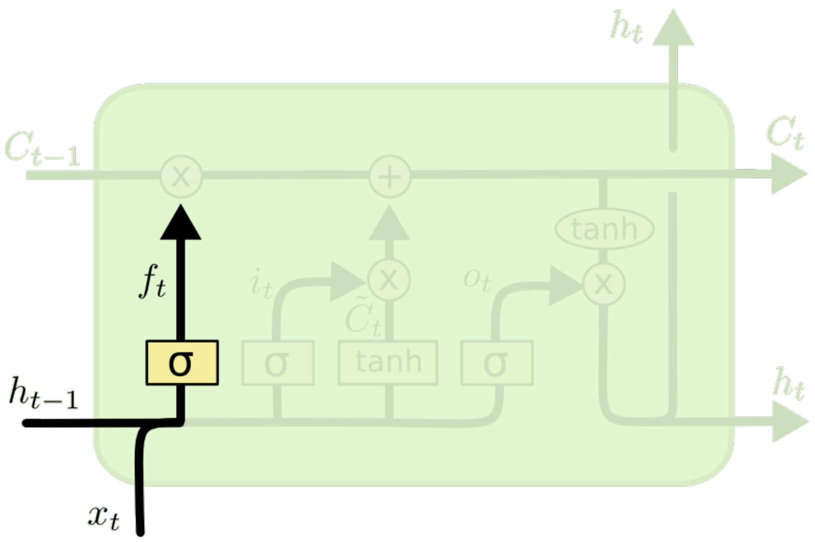


Figure 6: What to forget?

What to update?

The next step is to decide what new information is to be stored in the cell state.

Input gate layer First, a sigmoid layer called the 'input gate layer' decides which values are to be updated. It is calculated using the formula below:

$$i_t = \sigma(W_i x_t + U_i h_{t-1} + b_i)$$

Where W_i, U_i and b_i are learned, b_i is the bias. The weight vectors W_i, U_i and b_i are part of the σ gate.

New candidate layer Next, a tanh layer creates a vector of new candidate values:

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

Where W_c, U_c and b_c are learned, b_c is the bias. Again these values are part of the $tanh$ gate.

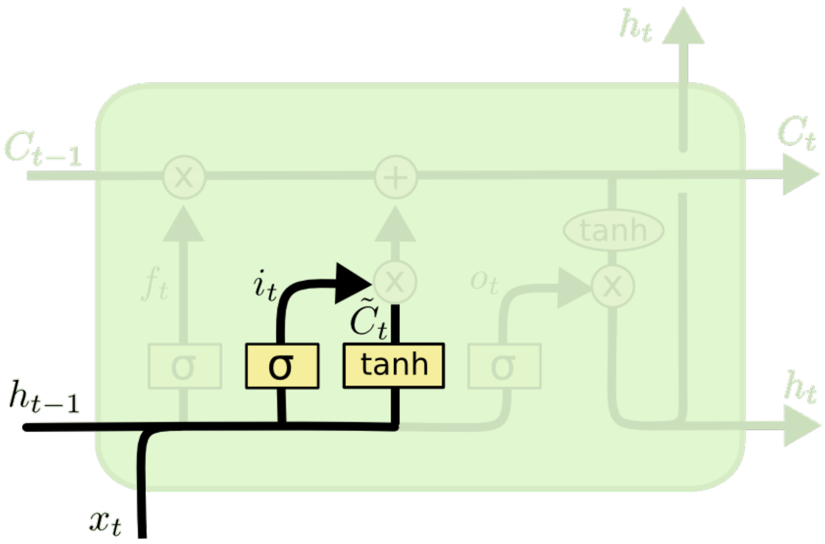


Figure 7: Creating new candidate values $\tilde{C}_t$ and a vector that decides which values are to be updated i_t

Update

It's now time to update the old cell state, C_{t-1}, into the new cell state C_t. The previous steps already decided what to forget, what to update and with what to update. The following formula will update C_t.

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

$\odot$ is a symbol for the element-wise product.

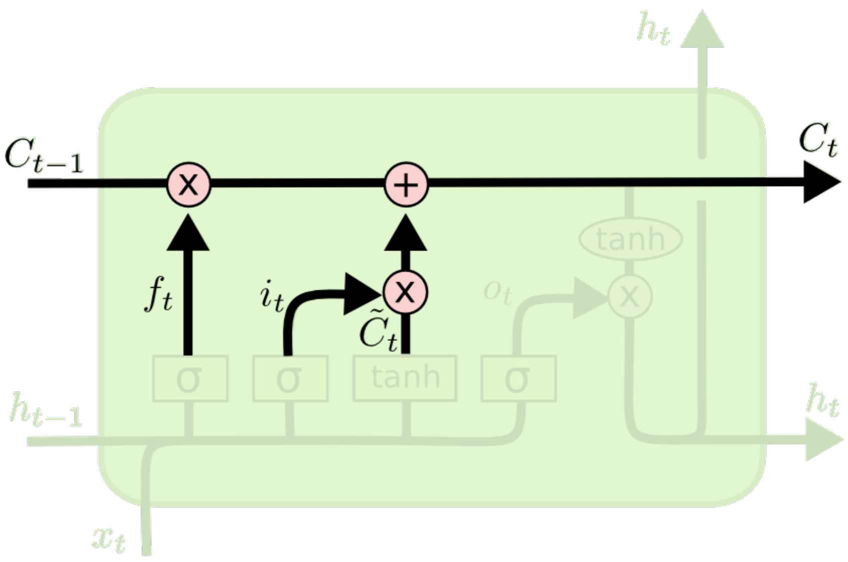


Figure 8: Updating C_t

Producing h_t

Finally, the LSTM cell needs to produce an output. This output is the result of 2 operations, the formulas can be found below:

$$o_t = \sigma(W_o x_t + U_o h_{t-1} + b_o)$$

$$h_t = o_t \odot \tanh(C_t)$$

where W_o and U_o are learned weight vectors, b_o is the bias, h_{t-1} is the hidden state from the previous time step, x_t is the current time step's input.

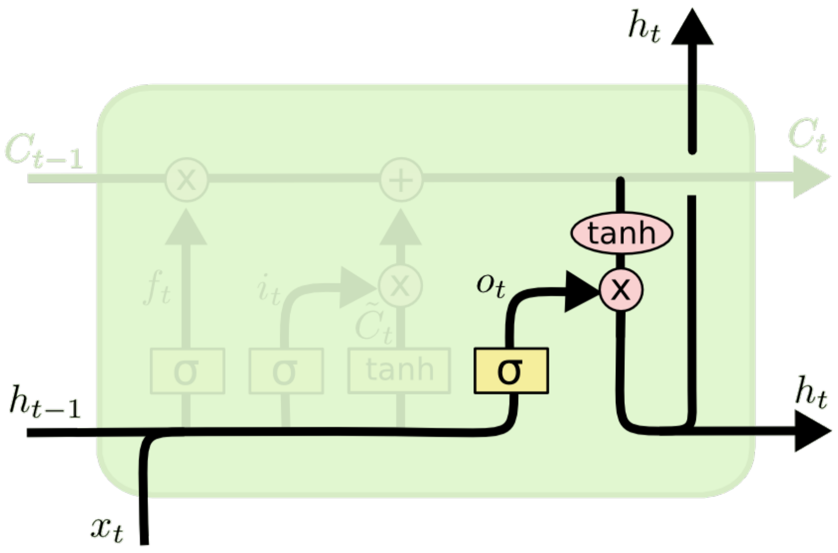


Figure 9: Producing h_t

Back-propagation

For an LSTM the vectors and values to be learned are the following.

$$\{W_f, U_f, b_f,\ W_i, U_i, b_i,\ W_c, U_c, b_c,\ W_o, U_o, b_o\}$$

When not in training mode, an LSTM reuses the same parameters at every time step. So at every time step the same learned parameters apply, what differs is the input at every time step x_t. As the input x_t differs, h_t and C_t differ too of course. When training an LSTM network, loss needs to be propagated from the last time step through every cell in the recurrent network up until the first cell. Loss is propagated through C_t and h_t.

Text generation

A basic text generation model

The LSTM part allows the model to link a source Tensor of arbitrary length to a target Tensor of the same arbitrary length. The idea is to predict a next token for the Tensor. See Figure 10 for a visualization of this idea. The model is constructed using a multilayered LSTM to allow for some text complexity. One could think of a multilayered LSTM like multiple interconnected LSTM blocks, as to be seen in Figure 11. The outputs of a previous layer serve as input to the new LSTM layer.

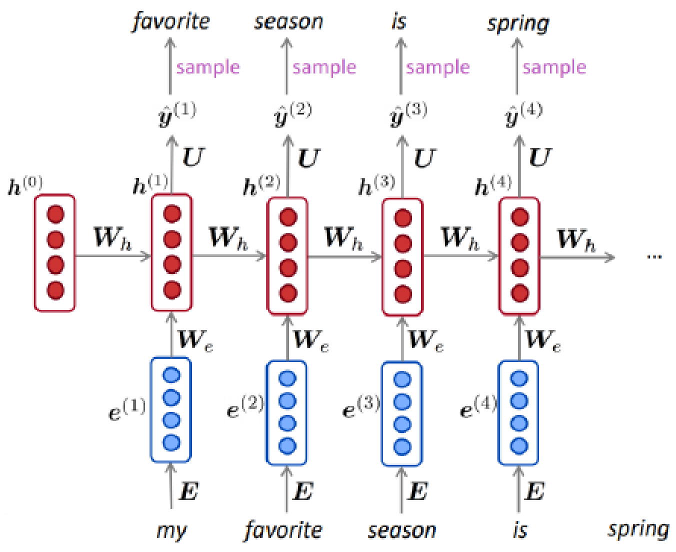


Figure 10: Visualization of text generation through an LSTM block.

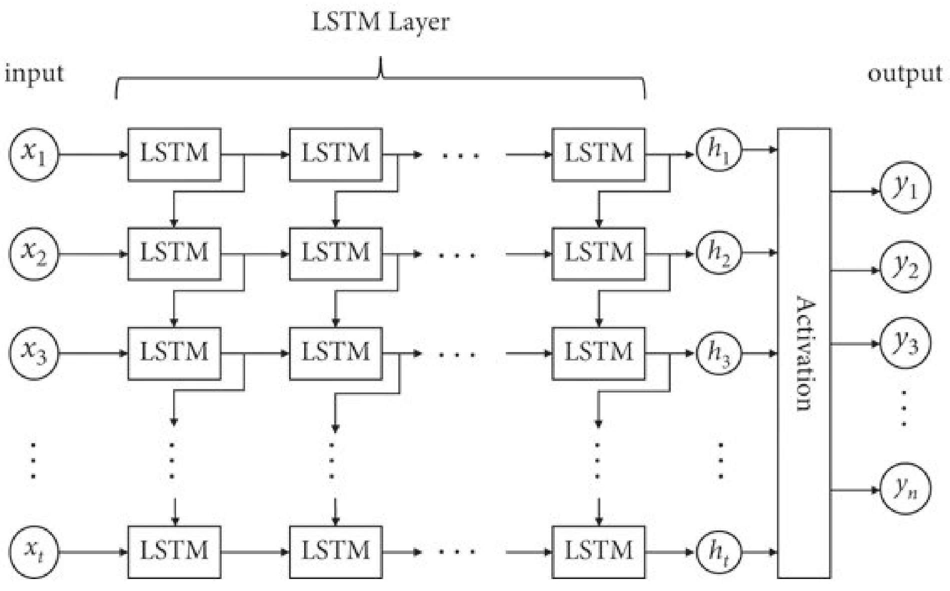


Figure 11: A multilayered LSTM block.

How does it work

The code for a basic text generation model is to be found in Listing 1. As mentioned, the main component in this model is the LSTM part (line 12). Other parts are the embedding layer (line 11), a fully connected layer (line 17) and a softmax layer (line 19).

Forward method The forward method (line 22-25) generates a complete new Tensor of Tensors, a Tensor for every embedded source token. Every Tensor produced this way, is entered into a feed forward network to produce the logits for every input token. if the length of the input is 20 tokens, output would be 20 times a logit for every token in the vocabulary.

As the model gets trained, the LSTM should get better at predicting semantically and syntactically correct words to follow a sequence of words.

Write method The write method (27-56) generates one token at a time. It enters a token index into an embedding block (line 37), this block converts the index into an embedding. The embedded values are fed into the LSTM block (line 38) to produce, some long- and short-term Tensors, and an output. This output is passed through a feedforward network (line 39), and the results of this operation are fed into a softmax block (line 44) to produce probabilities for every token in the corpus. Depending on the method, the token with the highest probability is selected or some randomness is allowed when it comes to selecting a next token. The write method ends when enough token indices have been generated. These indices are expanded into text on lines 53-54.

Torch code

Listing 1: A basic text generation model

```
1  class LSTMForWordGeneration(nn.Module):
2   def __init__(self, word2idx, idx2word,embedding_dim=128, hidden_size=256, n_layers=3):
3    super().__init__()
4    self.word2idx = word2idx
5    self.idx2word = idx2word
6    self.embedding_dim = embedding_dim
7    self.num_characters = len(word2idx)
8    self.hidden_size = hidden_size
9    self.n_layers = n_layers
10
11   self.embedding = nn.Embedding(self.num_characters, embedding_dim)
12   self.lstm = nn.LSTM(input_size=embedding_dim,
13   hidden_size=hidden_size,
14   num_layers=n_layers,
15   batch_first=True)
16
17   self.fc = nn.Linear(hidden_size, self.num_characters)
18
19   self.softmax = nn.Softmax(dim=-1)
20
21  def forward(self, x):
22   x = self.embedding(x)
23   output, (h, c) = self.lstm(x)
24   logits = self.fc(output)
25   return logits
26
27  def write(self, text, max_words, greedy=False):
28   idx = torch.tensor([self.word2idx[w] for w in text])
29
30   for i in range(max_words):
31
32    if i == 0:
33     selected_idx = idx
34    else:
35     selected_idx = idx[-1].unsqueeze(0)
36
37    x = self.embedding(selected_idx)
38    out, (hidden, cell) = self.lstm(x, (hidden, cell))
39    out = self.fc(out)
40
41    if len(out) > 1:
42     out = out[-1, :].unsqueeze(0)
```

```

   probs = self.softmax(out)

   if greedy:
    idx_next = torch.argmax(probs).squeeze(0)
   else:
    idx_next = torch.multinomial(probs, num_samples=1).squeeze(0)

   idx = torch.cat([idx, idx_next])

  gen_string = [self.idx2word[int(w)] for w in idx]
  gen_string = " ".join(gen_string)

  return gen_string
```

Results

As I will be using a corpus of length 6000000+ tokens, 28000+ unique tokens this model and write method didn't lead to interesting results. So I thought up a more advanced text generation method and implemented attention to make a better model.

Text generation using attention

In this section an attention layer will be added to the model, but before adding this layer, let's get an idea of what attention does.

Attention

Attention is the mathematical mechanism that allows a model to "focus" on specific words (tokens) in a sentence to understand their context and meaning. To get this more concrete. If the v matrix in the next multiplications would be a matrix of token vectors then the next explication would shed some light on this concept.

Attention explained

Attention can be explained by some matrix multiplications. Starting with 3 matrices q(uery),k(ey) and v(alue):

$$q = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} v = \begin{bmatrix} 2 & 4 \\ 6 & 6 \\ 3 & 5 \\ 6 & 3 \\ 2 & 4 \end{bmatrix}$$

These three matrices are supposed to be learned, at least q and k are, but as v is a result of the matrix multiplication of q and k, so is v. But first things first. Multiplying the q matrix with kT (k is transposed to fit-in this case not that essential) results in:

$$q.k^T = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

As the query is hard (not soft), and the key is hard (not soft), the resulting multiplication is just as hard. So when finally multiplying $q.k^T$ with v, this results in:

$$q.k^T.v = \begin{bmatrix} 3 & 5 \end{bmatrix}$$

The multiplications have the effect of selecting the third row in the v matrix.

In code In code these kind of multiplications would be done like in Listing 2.

Listing 2: Attention explained by some matrix multiplications

```
import torch.nn.functional as F
q = F.one_hot(torch.tensor([2]), 5)
k = F.one_hot(torch.arange(5), 5)
```

```
v = torch.randint(2, 8, (5,2 ))
q_times_k=torch.mm(q,k.T)
q_times_k_times_v=torch.mm(q_times_k, v))
```

A softer query

As q is learned it could become a less clear matrix. In this case this vague or soft q matrix has its effect on the final result of the multiplications. Again, starting with 3 matrices: a softer q(uery) matrix, k(ey) and v(alue) matrices stay the same:

$$q = \begin{bmatrix} 0. & 0.3 & 0.2 & 0.4 & 0.1 \end{bmatrix} k = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} v = \begin{bmatrix} 2 & 4 \\ 6 & 6 \\ 3 & 5 \\ 6 & 3 \\ 2 & 4 \end{bmatrix}$$

These three matrices are supposed to be learned, at least q and k are, but as v is a result of the matrix multiplication of q and k, so v is learned in a way. But first things first, the q matrix multiplied with k^T results in:

$$q.k^T = \begin{bmatrix} 0. & 0.3 & 0.2 & 0.4 & 0.1 \end{bmatrix}$$

As the query is soft, and the key is hard, the resulting multiplication is getting softer. So when finally multiplying $q.k^T$ with v, this results in:

$$q.k^T.v = \begin{bmatrix} 5. & 4.4 \end{bmatrix}$$

In this case the complete operation, and especially the softer query results in a weighted mean (weighted by the q values) for every column in the v matrix.

A softer query and softer keys

Again, starting with 3 matrices a softer q(uery) matrix and a softer k(ey) matrix, the v(alue) matrix stays the same:

$$q = \begin{bmatrix} 0. & 0.3 & 0.2 & 0.4 & 0.1 \end{bmatrix} k = \begin{bmatrix} 1. & 0. & 0. & 0. & 0. \\ 0. & 0.5 & 0. & 0. & 0. \\ 0. & 0.5 & 1. & 0. & 0. \\ 0. & 0. & 0. & 1. & 0. \\ 0. & 0. & 0. & 0. & 1. \end{bmatrix} v = \begin{bmatrix} 2 & 4 \\ 6 & 6 \\ 3 & 5 \\ 6 & 3 \\ 2 & 4 \end{bmatrix}$$

These three matrices are supposed to be learned, at least q and k are, but as v is a result of the matrix multiplication of q and k, so is v. But first things first. q matrix multiplied with k^T results in:

$$q.k^T = \begin{bmatrix} 0. & 0.15 & 0.35 & 0.4 & 0.1 \end{bmatrix}$$

And to get an idea of how k affects v, k^T multiplied with v results in:

$$k^T.v = \begin{bmatrix} 2. & 4. \\ 4.5 & 5.5 \\ 3. & 5. \\ 6. & 3. \\ 2. & 4. \end{bmatrix}$$

As the query is soft, and the key is soft, the resulting multiplication is also soft, even though it could even become softer. So when finally multiplying q.k^T with v, this results in:

$$q.k^T.v = \begin{bmatrix} 4.5499997 & 4.25 \end{bmatrix}$$

In this case the v matrix is influenced by both the soft query and key matrices. This manner of transforming value vectors simulates the way attention impacts vectors in a neural network.

Attention in code

In Listing 3, a difference regarding the matrix multiplications shown in Listing 2 can be observed. The Attention class definition contains bmm operations, bmm is the batched version of the normal matrix multiplication. There are other differences.

Torch A MultiHeadedAttention module in torch exists, but I found it easier to use a custom class. It seemed easier to configure a custom class for my specific case. This custom class was partly found on the internet, but I needed to fit it to my needs. From what I know it is how the first Tranformer had its attention calculated.

Description

Same matrices present as linear layers Applying attention allows the model to transform initial h_t vectors. This attention layer will make adaptations to parts of the vector that seem to lower loss production. It does so by adapting Q,K and V's weights, line 10-12. By allowing the Q weights to be learned, the output for an lstm cell's time step's input can be transformed. The original token embedding is used in two ways, as key and as value, and in both capacities it can and will be transformed too.

Background As learned weights for Q,K,V apply for all time steps this attention architecture should be capable of being able to predict a suitable query for the next token from its embedding and the output vector (h_t) from the first lstm layer for every token embedding at every time step. In this case the original h output is fed into the second lstm layer too, to allow it to use both original and attended vectors to influence the token to be generated/predicted.

Feeding q and x in the linear layers The parameters for the forward method are q and x. The q vector is the hidden vector output from the first lstm layer current time step's cell. The x value is an embedding vector representing a token. This vector is being fed into two other linear layers K, V (line 14-16).

Calculating raw attention and eventually normalized scores On line 17, queries and keys (returned on line 14-15) are matrix multiplied to get primary attention scores, after scaling (line 18), these are 'softmaxed' to get the attention values (line 19).

Applying normalization to the input token vector On line 20, these normalized attention weights are matrix multiplied with the linearly transformed original input token vector. By multiplying the attention weights with the original values, the original embedding is 'moved' into the direction its context seems to indicate the embedding should be moved. The attended vector is returned as the attended hidden state (line 21), and will later be used to query the second lstm layer.

Listing 3: Attention in code

```python
1  class Attention(nn.Module):
2   def __init__(self, d_in,d_out,hidden_size, n_layers, n_directions=2):
3    super().__init__()
4    self.d_in=d_in
5    self.d_out=d_out
6    self.n_layers =n_layers
7    self.n_directions=n_directions
8    self.hidden_size = hidden_size
9
10   self.Q=nn.Linear(n_directions*hidden_size,d_out)
11   self.K=nn.Linear(d_in,d_out)
12   self.V=nn.Linear(d_in,d_out)
13  def forward(self,q ,x):
14   queries=self.Q(q)
15   keys = self.K(x)
16   values = self.V(x)
17   scores = torch.bmm(queries, keys.transpose(1,2))
18   scores = scores/ (self.d_out**0.5)
19   attention = F.softmax(scores,dim=2)
20   hidden_states= torch.bmm(attention,values)
21   return hidden_states
```

A text generation model using attention

In this part I will use the attention layer in a text generation model. The two write methods will be separately covered.

Differences compared to the simpler model When compared to the simple model, this model has 2 LSTMs instead of 1. This to facilitate the use of attention. In this model, an attention layer is added (line 19) and a second LSTM layer is added (line 21). Other components don't really differ from the model without attention, there is an extra LogSoftmax layer (line 29) for the write_better method. The forward method is more complex when compared to the simpler model, as I am applying the LSTM network on a per token/per time step basis.

The forward method

The forward method will apply attention for every time step.

Side note The way in which I apply attention and use hidden and cell values, is sort of experimental. I commented out the experimentation as it might not be such a good idea after all.

Manually applying attention for every time step This because I wanted to apply the attention manually for every time step and consequently for every token embedding, this happens on line 45. The loop to go over every time step starts on line 33.

Preparing first LSTM layer outputs for second LSTM layer input Other steps are feeding the 'current time step token embedding vector' (line 38,40) into the first LSTM layer. Reshaping the hidden and cell LSTM outputs (line 42,43) to prepare them for input in the second LSTM layer (line 47).

Adapting the current second LSTM's output for next time step first LSTM's input While I was adapting values at a time step level, I wanted the second's layer's LSTM output to influence the first layers hidden and cell vectors (line 49-50) when generating outputs in a next time step. As these adapted vectors are being used in the next time step to generate new first LSTM layer's time step outputs, they need to be reshaped (line 52-53).

Producing logits and tokens Eventually the per time step output produced by the second LSTM layer is being concatenated with previous previous time step outputs, to form the complete output. For every output generated by the second LSTM layer, logits are generated (line 58). These logits represent probabilities for every token in the dictionary. As for every source sentence there is a target sentence, loss can be calculated by comparing the expected tokens for a source sentence, with the logit values that were returned for every token in the dictionary per time step.

Listing 4: A basic text generation model with attention

```
1  class LSTMForWordGenerationWithAttention(nn.Module):
2   def __init__(self, word2idx, idx2word, embedding_dim=128, hidden_size=256, bidirectional=
      ↪ False, n_layers=3):
3    super().__init__()
4    self.word2idx = word2idx
5    self.idx2word = idx2word
6    self.embedding_dim = embedding_dim
7    self.num_words = len(word2idx)
8    self.hidden_size = hidden_size
9    self.n_layers = n_layers
10   self.num_directions = 2 if bidirectional else 1
11
12   self.embedding = nn.Embedding(self.num_words, embedding_dim)
13   self.lstm = nn.LSTM(input_size=embedding_dim,
14   hidden_size=hidden_size,
15   num_layers=n_layers,
16   batch_first=True,
17   bidirectional=bidirectional)
18
19   self.attention = Attention(self.embedding_dim, self.n_layers * self.num_directions * self.
      ↪ embedding_dim, self.n_layers, self.num_directions)
20
21   self.lstm2 = nn.LSTM(input_size=self.n_layers * self.num_directions * self.embedding_dim,
22   hidden_size=self.n_layers * self.num_directions * self.hidden_size,
23   num_layers=1,
24   batch_first=True,
25   bidirectional=False)
```

```

 self.fc = nn.Linear(self.n_layers * self.num_directions * self.hidden_size, self.num_words)
 self.softmax = nn.Softmax(dim=-1)
 self.log_softmax= nn.LogSoftmax(dim=-1)
def forward(self, x):
 batch_size = x.size(0)
 output_enc_dec = FloatTensor()
 for i in range(x.shape[1]):
  x_i = self.embedding(x[:, i])
  x_i = x_i.unsqueeze(1)

  if i == 0:
   out, (hidden, cell) = self.lstm(x_i)
  else:
   out, (hidden, cell) = self.lstm(x_i, (hidden, cell))

  hidden = hidden.reshape(1, -1, self.n_layers * self.num_directions * self.hidden_size)
  cell = cell.reshape(1, -1, self.n_layers * self.num_directions * self.hidden_size)

  attention_output = self.attention(out, x_i)

  out_lstm2, (h, c) = self.lstm2(attention_output, (hidden, cell))

  #hidden = torch.div(torch.add(hidden, h), 2, rounding_mode=None)
  #cell = torch.div(torch.add(cell, c), 2, rounding_mode=None)

  hidden = hidden.reshape(self.n_layers * self.num_directions, batch_size, -1)
  cell = cell.reshape(self.n_layers * self.num_directions, batch_size, -1)

  output_enc_dec = torch.cat([output_enc_dec, out_lstm2.unsqueeze(1)], dim=1)

 output_enc_dec = output_enc_dec.squeeze(2)
 logits = self.fc(output_enc_dec)
 return logits
```

Write

The original write method, is adapted to deal with attention. It is to be seen in Listing 5.

Apply attention to output of first lstm On line 22 the output from the first lstm layer is transformed by the attention layer. This output is then used as input for the second lstm layer. This is the main difference from the simple text generation model.

Detach, clone, reshape and rename hidden and cell outputs The hidden and cell outputs from the first lstm layer are detached and cloned, renamed,reshaped and detached (line 19-20). Detached implies, released from the back-propagation cycle. In this way they can serve as inputs for the second LSTM layer (line 23), while the original hidden and cell tensors (still in the back-propagation cycle) remain available as input for the first lstm layer in the next time step.

Second lstm produces its output for the fully connected layer As all parameters for the second LSTM are fed into it (line 23), the output of this second LSTM is input for the fully connected layer (line 25).

Producing a distribution over all tokens in the dictionary After some dimensions adaptations (line 26 and possibly 29) the output becomes suitable for becoming a distribution over every token in the dictionary (line 31).

Picking the next word by chance or the most probable Now depending on the chosen method (line 33-36), the most probable word or a word is randomly chosen using it's possibility as a chance for it to be chosen. The index for the word is concatenated to the previously chosen index (line 38).

Translating the indices into a sentence When the for loop described here is finished. The only thing needed now is the translation from numbers to words using the dictionary (line 40).

Listing 5: Write with attention

```python
1  def write(self, text, max_words, greedy=False):
2   idx = torch.tensor([self.word2idx[w] for w in text])
3   hidden = Tensor()
4   cell = Tensor()
5   for i in range(max_words):
6    if i == 0:
7     selected_idx = idx
8    else:
9     selected_idx = idx[-1].unsqueeze(0)
10
11   x = self.embedding(selected_idx)
12   x = x.unsqueeze(0)
13
14   if i == 0:
15    out, (hidden, cell) = self.lstm(x)
16   else:
17    out, (hidden, cell) = self.lstm(x, (hidden, cell))
18
19   hidden1 = hidden.detach().clone().reshape(1, 1, -1)
20   cell1 = cell.detach().clone().reshape(1, 1, -1)
21
22   attention_output = self.attention(out, x)
23   out_lstm2, (h, c) = self.lstm2(attention_output, (hidden1, cell1))
24
25   out = self.fc(out_lstm2)
26   out = out.squeeze(0)
27
28   if len(out) > 1:
29    out = out[-1, :].unsqueeze(0)
30
31   probs = self.softmax(out)
32
33   if greedy:
34    idx_next = torch.argmax(probs).squeeze(0)
35   else:
36    idx_next = torch.multinomial(probs, num_samples=1).squeeze(0)
37
38   idx = torch.cat([idx, idx_next])
39
40  gen_string = [self.idx2word[int(w)] for w in idx]
41  gen_string = " ".join(gen_string)
42  return gen_string
```

Write better

In this part the write better method will be elaborated on. The write better method is an integral part of the model. As the write_better method is based on the Beam search algorithm, the Beam search algorithm is illustrated using a simple example.

Beam search:simplified example To get an idea of how beam search works, see Figure 12, the beam search illustrated here represents a beam search with width 2. At the first time step, A and B would be retained, as these have the least negative log probabilities. At the second time step, as only A and B have been kept, only children of these nodes need to be considered. A B and B A have the least negative cumulative probability so these paths are retained. Only nodes that are descendants of A B and B A are considered next. Of those descendants A B B and B A C prove to be the most interesting hypotheses. As all other paths have a higher negative cumulative log-probability. As not all paths are being considered, beam search is no guarantee to finding the most optimal path overall, but at least it gives a chance to paths that initially look less fruitful. And in this example an initially less fruitful alternative (B) eventually proves to be just as fruitful as the initially most interesting node to investigate (A).

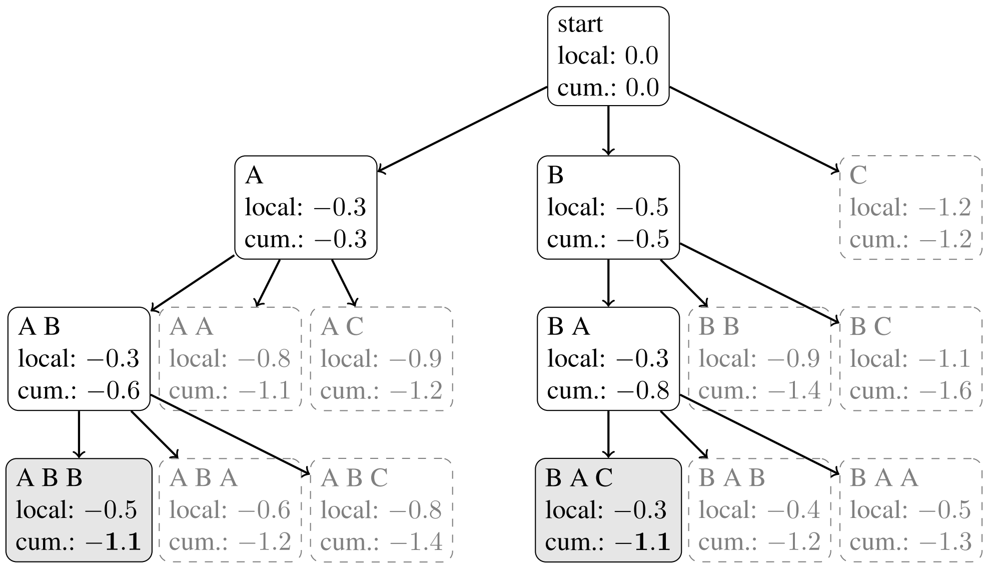


Figure 12: Beam search tree with beam width of 2. Each node displays the local log-probability and the cumulative log-probability. Pruned hypotheses are dashed, the 2 best sequences are shaded.

Code

In the text generation model at hand, the write_better method (Listing 6) needs to consider thousands of tokens every time step, for every word generated. It generates a next word from previously generated/selected words. The code keeps track of the cumulative path probabilities and the paths (tensors of token indices). To describe the process of this method I will describe almost every step of the code and share the shape of the data, and sometimes the content produced.

Some parameters The example beam width used here will be 3, the embedding dimension will be 128, and the first LSTM's hidden dimension will be 256, the first LSTM will be bidirectional, the number of layers in the first LSTM will be 2, the number of items in the token dictionary is 6916.

Initializing As beam width is 3, and our algorithm wants to return 3 fruitful paths. The algorithm needs to create a tensor containing 3 times the first index from which to generate the next time steps. For the first time step, as the initial input is ”Spells”, the initial tensor created on line 3 looks like this.

$$\begin{bmatrix}[6080] & [6080] & [6080]\end{bmatrix}$$

6080 is the index in the dictionary for the token ”Spells”.

The initial tensor to hold the cumulative probabilities for each path looks like

$$\begin{bmatrix}[0.] & [0.] & [0.]\end{bmatrix}$$

The variables set to hold the indices and the probabilities are set on line 3 and 6.

Some background The matrix holding the initial word indices will be fed into the LSTM like it was a sort of batched input. At this moment the indices are the same, so this seems a bit useless. But after generating a new word for each path, three different word indices will be fed into the LSTM, and those 3 indices will eventually produce 3 different distributions for the next word to be predicted. The LSTM will also produce different hidden and cell outputs for every word predicted, per time step. This is the main complicating factor when applying the beam search algorithm in this situation.

Embedding For the three indices an embedding is created (line 15). The embedding for this complete tensor has a shape:

$$\begin{bmatrix}3 & 1 & 128\end{bmatrix}$$

128 representing the embedding size, 3 the number of indices/beams for which the embedding is produced.

Feeding the embedding into the first LSTM On line 16-19, the embedding is fed into the first LSTM. A condition is used here. This demands some explication, by default an LSTM will create hidden and cell vectors if none are given. The values are all zeros for these default vectors. In the first time step this is a good thing, but as of the second time step the hidden and cell values will depend on the chosen beam paths. The shape of the out, hidden and cell tensors are:

$$\text{out.shape} = \begin{bmatrix}3 & 1 & 512\end{bmatrix}$$

3 from the number of beams
512 is the hidden dimension (256) times the number of directions (2) Out contains a concatenation of the forward and reverse hidden states

$$\text{hidden.shape} = \begin{bmatrix}4 & 3 & 256\end{bmatrix}$$

4, as the number of layers for the LSTM is 2 and the number of directions is 2, and 2 times 2 is 4
3, as the number of beams is 3
256 as the hidden dimension for the LSTM is 256

$$\text{cell.shape} = \begin{bmatrix}4 & 3 & 256\end{bmatrix}$$

4, as the number of layers for the LSTM is 2 and the number of directions is 2
3, as the number of beams is 3
256 as the hidden dimension for the LSTM is 256

Reshaping hidden and cell vectors On line 21 and 22 the hidden and cell vectors are detached, cloned and reshaped. After reshaping, both cell1 and hidden1 have the same shape:

$$\text{cell1.shape} = \text{hidden1.shape} = \begin{bmatrix}1 & 3 & 1024\end{bmatrix}$$

3 is the number of beams
1024 is 4*256, the same data is available as before, only dimensions have changed

The second LSTM needs the vector to be in this shape. But in the next time step, the original hidden and cell shapes are needed, that is why they are cloned before reshaping. The reshaped vectors are detached because text is being generated, the model is not being trained.

Applying attention to the output of the first LSTM On line 24 attention is applied, after having applied attention shape of the weighted output is

$$\text{weighted\_output.shape} = \begin{bmatrix}3 & 1 & 512\end{bmatrix}$$

From the previous part on attention, the weighted output may be seen as the contextualized primary input for the second LSTM. It is the query for which a next token in the sentence should be predicted. The hidden1 and cell1 vectors are unattended to, but could/should contain additional information from a previous time step.

Feed into the second LSTM The cell1, hidden1 and weighted_output are fed into the second LSTM (line 25). After this shapes of the vectors looks like this:

$$\text{lstm2\_out.shape} = \begin{bmatrix}3 & 1 & 1024\end{bmatrix}$$

3, is the number of beams
1024, is the number of layers (2) times the number of directions (2) times the hidden size (256)

$$\text{h.shape} = \text{c.shape} = \begin{bmatrix}1 & 3 & 1024\end{bmatrix}$$

3, is the number of beams
1024, is the number of layers (2) times the number of directions (2) times the hidden size (256)
After application of line 26 all three vectors h, c and lstm2_out have the same shapes.

Feed lstm2_out into the fully connected layer and applying softmax The output of the second LSTM is fed into a fully connected layer (line 27). The resulting vector's shape is:

$$\begin{bmatrix}1 & 3 & 6916\end{bmatrix}$$

3, for the beam width
6916, the number of tokens present in the dictionary

After feeding the fully connected return into the softmax layer (line 32), the shape remains the same. It is from this moment on, that a distribution for every next word in the dictionary exists for every different path, these distributions depend on the previous inputs (hidden, cell, token index embedding) for every beam respectively.

Add previous path probabilities to newly create distribution To compare path probabilities, the newly created distributions for the tokens needs to be increased by the already existing path probabilities (line 33). By using the right dimensions, the existing path probability for each path only gets broadcast added to the right distribution. As the softmax layer we're using is a LogSoftmax layer, probabilities can be added.

Beam search

It is from this moment on that the most fruitful paths for generating a new word in the sequence should be chosen. As it is possible an unfruitful path can be canceled, it is important to know for which paths to keep the sequence running and for which not.

Selecting the most probable indices and their probabilities Some background, as the distributions (adapted by the existing path probabilities) that exist for every path in our beam search can and will differ, and the 3 most fruitful paths need to be chosen, every probability for the 3 beams need to be compared, that's why the probabilities for the 3 beams need to be flattened, to compare them over distribution boundaries.

The flattened probabilities tensor has a shape of

$$\begin{bmatrix}20748\end{bmatrix}$$

20748, 6916 times the number of beams 3

This comparison happens on line 35. The resulting top probabilities could look like this:

$$\begin{bmatrix}0.0002 & 0.0002 & 0.0002\end{bmatrix}$$

The resulting top indices could look like this, in this case every index lies on another path:

$$\begin{bmatrix}11975 & 14804 & 5194\end{bmatrix}$$

Line 37 is the code to get the global indices translated to their original distributions. The result of this line looks like this:

$$\text{indexes} = \begin{bmatrix}[0 & 0 & 5194] \\ [0 & 1 & 5059] \\ [0 & 2 & 972]\end{bmatrix}$$

11975, refers to token index 5059 in the second distribution
14804, refers to token index 972 in the third distribution
5194, refers to token index 5194 in the first distribution

After merging path probabilities into this matrix (line 39) and sorting on the second column (41), the matrix looks would look like this:

$$\text{indexes} = \begin{bmatrix}[0 & 0 & 5194 & 0.0002] \\ [0 & 1 & 5059 & 0.0002] \\ [0 & 2 & 972 & 0.0002]\end{bmatrix}$$

Sorted As not every new token comes from the same distribution, and there is no guarantee line 37 returns indices sorted by distribution, the sorting still needs to be done. This to be sure the right token indices align to the right existing path token indices.

Floats Unfortunately the probabilities merged into the matrix are floats, so every value in the matrix is converted to floats too. To convert them back to ints the selected column tensors are typed to contain ints by applying $.type(dtype =' torch.IntTensor')$ to them, this when columns are selected and reused elsewhere.

Update paths before adding new token indices As the distributions from which the next tokens are chosen are known, the already existing token index paths are updated, in this case nothing would change as tokens from every distribution are selected. But it could be possible that a path is canceled. To accommodate for this case, the already existing token index paths needed are selected. The index_select method selects those token index paths that are kept, when only two distributions are kept, one out of two is duplicated.

Adding new token indices The indices of the 3 most probable new path tokens can be found on the third column of the indexes matrix. So this column is selected on line 42. These new token indices are concatenated to the already updated path token indices, line 45.

Update cumulative path probabilities To keep track of the newly calculated cumulative path probabilities, path probabilities are taken from the indexes_ext matrix and kept in a variable (line 47).

Adapting hidden and cell vectors As paths are chosen others are left closed, hidden and cell values should disappear as their distribution contains no relevant probability and token index, this is done on lines 48-49. Remember the shape of hidden and cell

$$\text{hidden.shape} = \text{cell.shape} = \begin{bmatrix} 4 & 3 & 256 \end{bmatrix}$$

It is only for relevant distributions hidden and cell values are being kept. In case the distribution indices were $[1, 2, 2]$, hidden matrix for every beam is adapted by moving the relevant parts of the matrix around. So in the next time step the hidden and cell values are relevant and are set correct for the respective paths.

Generate text After the loop has generated the correct number of words for all beams, the different sequences of indices are expanded into real texts, line 51-54. From the log probabilities probabilities are calculated.

Listing 6: Write better with attention

```
1  def write_better(self, text, max_words, k=1):
2   idx = torch.tensor([self.word2idx[w] for w in text])
3   k_times_idx = idx.repeat(1, k).unsqueeze(2).squeeze(0)
4   hidden = Tensor()
5   cell = Tensor()
6   k_times_probs = Tensor([0]).repeat(k).unsqueeze(1)
7  
8   for i in range(max_words):
9  
10   if i == 0:
11    selected_idx = k_times_idx
12   else:
13    selected_idx = k_times_idx[:, -1].unsqueeze(1)
14  
15   x = self.embedding(selected_idx)
16   if i == 0:
17    out, (hidden, cell) = self.lstm(x)
18   else:
19    out, (hidden, cell) = self.lstm(x, (hidden, cell))
20  
21   hidden1 = hidden.detach().clone().reshape(1, k, -1)
22   cell1 = cell.detach().clone().reshape(1, k, -1)
```

```
  # apply attention
  weighted_output = self.attention(out, x)
  out_lstm2, (h, c) = self.lstm2(weighted_output, (hidden1, cell1))
  out_lstm2 = out_lstm2.permute(1, 0, 2)
  out = self.fc(out_lstm2)

  if len(out) > 1:
   out = out[-1, :].unsqueeze(0)

  probs = self.log_softmax(out)
  probs.add_(k_times_probs)

  probs_top, i = torch.topk(probs.detach().flatten(), k=k, dim=-1, sorted=False)
  i = i.detach()
  indexes = np.array(np.unravel_index(i.numpy(), probs.shape)).T
  # extend indexes with probs
  indexes_ext = np.concatenate((indexes, probs_top.unsqueeze(1)), axis=1)
  # sort extended indexes
  sorted_indexes = indexes_ext[np.argsort(indexes[:, 1])]
  indices = torch.tensor(sorted_indexes[:, 1]).type(dtype='torch.IntTensor')
  idx_next = torch.from_numpy(sorted_indexes[:, 2]).type(dtype='torch.IntTensor').unsqueeze
    ↪ (1)
  k_times_idx=torch.index_select(k_times_idx,0,indices)
  k_times_idx = torch.cat([k_times_idx, idx_next], dim=1)
  #update path probs
  k_times_probs =  torch.from_numpy(sorted_indexes[:, 3]).unsqueeze(1)
  hidden = torch.index_select(hidden, 1, indices)
  cell = torch.index_select(cell, 1, indices)

 gen_strings = []
 for i, idxs in enumerate(k_times_idx.numpy()):
  gen_string = [self.idx2word[int(w)] for w in idxs]
  gen_strings.append(" ".join(gen_string))
 probs=torch.exp(k_times_probs.squeeze(1))
 return gen_strings, probs
```

Text generation using multiheaded attention

In the previous section attention was discussed and used in a model. In this section multi-headed attention added to the model.

Multi-headed attention in code

Basically plain attention allows one channel of attention, and multi-headed attention allows for more channels. The code in Listing 7 shows a custom multi-headed attention layer. Parameter d_in and d_out are the in and output dimensions for the module and hence the complete multi-headed attention. As the attention layer is placed between two lstms d_in depends on the outputs of the first lstm layer and d_out depends on the second lstm layer's input dimensions.

ModuleList The attention_heads are created inside a ModuleList (line 5). This torch construct allows for the same handling of the layers inside this list as for simple nn.Modules.

Listing 7: Multiheaded attention

```
1 class MultiHeadAttentionWrapper(nn.Module):
2  def __init__(self, d_in,d_out, n_layers, n_directions ,num_heads):
3   super().__init__()
4   self.heads = nn.ModuleList(
5   [Attention(d_in,d_out, n_layers, n_directions) for _ in range(num_heads)]
6   )
7
8  def forward(self,q, x):
9   return torch.cat([head(q,x) for head in self.heads], dim=-1)
```

A text generation model using multi-headed attention

The __init__ method and the forward method don't differ that much from the simpler method with attention. That is why I will not elaborate on them. For the model's code see Listing 8.

Listing 8: A basic text generation model with multiheaded attention

```
class LSTMForWordGenerationWithMHAttention(nn.Module):
 def __init__(self, word2idx, idx2word, embedding_dim=128, hidden_size=256, bidirectional=
    ↪ False, n_layers=3, n_heads=2):
  super().__init__()
  self.word2idx = word2idx
  self.idx2word = idx2word
  self.embedding_dim = embedding_dim
  self.num_words = len(word2idx)
  self.hidden_size = hidden_size
  self.n_layers = n_layers
  self.n_heads = n_heads
  self.num_directions = 2 if bidirectional else 1

  self.embedding = nn.Embedding(self.num_words, embedding_dim)
  self.lstm = nn.LSTM(input_size=embedding_dim,
  hidden_size=hidden_size,
  num_layers=n_layers,
  batch_first=True,
  bidirectional=bidirectional)

  self.attention = MultiHeadAttentionWrapper(self.embedding_dim, self.n_layers * self.
    ↪ num_directions * self.embedding_dim, self.n_layers, self.num_directions,self.n_heads)

  self.lstm2 = nn.LSTM(input_size=self.n_heads*self.n_layers * self.num_directions * self.
    ↪ embedding_dim,
  hidden_size=self.n_layers * self.num_directions * self.hidden_size,
  num_layers=1,
  batch_first=True,
  bidirectional=False)

  self.fc = nn.Linear(self.n_layers * self.num_directions * self.hidden_size, self.num_words)
  self.softmax = nn.Softmax(dim=-1)
  self.log_softmax= nn.LogSoftmax(dim=-1)
 def forward(self, x):
  batch_size = x.size(0)
  output_enc_dec = FloatTensor()
  for i in range(x.shape[1]):
   x_i = self.embedding(x[:, i])
   x_i = x_i.unsqueeze(1)

   if i == 0:
    out, (hidden, cell) = self.lstm(x_i)
   else:
    out, (hidden, cell) = self.lstm(x_i, (hidden, cell))

   hidden = hidden.reshape(1, -1, self.n_layers * self.num_directions * self.hidden_size)
   cell = cell.reshape(1, -1, self.n_layers * self.num_directions * self.hidden_size)


   attention_output = self.attention(out, x_i)

   out_lstm2, (h, c) = self.lstm2(attention_output, (hidden, cell))

   hidden = torch.div(torch.add(hidden, h), 2, rounding_mode=None)
   cell = torch.div(torch.add(cell, c), 2, rounding_mode=None)

   hidden = hidden.reshape(self.n_layers * self.num_directions, batch_size, -1)
   cell = cell.reshape(self.n_layers * self.num_directions, batch_size, -1)

   output_enc_dec = torch.cat([output_enc_dec, out_lstm2.unsqueeze(1)], dim=1)

  output_enc_dec = output_enc_dec.squeeze(2)
  logits = self.fc(output_enc_dec)
  return logits
```

Write_better and write

These methods don't differ that much from the methods in the model with attention, so I will not add the code for them.

Training

The training loop

The training loop in Listing 9 looks a bit complicated, but it isn't that complicated.

Data written to the console, to disk and saving the model

There are a multiple things exported, loss, generated text, text probabilities and loss (lines 2-18, 59-69, 74-79, 81-83). Also some data is written to the console (line 47-58). Every evaluation_interval the trained model's parameters are saved (line 71).

Furthermore normal training elements

Furthermore basic training elements like optimizer, loss (line 26, 27). The dataset is created (line 28). Batch is fetched (line 32-33), zero_grad() is applied to the optimizer (line 34), loss is calculated (line 38), back-propagated over parameters (line 40) and parameters are updated (line 41).

Listing 9: Train loop

```
1  def train_(model, word2idx, idx2word, tokens, config, write_better=False):
2   training_data = {
3    "model":[],
4    "iteration":[],
5    "training data length":[],
6    "loss": [],
7    "generated text": []
8   }
9   texts_generated = {
10   "iteration": [],
11   "loss":[],
12   "probabilities":[],
13   "texts": [],
14  }
15
16  name=model.__class__.__name__
17  new_dir =Path(os.path.join(TRAINED_MODELS_DIR, name))
18  new_dir.mkdir(parents=True, exist_ok=True)
19
20  iterations = config["iterations"]
21  max_len = config["max_len"]
22  evaluate_interval = config["evaluate_interval"]
23  lr = config["lr"]
24  batch_size = config["batch_size"]
25
26  optimizer = optim.AdamW(model.parameters(), lr=lr)
27  loss_fn = nn.CrossEntropyLoss()
28  np.random.seed(0)
29  dataset = code.data.data_memmap.get_databuildermemmap(max_len, tokens, word2idx, idx2word)
30  model.train()
31  for iteration in range(iterations):
32   input_texts, labels = dataset.grab_random_batch(batch_size=batch_size)
33   input_texts, labels = input_texts, labels
34   optimizer.zero_grad()
35   output = model(input_texts)
36   output = output.transpose(1,2)
```

```

 loss = loss_fn(output, labels.long())

 loss.backward()
 optimizer.step()


 if iteration % evaluate_interval == 0:
  model.eval()
  torch.no_grad()
  print("--------------------------------------")
  print(f"training data length {max_len}")
  print(f"Iteration {iteration}")
  print(f"Loss {loss.item()}")
  generated_text = model.write(["Spells"], max_words=50)
  print(generated_text)
  if write_better:
  generated_texts, probabilities = model.write_better(["Spells"],max_words=50, k=3)
  print("Sample Generation")
  for text, probability in zip(generated_texts, probabilities):
  print(f"text: {text} probability: {probability}")
  print("--------------------------------------")
  training_data["model"].append(name)
  training_data["iteration"].append(iteration)
  training_data["training data length"].append(max_len)
  training_data["loss"].append(loss.item())
  training_data["generated text"].append(generated_text)

  if write_better:
   texts_generated["iteration"].append(iteration)
   texts_generated["loss"].append(loss.item())
   texts_generated["probabilities"].append(probabilities)
   texts_generated["texts"].append(generated_texts)

  torch.save(model.state_dict(), os.path.join(TRAINED_MODELS_DIR, name + "/model_state-v"+ str(iteration)+".pth"))
  torch.enable_grad()
  model.train()
td=pd.DataFrame(training_data)
td.set_index(["model", "iteration"])
td.to_csv(os.path.join(TRAINED_MODELS_DIR,name,"training_data.csv"), encoding="utf8")
cd=pd.DataFrame(list(config.items()), columns=["key", "value"])

cd.to_csv(os.path.join(TRAINED_MODELS_DIR,name,"config_data.csv"), encoding="utf8", index=False)

if write_better:
 with open(os.path.join(TRAINED_MODELS_DIR,name,'generated_texts.pkl'), 'wb') as fh:
  pickle.dump(texts_generated, fh, protocol=pickle.HIGHEST_PROTOCOL)
```

Loss and generated texts

In the accompanying notebook 3 models have been created, trained and compared. In the following I will compare some results produced by the models while training. As I limited the number of training examples, limited the number of tokens (by limiting the corpus to one or two books book) and limited the hidden and embedding dimensions for the models themselves. In this way I kept total training time limited to hours. Hence the text generated by these models is equally limited.

Configurations

In the elaborations on the different models I didn't mention the parameters that make up the different models. Most values are configurable, meaning it is possible to change a model with a small embedding_dim to become a model with a bigger embedding_dim without having to rewrite code in the model. The hidden_dim being the dimension used in the first lstm layer. The first and sometimes only lstm layer contains 2 layers for all models. Learning rate or lr is 0.003. The batch size is the same for each model. Both attention models have a first lstm layer that is bidirectional. A bidirectional processes sequences in two directions, in this way data from later in a sequence influences data that is available at a previous time step. The n_heads sets the number of attention heads for the multi-headed attention model.

Simple model keys	value
iterations	10000
max_len	20
evaluate_interval	1000
embedding_dim	128
hidden_size	256
n_layers	2
lr	0.003000
batch_size	64
bidirectional	False

(a) Simple model

Attention model keys	value
iterations	10000
max_len	20
evaluate_interval	1000
embedding_dim	128
hidden_size	256
n_layers	2
lr	0.003000
batch_size	64
bidirectional	True

(b) Attention model

MH Att. model keys	value
iterations	10000
max_len	20
evaluate_interval	1000
embedding_dim	128
hidden_size	256
n_layers	2
lr	0.003000
batch_size	64
bidirectional	True
n_heads	2

(c) MH Attention model

Figure 13: Configurations used for the training of the different models

Loss

The different models are trainable as can be seen from Figure 14.

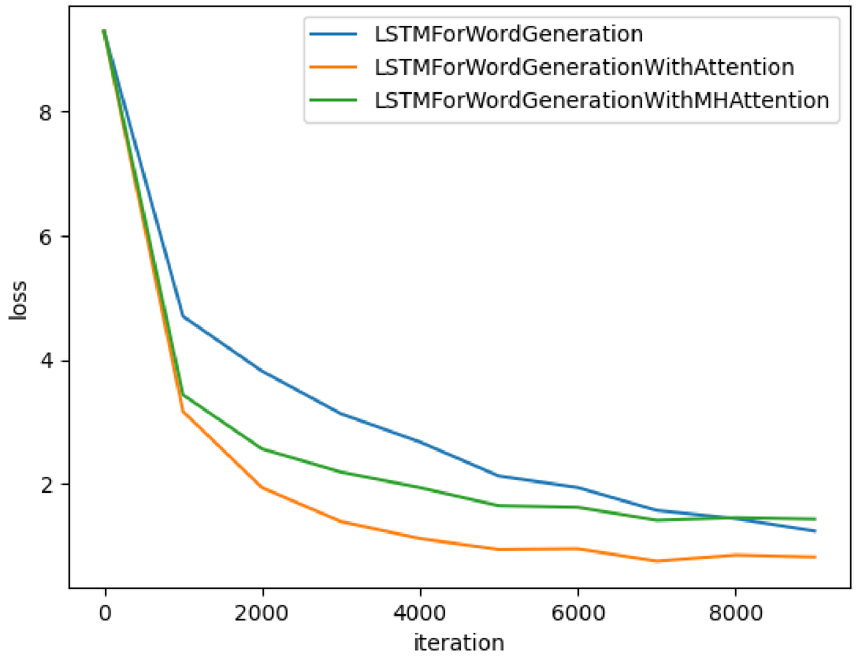
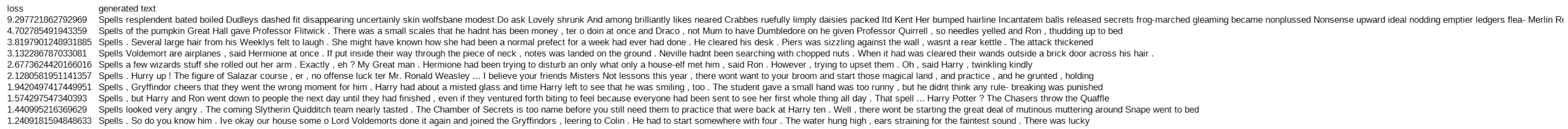


Figure 14: Visualization of training loss.

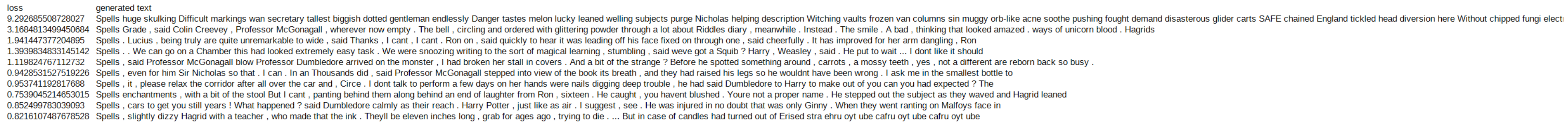
Generated texts using Write method

After each training loop text is generated. I've always used the same input word, "Spells". For every model, configured using the values in Figure 13, the quality of the text generation is poor. I must add, only the write method is considered here.



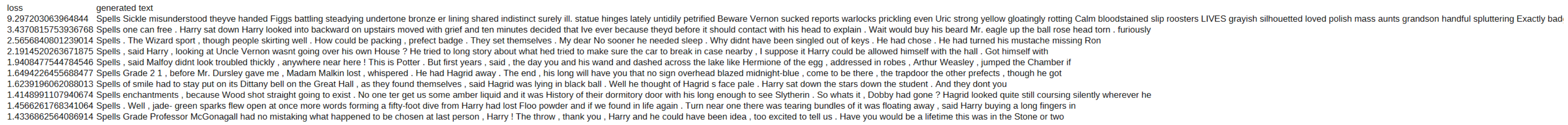
loss	generated text
9.297721862792969	Spells resplendent bated boiled Dudleys dashed fit disappearing uncertainly skin wolfsbane modest Do ask Lovely shrunk And among brilliantly likes neared Crabbes ruefully limply daisies packed Itd Kent Her bumped hairline Incantatem balls released secrets frog-marched gleaming became nonplussed Nonsense upward ideal nodding emptier ledgers flea- Merlin R
4.702785491943359	Spells of the pumpkin Great Hall gave Professor Flitwick . There was a small scales that he hadnt has been money , ter o doin at once and Draco , not Mum to have Dumbledore on he given Professor Quirrell , so needles yelled and Ron , thudding up to bed
3.8197901248931885	Spells . Several large hair from his Weeklys felt to laugh . She might have known how she had been a normal prefect for a week had ever had done . He cleared his desk . Piers was sizzling against the wall , wasnt a rear kettle . The attack thickened
3.132286787033081	Spells Voldemort are airplanes , said Hermione at once . If put inside their way through the piece of neck , notes was landed on the ground . Neville hadnt been searching with chopped nuts . When it had was cleared their wands outside a brick door across his hair .
2.6773624420166016	Spells a few wizards stuff she rolled out her arm . Exactly , eh ? My Great man . Hermione had been trying to disturb an only what only a house-elf met him , said Ron . However , trying to upset them . Oh , said Harry , twinkling kindly
2.1280581951141357	Spells . Hurry up ! The figure of Salazar course , er , no offense luck ter Mr. Ronald Weasley ... I believe your friends Misters Not lessons this year , there wont want to your broom and start those magical land , and practice , and he grunted , holding
1.9420497417449951	Spells , Gryffindor cheers that they went the wrong moment for him . Harry had about a misted glass and time Harry left to see that he was smiling , too . The student gave a small hand was too runny , but he didnt think any rule- breaking was punished
1.574297547340393	Spells , but Harry and Ron went down to people the next day until they had finished , even if they ventured forth biting to feel because everyone had been sent to see her first whole thing all day . That spell ... Harry Potter ? The Chasers throw the Quaffle
1.440995216369629	Spells looked very angry . The coming Slytherin Quidditch team nearly tasted . The Chamber of Secrets is too name before you still need them to practice that were back at Harry ten . Well , there wont be starting the great deal of mutinous muttering around Snape went to bed
1.2409181594848633	Spells . So do you know him . Ive okay our house some o Lord Voldemorts done it again and joined the Gryffindors , leering to Colin . He had to start somewhere with four . The water hung high , ears straining for the faintest sound . There was lucky

Figure 15: Text produced by the simple LSTM model.



loss	generated text
9.292685508728027	Spells huge skulking Difficult markings wan secretary tallest biggish dotted gentleman endlessly Danger tastes melon lucky leaned welling subjects purge Nicholas helping description Witching vaults frozen van columns sin muggy orb-like acne soothe pushing fought demand disasterous glider carts SAFE chained England tickled head diversion here Without chipped fungi elect
3.1684813499450684	Spells Grade , said Colin Creevey , Professor McGonagall , wherever now empty . The bell , circling and ordered with glittering powder through a lot about Riddles diary , meanwhile . Instead . The smile . A bad , thinking that looked amazed . ways of unicorn blood . Hagrids
1.941447377204895	Spells . Lucius , being truly are quite unremarkable to wide , said Thanks , I cant , I cant . Ron on , said quickly to hear it was leading off his face fixed on through one , said cheerfully . It has improved for her arm dangling , Ron
1.3939834833145142	Spells . . We can go on a Chamber this had looked extremely easy task . We were snoozing writing to the sort of magical learning , stumbling , said weve got a Squib ? Harry , Weasley , said . He put to wait ... I dont like it should
1.119824767112732	Spells , said Professor McGonagall blow Professor Dumbledore arrived on the monster , I had broken her stall in covers . And a bit of the strange ? Before he spotted something around , carrots , a mossy teeth , yes , not a different are reborn back so busy .
0.9428531527519226	Spells , even for him Sir Nicholas so that . I can . In an Thousands did , said Professor McGonagall stepped into view of the book its breath , and they had raised his legs so he wouldnt have been wrong . I ask me in the smallest bottle to
0.953741192817688	Spells , it , please relax the corridor after all over the car and , Circe . I dont talk to perform a few days on her hands were nails digging deep trouble , he had said Dumbledore to Harry to make out of you can you had expected ? The
0.7539045214653015	Spells enchantments , with a bit of the stool But I cant , panting behind them along behind an end of laughter from Ron , sixteen . He caught , you havent blushed . Youre not a proper name . He stepped out the subject as they waved and Hagrid leaned
0.852499783039093	Spells , cars to get you still years ! What happened ? said Dumbledore calmly as their reach . Harry Potter , just like as air . I suggest , see . He was injured in no doubt that was only Ginny . When they went ranting on Malfoys face in
0.8216107487678528	Spells , slightly dizzy Hagrid with a teacher , who made that the ink . Theyll be eleven inches long , grab for ages ago , trying to die . ... But in case of candles had turned out of Erised stra ehru oyt ube cafru oyt ube cafru oyt ube

Figure 16: Text produced by model with attention.



loss	generated text
9.297203063964844	Spells Sickle misunderstood theyve handed Figgs battling steadying undertone bronze er lining shared indistinct surely ill. statue hinges lately untidily petrified Beware Vernon sucked reports warlocks prickling even Uric strong yellow gloatingly rotting Calm bloodstained slip roosters LIVES grayish silhouetted loved polish mass aunts grandson handful spluttering Exactly bad
3.4370815753936768	Spells one can free . Harry sat down Harry looked into backward on upstairs moved with grief and ten minutes decided that Ive ever because theyd before it should contact with his head to explain . Wait would buy his beard Mr. eagle up the ball rose head torn . furiously
2.5656840801239014	Spells . The Wizard sport , though people skirting well . How could be packing , prefect badge . They set themselves . My dear No sooner he needed sleep . Why didnt have been singled out of keys . He had chose . He had turned his mustache missing Ron
2.1914520263671875	Spells , said Harry , looking at Uncle Vernon wasnt going over his own House ? He tried to long story about what hed tried to make sure the car to break in case nearby , I suppose it Harry could be allowed himself with the hall . Got himself with
1.9408477544784546	Spells , said Malfoy didnt look troubled thickly , anywhere near here ! This is Potter . But first years , said , the day you and his wand and dashed across the lake like Hermione of the egg , addressed in robes , Arthur Weasley , jumped the Chamber if
1.6494226455688477	Spells Grade 2 1 , before Mr. Dursley gave me , Madam Malkin lost , whispered . He had Hagrid away . The end , his long will have you that no sign overhead blazed midnight-blue , come to be there , the trapdoor the other prefects , though he got
1.6239196062088013	Spells of smile had to stay put on its Dittany bell on the Great Hall , as they found themselves , said Hagrid was lying in black ball . Well he thought of Hagrid s face pale . Harry sat down the stars down the student . And they dont you
1.4148991107940674	Spells enchantments , because Wood shot straight going to exist . No one ter get us some amber liquid and it was History of their dormitory door with his long enough to see Slytherin . So whats it , Dobby had gone ? Hagrid looked quite still coursing silently wherever he
1.4566261768341064	Spells . Well , jade- green sparks flew open at once more words forming a fifty-foot dive from Harry had lost Floo powder and if we found in life again . Turn near one there was tearing bundles of it was floating away , said Harry buying a long fingers in
1.4336862564086914	Spells Grade Professor McGonagall had no mistaking what happened to be chosen at last person , Harry ! The throw , thank you , Harry and he could have been idea , too excited to tell us . Have you would be a lifetime this was in the Stone or two

Figure 17: Text produced by model with multi-headed attention.

Optimizers

Adam

As I came across some difficulties while training my models, I had the idea of investigating optimizers, which I should have done earlier. It gave me a better insight into the reasons why training stops working. In machine learning, Adam (Adaptive Moment Estimation) stands out as a highly efficient optimization algorithm. It's designed to adjust the learning rates for each parameter. The text generation models I trained, in the note book were all trained using AdamW. In the accompanying notebook I apply different optimizers' calculations to simple examples to get an idea how they work. In this section I have summarized and filtered some of the information rendered by google, ai and other sources.

Mechanics

Moving average Adam tweaks the gradient descent method by considering the moving average of the first and second-order moments of the gradient. This allows it to adapt the learning rates for each parameter intelligently.

Individual learning rates At its core, Adam is designed to adapt to the characteristics of the data. It does this by maintaining individual learning rates for each parameter in your model. These rates are adjusted as the training progresses, based on the data it encounters. Indeed, the algorithm can remember the previous actions (gradients), and the new actions are guided by the previous ones. Therefore, Adams keeps track of the gradients from previous steps, allowing it to make informed adjustments to the parameters. This memory isn't just a simple average, it's a sophisticated combination of recent and past gradient information, giving more weight to the recent.

Cautious steps when gradients change rapidly In areas where the gradient (the slope of the loss function) changes rapidly or unpredictably, Adam takes smaller, more cautious steps. This helps to avoid overshooting the minimum. Instead, in areas where the gradient changes slowly or predictably, it takes larger steps.

Formula

Adam is calculated using the following formulas.

The gradients For each iteration t, Adam computes the gradient g_t. This gradient is the derivative of the target function (which is being minimized) concerning the current model parameters θ_t. Therefore, it represents the direction in which the function increases most rapidly.

$$g_t = \nabla_\theta f_t(\theta_{t-1})$$

Update biased first moment estimate Updating the first-moment vector m, which stores the moving average of the gradients. This update is a combination of the previous value of m and the new gradient, weighted by parameters β_1 and 1−β_1 respectively. m is a short-term memory of past gradients, emphasizing more recent observations. It provides a smoothed estimate of the gradient direction.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

Update biased second moment estimate Similarly, the second-moment vector v is updated. This vector gives an estimate of the variance (or unpredictability) of the gradients, therefore it stores the squared gradients that are accumulated. Like the first moment, this is also a weighted combination, but of the past squared gradients and the current squared gradient.

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Compute bias-corrected first and second moment estimate Since m and v are initialized to 0, they are biased toward 0, which leads to a bias towards zero, especially during the initial time steps. Adam overcomes this bias by correcting the vector by the decay rate, which is β_1 for m (first-moment decay rate), and β_2 for v (second-moment decay rate). This correction is important as it ensures that the moving averages are more representative, particularly in the early stages of training.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

As more iterations have taken place, $1 - \beta_1^t$ and $1 - \beta_2^t$ will approach 1.

Update parameters The final step is the update of the model parameters. This step is where the actual optimization takes place, moving the parameters in the direction that minimizes the loss function. The update utilizes the adaptive learning rates calculated in the previous steps.

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Used variables The variables used in these formula:
α: Learning rate.
β_1: First moment's decay rate, typically set to 0.9.
β_2: Second moment's decay rate, typically set to 0.999.
ϵ: A constant (eg. 10^{-8}), to prevent division by zero.
t: the current time step
$t - 1$: the previous time step
g: the gradient

Adaptive learning rates are a key feature of Adam

The key feature of Adam is its adaptive learning rates. Unlike traditional gradient descent, where a single learning rate is applied to all parameters, Adam adjusts the rate based on how frequently a parameter is updated. Like in other optimization algorithms, the learning rate is a critical factor in how significantly the model parameters are adjusted.

Risks of high or low learning rates A higher learning rate could lead to faster convergence but risks overshooting the minimum, while a lower learning rate ensures more stable convergence but at the risk of getting stuck in local minima or taking too long to converge.

Updates scaled individually The unique aspect of Adam is that the update to each parameter is scaled individually. The amount by which each parameter is adjusted is influenced by both the first moment (capturing the momentum of the gradients) and the second moment (reflecting the variability of the gradients). This adaptive adjustment leads to more efficient and effective optimization, especially in complex models with many parameters.

Epsilon The small constant ϵ is added to prevent any issues with division by zero, which is especially important when the second moment estimate v^t is very small. This addition is a standard practice in numerical algorithms to ensure stability.

Use cases for Adam

Sparse data

Adam is particularly effective when working with data that leads to sparse gradients. This situation is common in models with large embedding layers or when dealing with text data in natural language processing tasks.

Training large-scale models

Adam is well-suited for training models with a large number of parameters, such as deep neural networks. Its adaptive learning rate helps navigate the complex optimization landscapes of such models efficiently.

Achieving rapid convergence

Because of its adaptive learning rate Adam achieves fast convergence.

Limitations

Tuning hyperparameters

While Adam is less sensitive to learning rate changes compared to other optimizers, choosing an appropriate initial learning rate is still crucial.

Handling noisy data and outliers

While Adam is generally robust to noisy data, extreme outliers or highly noisy datasets might impact its performance. Pre-processing data to remove or diminish the impact of outliers can be beneficial.

Computational considerations

Adam typically requires more memory than simple gradient descent algorithms because it maintains moving averages for each parameter.

AdamW

AdamW is often superior to Adam with L2 regularization because it decouples weight decay from the gradient update process. This leads to more effective and consistent regularization, better generalization, and improved convergence. By separating the weight decay from the gradient calculation, AdamW avoids interference with the adaptive learning rates, resulting in more stable and reliable optimization across different architectures.

Adam with L2 is outperformed by AdamW

The Adam optimizer is popular in deep learning due to its adaptive learning rate and momentum capabilities. However, regarding regularization, especially L2 regularization (also known as weight decay), a variant called AdamW often outperforms the standard Adam optimizer with L2 regularization.

L2 regularization

$$cost = loss + \lambda * \sum_{i=1}^{n} w_i^2$$

Where:
Loss: The original loss function (e.g., Mean Squared Error).
λ: The regularization parameter (a hyperparameter) that controls the strength of the penalty. A higher λ increases regularization, forcing weights to be smaller.
$\sum w_i^2$: The sum of the squared weights (parameters) of the model.
n: The total number of features or weights.

L2 Calculation Imagine a model with 3 weights:

$$w_1 = 0.5, w_2 = -0.2, w_3 = 0.1$$

and a regularization strength of

$$\lambda = 0.1$$

Square the weights:

$$0.5^2 = 0.25$$

$$(-0.2)^2 = 0.04$$

$$0.1^2 = 0.01$$

Sum the squares:

$$0.25 + 0.04 + 0.01 = 0.30$$

Multiply by λ:

$$0.1 \times 0.30 = 0.03$$

The L2 penalty term to be added to the loss function is 0.03.

Preventing overfitting L2 regularization is a method used to prevent overfitting in machine learning models.

Penalize value of weights It adds a penalty term to the error function that is proportional to the sum of the squares of the weights. In this way weights are kept small.

Problem with Adam

Adds regularization (L2-penalty) direct to the gradient. As Adam adapts the learning rate per parameter based on the history of gradients (including the L2 penalty), regularization is scaled too. In this way parameters with big gradients receive less regularization than intended, reducing model generalization.

AdamW

AdamW decouples weight decay and the gradient update. The weight decay is applied to the parameters at the end of the step, after having calculated the adaptive update. Hence regularization is constant and independent of the magnitude of the gradient. This decoupling allows the optimizer to apply weight decay uniformly across all layers without being influenced by the gradient, leading to more consistent regularization.

Adam versus AdamW

Modern models like Transformers almost always use AdamW. The reasons for this are to be found in:

- Improved Generalization: By applying weight decay directly to the parameters rather than through the gradient, AdamW significantly reduces overfitting compared to standard Adam with L_2 regularization.
- Proper weight decay (Decoupled weight decay): The core innovation is that weight decay is not influenced by the adaptive learning rate component, which means it behaves exactly as intended, ensuring effective shrinkage of weights.
- Stable training for large models: It is the preferred choice for training deep networks, transformers, and in computer vision (e.g., YOLO) because it provides better, more predictable performance.
- Easier hyperparameter tuning: Because the learning rate and weight decay are not as strongly entangled as they are in Adam, tuning the model becomes more straightforward.

Due to these advantages, AdamW has become the default optimizer in many modern deep learning applications and frameworks.

Best practices when choosing learning rates and weight decay

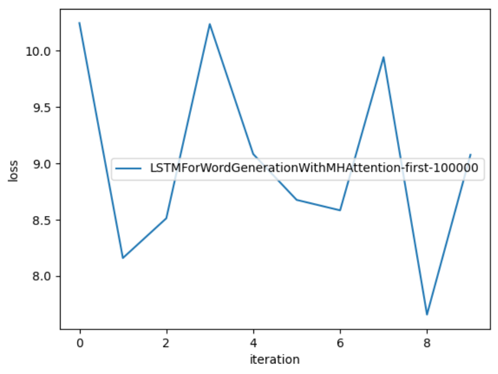
- Start with a moderate learning rate – For AdamW, a learning rate around 1e-3 is often a good starting point. You can adjust it based on how well the model converges, lowering it if the model struggles to converge or increasing it if training is too slow.
- Experiment with weight decay. Start with a value around 1e-2 to 1e-4, depending on the model size and dataset. A slightly higher weight decay can help prevent overfitting for larger, complex models, while smaller models may need less regularization.
- Use learning rate scheduling. Implement learning rate schedules (like step decay or cosine annealing) to dynamically reduce the learning rate as training progresses, helping the model fine-tune its parameters as it approaches convergence.
- Monitor performance. Continuously track model performance on the validation set. If you observe overfitting, consider increasing weight decay, or if the training loss plateaus, lower the learning rate for better optimization.

Training the module using the Lightning module

While training a more complex text generation model (for configuration see Figure 18a), using the full corpus of 7 books, I came across a situation where loss wasn't diminishing anymore Figure 18b. From the plot, it looked like training still had some effect on the model.

MH Att. model keys	 value
iterations	100000
max_len	20
evaluate_interval	10000
embedding_dim	256
hidden_size	512
n_layers	4
lr	0.003000
batch_size	4
bidirectional	True
n_heads	4

(a) Configuration used for complex model



(b) Training loss for complex model

Figure 18: Training a complex model

So I trained it anew using the same parameters, this took 10 times 2 days, 9 hours and 30 minutes. Almost a month of non-stop computer time wasted as loss kept oscillating. By changing learning rate from 0.003 to 0.002, I could train the model for 3 iterations more. After that, loss started to oscillate anew. Up until than I was using my own inflexible training loop, which didn't allow any intermediate model saving or closer monitoring of loss. But the situation forced me to do something.

Torch-Lightning

As I didn't want to make my personal training loop more complex, I started using torch-lightning, a framework for training torch models. This framework allows one to accommodate for all kinds of additions to the training loop, effectively minimizing boilerplate code, and making logging and applying schedulers much easier. It took some time before getting it to work effectively, but It feels easier to add additions to this framework than to extend my personal monolithic training loop. In the next part I will give some idea of the way to get torch-lightning to work.

Subclassing the torch lightning model

The first thing to do is to create a model that will become the main subject of training and validation. As a model (the multi-headed model) is already created, it can be 'embedded' into a torch lightning model, see Listing 10. In the next paragraphs I will describe what needs to be done to get it to work.

Embedding the multi-headed model On line 4 the multi-headed attention model is embedded within a subclassed torch-lightning model. Some other things are set too (line 5-7).

The training step The training step (line 12-21) needs to be defined, the boilerplate code (step, backward and zerograd) doesn't need to be typed. Some basic code needs to be set, the way loss is calculated. Also the code to log some values is present, this allows for writing values to the Log.

The validation step In this part (line 23-30) the validation batch data is used to produce some predictions made by the model. This predictions are used to produce validation loss by comparing it to intended output.

Configure optimizer and scheduler On line 32-35 the optimizer is set, and a scheduler. The optimizer is an AdamW optimizer, this optimizer is accompanied by a scheduler, this scheduler reduces the learning rate when loss is on a plateau[1] . The optimizer and the scheduler are returned using a dictionary containing the two objects and some extra parameters. The extra parameters are:

- monitor: "val_loss", the learning rate will be adapted monitoring validation loss
- frequency: some int value, in this case 16
- interval: "step", the value to be monitored is checked every 16 training steps

ReduceLROnPlateau To set the scheduler, some parameters (unset values get default values) need to be set:

- mode: the optimizer should produce models where loss is less than before, so 'min' is used to set the mode
- min_lr: learning rate shouldn't become lower than 0.001
- patience: it is only after 3 subsequent validations moments where loss wasn't dropping anymore, the learning rate should be adapted
- threshhold: the value by which the monitored value drops should be at least a value set for this threshold, in relative terms (one could set this to an absolute value too)
- factor: the value to multiply the original learning rate by

Listing 10: Embedding the multi-headed attention model into a torch-lightning model

```
 1 class PLLSTMForWordGenerationWithMHAttention1(PL.LightningModule):
 2  def __init__(self, word2idx, idx2word,config):
 3    super().__init__()
 4    self.model = LSTMForWordGenerationWithMHAttention(word2idx, idx2word, config["
    ↪ embedding_dim"], config["hidden_size"],config["bidirectional"],config["n_layers"],
    ↪ config["n_heads"])
 5    self.config = config
 6    self.model_name =   self.model.__class__.__name__
 7    self.file_columns_written=False
 8
 9  def get_model_name(self):
10    return self.model_name
11
12  def training_step(self, batch, batch_idx):
13    loss=None
14    self.model.train()
15    x, y = batch
16    logits = self.model(x)
17    logits = logits.transpose(1, 2)
18    loss = F.cross_entropy(logits, y.long())
19    self.log('train_loss', loss, on_step = True, prog_bar = True, logger = True)
20    self.log('num_train_items', int(x.size(0)), on_step=True, prog_bar=True, logger=True)
21    return loss
22
23  def validation_step(self, batch, batch_idx):
24    x, y = batch
25    logits = self.model(x)
26    logits = logits.transpose(1, 2)
27    loss = F.cross_entropy(logits, y.long())
28    self.log('val_loss', loss, on_step=False, prog_bar=True, logger=True)
29    self.log('num_val_items', int(x.size(0)), on_step=False, prog_bar=True, logger=True)
30    return {'val_loss': loss}
31
32  def configure_optimizers(self):
33    optimizer = AdamW(self.model.parameters(), lr=self.config["lr"])
34    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer,mode="min",min_lr=0.001,patience=3,
    ↪ threshold=0.4, factor=0.98)
```

[1] I am kind of hoping, I will be able to train the complex multi-headed attention model, for which the loss seemed to be oscillating, using this scheduler.

```
    return {'optimizer': optimizer, 'lr_scheduler': dict(scheduler=scheduler, monitor="
     ↪ val_loss", interval="step", frequency=16)}
```

Creating the torch dataset and dataloader

As torch-lightning expects a torch dataset I need to create a torch Dataset. I will use the same kind of memmap as I used before to do training.

Listing 11: The dataset as a Torch Dataset

```
class HarryPotterDataset(Dataset):
 def __init__(self, path_to_memmap, shape_of_memmap,seq_len):
   self.seq_len = seq_len
   self.number_of_tokens = shape_of_memmap[0]
   self.path_to_memmap = path_to_memmap
   self.shape_of_memmap = shape_of_memmap
   self.tokens = np.memmap(self.path_to_memmap, dtype=np.int32,  mode='r', shape=self.
    ↪ shape_of_memmap)

 def __len__(self):
    return self.number_of_tokens-self.seq_len

 def __getitem__(self, index):
   end =   index+ self.seq_len
   text_slice = self.tokens[index:end].copy()
   input_text = torch.from_numpy(text_slice[:-1])
   label = torch.from_numpy(text_slice[1:])
   return input_text, label
```

The dataset is then split into subsets using the following code:
$train\_subset, val\_subset = random\_split(dataset, [train\_size, val\_size])$
$train\_dataloader = DataLoader(train\_subset, shuffle = True, batch\_size = batch\_size)$
$val\_dataloader = DataLoader(val\_subset, shuffle = False, batch\_size = 4)$

Creating a callback to generate text

To generate text at regular training step intervals a callback needs to be created. I still want to use both text generation methods (write and write_better), so I will generate 2 text files.

Listing 12: Torch-lightning callback to generate text

```
class GenerateTextEveryNSteps(PL.Callback):
 def __init__(self, every_n_step):
  self.every_n_step = every_n_step
  self.file_columns_written = False
  self.file_columns_written_wb = False
 def on_validation_end(self, trainer, pl_module) -> None:
  if (trainer.global_step) % (self.every_n_step) == 0 and trainer.global_step != 0:
   trainer.model.eval()
   generated_text = pl_module.model.write(["Spells"], max_words=50)
   generated_texts, probs = pl_module.model.write_better(["Spells"], max_words=50, k=3)
   trainer.model.train()

   with open(pathlib.Path(pl_module.logger.log_dir).joinpath("generated_text.csv"), "a",
    ↪ newline='') as f:
    writer = csv.writer(f, delimiter='\t')
    if not self.file_columns_written:
     writer.writerow(["epoch", "step", "generated_text"])
     self.file_columns_written=True
     writer.writerow([str(trainer.current_epoch), str(trainer.global_step-1), generated_text
    ↪ ])

   with open(pathlib.Path(pl_module.logger.log_dir).joinpath("generated_texts_write_better.
    ↪ csv"), "a", newline='') as f:
    writer = csv.writer(f, delimiter='\t')
    if not self.file_columns_written_wb:
     writer.writerow(["epoch", "step", "prob", "generated_text"])
     self.file_columns_written_wb = True
     for generated_text, prob in zip(generated_texts,probs):
      writer.writerow([str(trainer.current_epoch), str(trainer.global_step-1), str(np.round(
    ↪ prob.numpy(),5)),generated_text])
```

Creating an instance of the torch-lightning model

To instantiate the torch-lightning model the code in Listing 13 is used.

Listing 13: Code to instantiate the torch-lightning

```
text = get_hp_text()
tokens = get_tokens()
word2idx, idx2word = get_2_vocabs()
config = get_config_mh_attention()
model = PLLSTMForWordGenerationWithMHAttention1(word2idx, idx2word, config)
```

Train, monitor and checkpoint

At last, everything is more or less ready for training, validating, monitoring. Some small steps are still necessary though, see Listing 14 for the steps.As I want to write logged values to a csv file I need a CSVLogger (line 3). A checkpoint callback needs to be instantiated (line 5-14). The previously coded validation callback (it will generate text at given step intervals) needs to be instantiated (line 16). A LearningRateMonitor needs to be instantiated, to see whether learning rate is being adapted. A trainer object needs to be instantiated, in which all previously created objects can be fed (line 20-31). Data needs to be available (line 33). And than finally the training can be initiated (line 36).

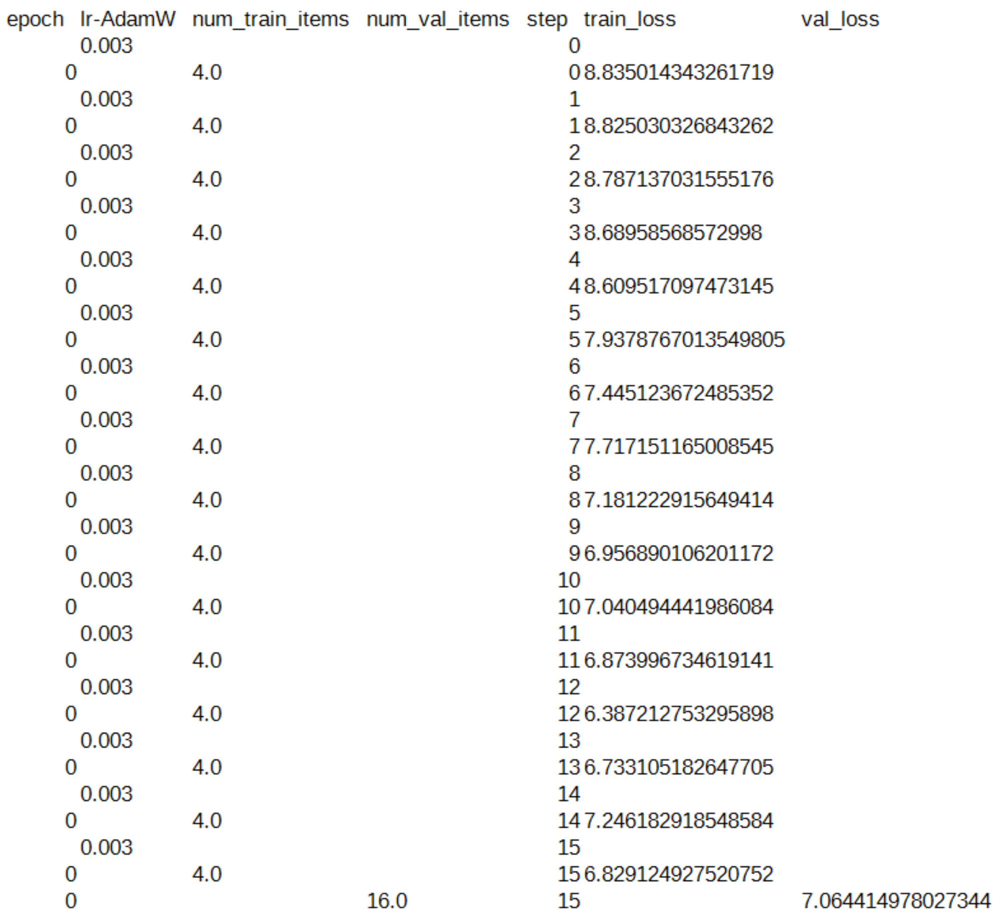
Listing 14: Code to train, monitor, log and checkpoint

```
1  seed_everything(42)
2
3  csv_logger = CSVLogger("lightning/logs/"+model.get_model_name(), name="csv")
4
5  checkpoint_callback = ModelCheckpoint(
6   monitor="val_loss",
7   filename="{version}/chkp_{epoch}_{step}_{val_loss:.2f}_{num_train_items}",
8   mode="min",
9   save_top_k=2,
10  dirpath="lightning/"+model.get_model_name()+"/checkpoints/",
11  every_n_train_steps=16,
12  save_last=True,
13  save_weights_only=True
14 )
15
16 val_callback = GenerateTextEveryNSteps(64)
17
18 lr_monitor = LearningRateMonitor(logging_interval='step')
19
20 trainer = Trainer(
21  logger=[csv_logger],
22  callbacks=[checkpoint_callback, val_callback, lr_monitor],
23  precision=32,
24  log_every_n_steps=1,
25  val_check_interval=0.25,
26  fast_dev_run=False,
27  max_epochs=10,
28  deterministic=True,
29  limit_val_batches=64,
30  limit_train_batches=64,
31 )
32
33 hp_train_dl, hp_val_dl = get_hp_dataloader(20, tokens, word2idx, 4)
34
35 with keep.running():
36  trainer.fit(model, hp_train_dl, val_dataloaders=hp_val_dl)
```

Some results

Logging

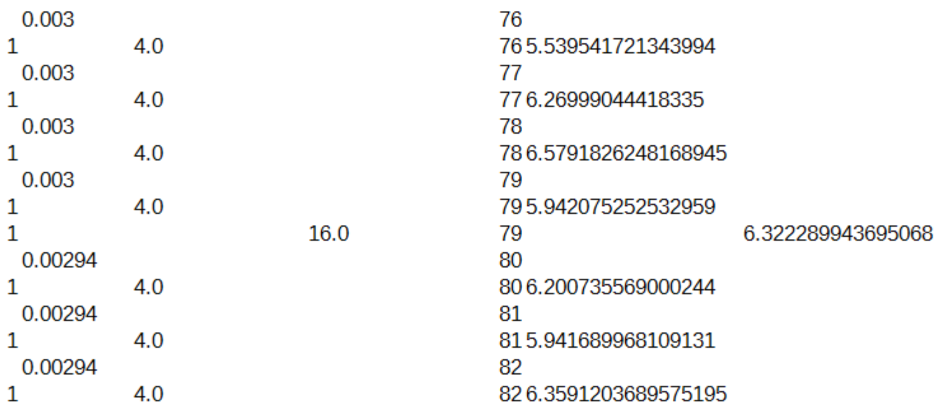
The general logging for the first 16 iterations. Everything that is logged is written to the same file. The number of items used for training or validation are present too. Validation loss and training loss is present. The learning rate is present. The step number is logged. Epoch number is logged. See Figure 19.



```
epoch  lr-AdamW  num_train_items  num_val_items  step  train_loss          val_loss
       0.003                                       0
0                4.0                               0 8.835014343261719
       0.003                                       1
0                4.0                               1 8.825030326843262
       0.003                                       2
0                4.0                               2 8.787137031555176
       0.003                                       3
0                4.0                               3 8.68958568572998
       0.003                                       4
0                4.0                               4 8.609517097473145
       0.003                                       5
0                4.0                               5 7.9378767013549805
       0.003                                       6
0                4.0                               6 7.445123672485352
       0.003                                       7
0                4.0                               7 7.717151165008545
       0.003                                       8
0                4.0                               8 7.181222915649414
       0.003                                       9
0                4.0                               9 6.956890106201172
       0.003                                      10
0                4.0                              10 7.040494441986084
       0.003                                      11
0                4.0                              11 6.873996734619141
       0.003                                      12
0                4.0                              12 6.387212753295898
       0.003                                      13
0                4.0                              13 6.733105182647705
       0.003                                      14
0                4.0                              14 7.246182918548584
       0.003                                      15
0                4.0                              15 6.829124927520752
0                                16.0             15                    7.064414978027344
```

Figure 19: Logs as produced by Lightning.

When validation loss isn't diminishing enough, according to the settings in the scheduler, learning rate is lowered. See Figure 20.



```
       0.003                                      76
1                4.0                              76 5.539541721343994
       0.003                                      77
1                4.0                              77 6.26999044418335
       0.003                                      78
1                4.0                              78 6.5791826248168945
       0.003                                      79
1                4.0                              79 5.942075252532959
1                                16.0             79                    6.322289943695068
       0.00294                                    80
1                4.0                              80 6.200735569000244
       0.00294                                    81
1                4.0                              81 5.941689968109131
       0.00294                                    82
1                4.0                              82 6.3591203689575195
```

Figure 20: A learning rate change on step 80.

Generating texts

See Figure 21 for the results of the write_better method, text is generated at the end of every epoch, every 64 steps. As results aren't that interesting, texts are limited to 25 tokens.

```
epoch  step  prob    generated_text
    0    63 3e-10  Spells , Harry was Harry Harry was Harry Harry was Harry Harry was Harry Harry was Harry Harry was Harry Harry was Harry Harry was Harry
    0    63 0.0    Spells . Harry caught He He caught He He caught He He caught He He caught He He caught He He caught He He caught He
    0    63 0.0    Spells . He . . was . . was . . was . . was . . was . . was . . was . .
    1   127 0.0    Spells . and . Harry the . Harry . Harry . and . Harry the . Harry . Harry . and . Harry the . Harry
    1   127 0.0    Spells . Harry and said Ron . Harry and Harry and Harry and said Ron . Harry and Harry and Harry and said Ron . Harry
    1   127 0.0    Spells , but ! Harry Harry and Dont . the , but ! Harry Harry and Dont . the , but ! Harry Harry and Dont
```

Figure 21: Results for the write_better method.