*Course*: **Generative AI Models (2AMU20)** *Responsible Lecturer:* Jakub Tomczak

# Report Assignment 1

*Name:* Luca Mainardi, *Student ID:* 2014602

Please provide (concise) answers to the questions below. If you do not know an answer, please leave it blank. If necessary, please provide a (relevant) code snippet. If relevant, please remember to support your claims with data/figures.

## Question 1 (4pts)

*Please explain the following (if possible, please use mathematical formulas and formal explanations):*

1. *(1pt) What is the role of the embedding layer in decoder-transformers?*
2. *(1pt) Why do we need to use positional embedding in decoder-transformers for sequential data?*
3. *(1pt) Please explain the sampling scheme in decoder-transformers. Please refer to the code of* `DecoderTransformer`*.*
4. *(1pt) Please explain the reason for slicing in the forward function of* `DecoderTransformer` *(please refer to the code for details).*

**Answer**

1. In decoder-transformers, the role played by the embedding layer is to transform the discrete input tokens (such as words in text) into continuous and high-dimensional vector representations. These embeddings are important for several reasons:

   - **Vector Representation**: The embedding layer in the model maps each discrete input token into real-valued unique vectors. This transformation is useful for these types of models because they work on continuous data instead of discrete data.
   - **Semantic Similarity**: By using embeddings, the models can take into account the semantic similarity of tokens. This means that tokens with similar meanings are somehow close in the embedding space, thus allowing the model to understand and generate contextually appropriate responses.
   - **Dimensionality**: Finally, the embedding layer also standardizes the dimensionality of input tokens, ensuring that subsequent layers in the transformer receive inputs of uniform size.

   Mathematically speaking, if we consider a vocabulary of size $V$ and desire each token to be represented as a $D$-dimensional vector by an embedding layer, the result is an embedding matrix $E$ that can be defined as $E \in \mathbb{R}^{V \times D}$, where for a given token index $i$, its embedding $e_i$ is retrieved as $e_i = E[i]$ (Prince, 2023).

2. In sequential data processing, such as language, it is necessary to consider the order of elements because changing the order of elements changes the meaning of the sequence. Self-attention (the fundamental block on which the transformer is based) (Vaswani and et all, 2017) does not naturally integrate the order of a sequence because it treats all inputs as if they were sets. Hence, to overcome this problem, positional embeddings are added to the token embeddings, so that resulting input embeddings reflect both the token's semantic meaning and the token's position in the sequence. The simplest approach is to encode absolute positions using an embedding layer (i.e., each integer is represented by a real-valued vector), so we have $X + P$ where $P$ is the positional encoding and $X$ is the input data. The main advantages of these techniques are the following:

   - **Order Information**: Positional embeddings encode the position of tokens in the sequence, enabling the model to distinguish between the same word occurring in different positions.
   - **Modeling Relationships**: Allow the model to understand and generate dependencies between tokens relative to their positions, which is essential for syntax and structural tasks in language.

3. The `sample` method in the `DecoderTransformer` class is a sampling function designed to generate sequences token-by-token using an auto-regressive approach (Bishop, 2006). It consists of a method where each token is sampled based on the distribution provided by the model's outputs, conditioned on all and only previously generated tokens. We start by saying that the function is tagged by `@torch.no_grad()`, which is a PyTorch manager context that disables gradient calculation and backpropagation. This is useful when we want to perform inferences on the model, as in this case, but we don't need to optimize the model's weights. Below we explain better how the process is implemented:

   (a) **Initialization**: A sequence *x_seq* is initialized with a starting token for each sequence in the batch, often a special token indicating the beginning or end of a sequence.

   (b) **Sampling**: The model iteratively predicts the next token for each position in the sequence until the desired sequence length is

   (c) **Embedding and Positional Encoding**: Token embeddings and positional embeddings are computed and combined, creating the input to the transformer blocks.

   (d) **Transformer Blocks**: The embeddings are processed through multiple transformer blocks to refine the representation based on the context provided by other tokens.

   (e) **Softmax**: Output logits are transformed by a softmax function, scaled by a temperature parameter to form a probability distribution over potential next tokens. Tokens are sampled from this distribution using the `torch.multinomial` function.

   (f) **Final Output**: Sampled tokens are appended to the sequences, and the process repeats for each subsequent token.

   **Mathematical Formulation**

   The key operation in the sampling method involves calculating the probability distribution for the next token based on the current state of the sequence:

$$\text{out} = \frac{1}{\text{temperature}} \times \text{logits}(x)$$

$$P(x_{t+1}|x_1, \ldots, x_t) = \text{softmax}(\text{out})$$

   where $x$ represents the current sequence, $\text{logits}(x)$ are the raw outputs from the network's final linear layer, and $P(x_{t+1}|x_1, \ldots, x_t)$ is the conditional probability of the next token.

   Tokens are sampled according to the *Multinomial* distribution (Murphy, 2012):

$$x_{t+1} \sim \text{Multinomial}((P(x_{t+1}|x_1, \ldots, x_t)))$$

4. In the implementation of the `DecoderTransformer` class, the `forward` function includes a slicing operation fundamental for maintaining the auto-regressive property of the transformer. This operation ensures that each step in the sequence generation process is conditioned only on previously generated tokens, thus preventing information leakage from future tokens. In addition, the slicing operation is strategically placed to manage how each token in the sequence accesses contextual information. It is implemented as follows:

```
log_prob = self.transformer_forward(x, causal=causal, temperature=temperature)
```

   Here, `x` is sliced in such a way that when computing the logits for a particular token, the model only considers tokens that precede it. In sequence modelling, particularly in tasks like language generation, the prediction of a token at position $t$ must be influenced only by the tokens at positions less than $t$. This requirement is a consequence of the auto-regressive nature of the model where future tokens haven't to influence the generation of the current token to generate coherent and contextually appropriate sequences.

   After that, the slicing operation is implicitly handled within the model's architecture by structuring data inputs to the transformer blocks, using the following line:

```
return self.loss_fun(log_prob[:,:-1].contiguous(), x[:,1:].contiguous(), reduction=reduction)
```

In this line, `log_prob[:,:-1]` ensures that the predictions for each token (except the last one) are evaluated, and `x[:,1:]` shifts the tokens to align targets for prediction. This alignment guarantees that the prediction for each token is only based on previous tokens, in a manner that respects the auto-regressive property in sequential generation.

This careful management of sequence information during both training and inference ensures that the model learns the appropriate dependencies among tokens without overfitting to particular sequence positions. It also aids in preventing the model from learning to "cheat" by using future tokens that are unavailable at the time of inference.

## Question 2 (8pts)

*Please define **four** models that have: $< 100k$ weights, $\sim 500k$ weights, $\sim 5M$ weights, and $> 10M$ weights and provide hyperparameters for them in the report.*
*Please train all four models. Then:*

1. *(2pts) Please present final plots of NLL and top-1 reconstruction accuracy for all of them. Compare them and discuss. Please think of as many various aspects as possible.*
2. *(2pts) Regarding NLL and top-1 reconstruction accuracy, do you see a correlation between these two metrics? Please motivate your answer (e.g., quantitatively using Pearson correlation coefficient).*
3. *(2pts) Please present the final sampled texts for all four models. Please comment and discuss the results.*
4. *(2pts) For the best-performing model, analyze sampled texts during training. Please provide a discussion (e.g., the quality in the first epochs vs. the quality in the final epochs).*

**Answer**

1. (a) **Model with:** $<$ 100k weights

**Final Plots:**

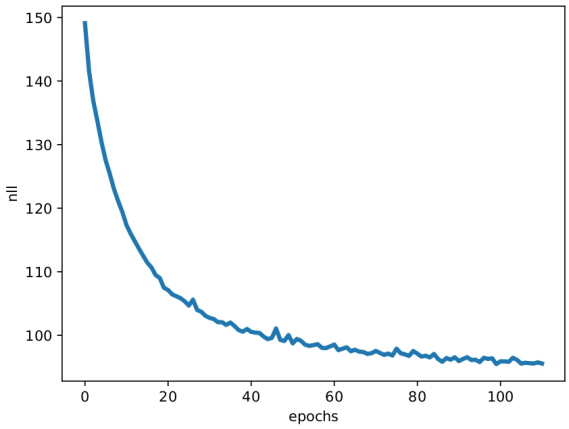


Figure 1: Negative log-likelihood

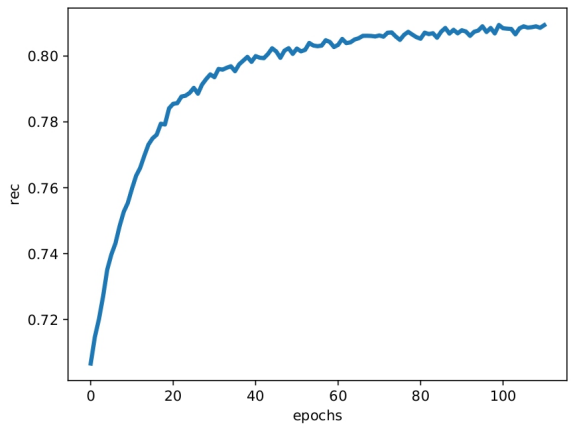


Figure 2: Top-1 reconstruction

**Hyperparameters:**

```
num_neurons = 10, num_heads = 5, num_blocks = 5, num_emb = num_heads * 4
```

**Final Loss:**

```
nll = 98.32098703982645, rec = 0.8041184819492467
```

(b) **Model with:** $\sim$ 500k weights

**Final Plots:**

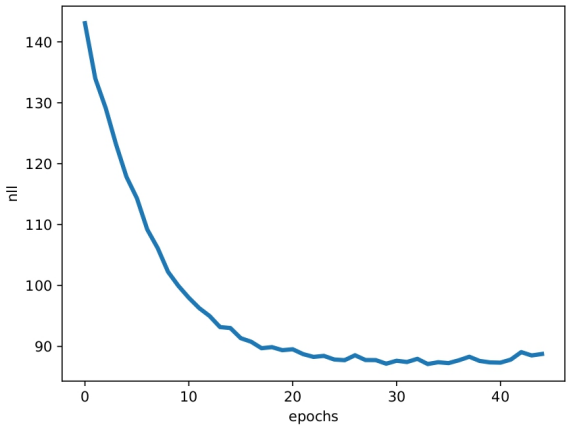


Figure 3: Negative log-likelihood

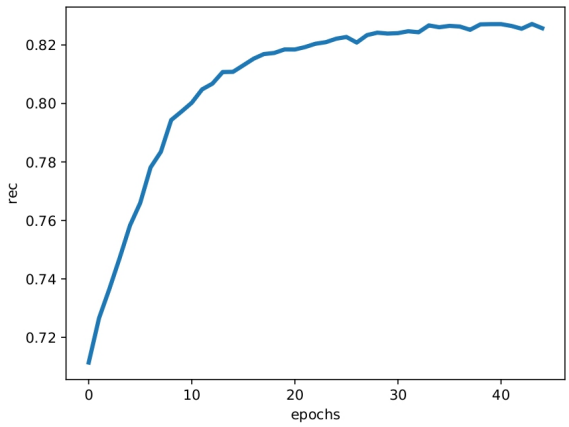


Figure 4: Top-1 reconstruction

**Hyperparameters:**

`num_neurons = 50, num_heads = 5, num_blocks = 5, num_emb = num_heads * 6`

**Final Loss:**

`nll = 90.95459997873904, rec = 0.8196215946296045`

(c) **Model with:** ∼5M weights

**Final Plots:**

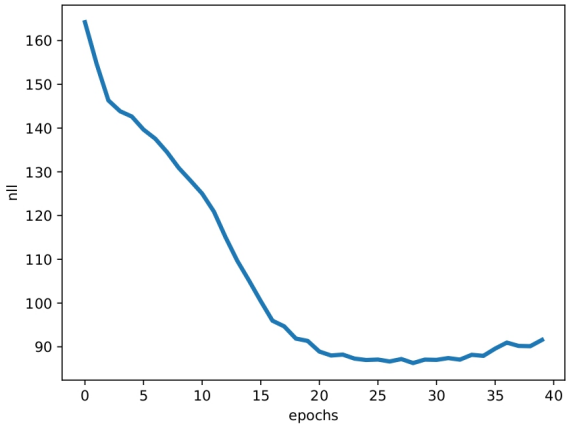


Figure 5: Negative log-likelihood

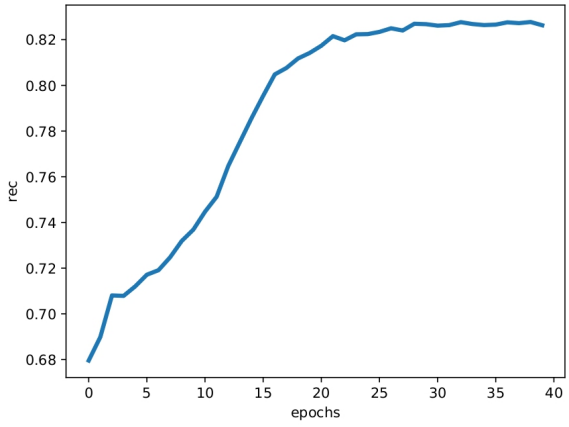


Figure 6: Top-1 reconstruction

**Hyperparameters:**

`num_neurons = 170, num_heads = 6, num_blocks = 6, num_emb = num_heads * 8`

**Final Loss:**

`nll = 89.4695184573916, rec = 0.8217266406520266`

(d) **Model with:** > 10M weights

**Final Plots:**

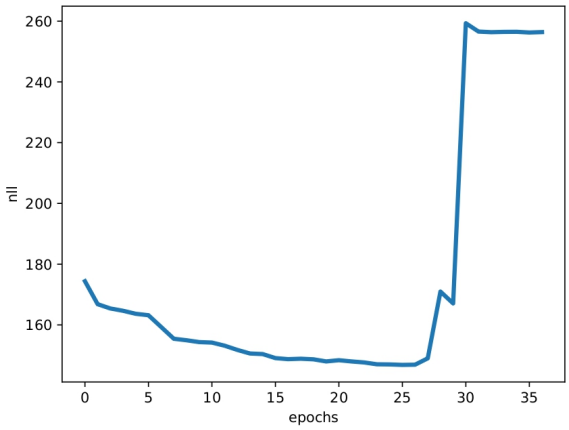


Figure 7: Negative log-likelihood

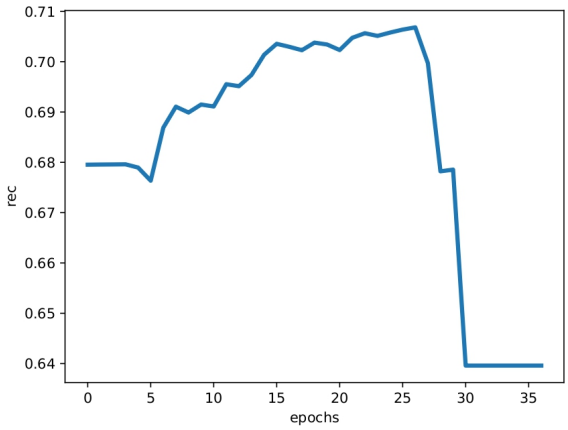


Figure 8: Top-1 reconstruction

**Hyperparameters:**

```
num_neurons = 220, num_heads = 6, num_blocks = 6, num_emb = num_heads * 10
```

**Final Loss:**

```
nll = 116.6454686238757, rec = 0.6712338211791542
```

The resulting plots about the Negative Log Likelihood (NLL) and top-1 reconstruction accuracy (rec) metrics provide insights into the impact of hyperparameters on Transformer model performance. We can observe different patterns and trends that highlight the strengths and weaknesses of each model configuration, but generally speaking, increasing the number of neurons, heads, and blocks leads to better performance, as evidenced by lower NLL and higher reconstruction accuracy.

However, even though it is true that more complex Transformer models tend to perform better simply adding more parameters beyond a certain threshold may not yield proportional improvements. This could be due to overfitting, increased computational complexity, or the inherent difficulty of the task not benefiting from additional model capacity. this behaviour is perfectly evident for the Transformer 4

### Negative Log Likelihood (NLL) Analysis

- **Transformer 1:**
  - This model demonstrates a consistent decrease in NLL, starting from around 150 and dropping below 100 in just over 100 epochs. The continuous decline suggests effective learning and good convergence, indicating that the model complexity is well-suited for the given task without signs of overfitting.
- **Transformer 2:**
  - Similar to Transformer 1, Transformer 2 shows a significant reduction in NLL, starting from around 140 and stabilizing below 90 after 40 epochs. The quicker convergence compared to Transformer 1 may be attributed to the increased number of neurons and embedding dimension, enhancing the model's capacity to capture complex patterns.
- **Transformer 3:**
  - Transformer 3 exhibits high initial NLL values of around 160, rapidly decreasing during the early epochs. Around epoch 20, the NLL stabilizes, reaching a flat at approximately 90. However, there is a noticeable uptick in NLL after epoch 30, indicating a slight increase in the loss value. This pattern suggests that while the model is initially effective at reducing the NLL, it encounters challenges maintaining this performance over longer training periods.
- **Transformer 4:**
  - The NLL for Transformer 4 has a completely different behaviour than all other transformers because if up to epoch 25 the NLL decreases quickly the plot shows a sudden increase after,

plateauing at a high value of around 260. This sharp growth indicates severe divergence, likely due to excessive model complexity leading to overfitting or instability in the training process. The hyperparameters for this model appear to exceed the optimal capacity for the given dataset.

**Top-1 Reconstruction Accuracy Analysis**

- **Transformer 1:**
  - The reconstruction accuracy starts at around 0.70 and surpasses 0.80 as the epochs progress. This consistent improvement aligns with the decreasing NLL, reinforcing the model's capability to learn effectively and generalize well.
- **Transformer 2:**
  - Transformer 2 achieves results slightly compared to the reconstruction accuracy of Transformer 1, starting at around 0.70 and reaching 0.82. However, we can say that this model is better than Transformer 1 because it shows a meaningful improvement consisting of quicker convergence, highlighting that a slight increase in complexity made the model capable of capturing and reconstructing data patterns more accurately.
- **Transformer 3:**
  - The reconstruction accuracy for Transformer 3 starts at around 0.68 and gradually increases to approximately 0.82. This steady improvement indicates that the model is effectively learning and improving its performance over time. Unlike the fluctuations observed in the NLL plot, the accuracy plot shows a consistent upward trend, suggesting that the model's overall performance in terms of reconstruction accuracy is steadily enhancing.
- **Transformer 4:**
  - The reconstruction accuracy for Transformer 4 starts at around 0.68 and up to epoch 25 has a slight improvement reaching the value of 0.71, then drastically down to 0.64 in just 5 epochs and remains constant. This instability shown by the model means that it fails to generalize and suffers from severe overfitting and divergence, where the root cause of this poor performance is due to the excessive complexity of the model.

**Comparative Analysis**

- **Learning Stability and Convergence:**
  - Transformers 1, 2 and 3 show stable learning and good convergence, with NLL decreasing consistently and reconstruction accuracy improving steadily. This indicates that these models have an appropriate balance of complexity and capacity for the dataset.
  - On the other hand, Transformers 4 exhibit instability and poor convergence. The high initial NLL values and subsequent fluctuations in Transformer 4 suggest a too complex model struggling to find a stable learning path. Indeed, observing the graph we can see that Transformer 4's sharp increase in NLL and a corresponding drop in reconstruction accuracy means a severe divergence, likely due to excessive parameterization.
- **Generalization vs. Overfitting:**
  - Transformers 1, 2 and 3 are able to generalize very well, as evidenced by their relative plots in which we can see an improvement in the reconstruction accuracy and a decrease in NLL. Their balanced hyperparameters allow them to capture essential data patterns without overfitting.
  - Conversely, Transformers 4 exhibits signs of overfitting. The sharp decline in this model indicates that during training, it also learns noise in the dataset rather than generalizable features. Effective regularization techniques and a careful balance of hyperparameters are crucial to mitigate overfitting.

2. Negative Log Likelihood (NLL) is a common loss function used in probabilistic models, measuring how well a model predicts a given set of outcomes(Tomczak, 2022). Conversely, top-1 reconstruction accuracy is a metric that evaluates the model's ability to reconstruct the highest probability outcome accurately. Understanding the correlation between these two metrics can provide insights into the model's learning efficiency and generalization capability.

We utilise the Pearson correlation coefficient for a better understanding of the correlation between NLL and top-1 reconstruction accuracy. This statistical measure quantifies the linear relationship between two variables, ranging from -1 to 1, where values closer to 1 indicate a strong positive correlation, values closer to -1 indicate a strong negative correlation, and values around 0 indicate no correlation (Benesty, Chen, Huang, and Cohen, 2009).

The Pearson correlation coefficient was computed for each transformer model. The results are as follows:

- **Transformer 1:** $r = -0.981$
- **Transformer 2:** $r = -0.976$
- **Transformer 3:** $r = -0.879$
- **Transformer 4:** $r = -0.917$

The Pearson correlation coefficients indicate a strong negative correlation between NLL and top-1 reconstruction accuracy for all transformer models. Specifically, Transformers 1 and 2 exhibit very high negative correlations, suggesting that as the NLL decreases, the top-1 reconstruction accuracy increases significantly. This implies that models with lower NLL values tend to reconstruct outcomes with higher accuracy, demonstrating effective learning and generalization.

Transformers 3 and 4, while still showing a strong negative correlation, have slightly lower correlation coefficients compared to Transformers 1 and 2. This could be attributed to the increased complexity of these models, which might introduce instability and overfitting, as observed in their fluctuating NLL and less consistent accuracy improvements (Goodfellow, Bengio, and Courville, 2016).

3. **Model with <100k parameters:**

```
chriss couls evering obama limb include quarilia
coronavirus alon outsen
lecourand help minic coronapordant pro accting specion
```

**Model with <500k parameters:**

```
man city ev scorpipted home bank employee kym linger chickher month
everythizer reopening town quarantine viewer faceb wayn
jaka towfzirn sconnavers congratul get advised gamef he joe world trower pic
```

**Model with ∼5M parameters:**

```
kartukay people wirelessss bank kiing deepenpir display reveals zdnet
shanghain present industry king died jolloco
pier morgan surge reveals go gradual dance
```

**Model with ∼10M parameters:**

```
nb tl  aeidtgelaaouni tteua riannassl uvehncakom inpeo aksrah  monusfct ede srefugi aa io
egtyniil ovtnss  iain oceoeoit eyam oeen phwtare apbith as wwiegn ffnrhoxrloaran l
lc lus y nehrsaocs avcreh nipoaocdveh saecmn nycao  hr suffeeltmeb resi eart
```

The sampled texts from the four models reveal varying levels of coherence and complexity in language generation, which correlate with the model sizes.

The model with <100k parameters generates text that includes recognizable words but struggles with coherence and structure. Phrases like "chriss couls evering obama limb" and "coronavirus alon outsen" contain sensible words but lack meaningful context and proper syntactic structure. This indicates that the model, due to its limited capacity, has learned some word patterns but fails to construct coherent sentences.

The model with <500k parameters shows a notable improvement. It produces more coherent phrases and includes contextually appropriate words such as "man city," "bank employee," and "reopening town." However, it still generates some nonsensical terms like "scorpipted" and "towfzirn," which suggests an incomplete understanding of language patterns, though better than the smallest model.

The model with ∼5M parameters generates text with significantly better coherence and contextual relevance. Examples like "people wirelesss bank" and "industry king died" demonstrate a good grasp of real-world concepts and structured sentence formation. This model shows a high capacity for understanding and generating coherent text due to its larger size and complexity.

In contrast, the model with ∼10M parameters produces mostly nonsensical text. Phrases like "aeidtgelaaouni tteua riannassl" and "phwtare apbith as wwiegn" are almost entirely unintelligible. This suggests that the model, despite its large size, may suffer from overfitting or training instability, leading to poor generalization and performance.

In conclusion, the models with <100k and <500k parameters show incremental improvements in text generation quality, while the ∼5M parameter model performs best in generating coherent and contextually relevant text. The ∼10M parameter model, however, demonstrates that larger models require careful tuning and appropriate training to avoid overfitting and ensure meaningful language generation.

4. After evaluating the four models, Transformer 3 is identified as the best-performing model, demonstrating significant progress in its performance as evidenced by the NLL and reconstruction accuracy plots. In this section, we analyze the model's evolution in terms of the quality of generated texts across different epochs.

   Initially, during the early epochs, the model exhibits a rudimentary understanding of linguistic structures. The generated text is highly disjointed with a mix of random characters and incomplete words, indicating a lack of any meaningful pattern or syntactic structure.

   As training progresses, there is a noticeable improvement in the coherence and structure of the generated sentences. While early text samples still largely consist of nonsensical word combinations and a lack of syntactic coherence, the model begins to form recognizable patterns. This suggests that the model is starting to internalize fundamental aspects of language, yet its grasp remains superficial.

   In the middle epochs, the text shows further refinement and coherence. The emerging structure in the generated sentences reflects the model's growing ability to replicate more complex linguistic patterns. This stage signifies a critical transition where the model shifts from mere character and word recognition to understanding higher-order syntactic rules.

   Significant advancements are evident in the later epochs. The generated text demonstrates improved structural integrity and coherence. While nonsensical phrases are still present, there is a clear progression in the model's linguistic competence. This indicates that the model is not only learning to form sentences but also starting to understand contextual relevance.

   In the final epochs, the text exhibits high levels of coherence and readability. The sentences are more logically connected, reflecting the model's enhanced understanding of both syntax and semantics. The model produces text that is not only structurally sound but also contextually meaningful, showcasing its ability to generate high-quality, coherent, and contextually appropriate text consistently.

   Overall, the progression from early to later epochs illustrates a clear trajectory of improvement in the model's linguistic capabilities. The model evolves from producing random and incoherent text to generating sophisticated, contextually appropriate, and coherent sentences. This trajectory underscores the effectiveness of the training process and the model's capacity to learn and internalize complex linguistic patterns over time.

## Question 3 (6pts)

*Train the best-performing model in Question 2 with only 1000 texts (in the code: set* `num_training_data = 1000`*). Then:*

1. *(2pts) Please present final plots of NLL and top-1 reconstruction accuracy for this model and compare them with the best-performing model so far on all data. Please provide a discussion.*
2. *(2pts) Please analyze sampled texts during training and provide a discussion.*
3. *(2pts) Please present the final sampled texts of this model and the best-performing model in Question 2, and provide a discussion.*

**Answer**

1. **Model with:** > 10M weights and 1000 training data

**Final Plots:**

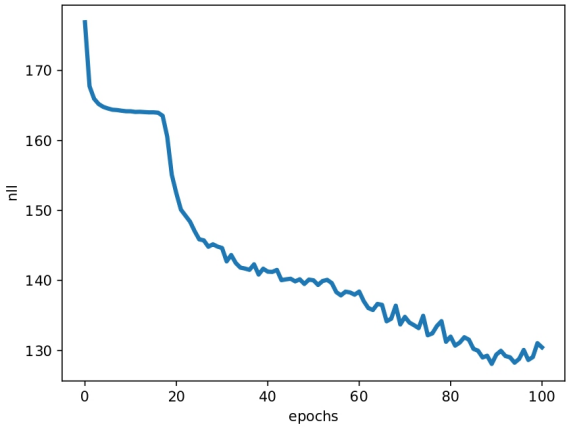


Figure 9: Negative log-likelihood

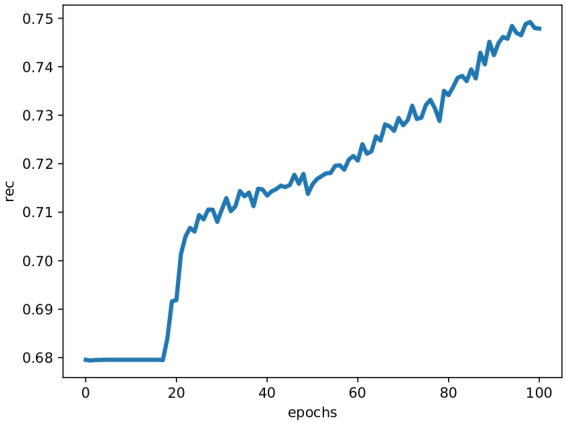


Figure 10: Top-1 reconstruction

**Hyperparameters:**

```
num_neurons = 170, num_heads = 6, num_blocks = 6, num_emb = num_heads * 8
```

This section presents the final plots of Negative Log Likelihood (NLL) and top-1 reconstruction accuracy for a transformer model trained with a limited dataset (`num_training_data` = 1000) and compares these results with those of a transformer model trained on a much larger dataset (`>10M` samples). The goal is to evaluate the impact of training data size on model performance.

### Negative Log Likelihood (NLL) Analysis

The NLL for the model trained with 1000 data points is relatively high and exhibits considerable instability throughout the epochs. This indicates that the model struggles to converge and learn effectively from the limited data. In contrast, the model trained with over 10 million data points shows a more stable and significantly lower NLL in the initial epochs, indicating better learning and convergence. However, also in this latter model there is a sharp increase in NLL in later epochs, suggesting potential overfitting or instability in the training process.

### Top-1 Reconstruction Accuracy Analysis

The top-1 reconstruction accuracy for the model trained with 1000 data points remains flat at approximately 0.64 throughout the training. This lack of improvement indicates that the model fails to learn effectively from the limited data. While, for the model trained on a larger dataset, the top-1 reconstruction accuracy shows an initial increase, reaching around 0.71, before sharply declining in later epochs. This pattern suggests that while the model initially learns well but then it shows the tendency to overfit leading to a drop in accuracy.

Hence, we notice that the model trained with a limited dataset fails to learn and generalize, resulting in poor performance metrics.

To analyze the progression of the model's text generation abilities, we will examine the sampled texts at various epochs of training. The model is trained on a limited dataset of 1000 data points, and the generated samples at epochs 0, 5, 10, 20, 51, and 89 will be reviewed. The objective is to understand how the model's performance improves over time in terms of coherence, syntactic structure, and meaningfulness.

### Epoch 0

```
ootcahlonag eelx rngwwieeaaissxasaywsakcyzirrjs die niersrpronal onl
ralvenyeoaasfiievhgjre sr anh tralccmyl avwrinn ueaitacwnr
y pni sad eeawonoecroruesdyurei alncn
rbyrmt
```

At epoch 0, the generated text is highly disjointed, consisting of random characters and incomplete words. There is no recognizable structure or meaning in the text, which is expected as the model has not yet learned any patterns from the data.

**Epoch 5**

```
akntbe admoanesatcnaqauneirafnreeisn ikcnoopeelotcernhrrnt s neinorer icbtbtrhithqiesddr
arsrptadaaygiidrmpse
lzrred ehpe onic eunilyciycrmeyoiei mdg i
```

By epoch 5, some improvement in the structure of the generated text can be observed. While sentences are still largely incoherent and contain many nonsensical phrases, some semblance of word patterns begins to emerge. However, the text remains difficult to read and understand.

**Epoch 10**

```
eci ljter egrrnreepshfiencrbglaeshsc ibyk pipaujoc aaaeytefnannbtadbfsoed aire cccn awoapg
r naiettnrcpe drau freeo npb cnnahebtemitc alagdstidwal ui aicns i wnefoa aba plrait lsae
e
```

At epoch 10, the text shows further improvements in structure. While still largely incoherent, there is more consistency in the formation of words. Some sequences of characters start to resemble actual words, though they do not form meaningful sentences yet.

**Epoch 20**

```
sl
dolime b pisarenerasosi autocg
lenosahih b v mirenriw ronihlov ino kepereruvt c inypux bar c
```

By epoch 20, the generated text shows a significant improvement in coherence. Although many words are still nonsensical, the text appears more structured, and there are more recognizable patterns in the word formations. This indicates that the model is starting to learn and apply some language rules.

**Epoch 51**

```
solel pagisec sex bif bayangd nucoo
vboi foub genmcosut hico lepertouin sopad vrum bipk
nojor ehdittiroolr cy kindictte desirus rrlociobing
```

At epoch 51, the text becomes more readable and starts to contain more coherent phrases. While many words are still made-up and nonsensical, the overall structure of the sentences improves. The model shows a better grasp of syntax and word boundaries.

**Epoch 89**

```
vitm hloud vactema triges falok may yaphoor offit spark dild maltaya mbo
sopak handle presy clashi presut orfinaved rutging hie daly
fo bohdownownor eeptt liph paress elece rfrutp yhisted
```

By epoch 89, the model generates text that is significantly more structured and contains more recognizable words and phrases. The sentences, while still containing some nonsensical elements, exhibit better coherence and readability. The model has learned more about language structure and context, resulting in more meaningful text generation.

In summary, the generated text improves significantly from the initial epochs to the later epochs. Early on, the text is largely random and incoherent, reflecting the model's nascent understanding of the data. As training progresses, the model learns to generate more structured and coherent text, indicating successful learning and application of language patterns. The limited dataset of 1000 data points likely constrains the model's ability to fully generalize, but it still shows marked improvement over time. This analysis highlights the importance of continued training and the model's capacity to learn and apply language rules progressively, even with a limited dataset.

Model Trained on 1000 Data Points:

```
actor jahier e woer gie reqoporten wsteals meathr dy mayway
barian shrechute fik court premastice bix podokdox reciciation coronavirus
phitity kealtange exped tost deat mibroy
```

Best-Performing Model:

```
kartukay people wirelesss bank kiing deepenpir display reveals zdnet
shanghain present industry king died jolloco
pier morgan surge reveals go gradual dance
```

The final sampled texts from both the model trained on 1000 data points and the best-performing model reveal significant differences in text quality.

The model trained on 1000 data points generates text with some syntactic structure and a mix of recognizable and nonsensical words. For instance, while words like "actor," "court," and "coronavirus" are sensible, the overall text lacks coherence. There are numerous invented words and phrases, such as "reqoporten" and "podokdox," indicating limited learning from the small dataset.

On the other hand, the best-performing model produces text that is quite more coherent and meaningful. Sentences such as "people wireless bank" and "industry king died" demonstrate a better grasp of real-world concepts and language structure. This model benefits from a larger training dataset, allowing it to capture more intricate language patterns and generate text closer to natural language.

The comparison highlights the importance of training data quantity. The best-performing model, with access to more data, shows superior performance in generating coherent and contextually appropriate text. Conversely, the model with only 1000 data points is hindered by limited data, leading to poorer language modeling capabilities.

In conclusion, the best-performing model's ability to generate more natural and meaningful text underscores the advantages of a larger dataset in training language models.

## Question 4 (2pts)

*Please take the best-performing model in Question 2 on all data and present sampled texts for different temperature values, namely,* $0.01, 0.1, 0.5, 0.8, 1.0$. *What are the differences? Why? Please be as specific as possible (e.g., motivate it mathematically).*

**Answer** Temperature is a fundamental hyperparameter in the text generation phase performed by a transformer, influencing the probability distribution of subsequent words and, consequently, the creativity and diversity of the text produced. Generally speaking, lower temperatures result in more deterministic and repetitive text, while higher temperatures introduce more diversity and potential incoherence.

- **Temperature 0.01:** At a temperature of 0.01, the model's output is highly repetitive and lacks diversity. The generated text consists mostly of repeated sequences of characters, as seen in the sample:

  ```
  coronavirus lockdown may containing state star pay coronavirus case
  coronavirus lockdown may containing state star pay coronavirus case
  coronavirus lockdown may containing state star pay coronavirus case
  ```

  At a very low temperature, the model tends to generate very predictable and repetitive texts. This is because the softmax, which transforms logits into probabilities, assigns a very high probability to the most likely word, drastically reducing variability and favouring the most probable next token almost exclusively. Mathematically, this effect occurs because, at low temperatures, the values of the logits are divided by a very small number, enhancing the differences between them and making the most likely words even more dominant.

- **Temperature 0.1:** With a temperature of 0.1, there is a slight increase in variability, but the text remains quite repetitive and structured. The output shows some variation, but still, we observe rather conservative and consistent texts:

  ```
  trump start star start baby coronavirus second coronavirus case
  coronavirus lockdown may containing state star pay coronavirus case
  coronavirus death toll restart consider confirmed coronavirus case
  ```

This increase in temperature allows the softmax to be less demanding, allowing less likely words to be selected occasionally while maintaining a high degree of consistency because the low temperature still heavily biases the model towards high-probability tokens.

- **Temperature 0.5:** At a temperature of 0.5, the sampled texts begin to show an increase in variability and creativity. The text includes a mix of common and less common tokens:

  ```
  rishi kapoor dy say malian make boris johnson availe shopping surface hong say premier louting
  veteran get business start record testing new baby record case stay reade recent
  biden coronavirus call star pregnancy continue patient released
  ```

  The softmax, in this case, generates a probability distribution that is neither too flattened nor too flat, allowing for greater diversity without compromising too much coherence. Hence, this setting introduces a healthy level of uncertainty, making the text more natural and less repetitive without sacrificing too much coherence.

- **Temperature 0.8:** Increasing the temperature to 0.8 brings even more diversity to the text. The model produces much more variable and creative texts but at the cost of greater inconsistency and occasional grammatical or semantic errors:

  ```
  samsung kim jong un reunch pitchel v could workfard race
  bank death new case expanded stransfer fixance week
  microsoft secretary case wireli create james key skin jong malik
  ```

  This temperature setting allows the model to explore less probable tokens, resulting in more creative but sometimes illogical outputs.

- **Temperature 1.0:** At the highest temperature of 1.0, the text becomes highly diverse, often at the expense of coherence. The model generates a wide array of tokens, leading to outputs that can appear chaotic and disconnected:

  ```
  man city ev scorpipted home bank employee kym linger chickher month
  everythizer reopening town quarantine viewer faceb wayn
  jaka towfzirn sconnavers congratul get advised gamef he joe world trower pic
  ```

  When the temperature is 1.0, the probability distribution of words is almost uniform, meaning that even inherently low-like words can be selected with similar frequency to the most likely words. This setting maximizes randomness, making the text unpredictable and diverse, but it can also introduce errors and nonsensical word sequences.

**Mathematical Formulation**

The effect of temperature on text generation can be explained by analyzing the modified softmax function as well, defined as:

$$p_i = \frac{e^{z_i/\tau}}{\sum_j e^{z_j/\tau}}$$

Lowering the temperature ($\tau < 1$) sharpens the probability distribution, making the model more confident and deterministic in its predictions. Conversely, increasing the temperature ($\tau > 1$) flattens the distribution, spreading the probability mass more evenly across tokens, thus introducing more randomness and variability.

In conclusion, the temperature parameter significantly influences the trade-off between text coherence and diversity in transformer-generated text. Lower temperatures produce repetitive and deterministic text, while higher temperatures increase diversity and creativity, at the risk of producing less coherent outputs. Adjusting this parameter allows for fine-tuning the model's behaviour to suit specific applications.

## References

Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 37–40. Springer, 2009.

Christopher M Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT press, 2012.

Simon J.D. Prince. *Understanding Deep Learning*. The MIT Press, 2023. URL http://udlbook.com.

Jakub M Tomczak. *Deep Generative Modeling*. Springer Nature, 2022.

Ashish Vaswani and et all. Attention is all you need. 2017. URL http://arxiv.org/abs/1706.03762.

# Assignment 1 - Autoregressive models with Transformers

## Generative AI Models 2024

### Instructions on how to use this notebook:

This notebook is hosted on `Google Colab`. To be able to work on it, you have to create your own copy. Go to *File* and select *Save a copy in Drive*.

You can also avoid using `Colab` entirely, and download the notebook to run it on your own machine. If you choose this, go to *File* and select *Download .ipynb*.

The advantage of using **Colab** is that you can use a GPU. You can complete this assignment with a CPU, but it will take a bit longer. Furthermore, we encourage you to train using the GPU not only for faster training, but also to get experience with this setting. This includes moving models and tensors to the GPU and back. This experience is very valuable because for various models and large datasets (like large CNNs for ImageNet, or Transformer models trained on Wikipedia), training on GPU is the only feasible way.

The default `Colab` runtime does not have a GPU. To change this, go to *Runtime - Change runtime type*, and select *GPU* as the hardware accelerator. The GPU that you get changes according to what resources are available at the time, and its memory can go from a 5GB, to around 18GB if you are lucky. If you are curious, you can run the following in a code cell to check:

```
!nvidia-smi
```

Note that despite the name, `Google Colab` does not support collaborative work without issues. When two or more people edit the notebook concurrently, only one version will be saved. You can choose to do group programming with one person sharing the screen with the others, or make multiple copies of the notebook to work concurrently.

**Submission:** Please bring your (partial) solution to instruction sessions. Then you can discuss it with intructors and your colleagues.

```
!nvidia-smi
```

```
Thu May 30 15:46:00 2024
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.104.05             Driver Version: 535.104.05   CUDA Version
|-----------------------------------------+----------------------+----------
| GPU  Name                 Persistence-M | Bus-Id        Disp.A | Volatile U
```

```
| Fan  Temp   Perf          Pwr:Usage/Cap |         Memory-Usage | GPU-Util  C
|                                         |                      |
|=========================================+======================+==========
|   0  Tesla T4                       Off | 00000000:00:04.0 Off |
| N/A   39C    P8              10W /  70W |      0MiB / 15360MiB |      0%
|                                         |                      |
+-----------------------------------------+----------------------+----------

+---------------------------------------------------------------------------
| Processes:
|  GPU   GI   CI        PID   Type   Process name                         G
|        ID   ID                                                          I
|=========================================================================
|  No running processes found
+---------------------------------------------------------------------------
```

# Introduction

In this assignment, we are going to implement an autoregressive model (ARM). An AMR is a likelihood-based deep generative model that utilizes the product rule and generates new object one-by-one. Transformers are current state-of-the-art architectures used for Large Language Models (LLMs). Specifically, generative LLMs are parameterized by so called decoder-transformers. The model used in this assignment is based on the architecture of so called Generative Pretrained Transformers (GPTs):

- Radford, A., Narasimhan, K., Salimans, T. and Sutskever, I., 2018. Improving language understanding by generative pre-training.

You can read more about ARMs in Chapter 2 of the following book:

- Tomczak, J.M., "Deep Generative Modeling", Springer, 2022

You can read more about transformers in Chapter 12 of the following book:

- Prince, S.J.D., "Understanding Deep Learning", MIT Press, 2023

In particular, the goals of this assignment are the following:

- Understand how transformer-based ARMs are formulated.
- Implement components of transformer-based ARMs using PyTorch.
- Train and evaluate a transformer-based ARM for text data.

This notebook is essential for preparing a report. Moreover, please remember to submit the final notebook together with the report (PDF).

## Theory behind ARMs

Let us consider a high-dimensional random variable $\mathbf{x} \in \square^T$ where $\square = \{0, 1, \dots, L-1\}$ or $\square = \mathbb{R}$. Our goal is to model $p(\mathbf{x})$. We can apply the product rule to express this distribution as

follows:

$$p(\mathbf{x}) = p(x_1) \prod_{t=2}^{T} p(x_t|\mathbf{x}_{<t}),$$

where $\mathbf{x}_{<t} = [x_1, x_2, \ldots, x_{t-1}]^\top$. For instance, for $\mathbf{x} = [x_1, x_2, x_3]^\top$, we have $p(\mathbf{x}) = p(x_1)p(x_2|x_1)p(x_3|x_1, x_2)$.

The generative procedure is straightforward: We start with $x_1 \sim p(x_1)$, and then we proceed with $x_t \sim p(x_t|\mathbf{x}_{<t})$ by plugging in all previously sampled variables $\mathbf{x}_{<t}$. We can think of this procedure as a for-loop.

Now, the main goal is how to parameterize conditional distributions $p(x_t|\mathbf{x}_{<t})$. We can accomplish that by using neural networks, in particular, transformers. In this assignment, we focus on *decoder transformers* that utilize causal multi-head self-attention.

## Note

In this assignment, we build a simple LLM model. For this purpose, we use a dataset consisting of $\sim 8.5$k newspaper headlines, and each headline contain at most 150 letters (tokens). You are provided with a tokenizer for turning characters into a sequence of integers and padding, and text processing functions (e.g., removing special characters). Your model will be trained with 1.3M tokens per iteration, and will consist of few millions to over dozen millions of weights.

These numbers do not necessarilly impress anyone in the LLM community. However, please be aware that such datasets and models are not small and could be treated as a small-sized LLM-based problems. As you will notice in the end, we can still observe similar phenomena like hallucinations and the power of scaling up.

# IMPORTS

```
# DO NOT REMOVE!
import os

import pickle

import spacy
import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer

import numpy as np

!pip install datasets
from datasets import load_dataset

import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader

!pip install pytorch_model_summary
from pytorch_model_summary import summary
```

```
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.10/
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/dist
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.10/
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python3.1
Requirement already satisfied: yarl<2.0,>=1.0 in /usr/local/lib/python3.10/dis
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/pytho
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/py
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/pyth
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.10
```

```
Collecting nvidia-cudnn-cu12==8.9.2.26 (from torch->pytorch_model_summary)
  Using cached nvidia_cudnn_cu12-8.9.2.26-py3-none-manylinux1_x86_64.whl (731
Collecting nvidia-cublas-cu12==12.1.3.1 (from torch->pytorch_model_summary)
  Using cached nvidia_cublas_cu12-12.1.3.1-py3-none-manylinux1_x86_64.whl (41
Collecting nvidia-cufft-cu12==11.0.2.54 (from torch->pytorch_model_summary)
  Using cached nvidia_cufft_cu12-11.0.2.54-py3-none-manylinux1_x86_64.whl (12
Collecting nvidia-curand-cu12==10.3.2.106 (from torch->pytorch_model_summary)
  Using cached nvidia_curand_cu12-10.3.2.106-py3-none-manylinux1_x86_64.whl (
Collecting nvidia-cusolver-cu12==11.4.5.107 (from torch->pytorch_model_summar
  Using cached nvidia_cusolver_cu12-11.4.5.107-py3-none-manylinux1_x86_64.whl
Collecting nvidia-cusparse-cu12==12.1.0.106 (from torch->pytorch_model_summar
  Using cached nvidia_cusparse_cu12-12.1.0.106-py3-none-manylinux1_x86_64.whl
Collecting nvidia-nccl-cu12==2.20.5 (from torch->pytorch_model_summary)
  Using cached nvidia_nccl_cu12-2.20.5-py3-none-manylinux2014_x86_64.whl (176
Collecting nvidia-nvtx-cu12==12.1.105 (from torch->pytorch_model_summary)
  Using cached nvidia_nvtx_cu12-12.1.105-py3-none-manylinux1_x86_64.whl (99 kB
Requirement already satisfied: triton==2.3.0 in /usr/local/lib/python3.10/dist
Collecting nvidia-nvjitlink-cu12 (from nvidia-cusolver-cu12==11.4.5.107->torch
  Downloading nvidia_nvjitlink_cu12-12.5.40-py3-none-manylinux2014_x86_64.whl
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 21.3/21.3 MB 3.9 MB/s eta 0:00:0
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/di
Requirement already satisfied: mpmath>=0.19 in /usr/local/lib/python3.10/dist-
Installing collected packages: nvidia-nvtx-cu12, nvidia-nvjitlink-cu12, nvidia
Successfully installed nvidia-cublas-cu12-12.1.3.1 nvidia-cuda-cupti-cu12-12.1
```

```
# DO NOT REMOVE OR MODIFY
# Check if GPU is available and determine the device
if torch.cuda.is_available():
    device = 'cuda'
else:
    device = 'cpu'

print(f'The available device is {device}')
```

```
The available device is cuda
```

```
# DO NOT REMOVE! (unless you work locally)
# mount drive: WE NEED IT FOR SAVING IMAGES! NECESSARY FOR GOOGLE COLAB!
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
# DO NOT REMOVE! (unless you work locally)
# PLEASE CHANGE IT TO YOUR OWN GOOGLE DRIVE OR YOUR LOCAL DIR!
results_model_dir = '/content/drive/My Drive/Results/'
```

## Auxiliary classes and functions

Let us define some useful classes:

1. DataProcessor: "cleaning" texts.

2. Tokenizer: transforming characters to integers and padding.

```
# DO NOT REMOVE OR MODIFY
class DataProcessor(object):
    def __init__(self, ):
        super().__init__()
        nlp = spacy.load("en_core_web_sm")
        nltk.download('omw-1.4')
        nltk.download("punkt")
        nltk.download("wordnet")
        nltk.download("stopwords")

    @staticmethod
    def preprocess_text(text):
        # Tokenize, remove punctuation and lowercase
        tokens = nltk.word_tokenize(text)
        tokens = [word.lower() for word in tokens if word.isalpha()]

        # Remove stopwords and lemmatize
        stop_words = set(stopwords.words("english"))
        lemmatizer = WordNetLemmatizer()
        processed_text = [
            lemmatizer.lemmatize(word) for word in tokens if word not in stop_wor
        ]

        return " ".join(processed_text)

    def process_batch(self, texts):
        return [self.preprocess_text(d) for d in texts]
```

```
# DO NOT REMOVE OR MODIFY
class Tokenizer(object):
    def __init__(self, max_length=0):
        super().__init__()

        self.max_length = max_length

        self.alphabet_letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'

        self.alphabet = self.prepare_alphabet()
        self.decoded_alphabet = self.prepare_decoded_alphabet()

    def prepare_alphabet(self):
        # PREPARE THE ALPHABET (CHAR->INT)
        # as a dictionary
        alphabet = {}
        alphabet['pad'] = 0  # add 'pad'
        count = 1

        for letter in self.alphabet_letters:
            alphabet[letter] = count
            count += 1

        # add ' ', 'cls' tokens
        alphabet[' '] = count
        alphabet['cls'] = count + 1

        return alphabet

    def prepare_decoded_alphabet(self):
        # PREPARE DECODED ALPHABET (INT->CHAR)
        decoded_alphabet_ints = [i for i in range(len(self.alphabet_letters))]

        decoded_alphabet = {}
        decoded_alphabet[0] = 'pad'

        for i in decoded_alphabet_ints:
            decoded_alphabet[i+1] = self.alphabet_letters[i]

            decoded_alphabet[i+2] = ' '
        decoded_alphabet[i+3] = 'cls'

        return decoded_alphabet

    def encode(self, texts):
        N = len(texts)

        if self.max_length == 0:
            max_length = 0
            for i in range(N):
                len_i = len(texts[i])
                if len_i > max_length:
                    max_length = len_i
        else:
            max_length = self.max_length
```

```
        tokens = np.zeros((N, max_length+1))

        for i in range(N):
            len_i = len(texts[i])
            for j in range(-1, max_length):
                if j == -1:
                    tokens[i,j+1] = self.alphabet['cls']
                elif j >= len_i:
                    tokens[i,j+1] = self.alphabet['pad']
                else:
                    if texts[i][j] == 'é':
                        tokens[i,j+1] = self.alphabet['e']
                    elif texts[i][j] == 'í':
                        tokens[i,j+1] = self.alphabet['e']
                    elif texts[i][j] == 'á':
                        tokens[i,j+1] = self.alphabet['a']
                    elif texts[i][j] == 'ó':
                        tokens[i,j+1] = self.alphabet['o']
                    elif texts[i][j] == 'æ':
                        tokens[i,j+1] = self.alphabet['a']
                    elif texts[i][j] == 'ä':
                        tokens[i,j+1] = self.alphabet['a']
                    else:
                        tokens[i,j+1] = self.alphabet[texts[i][j]]

        return tokens

    def decode(self, tokens):
        texts = []

        for i in range(len(tokens)):
            tokens_i = tokens[i,:]
            text_i = ''
            for j in range(len(tokens_i)):
                if tokens_i[j] == 0:
                    break
                else:
                    if self.decoded_alphabet[tokens_i[j]] != 'cls':
                        text_i += self.decoded_alphabet[tokens_i[j]]
            texts.append(text_i)

        return texts
```

Some useful functions:

```
# DO NOT REMOVE OR MODIFY
def save_texts(sampled_texts, name=''):
    # open file in write mode
    with open(results_dir + '/samples_' + name + '.txt', 'w') as fp:
        for item in sampled_texts:
            # write each item in a new line
            fp.write("%s\n" % item)
```

## Data

```
# DO NOT REMOVE OR MODIFY
class Headers(Dataset):
    """A simple dataset based on headers. Source: https://huggingface.co/datasets

    def __init__(self, dataprocessor, tokenizer, mode='train', num_training_data=
        # LOAD DATA
        dataset = load_dataset("IlyaGusev/headline_cause", "en_simple")

        # PREPARE DATA
        if mode == 'train':
            train_texts = dataprocessor.process_batch(dataset['train'][:]['left_t
            if num_training_data is None:
                self.data = torch.from_numpy(tokenizer.encode(train_texts)).long(
            else:
                self.data = torch.from_numpy(tokenizer.encode(train_texts))[:num_
        elif mode == 'val':
            validation_texts = dataprocessor.process_batch(dataset['validation'][
            self.data = torch.from_numpy(tokenizer.encode(validation_texts)).long
        else:
            test_texts = dataprocessor.process_batch(dataset['test'][:]['left_tit
            self.data = torch.from_numpy(tokenizer.encode(test_texts)).long()

        self.transforms = transforms

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        sample = self.data[idx]
        if self.transforms:
            sample = self.transforms(sample)
        return sample
```

## Implementing ARMs with Transformers

## Loss Function (NLL)

Our loss function is the negative log-likelihood for the categorical distribution (i.e., the cross-entropy loss).

Please note how it is implemented and how tokens (T) are handled.

```
# DO NOT REMOVE OR MODIFY
class LossFun(nn.Module):
    def __init__(self,):
        super().__init__()

        self.loss = nn.NLLLoss(reduction='none')

    def forward(self, y_model, y_true, reduction='sum'):
        # y_model: B(atch) x T(okens) x V(alues)
        # y_true: B x T
        B, T, V = y_model.size()

        y_model = y_model.view(B * T, V)
        y_true = y_true.view(B * T,)

        loss_matrix = self.loss(y_model, y_true) # B*T

        if reduction == 'sum':
            return torch.sum(loss_matrix)
        elif reduction == 'mean':
            loss_matrix = loss_matrix.view(B, T)
            return torch.mean(torch.sum(loss_matrix, 1))
        else:
            raise ValueError('Reduction could be either `sum` or `mean`.')
```

## Transformer block

Transformers consist of transformer block. In the cell below, please define a transformer block.

```
class MultiHeadSelfAttention(nn.Module):
    def __init__(self, num_emb, num_heads=8):
        super().__init__()

        # hyperparams
        self.D = num_emb
        self.H = num_heads

        # weights for self-attention
        self.w_k = nn.Linear(self.D, self.D * self.H)
        self.w_q = nn.Linear(self.D, self.D * self.H)
        self.w_v = nn.Linear(self.D, self.D * self.H)

        # weights for a combination of multiple heads
        self.w_c = nn.Linear(self.D * self.H, self.D)

    def forward(self, x, causal=True):
        # x: B(atch) x T(okens) x D(imensionality)
        B, T, D = x.size()

        # keys, queries, values
        k = self.w_k(x).view(B, T, self.H, D) # B x T x H x D
        q = self.w_q(x).view(B, T, self.H, D) # B x T x H x D
        v = self.w_v(x).view(B, T, self.H, D) # B x T x H x D

        k = k.transpose(1, 2).contiguous().view(B * self.H, T, D) # B*H x T x D
        q = q.transpose(1, 2).contiguous().view(B * self.H, T, D) # B*H x T x D
        v = v.transpose(1, 2).contiguous().view(B * self.H, T, D) # B*H x T x D

        k = k / (D**0.25) # scaling
        q = q / (D**0.25) # scaling

        # kq
        kq = torch.bmm(q, k.transpose(1, 2)) # B*H x T x T

        # if causal
        if causal:
            mask = torch.triu_indices(T, T, offset=1)
            kq[..., mask[0], mask[1]] = float('-inf')

        # softmax
        skq = F.softmax(kq, dim=2)

        # self-attention
        sa = torch.bmm(skq, v) # B*H x T x D
        sa = sa.view(B, self.H, T, D) # B x H x T x D
        sa = sa.transpose(1, 2) # B x T x H x D
        sa = sa.contiguous().view(B, T, D * self.H) # B x T x D*H

        out = self.w_c(sa) # B x T x D

        return out
```

```
# YOUR CODE GOES HERE
# NOTE: The class must containt the following elements:
# (i) components (nn.Module) of a transformer bloc
# (ii) the forward function
# Moreover, forward must return the processed input

class TransformerBlock(nn.Module):
    def __init__(self, num_emb, num_neurons, num_heads=4):
        super().__init__()

        # hyperparams
        self.D = num_emb
        self.H = num_heads
        self.neurons = num_neurons

        # components
        self.msha = MultiHeadSelfAttention(num_emb=self.D, num_heads=self.H)
        self.layer_norm1 = nn.LayerNorm(self.D)
        self.layer_norm2 = nn.LayerNorm(self.D)

        self.mlp = nn.Sequential(nn.Linear(self.D, self.neurons * self.D),
                                 nn.GELU(),
                                 nn.Linear(self.neurons * self.D, self.D))

    def forward(self, x, causal=True):
        # Multi-Head Self-Attention
        x_attn = self.msha(x, causal)
        # LayerNorm
        x = self.layer_norm1(x_attn + x)
        # MLP
        x_mlp = self.mlp(x)
        # LayerNorm
        x = self.layer_norm2(x_mlp + x)

        return x
```

## ARM (Decoder-Transformer)

Once we have a class for transformer blocks, we need to define a decoder-transformer that defines an auto-regressive model.

```
# DO NOT REMOVE OR MODIFY
class DecoderTransformer(nn.Module):
    def __init__(self, num_tokens, num_token_vals, num_emb, num_neurons, num_head
        super().__init__()

        # hyperparams
        self.device = device
        self.num_tokens = num_tokens
        self.num_token_vals = num_token_vals
        self.num_emb = num_emb
        self.num_blocks = num_blocks

        # embedding layer
        self.embedding = torch.nn.Embedding(num_token_vals, num_emb)

        # positional embedding
        self.positional_embedding = nn.Embedding(num_tokens, num_emb)

        # transformer blocks
        self.transformer_blocks = nn.ModuleList()
        for _ in range(num_blocks):
            self.transformer_blocks.append(TransformerBlock(num_emb=num_emb, num_

        # output layer (logits + softmax)
        self.logits = nn.Sequential(nn.Linear(num_emb, num_token_vals))

        # dropout layer
        self.dropout = nn.Dropout(dropout_prob)

        # loss function
        self.loss_fun = LossFun()

    def transformer_forward(self, x, causal=True, temperature=1.0):
        # x: B(atch) x T(okens)
        # embedding of tokens
        x = self.embedding(x) # B x T x D
        # embedding of positions
        pos = torch.arange(0, x.shape[1], dtype=torch.long).unsqueeze(0).to(self.
        pos_emb = self.positional_embedding(pos)
        # dropout of embedding of inputs
        x = self.dropout(x + pos_emb)

        # transformer blocks
        for i in range(self.num_blocks):
            x = self.transformer_blocks[i](x)

        # output logits
        out = self.logits(x)

        return F.log_softmax(out/temperature, 2)

    @torch.no_grad()
    def sample(self, batch_size=4, temperature=1.0):
        x_seq = np.asarray([[self.num_token_vals - 1] for i in range(batch_size)]
```

```
        # sample next tokens
        for i in range(self.num_tokens-1):
            xx = torch.tensor(x_seq, dtype=torch.long, device=self.device)
            # process x and calculate log_softmax
            x_log_probs = self.transformer_forward(xx, temperature=temperature)
            # sample i-th tokens
            x_i_sample = torch.multinomial(torch.exp(x_log_probs[:,i]), 1).to(sel
            # update the batch with new samples
            x_seq = np.concatenate((x_seq, x_i_sample.to('cpu').detach().numpy())

        return x_seq

    @torch.no_grad()
    def top1_rec(self, x, causal=True):
        x_prob = torch.exp(self.transformer_forward(x, causal=True))[:,:-1,:].con
        _, x_rec_max = torch.max(x_prob, dim=2)
        return torch.sum(torch.mean((x_rec_max.float() == x[:,1:].float().to(devi

    def forward(self, x, causal=True, temperature=1.0, reduction='mean'):
        # get log-probabilities
        log_prob = self.transformer_forward(x, causal=causal, temperature=tempera

        return self.loss_fun(log_prob[:,:-1].contiguous(), x[:,1:].contiguous(),
```

## Evaluation and training functions

**Please DO NOT remove or modify them.**

```
# DO NOT REMOVE OR MODIFY
def evaluation(test_loader, name=None, model_best=None, epoch=None, device='cuda'
    # EVALUATION
    if model_best is None:
        # load best performing model
        model_best = torch.load(name + '.model').to(device)

    model_best.eval()
    loss = 0.
    rec = 1.
    N = 0.
    for indx_batch, test_batch in enumerate(test_loader):
        loss_t = model_best.forward(test_batch.to(device), reduction='sum')
        loss = loss + loss_t.item()

        rec_t = model_best.top1_rec(test_batch.to(device))
        rec = rec + rec_t.item()

        N = N + test_batch.shape[0]
    loss = loss / N
    rec = rec / N

    if epoch is None:
        print(f'FINAL LOSS: nll={loss}, rec={rec}')
    else:
        print(f'Epoch: {epoch}, val nll={loss}, val rec={rec}')

    return loss, rec

def plot_curve(name, nll_val, ylabel='nll'):
    plt.plot(np.arange(len(nll_val)), nll_val, linewidth='3')
    plt.xlabel('epochs')
    plt.ylabel(ylabel)
    plt.savefig(name + '_' + ylabel + '_val_curve.pdf', bbox_inches='tight')
    plt.close()
```

```
# DO NOT REMOVE OR MODIFY
def training(name, max_patience, num_epochs, model, optimizer, training_loader, v
    nll_val = []
    rec_val = []
    best_nll = 1000.
    patience = 0

    # Main loop
    for e in range(num_epochs):
        # TRAINING
        model.train()
        for indx_batch, batch in enumerate(training_loader):
            loss = model.forward(batch.to(device))

            optimizer.zero_grad()
            loss.backward(retain_graph=True)
            optimizer.step()

        # Validation
        loss_val, r_val = evaluation(val_loader, model_best=model, epoch=e, devic
        nll_val.append(loss_val)  # save for plotting
        rec_val.append(r_val)

        if e == 0:
            print('saved!')
            torch.save(model, name + '.model')
            best_nll = loss_val

            sampled_tokens = model.sample(batch_size=64, temperature=1.0)
            sampled_texts = tokenizer.decode(sampled_tokens)
            save_texts(sampled_texts, name='epoch_' + str(e))

        else:
            if loss_val < best_nll:
                print('saved!')
                torch.save(model, name + '.model')
                best_nll = loss_val
                patience = 0

                sampled_tokens = model.sample(batch_size=64, temperature=1.0)
                sampled_texts = tokenizer.decode(sampled_tokens)
                save_texts(sampled_texts, name='epoch_' + str(e))
            else:
                patience = patience + 1

        if patience > max_patience:
            break

    nll_val = np.asarray(nll_val)
    rec_val = np.asarray(rec_val)

    np.save(name + '_nll_val.npy', nll_val)
    np.save(name + '_rec_val.npy', rec_val)

    return nll_val, rec_val
```

## Setup

**NOTE:** ***Please comment your code! Especially if you introduce any new variables (e.g., hyperparameters).***

```
# DO NOT REMOVE OR MODIFY
dataprocessor = DataProcessor()
tokenizer = Tokenizer(max_length=149)
```

```
[nltk_data] Downloading package omw-1.4 to /root/nltk_data...
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
# PLEASE MODIFY ACCORDING TO THE REPORT REQUIREMENTS
num_training_data = None  # None to take all training data

# DO NOT REMOVE OR MODIFY THE REST OF THIS CELL
#-dataset
train_dataset = Headers(dataprocessor, tokenizer, num_training_data=num_training_
validation_dataset = Headers(dataprocessor, tokenizer, mode="val")
test_dataset = Headers(dataprocessor, tokenizer, mode="test")

#-dataloaders
BATCH_SIZE = 32

training_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(validation_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

```
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:89: Us
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tal
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access pu
  warnings.warn(
```

| | |
|---|---|
| Downloading data: 100% | 1.13M/1.13M [00:00<00:00, 4.77MB/s] |
| Downloading data: 100% | 148k/148k [00:00<00:00, 984kB/s] |
| Downloading data: 100% | 145k/145k [00:00<00:00, 1.71MB/s] |
| Generating train split: 100% | 4332/4332 [00:00<00:00, 55708.87 examples/s] |
| Generating test split: 100% | 542/542 [00:00<00:00, 14066.83 examples/s] |
| Generating validation split: 100% | 542/542 [00:00<00:00, 18935.12 examples/s] |

# 1. Model with $< 100$k weights

```
# DO NOT REMOVE (but you can modify if necessary)
#-creating a dir for saving results
name = 'arm_transformer_1'  # NOTE: if you run multiple experiments, you would ov
results_dir = results_model_dir + name + '/'
if not(os.path.exists(results_dir)):
  os.mkdir(results_dir)
```

In the next cell, please initialize the model. Please remember about commenting your code!

```
# DO NOT REMOVE but PLEASE MODIFY WHENEVER YOU ARE ASKED FOR IT!
# NOTE: in order to obtain required sizes of your models, you can play with
#       various values of num_neurons, num_heads, num_blocks, num_emb
num_tokens = 150 # do not modify!
num_token_vals = 29  # do not modify!
num_neurons = 10 # please modify it
num_heads = 5 # please modify it
num_blocks = 5 # please modify it
num_emb = num_heads * 4  # please modify it but it must be a multiplication of nu
causal=True # do not modify!

lr = 1e-3 # learning rate; do not modify!
num_epochs = 1000 # max. number of epochs; do not modify!
max_patience = 10 # an early stopping is used, if training doesn't improve for lo


# DO NOT REMOVE OR MODIFY
model = DecoderTransformer(num_tokens=num_tokens, num_token_vals=num_token_vals,
model = model.to(device)
# Print the summary (like in Keras)
print(summary(model, torch.zeros(1, num_tokens, dtype=torch.long).to(device), sho
```

```
----------------------------------------------------------------------------
         Layer (type)        Output Shape         Param #     Tr. Param #
============================================================================
          Embedding-1        [1, 150, 20]             580             580
          Embedding-2        [1, 150, 20]           3,000           3,000
            Dropout-3        [1, 150, 20]               0               0
   TransformerBlock-4        [1, 150, 20]          16,620          16,620
   TransformerBlock-5        [1, 150, 20]          16,620          16,620
   TransformerBlock-6        [1, 150, 20]          16,620          16,620
   TransformerBlock-7        [1, 150, 20]          16,620          16,620
   TransformerBlock-8        [1, 150, 20]          16,620          16,620
            Linear-9        [1, 150, 29]             609             609
          LossFun-10                  []               0               0
============================================================================
Total params: 87,289
Trainable params: 87,289
Non-trainable params: 0
----------------------------------------------------------------------------
```

Please initialize the optimizer

```
# DO NOT REMOVE OR MODIFY
optimizer = torch.optim.AdamW([p for p in model.parameters() if p.requires_grad =
```

## Training and final evaluation

In the following two cells, we run the training and the final evaluation.

```
# DO NOT REMOVE OR MODIFY
# Training procedure
nll_val, rec_val = training(name=results_dir + name, max_patience=max_patience, n
```

```
saved!
Epoch: 96, val nll=96.45654049130823, val rec=0.8073132082105123
Epoch: 97, val nll=96.29552974912073, val rec=0.808532908393888
Epoch: 98, val nll=96.37968484238066, val rec=0.8068860219413504
Epoch: 99, val nll=95.4801428537967, val rec=0.8094368420843708
saved!
Epoch: 100, val nll=95.90658085108683, val rec=0.8084524091319404
Epoch: 101, val nll=95.89756499005419, val rec=0.8083224067828751
Epoch: 102, val nll=95.83862259643104, val rec=0.808235720954698
Epoch: 103, val nll=96.46901463582508, val rec=0.806613593083906
Epoch: 104, val nll=96.12918822791744, val rec=0.8084276558288349
Epoch: 105, val nll=95.5480542622809, val rec=0.8090344038396744
Epoch: 106, val nll=95.68711396952837, val rec=0.8086381609589411
Epoch: 107, val nll=95.6164253488238, val rec=0.8087743639506098
Epoch: 108, val nll=95.56041039048085, val rec=0.8090282137543513
Epoch: 109, val nll=95.7491603724631, val rec=0.8085886279595295
Epoch: 110, val nll=95.56086060921645, val rec=0.8093935044489223
```

```
# DO NOT REMOVE OR MODIFY
# Final evaluation
test_loss, test_rec = evaluation(name=results_dir + name, test_loader=test_loader

with open(results_dir + name + '_test_loss.txt', "w") as f:
    f.write('Test NLL: ' + str(test_loss)+'\n'+'Test REC: ' + str(test_rec))
    f.close()

plot_curve(results_dir + name, nll_val, ylabel='nll')
plot_curve(results_dir + name, rec_val, ylabel='rec')
```

```
FINAL LOSS: nll=98.32098703982645, rec=0.8041184819492467
```

## 2. Model with ~ 500k weights

```
# DO NOT REMOVE (but you can modify if necessary)
#-creating a dir for saving results
name = 'arm_transformer_2'  # NOTE: if you run multiple experiments, you would ov
results_dir = results_model_dir + name + '/'
if not(os.path.exists(results_dir)):
  os.mkdir(results_dir)
```

```
# DO NOT REMOVE but PLEASE MODIFY WHENEVER YOU ARE ASKED FOR IT!
# NOTE: in order to obtain required sizes of your models, you can play with
#       various values of num_neurons, num_heads, num_blocks, num_emb
num_tokens = 150 # do not modify!
num_token_vals = 29  # do not modify!
num_neurons = 50 # please modify it
num_heads = 5 # please modify it
num_blocks = 5 # please modify it
num_emb = num_heads * 6  # please modify it but it must be a multiplication of nu
causal=True # do not modify!

lr = 1e-3 # learning rate; do not modify!
num_epochs = 1000 # max. number of epochs; do not modify!
max_patience = 10 # an early stopping is used, if training doesn't improve for lo


# DO NOT REMOVE OR MODIFY
model = DecoderTransformer(num_tokens=num_tokens, num_token_vals=num_token_vals,
model = model.to(device)
# Print the summary (like in Keras)
print(summary(model, torch.zeros(1, num_tokens, dtype=torch.long).to(device), sho
```

```
----------------------------------------------------------------------------
         Layer (type)        Output Shape         Param #     Tr. Param #
============================================================================
          Embedding-1        [1, 150, 30]             870             870
          Embedding-2        [1, 150, 30]           4,500           4,500
            Dropout-3        [1, 150, 30]               0               0
   TransformerBlock-4        [1, 150, 30]         110,130         110,130
   TransformerBlock-5        [1, 150, 30]         110,130         110,130
   TransformerBlock-6        [1, 150, 30]         110,130         110,130
   TransformerBlock-7        [1, 150, 30]         110,130         110,130
   TransformerBlock-8        [1, 150, 30]         110,130         110,130
             Linear-9        [1, 150, 29]             899             899
          LossFun-10                  []               0               0
============================================================================
Total params: 556,919
Trainable params: 556,919
Non-trainable params: 0
----------------------------------------------------------------------------
```

```
# DO NOT REMOVE OR MODIFY
optimizer = torch.optim.AdamW([p for p in model.parameters() if p.requires_grad =
```

## Training and final evaluation

```
# DO NOT REMOVE OR MODIFY
# Training procedure
nll_val, rec_val = training(name=results_dir + name, max_patience=max_patience, n
```

```
Epoch: 8, val nll=102.24514331324954, val rec=0.7945237892376l
saved!
Epoch: 9, val nll=99.93455049296587, val rec=0.797227516385462
saved!
Epoch: 10, val nll=97.98789296730858, val rec=0.800248895624027
saved!
Epoch: 11, val nll=96.2922288957997, val rec=0.8048119087500766
saved!
Epoch: 12, val nll=94.99479579573628, val rec=0.8067745634550538
saved!
Epoch: 13, val nll=93.18348294078643, val rec=0.8107493970666865
saved!
Epoch: 14, val nll=93.02144788903944, val rec=0.8108113207940246
saved!
Epoch: 15, val nll=91.35416493996483, val rec=0.8130525821249424
saved!
Epoch: 16, val nll=90.77333006911613, val rec=0.8153124242691097
saved!
Epoch: 17, val nll=89.7092974335505, val rec=0.8168974028302295
saved!
Epoch: 18, val nll=89.88874281992332, val rec=0.8172750736954467
Epoch: 19, val nll=89.3865966796875, val rec=0.8184761965846663
saved!
Epoch: 20, val nll=89.5373969834669, val rec=0.8184576175309635
Epoch: 21, val nll=88.74224605771448, val rec=0.8192934391683319
saved!
Epoch: 22, val nll=88.28618664055293, val rec=0.820426467599904
saved!
Epoch: 23, val nll=88.45318626037823, val rec=0.8209527321847162
Epoch: 24, val nll=87.85545433579335, val rec=0.8221352654629528
saved!
Epoch: 25, val nll=87.74500795484029, val rec=0.8227420169928857
saved!
Epoch: 26, val nll=88.55405777934733, val rec=0.8208288935277734
Epoch: 27, val nll=87.77143437572072, val rec=0.8233735447000313
Epoch: 28, val nll=87.7577721852658, val rec=0.8242093716160398
Epoch: 29, val nll=87.16487217301372, val rec=0.8239059923319799
saved!
Epoch: 30, val nll=87.65155952706988, val rec=0.8240483924471584
Epoch: 31, val nll=87.44397333011416, val rec=0.8246984833720865
Epoch: 32, val nll=87.96594080625864, val rec=0.8243517664525781
Epoch: 33, val nll=87.10902568071091, val rec=0.8266920955418661
saved!
Epoch: 34, val nll=87.41185307238814, val rec=0.8260667737559638
Epoch: 35, val nll=87.26403290583198, val rec=0.826531126930265
Epoch: 36, val nll=87.73077572755707, val rec=0.826308232831779
Epoch: 37, val nll=88.30753363014587, val rec=0.8251938028089235
Epoch: 38, val nll=87.61891743001902, val rec=0.8270449972680574
Epoch: 39, val nll=87.38125463957276, val rec=0.8271254859727247
Epoch: 40, val nll=87.34234574096229, val rec=0.8271316936535149
Epoch: 41, val nll=87.84327072762915, val rec=0.8265187414809787
Epoch: 42, val nll=89.06991948764703, val rec=0.825559084705761
Epoch: 43, val nll=88.52237400181619, val rec=0.8271750260103232
Epoch: 44, val nll=88.7671195730512, val rec=0.8257262486813253
```

```
# DO NOT REMOVE OR MODIFY
# Final evaluation
test_loss, test_rec = evaluation(name=results_dir + name, test_loader=test_loader

with open(results_dir + name + '_test_loss.txt', "w") as f:
    f.write('Test NLL: ' + str(test_loss)+'\n'+'Test REC: ' + str(test_rec))
    f.close()

plot_curve(results_dir + name, nll_val, ylabel='nll')
plot_curve(results_dir + name, rec_val, ylabel='rec')
```

```
FINAL LOSS: nll=90.95459997873904, rec=0.8196215946296045
```

## 3. Model with ~5M weights

```
# DO NOT REMOVE (but you can modify if necessary)
#-creating a dir for saving results
name = 'arm_transformer_3'  # NOTE: if you run multiple experiments, you would ov
results_dir = results_model_dir + name + '/'
if not(os.path.exists(results_dir)):
  os.mkdir(results_dir)


# DO NOT REMOVE but PLEASE MODIFY WHENEVER YOU ARE ASKED FOR IT!
# NOTE: in order to obtain required sizes of your models, you can play with
#       various values of num_neurons, num_heads, num_blocks, num_emb
num_tokens = 150 # do not modify!
num_token_vals = 29  # do not modify!
num_neurons = 170 # please modify it
num_heads = 6 # please modify it
num_blocks = 6 # please modify it
num_emb = num_heads * 8  # please modify it but it must be a multiplication of nu
causal=True # do not modify!

lr = 1e-3 # learning rate; do not modify!
num_epochs = 1000 # max. number of epochs; do not modify!
max_patience = 10 # an early stopping is used, if training doesn't improve for lo


# DO NOT REMOVE OR MODIFY
model = DecoderTransformer(num_tokens=num_tokens, num_token_vals=num_token_vals,
model = model.to(device)
# Print the summary (like in Keras)
print(summary(model, torch.zeros(1, num_tokens, dtype=torch.long).to(device), sho
```

```
----------------------------------------------------------------------------
         Layer (type)        Output Shape         Param #     Tr. Param #
============================================================================
          Embedding-1       [1, 150, 48]           1,392           1,392
          Embedding-2       [1, 150, 48]           7,200           7,200
            Dropout-3       [1, 150, 48]               0               0
   TransformerBlock-4       [1, 150, 48]         847,968         847,968
   TransformerBlock-5       [1, 150, 48]         847,968         847,968
```

```
   TransformerBlock-6        [1, 150, 48]         847,968         847,968
   TransformerBlock-7        [1, 150, 48]         847,968         847,968
   TransformerBlock-8        [1, 150, 48]         847,968         847,968
   TransformerBlock-9        [1, 150, 48]         847,968         847,968
            Linear-10        [1, 150, 29]           1,421           1,421
          LossFun-11                  []               0               0
================================================================================
Total params: 5,097,821
Trainable params: 5,097,821
Non-trainable params: 0
--------------------------------------------------------------------------------
```

```
# DO NOT REMOVE OR MODIFY
optimizer = torch.optim.AdamW([p for p in model.parameters() if p.requires_grad =
```

## Training and final evaluation

```
# DO NOT REMOVE OR MODIFY
# Training procedure
nll_val, rec_val = training(name=results_dir + name, max_patience=max_patience, n
```

```
Epoch: 0, val nll=164.21112353335448, val rec=0.6795549674227669
saved!
Epoch: 1, val nll=154.65675765034018, val rec=0.6898697416720795
saved!
Epoch: 2, val nll=146.3340523920376, val rec=0.7080413125097972
saved!
Epoch: 3, val nll=143.85513857457912, val rec=0.7078555765187169
saved!
Epoch: 4, val nll=142.64299253344095, val rec=0.7120161443618831
saved!
Epoch: 5, val nll=139.65478988591155, val rec=0.7171178110411246
saved!
Epoch: 6, val nll=137.57329376333314, val rec=0.7190990483188981
saved!
Epoch: 7, val nll=134.52048529030213, val rec=0.7247455480793745
saved!
Epoch: 8, val nll=130.9090227077808, val rec=0.7318903405727936
saved!
Epoch: 9, val nll=128.0126752677439, val rec=0.7368743692376957
saved!
Epoch: 10, val nll=125.02991668588561, val rec=0.7446878387479324
saved!
Epoch: 11, val nll=120.93778513102514, val rec=0.7512444640437616
saved!
Epoch: 12, val nll=115.03936452267355, val rec=0.7647106216402512
saved!
Epoch: 13, val nll=109.66218189647716, val rec=0.775192559865128
saved!
Epoch: 14, val nll=105.14335221294108, val rec=0.7856187679670834
saved!
Epoch: 15, val nll=100.45079215074378, val rec=0.7954072723529435
saved!
Epoch: 16, val nll=95.97542400289726, val rec=0.8047933420132007
saved!
```

```
Epoch: 17, val nll=94.68222777869869, val rec=0.8076104062069827
saved!
Epoch: 18, val nll=91.87958326726822, val rec=0.8117709863669758
saved!
Epoch: 19, val nll=91.35726016561924, val rec=0.8141855929610474
saved!
Epoch: 20, val nll=88.9220862582161, val rec=0.8173307967801815
saved!
Epoch: 21, val nll=88.02198887223247, val rec=0.8215347057778897
saved!
Epoch: 22, val nll=88.22539742670376, val rec=0.8197206500711476
Epoch: 23, val nll=87.30658049073166, val rec=0.8223272038561832
saved!
Epoch: 24, val nll=86.98972086888837, val rec=0.8224324634594231
saved!
Epoch: 25, val nll=87.0928520399706, val rec=0.8233921061582671
Epoch: 26, val nll=86.64992846949954, val rec=0.8249647186251144
saved!
Epoch: 27, val nll=87.20877581944765, val rec=0.8240112501756731
Epoch: 28, val nll=86.28976992223535, val rec=0.8269521301522906
saved!
Epoch: 29, val nll=87.08008082766375, val rec=0.8267787760914032
Epoch: 30, val nll=87.02102942660287, val rec=0.8261348682136114
Epoch: 31, val nll=87.42186666678678, val rec=0.8263630647142473
```

```
# DO NOT REMOVE OR MODIFY
# Final evaluation
test_loss, test_rec = evaluation(name=results_dir + name, test_loader=test_loader

with open(results_dir + name + '_test_loss.txt', "w") as f:
    f.write('Test NLL: ' + str(test_loss)+'\n'+'Test REC: ' + str(test_rec))
    f.close()

plot_curve(results_dir + name, nll_val, ylabel='nll')
plot_curve(results_dir + name, rec_val, ylabel='rec')
```

```
FINAL LOSS: nll=89.4695184573916, rec=0.8217266406520266
```

## 4. Model with $> 10$M weights

```
# DO NOT REMOVE (but you can modify if necessary)
#-creating a dir for saving results
name = 'arm_transformer_4'  # NOTE: if you run multiple experiments, you would ov
results_dir = results_model_dir + name + '/'
if not(os.path.exists(results_dir)):
  os.mkdir(results_dir)
```

```
# DO NOT REMOVE but PLEASE MODIFY WHENEVER YOU ARE ASKED FOR IT!
# NOTE: in order to obtain required sizes of your models, you can play with
#       various values of num_neurons, num_heads, num_blocks, num_emb
num_tokens = 150 # do not modify!
num_token_vals = 29  # do not modify!
num_neurons = 220 # please modify it
num_heads = 6 # please modify it
num_blocks = 6 # please modify it
num_emb = num_heads * 10  # please modify it but it must be a multiplication of n
causal=True # do not modify!

lr = 1e-3 # learning rate; do not modify!
num_epochs = 1000 # max. number of epochs; do not modify!
max_patience = 10 # an early stopping is used, if training doesn't improve for lo


# DO NOT REMOVE OR MODIFY
model = DecoderTransformer(num_tokens=num_tokens, num_token_vals=num_token_vals,
model = model.to(device)
# Print the summary (like in Keras)
print(summary(model, torch.zeros(1, num_tokens, dtype=torch.long).to(device), sho
```

```
----------------------------------------------------------------------------
         Layer (type)        Output Shape         Param #     Tr. Param #
============================================================================
          Embedding-1        [1, 150, 60]           1,740           1,740
          Embedding-2        [1, 150, 60]           9,000           9,000
            Dropout-3        [1, 150, 60]               0               0
   TransformerBlock-4        [1, 150, 60]       1,685,040       1,685,040
   TransformerBlock-5        [1, 150, 60]       1,685,040       1,685,040
   TransformerBlock-6        [1, 150, 60]       1,685,040       1,685,040
   TransformerBlock-7        [1, 150, 60]       1,685,040       1,685,040
   TransformerBlock-8        [1, 150, 60]       1,685,040       1,685,040
   TransformerBlock-9        [1, 150, 60]       1,685,040       1,685,040
          Linear-10        [1, 150, 29]           1,769           1,769
         LossFun-11                  []               0               0
============================================================================
Total params: 10,122,749
Trainable params: 10,122,749
Non-trainable params: 0
----------------------------------------------------------------------------
```

```
# DO NOT REMOVE OR MODIFY
optimizer = torch.optim.AdamW([p for p in model.parameters() if p.requires_grad =
```

## Training and final evaluation

```
# DO NOT REMOVE OR MODIFY
# Training procedure
nll_val, rec_val = training(name=results_dir + name, max_patience=max_patience, n
```

```
Epoch: 0, val nll=256.3558255016144, val rec=0.6396022687538964
saved!
Epoch: 1, val nll=163.57773987041628, val rec=0.6797902364132589
```

```
saved!
Epoch: 2, val nll=163.18333834827607, val rec=0.6797469005373571
saved!
Epoch: 3, val nll=166.29448539480512, val rec=0.6787748671105867
Epoch: 4, val nll=166.12159824723247, val rec=0.6787500979715607
Epoch: 5, val nll=167.2141653813999, val rec=0.6780009480859961
Epoch: 6, val nll=167.8267430379382, val rec=0.6780257137059286
Epoch: 7, val nll=167.93117396505997, val rec=0.6779947562411263
Epoch: 8, val nll=167.89954649302354, val rec=0.6779761824660635
Epoch: 9, val nll=168.20901996007265, val rec=0.6778647345370472
Epoch: 10, val nll=168.17825643954683, val rec=0.677870926381917
Epoch: 11, val nll=168.26646012309732, val rec=0.6777594854910876
Epoch: 12, val nll=168.19519718634686, val rec=0.6777780539875101
Epoch: 13, val nll=168.17675646116814, val rec=0.6777594819719941
```

```
# DO NOT REMOVE OR MODIFY
# Final evaluation
test_loss, test_rec = evaluation(name=results_dir + name, test_loader=test_loader

with open(results_dir + name + '_test_loss.txt', "w") as f:
    f.write('Test NLL: ' + str(test_loss)+'\n'+'Test REC: ' + str(test_rec))
    f.close()

plot_curve(results_dir + name, nll_val, ylabel='nll')
plot_curve(results_dir + name, rec_val, ylabel='rec')
```

```
FINAL LOSS: nll=166.6454686238757, rec=0.6712338211791542
```

## Final sampled texts

```
# DO NOT REMOVE
# Sample texts: load best model
names = ['arm_transformer_1', 'arm_transformer_2', 'arm_transformer_3','arm_trans

# sample
temperature = 1.0 # you can modify it
num_samples = 64 # you can modify it

for name in names:
    results_dir = results_model_dir + name + '/'
    model_best = torch.load(results_dir + name + '.model')
    model_best = model_best.eval()

    sampled_tokens = model_best.sample(batch_size=num_samples, temperature=temper
    sampled_texts = tokenizer.decode(sampled_tokens)  # do not modify

    save_texts(sampled_texts, name='FINAL_' + str(temperature))
```

## **5.** Model with $\sim 5M$ weights and 1000 test data

```
# PLEASE MODIFY ACCORDING TO THE REPORT REQUIREMENTS
num_training_data = 1000  # None to take all training data

# DO NOT REMOVE OR MODIFY THE REST OF THIS CELL
#-dataset
train_dataset = Headers(dataprocessor, tokenizer, num_training_data=num_training_
validation_dataset = Headers(dataprocessor, tokenizer, mode="val")
test_dataset = Headers(dataprocessor, tokenizer, mode="test")

#-dataloaders
BATCH_SIZE = 32

training_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(validation_dataset, batch_size=BATCH_SIZE, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)


# DO NOT REMOVE (but you can modify if necessary)
#-creating a dir for saving results
name = 'arm_transformer_1000'  # NOTE: if you run multiple experiments, you would
results_dir = results_model_dir + name + '/'
if not(os.path.exists(results_dir)):
  os.mkdir(results_dir)


# DO NOT REMOVE but PLEASE MODIFY WHENEVER YOU ARE ASKED FOR IT!
# NOTE: in order to obtain required sizes of your models, you can play with
#       various values of num_neurons, num_heads, num_blocks, num_emb
num_tokens = 150 # do not modify!
num_token_vals = 29  # do not modify!
num_neurons = 170 # please modify it
num_heads = 6 # please modify it
num_blocks = 6 # please modify it
num_emb = num_heads * 8  # please modify it but it must be a multiplication of num_
causal=True # do not modify!

lr = 1e-3 # learning rate; do not modify!
num_epochs = 1000 # max. number of epochs; do not modify!
max_patience = 10 # an early stopping is used, if training doesn't improve for long


# DO NOT REMOVE OR MODIFY
model = DecoderTransformer(num_tokens=num_tokens, num_token_vals=num_token_vals, nu
model = model.to(device)
# Print the summary (like in Keras)
print(summary(model, torch.zeros(1, num_tokens, dtype=torch.long).to(device), show_
```

```
----------------------------------------------------------------------
      Layer (type)        Output Shape         Param #     Tr. Param #
```