Creating and analyzing a combined chatbot-image captioning transformer

Project 1 WS 2022/23

Michaela Stolz

Supervisor: Anja Meunier

January 25, 2023

Contents

1 Introduction

In recent years, transformer-based language models have revolutionized the area of natural language processing. Language models are now widely used tools for text generation, information retrieval and as general purpose chatbots, and their generation capabilities are also interesting for assisted communication research. One area of interest is the treatment of Broca's aphasia, which impairs patients' ability to express themselves verbally. By combining brain-computer interfaces with chatbot models, it might one day be possible to translate patients' thoughts into language. This project serves as a preliminary exploration of a possible language BCI architecture, but substitutes brain data with images.

The goal of this project is to combine a chatbot and image captioning transformer into a single model with the ability to answer questions based on images supplied as context. As an additional requirement, the influence of the chatbot and the image captioning model should be a fixed or learnable model parameter. In order to reduce the amount of required training data, existing pre-trained models are combined to solve the task. Different configurations are evaluated with the perplexity metric, investigating the quality of the generated output.

2 Large-scale language models

Neural networks have been very successful as language models, predicting an output sequence from an input sequence of words. They fall under the umbrella of sequence-to-sequence models, a type of neural network architecture used for tasks that involve variable-length sequences as their input and output. These models are prominent in most language tasks such as question answering, machine translation, text completion etc. The prototypical architecture is two-part and includes an encoder as well as a decoder neural network. An input sequence, e.g. a sequence of words, is first encoded into a fixed-sized vector representation by the encoder architecture, serving as input to the decoder model, which produces the output sequence.

2.1 Attention

Before the introduction of the transformer architecture, recurrent or convolutional neural networks were used as the encoder and decoder in sequence-to-sequence models for language modelling tasks.[1] These architectures have mostly been replaced by the transformer, which is entirely based on attention instead of recurrence.

As a precursor to transformer models, attention was first applied to the context vector in between the encoder and decoder. [2]. While previous encoder-decoder

models encoded the output of the encoder into a vector of fixed length, which was a bottleneck for performance, the attention mechanism circumvents this problem. Due to the fixed-length vector representation, the performance of the classical encoder-decoder architecture declines rapidly with an increased length of the input sentence. As a replacement of this fixed-length vector, Bahdanau et al. proposed an attention mechanism as an interface between the encoder and decoder. This takes the form of an alignment model inserted between the encoder and decoder, scoring how important the inputs at a certain position are for outputs at a certain position, e.g. matching words in different languages in a translation task. The matches do not need to be exact, since context words are also important for correct translation and words in text in general are interdependent. [2]

The transformer architecture goes one step further and replaces RNNs, often used as encoders and decoders in sequence-to-sequence-models, with attention-based non-recurrent networks. Instead of alignments between the in- and output, the self-attention used by the transformer models the relation between different parts of a sequence itself. The advantages of self-attention compared to recurrent and convolutional layers are the decreased amount of computational complexity, the increased amount of parallelization and the increased ability to deal with long-range dependencies. Attention is also an intuitive concept and might lead to more interpretability.[1]

2.2 Transformer architecture

The transformer consists of an encoder and a decoder, each of which has six identical layers with multiple sublayers. Each layer in the encoder contains a multi-head self-attention mechanism and a position-wise fully connected feed-forward network. In addition, a residual connection and a layer of normalization are employed after each encoder sublayer. The decoder layer is constructed similarly but contains a third sublayer performing multi-head attention between the other two sublayers. In addition, the multi-head self attention is masked in order to prevent the model from having information about future parts of the sequence.

Self-attention means that each layer has access to to all positions in the previous layers. The attention mechanism is described as a mapping of a query and a set of key-value pairs to an output, which is a weighted sum of the values, the weight assigned to each value being computed by a compatibility function of the query with the corresponding key. The attention used is scaled-dot-product attention: Here, the dot products of the queries with all keys are computed and divided by the square root of the keys' and values' dimension and the softmax function is applied to obtain the weights on the values. This can be done on a set of queries simultaneously. Cross-attention between the encoder and decoder, as described in the previous section, is analogous but computes the alignments between two different embeddings instead of an ambedding with itself. Multi-

head attention extends the simple attention mechanisms by linearly projecting the queries, keys and values multiple times with different learned projections to a specific set of dimensions. On each projected version, the attention function is performed in parallel, the outputs of which are concatenated and again projected for the final output.

In summary, there are two types of attention used in the transformer: Attention between the output layers of the encoder and the decoder layers and self-attention in the encoder and decoder layers. Before passing an input to a transformer model, it needs to be converted to a numerical representation. In the case of language models, input sentences are first converted to a one-hot encoding and later to word embeddings. After passing through the transformer, an output token is determined from the outputs of the final linear layer, which serves as the context for the next pass through the decoder, auto-regressively generating the next token. The start and end of the sentence are signaled by special tokens.

2.3 Generative pre-trained transformer

GPT is a language model by OpenAI first released in 2018, while the subsequent model GPT-2 is a scale-up of the original architecture. GPT stands for generative pre-trained transformer and can be pre-trained on a diverse unlabeled text corpus and later fine-tuned with labeled training data. With this semi-supervised approach, the model first learns a universal representation of language which is later adapted to more specific tasks with a small amount of training data. Compared to immediately training with a labeled corpus, pre-training can act as a regularizer and lead to improved the generalisability. In addition, unlabeled text is widely available, while labeled training corpora for specific tasks or domains are scarce. Pre-training is done by maximizing the probability of a transformer decoder to predict the next tokens in the context window of a sequence. Only the transformer decoder part is used, which performs better on longer input sequences compared to RNN and transformer encoder-decoder models.[3] The authors assume that redundant information about a language is learned in the encoder and decoder (in monolingual tasks) and that a decoder-only architecture allows easier optimization, which they believe to be reflected in increased quality of longer sequences. At inference, the input sequence is provided initially and the output is generated auto-regressively. [4]

By applying the transformer architecture to images with as little modification as possible, Dosovitskiy et al. have created a model called the VisionTransformer or ViT. For large training datasets, the ViT approached the state of the art at the time of its release is competitive with convolution-based neural networks like ResNets. The architecture is intentionally kept as close as possible to the original transformer, substituting the word embedding of the standard encoder with an image, which is transformed into a sequence of flattened 2D patches. Instead of applying this transformation to a raw input image, it e can also be

applied to a CNN feature map. Like GPT, ViT is pre-trained on large datasets and then fine-tuned for downstream task.

[5]

2.4 Performance metrics

A language model estimates the probability distribution of a sequence of tokens such as words, characters or sound. One standard metric to evaluate language models is perplexity.[6]

$$\text{ppl}(o) = \exp(-\frac{1}{t}\sum_{i=1}^{t}\log(P(o_i|o_{i-1}))) \tag{1}$$

Perplexity is related to cross-entropy and depends on the total probability of predicting a sequence of words correctly. Since the inverse is taken, smaller perplexity values indicate better model quality. The values are normalized by sequence length, but are not comparable between different models types, depending on factors such as vocabulary length and pre-processing type. Another disadvantage of perplexity is that it only evaluates the probability distribution rather than the actual generated output sentences. To my knowledge, the only quality measurement for the actual output would be evaluation by humans, which is both time-consuming and prone to biases and disagreement. Besides perplexity and similar measurements, language models are also evaluated on downstream natural language tasks, but this way of evaluation is not applicable to this project, since the created architecture is not a general-purpose language model.

3 Model architecture

3.1 Pretrained models

The architectures used are DialoGPT, a version of GPT-2 trained on dialogues, as well as VisionTransformer, an image captioning model also based on GPT-2. While DialoGPT is a decoder-only architecture, ViT consists of an encoder for processing the input image and GPT-2 is used as a decoder to generate the caption. The reason for choosing these architectures is their shared use of the GPT-2 tokenizer, as well as the existence of pre-trained models of high quality.

DialoGPT was trained on 147M multi-turn discussions from Reddit. The training set and model are chosen deliberately to capture The sentences in the dialogue are concatenated with end-of-text tokens as separators signaling turn-taking, which is referred to as the "dialogue history". A dialogue is modelled as a long text, where maximizing the conditional probability of an output turn equals the product of conditional probabilities of each turn depending on the previous turn. In addition, a mutual information maximization scoring function is used to prevent the generation of bland and uninformative samples. Top-K sampling is used to generate a set of hypothesis and the conditional probability of the source depending on the hypothesis is then used to rerank the hyptheses. This backwards prediction penalizes bland sentences, as hypotheses which occur frequently are associated with many different sources. [7] There is no information about the training dataset of the image captioner but examples for testing are taken from the COCO dataset.[1] Both models are loaded with huggingface's transformers library.

3.2 Version 1: Fixed parameter alpha

The simplest combination of the two models would be a fixed parameter set by the user before generating a sequence. The generation of a single word can be described as a classification task, selecting the correct token among all subwords in the dictionary. A language transformer's output is transformed into a probability distribution by applying the softmax function, and the output selected by choosing the token with the highest probability, sampling from the distribution, or another strategy. Since DialoGPT and ViT use the same vocabulary, a weighted average of the two models' probability distributions results in another probability distribution, and a token can be selected from the combined distribution.

$$\text{score}_{\text{ combined}} = \alpha \cdot \text{score}_{\text{ chatbot}} + (1 - \alpha) \cdot \text{score}_{\text{ captioner}} \tag{2}$$

To repeat the process, the output token is concatenated to the chatbot's and captioner's input, auto-regressively generating the whole sequence until a stop token is encountered. To find the optimal parameter for this model combination,

[1] https://huggingface.co/ydshieh/vit-gpt2-coco-en-ckpts

different parameters values are evaluated with the perplexity metric and a grid search is performed.

3.3 Version 2: Linear layer

While a single fixed weight parameter is easy to implement, it might be too restrictive to achieve good performance. It is possible that some tokens in the target distribution are more accurately predicted by the chatbot, while others are more in line with the captioner's output probabilities, a distinction not captured by the single parameter model.

In order to create a more flexible model, the architecture described in the previous section is adapted to multiple parameters. Each token in the vocabulary is assigned to one of the parameters, determining the weight of the chatbot's and captioner's output probabilities during the generation process. In the most extreme case, each token in the model's vocabulary would be assigned a separate parameter. Such a model would be equivalent to a sparse linear neural network layer with a connection from each neuron to a neuron in the chatbot's output layer, and a second neuron in the captioner's output layer. As an additional constraint, the weights connected to each output neuron sum up to 1, since they represent the proportion of the output that is determined by the chatbot versus the captioner for a single token.

Since the number of possible combinations of parameters is very high for a dictionary of 50257 entries, it would take too much time to determine the optimal parameters by hand. Instead, the parameters are found with machine learning methods, by training the model on a manually crafted dataset of questions, answers and images. However, due to the limited size of the dataset, the training is likely to result in overfitting. As a countermeasure, the number of parameters could be reduced by grouping tokens together and assigning a common parameter to each group. This leaves us with the challenge of finding meaningful ways to group the vocabulary items. For example, an ideal model combination might weight content words more heavily when they come from the image captioner, whereas functional words, which define the sentence structure, might be assigned more importance when produced by the chatbot. Not only does the grouping need to be meaningful, it also needs to be applicable to the subword tokens produced by the transformers' tokenizer.

In summary, there are three possible ways of assigning the parameters to tokens: Either by assigning a global parameter to all tokens, assigning a single parameter to each token or to predefined groups of tokens. While the architecture can be conceptualized as a linear layer, the need to pre-group the vocabulary items introduces more flexibility.

4 Implementation

4.1 Generation function

4.1.1 Library generation function

As a first prototype, the generation method of the transformers library is used. While the library provides a few different generation modes, greedy search is used, since it returns only one token with maximum score, which can easily be emulated in the combined model. In order to obtain the output probabilities, the generate function is run with the output parameters set to return the score information, which is the raw logits output before applying any modifications such as the softmax function. Only the score for the next token is obtained from both models, a vector the size of the tokenizer's vocabularies. After obtaining the logits from both models, the softmax function is applied to each and they are weighted with the parameter alpha and summed up to receive the combined score:

$$\text{score}_{\text{combined}} = \alpha \cdot \text{score}_{\text{chatbot}} + (1 - \alpha) \cdot \text{score}_{\text{captioner}}$$

The maximum value of the combined score determines the next token, which is appended to the input for the next forward pass through the decoder, unless an end token is created. Since unchanged parameters have to be set each time the generation function is called, this process is somewhat computationally wasteful but allows for a lot of flexibility by easily changing parameters or the search method.

4.1.2 Custom generation function

Besides the use of the library generation function, I also created a custom generation function which does not reset parameters and generation configurations after each forward pass. Based heavily on huggingface's library generation function, it pre-processes the input question and image before calling a sub-function depending on the token selection strategy. One challenge during this project stage was reproducing huggingface's way of generation, as it is encapsulated in a general-purpose library for different types of transformers with many configuration options. Despite the available documentation, it was necessary to look at the source code in detail.

Currently, greedy search and sampling are implemented as token selection strategies. Both proceed in a similar way, by passing the input to the models, combining the results and selecting the desired token, and appending it to the generated sequence to receive the input for the next iteration. Aside from small simplifications and the missing preprocessing re-computation, the process is almost identical to the one described in the previous section.

4.1.3 Configuration options

Configuration options for the combined weighted model include the use of different logit processors and different search methods. LogitsProcessor[2] is a class used by the transformers library to modify the prediction score and change aspects of the generation, such as setting the temperature, adding a penalty for repeated tokens or n-grams or constraints on the length of the output. Different search methods for selecting the next output tokens are available as well. The current version uses greedy search, preserving only the token with the maximum score at each step in the generation process. However, the transformers library also supports other generation methods such as contrastive search, multinomial sampling and beam search.

4.2 Custom models

I implemented the three possibilities of assigning tokens to parameters as three different machine learning models called SingleParameterModel, MultiParameterModel and AllParametersModel. The implementation was done in the PyTorch framework, as the pre-trained models are loaded with PyTorch as well.

While PyTorch provides support for densely connected linear layers out of the box, I am not aware of any standard implementation of sparse linear layers. In addition, the single and multi-parameter models require weight-sharing. Instead of using predefined layers, I chose to represent the model weights as a Parameter object, a subclass of the PyTorch tensor which is treated as a machine learning parameter. Any operations involving the Parameter are recorded by PyTorch's automatic differentiation engine and later used to compute gradients for the model.

The linear layer is implemented as an n-dimensional vector stored as a PyTorch parameter, each entry representing a distinct alpha. The weight-sharing between different tokens in the SingleParameterModel and the MultiParameterModel is implemented by storing each distinct alpha parameter only once and reusing it in the output computation during the model forward pass. The dependencies between the chatbot and captioner weights can be considered a type of weight-sharing too, as the captioner weights can be calculated by subtracting the chatbot weights from an equally large vector of 1s. The length of the PyTorch parameter vector depends on the model type: Although the SingleParameeterModel requires only a one-dimensional parameter, the length of the MultiParameterModel's weights parameter is equal to the number of token groups and for the AllParametersModel, it corresponds to the tokenizer's vocabulary size.

The model combinations are wrapped in custom PyTorch module subclasses.

[2] https://huggingface.co/docs/transformers/v4.24.0/en/internal/generation_utils#transformers.LogitsProcessor

During the forward pass, the models receive an input-tuple containing the tokenized and preprocessed question and image features. After passing the inputs to the chatbot and captioner models, the outputs are multiplied with the weights parameter to obtain the final result. Due to the weight-sharing, the SingleParameterModel and MultiParameterModel first need to duplicate the values before multiplying the resulting tensor with the chatbot and captioner output. This is done by duplicating the value and storing it in a tensor the size of the vocabulary size.

For the MultiParameterModel, the process is a bit more complicated: Since the grouping of the tokens is arbitrary, it needs to be passed to the model during the constructor call. The token assignment information is stored in a matrix of the shape [number of groups x number of vocabulary items]. Each row contains a mask consisting of 0 and 1 entries, with 1 indicating that the vocabulary item is part of the current group. Each token must be assigned to one and only one group. During the forward pass, the dot product of the weights tensor with the mask matrix results in the individual weight for each token. After that, the calculation proceeds in the same way as in the other models: The chatbot output is multiplied with the weights, while the captioner output is multiplied with 1 - the weights and the results are combined.

4.3 Training dataset

A manually crafted dataset of 450 question, answers and images is used for training and evaluating the models. The required input format for the models is obtained by tokenizing the questions and answers and extracting the image features with a pre-trained model. It seems that all information stored in the images is preserved, since the output feature size of the network exceeds the size of the images in the dataset.

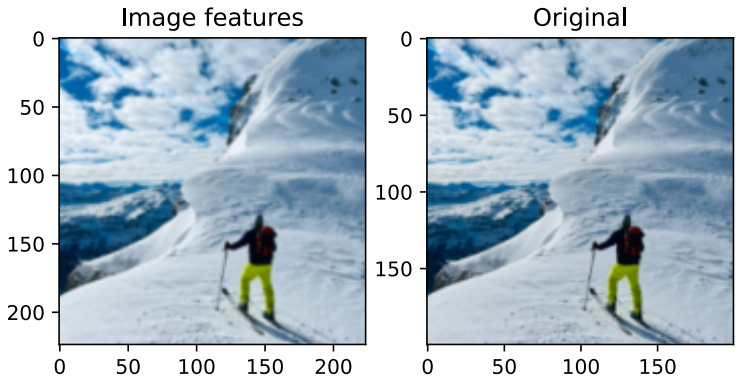

Figure 1: 224x224 image features, scaled to values between 0 and 1, compared to the original 200x200 image.

In order to accelerate the training process, the output of the chatbot and image captioning models is pre-computed and stored in a file. As the training does not fine-tune these pre-trained models and only adjusts the weights of the custom linear layer, re-computing the chatbot and captioner's forward passes in each epoch is not necessary. There are two different ways of pre-computing the dataset: It is possible to use the entire question and answer as an input sequence, as the transformer decoder's forward pass computes the output for each point in the sequence simultaneously. This does not affect the correctness of the output, since the decoder's causal mask prevents access to information about future tokens in the sequence.

Alternatively, the sequence can be generated completely auto-regressively by only feeding the decoder the sequence up to the token to be predicted. Teacher forcing, i.e. replacing the actually generated token with the correct one in each step, ensures that the model receives the correct input. This method is slower, because it requires multiple forward passes rather than just one, but it is the only method available when predicting new sequences rather than training data. For purposes of efficiency, huggingface's transformer implementation provides a parameter to pass past key values used by the attention computation to the model in each step of the generation process.

In theory, the two methods should be equivalent, but in practise, I found small differences of about e^{-8} to e^{-14} in many entries of the output tensors. Interestingly, the differences are only affected by the tensor's length rather than its entries after the current token. The differences affect the output systematically and lead to a different distribution of perplexities. For a comparison refer to the diagrams in section 5.2. At this point, the reason for this difference remains unknown and all training was done on data pre-computed on full input sequences.

4.4 Training process

The training process was the main challenge of the project due to the constraints, since finding errors and reasons for failure in machine learning processes can be difficult. As a first sanity check, I trained a single parameter model to reproduce the optimal alpha parameter found by grid search with perplexity. Adam was used as an optimizer with a learning rate of 0.001, and weight clamping between 0 and 1 was applied after each weight update. Categorical cross entropy and negative log likelihood were tried as loss functions, but neither arrived at the optimal parameter. When using weight clamping, the parameter always converged to 1 and otherwise increased indefinitely or jumped between very high values.

My first assumption was that the gradients were not computed correctly by PyTorch's automatic differentiation engine, as the loss only depends on one en-

try of the parameter variable, while the predefined linear layers usually have one weight per connection. Unfortunately, I was not able to check how exactly the gradients are computed, since it would have taken too much time to understand the library in detail. Visualizing the computation graph with the PyViz library did not yield any insights either. As a workaround, I computed and updated the gradients manually but found no changes in the training results.

When examining the results of the perplexity grid search more closely, I noticed that the location of the optimum depends on the normalization of the perplexity values. The perplexity computation consists of calculating the conditional probabilities of each token depending on the previous tokens, multiplying them, inverting the product and normalizing the results by taking the n-th square root, n being the number of tokens in the sequence. Without normalization, the optimal alpha parameter was almost 1, suggesting that the normalization skews the perplexity curve in the direction of the captioner. A possible reason is the chatbot's better ability to process short sentences correctly, which are weighted more strongly without normalization. As shown in table 1, it is likely that the

alpha	added prob.	multiplied prob.	perplexities
0	0.848	8.891e-14	6966.802
0.1	0.879	9.013e-09	162.563
0.2	0.910	1.998e-08	126.214
0.3	0.941	3.176e-08	110.111
0.4	0.972	4.334e-08	101.310
0.5	1.003	5.379e-08	96.405
0.6	1.034	6.226e-08	<u>94.209</u>
0.7	1.065	6.803e-08	94.476
0.8	1.096	7.080e-08	97.853
0.9	1.127	<u>7.093e-08</u>	107.470
1	<u>1.158</u>	6.994e-08	381.031

Table 1: Averaged sums and products of the conditional probabilities as well as averaged perplexities. Underlined values mark the column optimum. Note that the optimal value is a maximum for the conditional probabilities, but a minimum for the perplexities.

perplexity optimum can only be found by calculating the loss on whole sentences (as training on single tokens is equivalent to added rather than multiplied probabilities) and normalizing by sentence length. To circumvent this problem, I replaced the standard loss function by perplexity and added a manual gradient computation.

$$\text{ppl}(o) = exp(-\frac{1}{t}\sum_{i=1}^{t}\log(\text{P}(o_i|o_{i-1}))) =$$

$$\exp(-\frac{1}{t}\sum_{i=1}^{t}\log(\mathrm{P}(ch_i|o_{i-1}\cdot\alpha+\mathrm{P}(ca_i|o_{i-1})\cdot(1-\alpha)))$$

$$\frac{\partial \mathrm{ppl}}{\partial\alpha}=\exp(-\frac{1}{t}\sum_{i=1}^{t}\log(\mathrm{P}(o_i|o_{i-1})))\cdot(-\frac{1}{t})\sum_{i=1}^{t}(\frac{1}{\mathrm{P}(o_i|o_{i-1})}\cdot(\mathrm{P}(ch_i|o_{i-1})-\mathrm{P}(ca_i|o_{i-1}))) \tag{3}$$

Equation 3 computes the derivative of the perplexity with respect to the weights alpha and can be used directly to compute the gradient. The combined outputs at the i-th place in the sequence are denoted as o_i, and the chatbot's and captioner's individual outputs as ch_i and ca_i respectively. In case of a multi-dimensional weights parameter, the partial derivative with respect to each entry of the weights vector needs to be computed separately. In practise, the components of single output tokens

$$\frac{\mathrm{P}(ch_i|o_{i-1})-\mathrm{P}(ca_i|o_{i-1})}{\mathrm{P}(o_i|o_{i-1})}$$

are computed in the training loop, while the summation of the components depends on the model type and are implemented as a function inside the model classes. There might be a way to combine PyTorch optimizers with custom gradients, but I did not find a way to do so. For that reason, stochastic gradient descent is used instead of a more sophisticated optimization method such as Adam. This method was able to reproduce the one-dimensional alpha with optimal perplexity found with grid search and also showed plausible results when training the MultiParameterModel and the AllParametersModel. All models were trained with 5-fold cross validation for at least 30 epochs. Before the training process, the dataset was shuffled and all weights were initialized with a value of 0.5.

5 Results

5.1 Combined generation method

The fixed parameter generation method behaves as expected: Setting the parameter alpha to 0 or 1 produces the same output as the individual models, while values in between result in a combination of both. For demonstration purposes, the generation was run with values for alpha between 0 and 1 in increments of 0.1 on two images which are shown below. It seems that the image captioning model is strongly biased to start an image description with "a" and repetition is also common, which could be addressed by setting a repetition penalty in the logits processor.

Figure 2: "What is your favorite animal?"

alpha	answer
0 - 0.1	a cat laying on a blanket next to a cat laying on a bed
0.2 - 0.3	a cat laying on a blanket next to a cat laying on a blanket
0.4 - 0.7	a cat
0.8	I don't know, I don't have one.
0.9 - 1	I don't have one.

Figure 3: "What do you like to do in your free time?"

alpha	answer
0 - 0.3	a man riding skis down a snow covered slope
0.4 - 0.5	a lot of people on skis in the snow
0.6	a lot of people on skype talk about their free time irl irl irl irl irl...
0.7	a lot of people on here like to ski ive been skiing for a while ive been doing some tricks...
0.8	I like to ski, but I don't have a car.
0.9	I like to play video games, but I don't have a lot of time for that.
1	I like to play video games, but I also like to watch movies and read.

5.2 Perplexity evaluation

The results of the combined generation method are evaluated by calculating the perplexity per sentence for all samples in the dataset. To find the optimal parameter, a grid search is performed, investigating the values between 0 and 1 in steps of 0.1.

As shown in figure 4, combinations of the models have lower perplexities than the captioner (alpha = 0) or chatbot (alpha = 1) model alone. Furthermore, the chatbot's output probabilities are closer to the target distribution than the captioner's, which performs by far the worst. A possible reason for the worse performance of the captioner is the task it was trained on: The chatbot's training task of answering questions and holding a conversation produces structurally similar output to the target answers, especially at the beginning of an answer, when the image content is not yet relevant. On the other hand, image captions have a completely different structure, and the captioner does not receive the question as input, making it impossible to produce the correct structure at the beginning of a sentence. The perplexity reaches it's minimum at an alpha of about 0.6, with the weight slightly shifted in favor of the chatbot.

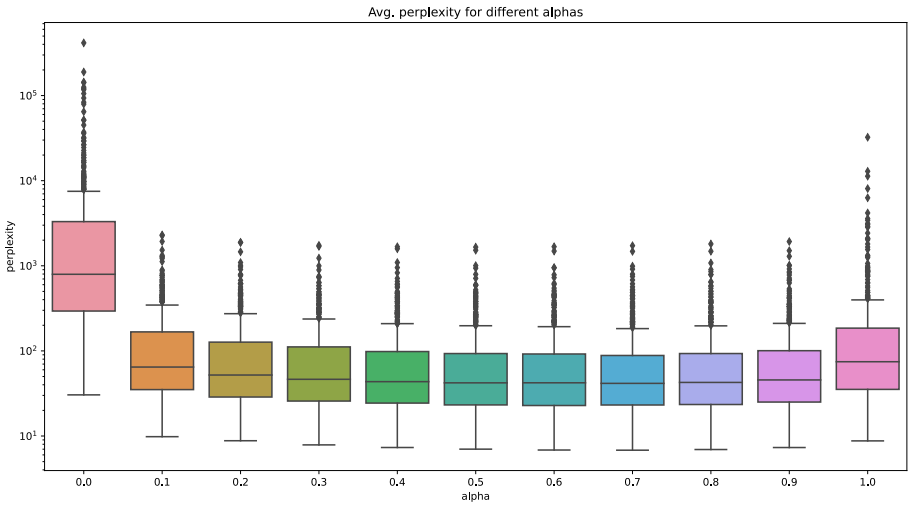

Figure 4: Perplexities at different alpha values calculated on the dataset pre-computed on entire sentences

When the dataset is pre-computed differently, the perplexities follow a different distribution as shown in figure 5. Unlike in figure 4, the dataset was pre-computed incrementally, feeding only the beginning of the sequence up to the current token to the models. More details about the generation process can be found in 4.3. On this dataset, the perplexity decrease until an alpha of 0.9, with a slight rise in perplexity in the chatbot only version with an alpha

of 1. The reason for this difference is unclear, and all other analyses and diagrams are based on the dataset pre-computed on full sequences rather than the incrementally pre-computed dataset.

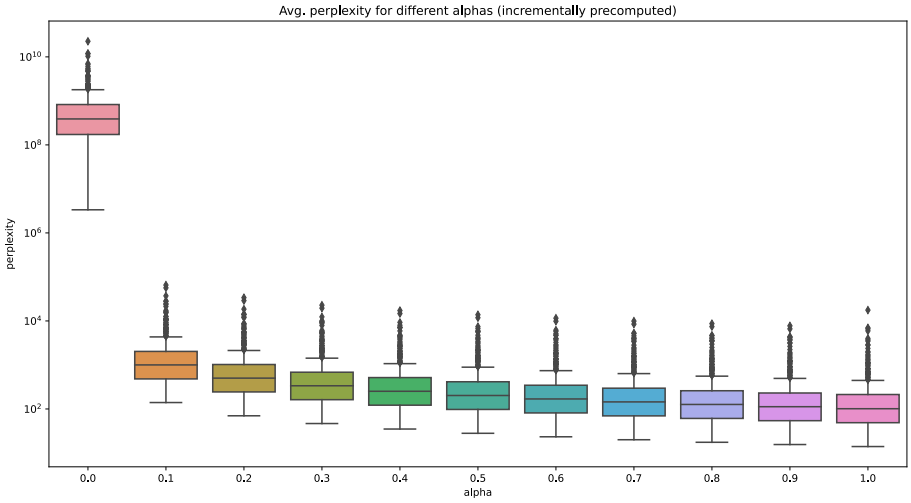

Figure 5: Perplexities at different alpha values calculated on the dataset pre-computed on incrementally generated sentences

To gain an intuition about extreme perplexity values, the questions and answers with the three lowest and highest perplexities are shown for each alpha. The most extreme perplexities were similar between versions with different alphas, with alphas from 0.2 - 0.5 and 0.6 - 0.8 being identical between the two versions.

The samples with the lowest perplexities were all very long sentences with common phrases, which presumably makes it easier to predict the next token from the previous context. Answers with references to objects or scenarios (e.g. "donuts") in the image seemed to be slightly easier to predict by the captioner model, while the chatbot seems to lean more towards common non-phrasal verbs.

The highest perplexities seem to occur for very short answer sentences, as the models have less context to base their predictions on. The pre-training tasks of the models are also apparent in the sentence structure: The chatbot considers sentences starting with "A" very unlikely, while this is the most preferred output token for the captioner. In fact, the most unlikely answer for the captioner was "Blue". The captioner likely predicted "a" as the sentence start with a very high probability, and due to the one-word-answer, additional tokens could not compensate for the low probability of the first expected token. In comparison, the chatbot seems to be more flexible in regard to the start token.

alpha	lowest perplexity question + answer combinations
0	What would you like to learn? I would like to know how to play the piano. If you could only eat one thing for the rest of your life, what would it be? If I had to choose only one thing, it would have to be donuts. When you look out the window, what do you see? I can see a cat and a dog fighting.
0.1	What would you like to learn? I would like to know how to play the piano. What would you like to learn? I would like to learn how to cook. If you could only eat one thing for the rest of your life, what would it be? If I had to choose only one thing, it would have to be donuts.
0.2 - 0.8	What would you like to learn? I would like to learn how to cook. What would you like to learn? I would like to know how to play the piano. What job did you want to do when you were growing up? I wanted to be a firefighter.
0.9	What would you like to learn? I would like to learn how to cook. What job did you want to do when you were growing up? I wanted to be a firefighter. What would you like to learn? I would like to learn how to draw.
1	What would you like to learn? I would like to learn how to cook. What would you like to learn? I would like to learn how to draw. Where do you like to go on holiday? I like to go to the beach.

Table 2: Lowest perplexity samples

alpha	highest perplexity question + answer combinations
0	What is your favorite color? Blue. What did you study in university? I studied violin. Where do you like to go on holiday? My balcony.
0.1	What are you afraid of? Big crowds. What is the craziest thing you can imagine a dog doing? Play cards. When you look out the window, what do you see? A helicpter flying by.
0.2 - 0.5	What is the craziest thing you can imagine a dog doing? Play cards. What are you afraid of? Big crowds. When you look out the window, what do you see? A helicpter flying by.
0.6 - 0.9	What is the craziest thing you can imagine a dog doing? Play cards. What are you afraid of? Big crowds. Where do you like to go on holiday? My balcony.
1	Tell me one object that is in your purse right now. A second t-shirt. What was the best gift you have ever gotten? A self-made cake. If you could only eat one thing for the rest of your life, what would it be? I guess, curry?

Table 3: Highest perplexity samples

Plotting the perplexity compared to the sentence length for different parameters alpha, we can see that the captioner's perplexity (alpha = 0) depends much more on sentence length than the chatbot's perpexity. Comparison between the chatbot and the captioner's perplexities of individual sentences reveals that on the lower end of the perplexity spectrum, perplexities are more similar be-

tween the chatbot and the captioner, while on the higher end, the values more different.

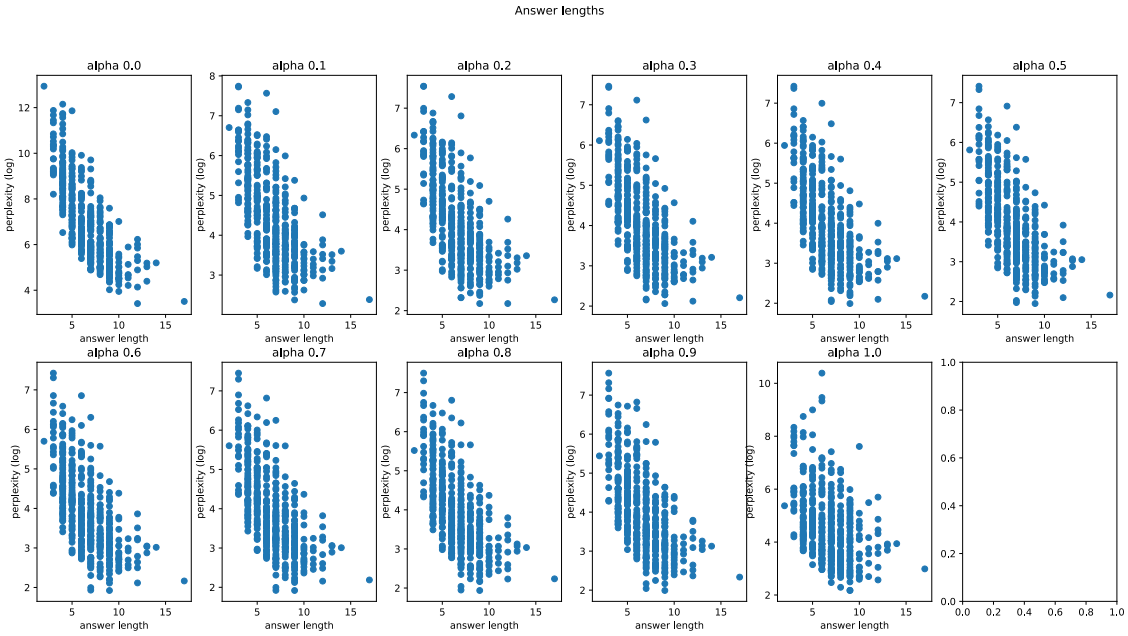

Figure 6: Answer lengths and perplexities at different alphas

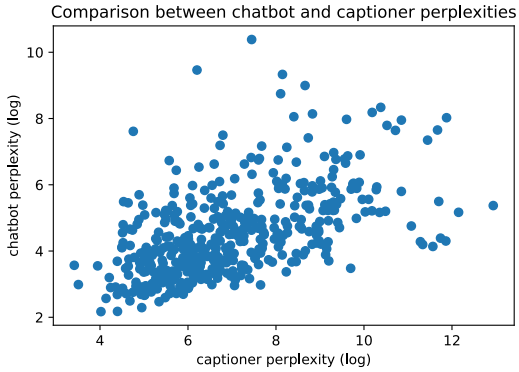

Figure 7: Chatbot (alpha=1) and captioner (alpha=0) perplexities

Given the closer alignment of the chatbot with the beginning of the sentence structure, one could speculate that the structure of the answer should be determined by the chatbot, while the captioner should have a stronger influence on the semantic content in order to deliver good results. While most token con-

I like to go to the beach.
I like city trips.
I enjoy hiking in the mountains.
Anywhere there is an ocean.
The desert is such an interesting place.
I spend holidays in my garden.
My balcony.
Anywhere I can bring my cat.
I like tiramisu.

Table 4: Example answers. Underlined tokens are marked as stopwords

tributes both to the structure and the content, function words, i.e. words from the closed lexicon mostly serve a grammatical function, while content words, i.e. words from the open lexicon, carry meaning on their own. Stopword lists were used to categorize each token in the dictionary as either a function or a content word, using stopwords as a proxy for function words. Since GPT-2's tokenizer works at a subword level, the distinction is not perfect, categorizing parts of content words as function words, such as "am" in "Tiramisu".

Regardless of the parameter alpha, a higher number of stopwords relative to the sentence length resulted in higher perplexities. A systematic difference between the alphas is not apparent, although high perplexities are slightly rarer at a lower stopword ratio for the chatbot compared to the captioner.

The stopword ratio is also more varied for shorter and medium length answers compared to long answers, which seem to approach a stopword ratio of 0.65. The slightly better performance of the chatbot at lower stopword ratios might be related to its increased ability to predict short answers.

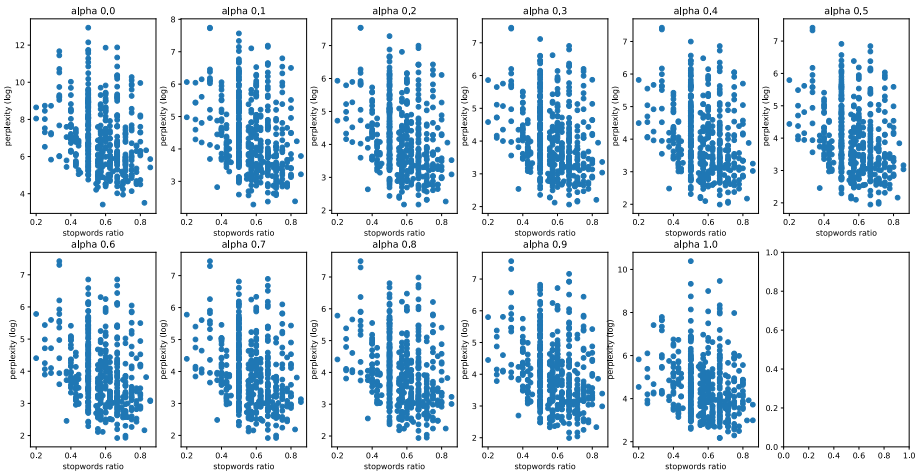

Figure 8: Stopword ratios and perplexities at different alphas

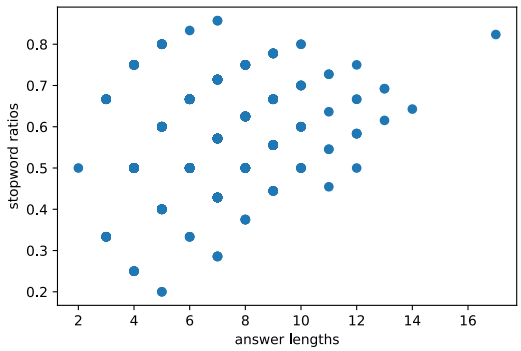

Figure 9: Answer length compared to perplexity

5.3 Cross-validation results

The SingleParameterModel, MultiParameterModel and the AllParametersModel were each trained for 30 epochs and evaluated with 5-fold cross validation. For the MultiParameterModel, a grouping of the vocabulary needs to be found. As a first idea for dividing the vocabulary space, the distinction between function and content words came to mind. The division was not expected to be meaningful, as the perplexities between sentences with different function/content word ratios did not differ strongly. Nevertheless, the trained models always learned a slightly higher parameter for the function words, suggesting that the chatbot

has a higher influence on the sentence structure than the captioner, which is consistent with the original hypothesis. The MultiParameterModel achieved a slightly higher validation loss than the SingleParameterModel.

The AllParametersModel has too many parameters to display them all, but

model type	avg. weights	avg. last training loss	avg. validation loss
SingleParameterModel	[0.6304]	46.762	94.047
MultiParameterModel	[0.6288, 0.5766]	47.521	94.152
AllParametersModel		52.776	79.963

Table 5: Average weights, last training loss and validation loss over 5 cross-validation folds for different model types. The MultiParameterModel was trained on the function/content word distinction, the first weight representing function, the second content words.

by filtering the highest and lowest parameters and finding the respective token, we can get an intuition what the model prioritizes. Of course, the weights are fitted very closely on the training data, as the number of parameters exceeds the number of training samples. The model is not very generalizable, since tokens which are not in the training corpus do not learn any weights.

5.3.1 Random and ordered partitions

no.	type	avg. weights	avg. last training loss	avg. validation loss
3	random	[0.5637, 0.6168, 0.5623]	65.601	94.479
5	random	[0.5565, 0.6652, 0.4074, 0.5359, 0.7321]	64.529	91.357
8	random	[0.5158, 0.3639, 0.6269, 0.6137, 0.5655, 0.7425, 0.5649, 0.7108]	64.365	91.320
10	random	[0.5670, 0.7216, 0.7734, 0.5842, 0.5237, 0.5831, 0.2734, 0.6347, 0.4111, 0.6831]	66.970	89.282
3	ordered	[0.6485, 0.4878, 0.5066]	47.126	93.613
5	ordered	[0.6498, 0.5926, 0.4837, 0.5098, 0.4899]	27.868	94.254
8	ordered	[0.6760, 0.5414, 0.5567, 0.5291, 0.4611, 0.5123, 0.5244, 0.4638]	47.106	94.072
10	ordered	[0.6698, 0.5157, 0.6781, 0.4841, 0.5169, 0.4531, 0.4861, 0.5325, 0.4936, 0.4860]	47.894	93.615

Table 6: Random and ordered partitions

To examine the influence of the number of parameters on the average perplexity, the MultiParameterModel was trained on random partitions of the tokenizer dictionary. Each vocabulary items in the group was randomly assigned to one of 3 to 10 groups without any regard for order or similarity. Since the number of items is high, it can be assumed that the groups were about equal in size. The resulting perplexities decrease as the number of partitions increases. While 3 random partitions perform about as well as the SingleParameterModel, there is a slight improvement with each increase of the number of partitions.

I could not find any information about the token order in the dictionary, whether there is a relation with the frequency of occurence, or whether the order is arbitrary. If adjacent tokens are similar in some regard, assigning them to a common parameter might improve the performance of the models compared to random parameter assignment. The partitions were created by dividing the dictionary space into 3 to 10 consecutive parts of equal size. Interestingly, the model performed performed worse with ordered rather than random partitions, and roughly achieve the same loss as the SingleParameterModel. The weights also stay in a narrower range around the initial value of 0.5 compared to the random partitions.

5.3.2 Clustering results

Word vectors or word embeddings are numerical representations of words based on the idea that co-occurence of words in text is related to semantic similarity. The cosine similarity is used as a similarity metric. Since the standard corpus in gensim's Word2Vec library is very small, I used 300d-dimensional pre-trained LexVec vectors[8]. 5576 tokens, about 10% of the tokenizer's vocabulary, did not have pre-trained word vectors, consisting mainly of unintellegible symbol combinations. After retrieving the word vectors of all vocabulary tokens which were in the corpus, I clustered them with the K-Means and DBSCAN algorithm.

K-Means is a simple clustering algorithm which assigns data points to clusters in such a way that the distance to the cluster centers is minimized. The scikit-learn implementation of K-Means only allows using the Euclidean distance as a distance metric, while the correct similarity measure for word vectors is the cosine distance. As the word vectors are normalized to unit vectors, there is a close relation between the Euclidean and the cosine distance. The number of clusters is a K-Means parameter and was set to values between 2 and 19, and the resulting clusterings were interpreted as groups for the MultiParameterModel. Additionally, the tokens without word vectors were interpreted as a group as well. Word2Vec also has a functionality to locate the word in its corpus that are the most similar to a given vector. By doing this with the cluster centroids found by K-Means, we obtain some information cluster's content. Up to and including 7 clusters, the cluster sizes are very unbalanced, with one cluster including almost half of the word vectors.

n	centroid	weight	cluster size	avg. LTL	avg. VL
3	atters (0.72), hered (0.72)	0.548	7157	63.222	94.571
	incidentally (0.55), uncoincidentally (0.54),	0.625	37524		
		0.521*	5576*		
5	Matr (0.62), Cald (0.61), Streer (0.60)	0.541	2679	68.690	92.540
	incidentally (0.59), presumably (0.58)	0.491	21856		
	Bridgekeeper (0.52), GArden (0.51)	0.743	13461		
	atters (0.72), hered (0.72)	0.543	6685		
7	alluding (0.65), infuriated (0.64)	0.553	8161	69.039	92.849
	atters (0.75), merical (0.73)	0.570	3751		
	plear (0.70), intotdeauna (0.68)	0.511	3657		
	PARAD (0.69), STANDA (0.67)	0.635	2046		
	Comination (0.52), Foundering (0.52)	0.747	11205		
	displying (0.51), combintation (0.50)	0.454	13334		
	Matr (0.62), Cald (0.61)	0.558	2527		

Table 7: K-Means results

Unlike K-Means, DBSCAN does not allow specifying the amount of clusters to be found. Cluster assignment is based on density and data points in scarce environments are labeled as noise. Two parameters can be used to fine-tune how dense the desired clusters are supposed to be: the number of minimum points in the immediate neighborhood to consider an area as a cluster, as well as the maximum distance between the points. Setting the number of minimum points to a low value, I try to find a suitable value for the maximum distance that results in multiple clusters of roughly equal size. Selecting a few values for the maximum distance manually, it appears that on the word vector data set, the algorithm either finds many very small clusters or very few large clusters, suggesting that most word vectors have similar distances to each other. Neither clustering method led to a low validation loss.

6 Project summary

Both versions of the architecture were implemented successfully, resulting in a fixed parameter generation method as well as trainable machine learning models. Evaluation with perplexity showed that a combination approximates the dataset's probability distribution more closely than each of the pre-trained language models. Training the machine learning models confirmed the optimal parameter found with grid search, but required some adaptation of the standard PyTorch training loop: Perplexity was adopted as a loss function and a custom gradient function replaced the automatic differentiation process.

In order to limit the number of parameters and prevent overfitting, the vocabulary is pre-grouped. Different ways of grouping the vocabulary were explored and evaluated. Neither randomly nor systematically partitioning the dictionary led to an improvement in model quality compared to the single parameter version. The only notable decrease in perplexity was achieved by using a high number of parameters, and even then, randomly partitioning the vocabulary space is more successful than either clustering the partitions or grouping consecutive slices of data. It is possible that the advantage of randomness lies in grouping high impact tokens with tokens not in the training corpus, but this was not investigated further. Additionally, there are a few outliers with very high perplexity in the dataset and removing them might affect both the weights and the average perplexity.

Of course, there are many more ways to group the vocabulary space. The AllParametersModel's learned weights could for example be used as a basis for the partitioning, with similar weights being grouped together. The number and composition of the groups could even be learned automatically, adding a clustering objective to the training algorithm's classification objective. However, the strongest limitation on the model quality are the pre-trained models: The chatbot and captioners' output distributions remain fixed, and tokens which are not predicted well by either are not learnable with the current method. The AllParametersModel could be used to estimate the lower bound for the combined model's perplexity by deliberately overfitting it on and then validating it on the entire dataset.

While the evaluation has focused entirely on perplexity, it is unclear whether actually generated answers are meaningful, as this can only be assessed by humans. The generated output also depends on the method of selecting tokens from the probability distribution, which was not the focus of the present project. Currently, only greedy generation and sampling are available, which are outperformed by more sophisticated methods such as contrastive sampling and beam search. A future continuation should include a human assessment of the optimal model that can be achieved with the current architecture, and further steps should be taken based on the results. If the model quality is suitable for real-world applications, the existing models could be improved by finding meaningful

vocabulary space partitions or cleaning and extending the training dataset to more closely approximate the ideal model. If the ideal model's quality is not sufficient, the linear layer connecting the pre-trained models could be replaced by a transformer layer, taking positional information and previous context into account. Whether the model is useful in practise remains an open question.

7 Timeline

In general, I was able to follow the project schedule and complete most tasks before their deadlines. The most time-consuming part of the process was fixing unexpected errors, especially finding the reason for the initial training process' poor outcomes, which caused a delay of about a week.

- until 23. 11.: I will set up different configurations of version 1 to create different options for the analysis and test whether quality issues could be reduced by additional parameters such as a repetition penalty. Optionally, the efficiency of the architecture could be improved by adding a custom generation method.
- until 14. 12.: I will run the generation method of version 1 with different parameters on the dataset. A suitable measurement for assessing model quality (e.g. perplexity) should be found and assessed for different values of the parameter alpha.
- until 28. 12.: I will set up and try to train version 2.
- until 11. 1.: I will apply the methods of analysis from version 1 to version 2 and possibly some additional techniques.
- until 25. 1.: I will summarize the project results in a final report.

References

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[2] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014.

[3] A. Radford and K. Narasimhan, "Improving language understanding by generative pre-training," 2018.

[4] P. J. Liu, M. Saleh, E. Pot, B. Goodrich, R. Sepassi, L. Kaiser, and N. Shazeer, "Generating wikipedia by summarizing long sequences," *CoRR*, vol. abs/1801.10198, 2018.

[5] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An image is worth 16x16 words: Transformers for image recognition at scale," 2020.

[6] C. Huyen, "Evaluation metrics for language modeling," 2019.

[7] Y. Zhang, S. Sun, M. Galley, Y.-C. Chen, C. Brockett, X. Gao, J. Gao, J. Liu, and B. Dolan, "Dialogpt: Large-scale generative pre-training for conversational response generation," 2019.

[8] A. Salle, M. Idiart, and A. Villavicencio, "Enhancing the lexvec distributed word representation model using positional contexts and external memory," 2016.