By Mahir Uysal

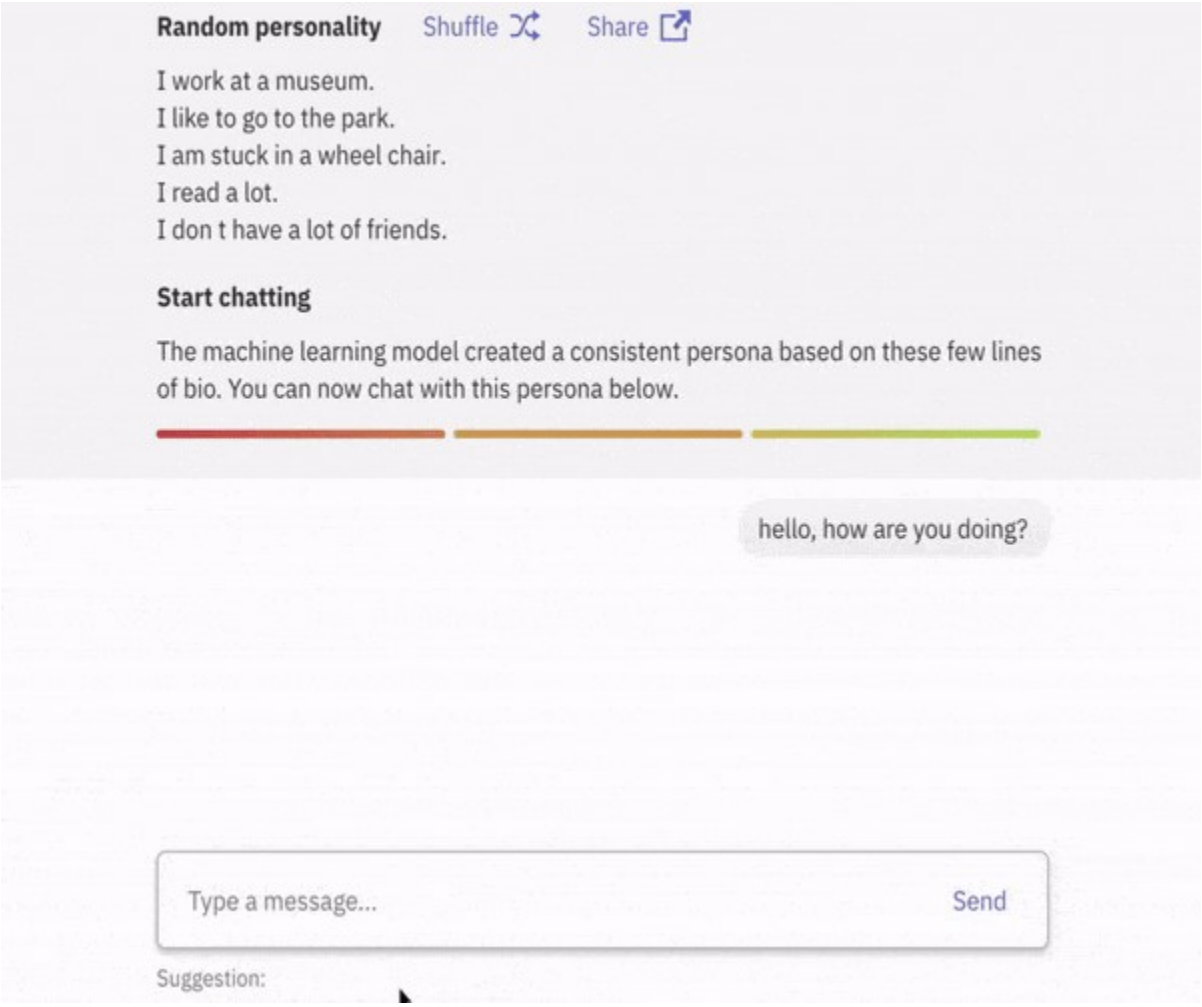
How to build a State-of-the-Art Conversational AI with Transfer Learning

Thomas Wolf Follow
May 9, 2019 · 12 min read

A few years ago, creating a chatbot -as limited as they were back then- could take months, from designing the rules to actually writing thousands of answers to cover some of the conversation topics.

With the recent progress in deep-learning for NLP, we can now get rid of this petty work and build much more powerful conversational AI in just a matter of hours as you will see in this tutorial.

> *We've set up a demo running the pretrained model we'll build together in this tutorial at convai.huggingface.co. Be sure to check it out!* 🎮



Online demo of the pretrained model we'll build in this tutorial at convai.huggingface.co. The "suggestions" (bottom) are also powered by the model putting itself in the shoes of the user.

Here is what we will learn and play with today:

- How you can use **Transfer Learning** to build a **State-of-the-Art dialog agent** based on **OpenAI GPT** and **GPT-2** Transformer language models,
- How you can **reproduce** the model we used in the NeurIPS 2018 dialog competition ConvAI2 which **won the automatic metrics track**,
- How we distilled 3k+ lines of competition code in less than **250 lines of commented training code** (with distributed & FP16 options!), and

- How you can train this model for **less than $20** on a cloud instance, or just use our open-sourced **pre-trained model.**

> *Together with this post, we released a clean and commented code base with a pretrained model! Check the Github repo here* 🚀

The story of this post began a few months ago in Montreal 🇨🇦 where Hugging Face finished 1st 🏆 in the automatic track of the Conversational Intelligence Challenge 2 (ConvAI2), a dialog competition at NeurIPS 2018.

Our secret sauce was a **large-scale pre-trained language model, OpenAI GPT,** combined with a **Transfer Learning fine-tuning technique**.

With the fast pace of the competition, we ended up with over 3k lines of code exploring many training and architectural variants.

Clearly, publishing such raw code would not have been fair.

In the meantime, we had started to build and open-source a repository of transfer learning models called `pytorch-pretrained-BERT` which ended up being downloaded more than 150 000 times and offered implementations of large-scale language models like OpenAI GPT and it's successor GPT-2 🦄

A few weeks ago, I decided to re-factor our competition code in a clean and commented code-base built on top of `pytorch-pretrained-BERT` and to write a detailed blog post explaining our approach and code.

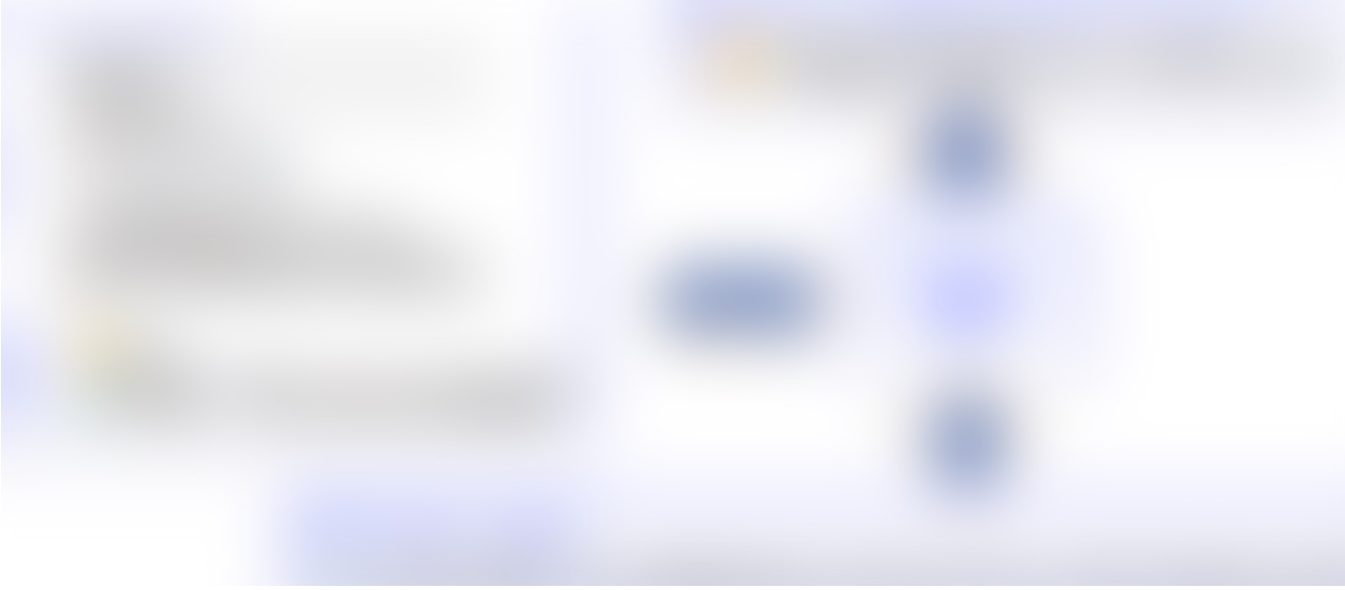
So here we are, let's dive in 🚀

An AI with a personality 🤠

We'll build a **conversational AI** with a **persona**.

Our dialog agent will have a *knowledge base* to store a few sentences describing who it is (*persona*) and a *dialog history*. When a *new utterance* will be received from a user, the agent will combine the content of this knowledge base with the newly received utterance to generate *a reply*.

Here is the general scheme:



When we train a deep-learning based dialog agents, in an end-to-end fashion, we are facing a major issue:

> *Dialog datasets are small and it's hard to learn enough about language and common-sense from them to be able to generate fluent and relevant responses.*

Some approaches try to solve this by filtering the output of the model to improve the quality using smart beam search. Here we'll take another path that gathered tremendous interest over the last months: **Transfer Learning.**

The idea behind this approach is quite simple:

- start by **pretraining** a language model on a very large **corpus** of text to be able to generate long stretches of contiguous coherent text,
- **fine-tune** this language model to adapt it to our end-task: **dialog**.

Pretraining a language model is an expensive operation so it's usually better to start from a model that has already been pretrained and open-sourced.

What would be a good pretrained model for our purpose?

The bigger the better, but we also need a model that *can generate text*. The most commonly used pretrained NLP model, BERT, is pretrained on full sentences only and is not able to complete unfinished sentences. Two other models, open-sourced by OpenAI, are more interesting for our use-case: **GPT** & **GPT-2**.

Let's have a quick look at them 🔎

🦄 OpenAI GPT and GPT-2 models

In 2018 and 2019, Alec Radford, Jeffrey Wu and their co-workers at OpenAI open-sourced two language models trained on a very large amount of data: **GPT** and **GPT-2** (where *GPT* stands for *Generative Pretrained Transformer)*.

A decoder/causal Transformer attends to the left context to generate next words

GPT and GPT-2 are two very similar Transformer-based language models. These models are called *decoder* or *causal* models which means that they use the left context

to predict the next word (see left figure).

Many papers and blog posts describe Transformers models and how they use attention mechanisms to process sequential inputs so I won't spend time presenting them in details. A few pointers if you are not familiar with these models: Emma Strubell's EMNLP slides are my personal favorite and Jay Alammar's "Illustrated Transformer" is a very detailed introduction.

In `pytorch-pretrained-BERT` OpenAI GPT's model and its tokenizer can be easily created and loaded from the pretrained checkpoint like this:

For our purpose, a language model will just be a model that takes as input a sequence of tokens and **generates a probability distribution over the vocabulary for the next token following the input sequence**. Language models are usually trained in a parallel fashion, as illustrated on the above figure, by predicting the token following each token in a long input sequence.

You probably noticed we've loaded a model called *OpenAIGPTDoubleHeadsModel* which sounds a bit more complex than the language model we've just talked about and you're right!

Pretraining these models on a large corpus is a costly operation, so we'll start from a model and tokenizer pretrained by OpenAI. The tokenizer will take care of splitting an input string in tokens (words/sub-words) and convert these tokens in the correct numerical indices of the model vocabulary.

This is because we need to adapt our model to *dialog*. Let's see how this goes!

🐣 Adapting a language model to a dialog task

Our language model is trained with a **single** input: a sequence of words.

But as we saw earlier, in a dialog setting, our model will have to use **several types** of *contexts* to generate an output sequence:

- one or several *persona sentences*,
- the *history of the dialog* with at least the last utterance from the user,
- the *tokens of the output sequence* that have already been generated since we generate the output sequence word by word.

> *How can we build an input for our model from these various contexts?*

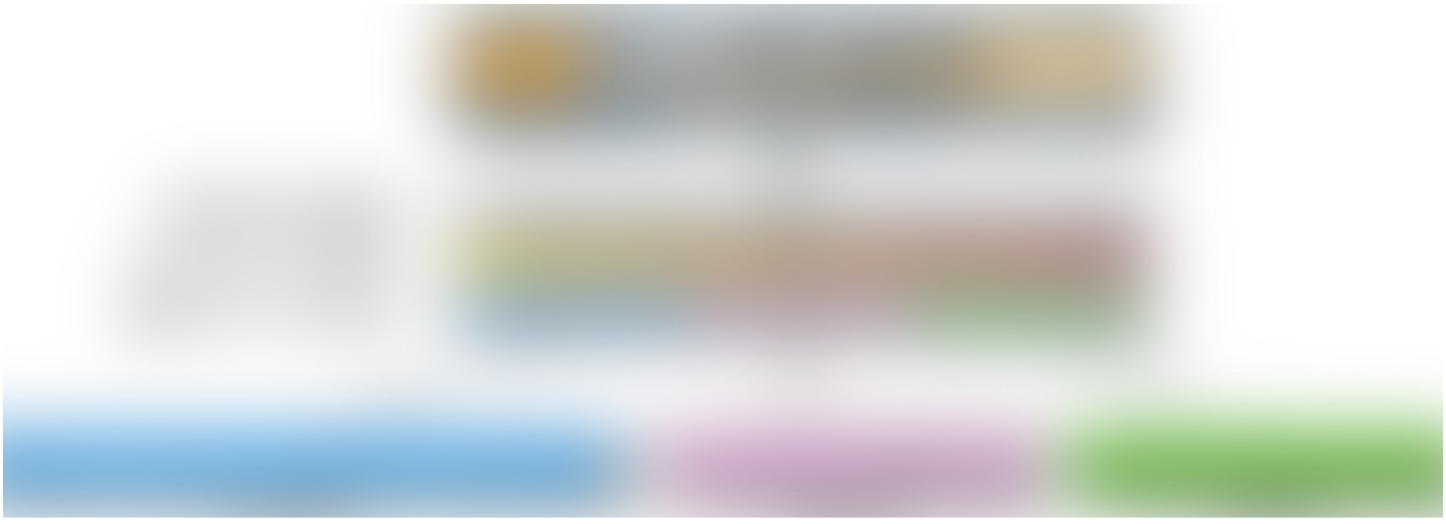
A simple answer is just to **concatenate** the context segments in a single sequence, putting the reply at the end. We can then **generate a completion** of the reply token by token by continuing the sequence:

Input sequence: a concatenation of persona (blue), history (pink) and reply (green) with delimiters (light pink). Here we generate the word “you” to complete the reply.

There are two issues with this simple setup:

- *Our transformer is color-blind!* The delimiter tokens only give it a weak idea of which segment each word belongs to. For example, the word “NYC” is indicated in blue (persona) in our illustration but our model will have a hard time extracting this information from the delimiters alone: **we should add more information about the segments.**
- *Our transformer is position-blind!* Attention is a symmetrical dot-product so **we should add position information for each token.**

An easy way to add this information is to build three parallel input sequences for *word*,



Summing three types of inputs embeddings indicating words (grey), position (gradient) and segments (blue/pink

/green)

How do we implement this?

First, we'll add *special tokens* to our vocabulary for delimiters and segment indicators. These tokens were not part of our model's pretraining so we will need to **create** and **train new embeddings** for them.

Adding special tokens and new embeddings to the vocabulary/model is quite simple with `pytorch-pretrained-BERT` classes. Let's add five special tokens to our tokenizer's vocabulary and model's embeddings:

> *These special-tokens methods respectively add our five special tokens to the vocabulary of the tokenizer and create five additional embeddings in the model.*

Now we have all we need to build our input sequence from the *persona*, *history,* and *beginning of reply* contexts. Here is a simple example:

👑 Multi-tasks losses

We have now initialized our pretrained model and built our training inputs, all that remains is to choose a **loss** to **optimize** during the fine-tuning.

> We will use a multi-task loss combining language modeling with a next-sentence prediction objective.

> *The next-sentence prediction objective is a part of BERT pretraining. It consists in randomly sampling distractors from the dataset and training the model to distinguish whether an input sequence ends with a gold reply or a distractor. It trains the model to look at the global segments meaning besides the local context.*

Now you see why we loaded a "*Double-Head*" model. One head will compute language modeling predictions while the other head will predict next-sentence classification labels. Let's have a look at how losses are computed:

Multi-task training objective — the model is provided with two heads for language modeling prediction (orange) and next-sentence classification (blue)

The **total loss** will be the weighted sum of the **language modeling loss** and the **next-sentence prediction loss** which are computed as follow:

- **Language modeling**: we project the hidden-state on the word embedding matrix to get logits and apply a *cross-entropy loss on the portion of the target corresponding to the gold reply* (green labels on the above figure).
- **Next-sentence prediction**: we pass the hidden-state of the *last* token (the *end-of-sequence* token) through a linear layer to get a score and apply a *cross-entropy loss to classify correctly a gold answer among distractors*.

Let's see how we can code this:

We now have all the inputs required by our model and we can run a forward pass of the model to get the two losses and the total loss (as a weighted sum):

We are ready to start the training 🎉

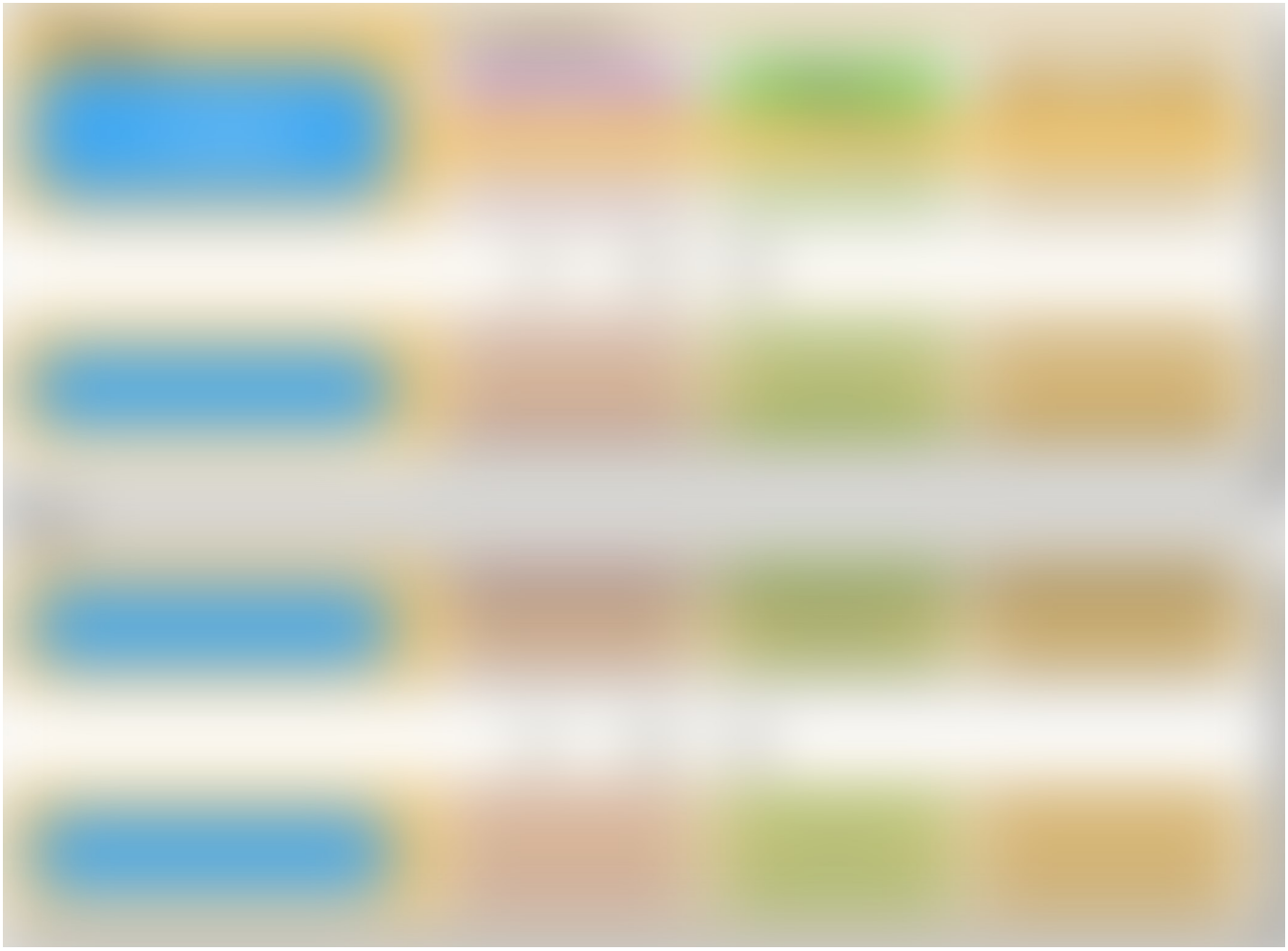
🦊 Training on a dialog dataset

The ConvAI2 competition used an interesting dataset released by Facebook last year: PERSONA-CHAT.

It's a rather large dataset of dialog (10k dialogs) which was created by crowdsourcing *personality sentences* and asking paired crowd workers to *chit-chat* while playing the part of a given character (an example is given on the left figure).

This dataset is available in raw tokenized text format in the nice Facebook's ParlAI library. To bootstrap you, we also uploaded a JSON formatted version that you can download and tokenize using GPT's tokenizer like this:

The JSON version of PERSONA-CHAT gives quick access to all the relevant inputs for training our model as a nested dictionary of lists:



Organization of the JSON version of PERSONA-CHAT

Using the awesome PyTorch ignite framework and the new API for Automatic Mixed Precision (FP16/32) provided by NVIDIA's apex, we were able to distill our +3k lines of competition code in less than **250 lines of training code** with distributed and FP16 options!

We've covered the essential parts of the code in the above gists so I'll just let you read the commented code to see how it all fits together.

> *The training (`train.py`) code is here ⇨ 🎮*

Training this model on an AWS instance with 8 V100 GPU takes less than **an hour** (currently less than $25 on the biggest p3.16xlarge AWS instance) and gives results close to the SOTA obtained during the ConvAI2 competition with **Hits@1 over 79,**

perplexity of 20.5 and F1 of 16.5.

A few differences explain the slightly lower scores vs our competition model, they are detailed in the readme of the code repo here and mostly consists in tweaking the position embeddings and using a different decoder.

Talking with the Model — the Decoder

The amazing thing about dialog models is that you can talk with them 🤗

To interact with our model, we need to add one thing: a **decoder** that will build full sequences from the next token predictions of our model.

Now there have been very interesting developments in decoders over the last few months and I wanted to present them quickly here to get you up-to-date.

The two most common decoders for language generation used to be **greedy-decoding** and **beam-search**.

Generating a sentence word by word (source)

Greedy-decoding is the simplest way to generate a sentence: at each time step, we select the most likely next token according to the model until we reach end-of-sequence tokens. One risk with greedy decoding is that a *highly probable* token may be hiding after a *low-probability* token and be missed.

Beam-search try to mitigate this issue by maintaining a beam of several possible

sequences that we construct word-by-word. At the end of the process, we select the best sentence among the beams. Over the last few years, beam-search has been the *standard decoding algorithm* for almost all language generation tasks including dialog (see the recent [1]).

However several developments happened in 2018/early-2019. First, there was growing evidence that beam-search was strongly *sensitive to the length* of the outputs and best results could be obtained when the output length was *predicted* before decoding ([2, 3] at EMNLP 2018). While this makes sense for **low-entropy tasks** like translation where the output sequence length can be roughly predicted from the input, it seems arbitrary for **high-entropy tasks** like dialog and story generation where outputs of widely different lengths are usually equally valid.

In parallel, at least two influential papers ([4, 5]) on high-entropy generation tasks were published in which greedy/beam-search decoding was replaced by ***sampling*** from the next token distribution at each time step. These papers used a variant of sampling called ***top-k sampling*** in which the decoder *sample only from the top-k most-probable tokens* (k is a hyper-parameter).

The last stone in this recent trend of work is the study recently published by Ari Holtzman et al. [6] which showed that the distributions of words in texts generated using *beam-search* and *greedy decoding* is very different from the

Left: Probability assigned to tokens generated by humans and beam search using GPT-2 (Note the strong variance in human text not reproduced by beam-search). Right: N-gram distributions in human and machine-generated texts (Note the complete separation between greedy/beam-search and sampling decoding methods).

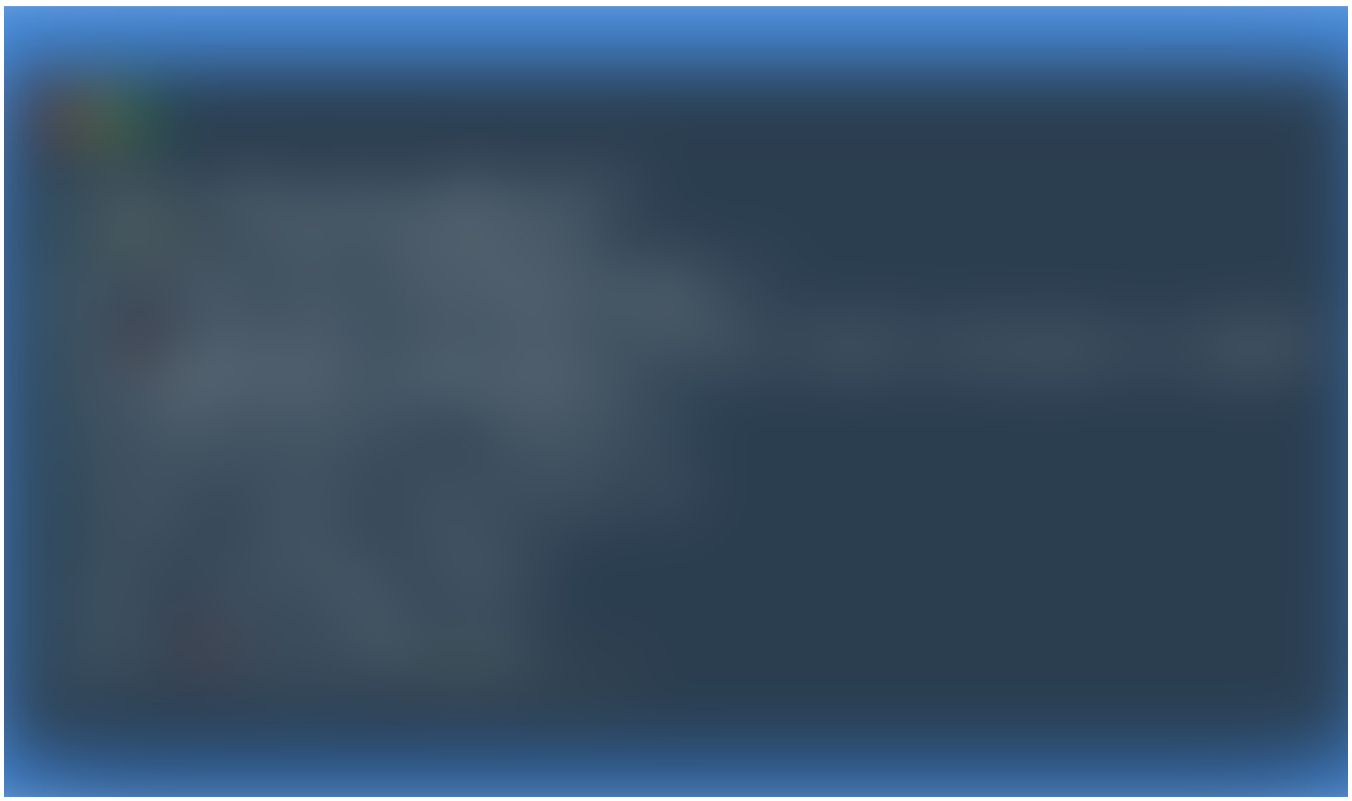
Currently, the two most promising candidates to succeed beam-search/greedy decoding are ***top-k*** and ***nucleus (or top-p) sampling***. The general principle of these two methods is to sample from the next-token distribution after having filtered this distribution to keep only the top k tokens (*top-k*) or the top tokens with a cumulative probability just above a threshold (*nucleus/top-p*).

Here is how we can decode using ***top-k*** and/or ***nucleus/top-p*** sampling:

We are now ready to talk with our model 🚀

The interactive script is here (`interact.py`) and if you don't want to run the script you can also just play with our live demo which is here 🎮

Here is an example of dialog:



Example using the interactive scripts with default settings — Bot personality: I read twenty books a year. I'm a stunt double as my second job. I only eat kosher. I was raised in a single parent household.

Conclusion

We've come to the end of this post describing how you can build a simple state-of-the-art conversational AI using transfer learning and a large-scale language model like OpenAI GPT.

As we learned at Hugging Face, getting your conversational AI up and running quickly is the best recipe for success so we hope it will help some of you do just that!

Be sure to check out the associated demo and code:

- the live demo is here and
- the open-sourced code and pretrained models are here.

As always, if you liked this post, give us a few 👏 to let us know and share the news around you!

References:

[1] ^ *Importance of a Search Strategy in Neural Dialogue Modelling* by Ilya Kulikov, Alexander H. Miller, Kyunghyun Cho, Jason Weston (http://arxiv.org/abs/1811.00907)

[2] ^ *Correcting Length Bias in Neural Machine Translation* by Kenton Murray, David Chiang (http://arxiv.org/abs/1808.10006)

[3] ^ *Breaking the Beam Search Curse: A Study of (Re-)Scoring Methods and Stopping Criteria for Neural Machine Translation* by Yilin Yang, Liang Huang, Mingbo Ma (https://arxiv.org/abs/1808.09582)

[4] ^ *Hierarchical Neural Story Generation* by Angela Fan, Mike Lewis, Yann Dauphin (https://arxiv.org/abs/1805.04833)

[5] ^ *Language Models are Unsupervised Multitask Learners* by Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever (https://openai.com

/blog/better-language-models/)

[6] ^ *The Curious Case of Neural Text Degeneration* by Ari Holtzman, Jan Buys, Maxwell Forbes, Yejin Choi (https://arxiv.org/abs/1904.09751)

[7] ^ *Retrieve and Refine: Improved Sequence Generation Models For Dialogue* by Jason Weston, Emily Dinan, Alexander H. Miller (https://arxiv.org/abs/1808.04776)

[8] ^ *The Second Conversational Intelligence Challenge (ConvAI2)* by Emily Dinan et al. (https://arxiv.org/abs/1902.00098)

Thanks to Victor Sanh, Clément Delangue, and Pierric Cistac.

Machine Learning NLP Deep Learning Chatbots AI