A Declarative Specification for Machine Learning Architectures

A Declarative Specification for Machine Learning Architectures

a thesis presented
by
Gordon C. Kamer
to
The Department of Computer Science

in partial fulfillment of the requirements
for the degree of
Bachelor of Arts
in the subject of
Computer Science

Harvard College
Cambridge, Massachusetts
March 2023

Thesis advisor: Professor Boaz Barak Gordon C. Kamer

A Declarative Specification for Machine Learning Architectures

Abstract

Going from the description of a model architecture in a figure to its implementation can be a fraught process. This work presents a markup language for specifying model architectures; an associated Python package used to convert models into a runnable format; utilities to import the model into PyTorch; a publicly available model repository; and a tool to visualize the resulting models. This paper also provides a review of some popular machine learning architectures, examples of models created using the markup, and associated experiments. The system is named Agrippa[*], determined through a bracket-poll tournament conducted by the author's roommates.

The language simplifies certain aspects of model development: parameters are named, explicit, and can be specified as being frozen or shared; models can be imported into different projects with few code changes; coherent parts of the architecture can be organized into self-contained blocks; and parameter initialization techniques are explicit. The language syntax is simply XML. Models compiled using the Agrippa Python package are converted into the ONNX format, a neural network interchange format supported by a variety of machine learning frameworks.

The web component of Agrippa can be found at `http://agrippa.build`, which contains the visualization tool, model repository, and documentation.[†]

At this stage of development, the system has a number of limitations: models may not be compiled if they are larger than 2GB due to a formal limit imposed by the ONNX file format; not all operations available in ONNX are supported by the compiler; the visualization tool does not yet support visual editing; and certain techniques, like dropout layers and batch normalization, require workarounds. There are a number of plausible ways each of these limitations may be addressed in the future.

[*]Marcus Vipsanius Agrippa was the most trusted deputy of the Roman Emperor Augustus. The reader should feel free to devise a related acronym, though none was intended.

[†]The source code for the web component can be found at `https://github.com/gkamer8/agrippa-zoo` and the open source Python package repository can be found at `https://github.com/gkamer8/agrippa-pkg`.

Contents

Acknowledgments

I would like to thank my parents for their love and support. I would also like to thank my thesis advisor, Boaz Barak, and my second thesis reader, Matin Wattenberg, for their feedback and generosity with their time.

1 Introduction

If you were to travel back in time a few decades with the knowledge you have today about the state of the art in artificial intelligence, you would probably be unable to implement any of your ideas. The first challenge would be hardware. The second challenge would be the lack of associated tools: auto differentiation libraries, Python, PyTorch, JAX, and so on. The goal of this work is to prototype a system I hope might make dealing with neural networks a bit easier for some people.

The heart of the system is a markup language for machine learning architectures. By declaring the contours of the model in a markup, the language provides a more direct mapping between the kinds of figures you might see in a paper and the underlying implementation. This system also permits straightforward visualization. In an ideal world, many users would rarely deal directly with markup files in favor of visual editing tools. For now, however, the visualization component does not support editing. In any case, the visualization can be a good tool for exposition, education, and debugging. The underlying XML is also straightforward to work with.

1.1 Motivation

There are a variety of problems with how machine learning models are constructed today that this system is intended to alleviate. Many novel architectures are implemented in idiosyncratic codebases. Reading, understanding, and, importantly, modifying models can be more difficult than they should be. The description of a model written in a high-level framework can be under-specified because certain important details about a model are often hidden deep inside library functions, including: the name of a parameter so that you can search for its value; those parameters' initialization schemes; frozen or shared parameters; and the implementation of certain portions of a model (such as an attention mechanism). Moreover, parts of a model can be dependent on peculiarities of implementation at the far reaches of the codebase.

Consider the case of modifying a pre-trained model. In the HuggingFace Transformers library, the BioGptModel class is a subclass of BioGptPreTrainedModel, which is a subclass of PreTrainedModel - not to be confused with AutoModel or AutoModelForCausalLM, which is how most documentation accesses models in the library. Editing any number of these classes must be done for many modifications, since the API for configuring the model is restricted to only the most basic hyperparameters. If you want to make a small change in how embeddings are calculated, you may find that inside BioGptModel, there is a reference to nn.Embedding — and now you are in a completely different library. Then, even after these changes, how can you be sure that the pre-trained weights are still mapped how you want them? What if the model you want to use exists in code in a different framework from the few you are comfortable with? The general task of modifying other people's models is not impossible but also not accessible. For people outside the AI guild, there should be some more direct way to turn the idea for a model into its implementation — or even simply inspect some of the hidden but critical details deep inside the usual layers of abstraction.

1.2 Contributions

The first chapter of this work is a literature review discussing other ML tools (complements and alternatives to this paper's system) as well as model architecture design spaces that are particularly amenable to a declarative approach. The overview of model architectures is meant to be both expository and provide context to the sections that follow.

Second, the paper gives a technical overview of Agrippa. It explains the basics of the architecture markup language, which is a completely novel approach to packaging ML models; the associated Python package; and the online component to Agrippa, which is comprised of a visualization tool and a model zoo. All of the aforementioned software was written by the author for this project with the exception of the ONNX to PyTorch conversion tool, which was merely modified.[*]

The third chapter briefly summarizes some experiments conducted using Agrippa. Two of the experiments are reproductions of existing work, and the third is a short foray into a novel scheme for expanding the length of the context in Transformers.

Finally, the thesis concludes with a discussion of the system's limitations and possibilities for future work.

[*]The ONNX to PyTorch conversion tool is a utility provided in the Agrippa Python package, for which the author's main contribution is the XML to ONNX compilation method. The original ONNX to PyTorch conversion package can be found here: https://github.com/ENOT-AutoDL/onnx2torch

2 Literature Review

The AI-paper-writing industrial complex has grown nearly as much as the parameter counts of our largest models. This AI summer is made possible by conceptual advances in machine learning models as well as their associated scaffolding, including the various machine learning frameworks, data engineering, and so on. This review focuses first on tools which are analogous or complementary to Agrippa. Second, the review includes an expository section discussing the transformer model and some of its variants. The expository section is meant both to serve as a reference as well as to give context to the design choices explained in the following chapters.

2.1 Software Tools for Machine Learning

The purpose of Agrippa is to make understanding, sharing, building, and re-using AI models easier. A variety of other tools have made similar efforts.

Perhaps the most important of these tools are the main machine learning frameworks, which provide auto-differentiation and pre-packaged model components. PyTorch[1], TensorFlow[2], and JAX[3] are some of the most popular. All of these frameworks are imperative. They are also designed to be performant on GPUs or in multi-node settings, making them especially useful for scaling models beyond a billion parameters. HuggingFace provides a litany of packages built on top of PyTorch that allows for simplified multi-node training, model sharing, tokenization, and other benefits. Agrippa is similar to HuggingFace's tools and is meant to be complementary to these frameworks.

The Agrippa Python package compiles architecture markups into ONNX[4], which stands for Open Neural Network Exchange. ONNX is an open source specification for neural networks. The ONNX format specifies a computational graph that is meant to be as general as possible. Inference engines such as ONNX Runtime[5] by Microsoft allow ONNX models to be run with high performance. As an interchange format, exporting models to ONNX is supported by PyTorch and TensorFlow. TensorFlow also natively supports importing ONNX files. Agrippa is tied closely to ONNX.

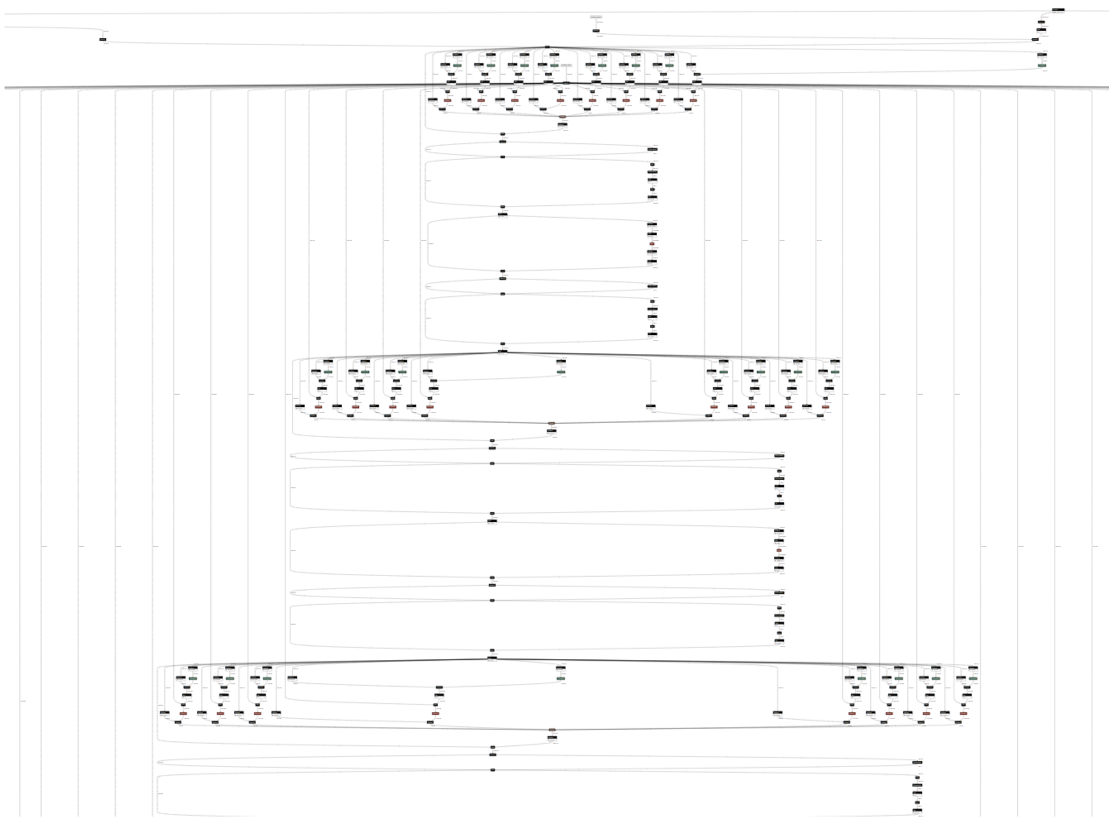


Figure 2.1: How a large Transformer model might appear in Netron.

Due to the ONNX format's graph structure, models can be visualized easily. Netron is an open source tool that takes an ONNX file as input and shows the individual computational nodes of the file, including various other metadata, such as tensor shapes and names. For small models, this visualization is sufficient; however, for large models, the lack of user-identified structure prevents straightforward understanding. This observation is the motivation behind the *block* construct in Agrippa.

TensorBoard is a tool for working with TensorFlow that allows for visualization of the model weights, the model computational graph, and a variety of metrics. The TensorFlow Graph Visualizer[6] is conceptually similar to the system presented in this work: it takes a

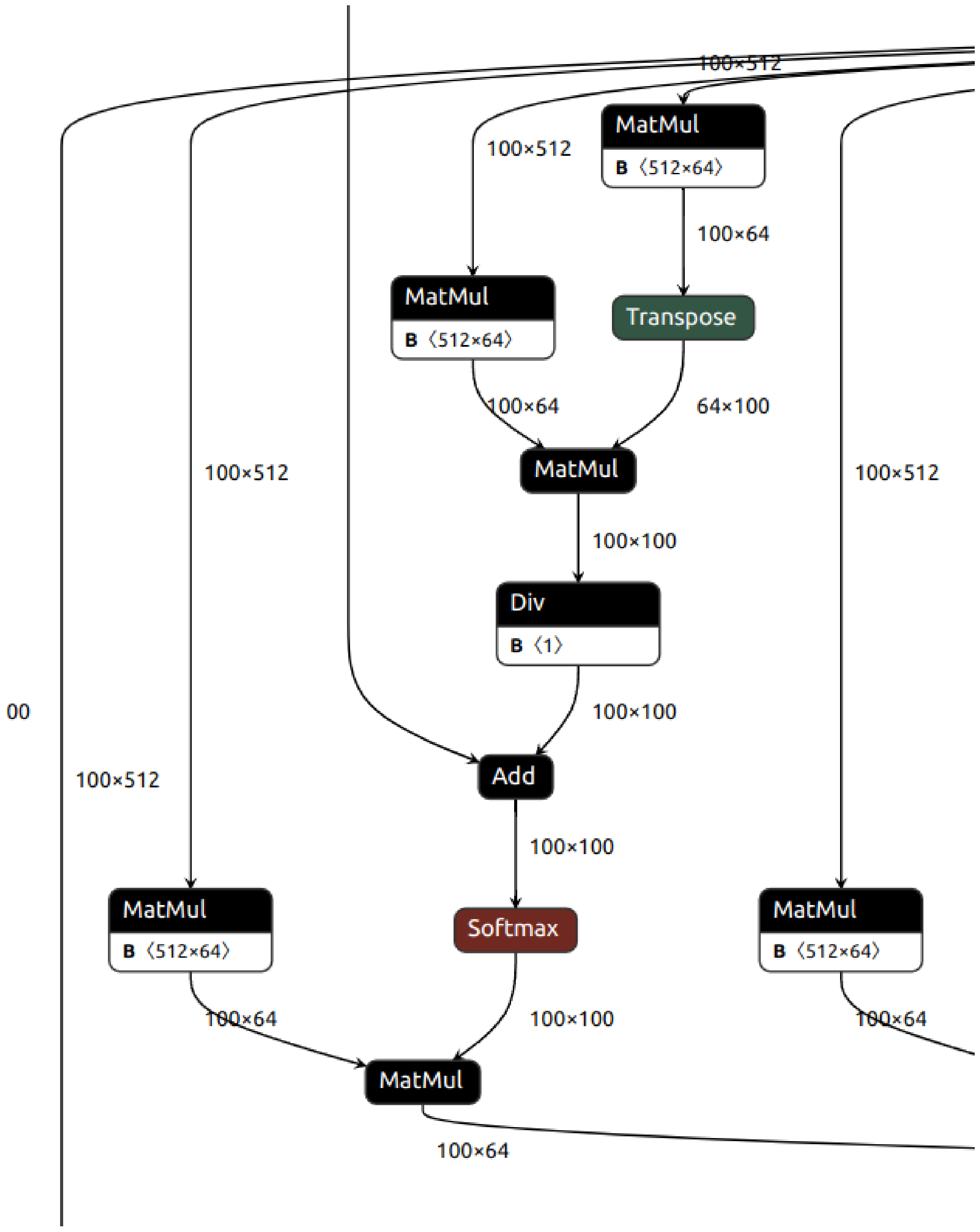


Figure 2.2: A small portion of a Transformer in Netron.

TensorFlow model as input and provides a visualization interface for the computational graph. Unlike in the system presented here, the TensorFlow visualizer tries to automatically construct a reasonable visualization based on patterns found in the TensorFlow computational graph. When using this paper's system, structural annotations are more explicit, hopefully allowing for greater clarity, unambiguous visual editing, and other benefits. However, the TensorFlow suite of tools is altogether a more developed product.

A smaller existing visualization tool is Net2Vis[7], which focuses on producing visualizations for publication. The problem identified is that figures in different papers are similar but sometimes lack coherence or even accuracy. The Net2Vis tool provides conceptually uniform figures automatically. The authors note that typically readers want "arbitrary aggregations of multiple layers," which makes TensorBoard prohibitively inflexible. The Agrippa visualization is not meant to fit the model architecture in one picture, allowing the user to dynamically interact with the graph to see nested subcomponents.

2.2 Transformers

Introduced by scientists at Google in 2017, the Transformer[8] has become the most popular model for sequence-to-sequence tasks in deep learning. Its modularity provides a large space for architecture search, and, over the years, many variants of the Transformer have emerged. Part of the motivation behind creating Agrippa was to find a way to easily explore the design space of Transformer models. This section provides a detailed review of the architecture as well as a number of popular modifications. The experiments that follow are dependent on a careful understanding of the Transformer.

2.2.1 Base Architecture

This section is unnecessary for those familiar with the Transformer architecture, but it may serve as a good reference. I have tried to provide some additional intuition and clarity on parts that confused me when reading the original paper for the first time a little more than a year ago.

The classical Transformer has two main parts: the decoder and the encoder, which have similar structures. Each takes as input a sequence of tokens, which can be thought of as matrices with one-hot encodings for each of the tokens. Those tokens are embedded with a linear transformation to make a matrix of width equal to the dimension of the model, d_{model} (a hyperparameter), and height equal to the length of the sequence. Each row in the matrix represents a token embedding, and within the decoder, those rows are slowly morphed, mostly independently, into target token embeddings. Tokens interact with one another inside the "attention blocks" but not elsewhere. Finally, the embeddings that the decoder produces are inverted, and the result is a matrix with height equal to the length of the sequence and width equal to the size of the vocabulary such that the entries in a row can be used to estimate the probability that the row corresponds to the token represented by the column.

The ultimate goal of the encoder is to produce a matrix of embeddings that distill some knowledge about context that the decoder should act upon. The point of dividing the model into an encoder and a decoder is to allow the encoder to have no mask (each token can interact with every other token) while the decoder has a mask (each token can only see the previous tokens). For example, in the translation context, the encoder can take as input the original language while the decoder can take the target language; each place in the

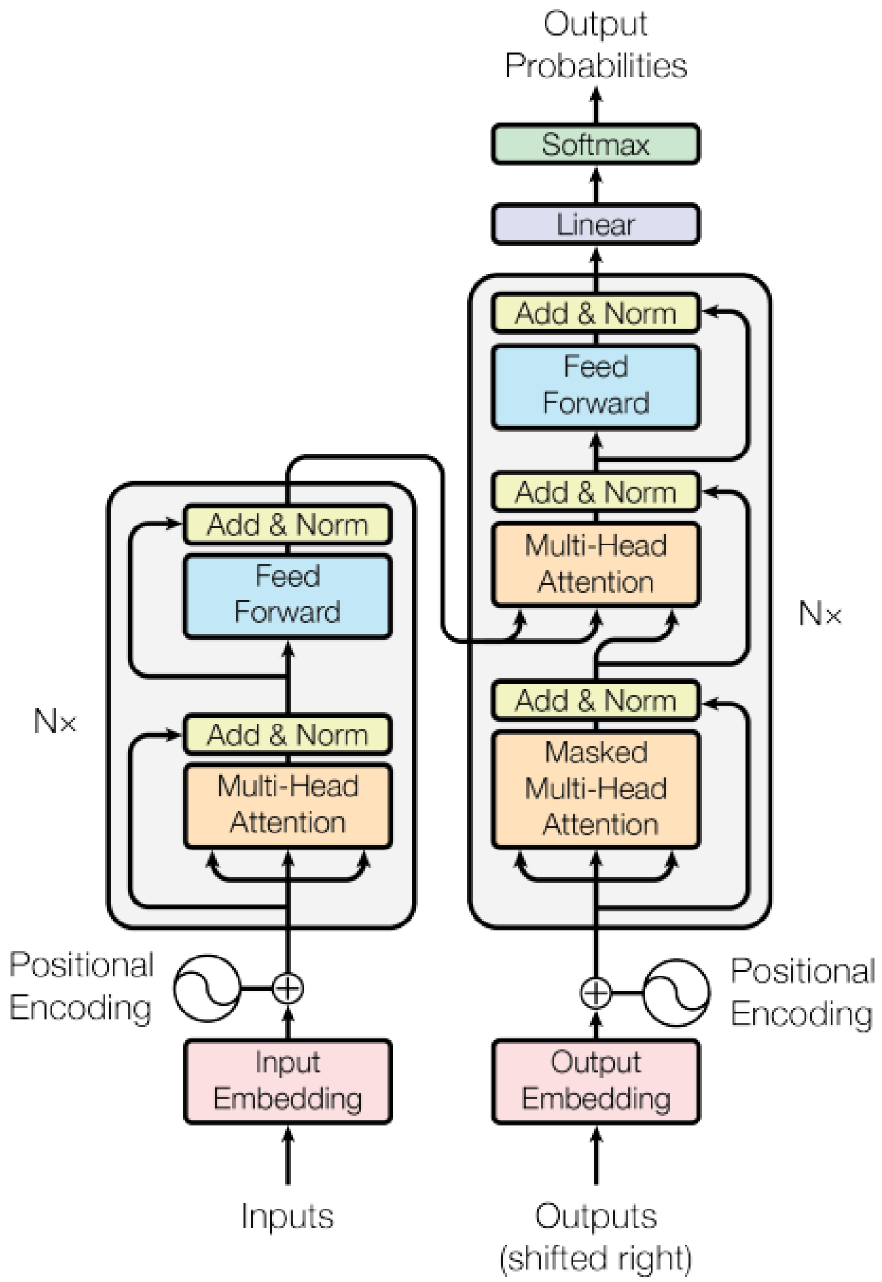


Figure 2.3: The Transformer, as seen in Vaswani et al. 2017

decoder is attempting to predict the next token, so its attention is masked, but there is no need to mask the interactions in the encoder.

It often goes unstated that none of the parameters has a shape dependent on the length of the input sequence; every place in the sequence is acted on by the same weights, though the tokens interact in the attention block. In a CNN, for example, a pattern that is learned using examples that appear in one location of an image are automatically detected everywhere in the image, due to the properties of convolutions. The analogous property of Transformers is that they can learn properties of a token or combination of tokens without seeing those tokens appear in every possible location in a sequence.

Since the model has no inherent understanding of position in a sequence, it is necessary to morph the learned token embeddings so that they carry some information about their place in the sequence. The standard way to do this is by adding a positional encoding to the token embeddings, though there are other possibilities.[*]

In both decoder and encoder, the embedding matrix passes through a "multi-head self-attention" block. In this block, token embeddings are altered by three learned linear projections into queries, keys, and values.[†] It can be thought of as "self-attention" since the three projections take the same matrix as input. Each query and key vector (that is, each row of the matrix) has its dot product taken with respect to each of the others, forming an attention matrix that is square with height and width equal to the length of the sequence (that is, its size is N^2, making this procedure quadratic in complexity). Each entry in this matrix can be thought of as an attention score between every pair of tokens. Different learned projections will result in different kinds of attention scores; one set of projections may try

[*]The reader should refer to the original paper[8] for details as well as experiments with alternative methods.
[†]Some of these projections can be the same, but they don't have to be.

to find semantically similar words, but another might seek to match nouns with their respective article. There is typically in a model always some projection that simply tries to match each token with the token immediately preceding it. This matrix of attention scores is multiplied by the values[‡] (a learned linear projection of the input embeddings). Finally, this process can be repeated across different "heads" (learned query/key/value projections), the outputs concatenated, and ultimately projected back into a matrix with width equal to d_{model}.

The output of the self-attention layer is then added to the input to the decoder layer in a residual fashion. This result is then normalized using a Layer Normalization.[9] The normalization is necessary if we want the result to have the same scale as the inputs to the residual addition. In an encoder layer, all that remains is to pass the embedding matrix through a position-wise feed-forward neural network; the result is then residually connected and normalized as in the previous step. Each layer in the encoder is stacked on top of one another, and only the final output is sent as input to the decoder.

In the decoder, following the original self-attention, the resulting embedding matrix is passed along with the encoder output into a cross-attention layer. In this layer, the previous self-attention output is projected into values while the encoder output is projected into keys and queries. The same attention operation is performed, but now the token inputs to the decoder are interacting with the outputs of the encoder rather than simply among themselves. The cross-attention output is again residually added and normalized before going through a position-wise feed-forward network, added and normalized again, and

[‡]Actually, the queries and keys are first divided by $\sqrt{d_{queries}}$ and $\sqrt{d_{keys}}$. Note also that queries and keys can exist in a space with different dimension than d_{model} (and similarly for the values). This process reduces computational complexity and promotes numerical stability.

then finally outputted to the next layer.

After the very last layer of the decoder, each token embedding (which is a vector with d_{model} elements) is up-projected into the size of the entire vocabulary (optionally using the transpose of the original embedding matrix). A final Softmax is performed before the resulting probabilities are compared with the empirical target probabilities. The loss is calculated and backpropogated through the entire model during training. During inference, the probability distribution of tokens is sampled.

A key benefit of the Transformer is that all of the tokens in a sequence are processed in parallel. This behavior is not present in RNNs, which generally process tokens sequentially. This property has allowed Transformers to be scaled to hundreds of billions of parameters trained on hundreds of billions of tokens. It might be damning that the attention mechanism is quadratic in the length of the sequence, but the length of the sequence can be small compared to the dimension of the model at scale. Moreover, every significant operation in a Transformer is a highly-optimized matrix multiply.

2.2.2 Architectural Modifications

A recurring challenge during the training of large transformers is to keep the scale of the gradients constant throughout the model. If gradient values grow too large, training can become unstable. In the original paper, the authors used a warmup stage during training: the learning rate for SGD was first raised from a low value before being decreased again. The normalizations are intended to help, but another trick was supposed to be needed in order to remove the need for warmup and make the model more natively stable. It turns out that by moving the first Layer Norm to start of a decoder/encoder layer and the second to immediately before the FFN is applied, larger models can be trained with more stability and without the warmup at all.[10] This stability might sometimes come at the cost of decreased performance.

Another solution to the same problem is to better initialize weights in deep models. DeepNet[11] scaled the initialization of the gain values of a each normalization according to the size of the model. Wang et al. also added a scaling factor to the residual addition. These tweaks permitted an increase in the depth of the Transformer by a couple of orders of magnitude and increased performance.

The training of large models is extremely costly, so architectures are chosen cautiously. A community-led project, BlOOM[12], sought to provide an open source alternative to GPT-3. The BLOOM model uses a "Stable Embedding", a layer-norm immediately following the embedding layer, again to address stability problems during training.

Other modifications are designed to decrease computational complexity. These modifications usually involve changes in the attention block. A "Reformer" Transformer replaces dot-product attention with a "locally-sensitive hashing function", reducing the complexity of the model from $O(N^2)$ to $O(NlogN)$ (where N is the length of the sequence).[13] Furthermore, the Reformer changes the computations performed in the Transformer so that each layer's activations is reversible; during training, it is therefore unnecessary to store intermediate activation values since they can be inferred by the final activations. Another Transformer variant, the "Flashformer", similarly tries to modify the attention block: it uses an operation based on Gated Linear Units and a corresponding linear approximation to reduce the complexity of attention from $O(N^2)$ to $O(N)$, though there are some complications.[14]

2.2.3 Pre-Training

Many tasks in natural language processing involve questions that tap into the same essential knowledge-base but produce different outputs. For example, a sentiment analysis program might benefit from knowing the impact of relative word positioning, which in turn might also be useful for an automatic sentence diagramming tool. Moreover, data for very specific tasks is usually limited. One way this knowledge can be shared is through pre-training. Pre-training initializes a model with weights formed from another model trained on a larger dataset and a more general task. Hopefully, some of the knowledge from the original model can be repurposed in the new model.

Perhaps the greatest triumph in pre-training is the GPT (Generative Pre-Trained Transformer) series of models. GPT is a decoder-only architecture (we do away with the encoder entirely as well as the corresponding cross-attention mechanism). The objective for the model is simply to predict the next token. We mask future tokens in the usual way, and the target tokens are simply a left-shift of the input.[§] Since there exist billions of tokens of text available on the internet, there is ample training data.

The key insight of GPT is to recognize that simple next-token prediction can require a great deal of knowledge about the world. In the original GPT paper[15], the goal was to improve performance on a variety of classic NLP problems: the authors found that by first training a GPT on a dataset of a few thousand books, the architecture could be modified slightly, fine-tuned on new problems, and could then beat the state of the art in a number of domains, such as a semantic similarity task. The following version, GPT-2[16], was found

[§]More precisely, the input is a right-shift of the target output, and the new beginning token is a special "Beginning of Sequence" token.

to learn tasks that were unexpectedly present in the dataset. For example, an article might naturally give the text of a speech in its original language followed by an English translation. Thus the model learned these tasks in a completely unsupervised fashion. The key development in GPT-3 [17] was that the language model could be given entirely new tasks with just a few examples (few-shot) and perform adequately.

While GPT-3 had clearly many capabilities, it was still essentially just trying to predict the next token. This property required tricks to produce socially desirable output, such as a friendly conversation. The primary development in AI over the past number of months has been the application of Reinforcement Learning with Human Feedback (RLHF) to large language models like GPT-3. RLHF [18] seeks to bootstrap a reward model with an LLM's pre-trained weights and train the reward model to predict a give a human-labeled score to text (in practice, this is an ELO score). Feedback from the reward model can be used to update the language model's weights using stochastic gradient *ascent*. The relevant algorithm to do the updates has been PPO [19] for the time being.

Pre-training has also been useful to bootstrap smaller models. Community-led projects have benefited from combining pre-trained models in clever ways. For example, DALLE-Mini [20] reproduces in a small package (and very cheaply) some results in image generation by using a pre-trained image autoencoder, image embedding model, text embedding model, and Transformer encoder. Only a Transformer decoder is trained.

2.3 Conclusion

This literature review has meant to be expository but also has meant to illustrate the kinds of innovations that might be easier using Agrippa. In each of these cases, models were bootstrapped using either an existing architectural idea or actual model weights. Navigating heterodoxical code written by academics is unpleasant. The declarative format promotes locality in the code, sensible visualizations, and modularity. It is difficult to tell when a researchers says the he uses a Transformer decoder whether he is also employing any number of the tricks above: novel initialization, normalization schemes, weight sharing, and the like. In a model built using Agrippa, it is straightforward to peer into the structure of a model and access any computational block. Moreover, since parameters are explicit, it is also straightforward to access model weights directly and import them into new places with confidence.

3
Technical Overview

In broad strokes, the system works like this: first, a user designs an architecture markup; second, the user exports the model using the Agrippa Python package; third, the user trains the model in PyTorch using the PyTorch conversion tool provided in the package; and lastly, the trained model weights can be exported and bundled with the architecture markup so that they model can be deployed. Optionally, a user can upload the project to the Agrippa model zoo and at any time use the associated model visualization tool to examine the architecture. Detailed documentation for each step of the process can be found at https://agrippa.build/docs.

3.1 Markup Language

The primary goal of the language is to provide a format that can be easily ingested by visualizers or other development tools. Since a full editing program is not yet developed, it is also human-readable.

First, the specification is meant to be transparent: the details of the model down to the precise computations undertaken are spelled out. Second, the language is meant to provide for a more coherent organization. All of the important details about a particular section of the model are locally concentrated. Lastly, it is meant to be portable: sections of a model can be rearranged into new models, quickly modified, and imported into other projects.

To be concrete, every model is contained inside the root model tag. Inside the model tag are the root exports and imports of the model (the outputs and inputs) and a series of other block tags. Block tags, as well as the blocks nested inside them, contain node tags, which in turn specify the precise operations. Consider a simple linear regression:

```
<model>
   <import from="x" dim="[var(n)]" />
   <block title="Linear">
      <node op="Mul" title="Apply Betas">
         <params name="W" dim="[var(n)]" />
         <input src="x" />
         <output name="wx" />
      </node>
      <node op="ReduceSum" axes="[0]" title="Sum">
         <input src="wx" />
         <output name="wx_sum" />
      </node>
      <node op="Add" title="Apply Bias">
         <params name="b" dim="[1]" />
         <input src="wx_sum" />
         <output name="y" />
      </node>
   </block>
   <export from="y" dim="[1]" />
</model>
```

Here, one root import is provided, which is a vector of variable size “n”. When exporting the model, we can supply the value of “n” in a dictionary of variable bindings. A block titled “Linear” contains all of the operations in this case. The input vector is multiplied element-wise by a vector of weights (named “W”) before being summed and added to a bias term. The export of the model is the output of the last node (titled “Apply Bias”).

Blocks are the primary structural innovation: blocks define logically coherent parts of the model and are collapsed into one unit upon visualization. In the online tool, users can click on a block to see inside, which could include more nested blocks. Blocks can also be sourced from other files using the “src” attribute. Sourced blocks are given a name that determines how you can pass inputs and use their outputs in the rest of the project.

Another useful abstraction is the usage of bindings for tag attributes. In the above example, the size of the first dimension of the input vector is bound to “n”. In a more complex situation, the dimensions of an operation could be bound to a list of variables like “[var(nbatch), var(height), var(width), var(channels)]”. Making dimensionality explicit and human-readable can greatly increase the transparency of the model.

3.2 Python Package

The Python package contains code to compile the architecture files, among some other utilities. The compiled format is an ONNX file, which can then be imported into PyTorch using the provided package. The Agrippa package can be installed with pip using "pip install agrippa". Here is some driver code to perform a linear regression:

```
import agrippa
import torch

proj_name = "lin-reg"

onnx_out = 'testing.onnx'

bindings = {
   'n': 5
}

agrippa.export(proj_name, onnx_out, bindings=bindings)
torch_model = agrippa.onnx_to_torch(onnx_out)
target_weights = torch.rand((bindings['n'],))
target_bias = torch.rand((1,))

def get_pair():
   x = torch.rand((bindings['n'],))
   y = torch.sum(target_weights * x) + target_bias
   return x, y

optimizer = torch.optim.SGD(torch_model.parameters(), lr=0.1)
loss_fn = torch.nn.MSELoss()

for _ in range(5000):
   x, y = get_pair()
   optimizer.zero_grad()
   outputs = torch_model(x)
   loss = loss_fn(outputs, y)
   loss.backward()
```

```
    optimizer.step()

agrippa.utils.save_torch_model(torch_model, proj_name)
```

This code assumes that we have placed our architecture file in a directory called "lin-reg". We set $n = 5$ and export the model to an ONNX file. The ONNX file is imported into PyTorch, and we use an ordinary PyTorch training loop to train the model using Stochastic Gradient Descent. Here, we generate the data by first randomly selecting target weights and a target bias term. After 5000 examples, the model parameters will converge to their target values. After training, we save the PyTorch model's weights (only a model first imported using the package tool will save correctly). To check that our model is correct, we can use some additional utitilies provided by the package:

```
biases = agrippa.utils.find_params("b", proj_name)
weights = agrippa.utils.find_params("W", proj_name)

print(f"Target weights: {target_weights}")
print(f"Actual weights: {weights}")
print()
print(f"Target bias: {target_bias}")
print(f"Actual bias: {biases}")
```

Here, a utility is being used to find the parameters' values from the project's weights file based on the names we assigned to those parameters.

3.3 ONNX Format

The Open Neural Network Exchange (ONNX) format developed at MIT is an open standard for machine learning interoperability.[5] ONNX files can be efficiently deployed on

Microsoft Azure using Microsoft's ONNX Runtime inference engine. In the ONNX specification, models are stored as a computational graph in which nodes specify operations. The nodes are given as input other nodes or tensors (parameters) that are stored in the file. The ONNX documentation details which operations are supported and gives their associated conventions. Many of the operations are based off of Numpy or PyTorch operations. The Agrippa node op types mostly mirror the ONNX operators.

3.4 PyTorch Import

ONNX Runtime nominally supports network training, but the support is lackluster and seems like it might be deprecated sometime soon. Therefore, utilities are provided to import ONNX into PyTorch. This support is native in some frameworks (namely TensorFlow) but not in PyTorch, though there are community tools for doing so. It should be noted that in the markup langauge, there are methods for specifying certain properties of training: for example, the "frozen" parameter changes whether a parameter should be allowed to change during training. There is no effect on the compiled ONNX file except as to modify the parameter's name to signal that it is a constant. Only by using the provided PyTorch import tool is the frozen value loaded into PyTorch as a constant buffer, not a model parameter.

3.5 Visualizer

Online at `https://agrippa.build`, users can make accounts and upload their projects. After uploading, the project can be viewed inside the Workspace, which provides a visualization of each of the architecture files. The visualization is made with respect to users' blocks.

When a block is clicked, users can see nested blocks or nodes contained inside. The following figures show what a Transformer Decoder might look like inside the tool. In the second example, the user has navigated to the attention block and selected the Softmax operation.

An offline version of the visualization tool could be added to the Python package in the future.

3.6 Zoo

The Agrippa website also contains a model zoo featuring uploaded models.* Creators can specify input and output tags that describe what their model is meant to accept as input and produce as output. For example, in the case of a Transformer, the input tags could be "text", "tokens", and "2d matrix". The output tags could be "probabilities" and "2d matrix".

One problem with existing model repositories is that there is little distinction made between very specific instantiations of a model and the more general architecture being used. For example, it is hard to find a plain CNN or GPT without a variety of tricks that tie the model close to a particular problem. Uploaders can tag their models as "canonical" if they are meant to represent a general architecture.

*Ryan Linnihan and Jared Simpson, my roommates, performed some programming work on the website. Specifically, they are responsible for input validation on user authentication requests as well as the search feature for the documentation.

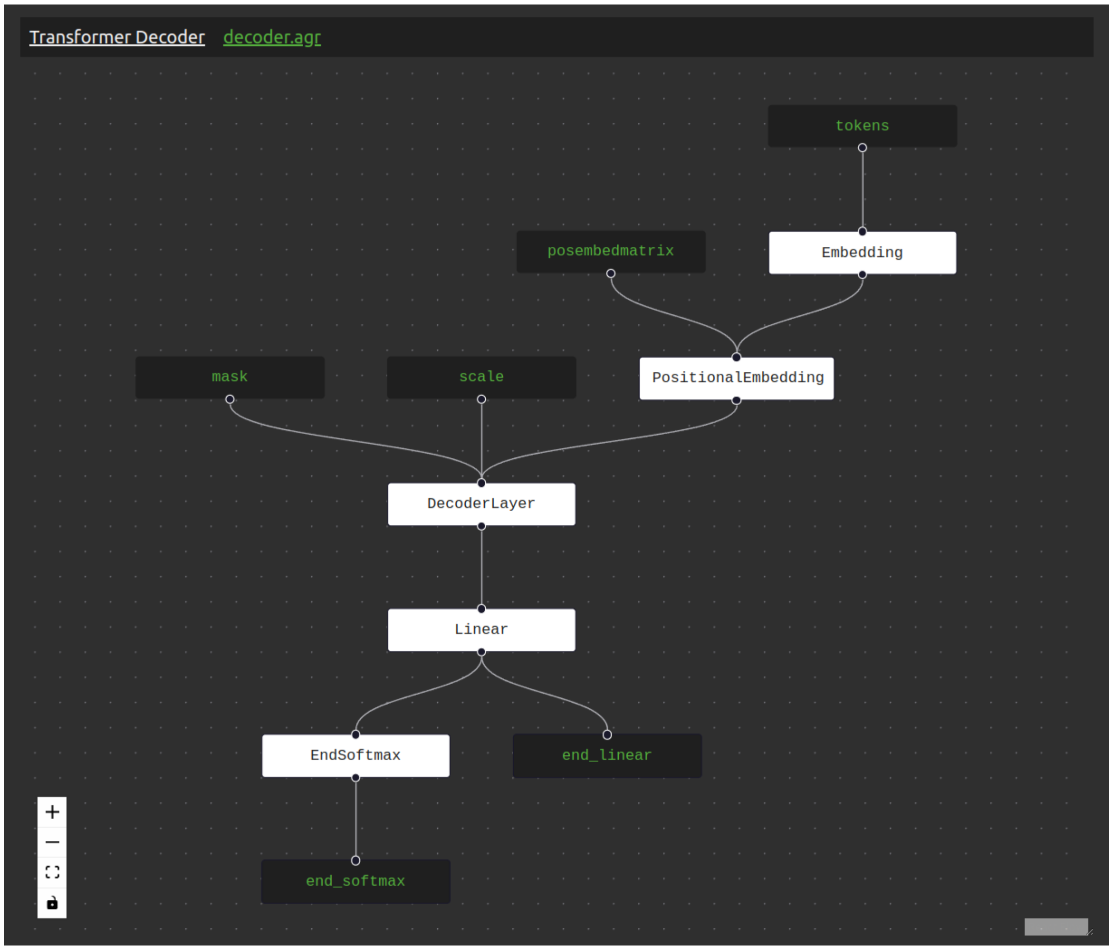
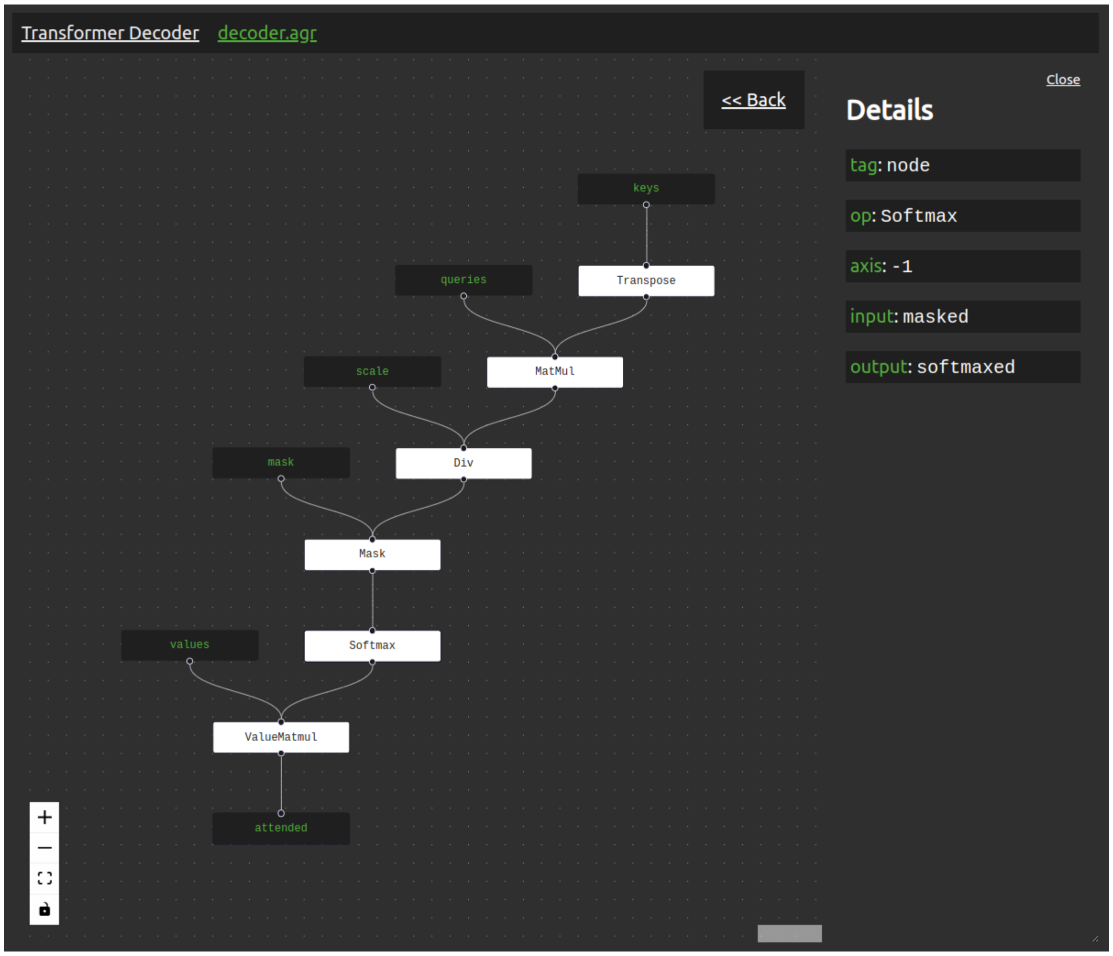


Figure 3.1: A Transformer Decoder as seen in the model visualization tool. In this implementation, the attention scalar, mask, and positional embedding matrix as inputs in addition to the tokens. The model outputs both the final linear projection as well as the Softmax of the projection.



Details

tag: node

op: Softmax

axis: -1

input: masked

output: softmaxed

Figure 3.2: An attention block inside a Transformer Decoder. In this example, the Softmax operation has been selected and its details are provided in a sidebar.

4 Experiments

Testing the system required a variety of experiments. Some are reproductions of existing work, the descriptions of which are recapitulated here for reference and additional exposition. The context embedding experiment is original. Details about the experimental setup can be found in Appendix A, and code is available along with other tests in the "tests" directory inside the Agrippa package GitHub repository[*]: https://github.com/gkamer8/agrippa-pkg.

4.1 Wikipedia GPT

First, I trained a decoder-only Transformer on a dataset of Wikipedia articles. The architecture is shown in Figure 4.1. The weights between the input embeddings and the final linear projection are shared. The input embeddings are put through a layer norm before the positional embedding is added; this technique is used in the BLOOM model.[12] The decoder layer itself uses a Pre-LN architecture.[†] You can see that in the implementation, every constant used in the model is passed as an input; this design need not be the case using the more advanced features of the language developed after this experiment was completed.[‡] The dataset used is available in Appendix A.

Over-parameterized machine learning models are usually considered a black box. However, there is some structure that allows an analysis of some of the interior. When testing the model, I found that the model often continues the sentence "Abraham Lincoln was an American layer, politician, and a" with the token, "Republican". Because the model is small and only saw the Abraham Lincoln Wikipedia page a few times in its dataset, I was surprised to see that the model had seemingly memorized the fact that Abraham Lincoln was a Republican.

There are two questions we might like to ask: does the model really know that Lincoln was a Republican, and if so, what weights are responsible? The residual addition as well as

[*]The relevant directory is tests/transformers/{embed, exp-wiki, translate2}

[†]Anecdotally, the precise variant of Transformer did not seem to affect performance too much in this case. The model and dataset are both unambitious.

[‡]The future experiments do not have this feature.

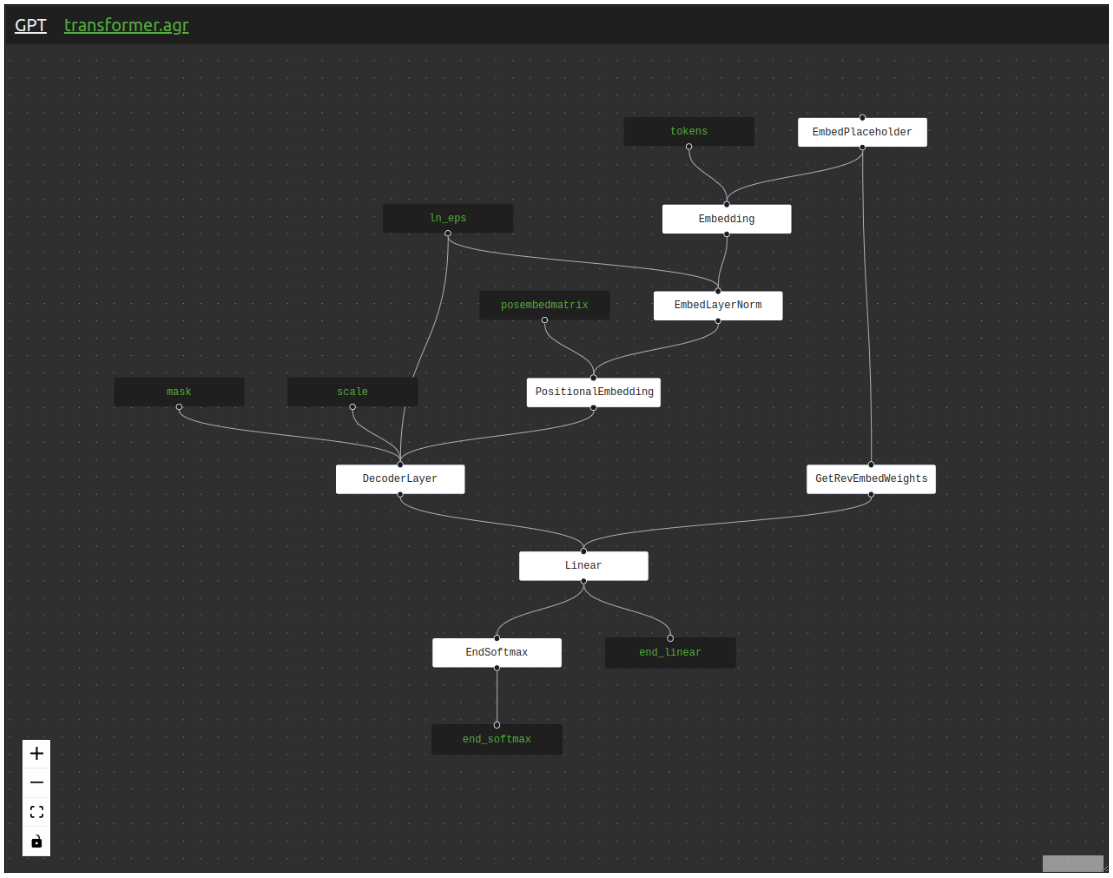


Figure 4.1: The architecture used in the Wikipedia GPT experiment, seen using the Agrippa visualization tool

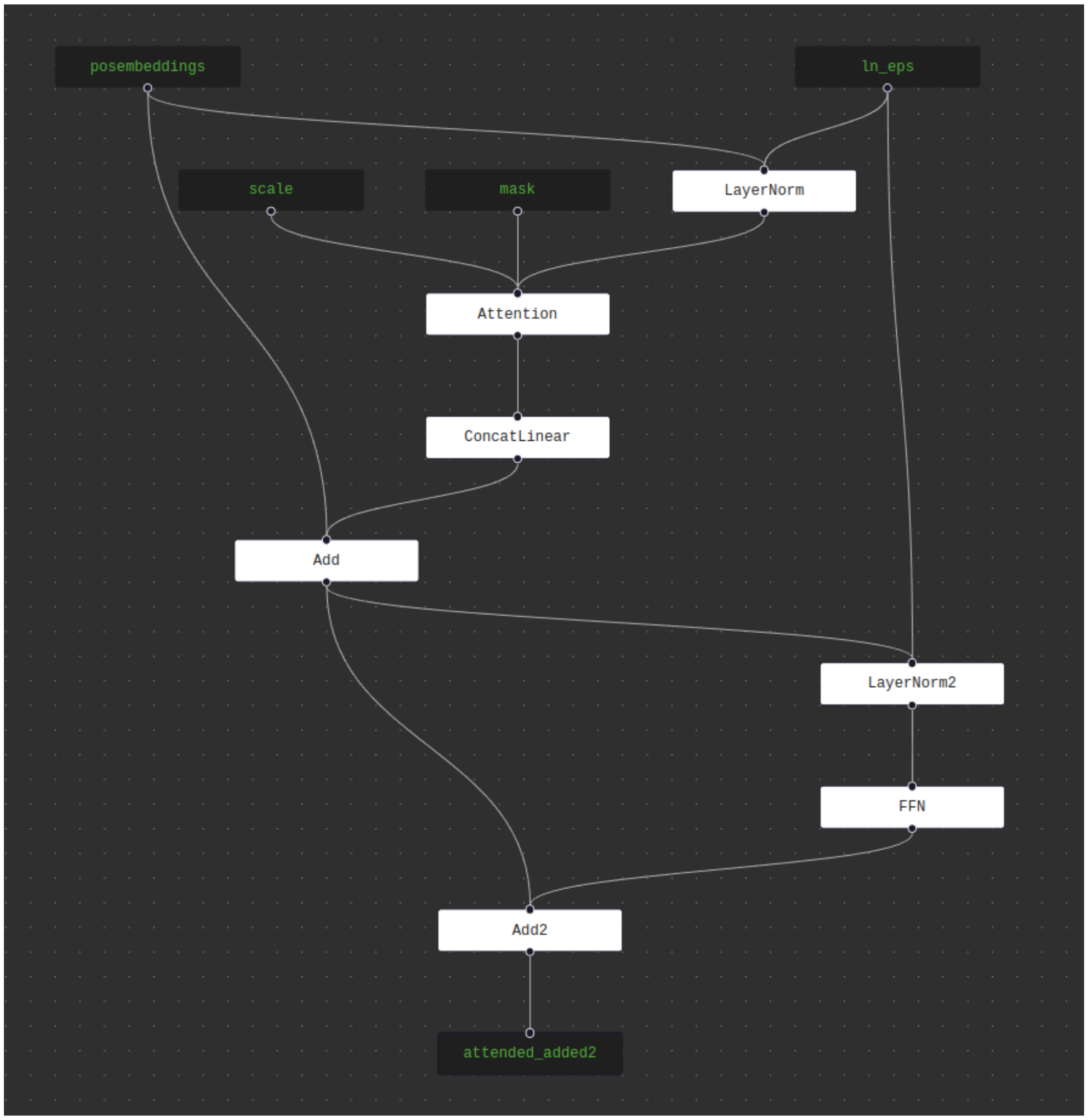


Figure 4.2: The DecoderLayer used in the Wikipedia GPT experiment, seen using the Agrippa visualization tool

Layer	Republican	person	Democrat	the	eos
1	0.0	0.0	0.0	0.0	0.56
2	0.0	0.0	0.0	0.0	0.86
3	2e-5	1e-5	0.0	0.0	0.95
4	1.3e-4	3e-5	2e-5	0.0	0.87
5	0.0013	7.4e-4	1.6e-4	0.0	0.79
6	0.027	0.0027	0.006	0.0	5e-5

Figure 4.3: The predicted probability for each token at each layer in the model (the Softmax'd component of each token). The sentence begins, "Abraham Lincoln was an American lawyer, politician, and a"

Layer	Republican	person	Democrat	the	eos
1	0.0	0.0	0.0	0.0	0.57
2	0.0	0.0	0.0	0.0	0.89
3	2e-5	1e-5	0.0	0.0	0.96
4	1.7e-4	4e-5	2e-5	0.0	0.86
5	0.0013	9.4e-4	2.1e-4	0.0	0.78
6	0.038	0.0056	0.0054	0.0	8e-5

Figure 4.4: The predicted probability for each token at each layer in the model (the Softmax'd component of each token). The sentence begins, "Joe Biden was an American lawyer, politician, and a"

the embedding weight sharing gives an inductive bias toward minor modifications of the tokens as they traverse each layer in the model. We can apply the reverse embedding and Softmax at each intermediate layer to see the status of the generation. This modification is relatively straightforward using Agrippa. These values are given in figures 4.3 and 4.4. Since the token begins as the end of sequence token (appearing in the column marked "eos"), the value for end of sequence is high until the final layer. Other positions in the sequence depend on information given in the original token, so that its value is high is not a surprise; however, it might be surprising to note that it is so high even up to the last layer.

Instead of just looking at the word "Republican", we can examine its predicted probability relative to other words that might fit, such as "Democrat" and "person".[§] The token

[§]Other tokens were examined, but "person" seemingly worked well as a control.

"the" is also included as an example of a common token that does not fit the context, forming a kind of control token. In the Lincoln case, the Republican token first appears significant after layer 3, but increases 20 fold in the final layer. The probability associated with "Democrat" is lower and first appears later in layer 4. Probabilities for the control token "person" were slightly less than for "Republican" until layer 5 and similarly for Democrat.

Has the model actually learned that Lincoln was a Republican? I test this idea by comparing the model dynamics to a similar sentence: "Joe Biden was an American lawyer, politician, and a". In this case, the model also predicts "Republican" with high probability and "Democrat" with about the same probability as before. The internal layer activations are also extremely similar; at layer 5, the relative probabilities for "Republican" are the same to four decimals between the Lincoln and Biden sentences. It seems likely the model instead learned to associate the names Joe Biden and Abraham Lincoln to political contexts, or the dataset included a lot of articles about Republicans.

4.2 En-De Translation

A traditional encoder-decoder Transformer was also tested. This time, I used the experiment setup of the original paper and tested English to German translation on a standard dataset. The model seen here is the "base" version. The performance of this model did not reach the original's performance, though some tricks were left out, like dropout layers and beam decoding. The performance was not good enough to merit a reasonable BLEU score. However, it appears that the model can write German that is somewhat related to the English.¶

¶Part of the problem might be that the dataset seemed to have many mismatched pairs (on the order of 1 in 20). The English to French dataset was even worse: sometimes passages appeared in Russian, and obvious

Original English

I declare resumed the session of the European Parliament adjourned on Friday 17 December 1999, and I would like once again to wish you a happy new year in the hope that you enjoyed a pleasant festive period.

German Label

Ich erkläre die am Freitag, dem 17. Dezember unterbrochene Sitzungsperiode des Europäischen Parlaments für wiederaufgenommen, wünsche Ihnen nochmals alles Gute zum Jahreswechsel und hoffe, daß Sie schöne Ferien hatten.

Sampled German

Ich erinnere Sie daran, dass das Europäische Parlament am Montag, dem 15. Oktober, eine Sitzungsperiode abgehalten hat, und ich hoffe, dass Sie eine Sitzungsperiode für das kommende Jahr haben werden, um eine Sitzung des Parlaments zu veranstalten.

Figure 4.5: Sample output from the English to German translation model using greedy decoding.

Appearing in figure 4.5 is a sample translation from the test set. The model starts correctly, matching "I" to "Ich" and also gives the correct second token, "er", the beginning of the word "erkläre". However, the second phoneme of "erkläre" is incorrectly given as "innere". The word "erinnere" can be translated as "remind", "remember", or "recall".

The model learns that days of the week in English should match days of the week in German, but it matches the wrong one: "Friday" in German is "Freitag"; "Montag" is "Monday". Moreover, the date is wrong: the model translates "17 December" as "15. Oktober". Clearly the model learns some German, but the faithfulness of the translation is poor.

problems were spotted in something on the order of 1 in 5 examples. It is unclear to me whether this problem exists in the WMT 2014 dataset or just the version available on HuggingFace, which was the one used here.

4.3 Context Embedding Model

Transformers improve on RNNs by processing a sequence in parallel; however, the computational complexity of RNNs is linear in the length of the sequence, while Transformers' complexity is quadratic. The context lengths of even the most ambitious models are only a few thousand tokens. If Transformers are to draw inferences from texts of, say, a book length, there needs to be some way to compress the context in a way a Transformer can understand. There is some work on addressing this issue, usually by overhauling the attention mechanism[13] [14] or by storing in some way previous activations recurrently[21] [22].

In this section, I experiment with a novel approach that encodes a sequence into one embedding (a vector of length equal to the dimension of the model). That embedding provides conditioning information to a decoder, effectively increasing its context length. During training, the context embedding is prepended to a decoder's input, and the decoder is meant to generate the original text given the embedding. The embedding is the last output of the encoder, which corresponds to a special summarize token. The encoder and decoder each have the same architecture, which is simply the GPT architecture used in the above Wikipedia experiment.[‖] Any ordinary Transformer could be fine-tuned using this procedure to produce or use document embeddings; there are no architectural modifications.

There are a few tricks in the implementation that might be worth explaining. The output from the encoder is a matrix with rows corresponding to each token and columns corresponding to each dimension of the model. That output is masked using element-wise multiplication. Here, the mask contains 0's in all places except the last row. The decoder's

[‖]However, the encoder's mask is all zeros, while the decoder's mask is not. Also, this model scales input embeddings by $\sqrt{d_{model}}$ as in the original Transformer paper. Details are in Appendix A.

= Robert Boulter =
Robert Boulter is an English film , television and theatre actor . He had a guest @-@ starring role on the television series The Bill in 2000 . This was followed by a starring role in the play Herons written by Simon Stephens , which was performed in 2001 at the Royal Court Theatre . He had a guest role in the television series Judge John Deed in 2002 . In 2004 Boulter landed a role as " Craig " in the episode "

Figure 4.6: Example from the Wikitext dataset

embeddings are masked using the complementary matrix (following the input embedding step) and added to the masked encoder output. The positional embeddings are modified so that the final row has the embedding it would have if it were first in the sequence, and the decoder's mask is similarly modified such that every token can see the final one (the document embedding) but the document embedding cannot see any other embeddings. It is possible using this implementation to allow the encoder to output any number of embeddings in any places and for the decoder to properly receive those embeddings, provided the user sets up the positional embeddings and masks correctly.[**]

Figures 4.6 through 4.8 show some sample input and output from a checkpoint of the model. First, we will look at the output from the decoder for an example in the test set, given the encoder output for that example (just as would appear in training). Next, we will feed the decoder the correct text (about British actor Robert Boulter) but the wrong embedding. The "anti-example" is from the Wikipedia article for Japanese Ise-class Battleships from World War 2.

First we can note that the model, with the correct embedding, predicted the word "actor" before the built-up context had mentioned anything about acting. In the version with the wrong embedding, the model produced "naval" instead. Moreover, with the wrong

[**]The implementation is provided in the Agrippa model zoo.

. The sisters were then reduced to reserve until they were sunk during American airstrikes in July . After the war they were scrapped in 1946 – 47 .
= = Background = =
The design of the Fusō @-@ class battleships was shaped both by the ongoing international naval arms race and a desire among Japanese naval planners to maintain a fleet of capital ships powerful enough to defeat the United States Navy in an encounter in Japanese territorial waters . The IJN 's fleet of battles

Figure 4.7: Anti-example from the Wikitext dataset

Actual Token	Top Prediction	w/ Wrong Embedding
Robert	Robert	“
B	B	B
oul	oul	oul
ter	ter	ter
is	is	was
an	a	considered
English	American	American
film	actor	naval
,	actor	that
television	directed	but
and	and	note
theater	television	it
actor	star	not

Figure 4.8: Output of the context embedding decoder given masked tokens from the example, with either the matching embedding from the encoder or the embedding from the anti-example. (tokens 8-20)

embedding, the model used the past tense verbs “was” and “considered” while the correct model predicted present-tense verbs, matching their respective embedding contexts.

One potential application for these context embeddings is to use them as measures of semantic similarity between text for search or other tasks. Unfortunately, the cosine similarity between embeddings does not seem to be a good measurement of semantic similarity. Using the STSBenchmark test dataset[23], context embedding cosine similarity scores have a 0.27 Pearson correlation with the label scores.

While these embeddings might not function well at the moment as search embeddings, the fact remains that the embedding gives successfully helps model perplexity.[††] Moreover, if one finds that it is not effective for longer sequences to compress context into one token of information, it is possible that the decoder could be trained to glean information from multiple embeddings, each for some small part of the context (perhaps 100 tokens or so). Future work on larger models should consider using this type of embedding to increase effective context without increasing computational cost.

[††]Training graphs are available in Appendix A. There is some ambiguity here over how to compare this model and the original Wikipedia GPT experiment. It would be strange if, given the context embedding over nothing, the decoder performed worse. Moreover, the previous experiment operated on a different sequence length and dataset. A proper benchmark might involve testing this approach against other context-compressors in the literature, though no other method is exactly analogous. Further work is needed to evaluate this method.

5 Conclusion

This paper presents a first step in a quest for a better framework for working with large machine learning models. The dominant imperative approach does not seem to match the way the models are actually presented. Model architectures are reasoned about like graphs. Unlike a regular computer program, researchers want to swap out chunks of that graph, and doing so should not cause the model to cease functioning. Moreover, researchers often want associated data from files to remain mapped to part of the program. This behavior is not what is optimized for in the imperative context. One can almost think of the Agrippa style like using HTML, CSS, and Javascript: XML (HTML) to define an architecture, a weights file (CSS) to give non-structural information about the architecture, and Python (Javascript) to drive the model.

Before this system can be competitive with other modern approaches, a variety of problems must be dealt with. Some of these problems can be fixed with time, but others might guide researchers toward related but separate systems.

With more work, visual editing of the markup should be straightforward. More operations from the ONNX specification can be added, including the ONNX feature that allows users to define their own operations. Simpler methods for Dropout layers and Batch normalization should also be added. ONNX has a feature that permits models to have separate behavior for inference and training; this distinction could be added, though it could complicate the PyTorch conversion process. The frozen weight feature is an example of a similar modification that is already implemented.

Researchers might also be able to fix deeper flaws with the system developed here. For example, the "rep" and "stretch" attributes (allowing simple repeated blocks) greatly complicates the compilation and leads to some unintuitive behavior. It might be better to simply let users explicitly define loop constructs themselves, such as in System Verilog, that are meant to be unrolled during compilation.

In the long term, future work could turn a system like this into a proper IDE for AI models. In addition to web development (with its complicated stack of development environments), computer graphics might offer a good analogy. Programs like Xcode combine linkers, graphics files, code, and visualizations to make designing applications on OSX and iOS significantly easier. If the architecture is analogous to a frontend layout, then a markup language would be a better way to specify it. Python packages alone do not seem to be an adequate tool for integrating the spiderweb of dependencies in a state of the art model.

A
Experiment Details

A.1 Initialization

In every experiment, Layer Normalization gain parameters were initialized to 1 and bias parameters to 0. All other parameters were initialized using Xavier initialization.

A.2 Token Embeddings

Input and output embeddings were shared in all cases. Furthermore, the input embeddings (but not the output embeddings) were scaled by a factor of $\sqrt{d_{model}}$ in the translation and context embedding experiments. This mimics the procedure in the original paper. All models were trained using the GPT-2 Tokenizer.

A.3 Hyperparameters

"Embed" is the context embedding model, "Wikipedia" is the GPT trained on Wikipedia, and "Translate" is the En-De translation model.

Hyperparameter	Embed	Translate	Wikipedia
Max seq length	100	100	64
d_{model}	512	512	1024
d_{values}	64	64	64
$d_{queries}$	64	64	64
d_{keys}	64	64	64
$nheads$	8	8	16
$nlayers$	6	6	6
FFN hidden dim	2048	2048	4096
Full batch size	640*	256	1000*
Warmup	4000	4000	2000

The learning rate decay schedule in all experiments is the same as the "Noam (Shazeer)" schedule found in the original Transformer paper. In the case of the context embedding experiment, the batch size was increased from 256 to 640 after 5 epochs. For the Wikipedia GPT, the batch size was increased from 400 to 1000 after 8000 optimization steps.

A.4 Datasets

The context embedding experiment used a dataset of English Wikipedia available on Hugging Face[24]. The Wikipedia GPT used a slightly different dataset of Wikipedia articles with different formatting, available at this link: `https://huggingface.co/datasets/lucadiliello/english_wikipedia/tree/main/data`. The files used from that link were 5, 7, 8, 9, 10, 12, 13, 14, 16, 19, and 20; others were not used due to hardware and time constraints.

The En-De translation dataset was WMT 14, which was accessed using HuggingFace datasets: `https://huggingface.co/datasets/wmt14`.

A.5 Hardware

All experiments were conducted on 1 NVIDIA A100 (40gb, variably PCIe or SXM4) rented from the Lambda Labs GPU cloud.

A.6 Training Graphs

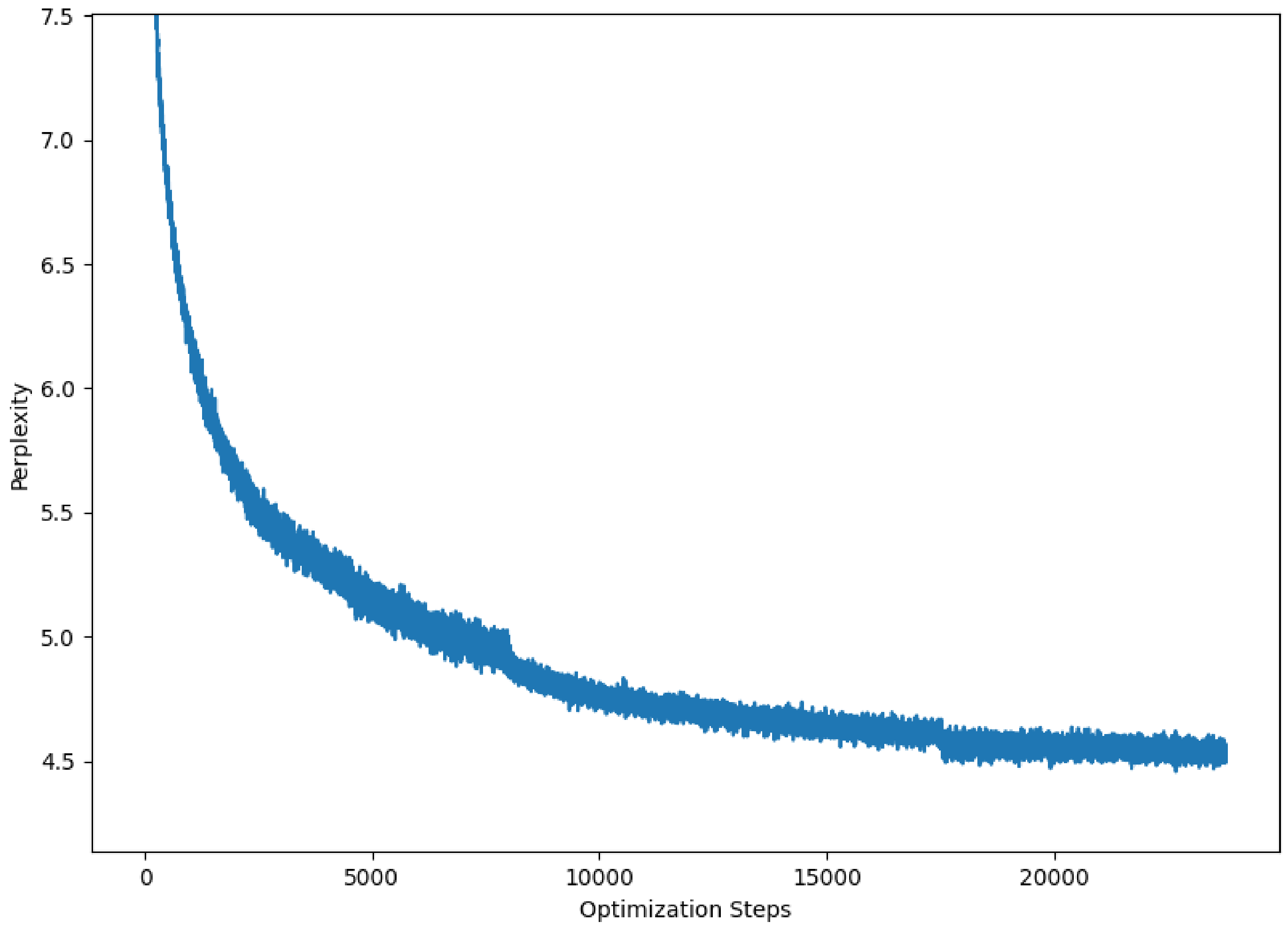


Figure A.1: Training loss for the Wikipedia GPT experiment

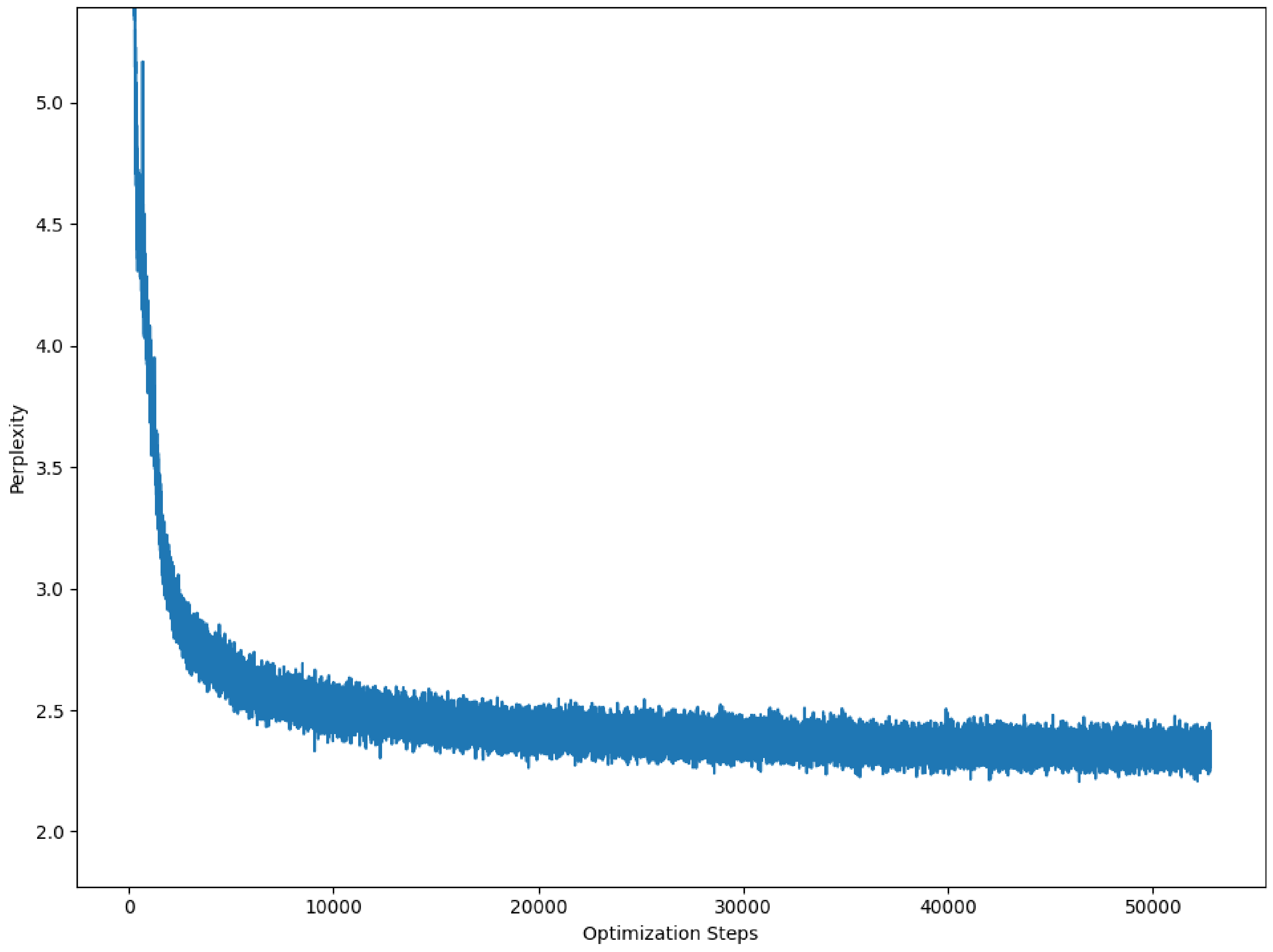


Figure A.2: Training loss for the En-De translation experiment

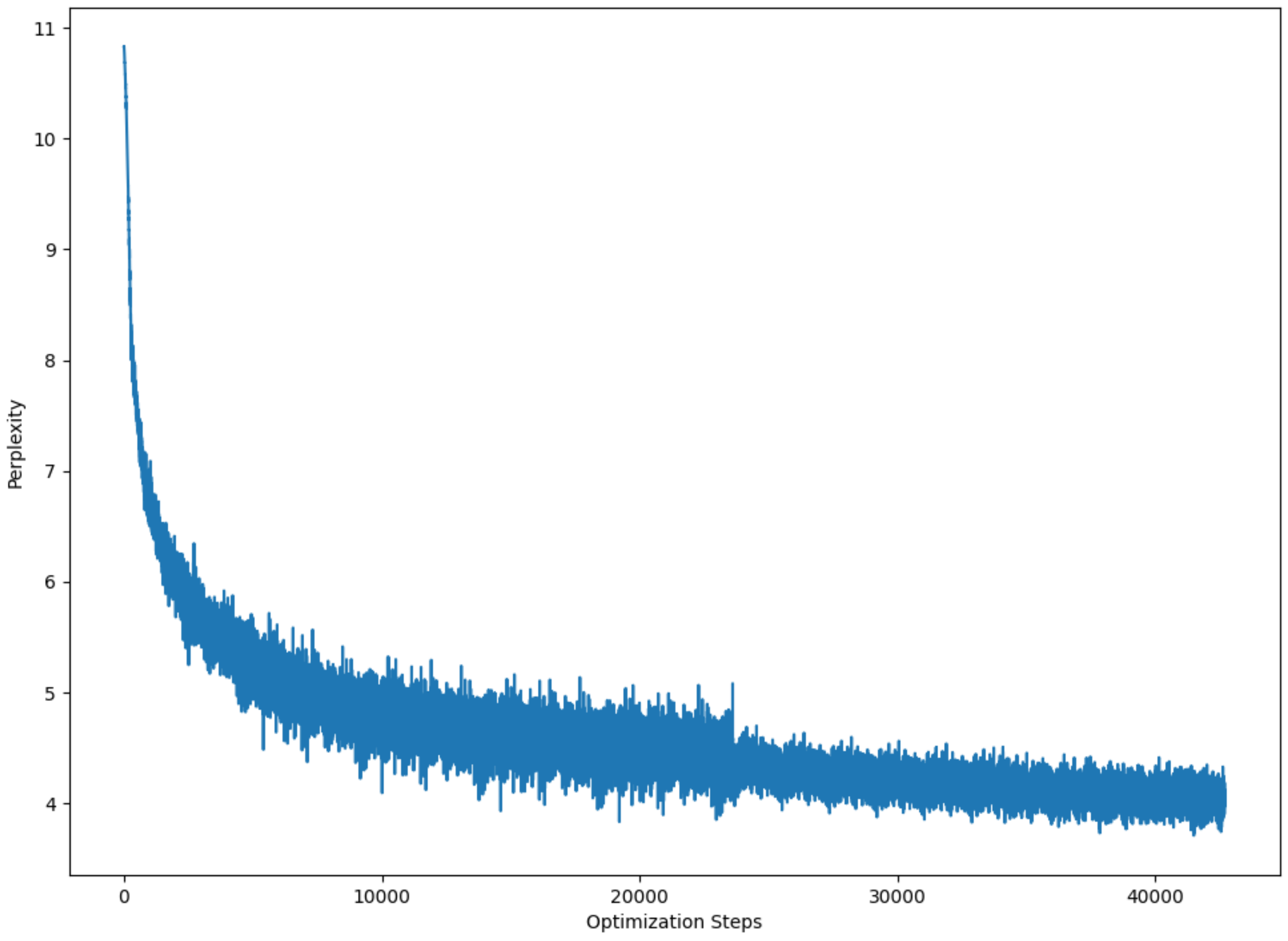


Figure A.3: Training loss for the context embedding experiment

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," Dec. 2019. arXiv:1912.01703 [cs, stat].

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, and M. Devin, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[3] R. Frostig, M. J. Johnson, and C. Leary, "Compiling machine learning programs via high-level tracing,"

[4] "ONNX: Open standard for machine learning interoperability."

[5] "Faster Scalable ML Model Deployment Using ONNX and Open Source Tools," in *2020 IEEE Infrastructure Conference*, pp. i–i, Oct. 2020.

[6] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg, "Visualizing Dataflow Graphs of Deep Learning Models in TensorFlow," *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, pp. 1–12, Jan. 2018. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[7] A. Bäuerle, C. van Onzenoodt, and T. Ropinski, "Net2Vis – A Visual Grammar for Automatically Generating Publication-Tailored CNN Architecture Visualizations," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, pp. 2980–2991, June 2021. Conference Name: IEEE Transactions on Visualization and Computer Graphics.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," Dec. 2017. arXiv:1706.03762 [cs].

[9] J. L. Ba, J. R. Kiros, and G. E. Hinton, “Layer Normalization,” July 2016. arXiv:1607.06450 [cs, stat].

[10] R. Xiong, Y. Yang, D. He, K. Zheng, S. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T.-Y. Liu, “On Layer Normalization in the Transformer Architecture,” June 2020. arXiv:2002.04745 [cs, stat].

[11] H. Wang, S. Ma, L. Dong, S. Huang, D. Zhang, and F. Wei, “DeepNet: Scaling Transformers to 1,000 Layers,” Mar. 2022. arXiv:2203.00555 [cs].

[12] B. Workshop, T. L. Scao, A. Fan, C. Akiki, E. Pavlick, S. Ilić, D. Hesslow, R. Castagné, A. S. Luccioni, F. Yvon, M. Gallé, J. Tow, A. M. Rush, S. Biderman, A. Webson, P. S. Ammanamanchi, T. Wang, B. Sagot, N. Muennighoff, A. V. del Moral, O. Ruwase, R. Bawden, S. Bekman, A. McMillan-Major, I. Beltagy, H. Nguyen, L. Saulnier, S. Tan, P. O. Suarez, V. Sanh, H. Laurençon, Y. Jernite, J. Launay, M. Mitchell, C. Raffel, A. Gokaslan, A. Simhi, A. Soroa, A. F. Aji, A. Alfassy, A. Rogers, A. K. Nitzav, C. Xu, C. Mou, C. Emezue, C. Klamm, C. Leong, D. van Strien, D. I. Adelani, D. Radev, E. G. Ponferrada, E. Levkovizh, E. Kim, E. B. Natan, F. De Toni, G. Dupont, G. Kruszewski, G. Pistilli, H. Elsahar, H. Benyamina, H. Tran, I. Yu, I. Abdulmumin, I. Johnson, I. Gonzalez-Dios, J. de la Rosa, J. Chim, J. Dodge, J. Zhu, J. Chang, J. Frohberg, J. Tobing, J. Bhattacharjee, K. Almubarak, K. Chen, K. Lo, L. Von Werra, L. Weber, L. Phan, L. B. allal, L. Tanguy, M. Dey, M. R. Muñoz, M. Masoud, M. Grandury, M. Šaško, M. Huang, M. Coavoux, M. Singh, M. T.-J. Jiang, M. C. Vu, M. A. Jauhar, M. Ghaleb, N. Subramani, N. Kassner, N. Khamis, O. Nguyen, O. Espejel, O. de Gibert, P. Villegas, P. Henderson, P. Colombo, P. Amuok, Q. Lhoest, R. Harliman, R. Bommasani, R. L. López, R. Ribeiro, S. Osei, S. Pyysalo, S. Nagel, S. Bose, S. H. Muhammad, S. Sharma, S. Longpre, S. Nikpoor, S. Silberberg, S. Pai, S. Zink, T. T. Torrent, T. Schick, T. Thrush, V. Danchev, V. Nikoulina, V. Laippala, V. Lepercq, V. Prabhu, Z. Alyafeai, Z. Talat, A. Raja, B. Heinzerling, C. Si, D. E. Taşar, E. Salesky, S. J. Mielke, W. Y. Lee, A. Sharma, A. Santilli, A. Chaffin, A. Stiegler, D. Datta, E. Szczechla, G. Chhablani, H. Wang, H. Pandey, H. Strobelt, J. A. Fries, J. Rozen, L. Gao, L. Sutawika, M. S. Bari, M. S. Al-shaibani, M. Manica, N. Nayak, R. Teehan, S. Albanie, S. Shen, S. Ben-David, S. H. Bach, T. Kim, T. Bers, T. Fevry, T. Neeraj, U. Thakker, V. Raunak, X. Tang, Z.-X. Yong, Z. Sun, S. Brody, Y. Uri, H. Tojarieh, A. Roberts, H. W. Chung, J. Tae, J. Phang, O. Press, C. Li, D. Narayanan, H. Bourfoune, J. Casper, J. Rasley, M. Ryabinin, M. Mishra, M. Zhang, M. Shoeybi, M. Peyrounette, N. Patry, N. Tazi, O. Sanseviero, P. von Platen, P. Cornette, P. F. Lavallée, R. Lacroix, S. Rajbhandari, S. Gandhi, S. Smith,

S. Requena, S. Patil, T. Dettmers, A. Baruwa, A. Singh, A. Cheveleva, A.-L. Ligozat, A. Subramonian, A. Névéol, C. Lovering, D. Garrette, D. Tunuguntla, E. Reiter, E. Taktasheva, E. Voloshina, E. Bogdanov, G. I. Winata, H. Schoelkopf, J.-C. Kalo, J. Novikova, J. Z. Forde, J. Clive, J. Kasai, K. Kawamura, L. Hazan, M. Carpuat, M. Clinciu, N. Kim, N. Cheng, O. Serikov, O. Antverg, O. van der Wal, R. Zhang, R. Zhang, S. Gehrmann, S. Mirkin, S. Pais, T. Shavrina, T. Scialom, T. Yun, T. Limisiewicz, V. Rieser, V. Protasov, V. Mikhailov, Y. Pruksachatkun, Y. Belinkov, Z. Bamberger, Z. Kasner, A. Rueda, A. Pestana, A. Feizpour, A. Khan, A. Faranak, A. Santos, A. Hevia, A. Unldreaj, A. Aghagol, A. Abdollahi, A. Tammour, A. HajiHosseini, B. Behroozi, B. Ajibade, B. Saxena, C. M. Ferrandis, D. Contractor, D. Lansky, D. David, D. Kiela, D. A. Nguyen, E. Tan, E. Baylor, E. Ozoani, F. Mirza, F. Ononiwu, H. Rezanejad, H. Jones, I. Bhattacharya, I. Solaiman, I. Sedenko, I. Nejadgholi, J. Passmore, J. Seltzer, J. B. Sanz, L. Dutra, M. Samagaio, M. Elbadri, M. Mieskes, M. Gerchick, M. Akinlolu, M. McKenna, M. Qiu, M. Ghauri, M. Burynok, N. Abrar, N. Rajani, N. Elkott, N. Fahmy, O. Samuel, R. An, R. Kromann, R. Hao, S. Alizadeh, S. Shubber, S. Wang, S. Roy, S. Viguier, T. Le, T. Oyebade, T. Le, Y. Yang, Z. Nguyen, A. R. Kashyap, A. Palasciano, A. Callahan, A. Shukla, A. Miranda-Escalada, A. Singh, B. Beilharz, B. Wang, C. Brito, C. Zhou, C. Jain, C. Xu, C. Fourrier, D. L. Periñán, D. Molano, D. Yu, E. Manjavacas, F. Barth, F. Fuhrimann, G. Altay, G. Bayrak, G. Burns, H. U. Vrabec, I. Bello, I. Dash, J. Kang, J. Giorgi, J. Golde, J. D. Posada, K. R. Sivaraman, L. Bulchandani, L. Liu, L. Shinzato, M. H. de Bykhovetz, M. Takeuchi, M. Pàmies, M. A. Castillo, M. Nezhurina, M. Sänger, M. Samwald, M. Cullan, M. Weinberg, M. De Wolf, M. Mihaljcic, M. Liu, M. Freidank, M. Kang, N. Seelam, N. Dahlberg, N. M. Broad, N. Muellner, P. Fung, P. Haller, R. Chandrasekhar, R. Eisenberg, R. Martin, R. Canalli, R. Su, R. Su, S. Cahyawijaya, S. Garda, S. S. Deshmukh, S. Mishra, S. Kiblawi, S. Ott, S. Sang-aroonsiri, S. Kumar, S. Schweter, S. Bharati, T. Laud, T. Gigant, T. Kainuma, W. Kusa, Y. Labrak, Y. S. Bajaj, Y. Venkatraman, Y. Xu, Y. Xu, Y. Xu, Z. Tan, Z. Xie, Z. Ye, M. Bras, Y. Belkada, and T. Wolf, "BLOOM: A 176B-Parameter Open-Access Multilingual Language Model," Dec. 2022. arXiv:2211.05100 [cs].

[13] N. Kitaev, �. Kaiser, and A. Levskaya, "Reformer: The Efficient Transformer," Feb. 2020. arXiv:2001.04451 [cs, stat].

[14] W. Hua, Z. Dai, H. Liu, and Q. V. Le, "Transformer Quality in Linear Time," June 2022. arXiv:2202.10447 [cs].

[15] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving Language Understanding by Generative Pre-Training,”

[16] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,”

[17] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language Models are Few-Shot Learners,” July 2020. arXiv:2005.14165 [cs].

[18] Y. Bai, A. Jones, K. Ndousse, A. Askell, A. Chen, N. DasSarma, D. Drain, S. Fort, D. Ganguli, T. Henighan, N. Joseph, S. Kadavath, J. Kernion, T. Conerly, S. El-Showk, N. Elhage, Z. Hatfield-Dodds, D. Hernandez, T. Hume, S. Johnston, S. Kravec, L. Lovitt, N. Nanda, C. Olsson, D. Amodei, T. Brown, J. Clark, S. McCandlish, C. Olah, B. Mann, and J. Kaplan, “Training a Helpful and Harmless Assistant with Reinforcement Learning from Human Feedback,” Apr. 2022. arXiv:2204.05862 [cs].

[19] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, “Proximal Policy Optimization Algorithms,” Aug. 2017. arXiv:1707.06347 [cs].

[20] B. Dayma and P. Cuenca, “DALL·E Mini – Generate Images From Any Text Prompt,” Oct. 2022.

[21] J. W. Rae, A. Potapenko, S. M. Jayakumar, and T. P. Lillicrap, “Compressive Transformers for Long-Range Sequence Modelling,” Nov. 2019. arXiv:1911.05507 [cs, stat].

[22] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. Le, and R. Salakhutdinov, “Transformer-XL: Attentive Language Models beyond a Fixed-Length Context,” *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pp. 2978–2988, 2019. Conference Name: Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics Place: Florence, Italy Publisher: Association for Computational Linguistics.

[23] D. Cer, M. Diab, E. Agirre, I. Lopez-Gazpio, and L. Specia, “SemEval-2017 Task 1: Semantic Textual Similarity Multilingual and Cross-lingual Focused Evaluation Proceedings of the 10th International Workshop on Semantic Evaluation,” in *SemEval-*

2017 Task 1: Semantic Textual Similarity Multilingual and Cross-lingual Focused Evaluation.

[24] S. Merity, C. Xiong, J. Bradbury, and R. Socher, "Pointer sentinel mixture models," 2016.