Assignment 2

Deadline: November 20, 2025 by 2pm.
Assignment structure: There are two sections of problems. One includes written problems. The other set involves programming. More information about these are in the Notebook "nmt.ipynb".
Submission:

- You must submit your solutions of written problems as a PDF file through **GradeScope**. Note that the PDF file of written problems needs to be concatenated with the PDF file generated from the Notebook before being submitted to GradeScope. You may access the GradeScope page through the navigation bars of the CourseWorks page. Please make sure you link the solutions and questions correctly to receive credits on Gradescope. Please include justification for all your answers - an answer with no work shown will not receive credit.
- The `hw2_code_<YOUR_UNI>.ipynb` iPython Notebook, where `<YOUR_UNI>` is your uni. This file needs to be submitted to the **CourseWorks assignment page**.

Please read the following instructions before attempting the assignment.

Collaboration Policy: You are welcome to work together with other students on the homework. However, you must write up your solutions to the assignment individually. You are also welcome to use online resources that you find helpful (e.g. tutorials, course notes, textbooks, etc.). However, directly seeking and copying/paraphrasing answers or hints for the homework questions is prohibited. If you submit an assignment that contains copied/paraphrased text or code, either from someone else or online, you will receive a 0 for the homework. Note also that if you rely too much on outside resources, you may not learn the material (the main goal!) and would likely do poorly on exams (where such resources are not available).

Generative AI Policy. You may ask general questions about concepts related to the homework problems. However, you may not directly ask them for hints or answers to the homework questions. For example, you should not be copying and pasting directly from the assignment handout to a chat interface. In addition, you may not directly use the output of a chatbot (or a paraphrased output) in your answers.

1 Written Problems: Batch Size vs. Learning Rate

When training neural networks, it is important to select an appropriate batch size. In this question, we will investigate the effect of batch size on some important quantities in neural network training.

Batch size affects the stochasticity in optimization, and therefore affects the choice of learning rate. We demonstrate this via a simple model called the noisy quadratic model (NQM). Despite the simplicity, the NQM captures many essential features in realistic neural network training.

For simplicity, we only consider the scalar version of the NQM. We have the quadratic loss $\mathcal{L}(w) = \frac{1}{2}aw^2$, where $a > 0$ and $w \in \Re$ is the weight that we would like to optimize. Assume that we only have access to a noisy version of the gradient – each time when we make a query for the gradient, we obtain $g(w)$, which is the true gradient $\nabla\mathcal{L}(w)$ with additive Gaussian noise:

$$\nabla\mathcal{L}(w) + \epsilon, \;\; \epsilon \sim \mathcal{N}(0, \sigma^2)$$

One way to reduce noise in the gradient is to use minibatch training. Let B be the batch size, and denote the minibatch gradient as $g_B(w)$:

$$g_B(w) = \frac{1}{B}\sum_{i=1}^{B} g_i(w), \;\; \text{where } g_i(w) = \nabla\mathcal{L}(w) + \epsilon_i, \;\; \epsilon_i \sim \mathcal{N}(0, \sigma^2)$$

(a) As batch size increases, how do you expect the optimal learning rate to change? Briefly explain in 2-3 sentences.

(Hint: Think about how the minibatch gradient noise changes with B.)

1.1 Training steps vs. batch size

For most of neural network training in real-world applications, we often observe the relationship of training steps and batch size for reaching a certain validation loss as illustrated in Figure 1.

(a) For the three points (A, B, C) in Figure 1, which one has the most efficient batch size (in terms of best resource and training time trade-off)? Assume that you have access to scalable (but not free) compute such that minibatches are parallelized efficiently. Briefly explain in 1-2 sentences.

(b) Figure 1 demonstrates that there are often two regimes in neural network training: the noise-dominated regime and the curvature-dominated regime. In the noise-dominated regime, the bottleneck for optimization is that there exists a large amount of gradient noise. In the curvature-dominated regime, the bottleneck of optimization is the ill-conditioned loss landscape. For points A and B in Figure 1, which regimes do they belong to, and what would you do to accelerate training? Fill each of the blanks with one best suited option.

Point A: Regime: Potential way to accelerate training:

Point B: Regime: Potential way to accelerate training:

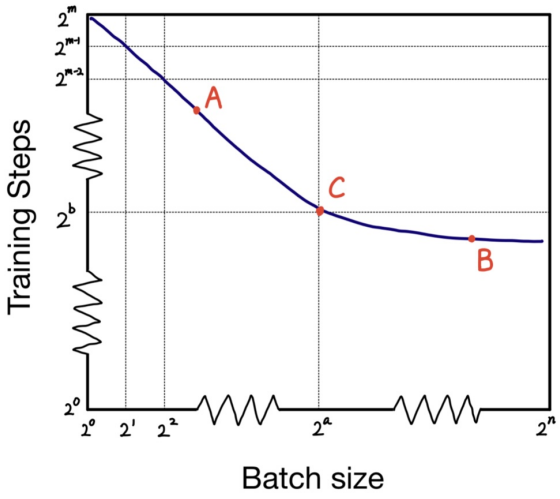

Figure 1: A cartoon illustration of the typical relationship between training steps and the batch size for reaching a certain validation loss (based on Shallue et al. [2018]). Learning rate and other related hyperparameters are tuned for each point on the curve.

Options:

- Regimes: noise dominated / curvature dominated.
- Potential ways to accelerate training: use higher order optimizers / seek parallel compute

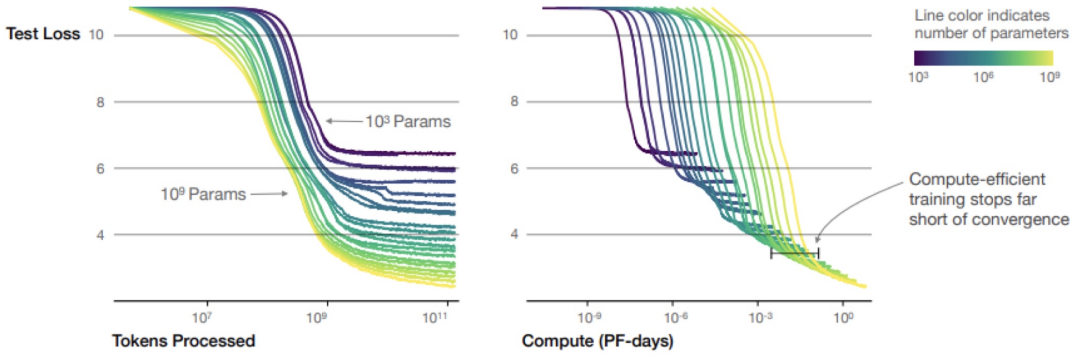

Figure 2: Test loss of language models of different sizes, plotted against the dataset size (tokens processed) and the amount of compute (in petaflop-days).

1.2 Model size, dataset size and compute

We have seen in the previous section that batch size is an important hyperparameter during training. Besides efficiently minimizing the training loss, we are also interested in the test loss. Recently, researchers have observed an intriguing relationship between the test loss and hyperparameters such as the model size, dataset size and the amount of compute used. We explore this relationship for neural language models in this section. The figures in this question (Figure 2 and Figure 3) are from Kaplan et al. [2020].

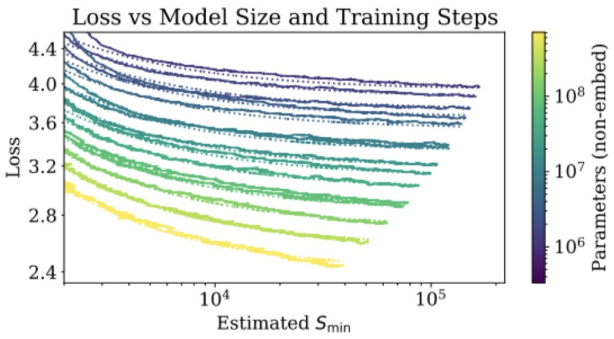

Figure 3: Test loss for different sized models after the initial transient period, plotted against the number of training steps (S_{min}) when using the critical batch sizes (the batch sizes that separate the two regimes as discussed above).

(a) Previously, you have trained a neural language model and obtained somewhat adequate performance. You have now secured more compute resources (in PF-days), and want to improve the model test performance (assume you will train from scratch). Which of the following is the best option? Give a brief explanation (2-3 sentences).

A. Train the same model with the same batch size for more steps.

B. Train the same model with a larger batch size (after tuning the learning rate), for the same number of steps.

C. Increase the model size.

2 Programming Part

2.1 Introduction

In this assignment, you will explore common tasks and model architectures in Natural Language Processing (NLP). Along the way, you will gain experience with important concepts like attention mechanisms, transformer language models, and scaling laws.

Setting Up

We recommend that you use Colab (https://colab.research.google.com/) for the assignment. To setup the Colab environment, just open the notebooks for each part of the assignment and make a copy in your own Google Drive account. The data files for the assignment can be downloaded on the class website ('pig_latin_small.txt' and 'pig_latin_large.txt').

Deliverables

Each section is followed by a checklist of deliverables to add in the assignment writeup. To also give a better sense of our expectations for the answers to the conceptual questions, we've put maximum sentence limits. You will not be graded for any additional sentences.

2.2 Neural machine translation (NMT)

Neural machine translation (NMT) is a subfield of NLP that aims to translate between languages using neural networks. In this section, will we train a NMT model on the toy task of English → Pig Latin. Please read the following background section carefully before attempting the questions.

2.2.1 Background

The task

Pig Latin is a simple transformation of English based on the following rules:

1. If the first letter of a word is a *consonant*, then the letter is moved to the end of the word, and the letters "ay" are added to the end: **team** → **eamtay**.

2. If the first letter is a *vowel*, then the word is left unchanged and the letters "way" are added to the end: **impress** → **impressway**.

3. In addition, some consonant pairs, such as "sh", are treated as a block and are moved to the end of the string together: **shopping** → **oppingshay**.

To translate a sentence from English to Pig-Latin, we apply these rules to each word independently:

i went shopping → iway entway oppingshay

Our goal is to build a NMT model that can learn the rules of Pig-Latin *implicitly* from (English, Pig-Latin) word pairs. Since the translation to Pig Latin involves moving characters around in a string, we will use a *character-level* transformer model. Because English and Pig-Latin are similar in structure, the translation task is almost a copy task; the model must remember each character in the input and recall the characters in a specific order to produce the output. This makes it an ideal task for understanding the capacity of NMT models.

The data

The data for this task consists of pairs of words $\{(s^{(i)}, t^{(i)})\}_{i=1}^{N}$ where the *source* $s^{(i)}$ is an English word, and the *target* $t^{(i)}$ is its translation in Pig-Latin. [1] Some examples are:

{ **(the, ethay), (family, amilyfay), (of, ofway), ...** }

In this assignment, you will investigate the effect of dataset size on generalization ability. We provide a small and large dataset. The small dataset is composed of a subset of the unique words from the book "Sense and Sensibility" by Jane Austen, totaling 3198 words. The vocabulary consists of 29 tokens: the 26 standard alphabet letters (all lowercase), the dash symbol -, and two special tokens <SOS> and <EOS> that denote the start and end of a sequence, respectively. [2] The second, larger dataset is obtained from Peter Norvig's natural language corpus. [3] It contains the top 20,000 most used English words, which is combined with the previous data set to obtain 22,402 unique words. This dataset contains the same vocabulary as the previous dataset.

The model

[1] In order to simplify the processing of mini-batches of words, the word pairs are grouped based on the lengths of the source and target. Thus, in each mini-batch, the source words are all the same length, and the target words are all the same length. This simplifies the code, as we don't have to worry about batches of variable-length sequences.

[2] Note that for the English-to-Pig-Latin task, the input and output sequences share the same vocabulary; this is not always the case for other translation tasks (i.e., between languages that use different alphabets).

[3] https://norvig.com/ngrams/

Translation is a sequence-to-sequence (seq2seq) problem. The goal is to train a model to transform one sequence into another. A transformer model [Vaswani et al., 2017] uses an encoder-decoder architecture and relies entirely on an attention mechanism to draw global dependencies between the input sequence and the output sequence. The encoder processes the input sequence in parallel using stacked self-attention and point-wise fully connected layers, as shown in Figure 5. Given the hidden representations of each input token processed through an encoder, the decoder then generates an output sequence one at a time. The model is auto-regressive when generating the output tokens.

Specifically, input characters are passed through an embedding layer before being fed into an encoder model. If H is the dimension of the encoder hidden state, we learn a $29 \times H$ embedding matrix, where each of the 29 characters in the vocabulary is assigned a H-dimensional embedding. At each time step, the decoder outputs a vector of unnormalized log probabilities given by a linear transformation of the decoder hidden state. When these probabilities are normalized (i.e., by passing them through a softmax), they define a distribution over the vocabulary, indicating the most probable characters for that time step. The model is trained via a cross-entropy loss between the decoder distribution and ground-truth at each time step.

2.2.2 Transformers for NMT (Attention Is All You Need)

In order to answer the following questions correctly, please make sure that you have run the code from **nmt.ipynb, Part1, Training and evaluation code** prior to answering the following questions.

1. In lecture, we learned about Scaled Dot-product Attention used in the transformer models. The function f is a dot product between the linearly transformed query and keys using weight matrices W_q and W_k:

$$\begin{aligned} \tilde{\alpha}_i^{(t)} &= f(Q_t K_i) = \frac{(W_q Q_t)^T (W_k K_i)}{\sqrt{d}} \\ \alpha_i^{(t)} &= \text{softmax}(\tilde{\alpha}^{(t)})_i \\ c_t &= \sum_{i=1}^{T} \alpha_i^{(t)} W_v V_i \end{aligned}$$

where d is the dimension of the query and the W_v denotes weight matrix project the value to produce the final context vectors.

Implement the scaled dot-product attention mechanism. Fill in the forward methods of the **ScaledDotAttention** class. Use the PyTorch torch.bmm (or @) to compute the dot product between the batched queries and the batched keys in the forward pass of the **ScaledDotAttention** class for the unnormalized attention weights.

The following functions are useful in implementing models like this. You might find it useful to get familiar with how they work (you can consult PyTorch documentation):

- squeeze
- unsqueeze
- expand as
- cat

- view
- bmm (or @)

Your forward pass needs to work with both 2D query tensor **(batch_size** $\times$ **(1)** $\times$ **hidden_size)** and 3D query tensor **(batch_size** $\times$ **k** $\times$ **hidden-size)**.

2. Implement the causal scaled dot-product attention mechanism. Fill in the forward method in the **CausalScaledDotAttention** class. It will be mostly the same as the **ScaledDotAttention** class. The additional computation is to mask out the attention to the future time steps. You will need to add **float("-inf")** to some of the entries in the unnormalized attention weights. You may find the **torch.tril** , **torch.triu**, or **masked_fill** methods handy for this part.

3. We will now use **ScaledDotAttention** as the building blocks for a simplified transformer [Vaswani et al., 2017] encoder.

The encoder consists of three components:

1. Positional encoding: To encode the position of each word, we add to its embedding a constant vector that depends on its position:

   p^{th} word embedding = input embedding + positional encoding(p)

   We follow the same positional encoding methodology described in Vaswani et al. [2017]. That is we use sine and cosine functions:

$$\begin{aligned} \text{PE}(\text{pos},2i) &= \sin\frac{\text{pos}}{10000^{2i/d_{model}}} \\ \text{PE}(\text{pos},2i+1) &= \cos\frac{\text{pos}}{10000^{2i/d_{model}}} \end{aligned}$$

   Since we always use the same positional encodings throughout the training, we pre-generate all those we'll need while constructing this class (before training) and keep reusing them throughout the training.

2. A **ScaledDotAttention** operation.
3. An MLP.

For this question, describe why we need to represent the position of each word through this positional encoding in one or two sentences. Additionally, describe the advantages of using this positional encoding method, as opposed to other positional encoding methods such as a one-hot encoding in one or two sentences.

4. In the code notebook, we have provided an experimental setup to evaluate the performance of the Transformer as a function of hidden size and data set size. Run the Transformer model using hidden size 16 versus 32, and using the small versus large dataset (in total, 4 runs). We suggest using the provided hyper-parameters for this experiment.

Run these experiments, and report the effects of increasing model capacity via the hidden size, and the effects of increasing dataset size. In particular, report your observations on how loss as a function of gradient descent iterations is affected, and how changing model/dataset size affects the generalization of the model. Are these results what you would expect?

In your report, include the two loss curves output by **save_loss_comparison_by_hidden** and **save_loss_comparison_by_dataset**, the lowest attained validation loss for each run, and your response to the above questions.

Deliverables

Create a section in your report called Scaled Dot Product Attention. Add the following:

- Screenshots of your **ScaledDotProduct, CausalScaledDotProduct** implementations. Highlight the lines you've added.
- Your written answer to question 3.
- The two loss curves plots output by the experimental setup in question 4, and the lowest validation loss for each run.
- Your response to the written component of question 4. Your analysis should not exceed six sentences.

2.3 Decoder Only NMT

In this subsection, we will train a decoder-only NMT model using the **CausalAttention** mechanism. The key difference between this approach and the previous encoder-decoder approach is that we do not encode a hidden state of the input sequence first using an encoder. Instead, we feed both the input sequence and the target sequence to a decoder simultaneously, as in Figure 6. The input sequence and the target sequence will be separated using an end-of-prompt token (**EOP**). The concatenated input to the decoder will have **SOS** token added at the beginning, and the concatenated target will have **EOS** token added at the end. In our provided notebook, the decoder will process this concatenated input using *causal attention*, but we compute the cross-entropy loss by using the output tokens beginning from the output of **<EOP>** only.

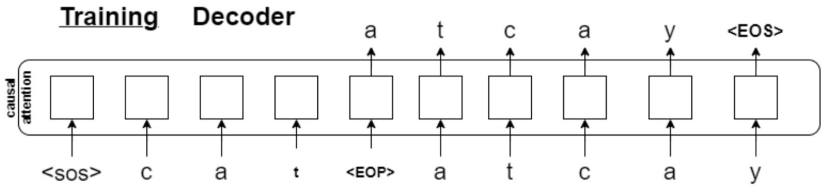

Figure 4: Training the decoder-only NMT model.

For test-time translations, we first feed the input sequence to a trained decoder, enclosed by a **SOS** token and a **EOP** token, as shown in Figure 4. We obtain the first translated token *a* in this case and concatenate the input sequence with the generated token. Then we feed the concatenated sequence to the decoder and obtain two tokens *a* and *t*. This procedure is repeated until reaching the maximum target length or generating a **<EOS>** token.

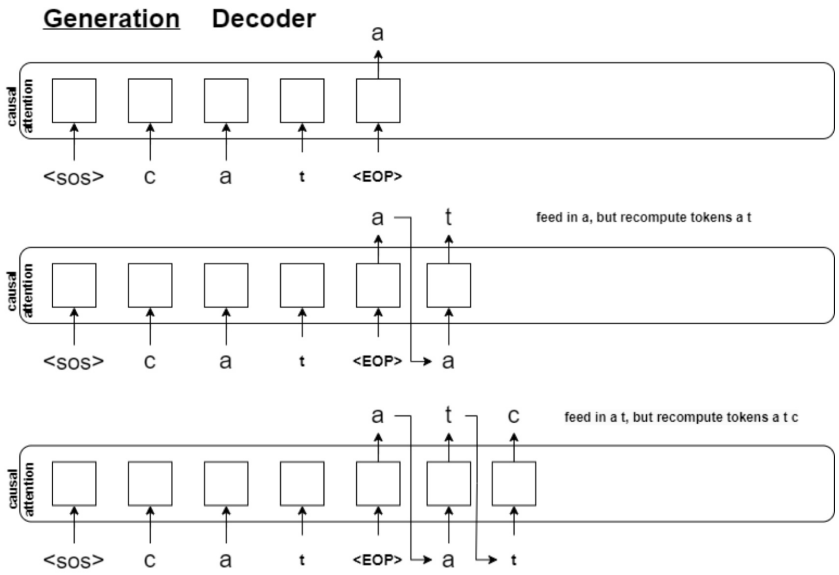

Figure 5: Translating a text using the decoder-only NMT model.

We will also experiment with one final form of attention: ***multi-head attention***. You will remember from lecture that multi-head attention has the following form:

$$MultiHead(Q, K, V) = Concat(head_1, ..., head_h)W^O,$$
$$head_i = Attention(QW_i^Q, QW_i^Q, QW_i^Q).$$

In order to answer the following questions correctly, please make sure that you have run the code from **nmt.ipynb, Part2, Training and evaluation code** prior to answering the following questions.

1. *Implement the forward function in* **DecoderOnlyTransformer**.

2. *Construct the input tensors and the target tensors for training a decoder.* For this question, we ask you to implement the missing line in the function decoder_generate that takes in an input sequence and returns a concatenated sequence of the form

<SOS> input sequence **<EOP>**

as in the input to the decoder shown in Figure 5.

3. *Train the model*. Now, run the training and testing code block to see the generated translation using a decoder-only model. Comment on the pros and cons of the decoder-only approach. How is the quality of your generated results compared to the ones using the encoder-decoder model?

4. *Implement multi-head attention in* **MultiHeadCausalScaledDotAttention**. This will involve both initializing W^0 appropriately and implementing the forward method.

5. *Compare increasing hidden dimension and increasing number of attention heads.* You'll next train 5 different models with varying hidden dimension and number of attention heads. To analyze their parameter efficiency, you'll need to implement **calc_parameters_decoder**. What tends to improve performance more:

- Increasing the number of attention heads for a fixed hidden dimension size?
- Increasing the hidden dimensionality for a single attention head?

Which of these techniques (increasing number of attention heads vs. increasing hidden dimensionality) seems to be more parameter efficient?

Deliverables

Create a section in your report called **Decoder Only NMT**. Add the following:

- Your answer to Question 1. (Screenshots of your implementations)
- Your answer to Question 2. (Screenshots of your implementations)
- Your written response to Question 3.
- Your answer to Question 4. (Screenshots of your implementations)
- Your written response to Question 5, as well as screenshots the loss curves and your implementation of **calc_parameters_decoder**.

2.4 Scaling Law and IsoFLOP Profiles

While in the previous section we analyzed parameter efficiency by comparing differences in performance for increasing number of attention heads and number of hidden dimensions, this section will analyze how to best use a fixed compute budget and other adjacent questions.

This section will give you hands-on experience charting scaling law curves to forecast neural network performance. A scaling law is a fundamental concept that describes how the performance of a neural network changes with its size. Specifically, it relates the number of parameters or computations required by a neural network to achieve a certain level of performance, such as accuracy or loss. The scaling law provides a useful tool for predicting the performance of neural networks as they are scaled up or down.

IsoFLOP is a method proposed in the "Training Compute-Optimal Large Language Models" paper [Hoffmann et al., 2022] to study the scaling law of large language models. The authors of the paper used IsoFLOP to study the effect of model size on the performance of large language models and to determine the optimal model size that maximizes performance for a given computational budget.

The motivation for using IsoFLOP to forecast neural network performance is twofold. Firstly, it provides a more accurate and efficient way to explore the scaling law of large language models than traditional methods, which involve training multiple models at different sizes. Secondly, IsoFLOP allows for a better understanding of the trade-off between model size and training cost, which is crucial for designing large-scale neural network architectures that are both efficient and effective. By leveraging IsoFLOP, researchers can gain insights into the scaling properties of neural networks, such as their accuracy and computational efficiency, and optimize their performance for specific applications and computational resources.

In this question, we will plot the scaling law curve for the decoder-only translation models from the previous section. The notebook provided trains seven translation models with different model sizes and varies the FLOP counts by training for different numbers of epochs. You are asked to complete the functions to make the final IsoFLOP curve consisting of models ranging from 0.08 TFLOPs to 1.28 TFLOPs.

1. Train seven decoder-only translation models using the code provided and plot the validation loss as the function of FLOPs. Comment on any interesting thing you observe. Does a larger model always have a smaller validation loss? (Hint: See Section 1.2).

2. Create the Compute Optimal Model plot by fitting a line to the target FLOPs and the optimal model parameters using the code provided. Based on the plot, estimate the optimal number of parameters when we have a compute budget of 1e15.

3. Plot Compute Optimal Token using the code provided. Now, given the Compute Optimal Model plot and Compute Optimal Token plot, is the training setup in Section 2.3.3 compute optimal? If not, how should we change it?

Deliverables

Create a section in your report called **Scaling Law and IsoFLOP Profiles**. Add the following:

- Your written response to Question 1. Your answer should not exceed 3 sentences.
- Your answer to Question 2. (The optimal number of parameters given 1e15 FLOPs and the process of how you estimate it.)
- Your written response to Question 3. Your answer should not exceed 3 sentences.

References

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, and Aidan Clark. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022. Available at https://arxiv.org/abs/2203.15556.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, , and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. Available at https://arxiv.org/abs/2001.08361.

Christopher J Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, , and George E Dahl. Measuring the effects of data parallelism on neural network training. *arXiv preprint arXiv:1811.03600*, 2018. Available at https://arxiv.org/abs/1811.03600.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, , and Illia Polosukhin. Attention is all you need. pages 5998–6008, 2017.