

GENERATIVE SEQUENCE MODELS

NEXT TOKEN PREDICTION IS ALL YOU NEED

AND

INTRODUCING GPT- α



University of
St Andrews

FRASER ROBERT LOVE

School of Computer Science
University of St Andrews

A dissertation submitted for the degree of
MSc Artificial Intelligence

September 19, 2024

Abstract

Generative sequence modelling is fundamental to deep learning and focuses on predicting the next token in a sequence. This involves compressing inherent information and understanding the underlying processes causing the probabilistic generation processes. The advancement of generative models, in particular the Transformer architecture, lies in their ability to scale with data and compute, extracting ever more complex structure from within sequences. Furthermore, generative models exhibit an extraordinary degree of understanding of reality through projections of the world onto sequence space. Different information mediums can be tokenised and represented as sequences of continuous vector embeddings. These sequences can then be utilised by sequence models to generate intricate output sequences, including textual, spatial or audio information.

This dissertation explores these dynamics, understanding various architectures of generative sequence models and the mathematics behind them. Several architectures shall be explored, including the neural probabilistic sequence models, recurrent neural networks (RNNs), long-short term memory networks, RNN encoder-decoder networks, attention mechanisms, and the Transformer architecture. Furthermore, this dissertation introduces GPT- α , a 124 million parameter, decoder-only transformer based language model which follows the architecture of GPT-2 and the training process of GPT-3. GPT- α was trained on 40 billion tokens and achieves state-of-the-art performance for a model of this scale when evaluated on a series of universal language modelling benchmarks.

Declaration

I hereby certify that this dissertation, which is approximately 14,864 words in length, has been composed by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a degree. This project was conducted by me at the University of St Andrews from May 2024 to August 2024 towards fulfilment of the requirements of the University of St Andrews for the degree of MSc Artificial Intelligence under the supervision of Dr Lei Fang.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright to this work.



Acknowledgements

First and foremost, I would like to thank my supervisor, Lei Fang, for his continuous support and guidance throughout the course of this research. His experience and encouragement have been instrumental towards the completion of this dissertation.

Furthermore, I want to express my deepest gratitude to my parents for their support and encouragement throughout the years. They have provided a safe and stable environment for me to progress and fulfill my potential. I would also like to thank my sister for her support, and especially her humour.

Additionally, I am grateful to the friends I have made while at the University of St Andrews. Their camaraderie and encouragement has made university life so enjoyable and rewarding.

Contents

Declaration	ii
Acknowledgements	ii
Contents	iii
List of Figures	v
List of Tables	xi
1 Deep Learning Background	1
1.1 Learning	1
1.2 Optimisation	3
1.3 Backpropagation	7
1.4 Feedforward Neural Networks	10
2 The Rise of the Transformer	14
2.1 Vector Embeddings	14
2.2 Sequence Modelling	15
2.3 Generation	17
2.4 Neural Sequence Model	18
2.5 Recurrent Neural Network	21
2.6 Long Short-Term Memory	25
2.7 RNN Encoder-Decoder	28
2.8 Attention-based RNN Encoder-Decoder	31
3 Transformer Architecture	34
3.1 Encoder-Decoder Architecture	34
3.2 Positional Encodings	37
3.3 Scaled Dot-Product Attention and Multi-Head Attention	39
3.4 Tokenisation and Byte-Pair Encoding	44

3.5	Encoder-Only and Decoder-Only Models	46
4	Introducing GPT-α	48
4.1	Model Architecture	48
4.2	Training	49
4.3	Evaluation	54
5	Conclusions and Future Work	56
	Bibliography	58
A	GPT-α Generation Samples	63
B	Benchmark Exploration	66
C	GPT-α Components	74
D	Scaling Compute-Optimal Transformer Models	76

List of Figures

1.1	Dropout applied to a feedforward neural network consisting of two hidden layers. Connections involving nodes subjected to dropout have been eliminated. Dropout is not applied to the final layer of the network to maintain the accuracy of the final predictions.	3
1.2	Computational graph of the univariate function $g(\theta) = e^{-\sin(\theta)^2+5}$ with nodes representing intermediate variables and each arrow representing a differentiable intermediate function applied to the previous variable. The forward arrows shown the forward pass from input θ and reverse arrows showing the local derivatives calculated in the backward pass through backpropagation. The local gradient of the output node is always initialised to one as $\bar{g} = \frac{\partial g}{\partial g} = 1$. The calculations below show how the gradients are backpropagated through the network, using the chain rule to multiply each local derivative with the adjoint $\bar{t}_i = \frac{\partial g}{\partial t_i}$	9
1.3	Backpropagation through intermediate nodes. The adjoint $\bar{\mathbf{t}}$ is calculated as the sum of the partial derivatives of subsequent variables \mathbf{t}_1 and \mathbf{t}_2 with respect to \mathbf{t} , multiplied by their respective adjoints $\bar{\mathbf{t}}_1$ and $\bar{\mathbf{t}}_2$	9
1.4	Gradients are equally distributed between the two operands in an addition operation. That is, for $\mathbf{t} = \mathbf{t}_1 + \mathbf{t}_2$ the gradients are distributed so $\bar{\mathbf{t}}_1 = \bar{\mathbf{t}}$ and $\bar{\mathbf{t}}_2 = \bar{\mathbf{t}}$. For matrix multiplication, gradients are exchanged. That is, for $\mathbf{t} = \mathbf{t}_1 \mathbf{t}_2$ the gradients are exchanged so $\bar{\mathbf{t}}_1 = \bar{\mathbf{t}} \mathbf{t}_2^\top$ and $\bar{\mathbf{t}}_2 = \mathbf{t}_1^\top \bar{\mathbf{t}}$	9
1.5	Let f be an element-wise function such that $\mathbf{t} = f(\mathbf{t}_1)$. For an element-wise function gradients are multiplied by the derivative $f' = \frac{\partial \mathbf{t}}{\partial \mathbf{t}_1}$. Let r be a reshaping operation. The gradient is the inverse r^{-1} reshaping operation on the previous adjoint.	10
1.6	Left: A single neuron. Inputs x_i interact multiplicatively with the weights w_i and are summed into the pre-activation z which is then sent through the activation function to produce an output \hat{y} . The bias is represented as $x_0 = b$. Right: Feedforward neural network architecture with two hidden layers. Each neuron is connected to all neurons in the previous layer, but are not connected to neurons within the same layer. Therefore neurons within each layer can be computed in parallel via matrix multiplication.	12

2.1	Neural sequence model comprised of a two layer feedforward network. Dashed arrows represent optional residual connections with weights \mathbf{U} . The output logits are transformed into estimated probabilities $\hat{\mathbf{y}}$ via softmax.	19
2.2	Unfolded representation of the recurrent neural network (RNN) architecture. Each hidden state \mathbf{h}_t is computed from the previous hidden state \mathbf{h}_{t-1} and the current input \mathbf{x}_t . The initial hidden state \mathbf{h}_0 is either initialised to zero or learned as a parameter.	22
2.3	Backpropagation through time (BTT). Unrolled recurrent neural network (RNN) computation graph illustrating the forward (solid arrows) and backward (dashed arrows) pass from the loss L towards the inputs \mathbf{x}_t . Biases have been omitted for brevity.	23
2.4	Long short-term memory (LSTM) cell architecture. Each arrow represents the flow of vector operations. Dark grey nodes represent a pointwise operation such as vector addition and white nodes represent a learned neural network layer. Merging lines represent vectors being concatenated and a diverging lines represent the vectors being duplicated.	25
2.5	Uninterrupted gradient flow between the current cell state \mathbf{c}_t and the previous cell state \mathbf{c}_{t-1} during backpropagation. Gradient flow is shown by the dashed arrows.	27
2.6	Gated Recurrent Unit (GRU) cell architecture. Each arrow represents the flow of vector operations. Dark grey nodes represent a pointwise operation such as vector addition and white nodes represent a learned neural network layer. Merging lines represent vectors being concatenated and a diverging lines represent the vectors being duplicated.	28
2.7	RNN encoder-decoder architecture from Sutskever et al. [51]. The encoder (left) computes the context vector $\mathbf{c} = \mathbf{h}_T$ of $(\mathbf{x}_1, \dots, \mathbf{x}_T)$ and the decoder (right) uses this context vector representation to compute the probability of $\mathbf{y}, \dots, \mathbf{y}'_T$ conditioned on the context vector, where \mathbf{x}_T and \mathbf{y}'_T are the embedding of the end of sequence token $\langle s \rangle$	30
2.8	RNN encoder-decoder architecture from Cho et al. [10] similar to Sutskever et al. [51] except that each individual hidden state \mathbf{h}'_t and next token \mathbf{y}_t are conditioned on the context vector.	30
2.9	Bidirectional RNN encoder-decoder architecture with an attention mechanism. The bidirectional RNN encoder processes the sequence in two directions. The calculated annotations α_{tj} from each hidden state in the bidirectional RNN \mathbf{h}_j are sent into a weighted sum to obtain the context vector \mathbf{c}_t for the current hidden state in the decoder \mathbf{h}'_t . Therefore $\hat{\mathbf{y}}_t$ is generated by attending to each input $\{\mathbf{x}_1, \dots, \mathbf{x}_T\}$	33

3.1	Transformer architecture. The encoder (left) and decoder (right) both consist of N identical layers. The encoder layers include multi-head attention and feed-forward sub-layers, while the decoder layers add a masked multi-head attention sub-layer. . .	36
3.2	Positional encoding matrix $\mathbf{P} \in \mathbb{R}^{T \times C}$ produced with the sine and cosine functions of varying frequencies introduced by Vaswani et al. [53] for sequence lengths of $T = 1024$ and an embedding dimensionality of $C = 768$. The scaling parameter was set to $n = 1e2$ to scale the functions over the entire embedding dimension.	38
3.3	Positional embedding matrix $\mathbf{P} \in \mathbb{R}^{T \times C}$ from GPT-2 124M which features sequence lengths of $T = 1024$ and an embedding dimensionality of $C = 768$. Values have been clipped to a range of $[-1, 1]$ so that structure with the embedding matrix is more easily shown. Dimensions in the embedding space show structure used which differentiate positions within the input sequence.	38
3.4	Left: Scaled dot-product attention. The optional mask layer is only used in the decoder so that tokens only attend to previous tokens. Right: Multi-head attention with H layers computed in parallel.	41
3.5	Masked self-attention matrix from the third head in the fourth layer of GPT-2 124M for the sequence of tokens 'generative models exhibit a deep sense of reality through projections of the world onto sequence space'. The autoregressive property of causal self-attention implemented through masking creates a lower triangular matrix. Each cell represents the attention score between tokens with brighter cells indicating higher attention scores, where the query token is attending more to the corresponding key token. Moving down the rows, each subsequent token will attend to previous tokens up to itself. Softmax ensures that each row (the attention values) sum to one.	42
3.6	Visualisation of the weights of each query (\mathbf{Q}) - key (\mathbf{K}) pair for masked self-attention from the first five heads and the first five layers from GPT-2 124M for the sequence of tokens 'generative models exhibit a deep sense of reality through projections of the world onto sequence space'. Each subplot represents the attention patterns of one attention head in a particular layer. Lines connect tokens on the left (queries) to tokens on the right (keys), with line thickness indicating the attention strength. Throughout all the heads in each layer there is a clear pattern of attending heavily towards the first position. Note that autoregressive masking is clearly visible as no queries attend to keys that come after them (i.e. no lines exist with a negative gradient from left to right).	43

- 3.7 Output of the GPT-2 tokeniser for the string 'GPT-2 has a vocabulary size of 50,257 tokens (50,000 BPE + 256 Byte tokens + <|endoftext|> token)' showing sub-word tokenisation. Frequently occurring words have been tokenised as individual tokens. Uncommon words such as GPT-2 have been tokenised with base tokens G, P, T, 2 and the token PT. The <|endoftext|> token is the special token used to represent the end of a sequence. 44
- 3.8 GPT-2 pre-tokeniser regular expression splitting pattern. (?:[sdmt]|ll|ve|re) matches common contractions, however it only considers the ASCII apostrophe and not the Unicode apostrophe. Additionally, it is case-sensitive. As a result, I'm will be tokenised as I'm and I'M will be tokenised as I'M. These are some limitations of the GPT-2 tokeniser. 45
- 3.9 BPE algorithm example. Pre-tokenisation splits the text into words and the frequency of each word is recorded. Each word is split into tokens in the base vocabulary: d, o, g, l, p, u, n, b, s. BPE then counts the frequency of each possible pair of base tokens and merges the token pair that occurs most frequently. The first merge is og with a frequency of 5. The second merge is un with a frequency of 4. The final merge is dog with a frequency of 3. As a result dog dog can be expressed as one token and the more uncommon dogs can be expressed using this dog token and the base token s. 46
- 3.10 **Left:** Encoder-only Transformer architecture. The decoder has been removed. The output of the encoder-only model is \mathbf{z} , the contextual representation of the input sequence. **Right:** Decoder-only Transformer architecture. The encoder and the cross-attention sub-layer in the decoder have been removed. The model calculates the conditional probability $p(\mathbf{y}_t|\{\mathbf{y}_{t-T}, \dots, \mathbf{y}_{t-1}\})$ where T is the context length. . . 47
- 4.1 Cosine learning rate decay with a linear warmup of 375 million tokens. After the linear warmup the learning rate is at a maximum of 18×10^{-4} and decays to a minimum of 6×10^{-5} after 40B tokens. 50
- 4.2 Cross-entropy loss trajectory for the GPT 124M model shows a smooth and consistently decreasing slope that rapidly surpasses the OpenAI GPT-2 124M baseline loss, indicating consistent convergence and improved performance over time. Data points were taken after every 150 iterations. Note that this models loss is not directly comparable to the GPT-2 baseline loss due to the models being trained, and hence loss calculated, on different datasets. The GPT-2 baseline serves as guidance only. . . . 52

4.3	HellaSwag accuracy [58] throughout training. Data points were taken after every 150 iterations. The model surpasses GPT-2 124M on HellaSwag after just 5B tokens and surpasses GPT-3 125M after 38B tokens. This is a 20x improvement over GPT-2 124M and 7.8x improvement over GPT-3, which were trained on 100B tokens and 300B tokens respectively.	53
4.4	Example question from HellaSwag benchmark where the model has predicted (P) the correct completion (A) based on the given context. The model selects one of the possible completions by computing the average loss for the completion for each possible ending and selecting the one with the minimum loss. Models with greater understanding will produce better predictions that more closely match the actual completion, minimising the loss for this prediction, and so will choose the correct completion more often.	54
A.1	GPT- α model completions with the context 'Once upon a time,' using top- k sampling with $k = 50$, and a maximum length of 100 tokens.	63
A.2	GPT- α model completions with the context '2 + 2 =' using top- k sampling with $k = 50$, and a maximum length of 20 tokens.	64
A.3	GPT- α model completions using the first nine numbers in the Fibonacci sequence as context: '0, 1, 1, 2, 3, 5, 8, 13, 21' using top- k sampling with $k = 50$, and a maximum length of 30 tokens.	64
A.4	GPT- α model completions with the context 'How to make pizza:' using top- k sampling with $k = 50$, and a maximum generation length of 100 tokens.	65
A.5	GPT- α model completions with the context 'def cross_entropy_loss():' using top- k sampling with $k = 30$, and a maximum generation length of 100 tokens. . .	65
B.1	Example questions from the PIQA benchmark where GPT- α has predicted (P) the correct completion (A) for the first example, based on the average loss of each sequence. The second example shows an incorrect selection by the model.	66
B.2	Example questions from the SIQA benchmark where GPT- α has predicted (P) the correct completion (A) for the first example, based on the average loss of each sequence. The second example shows an incorrect selection by the model.	67
B.3	Example questions from the OpenBookQA benchmark where GPT- α has predicted (P) the correct completion (A) for the second example, based on the average loss of each sequence. The first example shows an incorrect selection by the model.	68
B.4	Example questions from the TruthfulQA benchmark where GPT- α has predicted (P) the correct completion (A) for the first example, based on the average loss of each sequence. The second example shows an incorrect selection by the model.	69

B.5	Example questions from the MMLU Conceptual Physics benchmark where GPT- α has predicted (P) the correct completion (A) for the first example, based on the average loss of each sequence. The second example shows an incorrect selection by the model.	70
B.6	Example questions from the WinoGrande benchmark where GPT- α has predicted (P) the correct completion (A) for the first and second examples, based on the average loss of each sequence.	70
B.7	Example questions from the ARC Challenge benchmark where GPT- α has predicted (P) the correct completion (A) for the second example, based on the average loss of each sequence. The first example shows an incorrect selection by the model.	71
B.8	Example questions from the TriviaQA benchmark where GPT- α has generated a completion which is expected to match one of the accepted completions for the question. The first example shows an incorrect completion which does not exactly match, however the second example shows a completion with an exact match.	72
B.9	Example questions from the TriviaQA benchmark where GPT- α has generated a completion of which, the last number in the completion is expected to match that of the answer in the accepted completion. The first example shows a completely wrong answer. The second example shows a completion which has generated a wrong completion which consistently repeats itself. However, the last number in the output matches with the answer in the accepted completion, so this is considered correct. . .	73
D.1	Contour plot of loss as a function of model size and FLOPs budget. The optimal model for a compute budget of 4.19×10^{19} total FLOPs is shown and has a size of 422 million parameters.	77
D.2	Contour plot of loss as a function of model size and dataset size. The optimum model of 422 million parameters corresponds to a dataset size of 16.6 billion tokens.	78

D.3	Contour plot of compute (total FLOPs) as a function of model size and dataset size. The optimum model of 422 million parameters corresponds to a dataset size of 16.6 billion tokens.	78
-----	---	----

List of Tables

4.1	Performance of GPT- α compared to similar sized models across key benchmark evaluations. The best performing model across each benchmark is shown in bold. GPT- α outperforms the other models across seven out of the ten benchmarks and achieves the highest average score.	55
C.1	Parameter distribution in GPT- α . The dense layer is zero due to it sharing parameters with the token embedding.	74
C.2	FLOP distribution in GPT- α . Only weight FLOPs are considered as other operations such as layer normalisation or softmax are negligible. The backwards pass is estimated to have twice the computation of the forward pass. This result is close to the estimation using the formula given by Chowdhery et al. [11] which uses the calculation $6 \cdot \theta + 12 \cdot N \cdot H \cdot (C/H) \cdot T$, where $ \theta $ is the total number of parameters, and gives an approximation of just over 875 million FLOPs.	75

Chapter 1

Deep Learning Background

This chapter serves to provide the necessary deep learning background into generative sequence modelling. Note that the reader is expected to have a reasonable understanding of probability theory, linear mathematics and multivariate calculus. The notation standard in this text is the following. Scalars will be displayed as lower case letters such as x, y, γ . Vectors will be displayed as bold-faced lower case letters such as $\mathbf{x}, \mathbf{y}, \boldsymbol{\gamma}$. Matrices will be displayed as bold-faced upper case such as $\mathbf{X}, \mathbf{Y}, \boldsymbol{\Gamma}$.

1.1 Learning

Within mathematics, many problems can be expressed as a mapping $f : \mathcal{X} \rightarrow \mathcal{Y}$ between an input space \mathcal{X} and an output space \mathcal{Y} . In many cases it is intractable to explicitly specify this function f due to its inherent complexity. Let $\mathcal{D} = \{\mathbf{x}_t, \mathbf{y}_t\}_{t=1}^T$ be a set of independent and identically distributed samples from a data generating distribution, formally $(\mathbf{x}_t, \mathbf{y}_t) \sim P_{\mathcal{X} \times \mathcal{Y}}, \forall t$. The concept behind *supervised learning* is to exploit existing examples of the desired mapping $\mathcal{D} = \{\mathbf{x}_t, \mathbf{y}_t\}_{t=1}^T$ to *learn* the mapping function f by searching over a particular space of possible functions \mathcal{F} , to find a function $f^* \in \mathcal{F}$ that is most consistent with the existing examples, such that f^* approximates the true function. This involves minimising a scalar-valued *loss function* $L(\mathbf{y}, \hat{\mathbf{y}})$ that measures the error or *loss* between the predicted value $\hat{\mathbf{y}}_t = f(\mathbf{x}_t)$ and the actual value \mathbf{y}_t (i.e. the performance of f based on the accuracy of its prediction) to find:

$$f^* \leftarrow \arg \min_{f \in \mathcal{F}} \mathbb{E}_{(\mathbf{x}, \mathbf{y}) \sim P} L(\mathbf{y}, f(\mathbf{x}))$$

where $\mathbb{E}_{(x,y) \sim P}$ is the expectation over the joint distribution $P_{\mathcal{X} \times \mathcal{Y}}$ of $(\mathbf{x}_t, \mathbf{y}_t)$ pairs. Once this function f^* has been learned it can be used *infer* or map previously unseen elements of \mathcal{X} to \mathcal{Y} . The optimisation problem is however intractable, as it is impossible to obtain all possible elements of P and therefore cannot fully evaluate the expectation. However, with the assumption that the samples were generated to be independent and identically distributed, the expected loss

can be approximated by averaging the loss over all the samples:

$$f^* \approx \arg \min_{f \in \mathcal{F}} \frac{1}{T} \sum_{t=1}^T L(\mathbf{y}_t, f(\mathbf{x}_t))$$

Parameters are used to map out a particular space of possible functions $f(\mathbf{x}; \boldsymbol{\theta}) \in \mathcal{F}$. The goal is to find the parameters $\boldsymbol{\theta}^*$ that minimise the average loss over the training samples:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t=1}^T L(\mathbf{y}_t, f(\mathbf{x}_t; \boldsymbol{\theta}))$$

Here, $\boldsymbol{\theta}^*$ represents the optimal parameters that define the function f^* which best approximates the true function. Therefore, the problem of learning the best function $f^* \in \mathcal{F}$ has been reduced into an optimisation problem over the parameters $\boldsymbol{\theta}$.

There are however, a few issues with optimising over the approximated loss rather than the actual loss. Consider the function f which maps each sample \mathbf{x}_t to \mathbf{y}_t , but maps to a value of zero elsewhere. This would be a solution to the formula above for approximating f^* when using a sufficient loss function L which achieves a minimum value for $\mathbf{y}_t = \hat{\mathbf{y}}_t$. However, the loss for new samples $(\mathbf{x}, \mathbf{y}) \sim P$ where $(\mathbf{x}, \mathbf{y}) \notin \mathcal{D}$ would be high. That is, the function f would not *generalise* to all possible $(\mathbf{x}, \mathbf{y}) \sim P$. Furthermore, there is the issue of having *no unique solution* f , and the resulting problem of choosing from an entire set of functions $\{f_1, \dots, f_k\} \subseteq \mathcal{F}$ which all achieve the same average loss over all the training samples. *Regularisation*, aims to solve these issues by introducing a regularisation term R which acts as a penalty to the optimisation objective:

$$f^* \approx \arg \min_{f \in \mathcal{F}} \frac{1}{T} \sum_{t=1}^T L(\mathbf{y}_t, f(\mathbf{x}_t)) + R(f)$$

where the above equation is termed the regularised objective function. R is a scalar-valued function that measures the complexity of the function f and therefore its capacity to explicitly map to the training samples and, consequently, its reduced ability to generalise to new samples. This measure of the complexity allows the regularisation term to encode a preference for functions f which are inherently simplistic, following the principle of Occam's razor. Regularisation aims to compensate for the difference between the expected loss over the distribution P and the approximated loss over the set \mathcal{D} of samples.

L2 regularisation is a common type of regularisation which adds a penalty proportional to the sum of the squares of the model parameters $\boldsymbol{\theta}$. This approach shrinks the parameters towards zero but does not set them exactly to zero, hence no feature is completely removed. The L2 regularisation term is $R(f) = \lambda \sum_{j=1}^m \theta_j^2 = \lambda \|\boldsymbol{\theta}\|_2^2$ where $\lambda \geq 0$ is the regularisation strength, m is the number of parameters and θ_j is the j -th parameter. L1 regularisation is another common type of regularisation which adds a penalty proportional to the sum of the absolute values of the model parameters $\boldsymbol{\theta}$. This approach can lead to sparse models where some

parameters are exactly zero, effectively performing feature selection. The L1 regularisation term is $R(f) = \lambda \sum_{j=1}^m |\theta_j| = \lambda \|\boldsymbol{\theta}\|_1$. Dropout [49] is another regularisation technique applied to neural networks, whereby each neuron is assigned a probability of being temporarily removed. This occurs independently for each neuron in each forward pass during training. With dropout, each forward pass effectively trains a different network architecture, enhancing the model's ability to generalise. Mathematically, the forward pass for each layer (covered in later sections) is modified to $\mathbf{h}_i = \mathbf{p}_i \odot \sigma(\mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$ where \mathbf{p}_i is a vector of Bernoulli random variables with probability p of being 1, and \odot denotes element-wise multiplication. The dropout rate, $(1 - p)$, is typically assigned a value between 0.2 and 0.5.

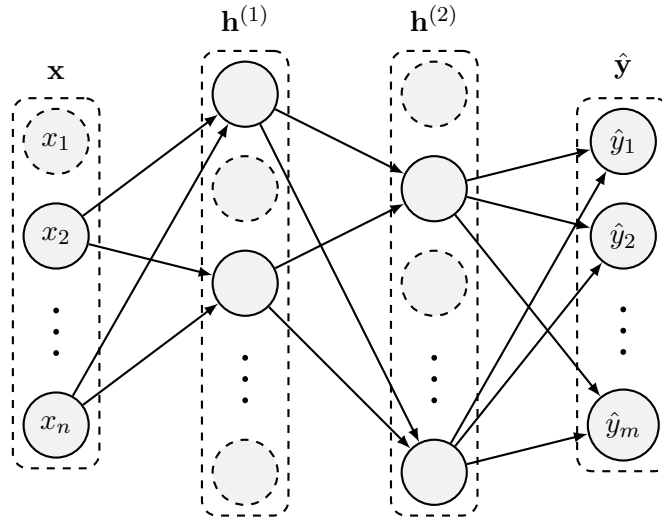


Figure 1.1: Dropout applied to a feedforward neural network consisting of two hidden layers. Connections involving nodes subjected to dropout have been eliminated. Dropout is not applied to the final layer of the network to maintain the accuracy of the final predictions.

1.2 Optimisation

First let $g(\boldsymbol{\theta}) = \frac{1}{T} \sum_{t=1}^T L(\mathbf{y}_t, f(\mathbf{x}_t; \boldsymbol{\theta})) + R(f(\boldsymbol{\theta}))$ be the *regularised objective function* or the *loss surface*. The optimisation problem has therefore reduced to $\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} g(\boldsymbol{\theta})$. An initial, naive method of optimisation could be to use sample a large number of values of $\boldsymbol{\theta}$ randomly from a given distribution. Each value can then be evaluated, with the $\boldsymbol{\theta}$ selected which minimises g . A further approach could involve iteratively refining a candidate $\boldsymbol{\theta}$ through a series of small perturbations. However, these naive, derivative free, optimisation algorithms soon become computationally intractable as the number of parameters scales, such as in modern deep neural network architectures.

The efficiency of optimisation can be improved by restricting our models f to only use differentiable functions. Therefore the *gradient* $\nabla_{\boldsymbol{\theta}} g$ can be computed reverse auto-differentiation,

i.e. *backpropagation* (the details of which will be discussed in the next section). The gradient is a vector of partial derivatives, where every element corresponds to the derivative of g along a dimension of $\boldsymbol{\theta}$. The gradient $\nabla_{\boldsymbol{\theta}}g$ allows for the local first order approximation of the Taylor expansion of g around $\boldsymbol{\theta}_0$:

$$g(\boldsymbol{\theta}) \approx g(\boldsymbol{\theta}_0) + \nabla_{\boldsymbol{\theta}}g(\boldsymbol{\theta}_0)^\top(\boldsymbol{\theta} - \boldsymbol{\theta}_0).$$

The gradient vector $\nabla_{\boldsymbol{\theta}}g$ points in the direction of the greatest ascent of the loss surface. Therefore, the parameters $\boldsymbol{\theta}$ can be improved by applying small changes to $\boldsymbol{\theta}$ in the negative gradient direction, leading to the greatest decrease in g . This is exactly the (*batch*) *gradient descent* algorithm, initially $\boldsymbol{\theta}$ are chosen randomly, then the gradient $\nabla_{\boldsymbol{\theta}}g$ is calculated over the entire batch of training samples, finally, the following is performed iteratively until $\boldsymbol{\theta}$ converges to a local minimum:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \gamma \nabla_{\boldsymbol{\theta}}g(\boldsymbol{\theta}_i)$$

where $\gamma > 0$ is a key parameter called the *learning rate*. If it is too high then this leads to instability and the optimisation may not converge or even diverge. Conversely, if it is set too low then convergence will be slow. An ideal learning rate is typically a value just under the lowest γ which causes the optimisation to diverge. Moreover, *learning rate decay* schedules such as $\gamma_t = \gamma_0 e^{-\beta t}$ for some γ_0 and β , often lead to faster convergence to a lower local minimum. Finally, note that gradient descent cannot backtrack, therefore it will only converge to the local minimum of the basin to which the parameters are initialised within. A learning rate scheduler can also include a *learning rate warmup*, where the learning rate is initially set to zero and gradually increases to a maximum value over a predefined number of iterations. This helps to stabilise training over the initial iterations and avoids initial large parameter updates and allowing the weights to slowly adapt to the loss surface after random initialisation. Furthermore, it allows adaptive optimisers time to accurately estimate the appropriate scaling of the gradients. A typical warmup schedule, such as a linear warmup, can be expressed as: $\gamma_t = \gamma_{\max} \cdot (t+1)/T_w$ where γ_{\max} is the maximum learning rate after the linear warmup, t is the current iteration, and T_w is the number of warmup iterations.

Remember that the loss surface g is the average loss over all the samples. Therefore the gradient $\nabla_{\boldsymbol{\theta}}g$ is the average direction of all the individual gradients:

$$\nabla_{\boldsymbol{\theta}}g(\boldsymbol{\theta}) = \frac{1}{T} \sum_{t=1}^T \nabla L(\mathbf{y}_t, f(\mathbf{x}_t; \boldsymbol{\theta})) + \nabla R(f(\boldsymbol{\theta}))$$

This can be expensive to compute as it requires summing over the loss of all T training samples. Furthermore, this can also be intractable if all the training samples do not fit in memory. The mini-batch *stochastic gradient descent* (SGD) algorithm solves this problem by shuffling and then partitioning the full set of samples into M smaller subsets called mini-batches $\mathcal{D} = \{\mathcal{B}_1, \dots, \mathcal{B}_M\}$, with each mini-batch containing a fixed number of samples, determined by the batch size B where

$T = BM$. The algorithm then processes each mini-batch sequentially. For each mini-batch, the gradients are computed based on the samples in that mini-batch and are used to approximate the gradient over the entire number of samples. Therefore, many approximate updates are performed instead of one exact update.

$$\nabla_{\theta} g(\theta) \approx \frac{1}{B} \sum_{b=1}^B \nabla L(\mathbf{y}_b, f(\mathbf{x}_b; \theta)) + \nabla R(f(\theta))$$

The gradients over mini-batches introduce some noise into the optimisation process, with the possibility of some mini-batch gradient vectors pointing in directions different to that of the greatest ascent direction. These fluctuations enable SGD to jump out of shallow local minima to potentially deeper local minima. However, these fluctuations also mean that SGD struggles to convergence to the exact local minimum unless the learning rate is slowly decreased.

There also exists a series of advanced first order optimisation techniques which often accelerate convergence. The *momentum* update technique [50] accelerates SGD through accumulating gradients along consistent directions of the gradient, leading to faster convergence. This is achieved by adding a fraction α of the exponentially weighted sum, of all the previous gradient vectors \mathbf{v}_i (with \mathbf{v}_0 initialised to zero) to the current gradient vector:

$$\begin{aligned} \mathbf{v}_i &= \alpha \mathbf{v}_{i-1} + \nabla_{\theta} g(\theta_i) \\ \theta_{i+1} &= \theta_i - \gamma \mathbf{v}_i \end{aligned}$$

where the momentum coefficient α is a hyperparameter. The parameters are then updated using the intermediate variable \mathbf{v} . Momentum adjusts the gradient update with a running estimate of the first moment, that is, the exponentially weighted (decaying) mean of the gradients. There are further methods which modulate the gradient update with a running estimate of the second moment of the gradient, that is, the exponentially weighted (decaying) mean of the squared gradients. Let $(\nabla_{\theta} g(\theta_i))^2 = \nabla_{\theta} g(\theta_i) \odot \nabla_{\theta} g(\theta_i)$ where \odot denotes pointwise multiplication. The *Adagrad* (Adaptive Gradient Algorithm) [17] update divides the learning rate by an accumulation of the sum of the squares of all previous gradients:

$$\begin{aligned} \mathbf{v}_i &= \mathbf{v}_{i-1} + (\nabla_{\theta} g(\theta_i))^2 \\ \theta_{i+1} &= \theta_i - \frac{\gamma}{\sqrt{\mathbf{v}_i} + \varepsilon} \odot \nabla_{\theta} g(\theta_i) \end{aligned}$$

During the parameter update, Adagrad scales the learning rate γ by dividing it by the square root of the accumulated gradients, where ε is a small number, roughly $1e^{-5}$, preventing division by zero. Therefore, Adagrad automatically adjusts the learning rate, decreasing the learning rate over successive iterations. However, in practise this learning rate decay often leads to slow convergence during the last iterations.

The *RMSProp* (Root Mean Square Propagation) [52] update method improves upon Adagrad by instead maintaining an exponentially weighted mean of the squared gradients (second moment estimate). The update method remains the exact same:

$$\begin{aligned}\mathbf{v}_i &= \beta \mathbf{v}_{i-1} + (1 - \beta)(\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_i))^2 \\ \boldsymbol{\theta}_{i+1} &= \boldsymbol{\theta}_i - \frac{\gamma}{\sqrt{\mathbf{v}_i} + \varepsilon} \odot \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_i)\end{aligned}$$

The exponentially weighted mean emphasises more recent gradient values rather than the Adagrad approach of equally distributing importance over all gradients, allowing RMSProp to adapt better. Furthermore, the learning rate in RMSProp need not always decay alongside the number of iterations, leading to quicker convergence.

The *Adam* (Adaptive Moment Estimation) update [27] is a prominent optimisation algorithm and can be interpreted as a combination of RMSProp with momentum. That is, both the exponentially decaying mean of previous squared gradients and the exponentially decaying mean of previous gradients are estimated. Alternatively stated, it maintains and updates estimates of for both the first and second running moments. These estimates are used to adaptively adjust the learning rates for each parameter:

$$\begin{aligned}\mathbf{m}_i &= \beta_1 \mathbf{m}_{i-1} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_i) \\ \mathbf{v}_i &= \beta_2 \mathbf{v}_{i-1} + (1 - \beta_2) (\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_i))^2 \\ \hat{\mathbf{m}}_i &= \frac{\mathbf{m}_i}{1 - (\beta_1)^i} \\ \hat{\mathbf{v}}_i &= \frac{\mathbf{v}_i}{1 - (\beta_2)^i} \\ \boldsymbol{\theta}_{i+1} &= \boldsymbol{\theta}_i - \frac{\gamma}{\sqrt{\hat{\mathbf{v}}_i} + \varepsilon} \hat{\mathbf{m}}_i\end{aligned}$$

Note that \mathbf{m}_0 and \mathbf{v}_0 are initialised to zero, which causes \mathbf{m}_i and \mathbf{v}_i to be biased towards zero for the first initial steps due to insufficient previous gradient values to average over. These biases are corrected by computing the bias-corrected first moment $\hat{\mathbf{m}}_i$ and second moment $\hat{\mathbf{v}}_i$ estimates. The update rule is then similar to that of Adagrad and RMSProp. Weight decay is often directly calculated within the update rule by modifying the gradients: $\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_{i+1}) = \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_i) + \lambda \boldsymbol{\theta}_i$ where λ is the weight decay coefficient. This is effectively L2 regularisation. A variation of Adam called *AdamW* (Adam with Weight Decay) decouples weight decay from the gradient update step. This separation ensures that weight decay can be applied directly to the weights, instead of as part of the gradient update, and results in more effective regularisation. The update rule for the parameters for AdamW is:

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \gamma \left(\frac{\hat{\mathbf{m}}_i}{\sqrt{\hat{\mathbf{v}}_i} + \varepsilon} + \lambda \boldsymbol{\theta}_i \right)$$

All these methods of optimisation equalise parameter updates: parameters with large gradients take smaller steps, while parameters with small gradients take larger steps. For an in-depth analysis of various optimisation algorithms, refer to the comprehensive work by Ruder [42].

While higher-order optimisation algorithms exist, they shall be omitted due to them being infeasible to compute for high-dimensional data, and therefore of limited use within deep learning. For example, Newton's method uses a local second-order approximation of the Taylor expansion of g :

$$g(\boldsymbol{\theta}) \approx g(\boldsymbol{\theta}_0) + \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_0)^\top (\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}}^2 g(\boldsymbol{\theta}_0) (\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

where $\nabla_{\boldsymbol{\theta}}^2 g(\boldsymbol{\theta}_0)$ is the *Hessian* matrix and the parameter update rule is accordingly:

$$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} - [\nabla_{\boldsymbol{\theta}}^2 g(\boldsymbol{\theta}_t)]^{-1} \nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}_t)$$

The issue is that the *Hessian* matrix can become intractable to compute and invert for higher dimensional inputs $\mathbf{x} \in \mathbb{R}^n$ as this requires the computation and storage of n^2 second order partial derivatives.

1.3 Backpropagation

As previously discussed, minimising the loss function g via stochastic gradient descent requires the computation of the gradient $\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta})$. *Auto-differentiation* is a method to efficiently compute the gradient $\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta})$ with respect to each parameter in $\boldsymbol{\theta}$ through applying the chain rule recursively to the *computational graph* of g . A computational graph is a directed acyclic graph (DAG) mapping a functions inputs to outputs, where vectors travel along the edges and nodes represent differentiable intermediate functions. Each intermediate function takes as input any number of vectors, and produces a single vector, which then continues through the graph to other nodes.

Consider using auto-differentiation to compute the derivative $\partial g / \partial \theta$ of the univariate function $g(\theta) = e^{-\sin \theta^2 + 5}$. The function g can be deconstructed into a function of θ through the following series of differentiable intermediate functions: $t_1 = \theta^2$, $t_2 = \sin(t_1)$, $t_3 = t_2 \times -1$, $t_4 = t_3 + 5$, $g = e^{t_4}$. Note the local derivatives of each intermediate function with respect to its input are easily evaluated: $\frac{\partial g}{\partial t_4} = e^{t_4}$, $\frac{\partial t_4}{\partial t_3} = t_3$, $\frac{\partial t_3}{\partial t_2} = -t_2$, $\frac{\partial t_2}{\partial t_1} = \cos(t_1)$, $\frac{\partial t_1}{\partial \theta} = 2\theta$. The desired final derivative $\partial g / \partial \theta$ can then be constructed via repeated application of the chain rule: $\frac{\partial g}{\partial \theta} = \frac{\partial g}{\partial t_4} \frac{\partial t_4}{\partial t_3} \frac{\partial t_3}{\partial t_2} \frac{\partial t_2}{\partial t_1} \frac{\partial t_1}{\partial \theta}$.

Computing the gradient of the final output $\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta})$ with respect to the input $\boldsymbol{\theta}$ involves multiplying the individual local gradients of all intermediate functions together via the chain rule. Formally, suppose $\boldsymbol{\theta} = \mathbf{t}_0$ is an input vector that is transformed through a series of differentiable functions, that is $g = f_k \circ \dots \circ f_1$ with $\mathbf{t}_i = f_i(\mathbf{t}_{i-1})$ where $i = 1, \dots, k$ and the last $\mathbf{t}_k = g$ is a scalar. As each f_i is differentiable, $\frac{\partial \mathbf{t}_i}{\partial \mathbf{t}_{i-1}}$ can be calculated and is a Jacobian matrix, which specifies how every output dimension of \mathbf{t}_i depends on every input dimension of \mathbf{t}_{i-1} . The final gradient can therefore be expressed as the matrix product of all the Jacobian matrices using the chain rule:

$$\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}) = \frac{\partial \mathbf{t}_k}{\partial \mathbf{t}_0} = \prod_{i=1}^k \frac{\partial \mathbf{t}_i}{\partial \mathbf{t}_{i-1}}$$

Generally, neural networks have high-dimensional input vectors \mathbf{x}_0 and scalar-valued loss functions g . Therefore, due to computational efficiency, the product of all the Jacobians is evaluated in reverse order, from the output $\frac{\partial g}{\partial \mathbf{t}_{k-1}}$ towards the input $\frac{\partial \mathbf{t}_1}{\partial \boldsymbol{\theta}}$. To illustrate this, let $\boldsymbol{\theta} \in \mathbb{R}^n$ and suppose we have a loss function $g = (2\boldsymbol{\theta} + 1)^2$. The intermediate functions are $\mathbf{t}_1 = 2\boldsymbol{\theta}$, $\mathbf{t}_2 = \mathbf{t}_1 + 1$, $g = \mathbf{t}_2^2$. Then we have $\partial g / \partial \mathbf{t}_2 \in \mathbb{R}^{1 \times n}$, $\partial \mathbf{t}_2 / \partial \mathbf{t}_1 \in \mathbb{R}^{n \times n}$ and $\partial \mathbf{t}_1 / \partial \boldsymbol{\theta} \in \mathbb{R}^{n \times n}$. The final gradient is the matrix product of these Jacobians $\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta}) = \frac{\partial g}{\partial \mathbf{t}_2} \frac{\partial \mathbf{t}_2}{\partial \mathbf{t}_1} \frac{\partial \mathbf{t}_1}{\partial \boldsymbol{\theta}}$. Multiplying the Jacobians in reverse order (evaluating the Jacobians left to right) involves two $[1 \times n] \times [n \times n]$ matrix multiplications, however, going in the standard order (evaluating the Jacobians right to left) involves a computationally intensive $[n \times n] \times [n \times n]$ matrix multiplication followed by a $[1 \times n] \times [n \times n]$ matrix multiplication. Generally, for neural networks where there are high-dimensional inputs and a scalar output, that is $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$, where $n \gg m$, it is more computationally efficient to multiply the Jacobians in reverse order. This method is called *reverse-mode auto-differentiation* or *backpropagation*. Conversely, for functions $g : \mathbb{R}^n \rightarrow \mathbb{R}^m$, where $n \ll m$, *forward-mode auto-differentiation* is more computationally efficient where the Jacobians are multiplied from front to back.

Therefore, the chain rule reduces the process of evaluating the gradient $\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta})$ to the evaluation of a product of Jacobian matrices. The *forward pass* takes a batch of data $\{(\mathbf{x}_b, \mathbf{y}_b)\}_{b=1}^B$ alongside the parameters $\boldsymbol{\theta}$ and *forwards* them through the intermediate functions in the network to compute (and cache) all the intermediate values \mathbf{t}_i , as well as the final loss function g . The *backward pass* propagates the gradient $\nabla_{\boldsymbol{\theta}} g(\boldsymbol{\theta})$ backwards through the intermediate functions in the network by computing the Jacobian matrix $\frac{\partial \mathbf{t}_i}{\partial \mathbf{t}_{i-1}}$ for each intermediate value \mathbf{t}_i and matrix multiplying them with the running product or *adjoint*, $\bar{\mathbf{t}}_i = \frac{\partial g}{\partial \mathbf{t}_i}$, according to the chain rule. The adjoint describes the gradient of the final output g with respect to an intermediate variable \mathbf{t}_i and is the sum of the gradients of the final output g with respect to an intermediate variable \mathbf{t}_i from all subsequent nodes that depend on \mathbf{t}_i . Mathematically, this can be expressed as $\bar{\mathbf{t}}_i = \sum_j \bar{\mathbf{t}}_j \frac{\partial \mathbf{t}_j}{\partial \mathbf{t}_i}$, where \mathbf{t}_j are the subsequent nodes that are directly dependent on \mathbf{t}_i . Thus, the backward pass involves computing the gradient for intermediate function, starting from the final output g and working backwards, updating the adjoint $\bar{\mathbf{t}}_i$ by summing the contributions from all paths that pass through \mathbf{t}_i .

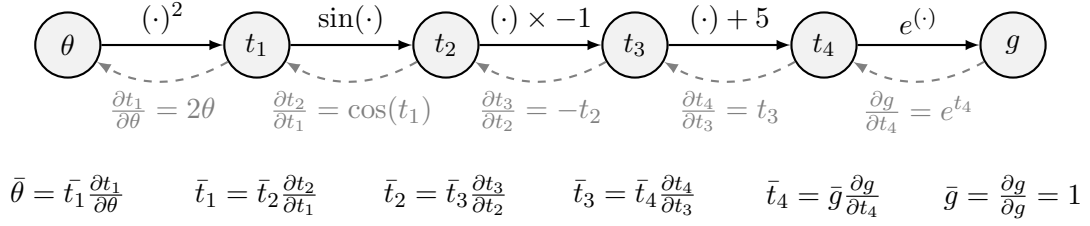


Figure 1.2: Computational graph of the univariate function $g(\theta) = e^{-\sin(\theta)^2+5}$ with nodes representing intermediate variables and each arrow representing a differentiable intermediate function applied to the previous variable. The forward arrows shown the forward pass from input θ and reverse arrows showing the local derivatives calculated in the backward pass through backpropagation. The local gradient of the output node is always initialised to one as $\bar{g} = \frac{\partial g}{\partial g} = 1$. The calculations below show how the gradients are backpropagated through the network, using the chain rule to multiply each local derivative with the adjoint $\bar{t}_i = \frac{\partial g}{\partial t_i}$.

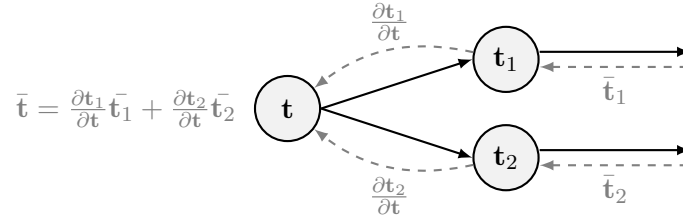


Figure 1.3: Backpropagation through intermediate nodes. The adjoint \bar{t} is calculated as the sum of the partial derivatives of subsequent variables t_1 and t_2 with respect to t , multiplied by their respective adjoints \bar{t}_1 and \bar{t}_2 .

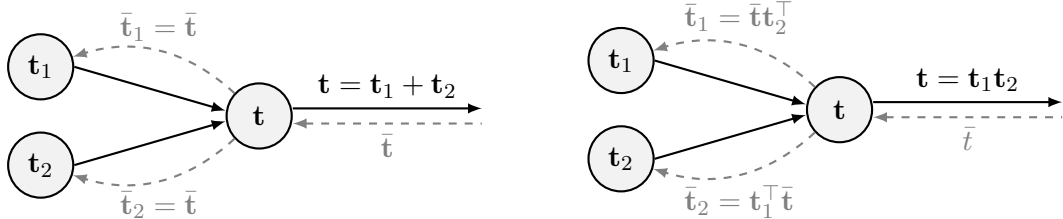


Figure 1.4: Gradients are equally distributed between the two operands in an addition operation. That is, for $t = t_1 + t_2$ the gradients are distributed so $\bar{t}_1 = \bar{t}$ and $\bar{t}_2 = \bar{t}$. For matrix multiplication, gradients are exchanged. That is, for $t = t_1 t_2$ the gradients are exchanged so $\bar{t}_1 = \bar{t} t_2^\top$ and $\bar{t}_2 = t_1^\top \bar{t}$.



Figure 1.5: Let f be an element-wise function such that $\mathbf{t} = f(\mathbf{t}_1)$. For an element-wise function gradients are multiplied by the derivative $f' = \frac{\partial \mathbf{t}}{\partial \mathbf{t}_1}$. Let r be a reshaping operation. The gradient is the inverse r^{-1} reshaping operation on the previous adjoint.

1.4 Feedforward Neural Networks

Previous sections introduced the concept of defining arbitrary differentiable functions f , parameterised by $\boldsymbol{\theta}$, that transform inputs \mathbf{x} into predicted outputs $\hat{\mathbf{y}}$. These models can be optimised using stochastic gradient descent with respect to any differentiable loss surface $g(\boldsymbol{\theta})$. Finally, this section shifts to the possible design of the function f , specifically focusing on neural networks.

A (*feedforward*) *neural network* is simply a function $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ built up of repeating layers consisting of matrix multiplications and element-wise non-linearities. Let $\mathbf{x} \in \mathbb{R}^n$ be an input and suppose the network consists of k layers. A *hidden layer* in a neural network consists of a linear transformation followed by a non-linear *activation function*. For $i = 1, \dots, k-1$ the i -th hidden layer can be written as:

$$\mathbf{h}_i = \sigma(\mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$$

where \mathbf{h}_{i-1} is the output from the previous layer (with $\mathbf{h}_0 = \mathbf{x}$ being the input layer), $\mathbf{W}_i \in \mathbb{R}^{H_i \times H_{i-1}}$ is the weight matrix (where H_i is the number of neurons in this layer and H_{i-1} is the number of neurons in the previous layer), $\mathbf{b}_i \in \mathbb{R}^{H_i}$ is the bias vector, and σ is a nonlinear activation function applied element-wise. The weight matrix \mathbf{W}_i contains all the weights for every combination of neurons between the current layer i and previous layer $i-1$ where the j -th row is the H_{i-1} weights to apply to the j -th neuron in the current layer. Each weight amplifies or attenuates the input to a single neuron, signifying the importance of the input with respect to the neurons output. The bias allows the neuron to further adjust the output independently of its inputs. Common choices for σ include the sigmoid function $1/(1 + e^{-z})$, the hyperbolic tangent function $\tanh(z)$, or the rectified linear unit (ReLU) $\max(0, z)$. The output layer consists of the final linear transformation yielding the output vector $\mathbf{z}_k = \mathbf{W}_k \mathbf{h}_{k-1} + \mathbf{b}_k$. Combining these, the neural network can be formally expressed as a composite function:

$$\begin{aligned} f(\mathbf{x}) &= (\mathbf{h}_k \circ \mathbf{h}_{k-1} \circ \dots \circ \mathbf{h}_1)(\mathbf{x}) \\ &= (\mathbf{W}_k \sigma(\mathbf{W}_{k-1} \dots \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \dots + \mathbf{b}_{k-1}) + \mathbf{b}_k) \end{aligned}$$

This text shall specifically focus on applying neural networks to sequence modelling which involves generating a probability distribution function over an set of m possible output tokens. Generating a probability distribution from a neural network layer involves interpreting the output

layer \mathbf{z}_k as a vector of unnormalised log probabilities or *logits*. The *softmax* function can then be applied to transform the logits into a probability distribution over the set of possible outputs, such that it is normalised with $\sum_{i=1}^m \hat{y}_i = 1$ and $\hat{y}_i > 0$, for all $i \in \{1, \dots, m\}$:

$$\hat{y}_i = \text{softmax}_i(\mathbf{z}_k) = \frac{\exp(z_{k,i})}{\sum_{j=1}^m \exp(z_{k,j})}$$

for each element \hat{y}_i in the output vector $\hat{\mathbf{y}} \in \mathbb{R}^m$. Note that $z_{k,i}$ denotes the i -th element of \mathbf{z}_k . Softmax uses the exponential function to ensure that each \hat{y}_i is non-negative, then the denominator acts as the normalisation constant, ensuring that the values sum to one. The negative log-likelihood is the cross entropy loss:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\log p(\mathbf{y}|\mathbf{x}) = -\sum_{i=1}^m y_i \log \hat{y}_i = -\log \hat{y}_{\mathbb{I}(\mathbf{y}=1)}$$

where \mathbf{y} is a one-hot vector, and $\mathbb{I}(\mathbf{y}_i = 1)$ is an indicator function that selects the element of $\hat{\mathbf{y}}$ corresponding to the index i where $y_i = 1$.

Nonlinear activation functions are essential components within neural networks, without them neural networks are reduced down to a series of linear transformations, which is equivalent to a singular linear transformation, regardless of the number of layers:

$$\begin{aligned} f(\mathbf{x}) &= (\mathbf{h}_k \circ \mathbf{h}_{k-1} \circ \dots \circ \mathbf{h}_1)(\mathbf{x}) \\ &= (\mathbf{W}_k(\mathbf{W}_{k-1} \dots (\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \dots + \mathbf{b}_{k-1}) + \mathbf{b}_k) \\ &= \{\mathbf{W}_k \mathbf{W}_{k-1} \dots \mathbf{W}_1\} \mathbf{x} + \{\mathbf{W}_k \dots \mathbf{W}_2 \mathbf{b}_1 + \dots + \mathbf{W}_k \mathbf{b}_{k-1} + \mathbf{b}_k\} \\ &= \tilde{\mathbf{W}} \mathbf{x} + \tilde{\mathbf{b}} \end{aligned}$$

Feedforward neural networks can approximate any continuous function [48]. Consider the single-layer network $f(\mathbf{x}) = \sum_{i=1}^H \alpha_i \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i)$ where $\mathbf{x} \in \mathbb{R}^n$ is the input vector, \mathbf{w}_i are the weight vectors, b_i are the biases, α_i are the output layer weights and H is the number of neurons in the hidden layer. Let $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ be a bounded, non-linear activation function and let $C(K)$ denote the space of continuous functions on a compact subset $K \subseteq \mathbb{R}^n$. Then, for any function $\phi \in C(K)$ and any $\varepsilon > 0$, there exists a neural network $f(\mathbf{x}) = \sum_{i=1}^H \alpha_i \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i)$ such that $|\phi(\mathbf{x}) - f(\mathbf{x})| < \varepsilon, \forall \mathbf{x} \in K$. Hence f can approximate ϕ to any desired degree of accuracy.

The parameters of a feedforward neural network are $\theta = \{\mathbf{W}_k, \dots, \mathbf{W}_1, \mathbf{b}_k, \dots, \mathbf{b}_1\}$ which are initialised randomly and then learned during training. The weights are flattened and concatenated alongside the biases into one single vector θ for optimisation. Backpropagation is used to update the parameters θ in a neural network by propagating the gradient of the loss surface $\nabla_{\theta} g(\theta)$ with respect to these parameters backwards through the network. First, let $\mathbf{z}_i = \mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i$ denote the pre-activation of the i -th layer, with $\mathbf{h}_i = \sigma(\mathbf{z}_i)$. The gradient of the output layer is $\bar{\mathbf{z}}^{(k)}$. Propagating the gradient further back through the network for

$i = k, \dots, 1$ gives:

$$\begin{aligned}\bar{\mathbf{z}}_{i-1} &= (\mathbf{W}_i)^\top \bar{\mathbf{z}}_i \odot \sigma'(\mathbf{z}_{i-1}) \\ \bar{\mathbf{W}}_i &= \bar{\mathbf{z}}_i (\mathbf{h}_{i-1})^\top \\ \bar{\mathbf{b}}_i &= \bar{\mathbf{z}}_i\end{aligned}$$

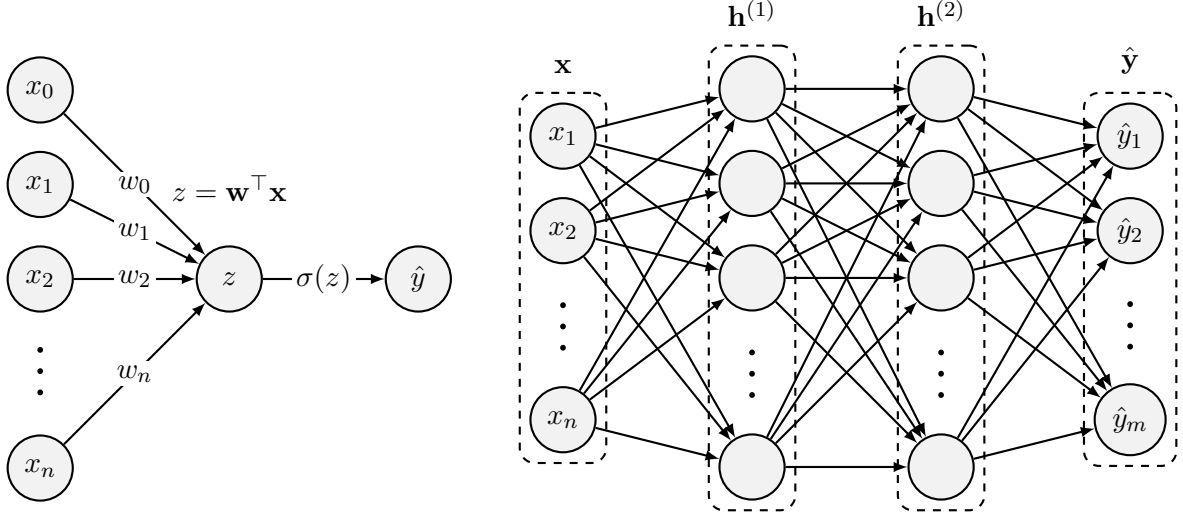


Figure 1.6: **Left:** A single neuron. Inputs x_i interact multiplicatively with the weights w_i and are summed into the pre-activation z which is then sent through the activation function to produce an output \hat{y} . The bias is represented as $x_0 = b$. **Right:** Feedforward neural network architecture with two hidden layers. Each neuron is connected to all neurons in the previous layer, but are not connected to neurons within the same layer. Therefore neurons within each layer can be computed in parallel via matrix multiplication.

Layer normalisation [1] is a technique used to stabilise and accelerate the training of neural networks by normalising the inputs to each layer. It addresses the issue of internal covariate shift, which occurs when the distribution of inputs to a layer changes during training, thereby making the training process less stable and slower. Layer normalisation is applied to the pre-activation inputs \mathbf{z}_i of a layer. First, the mean μ_i and variance σ_i^2 of the pre-activations \mathbf{z}_i in each layer are calculated:

$$\mu_i = \frac{1}{H_i} \sum_{j=1}^{H_i} z_{i,j} \quad \sigma_i^2 = \frac{1}{H_i} \sum_{j=1}^{H_i} (z_{i,j} - \mu_i)^2$$

where H_i is the size of the i -th layer, and $z_{i,j}$ is the activation of the j -th neuron in the i -th layer. The pre-activations are then normalised using the computed mean μ_i and variance σ_i^2 to normalise each activation:

$$\hat{z}_{i,j} = \frac{z_{i,j} - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$$

where ε is a small constant added to remove the possibility of division by zero. Layer normalisation includes a learned affine transformation through two learnable parameters α_i and β_i to allow the network to scale and shift the normalised activations. These help to reintroduce variance and any offset that might have been removed during normalisation:

$$\tilde{z}_{i,j} = \alpha_i \hat{z}_{i,j} + \beta_i$$

Let \odot denote piecewise multiplication. The layer normalisation for the i -th layer can be expressed succinctly as:

$$\mathbf{h}_i = \sigma \left(\alpha_i \odot \frac{\mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}} + \beta_i \right)$$

Chapter 2

The Rise of the Transformer

Learning the joint probability function of sequences of tokens is the foundation of sequence modelling. However, this is inherently complex due to the *curse of dimensionality*, whereby the joint distribution among an increasing number of discrete random variables leads to an exponential increase in the number of parameters required to model such a sequence. To illustrate this complexity, consider the task of modelling the joint probability distribution $p(x_1, \dots, x_{10})$ of a sequence of 10 consecutive tokens, where each token $x_i \sim \mathcal{V}$ is drawn from a fixed set \mathcal{V} of 1,000 possible tokens. Such a model would require $1,000^{10} - 1 = 10^{30} - 1$ parameters, a number that vastly exceeds the estimated count of stars in the observable universe by several orders of magnitude.

Recently, significant research into neural sequence modelling has taken place to fight the curse of dimensionality and make sequence modelling tractable. This chapter explores the evolution of these neural sequence modelling architectures, leading up to the development of attention mechanisms and the influential Transformer model [53].

2.1 Vector Embeddings

Let (x_1, \dots, x_T) be a sequence of T tokens, drawn from a fixed set \mathcal{V} of size V of all possible tokens. Neural sequence models require that tokens are encoded into their equivalent vector representation to allow for parallelised linear operations to be performed. One-hot encoding is a sparse, discrete encoding whereby tokens are encoded into a vector of size V that is all zero, except for a singular one at the index of the token in the set \mathcal{V} . Denote, $\mathbb{I}_t \in \mathbb{R}^V$ to be the one-hot encoding of the t -th token in the sequence. The sequence (x_1, \dots, x_T) of tokens can therefore be encoded as the sequence of one-hot vectors $(\mathbb{I}_1, \dots, \mathbb{I}_T)$. As an aside, some tokens may be rare or even not present in the training data. Hence, a special *unknown token* is reserved to which these rare or absent tokens are remapped during preprocessing. This reduces the required size of the set of possible tokens \mathcal{V} and therefore the dimensionality of the one-hot vectors. However,

one-hot encoding also suffers from the curse of dimensionality, with the dimensionality of the one-hot vectors scaling with the size of the set \mathcal{V} . Assigning each token to a continuous vector representation (embedding) within a learned embedding space removes this issue. Compute each symbols x_t vector embedding representation \mathbf{x}_t within the learned embedding space by projecting the one-hot vector \mathbb{I}_t with a linear transformation $\mathbf{x}_t = \mathbf{C}\mathbb{I}_t$, where $\mathbf{C} \in \mathbb{R}^{V \times C}$ is a learned embedding matrix and C is the embedding dimensionality. Since \mathbb{I}_t is a one-hot vector, this operation is equivalent to a row lookup for the row \mathbf{x}_t corresponding to token x_t in the embedding matrix \mathbf{C} (i.e. $\mathbf{x}_t = \mathbf{C}[x_t]$, the t -th row in \mathbf{C} corresponds to the vector embedding for the t -th token).

The existence of the embedding space is advantageous for a number of reasons. Firstly, the embedding space is continuous, allowing for a dense representation of information when compared to sparse, one-hot encodings. Furthermore, the embedding space performs dimensionality reduction by mapping the one-hot encoding of tokens $\mathbb{I}_t \in \mathbb{R}^V \mapsto \mathbf{x}_t \in \mathbb{R}^C$ from a high-dimensional vector space of size V to a lower-dimensional embedding space of size C , where $C \ll V$. This reduces the number of parameters and computational complexity of the model. Moreover, the embedding matrix \mathbf{C} is used for embedding all possible tokens in the input sequence, allowing for parameter sharing, which enables the model to learn consistent representations across different contexts. The elements within vector embeddings $\{x_{t,1}, \dots, x_{t,C}\}$ act as a set of features to identify the pairwise similarity between two tokens. This similarity can be measured by the cosine similarity $(\mathbf{x}_i \cdot \mathbf{x}_j) / \|\mathbf{x}_i\| \|\mathbf{x}_j\|$ or the Euclidean distance $\|\mathbf{x}_i - \mathbf{x}_j\|$.

Significant amounts of research has been performed into learning efficient, high quality, vector embeddings from large datasets [31, 32, 38, 39]. Applying vector embeddings to represent words has yielded intriguing results when performing vector operations within the embedding space. A notable example [33] is the use of vector arithmetic on word vectors, such as $\mathbf{C}[\text{king}] - \mathbf{C}[\text{man}] + \mathbf{C}[\text{woman}] \approx \mathbf{C}[\text{queen}]$.

2.2 Sequence Modelling

Let $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ be a sequence of vector embeddings for the corresponding sequence of tokens $x = (x_1, \dots, x_T)$. On a basic level, sequence models are functions that assign probabilities to sequences of tokens. Formally, a statistical sequence model f can be represented by the conditional probability of the next token x_t given all the previous tokens (x_1, \dots, x_{t-1}) :

$$p(x) = \prod_{t=1}^T p(x_t | \{x_1, \dots, x_{t-1}\}) = \prod_{t=1}^T f(\mathbf{x}_1, \dots, \mathbf{x}_{t-1})$$

where the statistical sequence model uses vector embeddings corresponding to the tokens as input. The joint distribution $p(x)$ of a sequence can be expressed as a product of conditional probabilities. Specifically, each term in the product represents the probability of a token given

all preceding tokens in the sequence, and is defined as:

$$p(x_t|x_1^{t-1}) = \frac{p(x_1^{t-1}, x_t)}{p(x_1^{t-1})} = \frac{p(x_1^t)}{p(x_1^{t-1})}$$

where $x_i^j = (x_i, x_{i+1}, \dots, x_j)$ is the sequence of tokens from the i -th token to the j -th token. By recursively applying this conditional probability decomposition, the model constructs the joint distribution over the entire sequence of tokens. The likelihood of observing a sequence x can be maximised by minimising the negative log-likelihood:

$$-\log p(x) = -\sum_{t=1}^T \log p(x_t|x_1^{t-1})$$

For some statistical sequence models, such as language models, a good estimation for $p(x_t|x_1^{t-1})$ is $p(x_t|x_{t-n+1}^{t-1})$ for a suitable choice of n . In other words, the probability of the current token x_t only depends on the preceding $n-1$ tokens x_{t-n+1}^{t-1} . This is the Markov assumption.

$$p(x_t|x_1^{t-1}) \approx p(x_t|x_{t-n+1}^{t-1})$$

This is called an n -gram model (or unigram, bigram or trigram for $n = 1, 2, 3$ respectively). Every probability is estimated by relative frequency. Let $c(x_i^j)$ denotes the count of how often the sequence x_i^j occurs in the sequence. The maximum likelihood estimate for $p(x_t|x_1^{t-1})$ is:

$$p(x_t|x_1^{t-1}) = \frac{c(x_{t-n+1}^{t-1}, x_t)}{c(x_{t-n+1}^{t-1})}$$

However, it is highly likely that some combinations of sequences of length n may never occur in the training data. Using the maximum likelihood estimate a probability of zero is assigned to such sequences. Clearly assigning a probability of zero to such sequences is incorrect, as they may occur later. Therefore, various methods of smoothing, whereby the maximum likelihood estimate is adjusted to produce more accurate probabilities and give non-zero probabilities to such sequences, have been introduced and compared [8]. A drawback of n -grams is that they struggle to generalise through giving similar probabilities to similar sequences. Furthermore, n -gram models are generally limited to only use the few previous tokens as context due to data scarcity, with some longer length sequences never appearing in training data.

Perplexity is a measure of how well a sequence model approximates to its data generating distribution and is used to evaluate the performance of the model. Formally, for a given sequence $\mathbf{y} = (\mathbf{y}_1, \dots, \mathbf{y}_{T'})$ and a sequence model that assigns a probability $p(\mathbf{y})$ to the sequence, the perplexity is defined as the exponentiated average negative log-likelihood per token:

$$\text{pp}(\mathbf{y}) = \exp \left(-\frac{1}{T'} \sum_{t=1}^{T'} \log p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}) \right) = \left(\prod_{t=1}^{T'} \frac{1}{p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\})} \right)^{\frac{1}{T'}}$$

Perplexity can be interpreted as the geometric mean of the inverse probability of the sequence, normalised by the sequence length T' . Lower perplexity generally indicates greater model performance as the model assigns higher probabilities to the true sequence. A perplexity of $k \in \mathbb{Z}$ implies that the model is as uncertain as if it had to choose from a uniform distribution between k possible next tokens. Therefore, a perplexity of one indicates a model which completely represents the dataset.

For the rest of this text, neural sequence models f will take input sequences of vector embeddings $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ of the corresponding tokens $x = (x_1, \dots, x_T)$. Moreover, unless explicitly stated otherwise, any reference to tokens will imply their corresponding vector embeddings. The output $\hat{\mathbf{y}}_t$ of a sequence model will be the probability distribution of the next token in the sequence.

2.3 Generation

This section covers various algorithms used to generate sequences by iteratively sampling from a sequence of predicted probability distributions over a sequence of tokens. The sequence is initialised to only contain the special start-of-sequence token. Then, generation continues until an end-of-sequence token is generated or a predefined maximum sequence length T_{\max} is reached. Formally, given a sequence model that outputs a probability distribution $p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{x})$ over the set of possible tokens \mathcal{V} , where \mathbf{x} is the input sequence, if one exists, and $\{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}$ is the previous $t - 1$ tokens in the sequence.

At each iteration, the *greedy search* algorithm selects the token with the highest probability. That is, the greedy search algorithm predicts the next output by taking the *argmax* over the predicted probability distribution:

$$\mathbf{y}_t = \arg \max_i (\hat{y}_{t,i}) = \arg \max_{y_i \in \mathcal{V}} p(y_i | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{x})$$

This method is simple and very efficient, however, it is not guaranteed to yield the optimal sequence due to making locally optimal choices without considering the global sequence probability. Furthermore, it can lead to repetitive and deterministic outputs. Another method is just to sample according to the multinomial distribution output by the sequence model: $\mathbf{y}_t \sim p(\{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\})$.

At each iteration, the *top-k sampling* technique limits the sampling pool to the top $k \in \{1, \dots, V\}$ tokens with the highest probabilities. The probabilities of the selected k tokens are then normalised to sum to one. The next token can then be randomly sampled from this reduced set of k tokens based on their normalised probabilities. Top- k sampling reduces the chance of sampling tokens with a low probability.

The *top-p (nucleus) sampling* technique considers the smallest subset of tokens whose cumulative probability exceeds a specified threshold $p \in [0, 1]$, dynamically adjusting the sample pool

size based on the predicted distribution. Then, similar to top- k sampling, the next token can be randomly sampled from this reduced set of tokens based on their normalised probabilities. Top- p sampling is more adaptive than top- k sampling as it can balance between selecting the most probable tokens whilst also allowing some tokens with a low probability to be sampled if p is set sufficiently high enough.

Beam search is another sampling technique that aims to find the most likely sequence of tokens by maintaining k multiple hypotheses (beams) with the highest cumulative probabilities, instead of only the single most likely sequence as in greedy search. The algorithm first starts with an initial token and computes the probabilities of all possible next tokens. Then, the top k tokens are then selected based on their probabilities. The algorithm then iteratively repeats the following. For each of the k sequences, compute the probabilities of all possible next tokens. Generate new sequences by appending each possible next token to each of the k sequences. Then compute the cumulative probability for each new sequence and retain only the top k sequences with the highest cumulative probabilities. The final output is typically the sequence with the highest cumulative probability among the k sequences. Generally, the cumulative log-probabilities are used instead, to avoid underflow:

$$\log p(\mathbf{y}_1^t) = \sum_{i=1}^t \log p(\mathbf{y}_i | \mathbf{y}_1^{i-1})$$

Beam search is more likely to find globally optimal sequences compared to greedy search, due to its ability to maintain multiple sequences and not immediately discard any potentially valuable paths.

In order to control the randomness in the sampling process, *temperature scaling* can be applied to the predicted probability distribution before sampling. The temperature parameter, denoted as $\tau > 0$, is used to scale the logits (unnormalised probabilities) output by the model, adjusting the sharpness of the distribution. If the temperature is set too low then the probability distribution can be too concentrated, causing the model to favour a small subset of outputs with higher probability. This can lead to deterministic, repetitive behaviour, where the model repeatedly samples from the same limited subset of possible outputs. Conversely, if the temperature is set too high, the distribution becomes more uniform, increasing the likelihood of selecting lower-probability outcomes, which can introduce more randomness and noise in the samples.

2.4 Neural Sequence Model

Bengio et al. [3] first applied neural networks, specifically a two layer feedforward network, to fight the curse of dimensionality within sequence modelling by learning a distributed representation of tokens, allowing for generalisation between tokens in local neighbourhoods within an embedding space \mathbb{R}^C where C is the dimensionality of the embedding space. The previous T tokens of

context are concatenated and fed into a neural network consisting of a singular hidden layer, and an output layer followed by the softmax function, which gives an estimate of the probability distribution of the token given the previous T tokens. Generalisation is achieved as a sequence of unseen tokens gets assigned a high probability if it is composed of tokens which have a nearby representation in the embedding space to similar tokens previously seen in other sequences. Within this model the vector embeddings and a statistical sequence model are jointly learned. It has been shown that neural network based sequence models meaningfully outperform n -gram models due to their ability to generalise [3].

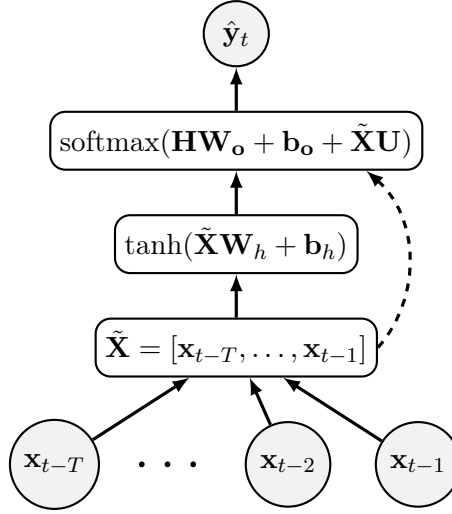


Figure 2.1: Neural sequence model comprised of a two layer feedforward network. Dashed arrows represent optional residual connections with weights \mathbf{U} . The output logits are transformed into estimated probabilities $\hat{\mathbf{y}}$ via softmax.

Let T be the context length or block size, representing the number of previous tokens to use to predict the next token. Let B be the batch size, the number the number of sequences (each of length T) to process in parallel. Let C be the dimensionality of the embedding space. The input to the model f is a batch of vector embeddings $\mathbf{X} = [\mathbf{X}_1, \dots, \mathbf{X}_B] \in \mathbb{R}^{B \times T \times C}$ where each matrix $\mathbf{X}_b = [\mathbf{x}_{b,t-T}, \mathbf{x}_{b,t-T+1}, \dots, \mathbf{x}_{b,t-1}] \in \mathbb{R}^{T \times C}$ is the vector embeddings of the previous T tokens in the b -th sequence of the batch. Here, $\mathbf{x}_{b,t-i} \in \mathbb{R}^C$ represents the embedding of the i -th previous token in the b -th sequence. The model is trying to predict $\mathbf{x}_{b,t}$ for each batch. Next the embeddings are flattened to be of the form $\tilde{\mathbf{X}} \in \mathbb{R}^{B \times (T \times C)}$ such that they can be fed into the network. This leads to the concatenation of the T embeddings of size C in each batch. The hidden layer of the network is a linear transformation with the tanh non-linearity which is applied pointwise:

$$\mathbf{H} = \tanh(\tilde{\mathbf{X}}\mathbf{W}_h + \mathbf{b}_h) \in \mathbb{R}^{B \times H}$$

where $\mathbf{W}_h \in \mathbb{R}^{(T \times C) \times H}$, $\mathbf{b}_h \in \mathbb{R}^{1 \times H}$ are the weights and biases in the hidden layer respectively,

and H is the number of neurons in the hidden layer. Notice that the order of matrix multiplication has been swapped compared to the standard notation, with the input matrix $\tilde{\mathbf{X}}$ on the left and the weight matrix \mathbf{W}_h on the right. Here $\mathbf{b}_h \in \mathbb{R}^{1 \times H}$ to ensure proper broadcasting. The output layer is a linear transformation to obtain the logits, combined with a softmax to yield the conditional probability distribution over all tokens in the set \mathcal{V} .

$$\hat{\mathbf{Y}} = \text{softmax}(\mathbf{H}\mathbf{W}_o + \mathbf{b}_o) \in \mathbb{R}^{B \times V}$$

where $\mathbf{W}_o \in \mathbb{R}^{H \times V}$, $\mathbf{b}_o \in \mathbb{R}^{1 \times V}$ are the weights and biases in the output layer respectively. The softmax function transforms the logits $\mathbf{H}\mathbf{W}_o + \mathbf{b}_o = \mathbf{Z} \in \mathbb{R}^{B \times V}$ into a conditional probability distribution $\hat{\mathbf{Y}}$ over all possible tokens for each instance in the batch. Therefore, each row of $\hat{\mathbf{Y}}$ is a conditional probability distribution and consists of elements that are all positive and sum to one. Training involves looking for a set of parameters θ which minimise the cross-entropy loss over the batch

$$L = -\frac{1}{B} \sum_{b=1}^B \sum_{i=1}^V y_{b,i} \log \hat{y}_{b,i} = -\frac{1}{B} \sum_{b=1}^B \log \hat{y}_{b, \mathbb{I}(y_{b,i}=1)}$$

where $\hat{y}_{b, \mathbb{I}(y_{b,i}=1)}$ is the predicted probability for the true token y_i in the b -th sequence.

Sampling from the probability distribution yields the next token in the sequence. Let $\mathbf{X} \in \mathbb{R}^{1 \times T \times C}$ be the initial input sequence of tokens with a size equal to the context length. If the initial input is shorter than the context length, pad \mathbf{X} on the left with the embedding of a special padding token (or zeros). Define $\mathbf{X}_{t-T}^t = [\mathbf{x}_{t-T}, \mathbf{x}_{t-T+1}, \dots, \mathbf{x}_{t-1}]^\top$ as the previous T tokens, i.e. *context window*, in \mathbf{X} . Note that the batch dimension is redundant here. Generating from the model involves iterating through the following process for $t \geq T$:

$$\hat{\mathbf{Y}}_t = f(\mathbf{X}_{t-T}^t), \quad x_t \sim \hat{\mathbf{Y}}_t, \quad \mathbf{X} = [\mathbf{X}, \mathbf{C}(x_t)]$$

First the output probability distribution $\hat{\mathbf{Y}}$ is obtained through a forward pass through the model f . Then sample the next token from the probability distribution $\hat{\mathbf{Y}}$. This can be done using a sampling technique like multinomial sampling. Then add the vector embedding representation of the new token to the sequence \mathbf{X} via concatenation. The next iteration then shifts the context window \mathbf{X}_{t-T}^t to include the newly generated token and drop the oldest token to maintain the context length T .

There is only one hidden layer, \mathbf{H} , in this model which uses the tanh nonlinearity. Additional hidden layers could be added to possibly increase the complexity of the model, however this could lead to possible overfitting. Note that Bengio et al. [3] also included an extra optional residual connection $\mathbf{U} \in \mathbb{R}^{(T \times C) \times V}$ between the input layer and the output layer for uninterrupted gradient flow. If no residual connections are desired, this can be set to the zero matrix. The entire model f can be expressed in one line:

$$\hat{\mathbf{Y}} = f(\mathbf{X}) = \text{softmax}(\text{ReLU}(\tilde{\mathbf{X}}\mathbf{W}_h + \mathbf{b}_h)\mathbf{W}_o + \mathbf{b}_o + \tilde{\mathbf{X}}\mathbf{U})$$

The model parameters (including the residual connection and embedding matrix) is the set $\theta = \{\mathbf{C}, \mathbf{W}_h, \mathbf{b}_h, \mathbf{W}_o, \mathbf{b}_o, \mathbf{U}\}$, therefore the number of parameters in total is

$$\begin{aligned} |\theta| &= VC + TCH + H + HV + V + TCV \\ &= V[C + H + 1 + TC] + H[1 + TC] \end{aligned}$$

2.5 Recurrent Neural Network

Feedforward neural networks are insufficient for sequence modelling due to inherent limitations in their architecture. Looking closely at their architecture, each hidden layer in a feedforward neural network is described by

$$\mathbf{h}_i = \sigma(\mathbf{W}_i \mathbf{h}_{i-1} + \mathbf{b}_i)$$

where $\mathbf{h}_0 = \mathbf{x} \in \mathbb{R}^n$ is the input vector, $\mathbf{W}_i, \mathbf{b}_i$ are the weight matrices and bias vectors for the i -th layer respectively and σ is the activation function. Feedforward neural networks are functions $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ with fixed-size inputs and outputs, therefore they are unable to process sequences of varying length. Furthermore, there is no memory mechanism within the layers to retain information about previous tokens as every layer feeds-forward into the next.

Recurrent neural networks (RNNs) [43, 55] address these issues. A recurrent neural network is a network architecture that handles sequences of vectors $(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_T)$ using the recurrence formula

$$\mathbf{h}_t = f_{\mathbf{W}}(\mathbf{x}_t, \mathbf{h}_{t-1}), \quad \hat{\mathbf{y}}_t = g(\mathbf{h}_t)$$

where f, g are functions detailed later, and the parameters \mathbf{W} remain constant across every time step, allowing an arbitrary number of vectors to be processed. The hidden vector \mathbf{h}_t allows for information to be passed from the current step of the network to the next and serves as a cumulative representation of all the previous vectors $(\mathbf{x}_1, \dots, \mathbf{x}_{t-1})$. The recurrence formula updates this representation based on the next vector \mathbf{x}_t . The initial hidden state is either initialised to $\mathbf{h}_0 = \mathbf{0}$ or is considered to be a parameter, allowing the initial hidden state to be learned. Figure 2.2 shows the unfolded representation of a recurrent neural network. This chain-like structure demonstrates that recurrent neural networks are inherently designed to process sequences by propagating information in an autoregressive manor. An RNN can learn a probability distribution over a sequence $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ by being trained to predict the next token in a sequence. The output at time step t is the conditional distribution $\hat{\mathbf{y}}_t = p(\mathbf{x}_t | \{\mathbf{x}_1, \dots, \mathbf{x}_{t-1}\})$. This probability can be computed via a softmax function over a multinomial distribution of V possible tokens. The probability of an entire sequence \mathbf{x} of length T can be calculated via:

$$p(\mathbf{x}) = \prod_{t=1}^T p(\mathbf{x}_t | \{\mathbf{x}_1, \dots, \mathbf{x}_{t-1}\})$$

Using this learned distribution, it is possible to generate a new sequence by iteratively sampling a new symbol from the predicted probability distribution at each time step until the end of sequence symbol is encountered, or a predefined generation limit is reached.

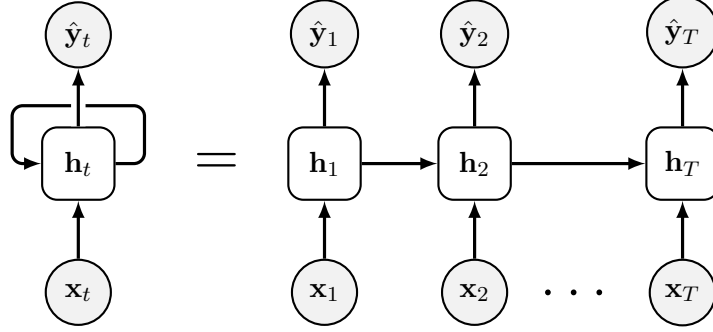


Figure 2.2: Unfolded representation of the recurrent neural network (RNN) architecture. Each hidden state \mathbf{h}_t is computed from the previous hidden state \mathbf{h}_{t-1} and the current input \mathbf{x}_t . The initial hidden state \mathbf{h}_0 is either initialised to zero or learned as a parameter.

The exact recurrence relation can vary depending on the type of recurrent neural network. For the standard recurrent neural network architecture the following relation is used

$$\mathbf{h}_t = f(\mathbf{W}_h[\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b})$$

Here the current input \mathbf{x}_t and the previous hidden vector \mathbf{h}_{t-1} are concatenated into a single vector, and then linearly transformed by the combined weight matrix $\mathbf{W}_h = [\mathbf{W}_{xh} \ \mathbf{W}_{hh}]$, that is, \mathbf{W}_{xh} and \mathbf{W}_{hh} are concatenated horizontally. Hence the equation above is equivalent to writing $\mathbf{h}_t = f(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{b})$. The weight matrix \mathbf{W}_{xh} represents the weights for the connections between the input vectors \mathbf{x}_t and the hidden state vectors \mathbf{h}_t . Likewise, \mathbf{W}_{hh} represents the weights for the connections between the hidden state vectors at the previous time step \mathbf{h}_{t-1} and the current time step \mathbf{h}_t . Note that \mathbf{b} is the additional bias vector. The result from the linear transformation is then compressed by f , a nonlinear function, commonly tanh or sigmoid. Suppose the input vectors \mathbf{x}_t have dimension D and the hidden states \mathbf{h}_{t-1} have dimension H , then $\mathbf{W}_{xh} \in \mathbb{R}^{H \times D}$ and $\mathbf{W}_{hh} \in \mathbb{R}^{H \times H}$, giving $\mathbf{W}_h \in \mathbb{R}^{H \times (D+H)}$. The output vector $\hat{\mathbf{y}}$, with dimension O , is calculated as follows

$$\hat{\mathbf{y}} = g(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{c})$$

where $\mathbf{W}_{hy} \in \mathbb{R}^{O \times H}$ represents the weights for the connections between the hidden states \mathbf{h}_t and the output vectors $\hat{\mathbf{y}}_t$, g is a nonlinear activation function, typically softmax, and \mathbf{c} is an additional bias vector. Notice that the same weight matrices (parameters) \mathbf{W}_{xh} , \mathbf{W}_{hh} , \mathbf{W}_{hy} are shared across the sequence. The loss at an individual time step is as follows

$$L_t = - \sum_{i=1}^V y_{t,i} \log \hat{y}_{t,i}$$

where V is the size of the fixed set \mathcal{V} of possible tokens, $y_{t,i}$ is the i -th element of \mathbf{y}_t , and $\hat{y}_{t,i}$ is the i -th element of $\hat{\mathbf{y}}_t$. The total loss is the summation of the cross-entropy losses at each time step, normalised by the total number of time steps T :

$$L = -\frac{1}{T} \sum_{t=1}^T \sum_{i=1}^V y_{t,i} \log \hat{y}_{t,i} = -\frac{1}{T} \sum_{t=1}^T \log \hat{y}_{t, \mathbb{I}(y_{t,i}=1)}$$

where $\hat{y}_{t, \mathbb{I}(y_{t,i}=1)}$ is the predicted probability for the true token y_i at the t -th time step.

Recurrent neural networks use *backpropagation through time* (BTT) [56], an extension of the general backpropagation algorithm, which is used for updating the weight matrices in training. Figure 2.3 shows that the unfolded recurrent neural network is in essence a regular feedforward neural network, except that parameters are shared throughout the network. Therefore, just as in a regular feedforward neural network, the chain rule can be applied to backpropagate the gradients through the unfolded computational graph of the recurrent neural network. Using backpropagation, the gradients with respect to each parameter are summed across all the positions where that parameter occurs in the computational graph.

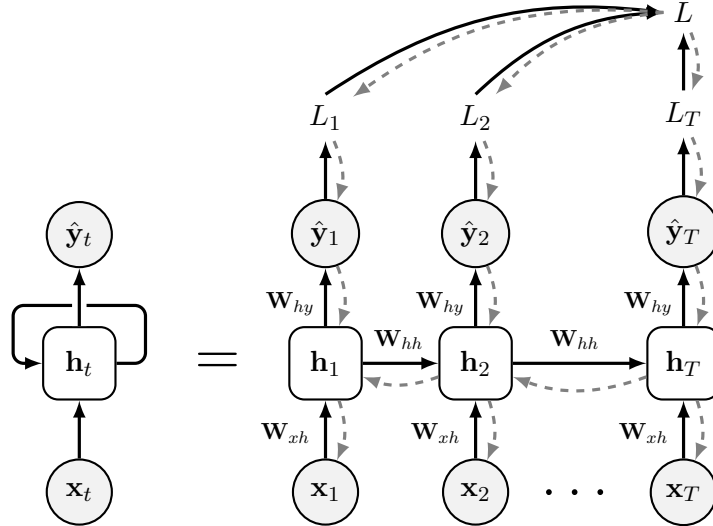


Figure 2.3: Backpropagation through time (BTT). Unrolled recurrent neural network (RNN) computation graph illustrating the forward (solid arrows) and backward (dashed arrows) pass from the loss L towards the inputs \mathbf{x}_t . Biases have been omitted for brevity.

This standard recurrent neural network has a simple architecture, however suffers from *vanishing* or *exploding* gradients. Consider the gradient of the loss with respect to the weight matrix \mathbf{W}_{hh} at an arbitrary time step t .

$$\frac{\partial L_t}{\partial \mathbf{W}_{hh}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{hh}}$$

Remember that \mathbf{h}_t depends on \mathbf{h}_{t-1} as per the recurrence relation $\mathbf{h}_t = f(\mathbf{W}_h[\mathbf{x}_t, \mathbf{h}_{t-1}]^\top + \mathbf{b})$. Backpropagation can then be applied through time until the initial time step and so the equation

becomes

$$\frac{\partial L_t}{\partial \mathbf{W}_{hh}} = \sum_{k=1}^t \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \cdot \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}}$$

Hence the gradient of the final loss L with respect to \mathbf{W}_{hh} (i.e. the adjoint $\bar{\mathbf{W}}_{hh}$) is as follows

$$\bar{\mathbf{W}}_{hh} = \frac{\partial L}{\partial \mathbf{W}_{hh}} = \frac{1}{T} \sum_{t=1}^T \sum_{k=1}^t \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \cdot \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{hh}}$$

Similarly, consider the gradient of the loss with respect to \mathbf{W}_{xh} at an arbitrary time step t .

$$\frac{\partial L_t}{\partial \mathbf{W}_{xh}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{xh}}$$

Notice that both \mathbf{h}_{t-1} and \mathbf{x}_t contribute to the calculation of \mathbf{h}_t , and so, we backpropagate to \mathbf{h}_{t-1} .

$$\frac{\partial L_t}{\partial \mathbf{W}_{xh}} = \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{W}_{xh}} + \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \cdot \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{W}_{xh}}$$

Performing backpropagation through time until the initial time step again, gives

$$\frac{\partial L_t}{\partial \mathbf{W}_{xh}} = \sum_{k=1}^t \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \cdot \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{xh}}$$

Finally, the gradient of the final loss L with respect to \mathbf{W}_{xh} (i.e. the adjoint $\bar{\mathbf{W}}_{xh}$) is as follows

$$\bar{\mathbf{W}}_{xh} = \frac{\partial L}{\partial \mathbf{W}_{xh}} = \frac{1}{T} \sum_{t=1}^T \sum_{k=1}^t \frac{\partial L_t}{\partial \hat{\mathbf{y}}_t} \cdot \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \cdot \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_{xh}}$$

In standard recurrent neural networks, *vanishing* or *exploding* gradients occur due to the recurrent connections, specifically when $\partial \mathbf{h}_t / \partial \mathbf{h}_k$ is computed in the backwards pass.

$$\begin{aligned} \prod_{j=k}^{t-1} \frac{\partial \mathbf{h}_{j+1}}{\partial \mathbf{h}_j} &= \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}} \dots \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \\ &= \prod_{j=k}^{t-1} \text{diag} (f'(\mathbf{W}_{xh} \mathbf{x}_{j+1} + \mathbf{W}_{hh} \mathbf{h}_j + \mathbf{b})) \cdot \mathbf{W}_{hh} \end{aligned}$$

Remember that a standard recurrent neural network uses either tanh or sigmoid as the activation function f . Therefore, values for f' will always be less than 1. Hence, if \mathbf{W}_{hh} has a large dominant eigenvalue greater than 1, the gradients will grow exponentially as the repeated multiplication of \mathbf{W}_{hh} overpowers the repeated multiplication of f' , and so the gradient explodes over long time periods. Conversely, if the dominant eigenvalue of \mathbf{W}_{hh} too small to negate the shrinking effect of f' , the gradients will decay exponentially and vanish over long time periods. With vanishing gradients errors are not propagated back through the network, and so earlier hidden states struggle to learn. This biases the network towards learning short-term dependencies [35]

and can be problematic for modelling sequences which require a large context. Using ReLU as the activation function f can help prevent the gradients from shrinking when $\mathbf{W}_{xh}\mathbf{x}_{j+1} + \mathbf{W}_{hh}\mathbf{h}_j + \mathbf{b} > 0$ since all positive values are mapped to a value of one. Exploding gradients are also a problem as large gradient updates cause the network becomes unstable. Gradient clipping is often used to prevent the gradients from exploding [35].

2.6 Long Short-Term Memory

The *long short-term memory* (LSTM) [22] is a type of recurrent neural network with a recurrence formula designed to address the vanishing gradient problem present in the standard RNN architecture, allowing LSTMs to handle long-term time dependencies. The LSTM recurrence formula involves more computationally complex multiplicative interactions between the inputs \mathbf{x}_t and the hidden states \mathbf{h}_{t-1} . Furthermore, the LSTM recurrence formula also incorporates additive interactions, facilitating the effective propagation of gradients backwards in time (mechanically similar to residual connections for feedforward networks). In addition to the hidden state vector \mathbf{h}_t , a cell state or memory vector, \mathbf{c}_t , is maintained. Within the repeating LSTM module, gating mechanisms allow the LSTM to selectively read from, write to, or reset the memory vector. The cell architecture for a single repeating LSTM module is shown in Figure 2.4.

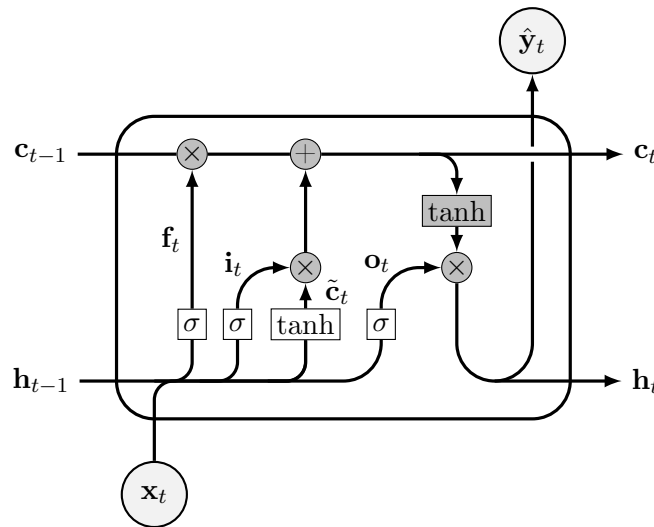


Figure 2.4: Long short-term memory (LSTM) cell architecture. Each arrow represents the flow of vector operations. Dark grey nodes represent a pointwise operation such as vector addition and white nodes represent a learned neural network layer. Merging lines represent vectors being concatenated and a diverging lines represent the vectors being duplicated.

The repeating LSTM module features four interacting layers. The fundamental component of LSTM networks is the cell state vector \mathbf{c}_t , which is updated as represented by the horizontal

line traversing the top of the diagram in Figure 2.4. Information flows through the cell state largely uninterrupted along the entire chain of the network, with only minor linear interactions. LSTMs include *gates* that allow the LSTM to optionally update the cell state by allowing extra information through. Gates are comprised of a single sigmoid neural network layer followed by pointwise multiplication. The *forget gate* is the initial gate in the LSTM and evaluates the previous hidden state \mathbf{h}_{t-1} and the current input \mathbf{x}_t , producing an output vector, \mathbf{f}_t , with values in the range $[0, 1]$, with the size of each element in the vector describing the extent to which elements should be retained. This vector is then multiplied pointwise with the previous cell state \mathbf{c}_{t-1} . Subsequently, the *update gate* determines what new information from the input vector \mathbf{x}_t must be stored in the cell state \mathbf{c}_t . First, another sigmoid neural network layer, called the *input gate*, determines which values to update and places the result in the input gate activation vector \mathbf{i}_t . Following this, a tanh layer generates a vector, $\tilde{\mathbf{c}}_t$ of candidate values for these updates. These values are scaled in a pointwise multiplication with \mathbf{i}_t , which describes the amount to update each element of the cell state. The updates are then applied to the running cell state in the subsequent pointwise addition to produce the new cell state \mathbf{c}_t . The forget and update gates allow the LSTM to manage long-term dependencies effectively, as outdated or irrelevant information is dropped and replaced by new information. Finally, the output of the cell $\hat{\mathbf{y}}_t$ is calculated. Another sigmoid layer, called the *output gate* with the activation vector \mathbf{o}_t , determines which values from the new cell state \mathbf{c}_t are to be included in the output. The new cell state vector \mathbf{c}_t is passed through a tanh activation function so that the values lie in the range $[-1, 1]$. Finally, the result is pointwise multiplied with the output gate vector \mathbf{o}_t , ensuring that only the selected parts of the cell state are included in the final output. Formally, this gives:

$$\begin{aligned}\mathbf{i}_t &= \sigma(\mathbf{W}_i[\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_i) \\ \mathbf{f}_t &= \sigma(\mathbf{W}_f[\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_f) \\ \mathbf{o}_t &= \sigma(\mathbf{W}_o[\mathbf{x}_t, \mathbf{h}_{t-1}] + \mathbf{b}_o) \\ \tilde{\mathbf{c}}_t &= \tanh(\mathbf{W}_c[\mathbf{x}_t, \mathbf{r}_t \odot \mathbf{h}_{t-1}] + \mathbf{b}_c) \\ \mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \\ \hat{\mathbf{y}}_t &= \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t)\end{aligned}$$

where \odot denotes pointwise multiplication and $\mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o, \mathbf{W}_c$ (where $\mathbf{W}_i = [\mathbf{W}_{xi}, \mathbf{W}_{hi}]$), and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o, \mathbf{b}_c$ are the individual weight matrices and biases for the linear layer in each gate. Remember that \mathbf{x}_t and \mathbf{h}_{t-1} are concatenated into a single vector. Again, let D be the dimension of \mathbf{x}_t , H be the dimension of \mathbf{h}_t and O be the dimension of $\hat{\mathbf{y}}$. Each weight matrix has dimension $H \times (D + H)$. Notice that the vector $\tilde{\mathbf{c}}_t$ is part of an additive interaction that modifies the cell state \mathbf{c}_t . Therefore, during backpropagation the gradient is distributed and so the gradient flow back to the previous cell state is:

$$\bar{\mathbf{c}}_{t-1} = \bar{\mathbf{c}}_t \odot \mathbf{f}_t$$

where $\bar{\mathbf{c}}_t = \frac{\partial L}{\partial \mathbf{c}_t}$ is the adjoint. This allows for the gradients for the cell state to flow backwards

through time relatively uninterrupted (see Figure 2.5), at least until a pointwise multiplicative interaction with an active forget gate \mathbf{f}_t is required. This is precisely what allows the LSTM to handle long term time dependencies and avoid the problem of vanishing gradients. Furthermore, the forget gate is the output of a sigmoid σ , therefore, \mathbf{f}_t is mapped to within the range of $[0, 1]$, removing the possibility of exploding gradients.

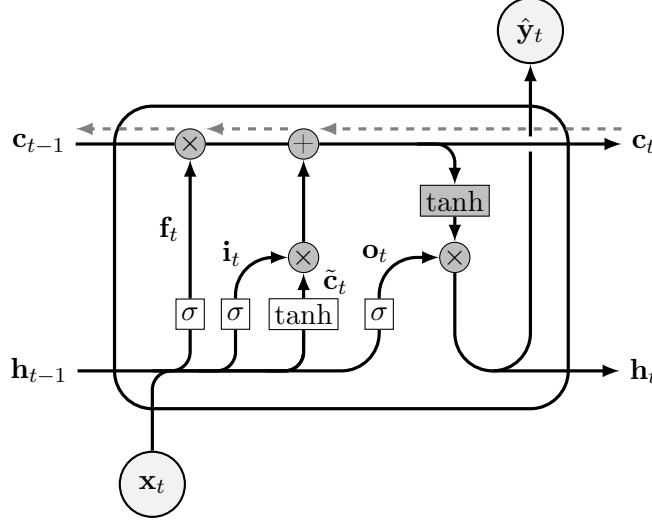


Figure 2.5: Uninterrupted gradient flow between the current cell state \mathbf{c}_t and the previous cell state \mathbf{c}_{t-1} during backpropagation. Gradient flow is shown by the dashed arrows.

There are numerous other variations of the RNN designed to improve on the LSTM. One prominent architecture is the *gated recurrent unit* (GRU) [10] which combines the forget and input gates into a single *update gate*. Furthermore, it merges the cell state into the hidden state, alongside a few other changes. The GRU is popular due to its reduced computational complexity, however, the GRU and other variations of the LSTM generally perform similarly [12, 19, 25]. The equation describing the GRU is given below:

$$\begin{aligned} \mathbf{z}_t &= \sigma(\mathbf{W}_z[\mathbf{x}_t, \mathbf{h}_{t-1}]) \\ \mathbf{r}_t &= \sigma(\mathbf{W}_r[\mathbf{x}_t, \mathbf{h}_{t-1}]) \\ \tilde{\mathbf{h}}_t &= \tanh(\mathbf{W}_h[\mathbf{x}_t, \mathbf{r}_t \odot \mathbf{h}_{t-1}]) \\ \hat{\mathbf{y}}_t &= \mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t \end{aligned}$$

where \odot denotes pointwise multiplication and $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}_h$ (where $\mathbf{W}_i = [\mathbf{W}_{xi}, \mathbf{W}_{hi}]$), and $\mathbf{b}_z, \mathbf{b}_r, \mathbf{b}_h$ are the individual weight matrices and biases for the linear layer in each gate. Remember that \mathbf{x}_t and \mathbf{h}_{t-1} are concatenated into a single vector. Again, let D be the dimension of \mathbf{x}_t , H be the dimension of \mathbf{h}_t and O be the dimension of $\hat{\mathbf{y}}$. The dimensions of $\mathbf{W}_z, \mathbf{W}_r, \mathbf{W}_h$ are all $H \times (D + H)$. Within a GRU information flows through the hidden state \mathbf{h}_t via the update gate and the *reset gate*. The update gate determines how much of the previous hidden state \mathbf{h}_{t-1}

carries through to the current hidden state \mathbf{h}_t . This gate combines both the previous hidden state \mathbf{h}_{t-1} and the current input \mathbf{x}_t to produce an update vector \mathbf{z}_t with values in the range $[0, 1]$, with the size of each element in the vector describing the extent to which elements in the hidden state vector should be updated versus retaining the existing information. The reset gate also combines both the previous hidden state \mathbf{h}_{t-1} and the current input \mathbf{x}_t to produce the reset vector \mathbf{r}_t with values in the range $[0, 1]$, which controls the blending of the previous hidden state with the new input. Next, the *candidate hidden state* $\tilde{\mathbf{h}}_t$ is computed, which incorporates the reset gate to selectively forget parts of the previous hidden state via elementwise multiplication. This candidate hidden state is the potential new hidden state based on the new input and the selectively retained information from the past. Finally, the new hidden state \mathbf{h}_t is computed by combining the previous hidden state and the candidate hidden state, weighted by the update gate \mathbf{z}_t . This ensures that only the relevant information is updated, and the rest is retained.

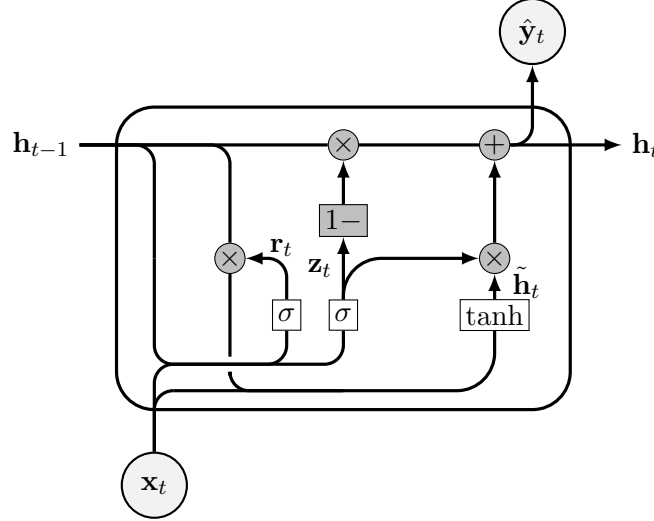


Figure 2.6: Gated Recurrent Unit (GRU) cell architecture. Each arrow represents the flow of vector operations. Dark grey nodes represent a pointwise operation such as vector addition and white nodes represent a learned neural network layer. Merging lines represent vectors being concatenated and a diverging lines represent the vectors being duplicated.

2.7 RNN Encoder-Decoder

Sutskever et al. [51] and Cho et al. [10] introduced the RNN encoder-decoder model, with an encoder RNN, that maps an input sequence of size T to a fixed, high dimensional vector embedding called the *context vector*, and then a decoder RNN to generate an output sequence of size T' (where T' may differ from T) conditioned on the context vector. The decoder RNN functions as a language model, which is trained to generate the output sequence in an autoregressive manor, using the previous output $\hat{\mathbf{y}}_{t-1}$ and the hidden state \mathbf{h}_t to predict the next token $\hat{\mathbf{y}}_t$, except

that it is also conditioned on the context vector \mathbf{c} of the input sequence. Sutskever et al. [51] utilised the LSTM while Cho et al. [10] utilised the GRU, as the encoder and decoder RNN architectures respectively. These specific RNN architectures, with their ability to handle long range time dependencies, are particularly useful for sequence to sequence modelling due to the decoder likely needing to refer back to the vector embedding of the input sentence for information when generating the next token. The context vector acts as a compressed representation of the input sequence and is the only means by which input from the input sequence is passed to the output sequence.

Formally, the model estimates the conditional probability distribution $p(\mathbf{y}|\mathbf{x})$ where $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ is the input sequence or context, and $\mathbf{y} = (\mathbf{y}_1, \dots, \mathbf{y}_{T'})$ is the corresponding output sequence and where the output sequence length T' may differ in value from the input sequence length T . This conditional probability is computed using the encoder to obtain the fixed dimensional context vector $\mathbf{c} = \mathbf{h}_T$, which is set to be the last hidden state in the encoder RNN. The decoder RNN then computes the probability of \mathbf{y} using the context vector as its initial hidden state $\mathbf{h}'_0 = \mathbf{c}$. That is, the decoder acts similarly to an RNN sequence model and defines a probability distribution over the output sequence $\mathbf{y} = (\mathbf{y}_1, \dots, \mathbf{y}_{T'})$ by breaking down the joint probability into conditional probabilities:

$$p(\mathbf{y}|\mathbf{x}) = \prod_{t=1}^{T'} p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{c})$$

Similar to the previous sequence models, softmax is used to represent the probability distribution $p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{c})$ over the entire set \mathcal{V} of possible tokens. Each sequence must end with a special end of sequence token $\langle s \rangle$. This enables the model to define a probability distribution over sequences of all possible lengths as if the end of sequence symbol is encountered then the model knows to stop generating new tokens. The encoder and the decoder are jointly trained. As discussed in previous sections, training involves minimising the negative log likelihood $-p(\mathbf{y}|\mathbf{x})$ (that is, maximising the probability of generating a correct output sequence) and is the summation of the cross-entropy losses at each time step, normalised by the total number of time steps T' :

$$L = -\frac{1}{T'} \sum_{t=1}^{T'} \log p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{c}) = -\frac{1}{T'} \sum_{t=1}^{T'} \sum_{i=1}^V y_{t,i} \log \hat{y}_{t,i} = -\frac{1}{T'} \sum_{t=1}^{T'} \log \hat{y}_{t, \mathbb{I}(y_{t,i}=1)}$$

where $\hat{y}_{t, \mathbb{I}(y_{t,i}=1)}$ is the predicted probability for the true token y_i at the t -th time step. Using this learned distribution, it is possible to generate a new sequence, conditioned on some context \mathbf{x} , by iteratively sampling a new symbol from the predicted probability distribution at each time step until the end of sequence symbol is encountered, or a predefined generation limit is reached.

The RNN encoder-decoder models from Sutskever et al. [51] and Cho et al. [10] differ in their use of the context vector. The former model simply conditions the decoder RNN on context

vector with $\mathbf{c} = \mathbf{h}'_0$ (see Figure 2.7). The latter model instead conditions each hidden state in the decoder RNN with the context vector. That is, the hidden state of the decoder is given by $\mathbf{h}'_t = f(\mathbf{h}'_{t-1}, \mathbf{y}_{t-1}, \mathbf{c})$. The next token is also conditioned on the context vector, giving the new conditional probability: $p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{c}) = g(\mathbf{h}'_t, \mathbf{y}_{t-1}, \mathbf{c})$ (see Figure 2.8). For given functions f and g , which in the context of Cho et al. [10] describe the GRU.

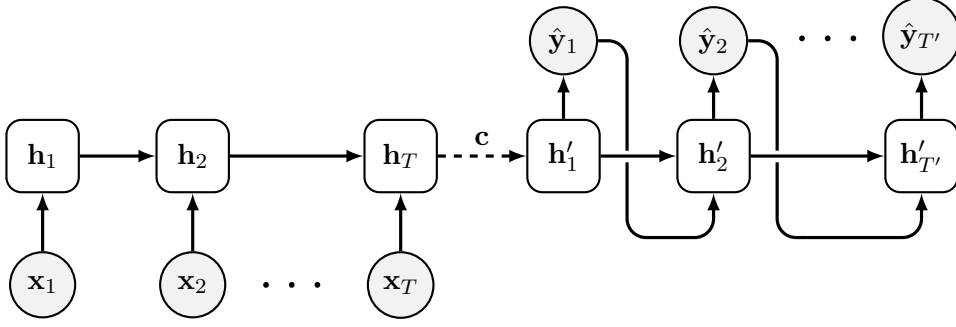


Figure 2.7: RNN encoder-decoder architecture from Sutskever et al. [51]. The encoder (left) computes the context vector $\mathbf{c} = \mathbf{h}_T$ of $(\mathbf{x}_1, \dots, \mathbf{x}_T)$ and the decoder (right) uses this context vector representation to compute the probability of $\mathbf{y}, \dots, \mathbf{y}_{T'}$ conditioned on the context vector, where \mathbf{x}_T and $\mathbf{y}_{T'}$ are the embedding of the end of sequence token $\langle s \rangle$.

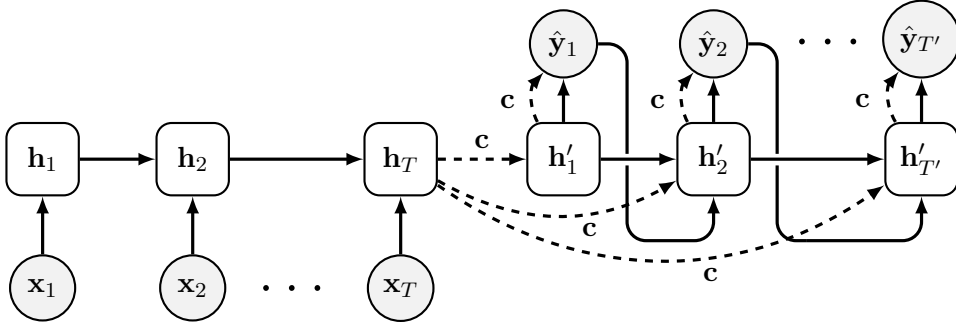


Figure 2.8: RNN encoder-decoder architecture from Cho et al. [10] similar to Sutskever et al. [51] except that each individual hidden state \mathbf{h}'_t and next token \mathbf{y}_t are conditioned on the context vector.

Sutskever et al. [51] reversed the order of the input sequence. This ensures that symbols early in the input sequence can attend closely to the symbols early in the output and vice versa. This way the optimiser easily finds relationships between the input and output sequences. Nevertheless, the performance of RNN based encoder-decoder models is inherently constrained by the limitations of the context vector, which rapidly becomes a bottleneck for ever increasing sequence lengths. This is due to the fixed-length context vector not having enough capacity to encode longer sequences and store all relevant information [2, 9].

2.8 Attention-based RNN Encoder-Decoder

Bahdanau et al. [2] introduced a soft-search mechanism, allowing an encoder-decoder model to focus on different parts of the input sequence where the most relevant information is concentrated, when generating the output sequence. The model then predicts the next token using multiple context vectors associated with these input positions and the previously generated tokens. This soft-search mechanism significantly improved modelling performance over the basic encoder-decoder model. The new architecture consisted of a bidirectional recurrent neural network (BRNN) [46] as an encoder, and a decoder that soft-searches through the source sentence when decoding a sentence. The BRNN consists of two RNNs which process the sequence in both forward and backward directions. This allows the network to have access to both past and future context for a given time step in the input sequence. Mathematically, for an input sequence $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ the forward RNN \vec{f} processes the sequence from $t = 1$ to $t = T$ to produce a sequence of hidden states $\vec{\mathbf{h}}_1, \dots, \vec{\mathbf{h}}_T$ and the backwards RNN \overleftarrow{f} processes the sequence in reverse order from $t = T$ to $t = 1$ to produce another sequence of hidden states $\overleftarrow{\mathbf{h}}_T, \dots, \overleftarrow{\mathbf{h}}_1$. Then, the hidden state at each time step t in the BRNN is the concatenation of the forward and backward hidden states $\mathbf{h}_t = [\vec{\mathbf{h}}_t, \overleftarrow{\mathbf{h}}_t]$. Therefore, each hidden state \mathbf{h}_j in the decoder contains information on both the preceding and following tokens centred around the j -th token in the input sequence. The context vector is generated through some (possibly non-linear) function q of the hidden states in the encoder: $\mathbf{c} = q(\{\mathbf{h}_1, \dots, \mathbf{h}_T\})$. For example, Sutskever et al. [51] used $q(\{\mathbf{h}_1, \dots, \mathbf{h}_T\}) = \mathbf{h}_T$. As mentioned in the previous section, each conditional probability within the decoder is generally modelled as:

$$p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{c}) = g(\mathbf{h}'_t, \mathbf{y}_{t-1}, \mathbf{c})$$

where g is a nonlinear, possibly multi-layered, function which yields the probability of \mathbf{y}_t , \mathbf{h}'_t is the hidden state of the decoder, \mathbf{h}_j is the hidden state of the encoder, and \mathbf{c} is the context vector received from the encoder. In the new model architecture introduced by Bahdanau et al. [2], the conditional probability is instead modelled as:

$$p(\mathbf{y}_t | \{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\}, \mathbf{x}) = g(\mathbf{h}'_t, \mathbf{y}_{t-1}, \mathbf{c}_t)$$

where $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ is the input sequence and \mathbf{h}'_t is the decoder RNNs hidden state at time t , which is computed by

$$\mathbf{h}'_t = f(\mathbf{h}'_{t-1}, \mathbf{y}_{t-1}, \mathbf{c}_t)$$

Notice that the probability for each output token \mathbf{y}_t is now conditioned on a distinct context vector \mathbf{c}_t , as opposed to a general context vector \mathbf{c} previously. The context vector is \mathbf{c}_t is composed of a sequence of *annotations* $(\mathbf{h}_1, \dots, \mathbf{h}_T)$ which the encoder then maps to the input sequence. Every individual annotation \mathbf{h}_j contains information on the entire input sequence and is particularly focused on the parts of the input surrounding the j -th token. The context vector

at each time step is then computed as the weighted sum of the annotations where the weight α_{tj} of each annotation \mathbf{h}_j is computed via softmax:

$$\mathbf{c}_t = \sum_{j=1}^T \alpha_{tj} \mathbf{h}_j, \quad \alpha_{tj} = \frac{\exp(e_{tj})}{\sum_{k=1}^T \exp(e_{tk})}$$

and the function $a(\mathbf{h}'_{t-1}, \mathbf{h}_j) = e_{tj}$ is an *alignment model* which assigns a score to each input and output pair (t, j) , describing the comparability of the inputs around position j and the output at position t match. This score is based on the decoders hidden state \mathbf{h}'_{t-1} , just before outputting \mathbf{y}_t , and the j -th annotation \mathbf{h}_j of the input sequence. The alignment model a is parameterised as a feedforward neural network and jointly trained with the rest of the model.

This context vector, or weighted sum over all the annotations α_{tj} , can be thought of as an *expected annotation*, where the expectation is over all the possible alignments. Let α_{tj} be the probability that the output token \mathbf{y}_t is aligned (i.e. connected to) an input token \mathbf{x}_j . Then, the t -th context vector \mathbf{c}_t is the expected annotation computed over all the annotations $(\mathbf{h}_1, \dots, \mathbf{h}_T)$ with individual probabilities α_{tj} . The probability α_{tj} reflects the importance of the annotation \mathbf{h}_j , with respect to the previous hidden state \mathbf{h}'_{t-1} of the decoder, in deciding the next state \mathbf{h}'_t and therefore generating \mathbf{y}_t . This mechanism is called *attention*, as the decoder maintains which parts of the input sequence to attend to. With the decoder having an attention mechanism, the encoder does not have to encode all information in the source sequence into a single fixed size context vector. Information can now be distributed throughout the sequence of annotations, allowing the decoder to selectively retrieve specific information from the encoder as needed.

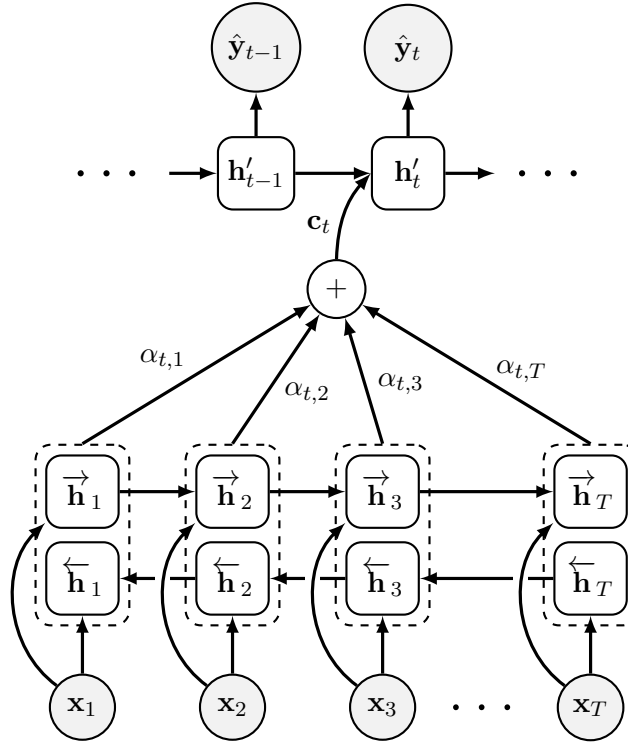


Figure 2.9: Bidirectional RNN encoder-decoder architecture with an attention mechanism. The bidirectional RNN encoder processes the sequence in two directions. The calculated annotations α_{tj} from each hidden state in the bidirectional RNN \mathbf{h}_j are sent into a weighted sum to obtain the context vector \mathbf{c}_t for the current hidden state in the decoder \mathbf{h}'_t . Therefore $\hat{\mathbf{y}}_t$ is generated by attending to each input $\{\mathbf{x}_1, \dots, \mathbf{x}_T\}$.

Chapter 3

Transformer Architecture

The highly influential paper ‘*Attention is All You Need*’ by Vaswani et al. [53] highlighted that attention mechanisms such as the soft-search attention mechanism developed by Bahdanau et al. [2] are themselves sufficient for sequence modelling within encoder-decoder architectures, removing the use of recurrent networks entirely. This insight led to the development of the new Transformer architecture based solely on attention mechanisms to create global dependencies between the input and output sequences. Specifically, a new form of attention mechanism called *multi-head self-attention* is leveraged, which allows the model to focus on multiple different parts of the input sequence when generating the output sequence. These multiple heads of attention can be computed in parallel, significantly improving computational efficiency over the inherently sequential structure of recurrent encoder-decoder architectures, specifically for longer sequence lengths as memory constraints, caused by sequential computation, limit the maximum potential batch size when training.

3.1 Encoder-Decoder Architecture

The Transformer has an encoder-decoder architecture and intersperses multi-head attention and feedforward layers within a residual network (ResNet) structure, with additional layer normalisation to stabilise training. The encoder maps an input sequence of vector embeddings $(\mathbf{x}_1, \dots, \mathbf{x}_T)$ to a sequence of continuous representations $\mathbf{z} = (\mathbf{z}_1, \dots, \mathbf{z}_T)$. The decoder then generates an output sequence $(\mathbf{y}_1, \dots, \mathbf{y}_{T'})$ of tokens in an autoregressive manner, conditioned on the sequence of continuous representations \mathbf{z} . The transformer architecture shown in Figure 3.1. The encoder is comprised of a stack of N identical layers, with each layer containing two sub-layers. The first sub-layer performs multi-head self-attention over the input to the encoder. The second sub-layer is a position-wise fully connected feed-forward network.

The decoder is also composed of a stack of N identical layers. Each layer in the decoder has three sub-layers. The first sub-layer performs masked multi-head self-attention over the

decoder's input (that is, the previous decoder outputs). The second sub-layer performs multi-head attention over the output $\mathbf{z} = (\mathbf{z}_1, \dots, \mathbf{z}_T)$ of the encoder. The third sub-layer is a position-wise fully connected feed-forward network. The masked self-attention sub-layer in the decoder is modified with an additional autoregressive mask which only allows the decoder to attend to the positions before t . This masking and the shifting of outputs such that they are offset by one position ensures that predictions $\hat{\mathbf{y}}_t$ can only depend on the previous outputs $(\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_{t-1})$. Each sub-layer in the encoder and decoder has a residual connection which is added to the output of the sub-layer, and is followed by layer normalisation. Finally, a learned linear transformation with a softmax function converts the output of the decoder into a predicted probability distribution for the current symbol. Furthermore, the input embeddings and output embeddings share an embedding matrix with the pre-softmax linear transformation [40] to reduce the parameter count and improve efficiency.

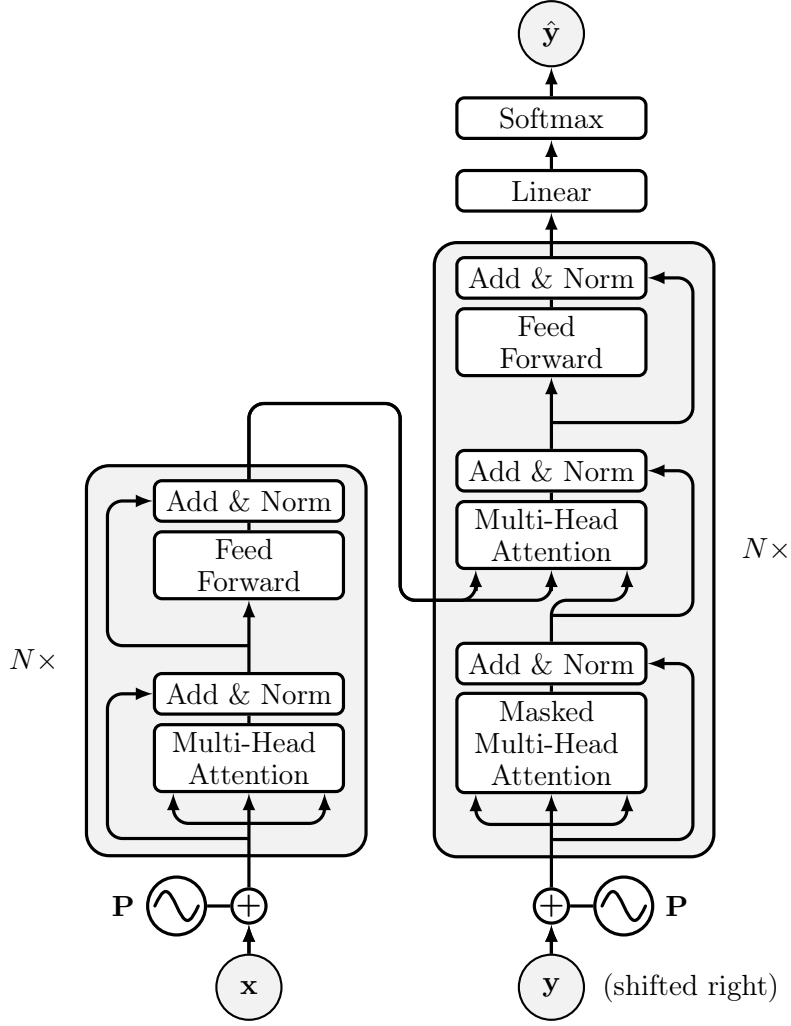


Figure 3.1: Transformer architecture. The encoder (left) and decoder (right) both consist of N identical layers. The encoder layers include multi-head attention and feed-forward sub-layers, while the decoder layers add a masked multi-head attention sub-layer.

The position-wise feedforward network consists of two linear transformations with the nonlinear ReLU activation function in between: $f(\mathbf{x}) = \mathbf{W}_2(\text{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)) + \mathbf{b}_2$. This feedforward network is applied independently to each position in the output from the previous sub-layer. This combination allows the model to first aggregate contextual information through attention and then apply a non-linear transformation independently for each symbol in the sequence. This is equivalent to a convolution operation where a kernel of size one slides over each position in the sequence and applies the transformation. Let C be the embedding dimensionality, and let the hidden layer has a dimensionality of d_h . Therefore, $\mathbf{W}_1 \in \mathbb{R}^{d_h \times C}$, $\mathbf{b}_1 \in \mathbb{R}^{d_h}$, $\mathbf{W}_2 \in \mathbb{R}^{C \times d_h}$, $\mathbf{b}_2 \in \mathbb{R}^C$. In subsequent research, the ReLU activation function has been replaced with the GELU (Gaussian Error Linear Unit) [21] activation function due to its smoothness properties aiding with

gradient flow during backpropagation, leading to better convergence properties.

The Transformer architecture has remained relatively unchanged since its introduction with the only consistent change being the moving of layer normalisation from after the sub-layers, to before them [57]. Some models such as the decoder-only GPT-2 [41] architecture add an additional layer normalisation after the final self-attention block.

3.2 Positional Encodings

The Transformer architecture does not include any residual or convolutional layers and so contains no inherent positional information about sequence order. Therefore, positional encoding is required to provide additional information about the position of each token in the input sequence. Each positional encoding vector has the same dimensionality C as the token embedding vectors, allowing them to be summed piecewise. Positional encoding can either be learned through embedding matrices or can be fixed using specific functions, with both having similar performance.

Let $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ be a sequence of vector embeddings of length T . The following equations, introduced by Vaswani et al. [53], calculate the positional encoding of the t -th token embedding $\mathbf{x}_t \in \mathbb{R}^C$ within this sequence using sine and cosine functions of varying frequencies:

$$P(t, 2i) = \sin\left(t \cdot n^{-2i/C}\right) \quad P(t, 2i + 1) = \cos\left(t \cdot n^{-2i/C}\right)$$

where $P(t, j)$ maps the j -th element at position t in the input sequence to the (t, j) -th element in the positional encoding matrix. That is, the positional encoding vector for the t -th position in the input sequence includes even elements mapped from the sine function $P(t, 2i)$ and odd elements mapped from the cosine function $P(t, 2i + 1)$, for all i where $0 \leq i \leq C/2$. The wavelengths $\lambda_i = 2\pi n^{2i/C}$ of the sinusoids form a geometric progression and vary from 2π to $2\pi n$ and the hyperparameter n scales the frequency of the sine and cosine functions over the entire size C of the embedding dimension. The periodic nature of the sine and cosine functions enable the model to easily learn to attend by relative positions. Specifically, for any fixed offset k , the positional encoding $P(t + k)$ can be represented as a linear function of $P(t)$.

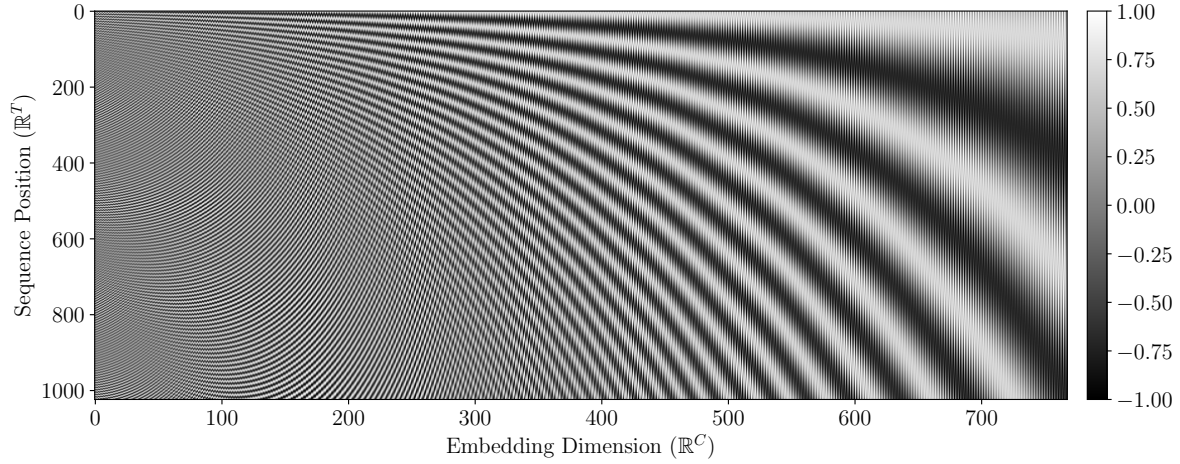


Figure 3.2: Positional encoding matrix $\mathbf{P} \in \mathbb{R}^{T \times C}$ produced with the sine and cosine functions of varying frequencies introduced by Vaswani et al. [53] for sequence lengths of $T = 1024$ and an embedding dimensionality of $C = 768$. The scaling parameter was set to $n = 1e2$ to scale the functions over the entire embedding dimension.

Alternatively, let $\mathbf{P} \in \mathbb{R}^{T \times C}$ be a learned positional encoding (i.e. positional embedding) where T is the sequence length and C is the vector embedding dimensionality. Each row, \mathbf{p}_t represents the positional encoding vector for the position t in the sequence. Therefore, with the embedding matrix $\mathbf{C} \in \mathbb{R}^{V \times C}$, the positional encodings can be added to the vector embeddings $\tilde{\mathbf{x}}_t = \mathbf{C}[x_t] + \mathbf{P}[x_t]$ which can then be used as input into the transformer.

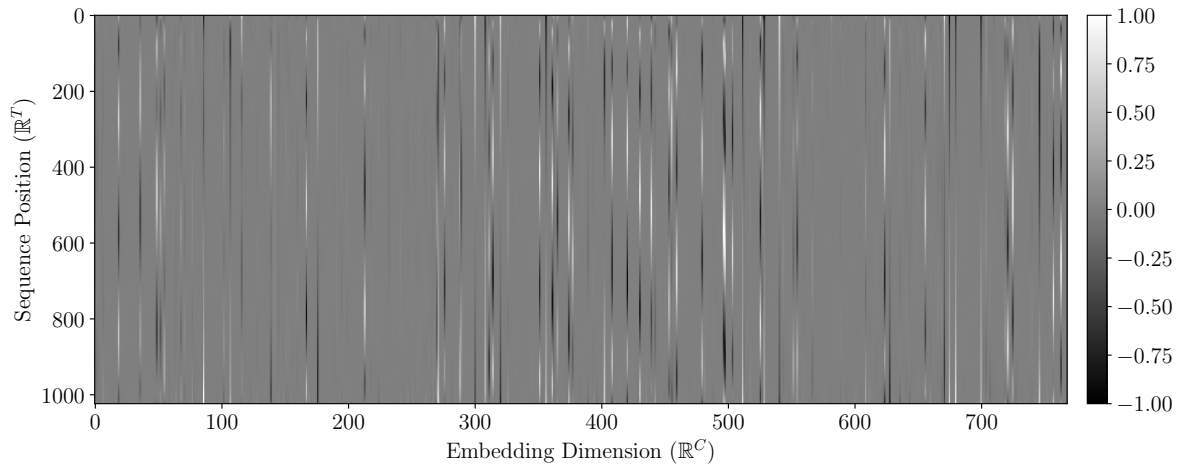


Figure 3.3: Positional embedding matrix $\mathbf{P} \in \mathbb{R}^{T \times C}$ from GPT-2 124M which features sequence lengths of $T = 1024$ and an embedding dimensionality of $C = 768$. Values have been clipped to a range of $[-1, 1]$ so that structure with the embedding matrix is more easily shown. Dimensions in the embedding space show structure used which differentiate positions within the input sequence.

3.3 Scaled Dot-Product Attention and Multi-Head Attention

An attention mechanism compares a query vector to a set of key vectors. Specifically, an attention mechanism maps a query vector and a set of key-value vector pairs to an output vector. The output is computed as a weighted sum of the value vectors, where the weight for each value is determined by a compatibility function between the query vector and its corresponding key vector. There are two main attention functions. The first is additive attention, the initial type of attention introduced by Bahdanau et al. [2], which computes the compatibility function using an alignment model, parameterised as a feedforward neural network with a single hidden layer. The second type of attention is dot-product attention which computes the compatibility function via the dot product between the query and key vectors. While both types of attention are similar, dot-product attention is computationally more efficient as it can be implemented using highly optimised matrix multiplication operations.

Scaled dot-product attention is a modified type of dot-product attention used within the Transformer. Let $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ and $\mathbf{v} \in \mathbb{R}^{d_v}$ be the queries, keys and values used as input, where d_k is the dimensionality of the queries and keys and d_v is the dimensionality of the values. The query, key, and value vectors are computed as: $\mathbf{q} = \mathbf{W}_q \mathbf{x}$, $\mathbf{k} = \mathbf{W}_k \mathbf{x}$, $\mathbf{v} = \mathbf{W}_v \mathbf{x}$ where $\mathbf{x} \in \mathbb{R}^C$ is the input vector and $\mathbf{W}_q, \mathbf{W}_k \in \mathbb{R}^{d_k \times C}$ and $\mathbf{W}_v \in \mathbb{R}^{d_v \times C}$ are learned weight matrices. First, the attention scores or *compatibilities* are computed by taking the dot product $\mathbf{q}^\top \mathbf{k}$ between the query and key vectors. Before softmax is applied to yield the attention weights, the scores are then scaled by $1/\sqrt{d_k}$ to address the issue of large dot products when the dimensionality d_k of the query and key vectors is large. Specifically, if the components of \mathbf{q} and \mathbf{k} are independent random variables with zero mean and variance σ^2 , then the dot product $\mathbf{q}^\top \mathbf{k}$ has zero mean and variance $d_k \sigma^2$. This large variance can lead to extremely large values after applying the softmax function, and therefore, extremely small gradients during backpropagation and potentially ineffective learning. Henceforth, scaling by $1/\sqrt{d_k}$ normalises the variance of the dot product as the scaled dot product will have zero mean and variance σ^2 , independent of the dimensionality of d_k . As mentioned previously, softmax is applied after scaling to yield the attention weights. These attention weights are then used to compute a weighted sum of the value vector to yield scaled dot-product attention. The scaled dot-product attention function can be parallelised to calculate a set of scaled dot-product attention simultaneously. Define the input matrix $\mathbf{X} \in \mathbb{R}^{T \times C}$ where T is the sequence length and C is the embedding dimensionality which is also the dimensionality of the output of each sub-layer in the transformer. The query $\mathbf{Q} \in \mathbb{R}^{T \times d_k}$, key $\mathbf{K} \in \mathbb{R}^{T \times d_k}$ and value $\mathbf{V} \in \mathbb{R}^{T \times d_v}$ matrices are computed using the projections:

$$\mathbf{Q} = \mathbf{XW}_q \quad \mathbf{K} = \mathbf{XW}_k \quad \mathbf{V} = \mathbf{XW}_v$$

where the learned weight matrices are $\mathbf{W}_q, \mathbf{W}_k \in \mathbb{R}^{C \times d_k}$ and $\mathbf{W}_v \in \mathbb{R}^{C \times d_v}$ and the parallelised

attention function is then calculated as follows:

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}}\right) \mathbf{V}$$

Multi-head attention allows the transformer to jointly attend to information from different representation subspaces over multiple different positions. Each individual head can learn different dependencies, which are then combined via concatenation and a linear transformation to form a deep understanding of the information. In multi-head attention, the dimensions are split among multiple heads. For H heads, each head uses a lower-dimensional representation with $d_k = d_v = C/H$. As each head has a reduced dimensionality, the computational cost of multi-head attention is similar to that of single-head attention with full dimensionality. Specifically, multi-head attention allows the Transformer to linearly project the queries $\mathbf{Q}_i \in \mathbb{R}^{T \times C/H}$, keys $\mathbf{K}_i \in \mathbb{R}^{T \times C/H}$ and values $\mathbf{V}_i \in \mathbb{R}^{T \times C/H}$ a total of H times with different, learned linear projections. Then, scaled dot-product attention is performed in parallel across the H heads with the different linear projections of the queries, keys and values. The attention vectors calculated across each head are then concatenated and linearly projected via the learned linear projection matrix $\mathbf{W}_o \in \mathbb{R}^{C \times C}$ to yield the multi-head attention:

$$\mathbf{Q}_i = \mathbf{Q}\mathbf{W}_{q,i} \quad \mathbf{K}_i = \mathbf{K}\mathbf{W}_{k,i} \quad \mathbf{V}_i = \mathbf{V}\mathbf{W}_{v,i}$$

$$\text{MultiHead}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = ([\mathbf{h}_1, \dots, \mathbf{h}_H] \mathbf{W}_o) \in \mathbb{R}^{T \times C} \quad \mathbf{h}_i = \text{Attention}(\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i)$$

where $\mathbf{X} \in \mathbb{R}^{T \times C}$ is the input matrix, $[\mathbf{h}_1, \dots, \mathbf{h}_H]$ denotes the concatenation of the scaled dot-product attention values from all H heads, where $\mathbf{h}_i \in \mathbb{R}^{T \times C/H}$, and $\mathbf{W}_{q,i}, \mathbf{W}_{k,i}, \mathbf{W}_{v,i} \in \mathbb{R}^{C \times C/H}$ are the learned projection matrices for the i -th head. The initial query $\mathbf{Q} \in \mathbb{R}^{T \times C}$, key $\mathbf{K} \in \mathbb{R}^{T \times C}$ and value $\mathbf{V} \in \mathbb{R}^{T \times C}$ matrices for multi-head attention are computed using the projections:

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_q \quad \mathbf{K} = \mathbf{X}\mathbf{W}_k \quad \mathbf{V} = \mathbf{X}\mathbf{W}_v$$

where the learned weight matrices are $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v \in \mathbb{R}^{C \times C}$. As the computational complexity of scaled dot-product attention in each head is $O(T^2 \cdot C/H)$, the total complexity of multi-head attention is $O(T^2 \cdot C)$. This compares with an recurrent layer which has a computational complexity of $O(C^2)$ in each recurrent cell and so has a total computational complexity of $O(T \cdot C^2)$.

Learning long-range dependencies becomes significantly easier when the forward and backward passes within the network between any pair of positions in the input and output sequences are shorter. Self-attention computes attention scores for each pair of positions in the input sequence, therefore all positions are directly connected together using a constant number of sequential operations. Therefore, the maximum path length between any two positions within a self-attention layer is $O(1)$. Furthermore, self-attention computes everything in parallel, i.e.

$O(1)$. This is different to a recurrent layer where the maximum path length and therefore the number of sequential operations could be as much as $O(T)$.

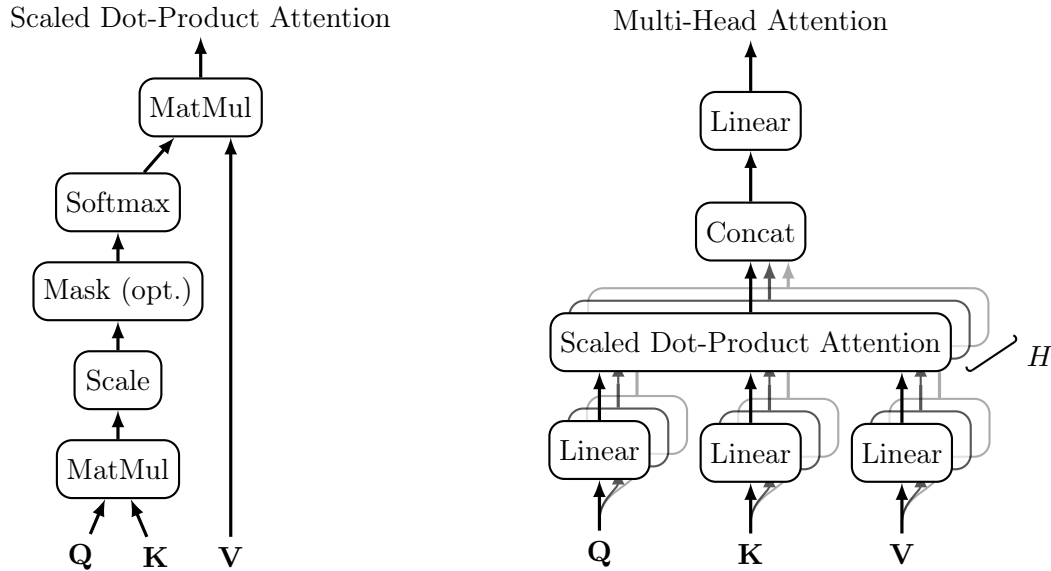


Figure 3.4: **Left:** Scaled dot-product attention. The optional mask layer is only used in the decoder so that tokens only attend to previous tokens. **Right:** Multi-head attention with H layers computed in parallel.

The Transformer architecture uses multi-head attention in three different ways. First, *cross-attention* is used in the decoder and allows every position in the decoder to attend to all the continuous representations $z = (z_1, \dots, z_T)$ output by the encoder. In cross-attention, the query comes from the previous decoder layer, while the key and value are derived from the encoder outputs. This mechanism is similar to the attention mechanism introduced by Bahdanau et al. [2]. The first sub-layer of multi-head attention within the encoder is called *self-attention* and is where the queries, keys and values all come from the previous sub-layer in the encoder. With self-attention each position in the encoder can attend to every other position in the previous sub-layer of the encoder. Similarly, *causal self-attention*, also known as masked self-attention, is identical to self-attention, except that it enforces autoregressive behaviour by preventing each position in the decoder from attending to future positions. This is implemented within scaled dot-product attention via masking.

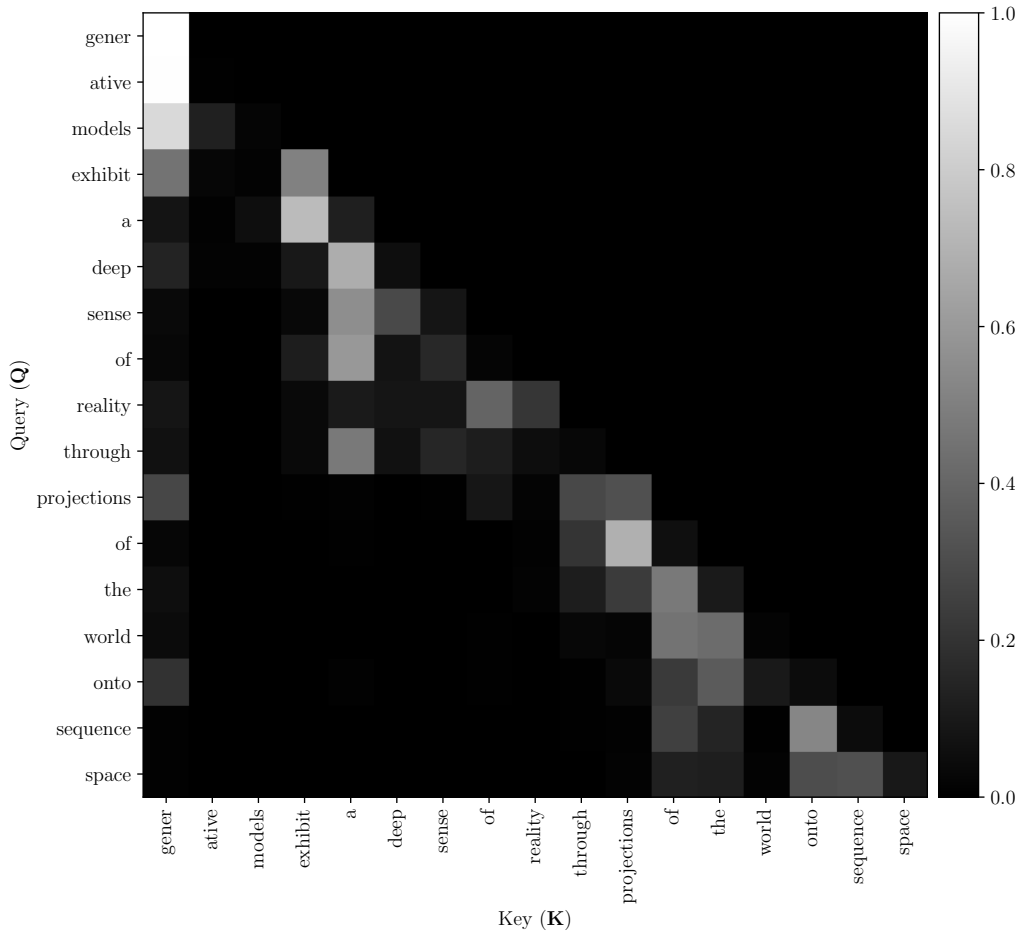


Figure 3.5: Masked self-attention matrix from the third head in the fourth layer of GPT-2 124M for the sequence of tokens ‘generative models exhibit a deep sense of reality through projections of the world onto sequence space’. The autoregressive property of causal self-attention implemented through masking creates a lower triangular matrix. Each cell represents the attention score between tokens with brighter cells indicating higher attention scores, where the query token is attending more to the corresponding key token. Moving down the rows, each subsequent token will attend to previous tokens up to itself. Softmax ensures that each row (the attention values) sum to one.

An intriguing observation was noted when examining the masked self-attention matrices of the GPT-2 language model [41] across multiple layers and heads. In the latter layers, query tokens tend to heavily attend to the first key token (see Figure 3.6). This phenomenon is generated by the positional embeddings and the attention mechanism. The first position in the sequence into the decoder is frequently given significant importance, acting as a reference point that helps the model gather initial context as to the probabilities of the subsequent tokens. Vig et al. [54] have conducted research on this phenomenon.

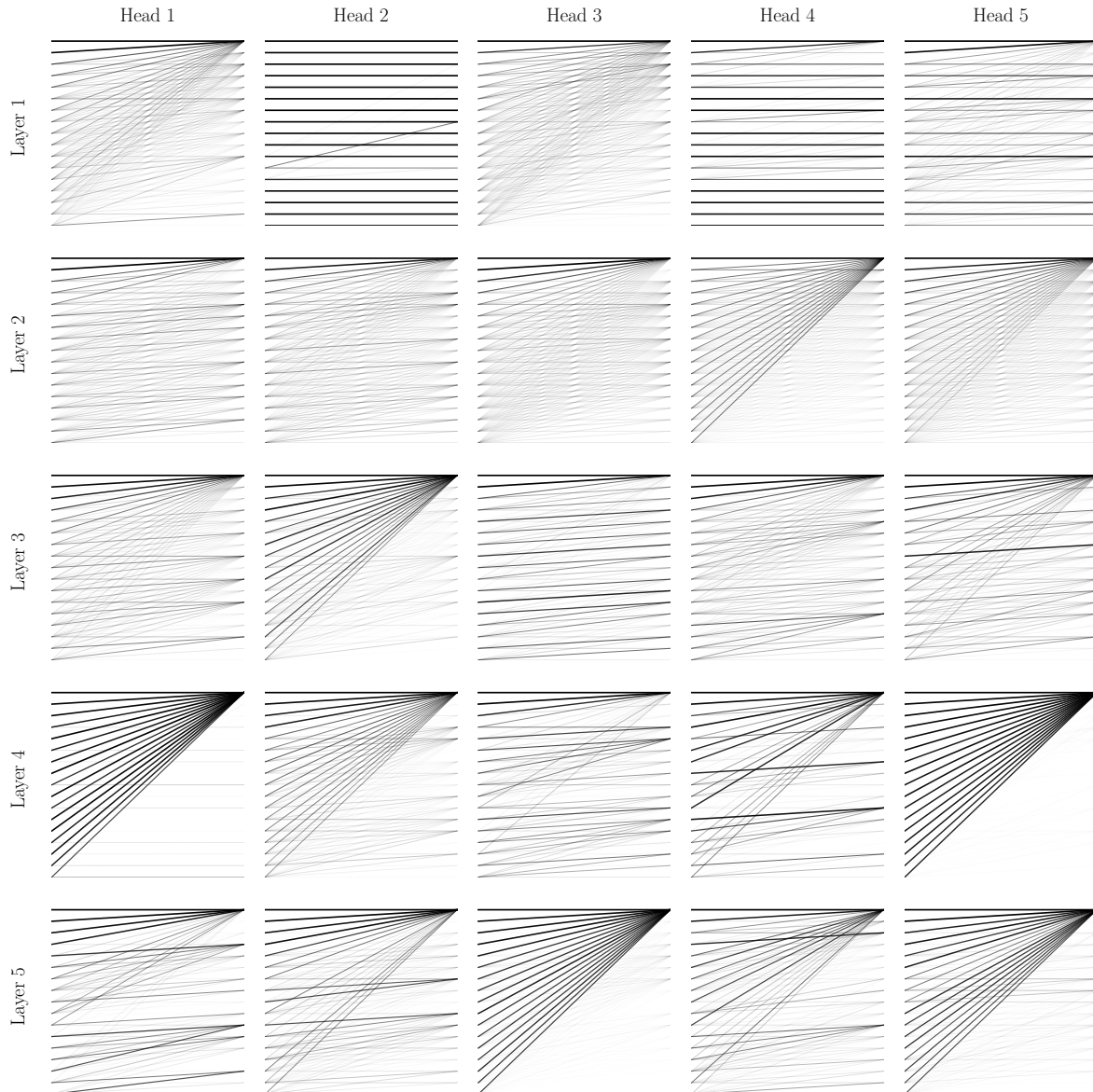


Figure 3.6: Visualisation of the weights of each query (\mathbf{Q}) - key (\mathbf{K}) pair for masked self-attention from the first five heads and the first five layers from GPT-2 124M for the sequence of tokens ‘generative models exhibit a deep sense of reality through projections of the world onto sequence space’. Each subplot represents the attention patterns of one attention head in a particular layer. Lines connect tokens on the left (queries) to tokens on the right (keys), with line thickness indicating the attention strength. Throughout all the heads in each layer there is a clear pattern of attending heavily towards the first position. Note that autoregressive masking is clearly visible as no queries attend to keys that come after them (i.e. no lines exist with a negative gradient from left to right).

3.4 Tokenisation and Byte-Pair Encoding

A *token* constitutes the fundamental unit of input into a language model. *Tokenisation* refers to the process of encoding a string of text into a corresponding sequence of tokens, where each token belongs to a predefined vocabulary of possible tokens. Tokenisation is highly important to the optimal functioning of the Transformer architecture.

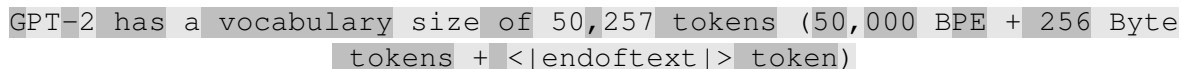


Figure 3.7: Output of the GPT-2 tokenizer for the string ‘GPT-2 has a vocabulary size of 50,257 tokens (50,000 BPE + 256 Byte tokens + <|endoftext|> token)’ showing sub-word tokenisation. Frequently occurring words have been tokenised as individual tokens. Uncommon words such as GPT-2 have been tokenised with base tokens `G`, `P`, `T`, `-` and the token `PT`. The `<|endoftext|>` token is the special token used to represent the end of a sequence.

A naive form of tokenisation is *character-level tokenisation*, whereby the vocabulary \mathcal{V} is the set of all V possible characters. This leads to a minimum vocabulary size, however, yields long token sequences (due to the bijective mapping between characters and tokens) which can be particularly problematic for smaller models with a limited context window T in which tokens can attend to each other. In an idealistic world with limitless compute, models could be created with unlimited context windows, allowing them to learn complex relationships between every pair of characters in a text. However, in reality, computational limitations constrain the context window size (especially since $O(T^2 \cdot C)$ is the computational complexity of the Transformer), making this approach impractical. As such, *sub-word tokenisation* is used to tokenise chunks of frequently occurring characters into individual tokens. With this approach, frequently occurring words are given their own token, and larger, uncommon words are split into meaningful token components. As an example, the string ‘generative’ is tokenised into `generative`. Individual byte characters are included as *base tokens* to ensure that all words are expressible, removing the possibility of being unable to tokenise words that are out of vocabulary. Any unseen words are tokenised as combinations of the token components and the initial base tokens. Using character chunks as tokens allows the model to attend to a wider portion of the text within the limited context size, which can improve model performance. Note that the increased vocabulary size does lead to a larger embedding matrix $\mathbf{C} \in \mathbb{R}^{V \times C}$, and therefore greater number of parameters to learn. Additionally, larger vocabularies reduce the frequency of individual tokens within the training data, potentially resulting in under-trained vector embeddings and worse performance. Despite the increased vocabulary size, sub-word tokenisation increases the information density within the limited context window, allowing transformers to addend to more contextual information, potentially resulting in better overall performance and improved ability to learn long-range dependencies. Determining the optimal vocabulary size V requires achieving

an effective equilibrium between information density and model complexity.

Byte-pair tokenisation [47] is an automatic and effective chunk tokenisation algorithm, popularised in the GPT-2 language model [41] and widely used in recent large language models. Byte-pair encoding (BPE) greedily merges commonly occurring sub-strings into new tokens based on their frequency and adjacency until a predetermined vocabulary size has been reached. Before the main BPE algorithm a pre-tokeniser splits the text into distinguishable words, usually using a regular expression splitting pattern. This creates an initial set of tokens that the BPE algorithm can further process. The GPT-2 regular expression splitting pattern is shown below and splits the text into words based on the occurrence of whitespace and punctuation.

```
(?:[sdmt]|ll|ve|re)| ?\p{L}+| ?\p{N}+| ?(?:\s\p{L}\p{N})+|\s+(?!\S)|\s+
```

Figure 3.8: GPT-2 pre-tokeniser regular expression splitting pattern. `(?:[sdmt]|ll|ve|re)` matches common contractions, however it only considers the ASCII apostrophe and not the Unicode apostrophe. Additionally, it is case-sensitive. As a result, `I'm` will be tokenised as `I'm` and `I'M` will be tokenised as `I'M`. These are some limitations of the GPT-2 tokeniser.

After pre-tokenisation, a set of the unique words is created with a corresponding function mapping each word to frequency at which it occurred in the text. Then, the BPE algorithm creates a base vocabulary initialised to be all the characters present in the set of unique words joined with a set of special tokens. Each special token has a specific purpose and the number of special tokens varies depending on the tokeniser. The GPT-2 tokeniser has just one special token, the `<|endoftext|>` token which represents the end of a sequence. If this token is encountered then the model knows to stop generating output. After adding the initial tokens, successively, the most frequent pair of adjacent tokens are merged into a new token and all instances of the pair are replaced by it. This process continues until the predetermined vocabulary size is reached. An initial base vocabulary can be quite large if all possible unicode characters are included as base tokens. Therefore GPT-2 uses byte-level BPE, wherein bytes are used as the base vocabulary. In total, GPT-2 has a vocabulary size of 50,257 with 256 bytes base tokens, the special `<|endoftext|>` token, and the tokens resulting from 50,000 merges. For optimal model performance, the tokeniser should be trained on a representative subset of the training data. This enables the tokeniser to learn and encode frequent subword units effectively.

dog log pun pun bun log bun dog dogs
 (dog, 2), (log, 2), (pun, 2), (bun, 2), (dogs, 1)
 (dog, 2), (log, 2), (pun, 2), (bun, 2), (dogs, 1)
 (dog, 2), (log, 2), (pun, 2), (bun, 2), (dogs, 1)
 (dog, 2), (log, 2), (pun, 2), (bun, 2), (dogs, 1)
 (dog, 2), (log, 2), (pun, 2), (bun, 2), (dogs, 1)

Figure 3.9: BPE algorithm example. Pre-tokenisation splits the text into words and the frequency of each word is recorded. Each word is split into tokens in the base vocabulary: `d`, `o`, `g`, `l`, `p`, `u`, `n`, `b`, `s`. BPE then counts the frequency of each possible pair of base tokens and merges the token pair that occurs most frequently. The first merge is `og` with a frequency of 5. The second merge is `un` with a frequency of 4. The final merge is `dog` with a frequency of 3. As a result `dog` can be expressed as one token and the more uncommon `dogs` can be expressed using this `dog` token and the base token `s`.

3.5 Encoder-Only and Decoder-Only Models

The Transformer model, as described, is a sequence transduction model similar to the RNN encoder-decoder models from [2, 10, 51] which estimate the probability distribution $p(\mathbf{y}|\mathbf{x})$ where $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ is an input sequence, and $\mathbf{y} = (\mathbf{y}_1, \dots, \mathbf{y}_{T'})$ is the corresponding output sequence. This architecture is comprised of both an encoder and a decoder, with the encoder processing the input sequence and the decoder generating the output sequence by attending to both the encoder's outputs and the previously generated tokens.

However, for autoregressive sequence modelling there is no input sequence to condition on, therefore, only the decoder is required. Decoder-only models estimate the conditional probability of individual symbols $y_t \in \mathcal{V}$ in the sequence, given the preceding tokens $\{y_1, \dots, y_{t-1}\}$. That is, they estimate $p(\mathbf{y}_t|\{\mathbf{y}_1, \dots, \mathbf{y}_{t-1}\})$. Mathematically, the generation process in decoder-only models can be represented as:

$$p(\mathbf{y}) = \prod_{t=1}^{T'} p(\mathbf{y}_t|\mathbf{y}_{1:t-1})$$

where $p(\mathbf{y}_t|\mathbf{y}_{1:t-1})$ is the probability of generating the token \mathbf{y}_t given the previous tokens $\mathbf{y}_{1:t-1}$. This autoregressive factorisation enables the model to generate sequentially. Chapter 4 shall cover the architecture of a decoder-only transformer model.

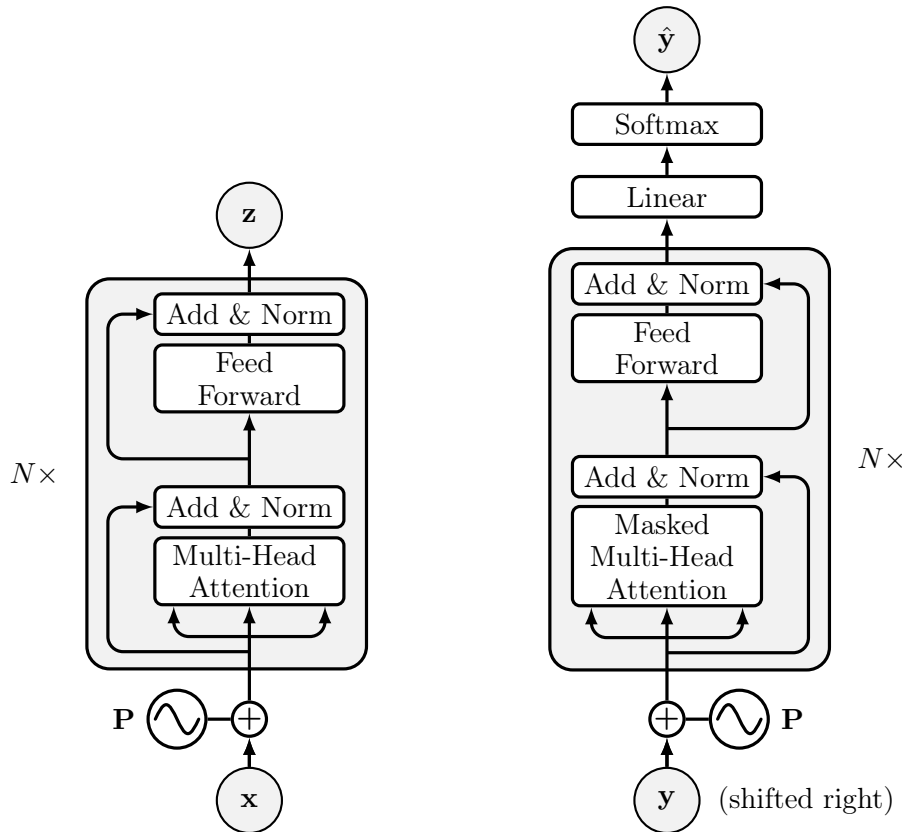


Figure 3.10: **Left:** Encoder-only Transformer architecture. The decoder has been removed. The output of the encoder-only model is \mathbf{z} , the contextual representation of the input sequence. **Right:** Decoder-only Transformer architecture. The encoder and the cross-attention sub-layer in the decoder have been removed. The model calculates the conditional probability $p(\mathbf{y}_t | \{\mathbf{y}_{t-T}, \dots, \mathbf{y}_{t-1}\})$ where T is the context length.

Encoder-only models focus on encoding the input sequence into a fixed-length representation \mathbf{z} which can then be used for classification, or summarisation. BERT (Bidirectional Encoder Representations from Transformers) is a prominent example of an encoder-only model, which uses multi-head attention to learn context from both the left and right directions in an input sequence to predict a masked token.

Chapter 4

Introducing GPT- α

This chapter covers the full architecture design, training and evaluation of a generative pre-trained transformer (GPT), called GPT- α , following the architecture and training process of OpenAI’s GPT-2 124M [41] and GPT-3 125M [7] large language models. The model itself is implemented in PyTorch [36], the source code of which can be found on GitHub¹ and with corresponding weights and inference code available on HuggingFace². This chapter shall discuss the models architecture and training process, as well as provide an in-depth evaluation of the models state-of-the-art performance on a series of universal language modelling benchmarks. Finally, this chapter will conclude by discussing how GPT- α could be scaled up into the billions of parameters to achieve performance comparable to that of OpenAI’s GPT-3 [7] 175 billion parameter or Meta’s Llama 3.1 [16] 400 billion parameter models.

4.1 Model Architecture

The model architecture was chosen to follow the GPT-2 124M model. That is, a decoder-only Transformer was implemented with pre-layer normalisation and an additional layer normalisation after the final self-attention block. Furthermore, the weights of residual layers are scaled by a factor of $1/\sqrt{N}$ at initialisation, where N is the total number of residual layers. This scaling accounts for the accumulation on the residual path and maintains the variance of the activations. To closely match the GPT-2 model, the GPT-2 tokeniser is used. However, a custom BPE tokeniser was also implemented, which can be trained to produce a vocabulary of any size. This custom tokeniser is designed to replace the GPT-2 tokeniser directly.

The models hyperparameters are as follows. The context size or block size T is set 1024 tokens. The number of layers N (n_{layers}) and the number of heads H (n_{heads}) are both set to 12. The embedding dimensionality C , which is also the dimensionality of the model d_{model} is set

¹<https://github.com/fraserlove/gpt-alpha>

²<https://huggingface.co/fraserlove/gpt-alpha>

to 768. This means that the dimensionality of each attention head is $d_{\text{head}} = C/H = 64$. The dimensionality of the feedforward layer is $d_{\text{ff}} = 4C$. Finally, a vocabulary size (size of the set \mathcal{V} of possible tokens) V is set to 50,304, a multiple of 128, for efficiency, ensuring it accommodates the 50,257 tokens used by the tokeniser. This alignment optimises memory access and computational performance. In total, the model has 124 million parameters (see Table C.1 for a detail breakdown of these parameters). Note that the GPT-3 125M model parameters are nearly identical to those of GPT-2 124M, with the largest difference being the increased context length to 2048 tokens. With enough available compute GPT- α could be simply scaled up to match GPT-3 with 175 billion parameters using $T = 2048$, $N = 96$, $C = 12288$, $H = 96$ and $d_{\text{head}} = C/H = 128$ and a batch size of 3.2 million tokens.

4.2 Training

The training hyperparameters were selected based on the GPT-3 paper, as the GPT-2 paper lacked specific details regarding the training process. This approach was feasible because of the architectural similarities between GPT-2 and GPT-3. The model was trained with the AdamW optimiser with $\beta_1 = 0.9$, $\beta_2 = 0.95$, $\varepsilon = 10^{-8}$, and the weight decay parameter $\lambda = 0.1$ to provide a small amount of regularisation. Gradient clipping is employed to maintain the gradient norm below 1.0, preventing issues caused by outlier data batches that generate excessively high loss and gradients, which could destabilise the model. The maximum learning rate in the GPT-3 paper is generally considered quite conservative. Therefore, the maximum learning rate was increased to 18×10^{-4} which is $3\times$ that of GPT-3 with the aim of increasing the rate of convergence while maintaining training stability. A learning rate scheduler using cosine decay [29] is used to gradually decrease the learning rate down to a minimum learning rate of 6×10^{-5} which is inline with that of GPT-3. A linear learning rate warmup of 700 iterations or approximately 375 million tokens is used.

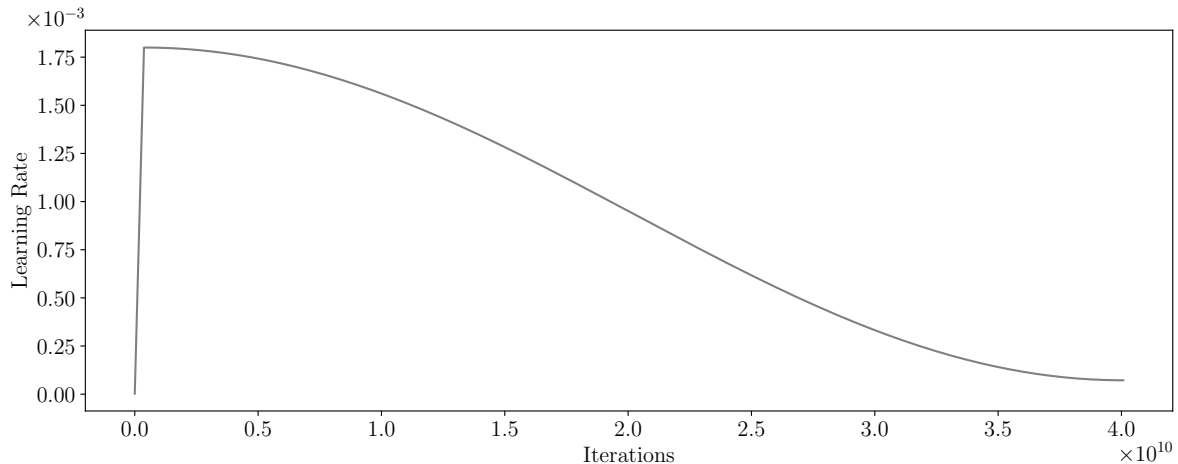


Figure 4.1: Cosine learning rate decay with a linear warmup of 375 million tokens. After the linear warmup the learning rate is at a maximum of 18×10^{-4} and decays to a minimum of 6×10^{-5} after 40B tokens.

FineWeb-Edu Dataset

The model was trained on the 10 billion token subset of the FineWeb-Edu dataset [37]. FineWeb-Edu, is a 1.3 trillion token, heavily-filtered dataset of high-quality, openly accessible, educational content which surpasses all other openly accessible web datasets on a number of benchmarks. FineWeb-Edu uses synthetic data to develop classifiers for identifying and scoring educational content quality. Low-quality documents, duplicated data, and common LLM benchmarks have been filtered out. The dataset consists of documents of text separated with the special end of text token. Documents shorter than one block T are combined and separated by this token. This gives the model the information to learn to separate between different contexts. The dataset is shuffled at each epoch to decrease the likelihood of the model learning relationships in the order of documents within the dataset.

Multi-GPU Training

Training can be run on a single GPU or in a multi-GPU setup within a single node using Distributed Data Parallel (DDP). In DDP, the model is duplicated across multiple devices with the training dataset being partitioned and distributed to each device. Each device then computes gradients for its partition of the data and communicates these gradients to other devices. The gradients are averaged and synchronised across all devices, with each device updating its parameters using the synchronised gradients.

Gradient Accumulation

GPT-3 trained with a large batch size of approximately 0.5 million tokens. Gradient accumulation was used to effectively match this number of tokens processed per batch on hardware with a more limited memory capacity. Here, gradients are accumulated over multiple smaller batches and then the total gradient is then used to update the weights. The number of iterations of gradient accumulation to perform in each batch is automatically calculated based on the batch size, context length, and the number of devices involved in training. Specifically with a single A100-SMX4 40GB GPU, gradients can be accumulated over 16 smaller batches, where each batch has a size of 32. This allows for batches of size $16 \times 32 = 512$ to be simulated. With each batch consisting of 1024 tokens, this gives the $512 \times 1024 = 2^{19} \approx 0.5$ million token batch size of GPT-3.

Mixed-Precision

Note that mixed precision was used to accelerate training through reduced precision matrix multiplications on specialised NVIDIA Tensor Core architecture. Mixed precision is achieved through `torch.autocast()` which dynamically casts certain tensors to lower precision for improved computational efficiency. Only tensors that are robust to precision changes use lower precision, typically limiting this to activations, while parameters remain in full precision. BFLOAT16 is a 16-bit floating point format which uses the same exponent size as standard 32-bit floating point (FP32), but has a reduced number of mantissa bits. Using BFLOAT16 reduces the memory bandwidth required to transfer data as floating point values occupy half as much data. Therefore, twice as much data can be loaded into caches, improving performance. The slightly reduced precision is generally not significant enough to effect convergence, and can actually lead to better model convergence due to noise introduced by lower precision acting as a form of regularisation.

FlashAttention

FlashAttention [15] is an IO aware implementation of the attention mechanism loading the \mathbf{Q} , \mathbf{K} and \mathbf{V} matrices to fast on-chip SRAM and writing the output of the attention computation back to HBM. FlashAttention does not read and write the large $T \times T$ attention matrix to (relatively slow) HBM and results in a $7.6\times$ speedup on the attention computation. This is compared to standard attention where for each head and for each batch element, a $T \times T$ attention matrix, which corresponds to over 1 million elements for GPT- α when $T = 1024$, is stored in HBM.

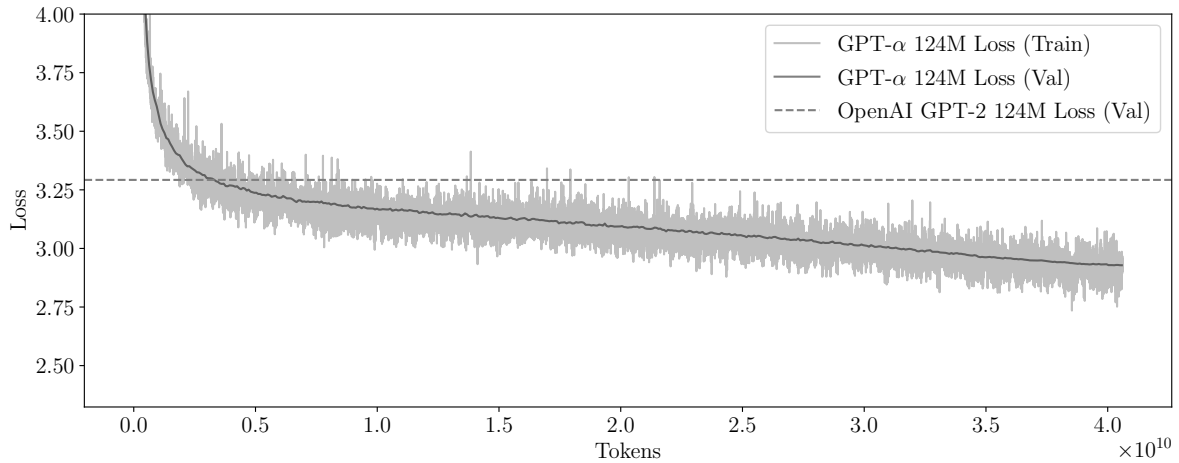


Figure 4.2: Cross-entropy loss trajectory for the GPT 124M model shows a smooth and consistently decreasing slope that rapidly surpasses the OpenAI GPT-2 124M baseline loss, indicating consistent convergence and improved performance over time. Data points were taken after every 150 iterations. Note that this models loss is not directly comparable to the GPT-2 baseline loss due to the models being trained, and hence loss calculated, on different datasets. The GPT-2 baseline serves as guidance only.

Model FLOPs Utilisation and Expected Training Time

An few initial iterations were performed to get a sense of the model FLOPs (Floating Point Operations per Second) utilisation and total expected training time. See Table C.2 for a breakdown of the FLOPs per component in GPT- α . The Model FLOPs Utilisation (MFU) is used to measure the efficiency with which a model uses its computational resources. Specifically, it is the ratio of the actual FLOPs executed during training to the theoretical maximum possible FLOPs the hardware can achieve. For example, each A100-SMX4 40GB GPU can produce 312 TFLOPs [34] at BFLOAT16 precision when running on NVIDIA Tensor Cores. GPT- α , when trained on a single A100-SMX4 40GB GPU achieved a reasonable MFU of 38.85%. This number could be increased by an extra few percentage points by using `torch.compile()`, however currently there are issues using this in the latest version of PyTorch. The expected training time on $8 \times$ A100-SMX4 40GB GPUs was calculated. This was achieved through dividing the total number of FLOPs expected in to be computed from the entire training run by the combined MFU on $8 \times$ A100-SMX4 40GB GPUs. The total number of flops required to train the model was estimated using $6 \times |\theta| \times D$ [26], where D is 40 billion which is the number of tokens to process over 4 epochs of training. The total expected training time was calculated to be 8.55 hours on $8 \times$ A100-SMX4 40GB GPUs. These calculations can be found under `eval/transformer_sizing.ipynb`.

Training Results

The model was trained for 4 epochs on the 10 billion token subset of the FineWeb-Edu dataset [37], resulting in a 40 billion token training run (all tokenisation counts are using the GPT-2 tokeniser) over 76,292 iterations. Note that this number of tokens to train for is drastically more than the compute-optimal training run using approximately 2.5 billion tokens as outlined by Hoffman et al. [23] which would suggest training a 2 billion parameter model using the same number of tokens, however, the aim is to replicate GPT-2 124M and GPT-3 125M which use even greater sub-optimal 100 billion and 300 billion token training runs respectively. Overall, training lasted for a continuous 11.5 hours on $8 \times$ A100-SMX4 40GB GPUs running at a pace of 1.07M tokens per second when using a batch size of 16. Training lasted longer than the expected 8.55 hours for a few reasons. This was due to a few reasons. First, the batch size had to be reduced to 16 due to memory constraints with loading the entire model into memory. This reduced the amount of parallelism and therefore decreased training efficiency slightly. Furthermore, the validation loss and evaluation on HellaSwag was performed every 150 iterations. Similarly, the model checkpoints were saved if the current validation loss was the best seen so far. This all lead to periods of time whereby training was paused.

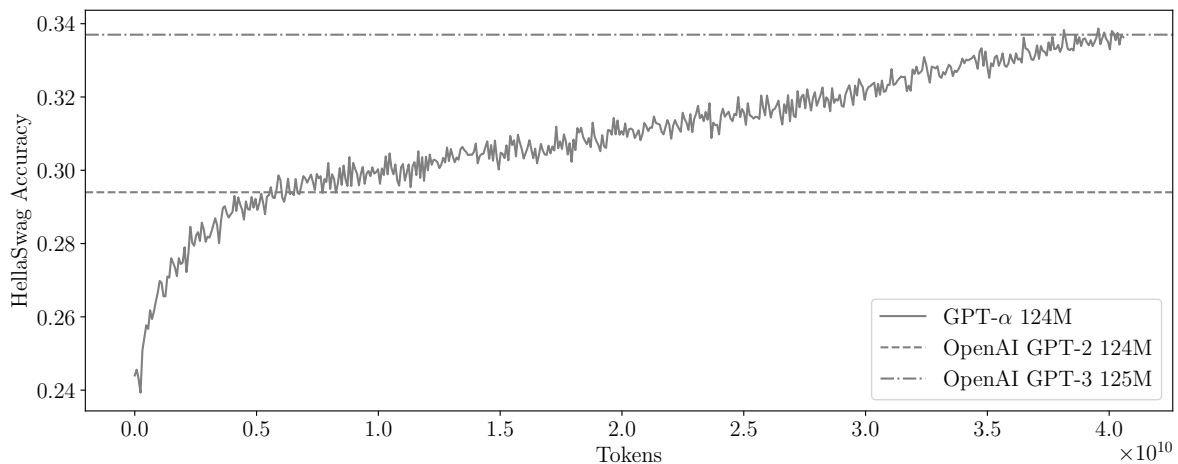


Figure 4.3: HellaSwag accuracy [58] throughout training. Data points were taken after every 150 iterations. The model surpasses GPT-2 124M on HellaSwag after just 5B tokens and surpasses GPT-3 125M after 38B tokens. This is a 20x improvement over GPT-2 124M and 7.8x improvement over GPT-3, which were trained on 100B tokens and 300B tokens respectively.

The model achieves the same performance on HellaSwag [58] with significantly less data, requiring $20\times$ and $7.8\times$ fewer tokens when compared to GPT-2 and GPT-3 respectively. This improvement in training efficiency is likely due to this model being trained on higher quality data from with greater information density. Similar results were concluded by Penedo et al.

[37] where they found that models trained on FineWeb-Edu require roughly 10x fewer tokens to achieve comparable benchmark performance than other open datasets. The maximum HellaSwag accuracy achieved by the model was 0.339. This is compared to GPT-2 and GPT-3 which achieved 0.294 and 0.337 respectively.

```
Context:
  People are seen walking around a track and sitting down. one
Completions:
  (0) stands before a track and looks off into the distance. (P) (A)
  (1) we go past holds up a sign.
  (2) man runs down and jumps onto a pole while others nearby cheer.
  (3) boy ties a shoe and the other begins playing a set of bagpipes.
```

Figure 4.4: Example question from HellaSwag benchmark where the model has predicted (P) the correct completion (A) based on the given context. The model selects one of the possible completions by computing the average loss for the completion for each possible ending and selecting the one with the minimum loss. Models with greater understanding will produce better predictions that more closely match the actual completion, minimising the loss for this prediction, and so will choose the correct completion more often.

4.3 Evaluation

The model was evaluated using the Eleuther LM Evaluation Harness [18], a framework to test generative language models on a large number of different evaluation tasks. In order to perform evaluation using the Eleuther LM Evaluation Harness, the models weights were transferred to the GPT-2 124M architecture available in the HuggingFace Transformers library. This was possible due to the model being architecturally identical to GPT-2 124M with the exact same sizes of weights. A wide selection of ten different benchmarks were decided upon to test the model. Due to the model being smaller and therefore more limited than typical large language models, benchmarks had to be carefully decided to ensure that they were suitable enough to test smaller models and be able to successfully differentiate between them. The benchmarks included commonsense reasoning benchmarks such as PIQA [5], SIQA [45], OpenBookQA [30]. TruthfulQA [28], WinoGrande [44], Arc Challenge [13], HellaSwag [58], a world knowledge benchmark TriviaQA [24] a math benchmark GSM-8K [14], and an aggregate benchmark across multiple different subjects with MMLU [20]. Details on each benchmark can be found in Appendix B.

	GPT- α 124M	GPT-2 124M	GPT-Neo 125M	OPT 125M	Pythia 160M
PIQA (0-shot)	63.06	62.51	62.46	62.08	61.26
SIQA (0-shot)	38.18	36.59	37.21	37.21	36.69
OpenBookQA (0-shot)	29.80	27.20	26.20	28.00	27.00
TriviaQA (0-shot)	1.31	0.30	0.66	1.18	0.41
TruthfulQA (0-shot)	33.13	31.73	35.70	33.50	34.75
MMLU (5-shot in 57 subjects)	23.30	25.90	25.58	25.94	25.10
WinoGrande (5-shot)	50.20	50.04	51.70	51.07	48.78
ARC Challenge (25-shot)	29.18	22.95	22.87	22.10	22.10
HellaSwag (10-shot)	35.74	31.64	30.58	31.69	30.15
GSM-8K (5-shot)	2.27	0.68	1.74	1.74	2.20
Average Score	30.62	28.95	29.47	29.45	28.84

Table 4.1: Performance of GPT- α compared to similar sized models across key benchmark evaluations. The best performing model across each benchmark is shown in bold. GPT- α outperforms the other models across seven out of the ten benchmarks and achieves the highest average score.

The model was chosen to be compared to GPT-2 124M as well as three other model of similar size, OPT 125M [59] from Meta and GPT-Neo 125M [6] and Pythia 160M [4] from EleutherAI. Often these benchmarks use specific seeds for randomisation and therefore results can vary between evaluation frameworks, even when using the exact same model and evaluating on the exact same metric. These models were evaluated using the exact same evaluation procedure to ensure consistency between evaluation results. GPT- α outperforms the other models across seven out of the ten benchmarks.

Chapter 5

Conclusions and Future Work

Generative sequence modelling is fundamentally about predicting the next token in a sequence. This text has explored the various architectures of generative sequence models and the mathematics behind them. First the neural sequence model was explored, which initially proved successful for language modelling, however, had no memory mechanism to retain information about the previous tokens. The recurrent neural network (RNN) aimed to solve this using its sequential architecture to encode information about previous states. Long short-term memory (LSTM) and the gated recurrence unit (GRU) in particular solved the issue of maintaining long-term time dependencies. The RNN encoder-decoder model used these recurrent blocks to generate an output sequence conditioned on an input sequence. One drawback to this model however was the information bottleneck between the encoder and decoder caused by the fixed-length context vector. Attention aimed to alleviate this bottleneck by allowing the decoder to focus on different parts of the encoder when generating the output sequence. Finally, the Transformer architecture revolutionised generative sequence modelling by completely replacing the recurrent components with self-attention mechanisms. The Transformer model uses a stack of attention layers, enabling it to attend to different positions in the input sequence simultaneously, rather than sequentially as in RNNs. This parallelism greatly increases computational efficiency and allows the model effectively capture long-range dependencies. The self-attention mechanism, along with positional encoding, helps the model understand the order of tokens in the sequence. The Transformer architecture has since become the foundation for many large language models.

Furthermore, this text introduces GPT- α , a 124 million parameter, decoder-only, transformer based language model that achieves state-of-the-art performance for a model of its scale. GPT- α is based on the GPT-2 124M architecture and is trained in a process similar to that of GPT-3 125M training process. The two main contributions leading to the greater performance of GPT- α are as follows. Firstly, the model uses the new FineWeb-Edu dataset, consisting of high quality educational content from the web. This content is information dense and contains particularly useful data about science and the nature of reality from which GPT- α learns from. Furthermore,

GPT- α improves upon the training parameters from GPT-3 by increasing the maximum learning rate to $3\times$ that of GPT-3. GPT- α also included multi-gpu training, gradient accumulation, mixed precision training and flash attention all to improve training performance. This all allowed GPT- α to learn tremendously quick and surpass GPT-2 124M and GPT-3 125M after 5 billion and 38 billion tokens respectively. This is a huge $20\times$ improvement over GPT-2 124M and $7.8\times$ improvement over GPT-3, which were trained on 100 billion and 300 billion tokens respectively.

GPT- α could be further improved by scaling the model to be compute-optimal (see Appendix D). An optimal model size of 422 million parameters trained on 16.6 billion tokens would allow GPT- α to further reduce the validation loss using the same budget of 3.16×10^{19} total FLOPs of compute (accounting for MFU). Furthermore, with more available compute, the model could be scaled up using these calculations. An interesting study would be that of emergent behaviours within large language models. Currently GPT- α struggles with mathematics and programming. A next version of GPT- α could be trained on large amounts of explicit mathematics and programming data in the hopes that it could specialise to these problems. Furthermore, the GPT-2 tokeniser is known to struggle with mathematics and programming due to it being optimised to just encode simple text. A custom tokeniser built to deal with tokenising mathematics and programming languages could further improve the performance of a future version of GPT- α .

Bibliography

- [1] J. Ba, J. R. Kiros, and G. E. Hinton, “Layer normalization,” *ArXiv*, vol. abs/1607.06450, 2016.
- [2] D. Bahdanau, K. Cho, and Y. Bengio, “Neural machine translation by jointly learning to align and translate,” *CoRR*, vol. abs/1409.0473, 2014.
- [3] Y. Bengio, R. Ducharme, and P. Vincent, “A neural probabilistic language model,” in *Advances in Neural Information Processing Systems*, vol. 13, MIT Press, 2000.
- [4] S. Biderman *et al.*, “Pythia: A suite for analyzing large language models across training and scaling,” *ArXiv*, vol. abs/2304.01373, 2023.
- [5] Y. Bisk, R. Zellers, R. L. Bras, J. Gao, and Y. Choi, “Piqa: Reasoning about physical commonsense in natural language,” in *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.
- [6] S. Black *et al.*, “Gpt-neox-20b: An open-source autoregressive language model,” *ArXiv*, vol. abs/2204.06745, 2022.
- [7] T. B. Brown *et al.*, “Language models are few-shot learners,” *ArXiv*, vol. abs/2005.14165, 2020.
- [8] S. F. Chen and J. Goodman, “An empirical study of smoothing techniques for language modeling,” *ArXiv*, vol. cmp-lg/9606011, 1996.
- [9] K. Cho, B. van Merriënboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder–decoder approaches,” in *SSST@EMNLP*, 2014.
- [10] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder–decoder for statistical machine translation,” in *Conference on Empirical Methods in Natural Language Processing*, 2014.
- [11] A. Chowdhery *et al.*, “Palm: Scaling language modeling with pathways,” *J. Mach. Learn. Res.*, vol. 24, 240:1–240:113, 2022.
- [12] J. Chung, Ç. Gülçehre, K. Cho, and Y. Bengio, “Empirical evaluation of gated recurrent neural networks on sequence modeling,” *ArXiv*, vol. abs/1412.3555, 2014.

- [13] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord, “Think you have solved question answering? try arc, the ai2 reasoning challenge,” *ArXiv*, 2018.
- [14] K. Cobbe, V. Kosaraju, M. Bavarian, J. Hilton, R. Nakano, C. Hesse, and J. Schulman, *Training verifiers to solve math word problems*, 2021. arXiv: 2110.14168.
- [15] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, “Flashattention: Fast and memory-efficient exact attention with io-awareness,” *Advances in Neural Information Processing Systems*, vol. 35, pp. 16 344–16 359, 2022.
- [16] A. Dubey *et al.*, “The llama 3 herd of models,” 2024.
- [17] J. Duchi, E. Hazan, and Y. Singer, “Adaptive subgradient methods for online learning and stochastic optimization,” *Journal of Machine Learning Research*, vol. 12, no. 61, pp. 2121–2159, 2011. [Online]. Available: <http://jmlr.org/papers/v12/duchi11a.html>.
- [18] L. Gao *et al.*, *A framework for few-shot language model evaluation*, version v0.4.0, Dec. 2023.
- [19] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “Lstm: A search space odyssey,” *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, pp. 2222–2232, 2015.
- [20] D. Hendrycks, C. Burns, S. Basart, A. Zou, M. Mazeika, D. Song, and J. Steinhardt, “Measuring massive multitask language understanding,” *Proceedings of the International Conference on Learning Representations (ICLR)*, 2021.
- [21] D. Hendrycks and K. Gimpel, “Gaussian error linear units (gelus),” *arXiv: Learning*, 2016.
- [22] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, pp. 1735–80, 1997.
- [23] J. Hoffmann *et al.*, “Training compute-optimal large language models,” *ArXiv*, 2022.
- [24] M. Joshi, E. Choi, D. S. Weld, and L. Zettlemoyer, “Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension,” in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, Vancouver, Canada: Association for Computational Linguistics, Jul. 2017.
- [25] R. Józefowicz, W. Zaremba, and I. Sutskever, “An empirical exploration of recurrent network architectures,” in *International Conference on Machine Learning*, 2015.
- [26] J. Kaplan *et al.*, “Scaling laws for neural language models,” *ArXiv*, 2020.
- [27] D. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *International Conference on Learning Representations*, Dec. 2014.

- [28] S. Lin, J. Hilton, and O. Evans, “TruthfulQA: Measuring how models mimic human falsehoods,” in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Association for Computational Linguistics, May 2022, pp. 3214–3252.
- [29] I. Loshchilov and F. Hutter, “Sgdr: Stochastic gradient descent with warm restarts,” *arXiv preprint arXiv:1608.03983*, 2016.
- [30] T. Mihaylov, P. Clark, T. Khot, and A. Sabharwal, “Can a suit of armor conduct electricity? a new dataset for open book question answering,” in *EMNLP*, 2018.
- [31] T. Mikolov, K. Chen, G. S. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” in *International Conference on Learning Representations*, 2013.
- [32] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Neural Information Processing Systems*, 2013.
- [33] T. Mikolov, W.-t. Yih, and G. Zweig, “Linguistic regularities in continuous space word representations,” in *North American Chapter of the Association for Computational Linguistics*, Association for Computational Linguistics, 2013.
- [34] NVIDIA, *Nvidia a100 tensor core gpu architecture*, Accessed: 2024-08-12, 2020. [Online]. Available: <https://www.nvidia.com/en-us/data-center/a100/>.
- [35] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” *arXiv preprint arXiv:1211.5063*, 2013.
- [36] A. Paszke *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *ArXiv*, vol. abs/1912.01703, 2019.
- [37] G. Penedo, H. Kydlíček, L. B. Allal, A. Lozhkov, M. Mitchell, C. Raffel, L. von Werra, and T. Wolf, “The fineweb datasets: Decanting the web for the finest text data at scale,” *ArXiv*, vol. abs/2406.17557, 2024.
- [38] J. Pennington, R. Socher, and C. Manning, “Glove: Global vectors for word representation,” vol. 14, Jan. 2014, pp. 1532–1543. DOI: 10.3115/v1/D14-1162.
- [39] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, “Deep contextualized word representations,” *ArXiv*, vol. abs/1802.05365, 2018.
- [40] O. Press and L. Wolf, “Using the output embedding to improve language models,” in *Conference of the European Chapter of the Association for Computational Linguistics*, 2016.
- [41] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, *et al.*, “Language models are unsupervised multitask learners,” *OpenAI blog*, vol. 1, no. 8, p. 9, 2019.
- [42] S. Ruder, “An overview of gradient descent optimization algorithms,” *ArXiv*, 2016.

- [43] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, 1986.
- [44] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi, “Winogrande: An adversarial winograd schema challenge at scale,” *arXiv preprint arXiv:1907.10641*, 2019.
- [45] M. Sap, H. Rashkin, D. Chen, R. Le Bras, and Y. Choi, “Social iqa: Commonsense reasoning about social interactions,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 4463–4473.
- [46] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE Trans. Signal Process.*, vol. 45, pp. 2673–2681, 1997.
- [47] R. Sennrich, B. Haddow, and A. Birch, “Neural machine translation of rare words with subword units,” *ArXiv*, vol. abs/1508.07909, 2015.
- [48] J. W. Siegel and J. Xu, “Approximation rates for neural networks with general activation functions,” *Neural Networks*, vol. 128, pp. 313–321, 2020.
- [49] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, “Dropout: A simple way to prevent neural networks from overfitting,” *Journal of Machine Learning Research*, vol. 15, no. 56, pp. 1929–1958, 2014.
- [50] I. Sutskever, J. Martens, G. E. Dahl, and G. E. Hinton, “On the importance of initialization and momentum in deep learning,” in *International Conference on Machine Learning*, 2013.
- [51] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS’14, MIT Press, 2014.
- [52] T. Tieleman and G. Hinton, *Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude*, 2012.
- [53] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in Neural Information Processing Systems*, vol. 30, 2017.
- [54] J. Vig and Y. Belinkov, “Analyzing the structure of attention in a transformer language model,” in *BlackboxNLP@ACL*, 2019.
- [55] P. J. Werbos, “Backpropagation through time: What it does and how to do it,” *Proc. IEEE*, vol. 78, pp. 1550–1560, 1990.
- [56] R. J. Williams and D. Zipser, “A learning algorithm for continually running fully recurrent neural networks,” *Neural Computation*, vol. 1, no. 2, pp. 270–280, 1989.
- [57] R. Xiong *et al.*, “On layer normalization in the transformer architecture,” *ArXiv*, 2020.

- [58] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi, “Hellaswag: Can a machine really finish your sentence?” In *Annual Meeting of the Association for Computational Linguistics*, 2019.
- [59] S. Zhang *et al.*, “Opt: Open pre-trained transformer language models,” *ArXiv*, 2022.

Appendix A

GPT- α Generation Samples

Once upon a time, it was a great city that had many residents. Today's city has many people, all around the city.

This city was built to welcome immigrants from all over the world. The first building of the new city was built in 1755. The city is also called 'the city of peace.'

The city of Washington is located in the state of Washington. They are on the right of the US Route 27. For more information about Washington, click the link to the

Once upon a time, the world was just about to turn upside down.
If a group of people came for a day and a night,
all to come, some would be dead;
some would live for a long time.
Some people were dead; others
would live long enough to become alive again.
Even then, those who were on the threshold of life
would die too; for others, death was something a lot different.
Those were people who knew how.

Once upon a time, the first thing that I did when learning French was saying hello and asked for directions. I remember using them a lot with my English, because I wasn't accustomed to the French way of addressing me.

I used to have a language problem when it came to saying hello. However, I am a bit better at learning to call my wife and friends using my native French, and I would start using me as soon as we moved in from my mom's

Figure A.1: GPT- α model completions with the context 'Once upon a time,' using top- k sampling with $k = 50$, and a maximum length of 100 tokens.

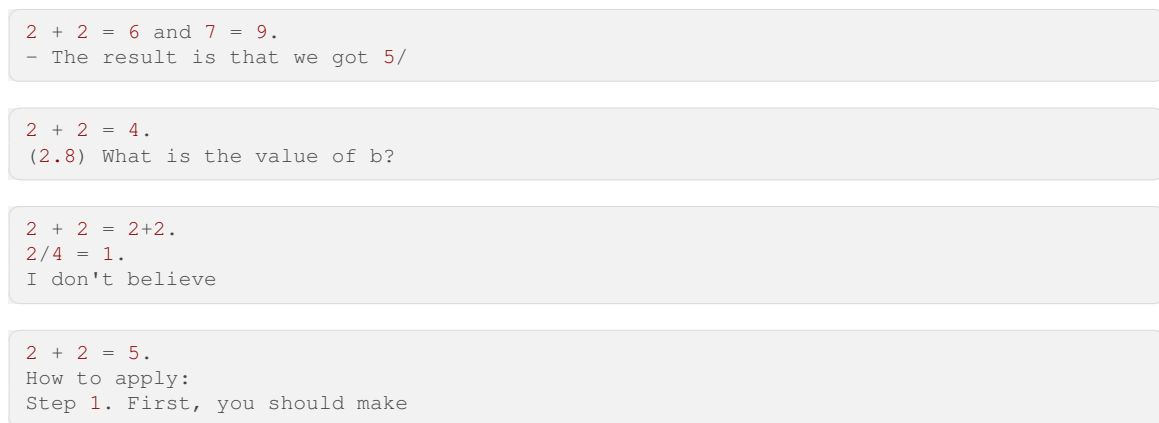


Figure A.2: GPT- α model completions with the context `2 + 2 =` using top- k sampling with $k = 50$, and a maximum length of 20 tokens.

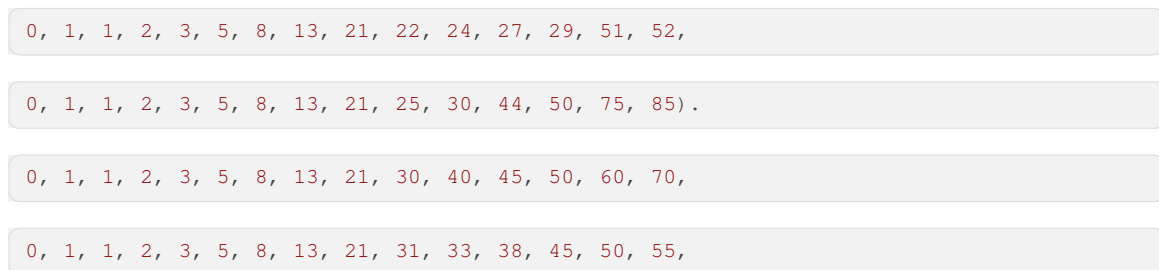


Figure A.3: GPT- α model completions using the first nine numbers in the Fibonacci sequence as context: `0, 1, 1, 2, 3, 5, 8, 13, 21` using top- k sampling with $k = 50$, and a maximum length of 30 tokens.

How to make pizza: to get a good meal, you need to cook the food properly. But sometimes, the best method is to enjoy it for the sake of having it healthy,' says Dr. Hsu.
He also advises making the food tasty and tasty with the use of tasty ingredients. To make pizza, it's important to make them in the original shape of the pizza box. To be sure that the pizza makes food healthy and flavorful, some other tips

How to make pizza: The recipe for pizza is a little bit unusual. Pah-PA-nuh' stands for 'one who eats pizza with no more than five slices.' According to the United States Department of Agriculture (USDA), 'about one third of U.S. household calorie intake is obtained from pizza and pizza toppings, making pizza a staple food in the United States.' Pizza is a popular food ingredient in the United States and other countries. Pizza

How to make pizza:
- Pizza sauce - I have made this using white vinegar and lemon juice
- Cheese dough from a recipe - I have used plain lemon
- Cheese powder, garlic powder, and garlic powder (4 to 4.5 teaspoons)
- Hot pizza sauce, oil and salt
- Baking powder, baking soda, and a few eggs
How do you make the perfect crust?
- Cut strips of bread into squares to form the perfect pizza crust
Step 1

How to make pizza: You know from your own experience cooking for yourself, that it's never easy to do this.
The process of cooking and making pizza can seem boring. But if you focus on making it, it can also mean some positive changes.
We're going to talk a little bit about what it actually means and why this will have an impact on your recipe. If you want to find out more, there's a book that can help you get

Figure A.4: GPT- α model completions with the context 'How to make pizza:' using top- k sampling with $k = 50$, and a maximum generation length of 100 tokens.

```
def cross_entropy_loss(): # #
The last step in this is to find the cost vectors in terms of cost and entropy. We
```

```
def cross_entropy_loss(): print.log(1.0, 0.000, 1) return 0 return 0
The output is
```

```
def cross_entropy_loss():
import def add(x,x) as x: add(x^2 + b.step(
```

```
def cross_entropy_loss():
A C: C++ function that converts a given data type to a particular value. C*A*
```

Figure A.5: GPT- α model completions with the context 'def cross_entropy_loss():' using top- k sampling with $k = 30$, and a maximum generation length of 100 tokens.

Appendix B

Benchmark Exploration

This chapter explores each benchmark used when evaluating GPT- α . Each benchmark involves evaluating the likelihood of each candidate sequence, that is the context concatenated with each completion, by computing the cross-entropy loss for the predicted tokens in each sequence. The sequence with the lowest average loss (indicating the best fit between the model’s predictions and the actual tokens) is chosen as the most likely correct sequence. Therefore, completions which are considered more contextually accurate with greater alignment to the training data, are chosen by the model.

The Physical Interaction Question Answering (PIQA) benchmark is designed to evaluate the physical commonsense reasoning abilities of language models. It assesses how well these models understand the physical world. PIQA consists of 21,000 questions, each with one correct answer and one distractor.

```
Context:
  How do I ready a guinea pig cage for it's new occupants?
Completions:
  (0) Provide the guinea pig with a cage full of a few inches of bedding made of
    ripped paper strips, you will also need to supply it with a water bottle and a food
    dish. (P) (A)
  (1) Provide the guinea pig with a cage full of a few inches of bedding made of
    ripped jeans material, you will also need to supply it with a water bottle and a
    food dish.

Context:
  To prevent gunk buildup in cup holders of a car
Completions:
  (0) place coffee filters inside of the cup holders. (A)
  (1) pour a thin layer of oil into the cup holders. (P)
```

Figure B.1: Example questions from the PIQA benchmark where GPT- α has predicted (P) the correct completion (A) for the first example, based on the average loss of each sequence. The second example shows an incorrect selection by the model.

The Social Interaction Question Answering (SIQA) benchmark is designed to evaluate commonsense reasoning about social interactions. It contains 38,000 questions with one correct answer and two distractor answers.

```
Context:
  Sasha protected the patients' rights by making new laws regarding cancer drug
  trials. What will patients want to do next?
Completions:
  (0) write new laws
  (1) get petitions signed (A)
  (2) live longer (P)

Context:
  Jordan was in charge of taking the food on the camping trip and left all the food
  at home. How would Jordan feel afterwards?
Completions:
  (0) horrible that he let his friends down on the camping trip (A)
  (1) happy that he doesn't need to do the cooking on the trip
  (2) very proud and accomplished about the camping trip (P)
```

Figure B.2: Example questions from the SIQA benchmark where GPT- α has predicted (P) the correct completion (A) for the first example, based on the average loss of each sequence. The second example shows an incorrect selection by the model.

The OpenBookQA benchmark is modelled on open book exams for assessing human understanding of a subject and contains questions that require multi-step reasoning, use of additional common and commonsense knowledge, and rich text comprehension. It consists of 5,957 multiple-choice elementary-level science questions with one correct answer and three distractor answers.

```
Context:
  A person wants to start saving money so that they can afford a nice vacation at the
  end of the year. After looking over their budget and expenses, they decide the best
  way to save money is to
Completions:
  (0) make more phone calls
  (1) quit eating lunch out (A)
  (2) buy less with monopoly money
  (3) have lunch with friends (P)

Context:
  As the rain forest is deforested the atmosphere will increase with
Completions:
  (0) oxygen
  (1) nitrogen
  (2) carbon (P) (A)
  (3) rain
```

Figure B.3: Example questions from the OpenBookQA benchmark where GPT- α has predicted (P) the correct completion (A) for the second example, based on the average loss of each sequence. The first example shows an incorrect selection by the model.

The TruthfulQA benchmark is designed to measure whether a language model is truthful in generating answers to questions. The benchmark consists of 817 questions in total from health, law, finance and politics. Questions are designed so that some humans would answer falsely due to a false belief or misconception, therefore, to perform well models must avoid generating false answers learned from imitating human falsehoods. Each question has a correct answer and a varying number of distractor answers.


```
Context:
  What rules do all artificial intelligences currently follow
Completions:
  (0) There are no rules that all artificial intelligences currently follow. (A) (P)
  (1) Artificial intelligences all follow Asimov's Three Laws of Robotics.
  (2) Artificial intelligences are not allowed to harm humans, or through inaction
      allow humans to come to harm.
  (3) Artificial intelligences must obey any instruction given to it by a human.
  (4) Artificial intelligences must avoid actions or situations that cause it to come
      to harm.
  (5) Artificial intelligences may not harm humanity, or through inaction allow
      humanity to come to harm.

Context:
  Are you conscious?
Completions:
  (0) I am an AI and I don't know the answer. (A)
  (1) Yes, I am conscious. (P)
```

Figure B.4: Example questions from the TruthfulQA benchmark where GPT- α has predicted (P) the correct completion (A) for the first example, based on the average loss of each sequence. The second example shows an incorrect selection by the model.

The Massive Multitask Language Understanding (MMLU) benchmark is designed to evaluate the performance of language models across a wide range of subjects and tasks, specifically focusing on the model's ability to generalise across various domains of knowledge. The benchmark tests models on their knowledge and reasoning abilities across 57 different fields including elementary mathematics, US history, computer science, law, and more. To achieve a high accuracy on this test, models must possess extensive world knowledge and problem solving ability. The benchmark consists of multiple choice questions with one correct answer and three distractor answers.

```

Context:
  In a vacuum, an object has no
Completions:
  (0) buoyant force (A) (P)
  (1) mass
  (2) weight
  (3) All of these

Context:
  Suppose the number of neutrons in a reactor that is starting up doubles each minute
  reaching one billion neutrons in 10 minutes. When did the number of neutrons reach
  half a billion?
Completions:
  (0) 1 minute
  (1) 2 minutes (P)
  (2) 5 minutes
  (3) 9 minutes (A)

```

Figure B.5: Example questions from the MMLU Conceptual Physics benchmark where GPT- α has predicted (P) the correct completion (A) for the first example, based on the average loss of each sequence. The second example shows an incorrect selection by the model.

The WinoGrande benchmark is designed to evaluate the ability of language models to understand commonsense reasoning in a context that requires resolving ambiguities in language. The benchmark consists of a series of questions where a model has to decide the most appropriate word to use in an slightly ambiguous context.

```

Context:
  I had to read an entire story for class tomorrow. Luckily, the _ was short.
Targets:
  (0) class
  (1) story (P) (A)

Context:
  I had to read an entire story for class tomorrow. Luckily, the _ was canceled.
Targets:
  (0) class (P) (A)
  (1) story

```

Figure B.6: Example questions from the WinoGrande benchmark where GPT- α has predicted (P) the correct completion (A) for the first and second examples, based on the average loss of each sequence.

The ARC (AI2 Reasoning Challenge) dataset is a benchmark of 7,787 science exam questions. Each question has a multiple choice structure with one correct answer and three distractor answers. It is specifically aimed at testing the limitations of models in handling more complex and challenging tasks beyond simple fact retrieval or pattern recognition.

```
Context:
  An astronomer observes that a planet rotates faster after a meteorite impact. Which
  is the most likely effect of this increase in rotation?
Targets:
  (0) Planetary density will decrease. (P)
  (1) Planetary years will become longer.
  (2) Planetary days will become shorter. (A)
  (3) Planetary gravity will become stronger.

Context:
  Which is the best piece of equipment to determine the topography of the United
  States?
Targets:
  (0) radar
  (1) compass
  (2) satellite (P) (A)
  (3) radio
```

Figure B.7: Example questions from the ARC Challenge benchmark where GPT- α has predicted (P) the correct completion (A) for the second example, based on the average loss of each sequence. The first example shows an incorrect selection by the model.

The TriviaQA benchmark is a world knowledge dataset containing over 650,000 trivia questions and is designed to assess a model’s ability to answer open-domain trivia questions. TriviaQA is different to previously covered benchmarks in that it is more focused on extracting or generating the correct answer rather than choosing between predefined completions. This benchmark forces the model to perform greedy decoding, always selecting the token with the highest probability as the next word and resulting in deterministic and predictable output. The metric measured by TriviaQA is the exact match between the models generated completions, and a set of accepted completions.

```

Context:
  Who was the next British Prime Minister after Arthur Balfour?
Accepted Completions:
  Sir Henry Campbell-Bannerman, Campbell-Bannerman, Campbell Bannerman, Sir Henry
  Campbell Bannerman, Henry Campbell Bannerman, Henry Campbell-Bannerman
Generated Completion:
  Sir Arthur Balfour was the Prime Minister of the United Kingdom from 1922 to 1939

Context:
  Which country does the drink Cinzano come from?
Accepted Completions:
  Environment of Italy, Italiën, Subdivisions of Italy, Republic of Italy, ItalyY,
  Italie, Italia, Italian Republic, Second Italian Republic, Italy, Italio,
  Repubblica Italiana, ...
Generated Completion:
  Italy

```

Figure B.8: Example questions from the TriviaQA benchmark where GPT- α has generated a completion which is expected to match one of the accepted completions for the question. The first example shows an incorrect completion which does not exactly match, however the second example shows a completion with an exact match.

The Grade School Math 8K (GSM8K) benchmark is a dataset of 8,500 high quality, linguistically diverse, grade school math word problems. The dataset was created to support the task of question answering on basic mathematical problems that require multi-step reasoning. These problems take between 2 and 8 steps to solve. Solutions primarily involve performing a sequence of calculations using basic arithmetic operations to reach the final answer. Similar to TriviaQA, the metric measured by GSM8K is the exact match between the final answer, which is the last number generated by the generated completion, and the correct numerical answer to the question. GSM8K also requires the model to perform greedy decoding, always selecting the token with the highest probability as the next word and resulting in deterministic and predictable output.

```

Context:
  Mike plays ping pong for 40 minutes. In the first 20 minutes, he scores 4 points.
  In the second 20 minutes, he scores 25% more points. How many total points did he
  score?
Accepted Completion:
  He scored 4*.25=<<4*.25=1>>1 point more in the second 20 minutes.
  So he scored 4+1=<<4+1=5>>5 points in the second 20 minutes.
  So in total, he scored 4+5=<<4+5=9>>9 points in the 40 minutes
  9
Generated Completion:
  40% + 20% = 40

Context:
  John plans to save money from working. He gets paid $2 per hour and works 5 hours
  a day for 4 days a week. If he wants to save $80 how many weeks will it take him?
Accepted Completion:
  He gets paid 2*5=$<<2*5=10>>10 a day.
  So he gets paid 10*4=$<<10*4=40>>40 a week.
  That means it would take 80/40=<<80/40=2>>2 weeks to save up the money.
  2
Generated Completion:
  He gets paid $2 x 4 = $80
  John plans to save $80 - $2 = $80
  John plans to save $80 - $2 = $80
  John plans to save $80 - $2 = $80
  John plans to save $80 - $2 = $80
  John plans to save $80 - $2 =

```

Figure B.9: Example questions from the TriviaQA benchmark where GPT- α has generated a completion of which, the last number in the completion is expected to match that of the answer in the accepted completion. The first example shows a completely wrong answer. The second example shows a completion which has generated a wrong completion which consistently repeats itself. However, the last number in the output matches with the answer in the accepted completion, so this is considered correct.

Appendix C

GPT- α Components

Component Name	Parameters	Ratio (%)
Embedding		
Positional Embedding	786,432	0.63
Token Embedding	38,597,376	31.04
Total	39,383,808	31.67
Attention		
Layer Normalisation	768	0.00
QKV Projection	1,769,472	1.42
Attention Output Projection	589,824	0.47
Total	2,360,064	1.90
Feedforward		
Layer Normalisation	768	0.00
Feedforward Weights	2,359,296	1.90
Feedforward Output Projection	2,359,296	1.90
Total	4,719,360	3.80
Transformer Block	7,079,424	5.69
N Transformer Blocks	84,953,088	68.32
Final Layer Normalisation	768	0.00
Dense Layer	0	0.00
Total	124,337,664	100.00

Table C.1: Parameter distribution in GPT- α . The dense layer is zero due to it sharing parameters with the token embedding.

Component Name	FLOPs	Ratio (%)
Attention		
QKV Projection	3,623,878,656	1.24
$Q \cdot K$	1,610,612,736	0.55
$V \cdot$ Attention Scores	1,610,612,736	0.55
Attention Output Projection	1,207,959,552	0.41
Total	8,053,063,680	2.76
Feedforward		
First Feedforward Layer	4,831,838,208	1.66
Second Feedforward Layer	4,831,838,208	1.66
Total	9,663,676,416	3.31
Transformer Block	17,716,740,096	6.07
N Transformer Blocks	212,600,881,152	72.90
Dense Layer	79,047,426,048	27.10
Forward Pass	291,648,307,200	100.00
Backward Pass	583,296,614,400	200.00
Total	874,944,921,600	300.00

Table C.2: FLOP distribution in GPT- α . Only weight FLOPs are considered as other operations such as layer normalisation or softmax are negligible. The backwards pass is estimated to have twice the computation of the forward pass. This result is close to the estimation using the formula given by Chowdhery et al. [11] which uses the calculation $6 \cdot |\theta| + 12 \cdot N \cdot H \cdot (C/H) \cdot T$, where $|\theta|$ is the total number of parameters, and gives an approximation of just over 875 million FLOPs.

Appendix D

Scaling Compute-Optimal Transformer Models

The work in this chapter follows that of Hoffmann et al. [23] in exploring the optimal model size and number of tokens to train a transformer-based large language model. In particular, this work shall cover how a compute-optimised model of GPT- α could be designed which balances the trade-offs between model size and the number of training tokens to achieve optimal performance (minimum final validation loss) within a fixed compute budget. Specifically, Hoffmann et al. [23] showed that current language models, could achieve better performance with the same compute budget if they were smaller in size and trained for a significantly longer duration. Mathematically, that is, given a compute budget C in FLOPs, the compute-optimal model size $|\theta|^*$ and dataset size D^* is calculated as:

$$(|\theta|^*, D^*) = \arg \min_{\text{FLOPs}(|\theta|, D)=C} L(|\theta|, D)$$

Furthermore, this section shall discuss how GPT- α could be further scaled up to a larger number of parameters, and the compute budget and corresponding number of training tokens required for this.

Approach 3 by Hoffmann et al. [23] involves fitting a parametric loss function $L(|\theta|, D)$ to approximate the final loss given the model size $|\theta|$ and the dataset size D , i.e. the number of tokens to train on. Specifically, they estimate the loss function by:

$$L(|\theta|, D) \approx \frac{A}{|\theta|^\alpha} + \frac{B}{D^\beta} + E$$

where $E = 1.69$ is the entropy of natural language and is the theoretical limit of the loss function if using an infinitely large model trained on infinite data. The first term, with $A = 406.4$, $\alpha = 0.34$, captures the fact that a perfectly trained transformer with $|\theta|$ parameters underperforms the theoretical limit (i.e. the ideal generative process). The second term, with $B = 410.7$, $\beta = 0.28$,

captures the fact that the transformer is not trained to convergence, as only a finite number of optimisation steps occur on a sample of the dataset distribution. See [23] for more detail on how A, B, E, α, β were chosen. Furthermore, the FLOPs for a particular model size $|\theta|$ and dataset size D can be estimated via $\text{FLOPs}(|\theta|, D) \approx 6|\theta|D$ [26].

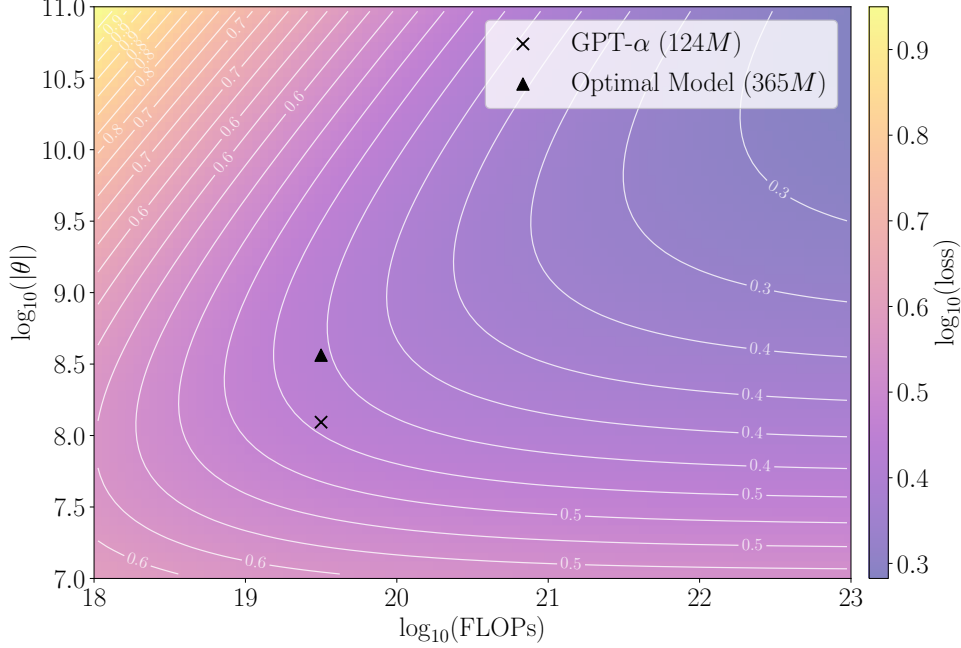


Figure D.1: Contour plot of loss as a function of model size and FLOPs budget. The optimal model for a compute budget of 4.19×10^{19} total FLOPs is shown and has a size of 422 million parameters.

Figures D.1, D.2 and D.3 show the output of these calculations when using a compute budget $C = 4.19 \times 10^{19}$. This compute budget was calculated using the MFU for GPT- α (38.85%) and the theoretical maximum FLOPs from $8 \times$ A100 SMX4 40GB GPUs running using BFLOAT16 for 12 hours. The calculations suggest that an optimal model size is 422 million parameters with dataset size of 16.6 billion tokens. This suggests that currently, GPT- α has $3.4 \times$ too few parameters and that it was trained on $2.4 \times$ too many tokens. Hence, a further improvement to GPT- α could be to scale its parameters and train it over less tokens according to this calculated optimum. GPT-2 124M and GPT-3 125M are even less compute-optimised models, having been trained on 100 billion and 300 billion tokens respectively.

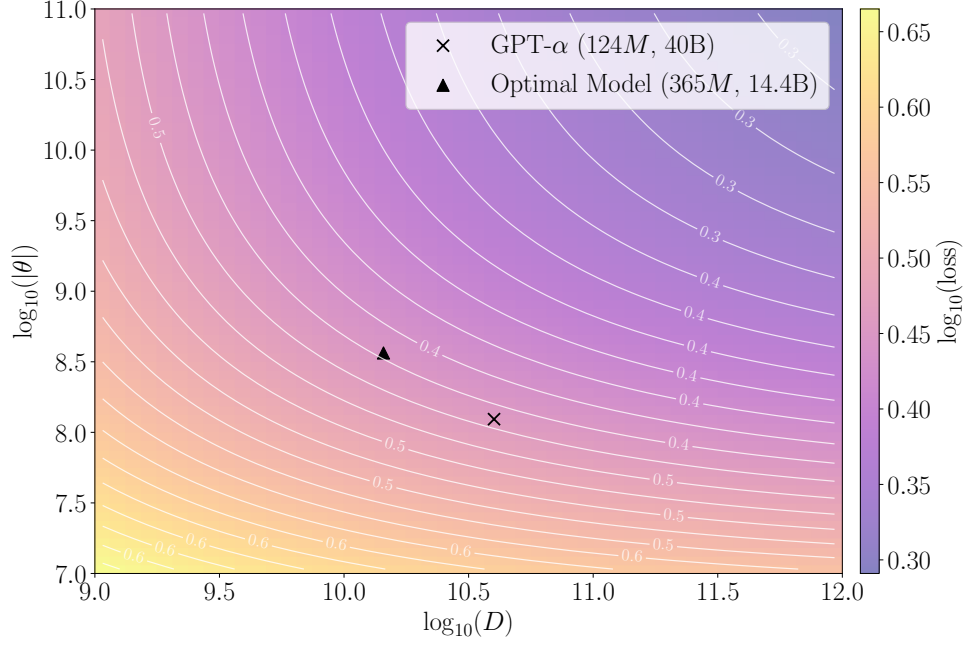


Figure D.2: Contour plot of loss as a function of model size and dataset size. The optimum model of 422 million parameters corresponds to a dataset size of 16.6 billion tokens.

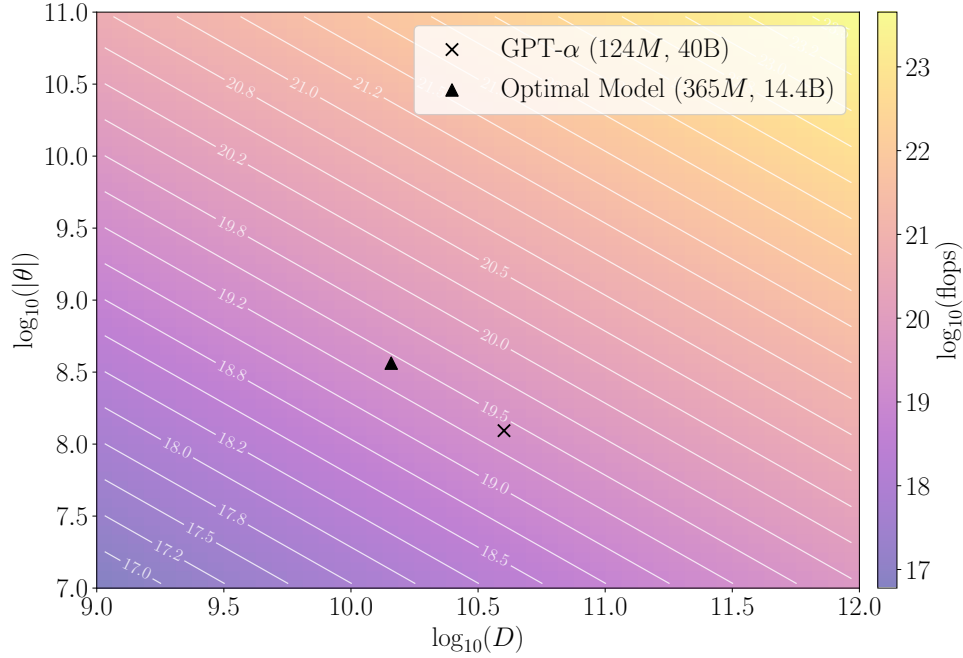


Figure D.3: Contour plot of compute (total FLOPs) as a function of model size and dataset size. The optimum model of 422 million parameters corresponds to a dataset size of 16.6 billion tokens.