Learning Notes

Yangyang Li
yangyang.li@northwestern.edu

Update on January 2, 2024

Contents

List of Figures

LIST OF FIGURES

List of Theorems

List of Definitions

LIST OF DEFINITIONS

Acronyms

BERT	Bidirectional Encoder Representations from Transformers 81
CGAN	Conditional Generative Adversarial Nets 68
DCGAN	Deep Convolutional GAN 68, 70
DDIM	Denoising Diffusion Implicit Models 78
DDM	Denoising Diffusion Models 75
GAN	Generative Adversarial Nets 4, 63, 66–68, 70, 84–91, 93, 94
GPT	Generative Pre-trained Transformer 84
GPU	Graphics Processing Unit 93
GRU	Gated Recurrent Unit 7, 74
KL	Kullback-Leibler 64, 65
LSTM	Long Short-Term Memory 7, 71, 73, 74
MAP	Maximum A Posteriori Estimation 6
MLE	Maximum Likelihood Estimation 5, 6
MuseGAN	Multi-track Sequential Generative Adversarial Networks 91
PDF	Probability Density Function 6
ProGAN	Progressive Growing Generative Adversarial Networks 84, 89, 91
RNN	Recurrent Neural Network 74
SAGAN	Self-Attention Generative Adversarial Networks 91
StyleGAN	Style-Based Generator Architecture for Generative Adversarial Networks 7, 89–91
VAE	Variational Autoencoder 65, 74, 91
ViT VQ-GAN	Vision Transformer Vector Quantized GAN 91
VQ-GAN	Vector Quantized GAN 91
VQ-VAE	Vector Quantized Variational Autoencoder 91
WGAN-GP	Wasserstein GAN with Gradient Penalty 7, 67, 68

Preface

Contents

0.1 Features of this template

> *TeX, stylized within the system as LaTeX, is a typesetting system which was designed and written by Donald Knuth and first released in 1978. TeX is a popular means of typesetting complex mathematical formulae; it has been noted as one of the most sophisticated digital typographical systems.*
>
> - Wikipedia

0.1.1 crossref

different styles of clickable definitions and theorems

- nameref: Gaussian distribution
- autoref: Definition A.1, **??**
- cref: Definition A.1,
- hyperref: Gaussian,

0.1.2 ToC (Table of Content)

- mini toc of sections at the beginning of each chapter
- list of theorems, definitions, figures
- the chapter titles are bi-directional linked

0.1.3 header and footer

fancyhdr

- right header: section name and link to the beginning of the section
- left header: chapter title and link to the beginning of the chapter
- footer: page number linked to ToC of the whole document

0.1.4 bib

- titles of reference is linked to the publisher webpage e.g., [Kit+02]
- backref (go to the page where the reference is cited) e.g., [Chi09]
- customized video entry in reference like in [Bab16]

0.1.5 preface, index, quote (epigraph) and appendix

index page at the end of this document...

Part I

Machine Learning

Chapter 1

Probability

Contents

1.1 Basic Concepts

Permutation is an arrangement of objects in which the order is important.

$$P(n,r) = \frac{n!}{(n-r)!}$$

Combination is an arrangement of objects in which the order is not important.

$$C(n,r) = \frac{P(n,r)}{r!} = \frac{n!}{r!\,(n-r)!}$$

in which $0 \leq r \leq n$.

1.2 Maximum Likelihood Estimation

$$\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N)^{\mathrm{T}}, \boldsymbol{x}_i = (x_{i1}, x_{i2}, \ldots, x_{ip})^{\mathrm{T}} \tag{1.1}$$

in which N is the number of samples, p is the number of features. The data is sampled from a distribution $p(\boldsymbol{x} \mid \theta)$, where θ is the parameter of the distribution.

For N i.i.d. samples, the likelihood function is $p(\boldsymbol{X} \mid \theta) = \prod_{i=1}^{N} p(\boldsymbol{x}_i \mid \theta))$

In order to get θ, we use Maximum Likelihood Estimation (MLE) to maximize the likelihood function.

$$\theta_{\texttt{MLE}} = \underset{\theta}{\operatorname{argmax}} \log p(\boldsymbol{X} \mid \theta) = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i \mid \theta) \tag{1.2}$$

1.3 Maximum A Posteriori Estimation

In Bayes' theorem, the θ is not a constant value, but $\theta \sim p(\theta)$. Hence,

$$p(\theta \mid \boldsymbol{X}) = \frac{p(\boldsymbol{X} \mid \theta)p(\theta)}{p(\boldsymbol{X})} = \frac{p(\boldsymbol{X} \mid \theta)p(\theta)}{\int\limits_{\theta} p(\boldsymbol{X} \mid \theta)p(\theta)d\theta} \tag{1.3}$$

In order to get θ, we use Maximum A Posteriori Estimation (MAP) to maximize the posterior function.

$$\theta_{\texttt{MAP}} = \underset{\theta}{\operatorname{argmax}}\, p(\theta \mid \boldsymbol{X}) = \underset{\theta}{\operatorname{argmax}}\, \frac{p(\boldsymbol{X} \mid \theta)p(\theta)}{p(\boldsymbol{X})} \tag{1.4}$$

After θ is estimated, then calculating $\frac{p(\boldsymbol{X} \mid \theta)\cdot p(\theta)}{\int\limits_{\theta} p(\boldsymbol{X} \mid \theta)p(\theta)d\theta}$ to get the posterior distribution. We can use the posterior distribution to predict the probability of a new sample $\boldsymbol{x}$.

$$p(x_{\texttt{new}} \mid \boldsymbol{X}) = \int\limits_{\theta} p(x_{\texttt{new}} \mid \theta) \cdot p(\theta \mid \boldsymbol{X})d\theta \tag{1.5}$$

1.4 Gaussian Distribution

Gaussian distribution is also called normal distribution.

$$\boldsymbol{\theta} = (\mu, \sigma^2), \quad \mu = \frac{1}{N}\sum_{i=1}^{N} x_i, \quad \sigma^2 = \frac{1}{N}\sum_{i=1}^{N}(x_i - \mu)^2 \tag{1.6}$$

For MLE,

$$\boldsymbol{\theta} = (\mu, \Sigma) = (\mu, \sigma^2), \quad \theta_{\texttt{MLE}} = \underset{\theta}{\operatorname{argmax}} \log p(\boldsymbol{X} \mid \theta) = \underset{\theta}{\operatorname{argmax}} \sum_{i=1}^{N} \log p(\boldsymbol{x}_i \mid \theta) \tag{1.7}$$

Generally, the Probability Density Function (PDF) of a Gaussian distribution is:

$$p(x \mid \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^p \det(\Sigma)}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu)^{\mathrm{T}} \Sigma^{-1} (\boldsymbol{x} - \mu)\right) \tag{1.8}$$

in which μ is the mean vector, Σ is the covariance matrix, $\det$ is the determinant of matrix. $\det$ is the product of all eigenvalues of a matrix.

Hence,

$$\log p(\boldsymbol{X} \mid \theta) = \sum_{i=1}^{N} \log p(x_i \mid \theta) = \sum_{i=1}^{N} \log \frac{1}{\sqrt{(2\pi)^p \det(\Sigma)}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \mu)^{\mathrm{T}} \Sigma^{-1} (\boldsymbol{x} - \mu)\right) \tag{1.9}$$

Let's only consider 1 dimension case for brevity, then

$$\log p(\boldsymbol{X} \mid \theta) = \sum_{i=1}^{N} \log p(x_i \mid \theta) = \sum_{i=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\frac{(x - \mu)^2}{\sigma^2}\right) \tag{1.10}$$

Let's get the optimal value for μ,

$$\mu_{\texttt{MLE}} = \underset{\mu}{\operatorname{argmax}} \log p(\boldsymbol{X} \mid \theta) = \underset{\mu}{\operatorname{argmin}} \sum_{i=1}^{N} \frac{1}{2} (x_i - \mu)^2 \tag{1.11}$$

So,

$$\frac{\partial \log p(\boldsymbol{X} \mid \theta)}{\partial \mu} = \sum_{i=1}^{N} (\mu - x_i) = 0 \rightarrow \mu_{\texttt{MLE}} = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{1.12}$$

Let's get the optimal value for σ^2,

$$\begin{aligned}
\sigma_{\texttt{MLE}} &= \underset{\sigma}{\operatorname{argmax}} \log p(\boldsymbol{X} \mid \theta) \\
&= \underset{\sigma}{\operatorname{argmax}} \sum_{i=1}^{N} \log \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2} \frac{(x-\mu)^2}{\sigma^2}\right) \\
&= \underset{\sigma}{\operatorname{argmax}} \sum_{i=1}^{N} \left[-\log \sqrt{2\pi\sigma^2} - \frac{(x-\mu)^2}{2\sigma^2}\right] \\
&= \underset{\sigma}{\operatorname{argmin}} \sum_{i=1}^{N} \left[\log \sigma + \frac{(x-\mu)^2}{2\sigma^2}\right]
\end{aligned}$$

Hence,

$$\frac{\partial}{\partial \sigma} \sum_{i=1}^{N} \left[\log \sigma + \frac{(x_i-\mu)^2}{2\sigma^2}\right] = 0 \rightarrow \sigma^2_{\texttt{MLE}} = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2 \tag{1.13}$$

$\mathbb{E}_D[\mu_{\texttt{MLE}}]$ is unbaised.

$$\mathbb{E}_D[\mu_{\texttt{MLE}}] = \mathbb{E}_D\left[\frac{1}{N} \sum_{i=1}^{N} x_i\right] = \frac{1}{N} \sum_{i=1}^{N} \mathbb{E}_D[x_i] = \frac{1}{N} \sum_{i=1}^{N} \mu = \mu \tag{1.14}$$

However, $\mathbb{E}_D\left[\sigma^2_{\texttt{MLE}}\right]$ is biased.

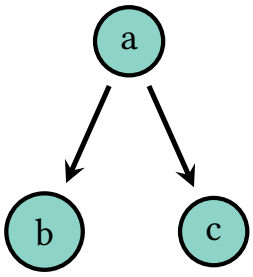

Figure 1.1: A simple Bayesian network.

$$\mathbb{E}_D\left[\sigma^2_{\texttt{MLE}}\right] = \mathbb{E}_D\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \mu_{\texttt{MLE}}\right)^2\right] \tag{1.15}$$

$$= \mathbb{E}_D\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i - \mu_{\texttt{MLE}}\right)^2\right] \tag{1.16}$$

$$= \mathbb{E}_D\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i^2 - 2x_i\mu_{\texttt{MLE}} + \mu^2_{\texttt{MLE}}\right)\right] = \mathbb{E}_D\left[\sum_{i=1}^{N}x_i^2 - 2\frac{1}{N}\sum_{i=1}^{N}x_i\mu_{\texttt{MLE}} + \mu^2_{\texttt{MLE}}\right] \tag{1.17}$$

$$= \mathbb{E}_D\left[\frac{1}{N}\sum_{i=1}^{N}\left(x_i^2 - \mu^2\right) + \mu^2 - \mu^2_{\texttt{MLE}}\right] \tag{1.18}$$

$$= \sigma^2 - \mathbb{E}_D\left[\mu^2_{\texttt{MLE}} - \mu^2\right] \tag{1.19}$$

$$= \sigma^2 - \left(\mathbb{E}_D\left[\mu^2_{\texttt{MLE}}\right] - \mathbb{E}_D\left[\mu^2_{\texttt{MLE}}\right]\right) \tag{1.20}$$

$$= \sigma^2 - \texttt{Var}\left[\mu_{\texttt{MLE}}\right] = \sigma^2 - \texttt{Var}\left[\frac{1}{N}\sum_{i=1}^{N}x_i\right] \tag{1.21}$$

$$= \sigma^2 - \frac{1}{N^2}\sum_{i=1}^{N}\texttt{Var}\left[x_i\right] = \frac{N-1}{N}\sigma^2 \tag{1.22}$$

$$\tag{1.23}$$

1.5 Bayesian Network

1.6 Probability Graph

this section is not finished yet. Need to be reviewed p54

1.6.1 Variables Elimination

1.6.2 Belief propagation

Belief propagation is mainly used for tree data structure, and equals Section 1.6.1 with caching.

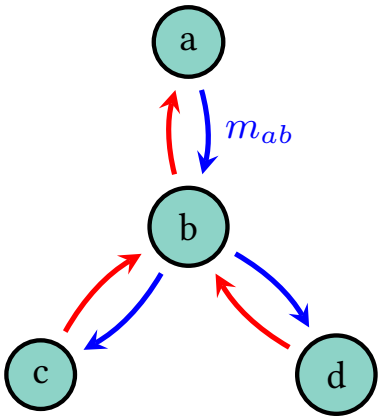

Figure 1.2: Belief propagation.

1.6.3 Max-product Algorithm

1.6.4 Factor Graph

1.7 Expectation Maximum

$$\Theta^{(t+1)} = \underset{\Theta}{\operatorname{argmax}} \int_Z \log P(x, z \mid \theta) \cdot P(z \mid x, \Theta^{(t)}) \, dz$$

continue on p60

1.8 Gaussian Mixture Model

$$Q\left(\Theta, \Theta^{(t)}\right) = \int_Z \log P(x, z \mid \theta) \cdot P(z \mid x, \Theta^{(t)}) \, dz$$

1.9 Hidden Markov Model

Chapter 2

Machine Learning Concepts

2.1 Complexity of matrix multiply

Assume $\mathbf{A}$ is $m \times n$ and $\mathbf{B}$ is $n \times p$. The naive algorithm takes $O(mnp)$ time.

2.2 Representation of matrix

Row major, col major, stride

How to calculate dimension of convolutional layer

$$\left\lfloor \frac{n_h - k_h + p_h + s_h}{s_h} \right\rfloor \times \left\lfloor \frac{n_w - k_w + p_w + s_w}{s_w} \right\rfloor$$

In which n_h means height of input, k_h means height of filter, p_h means padding of height, s_h means stride of height. So as n_w, k_w, p_w, s_w.

2.3 Training Tricks

Batch normalization

$$\mathrm{BN}(\boldsymbol{x}) = \gamma \odot \frac{\boldsymbol{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \beta$$

Batch normalization is a technique that drastically reduces this problem. The solution is surprisingly simple. During training, a batch normalization layer calculates the mean and standard deviation of each of its input channels across the batch and normalizes by subtracting the mean and dividing by the standard deviation. There are then two learned parameters for each channel, the scale (gamma) and shift (beta) Fig. 2.1. The output is simply the normalized input, scaled by gamma and shifted by beta [Fos22].

When it comes to prediction, we may only want to predict a single observation, so there is no batch over which to calculate the mean and standard deviation. To get around this problem, during training a batch normalization layer also calculates the moving average of the mean and standard deviation of each channel and stores this value as part of the layer to use at test time [Fos22].

How many parameters are contained within a batch normalization layer? For every channel in the preceding layer, two weights need to be learned: the scale (gamma) and shift (beta). These are the trainable parameters. The moving average and standard deviation also need to be calculated for each channel, but since they are derived from the data passing through the layer rather than trained through backpropagation, they are called nontrainable parameters. In total, this gives four parameters for each channel in the preceding layer, where two are trainable and two are nontrainable [Fos22].

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Figure 2.1: Batch Normalization

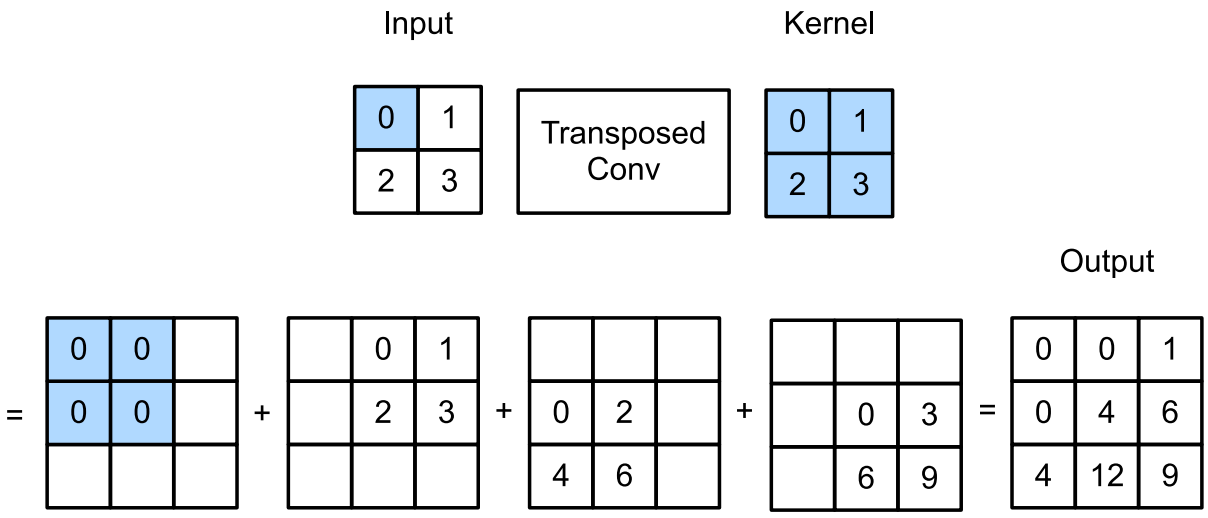

Figure 2.2: Transposed Convolution [Zha+23, Chapter 14.10]

Transposed Convolution Ignoring channels for now, let's begin with the basic transposed convolution operation with stride of 1 and no padding. Suppose that we are given a input tensor $n_h \times n_w$ and a kernel $k_h \times k_w$. Sliding the kernel window with stride of 1 for n_w times in each row and n_h times in each column yields a total of $n_h n_w$ intermediate results. Each intermediate result is a $(n_h + k_h - 1) \times (n_w + k_w - 1)$ tensor that are initialized as zeros. To compute each intermediate tensor, each element in the input tensor is multiplied by the kernel so that the resulting tensor $k_h \times k_w$ replaces a portion in each intermediate tensor. Note that the position of the replaced portion in each intermediate tensor corresponds to the position of the element in the input tensor used for the computation. In the end, all the intermediate results are summed over to produce the output [Zha+23, Chapter 14.10].

Different from in the regular convolution where padding is applied to input, it is applied to output in the transposed convolution. For example, when specifying the padding number on either side of the height and width as 1, the first and last rows and columns will be removed from the transposed convolution output.

Consider implementing the convolution by multiplying matrices. Given an input vector $\mathbf{x}$ and a weight matrix $\mathbf{W}$, the forward propagation function of the convolution can be implemented by multiplying its input with the weight matrix and outputting a vector $y = \mathbf{W}\mathbf{x}$. Since backpropagation follows the chain rule and $\nabla_x \mathbf{y} = \mathbf{W}^\top$, the backpropagation function of the convolution can be implemented by multiplying its input with the transposed weight matrix $\mathbf{W}^\top$. Therefore, the transposed convolutional layer can just exchange the forward propagation function and the backpropagation function of the convolutional layer: its forward propagation and backpropagation functions multiply their input vector with $\mathbf{W}^\top$ and $\mathbf{W}$, respectively [Zha+23].

Softmax vs LogSoftmax The softmax function is used to convert a vector of values to a probability distribution. The probabilities of each value are proportional to the exponential of the original values. The softmax function for a vector $\mathbf{z}$ is defined as:

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

where $\mathbf{z}$ is the input vector, K is the number of classes (or the length of the vector), z_i is the i^{th} element of the input vector, and e is the base of the natural logarithm.

The *log_softmax* function is essentially the logarithm of the softmax function. It's defined as:

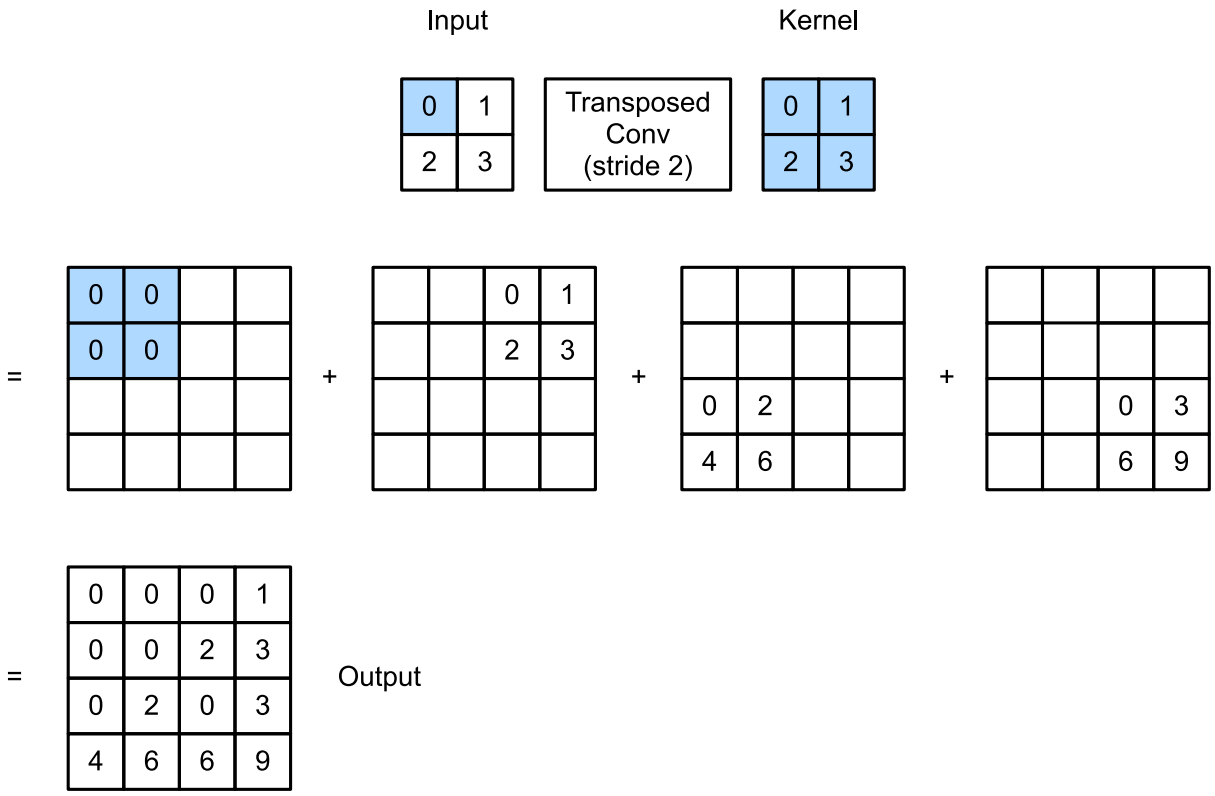

Figure 2.3: Transposed convolution with a kernel with stride of 2×2. The shaded portions are a portion of an intermediate tensor as well as the input and kernel tensor elements used for the computation. [Zha+23]

$$\log\_\text{softmax}(\mathbf{z})_i = \log\left(\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}\right)$$

Using the property of logarithms that $\log(a/b) = \log(a) - \log(b)$, this can be simplified to:

$$\log\_\text{softmax}(\mathbf{z})_i = z_i - \log\left(\sum_{j=1}^{K} e^{z_j}\right)$$

When you're training a model using a loss function like Negative Log-Likelihood (NLL), the combination of *log_softmax* and NLL can be computed more efficiently and with better numerical stability. The NLL for a true class c and predicted log probabilities $\log(p_i)$ is:

$$\text{NLL} = -\log(p_c)$$

When you use the output from *log_softmax* in the NLL loss, you get:

$$\text{NLL} = -\log\left(\frac{e^{z_c}}{\sum_{j=1}^{K} e^{z_j}}\right)$$

which simplifies to:

$$\text{NLL} = -z_c + \log\left(\sum_{j=1}^{K} e^{z_j}\right)$$

This form is more numerically stable and efficient for computers to calculate, especially for large vectors where values of e^{z_j} can become very large or small, leading to floating-point precision issues.

In summary, using *log_softmax* provides a more numerically stable and computationally efficient way to handle the softmax outputs, especially when combined with loss functions like NLL during model training.

Temperature in Softmax Temperature is a hyperparameter that controls the steepness of the softmax function.

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i/T}}{\sum_{j=1}^{K} e^{z_j/T}}.$$

Part II

Algorithm and Data Structure

Chapter 3

Algorithm

Contents

3.1 Graph

Part III

Programming

Chapter 4

Leetcode

Contents

4.1 Problems Sets

- https://leetcode.com/list/rab78cw1/https://www.techinterviewhandbook.org/grind75?weeks=8
- dfs recursive = bfs iterative (use stack)
- two pointer
- check size (is_empty, compare, larger, .etc.)
- hash map (fixed array)

4.2 Tips

- postorder: LRN
- preorder: NLR
- inorder: LNR
- virtual head pointers
- two pointers (left and right, fast and slow)
- BST (value of left child is smaller, that of right child is larger)
- Backtracking (terminated option, available paths, current paths, if has
- duplicates(available paths or result))
- BFS (queue;while;for)
- sliding windows (tow pointers) for substring problems
- best-time-to-buy-and-sell-template.cpp
- (p + m - 1) / m equal to ceil(p / m)

- be careful about int overflow and underflow
- binary search problem: 1. left, right, f(x) increase/decrease, target

4.3 Problems

4.3.1 1 Two Sum

- hash map

4.3.2 20 valid parentheses

- stack

4.3.3 21 Merge Two Sorted List

- recursive

4.3.4 121 Best time to buy and sell stock

- record value

4.3.5 226 Invert Binary Tree

- DFS recursive
- BFS iterative via stack

4.3.6 242 Valid Anagram

- count freq then verify freq

4.3.7 704 Binary Search

- two pointers

4.3.8 733 Flood Fill

- graph matrix
- DFS via recursive or BFS via iterative

4.3.9 235 lowest-common-ancestor-of-a-binary-search-tree

- Attributes of BST, the value of left subtree is less than root, that of left larger than root.
- Recursive
- iterative

4.3.10 110 balanced-binary-tree

- recursive DFS
- use -1 means unbalanced

4.3.11 141 linked-list-cycle

- two points fast and slow
- check if they meet again

4.3.12 232 implement-queue-using-stacks

- Two stacks, input and output
- move when output is empty
- amortized cost for each operation is O(1).

4.3.13 278 first-bad-version

- Binary Search
- use val directly instead of index

4.3.14 383 ransom-note

- hash map (fixed array)
- check size first

4.3.15 70 climbing-stairs

- Fibonacci like
- recursive (from top to down, memorization)
- iterative (from bottom to top, space O(1))

4.3.16 409 longest-palindrome

- use even char freq
- cal odd char freq $(\text{s.size}() - \text{odd} + (\text{odd} > 0))$

4.3.17 206 reverse-linked-list

- recursive
- iterative (three pointes (prev, curr,next))

4.3.18 169 majority-element

- moorse vote

4.3.19 67 add-binary

- carry
- ASCII to int, vicewise

4.3.20 543 diameter-of-binary-tree

- postorder traversal

4.3.21 876 middle-of-the-linked-list

- fast and slow pointers
- get n and n/2 at same time

4.3.22 104 maximum-depth-of-binary-tree

- DFS recursive
- postorder

4.3.23 217 contains-duplicate

- Hash set
- unorder_map to hash map
- map to BST

4.3.24 53 maximum-subarray

- DP
- use subproblem to fix global problem

4.3.25 57 insert-interval

- three situations
- not—overlap—not

4.3.26 542 01-matrix

- DP first left top, then bottom, right
- BFS, first 0, then unseen

4.3.27 973 k-closest-points-to-origin

- sort/n_element/partial_sort
- max-heap
- randomized quicksort

4.3.28 3 longest-substring-without-repeating-characters

- slide window (two points)

4.3.29 15 3sum

- sort
- two points

4.3.30 102 binary-tree-level-order-traversal

- stack/queue BFS
- DFS recursive / keep level

4.3.31 133 clone-graph

- DFS
- Amazing

4.3.32 207 course-schedule

- https://www.geeksforgeeks.org/kahns-algorithm-vs-dfs-approach-a-comparative-analysis/
- BFS kahn's algorithm topological sort
- DFS better for larger V
- topological sort

4.3.33 322 coin-change

- DP

4.3.34 238 product-of-array-except-self

- prefix
- suffix
- space O(1)

4.3.35 155 min-stack

- keep value and min at the same time

4.3.36 98 validate-binary-search-tree

- Recursive

4.3.37 200 number-of-islands

- find 1 and DFS clear 1

4.3.38 33 search-in-rotated-sorted-array

- Binary Search
- check if mid and target are in same side
- if not same side assume one side is inf/-inf value let left/right advanced

4.3.39 39 combination-sum

- Backtracking

4.3.40 86.partition-list

- dummy pointers
- two pointers

4.3.41 23.merge-k-sorted-lists

- heap/priority_queue
- merge two lists iteratively

4.3.42 142.linked-list-cycle-ii.cpp

- two pointers

4.3.43 160.intersection-of-two-linked-lists.cpp

- contact two lists
- make sure two lists have same distance from the end

4.3.44 26.remove-duplicates-from-sorted-array

- fast and slow pointers

4.3.45 27.remove-element.cpp

- fast and slow pointers

4.3.46 283.move-zeroes.cpp

- fast and slow pointers
- snowball

4.3.47 5.longest-palindromic-substring

- dp
- two pointers (expand from center/narrow from both sides)

4.3.48 669.trim-a-binary-search-tree.cpp

- BST (value of left child is smaller, that of right child is larger)
- postorder

4.3.49 124.binary-tree-maximum-path-sum.cpp

- postorder
- discard nodes with negative value

4.3.50 46.permutations.cpp

- Backtracking

4.3.51 51.n-queens.cpp

- Backtracking(current path, options, terminated option)

4.3.52 78.subsets.cpp

- backtracking

4.3.53 77.combinations.cpp

- backtracking

4.3.54 90.subsets-ii.cpp

- backtracking

4.3.55 40.combination-sum-ii.cpp

- backtracking

4.3.56 47.permutations-ii.cpp

- backtracking

4.3.57 111.minimum-depth-of-binary-tree.cpp

- BFS

4.3.58 752.open-the-lock.cpp

- BFS

4.3.59 76.minimum-window-substring.cpp

- sliding windows

4.3.60 438.find-all-anagrams-in-a-string.cpp

- sliding windows

4.3.61 337.house-robber-iii.cpp

- 213.house-robber-ii.cpp
- 198.house-robber.cpp
- DP
- recursive with memorization
- DP table -¿ optimize table

4.3.62 116. Populating Next Right Pointers in Each Node

- Three children tree traverse
- BFS $O(N)$ space $O(1)$

Chapter 5

C++

Contents

5.1 Code Snippets

5.1.1 Random Number Generation

```
#include <algorithm>
#include <iostream>
#include <iterator>
#include <random>

int main() {
  std::random_device rd;
  std::mt19937 rng(rd());
  std::uniform_int_distribution<int> dist6(1, 6);
  std::generate_n(std::ostream_iterator<int>(std::cout, " "), 10,
                  [&dist6, &rng]() { return dist6(rng); });
  return 0;
}
```

5.1.2 Quick Sort

```
#include <vector>
using std::vector;

class Solution {
public:
  void sort(vector<int> &nums, int low, int high) {
    if (low >= high)
      return;

    int p = partition(nums, low, high);
```

```
    sort(nums, low, p - 1); // Changed p to p-1
    sort(nums, p + 1, high);
  }

  int partition(vector<int> &nums, int low, int high) {
    int pivot = low, l = pivot + 1, r = high;

    while (l <= r) {
      if (nums[l] < nums[pivot])
        ++l;
      else if (nums[r] >= nums[pivot])
        --r;
      else
        std::swap(nums[l], nums[r]);
    }
    std::swap(nums[pivot], nums[r]);
    return r;
  }

  void shuffle(vector<int> &nums) {
    std::srand((unsigned)time(nullptr));
    int n = nums.size();

    for (int i = 0; i < n; ++i) {
      int r = i + rand() % (n - i);
      std::swap(nums[i], nums[r]);
    }
  }

  vector<int> sortArray(vector<int> &nums) {
    shuffle(nums);                  // Shuffle the array before sorting
    sort(nums, 0, nums.size() - 1); // Changed nums.size() to nums.size() - 1
    return nums;
  }
};
```

On average, the algorithm takes $O(n \log n)$ time. In the worst case, it takes $O(n^2)$ time. We shuffle the array before sorting to avoid the worst case.

5.2 Code Minning

5.2.1 Deep Learning

https://ensmallen.org/docs.html
https://github.com/mlpack/mlpack/blob/master/doc/quickstart/cpp.md https://github.com/flashlight/flashlight?tab=readme-ov-file#build-options
https://github.com/pytorch/pytorch
https://mp.weixin.qq.com/s/rx7SPYr-sEOz_GOYRfSDOw

Chapter 6

Rust

Chapter 7

Python

Contents

7.1 Deep Learning

7.1.1 Print Attention Score from Decoder of Transformer

```
import numpy as np
from IPython.display import display, HTML

def print_probs(info, vocab, top_k=5):
    for i in info:
        highlighted_text = []
        for word, att_score in zip(
            i["prompt"].split(), np.mean(i["atts"], axis=0)
        ):
            highlighted_text.append(
                '<span style="background-color:rgba(135,206,250,'
                + str(att_score / max(np.mean(i["atts"], axis=0)))
                + ');">'
                + word
                + "</span>"
            )
        highlighted_text = " ".join(highlighted_text)
        display(HTML(highlighted_text))

        word_probs = i["word_probs"]
        p_sorted = np.sort(word_probs)[::-1][:top_k]
        i_sorted = np.argsort(word_probs)[::-1][:top_k]
        for p, i in zip(p_sorted, i_sorted):
            print(f"{vocab[i]}:   \t{np.round(100*p,2)}%")
        print("--------\n")
```

7.1.2 Transformer

Part IV

Research

Chapter 8

Paper Writing

Contents

8.1 Resources

- https://github.com/MLNLP-World/Paper-Writing-Tips
- https://github.com/dspinellis/latex-advice
- https://github.com/guanyingc/latex_paper_writing_tips
- https://github.com/yzy1996/English-Writing
- https://github.com/guanyingc/cv_rebuttal_template
- https://www.overleaf.com/gallery/tagged/tikz/page
- https://color.adobe.com/zh/explore
- https://github.com/acmi-lab/cmu-10717-the-art-of-the-paper/tree/main/resources
- https://github.com/daveshap/ChatGPT_Custom_Instructions/blob/main/academic_copy_reviser.md
- https://github.com/sylvainhalle/textidote?tab=readme-ov-file
- https://github.com/egeerardyn/awesome-LaTeX
- https://texfaq.org
- https://github.com/xinychen/awesome-latex-drawing
- https://pudding.cool/2022/02/plain

Table 8.1: Writing Tools

Tool	URL
Latex Table Generator	tablesgenerator.com
Latex Equation Editor	https://www.codecogs.com/latex/eqneditor.php
Latex Equation and Figure Editor	https://www.mathcha.io/editor
Equation Image to Latex	https://mathpix.com/
Figure Inspiration	https://echarts.apache.org/examples/
Figure Creation	https://altair-viz.github.io/
Online Figure Editor	https://www.tldraw.com/
Identify Font	https://www.fontsquirrel.com/matcherator
Identify Color	https://imagecolorpicker.com/
Identify Math Symbol	https://detexify.kirelabs.org/classify.html
Better Labeling Tool	https://labelstud.io/
Fix Wrong Citation	https://github.com/yuchenlin/rebiber
Have a good name for model	http://acronymify.com/
Deadline Reminder	https://aideadlin.es/?sub=ML,CV,NLP
Great Paper/Model Info	https://paperswithcode.com/
Same Meaning Words	https://www.thesaurus.com/
Compare Same Meaning Words	https://wikidiff.com/neglect/omit
Common Words for Academic Writing	http://www.esoda.org/
Doi to Bib	https://www.doi2bib.org/
Figure Gallery of Vega	https://vega.github.io/vega-lite/examples/
Figure Gallery of Graphviz	https://graphviz.org/gallery/
UML Diagram	https://plantuml.com/
Color Picker	https://colorbrewer2.org
Abstract/Title Generator	https://x.writefull.com/
Online BibTeX Formatter	https://flamingtempura.github.io/bibtex-tidy

- https://www.youtube.com/watch?v=VK51E3gHENc&ab_channel=MicrosoftResearch
- https://www.cs.jhu.edu/~jason/advice/write-the-paper-first.html
- https://github.com/yzy1996/English-Writing

8.2 Writing Tools

8.3 Writing Tips

Your draft can describe your motivation, formal problem, model, algorithms, and experiments before you actually build anything. (Ideally, it will also explain why you did it this way rather than some other way, and point out gaps that remain for future work.) By showing others the draft at this stage, you'll get important feedback before you invest time in the "wrong" work.

If you run into trouble while doing the work, then I may have difficulty diagnosing or even understanding the problem you are facing. We may waste a whole meeting just getting aligned. But if you can show me a precise writeup of what you're doing, then I can be much more helpful. In general, meetings are very productive when we have a concrete document in front of us.

Of course, there are many kinds of concrete documents. You could instead write up the notes from our (usually extensive) discussions as private documents for further discussion. Still, writing up in the form of

a final paper makes you (1) integrate everything in one place, (2) decide which ideas will be made central for this paper, and (3) focus on the coherence and impact of the end product. (Additional discussion and brainstorming can still go into the document—those subsections can be cut or moved to an appendix for the conference paper, but retained for a longer version as a tech report, journal article, or dissertation.)[1]

What needs to be done? Assuming the idea is indeed a good one, then writing a draft makes you sharpen the message of the paper. Then you can figure out what work needs to be done to support exactly that message.

- Your introduction will make some claims. Often you will realize that some interesting additional experiment would really test those claims.
- Writing the literature review will help you design your experiments. It may influence what datasets you use, what comparisons you do, and what you are trying to prove you can do that other people can't. However, don't write the lit review first. Write out your own ideas before comparing them with the literature. This increases the chance that you'll find new angles on the problem.
- Writing the experimental section is possible even before you've done the experiments. Explain the full experimental design. Make empty result tables with row and column headers and explanatory captions. Make empty graphs with axes and captions. The actual results will be missing, but it will be clear what work is necessary. If you are having trouble switching your brain from experiments to writing, just write up your current experiments first. Although writing an intro will also help, so you can explain why you are doing the experiments.

Honest writing may lead you to realize that proving your point requires more work than you'd thought (which is why you'd better write early).

How to give a wonderful talk?

- What is the question?
- Why is it interesting?
- How do you propose to answer it?

- e.g. means "for example".
- i.e. means "that is".
- et al. means "and others of the same kind".
- etc. means "and others", do not specify people.

Avoid extreme meaning words

1. Don't spend too much time reading other people's research, waiting for inspiration to strike you.
   (a) Reading research should be a regular part of any economist's routine.
   (b) But it is not a substitute for doing your own.
2. Don't set the bar too high. You don't need to win the Nobel.
3. But don't set the bar too low!
   (a) Be wary of picking a topic that is of interest to a small number of people (e.g. you and your adviser).

[1] https://www.cs.jhu.edu/~jason/advice/write-the-paper-first.html

Table 8.2: Alternative Terms for Common Words

Original Term	Alternative Terms
obvious	straightforward
always	usually, generally, often
never	seldom, rare
avoid, eliminate	reduce, mitigate, alleviate, relieve
a lot of	much, many
do (verb)	perform, conduct, carry out
big	large, significant
like	such as, for example
think	believe, consider, argue, claim, suggest
talk	discuss, describe, present, show
look at	examine, investigate, explore, study
get	obtain, acquire, receive, gain
keep	maintain, retain, preserve
climb	increase, rise, grow, improve, ascend
really	very, extremely, highly

(b) In particular, think about topics that will interest those beyond this island

4. Your ideas and results won't sell themselves.
5. How you communicate your work is of crucial importance.
6. There is no point in having an interesting piece of research that nobody understands or sees the point of.
7. Many economists think of themselves as primarily experts in technical methods: Econometrics, economic theory, data expertise.
8. This "white coat" mentality—that we are mainly scientists who then do a write-up of our results—is deeply wrong.
9. Writing is an essential part of the research process, not a last-minute thing to be rushed.
10. Do not wait to write
11. Identify your key idea explicitly
12. Tell a story paper structure
13. Nail your contributions to the mast introduction
14. Related work
15. Put your readers first the main body
16. Listen to your readers

Latex Writing Tips

- use `\usepackage{microtype}` to improve the appearance of the text.
- use `\tablename~\ref{tab}` to reference table.
- use `\figurename~\ref{fig}` to reference figure.

- `Eq.~\eqref{eq1}`
- comments from different people can be distinguished by different colors. `\newcommand{\yl}[1]{{\color{blue}{[`
- use `\newcommand{\method}{ABC\xspace}` to define model name or method
- mathematic notation https://www.deeplearningbook.org/contents/notation.html

8.4 PhD Theses Tips

The PhD Thesis Tips from https://www.karlwhelan.com/Teaching/PhD/phd-writing-talk.pdf
What makes a good thesis? Forget about objectivity—beauty is in the eye of the beholder. A good thesis is one that readers think is good. So, you need to explain well what you are doing to somebody else. Your ideas and results won't sell themselves. How you communicate your work is of crucial importance. There is no point in having an interesting piece of research that nobody understands or sees the point of. The "white coat" mentality—that we are mainly scientists who then do a write-up of our results—is misguided. Writing is an essential part of the research process, not a last-minute thing to be rushed. This is particularly true of PhD theses which are read very carefully by externs, who are hoping that you have explained what you have done in a clear fashion.

Writing Skills: More Important Than You Think What makes a good thesis? Forget about objectivity—beauty is in the eye of the beholder. A good thesis is one that readers think is good. So, you need to explain well what you are doing to somebody else Your ideas and results won't sell themselves. How you communicate your work is of crucial importance. There is no point in having an interesting piece of research that nobody understands or sees the point of. The "white coat" mentality—that we are mainly scientists who then do a write-up of our results—is misguided. Writing is an essential part of the research process, not a last-minute thing to be rushed. This is particularly true of PhD theses which are read very carefully by externs, who are hoping that you have explained what you have done in a clear fashion.

Start by Avoiding Very Bad Writing People can be quite sensitive about their writing. If you are a native English speaker and you have a degree, you probably think you know how to write and communicate well. Chances are, you might be wrong. Most MA graduates and even many professional PhD-qualified economists write very poorly. Good writing is hard to define. Bad writing is easy to spot. A badly-written thesis will have:

1. Mis-spelled words.
2. Missing words.
3. Sentences that don't make sense or aren't proper sentences.
4. Poor use of punctuation – full stops, commas etc.

This annoys the reader because it makes things harder to read (and understand) but also because it's so easy to prevent. It suggests you didn't take the time to be careful.

What To Do About It? Read, re-read, edit, and re-edit. And do this as you go along. Read and edit after you've written a page or so. This can correct most of the common errors of style, grammar, and spelling that occur in the writing process. Most of you actually do know what a sentence is. In addition to catching typos, a quick re-read and edit allows you to check that what you've written gets across what you've been trying to say. Sometimes you can think you've made a point clearly but then you read what you've written and it's not so good. Read your stuff aloud or slowly to yourself. Does it sound right? Are you writing proper sentences? Are you over-using jargon or certain particular phrases? Use spell checks but don't rely on them. Grammar.

There are rules about how to use commas, colons, semi-colons, full stops, about what defines a sentence. Try to learn them.

8.5 Abstract

8.6 Introduction

1. Introductions are crucial because they set up the reader to understand what your topic is and what you are going to do in your paper.
2. Quickly explain two things:
   (a) Why is the topic of your paper interesting?
   (b) What did YOU do? What is YOUR contribution? A new question? An existing question but new methodology? Existing question, existing methodology, new data (e.g. no previous Irish application)?
3. Be willing to give an outline of what your results are but don't get into too many details.
4. Because of its importance, spend a lot of time on the introduction.
5. But don't make it too long. Three pages is a limit. Two is better.
6. I often start writing the introduction as soon I have some results and then keep adjusting it as the paper evolves.
7. Conclusions, on the other hand, should be kept short and to the point. Don't repeat lots of stuff from the intro.

8.7 Literature Reviews

1. You need to explain your contribution.
2. So it needs to be put in context.
3. This will require discussion of previous studies in this area.
4. Remember, though, the purpose is to set up your contribution, and distinguish it from previous work.
5. Don't simply provide a long list of separate descriptions of weakly related studies. (X (2007) did this. Y(2009) did that ...)
6. Grouping studies together by type may be a better way of explaining it than listing lots of separate individual studies.
7. A well-focused literature review is usually a useful component of a PhD thesis. It shows externs and your supervisor that you have read the literature.
8. However, it is not essential that articles submitted for publication have a literature review. You may be able to summarise the existing literature in your introduction or work it into your opening section that sets up what you are doing.

8.8 Method

8.9 Result

8.10 Conclusion

Part V

Book Reading

Chapter 9

Dive into Deep Learning

Contents

9.1 Linear Neural Networks

9.1.1 Exercise 3.1.6

Exercise 1: Let's solve this analytically.

To minimize the sum of squared differences, we take the derivative of the function with respect to (b), set it equal to zero, and solve for (b). The function to minimize is:

$$f(b) = \sum_{i=1}^{n} (x_i - b)^2$$

Taking the derivative with respect to (b) gives:

$$f'(b) = \sum_{i=1}^{n} -2\,(x_i - b)$$

Setting this equal to zero and solving for (b) gives:

$$0 = \sum_{i=1}^{n} -2(x_i - b)$$

Solving for (b) gives:

$$b = \frac{1}{n} \sum_{i=1}^{n} x_i$$

So the value of (b) that minimizes the sum of squared differences is the mean of the x_i.

This problem and its solution relate to the normal distribution because the sum of squared differences is the basis for the maximum likelihood estimation of the mean of a normal distribution. In other words, the mean of a normal distribution is the value that maximizes the likelihood of the observed data, which is equivalent to minimizing the sum of squared differences from the mean.

If we change the loss function to the sum of absolute differences, the problem becomes finding the median of the x_i. This is because the median is the value that minimizes the sum of absolute differences. Unlike the

mean, the median is not sensitive to extreme values, so it is a more robust measure of central tendency when there are outliers in the data.

Exercise 2: An affine function is a function composed of a linear transformation and a translation. In the context of machine learning, an affine function often takes the form $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b)$, where $\mathbf{x}$ is an input vector, $\mathbf{w}$ is a weight vector, and (b) is a bias term.

We can show that this affine function is equivalent to a linear function on the augmented vector $(\mathbf{x}, 1)$ by considering a new weight vector $\mathbf{w}' = (\mathbf{w}, b)$ and an augmented input vector $\mathbf{x}' = (\mathbf{x}, 1)$. Then the affine function can be written as a linear function:

$$f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b = \mathbf{x}'^\top \mathbf{w}'$$

Here's the proof:

The original affine function is $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{w} + b$.

We define $\mathbf{x}' = (\mathbf{x}, 1))and(\mathbf{w}' = (\mathbf{w}, b)$.

The dot product $\mathbf{x}'^\top \mathbf{w}')is(\mathbf{x}^\top \mathbf{w} + 1 \cdot b$, which is exactly the original affine function.

Therefore, the affine function $\mathbf{x}^\top \mathbf{w} + b$ is equivalent to the linear function $\mathbf{x}'^\top \mathbf{w}'$ on the augmented vector $(\mathbf{x}, 1)$. This equivalence is often used in machine learning to simplify the notation and the implementation of algorithms.

Exercise 3: A quadratic function of the form $f(\mathbf{x}) = b + \sum_i w_i x_i + \sum_{j \leq i} w_{ij} x_i x_j$ can be implemented in a deep network using a fully connected layer followed by an element-wise multiplication operation and another fully connected layer. Here's how you can do it:

Input Layer: The input to the network is the vector $\mathbf{x}$.

First Fully Connected Layer: This layer applies a linear transformation to the input vector. The weights of this layer represent the w_i terms in the quadratic function. The output of this layer is $b + \sum_i w_i x_i$.

Element-wise Multiplication Layer: This layer computes the element-wise product of the input vector with itself, resulting in the vector $\mathbf{x} \odot \mathbf{x}$, where $\odot$ denotes element-wise multiplication. This operation corresponds to the $x_i x_j$ terms in the quadratic function.

Second Fully Connected Layer: This layer applies a linear transformation to the output of the element-wise multiplication layer. The weights of this layer represent the w_{ij} terms in the quadratic function. The output of this layer is $\sum_{j \leq i} w_{ij} x_i x_j$.

Summation Layer: This layer adds the outputs of the first fully connected layer and the second fully connected layer to produce the final output of the network, which is the value of the quadratic function.

Note that this network does not include any activation functions, because the quadratic function is a polynomial function, not a nonlinear function. If you want to model a more complex relationship between the input and the output, you can add activation functions to the network.

Exercise 4: If the design matrix $\mathbf{X}^\top \mathbf{X}$ does not have full rank, it means that there are linearly dependent columns in the matrix $\mathbf{X}$. This can lead to several problems:

The matrix $\mathbf{X}^\top \mathbf{X}$ is not invertible, which means that the normal equations used to solve the linear regression problem do not have a unique solution. This makes it impossible to find a unique set of regression coefficients that minimize the sum of squared residuals.

The presence of linearly dependent columns in $\mathbf{X}$ can lead to overfitting, as the model can "learn" to use these redundant features to perfectly fit the training data, but it will not generalize well to new data.

One common way to fix this issue is to add a small amount of noise to the entries of $\mathbf{X}$. This can help to make the columns of $\mathbf{X}$ linearly independent, which ensures that $\mathbf{X}^\top \mathbf{X}$ has full rank and is invertible. Another common approach is to use regularization techniques, such as ridge regression or lasso regression,

which add a penalty term to the loss function that discourages the model from assigning too much importance to any one feature.

If you add a small amount of coordinate-wise independent Gaussian noise to all entries of $\mathbf{X}$, the expected value of the design matrix $\mathbf{X}^\top\mathbf{X}$ remains the same. This is because the expected value of a Gaussian random variable is its mean, and adding a constant to a random variable shifts its mean by that constant. So if the noise has mean zero, the expected value of $\mathbf{X}^\top\mathbf{X}$ does not change.

Stochastic gradient descent (SGD) can still be used when $\mathbf{X}^\top\mathbf{X}$ does not have full rank, but it may not converge to a unique solution. This is because SGD does not rely on the invertibility of $\mathbf{X}^\top\mathbf{X}$, but instead iteratively updates the regression coefficients based on a randomly selected subset of the data. However, the presence of linearly dependent columns in $\mathbf{X}$ can cause the loss surface to have multiple minima, which means that SGD may converge to different solutions depending on the initial values of the regression coefficients.

Exercise 5: The negative log-likelihood of the data under the model is given by:

The likelihood function for the data is $P(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} \frac{1}{2} \exp\big(-|y_i - (\mathbf{x}_i^\top \mathbf{w} + b)|\big)$, where y_i is the (i)-th target value, $\mathbf{x}_i$ is the (i)-th input vector, $\mathbf{w}$ is the weight vector, and (b) is the bias term.

Taking the negative logarithm of this gives the negative log-likelihood:
$-\log P(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \big(\log 2 + |y_i - (\mathbf{x}_i^\top \mathbf{w} + b)|\big)$.

Unfortunately, there is no closed-form solution for this problem. The absolute value in the log-likelihood function makes it non-differentiable at zero, which means that we cannot set its derivative equal to zero and solve for $\mathbf{w}$ and b.

A minibatch stochastic gradient descent algorithm to solve this problem would involve the following steps: Initialize $\mathbf{w}$ and b with random values. For each minibatch of data: Compute the gradient of the negative log-likelihood with respect to $\mathbf{w}$ and b. The gradient will be different depending on whether $y_i - (\mathbf{x}_i^\top \mathbf{w} + b)$ is positive or negative. Update $\mathbf{w}$ and b by taking a step in the direction of the negative gradient. Repeat until the algorithm converges. One issue that could arise with this algorithm is that the updates could become very large if $y_i - (\mathbf{x}_i^\top \mathbf{w} + b)$ is close to zero, because the gradient of the absolute value function is not defined at zero. This could cause the algorithm to diverge.

One possible solution to this problem is to use a variant of gradient descent that includes a regularization term, such as gradient descent with momentum or RMSProp. These algorithms modify the update rule to prevent the updates from becoming too large. Another solution is to add a small constant to the absolute value inside the logarithm to ensure that it is always differentiable.

Exercise 6: The composition of two linear layers in a neural network essentially results in another linear layer. This is due to the property of linearity: the composition of two linear functions is another linear function.

Mathematically, if we have two linear layers L_1 and L_2 such that $L_1(x) = Ax + b$ and $L_2(x) = Cx + d$, then the composition $L_2(L_1(x)) = L_2(Ax + b) = C(Ax + b) + d = (CA)x + (Cb + d)$, which is another linear function.

This means that a neural network with two linear layers has the same expressive power as a neural network with a single linear layer. It cannot model complex, non-linear relationships between the input and the output.

To overcome this limitation, we typically introduce non-linearities between the linear layers in the form of activation functions, such as the ReLU (Rectified Linear Unit), sigmoid, or tanh functions. These non-linear activation functions allow the neural network to model complex, non-linear relationships and greatly increase its expressive power.

Exercise 7: Regression for realistic price estimation of houses or stock prices can be challenging due to the complex nature of these markets. Prices can be influenced by a wide range of factors, many of which may not

be easily quantifiable or available for use in a regression model.

The additive Gaussian noise assumption may not be appropriate for several reasons. First, prices cannot be negative, but a Gaussian distribution is defined over the entire real line, meaning it allows for negative values. Second, the Gaussian distribution is symmetric, but price changes in real markets are often not symmetric. For example, prices may be more likely to increase slowly but decrease rapidly, leading to a skewed distribution of price changes. Finally, the Gaussian distribution assumes that large changes are extremely unlikely, but in real markets, large price fluctuations can and do occur.

Regression to the logarithm of the price can be a better approach because it can help to address some of the issues with the Gaussian noise assumption. Taking the logarithm of the prices can help to stabilize the variance of the price changes and make the distribution of price changes more symmetric. It also ensures that the predicted prices are always positive, since the exponential of any real number is positive.

When dealing with penny stocks, or stocks with very low prices, there are several additional factors to consider. One issue is that the prices of these stocks may not be able to change smoothly due to the minimum tick size, or the smallest increment by which the price can change. This can lead to a discontinuous distribution of price changes, which is not well modeled by a Gaussian distribution. Another issue is that penny stocks are often less liquid than higher-priced stocks, meaning there may not be enough buyers or sellers to trade at all possible prices. This can lead to large price jumps, which are also not well modeled by a Gaussian distribution.

The Black-Scholes model for option pricing is a celebrated model in financial economics that makes specific assumptions about the distribution of stock prices. It assumes that the logarithm of the stock price follows a geometric Brownian motion, which implies that the stock price itself follows a log-normal distribution. This model has been very influential, but it has also been criticized for its assumptions, which may not hold in real markets. For example, it assumes that the volatility of the stock price is constant, which is often not the case in reality.

Exercise 8: The Gaussian additive noise model may not be appropriate for estimating the number of apples sold in a grocery store for several reasons: The Gaussian model allows for negative values, but the number of apples sold cannot be negative. The Gaussian model is a continuous distribution, but the number of apples sold is a discrete quantity. The Gaussian model assumes that large deviations from the mean are extremely unlikely, but in reality, there could be large fluctuations in the number of apples sold due to various factors (e.g., seasonal demand, promotions).

The Poisson distribution is a discrete probability distribution that expresses the probability of a given number of events (in this case, the number of apples sold) occurring in a fixed interval of time or space. The parameter λ is the rate at which events occur. The expected value of a Poisson-distributed random variable is indeed λ. This can be shown as follows:

The expected value (E[k]) of a random variable (k) distributed according to a Poisson distribution is given by:

$$E[k] = \sum_{k=0}^{\infty} k \cdot p(k \mid \lambda) = \sum_{k=0}^{\infty} k \cdot \frac{\lambda^k e^{-\lambda}}{k!}$$

By the properties of the Poisson distribution, this sum equals λ.

The loss function associated with the Poisson distribution is typically the negative log-likelihood. Given observed counts $y_1, y_2, \ldots, y_n$ and predicted rates $\lambda_1, \lambda_2, \ldots, \lambda_n$, the negative log-likelihood is:

$$L(\lambda, y) = \sum_{i=1}^{n} (\lambda_i - y_i \log \lambda_i)$$

If we want to estimate $\log \lambda$ instead of λ, we can modify the loss function accordingly. One possible choice is to use the squared difference between the logarithm of the predicted rate and the logarithm of the observed count:

$$L\left(\log \lambda, y\right) = \sum_{i=1}^{n} \left(\log \lambda_i - \log y_i\right)^2$$

This loss function penalizes relative errors in the prediction of the rate, rather than absolute errors. It can be more appropriate when the counts vary over a wide range, as it gives equal weight to proportional errors in the prediction of small and large counts.

9.1.2 Exercise 3.3.5

1. What will happen if the number of examples cannot be divided by the batch size. How would you change this behavior by specifying a different argument by using the framework's API? PyTorch: In PyTorch's *DataLoader*, there's an argument called *drop_last*. If set to True, it will drop the last incomplete batch. By default, it's set to *False*.

```
train_loader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
    shuffle=True, drop_last=True)
```

2. Suppose that we want to generate a huge dataset, where both the size of the parameter vector w and the number of examples num_examples are large. 1. What happens if we cannot hold all data in memory?

2. How would you shuffle the data if it is held on disk? Your task is to design an efficient algorithm that does not require too many random reads or writes. Hint: pseudorandom permutation generators 75 allow you to design a reshuffle without the need to store the permutation table explicitly (Naor and Reingold, 1999).

1. What happens if we cannot hold all data in memory?

If we cannot hold all the data in memory:

Performance Impact: Reading data directly from disk is much slower than reading from memory. If the training algorithm frequently accesses the disk, it can significantly slow down the training process.

Out-of-Core Processing: You'll need to use "out-of-core" or "external memory" algorithms, which are designed to process data that is too large to fit into a computer's main memory. These algorithms break the data into smaller chunks that can fit into memory, process each chunk, and then combine the results.

Streaming: Data can be streamed from the disk in mini-batches. Only one mini-batch is loaded into memory at a time, and after processing it, the next mini-batch is loaded. This is common in deep learning when dealing with large datasets.

Distributed Computing: Another approach is to distribute the dataset across multiple machines in a cluster. Each machine processes a subset of the data. Frameworks like TensorFlow and PyTorch have support for distributed training.

2. How would you shuffle the data if it is held on disk?

Shuffling large datasets on disk without too many random reads or writes can be challenging. Here's a method using pseudorandom permutation generators:

1. Pseudorandom Permutation: Use a pseudorandom permutation generator to generate a sequence of indices that represents the shuffled order of your data. The beauty of pseudorandom permutation generators is that they can generate the nth element of the sequence without generating the first n-1 elements, allowing for efficient random access.

2. Sequential Reads and Writes: - Read the data from the disk sequentially in large chunks (to benefit from sequential read speeds). - For each item in the chunk, use the pseudorandom permutation generator to determine its new location in the shuffled dataset. - Write the shuffled data back to the disk sequentially in large chunks.

3. In-Place Shuffling: If you have some memory available (but not enough to hold the entire dataset), you can load chunks of data into memory, shuffle them using the pseudorandom permutation, and then write them back to disk. This reduces the number of disk writes.

4. External Sorting: Another approach is to use techniques from external sorting. Divide the data into chunks that fit into memory, shuffle each chunk in memory, and then merge the chunks together in a shuffled manner.

5. Temporary Storage: If possible, use a fast intermediate storage (like an SSD) to temporarily store data chunks during the shuffling process. This can speed up both reads and writes.

Remember, the key is to minimize random disk accesses, as they are much slower than sequential accesses. Leveraging pseudorandom permutations allows for an efficient reshuffling without the need to store large permutation tables explicitly.

3. Implement a data generator that produces new data on the fly, every time the iterator is called.

```
import torch
from torch.utils.data import Dataset, DataLoader

class RandomDataset(Dataset):
    def __init__(self, num_samples, tensor_size):
        """
        Args:
            num_samples (int): Number of samples in the dataset.
            tensor_size (tuple): Size of the tensor to be generated.
        """
        self.num_samples = num_samples
        self.tensor_size = tensor_size

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        # Generate a random tensor on the fly
        sample = torch.randn(self.tensor_size)
        return sample

# Parameters
num_samples = 1000
tensor_size = (3, 32, 32)  # Example: 3-channel, 32x32 image

# Create dataset and dataloader
random_dataset = RandomDataset(num_samples, tensor_size)
dataloader = DataLoader(random_dataset, batch_size=32, shuffle=True)

# Iterate over the dataloader
for batch in dataloader:
    print(batch.shape)  # Should print torch.Size([32, 3, 32, 32])
                        # for all batches except possibly the last one
```

4. How would you design a random data generator that generates the same data each time it is called?

```
torch.manual_seed(idx)
```

9.1.3 Exercise 3.5.6

1. How would you need to change the learning rate if you replace the aggregate loss over the minibatch with an average over the loss on the minibatch? When you replace the aggregate loss over the minibatch with an average over the loss on the minibatch, you're essentially scaling the loss by a factor of $\frac{1}{\text{batch size}}$. If you're using gradient descent or a variant thereof, the update rule is:

$$\theta_{\text{new}} = \theta_{\text{old}} - \eta \nabla L$$

Where: - θ represents the parameters of the model. - η is the learning rate. - ∇L is the gradient of the loss function L with respect to the parameters.

If you change the loss from an aggregate to an average, the gradient ∇L will also be scaled by $\frac{1}{\text{batch size}}$. To compensate for this scaling and to ensure that the magnitude of the parameter updates remains the same, you would need to multiply the learning rate η by the batch size.

In other words, if your original learning rate was η_{original} when using aggregate loss, and you switch to using average loss, your new learning rate should be:

$$\eta_{\text{new}} = \eta_{\text{original}} \times \text{batch size}$$

However, in practice, many deep learning frameworks, including TensorFlow and PyTorch, compute the average loss over a minibatch by default. So, if you're already using one of these frameworks, you likely don't need to adjust the learning rate when switching between aggregate and average loss, unless you were explicitly scaling the loss by the batch size yourself.

4. What is the effect on the solution if you change the learning rate and the number of epochs? Does it keep on improving? The learning rate and the number of epochs are two critical hyperparameters in training neural networks and other machine learning models. Their values can significantly impact the convergence and performance of the model. Let's discuss the effects of changing each:

1. Learning Rate (η): - Too High: If the learning rate is set too high, the model might overshoot the minima in the loss landscape, leading to divergence. The loss might oscillate or even explode. - Too Low: If the learning rate is too low, the model will converge very slowly. It might get stuck in local minima or plateaus in the loss landscape. - Adaptive: Many modern optimizers (like Adam, RMSprop) adjust the learning rate dynamically based on recent gradients, which can help in finding a good balance.

2. Number of Epochs: - Too Few: If the number of epochs is too low, the model might underfit the data, as it hasn't seen the data enough times to learn the underlying patterns. - Too Many: If the number of epochs is too high, the model might overfit the data, especially if there isn't an early stopping mechanism or regularization in place. Overfitting means the model performs well on the training data but poorly on unseen data.

Does it keep on improving?

- Not necessarily. As training progresses: - The loss typically decreases and the model's performance on the training set improves. - However, after a certain point, the model might start to overfit, and its performance on a validation set might start to degrade. - The optimal number of epochs is often where the performance on the validation set is maximized.

- The learning rate plays a crucial role in this: - A good learning rate can help the model converge to a good solution faster. - Learning rate schedules or decay can be beneficial. Starting with a larger learning rate (to progress quickly) and reducing it over time (to fine-tune) can be effective.

In summary, there's a balance to strike. It's essential to monitor both training and validation metrics during training to determine the best values for the learning rate and the number of epochs. Experimentation and hyperparameter tuning, possibly with techniques like cross-validation or Bayesian optimization, can help find the optimal values for a given problem and dataset.

9.1.4 Exercise 4.1.5

1. We can explore the connection between exponential families and softmax in some more depth 1. Compute the second derivative of the cross-entropy loss $l(y, \hat{y})$ for softmax. 2. Compute the variance of the distribution given by softmax(o) and show that it matches the second derivative computed above.

1. Second Derivative of Cross-Entropy Loss for Softmax, The cross-entropy loss for softmax is given by:

$$l(y, \hat{y}) = -\sum_i y_i \log(\hat{y}_i)$$

Where $\hat{y}_i$ is the softmax output:

$$\hat{y}_i = \frac{\exp(o_i)}{\sum_j \exp(o_j)}$$

Let's compute the second derivative with respect to the logits o.
First, the derivative of $\hat{y}_i$ with respect to o_k is:

$$\frac{\partial \hat{y}_i}{\partial o_k} = \hat{y}_i(\delta_{ik} - \hat{y}_k)$$

Where δ_{ik} is the Kronecker delta (it's when $i = k$ and 0 otherwise). Using the above, the second derivative is:

$$\frac{\partial^2 l}{\partial o_i \partial o_k} = -\sum_j y_j \frac{\partial^2 \log(\hat{y}_j)}{\partial o_i \partial o_k}$$

$$= -\sum_j y_j \frac{\partial}{\partial o_i} \left(\frac{\hat{y}_j(\delta_{jk} - \hat{y}_k)}{\hat{y}_j} \right)$$

$$= -\sum_j y_j \left(\delta_{jk} - \hat{y}_k\right) \frac{\partial \hat{y}_j}{\partial o_i}$$

$$= -\sum_j y_j \left(\delta_{jk} - \hat{y}_k\right) \hat{y}_j \left(\delta_{ji} - \hat{y}_i\right)$$

2. Variance of the Distribution Given by Softmax
The variance $\mathrm{Var}[X]$ of a discrete random variable X with possible outcomes $x_1, x_2, \ldots$ and probabilities $p_1, p_2, \ldots$ is:

$$\mathrm{Var}[X] = \sum_i p_i \left(x_i - \mathbb{E}[X]\right)^2$$

For the softmax distribution, X can take on values corresponding to the logits o with probabilities $\hat{y}$. Thus, the variance becomes:

$$\mathrm{Var}[X] = \sum_i \hat{y}_i \left(o_i - \mathbb{E}[X]\right)^2$$

Where $\mathbb{E}[X] = \sum_i \hat{y}_i o_i$.

Now, the second derivative of the cross-entropy loss with respect to the logits, as derived above, gives us a measure of the curvature of the loss. When we compare this with the variance of the softmax distribution, we can see that they are related. The variance captures the spread of the logits, and the second derivative captures the sensitivity of the loss to changes in the logits. In the context of exponential families, this relationship between variance and the second derivative of the log-likelihood is a known property.

To show that they match, one would equate the expressions derived in the two sections above and simplify. The detailed algebra can be quite involved, but the key insight is understanding the relationship between the curvature of the loss (second derivative) and the spread of the logits (variance).

5. Softmax gets its name from the following mapping: $RealSoftMax(a, b) = log(exp(a) + exp(b))$

1. Prove that RealSoftMax$(a, b) > \max(a, b)$:

Given:

$$\text{RealSoftMax}(a, b) = \log(\exp(a) + \exp(b))$$

Now, since $\exp(a)$ and $\exp(b)$ are both positive, we have:

$$\exp(a) + \exp(b) > \exp(a)$$

$$\exp(a) + \exp(b) > \exp(b)$$

Taking the logarithm of both sides (logarithm is a monotonically increasing function):

$$\log(\exp(a) + \exp(b)) > \log(\exp(a)) = a$$

$$\log(\exp(a) + \exp(b)) > \log(\exp(b)) = b$$

Thus, RealSoftMax$(a, b) > \max(a, b)$.

2. How small can you make the difference between both functions?

Without loss of generality, let's set $b = 0$ and $a \geq b$. Then:

$$\text{RealSoftMax}(a, 0) = \log(\exp(a) + 1)$$

The difference between the two functions is:

$$\text{RealSoftMax}(a, 0) - a = \log(\exp(a) + 1) - a$$

As a becomes large, the difference approaches 0, but it's always positive because of the "+1" inside the logarithm.

3. Prove that this holds for λ^{-1}RealSoftMax $(\lambda a, \lambda b)$, provided that $\lambda > 0$:

Given:

$$\lambda^{-1}\text{RealSoftMax}\left(\lambda a, \lambda b\right) = \lambda^{-1} \log(\exp(\lambda a) + \exp(\lambda b))$$

Using the property of logarithms:

$$= \log(\exp(\lambda a) + \exp(\lambda b))^{\lambda^{-1}}$$

Since $\exp(\lambda a)$ and $\exp(\lambda b)$ are both positive and $\lambda > 0$, the proof from part applies here as well, and we can conclude that λ^{-1}RealSoftMax$(\lambda a, \lambda b) > \max(a, b)$.

4. Show that for $\lambda \to \infty$ we have λ^{-1}RealSoftMax$(\lambda a, \lambda b) \to \max(a, b)$:

As λ becomes very large, the term $\exp(\lambda a)$ will dominate $\exp(\lambda b)$ if $a > b$. Thus:

$$\lambda^{-1} \log(\exp(\lambda a) + \exp(\lambda b)) \approx \lambda^{-1} \log(\exp(\lambda a)) = a$$

Similarly, if $b > a$, the result will be b. Therefore, as $\lambda \to \infty$, $\lambda^{-1}\text{RealSoftMax}(\lambda a, \lambda b) \to \max(a, b)$.
5. Construct an analogous softmin function:
The softmin function can be constructed using the negative of the inputs to the softmax function:

$$\text{RealSoftMin}(a, b) = -\text{RealSoftMax}(-a, -b)$$

$$= \log(\exp(-a) + \exp(-b))$$

6. Extend this to more than two numbers:
For n numbers $a_1, a_2, \ldots, a_n$:

$$\text{RealSoftMax}(a_1, a_2, \ldots, a_n) = \log(\exp(a_1) + \exp(a_2) + \ldots + \exp(a_n))$$

Similarly, the softmin for n numbers is:

$$\text{RealSoftMin}(a_1, a_2, \ldots, a_n) = \log(\exp(-a_1) + \exp(-a_2) + \ldots + \exp(-a_n))$$

9.1.5 Exercise 4.5.5

format	format_min	format_max	exp_min	exp_max	test_min	exp(max-1)	exp(max+1)
torch.float64	-1.80e+308	1.80e+308	-1.80e+308	7.10e+02	0.00e+00	6.61e+307	inf
torch.float32	-3.40e+38	3.40e+38	-3.40e+38	8.87e+01	0.00e+00	1.25e+38	inf
torch.bfloat16	-3.39e+38	3.39e+38	-3.39e+38	8.85e+01	0.00e+00	1.00e+38	inf
torch.float16	-6.55e+04	6.55e+04	-6.55e+04	1.11e+01	0.00e+00	2.41e+04	inf
torch.int8	-1.28e+02	1.27e+02	-1.28e+02	4.84e+00	0.00e+00	4.67e+01	8.90e+01

Deep learning uses many different number formats, including FP64 double precision (used extremely rarely), FP32 single precision, BFLOAT16 (good for compressed representations), FP16 (very unstable), TF32 (a new format from NVIDIA), and INT8. Compute the smallest and largest argument of the exponential function for which the result does not lead to numerical underflow or overflow 1. FP64 (Double Precision): - Smallest positive normalized number 2^{-1022} - Largest finite representable number $(2 - 2^{-52}) \times 2^{1023}$ - For the exponential function, the range of x for which e^x does not overflow would be approximately $[-1022 \log(2), 1023 \log(2)]$.

2. FP32 (Single Precision): - Smallest positive normalized number: 2^{-126} - Largest finite representable number: $(2 - 2^{-23}) \times 2^{127}$ - For the exponential function, the range of x for which e^x does not overflow would be approximately $[-126 \log(2), 127 \log(2)]$.

3. BFLOAT16: - This format has the same exponent bits as FP32 but fewer precision bits. - The range for the exponential function would be similar to FP32.

4. FP16: - Smallest positive normalized number: 2^{-14} - Largest finite representable number: $(2 - 2^{-10}) \times 2^{15}$ - For the exponential function, the range of x for which e^x does not overflow would be approximately $[-14 \log(2), 15 \log(2)]$.

5. TF32: - This is a new format introduced by NVIDIA, and it has 10 bits for the exponent and 19 bits for the fraction. - The range would be somewhere between FP32 and FP16.

6. INT8: - This is an integer format, so it doesn't directly apply to the exponential function in the same way as the floating-point formats. However, if used in quantized neural networks, the range would be $[-128, 127]$.

For the exact ranges, especially for the newer formats like TF32, one would need to refer to the official documentation or specifications provided by the manufacturers.

To avoid overflow or underflow when computing the exponential function, it's common to clip or bound the input values to the function within the safe range for the given number format.

9.1.6 Exercise 7.1.6

1. Assume that the size of the convolution kernel is $\Delta = 0$. Show that in this case the convolution kernel implements an MLP independently for each set of channels. This leads to the Network in Network architectures When the size of the convolution kernel is $\Delta = 0$, it means that the kernel effectively has a size of 1×1. A 1×1 convolution kernel operates on each pixel individually, without considering its neighbors.

Let's break down the implications of this:

1. Independent Operation on Each Pixel: A 1×1 convolution operates on each pixel independently. For an input with C_{in} channels, each pixel is represented by a C_{in}-dimensional vector. The 1×1 convolution transforms this C_{in}-dimensional vector into a C_{out}-dimensional vector, where C_{out} is the number of output channels.

2. MLP for Each Set of Channels: The transformation from C_{in} to C_{out} for each pixel can be viewed as a fully connected layer (or MLP) applied independently to each spatial location. This is because the weights of the 1×1 convolution kernel can be seen as the weights of a fully connected layer that connects each input channel to each output channel.

3. Network in Network (NiN): The idea of using 1×1 convolutions to implement an MLP for each set of channels led to the Network in Network (NiN) architecture. In NiN, multiple 1×1 convolutional layers are stacked, allowing for deep per-pixel transformations. This introduces additional non-linearity for each pixel without changing the spatial dimensions.

In summary, a 1×1 convolution kernel effectively applies an MLP to each pixel independently, transforming the channels at each spatial location. This concept is central to the Network in Network (NiN) architecture, which uses 1×1 convolutions to introduce additional non-linearities in the network without altering the spatial dimensions.

5. What happens with convolutions when an object is at the boundary of an image? When an object is at the boundary of an image, and we apply convolutions, several issues and considerations arise:

1. Loss of Information: The primary issue is that for pixels on the boundary, there aren't enough neighboring pixels to apply the convolution kernel fully. This means that these boundary pixels might not be processed adequately, leading to potential loss of information about objects at the image boundary.

2. Padding: To address the boundary issue, one common approach is to add padding around the image. Padding involves adding extra pixels around the boundary of the image. There are different types of padding: - Zero Padding: Adding pixels with a value of zero. - Reflect Padding: The boundary pixels are reflected. For instance, if the last row of an image is $[a, b, c]$, it becomes $[a, b, c, c, b, a]$ after reflect padding. - Replicate Padding: The boundary pixels are replicated. Using the previous example, it becomes $[a, b, c, c, c, c]$.

Padding ensures that every pixel, including those on the boundary, has a full neighborhood of pixels to apply the convolution kernel.

3. Valid vs. Same Convolutions: - Valid Convolution: No padding is used. The resulting feature map after convolution is smaller than the input image because the kernel can only be applied to positions where it fits entirely within the image boundaries. - Same Convolution: Padding is added such that the output feature map has the same spatial dimensions as the input image.

4. Edge Effects: Even with padding, objects at the boundary might be affected differently than objects in the center of the image. This is because the artificial values introduced by padding might not represent the actual image content. This can lead to edge artifacts in the output feature map.

5. Dilated Convolutions: In dilated convolutions, where the kernel elements are spaced out by introducing gaps, the boundary effects can be even more pronounced, especially if the dilation rate is high.

6. Pooling Layers: The boundary effects can also impact pooling layers (like max-pooling or average-pooling). If the pooling window doesn't fit entirely within the image or feature map boundaries, padding might be needed, or the pooling window might be truncated.

6. Prove that the convolution is symmetric, $f * g = g * f$ To prove that convolution is symmetric, we'll start with the definition of the convolution operation for two functions $f(t)$ and $g(t)$:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)\, g(t - \tau)\, d\tau$$

Now, let's express the convolution of g and f:

$$(g * f)(t) = \int_{-\infty}^{\infty} g(\tau)\, f(t - \tau)\, d\tau$$

To show that these two expressions are equivalent, we'll make a substitution:
Let $\lambda = t - \tau$ This implies $d\lambda = -d\tau$
When $\tau = -\infty$, $\lambda = t + \infty = \infty$ When $\tau = \infty$, $\lambda = t - \infty = -\infty$
Substituting these values into the convolution of g and f, we get:

$$(g * f)(t) = \int_{\infty}^{-\infty} g(t - \lambda) f(\lambda)(-d\lambda)$$

Now, reversing the limits of integration:

$$(g * f)(t) = \int_{-\infty}^{\infty} g(t - \lambda) f(\lambda)\, d\lambda$$

Comparing this with the expression for $(f * g)(t)$, we see that the two expressions are equivalent:

$$(f * g)(t) = (g * f)(t)$$

Thus, we've proven that the convolution operation is symmetric:

$$f * g = g * f$$

Exercise 7.2.8

4. How do you represent a cross-correlation operation as a matrix multiplication by changing the input and kernel tensors? Cross-correlation can be represented as a matrix multiplication by converting the input and kernel tensors into appropriate matrix and vector forms, respectively. This process is often referred to as "toeplitzization" or "im2col" (image to column) transformation. Here's how you can achieve this:

1. Flatten the Kernel: - First, you reshape the kernel into a column vector. - If the kernel is of size $k \times k$, the resulting vector will be of size $k^2 \times 1$.

2. Toeplitz Matrix for Input: - For each position in the output, extract a $k \times k$ patch from the input, and flatten it into a row vector. - Stack these row vectors on top of each other to form a matrix. This matrix has a special structure called a Toeplitz matrix. - If the input is of size $n \times n$ and the kernel is $k \times k$, then the resulting matrix will be of size $(n - k + 1)^2 \times k^2$.

3. Matrix Multiplication: - Multiply the Toeplitz matrix (from step 2) with the flattened kernel (from step 1). The result will be a column vector representing the flattened output of the cross-correlation operation. - Reshape this column vector back into a 2D matrix to get the final output.

Mathematically, if M is the Toeplitz matrix derived from the input and K is the column vector derived from the kernel, then the output O is given by:

$$O = M \times K$$

This representation is particularly useful for certain computational platforms and algorithms that can efficiently handle matrix multiplications. In deep learning frameworks, this transformation is often used under the hood to speed up convolution operations, especially when using GPUs.

Note: This explanation assumes 2D inputs and kernels for simplicity, but the concept can be extended to higher dimensions.

```
def im2col(inputs, kernel_size):
    # Extract patches from the input tensor
    k, _ = kernel_size
    unfold = torch.nn.Unfold(kernel_size=k, stride=1, padding=0)
    cols = unfold(inputs)
    return cols.transpose(1, 2)
```

Chapter 10

Generative Deep Learning

Contents

10.1 Intro to Deep Generative Models

What is Generative Modeling?

Generative modeling can be broadly defined as follows: Generative modeling is a branch of machine learning that involves training a model to produce new data that is similar to a given dataset [Fos22].

Discriminative models vs Generative models

The Generative Modeling Framework

- We have a dataset of Observations $\mathbf{X}$.
- We assume that the observations have been generated according to some unknown distribution $\mathcal{P}_{\text{data}}$
- We want to build a generative model $\mathcal{P}_{\text{model}}$ that mimics $\mathcal{P}_{\text{data}}$ to generate observations that appear to have been drawn from $\mathcal{P}_{\text{data}}$
- Therefore, the desirable properties of $\mathcal{P}_{\text{data}}$ are:

  1. Accuracy: if $\mathcal{P}_{\text{model}}$ is high for a generated observation, it should look like it has been drawn from $\mathcal{P}_{\text{data}}$. If $\mathcal{P}_{\text{model}}$ is low, it should look like it has not been drawn from $\mathcal{P}_{\text{data}}$

  2. Generation: it should be possible to easily sample a new observation from $\mathcal{P}_{\text{model}}$

  3. Representation: it should be possible to understand how different high-level features in the data are represented by $\mathcal{P}_{\text{model}}$

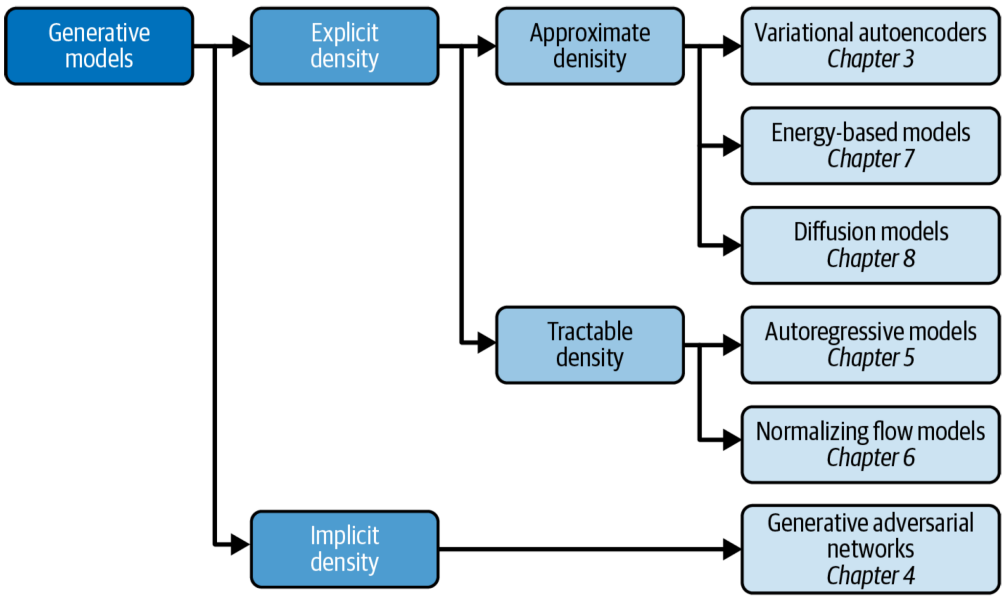

Figure 1-10. A taxonomy of generative modeling approaches

Figure 10.1: Generative Models Taxonomy

Generative Model Taxonomy

- Explicitly model the density function, but constrain the model in some way, so that the density function s tractable (i.e. it can be calculated)
- Explicitly model a tractable approximation of the density function.
- Implicitly model the density function, through a stochastic process that directly generates data.

If you do choose to use batch normalization before activation, you can remember the order using the acronym **BAD (batch normalization, activation, then dropout)**

10.2 Variational Autoencoders

The Reparameterization Trick Rather than sample directly from a normal distribution with parameters z_mean and z_log_var, we can sample epsilon from a standard normal and then manually adjust the sample to have the correct mean and variance. This is known as the reparameterization trick, and it's important as it means gradients can backpropagate freely through the layer. By keeping all of the randomness of the layer contained within the variable epsilon, the partial derivative of the layer output with respect to its input can be shown to be deterministic (i.e., independent of the random epsilon), which is essential for backpropagation through the layer to be possible.

Kullback-Leibler (KL) Divergence

$$D_{\mathrm{KL}}\left[N\left(\mu,\sigma\right)||N\left(0,1\right)\right]=\frac{1}{2}\sum_{i=1}^{n}\left(\sigma_i^2+\mu_i^2-1-\log\sigma_i^2\right).$$

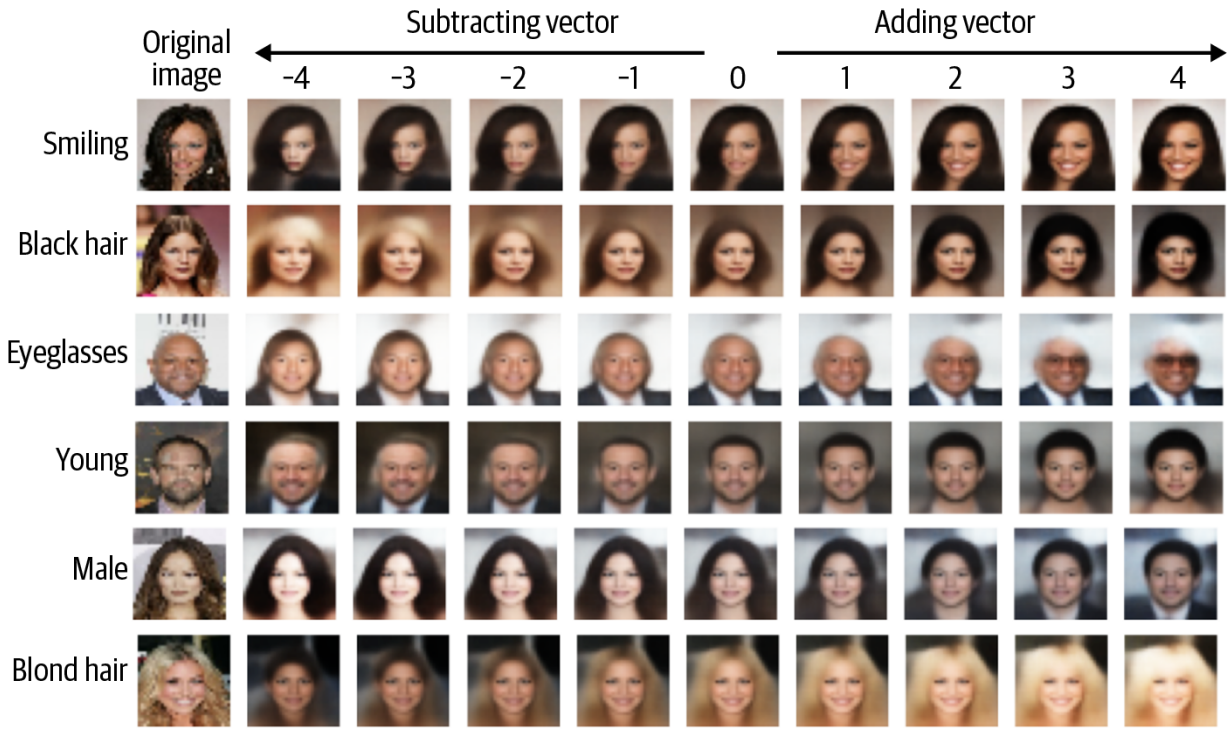

Figure 10.2: Adding and subtracting features to and from faces

In summary, the KL divergence term penalizes the network for encoding observations to z_mean and z_log_var variables that differ significantly from the parameters of a standard normal distribution, namely $z\_mean = 0$ and $z\_log\_var = 0$.

Why does this addition to the loss function help?

Firstly, we now have a well-defined distribution that we can use for choosing points in the latent space—the standard normal distribution. Secondly, since this term tries to force all encoded distributions toward the standard normal distribution, there is less chance that large gaps will form between point clusters. Instead, the encoder will try to use the space around the origin symmetrically and efficiently.

In the original Variational Autoencoder (VAE) paper, the loss function for a VAE was simply the addition of the reconstruction loss and the KL divergence loss term. A variant on this (the β-VAE) includes a factor that weights the KL divergence to ensure that it is well balanced with the reconstruction loss. If we weight the reconstruction loss too heavily, the KL loss will not have the desired regulatory effect and we will see the same problems that we experienced with the plain autoencoder. If the KL divergence term is weighted too heavily, the KL divergence loss will dominate and the reconstructed images will be poor. This weighting term is one of the parameters to tune when you're training your VAE [Fos22].

Latent Space Arithmetic One benefit of mapping images into a lower-dimensional latent space is that we can perform arithmetic on vectors in this latent space that has a visual analogue when decoded back into the original image domain (Figure 10.2).

For example, suppose we want to take an image of somebody who looks sad and give them a smile. To do this we first need to find a vector in the latent space that points in the direction of increasing smile. Adding this vector to the encoding of the original image in the latent space will give us a new point which, when decoded, should give us a more smiley version of the original image [Fos22].

Figure 10.3: Artifacts when using convolutional transpose layers

10.3 Generative Adversarial Networks

Upsampling vs Transposed Convolution The UpSampling2D layer simply repeats each row and column of its input in order to double the size. The Conv2D layer with stride 1 then performs the convolution operation. It is a similar idea to convolutional transpose, but instead of filling the gaps between pixels with zeros, upsampling just repeats the existing pixel values.

It has been shown that the Conv2DTranspose method can lead to artifacts, or small checkerboard patterns in the output image (see Figure 10.3) that spoil the quality of the output. However, they are still used in many of the most impressive GANs in the literature and have proven to be a powerful tool in the deep learning practitioner's toolbox.

Adding Noise to the Labels A useful trick when training GANs is to add a small amount of random noise to the training labels. This helps to improve the stability of the training process and sharpen the generated images. This label smoothing acts as way to tame the discriminator, so that it is presented with a more challenging task and doesn't overpower the generator.

Another requirement of a successful generative model is that it doesn't only reproduce images from the training set. To test this, we can find the image from the training set that is closest to a particular generated example. A good measure for distance is the L1 distance, defined as:

$$\mathrm{L1}(\mathbf{a}, \mathbf{b}) = \frac{1}{n} \sum_{i=1}^{n} |a_i - b_i|.$$

10.3.1 GAN Training Tips

While GANs are a major breakthrough for generative modeling, they are also notoriously difficult to train. We will explore some of the most common problems and challenges encountered when training GANs in this section, alongside potential solutions. In the next section, we will look at some more fundamental adjustments to the GAN framework that we can make to remedy many of these problems [Fos22].

Discriminator overpowering the generator If the discriminator becomes too strong, the signal from the loss function becomes too weak to drive any meaningful improvements in the generator. In the worst-case scenario, the discriminator perfectly learns to separate real images from fake images and the gradients vanish completely, leading to no training whatsoever,

If you find your discriminator loss function collapsing, you need to find ways to weaken the discriminator. Try the following suggestions:

- Increase the rate parameter of the Dropout layers in the discriminator to dampen the amount of information that flows through the network.
- Reduce the learning rate of the discriminator.
- Reduce the number of convolutional filters in the discriminator.
- Add noise to the labels when training the discriminator.
- Flip the labels of some images at random when training the discriminator.

Generator overpowering the discriminator If the discriminator is not powerful enough, the generator will find ways to easily trick the discriminator with a small sample of nearly identical images. This is known as mode collapse.

For example, suppose we were to train the generator over several batches without updating the discriminator in between. The generator would be inclined to find a single observation (also known as a mode) that always fools the discriminator and would start to map every point in the latent input space to this image. Moreover, the gradients of the loss function would collapse to near 0, so it wouldn't be able to recover from this state.

Even if we then tried to retrain the discriminator to stop it being fooled by this one point, the generator would simply find another mode that fools the discriminator, since it has already become numb to its input and therefore has no incentive to diversify its output.

If you find that your generator is suffering from mode collapse, you can try strengthening the discriminator using the opposite suggestions to those listed in the previous section. Also, you can try reducing the learning rate of both networks and increasing the batch size.

Uninformative Loss Since the deep learning model is compiled to minimize the loss function, it would be natural to think that the smaller the loss function of the generator, the better the quality of the images produced. However, since the generator is only graded against the current discriminator and the discriminator is constantly improving, we cannot compare the loss function evaluated at different points in the training process. Indeed the loss function of the generator actually increases over time, even though the quality of the images is clearly improving. This lack of correlation between the generator loss and image quality sometimes makes GAN training difficult to monitor.

Hyperparameter As we have seen, even with simple GANs, there are a large number of hyperparameters to tune. As well as the overall architecture of both the discriminator and the generator, there are the parameters that govern batch normalization, dropout, learning rate, activation layers, convolutional filters, kernel size, striding, batch size, and latent space size to consider. GANs are highly sensitive to very slight changes in all of these parameters, and finding a set of parameters that works is often a case of educated trial and error, rather than following an established set of guidelines.

WGAN-GP WGAN-GP (Figure 10.4) brings a meaningful loss metric that correlates with the generator's convergence and sample quality, and improved stability of the optimization process.

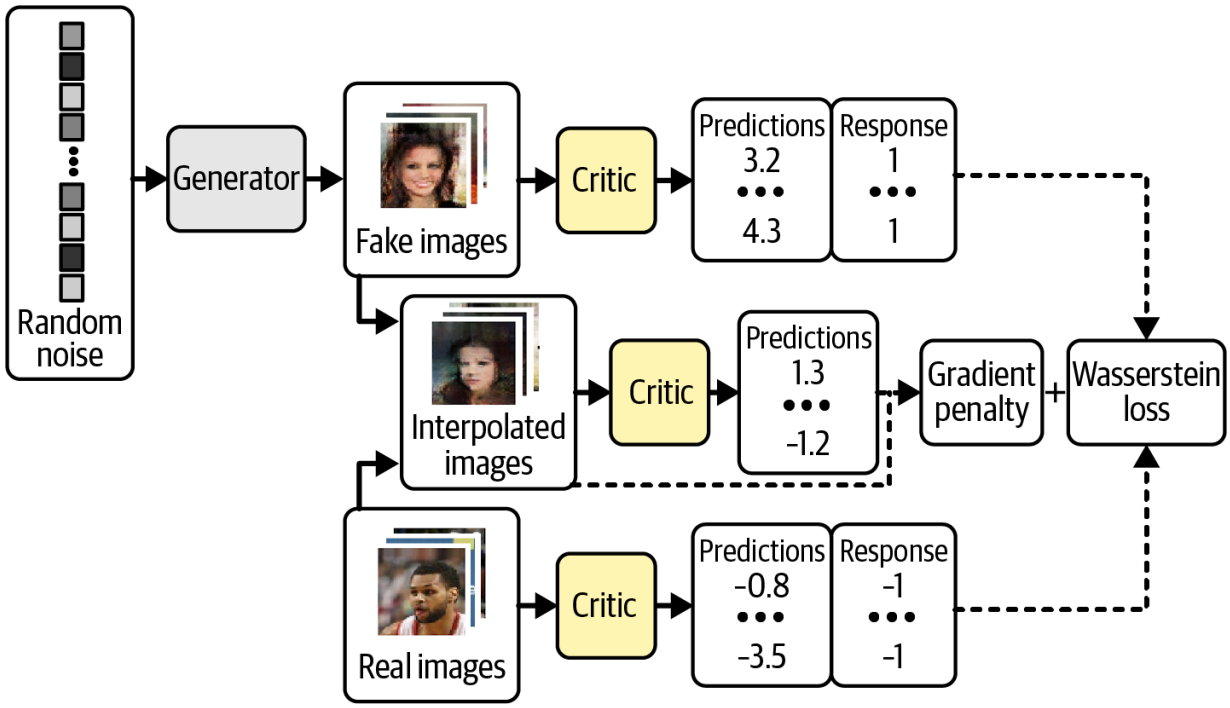

Figure 10.4: The WGAN-GP critic training process

$$-\frac{1}{n}\sum_{i=1}^{n}\left(y_i \log\left(p_i\right) + \left(1-y_i\right)\log\left(1-p_i\right)\right) \tag{10.1}$$

One last consideration we should note before training a WGAN-GP is that batch normalization shouldn't be used in the critic. This is because batch normalization creates correlation between images in the same batch, which makes the gradient penalty loss less effective. Experiments have shown that WGAN-GPs can still produce excellent results even without batch normalization in the critic. We have now covered all of the key differences between a standard GAN and a WGAN-GP. To recap:

- A WGAN-GP uses the Wasserstein loss.
- The WGAN-GP is trained using labels of 1 for real and −1 for fake.
- There is no sigmoid activation in the final layer of the critic.
- Include a gradient penalty term in the loss function for the critic.
- Train the critic multiple times for each update of the generator.
- There are no batch normalization layers in the critic.

Conditional Generative Adversarial Nets (CGAN)

Summary we explored three different GAN models: the Deep Convolutional GAN (DCGAN), the more sophisticated WGAN-GP, and the CGAN.

All GANs are characterized by a generator versus discriminator (or critic) architecture, with the discriminator trying to "spot the difference" between real and fake images and the generator aiming to fool the

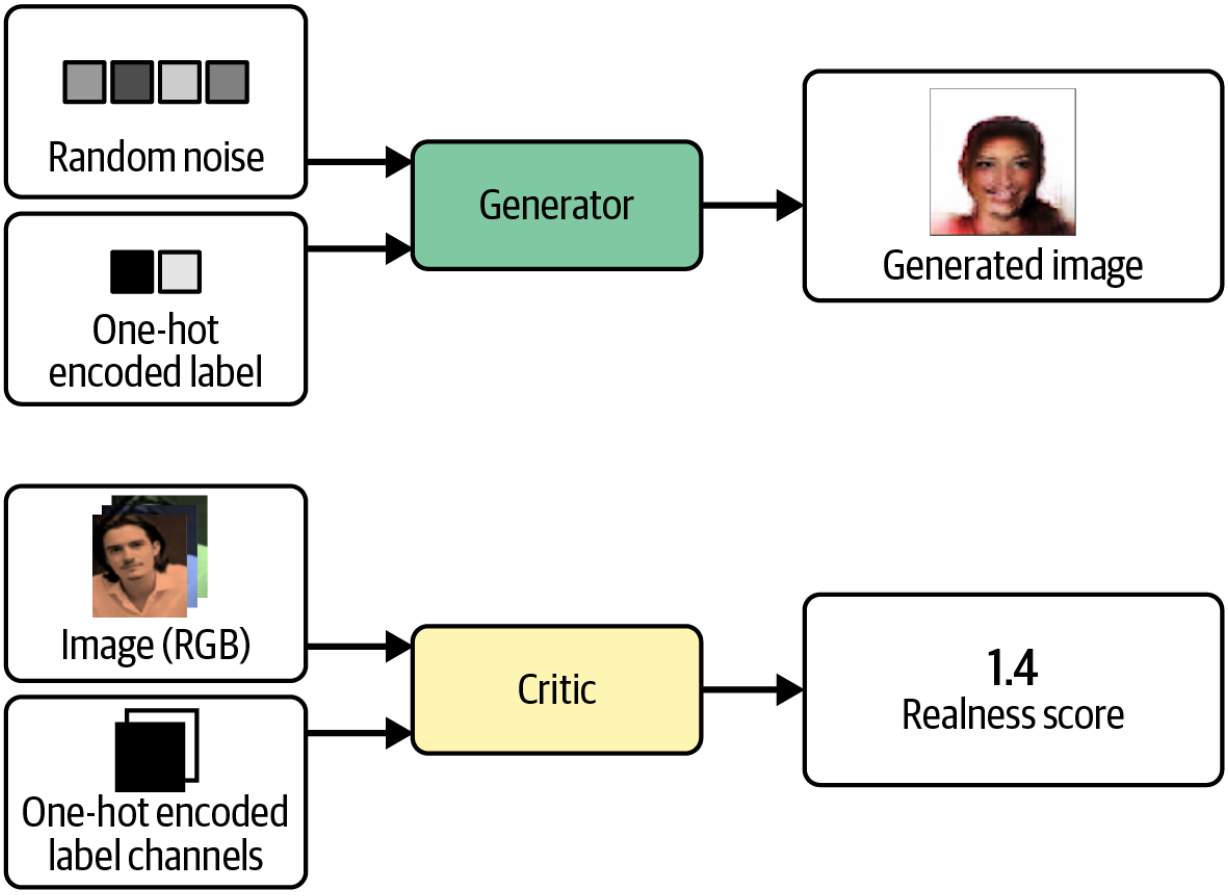

Figure 10.5: Inputs and outputs of the generator and critic in a CGAN

discriminator. By balancing how these two adversaries are trained, the GAN generator can gradually learn how to produce similar observations to those in the training set.

We first saw how to train a DCGAN to generate images of toy bricks. It was able to learn how to realistically represent 3D objects as images, including accurate representations of shadow, shape, and texture. We also explored the different ways in which GAN training can fail, including mode collapse and vanishing gradients [Fos22].

10.4 Autoregressive Models

Working with Text Data There are several key differences between text and image data that mean that many of the methods that work well for image data are not so readily applicable to text data. In particular:

- Text data is composed of discrete chunks (either characters or words), whereas pixels in an image are points in a continuous color spectrum. We can easily make a green pixel more blue, but it is not obvious how we should go about making the word cat more like the word dog, for example. This means we can easily apply backpropagation to image data, as we can calculate the gradient of our loss function with respect to individual pixels to establish the direction in which pixel colors should be changed to minimize the loss. With discrete text data, we can't obviously apply backpropagation in the same way, so we need to find a way around this problem.
- Text data has a time dimension but no spatial dimension, whereas image data has two spatial dimensions but no time dimension. The order of words is highly important in text data and words wouldn't make sense in reverse, whereas images can usually be flipped without affecting the content. Furthermore, there are often long-term sequential dependencies between words that need to be captured by the model: for example, the answer to a question or carrying forward the context of a pronoun. With image data, all pixels can be processed simultaneously.
- Text data is highly sensitive to small changes in the individual units (words or characters). Image data is generally less sensitive to changes in individual pixel units—a picture of a house would still be recognizable as a house even if some pixels were altered—but with text data, changing even a few words can drastically alter the meaning of the passage, or make it nonsensical. This makes it very difficult to train a model to generate coherent text, as every word is vital to the overall meaning of the passage.
- Text data has a rules-based grammatical structure, whereas image data doesn't follow set rules about how the pixel values should be assigned. For example, it wouldn't make grammatical sense in any context to write "The cat sat on the having." There are also semantic rules that are extremely difficult to model; it wouldn't make sense to say "I am in the beach," even though grammatically, there is nothing wrong with this statement.

Tokenization If you use word tokens:

- All text can be converted to lowercase, to ensure capitalized words at the start of sentences are tokenized the same way as the same words appearing in the middle of a sentence. In some cases, however, this may not be desirable; for example, some proper nouns, such as names or places, may benefit from remaining capitalized so that they are tokenized independently.
- The text vocabulary (the set of distinct words in the training set) may be very large, with some words appearing very sparsely or perhaps only once. It may be wise to replace sparse words with a token for unknown word, rather than including them as separate tokens, to reduce the number of weights the neural network needs to learn.

- Words can be stemmed, meaning that they are reduced to their simplest form, so that different tenses of a verb remained tokenized together. For example, browse, browsing, browses, and browsed would all be stemmed to brows.
- You will need to either tokenize the punctuation, or remove it altogether.
- Using word tokenization means that the model will never be able to predict words outside of the training vocabulary.

If you use character tokens:

- The model may generate sequences of characters that form new words outside of the training vocabulary—this may be desirable in some contexts, but not in others.
- Capital letters can either be converted to their lowercase counterparts, or remain as separate tokens.
- The vocabulary is usually much smaller when using character tokenization. This is beneficial for model training speed as there are fewer weights to learn in the final output layer.

LSTM A recurrent layer has the special property of being able to process sequential input data $x_1, \ldots, x_n$. It consists of a cell that updates its hidden state, h_t, as each element of the sequence x_t is passed through it, one timestep at a time.

The hidden state is a vector with length equal to the number of units in the cell. it can be thought of as the cell's current understanding of the sequence. At timestep t, the cell uses the previous value of the hidden state,h_{t-1}, together with the data from the current timestep x_t to produce an updated hidden state vector, h_t. This recurrent process continues until the end of the sequence. Once the sequence is finished, the layer outputs the final hidden state of the cell, h_n, It's important to remember that all of the cells in this diagram share the same weights (as they are really the same cell).

We represent the recurrent process by drawing a copy of the cell at each timestep and show how the hidden state is constantly being updated as it flows through the cells (Figure 10.6).

The hidden state is updated in six steps (Figure 10.7):

1. The hidden state of the previous timestep, h_{t-1}, and the current word embedding, x_t, are concatenated and passed through the forget gate. This gate is simply a dense layer with weights matrix $\mathbf{W}_f$, bias b_f, and a sigmoid activation function. The resulting vector, $f_t l$, has length equal to the number of units in the cell and contains values between 0 and 1 that determine how much of the previous cell state, C_{t-1}, should be retained.
2. The concatenated vector is also passed through an input gate that, like the forget gate, is a dense layer with weights matrix $\mathbf{W}_i$, bias b_i, and a sigmoid activation function. The output from this gate, i_t, has length equal to the number of units in the cell and contains values between 0 and 1 that determine how much new information will be added to the previous cell state, C_{t-1}.
3. The concatenated vector is passed through a dense layer with weights matrix $\mathbf{W}_C$, bias b_C, and a tanh activation function to generate a vector $\tilde{C}_t$ that contains the new information that the cell wants to consider keeping. It also has length equal to the number of units in the cell and contains values between −1 and 1.
4. f_t and C_{t-1} are multiplied element-wise and added to the element-wise multiplication of i_t and $\tilde{C}_t$. This represents forgetting parts of the previous cell state and then adding new relevant information to produce the updated cell state, C_t.
5. The concatenated vector is passed through an output gate: a dense layer with weights matrix $\mathbf{W}_o$, bias b_o, and a sigmoid activation. The resulting vector, o_t, has length equal to the number of units in the cell and stores values between 0 and 1 that determine how much of the updated cell state, C_t, to output from the cell.

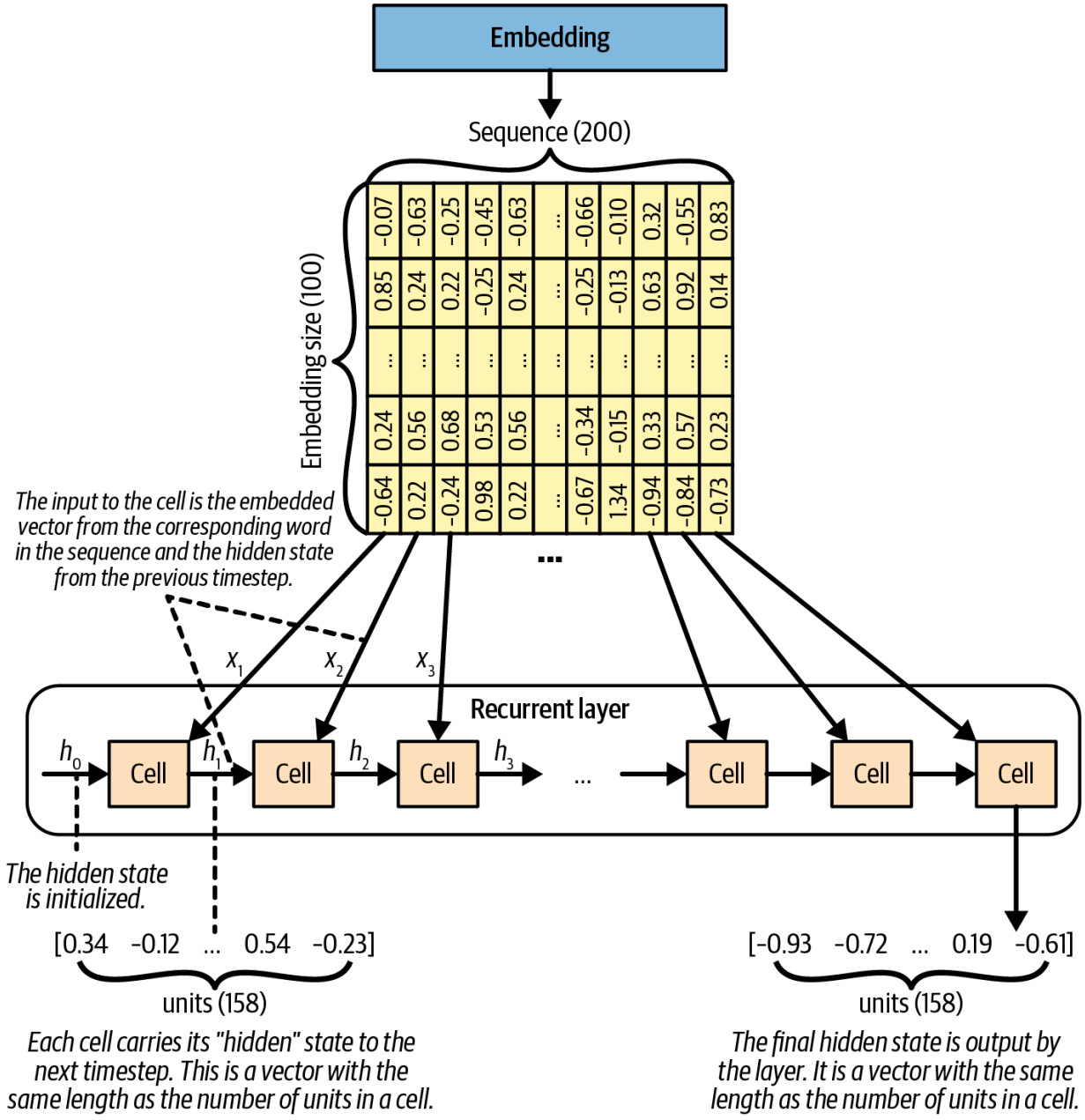

Figure 10.6: How a single sequence flows through a recurrent layer

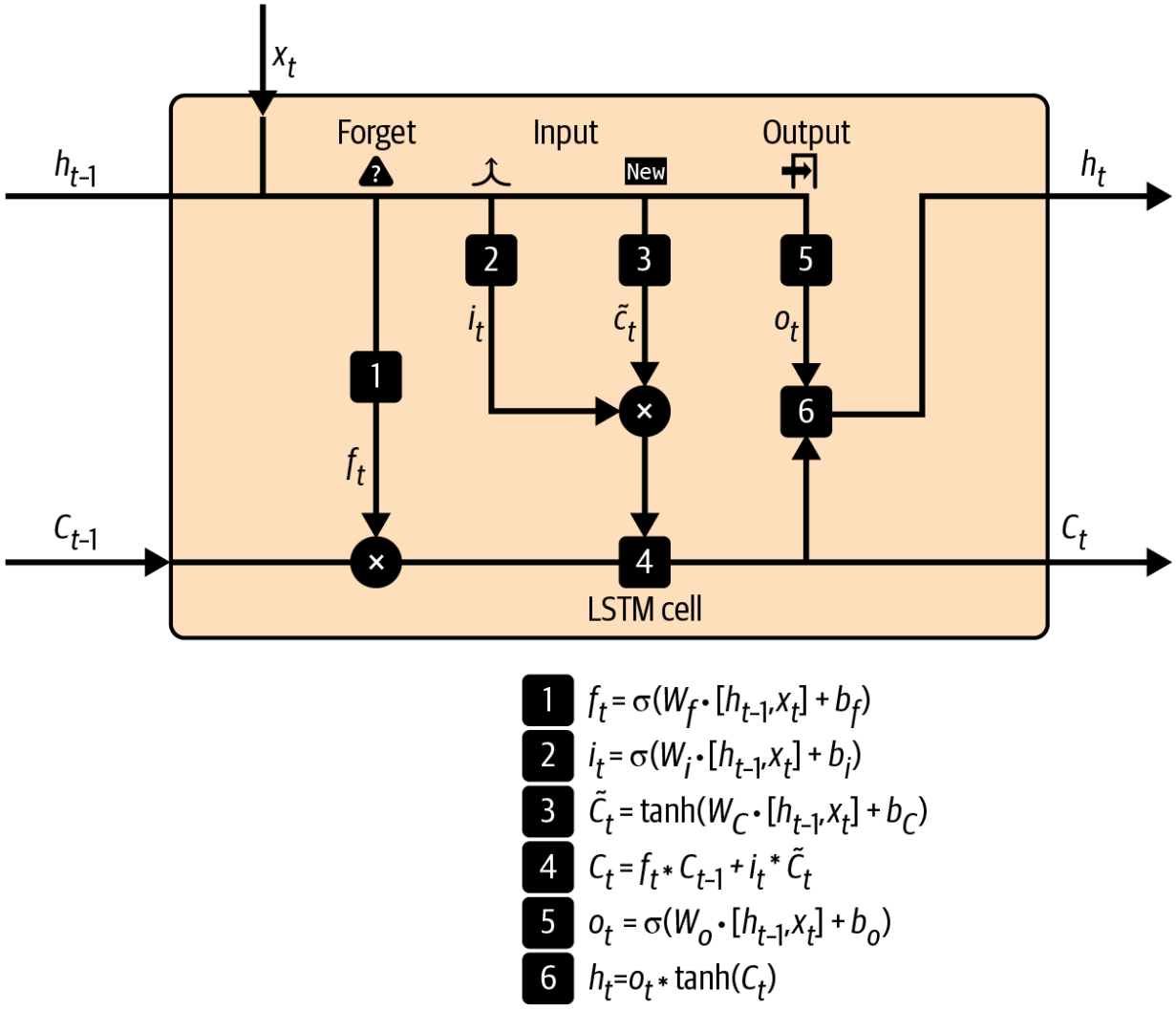

Figure 10.7: An LSTM Cell

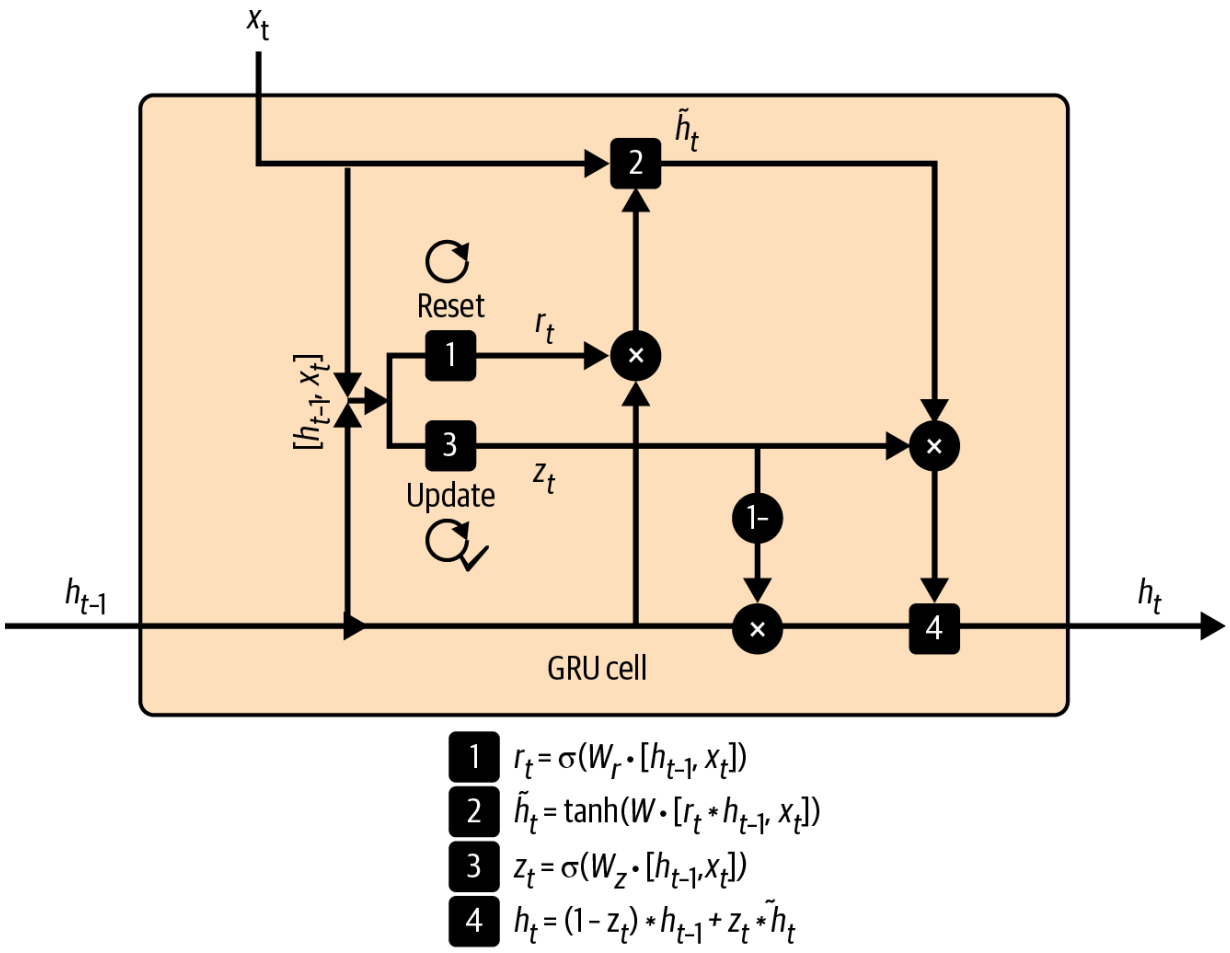

Figure 10.8: A single GRU cell

6. o_t is multiplied element-wise with the updated cell state, C_t, after a tanh activation has been applied to produce the new hidden state, h_t.

GRU Another type of commonly used Recurrent Neural Network (RNN) layer is the GRU (Figure 10.8). The key differences from the LSTM unit are as follows:

1. The forget and input gates are replaced by reset and update gates.
2. There is no cell state or output gate, only a hidden state that is output from the cell.

10.5 Normalizing Flow Models

Normalizing flows share similarities with both autoregressive models and VAEs. Like autoregressive models, normalizing flows are able to explicitly and tractably model the data-generating distribution $\mathcal{P}_x$. Like VAEs, normalizing flows attempt to map the data into a simpler distribution, such as a Gaussian distribution. The key difference is that normalizing flows place a constraint on the form of the mapping function, so that it is invertible and can therefore be used to generate new data points.

Summary A normalizing flow model is an invertible function defined by a neural network that allows us to directly model the data density via a change of variables. In the general case, the change of variables equation requires us to calculate a highly complex Jacobian determinant, which is impractical for all but the simplest of examples.

To sidestep this issue, the RealNVP model restricts the form of the neural network, such that it adheres to the two essential criteria: it is invertible and has a Jacobian determinant that is easy to compute.

It does this through stacking coupling layers, which produce scale and translation factors at each step. Importantly, the coupling layer masks the data as it flows through the network, in a way that ensures that the Jacobian is lower triangular and therefore has a simple-to-compute determinant. Full visibility of the input data is achieved through flipping the masks at each layer.

By design, the scale and translation operations can be easily inverted, so that once the model is trained it is possible to run data through the network in reverse. This means that we can target the forward transformation process toward a standard Gaussian, which we can easily sample from. We can then run the sampled points backward through the network to generate new observations.

The RealNVP paper also shows how it is possible to apply this technique to images, by using convolutions inside the coupling layers, rather than densely connected layers. The GLOW paper extended this idea to remove the necessity for any hardcoded permutation of the masks. The FFJORD model introduced the concept of continuous time normalizing flows, by modeling the transformation process as an ODE defined by a neural network.

Overall, we have seen how normalizing flows are a powerful generative modeling family that can produce high-quality samples, while maintaining the ability to tractably describe the data density function. [Fos22].

10.6 Energy-Based Models

Energy-based models attempt to model the true data-generating distribution using a *Boltzmann distribution* (10.2) where $E(x)$ is know as the *energy function* of an observation x.

$$p(\mathbf{x}) = \frac{e^{-E(\mathbf{x})}}{\int_{\hat{\mathbf{x}} \in \mathbf{X}} e^{-E(\hat{\mathbf{x}})}} \tag{10.2}$$

10.7 Diffusion Models

Denoising Diffusion Models (DDM) The core idea behind a denoising diffusion model is simple—we train a deep learning model to denoise an image over a series of very small steps.

The Forward Diffusion Process Suppose we have an image $\mathbf{x}_0$ that we want to corrupt gradually over a large number of steps ($T = 1000$), so that eventually it is indistinguishable from standard Gaussian noise (i.e., $\mathbf{x}_T shouldhavezeormeanandunitvariance$).

We can define a function q that adds a small amount of Gaussian noise with variance β_t to an image $\mathbf{x}_{t-1}$ to generate a new image $\mathbf{x}_t$. If we keep applying this function, we will generate a sequence of progressively noisier images ($x_0, \ldots x_T$), as shown in Figure 10.9.

We can write this update process mathematically as follows (here, ϵ_{t-1} is a standard Gaussian with zeor mean and unit variance):

$$\mathbf{x}_t = \sqrt{1 - \beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon_{t-1}.$$

Note that we also scale the input image $\mathbf{x}_{t-1}$, to ensure that the variance of the output image $\mathbf{x}_t$ remains constant over time. This way, if we normalize our original image $\mathbf{x}_0$ to have zeor mean and unit variance, the $\mathbf{x}_t$ will approximate a standard Gaussian distribution for large enough T, by induction, as follows.

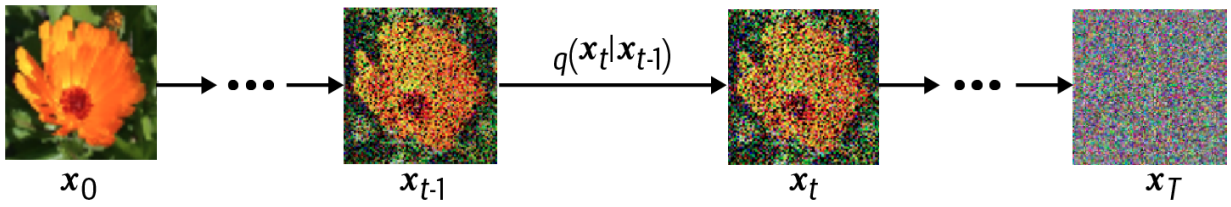

Figure 10.9: The forward diffusion process

If we assume that $\mathbf{x}_{t-1}$ has zeor mean and unit variance then $\sqrt{1-\beta_t}\mathbf{x}_{t-1}$ will have variance $1-\beta_t$ and $\sqrt{\beta_t}\epsilon_{t-1}$ will have variance β_t, using the rule that $\text{Var}(aX) = a^2\text{Var}(X)$. Adding these together, we obtain a new distribution $\mathbf{x}_t$ with zero mean and variance $1-\beta_t+\beta_t = 1$, using the rule that $\text{Var}(X+Y) = \text{Var}(X)\,\text{Var}(Y)$ for independent X and Y. Therefore, if $\mathbf{x}_0$ is normalized to a zero mean and unit variance, then we guarantee that this is also true for all $\mathbf{x}_t$, including the final image $\mathbf{x}_T$, which will approximate a standard Gaussian distribution. This is exactly what we need, as we want to be alble to easily sample $\mathbf{x}_T$ and then apply a reverse diffusion process through our trained neural network model.

In other words, our forward nosing process q can also be written as follows:

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}).$$

The Reparmeaterization Trick It would also be useful to be able to jump straight from an image $\mathbf{x}_0$ to any noised version of the image $\mathbf{x}_t$ without having to go through t applications of q.

If we define $\alpha = 1-\beta_t$ and $\overline{\alpha}_t = \prod_{i=1}^{t}\alpha_i$, then we can write the following:

$$\begin{aligned}
\mathbf{x}_t &= \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\epsilon_{t-1} \\
&= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\epsilon \\
&= \dots \\
&= \sqrt{\overline{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\overline{\alpha}_t}\epsilon.
\end{aligned}$$

Note that the second line uses the fact that we can add two Gaussians to obtain a new Gaussian. We therefore have have a way to jump from the original image $\mathbf{x}_0$ to nay step of the forward diffusion process $\mathbf{x}_t$. Moverover, we can define the diffusion schedule using the $\overline{\alpha}_t$values, instead of the original β_t values, with the interpretation that $\overline{\alpha}_t$ is the variance due to the signal (the original image, $\mathbf{x}_0$) and $1-\overline{\alpha}_t$ is the variance due to the noise (ϵ).

The forward diffusion process q can therefore also be written as follows:

$$q\left(\mathbf{x}_t \mid \mathbf{x}_0\right) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{\overline{\alpha}_t}\mathbf{x}_0, \left(1-\overline{\alpha}_t\right)\mathbf{I}\right).$$

Diffusion Schedule Notice that we are also free to choose a different β_t at each timestep—they don't all have be the same. How the β_t (or $\hat{\alpha}_t$) values change with t is called the diffusion schedule.

In the original paper [Hoo+21], the authors chose a linear diffusion schedule for β_t that is, β_t increases linearly with t, from $\beta_1 = 0.0001$ to $\beta_T = 0.02$. This ensures that in the early stages of the noising process we take smaller noising steps than in the later stages, when the image is already very noisy.

In a later paper it was found that a cosine diffusion schedule outperformed the linear schedule from the original paper [Hoo+21]. A cosine schedule defines the following values of $\hat{\alpha}_t$:

$$\hat{\alpha}_t = \cos\left(\frac{\pi t}{2T}\right).$$

The updated equation is therefore as follows (using the trigonometric identity $\cos^2(x) + \sin^2 = 1$)

$$\mathbf{x}_t = \cos\left(\frac{t}{T} \cdot \frac{\pi}{2}\right)\mathbf{x}_0 + \sin\left(\frac{t}{T} \cdot \frac{\pi}{2}\right)\epsilon.$$

This equation is a simplified version of the actual cosine diffusion schedule used in the paper. The authors also add an offset term and scaling to prevent the noising steps from being too small at the beginning of the diffusion process. We can code up the cosine and offset cosine diffusion schedules as shown below:

```
def cosine_diffusion_schedule(diffusion_times):
    signal_rates = tf.cos(diffusion_times * math.pi / 2)
    noise_rates = tf.sin(diffusion_times * math.pi / 2)
    return noise_rates, signal_rates

def offset_cosine_diffusion_schedule(diffusion_times):
    min_signal_rate = 0.02
    max_signal_rate = 0.95
    start_angle = tf.acos(max_signal_rate)
    end_angle = tf.acos(min_signal_rate)
    diffusion_angles = start_angle + diffusion_times * (end_angle - start_angle)
    signal_rates = tf.cos(diffusion_angles)
    noise_rates = tf.sin(diffusion_angles)
```

The Reverse Diffusion Process There are many similarities between the the reverse diffusion proceess and the decoder of a variational autoencoder.

Sampling from the Denoising Diffusion Model In order to sample images from trained model, we need to apply the reverse diffusion process, that is, we need to start with random noise adn use the model to gradually undo the noise, until we are left with recognizable picture of a flower. However, we do not want to undo the noise all in one go in which predicting an image from pure random noise in one shot is clearly not going to work. We would rather mimic the forward process and undo the predicted noise gradually over many small steps, to allow the model to adjust to its own predictions.

To achieve this, we can jump from x_t to x_{t-1} in two steps, first by using our model's noise prediction to calculate an estimate for the original image x_0 and the by replying the predicted noise to this image, but only over $t-1$ timesteps, to produce x_{t-1} (Figure 10.10).

If we repeat this process over a number of steps, we'll eventually get back to an estimate for x_0 that has been guided gradually over many small steps. In fact, we are free to choose the number of steps we take, and crucially, it doesn't have to be the same as the large number of steps in the training noising process (i.e., 1.000). It can be much smaller—in this example we choose 20.

The following equation [SME22] this process mathematicaly:

$$\mathbf{x}_{t-1} = \sqrt{\overline{\alpha}_{t-1}}\left(\frac{\mathbf{x}_t - \sqrt{1-\overline{\alpha}_t}\epsilon_\theta^{(t)}(\mathbf{x}_t)}{\sqrt{\overline{\alpha}_t}}\right) + \sqrt{1-\overline{\alpha}_{t-1}-\sigma_t^2}\cdot\epsilon_\theta^{(t)}(\mathbf{x}_t) + \sigma_t\epsilon_t.$$

Let's break this down. The first term inside the brackets on the righthand side of the equation is the estimated image x_0, calculated using the noise predicted by our network $\epsilon_\theta^{(t)}$. We then scale this by the $t-1$

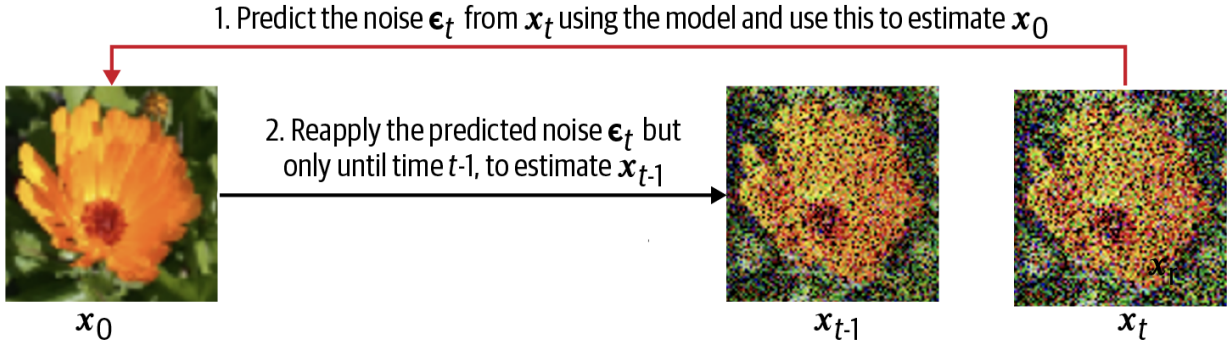

Figure 10.10: One step of the sampling process for our diffusion model [Fos22]

signal rate $\sqrt{\overline{\alpha}_t - 1}$ and reapply the predicted noise, but this time scaled by the $t-1$ noise rate $\sqrt{1-\overline{\alpha}_{t-1}\sigma_t^2}$. Additional Gaussian random noise $\sigma_t\epsilon_t$ is also added, with the factors σ_t determining how random we want our generation process to be.

The special case $\sigma_t = 0$ for all t corresponds to a type of model known as a Denoising Diffusion Implicit Models (DDIM), introduced by Song, Meng, and Ermon [SME22]. With DDIM, the generation process is entirely deterministic, which is the same random noise input will always give the same output. This is desirable as the we have a well-defined mapping between samples from the latent space and the generated output in pixel space.

Analysis of the Diffusion Model We can see in Figure 10.11 that the quality of the generations does indeed improve with the number of diffusion steps. With one giant leap from the initial sampled noise, the model can only predict a hazy blob of color. With more steps, the model is able to refine and sharpen its generations. However, the time taken to generate the images scales linearly with the number of diffusion steps, so there is a trade-off. There is minimal improvement between 20 and 100 diffusion steps, so we choose 20 as a reasonable compromise between quality and speed in this example [Fos22].

The reverse diffusion process is parameterized by a U-Net that tries to predict the noise at each timestep, given the noised image and the noise rate at that step. A U-Net consists of DownBlocks that increase the number of channels while reducing the size of the image and UpBlocks that decrease the number of channels while increasing the size. The noise rate is encoded using sinusoidal embedding [Fos22].

Sampling from the diffusion model is conducted over a series of steps. The U-Net is used to predict the noise added to a given noised image, which is then used to calculate an estimate for the original image. The predicted noise is then reapplied using a smaller noise rate. This process is repeated over a series of steps (which may be significantly smaller than the number of steps used during training), starting from a random point sampled from a standard Gaussian noise distribution, to obtain the final generation [Fos22].

We saw how increasing the number of diffusion steps in the reverse process improves the image generation quality, at the expense of speed. We also performed latent space arithmetic in order to interpolate between two images [Fos22].

10.8 Transformers

A recurrent layer tries to build up a generic hidden state that captures an overall representation of the input at each timestep. A weakness of this approach is that many of the words that have already been incorporated into the hidden vector will not be directly relevant to the immediate task at hand (e.g., predicting the next word), as we have just seen. Attention heads do not suffer from this problem, because they can pick and

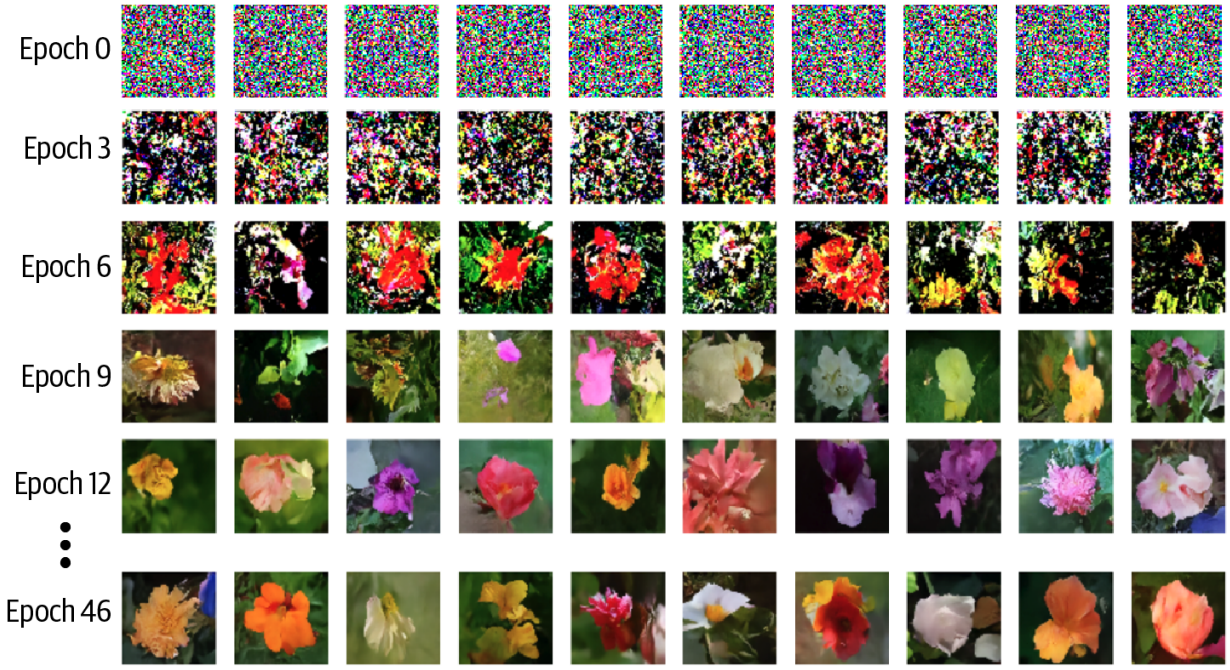

Figure 10.11: Samples from the diffusion model at different epochs of the training process [Fos22]

choose how to combine information from nearby words, depending on the context.

The query (Q) can be thought of as a representation of the current task at hand (e.g., "What word follows too?"). In this example, it is derived from the embedding of the word too, by passing it through a weights matrix W_Q to change the dimensionality of the vector from d_e to d_k (Figure 10.12).

The key vectors (K) are representations of each word in the sentence—you can think of these as descriptions of the kinds of prediction tasks that each word can help with. They are derived in a similar fashion to the query, by passing each embedding through a weights matrix W_K to change the dimensionality of each vector from d_e to d_k. Notice that the keys and the query are the same length (d_k).

Inside the attention head, each key is compared to the query using a dot product between each pair of vectors (QK^T). This is why the keys and the query have to be the same length. The higher this number is for a particular key/query pair, the more the key resonates with the query, so it is allowed to make more of a contribution to the output of the attention head. The resulting vector is scaled by $\sqrt{d_k}$ to keep the variance of the vector sum stable (approximately equal to 1), and a softmax is applied to ensure the contributions sum to 1. This is a vector of *attention weights*.

The value vectors (V) are also representations of the words in the sentence—you can think of these as the unweighted contributions of each word. They are derived by passing each embedding through a weights matrix W_V to change the dimensionality of each vector from $d_e$to d_v. Notice that the value vectors do not necessarily have to have the same length as the keys and query (but often do, for simplicity).

The value vectors are multiplied by the attention weights to give the attention for a given Q, K, and V, as shown in Equation 10.3.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{10.3}$$

Multi-Head Attention The concatenated outputs are passed through one final weights matrix W_O to project the vector into the desired output dimension, which in our case is the same as the input dimension of

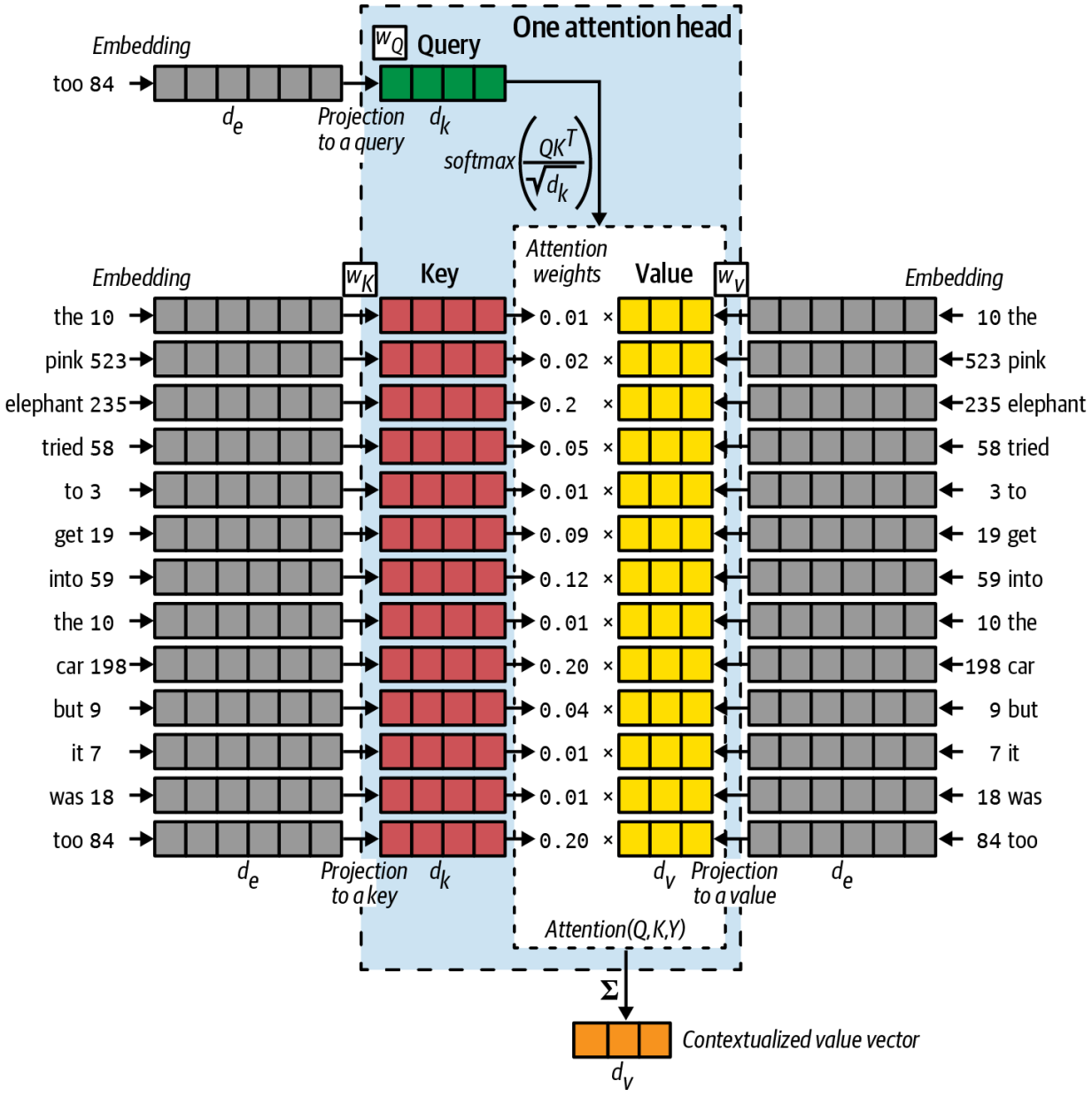

Figure 10.12: The mechanics of an attention head [Fos22]

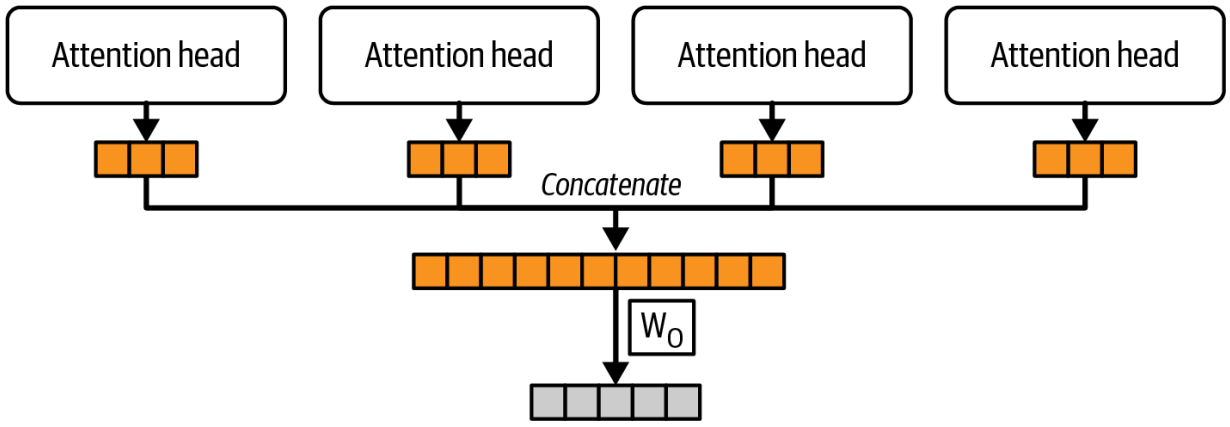

Figure 10.13: The Multi-Head Attention

the query (d_e), so that the layers can be stacked sequentially on top of each other (Figure 10.13).

Causal Masking Causal masking is only required in decoder Transformers such as GPT, where the task is to sequentially generate tokens given previous tokens. Masking out future tokens during training is therefore essential (Figure 10.14).

Other flavors of Transformer (e.g., encoder Transformers) do not need causal masking, because they are not trained to predict the next token [Vas+17]. For example Google's Bidirectional Encoder Representations from Transformers (BERT) predicts masked words within a given sentence, so it can use context from both before and after the word in question. We will explore the different types of Transformers in more detail at the end of the chapter.

This concludes our explanation of the multihead attention mechanism that is present in all Transformers. It is remarkable that the learnable parameters of such an influential layer consist of nothing more than three densely connected weights matrices for each attention head (W_Q, W_K, W_V) and one further weights matrix to reshape the output (W_O). There are no convolutions or recurrent mechanisms at all in a multihead attention layer!

The Transformer Block (Figure 10.15) In contrast, layer normalization in a Transformer block normalizes each position of each sequence in the batch by calculating the normalizing statistics across the channels. It is the complete opposite of batch normalization, in terms of how the normalization statistics are calculated. A diagram showing the difference between batch normalization and layer normalization is shown in Figure 10.16.

Layer normalization was used in the original GPT paper and is commonly used for text-based tasks to avoid creating normalization dependencies across sequences in the batch Figure 10.16. However, recent work such as Shen et al. [She+20] challenges this assumption, showing that with some tweaks a form of batch normalization can still be used within Transformers, outperforming more traditional layer normalization.

Positional Embeddings To construct the joint token–position encoding, the token embedding is added to the positional embedding, as shown in Figure 10.17. This way, the meaning and position of each word in the sequence are captured in a single vector.

Summary Attention heads can be grouped together to form what is known as a multihead attention layer. These are then wrapped up inside a Transformer block, which includes layer normalization and skip connections around the attention layer. Transformer blocks can be stacked to create very deep neural networks.

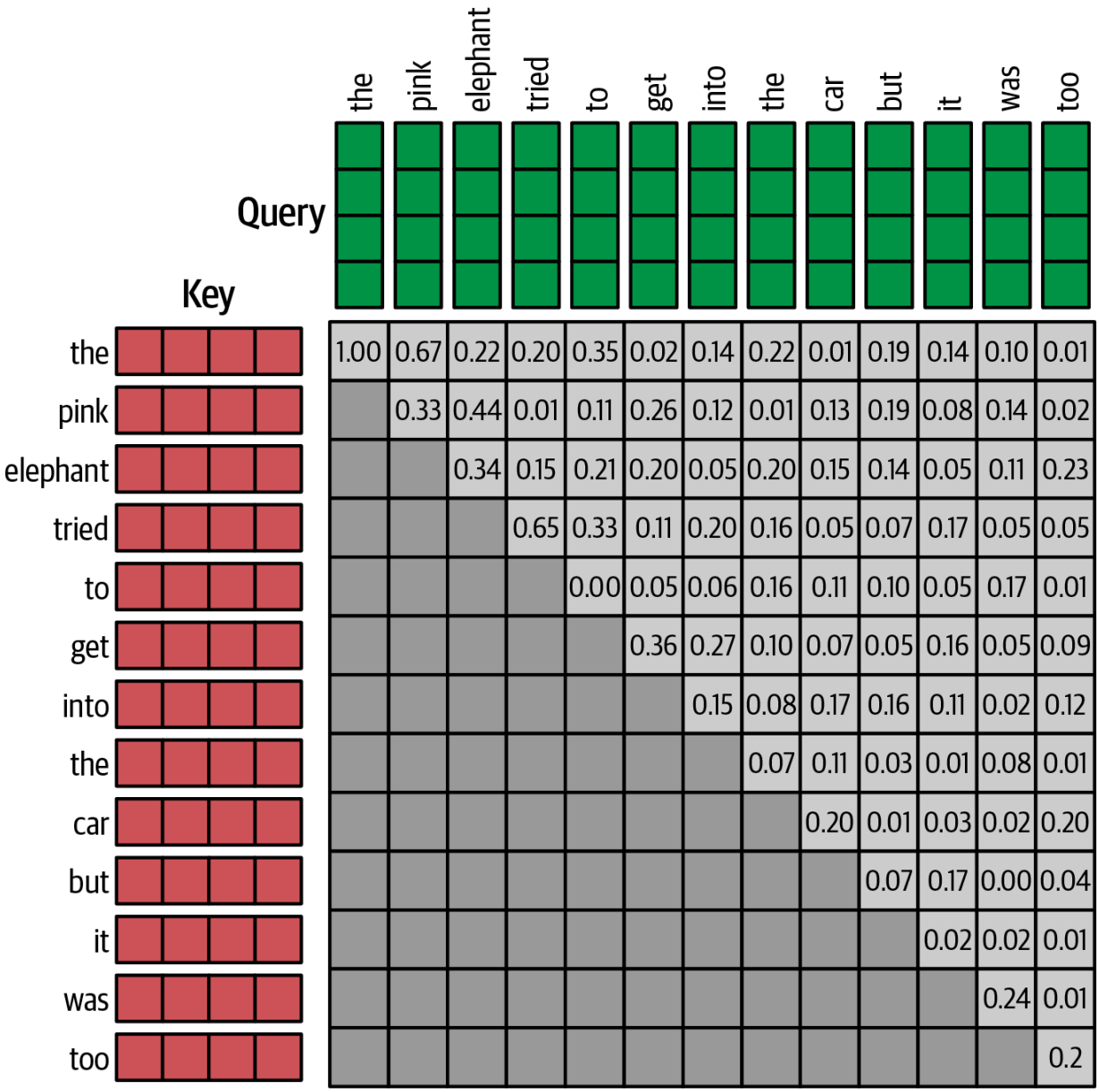

Figure 10.14: Matrix calculation of the attention scores for a batch of input queries, using a causal attention mask to hide keys that are not available to the query (because they come later in the sentence)

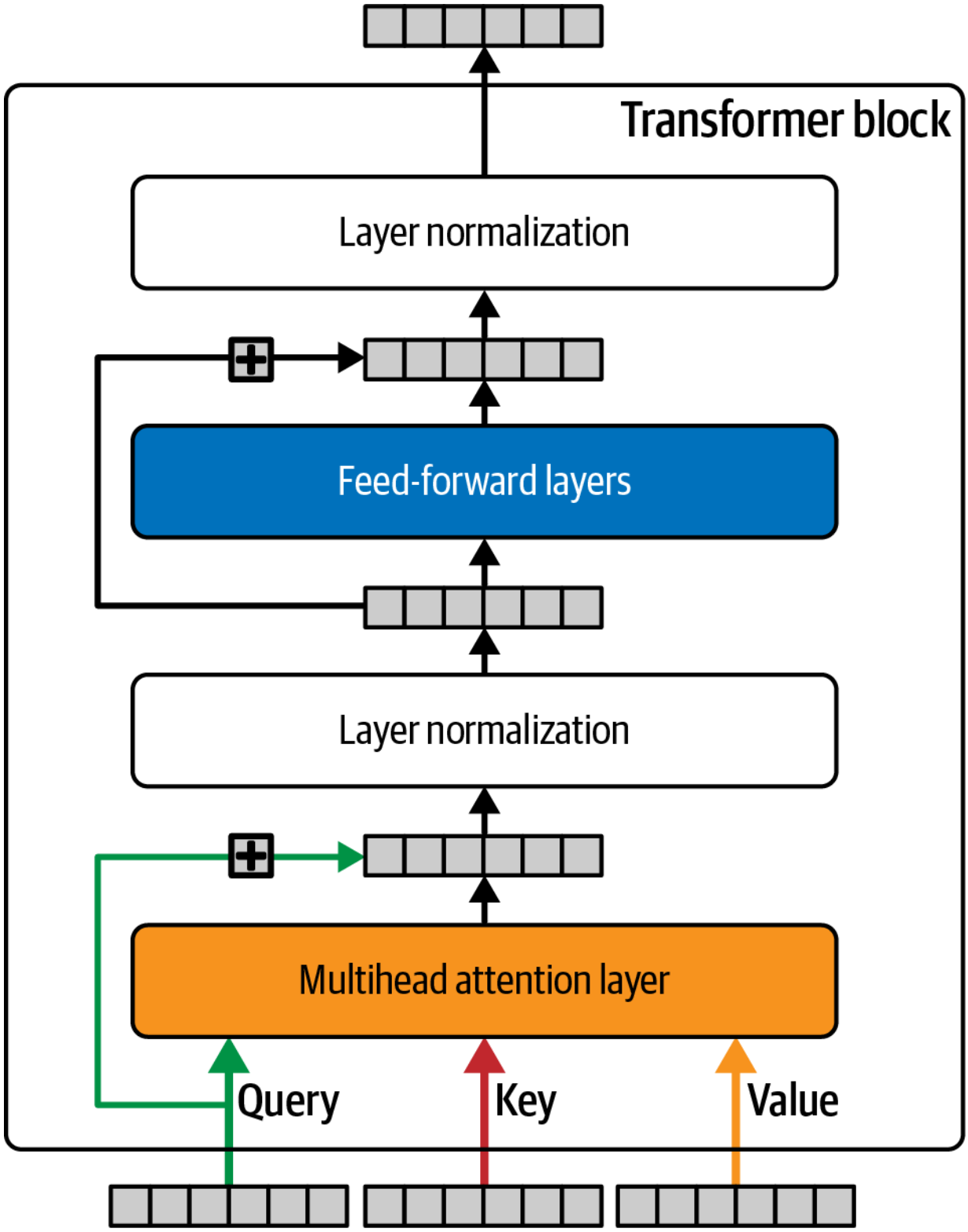

Figure 10.15: A Transformer Block [Fos22]

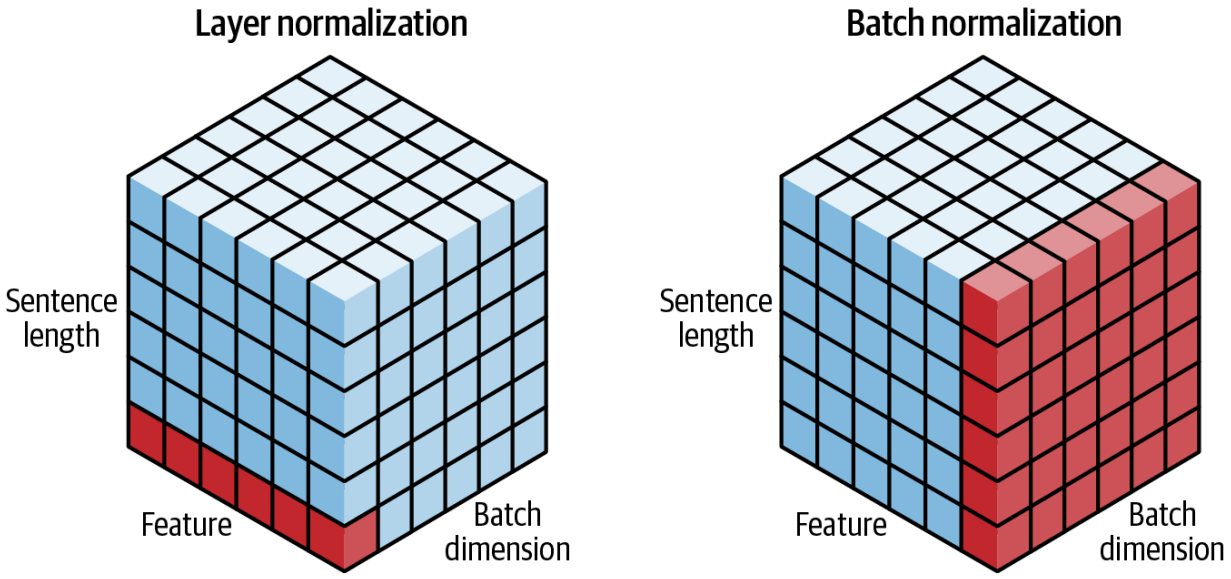

Figure 10.16: Layer normalization versus batch normalization—the normalization statistics are calculated across the blue cells [She+20]

Causal masking is used to ensure that Generative Pre-trained Transformer (GPT) cannot leak information from downstream tokens into the current prediction. Also, a technique known as positional encoding is used to ensure that the ordering of the input sequence is not lost, but instead is baked into the input alongside the traditional word embedding.

When analyzing the output from GPT, we saw it was possible not only to generate new text passages, but also to interrogate the attention layer of the network to understand where in the sentence it is looking to gather information to improve its prediction. GPT can access information at a distance without loss of signal, because the attention scores are calculated in parallel and do not rely on a hidden state that is carried through the network sequentially, as is the case with recurrent neural networks.

We saw how there are three families of Transformers (encoder, decoder, and encoder-decoder) and the different tasks that can be accomplished with each.

The Illustrated Transformer [alammar2018illustrated] Matrix calculation of self-attention is shown in Figure 10.19.

Finally, since we're dealing with matrices, we can condense steps two through six in one formula to calculate the outputs of the self-attention layer.

Let me try to put them all in one visual so we can look at them in one place (Figure 10.20).

10.9 Advanced GANs

The GAN Zoo GitHub repository is a great resource for exploring the different types of GANs that have been proposed over the years.

Progressive Growing Generative Adversarial Networks (ProGAN) [Kar+18] ProGAN is a technique developed by NVIDIA Labs in 2017 to improve both the speed and stability of GAN training. Instead of immediately training a GAN on fullresolution images, the ProGAN paper suggests first training the generator and

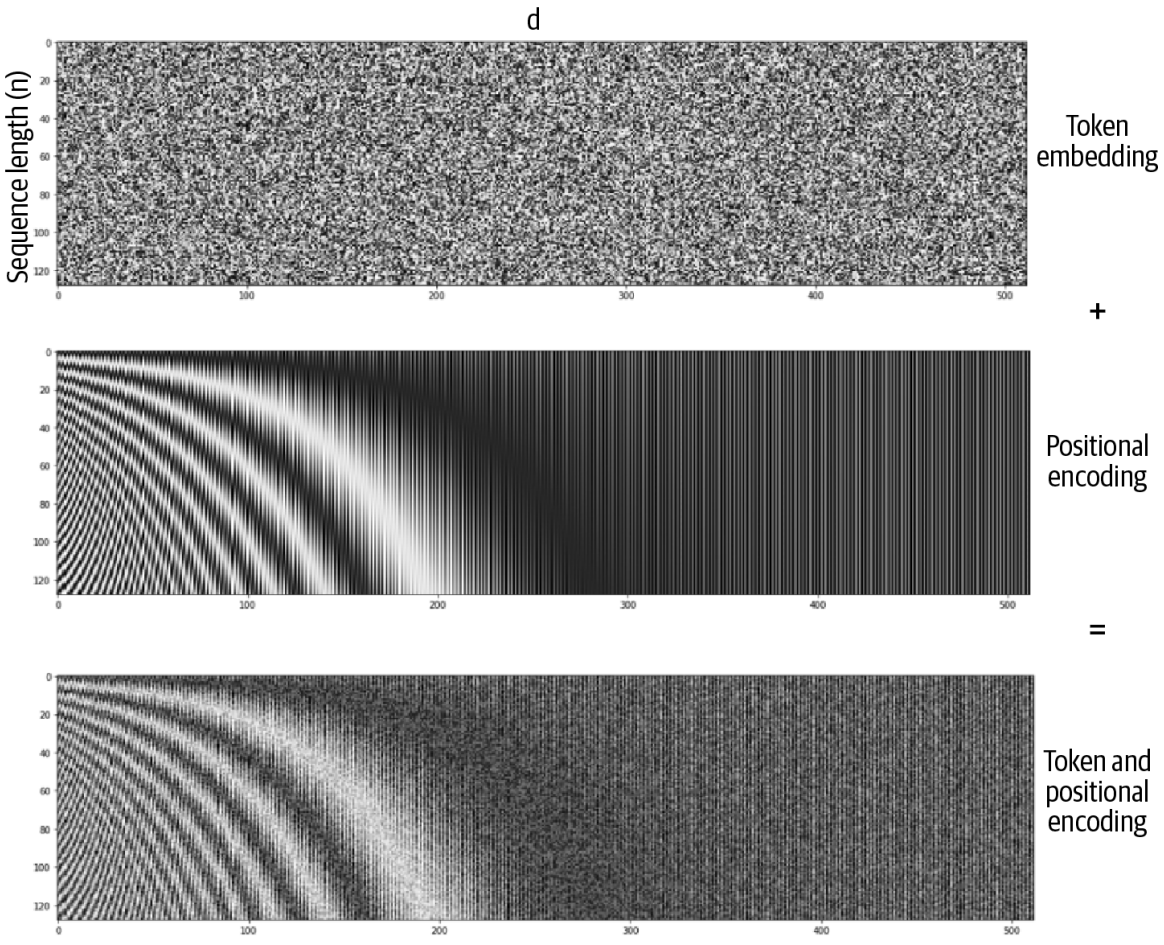

Figure 10.17: The token embeddings are added to the positional embeddings to give the token position encoding Figure 10.17

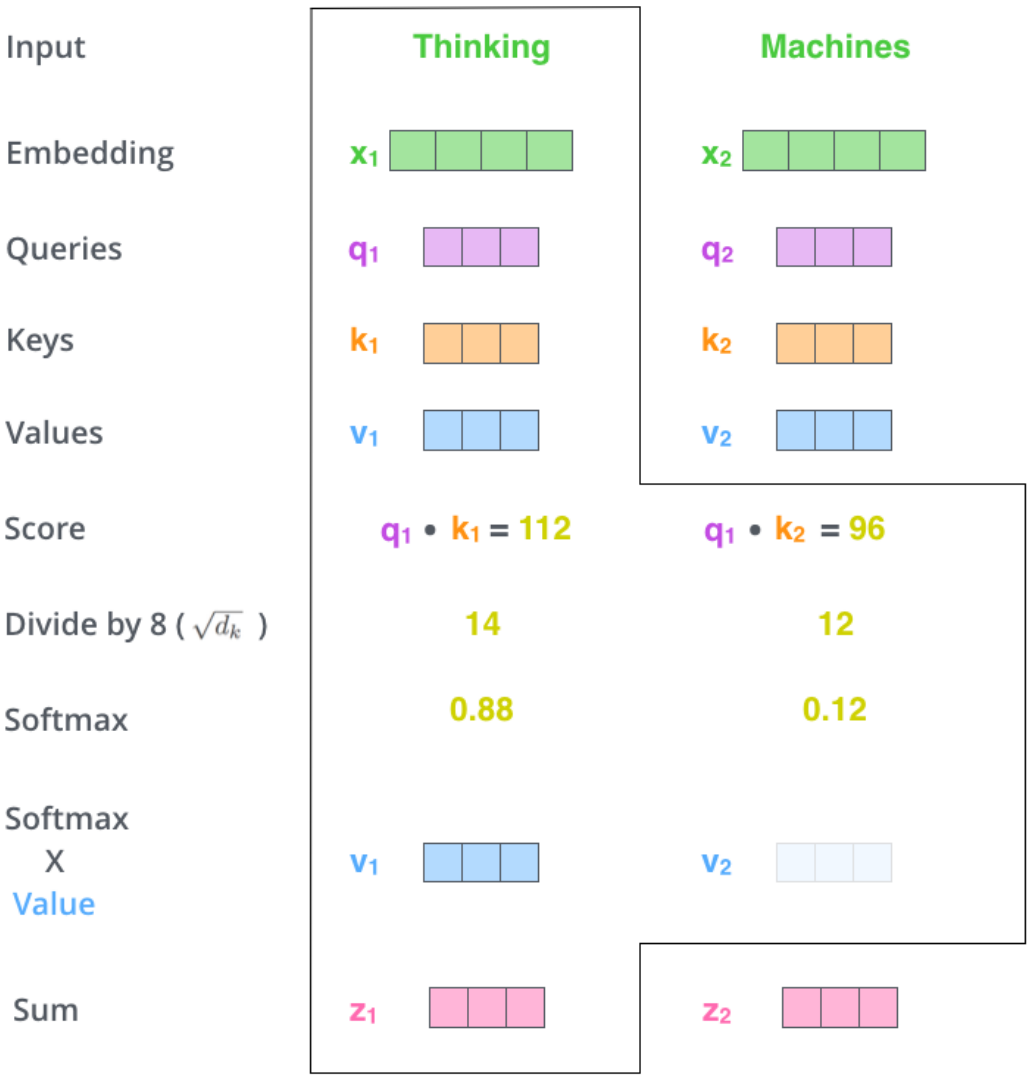

Figure 10.18: Self-attention Output

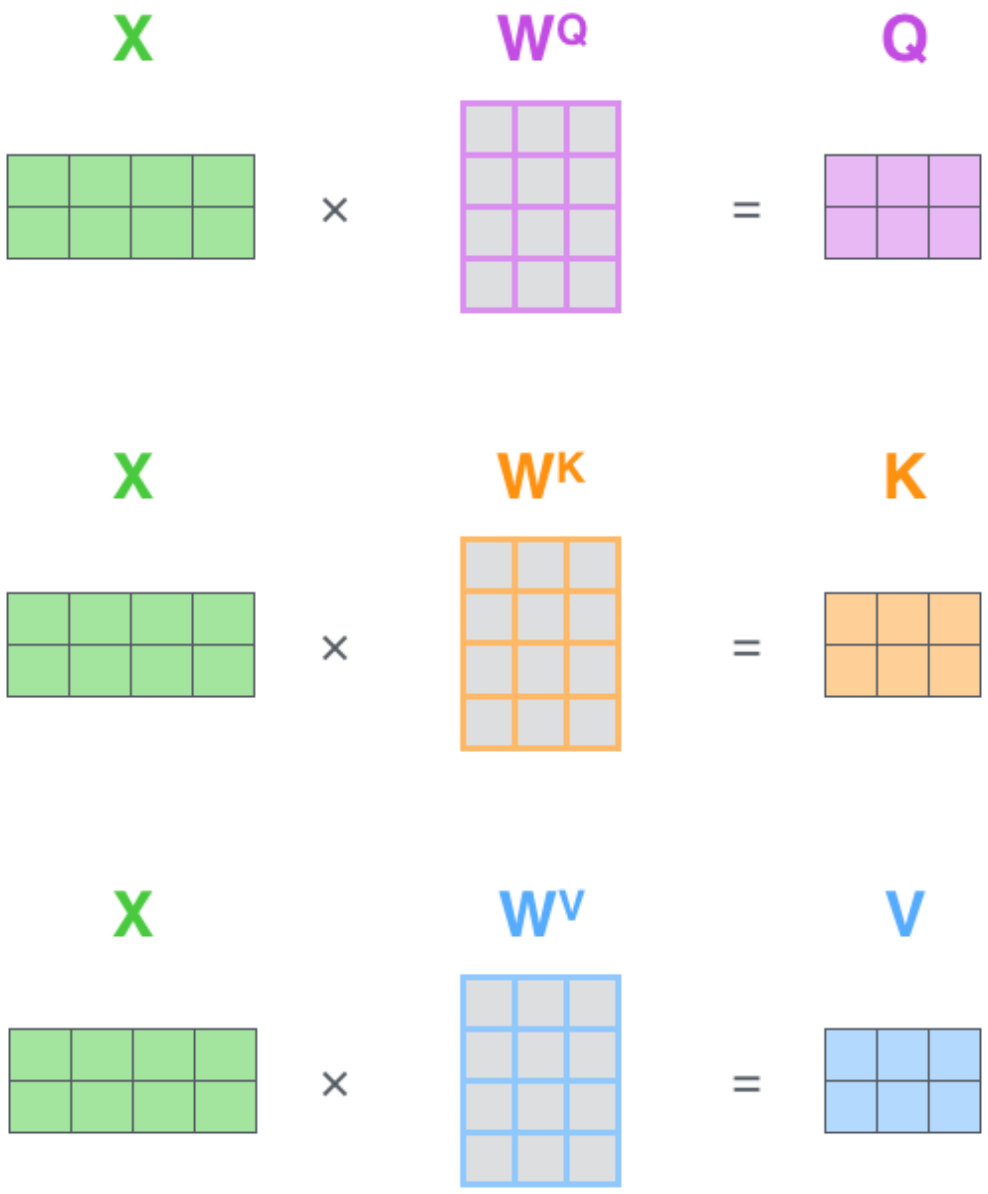

Figure 10.19: Self-attention Matrix Calculation

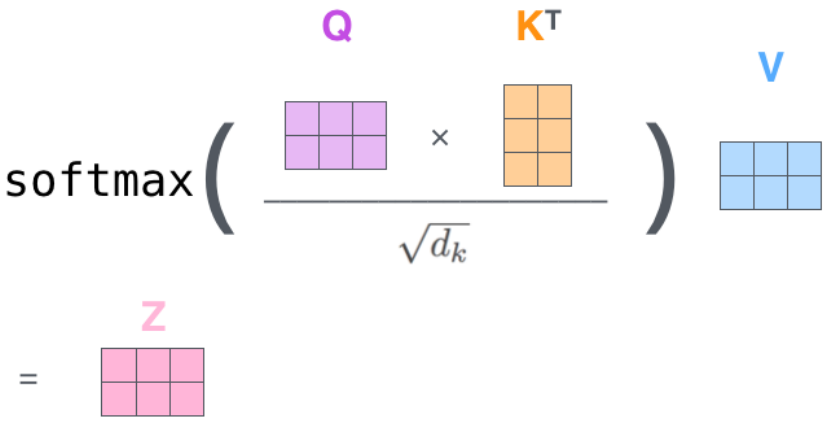

Figure 10.20: Self-attention Matrix Calculation

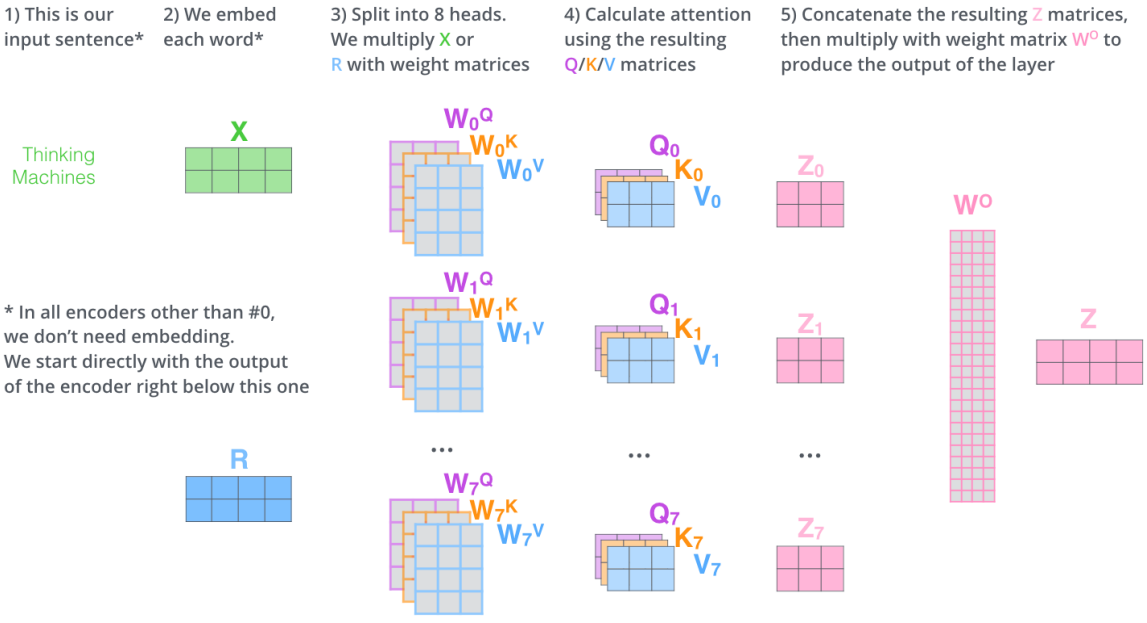

Figure 10.21: Transformer Multi-headed Self-attention Recap

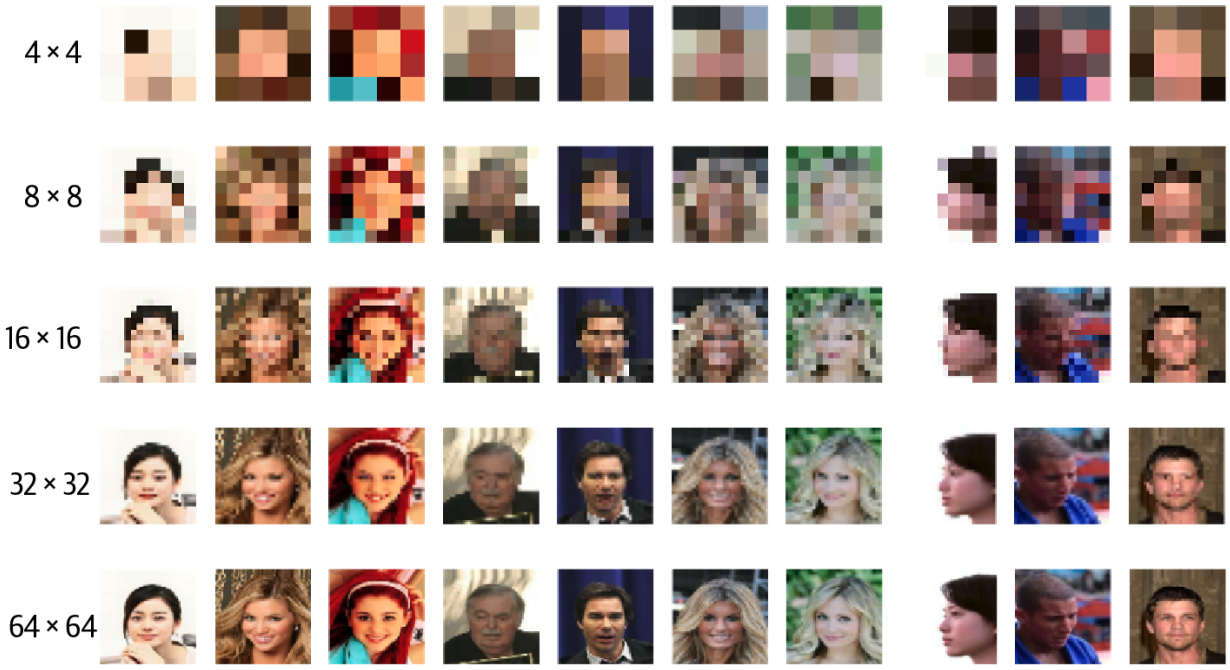

Figure 10.22: Images in the dataset can be compressed to lower resolution using interpolation

discriminator on low-resolution images of, say, 4×4 pixels and then incrementally adding layers throughout the training process to increase the resolution.

In a normal GAN, the generator always outputs full-resolution images, even in the early stages of training. It is reasonable to think that this strategy might not be optimal—the generator might be slow to learn high-level structures in the early stages of training, because it is immediately operating over complex, high-resolution images. Wouldn't it be better to first train a lightweight GAN to output accurate low-resolution images and then see if we can build on this to gradually increase the resolution?

This simple idea leads us to progressive training, one of the key contributions of the ProGAN paper. The ProGAN is trained in stages, starting with a training set that has been condensed down to 4×4 pixel images using interpolation, as shown in Figure 10.22.

Features:

- Progressive training
- Minibatch standard deviation
- Equalized learning rate
- Pixelwise feature vector normalization
- Scaled images and pixel values

StyleGAN [KLA18] StyleGAN is a GAN architecture from 2018 that builds on the earlier ideas in the ProGAN paper. In fact, the discriminator is identical; only the generator is changed (Figure 10.23).

Features:

- The Mapping Network
- The Adaptive Instance Normalization Layer
- The Style Mixing Regularization

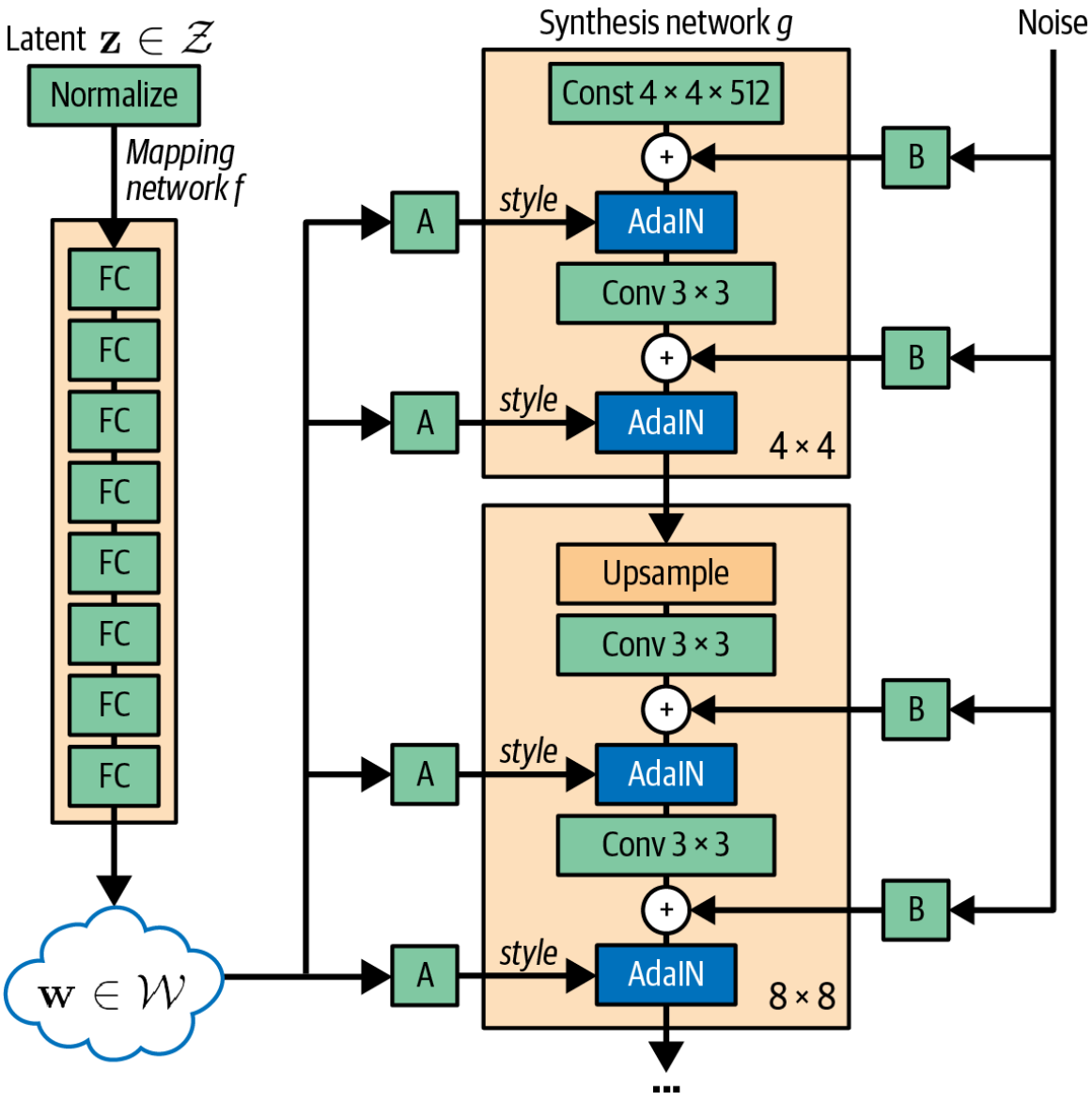

Figure 10.23: The StyleGAN Generator Architecture

- Stochastic Variation

StyleGAN2

- Weight Modulation and Demodulation
- Path Length Regularization
- No Progressive Growing

Summary We started by exploring the concept of progressive training that was pioneered in the 2017 ProGAN paper [Kar+18]. Several key changes were introduced in the 2018 StyleGAN paper that gave greater control over the image output, such as the mapping network for creating a specific style vector and synthesis network that allowed the style to be injected at different resolutions. Finally, StyleGAN2 replaced the adaptive instance normalization of StyleGAN with weight modulation and demodulation steps, alongside additional enhancements such as path regularization. The paper also showed how the desirable property of gradual resolution refinement could be retained without having to the train the network progressively.

We also saw how the concept of attention could be built into a GAN, with the introduction of Self-Attention Generative Adversarial Networks (SAGAN) in 2018 [Zha+19]. This allows the network to capture long-range dependencies, such as similar background colors over opposite sides of an image, without relying on deep convolutional maps to spread the information over the spatial dimensions of the image. BigGAN was an extension of this idea that made several key changes and trained a larger network to improve the image quality further [BDS19].

In the Vector Quantized GAN (VQ-GAN) paper, the authors show how several different types of generative models can be combined to great effect. Building on the original Vector Quantized Variational Autoencoder (VQ-VAE) paper that introduced the concept of a VAE with a discrete latent space, VQ-GAN additionally includes a discriminator that encourages the VAE to generate less blurry images through an additional adversarial loss term. An autoregressive Transformer is used to construct a novel sequence of code tokens that can be decoded by the VAE decoder to produce novel images. The Vision Transformer Vector Quantized GAN (ViT VQ-GAN) paper extends this idea even further, by replacing the convolutional encoder and decoder of VQ-GAN with Transformers.

10.10 Music Generation

One of the most popular techniques for music generation is the Transformer, as music can be thought of as a sequence prediction problem. These models have been adapted to generate music by treating musical notes as a sequence of tokens, similar to words in a sentence. The Transformer model learns to predict the next note in the sequence based on the previous notes, resulting in a generated piece of music.

Multi-track Sequential Generative Adversarial Networks (MuseGAN) takes a totally different approach to generating music [Don+17]. Unlike Transformers, which generate music note by note, MuseGAN generates entire musical tracks at once by treating music as an image, consisting of a pitch axis and a time axis. Moreover, MuseGAN separates out different musical components such as chords, style, melody, and groove so that they can be controlled independently.

Sine Positional Embeddings Using simple Embedding layer to encode the position of each token, effectively mapping each integer position to a distinct vector that was learned by the model. We therefore needed to define a maximum length (N) that the sequence could be and train on this length of sequence. The downside to this approach is that it is then impossible to extrapolate to sequences that are longer than this maximum length. You would have to clip the input to the last N tokens, which isn't ideal if you are trying to generate long-form content.

To circumvent this problem, we can switch to using a different type of embedding called a sine position embedding. This is similar to the embedding to encode the noise variances of the diffusion model. Specifically, the following function is used to convert the position of the word (pos) in the input sequence into a unique vector of length d:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right).$$

$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{(2i+1)/d}}\right).$$

For small i, the wavelength of this function is short and therefore the function value changes rapidly along the position axis. Larger values of i create a longer wavelength. Each position thus has its own unique encoding, which is a specific combination of the different wavelengths.

Notice that this embedding is defined for all possible position values. It is a deterministic function (i.e., it isn't learned by the model) that uses trigonometric functions to define a unique encoding for each possible position.

Multiple Inputs and Outputs There are many ways of handling the dual stream of inputs. We could create tokens that represent each note–duration pair and then treat the sequence as a single stream of tokens. However, this has the downside of not being able to represent note–duration pairs that have not been seen in the training set.

An alternative approach would be to interleave the note and duration tokens into a single stream of input and let the model learn that the output should be a single stream where the note and duration tokens alternate. This comes with the added complexity of ensuring that the output can still be parsed when the model has not yet learned how to interleave the tokens correctly.

There is no right or wrong way to design your model—part of the fun is experimenting with different setups and seeing which works best for you!

Chapter 11

Deep Learning Foundations and Concepts

Contents

11.1 Transformer

One major advantage of transformers is that transfer learning is very effective, so that a transformer model can be trained on a large body of data and then the trained model can be applied to many downstream tasks using some form of fine-tuning. A large-scale model that can subsequently be adapted to solve multiple different tasks is known as a *foundation model* [BB23].

Furthermore, transformers can be trained in a self-supervised way using unlabelled data, which is especially effective with language models since transformers can exploit vast quantities of text available from the internet and other sources. The *scaling hypothesis* asserts that simply by increasing the scale of the model, as measured by the number of learnable parameters, and training on a commensurately large data set, significant improvements in performance can be achieved, even with no architectural changes. Moreover, the transformer is especially well suited to massively parallel processing hardware such as Graphics Processing Unit (GPU), allowing exceptionally large neural network language models having of the order of a trillion (10^{12}) parameters to be trained in reasonable time [BB23].

11.2 GAN

Appendix A

Formulas

A.1 Gaussian distribution

Definition A.1 (Gaussian distribution)**.** *Gaussian distribution*

Theorem A.1 (Central limit theorem)**.**

Bibliography

[Bab16] László Babai. "Graph Isomorphism in Quasipolynomial Time". Jan. 19, 2016. arXiv: 1512.03547 [cs, math] (cit. on p. 2).

[BB23] Christopher M. Bishop and Hugh Bishop. *Deep Learning: Foundations and Concepts*. Berlin, Germany: Springer, Nov. 2023 (cit. on p. 93).

[BDS19] Andrew Brock, Jeff Donahue, and Karen Simonyan. *Large Scale GAN Training for High Fidelity Natural Image Synthesis*. 2019. arXiv: 1809.11096 [cs.LG] (cit. on p. 91).

[Chi09] Andrew M. Childs. *Universal Computation by Quantum Walk*. Physical Review Letters 102.18 (May 4, 2009), p. 180501. arXiv: 0806.1972 (cit. on p. 2).

[Don+17] Hao-Wen Dong et al. *MuseGAN: Multi-track Sequential Generative Adversarial Networks for Symbolic Music Generation and Accompaniment*. 2017. arXiv: 1709.06298 [eess.AS] (cit. on p. 91).

[Fos22] David Foster. *Generative Deep Learning*. "O'Reilly Media, Inc.", June 28, 2022. 444 pp. (cit. on pp. 11, 63, 65, 66, 70, 75, 78–80, 83).

[Hoo+21] Emiel Hoogeboom et al. *Autoregressive Diffusion Models*. 2021. arXiv: 2110.02037 [cs, stat]. preprint (cit. on p. 76).

[Kar+18] Tero Karras et al. *Progressive Growing of GANs for Improved Quality, Stability, and Variation*. OpenReview (Feb. 2018) (cit. on pp. 84, 91).

[Kit+02] Alexei Yu Kitaev et al. *Classical and quantum computation*. 47. American Mathematical Soc., 2002 (cit. on p. 2).

[KLA18] Tero Karras, Samuli Laine, and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*. CoRR abs/1812.04948 (2018). arXiv: 1812.04948 (cit. on p. 89).

[She+20] Sheng Shen et al. *PowerNorm: Rethinking Batch Normalization in Transformers*. June 28, 2020. arXiv: 2003.07845 [cs]. URL: http://arxiv.org/abs/2003.07845 (visited on 12/26/2023). preprint (cit. on pp. 81, 84).

[SME22] Jiaming Song, Chenlin Meng, and Stefano Ermon. *Denoising Diffusion Implicit Models*. Oct. 5, 2022. arXiv: 2010.02502 [cs]. URL: http://arxiv.org/abs/2010.02502 (visited on 12/26/2023). preprint (cit. on pp. 77, 78).

[Vas+17] Ashish Vaswani et al. "Attention Is All You Need". Dec. 5, 2017. arXiv: 1706.03762 [cs] (cit. on p. 81).

[Zha+19] Han Zhang et al. *Self-Attention Generative Adversarial Networks*. 2019. arXiv: 1805.08318 [stat.ML] (cit. on p. 91).

[Zha+23] Aston Zhang et al. *Dive into Deep Learning*. Cambridge University Press, Dec. 7, 2023. 581 pp. (cit. on pp. 13, 14).

BIBLIOGRAPHY

Alphabetical Index