Mathematical Engineering of Deep Learning

Book Draft

Benoit Liquet, Sarat Moka and Yoni Nazarathy

February 28, 2024

Contents

8 Specialized Architectures and Paradigms - DRAFT

In each of the chapters 5, 6, and 7, we presented one concrete deep learning paradigm, namely feedforward networks, convolutional neural networks, and sequence models respectively. Such models are useful in their own right, yet in the world of deep learning one often integrates them within more complex architectures for specific activities. For example the convolutional neural networks of Chapter 6 may be inter-connected with sequence models of Chapter 7 for applications that involve both images and text. In addition, other specialized architectures and paradigms have also emerged where in each case, non-trivial ideas are employed to create powerful models. In the current chapter we present such ideas emerging from different domains, yet all using deep neural networks. Some of these domains include generative modelling, where we focus on diffusion models and generative adversarial networks, after an overview of variational autoencoders. Other domains are in the area of automatic control and decision making where we present concepts of reinforcement learning. Finally, we explore the domain of graph neural networks, an area that is proving to be ever so useful for complex problems that can be represented with graph structures. Without space constraints, each of these topics deserves its own chapter or a sequence of chapters, yet within this single chapter we hope that the reader gains an overarching view.

In Section 8.1 we introduce generative modelling principles. For this we introduce principles of variational autoencoders which are the basis for diffusion models, the topic of Section 8.2. In Section 8.3, we describe the ideas of generative adversarial networks, which also have some pinnings in game theory. These two variants, namely diffusion models of Section 8.2 and generative adversarial networks of Section 8.3, have become the most popular means of generative modelling to date. We continue in Section 8.4 where we outline principles of reinforcement learning. Towards that end we first define Markov decision processes and discuss principles of optimal control, and then tie the ideas to deep reinforcement learning. Note that while the application domain of reinforcement learning differs from the generative modelling domain of the earlier sections, ideas of Markov chains used in diffusion models, reappear in reinforcement learning of Section 8.4. Finally, graph neural networks are introduced in Section 8.5. As we see, graph neural networks generalize the convolutional neural networks of Chapter 6 while allowing us to have general graph structures within the data in contrast to simple spatial connections.

8.1 Generative Modelling Principles

The field of *generative modelling* deals with algorithms and models for creating (generating) data such as fake images, generated text, or similar. In this space we often think probabilistically and assume that the data has an underlying probability distribution. Our goal is then to train models that generate random yet realistic data from that distribution, with or without explicitly capturing the form of the distribution. Generative modelling can be

applied both in the supervised case (features x and labels y) and the unsupervised case (no labels y). In Chapter 2 towards the end of Section 2.2 we mentioned a few names of supervised learning generative modelling approaches such as naive Bayes and others. In contrast, now we only focus on unsupervised learning. In such a case observed data $\mathcal{D}$ is composed of $\{x^{(1)}, \ldots, x^{(n)}\}$ and we assume that all $x^{(i)}$ are distributed according to a probability distribution, which we simply denote as $p(x)$. Note that in general $x^{(i)}$ is a high dimensional object such as a high resolution color image, and hence $p(x)$ is a complicated distribution.

There are many generative approaches, yet our focus in this chapter is on two approaches that have become very popular due to their ability to generate data that appears realistic. One approach that has recently become popular is the *diffusion model* approach, and this is the focus of Section 8.2. Another approach is the *generative adversarial network* (GAN) approach which is the focus of Section 8.3. The ideas of the diffusion approach require understanding of *variational autoencoders*. For this purpose, in the bulk of this section, we introduce variational autoencoders, a class of generative models that is also interesting and useful in their own right.

In Figure 8.1 we present a schematic of both the GAN approach and the diffusion approach. In both cases, an underlying idea is to first generate random noise. The space of this noise is typically called the *latent space*, and it has a very simple distribution, denoted[1] as $p(z)$, for example a multivariate standard normal. In both approaches, the sample z is processed as input to a model whose output is a point x, which is approximately distributed according to $p(x)$ and hence "looks realistic". In the GAN case, one often uses z which is of lower dimension than x; for example x may be a $3 \times 200 \times 200$ dimensional color image while z may be a vector of length 100. In contrast, in the diffusion models case, the latent variable z has the same dimension as the target sample x.

Both the diffusion approach and the GAN approach embody the conditional distribution of the data given the noise, denoted as $p(x \mid z)$. The GAN approach learns an approximate algorithm for sampling from $p(x \mid z)$ with a so called *generator network*, without explicitly learning $p(x \mid z)$. In contrast, diffusion models are probabilistic models which approximately learn $p(x \mid z)$ within a so-called *decoder*. One additional difference between the approaches is that GANs generate x from z in one shot, i.e., via one application of the generator network. In contrast diffusion models iterate over multiple steps of the decoder, starting with the latent noise variable and eventually reaching the target output; see also Figure 8.4.

Both the diffusion decoder and the GAN generator use deep neural networks with learned parameters. As illustrated in Figure 8.1, these parameters are learned by training the models using the training data $\mathcal{D}$. In both cases an auxiliary network (not illustrated in Figure 8.1) also plays a part in training. In GANs this auxiliary network is called the *discriminator* and to train the generator we train the discriminator network in parallel. For diffusion models the auxiliary network is called an *encoder* and in this case, it is fixed in advance and has no learned parameters. More details are in Section 8.3 for GANs and Section 8.2 for diffusion models. We begin the study of generative models with variational autoencoders which lay down the foundations for understanding diffusion models.

[1] Observe that in the context of this chapter we use $p(\cdot)$ in multiple ways, where the actual distribution used should be inferred from the argument of the function. For example $p(x)$ is the distribution of the data while $p(z)$ is the distribution of the latent space.

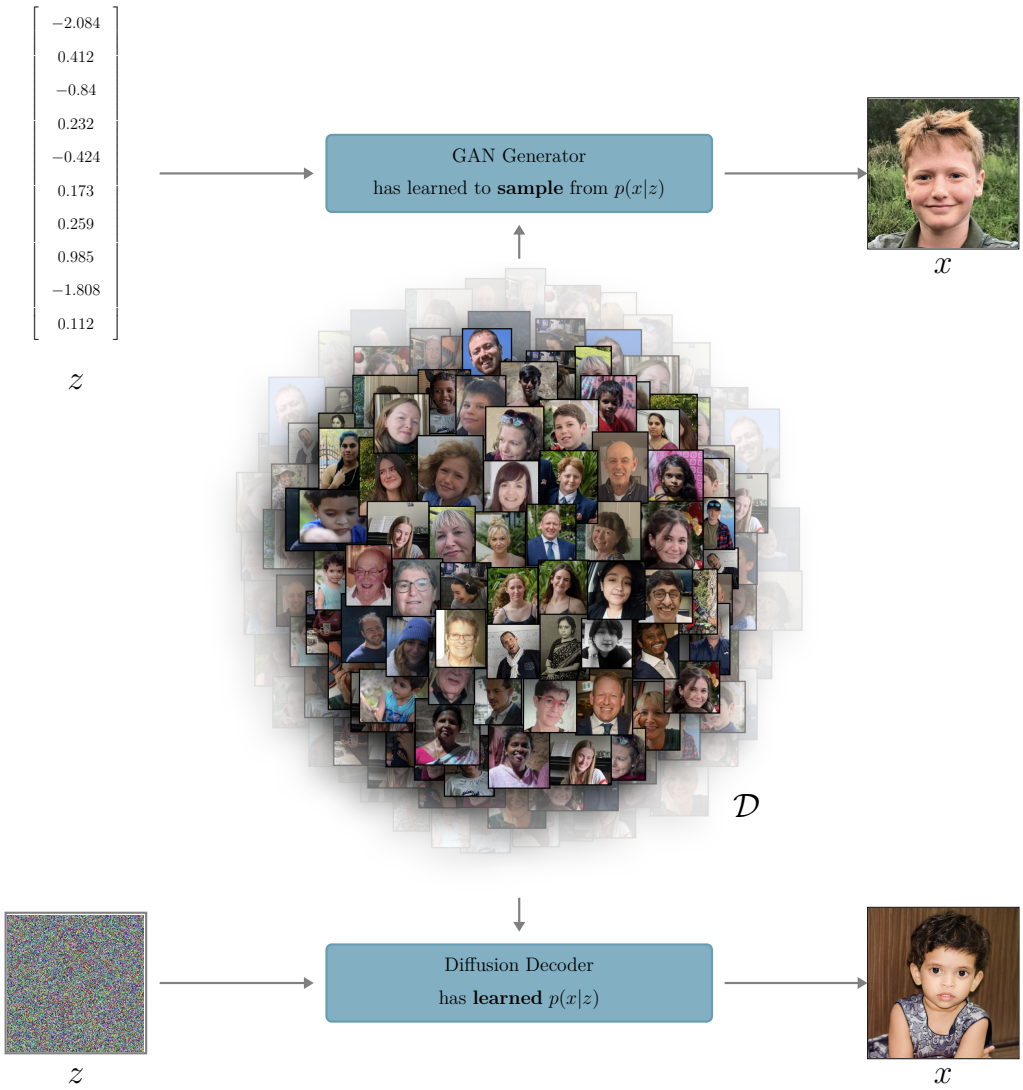


Figure 8.1: Generative adversarial networks (GANs) and diffusion models are different approaches for generative models. Both use a latent space z and are able to sample from the conditional distribution $p(x \mid z)$ to generate x. For image generation, GANs typically have a smaller dimensional z while diffusion models use z of the same dimension as the image. Both cases are trained using data $\mathcal{D}$ where the resulting model in GANs is called a generator, and the resulting model in diffusion models is called a decoder.

Variational Autoencoders

In Section 3.5 we introduced basics of autoencoders within the context of shallow autoencoders. We now focus on variational autoencoders which are probabilistic enhancements of autoencoders. We first focus on a statistical perspective of variational autoencoders and then see a general framework for generating data from $p(x)$ after learning it.

Like many statistical models, a variational autoencoder is a *parametric model* where we assume some parameters θ determine $p(x)$ and hence denote the distribution as $p_\theta(x)$. By learning θ we can learn the distribution and then also generate samples from $p_\theta(x)$. A key principle for estimation of θ is maximum likelihood estimation (MLE). In Section 3.1, we briefly introduced this commonly used approach in the context of logistic regression.

With the maximum likelihood approach, similarly to (3.10) of Chapter 3, our estimate of θ for the data $\mathcal{D}$ is

$$\widehat{\theta}_{MLE} := \operatorname*{argmax}_{\theta} \frac{1}{n} \sum_{i=1}^{n} \log p_\theta(x^{(i)}), \tag{8.1}$$

where the optimized quantity (barring the $1/n$ term) is the log-likelihood under the assumption of independent and identically distributed elements of $\mathcal{D}$. In practice, solving (8.1) requires information about the expression or form of $p_\theta(x)$.

In variational autoencoders, as alluded to at the start of the section, an unseen latent variable z plays a key role. More precisely we suppose that $p_\theta(x)$ is coupled with z which has distribution $p(z)$. The distribution $p(z)$ is assumed to be simple and does not depend on the parameters θ. The relationship involving z and x can then be represented as,

$$p_\theta(x, z) = p_\theta(x \,|\, z)\, p(z). \tag{8.2}$$

Hence in summary, both the joint distribution $p_\theta(x, z)$ and the conditional distribution $p_\theta(x \,|\, z)$ are paramterized by θ, but $p(z)$ is not parameterized by θ.

We can use (8.2) to obtain $p_\theta(x)$ by marginalizing the joint distribution over z as,

$$p_\theta(x) = \int p_\theta(x \,|\, z)\, p(z) dz. \tag{8.3}$$

Such a representation is helpful in expressing complex $p_\theta(x)$ using relatively simple expressions for $p_\theta(x \,|\, z)$ and $p(z)$. However, even in this setting, the integral (8.3) is intractable and hence indirect optimization using ELBO, defined and described in the sequel, is used.

To parameterize $p_\theta(x \,|\, z)$ and describe $p(z)$ we make use of *multivariate normal distributions* where the probability density for a random vector u is represented via $\mathcal{N}(u\,;\, \mu, \Sigma)$ with μ denoting the *mean vector* and Σ denoting the *covariance matrix*. A particular simple case is the *standard multivariate normal* where $\mu = 0$ (zero vector) and $\Sigma = I$ (identity matrix). A slightly more general case is setting $\Sigma = \sigma^2 I$, where the positive scalar σ^2 determines the variance shared among all coordinates.

We sometimes reparametrize[2] a covariance matrix $\Sigma \in \mathbb{R}^{p \times p}$ via

$$\Sigma = \Gamma\, \Gamma^\top, \quad \text{where} \quad \Gamma \in \mathbb{R}^{p \times p}. \tag{8.4}$$

Covariance matrices are always positive semi definite, and characterizing this constraint on the individual entries of the matrix can be difficult. In contrast, with reparameterizations to Γ, one may end up with an unconstrained matrix Γ which is easier to work with.

[2]For example Γ in $\Sigma = \Gamma\, \Gamma^\top$ can be obtained via a Cholesky factorization in which case Γ is an unconstrained lower triangular matrix. Other factorizations such as the spectral decomposition (or singular value decomposition) can also be used.

Starting with $p_\theta(x \mid z)$ we assume that this conditional distribution is multivariate normal where the mean vector and covariance matrix are functions of z that are both parameterized by θ. In particular, keeping in mind that m is the dimension of z, and p is the dimension[3] of x, we assume that we have learned functions,

$$\mu_\theta : \mathbb{R}^m \to \mathbb{R}^p \quad \text{and} \quad \Sigma_\theta : \mathbb{R}^m \to \mathbb{R}^{p \times p}, \tag{8.5}$$

each parameterized by θ. With these, we have the conditional distribution set as

$$p_\theta(x \mid z) = \mathcal{N}\big(x \,;\, \mu_\theta(z), \Sigma_\theta(z)\big). \tag{8.6}$$

In this case with (8.3) and (8.6), $p_\theta(x)$ is a *Gaussian mixture model*[4] with *mixture components* $p_\theta(x \mid z)$ indexed by the values of z, and corresponding *mixture weights* provided by $p(z)$. It is well known that any distribution can essentially be closely approximated by a Gaussian mixture model if the mixture components and mixture weights are properly selected.[5]

While the general form of Gaussian mixture models is very versatile, in variational autoencoders we make some simplifying assumptions. First and most importantly we assume that the distribution of the latent variable z is multivariate standard normal. Namely, the mixture weights are,

$$p(z) = \mathcal{N}(z \,;\, 0, I). \tag{8.7}$$

Further, it is common to reduce the complexity of the mixture components (8.6) and assume that the covariance function is simply $\sigma^2 I$ where σ^2 is a pre-determined (not learned) hyper-parameter. Thus, (8.6) is reduced to

$$p_\theta(x \mid z) = \mathcal{N}\big(x \,;\, \mu_\theta(z), \sigma^2 I\big), \tag{8.8}$$

and now (8.8) together with (8.3) implies that θ parameterizes the distribution of x only via the mean function $\mu_\theta(\cdot)$. In the context of deep learning, this mean function is taken as a neural network with θ representing the weights and biases. However in the general framework of variational autoencoders the mean function $\mu_\theta(\cdot)$ could be any model and not necessarily a deep neural network.

With the model defined, suppose now that based on data $\mathcal{D}$ we manage to approximate the maximum likelihood estimate (8.1) and learn θ for $\mu_\theta(\cdot)$. We can now use the model as a generative model similar to the spirit of Figure 8.1. In particular to generate a new random data sample x^*, we first generate a random latent sample z^* from the standard multivariate normal distribution. We then compute $\mu_\theta(z^*)$ using the learned model, and then generate x^* using (8.8) where the mean taken is $\mu_\theta(z^*)$. In this sense, every random z^* yields its own $\mu_\theta(z^*)$ which in turn yields a random x^*. This generative process can be illustrated as follows:

$$z^* \quad \xrightarrow{\mu_\theta(\cdot)} \quad \mu_\theta(z^*) \quad \xrightarrow{p_\theta(x \mid z)} \quad x^*. \tag{8.9}$$

Note that the diffusion models alluded to in Figure 8.1 are a variant of variational autoencoders and we cover the details of this powerful class of generative models the Section 8.2.

[3] In case x is an image we vectorize it and assume the dimension is p.

[4] A Gaussian mixture model is a probability mixture model where a collection Gaussian distributions are "weighted" to create a new probability distribution. As with any mixture model, it is composed of *mixture components*, and *mixture weights*, both of which appear under the integral sign as in (8.3).

[5] Mathematically this property can be phrased as the fact that Gaussian mixture models are dense in the space of probability distributions.

As already mentioned, variational autoencoders are related to the standard autoencoders presented in Section 3.5, yet standard autoencoders are deterministic while variational autoencoders are probabilistic. It is this this difference that allows variational autoencoders to be used as generative models while standard autoencoders on their own cannot. To see this distinction, assume we were to try and carry out a generative procedure similar to (8.9) with a decoder trained on a standard autoencoder. This can be represented as follows:

$$z^* \xrightarrow{\text{Decoder}} x^*,$$

where again we would sample z^* from some predetermined distribution. However the standard autoencoder on its own does not capture the distribution in the latent space where meaningful samples of z^* are present. To see this, consider for example Figure 3.14 in Chapter 3, and observe that z^* would be some random point in the latent space and then mapped to a random output x^* via the decoder. With this, there is no guarantee to get any realistic x^* because the model does not capture the region of interest in the latent space. Hence standard autoencoders, while useful for other purposes as discussed in Section 3.5, are not useful as generative models on their own right.

The Encoder-Decoder Architecture for Variational Autoencoders

As alluded to above, a variational autoencoder has both an *encoder* and a *decoder*. The main object used for generative models, $p_\theta(x \mid z)$, is represented via the decoder and as described above this distribution is parameterized by the mean function $\mu_\theta(\cdot)$ as well as sometimes by the covariance function $\Sigma_\theta(\cdot)$. However, for learning the parameters θ, we require the full (variational) encoder-decoder architecture as illustrated in Figure 8.2.

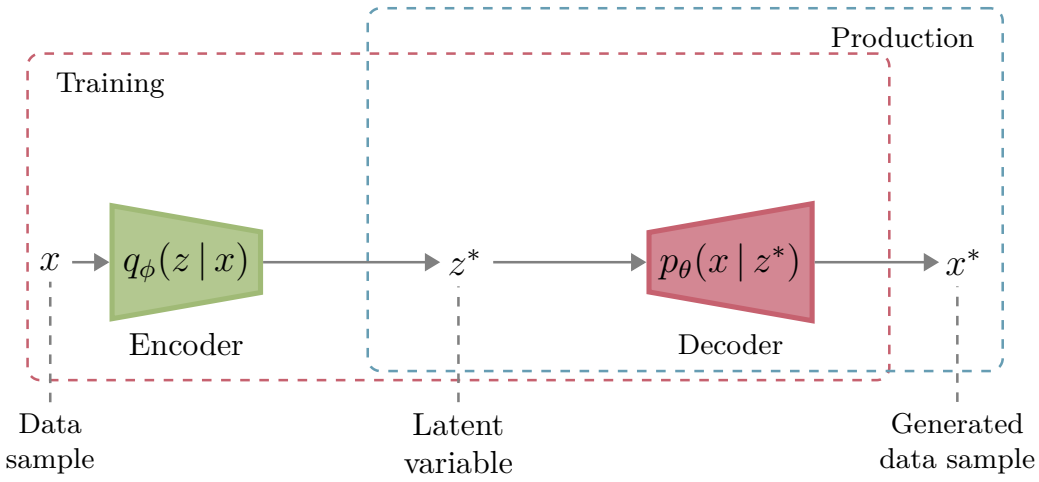


Figure 8.2: The variational autoencoder architecture comprised of an encoder $q_\phi(z \mid x)$ and a decoder $p_\theta(x \mid z)$.

Like the decoder, the encoder can be a neural network parameterized by weights and biases denoted as ϕ. Also like the decoder, the encoder is probabilistic in the sense that it describes a probability distribution, denoted as $q_\phi(z \mid x)$ and known as the *variational posterior* of the latent variable, that provides a distribution of the latent variable z given the data sample x. That is, the encoder transforms an input data sample $x \in \mathbb{R}^p$ from $\mathcal{D}$ to a latent variable

$z \in \mathbb{R}^m$. Like the decoder, the encoder has a mean function and a covariance function which we denote respectively as,

$$\mu_\phi : \mathbb{R}^p \to \mathbb{R}^m \quad \text{and} \quad \Sigma_\phi : \mathbb{R}^p \to \mathbb{R}^{m \times m}, \tag{8.10}$$

and the conditional distribution of the latent variable z given a data point x is assumed to be normally distributed. Namely,

$$q_\phi(z \,|\, x) = \mathcal{N}\big(z \,;\, \mu_\phi(x), \Sigma_\phi(x)\big). \tag{8.11}$$

Hence compare the encoder's (8.10) and (8.11) with the decoder's (8.5) and (8.6) respectively.

Taking all $x \in \mathcal{D}$ under consideration, the *latent variable sample marginal distribution* is given by,

$$q_{\phi,\mathcal{D}}(z) = \frac{1}{n} \sum_{x \in \mathcal{D}} q_\phi(z \,|\, x). \tag{8.12}$$

Ideally, like (8.7), we would like $q_{\phi,\mathcal{D}}(z)$ to be a standard normal distribution in which case it no longer depends on ϕ and $\mathcal{D}$. This is then set as a training goal when jointly training the encoder (learning ϕ) and decoder (learning θ) in a variational autoencoder. To achieve such a goal, we aim to minimize a loss function of the general form,

$$C(\phi, \theta \,;\, \mathcal{D}) = C_{\text{PriorMatching}}(\phi) + C_{\text{Reconstruction}}(\phi, \theta). \tag{8.13}$$

In general this loss depends on the data $\mathcal{D}$, yet for simplicity we omit this relationship for the two terms on the right hand side. Minimization of the first term, $C_{\text{PriorMatching}}(\phi)$, aims to set the latent distribution (8.12) to be as close as possible to a standard normal. It is called *prior matching* because when treating the variational autoencoder as a Bayesian inference model, the latent distribution can be viewed as a prior distribution. Minimization of the second term, $C_{\text{Reconstruction}}(\phi, \theta)$, aims to capture maximum likelihood estimation with respect to both encoder and decoder parameters and hence achieve optimal *reconstruction* of the output distribution. Details of implementing these loss functions are below, after we introduce the concept of the evidence lower bound.

Relations to Maximal Likelihood and ELBO

Our aim with training variational autoencoders is that minimization of (8.13) will act as a means for maximum likelihood estimation as in (8.1). In our exposition here we focus on a simple stochastic gradient descent setting and thus we consider a single datapoint $x \in \mathcal{D}$. Extending to multiple datapoints, or mini-batches is straightforward. Our goal is to optimize,

$$\max_{\theta \in \mathbb{R}^p} \log p_\theta(x). \tag{8.14}$$

For this goal, let us decompose $p_\theta(x)$ where in our decomposition, we use the encoder distribution $q_\phi(z \,|\, x)$ as well as distributions associated with the joint distribution of the decoder from (8.2). Namely we use the joint distribution $p_\theta(x, z)$, as well as $p_\theta(z \,|\, x)$ which uses the decoder to describe the distribution of the latent variable z given x. This distribution can be obtained via,

$$p_\theta(z \,|\, x) = \frac{p_\theta(x, z)}{p_\theta(x)}. \tag{8.15}$$

Note that $p_\theta(x, z)$ and $p_\theta(z \mid x)$ are not operationally used (in a learning or inference algorithms) but rather only appear theoretically for the decomposition. A key quantity in the decomposition is called the *evidence lower bound (ELBO)*, and is defined as,

$$\mathsf{ELBO}(\theta, \phi\,;\, x) = \int q_\phi(z \mid x) \log \frac{p_\theta(x, z)}{q_\phi(z \mid x)} dz. \tag{8.16}$$

Note that ELBO is an expectation of $\log \frac{p_\theta(x,Z)}{q_\phi(Z \mid x)}$ where Z is distributed according to $q_\phi(\cdot \mid x)$. With ELBO defined, the decomposition of the log-likelihood $\log p_\theta(x)$ is,

$$\log p_\theta(x) = \mathsf{ELBO}(\theta, \phi\,;\, x) + D_{\mathsf{KL}}\left(q_\phi(z|x) \,\|\, p_\theta(z|x)\right), \tag{8.17}$$

where $D_{\mathsf{KL}}\left(q_\phi(z|x) \,\|\, p_\theta(z|x)\right)$ denotes the KL-divergence (see (B.6) in Appendix B) which is a measure of how $q_\phi(z \mid x)$ is different from $p_\theta(z \mid x)$. Note that the KL-divergence is always non-negative, and equal to zero if and only if both distributions are identical. To see (8.17), observe that,

$$\underbrace{\int q_\phi(z \mid x) \log \frac{p_\theta(x, z)}{q_\phi(z \mid x)} dz}_{\mathsf{ELBO}(\theta, \phi\,;\, x)} + \underbrace{\int q_\phi(z \mid x) \log \frac{q_\phi(z \mid x)}{p_\theta(z \mid x)} dz}_{D_{\mathsf{KL}}(q_\phi(z|x) \,\|\, p_\theta(z|x))} = \int q_\phi(z \mid x) \log \frac{p_\theta(x, z)}{p_\theta(z \mid x)} dz = \log p_\theta(x),$$

where moving from left to right, in the first step we combine the log terms and in the second step we use (8.15).

A consequence of the decomposition (8.17) as well as the non-negativity of the KL-divergence is that ELBO is a lower bound on the log-likelihood. Namely,

$$\log p_\theta(x) \geq \mathsf{ELBO}(\theta, \phi\,;\, x), \tag{8.18}$$

hence the name "lower bound" in ELBO. Note that the log-likelihood is sometimes called the *evidence*, and hence the term "evidence" in ELBO. Equality holds in (8.18) if and only if $q_\phi(z \mid x)$ and $p_\theta(z \mid x)$ are the same.

Now let us assume that the encoder model is flexible enough to yield any mean and covariance function as in (8.10). With this assumption on the flexibility of the encoder, there exists some ϕ such that $q_\phi(z \mid x)$ is equal to $p_\theta(z \mid x)$. Since $\log p_\theta(x)$ does not depend on ϕ we have from (8.18) and (8.17) that such a ϕ maximizes the evidence lower bound. Namely,

$$\log p_\theta(x) = \max_{\phi \in \mathbb{R}^m} \mathsf{ELBO}(\theta, \phi\,;\, x). \tag{8.19}$$

With such a representation of $\log p_\theta(x)$ for any θ, we can also maximize (8.19) over θ to obtain,

$$\max_{\theta \in \mathbb{R}^p} \log p_\theta(x) = \max_{\theta \in \mathbb{R}^p,\, \phi \in \mathbb{R}^m} \mathsf{ELBO}(\theta, \phi\,;\, x). \tag{8.20}$$

We thus see that the maximum likelihood estimate (8.1), when constrained to a single $x \in \mathcal{D}$, can be obtained by maximization of ELBO in terms of both the encoder parameters ϕ and the decoder parameters θ. This general idea of indirect likelihood optimization via maximization of ELBO (with the additional ϕ set of parameters for the encoder) is the crux of training variational autoencoders. The importance of this approach is that approximate maximization of ELBO is computationally feasible in contrast to direct maximum likelihood estimation, where the integration in (8.3) is a computational barrier.

Details of the Loss Function

We now see how minimization of the encoder-decoder loss function (8.13) can be constructed for approximate maximization of ELBO. For this let us expand the expression in (8.16) as,

$$\begin{aligned}\mathsf{ELBO}(\theta,\phi\,;\,x) &= \int q_\phi(z\,|\,x)\log\frac{p_\theta(x\,|\,z)\,p(z)}{q_\phi(z\,|\,x)}dz \\ &= \underbrace{\int q_\phi(z\,|\,x)\log p_\theta(x\,|\,z)dz}_{\mathbb{E}\log p_\theta(x\,|\,Z)} \;-\; \underbrace{\int q_\phi(z\,|\,x)\log\frac{q_\phi(z\,|\,x)}{p(z)}dz}_{D_{\mathsf{KL}}(q_\phi(z|x)\,\|\,p(z))}. \end{aligned} \tag{8.21}$$

In the first equality we used the representation of the joint distribution $p_\theta(x,z)$ from (8.2). Then in the second equality we expand the logarithm. The resulting first term is an expectation of the function $\log p_\theta(x\,|\,Z)$ when Z is distributed according to $q_\phi(\cdot\,|\,x)$. For the second term, keeping in mind that $p(z)$ as in (8.7) is a parameter-free multivariate standard normal, we have a KL-divergence that measures how close the encoder distribution is to the standard normal. Keep in mind that this a different application of the KL-divergence to the one used for lower bounding the evidence in (8.17).

With the ELBO expression present, we can now fill in the details for the terms in the loss expression (8.13), such that minimization of this loss works towards maximization of ELBO. Our focus is on a stochastic gradient descent approach as in Section 4.2 where at any gradient descent step, instead of considering the whole data $\mathcal{D}$, we consider a single arbitrary $x \in \mathcal{D}$. In such a case, the loss components on the right hand side of (8.13) can be defined based on a single data sample $x \in \mathcal{D}$ where, as shown below, for the second component we also generate a random latent variable $z^* \in \mathbb{R}^m$ from (8.11).

The idea of minimizing the first term of (8.13), $C_{\text{PriorMatching}}(\phi)$, is to drive the parameters such that $q_{\phi,\mathcal{D}}(z)$ of (8.12) is approximately a multivariate standard normal as in (8.7). For this we set,

$$C_{\text{PriorMatching}}(\phi) = \mu_\phi(x)^\top\mu_\phi(x) - \log\det\big(\Sigma_\phi(x)\big) + \operatorname{tr}\big(\Sigma_\phi(x)\big), \tag{8.22}$$

where $\det(\cdot)$ is the determinant of a matrix and $\operatorname{tr}(\cdot)$ is the trace of a matrix. Minimization of this expression is equivalent to minimization of the KL-divergence where the first argument is a multivariate normal distribution with mean vector $\mu_\phi(x)$ and covariance matrix $\Sigma_\phi(x)$, and the second argument is an m-dimensional standard multivariate normal distribution. This is because,

$$D_{\mathsf{KL}}\left(q_\phi(z|x)\,\|\,p(z)\right) = \frac{1}{2}C_{\text{PriorMatching}}(\phi) - \frac{m}{2},$$

as can be verified with (B.11) of Appendix B.

Moving onto the second term in (8.13), using a single x and a single z^*, we construct this term as,

$$C_{\text{Reconstruction}}(\phi,\theta) = \big(x-\mu_\theta(z^*)\big)^\top\Sigma_\theta(z^*)^{-1}\big(x-\mu_\theta(z^*)\big) + \log\det\big(\Sigma_\theta(z^*)\big). \tag{8.23}$$

This term aims to capture the $\mathbb{E}\log p_\theta(x\,|\,Z)$ term in (8.21). Observe that $C_{\text{Reconstruction}}(\phi,\theta)$ is a random variable with distribution influenced by ϕ through the distribution of z^*. The form of (8.23) arises from the log-density expression in (B.8) of Appendix B. We do not have an explicit expression for the expectation $\mathbb{E}\log p_\theta(x\,|\,Z)$ term of (8.21), therefore, as

a simple estimate we use a single sample z^* of the latent variable from the distribution $q_\phi(z \,|\, x)$ and rely on the approximate relationship,

$$\mathbb{E} \log p_\theta(x \,|\, Z) \approx \log p_\theta(x \,|\, z^*), \tag{8.24}$$

as a proxy for the optimization. Now considering (8.23) we have,

$$\log p_\theta(x \,|\, z^*) = -\frac{1}{2} C_{\text{Reconstruction}}(\phi, \theta) + \frac{1}{2} \log(2\pi).$$

Hence minimization of $C_{\text{Reconstruction}}(\phi, \theta)$ maximizes $\log p_\theta(x|z^*)$, which approximates maximization of $\mathbb{E} \log p_\theta(x \,|\, Z)$ via (8.24).

We have thus seen that the encoder-decoder architecture, learned via stochastic gradient descent with loss function (8.13) and individual terms (8.22) and (8.23) approximates maximization of ELBO, and thus facilitates maximum likelihood estimation of θ.

The Reparameterization Trick

During training of the variational autoencoder, we need to compute the gradient of the loss function (8.13). With standard backpropagation we can compute the gradient of the $C_{\text{PriorMatching}}(\phi)$ component in a straight forward manner. However, since the $C_{\text{Reconstruction}}(\phi, \theta)$ component is based on a random sample z^*, it is not obvious how to use backpropagation through the number random generator. Luckily, via a reparametrization of the random vector z^* as an affine function of a standard random vector ϵ^*, we can overcome this difficulty. Specifically, if ϵ^* is a standard m-dimensional multivariate normal random vector, then we may set,

$$z^* = \mu_\phi(x) + \Gamma_\phi(x)\, \epsilon^*, \tag{8.25}$$

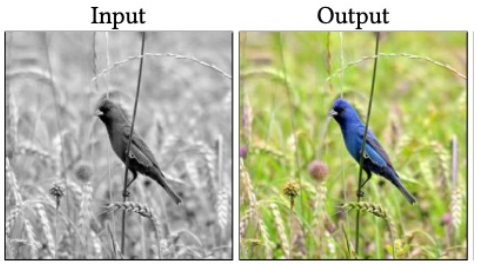
where as before $\mu_\phi(x)$ is the learned mean function of the encoder. With the reparameterization (8.25), the desired learned covariance function $\Sigma_\phi(x)$ is decomposed using $\Sigma_\phi(x) = \Gamma_\phi(x)\, \Gamma_\phi(x)^\top$, as in (8.4). Now $\Gamma_\phi(\cdot)$ is learned in the encoder neural network in place of $\Sigma_\phi(\cdot)$ of (8.10). With this, the distribution of z^* is as required, and further there are no longer issues with enforcing the positive definite constraint that is required for the covariance matrix.

Importantly, backpropagation can now be carried out, because the random numbers generated in ϵ^* are now generated independently of the parameters whose gradients we seek. In some cases, the desired covariance matrix is diagonal. In such cases, the reparameterization is especially simple since $\Gamma_\phi(x)$ is also diagonal with each entry being the square root of the associated diagonal entry of $\Sigma_\phi(x)$.

8.2 Diffusion Models

Diffusion models are a class of generative models that have shown great promise in creating both realistic looking images, as well as highly impressive surreal artwork. Figure 8.3 presents a few examples with several paradigms of application including image generation, colorization, style transfer, and text to image creation. The ease of use from a user's perspective of such platforms, and the quality of the images created is impressive.

(a)



(b)

(c)

(d)

Figure 8.3: Images generated via various diffusion architectures[6] with various types of paradigms. (a) A 256×256 generated images based on a label. (b) Image-to-image generation (colorization). (c) Applying a style (left column) to an image (middle column). (d) A text-to-image application.

The principles underlying diffusion models hinge on variational autoencoders presented in the previous section, as well as on a generalisation of such models, called *hierarchical variational autoencoders* and in particular *Markovian hierarchical variational autoencoders*. Diffusion models are a special case of such models, hence in exploring diffusion models, we first study Markovian hierarchical variational autoencoders. We then construct diffusion models as a special case of Markovian hierarchical variational autoencoders, based on auto-regressive Gaussian processes.

Hierarchical Variational Autoencoders

In the autoencoders of Section 8.1 the encoder acts as a *noising mechanism* that converts the input x into a noisy latent variable z. On the other hand the decoder can be viewed as a *denoising mechanism* for converting a noisy latent variable z to meaningful output. The main idea of *hierarchical variational autoencoders* is to break up this process into multiple levels. In the encoder, the input x is noised slightly to create z_1, then z_1 is further noised to create z_2 up until some final level T with a fully noisy latent variable z_T. In the decoder the reverse process takes place where each step from z_t to z_{t-1} enacts a slight denoiseing operation.

Hence in hierarchical variational autoencoders, instead of a single latent variable z, we have a sequence of T levels with latent variables, $z_1, \ldots, z_T$. Such models also enforce a specific dependence structure within this sequence and are called "hierarchical", since in the encoder each z_t can be generated based on the values of the lower-level latent variables $z_1, \ldots, z_{t-1}$ and in the decoder each latent variable z_t can be generated based on the values of the higher latent variables $z_{t+1}, \ldots, z_T$. In such a model, one can view the input $x \in \mathcal{D}$ as the 0-th level, namely the complete sequence of levels is $x, z_1, \ldots, z_T$, where we can treat x as z_0.

The most common hierarchical variational autoencoders are *Markovian* in which case the sequence $x, z_1, \ldots, z_T$ is a *Markov chain*, and the model is called a *Markovian hierarchical variational autoencoder*. That is, the sequence follows the *Markov property* which can be defined in multiple ways. One way is that for any $t = 1, \ldots, T-1$, given the value of z_t, the sequence $x, z_1, \ldots, z_{t-1}$ and the sequence $z_{t+1}, \ldots, z_T$ are independent.

Considering the decoder, a consequence of the Markov property is that the joint distribution $p_\theta(x, z_1, \ldots, z_T)$ can be described as a product of *one step transition probabilities*. We have,

$$p_\theta(x, z_1, \ldots, z_T) = p_{\theta_1}(x \,|z_1) p_{\theta_2}(z_1 \,|\, z_2) \cdots p_{\theta_T}(z_{T-1} \,|\, z_T) p(z_T), \tag{8.26}$$

where $\theta = (\theta_1, \ldots, \theta_T)$ are the decoder parameters, and each $p_{\theta_t}(z_{t-1} \,|\, z_t)$ is the decoder conditional one step transition probability from level t to level $t-1$. Note that in this expression, $p(z_T)$, the distribution of level T, is parameter free. Since it is assumed to be fully noisy, it is often taken as a multivariate standard normal similar to (8.7) from the (non-hierarchical) variational autoencoder.

In such models, the one step transition probabilities $p_{\theta_t}(\cdot \,|\, z_t)$ are generally represented as multivariate Gaussian distributions where for level t, we learn neural networks for the mean function $\mu_{\theta_t}(z_t)$ and potentially for the covariance function $\Sigma_{\theta_t}(z_t)$. Hence the learned parameters θ_t describe the distribution at level $t-1$ given the value at the previous level, z_t.

[6]Image (a) is thanks to Ho, et al. [183]. Image (b) is thanks to Saharia, et. al. [358]. Image (c) is thanks to Wang, et. al. [418]. Image (d) is thanks to Nichol, et. al. [307].

Observe that the Markovian hierarchical variational autoencoder joint distribution in (8.26) is the parallel of the joint distribution (8.2) for the non-hierarchical case. A potential benefit of using (8.26) is that by breaking up the parameterization of the decoder into T smaller steps, each with its own learned parameters, more expressive models can be achieved.

A decoder parameterized by $\theta = (\theta_1, \ldots, \theta_T)$ implements a generative model, similarly to the non-hierarchical case. First z_T^* is drawn from the distribution $p(z_T)$, and then for each transition we compute $\mu_{\theta_t}(z_t^*)$ and $\Sigma_{\theta_t}(z_t^*)$ and then draw z_{t-1}^* from the transition probability $p_{\theta_t}(\cdot \,|\, z_t^*)$ parametrized by these values. Hence while a non-hierarchical variational autoencoder draws z^* once and then generates x^*, here we draw z_T^*, then z_{T-1}^*, and so forth until attaining a generated data sample $x^* = z_0^*$.

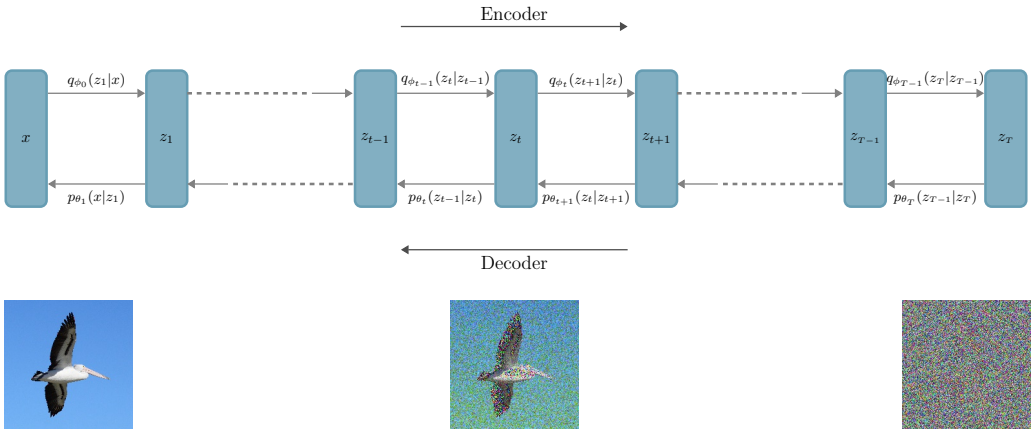


Figure 8.4: A Markovian hierarchical variational autoencoder has a latent space $z_1, z_2, \ldots, z_T$, with $x = z_0$. The encoder transforms from z_t to z_{t+1} (adding noise) and the decoder works in the opposite direction (removing noise).

Like non-hierarhcial variational autoencoders, in Markovian hierarchical variational autoencoders, the encoder serves as an aid for training the generative model (the decoder). The encoder also incorporates a Markovian structure within levels. The encoder parameters are $\phi = (\phi_0, \ldots, \phi_{T-1})$, where this time for $t = 0, \ldots, T-1$, we have conditional one step transition probabilities, $q_{\phi_t}(z_{t+1} \,|\, z_t)$ describing the transition from level t to level $t+1$. Like the decoder, in the encoder we can assume Gaussian transition probabilities which are this time parameterized by $\mu_{\phi_t}(z_t)$ and $\Sigma_{\phi_t}(z_t)$. See Figure 8.4.

Considering the encoder, it is useful to represent the conditional distribution of $z_1, \ldots, z_t$ given x for any $t = 1, \ldots, T$, denoted as $q_\phi(z_1, \ldots, z_t \,|\, x)$. Due to the Markovian assumption we have,

$$q_\phi(z_1, \ldots, z_t \,|\, x) = q_{\phi_0}(z_1 \,|\, x) q_{\phi_1}(z_2 \,|\, z_1) \cdots q_{\phi_{t-1}}(z_t \,|\, z_{t-1}), \quad \text{for} \quad t = 1, \ldots, T. \tag{8.27}$$

Further, the distribution of z_t for a given x is denoted as $q_\phi(z_t \,|\, x)$ which may formally be represented as a marginal distribution from $q_\phi(z_1, \ldots, z_t \,|\, x)$, and obtained via,

$$q_\phi(z_t \,|\, x) = \int \cdots \int q_\phi(z_1, \ldots, z_t \,|\, x) \; dz_1 \cdots dz_{t-1}. \tag{8.28}$$

In the specific diffusion models below this distribution is constructed with an explicit form, yet for now the general expression (8.28) is appropriate.

With this notation and the construction of the encoder and decoder in place, we can seek to approximate maximum likelihood estimation of θ. This is done in a similar manner to the non-hierarchical variational autoencoder case outlined in Section 8.1. Specifically we maximize an ELBO term over both ϕ and θ.

The ELBO term in this case is similar to (8.16) yet can be represented via $z_1, \ldots, z_T$ in place of z. Namely,

$$\mathsf{ELBO}(\theta, \phi\,;\, x) = \int q_\phi(z_1, \ldots, z_T \,|\, x) \log \frac{p_\theta(x, z_1, \ldots, z_T)}{q_\phi(z_1, \ldots, z_T \,|\, x)} dz_1 \cdots dz_T. \tag{8.29}$$

Now using the Markovian structure of (8.26), (8.27), and after some extensive manipulation, the following expansion of $\mathsf{ELBO}(\theta, \phi\,;\, x)$ arises:

$$\begin{aligned}
&\underbrace{\int q_{\phi_0}(z_1 \,|\, x) \log p_{\theta_1}(x \,|\, z_1) dz_1}_{\mathbb{E} \log p_{\theta_1}(x \,|\, Z_1)} && \text{(Reconstruction)} \\
&- \underbrace{\int_{z_T} q_\phi(z_T \,|\, x) \log \frac{q_\phi(z_T \,|\, x)}{p(z_T)} dz_T}_{D_{\mathsf{KL}}(q_\phi(z_T|x) \,\|\, p(z_T))} && \text{(Prior matching)} \\
&- \sum_{t=2}^{T} \underbrace{\int_{z_t} q_\phi(z_t \,|\, x) \int q_\phi(z_{t-1} \,|\, z_t,\, x) \log \frac{q_\phi(z_{t-1} \,|\, z_t,\, x)}{p_{\theta_t}(z_{t-1} \,|\, z_t)} dz_t.}_{\mathbb{E}\, D_{\mathsf{KL}}\left(q_\phi(z_{t-1}|Z_t,x) \,\|\, p_{\theta_t}(z_{t-1}|Z_t)\right)} && \text{(Denoising matching)}
\end{aligned} \tag{8.30}$$

With this expansion ELBO is now represented in terms of three types of terms, namely a *reconstruction term*, a *prior matching term*, and *denoising matching terms*. The first two terms are also present in non-hierarchical variational autoencoders whereas the denoising matching terms arise due to the multi-level structure of the latent space.

The reconstruction term in (8.30) is an expectation of the conditional log-likelihood with respect to Z_1, distributed according to $q_{\phi_0}(z_1 \,|\, x)$. Maximization of this term drives maximum likelihood estimation. The prior matching term in (8.30) is the KL-divergence between the standard normal distribution $p(z_T)$ and the encoder based last step distribution $q_\phi(z_T \,|\, x)$. Minimization of this KL-divergence attempts to enforce that z_T at the exit of the encoder is distributed approximately as a standard normal.

In the additional $T - 1$ terms, we use $q_\phi(z_{t-1} \,|\, z_t, x)$ which can be viewed as a denoising distribution in the encoder. Note that in contrast to the direction of the encoder (z_{t-1} to z_t), this distribution is in the opposite direction. The expected KL-divergence in each denoising matching term is between $q_\phi(z_{t-1} \,|\, z_t, x)$ and the decoder's one step transition $p_{\theta_t}(z_{t-1} \,|\, z_t)$. Ideally both the encoder and the decoder are similar at level t and minimization of the expected KL-divergence enforces the denoising to be as close as possible. Note that when represented as an expectation, the expectation is with respect to the random variable Z_t distributed as $q_\theta(z_t \,|\, x)$.

As is evident, the $\mathsf{ELBO}(\theta, \phi\,;\, x)$ expressions rely on $q_\phi(z_t \,|\, x)$ which in general needs to be computed via (8.27) and (8.28). It also relies on $q_\phi(z_{t-1} \,|\, z_t, x)$, which is generally difficult to compute. We now see, that in the special case of a diffusion model, with the assumptions imposed, the expressions for $q_\phi(z_t \,|\, x)$ and $q_\phi(z_{t-1} \,|\, z_t, x)$ simplify and are efficient to compute.

The process of learning parameters θ for Markovian hierarchical variational autoencoders is in principle similar to learning variational autoencoders as presented in Section 8.1. Specifically, we approximately maximize $\mathsf{ELBO}(\theta, \phi\,;\, x)$ over both θ and ϕ using estimates obtained via samples of the latent variables. The resulting decoder with parameters θ emerges and approximates maximum likelihood estimation as in the non-hierarchical case.

The Diffusion Model Assumptions

A *diffusion model* is a Markovian hierarchical variational autoencoder with a few specific assumptions. All latent variables are of the same dimension of the data samples. Further, recalling that $x = z_0$, the one step encoder and decoder transition probabilities are respectively,

$$q(z_{t+1} \,|\, z_t) = \mathcal{N}(z_{t+1}\,;\, \sqrt{1-\beta_t}\, z_t,\, \beta_t I) \qquad \text{for} \qquad t = 0, \ldots, T-1, \tag{8.31}$$

$$p_{\theta_t}(z_{t-1} \,|\, z_t) = \mathcal{N}(z_{t-1}\,;\, \mu_{\theta_t}(z_t),\, \sigma_t^2 I) \qquad \text{for} \qquad t = T, \ldots, 1. \tag{8.32}$$

Here we have two sequences of scalar hyper-parameters. The encoder hyper-parameters are $\beta_0, \ldots, \beta_{T-1}$, each in the range $[0, 1]$. The decoder hyper-parameters are $\sigma_1^2, \ldots, \sigma_T^2$, each a positive number. Note that the encoder has no learned parameters and thus there are no ϕ_t subscripts for $q(z_{t+1} \,|\, z_t)$. The decoder learned parameters, $\theta_1, \ldots, \theta_T$, are used for the mean functions $\mu_{\theta_1}(\cdot), \ldots, \mu_{\theta_T}(\cdot)$, but not for the one step covariance matrices that are of the form $\sigma_1^2 I, \ldots, \sigma_T^2 I$. Each of these mean functions, $\mu_{\theta_t}(\cdot)$, is a neural network, and hence the model has T learned neural networks.

A key aspect of the diffusion model is the structure of the one step transition probabilities of the encoder (8.31). Since the encoder steps have a conditional mean vector and covariance matrix of $\sqrt{1-\beta_t}\, z_t$ and $\beta_t I$ respectively, they describe an *auto-regressive stochastic sequence* of the form,

$$z_{t+1} = \sqrt{1-\beta_t}\, z_t + \sqrt{\beta_t}\, \epsilon_t, \tag{8.33}$$

where ϵ_t is a multivariate standard normal vector. Conditioned on the value of z_t, the expected value of z_{t+1} resulting from (8.33) is $\sqrt{1-\beta_t}\, z_t$ and the covariance matrix is $\beta_t I$. Hence, this stochastic recursion then follows the next step distribution (8.31). The name "diffusion model" can be attributed to the fact that if (8.33) was in continuous time, then the process is a type of a stochastic diffusion process.

A strength of the recursion (8.33), is that it also enables closed form expressions of the multi-step transition probabilities $q_\phi(z_t \,|\, x)$. Such probabilities are heavily used in the ELBO expression (8.30). For this purpose it is useful to define the products γ_t for $t = 1, \ldots, T$, with

$$\gamma_t = (1-\beta_0) \cdots (1-\beta_{t-1}).$$

With this notation, standard recursive computations of means and variances involving geometric sums yield,

$$q(z_t \,|\, x) = \mathcal{N}(z_t \,;\, \sqrt{\gamma_t}\, x,\, (1-\gamma_t)I) \qquad \text{for} \quad t = 1, \ldots, T. \tag{8.34}$$

Hence for such an auto-regressive stochastic sequence, the distribution of z_t given the initial value x has a mean vector $\sqrt{\gamma_t}x$ and a covariance matrix $(1-\gamma_t)I$.

It is of further interest to have explicit expressions for $q(z_{t-1} \,|\, z_t, x)$, a probability appearing in the denoising matching term in (8.30). The diffusion model assumptions enable us to obtain this conditional probability as well. Namely,

$$\begin{aligned} q(z_{t-1} \,|\, z_t, x) &= \frac{q(z_t \,|\, z_{t-1}, x)\, q(z_{t-1} \,|\, x)}{q(z_t \,|\, x)} \\ &= \frac{q(z_t \,|\, z_{t-1})\, q(z_{t-1} \,|\, x)}{q(z_t \,|\, x)} \\ &= \mathcal{N}\Bigg(z_{t-1} \,;\, \underbrace{\frac{(1-\gamma_{t-1})\sqrt{1-\beta_{t-1}}}{1-\gamma_t}\, z_t + \frac{\sqrt{\gamma_{t-1}}\,\beta_{t-1}}{1-\gamma_t} x}_{\text{Mean vector}},\, \underbrace{\frac{\beta_{t-1}(1-\gamma_{t-1})}{1-\gamma_t} I}_{\text{Covariance Matrix}} \Bigg). \end{aligned} \tag{8.35}$$

The first equality follows from Bayes' rule of conditional probability, maintaining the condition on x. In the second equality we use the Markovian structure which implies that $q(z_t \,|\, z_{t-1}, x) = q(z_t \,|\, z_{t-1})$. Then to obtain the final expression we manipulate normal densities as given by (8.31) and (8.34). Note that this final manipulation requires a few algebraic steps involving completion of the squares; we omit the details. Hence we have explicit expressions for the conditional (on z_t and x) mean vector and covariance matrix of z_{t-1}.

Loss Function

Training a diffusion model involves learning the parameter vectors $\theta_1, \ldots, \theta_T$. This is the learning of T distinct neural networks together. While it is a special case of training general Markovian hierarchical variational autoencoders, the fact that there are no learned parameters in the encoder, and the fact that the decoder covariance functions are also without learned parameters, eases training.

With any Markovian hierarchical variational autoencoder, we would ideally like to maximize ELBO represented in (8.30), yet in the special case of a diffusion model, the closed form diffusion expressions (8.31), (8.32), (8.34), and (8.35) simplify the training process. Some of the general parameters and associated probabilities in (8.30) are now captured by closed form expressions and hyper-parameters in the diffusion model case.

Considering the ELBO expression (8.30) we see that for diffusion models we do not need a prior matching term, since this term only depends on the encoder parameters ϕ, and diffusion models do not have learned parameters for the encoder. With this, a single observation loss function for the diffusion model can then be represented as,

$$C(\theta_1 \ldots, \theta_T \,;\, x) = C_{\text{Reconstruction}}(\theta_1) + \sum_{t=2}^{T} C_{\text{DenoisingMatching}}(\theta_t), \tag{8.36}$$

where $x \in \mathcal{D}$. The loss component $C_{\text{Reconstruction}}(\theta_1)$ relates to the reconstruction term in (8.30), and the loss components $C_{\text{DenoisingMatching}}(\theta_t)$ relates to the denoising matching terms in (8.30). Note that for brevity we omit the dependance on x in the terms on the right hand side. These loss components are implemented as,

$$C_{\text{Reconstruction}}(\theta_1) = \frac{1}{\sigma_1^2} \left\| x - \mu_{\theta_1}(z_1^*) \right\|^2, \tag{8.37}$$

$$C_{\text{DenoisingMatching}}(\theta_t) = \frac{1}{\sigma_t^2} \left\| \frac{(1-\gamma_{t-1})\sqrt{1-\beta_{t-1}}}{1-\gamma_t} z_t^* + \frac{\sqrt{\gamma_{t-1}}\,\beta_{t-1}}{1-\gamma_t} x - \mu_{\theta_t}(z_t^*) \right\|^2, \tag{8.38}$$

for $t = 2, \ldots, T$. Here each z_t^* is generated randomly using (8.34) for given data sample x.

Minimization of the terms (8.37) and (8.38) is approximately equivalent to ELBO maximization, in a similar nature to the variational autoencoder. The reconstruction loss (8.37) is similar to the standard variational autoencoder reconstruction loss (8.23), based on the log-density expression (B.8) of Appendix B. Here we exploit the fact that there are no learned covariance parameters. Similarly to the standard variational autoencoder of Section 8.1, minimization of this reconstruction loss serves to approximately maximize $\mathbb{E} \log p_{\theta_1}(x \,|\, Z_1)$ in (8.30).

The denoising matching loss terms (8.38) are based on the KL-divergence expression (B.10) of Appendix B and in fact for every level t,

$$D_{KL}\left(q(z_{t-1} \,|\, z_t, x) \,\middle\|\, p_{\theta_t}(z_{t-1} \,|\, z_t) \right) = \frac{1}{2} C_{\text{DenoisingMatching}}(\theta_t) + \text{constant},$$

where the constant on the right hand side depends only on the (non-learnable) hyperparameters β_{t-1}, γ_{t-1}, and σ_t^2. From this expression, it is clear that minimization of the KL-divergence is equivalent to minimization of the Euclidean distance between the mean vectors of the distributions $q(z_{t-1} \,|\, z_t, x)$ and $p_{\theta_t}(z_{t-1} \,|\, z_t)$. Further, this term serves as an estimate of $\mathbb{E}\, D_{\mathsf{KL}}\left(q(z_{t-1}|Z_t, x) \,\|\, p_{\theta_t}(z_{t-1}|Z_t)\right)$ from (8.30), where the mean and covariance expressions of (8.35) are used for the inner integral, and the single z_t^* drawn from $q(z_t \,|\, x)$ serves as an approximation of the outer integral.

This loss formulation in (8.36), (8.37), and (8.38) presents us with a simple mechanism for learning the parameters of the neural networks $\mu_{\theta_1}(\cdot), \ldots, \mu_{\theta_T}(\cdot)$ using gradient based optimization where at each epoch we draw random $z_1^*, \ldots, z_T^*$ according to (8.34). Similar to other deep learning cases, we can also integrate mini-batch based learning and other gradient descent variations. As we show now, these loss expressions can even be further simplified using the reparameterization trick.

The Reparameterization Trick and Simplified Loss

In (8.25) of Section 8.1, we saw the *reparameterization trick* in the context of variational autoencoders. This allows us to carry out model training using backpropagation. A similar reparameterization trick can be applied to diffusion models. In the context of diffusion models, this trick simplifies the loss function (8.36) and often yields better numerical stability during training. Such simplification is achieved by replacing the neural networks $\mu_{\theta_1}(z_1), \ldots, \mu_{\theta_T}(z_T)$, which are used for sequentially denoising the latent variables to obtain

data sample x, with a different collection of neural networks $\hat{\mu}_{\theta_1}(z_1), \ldots, \hat{\mu}_{\theta_T}(z_T)$. Each such $\hat{\mu}_{\theta_t}(z_t)$ is trained to predict a standard noise component as constructed below.

Similar to the variational autoencoder case, the reparameterization trick we use is applied on the samples of the latent variables. In particular, based on (8.34) we can represent z_t^* as

$$z_t^* = \sqrt{\gamma_t}\, x + \sqrt{1-\gamma_t}\, \epsilon_t^*, \tag{8.39}$$

where ϵ_t^* denotes an independent multivariate standard normal vector with dimension equal to the dimension of x. Using this reparameterization trick, as we show below, at a given data sample x, the loss function can be of the form,

$$C(\theta_1 \ldots, \theta_T\,;\, x) = \sum_{t=1}^{T} C_{\text{Reparameterized}}(\theta_t), \tag{8.40}$$

where each $C_{\text{Reparameterized}}(\theta_t)$ is defined as,

$$C_{\text{Reparameterized}}(\theta_t) = \frac{\beta_{t-1}^2}{\sigma_t^2\,(1-\gamma_t)(1-\beta_{t-1})} \Big\| \epsilon_t^* - \hat{\mu}_{\theta_t}\Big(\underbrace{\sqrt{\gamma_t}\, x + \sqrt{1-\gamma_t}\, \epsilon_t^*}_{z_t^*} \Big) \Big\|^2, \tag{8.41}$$

and the dependence on x is notationally suppressed. Before delving into the mathematical formulation of obtaining the loss component (8.41), let us focus on how the diffusion model is trained.

When using (8.40) and (8.41) we train the neural networks $\hat{\mu}_{\theta_t}(\cdot)$ either with a single $x \in \mathcal{D}$ or with a mini-batch approach. Importantly, in each gradient evaluation during training, we generate a standard multivariate normal ϵ_t^* which is mapped to z_t^* using (8.39). We then obtain the gradient of $\hat{\mu}_{\theta_t}(z_t)$ at $z_t = z_t^*$ and this allows us to compute the gradient of $C_{\text{Reparameterized}}(\theta_t)$.

In the remaining part of the section, we provide a derivation of the loss expressions and then discuss how the trained diffusion model is used in the production. With (8.39), a sample of z_t^* can be obtained by first generating a sample of ϵ_t^* and then using a data sample x to get z_t^*. Thus, if the values of ϵ_t^* and the corresponding z_t^* are given, we can represent x as,

$$x = \sqrt{\frac{1}{\gamma_t}}\, z_t^* - \sqrt{\frac{1-\gamma_t}{\gamma_t}}\, \epsilon_t^*.$$

Using this, we can now manipulate the mean vector expression in (8.35). After some simplification, based on the fact that $\gamma_t/\gamma_{t-1} = 1-\beta_{t-1}$, we obtain

$$\frac{(1-\gamma_{t-1})\sqrt{1-\beta_{t-1}}}{1-\gamma_t}\, z_t^* + \frac{\sqrt{\gamma_{t-1}}\,\beta_{t-1}}{1-\gamma_t} x = \frac{1}{\sqrt{1-\beta_{t-1}}}\, z_t^* - \frac{\beta_{t-1}}{\sqrt{(1-\gamma_t)(1-\beta_{t-1})}}\, \epsilon_t^*.$$

Consequently, the denoising loss component in (8.38) can be expressed as

$$C_{\text{DenoisingMatching}}(\theta_t) = \frac{1}{\sigma_t^2} \left\| \frac{1}{\sqrt{1-\beta_{t-1}}}\, z_t^* - \frac{\beta_{t-1}}{\sqrt{(1-\gamma_t)(1-\beta_{t-1})}}\, \epsilon_t^* - \mu_{\theta_t}(z_t^*) \right\|^2. \tag{8.42}$$

Hence the neural network $\mu_{\theta_t}(z_t^*)$ attempts to predict $\frac{1}{\sqrt{1-\beta_{t-1}}}\, z_t^* - \frac{\beta_{t-1}}{\sqrt{(1-\gamma_t)(1-\beta_{t-1})}}\, \epsilon_t^*$. Now the essence of the reparameterization trick is to use a different neural network, denoted as $\hat{\mu}_{\theta_t}(z_t^*)$, that takes z_t^* to predict the noise ϵ_t^*. Note that only for the notational convenience we use the same notation θ_t to denote the parameters of new neural network as well.

With the new neural network, $\hat{\mu}_{\theta_t}(z_t^*)$, we replace $\mu_{\theta_t}(z_t^*)$ in (8.42) with a new random latent variable,

$$\frac{1}{\sqrt{1-\beta_{t-1}}}\, z_t^* - \frac{\beta_{t-1}}{\sqrt{(1-\gamma_t)(1-\beta_{t-1})}}\, \hat{\mu}_{\theta_t}(z_t^*), \tag{8.43}$$

where z_t^* is obtained from ϵ_t^* using (8.39). Now with basic manipulation the $C_{\text{DenoisingMatching}}(\theta_t)$ expression in (8.42) can be replaced by $C_{\text{Reparameterized}}(\theta_t)$ of (8.41).

With a similar argument, we can show that for $t = 1$, $C_{\text{Reconstruction}}(\theta_1)$ in (8.37) can be replaced with

$$C_{\text{Reparameterized}}(\theta_1) = \frac{\beta_0}{\sigma_1^2\,(1-\beta_0)}\, \left\| \epsilon_1^* - \hat{\mu}_{\theta_1}(z_1^*) \right\|^2,$$

As a consequence of these neural network replacements and the fact that $\gamma_1 = (1-\beta_0)$, the loss component for each t must be of the form (8.41) and hence the final loss function is given by (8.40).

In production, once the diffusion model is trained, to generate a realistic looking sample x^*, we start at $t = T$ with a sample z_T^* from a multivariate standard normal. Iteratively, as we decrement t, for each step we generate a different standard multivariate normal ϵ^* and compute $\hat{\mu}_{\theta_t}(z_t^*)$. By using

$$z_{t-1}^* = \underbrace{\frac{1}{\sqrt{1-\beta_{t-1}}}\, z_t^* - \frac{\beta_{t-1}}{\sqrt{(1-\gamma_t)(1-\beta_{t-1})}}\, \hat{\mu}_{\theta_t}(z_t^*)}_{\hat{z}_t^*} + \sigma_t \epsilon^*, \tag{8.44}$$

we obtain the (denoised) input to the next iteration, where $\hat{z}_t^*$ is given by (8.43). We repeat until $x^* = z_0^*$ yields a realistic looking generated data sample. Observe that randomness of this sample is due to the initial random z_T^* as well as to the ϵ^* that was generated in each iteration.

Notice from (8.44) that z_{t-1}^* follows a multivariate normal distribution with the conditional (on z_t^*) mean vector being $\hat{z}_t^*$ and the covariance matrix being $\sigma_t^2 I$. Since with the reparameterization trick, $\hat{z}_t^*$ is a replacement of $\mu_{\theta_t}(z_t^*)$, in production, the distribution of z_{t-1}^* is approximately equal to $p_{\theta_t}(\cdot \,|\, z_t^*)$ which is what we aim to achieve.

8.3 Generative Adversarial Networks

Generative adversarial networks or GANs, are generative deep learning architectures that have made great impact on the field of synthetic data generation. For example in Figure 8.5 we can see synthetic images, all created via various GAN architectures, with several paradigms of application. In this section our aim is to highlight the key ideas of generative adversarial networks, focusing on mathematical principles. The basic setup of using such a generative model to approximately create samples from the distribution $p(x \,|\, z)$ as was illustrated in Figure 8.1. We now present details of generative adversarial network architectures.

(a) **(b)** **(c)**

Figure 8.5: Images generated via various GAN architectures[7] with various types of paradigms. (a) Faces with 1024×1024 resolution. (b) Picture to picture generation. (c) A text to image application.

The GAN Approach to Generative Modelling

The general idea of GAN modelling is to train a neural network, called the *generator* and denoted here as $f_\theta^G(\cdot)$. This model applied to a noise vector z^*, generates a data point x^* that ideally appears similar to other data points $x \in \mathcal{D}$. Training is motivated by an *adversarial* game like approach, where we simultaneously train both the generator $f_\theta^G(\cdot)$, and another network called the *discriminator*, and denoted here as $f_\phi^D(\cdot)$. Hence training implies learning the generator parameters θ while in parallel also learning the discriminator's parameters ϕ.

The generator's purpose is to create "fake" data samples and the discriminator's purpose is to try and distinguish between fake and real samples. As such, the discriminator is a binary classifier, which when presented with some data point $\tilde{x}$, would ideally be able to output a probability value near 1 if $\tilde{x} \in \mathcal{D}$ (real) and a probability value near 0 if $\tilde{x} = x^*$ (fake).

While in a normal classification setting, having a well performing disciriminator $f_\phi^D(\cdot)$ is desirable, in the GAN setting, the discriminator is actually used to help train the generator $f_\theta^G(\cdot)$. Once training is complete, we would generally have that $f_\phi^D(\tilde{x})$ is on average near $1/2$ for both real $\tilde{x} \in \mathcal{D}$ and fake $\tilde{x} = x^*$, with $x^* = f_\phi^G(z^*)$. As such, during training of a GAN, we can expect to see a sequence of network parameters,

$$\underbrace{(\phi^{(1)}, \theta^{(1)})}_{\text{iteration 1}}, \underbrace{(\phi^{(2)}, \theta^{(2)})}_{\text{iteration 2}}, \ldots, \tag{8.45}$$

such that as training progresses to high iterations t, $\theta^{(t)}$ are the parameters of a generator that creates "sensible fake samples", while the discriminator parameters $\phi^{(t)}$, are no longer able to discriminate between "fake" and "real", and then yield performance near $1/2$.

Empirically it has been experienced that one needs to train both the generator and discriminator together, in order to achieve the desired behaviour. The alternative of starting with an already trained discriminator, and then training a generator is not feasible, first because

[7] Image (a) is thanks to Karras, et. al. [222]. Image (b) is thanks to Isola, et. al. [204]. Image (c) is thanks to Kang, et. al. [219].

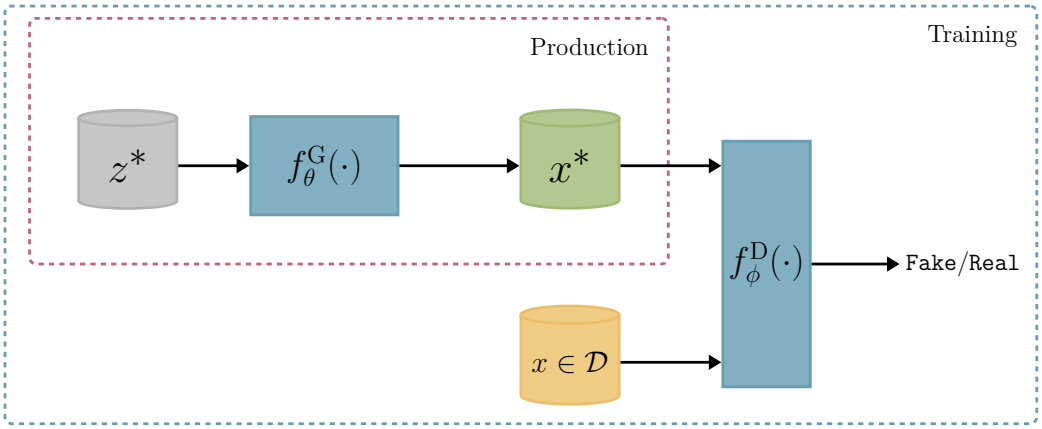


Figure 8.6: The GAN architecture for learning the generator $f_\theta^G(\cdot)$. A random z^* passes through the generator to produce x^*. The generator is trained simultaneously with the discriminator $f_\phi^D(\cdot)$, whose role is to determine if its input is fake, x^*, or real, $x \in \mathcal{D}$.

such a trained discriminator is not available without the generator trained, and secondly because the joint training provides a loss landscape for the generator parameters θ that is suitable for gradient based learning. In this sense, training a GAN, follows a *dynamic equilibrium* approach which can often be posed as a *mathematical game* between two players. In fact, as we present below, at the equilibrium point of this game, one can properly use mini-max game analysis to analyze some properties of the optimization.

Figure 8.6 illustrates the general GAN architecture where in training we use both the generator and discriminator networks, while in production only the generator network is used for creating samples from $p(x \mid z)$. Note that the latent space size of z, denoted m, is typically in the order of several dozen to several hundred dimensional and it is typically taken to be a multivariate standard normal distribution as in the previous generative models of this chapter.

Training a GAN

The key to training a GAN with steps of a sequence such as (8.45) is to alternate between learning ϕ for the discriminator, and learning θ for the generator. For the discriminator, training involves improving the binary classifier $f_\phi^D(\cdot)$ and for the generator, training is *adversarial* since it involves seeking parameters θ that yield the "worst possible" generator $f_\theta^G(\cdot)$ for a given discriminator $f_\phi^D(\cdot)$. We cast these alternative goals as a *minimax objective*, a notion that we now describe.

First note that as in Section 8.1 we use $\mathcal{D}$ to denote the set of data points and assume that there are n such data points. In addition we assume some set of random latent variable samples which we denote as $\mathcal{Z}^*$ with each $z^* \in \mathcal{Z}^*$ being an m-dimensional random vector. In practice, one does not need to randomly sample all of $\mathcal{Z}^*$ in one go, but can rather resample from a standard multivariate normal distribution every time we seek an arbitrary element of $\mathcal{Z}^*$. Yet for simplicity of the exposition and analysis we assume that the number of elements in $\mathcal{Z}^*$ is the same as in $\mathcal{D}$, i.e., n.

With this notation, based on $\mathcal{D}$ and $\mathcal{Z}^*$, for fixed generator parameters θ, the objective for the discriminator parameters ϕ can be taken as the classic binary cross entropy objective; recall (3.11) of Chapter 3. Specifically, from the discriminator's perspective we want $x \in \mathcal{D}$ to be considered as a positive example and we want $f_\theta^G(z^*)$ for $z^* \in \mathcal{Z}^*$ to be considered as a negative example. That is, the discriminator applied to x should be as close to 1 as possible, and the discriminator applied to $f_\theta^G(z^*)$ should be as close to 0 as possible. Now given the generator's parameters θ, together with $\mathcal{D}$ and $\mathcal{Z}^*$, a binary cross entropy loss function (scaled by a factor of 2) can be formulated as,

$$C_\theta(\phi\,;\,\mathcal{D},\mathcal{Z}^*) = -\frac{1}{n}\Big(\sum_{x\in\mathcal{D}} \log\big(f_\phi^D(x)\big) + \sum_{z^*\in\mathcal{Z}^*} \log\big(1 - f_\phi^D\big(f_\theta^G(z^*)\big)\big)\Big). \tag{8.46}$$

Hence the objective of learning the discriminator parameters is,

$$\min_\phi C_\theta(\phi\,;\,\mathcal{D},\mathcal{Z}^*). \tag{8.47}$$

The generator's learning process is adversarial as the generator wishes for $C_\theta(\phi\,;\,\mathcal{D},\mathcal{Z}^*)$ to be as high as possible. Specifically, from the generator's point of view, we seek to find parameters θ that are the worst possible parameters (maximization of C_θ) for the optimal choice of the discriminator. That is, we seek to solve

$$\max_\theta \Big(\min_\phi C_\theta(\phi\,;\,\mathcal{D},\mathcal{Z}^*)\Big). \tag{8.48}$$

The joint objective (8.48) is generally called a minimax objective as it captures the competing goals of both players in the game.

To implement (8.48) we use an iterative algorithm that goes through iterates as in (8.45) and in each iteration alternates between ϕ and θ, with multiple steps within the iteration for each. Specifically, at iteration t we first carry out mini-batch based gradient descent steps on the loss function (8.46) where $\theta = \theta^{(t-1)}$ is fixed and we improve ϕ, starting with $\phi^{(t-1)}$ in the iteration. The result of these mini-batch gradient descent steps for iteration t is $\phi^{(t)}$.

In the second part of iteration t, we seek to maximize (8.46) over θ while keeping $\phi^{(t)}$ fixed. Now since the first term in (8.46) does not depend on θ, this maximization is equivalent to gradient descent steps (minimization) for,

$$\min_\theta \frac{1}{n}\sum_{z^*\in\mathcal{Z}^*} \log\big(1 - f_{\phi^{(t)}}^D\big(f_\theta^G(z^*)\big)\big). \tag{8.49}$$

Such iterative gradient based learning, ideally approximates a solution of (8.48). In practice one often tunes the number of mini-batch discriminator steps and the number of z^* samples for the generator steps, carried out per iteration t. One problem that sometimes arises during training is called *mode collapse* and it is a situation where the generator is stuck at a point in the parameter space of θ where it generates samples that do not cover the breadth of the data $\mathcal{D}$. In general, when carrying out diagnostics of GAN training, we expect the performance of the discriminator to converge to about 1/2.

Minimization of the Jensen-Shannon Divergence

The training procedure outlined above can be cast in a theoretical setting. Motivated by the discriminator loss (8.46) we can describe a theoretical construct which abstracts what a GAN training procedure actually optimizes. Specifically, we now shift to probabilistic thinking, similar to the notions of Section 8.1 in the context of variational autoencoders and Section 8.2 in the context of diffusion models.

For this let us define three probability distributions used in the analysis. First, the distribution of the data $\mathcal{D}$ is assumed to be captured by $p_{\mathcal{D}}(\cdot)$. Let us assume this is a probability distribution covering all of $\mathbb{R}^p$. Then the distribution of each of the latent variables $\mathcal{Z}^*$ is assumed be captured by $p_{\mathcal{Z}*}(\cdot)$. Let us assume that this a probability distribution over $\mathbb{R}^m$ which typically is multivariate standard Gaussian and hence covers the whole latent space. Finally, the distribution of the output of the generator is captured by $p_G(\cdot)$. This last distribution can in theory be obtained by applying the generator $f_\theta^G(\cdot)$ as a transformation on the random variables in $\mathcal{Z}^*$. Like the distribution of the data, this is a distribution over $\mathbb{R}^p$ and we assume it also covers the whole space.

Ideally we would like a GAN to learn the generator parameters θ such that $p_G = p_{\mathcal{D}}$, i.e., that the distribution of the generator is the same as the distribution of the data. To capture such a goal, it turns out that in the context of GANs, a good measure of the distance between the two distributions is the Jenson-Shannon divergence, defined in (B.7) of Appendix B. Specifically, let us denote,

$$J_\theta = \mathrm{JSD}(p_G \,\|\, p_{\mathcal{D}}) = \frac{D_{\mathrm{KL}}(p_{\mathcal{D}} \,\|\, p_{\mathcal{M}}) + D_{\mathrm{KL}}(p_G \,\|\, p_{\mathcal{M}})}{2}, \;\; \text{where } p_{\mathcal{M}}(u) = \frac{1}{2}\big(p_G(u) + p_{\mathcal{D}}(u)\big),$$

and $D_{\mathrm{KL}}(\cdot \,\|\, \cdot)$ is the KL-divergence. Hence for generator parameters θ, the divergence value J_θ captures how far off our generator is from the actual data. In the ideal situation of $p_G = p_{\mathcal{D}}$ we have that $J_\theta = 0$, otherwise it is greater than 0. Note that upon representing the KL-divergences as integrals and manipulating the 2 constant we have,

$$J_\theta = \frac{1}{2}\int_{\mathbb{R}^p} p_{\mathcal{D}}(u) \log\Big(\frac{p_{\mathcal{D}}(u)}{p_{\mathcal{D}}(u) + p_G(u)}\Big) du + p_G(u) \log\Big(\frac{p_G(u)}{p_{\mathcal{D}}(u) + p_G(u)}\Big) du + \log 2. \quad (8.50)$$

Consider now the discriminator loss (8.46) and assume that $n \to \infty$. In this case, due to laws of large numbers, the loss can be rephrased in terms of expectations and expressed as,

$$\overline{C}(\phi, \theta\,;\, p_{\mathcal{D}}, p_{\mathcal{Z}^*}) = -\mathbb{E}_{p_{\mathcal{D}}} \log\big(f_\phi^D(X)\big) - \mathbb{E}_{p_{\mathcal{Z}^*}} \log\big(1 - f_\phi^D\big(f_\theta^G(Z)\big)\big), \quad (8.51)$$

where the first expectation is of the random variable X distributed according to $p_{\mathcal{D}}(\cdot)$ and the second expectation is with respect to the latent random variable Z distributed according to $p_{\mathcal{Z}^*}(\cdot)$. This expected loss can further be manipulated as,

$$\begin{aligned}
\overline{C}(\phi, \theta\,;\, p_{\mathcal{D}}, p_{\mathcal{Z}^*}) &= -\mathbb{E}_{p_{\mathcal{D}}} \log\big(f_\phi^D(X)\big) - \mathbb{E}_{p_G} \log\big(1 - f_\phi^D(W)\big) \\
&= -\int_{\mathbb{R}^p} p_{\mathcal{D}}(u) \log\big(f_\phi^D(u)\big) du - \int_{\mathbb{R}^p} p_G(u) \log\big(1 - f_\phi^D(u)\big) du \\
&= -\int_{\mathbb{R}^p} p_{\mathcal{D}}(u) \log\big(f_\phi^D(u)\big) du + p_G(u) \log\big(1 - f_\phi^D(u)\big) du,
\end{aligned}$$

where in the first equation we set $W = f_\theta^G(Z)$ which is a random variable distributed according to $p_G(\cdot)$. Hence the second expectation in the first equation is with respect to the

distribution $p_G(\cdot)$. Moving from the first equation to the second step we simply represent the expectations as integrals and then since both integrations are on the same domain we can move from the second step to the third step.

Now let us identify a pattern inside the final integral of the form $a\log(y)+b\log(1-y)$, where $a = p_{\mathcal{D}}(u)$, $b = p_G(u)$, and $y = f_\phi^D(u)$. It is easy to check that for $a, b > 0$, the function $g(y) = a\log(y) + b\log(1-y)$ is maximized at $y = a/(a+b)$. Hence, for any y,

$$a\log(y) + b\log(1-y) \leq a\log(\frac{a}{a+b}) + b\log(\frac{b}{a+b}) \tag{8.52}$$

with the inequality being an equality at $y = a/(a+b)$.

We can now use (8.52) to bound the integrand in the final expression for $\overline{C}(\phi, \theta\,;\, p_{\mathcal{D}}, p_{\mathcal{Z}^*})$ as follows,

$$-\int_{\mathbb{R}^p} p_{\mathcal{D}}(u)\log\big(\frac{p_{\mathcal{D}}(u)}{p_{\mathcal{D}}(u)+p_G(u)}\big)du + p_G(u)\log\big(\frac{p_G(u)}{p_{\mathcal{D}}(u)+p_G(u)}\big)du \leq \overline{C}(\phi, \theta\,;\, p_{\mathcal{D}}, p_{\mathcal{Z}^*}).$$

Or, using the expression for J_θ in (8.50) we have

$$2\log 2 - 2J_\theta \leq \overline{C}(\phi, \theta\,;\, p_{\mathcal{D}}, p_{\mathcal{Z}^*}), \tag{8.53}$$

where at the best theoretical generator parameters, θ, we have that $J_\theta = 0$ and the inequality is an equality with $\overline{C} = 2\log 2 \approx 1.386$.

Let us note some parallels between this Jensen-Shannon based analysis for GANS and the ELBO analysis of Section 8.1 for variational autoencoders. In the variational autoencoder case our inherent implicit goal is maximum likelihood estimation of θ and since we cannot achieve such estimation in a computationally efficient manner, we resort to (approximate) optimization of the lower bound, ELBO. Nevertheless, a theoretical inequality such as (8.18), and the additional optimization over the encoder's ϕ in (8.20) ensures that once we have optimized ELBO both in terms of θ and ϕ, then the desired maximum likelihood estimation is also achieved.

In our GAN case, the inequality (8.53) helps justify a similar idea. If we apply a minimax style optimization on it, as in (8.48) then we obtain,

$$2\log 2 = \max_\theta \Big(\min_\phi \overline{C}(\phi, \theta\,;\, p_{\mathcal{D}}, p_{\mathcal{Z}^*}) \Big),$$

which is the best possible value.

Variations to the Objective Function

While the ideas of GANs presented up to now are useful in their own right, there is much room for architectural improvements. On the one hand one may consider different structures for $f_\phi^D(\cdot)$ and $f_\theta^G(\cdot)$, such as for example setting the discriminator and generator neural networks to be convolutional networks. On the other hand, one may revise the objective function (8.46) as well as its expected value formulation (8.51), to yield better training performance. We now focus on this approach, where variations of the objective function are considered.

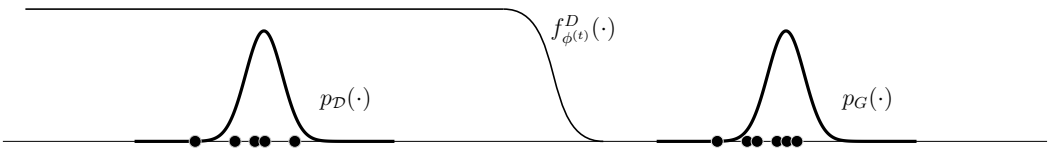


Figure 8.7: A schematic of a potential scenario during early phases of GAN training. The data distribution $p_{\mathcal{D}}(\cdot)$ and the generator distribution $p_G(\cdot)$ are far while the discriminator $f^D_{\phi^{(t)}}(\cdot)$ is already capable of separating the two distributions well. This reduces the "signal" for generator training.

In general, problems may occur in initial training iterations when generator parameter values, $\theta^{(t)}$, are nearly arbitrary. In such a case there is often extreme separation between $p_{\mathcal{D}}(\cdot)$ and $p_G(\cdot)$, because the latter distribution is nearly arbitrary. Such a difficult situation requires a "signal" from the discriminator that will force learning to improve subsequent $\theta^{(t)}$ values. However, an objective such as (8.46), does not always cater for such learning. In particular, see Figure 8.7, where for simplicity we assume the data is one dimensional and we plot data points together with $p_{\mathcal{D}}(\cdot)$ as well as the generator distribution $p_G(\cdot)$ associated with some arbitrary $\theta^{(1)}$. In such a case, after only a few iterations, the discriminator parameters $\phi^{(t)}$ may quickly be trained to separate between the two distributions. The problem with this is that at that point, when considering gradients for minimization of the generator, as in (8.49), there will often be a "lack of signal" (or nearly zero gradients) for learning the generator parameters $\theta^{(t)}$. This situation is quite common in practice, and often prohibits effective learning of GANs.

As a consequence of such scenarios, multiple alternative objective formulations have been proposed, some of which yield much better training performance. We now outline a few of these ideas where we first describe a simple modification called *non-saturating GAN* (NS-GAN). We then describe a deeper idea using *Wasserstein distances* which yields a framework called *Wasserstein GAN* (W-GAN), and finally we discuss improvements to W-GAN, which involve regularization concepts.

The first idea of *non-saturating GAN* (NS-GAN) is simply to modify the generator objective (8.49) to use $-\log(u)$ instead of $\log(1-u)$. Namely, the NS-GAN generator objective is,

$$\min_{\theta} \quad -\frac{1}{n}\sum_{z^*\in\mathcal{Z}^*}\log\left(f^D_{\phi^{(t)}}\left(f^G_{\theta}(z^*)\right)\right), \tag{8.54}$$

and NS-GAN retains the same discriminator objective (8.46). The motivation for such a modification is simply the fact that at $u \approx 0$, the derivative of $\log(1-u)$ (as in the original formulation) is approximately -1, while the derivative of derivative of $-\log(u)$ (as in NS-GAN) is nearly negative infinite. This may yield an improvement for initial phases of training and is particularly useful for cases when $p_{\mathcal{D}}(\cdot)$ is highly separated from $p_G(\cdot)$ and the discriminator parameters already yield good separation as in Figure 8.7. In this case, $f^D_{\phi^{(t)}}\left(f^G_{\theta^{(t)}}(z^*)\right) \approx 0$ and thus the NS-GAN generator objective (8.54) yields much higher magnitude gradients than the original generator objective (8.49), due to the nearly negative infinite derivative.

For both the original GAN and NS-GAN, as training progresses and the expected value of $f^D_{\phi^{(t)}}\left(f^G_{\theta}(z^*)\right)$ approaches $1/2$, the objectives (8.54) and (8.49) behave similarly. Hence in principle, the NS-GAN modification should not hamper with training. However, while the

simple idea of NS-GAN may appear like a good improvement, in practice it has shown to yield unstable gradient updates. We chose to present NS-GAN here, merely as simple idea where modification of the objective can potentially yield training performance improvement.

A more fundamental way to improve GANs is based on the *Wasserstein distance*. This is a concept used to determine the "distance" between two probability distributions, with a similar goal of JS-divergence, yet with a completely different approach that yields different properties of the distance metric.

Formally when presented with two probability distributions, in our case, $p_{\mathcal{D}}(\cdot)$, and $p_G(\cdot)$, the Wasserstein distance, $\mathcal{W}(\cdot,\cdot)$, may be represented as

$$\mathcal{W}\left(p_{\mathcal{D}}, p_G\right) = \inf_{p_\Pi \in \Pi(p_{\mathcal{D}}, p_G)} \mathbb{E}_{(x,\tilde{x})\sim p_\Pi} \|x - \tilde{x}\|. \tag{8.55}$$

Let us unpack (8.55) by assuming that our data and generator are each in $\mathbb{R}^p$. Here $\Pi\left(p_{\mathcal{D}}, p_G\right)$ is the space of all probability distributions over $\mathbb{R}^{2p}$ where the first p coordinates are for the data and the following p coordinates are for the generator. This space of distributions is defined such that each element distribution $p_\Pi \in \Pi\left(p_{\mathcal{D}}, p_G\right)$ has a marginal distribution of the first p coordinates of p_Π that agrees with $p_{\mathcal{D}}$, and likewise the marginal distribution of the remaining p coordinates of p_Π agrees with p_G. That is, for every distribution $p_\Pi \in \Pi\left(p_{\mathcal{D}}, p_G\right)$,

$$\begin{aligned} p_{\mathcal{D}}(x_1, \ldots, x_p) &= \int \cdots \int p_\Pi(x_1, \ldots, x_p, u_1, \ldots, u_p)\ du_1 \cdots du_p, \\ p_G(\tilde{x}_1, \ldots, \tilde{x}_p) &= \int \cdots \int p_\Pi(u_1, \ldots, u_p, \tilde{x}_1, \ldots, \tilde{x}_p)\ du_1 \cdots du_p. \end{aligned} \tag{8.56}$$

The infimum in (8.55) acts "like a minimum" and formally seeks the greatest lower bound. This minimization is over the expectation of the Euclidean norm $\|x - \tilde{x}\|$ with x and $\tilde{x}$ each elements of $\mathbb{R}^p$. Thus the pair $(x, \tilde{x}) \in \mathbb{R}^{2p}$ is distributed according to p_Π and hence x is (marginally) distributed according to $p_{\mathcal{D}}$ and $\tilde{x}$ is (marginally) distributed according to p_G. For clarity, note that the expectation in (8.55) can be represented as,

$$\mathbb{E}\|x - \tilde{x}\| = \int \cdots\cdots \int \sqrt{\sum_{i=1}^{p} (x_i - \tilde{x}_i)^2}\ \ p_\Pi(x_1, \ldots, x_p, \tilde{x}_1, \ldots, \tilde{x}_p)\ dx_1 \cdots dx_p\ d\tilde{x}_1 \cdots d\tilde{x}_p. \tag{8.57}$$

As an extreme example, consider the case where $p_{\mathcal{D}} = p_G$. Then one particular $p^*_\Pi \in \Pi\left(p_{\mathcal{D}}, p_G\right)$ is a distribution that only has support on elements $u_1, \ldots, u_p, u_{p+1}, \ldots, u_{2p}$ where $(u_1, \ldots, u_p) = (u_{p+1}, \ldots, u_{2p})$, and the density over each such element is $p_{\mathcal{D}}(u_1, \ldots, u_p)$ which is the same as $p_G(u_{p+1}, \ldots, u_{2p})$. In this case, using p^*_Π in (8.57) we get 0 because probability mass (or density) is only concentrated on points $(x, \tilde{x}) \in \mathbb{R}^{2p}$ where $x = \tilde{x}$. Hence this p^*_Π minimizes (8.57), and hence the Wasserstein distance is 0. However, when $p_{\mathcal{D}} \neq p_G$, one can show that the Wasserstein distance is strictly positive.

In general, the joint distribution p_Π captures an "earth moving" plan which describes how to shift mass from one distribution $p_{\mathcal{D}}$ to the other p_G. To get a feel for this interpretation, let us grossly simplify the situation and assume that $p_{\mathcal{D}}$ and p_G are discrete one dimensional ($p = 1$) distributions on a finite domain with values, such as 1, 2, and 3. In this case, the joint distribution p_Π is on the 3×3 grid, capturing the probability mass over the 9 possible joint locations. In this simplistic case, the marginal distribution assumptions (8.56) can be

written as,

$$\begin{aligned} p_{\mathcal{D}}(x) &= \textstyle\sum_{\tilde{x}=1}^{3} p_{\Pi}(x,\tilde{x}) \quad \text{for} \quad x = 1,2,3, \\ p_{G}(\tilde{x}) &= \textstyle\sum_{x=1}^{3} p_{\Pi}(x,\tilde{x}) \quad \text{for} \quad \tilde{x} = 1,2,3, \end{aligned} \tag{8.58}$$

where $p_{\Pi}(\cdot,\cdot)$, $p_{\mathcal{D}}(\cdot)$, and $p_G(\cdot)$ are probability masses.

Now we may interpret $p_{\Pi}(x,\tilde{x})$ as a "plan" of how much mass (or earth) to shift from location x to location $\tilde{x}$. For this, the marginal distribution assumptions (8.58) serve as constraints where the first constraints imply that mass is shifted properly out of $p_{\mathcal{D}}$ and the second constraints imply that all resulting mass ends up creating p_G. The Wasserstein distance is then the expectation of $\|x-\tilde{x}\|$ for the optimal plan, or optimal joint distribution. Note that in such a discrete finite case, one may actually formulate and solve this optimization problem using *linear programming*. While such an approach is not important in deep learning practice, ideas of *duality theory* from linear programming play a role as we describe below.

One important attribute of the Wasserstein distance is that it is sensitive to differences in the "location" of the distributions and captures such differences in a much better way than the JS-divergence (based on the KL-divergence). As an illustration let us focus on a case with $p = 1$ where we can easily plot $p_{\mathcal{D}}$ and p_G as marginals. Consider Figure 8.8 where we plot a red pair $(p_{\mathcal{D}}, p_G)$ with the distributions concentrated at quite far away locations, and a green pair $(p_{\mathcal{D}}, p_G)$ with the distributions closer together. We cannot exactly see the Wasserstein distance because it requires minimization over all possible $\Pi(p_{\mathcal{D}}, p_G)$ distributions. However, in this example any possible $p_{\Pi} \in \Pi(p_{\mathcal{D}}, p_G)$ is concentrated around the respective red or green cloud. Hence we roughly see the Wasserstein distance as marked in the figure, representing the difference between the center of the cloud and the diagonal $x = \tilde{x}$ line. Importantly, the distance for the red pair is much higher than that of the green pair. Now in contrast, if we were to consider the JS-divergence, then in both the red and the green pair cases, the divergence would be very close[8] to 0.

An important result for the Wasserstein distance, called the *Kantorovich-Rubinstein duality theorem*, provides us with a dual representation of the minimization problem in (8.55). This result implies the following representation of $\mathcal{W}(p_{\mathcal{D}}, p_G)$, for any $K > 0$,

$$\mathcal{W}(p_{\mathcal{D}}, p_G) = \frac{1}{K} \sup_{\|h\|_L \leq K} \Big\{ \mathbb{E}_{x\sim p_{\mathcal{D}}}\, h(x) \;-\; \mathbb{E}_{\tilde{x}\sim p_G}\, h(\tilde{x}) \Big\}. \tag{8.59}$$

Let us unpack (8.59). The supremum in (8.59) acts "like a maximum" and formally seeks the lowest upper bound. Continuing to assume that $p_{\mathcal{D}}$ and p_G are distributions over $\mathbb{R}^p$, this maximization searches over functions $h : \mathbb{R}^p \to \mathbb{R}$ that also satisfy the K-Lipschitz property, denoted by $\|h\|_L \leq K$. A function $h(\cdot)$ is K-Lipschitz if

$$\|h(u) - h(v)\| \leq K\|u - v\|, \qquad \text{for all} \qquad u, v \in \mathbb{R}^p.$$

This implies that the function does not ascend or descend in any direction with steepness greater than K. Note that in stating the Kantorovich-Rubinstein duality theorem using $K = 1$ is common, yet for our GAN purposes keeping K free is preferable.

The elegance and applicability of (8.59) is that instead of searching over all possible joint probability distributions (earth moving plans) as in (8.55), we now only need to search over

[8] In fact, one needs to assume that the support of the $p_{\mathcal{D}}$ and p_G distributions overlaps for the JS-divergence to be properly defined.

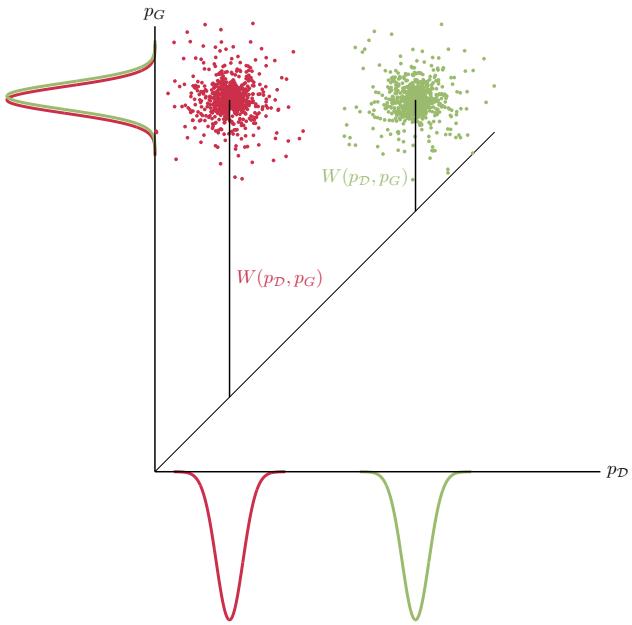


Figure 8.8: Two pairs of distributions where in the red pair, the distributions are farther away than in the green pair. While we cannot exactly see the Wasserstein distance in this case, we get an approximate feeling for it using the distances marked with vertical lines between the centers of the point clouds and the diagonal line.

all K-Lipschitz functions. It turns out, that in the context of generative adversarial networks, the latter formulation is much easier to implement. We omit further details of the duality derivation between (8.55) and (8.59), and now show how to use (8.59) in a GAN setting.

Since p_G is determined by the generator $f_\theta^G(\cdot)$, with a Wasserstein distance approach our overall goal in a generative setting is to find generator parameters θ that minimize $\mathcal{W}(p_{\mathcal{D}}, p_G)$. Replacing the supremum in (8.59) by a maximum, and dropping the $1/K$ constant term, we have and overall objective,

$$\min_{\theta} \quad \max_{\|h\|_L \leq K} \Big\{ \mathbb{E}_{x \sim p_{\mathcal{D}}} h(x) \; - \; \underbrace{\mathbb{E}_{\tilde{x} \sim p_G} h(\tilde{x})}_{\text{Depends on } \theta} \Big\}. \tag{8.60}$$

The key idea of W-GAN is now to treat the function $h(\cdot)$ as a discriminator, even though it is no longer a binary classifier. Following the previous notation, we still denote this discriminator as $f_\phi^D(\cdot)$ with parameters ϕ, even though now the architecture of this discriminator allows it to output any potential real value, not just in $[0, 1]$. With this, (8.60) can be approximated with the objective,

$$\min_{\theta} \quad \max_{\phi \in \Phi_K} \Big\{ \frac{1}{n} \sum_{x \in \mathcal{D}} f_\phi^D(x) \; - \; \frac{1}{n} \sum_{z^* \in \mathcal{Z}^*} f_\phi^D\big(f_\theta^G(z^*)\big) \Big\}, \tag{8.61}$$

where now Φ_K is some space of discriminator parameters that approximately enforces a K-Lipstchitz condition on $f_\phi^D(\cdot)$. That is, with every $\phi \in \Phi_K$ we have that $f_\phi^D(\cdot)$ is K-

Lipstchitz. Hence the discriminator objective, posed as a minimization problem constrained on Φ_k (and dropping the $1/n$ terms) is,

$$\min_{\phi\in\Phi_K}\Big\{-\sum_{x\in\mathcal{D}} f^D_\phi(x) \;+\; \sum_{z^*\in\mathcal{Z}^*} f^D_\phi\big(f^G_\theta(z^*)\big)\Big\}. \tag{8.62}$$

In training a W-GAN we iterate between discriminator and generator steps, and hence in the discriminator steps we carry out mini-batch gradient descent steps for (8.62). The constraint of keeping $\phi\in\Phi_K$ is discussed below. The outcome of such a discriminator iteration is $\phi^{(t)}$ and then in carrying out generator steps for (8.61) we carry out iterations for

$$\min_\theta \;\; -\sum_{z^*\in\mathcal{Z}^*} f^D_{\phi^{(t)}}\big(f^G_\theta(z^*)\big). \tag{8.63}$$

Compare the W-GAN discriminator problem (8.62) with the original GAN discriminator problem (8.47), and similarly compare the W-GAN generator problem (8.63) with the original GAN generator problem (8.49) (similarly we may compare to NS-GAN with generator problem as in (8.54)). Quite surprisingly, if we ignore the Φ_K constraint, the differences are only very minor where the W-GAN does not use logarithms. In practice, W-GAN generally overcomes the problems illustrated in Figure 8.7 by enhancing the algorithm with a much better discrimination signal.

We still need to specify how to enforce the Φ_K, K-Lipstchitz constraint in (8.62). For this there are multiple techniques. The most basic technique is called *weight clipping* where we constrain the weights for the neural network $f^D_\phi(\cdot)$ to lie in some range, e.g., $[-0.05, 0.05]$. For almost all standard deep learning architectures, this will mean that $f^D_\phi(\cdot)$ will be K-Lipstchitz for some K that depends on the architecture of the network and on the clipping half-range 0.05.

A different approach which in practice has generally shown better performance is *gradient penalty*. This approach relies on the following property of (8.59). If a function $h^*:\mathbb{R}^p\to\mathbb{R}$ attains the supremum in (8.59), then this function has with probability one, a unit gradient norm on any random point x_η that is a convex combination between a point drawn from $p_\mathcal{D}$ and a point drawn from p_G. That is, if we take x as a random point from $p_\mathcal{D}$ and x^* as a random point from p_G (for some generator), then for any $\eta\in[0,1]$, the point $x_\eta=\eta x+(1-\eta)x^*$ satisfies,

$$\|\nabla h^*(x_\eta)\|=1, \quad \text{with probability 1.} \tag{8.64}$$

We omit the proof, yet we use this property algorithmically to approximately enforce the constraint $\phi\in\Phi_K$. To do so, the gradient penalty approach uses (8.64) as an optimization constraint in place of $\phi\in\Phi_K$. This constraint is then integrated in the objective using an approximate Lagrange multiplier approach.

Specifically for some tunable $\lambda>0$, we modify the discriminator objective (8.62) to

$$\min_\phi\Big\{-\sum_{x\in\mathcal{D}} f^D_\phi(x) \;+\; \sum_{z^*\in\mathcal{Z}^*} f^D_\phi\big(f^G_\theta(z^*)\big)+\lambda\sum_{x_\eta\in\mathcal{D}_\eta\mathcal{Z}^*}\big(\|\nabla f^D_\phi(x_\eta)\|-1\big)^2\Big\}, \tag{8.65}$$

where the set of random points $\mathcal{D}_\eta\mathcal{Z}^*$ is a set of points based on pairs $x\in\mathcal{D}$ and $z^*\in\mathcal{Z}^*$, where each x has as single matching z^*, and for each pair we generate a uniformly random $\eta\in[0,1]$ and set $x_\eta=\eta x+(1-\eta)f^G_\theta(z^*)$.

In practice the gradient penalty approach works very well and out of all of the GAN objective variations that we presented, it is the most common approach used. In a typical algorithm that implements mini-batch gradient descent steps for (8.65), we would sample a subset of the data $\mathcal{D}$ and a matching sized sample of $\mathcal{Z}^*$ followed by a sample of $\mathcal{D}_\eta\mathcal{Z}^*$ with each element determined by a uniform η.

Beyond Data Generation with GANs

Up to now we considered the application of generative adversarial networks for generative modelling where an unlabelled dataset $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$ is used to train a generator $f_\theta^G(\cdot)$ and then this generator can create samples similar to those in $\mathcal{D}$. This is already useful for a variety of applications, one of which is *data augmentation*, where we can then use additional generated (fake) samples to train other models. Yet, there are many more tasks where generative adversarial networks can be employed. We now outline a few paradigms that extend the basic GAN architecture and allow us to handle additional tasks. We outline the *conditional generation paradigm*, the *image to image paradigm*, and the *style transfer paradigm*.

With the *conditional generation paradigm* the resulting generator from training is not just a function of the latent space, but is also a function of additional variables. Namely we can describe the generator as $f_\theta^G(z, w)$ where z is still a random latent variable and the newly introduced w contains some additional information. One such example of conditional generation, called a *conditional generative adversarial network* (C-GAN) is where the data is labeled and is of the form $\mathcal{D} = \{(x^{(1)}, y^{(1)}) \ldots, (x^{(n)}, y^{(n)})\}$. Here we can use the additional information w as some attribute vector that potentially encodes the particular label. Hence for example in a case of the MNIST digits dataset, our GAN $f_\theta^G(z, w)$ will create random digits, where w may indicate which digit to create using one-hot encoding. The basic way in which such a GAN is trained, is by also supplying the attribute vector to the discriminator during training. We omit further details.

A slightly different form of conditional generation is also described in the *auxiliary classifier generative adversarial network* (AC-GAN) architecture. Here the generator is similar to the C-GAN case where w is specifically a class of the desired sample to generate, yet in training, the discriminator has two outputs, one of which is the binary classification fake/real as before, and another is a probability vector determining which class is detected. Such a training process, often results in a better architecture for conditional generation than the original C-GAN.

Ideas of conditional generation can be extended in multiple ways, where one notable way is the *interpretable representation learning generative adversarial network* (Info-GAN) approach. Here we are back to unlabeled data such as $\mathcal{D} = \{x^{(1)}, \ldots, x^{(n)}\}$, and we use the GAN training process to separate some elements of w "out of z", where typically w is of smaller dimension than z. The resulting generator, $f_\theta^G(z, w)$, then uses w for tweaking specific attributes of the image, where these attributes are not controlled apriori, but are rather discovered during the learning process. As a concrete example, again in the case of MNIST digits, we may have $w \in \mathbb{R}^4$ where after the training process it turns out that the first coordinate of w controls the stroke widths of the digits, the second coordinate controls the slant of the digit, and other two coordinates of w either have some interpretation or not. The beauty of Info-GAN is in the self discovery of these properties. The general idea of training in such a framework is that the discriminator outputs additional variables that are

meant to match the inputs w. The analysis involves mutual information with basic ideas from information theory, and is beyond our scope.

With the *image to image paradigm*, or more generally *data to data paradigm* our goal is not to generate random general images or data examples, but rather to be able to *enhance* images or data. Assume a training dataset such as $\mathcal{D} = \{(x_{\text{in}}^{(1)}, x_{\text{out}}^{(1)}), \ldots, (x_{\text{in}}^{(n)}, x_{\text{out}}^{(n)})\}$, where each $x_{\text{in}}^{(i)}$ has lower information content than the corresponding $x_{\text{out}}^{(i)}$. For example $x_{\text{in}}^{(i)}$ may be a black and white image of a portion a street map, while $x_{\text{out}}^{(i)}$ may be a color satellite image of the same geographic location.[9] The goal of an image to image generator, say $\tilde{G}$, is to operate on input images similar to $x_{\text{in}}^{(i)}$ in structure, and create the corresponding enhanced output image. Then when presented with a new input image $\tilde{x}_{\text{in}}$, we would have that the trained generator applied to it, $f_{\theta}^{\tilde{G}}(\tilde{x}_{\text{in}})$, is the enhanced image. Hence for example in the geographic case, we can generate fake satellite images, based on street map images. Other applications include the color enhancement of images or films, and almost any domain where datasets such as $\mathcal{D} = \{(x_{\text{in}}^{(1)}, x_{\text{out}}^{(1)}), \ldots, (x_{\text{in}}^{(n)}, x_{\text{out}}^{(n)})\}$ appear.

The basic training architecture in the image to image paradigm is based on a discriminator that distinguishes between real pairs $(x_{\text{in}}^{(i)}, x_{\text{out}}^{(i)})$, and fake pairs $(x_{\text{in}}^{(i)}, x_{\text{out}}^{*(i)})$, where in the first pair, $x_{\text{out}}^{(i)}$ matches $x_{\text{in}}^{(i)}$ from the data, while the second pair, $x_{\text{out}}^{*(i)}$ is generated. That is, the discriminator, is the function $f_{\phi}^{\tilde{D}}(\tilde{x}_{\text{in}}, \tilde{x}_{\text{out}})$ that tries to determine if the pair it is fed with is completely real or if the second image fed to it is generated. In this sense, the image to image paradigm is similar to the C-GAN paradigm. Key differences include the fact, that in the image to image case, one would often want a complicated generator network such as a U-Net.[10]

In deep learning, *style transfer*, sometimes called *neural style transfer*, is an area broader than generative adversarial networks, mostly focusing on image data. The general theme of style transfer is to modify input images such that they appear to resemble a certain style. Examples that have been popularized include the recreation of arbitrary images using the distinctive drawing style of Vincent van Gogh. Within the context of generative adversarial networks, the *style transfer paradigm* uses GANs for style transfer and for similar image modifications.

A notable GAN architecture specifically focused on style transfer is the *style-GAN*. Here, multiple new ideas are introduced in the context of the generator network, specifically for image data using convolutional architectures. One of the notable features of StyleGAN is its ability to generate high-resolution images with remarkable detail and diversity. Style-GAN has been widely used in applications such as image synthesis, artistic expression, and creating realistic faces with customizable attributes.

Several of the key ideas of Style-GAN are specific to convolutional architectures and include *upsampling*, as well as *adaptive instance normalization* which is somewhat similar to batch normalization of Section 5.6 and layer normalization, used in Section 7.5. We do not focus on these ideas here but rather comment on the general structure of the generator which differs from generators used in other paradigms. The Style-GAN generator is composed of two distinct functions, a *mapping network* and a *synthesis network*, which we denote as $f_{\theta_m}^{G}(z_1)$ and $f_{\theta_s}^{G}(z_2, w)$ respectively.

[9] For this specific example and other examples of image to image generation, it is not hard to create such training data.

[10] See the notes and references in Chapter 6 for background on the U-Net architecture.

The mapping network's role is to convert a latent space noise vector z_1 into a more meaningful representation $w = f^G_{\theta_m}(z_1)$ which is an intermediate variable that is later amenable to manipulation. The synthesis network's role is somewhat similar to the Info-GAN generator where z_2 serves as noise, while w captures style information. One way to use a trained generator without any style modifications is via,

$$x^* = f^G_{\theta_s}\big(z_2^*, \underbrace{f^G_{\theta_m}(z_1^*)}_{w}\big),$$

where z_1^* and z_2^* are noise vectors, and x^* is the resulting random output image. However, in more advanced applications we may keep one latent noise vector fixed, while perturbing the other noise vector, and/or varying the intermediate variable w. Variations of this form enable Style-GAN to create meaningful modifications of images.

8.4 Reinforcement Learning

The topic of *reinforcement learning* deals with controlling dynamic processes over time. This is different than most tasks presented in the book, which on their own are typically applied at one instant of time. For example, classification is based on a static input x at some time, to determine an output $\hat{y}$ relevant for that time. Yet the classification task does not care about previous or future classification decisions or the time at which it is carried out. In contrast, with reinforcement learning, we have the task of making decisions as time evolves, where our decisions often affect the evolution of the system and thus our decisions need to take planning ahead into consideration.

Typical applications of reinforcement learning include decisions for automatic pilots, robot control, playing strategy games, financial management, and other applications that involve decisions over a time horizon. Indeed, in the second decade of this century, the deep learning variant of reinforcement learning, namely *deep reinforcement learning*, made big headlines with the game of Go, where the world's best Go players were eventually beaten by an engineered deep reinforcement learning platform. Earlier, it was shown that deep reinforcement learning systems can be trained to automatically play classic arcade Atari video games.

In general, the field of *control theory* is the engineering field where *automatic control* systems are designed and analyzed. This field has a rich history, including many advances made during the space programs of the 1950's and the 1960's, including concepts such as *Kalman filtering* and many related ideas. The area of reinforcement learning can be cast as part of the control theory world, yet it has much more of a computer science flavour to it, and with the advent of deep learning, has essentially become part of the deep learning toolkit. Nevertheless, today, ideas of reinforcement learning are part of the control theory world.

A schematic representation of reinforcement learning is in Figure 8.9. The basic setup is that a system, or *environment*, is controlled by an *agent*. Other processes may be taking place in the environment as well, perhaps not directly controlled by the agent, these are modelled via randomness. As time progresses the agent sends their control decisions in the form of *actions* to the environment, and in turn it receives *reward* from the environment as well as *observations*.

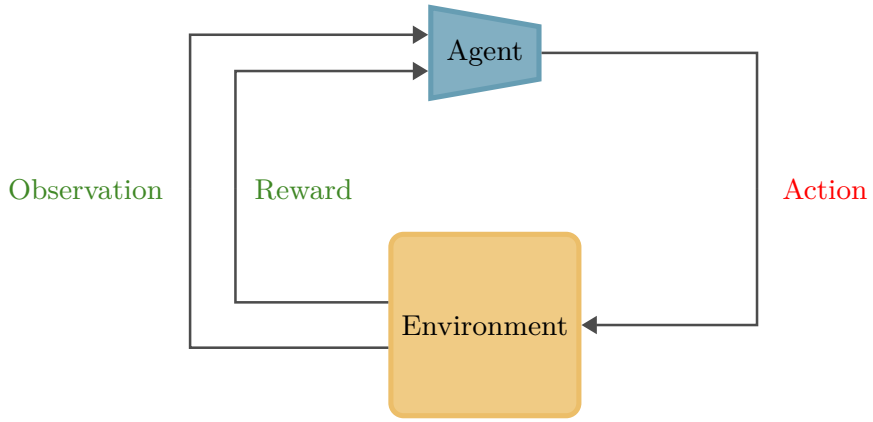


Figure 8.9: The setup in reinforcement learning is that an agent (or controller) applies actions to an environment (also known as system). The agent's decisions of which actions to take, are based on reward obtained together with observations. One illustrative example of the environment is a video game where multiple random things happen and the agent is a player.

Mathematically, we cast reinforcement learning in terms of an area called *Markov decision processes*.[11] Here, as we explain in this section, the evolution of the environment being controlled is modeled as a variant of a Markov chain, where the state evolution of the chain also depends on control decisions made. A key goal in the area of Markov decision processes is the study and application of algorithms for controlling the system in some optimal manner. Reinforcement learning, aims to solve the same problem, with the difference that in the Markov decision processes case we assume to have full knowledge of the system evolution model, whereas in the reinforcement learning case, the system model is unknown and hence needs to either be learned, or optimal control strategies need to be learned. With reinforcement learning, the agent often learn incrementally while controlling actual systems. In some cases learning is based on simulations of the system before deployment into the real world, while in other cases, operational systems are controlled and learned simultaneously.

Our focus of this section is to first present the Markov decision processes framework and to further discuss optimal strategies using Bellman equations. Then, after briefly discussing solution methods for such equations we move onto the reinforcement learning case, where the system model is unknown. There are multiple types of algorithms for reinforcement learning, and we focus only on the Q-learning approach. After understanding the basic Q-learning algorithm, we finally see how concepts of deep neural networks can be integrated by replacing the so-called Q-function with a deep neural network. This presentation here is merely an introduction to the field, and more readings are suggested at the end of the chapter.

Markov Decision Processes

Assume some system evolving over discrete time $t = 0, 1, 2, \ldots$, where at any time t, the system state is z_t. This state may describe the location of a robot, or the vector of velocities

[11] In certain cases the observations are only a partial description of the environment state, in which case the formal framework is that of *partially observable Markov decision processes*. Yet in our brief exposition, we only consider full state observations.

in multiple directions together with accelerations, or a discrete state of a game, or one of many other possibilities depending on the application.

Let us first ignore decisions and system control. In this case we can model the evolution of $z_0, z_1, z_2, \ldots$ as a *Markov chain*, where some transition kernel $p_t(z_{t+1} \,|\, z_t)$ determines the probability of moving from a given state z_t to state z_{t+1}, between time t and time $t+1$. In this form, the transitions appear to depend on the time t, yet we may also assume that the evolution is *time homogenous* and does not depend on time t. In this case, common to this section, the transition kernel can be presented with the notation $p(z_{t+1} \,|\, z_t)$, i.e., without a subscript t. As already stated in Section 8.2 in the context of Markovian hierarchical variational autoencoders, a Markov chain satisfies the Markov property. In our case here, this property is best interpreted as implying that given any history prior to time t, if we are given z_t, that history does not have an affect on the future. This then means that the *state information* at time t, namely z_t, is enough to determine probabilities of the future.

As a simple example, consider a scenario focusing on the engagement level of a student. Assume that there are 10 levels of engagement, $1, 2, \ldots, 10$, where at level 1 the student is not engaged at all and at the other extreme, at level 10, the student is maximally engaged. One may model the probability transitions of engagement as,

$$
\begin{aligned}
p(j \,|\, 1) &= \begin{cases} 0.6 & j = 1, \\ 0.4 & j = 2, \\ 0 & \text{otherwise}, \end{cases} \\
p(j \,|\, i) &= \begin{cases} 0.6 & j = i - 1, \\ 0.2 & j = i, \\ 0.2 & j = i + 1, \\ 0 & \text{otherwise}, \end{cases} \qquad \text{for} \qquad i = 2, \ldots, 9, \\
p(j \,|\, 10) &= \begin{cases} 0.6 & j = 9, \\ 0.4 & j = 10, \\ 0 & \text{otherwise}. \end{cases}
\end{aligned}
\tag{8.66}
$$

In this case, it is evident that at any time step, the student's engagement can either improve by one level, decrease by one level, or stay unchanged, with the probabilities specified by $p(\cdot \,|\, \cdot)$ as above. If at onset at time 0 we start at some given level $z_0 \in \{1, \ldots, 10\}$, as time progresses, the level will fluctuate according to the Markov chain stochastic process. The level of engagement is then the system or environment in this very simple case.

A Markov decision process is a generalization of a Markov chain where now some decision (also known as or control) is used by the agent. In this simple student engagement example, assume that at any time t we can choose to "stimulate" the student, for example by suggesting a prize, or "not to stimulate". Naturally, it is sensible to model the fact that stimulation will improve transitions up towards higher levels. For example, one model for these transitions

under stimulation captures transition probabilities via $p^{(1)}(z_{t+1} \mid z_t)$, as,

$$
\begin{aligned}
p^{(1)}(j \mid 1) &= \begin{cases} 0.2 & j = 1, \\ 0.8 & j = 2, \\ 0 & \text{otherwise}, \end{cases} \\
p^{(1)}(j \mid i) &= \begin{cases} 0.1 & j = i - 1, \\ 0.1 & j = i, \\ 0.8 & j = i + 1, \\ 0 & \text{otherwise}, \end{cases} \qquad \text{for} \qquad i = 2, \ldots, 9, \\
p^{(1)}(j \mid 10) &= \begin{cases} 0.2 & j = 9, \\ 0.8 & j = 10, \\ 0 & \text{otherwise}. \end{cases}
\end{aligned} \tag{8.67}
$$

Compare now the original transition probabilities (8.66) with the revised transition probabilities under "stimulation" (8.67). With these particular values in the model that we chose, transitions with "stimulation" generally have a higher probability of pushing engagement level up. In a Markov decision process, in addition to the state evolution $z_0, z_1, z_2, \ldots$, we also have an *action* chosen by the *controller* or *agent* for any time t. Specifically the sequence $a_0, a_1, a_2, \ldots$ denotes the actions, where in this example we can set that $a_t = 0$ implies "not to stimulate" at time t and that $a_t = 1$ implies to "stimulate" at time t.

If we now define $p^{(0)}(j \mid i)$ as the original probabilities (8.66), then each of the probabilities,

$$
p^{(a)}(j \mid i), \qquad \text{for} \quad a \in \{0, 1\}, \quad i, j \in \{1, \ldots, 10\}, \tag{8.68}
$$

constitutes the transition probability from $z_t = i$ to $z_{t+1} = j$ if action $a_t = a$ is chosen at time t. Hence (8.68) is a specification of the environment (or system) and of how the agent's actions affect that system.

Now with the action sequence, $a_0, a_1, a_2, \ldots$, the state sequence $z_0, z_1, z_2, \ldots$ does not evolve autonomously, but rather depends on the decisions made $a_0, a_1, a_2, \ldots$. So if for example the first decision is $a_0 = 0$, the second is $a_1 = 0$, and the third is $a_2 = 1$, then given some initial z_0, the first probabilities are $p^{(0)}(\cdot \mid z_0)$ which determine a random z_1; then given this value, the second transition probabilities are $p^{(0)}(\cdot \mid z_1)$ which determine a random z_2; and the third transition probabilities are $p^{(1)}(\cdot \mid z_2)$ which determine a random z_3, and so forth.

Having the sequence of decisions $a_0, a_1, a_2, \ldots$ fixed a-priori is called an *open loop control* and in our context this is not common. Instead, we typically wish to determine the action a_t based on the state z_t. This is called *closed loop control* or *feedback control*, because the control decision is based on the current state. A *control policy*, or just a *policy*[12] is a function of the form,

$$
g_{\text{policy}} : \underbrace{\{1, \ldots, 10\}}_{\text{State space}} \quad \to \quad \underbrace{\{0, 1\}}_{\text{Action space}},
$$

where in more general examples the *state space* and *action space* may be much more complex.

Any policy function, $g_{\text{policy}}(\cdot)$, induces a Markov chain specific for that policy. This is because with $a_t = g_{\text{policy}}(z_t)$, the transitions $p^{(a_t)}(\cdot \mid z_t)$ are used for that Markov chain. Hence in

[12] This is in fact called a deterministic Markovian policy.

this example a policy can be seen as a rule for mixing (8.66) and (8.67). With this specific example there are only $2^{10} = 1,024$ possible policies, yet with more complex state spaces or action spaces, the number of possible policies can be huge.

The activity of finding the best feedback control policy for a Markov decision process is the act of solving a Markov decision process. To do so, our model requires a *reward function* to be specified. This reward function applies to every time step t, and captures the benefit of being in a given state, and choosing a given action at that time. The reward function[13] can be viewed as part of the Markov decision process model specification, and is of the form,

$$r : \underbrace{\{1, \ldots, 10\}}_{\text{State space}} \times \underbrace{\{0, 1\}}_{\text{Action space}} \rightarrow \mathbb{R}.$$

Hence for example, $r(4, 0)$ is the reward obtained at a time where the state is $z_t = 4$ and the chosen action is $a_t = 0$ (no stimulation).

For our specific example, let us assume a reward function,

$$r(z, a) = z - 1.5a, \tag{8.69}$$

where higher student engagement levels have higher reward with a linear increase via the z component, and further, we pay a price of 1.5 reward units every time we set $a_t = 1$. Hence for example $r(4, 0) = 4$ and $r(4, 1) = 2.5$. From a modelling perspective, this latter reward being lower can be viewed as due to the cost of the prize for stimulation. Observe that the reward is always from the viewpoint of the agent controlling the system (the student in this particular example is the environment).

We now want to find a policy $g_{\text{policy}}(\cdot)$ that is best in terms of this reward over the whole time horizon. There are multiple ways to accumulate reward and in our exposition we consider only the *infinite horizon expected discounted reward* objective. Here, we first set $\gamma \in (0, 1)$ which a fixed hyper-parameter called a *discount factor*. Then the contribution at time t is taken to be,

$$\gamma^t \, r(z_t, \underbrace{g_{\text{policy}}(z_t)}_{a_t}).$$

These contributions are accumulated to form the infinite horizon expected discounted reward objective, which depends on the initial state $z_0 = z$ and is,

$$V_{g_{\text{policy}}}(z) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t \, r(z_t, \underbrace{g_{\text{policy}}(z_t)}_{a_t}) \;\middle|\; z_0 = z \right]. \tag{8.70}$$

The role of the discount factor, γ, is to capture the importance of near present times vs. far future times. With γ low (near 0), far future times t have little effect on this contribution. Similarly with γ high (near 1), these far future times have much more of an affect on the contribution.

Observe that this objective function, (8.70), is parameterized by the policy, $g_{\text{policy}}(\cdot)$, because the policy determines how actions a_t are chosen, given values of z_t. Our goal is to find an optimal policy that maximizes (8.70), yet it may appear that there are multiple objectives

[13] Here we focus on the time-homogenous reward function which does not depend on the current time.

here because (8.70) depends for every initial state $z_0 = z$. Nevertheless, a property of the type of Markov decision processes that we are using is that there exists a policy that can maximize $V_{g_{\text{policy}}}(z)$ for all initial states $z_0 = z$. Hence we seek,

$$g^*_{\text{policy}} = \operatorname*{argmax}_{g_{\text{policy}}} V_{g_{\text{policy}}}(z), \qquad \text{for all initial states} \quad z. \tag{8.71}$$

In our simple student engagement example, one may even find such a policy by enumerating all possible policies and for each policy evaluating the expectation in (8.70) either via Monte-Carlo simulation or via analytic properties of Markov chains. For example, if the discount factor is at $\gamma = 0.6$ then an optimal policy turns out to be,

$$g^*_{\text{policy}}(z) = \begin{cases} 0 & z \in \{1, 2\}, \\ 1 & z \in \{3, 4, 5, 6, 7\}, \\ 0 & z \in \{8, 9, 10\}. \end{cases} \tag{8.72}$$

It is not obvious a-priori why this is the best policy, but considering it we see that for low and high engagement levels (1, 2, 8, 9, and 10) it is not worth to pay the price for stimulation, whereas otherwise for intermediate engagement levels $(3, 4, 5, 6,$ and $7)$, it is worth stimulating the student. The strength of Markov decision processes is that they expose such policies, where the optimal control for any time t and current state z_t implicitly takes the future evolution into account via (8.70).

While such a Markov decision processes framework is in principle very powerful, we are faced with two problems. The first problem is to have better means than exhaustive search for solving (8.71) to find optimal policies, and we discuss such means shortly. The second problem is the fact that models can seldom be specified as we did here, with probabilities that correctly capture reality. That is, realistic scenarios would involve very complex transition kernels $p^{(a)}(j \,|\, i)$ in contrast to the simplistic specification of probabilities in (8.66) and (8.67). This second problem is handled by reinforcement learning methods.

Bellman Equations, the Value Function, and the Q-function

In characterizing the solution of (8.71) an important object is the *value function*. This real valued function, denoted as $V^*(z)$, where z is any element of the state space, is defined as,

$$V^*(z) = \max_{g_{\text{policy}}} V_{g_{\text{policy}}}(z), \tag{8.73}$$

where $V_{g_{\text{policy}}}(z)$ is from (8.70). Hence the value function determines the optimal infinite horizon expected discounted reward, when starting at a state $z_0 = z$.

A result in the study of Markov decisions processes is that we can characterize the value function via a non-linear system of equations called the *Bellman equations*. For the case of finite state and action spaces, these equations are,

$$V^*(z) = \max_a \underbrace{\Big\{ r(z, a) + \gamma \sum_{z'} p^{(a)}(z' \,|\, z)\, V^*(z') \Big\}}_{Q^*(z,a)}, \qquad \text{for all states} \quad z. \tag{8.74}$$

Here the maximum is over all actions a (e.g., $\{0, 1\}$ in the student engagement example). Further, inside the maximum on the right hand side, we have the *Q-function*, $Q^*(z, a)$, which

captures the "quality" of choosing action a on state z. Note that the summation in (8.74) is taken over all states z' (e.g., $\{1, \ldots, 10\}$ in the student engagement example).

One can informally derive the Bellman equations (8.74) via what is known as the *dynamic programming principle* where the first term of $Q^*(z, a)$ is the immediate reward, $r(z, a)$ and the second part is the expected next step reward, discounted by one time step and hence multiplied by γ. Theoretically it can be shown that any function $V^*(\cdot)$ that satisfies the Bellman equations is the value function as in (8.73).

With more complex state spaces that are not necessarily discrete, we can rewrite the Q-function as,

$$Q^*(z, a) = r(z, a) + \gamma \, \mathbb{E}\big[\, V^*(z_{t+1}) \,|\, z_t = z, a_t = a \big], \tag{8.75}$$

while keeping in mind that the time-homogenous assumptions imply that $Q^*(z, a)$ is the same for every time t and hence we can use z_0, a_0, and z_1 in place of z_t, a_t, and z_{t+1} respectively. With the formulation of the Q-function as in (8.75), we have a more general expression for the Bellman equation (8.74), where $\sum_{z'} p^{(a)}(z' \,|\, z)\, V^*(z')$ is replaced by $\mathbb{E}[V^*(z_1) \,|\, z_0 = z, a_0 = a]$. This formulation encompasses states spaces that are not discrete.

Importantly, knowing either the Q-function or the value function presents us with an optimal policy. If we know the Q-function, $Q^*(\cdot, \cdot)$, then we can determine an optimal policy $g^*_{\text{policy}}(\cdot)$ as in (8.71) via,

$$g^*_{\text{policy}}(z) = \operatorname*{argmax}_a \, Q^*(z, a). \tag{8.76}$$

If we know the value function, $V^*(\cdot)$, then we can first evaluate the Q-function via (8.75) where we compute the expectation using the explicit model (sometimes using Monte Carlo simulation if needed) and then with this computed Q-function we can evaluate (8.76).

The classic study of Markov decision processes deals with existence and optimality of solutions to the Bellman equations. It also deals with algorithms for solving these equations to find the value function and hence to find an optimal policy via (8.75) and (8.76). We briefly discuss such solution methods now.

Solving Bellman Equations

The two most common algorithms for solving Bellman equations are *value iteration* and *policy iteration*. We focus on value iteration on discrete state spaces. The algorithm is based on the recursion,[14]

$$Q^{(t+1)}(z, a) = r(z, a) + \gamma \sum_{z'} p^{(a)}(z' \,|\, z) \left(\max_{a'} Q^{(t)}(z', a') \right), \qquad \text{for all } \; z \text{ and } a. \tag{8.77}$$

Value iteration starts with some initial or arbitrary guess $Q^{(0)}(\cdot, \cdot)$ and then with each step t we apply (8.77) on all states z and all actions a to get from $Q^{(t)}(\cdot, \cdot)$ to $Q^{(t+1)}(\cdot, \cdot)$. The algorithm terminates when some distance measure applied to $Q^{(t)}(\cdot, \cdot)$ and $Q^{(t+1)}(\cdot, \cdot)$ is below a specified small threshold.

[14] Note that one often writes the recursion in terms of $V^*(\cdot)$ and not in terms of Q-functions as we did here. However, the two formulations are equivalent and our representation is preferable for understanding Q-learning in the sequel.

In quite general settings, convergence of repeated applications of (8.77) to the optimal Q-function is guaranteed and a proof of this relies on the fact that the right hand side of (8.77) is a *contraction mapping*. We omit the details. Yet with value iteration we do not have an indication at what iteration the optimal policy has been discovered. Hence one may need to apply (8.77) many times.

Note that policy iteration, the other algorithm that we mentioned above, remedies the situation. On finite state and action spaces, policy iteration needs to execute for only a finite number of steps before discovering an optimal policy. We do not discuss the policy iteration algorithm further here because Q-learning is based on value iteration.

Back to the simple student engagement example presented earlier, each $Q^{(t)}(\cdot,\cdot)$ can simply be implemented as a table with $10 \times 2 = 20$ entries since the state space is $\{1,\ldots,10\}$ and the action space is $\{0,1\}$. One sometimes informally calls this a *Q-table*. If we were to use value iteration for finding the optimal policy, we would first initialize $Q^{(0)}(\cdot,\cdot)$ with some 20 arbitrary values. We would then apply (8.77) for each $z \in \{1,\ldots,10\}$ and $a \in \{0,1\}$. This application would directly use the probabilities specified in (8.66) and (8.67), and the reward function specified in (8.69). Applying such value iteration steps is thus straightforward to execute recursively. After multiple steps, we can then determine the policy using (8.76) and this is in fact how we obtained the example optimal policy (8.72). However, more complex examples are harder to implement and importantly in realistic scenarios we often do not know the exact transition probabilities and reward function. Instead, we take the approach of learning the Q-function, which we describe now.

Q-learning

The idea with *Q-learning* is to learn the Q-function (8.75) and obtain some estimate $\hat{Q}(z,a)$ for all states z and all actions a. Learning the Q-function does not mean learning, the individual components, $r(\cdot,\cdot)$, $p^{(\cdot)}(\cdot\,|\,\cdot)$, and $V^*(\cdot)$. It rather means learning $\hat{Q}(\cdot,\cdot)$ as a whole. One can then use this estimate in (8.76) in place of $Q^*(z,a)$ to obtain a policy that is approximately optimal via,

$$\hat{g}_{\text{policy}}(z) = \operatorname*{argmax}_a \hat{Q}(z,a). \tag{8.78}$$

Before seeing how Q-learning learns $\hat{Q}(\cdot,\cdot)$, we mention that as a reinforcement learning algorithm, one sometimes applies Q-learning in parallel to controlling a system, or controlling a simulation of the system. This is different from other learning paradigms in this book where the learning and production activities are often separate. Such a mix of controlling and learning involves ongoing estimates of the Q-function, $\hat{Q}^{(t)}(\cdot,\cdot)$ where at any time t we use $\hat{Q}^{(t)}(\cdot,\cdot)$ in place of $\hat{Q}(\cdot,\cdot)$ for (8.78).

When carrying out such a mix of learning and controlling, we know that at time t, the latest $\hat{Q}^{(t)}(\cdot,\cdot)$ is only an estimate. Hence we also employ *state exploration* as part of the policy. For this one typical approach known as the *epsilon greedy* approach, uses some pre-specified decreasing sequence of probabilities, $\varepsilon_0, \varepsilon_1, \varepsilon_2, \ldots$, and at any time t, we control the system with a randomized policy,

$$\hat{g}^{(t)}_{\text{policy}}(z) = \begin{cases} \operatorname*{argmax}_a \hat{Q}^{(t)}(z,a), & \text{with probability } 1-\varepsilon_t, \\ \text{random action } a, & \text{with probability } \varepsilon_t. \end{cases} \tag{8.79}$$

With this epsilon greedy approach we know that at time t there is a chance of ε_t of selecting an arbitrary action that is most likely sub-optimal. Yet it allows us to potentially navigate the system to parts of the state space that would otherwise remain unexplored.

Now let us consider the Q-learning algorithm. Here the idea is to update some part of $\hat{Q}^{(t)}(\cdot,\cdot)$ at any time step based on the following available information: (i) the previous state z_t; (ii) the new state z_{t+1}; (iii) the previous action chosen a_t; (iv) the observed reward denoted as r_t which is the reward after applying the action at time t. For this, Q-learning relies on hyper-parameters, $\alpha_0, \alpha_1, \alpha_2, \ldots$, a pre-specified decreasing sequence of probabilities. The recursion of Q-learning is,

$$\hat{Q}^{(t+1)}(z,a) = \begin{cases} (1-\alpha_t)\,\hat{Q}^{(t)}(z,a) + \alpha_t\Big(\underbrace{r_t + \gamma \max_{a'} \hat{Q}^{(t)}(z_{t+1},a')}_{\text{Single sample Bellman estimate}} \Big), & \text{if } z_t = z, a_t = a \\ \hat{Q}^{(t)}(z,a), & \text{otherwise.} \end{cases} \tag{8.80}$$

Key to (8.80) is that we only update the Q-function estimate for one specific z_t, a_t pair at any time step. Further observe that this update is a weighted average of the previous entry $\hat{Q}^{(t)}(z,a)$ and a new component which we denote as the *single sample Bellman estimate*. This term is directly motivated by the value iteration recursion (8.77) and we can observe that it agrees with the right hand side of (8.77) if we were to ignore the probabilities $p^{(a)}(z' \mid z)$. This general form of approximation is called a *stochastic approximation* and its theoretical analysis allows one to prove certain convergence properties of Q-learning. We omit these details.

From an operational point of view, we can now integrate the epsilon-greedy control of (8.79) with the Q-learning recursion of (8.80) to develop a learning scheme that both controls the system and learns how to control it as time progresses. With such a scheme, calibration of the hyper-parameter sequences $\varepsilon_0, \varepsilon_1, \varepsilon_2, \ldots$ and $\alpha_0, \alpha_1, \alpha_2, \ldots$ is often delicate, and one often has to experiment with various hyper-parameter settings in order to get useful results. A theoretical result for Q-learning is that under certain conditions we need the sequence $\alpha_0, \alpha_1, \alpha_2, \ldots$ to satisfy,

$$\sum_{t=0}^{\infty} \alpha_t = \infty, \qquad \text{and} \qquad \sum_{t=0}^{\infty} \alpha_t^2 < \infty. \tag{8.81}$$

Hence the probabilities need to decay quickly enough, but not too quickly. So for example probabilities such as $\alpha_t = 1/(t+1)$ suffice since with these the first (harmonic) series diverges while the second series converges.

While theoretical results based on the condition (8.81) help to place Q-learning on a rigorous footing, from a practical perspective Q-learning on its own is difficult to use effectively. Even in our simple student engagment example where we try to learn the 10×2 Q-table, Q-learning can be challenging. At any time step we only update a single entry of this table. This is already slow, and is further slowed down due to the fact that if for non small times t, α_t is a low probability, then the averaging in (8.80) would keep the new entry $\hat{Q}^{(t+1)}(z,a)$ not far from the previous entry $\hat{Q}^{(t)}(z,a)$. In more complex and realistic scenarios where the state and action spaces are big and sometimes non-discrete, we cannot even tabulate the Q-function in a naive Q-table. Hence approximate Q-learning needs to be carried out. This brings us to *deep reinforcement learning*.

Deep Reinforcement Learning

The key idea with deep reinforcement learning is to approximate the Q-function, $Q^*(z,a)$, with a neural network, $f_\theta^Q(z,a)$. Such a setup allows us to deal with highly complex state spaces and action spaces. The parameters of this network are learned during a *deep Q-learning* algorithm where as the algorithm evolves, we have a sequence of learned parameters $\theta^{(1)}, \theta^{(2)}, \ldots$. With each such $\theta^{(t)}$, we can still use an epsilon greedy policy similar to (8.79) where the control decision is taken as,

$$g_{\text{policy}}(z, \theta^{(t)}) = \begin{cases} \underset{a}{\text{argmax}}\ f_{\theta^{(t)}}^Q(z,a), & \text{with probability } 1-\varepsilon_t, \\ \text{random action } a, & \text{with probability } \varepsilon_t. \end{cases} \tag{8.82}$$

The key is now how to learn $f_\theta^Q(\cdot,\cdot)$ and for this there are many variants of the deep Q-learning algorithm of which we only outline the simplest form. To create a loss function that will help to learn the parameters θ, we use a reinforcement learning concept called *temporal difference learning*. Here the loss at time t is given by,

$$C_t(\theta\,; z_t, a_t, r_t, z_{t+1}) = \Big(\underbrace{r_t + \gamma \max_a f_\theta^Q(z_{t+1}, a)}_{\text{Single sample Bellman estimate}} - f_\theta^Q(z_t, a_t) \Big)^2, \tag{8.83}$$

where the observed state is z_t, the system is controlled via the action a_t, a reward of r_t is obtained, and the resulting next state is z_{t+1}. With such a temporal difference loss, if the neural network parameters θ determine a good approximation for the actual Q-function, then the loss would be generally low, whereas if θ does not describe the Q-function well, then the loss is higher. To see this, revisit the value iteration equation (8.77).

Now with the neural network loss (8.83) defined, we have a very basic deep Q-learning algorithm. We control the system using the control policy of (8.82) and at any time step t we apply a single gradient descent update for the parameters θ based on the loss $C_t(\,\cdot\,; z_t, a_t, r_t, z_{t+1})$. The gradient descent update then modifies the Q-function estimate from $f_{\theta^{(t)}}^Q(\cdot,\cdot)$ to $f_{\theta^{(t+1)}}^Q(\cdot,\cdot)$. Note that like the Q-learning algorithm (without a neural network approximation) described above, in this framework every time involves both a control decision and learning.

In practice, this basic deep Q-learning algorithm typically does not perform well. One problem is the coupling of the gradient descent step and the control decision, both happening only once at any time t. In practice we often seek an algorithm that can on the one hand learn the Q-function approximation using multiple control steps, and on the other hand perform multiple gradient descent steps. For this, one popular variant is to maintain two copies of the network approximating the Q-function, $f_\theta^Q(z,a)$. One copy is updated only every several time steps and is used for the control decisions (8.82), and the other copy is updated with every gradient descent step. Another concept often used is *reply memory* where we control the system for some multiple time steps and use the combination of these time steps for gradient based learning. We do not dive into these technical details here.

8.5 Graph Neural Networks

In this section we introduce and explore neural networks for graph objects, called *graph neural networks* (GNNs). In a similar way to how convolutional neural networks are primarily used for image data, and recurrent neural networks are primarily used for sequence data, graph neural networks are applied on data organized as a (combinatorial) graph. The input data is typically of the form $G = (V, E)$ together with related feature data, where V is some vertex set of the graph, and E is the edge set of that graph. We introduce basic concepts of graphs in the sequel.

Abstractly, a graph neural network can be viewed as a function

$$f_\theta : \mathcal{G} \times \mathcal{F} \to \text{output}, \tag{8.84}$$

where $\mathcal{G}$ is the set of possible input graphs G, $\mathcal{F}$ is the set of possible input features, and the output may be a graph, a classification vector, a scalar value, or similar. Specifically, the features in $\mathcal{F}$ may include weights on edges, or much more complex features associated with edges and nodes. In similar spirit to other deep learning models, graph neural networks often implement $f_\theta(\cdot)$ via a composition of a sequence of steps or layers, each denoted $f^{[\ell]}_{\theta^{[\ell]}}(\cdot)$ as in (5.1). At each such layer ℓ, the input graph and features are transformed.

Applications of Graph Neural Networks

In contrast to domains such as image classification using convolutional neural networks, or natural language translation using sequence models, the applications associated with graph neural networks often require some transformation of the given problem into a graphical representation. Our focus in this section is not on such transformations for applications; see the notes and references at the end of this chapter for specific reading suggestions. An important point to highlight is that most applications of GNNs are not for mimicking human level performance (e.g., image recognition or text understanding), but are rather for gaining insights from complex data, that is otherwise hard to process. We now mention a few general application domains. See Figure 8.10 for an illustration.

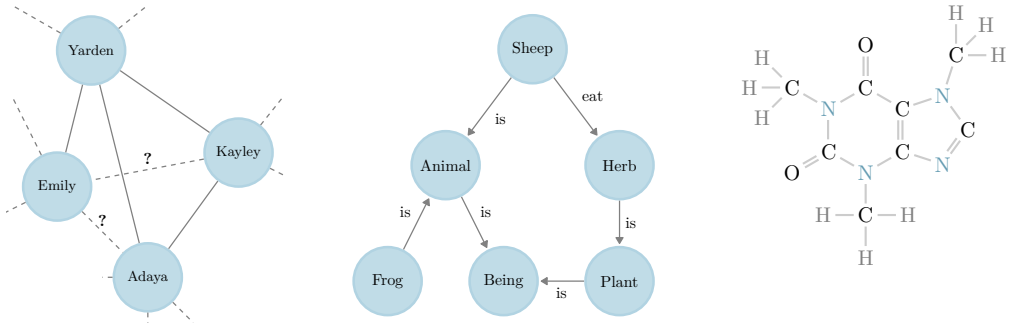


Figure 8.10: An illustration of graph structures arising in several application domains. (a) Connections in social networks described via graphs. The presence of edges with question marks is not known and can be predicted via a graph neural network. (b) Knowledge graphs capturing relationships between entities. Learning about such relationships can be assisted with graph neural networks. (c) Molecular bonds can be described via graphs. Thus classification of molecular structures and the design and discovery of new structures, is also aided by GNN.

One key area of application is social network analysis, where GNNs assist in identifying influential users, detecting communities, and predicting connections between individuals. Additionally, in recommendation systems, GNNs enhance the accuracy and personalization of suggestions by modelling user-item interactions within collaborative filtering settings. Another prominent application is in fraud detection, particularly in financial and e-commerce contexts, where GNNs uncover hidden relationships and anomalous patterns within transaction graphs. In the fields of biology and chemistry, GNNs excel at modelling molecular structures and interactions, offering valuable insights into drug discovery, protein-protein interaction prediction, and chemical property estimation. GNNs are also widely used in traffic analysis for optimizing transportation networks, predicting congestion, and enhancing routing and scheduling.

All of the above applications are generally cases where the tasks are not supposed to replace human level tasks, yet there are also situations where GNNs can enhance or replace human level tasks. Specifically, GNNs have applications in image analysis, enabling tasks like image segmentation, object tracking, and scene understanding, particularly when graphs represent relationships between image regions or structures. Further, in the natural language processing domain, GNNs enhance tasks such as named entity recognition, sentiment analysis, and text classification by analyzing text data represented as dependency trees or semantic graphs.

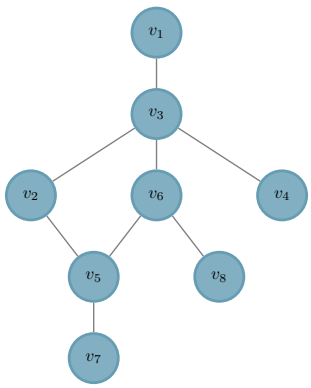
Graph Structures

As alluded to in (8.84) the input to a GNN consists of a graph and features which are elements of $\mathcal{G}$ and $\mathcal{F}$ respectively. We now discuss possible representations of such objects. We begin with a brief outline of graph-theoretic terminology. See Figure 8.11 as a guide.

Recall that a *graph* $G \in \mathcal{G}$, is often denoted $G = (V, E)$. The graph is composed of a *node set* V and an *edge set* E. The node set can be denoted as $V = \{v_1, v_2, \ldots, v_r\}$ where each v_i

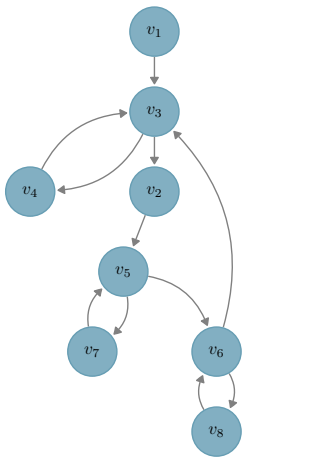
is a node (also known as a *vertex*), and each of the r nodes can represent a distinct entity in the application domain (e.g., a single person, an atom, etc.). The edge set E is a subset of $V \times V$, or in particular is composed of tuples of the form (v_i, v_j) where each such tuple represents an *edge* connecting v_i and v_j. In our terminology we do not allow the elements (v_i, v_i) to be in E, that is, there are no *self loops*.

In some cases the graph is a *directed graph*, where (v_i, v_j) is different from (v_j, v_i) for $i \neq j$, and the former represents an edge (arrow) from v_i to v_j, while the latter is in the opposite direction. In other cases, the graph is an *undirected graph* in which case we can either treat (v_i, v_j) as unordered, or more formally require that if $(v_i, v_j) \in E$ then also $(v_j, v_i) \in E$. In the undirected case, edges are not represented as arrows but rather as links between nodes.



	1	2	3	4	5	6	7	8
1	0	0	1	0	0	0	0	0
2	0	0	1	0	1	0	0	0
3	1	1	0	1	0	1	0	0
4	0	0	1	0	0	0	0	0
5	0	1	0	0	0	1	1	0
6	0	0	1	0	1	0	0	1
7	0	0	0	0	1	0	0	0
8	0	0	0	0	0	1	0	0

(a)



	1	2	3	4	5	6	7	8
1	0	0	1	0	0	0	0	0
2	0	0	0	0	1	0	0	0
3	0	1	0	1	0	0	0	0
4	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	1	0
6	0	0	1	0	0	0	0	1
7	0	0	0	0	1	0	0	0
8	0	0	0	0	0	1	0	0

(b)

Figure 8.11: Directed and undirected graphs each with their associated adjacency matrix. (a) An undirected graph has a symmetric adjacency matrix. (b) A directed graph has an adjacency matrix that is typically not symmetric.

For undirected graphs, the *degree* of a node v_i is the number of edges connected to it, or more formally the number of nodes v_j such that $(v_i, v_j) \in E$. For directed graphs we differentiate between the *out-degree* of node v_i and the *in-degree*. The former is the number of nodes v_j such that $(v_i, v_j) \in E$, while the latter is the number of nodes v_j such that $(v_j, v_i) \in E$. In the undirected case, both out-degree and in-degree are the same at each node.

In the context of undirected graphs, the *neighbours* of a node v_i are the nodes v_j that are connected to v_i with an edge. We denote the set of indices of these neighbours via $\mathcal{N}(v_i)$. Hence the degree is the number of elements in this set. See for example, the undirected graph illustrated in Figure 8.11 (a) where $\mathcal{N}(v_1) = 1$, $\mathcal{N}(v_2) = 2$, $\mathcal{N}(v_3) = 4$, $\mathcal{N}(v_4) = 1$, $\mathcal{N}(v_5) = 3$, etc.

A *path* between two nodes v_i and v_j is a sequence of nodes $(v_{k_1}, v_{k_2}, \ldots, v_{k_m})$ where $v_{k_1} = v_i$, $v_{k_m} = v_j$ and $(v_{k_\ell}, v_{k_{\ell+1}}) \in E$ for $\ell = 1, \ldots, m-1$. A graph is said to be *connected* if there is a path between any two nodes $v_i, v_j \in V$. One trivial graph (which is also connected) is the *complete graph* (also known as the *fully connected graph*) where $(v_i, v_j) \in E$ for all $v_i, v_j \in V$.

Mathematically, one way to represent a graph is via an *adjacency matrix* where we number the nodes as $1, 2, \ldots, r$. Such an $r \times r$ matrix A has entries that are either 0 or 1 where,

$$A_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases} \tag{8.85}$$

For undirected graphs, the adjacency matrix is symmetric ($A_{ij} = A_{ji}$) whereas for directed graphs it is not necessarily symmetric. In either case, since we do not allow self loops, $A_{ii} = 0$ for all i. When representing a graph in computer memory, sometimes an adjacency matrix is suitable, yet at other times, when the graph has fewer edges, sparser representations called *adjacency lists* are more suitable.

Since the node numbering is typically arbitrary, it is often useful to allow all permutations of the node numbering to be valid. Mathematically, we can express this property with the aid of an $r \times r$ *permutation matrix* P with,

$$P_{ij} = \begin{cases} 1 & \text{if } j \text{ replaces } i \text{ in the permutation,} \\ 0 & \text{otherwise.} \end{cases} \tag{8.86}$$

When P is applied to a vector x, namely $\tilde{x} = Px$, the result $\tilde{x}$ has x_j in at index i. Similarly when P is left multiplied to another matrix, the matrix's rows are permuted according to the permutation encoded in P. Also when $P^\top$ is right multiplied to another matrix, the matrix's columns are permuted as such. Finally, applying the permutation to the adjacency matrix we have, that the new matrix $PAP^\top$ represents the adjacency matrix after transforming the numbering of the nodes according to the permutation described by P.

A graph is called a *weighted graph* if for every edge $(v_i, v_j) \in E$ there is also an associated weight $w_{ij} \in \mathbb{R}$ with the edge. One can also associate weights with all elements of $V \times V$ and consider a weight of 0 as the absence of the edge. In this case the adjacency matrix can be enhanced to an *adjacency weight matrix* which we still denote as A, and have $A_{ij} = w_{ij}$.

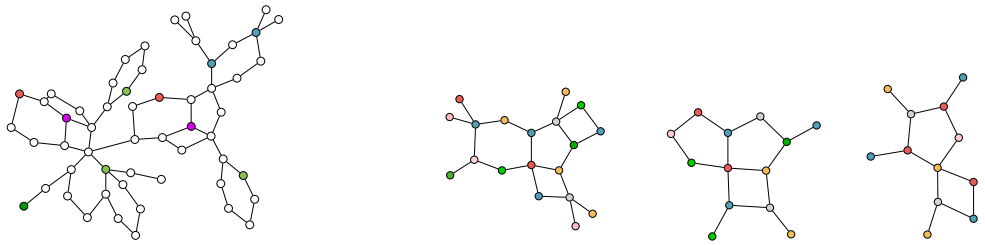


Figure 8.12: Transductive vs. inductive learning. (a) Transductive learning involves a single large input graph. (b) Inductive learning involves multiple separate input graphs.

With this representation,

$$A_{ij} = \begin{cases} w_{ij} & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

One example application of a weighted graph is a map of cities where edges between cities are roads and the weights are the distances in kilometers. Note that this is also a *planar graph* since it can be "drawn" on a plane. Not all graphs are planar. We also mention that a more general object than a graph is a *multi-graph* which permits multiple distinct edges (e.g., roads) between vertices. Some graph neural networks deal with multi-graphs, yet we do not go into this specialization in this section.

The definitions above associated with $G \in \mathcal{G}$ are general graph-theoretic concepts, not specific to graph neural networks. Yet in graph neural networks, there are also additional features, which we denote as elements of $\mathcal{F}$. Specifically, each individual node v_i can carry an associated feature vector $x_{(i)}$ which we generally consider an element of $\mathbb{R}^{p_V}$, for some $p_V \geq 1$. For example, in social networks applications where each individual node is a person, $x_{(i)}$ denotes the p_V features associated with the person such as age, marital status, etc. We call these features *node level features*.

Similarly to node level features, in certain cases we can also consider *edge level features*. Here we have the features $x_{(i \to j)}$ for edge (v_i, v_j), and we assume $x_{(i \to j)} \in \mathbb{R}^{p_E}$, for some $p_E \geq 1$. One can treat a weighted graph as a very simple case where $x_{(i \to j)} = w_{ij}$ and $p_E = 1$. However, in most cases that involve edge level features, $p_E > 1$. For example, in a transportation case where edges are roads (or transportation links) we may have each $x_{(i \to j)}$ represent multiple characteristics of the road such as the number of lanes, the speed limit, toll information, etc.

In this section's exposition of graph neural networks we generally ignore edge level features and focus on data with node level features. In such a case, the features can be organized in a $r \times p_V$ matrix X where each row represents the features of node v_i, namely $x_{(i)}$. That is, $\mathcal{F}$ is the set of all possible feature matrices[15] X. Note also that if we wish to apply a permutation to the node numbering as encoded in a permutation matrix P, then the permuted feature matrix is PX.

[15] Note that if one was to organize edge level features in such an object then a tensor of dimension $r \times r \times p_E$ would be appropriate.

The Structure of Input Data and Tasks

Similarly to the rest of the book, we generally denote the data via $\mathcal{D}$. For graph neural networks this data can come in various forms:

$$\mathcal{D} = \begin{cases} (G, X), & \text{for transductive learning (i),} \\ \{((G^{(1)}, X^{(1)}), y^{(1)}), \ldots, ((G^{(n)}, X^{(n)}), y^{(n)})\}, & \text{for inductive learning (ii),} \\ \widetilde{X}_{(G,X)}, & \text{for graph embedding (iii).} \end{cases} \tag{8.87}$$

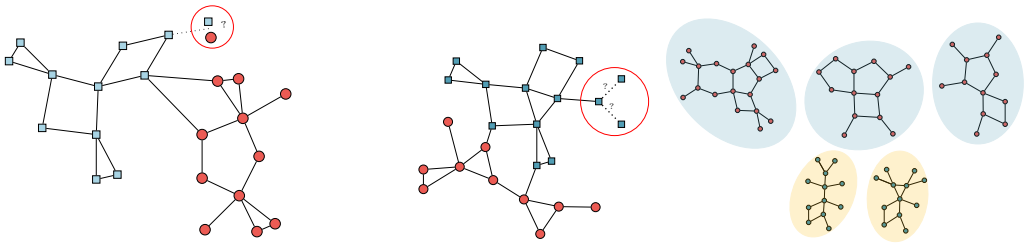
The forms (i) and (ii) in (8.87) are for the two overarching graph neural network learning paradigms, namely, *transductive learning* and *inductive learning*. In the case of transductive learning the data is one (big) input graph together with the features. In the case of inductive learning, the data consists of n different graphs (potentially each of them smaller), each with its own features as well, and also potential labels, $y^{(i)}$. See Figure 8.12 for an illustration of the difference between transductive and inductive learning.

To further understand the difference between transductive and inductive learning consider the following illustrative examples. For transductive learning, consider the social networks domain where we treat a big social network with some missing data as an input and then learn a model for predicting potential connections between individuals (predicting missing edges). As an inductive learning example consider classification of molecules where for example $y^{(i)} = 0$ implies a non-toxic molecule and $y^{(i)} = 1$ implies that molecule is toxic. The training data is then composed of many input graphs, some of which have $y^{(i)} = 0$ and some of which have $y^{(i)} = 1$. We then train a model that operates on an input graph (molecule structure) and outputs $\hat{y}$ which is the estimated probability of being toxic. It can then be transformed into a decision rule, as with any binary classifier.

Form (iii) of the data in (8.87) is different. In this case the data is no longer a graph but rather a transformation of graphical data into vector form using techniques called *graph embeddings*. Here, in a similar nature to word embeddings in the context of natural language processing (see Section 7.1), input graph data and features (G, X) are pre-processed to create a matrix $\widetilde{X}_{(G,X)}$ which summarizes the graph and the features via real valued vectors. In contrast to the first two forms of data (i) and (ii) in (8.87), which are used as input to graph neural networks, the graph embedding form (iii) can be used as input to feedforward networks or other (non graphical) models trained on $\widetilde{X}_{(G,X)}$. Interestingly, some graph embedding techniques themselves are based on graph neural networks; we omit the details.[16]

Given graphical data of the form (8.87), one can train a model as in (8.84) for various different tasks. These can be dichotomized as *tasks on nodes*, *tasks on edges*, or *tasks on graphs*. In the first case the output is associated with nodes and our main goal is to predict outcomes, or impute missing features, for specific nodes in the graph. In the second case the output is associated with edges and our goal is to determine the presence of certain edges that were not originally available in the data, or similarly predict outcomes associated with available edges. In the third case the output is associated with the whole graph and in this case we predict properties of the graph, including classification of graphs, regression, and similar. See Figure 8.13 for an illustration.

[16] Various techniques for graph embedding include DeepWalk, node2vec, GraphSAGE, LINE (Large-scale Information Network Embedding), and HOPE (High-Order Proximity preserved Embedding).



(a) **(b)** **(c)**

Figure 8.13: Various forms of tasks for graph neural networks. (a) Tasks on nodes deal with inference about individual nodes. For example classification of the type of node. (b) Tasks on edges deal with inference about individual edges. For example the existence of an edge or not. (c) Tasks on graphs deal with inference about the whole graph. For example classification of the graph.

Generally, tasks on graphs are carried out in a inductive setting as we require form (ii) of the training data from (8.87). In contrast, tasks on nodes or tasks on edges can be carried out both in an inductive and a transductive level.

The General Structure of a Graph Neural Network Model

In general, a graph neural network model is a function of graph data x from $\mathcal{D}$ of (8.87). Like (5.1) of Chapter 5 we construct the neural network $f_\theta(x)$ of (8.84) with a recursive computation of layers, $f_\theta(x) = f^{[L]}_{\theta^{[L]}}(f^{[L-1]}_{\theta^{[L-1]}}(\ldots(f^{[1]}_{\theta^{[1]}}(x))\ldots))$. Here $f^{[\ell]}_{\theta^{[\ell]}}(\cdot)$ is the ℓ-th layer, and $\theta^{[\ell]}$ are the parameters of that layer. As with other deep learning models, we denote $a^{[\ell]}$ as the result of $f^{[\ell]}_{\theta^{[\ell]}}(a^{[\ell-1]})$ and $a^{[0]} = x$, keeping in mind that x contains both the graph structure and features.

Some models are *dynamic graph neural networks* in which case the graph structure is modified as it is processed by the neural network layers. Other models, are *static graph neural networks* for which $G = (V, E)$ is not modified across layers, and thus G has the same value within each layer. In such a case, it is convenient to represent $a^{[\ell]} = (h^{[\ell]}, G)$ where $h^{[\ell]}$ is sometimes called the *hidden state*. We set $h^{[0]}$ as the input features from x and denote the neural network function as $f_\theta(h^{[0]}, G)$. With this representation, the function of each layer can be denoted as $f^{[\ell]}_{\theta^{[\ell]}}(h^{[\ell-1]}; G)$. Note that G is fixed for all layers, but with each application of the complete $f_\theta(h^{[0]}, G)$, a different graph structure G is possible. As with feedforward neural network models, the parameters of the model are $\theta = (\theta^{[1]}, \ldots, \theta^{[L]})$.

Our focus in this exposition is only on static graph neural networks for which the forward pass[17] is

$$\begin{aligned} h^{[1]} &= f^{[1]}_{\theta^{[1]}}(\overbrace{h^{[0]}}^{\text{Input features}}; G) \\ h^{[2]} &= f^{[2]}_{\theta^{[2]}}(h^{[1]}; G) \\ &\vdots \\ h^{[L-1]} &= f^{[L-1]}_{\theta^{[L-1]}}(h^{[L-2]}; G) \\ \underbrace{\hat{y}}_{\text{output}} &= f^{[L]}_{\theta^{[L]}}(h^{[L-1]}; G) \end{aligned}$$

The specific structure of $h^{[\ell]}$ can vary between applications and may be a matrix, or a higher dimensional tensor. In our case, for simplicity, let us assume it is a matrix of dimension $r \times p_V$ where the i-th row represents the node level features for node v_i in the graph, and there are r nodes in total. In general one may have different column dimensions (number of features) per layer ℓ, yet for simplicity let us assume that this is fixed as p_V throughout the layers.

An important requirement of the layer function $f^{[\ell]}_{\theta^{[\ell]}}(h^{[\ell]}; G)$ is *permutation invariance* where the order of nodes (and consequently, the order of entries in the adjacency matrix) does not affect the network's output. To understand this requirement, assume that we represent G via an adjacency matrix A, and hence the layer's operation can be represented with A in place of G, namely we can denote the layer's function as $f^{[\ell]}_{\theta^{[\ell]}}(h^{[\ell-1]}; A)$. Now for any permutation on the nodes represented via P as in (8.86), we require that,

$$f^{[\ell]}_{\theta^{[\ell]}}(\underbrace{Ph^{[\ell]}}_{\text{Permuted hidden state}}; \overbrace{PAP^\top}^{\text{Permuted adjacency matrix}}) = \underbrace{P\, f^{[\ell]}_{\theta^{[\ell]}}(h^{[\ell]}; A)}_{\text{Permuted hidden state at layer } \ell+1}. \tag{8.88}$$

The permutation invariance requirement in (8.88) then ensures that node numbering is indeed arbitrary and does not does not affect the operation of the model.

Let us now dive into the structure of a single layer except for the final layer. Namely, let us see the structure of $f^{[\ell]}_{\theta^{[\ell]}}(h^{[\ell-1]}; G)$ for $\ell = 1, \ldots, L-1$. Here, similarly to the convolutional neural networks of Chapter 6, graph neural networks try to enforce *locality* and *translation invariance*; see Section 6.1. The translation invariance property is analogous to the permutation invariance property of (8.88). Locality is enforced by requiring that the output of $f^{[\ell]}_{\theta^{[\ell]}}(h^{[\ell-1]}; G)$ for node v_i only depends on the neighbours of v_i. Specifically in our data representation it means that only the rows with indices of neighbours $\mathcal{N}(v_i)$ as well as v_i itself are used to compute the i-th row of $h^{[\ell]}$. Permutation invariance is enforced by using the same form of function (and same parameters) for each target output node, and not considering the actual index of a node but rather only the graphical structure.

[17] Compare with the feedforward network forward pass in (5.4).

Specifically, if we denote the i-th row of $h^{[\ell]}$ via $h^{[\ell]}_{(i)}$ then the computation of the layer associated with node v_i can be represented via,

$$h^{[\ell]}_{(i)} = f^{[\ell]}_{\text{node},\theta^{[\ell]}}\left(h^{[\ell-1]}_{(j)} \quad \text{for} \quad v_j \in \mathcal{N}(v_i) \cup \{v_i\}\right), \tag{8.89}$$

where the function $f^{[\ell]}_{\text{node},\theta^{[\ell]}}(\cdot)$ determines how the hidden state of a node in the next layer is determined by the hidden state of the node and its neighbours in the previous layer.

The final layer L is typically different because the output $\hat{y}$ is often not of the same dimension as $h^{[\ell]}$. In this case, we just retain the final layer action via the general function $f^{[L]}_{\theta^{[L]}}(\cdot)$.

Message Passing Schemes

The operation of (8.89) is based on a so-called *message passing* scheme where two steps called *aggregate* and *update*[18] are executed one after the other. These steps break up the operation of the $f^{[\ell]}_{\text{node},\theta^{[\ell]}}(\cdot)$ function via, a function $f_{\text{aggregate}}(\cdot)$ and a function $f^{[\ell]}_{\text{update},\theta^{[\ell]}}(\cdot)$ and are executed as,

$$\begin{aligned} m^{[\ell]}_{(i)} &= f_{\text{aggregate}}\left(h^{[\ell-1]}_{(j)} \quad \text{for} \quad v_j \in \mathcal{N}(v_i) \cup \{v_i\}\right), \\ h^{[\ell]}_{(i)} &= f^{[\ell]}_{\text{update},\theta^{[\ell]}}\left(h^{[\ell-1]}_{(i)},\ m^{[\ell]}_{(i)}\right). \end{aligned}$$

Observe that the aggregate function is the same for all layers and is not parameterized, while the parameters for layer ℓ, $\theta^{[\ell]}$ are associated only with the update function.

As is evident, the aggregation step utilizes all the hidden states of the neighbours of the node, say v_i, to achieve a summary, $m^{[\ell]}_{(i)}$, which we call the *message*. Note that in some cases it also uses node v_i itself (*self loops*) and in other cases not (*no self loops*). The message collects hidden state information of neighbouring nodes.

In our exposition, we assume that the message for node i is an element of $\mathbb{R}^{p_V}$. Similarly to the hidden state, for simplicity, we assume this dimension, p_V, does not depend on the layer ℓ.

Typical aggregation functions are the sum, the mean, or the element-wise maximum. In our exposition we focus on the sum as an illustrative simple case where,

$$m^{[\ell]}_{(i)} = \begin{cases} \sum_{j\,:\,v_j \in \mathcal{N}(v_i)} h^{[\ell-1]}_{(j)}, & \text{for sum aggregation without self loops,} \\ \left(\sum_{j\,:\,v_j \in \mathcal{N}(v_i)} h^{[\ell-1]}_{(j)}\right) + h^{[\ell-1]}_{(i)}, & \text{for sum aggregation with self loops.} \end{cases} \tag{8.90}$$

The aggregate and update steps are carried out for all nodes in the graph and thus one may view this scheme as messages passing along neighbours within the graph. Note that this type of architecture is sometimes called a *message passing neural network (MPNN)*; see Figure 8.14 for an illustration.

[18] Sometimes this is called *combine*.

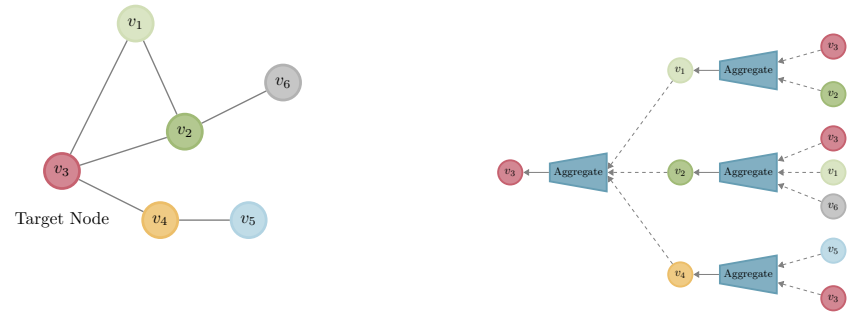


Figure 8.14: Aggregation in a message passing scheme. (a) An input graph where the messages for node v_3 are considered. (b) The application of the aggregation via multiple layers (2 layers in this case) yielding the message aggreagated for the node. Observe the neighbour of v_3 in each layer.

Considering the whole graph, note that like the $r \times p_V$ hidden state matrix $h^{[\ell]}$, we can also consider an $r \times p_V$ message matrix $m^{[\ell]}$, where the i-th row is $m^{[\ell]}_{(i)}$. Now focusing on the unweighted case, using the graph's adjacency matrix A from (8.85), one can verify that the message matrix resulting from the sum aggregation (8.90) is,

$$m^{[\ell]} = \begin{cases} A\, h^{[\ell-1]}, & \text{for sum aggregation without self loops,} \\ (A + I)\, h^{[\ell-1]}, & \text{for sum aggregation with self loops.} \end{cases} \tag{8.91}$$

The update step is where a neural network approach is used. In this step, the hidden state of the node, $h^{[\ell-1]}_{(i)}$, and the message $m^{[\ell]}_{(i)}$ are used to determine the hidden state of the node in the output of the ℓ-th layer. The update function is typically composed of an affine transformation together with a non-linear activation. A simple typical form of $f^{[\ell]}_{\text{update},\theta^{[\ell]}}(\cdot)$ is,

$$h^{[\ell]}_{(i)} = S\left(m^{[\ell]}_{(i)} W^{[\ell]}_{\text{m}} + h^{[\ell-1]}_{(i)} W^{[\ell]}_{\text{u}} + b^{[\ell]}\right), \tag{8.92}$$

where we treat the vectors as row vectors. Here $S(\cdot)$ is a vector activation function typically formed of scalar activation functions such as $\sigma_{\text{Sig}}(\cdot)$, similarly to other neural networks. With this form, the learned parameters $\theta^{[\ell]} = (W^{[\ell]}_{\text{m}}, W^{[\ell]}_{\text{u}}, b^{[\ell]})$ include weight matrices and a bias vector, and under our (simplifying) dimensionality assumptions, we have $W^{[\ell]}_{\text{m}}, W^{[\ell]}_{\text{u}} \in \mathbb{R}^{p_V \times p_V}$, and $b^{[\ell]} \in \mathbb{R}^{p_V}$ (considered as a row vector). Note that more generally, dimensions may vary across layers and hence the matrices may be non-square.

If we focus on sum aggregation without self loops, then (8.92) becomes,

$$h^{[\ell]}_{(i)} = S\left(\sum_{j:v_j \in \mathcal{N}(v_i)} \left(h^{[\ell-1]}_{(j)} W^{[\ell]}_{\text{m}}\right) + h^{[\ell-1]}_{(i)} W^{[\ell]}_{\text{u}} + b^{[\ell]}\right). \tag{8.93}$$

Further, in this case, if we consider the whole graph, we can use (8.91) to represent (8.93) via a network wide equation,

$$h^{[\ell]} = S\left(Ah^{[\ell-1]}W_{\mathrm{m}}^{[\ell]} + h^{[\ell-1]}W_{\mathrm{u}}^{[\ell]} + B^{[\ell]}\right), \tag{8.94}$$

where in similar nature to (5.10) of Chapter 5, $B^{[\ell]}$ is a $r \times p_V$ matrix with each row equal to the bias vector $b^{[\ell]}$. Note that here $S(\cdot)$ is taken as the activation function over the whole matrix, typically element wise.

It is also of interest to note that in case where we restrict the parameters with $W_{\mathrm{m}}^{[\ell]} = W_{\mathrm{u}}^{[\ell]}$, both denoted as $W^{[\ell]}$, then (8.94) is reduced to

$$h^{[\ell]} = S\left((A+I)h^{[\ell-1]}W^{[\ell]} + B^{[\ell]}\right). \tag{8.95}$$

This case of $W^{[\ell]} = W_{\mathrm{m}}^{[\ell]} = W_{\mathrm{u}}^{[\ell]}$ yields a similar update rule to what we would have if we consider sum aggregation with self loops; see (8.91).

In practice, one can make a choice if to use the formulation with more parameters (8.94) or the less parameterized formulation (8.95). With the latter, there can be some restriction on the expressivity of the graph neural network as there is no separation of the information from the node and from its neighbours. Nevertheless in some cases, the less parameterized case, (8.95) suffices.

Model Variants

We close this section with a few model variants of graph neural networks. Each of the variants has its advantages and disadvantages, and the applicability of the variants for applications is beyond our scope. Our purpose here is simply to explore the various basic ideas and equations associated with each of the models. We consider graph convolutional networks, spectral approaches, and the use of the attention mechanism.

In a *graph convolutional network*, (8.95) is modified to,

$$h^{[\ell]} = S\left(\tilde{D}^{-\frac{1}{2}}(A+I)\tilde{D}^{-\frac{1}{2}}h^{[\ell-1]}W^{[\ell]} + B^{[\ell]}\right), \tag{8.96}$$

where the matrix $\tilde{D}$ is a diagonal matrix where $\tilde{D}_{ii}$ is the degree of node v_i plus one. Thus $\tilde{D}^{-\frac{1}{2}}$ is a diagonal matrix with entries that are inverse of the square root of the degree plus one. At the node level, again representing vectors as row vectors, we can unpack (8.96) for node v_i as,

$$h_{(i)}^{[\ell]} = S\left(\sum_{j:v_j\in\mathcal{N}(v_i)}\left(\frac{1}{\sqrt{\tilde{D}_{ii}\tilde{D}_{jj}}}h_{(j)}^{[\ell-1]}W^{[\ell]}\right) + \frac{1}{\tilde{D}_{ii}}h_{(i)}^{[\ell-1]}W^{[\ell]} + b^{[\ell]}\right). \tag{8.97}$$

If we compare (8.97) with (8.93), we see that a graph convolutional network is similar to sum aggregation, yet with weighting in the summation proportional to the degrees. Specifically, when considering the update for the hidden state $h_{(i)}^{[\ell]}$ of node v_i, the weight of the hidden state of neighbouring nodes that have more neighbours than i is reduced, and conversely

the weight for nodes that have less neighbours is increased. This scaling acts as a form of regularization in the network.

Graph convolutional networks can be extended with a *spectral* approach. Specifically let us now outline key ideas of *spectral graph neural networks*, also known as *spectral convolutional graph neural networks*. Generally, in the world of signal processing and mathematics, a spectral approach deals with analyzing a transform of a signal in place of the signal itself. For example in the context of time signals, as briefly discussed in Section 6.2, instead of the signal, one may sometimes consider the *Fourier transform* of the signal. Then based on the so-called *convolution theorem*, the Fourier transform of the convolution between two signals, as for example in equation (6.2) of Chapter 6, can be represented as the product of the Fourier transforms of the individual signals. [19]

In the context of graphs, an analogy to the convolution theorem can be considered using eigenvalue decompositions of matrices, which is the topic of study of an area called *spectral graph theory*. Let us focus on undirected graphs that are connected, and thus the degree of each node is at least one.

In spectral graph theory, an important matrix associated with a graph with r nodes is the $r \times r$ *Laplacian matrix*, appearing in either the unnormalized form, or the normalized form, and defined as,

$$\mathcal{L} = \begin{cases} D - A & \text{(unnormalized)}, \\ I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} & \text{(normalized)}. \end{cases}$$

Here, similarly to $\tilde{D}$ from above, D a diagonal matrix where this time D_{ii} is the degree of node v_i.

One may consider a vector $u^{[\ell]} \in \mathbb{R}^r$ which represents some state values for each node in the graph, and the operation

$$u^{[\ell+1]} = \mathcal{L} u^{[\ell]}, \tag{8.98}$$

can encompass the effect of applying the Laplacian matrix of the graph (either unnormalized or normalized) on the state $u^{[\ell]}$ to achieve the next state $u^{[\ell+1]}$. Interpretations of this operation for specific graph contexts are beyond our scope, yet we mention that in the context of random walks on graphs, electrical networks (Kirchhoff laws), and other contexts, this operation is common.

A key idea in spectral graph networks is to modify the operation (8.98) to,

$$u^{[\ell+1]} = \tilde{\mathcal{L}} u^{[\ell]}, \tag{8.99}$$

where the modification from $\mathcal{L}$ to $\tilde{\mathcal{L}}$ is the essence of the learned parameters in the model and is further explained below. With this modification it is useful to use the *spectral decomposition* of $\mathcal{L}$ and learn how to modify the eigenvalues of $\mathcal{L}$ effectively. This is analogous to learning how to modify filters, as is done in Chapter 6, yet unlike Chapter 6, learning is on the spectral domain (eigenvalues) and not on the time domain (convolutions).

Now looking at the spectral decomposition, since the graph is undirected, $\mathcal{L}$, either in the unnormalized or normalized form, is a symmetric matrix, and further it can be shown to be

[19] Note that in Chapter 6 we actually do not discuss Fourier transforms, yet we point the reader there for general context of signals and systems.

positive semidefinite. The spectral decomposition of $\mathcal{L}$ is,

$$\mathcal{L} = U \Lambda U^\top, \tag{8.100}$$

where $U \in \mathbb{R}^{r \times r}$ has normalized eigenvectors of $\mathcal{L}$ as columns, and $\Lambda \in \mathbb{R}^{r \times r}$ is a diagonal matrix with corresponding eigenvalues, each of which is real and non-negative (due to being positive semidefinite). With this spectral decomposition, U is an orthogonal matrix, namely $U^\top U = I$ (the inverse of U is its transpose). Note that it is also customary to order the eigenvalues in the diagonal of Λ in descending order (and the associated eigenvectors in the columns of U are obviously ordered accordingly).

Now when we consider (8.98) and wish to modify it to (8.99), we do so by transforming the eigenvalues of $\mathcal{L}$, appearing in the diagonal of Λ. For example with so called *high pass filtering* we retain the high valued eigenvalues (above some threshold), while shrinking or zeroing out the other eigenvalues. Similarly, *low pass filtering* works in the other direction, retaining only low eigenvalues. In each such case, we may view the transformation of the eigenvalues as some function $F(\cdot)$ which applied to the diagonal matrix Λ is denoted as $F(\Lambda) \in \mathbb{R}^{r \times r}$. With this we have the modified Laplacian matrix as $\tilde{\mathcal{L}} = U F(\Lambda) U^\top$, and thus (8.99) is represented as,

$$u^{[\ell+1]} = U F(\Lambda) U^\top u^{[\ell]}. \tag{8.101}$$

Note that we may view (8.101) as first projecting $u^{[\ell]}$ onto the orthogonal eigenvector space via the transformation $U^\top u^{[\ell]}$, then applying individual (adjusted via $F(\cdot)$) eigenvalues on each coordinate of the basis via the left multiplication by the diagonal matrix $F(\Lambda)$, and finally, transforming back to the original basis via another left multiplication by U.

Having understood how the spectral decomposition can be used, we now return to the general setup of graph neural networks where as before $h^{[\ell]} \in \mathbb{R}^{r \times p_V}$ is the hidden state matrix which is updated from layer ℓ to layer $\ell + 1$. Now also denote $h^{[\ell]}_{(|i)}$ as a vector in $\mathbb{R}^r$ which is the i-th column of this matrix. This vector represents the hidden state information for each node in the network, based on the i-th hidden state feature at layer ℓ. With this notation, the update for this hidden state vector per feature i depends on all features in the previous layer and is,

$$h^{[\ell+1]}_{(|i)} = S\Big(\big(\sum_{k=1}^{p_V} U \, F^{[\ell]}_{(k,i)} \, U^\top h^{[\ell]}_{(|k)} \big) + b^{[\ell]}_{(i)} \Big), \qquad \text{for} \qquad i = 1, \ldots, p_V. \tag{8.102}$$

Here for each pair of features k and i, $F^{[\ell]}_{(k,i)}$ is an $r \times r$ diagonal matrix of learned parameters that we call *spectral weights*, and $b^{[\ell]}_{(i)}$ is a bias vector in $\mathbb{R}^r$. As always, $S(\cdot)$ is a vector activation function. For one such layer ℓ, we can observe that the total number of learned parameters for spectral weights is $r \times (p_V)^2$ and the total number of learned parameters for the bias vectors is $r \times p_V$.

To get a better feel for the update equation (8.102), compare it with (8.101). In (8.101) we see that $F(\Lambda)$ is a filter applied to the eigenvalues whereas in (8.102) we represent a learned $F(\Lambda)$ via $F^{[\ell]}_{(k,i)}$. Similarly to the concept of channels in convolutional neural networks of Chapter 6, the summation over all updates $U \, F^{[\ell]}_{(k,i)} \, U^\top h^{[\ell]}_{(|k)}$ integrates all the features from layer ℓ into the associated feature of layer $\ell + 1$.

The story of spectral graph neural networks does not stop here because ideally one would like to reduce the dimension of the learned parameters $F_{(k,i)}^{[\ell]}$ so that they do not depend on the whole graph and are independent of the number of nodes in the graph r. The study and application of spectral graph neural networks deals with such approaches and indeed in certain cases it has been shown that spectral graph neural networks generalize well and sometimes perform better than their convolutional graph neural networks counter parts. The details are beyond our scope.

A different variant is *graph attention networks* where the aggregation step uses an attention mechanism. Concepts of attention were heavily discussed in Chapter 7 in the context of sequence models. Specifically, in Section 7.4 we introduced the general concept of attention where attention weights are calculated as in (7.20) and are then used for linear combinations of inputs in (7.21). We then also used attention in the context of transformers as in Section 7.5.

In the graph context, attention can be incorporated via the aggregation step. Specifically, we can enhance the basic sum aggregation as in (8.90), focusing here only on the case without self loops, to be,

$$m_{(i)}^{[\ell]} = \sum_{j\,:\,v_j \in \mathcal{N}(v_i)} \alpha_{(i),j}^{[\ell]}\; h_{(j)}^{[\ell-1]}. \tag{8.103}$$

Here for node v_i, and each neigbour $v_j \in \mathcal{N}(v_i)$, we have attention weight $\alpha_{(i),j}^{[\ell]}$ where

$$\sum_{j\,:\,v_j \in \mathcal{N}(v_i)} \alpha_{(i),j}^{[\ell]} = 1.$$

Similarly to Chapter 7, attention weights are calculated using a scoring mechanism via an alignment function $s : \mathbb{R}^{p_V} \times \mathbb{R}^{p_V} \to \mathbb{R}$, where $s(h_{(i_1)}^{[\ell-1]}, h_{(i_2)}^{[\ell-1]})$ measures the proximity between the hidden states of nodes v_{i_1} and v_{i_2} at layer $\ell - 1$. The basic alignment function is the inner product between $h_{(i_1)}^{[\ell-1]}$ and $h_{(i_2)}^{[\ell-1]}$, yet more complex options with learned parameters are also possible. One option with linear re-weighting is,

$$s(h_{(i_1)}^{[\ell-1]}, h_{(i_2)}^{[\ell-1]}) = \big(h_{(i_1)}^{[\ell-1]} W_a^{[\ell]}\big)\big(h_{(i_2)}^{[\ell-1]} W_a^{[\ell]}\big)^\top \tag{8.104}$$

where $W_a^{[\ell]} \in \mathbb{R}^{p_V \times m'}$ is a matrix of learned parameters for layer ℓ with m' set as some dimension (recall here that our hidden state vectors are taken as rows).

Based on the alignment function, the attention weights are calculated for each node v_i. As with all attention mechanisms, we use a softmax function over neighbours indices j to obtain the attention weights,

$$\alpha_{(i),j}^{[\ell]} = \frac{e^{s(h_{(i)}^{[\ell-1]}, h_{(j)}^{[\ell-1]})}}{\sum_{k\,:\,v_k \in \mathcal{N}(v_i)} e^{s(h_{(i)}^{[\ell-1]}, h_{(k)}^{[\ell-1]})}},$$

which are then used in the aggregation step (8.103). Once the message is computed, it can be used in an update function as in (8.92).

Note that with our description of graph attention networks here, the learned parameters per layer are $W_a^{[\ell]}$ for the alignment function (8.104) as well as $\theta^{[\ell]} = (W_{\mathrm{m}}^{[\ell]}, W_{\mathrm{u}}^{[\ell]}, b^{[\ell]})$ used in the update (8.92). However, other options, typically with reduced parameters, reusing weight matrices either across layers, or between the alignment function and the update equation,

are also possible. Also, similar to the transformer architecture, multi-head attention has also been introduced and in certain graph neural network applications, such architectures are very popular.

Notes and References

This chapter covered a broad range of specialized architectures and paradigms where each section covers a major topic which could have in fact made a whole chapter. Hence in our notes and references about the topics of this chapter we only summarize key references and developments in each of the sub-fields. A further recent overarching text that we recommend is [336] with multiple chapters, one per each of the topics covered here.

The field of generative modelling has multiple origins. Early models include *hidden Markov models* and *Gaussian mixture models* with origins in the 1950's and 1960's; see chapters 11 and 17 of [298] for background. Somewhat more recently, some authors consider the study of *Boltzman machine* models introduced in the 1980's in [2], and deep Boltzman machines in [361], as the initial meaningful generative models in the context of deep learning. See also chapter 20 of [142] for an overview. A more recent survey of generative models in machine learning is [162] and a comparison of deep generative modelling approaches is in [46].

Up to 2014, while generative models were useful for some applications and certainly interesting, in terms of images, they lacked the ability to create real life looking data. The big advance came with the development of *generative adversarial networks* (GANs), in Goodfellow et al.'s work [143]. This opened up possibilities for creation of realistic looking images (and data) and is still a very active topic. *Variational autoencoders*, initially introduced in [234], grew into multiple directions and contemporary *diffusion models* such as [183], and those surveyed in [430], constitute the state of the art in image generative modelling. As of the time of publishing of this book, diffusion models and GANs still compete, with diffusion models generally able to produce more impressive images, while GANs are much faster in production since they do not require multiple neural networks.

Ideas of variational autoencoders are rooted in modern developments of *Bayesian statistics*. See [92] for an introductory general text on Bayesian statistics and [407] for an accessible review of the area. Specifically, the *variational Bayes* methods, a well-known optimization-based approach in the field of *approximate Bayesian computation*, captures the key ideas used in variational autoencoders. See [41] and [444] for reviews of variational Bayes. This approach also falls in the realm of *approximate Bayesian computation* and entails a method for approximating posterior distributions using simpler surrogate distributions. See [381] for a collection of approximate Bayesian computation methods. Specifically, for more details about variational autoencoders, see [235].

Our presentation of variational autoencoders was geared towards *hierarchical Markovian variational autoencoders* of which diffusion models are a special case. Nevertheless, variational autoencoders and their variants are interesting and useful in their own right. They have been applied to many fields. In image processing, prediction of the trajectory of pixels of an image is tackled in [414] and natural image modelling is in [152]. In the field of speech analysis, voice conversion is handled in [191] and speech synthesis in [8]. In the area of text processing as in [54], reccurent neural network based variational autoencoders for generating sentences are put forward and in [193], controlled text generation is handled. Another field is graph-based data analysis as in [236] where learning on graph-structured data is handled, and [210] which deals with molecular graph generation. As we presented diffusion models as special cases of *hierarchical variational autoencoders*, the literature on these models is also relevant. In particular, see [343] for an application in black box variational inference and [385] for a variant called ladder variational autoencoder.

Diffusion models, initially introduced in [384], gained significant prominence following [183], which showcased exceptional image synthesis results. These models were further improved with [104], where for the first time the prolonged dominance of GANs was broken. For recent surveys of diffusion models, refer to [71], [96], and [430]. In terms of applications in the realm of image processing, diffusion models are utilized for tasks such as *colorization*, *inpainting*, *uncropping*, and *restoration* as in [358]. Other image processing applications include super-resolution as in [360], and image editing as in [176]. There is an extensive study on applications of diffusion models for *text to image generation* such as for example the work in [359] which introduced *Imagen*. This system utilizes a transformer based large language model which is used for understanding text combined with a diffusion model used for image generation. The application of diffusion models extends to video data as well. Notable contributions include [163] where an approach for long-duration video completions is put forward, and [182] which introduced *Imagen Video*, a text-conditional video generation system based on a cascade of video diffusion models.

A related paradigm to variational autoencoders and diffusion models that we did not cover is *normalizing flows*. This paradigm was first introduced in [347] for representing the posterior in variational autoencoders. These generative models construct complex distributions by transforming a distribution through a series of invertible mappings. See chapter 16 of [336] for an overview and [322] for a recent survey.

As already mentioned, GANs were introduced in [143]. After the introduction of this paradigm, multiple generalizations appeared. Particular early variants included *C-GAN* [293], and convolutional versions of GANs as in [339]. The idea of *NS-GAN* was already introduced in [143]. The *W-GAN* concept first appeared in [15] and was later developed in [151] where the gradient penalty approach was introduced. See also [3] as well as the empirical comparison in [271] and the general GAN surveys [150] and [200]. The ideas of *AC-GAN* were developed in [316] and the ideas of *Info-GAN* were developed in [78]. The image to image paradigm in GANs is broad. A survey on this topic is [11] with initial ideas in [83], [204], and [417]. *Style-GAN* was developed in [224], and further developments are in [225] and [223]. Finally we mention a few general developments in GANs including *sequence GAN* from [437], and *EditGAN* from [264].

Modern approaches to deep reinforcement learning are surveyed extensively in the texts [125] and [393], whereas more dated accounts are [36] and [217]. At the more fundamental level, basics of Markov decision processes are presented in [33], [34], and [337]. An expository overview of engineering control theory is in [9]. The criterion for convergence of *Q-learning* as in our equation (8.81) is from [419]. The success of reinforcement learning in playing Atari video games is in [294], and later landmark results in the game of Go are documented in [377]. Apart from games, reinforcement learning is successfully used in several domains one of which is addressing hard combinatorial problems; see [282] for a survey of such approaches. Nowadays, reinforcement learning is applied for fine tuning large language models through human feedback as documented in [318] and [429]. Reinforcement learning is also critical for some robotic tasks as described in the survey [448]. However, as one would have thought initially that self driving cars are best handled as a system via reinforcement learning, in practice other techniques have so far prevailed; see [79] for a general survey. A related approach often used is *imitation learning*, see [286] for a survey of this technique in the context of autonomous vehicles.

General texts about graph neural networks are [273], and [157], with recent survey papers in [413] and [427] as well as a notable chapter in [336]. Early ideas in graph neural networks arose in [387] where a concept called at the time the *generalized recursive neuron* was introduced. Further ideas of graph neural networks were developed in [144] and [365]. These days graph neural networks are used in multiple applications such as those discussed in [242], [445], and [451] among many others. There are various techniques for graph embedding including *DeepWalk* [329], *node2vec* [148], *GraphSAGE* [156], *LINE* (Large-scale Information Network Embedding) [397], and *HOPE* (High-order Proximity preserved Embedding), [317]. For an introductory computer science overview of traditional graph algorithms and data structures see [93].

General ideas of *message passing schemes* in graph neural networks were introduced in [287]. Ideas of *graph convolutional networks* were developed in [18], [134], and [309]. Ideas of *spectral convolutional graph neural networks* were developed in [67] and [174]. *Graph attention networks* were developed in [411], motivated by the effective application of the attention mechanism to sequence models as in [410]. Indeed many developments in deep learning cross architectures, domains, and sub-disciplines, and the development of graph attention networks serve as one such example. We also close by mentioning *spatial-temporal graph neural networks* which are designed to deal with *dynamic graphs*, sometimes also with a spatial component. These models were introduced in [257] and [371], and are further described in the recent review [209].