Machine Perception

Cristian Perez Jensen

November 15, 2024

Note that these are not the official lecture notes of the course, but only notes written by a student of the course. As such, there might be mistakes. The source code can be found at `github.com/cristianpjensen/eth-cs-notes`. If you find a mistake, please create an issue or open a pull request.

Contents

List of symbols

$\doteq$	Equality by definition
$\approx$	Approximate equality
$\propto$	Proportional to
$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$i:j$	Set of natural numbers between i and j. *I.e.*, $\{i, i+1, \ldots, j\}$
$\mathbb{1}\{\text{predicate}\}$	Indicator function (1 if predicate is true, otherwise 0)
$\boldsymbol{v} \in \mathbb{R}^n$	n-dimensional vector
$\boldsymbol{M} \in \mathbb{R}^{m \times n}$	$m \times n$ matrix
$\mathbf{T} \in \mathbb{R}^{d_1 \times \cdots \times d_n}$	Tensor
$\boldsymbol{M}^\top$	Transpose of matrix $\boldsymbol{M}$
$\boldsymbol{M}^{-1}$	Inverse of matrix $\boldsymbol{M}$
$\det(\boldsymbol{M})$	Determinant of $\boldsymbol{M}$
$\frac{\mathrm{d}}{\mathrm{d}x} f(x)$	Ordinary derivative of $f(x)$ w.r.t. x at point $x \in \mathbb{R}$
$\frac{\partial}{\partial x} f(\boldsymbol{x})$	Partial derivative of $f(\boldsymbol{x})$ w.r.t. x at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^n$	Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{J}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^{n \times m}$	Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^m$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}^2_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^{n \times n}$	Hessian of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\theta} \in \Theta$	Parametrization of a model, where Θ is a compact subset of $\mathbb{R}^K$
$\mathcal{X}$	Input space
$\mathcal{Y}$	Output space
$\mathcal{D} \subseteq \mathcal{X} \times \mathcal{Y}$	Labeled training data

1 Neural networks

1.1 Multi-layer perceptron

The original perceptron [Rosenblatt, 1958] was a single layer perceptron with the following non-linearity,

$$\sigma(x) \doteq \mathbb{1}\{x > 0\}.$$

The classification of a single point can then be written as

$$\hat{y} = \mathbb{1}\{\boldsymbol{w}^\top \boldsymbol{x} + b > 0\}.$$

The learning algorithm then iteratively updates the weights for a data point that was classified incorrectly,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta \underbrace{(y_i - \hat{y}_i)}_{\text{residual}} \boldsymbol{x}_i,$$

where η is the learning rate. This is essentially gradient descent with a Hinge loss, where if $y < \hat{y}$, then we decrease the weights, while if $y > \hat{y}$, we increase the weights. If the data is linearly separable, the perceptron converges in finite time.

The problem with the single-layer perceptron was that it could not solve the XOR problem; see Figure 1.2. This can be solved by introducing hidden layers,

$$\hat{y} = \sigma(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \cdots \sigma(\boldsymbol{W}_1 \boldsymbol{x}))).$$

We call this architecture a multi-layer perceptron (MLP); see Figure 1.3. We then want to estimate the parameters $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k, \boldsymbol{b}_1, \ldots, \boldsymbol{b}_k\}$, using an optimization algorithm such as gradient descent, which we call "learning". We can compute the gradient by backpropagation, which we will see in a later section.

1.2 Loss functions

We need an objective to optimize for. We typically call this objective function the loss function, which we minimize. In classification, we typically optimize the *maximum likelihood estimate* (MLE),

$$\begin{aligned}
\operatorname*{argmax}_{\boldsymbol{\theta}} p(\mathcal{D} \mid \boldsymbol{\theta}) &\overset{\text{iid}}{=} \operatorname*{argmax}_{\boldsymbol{\theta}} \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) && \text{log is monotonic.} \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} -\sum_{i=1}^{n} \log p(y_i \mid \boldsymbol{\theta}).
\end{aligned}$$

If the model predicts the parameters of a Bernoulli distribution,[1] MLE is

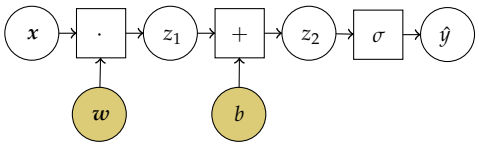

Figure 1.1. Computation graph of a perceptron [Rosenblatt, 1958], where $\sigma(x) = \mathbb{1}\{x > 0\}$.

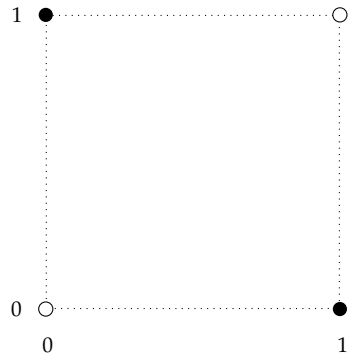

Figure 1.2. XOR problem. As can be seen, the data is not linearly separable, and thus not solvable by the perceptron.

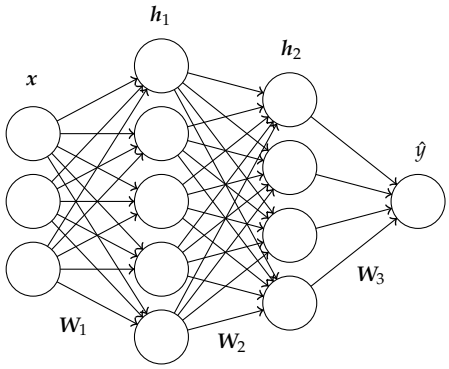

Figure 1.3. Example multi-layer perceptron architecture.

[1] *I.e.*, $y \in \{0, 1\}$, but it predicts the probability of $y = 1$ with $\hat{y} \in [0, 1]$.

equivalent to binary cross-entropy,

$$\begin{aligned}\mathcal{L}(\boldsymbol{\theta}) &= -\sum_{i=1}^{n} \log \text{Ber}(y_i \mid \hat{y}_i \doteq f(\boldsymbol{x}_i \mid \boldsymbol{\theta})) \\ &= -\sum_{i=1}^{n} \log \hat{y}_i^{y_i}(1-\hat{y}_i)^{1-y_i} \\ &= -\sum_{i=1}^{n} y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i).\end{aligned}$$

This loss is minimized if $y_i = \hat{y}_i$ for all $i \in [n]$; see Figure 1.4. We can extend this to multi-class classification by using the softmax and a categorical distribution.

If we the choose the model to be Gaussian, we end up minimizing the mean-squared error,

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \|\boldsymbol{y}_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\|_2^2.$$

Furthermore, the Laplacian distribution yields minimizing the ℓ_1 norm,

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \|\boldsymbol{y}_i - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)\|_1.$$

If we have prior information about the weights $p(\boldsymbol{\theta})$, we could also optimize for the *maximum a posteriori* (MAP),

$$\begin{aligned}\operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta} \mid \mathcal{D}) &= \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) p(\mathcal{D} \mid \boldsymbol{\theta}) \\ &\overset{\text{iid}}{=} \operatorname*{argmax}_{\boldsymbol{\theta}} p(\boldsymbol{\theta}) \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) \\ &= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log\left(p(\boldsymbol{\theta}) \prod_{i=1}^{n} p(y_i \mid \boldsymbol{\theta})\right) \\ &= \operatorname*{argmin}_{\boldsymbol{\theta}} -\log p(\boldsymbol{\theta}) - \sum_{i=1}^{n} \log p(y_i \mid \boldsymbol{\theta})\end{aligned}$$

Note that MAP and MLE are equivalent if $p(\boldsymbol{\theta})$ is uniform over the domain of weights. Assuming a Gaussian prior distribution over $\boldsymbol{\theta}$, MAP yields Ridge regression,

$$\mathcal{L}'(\boldsymbol{\theta}) = \mathcal{L}(\boldsymbol{\theta}) + \lambda \|\boldsymbol{\theta}\|_2^2.$$

1.3 Backpropagation

Typically, we cannot find the optimal parameters $\boldsymbol{\theta}^\star$ in closed form, so we must use an optimization algorithm. Optimization algorithms, such as gradient descent, typically require computing the gradient w.r.t. the parameters. Backpropagation is an algorithm for computing the gradient

f is a model that outputs the Bernoulli parameter. Note that this parameter must be in $[0, 1]$, thus we use a sigmoid non-linearity,

$$\sigma(x) = \frac{1}{1+\exp(-x)}.$$

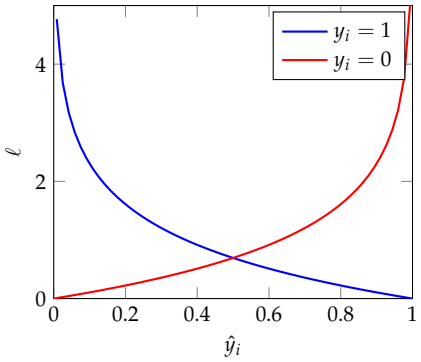

Figure 1.4. Loss function for if $y_i = 0$ and $y_i = 1$ in binary cross entropy.

of any function, given that we have access to the derivatives of the primitive functions it consists of.[2] It then computes the gradient by making use of dynamic programming, the chain rule, and sum rule.

[2] For example, to compute the gradient of $f(\boldsymbol{x}, \boldsymbol{y}) = \sigma(\boldsymbol{x}^\top \boldsymbol{y})$, we would need access to $\frac{\mathrm{d}}{\mathrm{d}x}\sigma(x)$, $\frac{\partial}{\partial \boldsymbol{x}}\boldsymbol{x}^\top \boldsymbol{y}$, and $\frac{\partial}{\partial \boldsymbol{y}}\boldsymbol{x}^\top \boldsymbol{y}$.

Let $\hat{y}_i < y_i$, then we want to increase $\hat{y}_i$ to match y_i. Furthermore, consider the following loss function,

$$\ell = \frac{1}{2}(y - \hat{y})^2, \quad \hat{y} = \sigma(\boldsymbol{w}^\top \boldsymbol{h}),$$

and the MLP architecture, then intuitively, we can do two things to increase $\hat{y}_i$:

1. We could increase the weight connected to a node with a high activation value in the previous hidden layer. Typically, we optimize by moving the weights in the direction of the gradient,

$$\frac{\partial \mathcal{L}}{\partial w_k} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{w}^\top \boldsymbol{h}} \frac{\partial \boldsymbol{w}^\top \boldsymbol{h}}{\partial w_k} = (y - \hat{y})\sigma'\left(\boldsymbol{w}^\top \boldsymbol{h}\right) h_k.$$

   This matches our intuition, because the amount that we increase w_k by is proportional to the activation connected to that weight h_k;

2. Or, we could increase the activation that is connected to a strong weight in the previous hidden layer. Again, we move the weights in the direction of the gradient,

$$\frac{\partial \mathcal{L}}{\partial h_k} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \boldsymbol{w}^\top \boldsymbol{h}} \frac{\partial \boldsymbol{w}^\top \boldsymbol{h}}{\partial h_k} = (y - \hat{y})\sigma'\left(\boldsymbol{w}^\top \boldsymbol{h}\right) w_k.$$

   This also matches our intuition, since the amount that we want to increase h_k by is proportional to w_k. We cannot increase h_k directly, but we can update the weights connected to h_k, which brings us back to the first case. This update will be proportional to $\frac{\partial \mathcal{L}}{\partial h_k}$ by the chain rule. In this way, we can recursively update all the weights using gradient information.

Thus, using first-order methods that iteratively move the weights in the direction of the gradient should work well. Gradient descent iteratively updates the parameters by the following,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \mathcal{L}(\hat{\boldsymbol{y}}, \boldsymbol{y}),$$

until the gradient is small.

1.4 Activation functions

In MLPs, the activation function should be non-linear, or the resulting MLP is just an affine mapping with extra steps. This is because the product of affine mappings are themselves affine mappings.

1.5 Universal approximation theorem

Theorem 1.1 (Universal approximation theorem [Hornik et al., 1989]). Let $\sigma : \mathbb{R} \to \mathbb{R}$ be a non-constant, bounded, and continuous activation function. Let I_m denote the m-dimensional unit hypercube $[0,1]^m$ and the space of real-valued function on I_m is denoted by $\mathcal{C}(I_m)$.

Let $g \in \mathcal{C}(I_m)$ be any function in the hypercube. Let $\epsilon > 0, n \in \mathbb{N}, v_i, b_i \in \mathbb{R}, \boldsymbol{w}_i \in \mathbb{R}^m$ for $i \in [n]$, then

$$g(\boldsymbol{x}) \approx f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sum_{i=1}^{n} v_i \sigma(\boldsymbol{w}_i^\top \boldsymbol{x} + b_i),$$

where $|f_{\boldsymbol{\theta}}(\boldsymbol{x}) - g(\boldsymbol{x})| < \epsilon$ for all $\boldsymbol{x} \in I_m$.

In words, this means that any continuous function can be approximated by a single hidden layer MLP with a continuous non-linear activation function with arbitrary precision. The universal approximation theorem holds for any single hidden layer network. However, this hidden layer may need to have infinite width to approximate f. In practice, deeper networks work better than wider networks.

2 Convolutional neural networks

When dealing with high-dimensional data, such as images, it is not practical to work with MLPs, because the amount of parameters would explode in size.[3] By making use of the locality of images, we can drastically decrease the number of parameters.

[3] Mapping a $256 \times 256 \times 3$ input image to a 1-dimensional output would already require nearly 2 million parameters.

2.1 Convolution

The *correlation* operator takes a filter $\mathbf{K}$, moves it along the entire image, and outputs the patch-wise multiplication, for each patch of the same size as the filter. It is defined as follows,

$$(\mathbf{K} \star \mathbf{I})[i,j] = \sum_{m=-k}^{k} \sum_{n=-k}^{k} \mathbf{K}[m,n]\mathbf{I}[i+m, j+n].$$

The *convolution* operator is very similar. The only difference is that the kernel is mirrored in a convolution,

$$(\mathbf{K} * \mathbf{I})[i,j] = \sum_{m=-k}^{k} \sum_{n=-k}^{k} \mathbf{K}[m,n]\mathbf{I}[i-m, j-n].$$

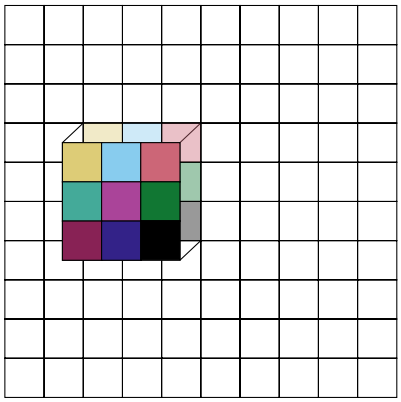

Figure 2.1. Illustration of applying a correlation to a pixel.

Theoretically, the convolution operator is more useful, because it is commutative.[4] In practice with neural networks, it does not matter, since the weights will just be updated to be the same, except that they are mirrored. Thus, we will only be referring to the convolution from now on.

[4] $\mathbf{I} * \mathbf{K} = \mathbf{K} * \mathbf{I}$, but $\mathbf{I} \star \mathbf{K} \neq \mathbf{K} \star \mathbf{I}$.

A convolution operator C is a linear, shift-equivariant transformation, *i.e.*,

$$C(\alpha \boldsymbol{x} + \beta) = \alpha C(\boldsymbol{x}) + \beta$$

Linearity.

$$T_{\boldsymbol{t}}(C(\boldsymbol{x})) = C(T_{\boldsymbol{t}}(\boldsymbol{x})).$$

Translation equivariant.

Since convolutions are linear, discrete convolutions can be implemented using matrix multiplication,

$$\mathbf{K} * \mathbf{I} = \begin{bmatrix} k_1 & 0 & 0 & \cdots & 0 \\ k_2 & k_1 & 0 & \cdots & 0 \\ k_3 & k_2 & k_1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & k_m \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ \vdots \\ I_n \end{bmatrix}.$$

2.2 Convolutional neural network

A *convolutional neural network* (CNN) [Krizhevsky et al., 2012] is composed of a sequence of *convolutional layers* and *pooling layers*, followed by a final *dense layer* (MLP).

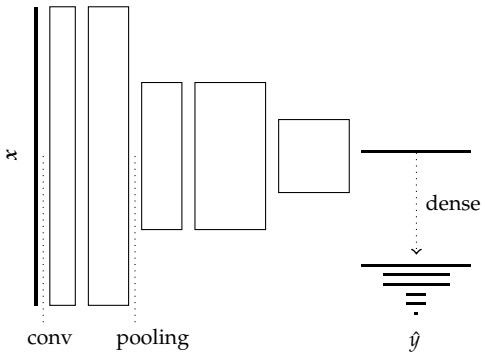

Figure 2.2. Example schematic of a CNN architecture.

This architecture is loosely inspired by the brain, which, at a high level, first extracts high-level features and then more and more specific features. Furthermore, the HMAX model of the brain distinguishes between simple and complex cells, which correspond to the linear layer and max-pool layer, respectively.

Convolutional layer. A convolutional layer applies many filters $\mathbf{W} \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}} \times k \times k}$ to an input image $\mathbf{Z} \in \mathbb{R}^{C_{\text{in}} \times H \times W}$ by convolution,

$$\boldsymbol{Z}_j^{(\ell)} = \sum_{k=1}^{C_{\text{in}}} \boldsymbol{W}_{kj}^{(\ell)} * \boldsymbol{Z}_k^{(\ell-1)} + b_j, \quad j \in [C_{\text{out}}].$$

This filter is learned. Hence, we need to derive its derivative. We will focus on the single filter case ($C_{\text{in}} = C_{\text{out}} = 1$) for simplicity, whose forward pass is computed by

$$z^{(\ell)}[i,j] = \sum_{m=-k}^{k} \sum_{n=-k}^{k} w^{(\ell)}[m,n] z^{(\ell-1)}[i-m, j-n] + b.$$

We can express the derivative of the cost function $\mathcal{L}$ w.r.t. the output of the $(\ell-1)$-th layer as the following,

$$\begin{aligned}
\delta^{(\ell-1)}[i,j] &= \frac{\partial \mathcal{L}}{\partial z^{(\ell-1)}[i,j]} \\
&= \sum_{i'} \sum_{j'} \frac{\partial \mathcal{L}}{\partial z^{(\ell)}[i',j']} \frac{\partial z^{(\ell)}[i',j']}{\partial z^{(\ell-1)}[i,j]} \\
&= \sum_{i'} \sum_{j'} \delta^{(\ell)}[i',j'] \frac{\partial}{\partial z^{(\ell-1)}[i,j]} \sum_{m} \sum_{n} w^{(\ell)}[m,n] z^{(\ell-1)}[i'-m, j'-n] + b \\
&= \sum_{i'} \sum_{j'} \delta^{(\ell)}[i',j'] w^{(\ell)}[i'-i, j'-j].
\end{aligned}$$

From this, we can see that we can compute all values of $\delta^{(\ell-1)}$ by a single convolution,

$$\delta^{(\ell-1)} = \delta^{(\ell)} * \text{Flip}\left(\boldsymbol{W}^{(\ell)}\right) = \delta^{(\ell)} \star \boldsymbol{W}^{(\ell)}.$$

Using this value, we can compute the derivative w.r.t. the weights, which we need for the parameter update in algorithms such as gradient descent,

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial w^{(\ell)}[m,n]} &= \sum_{i} \sum_{j} \frac{\partial \mathcal{L}}{\partial z^{(\ell)}[i,j]} \frac{\partial z^{(\ell)}[i,j]}{\partial w^{\ell}[m,n]} \\
&= \sum_{i} \sum_{j} \delta^{(\ell)}[i,j] \frac{\partial}{\partial w^{(\ell)}[m,n]} \sum_{m'} \sum_{n'} w^{(\ell)}[m',n'] z^{(\ell-1)}[i-m', j-n'] + b \\
&= \sum_{i} \sum_{j} \delta^{(\ell)}[i,j] z^{(\ell-1)}[i-m, j-n].
\end{aligned}$$

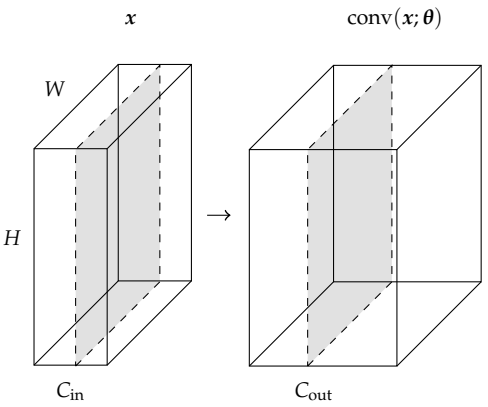

Figure 2.3. Schematic of a convolutional layer. Each input-output channel pair has its own kernel, so $\boldsymbol{\theta}$ has $K \times K \times C_{\text{in}} \times C_{\text{out}}$ parameters.

Again, this has the form of a convolution, thus we can compute all derivatives of $\boldsymbol{W}^{(\ell)}$ by convolution,

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}^{(\ell)}} = \delta^{(\ell)} * \boldsymbol{Z}^{(\ell-1)}.$$

As shown, we can use convolutions for both computing the forward pass, as well as the backward pass. Thus, we first do the forward pass, then compute all $\delta^{(\ell)}$ for all layers ℓ by convolution, and finally we can compute the derivative by convolution as well.

Pooling layers. Pooling layers makes the data more manageable. The most common pooling layer is *max pooling*, which outputs the maximum value for each patch. It can be seen as a non-linear convolutional filter, where it simply outputs the maximum value. Usually, pooling is done with a stride, such that the output becomes exponentially smaller; see Figure 2.4. Because of this, the *receptive field* becomes exponentially larger.

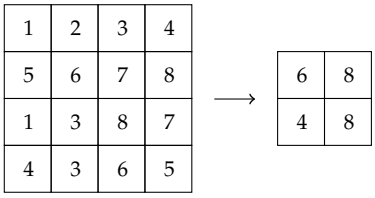

Figure 2.4. Toy example of max pooling.

The forward pass is computed as follows,

$$z^{(\ell)}[i,j] = \max\left\{ z^{(\ell-1)}[i',j'] \;\middle|\; i' \in [si : si+k], j' \in [sj : sj+k] \right\},$$

where s is the stride and k is the kernel size. Let $[i^\star, j^\star]$ be the indices which corresponded to the maximum value in the forward pass, then we can compute the error propagation in the backward pass by

$$\frac{\partial z^{(\ell)}[i,j]}{\partial z^{(\ell-1)}[i',j']} = \mathbb{1}\{[i',j'] = [i^\star, j^\star]\}.$$

Note that the max pooling layer has no learnable parameters. Hence, the backward pass is only a propagation of the error, and not used for a weight update.

Dense layer. The dense layer is simply a linear layer that maps the final convolutional layer to the network's output. All previous convolutional and pooling layers can be seen as extracting features from the image, while the final dense layer makes the actual prediction.

3 *Fully convolutional neural network*

Semantic segmentation is a computer vision task that involves assigning a semantic class to each pixel in an image. While in image classification, the model must output a single class for the entire image, semantic segmentation requires classifying a class for each pixel individually.

A naive approach would be to apply a single convolutional layer to an image, and then running a classifier on each individual pixel. However, this method is inefficient, because we have to run the classifier $H \times W$ times. Instead, we use the output of convolutional neural networks. A naive approach of using CNNs would be to simply apply n convolutional layers with no downsampling, and then considering the last output as the predicted segmentation map. However, this method is expensive.

In practice, the most common approach is to downsample the features obtained using convolution and pooling layers and then upsample them again. By downsampling, this method is more computationally efficient, has larger receptive field, and suffers less from "The curse of dimensionality." By upsampling, the model produces an output of the same resolution as the input.

3.1 *Upsampling methods*

Nearest neighbor. Nearest neighbor upsampling copies the same value into all corresponding pixels at a higher resolution; see Figure 3.1.

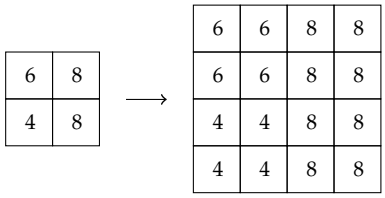

Figure 3.1. Nearest neighbor upsampling.

Bed of nails. Bed of nails upsampling only copies each value once into the output in the top left value, and pads the rest with zero; see Figure 3.2.

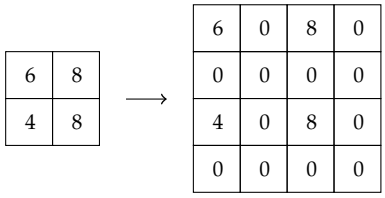

Figure 3.2. Bed of nails upsampling.

Max unpooling. Max unpooling also uses zero padding, like bed of nails. However, it also remembers the original position of the maximum value before the corresponding max pooling in the downsampling phase. This information is then used to place each element back in the correct position; see Figures 2.4 and 3.3.

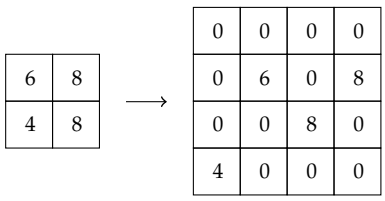

Figure 3.3. Max unpooling the output of Figure 2.4.

Transposed convolutions. Transposed convolution [Shelhamer et al., 2016] is a learned upsampling technique. This layer learns a kernel that is used to produce the terms whose sum will be the final output. Each term is obtained by multiplying all the element of the kernel by the same value of one single input pixel and then inserting the result in the correct position of a matrix of the same size as the output.

3.2 *U-net*

The *U-net* [Ronneberger et al., 2015] is an FCNN architecture, whose main idea is to combine global and local feature maps by copying corresponding tensors from earlier stages in each upsampling stage; see Figure 3.4.

This allows the network to capture both local and global context. In each upsampling, the corresponding output from the downsampling phase is appended to the output. The copied tensor can be seen as the "global" information, while the input of the upsampling layer is the "local" information. Combining these allows for more fine-grained outputs.

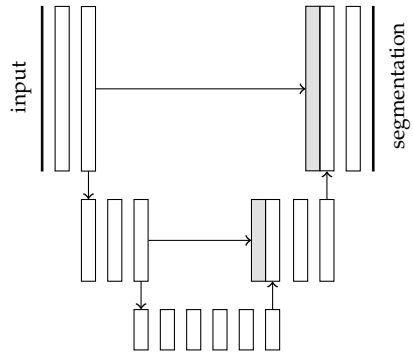

Figure 3.4. U-net architecture. Down arrows are downsampling layers, up arrows are upsampling layers, and right arrows copy.

4 Recurrent neural networks

Recurrent neural networks (RNN) are a type of neural network that processes sequential data, such as text and video. Unlike traditional neural networks, which take fixed-length inputs, RNNs can take inputs of variable length.[5] Generally, RNNs have the following form,

$$\boldsymbol{h}^{(t)} = f_{\boldsymbol{\theta}}\Big(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\Big).$$

In this way, an RNN behaves much like a dynamical system. If we unroll the timesteps, it becomes clear how $\boldsymbol{h}^{(t)}$ depends on $\boldsymbol{x}^{(1:t)}$,

$$\boldsymbol{h}^{(t)} = f_{\boldsymbol{\theta}}\Big(f_{\boldsymbol{\theta}}\Big(\cdots\Big(f_{\boldsymbol{\theta}}\Big(\boldsymbol{h}^{(0)}, \boldsymbol{x}^{(1)}\Big), \cdots\Big), \boldsymbol{x}^{(t-1)}\Big), \boldsymbol{x}^{(t)}\Big).$$

Because of this, we can see $\boldsymbol{h}^{(t)}$ as a representation of $\boldsymbol{x}^{(1:t)}$.

RNNs can have different applications, for example, *one to one*, where at each time step we have one input and one output,[6], *one to many*, where we have one input and we output a sequence of elements,[7] *many to one*, where we have a sequence of inputs and one output,[8] and *many to many*, where we map a sequence to another sequence of a different length.[9]

[5] This is useful for data structures such as text, where the number of words in a text is not fixed.

[6] *E.g.*, part-of-speech tagging.

[7] *E.g.*, image captioning, where the image is the one input, and the caption is a sequence of words.

[8] *E.g.*, sentiment classification, where the input is a text and the output is a single output that determines how positive or negative the text is.

[9] *E.g.*, machine translation, where we map a sentence in one language to a sentence of another.

4.1 Elman RNN

The Elman RNN [Elman, 1990] is characterized by a hidden vector $\boldsymbol{h}^{(t)}$, which forms the state of the network at timestep t. The hidden state is updated at each timestep by combining the previous hidden state with the input,

$$f_{\boldsymbol{\theta}}\Big(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\Big) = \tanh\Big(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)}\Big).$$

We use the tanh activation function, because it is centered at 0.

Since $\boldsymbol{h}^{(t)}$ represents the subsequence $\boldsymbol{x}^{(1:t)}$, we can use it as input to an MLP,

$$\hat{\boldsymbol{y}} = \boldsymbol{W}_y \boldsymbol{h}^{(t)}.$$

Then, we can compute the loss function as the sum of each individual loss function,

$$\mathcal{L} \doteq \sum_{t=1}^{T} \ell^{(t)}.$$

We use *backpropagation through time* (BPTT) to compute the gradient of an RNN. This involves first unrolling the RNN; see Figure 4.1. Then we can compute the gradient by backpropagation on the resulting computational graph,

$$\frac{\partial \ell^{(t)}}{\partial \boldsymbol{W}} = \sum_{k=1}^{t} \frac{\partial \ell^{(t)}}{\partial \hat{\boldsymbol{y}}^{(t)}} \frac{\partial \hat{\boldsymbol{y}}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial^{+} \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}},$$

where $\frac{\partial^{+} \boldsymbol{h}^{(k)}}{\partial \boldsymbol{W}}$ is the immediate derivative that treats $\boldsymbol{h}^{(k-1)}$ as constant w.r.t. $\boldsymbol{W}$.

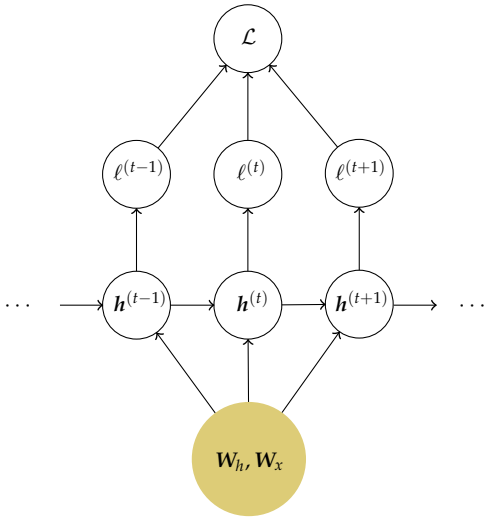

Figure 4.1. The computational graph of an unrolled recurrent neural network. The inputs $\boldsymbol{x}_{1:T}$ and outputs $\boldsymbol{y}_{1:T}$ are omitted.

Let's only consider the following term of the product,

$$\begin{aligned}\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} &= \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}} \\ &= \prod_{i=k+1}^{t} \frac{\partial}{\partial \boldsymbol{h}^{(i-1)}} \sigma\Big(\boldsymbol{W}_h \boldsymbol{h}^{(i-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(i)}\Big) \\ &= \prod_{i=k+1}^{t} \boldsymbol{W}_h^\top \operatorname{diag}\Big(\sigma'\Big(\boldsymbol{W}_h \boldsymbol{h}^{(i-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(i)}\Big)\Big).\end{aligned}$$

Assuming that the norm of the gradient of σ is upper bounded by some $\gamma \in \mathbb{R}$,[10] *i.e.*,

$$\left\|\operatorname{diag}\Big(\sigma'\Big(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)}\Big)\Big)\right\| < \gamma.$$

[10] For example, the gradient of tanh is bounded by 1.

Let λ_1 be the largest eigenvalue of $\boldsymbol{W}_h$, then we have two cases,

1. $\lambda_1 < \frac{1}{\gamma}$. Then we have the following,

$$\left\|\frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}\right\| \leq \|\boldsymbol{W}_h\| \left\|\operatorname{diag}\Big(\sigma'\Big(\boldsymbol{W}_h \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_x \boldsymbol{x}^{(t)}\Big)\Big)\right\| < \frac{1}{\gamma}\gamma = 1.$$

Triangle inequality and $\|\boldsymbol{W}_h\| = \lambda_1$.

   Let $\eta < 1$ be the upper bound of all gradients between $\boldsymbol{h}^{(i)}$ and $\boldsymbol{h}^{(i-1)}$, then by induction, we have

$$\left\|\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}\right\| = \left\|\prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}\right\| < \eta^{t-k}.$$

   This converges to zero, as $t \to \infty$. Thus, we have a *vanishing gradient*;

2. $\lambda_1 > \frac{1}{\gamma}$. Using the same logic as in the other case, this yields

$$\left\|\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}}\right\| = \left\|\prod_{i=k+1}^{t} \frac{\partial \boldsymbol{h}^{(i)}}{\partial \boldsymbol{h}^{(i-1)}}\right\| > \eta^{t-k}.$$

   for an upper bound $\eta > 1$. Thus, this diverges to ∞ as $t \to \infty$. Thus, we have a *exploding gradient*.

Thus, we will always have vanishing or exploding gradients when using an Elman RNN. This makes it hard for the architecture to capture long-term dependencies. A naive solution is to truncate BPTT to only go back m steps. However, then we still lose long-term signals, which we wanted to preserve.

4.2 *Long-short term memory*

We need to make sure there is constant error flow. For this, we need a connection between timesteps that avoids exploding and vanishing gradients. A simple solution is the leaky unit,

$$\begin{aligned}\hat{\boldsymbol{h}}^{(t)} &= f_{\boldsymbol{\theta}}\Big(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\Big) \\ \boldsymbol{h}^{(t)} &= \alpha \boldsymbol{h}^{(t-1)} + (1-\alpha)\hat{\boldsymbol{h}}^{t}.\end{aligned}$$

The *long-short term memory* (LSTM) architecture [Hochreiter and Schmidhuber, 1997] takes this idea further by keeping a separate memory cell. Access to this cell is protected through gates to make sure that there is always a path between units, such that errors can propagate; see Figure 4.2. Furthermore, in contrast to the leaky unit, the LSTM Learns how to "remember" and "forget" information.

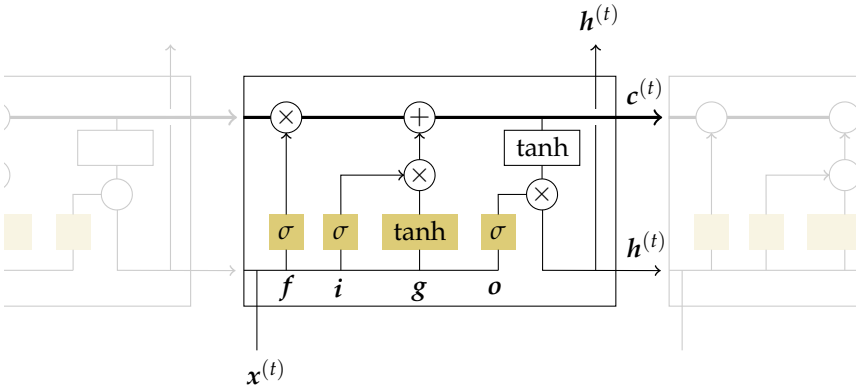

Figure 4.2. LSTM architecture. The yellow squares are neural networks, and the white squares are point-wise operators. As can be seen, there is an "information highway" that can easily propagate errors at the top, because of the minimal modifications made to it.

The cell of an LSTM consists of 4 gates. All gates get $\boldsymbol{h}^{(t-1)}$ and $\boldsymbol{x}^{(t)}$ as input. In particular, these gates have the following instructions,

- $f : \mathbb{R}^d \times \mathbb{R}^d \to [0,1]^d$ is the *forget gate* and has the role of scaling the old cell state $\boldsymbol{h}^{(t-1)}$. It "decides which information should be forgotten" from the previous cell state. It does so by outputting a vector in $[0,1]^d$ where 0 means forgetting completely and 1 means remembering everything;
- $g : \mathbb{R}^d \times \mathbb{R}^d \to [-1,1]^d$ is the *gate* that decides what to write in the cell state $\boldsymbol{c}^{(t)}$;
- $i : \mathbb{R}^d \times \mathbb{R}^d \to [0,1]^d$ is the *input gate* and has the role of "deciding which values of the cell state $\boldsymbol{c}^{(t)}$ should be updated" at the current time step. Again, it does so by outputting a vector in $[0,1]^d$ to decide what of the output of the g gate should be written to the cell state $\boldsymbol{c}^{(t)}$;
- $o : \mathbb{R}^d \times \mathbb{R}^d \to [0,1]^d$ is the *output gate* and has the role of "deciding which values of the current cell state $\boldsymbol{c}^{(t)}$ should be put in the output of the cell $\boldsymbol{h}^{(t)}$".

We first compute all the gates,

$$\begin{aligned}
\boldsymbol{f}^{(t)} &= \sigma\Big(\boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xf}\boldsymbol{x}^{(t)}\Big) \\
\boldsymbol{i}^{(t)} &= \sigma\Big(\boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xi}\boldsymbol{x}^{(t)}\Big) \\
\boldsymbol{o}^{(t)} &= \sigma\Big(\boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xo}\boldsymbol{x}^{(t)}\Big) \\
\boldsymbol{g}^{(t)} &= \tanh\Big(\boldsymbol{W}_{hg}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}_{xg}\boldsymbol{x}^{(t)}\Big).
\end{aligned}$$

You can also chain multiple LSTM units one after another, which results in this computation being performed multiple times per layer. If this is the case, then we replace $\boldsymbol{x}^{(t)}$ by the hidden vector of the previous unit for all units after the first.

Then, we compute the outputs that are propagated to the next layer,

$$\begin{aligned} \boldsymbol{c}^{(t)} &= \boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)} + \boldsymbol{i}^{(t)} \odot \boldsymbol{g}^{(t)} \\ \boldsymbol{h}^{(t)} &= \boldsymbol{o}^{(t)} \odot \tanh\Big(\boldsymbol{c}^{(t)}\Big). \end{aligned}$$

$\boldsymbol{f}^{(t)} \odot \boldsymbol{c}^{(t-1)}$ computes what to keep/forget and $\boldsymbol{i}^{(t)} \odot \boldsymbol{g}^{(t)}$ computes what to add.

The addition in the computation of $\boldsymbol{c}^{(t)}$ allows for gradients to directly propagate through $\boldsymbol{c}^{(t-1)} \odot \boldsymbol{f}$. Also, it allows the model to "select" what information should be retained. For example, at a high level in text, it might be helpful to store information such as gender and countries of origin. See [Olah, 2015] for more information about the gates.

4.3 Gradient clipping

While LSTMs are a great solution to the vanishing gradient problem, we still have the possibility of exploding gradients. This is what *gradient clipping* solves. The idea is to limit the maximum value of the gradient if it surpasses a predetermined threshold. In practice, the gradient descent step gets transformed into the following update rule,

$$\boldsymbol{\theta} \leftarrow \begin{cases} \boldsymbol{\theta} - \gamma \boldsymbol{g} & \|\boldsymbol{g}\| \leq k \\ \boldsymbol{\theta} - \gamma \frac{k}{\|\boldsymbol{g}\|} \boldsymbol{g} & \text{otherwise,} \end{cases}$$

where $\boldsymbol{g}$ is the gradient, γ is the learning rate, and k is the gradient threshold.

5 *Generative models*

In *discriminative models*, the goal is to learn a function that maps inputs $\boldsymbol{x}$ to their correct output y. On the other hand, *generative models* aim to learn the underlying hidden structure of a dataset by modeling the distribution p_{model} to generate new samples that resemble the distribution p_{data}.

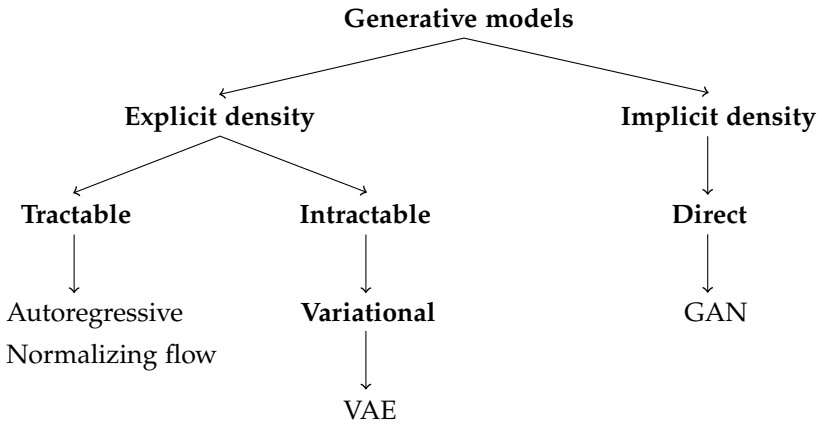

Figure 5.1. Taxonomy of generative models.

Generative models can be classified into two main categories:

- *Explicit models* explicitly define the probability distribution p_{model} and then sample from it;
- *Implicit models* define a model from which we can sample. By being able to sample from the model, it implicitly induces a probability distribution p_{model}.

See Figure 5.1 for further classifications of different models.

6 Autoencoders

Autoencoders [Kramer, 1991] are *generative models*. This means that their objective is to learn the underlying hidden structure of the data. They aim to model the distribution $p_{\text{model}}(\boldsymbol{x})$ that resembles $p_{\text{data}}(\boldsymbol{x})$ to generate new samples. Autoencoders are an *explicit* generative model, which means that they explicitly define the probability distribution $p_{\text{model}}(\boldsymbol{x})$ and then sample from it to generate new data points.

In machine learning, we often have high-dimensional data $\boldsymbol{x} \in \mathbb{R}^n$, such as images, audio, or time-series. Hence, it is crucial to find a low-dimensional representation that can effectively compress the data while preserving its essential information.

Autoencoders offer a solution by making use of the *encoder-decoder structure*; see Figure 6.1. The *encoder* f projects the input space $\mathcal{X}$ into a latent space $\mathcal{Z}$, while the *decoder* g maps the latent space $\mathcal{Z}$ back to the input space $\mathcal{X}$. The assumption made by the autoencoder architecture is that if the decoder is capable of reconstructing the original input solely from the compressed representation, then this compressed representation must be meaningful. Consequently, the composition $g \circ f$ aims to approximate the identity function on the data for a low reconstruction error.

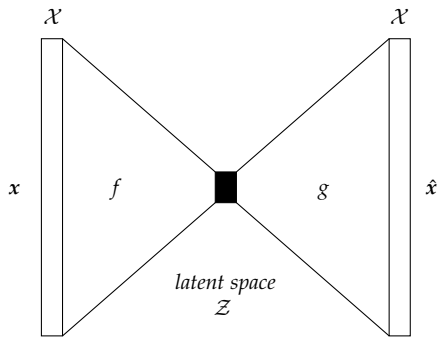

Figure 6.1. Autoencoder architecture.

Furthermore, to enable the generation of new samples from the latent space, the latent space must be well structured, characterized by *continuity* and *interpolation*. Continuity means that the entire space must be covered by the data points, while interpolation means that if we interpolate between two points, then the interpolation must also be a well behaved data point.

6.1 Linear autoencoders

If we restrict f and g to be linear, the encoder f becomes equivalent to the projection performed by *principal component analysis*. The advantage of such a reconstruction is that it can be found in a closed form, and it is interpretable. However, it is not very powerful.

6.2 Non-linear autoencoders

We can gain a lot of performance by allowing f and g to be non-linear. In this case, the encoder and decoder are implemented as neural networks. To train these networks, we optimize for the reconstruction error,

$$\boldsymbol{\phi}^\star, \boldsymbol{\psi}^\star \in \operatorname*{argmin}_{\boldsymbol{\phi},\boldsymbol{\psi}} \sum_{n=1}^{N} \|\boldsymbol{x}_n - g_{\boldsymbol{\psi}}(f_{\boldsymbol{\phi}}(\boldsymbol{x}_n))\|^2$$

We can distinguish between *undercomplete* and *overcomplete* latent spaces. A latent space is undercomplete if $\dim(\mathcal{Z}) < \dim(\mathcal{X})$, while it is over-

complete if $\dim(\mathcal{Z}) > \dim(\mathcal{X})$. The idea of an undercomplete hidden representation is to enable the network to learn the important features of the data by reducing the dimensionality of the hidden space. This prevents the autoencoder from simply copying the input and forces it to extract meaningful and discriminative features. Next, overcomplete latent spaces are useful for denoising and inpainting autoencoders, where we have an imperfect input and want a perfect output. The overcompleteness allows the model to extract more features from the transformed input, leading to improved performance.

6.3 Variational autoencoders

While autoencoders are good at reconstruction, they struggle at generating new high quality samples. This is due to the latent space not being "well-structured", meaning that there is no continuity or interpolation. There are large regions in the latent space where there are no observations, thus the model does not know what to output when it get an input from those regions.

Variational autoencoders (VAE) [Kingma and Welling, 2013] are designed to have a continuous latent space. It achieves this by making the encoder output a probability distribution over latent vectors, rather than a single latent vector. Generally, it outputs a mean vector $\boldsymbol{\mu}$ and standard deviation vector $\boldsymbol{\sigma}^2$ to form a Gaussian distribution over latents $\mathcal{N}(\boldsymbol{\mu}, \text{diag}(\boldsymbol{\sigma}^2))$. The idea is that even for the same input, the latent vector can be different, but in the same area. This means that data points cover areas in the latent space, rather than single points, ensuring continuity and interpolation.

However, since there are no limits on the values taken by $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}^2$, the encoder may learn to generate very different $\boldsymbol{\mu}$ for each class while minimizing $\boldsymbol{\sigma}^2$. This would mean that the encoder essentially outputs points again to decrease the reconstruction error. We can avoid this by minimizing the KL-divergence between the output distribution and a standard normal distribution.[11] Intuitively, this encourages the encoder to distribute the encodings evenly around the center of the latent space.

To train the model, we want to maximize the likelihood of the training data,

$$p(\boldsymbol{x}) = \int p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z}) p(\boldsymbol{z}) \mathrm{d}\boldsymbol{z}.$$

However, this is intractable, because we cannot compute it for all $\boldsymbol{z} \in \mathcal{Z}$. Thus, we define an approximation of the posterior, $q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})$, which is computed by the encoder. We can now derive the *evidence lower bound*

[11] The KL-divergence is defined as

$$D_{\mathrm{KL}}(p \| q) \doteq \mathbb{E}\left[\log\left(\frac{p(x)}{q(x)}\right)\right].$$

It is not symmetric and non-negative.

$p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})$ is induced by the decoder.

(ELBO),

$$
\begin{aligned}
\log p(\boldsymbol{x}) &= \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}[\log p(\boldsymbol{x})] && \boldsymbol{x} \text{ does not depend on } \boldsymbol{z}. \\
&= \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}\left[\log \frac{p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})p(\boldsymbol{z})}{p(\boldsymbol{z} \mid \boldsymbol{x})}\right] && \text{Bayes' rule.} \\
&= \mathbb{E}_{z\sim q_{\boldsymbol{\phi}}(\cdot|\boldsymbol{x})}\left[\log\left(\frac{p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})p(\boldsymbol{z})}{p(\boldsymbol{z} \mid \boldsymbol{x})}\frac{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}\right)\right] && {}^{q(\boldsymbol{z}|\boldsymbol{x})}/_{q(\boldsymbol{z}|\boldsymbol{x})} = 1. \\
&= \mathbb{E}_{\boldsymbol{z}|\boldsymbol{x}}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})] - \mathbb{E}_{\boldsymbol{z}|\boldsymbol{x}}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}{p(\boldsymbol{z})}\right] + \mathbb{E}_{\boldsymbol{z}|\boldsymbol{x}}\left[\log \frac{q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x})}{p(\boldsymbol{z} \mid \boldsymbol{x})}\right] \\
&= \mathbb{E}_{\boldsymbol{z}|\boldsymbol{x}}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})] - D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \| p(\boldsymbol{z})) + D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \| p(\boldsymbol{z} \mid \boldsymbol{x})) \\
&\geq \mathbb{E}_{\boldsymbol{z}|\boldsymbol{x}}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})] - D_{\mathrm{KL}}(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \| p(\boldsymbol{z})). && \text{KL-divergence is non-negative.}
\end{aligned}
$$

The first term of the ELBO encourages low reconstruction error, which encourages the latent space to be structured such that similar data is clustered together. The second term makes sure that the approximate posterior $q_{\boldsymbol{\phi}}$ does not deviate too far from the prior p. The second term can be computed in a closed-form, since both arguments are Gaussian,

$$
\begin{aligned}
q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) &= \mathcal{N}(\boldsymbol{z}; \boldsymbol{\mu}_{\boldsymbol{\phi}}(\boldsymbol{x}), \mathrm{diag}(\boldsymbol{\sigma}^2_{\boldsymbol{\phi}}(\boldsymbol{x}))) \\
p(\boldsymbol{z}) &= \mathcal{N}(\boldsymbol{z}; \mathbf{0}, \boldsymbol{I}).
\end{aligned}
$$

A minor problem is that, during training, we cannot compute the derivative of expectations w.r.t. the parameters that we wish to optimize. Thus, we must use the *reparametrization trick*, which involves treating the random sampling as a single noise term. In particular, instead of sampling $\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}, \mathrm{diag}(\boldsymbol{\sigma}^2))$, we sample $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and compute $\boldsymbol{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$. Using this trick, we can remove the mean and variance from the sampling operation, meaning that we can differentiate w.r.t. the model parameters.

After training, we can sample a latent vector $\boldsymbol{z} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ and pass it to the decoder, which will give a good output, because the latent space is well-structured.

6.4 β-VAE

VAEs still have problems with their latent space; the representations are still *entangled*.[12] This means that we do not have an explicit way of controlling the output. For example, in the MNIST dataset [Deng, 2012], we have no way of explicitly sampling a specific number. The β-VAE [Higgins et al., 2017] solves this problem by giving more weight to the KL term with an adjustable hyperparameter β that balances latent channel capacity and independence constraints with reconstruction accuracy. The intuition behind this is that if factors are in practice independent from each other, the model should benefit from disentangling them.

[12] The latent space is disentangled if every dimension changes a single feature of the output.

In practice, we want to force the KL loss to be under a threshold $\delta > 0$,

$$\begin{aligned} \underset{\boldsymbol{\phi}, \boldsymbol{\psi}}{\text{maximize}} \quad & \mathbb{E}_{\boldsymbol{x} \sim \mathcal{D}}\Big[\mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{x})}\big[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})\big]\Big] \\ \text{subject to} \quad & D_{\mathrm{KL}}\big(q_{\boldsymbol{\phi}}(\boldsymbol{z} \mid \boldsymbol{x}) \| p(\boldsymbol{z})\big) < \delta. \end{aligned}$$

Rewriting this as a Lagrangian, using the Karush-Kuhn-Tucker conditions, we get

$$\begin{aligned} \mathcal{L}(\boldsymbol{\phi}, \boldsymbol{\psi}, \beta) &= \mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{x})}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})] - \beta(D_{\mathrm{KL}}(q_{\boldsymbol{\phi}} \| p(\boldsymbol{z})) - \delta) \\ &= \mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{x})}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})] - \beta D_{\mathrm{KL}}(q_{\boldsymbol{\phi}} \| p(\boldsymbol{z})) + \beta\delta \\ &\geq \mathbb{E}_{\boldsymbol{z} \sim q_{\boldsymbol{\phi}}(\cdot \mid \boldsymbol{x})}[\log p_{\boldsymbol{\psi}}(\boldsymbol{x} \mid \boldsymbol{z})] - \beta D_{\mathrm{KL}}(q_{\boldsymbol{\phi}} \| p(\boldsymbol{z})). \end{aligned}$$

Thus, this becomes our new objective function that we wish to maximize.

7 *Autoregressive models*

We saw that VAEs are approximate models, since they cannot exactly compute the likelihood $p(\boldsymbol{x})$ to maximize it. *Autoregressive models* solve this by computing the likelihood with the chain rule,

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i \mid \boldsymbol{x}_{1:i-1}).$$

Autoregressive models use data from the same input variable at previous timesteps. This is where they get their name from; they perform regression of themselves. In particular, autoregressive models generate one element of the sequence at a time, conditioning on previously generated elements. Thus, it takes $\boldsymbol{x}_{1:k}$ as input and outputs x_{k+1}. Because of this, they can build a probability distribution over possible sequences, using the chain rule as above.

The hard part of this approach is that we must parametrize all possible conditional distributions $p(x_{k+1} \mid \boldsymbol{x}_{1:k})$. Suppose we have a binary image consisting of n pixels. Then, we need

$$\sum_{i=1}^{n} 2^{i-1} \in \mathcal{O}(2^n)$$

parameters to parametrize this model. Thus, we need to make additional assumptions to find more compact representations of the distribution.[13] We will explore the idea of learning a function $f_i : \{0,1\}^{i-1} \to [0,1]$, parametrized by $\boldsymbol{\theta}_i$, which takes as input the previous pixels and outputs the parameter for the Bernoulli distribution. The total number of parameters is

$$\sum_{i=1}^{n} |\boldsymbol{\theta}_i|.$$

Furthermore, we need to think about in which order we generate the pixels of the image. Intuitively, it would make sense to do left-right top-down or top-down left-right. However, in practice, randomly ordering the pixels works just as well. One just needs to make sure that the ordering remains the same for all data points.

[13] A naive solution would be to assume that all points are independent, which would result in

$$p(\boldsymbol{x}) = \prod_{i=1}^{n} p(x_i).$$

Then, we only require n parameters. However, in practice, this would result in a random sampling of pixels, making the generations incoherent.

7.1 *Fully visible sigmoid belief network*

In a *fully visible sigmoid belief network* (FVSBN) [Frey, 1998], each timestep has its own function f_i that is modeled by logistic regression,

$$f_i(\boldsymbol{x}_{1:i-1}) = \sigma\Big(\alpha_0^{(i)} + \alpha_1^{(i)} x_1 + \cdots + \alpha_{i-1}^{(i)} x_{i-1}\Big).$$

At the i-th timestep, we have i parameters denoted by $\boldsymbol{\theta}_i = [\alpha_0, \ldots, \alpha_{i-1}]$. Thus, the total number of parameters is

$$\sum_{i=1}^{n} |\boldsymbol{\theta}_i| = \sum_{i=1}^{n} i = \frac{n^2 + n}{2} \in \mathcal{O}\Big(n^2\Big),$$

which is much better than the exponential number of parameters we had before.

7.2 Neural autoregressive density estimator

The problem with FVSBN is that they are likely not expressive enough for any meaningful tasks, since they only consist of a single linear layer for each timestep. The *neural autoregressive density estimator* (NADE) [Uria et al., 2016] offers an alternative parametrization based on MLPs, where we have hidden layers, increasing expressivity, and the weights are shared between timesteps, decreasing the number of parameters. Specifically, the hidden layer activations can be computed by the following,

$$\begin{aligned}\boldsymbol{h}_i &= \sigma(\boldsymbol{W}_{:,1:i-1}\boldsymbol{x}_{1:i-1} + \boldsymbol{b}) \\ \hat{x}_i &= \sigma(\boldsymbol{V}_{i,:}\boldsymbol{h}_i + c_i).\end{aligned}$$

The advantage of shared parameters is that the total number of parameters gets reduced from $\mathcal{O}(n^2 d)$ to $\mathcal{O}(nd)$, and the hidden unit activations can be evaluated in $\mathcal{O}(nd)$ by using an alternative recursive definition,

$$\begin{aligned}\boldsymbol{a}_0 &= \boldsymbol{0} \\ \boldsymbol{a}_{i+1} &= \boldsymbol{a}_i + \boldsymbol{W}_{:,i}x_i \\ \boldsymbol{h}_i &= \sigma(\boldsymbol{a}_i + \boldsymbol{b}).\end{aligned}$$

Since NADE is a model for binary data, $\hat{x} \in [0, 1]$ is the probability $p(x_i \mid \boldsymbol{x}_{1:i-1})$ at each timestep i. NADE is trained by maximizing the average log-likelihood,

$$\begin{aligned}\frac{1}{T}\sum_{t=1}^{T}\log p\left(\boldsymbol{x}^{(t)}\right) &= \frac{1}{T}\sum_{t=1}^{T}\log\prod_{i=1}^{n} p\left(x_i^{(t)} \mid \boldsymbol{x}_{1:i-1}^{(t)}\right) \\ &= \frac{1}{T}\sum_{t=1}^{T}\sum_{i=1}^{n}\log p\left(x_i^{(t)} \mid \boldsymbol{x}_{1:i-1}^{(t)}\right).\end{aligned}$$

Thus, we optimize the log-likelihood exactly, which was not possible for VAEs.

During training, a *teacher forcing* approach is used, where the ground truth values of pixels are used for conditioning, rather than the predicted ones. This leads to more stable training. However, at inference time we use the predicted value, since we do not have access to the ground truth.

7.3 Masked autoencoder distribution estimation

The idea behind *masked autoencoder distribution estimation* (MADE) [Germain et al., 2015] is to construct an autoencoder which fulfills the autoregressive property, such that its outputs can be used as conditionals. In order to achieve this, no computational path between output unit $\hat{x}_{k+1}$ and any of the input units $x_{k+1}, \ldots, x_n$ may exist. This has the result that $\hat{x}_{k+1}$ is only conditioned on $\boldsymbol{x}_{1:k}$, relative to the ordering.

The way to do this is to first assign a number from 1 to n to each input unit x_i, which we call the ordering. Then, for each hidden unit, uniformly

sample an integer m, between 1 and $n-1$. In the hidden layers, we then only allow units in layer ℓ to propagate to units in layer $\ell+1$ with higher or equal value. Finally, we allow connections between the last hidden layer and the output only to units with value that is strictly greater. Let $m^{(\ell)}(k)$ be the value assigned to the k-th element of layer ℓ, then these constraints can be encoded in mask matrices,

$$\boldsymbol{M}_{ij}^{\boldsymbol{W}^{(\ell)}} = \mathbb{1}\left\{m^{(\ell-1)}(j) \leq m^{(\ell)}(i)\right\}$$
$$\boldsymbol{M}_{ij}^{\boldsymbol{V}} = \mathbb{1}\left\{m^{(n)}(j) < m^{(V)}(i)\right\}.$$

Then, we alter the weight matrices by

$$\bar{\boldsymbol{W}}^{(\ell)} = \boldsymbol{W}^{(\ell)} \odot \boldsymbol{M}^{\boldsymbol{W}^{(\ell)}}$$
$$\bar{\boldsymbol{V}} = \boldsymbol{V} \odot \boldsymbol{M}^{\boldsymbol{V}},$$

and use those instead.

However, a problem with this approach is that it requires very large hidden layers to retain expressivity. And, while it is possible to compute $p(\boldsymbol{x})$ in a single pass, sampling still requires n passes.

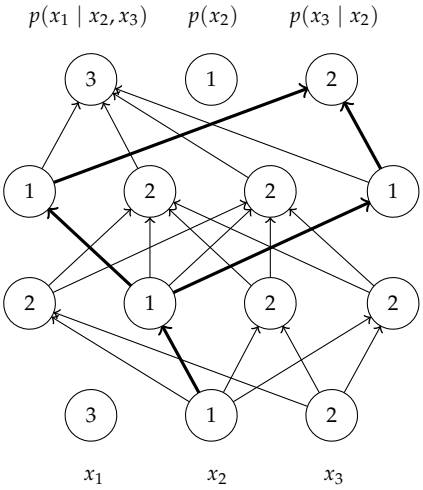

Figure 7.1. MADE masking with $n = 3$.

7.4 *Generating images*

Pixel-RNN. The idea behind *Pixel-RNN* [Van Den Oord et al., 2016] is to generate image pixels starting from the corner and modeling the dependency on previous pixels using an RNN. In particular, a pixel value is dependent on its top-left neighboring pixels and the RNNs hidden state; see Figure 7.2. However, the problem with this approach is that the generation of new pixels depends on the hidden state, making the generation process sequential, and thus slow.

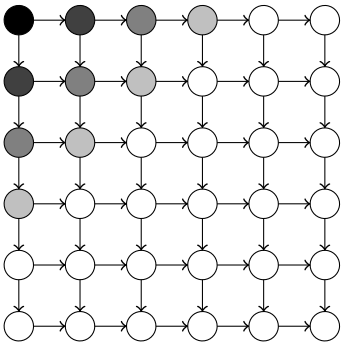

Figure 7.2. Pixel-RNN generation process.

Pixel-CNN. We can solve the problem of Pixel-RNN by assuming that pixel values only depend on a context region around the pixel. This is exactly what the *Pixel-CNN* does [Van Den Oord et al., 2016]. This allows for parallelization during training, because the context region values are known during training.[14] Just as Pixel-RNN, it starts from the top-left corner, but it models the dependencies with a CNN. During training, we need to make sure that only the previously generated pixels are used for prediction of the next, thus we need a mask that masks out all unknown pixel values. However, stacking layers of masked convolutions creates a blind spot in the convolution; see Figure 7.3. The solution to this is to combine horizontal and vertical stacks of convolutions [Van den Oord et al., 2016]. The former conditions on the row so far, while the latter conditions on all rows above. The final output is obtained by summing the two outputs.

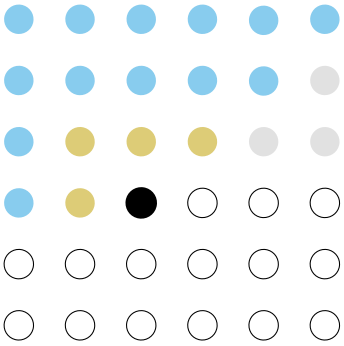

Figure 7.3. Pixel-CNN generation process. The black pixel depends explicitly on the yellow pixels, where the thick-lined pixels denote the masked convolutional layer. The black pixel should also depend on the gray pixels, but it does not due to the way the stacked masked convolutions work; a blind spot.

To enforce the autoregressive property, the model also needs to go over the color channels in an autoregressive manner. Thus the conditional

[14] However, we still have to sequentially predict every token during inference, but this is an issue with all autoregressive models.

probability is expressed as

$$p(\boldsymbol{x}_i \mid \boldsymbol{x}_{1:i-1}) = p(x_{i,\mathrm{R}} \mid \boldsymbol{x}_{1:i-1})p(x_{i,\mathrm{G}} \mid \boldsymbol{x}_{1:i-1}, x_{i,\mathrm{R}})p(x_{i,\mathrm{B}} \mid \boldsymbol{x}_{1:i-1}, x_{i,\mathrm{R}}, x_{i,\mathrm{G}}).$$

7.5 Generating audio

WaveNet [van den Oord et al., 2016] adapts the Pixel-CNN framework for audio data. However, audio typically has much longer time horizons, since they consist of 16000 samples per second. To be able to capture these long-term dependencies efficiently, WaveNet incorporates *dilated convolutions* [Yu and Koltun, 2016]. This allows for an exponential increase in the receptive field. In dilated convolutions, the filter is applied with gaps between the filter elements. WaveNet increases the dilation factor as we go up in the layers to attain an exponential receptive field; see Figure 7.4.

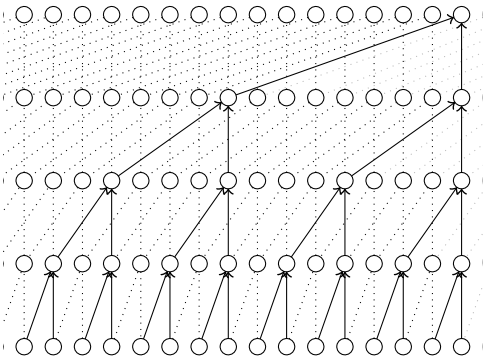

Figure 7.4. Stacked dilated convolutional layers in WaveNet.

7.6 Variational RNN

RNNs can also be used to generate sequences by sampling $\boldsymbol{h}^{(0)}$ and then sequentially predicting the next element, and updating the hidden state. However, the generation of new sequences is very slow, because of the sequential nature of the generation process. Another limitation of vanilla RNNs is that their structure is entirely deterministic and thus limited in expressive power.

The *variational RNN* [Chung et al., 2015] (VRNN) introduces stochasticity to the generations of the RNN, which are typically deterministic. It does so by adding a VAE and sampling the hidden state $\boldsymbol{h}^{(t)}$ from it. For inference (*i.e.*, encoding), it samples the new hidden state by

$$\begin{aligned}\boldsymbol{z}^{(t)} &\sim q_{\boldsymbol{\phi}}\Big(\cdot \mid \boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}\Big)\\ \boldsymbol{h}^{(t)} &\sim p_{\boldsymbol{\theta}}\Big(\cdot \mid \boldsymbol{h}^{(t-1)}, \boldsymbol{z}^{(t)}, \boldsymbol{x}^{(t)}\Big).\end{aligned}$$

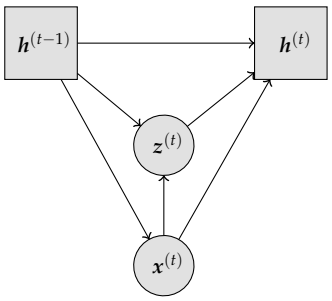

Figure 7.5. Computational graph of VRNN for inference.

For generation, $\boldsymbol{z}_t$ may not depend on $\boldsymbol{x}_t$, so we have

$$\begin{aligned}\boldsymbol{z}^{(t)} &\sim q_{\boldsymbol{\phi}}\Big(\cdot \mid \boldsymbol{h}^{(t-1)}\Big)\\ \boldsymbol{x}^{(t)} &\sim p_{\boldsymbol{\theta}}\Big(\cdot \mid \boldsymbol{h}^{(t)}, \boldsymbol{z}^{(t)}\Big)\\ \boldsymbol{h}^{(t)} &\sim p_{\boldsymbol{\theta}}\Big(\cdot \mid \boldsymbol{h}^{(t-1)}, \boldsymbol{z}^{(t)}, \boldsymbol{x}^{(t)}\Big).\end{aligned}$$

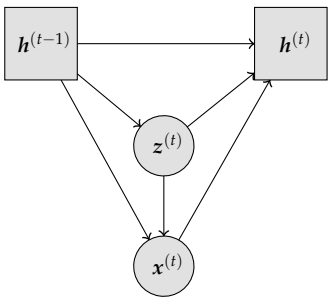

Figure 7.6. Computational graph of VRNN for generation.

The prior is also sampled from a learned distribution.

The *conditional VRNN* (C-VRNN) is an extension to the VRNN, where we can condition on *e.g.* style or, in the case of MNIST, which digit should be generated. During training, the model predicts the conditional variable, called the posterior, and hidden state from the input $\boldsymbol{x}^{(t)}$. From the hidden state it also predicts the conditional variable, called the prior. It then minimizes the reconstruction error from the style variable, and the KL divergence between prior and posterior. During inference, it only

uses the priors for next token prediction, predicted from the hidden state, since it does not have access to $\boldsymbol{x}^{(t)}$. Then, given the predicted token, it predicts the posterior conditional variables, which are used to predict the next hidden state $\boldsymbol{h}^{(t+1)}$.

7.7 Transformers

Transformers [Vaswani et al., 2017] are used in nearly all the state-of-the-art models at the moment. At the basis of transformers lies the self-attention mechanism, where we extract key, value, and query representations from the input. These are then used in a soft-lookup mechanism, where the query and key values decide how much attention is paid to the values.[15]

[15] Note the parallel with dictionaries in programming languages, which use a hard-lookup.

It computes the keys, values, and queries by

$$\begin{aligned}\boldsymbol{K} &= \boldsymbol{X}\boldsymbol{W}_K \\ \boldsymbol{V} &= \boldsymbol{X}\boldsymbol{W}_V \\ \boldsymbol{Q} &= \boldsymbol{X}\boldsymbol{W}_Q,\end{aligned}$$

where $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ and $\boldsymbol{W}_K, \boldsymbol{W}_V, \boldsymbol{W}_Q \in \mathbb{R}^{d\times d}$. Now, we have key, value, and query representations of the input that are all in $\mathbb{R}^{n\times d}$. Then, we can compute the output by

$$\boldsymbol{Y} = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}}\right)\boldsymbol{V}.$$

Intuitively, the softmax computes how much "attention" should be given to each of the previous timesteps by outputting a probability distribution over timesteps.

Since $\boldsymbol{Y} \in \mathbb{R}^{n\times d}$ is of the same dimensionality as $\boldsymbol{X}$, we can stack self-attention layers. This is the basis of the transformer architecture. However, this does not respect the autoregressive property, thus we need to use a mask $\boldsymbol{M}$ to prevent the model from accessing future timesteps, where

$$\boldsymbol{M} = \begin{bmatrix} -\infty & -\infty & \cdots & -\infty \\ 0 & -\infty & \cdots & -\infty \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & -\infty \end{bmatrix}.$$

Then, the attention computation becomes

$$\boldsymbol{Y} = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d}} + \boldsymbol{M}\right)\boldsymbol{V}.$$

The computational complexity of self-attention is $\mathcal{O}(n^2 d)$, which is quite high for large inputs, such as audio and images. But, we gain a maximum path length of $\mathcal{O}(1)$ to any of the previous timesteps, making sure that no information gets "forgotten", and it allows for easy error propagation during training, in contrast to RNNs.

8 Normalizing flows

As we have seen, VAEs learn meaningful representation through latent variables, but they suffer from intractable marginal likelihoods. On the other hand, autoregressive models have a tractable likelihood, but lack a latent space and direct feature learning mechanism. *Normalizing flows* [Rezende and Mohamed, 2015] try to have the best of both worlds; meaningful latent space and a tractable likelihood. They achieve this by leveraging the change of variable technique of integration.

8.1 Change of variables

In the one-dimensional, integration by substitution works as follows. Let $g : [a, b] \to I$ be a differentiable function with a continuous derivative and $f : I \to \mathbb{R}$ be a continuous function, where $I \subset \mathbb{R}$ is an interval. Then,

$$\int_{g(a)}^{g(b)} f(x)\mathrm{d}x = \int_a^b f(g(u))g'(u)\mathrm{d}u.$$

Similarly, we can make the same change of variables transformation to probability distributions. Let $z \sim p_Z$, $x = f(z)$, where $f(\cdot)$ is a monotonic and differentiable function with an inverse $z = f^{-1}(x)$. Then, the probability density function of x is

$$p_X(x) = p_Z\Big(f^{-1}(x)\Big)\left|\frac{\mathrm{d}f^{-1}(x)}{\mathrm{d}x}\right| = p_Z(z)\left|\frac{\mathrm{d}f^{-1}(x)}{\mathrm{d}x}\right|.$$

We can generalize this to any dimensionality by the following,

$$p_X(\boldsymbol{x}) = p_Z\Big(f^{-1}(\boldsymbol{x})\Big)\Big|\det\Big(\boldsymbol{J}_{\boldsymbol{x}} f^{-1}(\boldsymbol{x})\Big)\Big| = p_Z(\boldsymbol{z})|\det(\boldsymbol{J}_{\boldsymbol{z}} f(\boldsymbol{z}))|^{-1},$$

We need to normalize by the determinant of f, because the total probability must be 1. Recall that the determinant quantifies the volume change of an operation.

where the last equality is a property of Jacobians of invertible functions. We can view the absolute determinant as computing the volume of change of f. Normalizing flow models parametrize $f_{\boldsymbol{\theta}}$.

Thus, this gives us a way to compute the probability of $\boldsymbol{x}$ in an exact and tractable manner. The downside of this approach is that f must be invertible, thus we must carefully parametrize the model, which means that we cannot use any model we might want to use. Furthermore, this has the consequence that it must preserve dimensionality, thus the latent space must be as large as the data space.

From a computational perspective, we require the Jacobian of the transformation to be computed efficiently. In general, computing the Jacobian takes $\mathcal{O}(d^3)$ to compute for a $d \times d$ matrix. However, this is not fast enough. A way to achieve linear complexity is to design f such that its Jacobian is a triangular matrix, which takes $\mathcal{O}(d)$ to compute.[16] This requirement further reduces the number of modeling decisions we can make.

[16] The determinant of a triangular matrix is the product of its diagonal entries.

8.2 Coupling layers

A *coupling layer* [Dinh et al., 2014] is a type of network layer that effectively meets the above requirements of a normalizing flow function. It is invertible and offers efficient computation of the determinant. It consists of two functions; β and h. β can be any neural network and does not necessarily need to be invertible.[17] h is an element-wise operation that is invertible w.r.t. its first argument, given the second. $h(\boldsymbol{x}_A, \beta(\boldsymbol{x}_B))$ produces the first half of the input and $\boldsymbol{x}_B$ produces the second half.

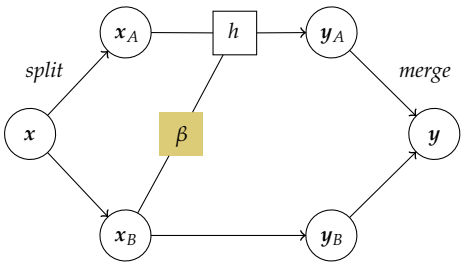

Figure 8.1. Diagram of a coupling layer. h is an invertible element-wise operation and β is can be arbitrarily complex and does not need to be invertible.

[17] This is very important, because requiring a neural network to be invertible would significantly reduce the number of available modeling decisions.

This gives the following function,

$$f : \begin{bmatrix} \boldsymbol{x}_A \\ \boldsymbol{x}_B \end{bmatrix} \mapsto \begin{bmatrix} h(\boldsymbol{x}_A, \beta(\boldsymbol{x}_B)) \\ \boldsymbol{x}_B \end{bmatrix}.$$

The inverse of this function is given by

$$f^{-1} : \begin{bmatrix} \boldsymbol{y}_A \\ \boldsymbol{y}_B \end{bmatrix} \mapsto \begin{bmatrix} h^{-1}(\boldsymbol{y}_A, \beta(\boldsymbol{y}_B)) \\ \boldsymbol{y}_B \end{bmatrix}.$$

The Jacobian matrix can be efficiently computed by

$$\begin{aligned} \boldsymbol{J}f(\boldsymbol{x}) &= \begin{bmatrix} \frac{\partial \boldsymbol{y}_A}{\partial \boldsymbol{x}_A} & \frac{\partial \boldsymbol{y}_A}{\partial \boldsymbol{x}_B} \\ \frac{\partial \boldsymbol{y}_B}{\partial \boldsymbol{x}_A} & \frac{\partial \boldsymbol{y}_B}{\partial \boldsymbol{x}_B} \end{bmatrix} \\ &= \begin{bmatrix} h'(\boldsymbol{x}_A, \beta(\boldsymbol{x}_B)) \frac{\partial \boldsymbol{x}_A}{\partial \boldsymbol{x}_A} & h'(\boldsymbol{x}_A, \beta(\boldsymbol{x}_b)) \frac{\partial \beta(\boldsymbol{x}_B)}{\partial \boldsymbol{x}_B} \\ \frac{\partial \boldsymbol{x}_B}{\partial \boldsymbol{x}_A} & \frac{\partial \boldsymbol{x}_B}{\partial \boldsymbol{x}_B} \end{bmatrix} \\ &= \begin{bmatrix} h'(\boldsymbol{x}_A, \beta(\boldsymbol{x}_B)) & h'(\boldsymbol{x}_A, \beta(\boldsymbol{x}_B))\beta'(\boldsymbol{x}_B) \\ \boldsymbol{0} & \boldsymbol{I} \end{bmatrix}. \end{aligned}$$

When implementing this, we notice that this layer leaves part of its input unchanged. The role of the two subsets in the partition thus gets exchanged in alternating layers, so that both subsets get updates. In practice, we often randomly choose the splits to ensure proper mixing.

8.3 Composing transformations

Often, a single non-linear transformation is insufficient to capture complex patterns. Especially, because a single layer leaves part of the input unchanged. Thus, to achieve more complex transformations, we can compose multiple transformations together. We then get the following function

$$\boldsymbol{x} = f(\boldsymbol{z}) = (f_m \circ \cdots \circ f_1)(\boldsymbol{z}).$$

Again using the change of variables, we can then compute the likelihood by

$$p_X(\boldsymbol{x}) = p_Z\Big(f^{-1}(\boldsymbol{x})\Big) \prod_{k=1}^{m} |\det(\boldsymbol{J}f_k(\boldsymbol{x}))|^{-1}.$$

$\det(\boldsymbol{AB}) = \det(\boldsymbol{A})\det(\boldsymbol{B})$.

8.4 Training and inference

At training time, we can learn the model by maximizing the exact log likelihood over the dataset,

$$\log p_X(\mathcal{D}) = \sum_{i=1}^{n} \log p_Z\Big(f^{-1}(\boldsymbol{x}_i)\Big) + \sum_{k=1}^{m} \log|\det(\boldsymbol{J} f_k(\boldsymbol{x}_i))|^{-1}.$$

At inference time, we generate a sample $\boldsymbol{x}$ by drawing a random $\boldsymbol{z} \sim p_Z$ and transform it via f, obtaining $\boldsymbol{x} = f(\boldsymbol{z})$. To evaluate the probability of an observation, we use the inverse transform to get its latent variable $\boldsymbol{z} = f^{-1}(\boldsymbol{x})$, and compute its probability at $p_Z(\boldsymbol{z})$. Generally, p_Z is chosen to be a simple distribution, such as $\mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$.

8.5 Architectures

The main difference between flow architectures is the choice of h and the way it splits the input into $\boldsymbol{x}_A$ and $\boldsymbol{x}_B$.

NICE. NICE [Dinh et al., 2014] splits the data by dividing the input into two parts $\boldsymbol{x}_A = \boldsymbol{x}_{1:d/2}$ and $\boldsymbol{x}_B = \boldsymbol{x}_{d/2+1:d}$, swapping the order randomly. In the coupling layer, it uses an additive coupling law and the output is computed as

$$\begin{bmatrix} \boldsymbol{y}_A \\ \boldsymbol{y}_B \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_A + \beta(\boldsymbol{x}_B) \\ \boldsymbol{x}_B \end{bmatrix}.$$

RealNVP. RealNVP [Dinh et al., 2016] employs a multi-scale architecture that uses two types of partitioning: checkerboard and channel-wise masking; see Figure 8.2. At each scale, the model alternates between checkerboard patterns. Then, it does a squeezing operation to go from $C \times H \times W$ dimensionality to $4C \times H/2 \times W/2$. Lastly, it uses channel-wise masking. This sequence of steps ensures that all data can interact with each other. Instead of a simple additive h as in NICE, RealNVP implements it as an affine mapping,

$$\begin{bmatrix} \boldsymbol{y}_A \\ \boldsymbol{y}_B \end{bmatrix} = \begin{bmatrix} \boldsymbol{x}_A \odot \exp(s(\boldsymbol{x}_B)) + t(\boldsymbol{x}_B) \\ \boldsymbol{x}_B \end{bmatrix}.$$

Here, s and t can be arbitrarily complex.

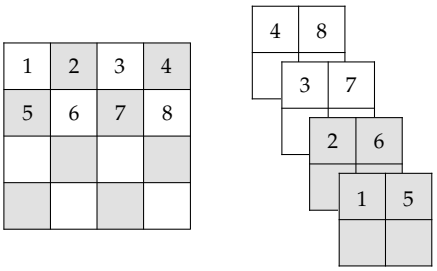

Figure 8.2. Masking used by RealNVP [Dinh et al., 2016].

GLOW. GLOW [Kingma and Dhariwal, 2018] is an architecture that utilizes invertible 1×1 convolutions, affine coupling layers, and multi-scale architecture. It consists of L levels, each of which is composed of K steps of flow. The L levels allow for effective processing of all parts of the input, while the K steps are used to increase the flexibility of the transformation f.

A step of flow consists of applying activation norm, an invertible 1×1 convolution, and a coupling layer, in order. The activation norm

is similar to batch norm, but it normalizes each input channel. As we saw, a permutation of the input is needed in order to be able to process the entire input. The 1×1 convolution with C filters is a generalization of a permutation in the channel dimension. This allows us to learn the required permutation. The coupling layer is implemented as in RealNVP, while the split is performed along the channel dimension only, because the convolution is 1×1.

9 Generative adversarial networks

So far, we have only seen generative models that optimize the likelihood. This has a nice interpretation and leads to nice theory, but there are cases where optimizing the likelihood will not give good results [Theis et al., 2016].

1. In the first case, we might have a good likelihood score, but poor samples. Let's say we have a model p that generates high-quality samples and a model q that generates noise. Now construct the mixture model $0.01p + 0.99q$.[18] The log-likelihood of this model is then

   $$\log(0.01p(\boldsymbol{x}) + 0.99q(\boldsymbol{x})) \geq \log(0.01p(\boldsymbol{x})) = \log(p(\boldsymbol{x})) - \log 100.$$

   The term $\log p(\boldsymbol{x})$ will be proportional to the dimensionality d, while $\log 100$ remains constant. For high-dimensional data, this results in a high log-likelihood for the mixture model;

2. In the second case, we might have a low likelihood score with high-quality samples. This occurs when the model overfits on the training data, meaning that it only outputs data points from the training dataset. This will result in low log-likelihood on the validation dataset, despite the samples being high-quality.

[18] This means that 99% of the time, the generated samples will be noise.

Generate adversarial networks (GAN) [Goodfellow et al., 2014] aim to solve this by introducing a *discriminator* (D), whose job it is to differentiate between real and fake images. The objective of the *generator* (G) is then to maximize the discriminator's classification loss by generating images similar to the dataset. By doing so, it implicitly induces a distribution over data points p_{model}.

Specifically, the generator $G : \mathbb{R}^d \to \mathbb{R}^n$ maps a simple d-dimensional distribution to a sample from the data distribution. The discriminator $D : \mathbb{R}^n \to [0, 1]$ assigns a probability to samples. Its objective is to assign probability 1 to samples from the dataset and probability 0 to samples generated by the generator.

This leads us to the following value function,

$$V(D, G) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}}[D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})}[\log(1 - D(G(\boldsymbol{z})))].$$

The discriminator aims to maximize it, while the generator aims to minimize it, which gives the following optimization problem,

$$G^\star, D^\star \in \operatorname*{argmin}_G \operatorname*{argmax}_D V(D, G).$$

9.1 Theoretical analysis

Optimal discriminator. Given access to p_{model} and p_{data}, we can compute a closed form solution for $D^\star$ by the following theorem.[19]

[19] In practice, we do not have access to p_{model}, since we parametrize it implicitly.

Theorem 9.1. For any generator G that induces p_{model}, the optimal discriminator is

$$D^\star(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_{\text{model}}(\boldsymbol{x})}.$$

Proof. Let G be a generator, then D is computed by

$$\begin{aligned}
D^\star &= \operatorname*{argmax}_D V(G, D) \\
&= \operatorname*{argmax}_D \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D(\boldsymbol{x})] + \mathbb{E}_{z\sim p_{\text{prior}}}[\log(1 - D(G(\boldsymbol{z})))] \\
&= \operatorname*{argmax}_D \int p_{\text{data}}(\boldsymbol{x}) \log D(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} + \int p(\boldsymbol{z}) \log(1 - D(G(\boldsymbol{z}))) \mathrm{d}\boldsymbol{z} \\
&= \operatorname*{argmax}_D \int p_{\text{data}}(\boldsymbol{x}) \log D(\boldsymbol{x}) \mathrm{d}\boldsymbol{x} + \int p_{\text{model}}(\boldsymbol{x}) \log(1 - D(\boldsymbol{x})) \mathrm{d}\boldsymbol{x} && \text{Law of unconscious statistician.} \\
&= \operatorname*{argmax}_D \int p_{\text{data}}(\boldsymbol{x}) \log D(\boldsymbol{x}) + p_{\text{model}}(\boldsymbol{x}) \log(1 - D(\boldsymbol{x})) \mathrm{d}\boldsymbol{x}.
\end{aligned}$$

Let $a, b > 0$ and consider $f(y) = a \log(y) + b \log(1 - y)$, then f's maximum is achieved at $y = \frac{a}{a+b}$, which we can easily prove by

$$\begin{aligned}
f'(y) = \frac{a}{y} - \frac{b}{1-y} = 0 \iff y = \frac{a}{a+b} && \text{Critical point.} \\
f''(y) = -\frac{a}{y^2} - \frac{b}{(1-y)^2} < 0, \quad \forall a, b > 0. && \text{Maximum point.}
\end{aligned}$$

Thus, we obtain

$$D^\star(\boldsymbol{x}) = \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{data}}(\boldsymbol{x}) + p_{\text{model}}(\boldsymbol{x})}, \quad \forall \boldsymbol{x} \in \mathcal{X}.$$

■

Global optimality. Now that we have found the optimal theoretical value for the discriminator, we want to know when we have found a global optimum of the value function. *I.e.*, when is $G^\star$ obtained.

Theorem 9.2. The generator is optimal if $p_{\text{model}} = p_{\text{data}}$ and at optimum, we have

$$V(G^\star, D^\star) = -\log 4.$$

Proof. Let $p = p_{\text{data}}$ and $q = p_{\text{model}}$. We have already found the optimal $D^\star$,

$$D^\star(\boldsymbol{x}) = \frac{p(\boldsymbol{x})}{p(\boldsymbol{x}) + q(\boldsymbol{x})}.$$

We substitute this into the value function,

$$\begin{aligned}
V(G, D^\star) &= \mathbb{E}_p\left[\log\left(\frac{p(\boldsymbol{x})}{p(\boldsymbol{x})+q(\boldsymbol{x})}\right)\right] + \mathbb{E}_q\left[\log\left(1-\frac{p(\boldsymbol{x})}{p(\boldsymbol{x})+q(\boldsymbol{x})}\right)\right] \\
&= \mathbb{E}_p\left[\log\left(\frac{p(\boldsymbol{x})}{p(\boldsymbol{x})+q(\boldsymbol{x})}\right)\right] + \mathbb{E}_q\left[\log\left(\frac{q(\boldsymbol{x})}{p(\boldsymbol{x})+q(\boldsymbol{x})}\right)\right] \\
&= \mathbb{E}_p\left[\log\left(\frac{2p(\boldsymbol{x})}{2(p(\boldsymbol{x})+q(\boldsymbol{x}))}\right)\right] + \mathbb{E}_q\left[\log\left(\frac{2q(\boldsymbol{x})}{2(p(\boldsymbol{x})+q(\boldsymbol{x}))}\right)\right] \\
&= \mathbb{E}_p\left[\log\left(\frac{2p(\boldsymbol{x})}{p(\boldsymbol{x})+q(\boldsymbol{x})}\right)\right] - \log 2 + \mathbb{E}_q\left[\log\left(\frac{2q(\boldsymbol{x})}{p(\boldsymbol{x})+q(\boldsymbol{x})}\right)\right] - \log 2 \\
&= \mathbb{E}_p\left[\log\left(\frac{2p(\boldsymbol{x})}{p(\boldsymbol{x})+q(\boldsymbol{x})}\right)\right] + \mathbb{E}_q\left[\log\left(\frac{2q(\boldsymbol{x})}{p(\boldsymbol{x})+q(\boldsymbol{x})}\right)\right] - \log 4 \\
&= D_{\mathrm{KL}}\left(p\middle\|\frac{p+q}{2}\right) + D_{\mathrm{KL}}\left(q\middle\|\frac{p+q}{2}\right) - \log 4 \\
&= 2D_{\mathrm{JS}}(p\|q) - \log 4,
\end{aligned}$$

where the Jensen-Shannon divergence is a symmetric and smoothed version of the KL divergence, defined as

$$D_{\mathrm{JS}}(p\|q) \doteq \frac{1}{2}D_{\mathrm{KL}}\left(p\middle\|\frac{p+q}{2}\right) + \frac{1}{2}D_{\mathrm{KL}}\left(q\middle\|\frac{p+q}{2}\right).$$

This divergence is non-negative, and equals 0 if and only if $p = q$. Thus, $G^\star \in \operatorname{argmin}_G V(D^\star, G)$ must satisfy $p_{\text{data}} = p_{\text{model}}$, and we obtain

$$V(G^\star, D^\star) = -\log 4.$$

■

Convergence. Under very strong assumptions, we can guarantee that a GAN converges.

Theorem 9.3. Assume that G and D have sufficient capacity, at each update step $D \to D^\star$, and p_{model} is updated to improve

$$\begin{aligned}
V(p_{\text{model}}, D^\star) &= \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}}[\log D^\star(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{x}\sim p_{\text{model}}}[\log(1 - D^\star(\boldsymbol{x}))] \\
&\propto \sup_D \int_{\boldsymbol{x}} p_{\text{model}}(\boldsymbol{x}) \log(1 - D(\boldsymbol{x}))\mathrm{d}\boldsymbol{x}.
\end{aligned}$$

Then, p_{model} converges to p_{data}.

Notice that p_{model} is updated directly here, rather than indirectly by optimizing G, which is actually what happens.

Proof. The argument of the supremum is convex in p_{model} and the supremum preserves convexity. Thus, $V(p_{\text{model}}, D^\star)$ is convex in p_{model} with global optimum as in Theorem 9.2. ■

However, Theorem 9.3 is a very weak result, because of how strong the assumptions are. In practice, G and D have finite capacity, D is optimized for only k steps and does not converge to $D^\star$, and due to the neural network parametrization of G, the objective is no longer convex. However, despite this, GANs work well in practice, because D does stay close to $D^\star$, providing meaningful gradients for G to optimize its generations.

while not converged **do**
  repeat k **times**
    $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n \sim p_{\text{data}}$
    $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
    $\mathcal{L}_D = \frac{1}{n} \sum_{i=1}^{n} \log(D(\boldsymbol{x}_i)) + \log(1 - D(G(\boldsymbol{z}_i)))$
    perform a gradient ascent step on $\mathcal{L}_D$
  end
  $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
  $\mathcal{L}_G = \frac{1}{n} \sum_{i=1}^{n} \log(1 - D(G(\boldsymbol{z}_i)))$
  perform a gradient descent step on $\mathcal{L}_G$
end while

Algorithm 1. Generative adversarial network training algorithm.

A possible issue with this training algorithm is that, early in learning, G is poor, which means that D can easily reject samples with high confidence. In this case, $\log(1 - D(G(\boldsymbol{z})))$ saturates, meaning that it approaches $-\infty$ as $D(G(\boldsymbol{z})) \to 1$. Instead, we can train G to maximize $\log D(G(\boldsymbol{z}))$, which does not have this problem; see Figure 9.1.

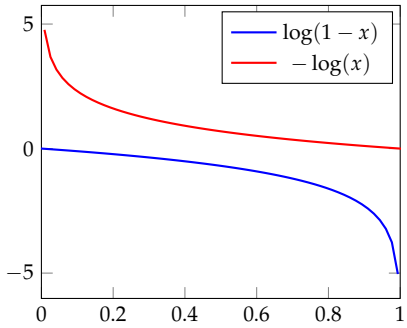

Figure 9.1. Solution to that the GAN loss function saturates for the generator.

Mode collapse. We speak of *mode collapse* when the generator learns to produce high-quality samples with very low variability, covering only a fraction of the data distribution. A simple example that explains this phenomenon is a generator that generates temperature values. The generator may learn to only output cold temperatures, which the discriminator counters by predicting all cold temperatures as "fake" and all warm temperatures as "real". Then, the generator exploits this by only generating warm temperatures. And, again the discriminator can counter this, which the generator counters, repeating cyclically.

The most common solution to mode collapse is the *unrolled GAN* [Metz et al., 2017]. The idea is to optimize the generator w.r.t. the last k discriminators. This results in the above not being able to occur, since the generator must not only fool the current discriminator, which might be unstable, but also the previous k ones.

Training instability. Since we optimize GANs as a two-player game, we need to find a Nash-Equilibrium, where, for both players, moving anywhere will only be worse than the equilibrium. However, this can lead to training instabilities, since making progress for one player may mean the other player being worse off.

Optimizing Jensen-Shannon divergence. It might be the case that the supports of p_{data} and p_{model} are disjoint. In this case, it is always possible to find a perfect discriminator with $D(\boldsymbol{x}) = 1, \forall \boldsymbol{x} \in \text{supp}(p_{\text{data}})$ and $D(\boldsymbol{x}) = 0, \forall \boldsymbol{x} \in \text{supp}(p_{\text{model}})$. Then, the loss function equals zero, meaning that there will be no gradient to update the generator's parameters.[20]

[20] As we saw in the proof of Theorem 9.2, GANs optimize the Jensen-Shannon divergence.

Nowozin et al. [2016] showed that the GAN objective can be generalized to an entire family of divergences. The Wasserstein GAN [Arjovsky et al., 2017] optimizes the Wasserstein distance between p_{model} and p_{data}. In this case, the loss does not fall to zero for disjoint supports, because it measures divergence by how different they are horizontally, rather than vertically. Intuitively, it measures how much "work" it takes to turn one distribution into the other.

Gradient penalty. To stabilize training, Mescheder et al. [2018] proposed adding a gradient penalty,

$$V(G, D) = \mathbb{E}_{\boldsymbol{x} \sim p_{\text{data}}} \left[\log D(\boldsymbol{x}) + \lambda \|\boldsymbol{\nabla} D(\boldsymbol{x})\|^2 \right] + \mathbb{E}_{\boldsymbol{x} \sim p_{\text{model}}} [\log(1 - D(\boldsymbol{x}))].$$

10 Diffusion models

Unlike VAEs and GANs, *diffusion models* [Ho et al., 2020] do not generate a sample in a single step. They do so in many small steps. They start from pure noise and iteratively denoise it. At the end of all the denoising steps, the goal is to obtain a sample from the data distribution. The diffusion chain can be traversed in two directions: going from noise to sample is called *denoising* and going from sample to noise is called *diffusion*. In general, we have

$$\boldsymbol{x}_0 \sim q, \quad \boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}),$$

where q is the data distribution and T is the total number of steps in the diffusion chain. $\boldsymbol{x}_t$ is then a noisy version of $\boldsymbol{x}_0$ at timestep t, where higher t means more noise. The goal of diffusion models is to train a model that can predict $\boldsymbol{x}_{t-1}$, given $\boldsymbol{x}_t$ to reverse the diffusion process. It then performs these small steps T times, starting from $\boldsymbol{x}_T \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$.

10.1 Diffusion step

To generate the training data for a diffusion model, we need an efficient way of computing $\boldsymbol{x}_t$ and $\boldsymbol{x}_{t+1}$. The diffusion chain is governed by a variance schedule $(\beta_t \in (0,1))_{t=1}^T$, where $\beta_t < \beta_{t+1}$ for all t.[21] A naive way of computing the diffusion steps is sequentially,

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

However, this is very inefficient, since to compute $\boldsymbol{x}_t$, we need to perform t diffusion steps. Luckily, there exists a closed-form solution for $\boldsymbol{x}_t$ by using the reparametrization trick,

$$\boldsymbol{x}_t = \sqrt{1-\beta_t}\boldsymbol{x}_{t-1} + \sqrt{\beta_t}\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}).$$

Let $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$. Using the new notation, we get

$$\boldsymbol{x}_t = \sqrt{\alpha_t}\boldsymbol{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}.$$

Using induction, we can find a closed-form solution,

$$\begin{aligned}
\boldsymbol{x}_t &= \sqrt{\alpha_t}(\sqrt{\alpha_{t-1}}\boldsymbol{x}_{t-2} + \sqrt{1-\alpha_{t-1}}\boldsymbol{\eta}) + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}, \quad \boldsymbol{\eta} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}) \\
&= \sqrt{\alpha_t\alpha_{t-1}}\boldsymbol{x}_{t-2} + \sqrt{\alpha_t(1-\alpha_{t-1})}\boldsymbol{\eta} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon} \\
&= \circledast
\end{aligned}$$

Using the properties of multivariate Gaussians, we get

$$\begin{aligned}
\sqrt{1-\alpha_t}\boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, (1-\alpha_t)\boldsymbol{I}) \\
\sqrt{\alpha_t(1-\alpha_{t-1})}\boldsymbol{\eta} &\sim \mathcal{N}(\mathbf{0}, \alpha_t(1-\alpha_{t-1})\boldsymbol{I}) \\
\implies \quad \sqrt{\alpha_t(1-\alpha_{t-1})}\boldsymbol{\eta} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon} &\sim \mathcal{N}(\mathbf{0}, (\alpha_t(1-\alpha_{t-1}) + (1-\alpha_t))\boldsymbol{I}) \\
&= \mathcal{N}(\mathbf{0}, (1-\alpha_t\alpha_{t-1})\boldsymbol{I}).
\end{aligned}$$

[21] There are two main ways of defining the scheduler; a linear schedule and a cosine schedule. It was found that linear schedulers add too much noise too quickly, making it hard for the model to learn. Thus, a cosine scheduler is usually preferred in practice.

Thus, we have

$$\sqrt{\alpha_t(1-\alpha_{t-1})}\boldsymbol{\eta} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon} = \sqrt{1-\alpha_t\alpha_{t-1}}\boldsymbol{\epsilon}', \quad \boldsymbol{\epsilon}' \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}).$$

Since $\boldsymbol{\epsilon}$ and $\boldsymbol{\epsilon}'$ are samples from the same distribution, we can continue using $\boldsymbol{\epsilon}$,

$$\circledast = \sqrt{\alpha_t\alpha_{t-1}}\boldsymbol{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\boldsymbol{\epsilon}$$

Continuing this pattern, we obtain

$$= \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}.$$

10.2 Denoising step

The reverse diffusion (denoising) process obviously has no closed-form solution. Thus, we want to parametrize a model $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)$ that performs the denoising from $\boldsymbol{x}_t$ to $\boldsymbol{x}_{t-1}$. We defined $q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1})$ to be a Gaussian with known parameters. For small enough β_t, we can also model $p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)$ as a Gaussian,

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t) = \mathcal{N}(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t), \boldsymbol{\Sigma}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)).$$

Similarly to VAEs, instead of predicting the full distribution (which would be very hard), we only need to predict the parameters of the Gaussian (which is not as hard). Using this model, we can compute the probability of the full process,

$$p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T}) = p(\boldsymbol{x}_T)\prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t).$$

10.3 Training

Similarly to VAEs, we can derive an ELBO to optimize the log-likelihood,

$$\begin{aligned}
\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_0) &= \log \int p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T}) \mathrm{d}\boldsymbol{x}_{1:T} \\
&= \log \int q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0) \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)} \mathrm{d}\boldsymbol{x}_{1:T} \\
&= \log \mathbb{E}_{\boldsymbol{x}_{1:T}\sim q(\cdot|\boldsymbol{x}_0)}\left[\frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)}\right] \\
&\geq \mathbb{E}_{\boldsymbol{x}_{1:T}\sim q(\cdot|\boldsymbol{x}_0)}\left[\log \frac{p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)}\right] && \text{Jensen's inequality.} \\
&\quad \vdots \\
&= \underbrace{\mathbb{E}_{\boldsymbol{x}_{1:T}\sim q(\cdot|\boldsymbol{x}_0)}[\log p_{\boldsymbol{\theta}}(\boldsymbol{x}_{0:T})]}_{\textit{reconstruction term}} - \underbrace{D_{\mathrm{KL}}(q(\boldsymbol{x}_T \mid \boldsymbol{x}_0) \| p(\boldsymbol{x}_T))}_{\textit{prior matching term}} \\
&\quad - \underbrace{\sum_{t=2}^{T} \mathbb{E}_{\boldsymbol{x}_t\sim q(\cdot|\boldsymbol{x}_0)}[D_{\mathrm{KL}}(q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}_0) \| p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t))]}_{\textit{denoising matching term}}.
\end{aligned}$$

In practice, we assume that the covariance matrix $\boldsymbol{\Sigma}_t = \sigma_t^2 \boldsymbol{I}$ of p and q are the same. Then, we can simplify the denoising matching term,

$$
\begin{aligned}
&\operatorname*{argmin}_{\boldsymbol{\theta}} D_{\mathrm{KL}}(q(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}_0) \| p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t)) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} D_{\mathrm{KL}}\big(\mathcal{N}(\boldsymbol{\mu}_q, \boldsymbol{\Sigma}_t) \| \mathcal{N}(\boldsymbol{\mu}_{\boldsymbol{\theta}}, \boldsymbol{\Sigma}_t)\big) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} \frac{1}{2}\left(\log \frac{\det(\boldsymbol{\Sigma}_t)}{\det(\boldsymbol{\Sigma})} - d + \operatorname{tr}\left(\boldsymbol{\Sigma}_t^{-1}\boldsymbol{\Sigma}_t\right) + (\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q)^\top \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q)\right) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} \frac{1}{2}(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q)^\top (\sigma_t^2 \boldsymbol{I})^{-1}(\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta}} \frac{1}{2\sigma_t^2} \|\boldsymbol{\mu}_{\boldsymbol{\theta}} - \boldsymbol{\mu}_q\|^2.
\end{aligned}
$$

Thus, we want to minimize the difference between the means of the two distributions. Assuming that we have a model that can predict the noise at timestep t, we can compute the two means by

$$
\begin{aligned}
\boldsymbol{\mu}_q(\boldsymbol{x}_t, t) &= \frac{1}{\sqrt{\alpha_t}}\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}\sqrt{\alpha_t}}\boldsymbol{\epsilon} \\
\boldsymbol{\mu}_q(\boldsymbol{x}_t, t) &= \frac{1}{\sqrt{\alpha_t}}\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}\sqrt{\alpha_t}}\hat{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t).
\end{aligned}
$$

A further simplification can be made by formulating the loss directly as the difference between the two actual and predicted noise,

$$
\|\boldsymbol{\epsilon} - \hat{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\|^2 = \left\|\boldsymbol{\epsilon} - \hat{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}\left(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t\right)\right\|^2.
$$

We then get the following algorithms for training and sampling a diffusion model.

Require: $\{\beta_t\}_{t=1}^T$
1: **while** not converged **do**
2: $\quad \boldsymbol{x}_0 \sim q$
3: $\quad t \sim \mathrm{Unif}([T])$
4: $\quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
5: $\quad \mathcal{L} = \left\|\boldsymbol{\epsilon} - \hat{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1-\bar{\alpha}_t}\boldsymbol{\epsilon}, t)\right\|^2$
6: $\quad$ perform a gradient descent step on $\mathcal{L}$
7: **end while**

Algorithm 2. Diffusion model training algorithm.

Require: $\{\beta_t\}_{t=1}^T$, $\hat{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}$
1: $\boldsymbol{x}_T \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$
2: **for** $t = T, \ldots, 1$ **do**
3: $\quad \boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$ if $t > 1$ else $\boldsymbol{z} = \boldsymbol{0}$
4: $\quad \boldsymbol{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\hat{\boldsymbol{\epsilon}}_{\boldsymbol{\theta}}(\boldsymbol{x}_t, t)\right) + \sigma_t \boldsymbol{z}$
5: **end for**
6: **return** $\boldsymbol{x}_0$

Algorithm 3. Diffusion model sampling algorithm.

Typically, $\boldsymbol{\epsilon}_{\boldsymbol{\theta}}$ is implemented as a U-net that is shared across timesteps.

10.4 Guidance

In use-cases such as text-to-image, we want to condition the generated image $\boldsymbol{x}$ on the input text y, meaning that we need to model a conditional distribution,

$$p_{\boldsymbol{\theta}}(\boldsymbol{x} \mid y) = p(\boldsymbol{x}_T) \prod_{t=1}^{T} p_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, y).$$

Contrastive language-image pretraining (CLIP) [Radford et al., 2021] is a large image-language model that has been trained on image-caption pairs. It consists of an image encoder and a text encoder network. By using a contrastive loss, CLIP is encouraged to encode the image and caption into similar embeddings.[22]

[22] A possible application of CLIP is *zero-shot classification*, which leverages the CLIP model to predict the class of an image without any training. It achieves this by predicting the class that maximizes the cosine similarity between the image and the class name.

The idea of *classifier guidance* is to guide denoising in a "direction" favoring images that are more reliably classified by a trained classifier. For this we need a pretrained unconditional diffusion model and a classifier trained on noisy images. Then, we guide the denoising in the "direction" of the classifier by injecting gradients of the classifier model into the sampling process. However, the problem is that this requires training a very specific classifier on noisy data, because we want to guide the diffusion model in all steps of the process. Furthermore, it is hard to interpret what the classifier guidance is doing.

Classifier-free guidance [Ho and Salimans, 2022] address these issues by jointly training a class-conditional and unconditional diffusion model. It then guides the generation process in the generation of conditioning by

$$\boldsymbol{\epsilon}^{\star}_{\boldsymbol{\theta}}(\boldsymbol{x}, y; t) = (1 + \rho)\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}, y; t) - \rho\boldsymbol{\epsilon}_{\boldsymbol{\theta}}(\boldsymbol{x}; t),$$

where y is the conditioning variable and is usually obtained by encoding text using CLIP and ρ controls the strength of the guidance.[23] However, the problem with this approach is that generation takes twice as long when compared to classifier guidance. Overall, guidance improves the quality of the outputs, but reduces the diversity of the generated images.

[23] In practice, we usually train a single model and just set the conditioning variable to all zero for the unconditional generation.

10.5 Latent diffusion models

Diffusion models for image generation typically operate on the original, high-dimensional image size, which can result in slow training and sampling. *Latent diffusion models* [Rombach et al., 2022] address this issue by operating on the latent space of a VAE. The VAE is trained beforehand and is frozen during training of the diffusion model. The diffusion model then does the diffusion reversal on the latent space, rather than the high-dimensional space of the data. This has the advantage that it significantly improves the efficiency of sampling and training. Furthermore, in this approach, the diffusion model only needs to focus on the "semantic" aspect of the image, because the VAE has already captured the relevant information in the latent space.

11 Reinforcement learning

Definition 11.1 (Markov decision process)**.** A Markov decision process (MDP) is a 5-tuple $\langle \mathcal{S}, \mathcal{A}, P, r, \gamma \rangle$, where

- $\mathcal{S}$ is the state space;
- $\mathcal{A}$ is the action space;
- $P : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ is the state transition model;
- $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the expected reward function;
- $\gamma \in [0, 1]$ is the discount factor. If $\gamma = 1$, all future rewards count as much as current reward. If $\gamma = 0$, only immediate rewards matter. Usually, we want a value between 0 and 1, because we want a balance between the two extremes.

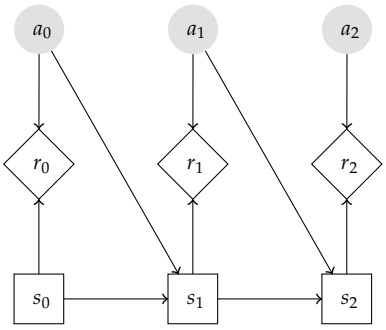

Figure 11.1. Diagram of an MDP.

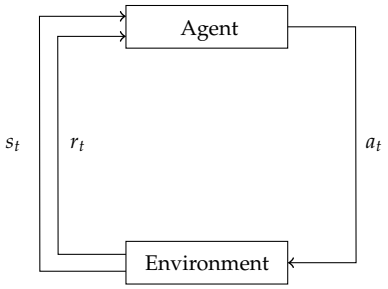

Figure 11.2. Reinforcement learning.

An MDP can be seen as a controlled Markov chain, because the next state s_{t+1} is independent of all states and actions before t if we know (s_t, a_t),

$$\mathbb{P}(s_{t+1} = s' \mid s_t = s, a_t = a, \ldots, s_0, a_0) = P(s' \mid s, a).$$

The performance objective is maximizing the expected discounted reward.

Definition 11.2 (Policy)**.** A policy $\pi : \mathcal{S} \to \Delta(\mathcal{A})$ expresses an agent's behavior by mapping states to a probability distribution over actions. A policy can be made deterministic by outputting only Dirac distributions.

With a fixed policy, we can then put the probability of the next action as

$$\mathbb{P}(a_t = a \mid s_t = s, \ldots, s_0, a_0) = \pi(a \mid s).$$

At any point in the Markov chain, the objective of an agent is to maximize the (discounted) return,

$$G_t = \sum_{k=0}^{\infty} \gamma^k r(s_{t+k}, a_{t+k}).$$

This can also be written in recursive form as

$$G_t = r(s_t, a_t) + \gamma G_{t+1}.$$

Definition 11.3 (Value function)**.** The value function $V^\pi : \mathcal{S} \to \mathbb{R}$ is defined as the expected return under policy π, starting from state s,

$$V^\pi(s) \doteq \mathbb{E}_\pi[G_0 \mid S_0 = s] = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \;\middle|\; S_0 = s\right].$$

From the definition of the value function, we can derive the Bellman consistency equation,

$$
\begin{aligned}
V^\pi(s) &\doteq \mathbb{E}_\pi[G_0 \mid S_0 = s] \\
&= \sum_{a\in\mathcal{A}} \pi(a \mid s)\mathbb{E}_\pi[r(s,a) + \gamma G_1 \mid S_0 = s, A_0 = a] && \text{Recursive definition of return and condition on action.} \\
&= \sum_{a\in\mathcal{A}} \pi(a \mid s)[r(s,a) + \gamma\mathbb{E}_\pi[G_1 \mid S_0 = s, A_0 = a]] && \text{Linearity of expectation.} \\
&= \sum_{a\in\mathcal{A}} \pi(a \mid s)\left[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)\mathbb{E}_\pi[G_1 \mid S_1 = s']\right] && \text{Condition on next state, which makes } (s_0,a_0) \text{ irrelevant for further return due to Markov property.} \\
&= \sum_{a\in\mathcal{A}} \pi(a \mid s)\left[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)\mathbb{E}_\pi[G_0 \mid S_0 = s']\right] && \text{Markov property.} \\
&= \sum_{a\in\mathcal{A}} \pi(a \mid s)\left[r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)V^\pi(s')\right].
\end{aligned}
$$

Furthermore, we have the Bellman optimality equation,

$$V^\star(s) = \max_{a\in\mathcal{A}}\left\{r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)V^\star(s')\right\}.$$

This must always hold for the optimal policy, because the optimal policy acts greedily w.r.t. the optimal value function. We can formalize this as the Bellman optimality operator

$$(\mathcal{T}\boldsymbol{v})(s) \doteq \max_{a\in\mathcal{A}}\left\{r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)\boldsymbol{v}(s')\right\}.$$

$\boldsymbol{v}$ is the vector containing all state values, where $\boldsymbol{v}_s = V^\pi(s)$.

This operator is a γ-contraction w.r.t. the ℓ_∞-norm,

$$\|\mathcal{T}\boldsymbol{v}' - \mathcal{T}\boldsymbol{v}\|_\infty \leq \gamma\|\boldsymbol{v}' - \boldsymbol{v}\|_\infty,$$

and monotonic,

$$\boldsymbol{v} \leq \boldsymbol{v}' \implies \mathcal{T}\boldsymbol{v} \leq \mathcal{T}\boldsymbol{v}'.$$

As we saw above, $\boldsymbol{v}^\star$ is the fixed-point of the Bellman optimality operator,

$$\boldsymbol{v}^\star = \mathcal{T}\boldsymbol{v}^\star.$$

Thus, by applying $\mathcal{T}$ iteratively, we converge linearly in γ,

$$\|\mathcal{T}\boldsymbol{v}_t - \boldsymbol{v}^\star\|_\infty = \|\mathcal{T}\boldsymbol{v}_t - \mathcal{T}\boldsymbol{v}^\star\|_\infty \leq \gamma\|\boldsymbol{v}_t - \boldsymbol{v}^\star\|.$$

This telescopes, giving

$$\|\boldsymbol{v}_t - \boldsymbol{v}^\star\| \leq \gamma^t\|\boldsymbol{v}_0 - \boldsymbol{v}^\star\|.$$

As $t \to \infty$, $\gamma^t \to 0$, showing convergence. This leads us to *value iteration*, which does exactly that; see Algorithm 4. After computing the optimal value function, we can get the optimal policy by acting greedily w.r.t. the returned value function,

$$\pi^\star(s) \in \operatorname*{argmax}_{a\in\mathcal{A}}\left\{r(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s' \mid s,a)V^\star(s')\right\}.$$

$v_0 \leftarrow \mathbf{0}$
while $\|v_t - v_{t-1}\|_\infty > \epsilon$ **do**
    $v_{t+1} = \mathcal{T} v_t$
end while
return v

Algorithm 4. Value iteration, where $\mathcal{T}$ is the Bellman optimality operator.

The runtime complexity of a single iteration is $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$.

Policy iteration is another algorithm that computes the optimal policy given a known MDP. Instead of only operating in the space of the value function, it alternates between computing the value function and the greedy policy w.r.t. the value function; see Algorithm 5. This algorithm converges in fewer iterations than value iteration, but has higher iteration cost of $\mathcal{O}(|\mathcal{S}| + |\mathcal{S}|^2|\mathcal{A}|)$.

$\pi_0 \leftarrow$ random policy
while $\|v^{\pi_t} - v^{\pi_{t-1}}\|_\infty > \epsilon$ **do**
    $V^{\pi_t} \leftarrow \textsc{ValueFunction}(\pi_t)$ ▷ Policy evaluation
    $\pi_{t+1} \leftarrow \textsc{GreedyPolicy}(V^{\pi_t})$ ▷ Policy improvement
end while

Algorithm 5. Policy iteration.

Reinforcement learning. Unlike in MDPs, in reinforcement learning (RL), we do not have access to the transition model P and reward function r. Thus, we cannot make use of value iteration or policy iteration. In some way, we must learn to act optimally within an environment by interacting with it. We can distinguish between two kinds of RL algorithms: *model-free* and *model-based*. In model-based RL, we learn the underlying MDP by approximating P and r. Then, we solve this learned MDP by value or policy iteration. However, we do not *need* the underlying MDP to act optimally. In model-free RL, we learn either the value function, which induces a greedy policy, or directly learn the policy. Generally, model-free algorithms are less expensive to run, while model-based algorithms are more sample efficient.

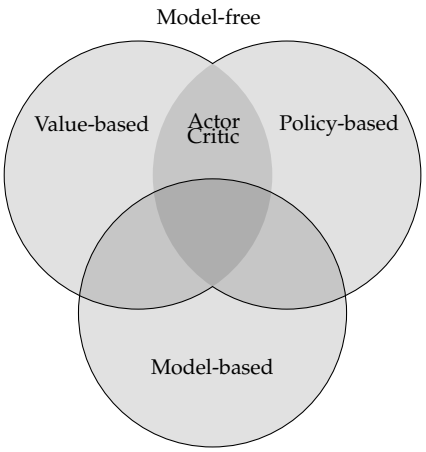

Figure 11.3. Overview of the kinds of reinforcement learning algorithms.

Furthermore, we can distinguish between RL algorithms in how they learn from data; *on-policy* and *off-policy*. On-policy algorithms must learn from actions that result from its own actions, while off-policy algorithms can learn from the trajectory data of any algorithm. Generally, off-policy are preferred, since we can collect data more efficiently.

Lastly, a way that RL differs from supervised learning is that the data depends on past actions, *i.e.*, they are highly correlated, which violates the *independently and identically distributed* assumption made by supervised learning. RL algorithms learn from trajectory data,

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, \ldots).$$

11.1 Monte Carlo methods

The most naive method is the *Monte Carlo method*, which is a value-based on-policy algorithm that estimates the value function by the empirical mean return,

$$V^\pi(s) \approx \frac{1}{n} \sum_{i=1}^{n} G(s)^i,$$

where $G(s)^i$ is the return of episode i, starting from s. The problem with this method is that we need to wait for a trajectory to finish before we can use it for approximating the value function.

11.2 Temporal difference learning

Temporal difference (TD) learning makes it possible to learn from transitions (s, a, r, s'), instead of full trajectories, by approximating the Bellman equation by a single reward, action, and next-state sample,

$$V(s) \leftarrow \alpha V(s) + (1 - \alpha)(r + \gamma V(s')),$$

where $\alpha > 0$ is the learning rate. We use *bootstrapping* here, which means that we use "old" information $V(s')$ as labels. This can also be interpreted as updating the value by the TD error,

$$\begin{aligned} \delta &= r + \gamma V(s') - V(s) \\ V(s) &\leftarrow V(s) + \alpha\delta. \end{aligned}$$

However, to find the optimal value function, we must visit all states sufficiently often. This means that we have to find a balance between exploration and exploitation. A commonly used method is the ϵ-greedy policy, which chooses a random action with small probability ϵ.

SARSA. *SARSA* [Rummery and Niranjan, 1994] is an on-policy algorithm that learns the Q-values of a policy π. It updates Q-values given a transition (s, a, r, s') as follows,

$$Q^\pi(s, a) \leftarrow \alpha Q^\pi(s, a) + (1 - \alpha)(r + \gamma Q^\pi(s', a')), \quad a' \sim \pi(\cdot, s').$$

This is on-policy, because it requires the policy for the update.

Q-learning. An off-policy version of SARSA is *Q-learning* [Watkins and Dayan, 1992]. Note that this learns the optimal Q-values, rather than the Q-values of a certain policy. It updates the Q-values by

$$Q(s, a) \leftarrow \alpha Q(s, a) + (1 - \alpha)(r + \gamma Q(s', a')), \quad a' \in \operatorname*{argmax}_{a \in \mathcal{A}} Q(s', a).$$

Notice that this is an off-policy algorithm, because nowhere in the learning algorithm does it depend on the policy. Given optimal Q-values, it is easy to compute the optimal policy by always acting greedily,

$$\pi(s) \in \operatorname*{argmax}_{a \in \mathcal{A}} Q(s, a).$$

11.3 Deep reinforcement learning

Notice that the policy and the value are simply functions,

$$\pi : \mathcal{S} \to \mathcal{A}, \quad V^{\pi} : \mathcal{S} \to \mathbb{R}.$$

Thus, we can use neural networks to approximate them. This is especially useful for large state and action spaces, since these would not fit in memory.

Replay buffer. The key component to training the following models is the *replay buffer*. Since transitions are highly correlated and we want to use supervised learning techniques, we cannot always use the latest transition for training. Instead, the replay buffer stores the last n transitions and randomly samples from them, making the data points independent and identically distributed.

Deep Q-learning. Deep Q-networks (DQN) [Mnih et al., 2013] estimate the value function of a large, potentially infinite, state space with a finite number of parameters $\boldsymbol{\theta}$,

$$V(s) \approx V_{\boldsymbol{\theta}}(s).$$

It does so in a very similar way to discrete Q-learning. However, instead of updating the Q-values directly, the parameters of the network are updated by the gradients of the following loss function given a transition (s, a, r, s'),

$$\ell(\boldsymbol{\theta}) = \left(Q_{\boldsymbol{\theta}}(s, a) - (r + \gamma Q_{\bar{\boldsymbol{\theta}}}(s', a'))\right)^2, \quad a' \in \operatorname*{argmax}_{a \in \mathcal{A}} Q_{\bar{\boldsymbol{\theta}}}(s', a).$$

Notice that we use different parameters $\bar{\boldsymbol{\theta}}$ for the target value. This is a copy of the parameters $\boldsymbol{\theta}$ that is only updated occasionally. This helps with convergence, because the target is more stable than if we used $\boldsymbol{\theta}$.

Policy search methods. The problem with DQNs is that they do not address potentially large action spaces. *Policy search methods* solve this by directly parametrizing the policy $\pi_{\boldsymbol{\theta}}$. However, it is not trivial to train such a model, since there is no way of knowing what the target action should be. Furthermore, we want the policy to be a probability distribution. Thus, instead we parameterize a Gaussian as the policy,

$$\pi(a \mid s) = \mathcal{N}\left(a; \mu_{\boldsymbol{\theta}}, \sigma^2_{\boldsymbol{\theta}}\right).$$

Recall that the probability of a trajectory τ is computed by

$$\pi_{\boldsymbol{\theta}}(\tau) = p(s_0) \prod_{t=0}^{T} \pi_{\boldsymbol{\theta}}(a_t \mid s_t) p(s_{t+1} \mid s_t, a_t).$$

And, ideally, we want our policy to make trajectories with high return more likely and trajectories with low return unlikely. Thus, we have the

following training objective that we want to maximize,

$$J(\boldsymbol{\theta}) \doteq \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}\left[\sum_{t=0}^{T}\gamma^t r(s_t, a_t)\right].$$

Then, we update the parameters by gradient ascent,

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \eta\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}).$$

But, since we take the expectation w.r.t. the parameters $\boldsymbol{\theta}$, we need to rederive it, such that we can compute its gradient. First, we can rewrite the expectation as

$$\begin{aligned} J(\boldsymbol{\theta}) &= \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}[r(\tau)] \\ &= \int \pi_{\boldsymbol{\theta}}(\tau) r(\tau) d\tau. \end{aligned}$$

Let $r(\tau) \doteq \sum_{t=0}^{T}\gamma^t r(s_t, a_t)$.

Now, we can rewrite the gradient,

$$\begin{aligned} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &= \int \nabla_{\boldsymbol{\theta}}\pi_{\boldsymbol{\theta}}(\tau) r(\tau) d\tau \\ &= \int \pi_{\boldsymbol{\theta}}(\tau)\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(\tau) r(\tau) d\tau \\ &= \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}[\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(\tau) r(\tau)]. \end{aligned}$$

By chain rule, $\nabla\log f(\boldsymbol{x}) = {}^{\nabla f(\boldsymbol{x})}/_{f(\boldsymbol{x})}$.

We can evaluate $\log\pi_{\boldsymbol{\theta}}$ as

$$\begin{aligned} \log\pi_{\boldsymbol{\theta}}(\tau) &= \log\left(p(s_0)\prod_{t=0}^{T}\pi_{\boldsymbol{\theta}}(a_t \mid s_t) p(s_{t+1} \mid a_t, s_t)\right) \\ &= \log p(s_0) + \sum_{t=0}^{T}\log\pi_{\boldsymbol{\theta}}(a_t \mid s_t) + \sum_{t=0}^{T}\log p(s_{t+1} \mid a_t, s_t). \end{aligned}$$

Thus, when we take the gradient, only the second term remains,

$$\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(\tau) = \sum_{t=0}^{T}\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a_t \mid s_t).$$

So, we are able to update $\boldsymbol{\theta}$ without knowing the MDP. Thus, we get the following objective gradient,

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = \mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}\left[\left(\sum_{t=0}^{T}\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right)\left(\sum_{t=0}^{T}\gamma^t r(s_t, a_t)\right)\right].$$

Despite that this value is unbiased, it exhibits high variance. To reduce the variance, we can introduce a baseline $b(\tau)$,

$$\mathbb{E}_{\tau\sim\pi_{\boldsymbol{\theta}}}\left[\left(\sum_{t=0}^{T}\nabla_{\boldsymbol{\theta}}\log\pi_{\boldsymbol{\theta}}(a_t \mid s_t)\right)(r(\tau) - b(\tau))\right],$$

which does not influence the unbiasedness.[24] Thus, we are allowed to shift the reward up or down to reduce the variance. REINFORCE [Sutton et al., 1999] sets this baseline to be

$$b_t(\tau) = \sum_{t'=0}^{t-1}\gamma^{t'} r(s_{t'}, a_{t'}).$$

[24] Critically, the baseline may not depend on the policy $\pi_{\boldsymbol{\theta}}$.

Thus, we get the following expectation,

$$\mathbb{E}_{\tau \sim \pi_{\boldsymbol{\theta}}} \left[\left(\sum_{t=0}^{T} \boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t) \right) G_t \right].$$

I.e., instead of total discounted reward, we only consider the reward from timestep t onward, which naturally has lower variance. Note that REINFORCE is an on-policy algorithm, because it requires training on its own trajectory data, since the expectation is w.r.t. trajectories from the policy itself.

Actor-critic methods. A natural next question is how we can make this off-policy. The key idea behind *actor-critic methods* is to introduce bias and reduce variance by learning a value network next to the policy network. We can then estimate G_t in the REINFORCE gradient by bootstrapping,

$$\boldsymbol{\nabla}_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(a_t \mid s_t)(V(s_t) - (r(s_t, a_t) + \gamma V(s_{t+1}))).$$

We can remove the expectation since nothing depends on τ anymore, since we estimate G_t as $V(s_t) - (r(s_t, a_t) + \gamma V(s_{t+1}))$. Note that if the TD error is zero, it means that the value network has learned the optimal value function and the policy network has learned the optimal policy, leading to no update.

12 Neural implicit representations

The most obvious way to represent 3D objects is by voxels, which are the 3D correspondent of pixels.[25] However, this has $\mathcal{O}(n^3)$ space complexity, which means that the resolution will be very limited. A second approach would be to model 3D objects as points, however this does not model connectivity. Third, we could use meshes, which is used by many downstream tasks. This approach discretizes the object into vertices and faces. However, the problem with this approach is that there will always be an approximation error and may lead to self-intersections.

[25] Essentially, the same as Minecraft blocks.

The approach that we will focus on is the implicit function representation, where the analytic function that represents the 3D surface is learned. This allows us to achieve zero approximation error and a smooth continuous surface with a fixed memory requirement. By the universal approximation theorem, we know that neural networks are able to learn an approximation of any continuous function and since we represent objects as continuous functions, we will be using them for this use case. The only problem with this approach is that a graphical visualization is not directly possible unless we convert the function to one of the previous approaches. Thus, it is still difficult to obtain high frequency details.

There are two kinds of functions that we can use:

- *Occupancy networks* [Mescheder et al., 2019] output the probability of being inside the surface,

$$f_{\boldsymbol{\theta}} : \mathbb{R}^3 \times \mathcal{Z} \to [0, 1].$$

$\mathcal{Z}$ is the set of conditioning variables.

  The surface is then defined to be the set

$$S = \{\boldsymbol{x} \in \mathbb{R}^3 \mid f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \tau\},$$

  for some $\tau \in [0, 1]$;

- *DeepSDF* [Park et al., 2019] outputs the signed distance from the surface (negative if inside, positive if outside),

$$f_{\boldsymbol{\theta}} : \mathbb{R}^3 \times \mathcal{Z} \to \mathbb{R}.$$

  The surface is then defined to be the set

$$S = \{\boldsymbol{x} \in \mathbb{R}^3 \mid f_{\boldsymbol{\theta}}(\boldsymbol{x}) = 0\}.$$

With these, we get a continuous representation with an arbitrary topology and resolution, while achieving a low memory footprint of $\mathcal{O}(|\boldsymbol{\theta}|)$.

12.1 Training with watertight meshes

It is not obvious how to train these networks. We can have multiple kinds of ground truth data, each requiring a different training regimen.

The simplest case is when we have watertight meshes as ground truth.[26] In the case of occupancy networks, we uniformly sample n points $(\boldsymbol{x}_i, y_i) \in \mathbb{R}^3 \times \{0, 1\}$ inside the surface and we train the model using binary cross entropy,

$$\mathcal{L}(\boldsymbol{\theta}) = -\sum_{i=1}^{n} y_i \log f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) + (1 - y_i) \log(1 - f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)).$$

To train a DeepSDF network, we can compute the distance to the mesh and train it as a regression problem. The problem with this ground truth is that it is very expensive.

[26] "Watertight" means that the meshes have no holes, thus the space is divided into inside and outside.

12.2 Training with point clouds

A cheaper alternative is to use point clouds. With this data, it would be very hard to train an occupancy network, because we have no concept of inside or outside. But, training DeepSDF would be possible by training the model to output 0 at the points. However, in this case, a trivial optimal solution with zero loss would be for the model to always output 0. Thus, we introduce the *Eikonal term* [Gropp et al., 2020] to the loss function, which pushes the gradient to be 1 everywhere,

$$\mathcal{L}(\boldsymbol{\theta}) = \sum_{i=1}^{n} \underbrace{|f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)|^2}_{\text{Vanish term}} + \lambda \underbrace{\mathbb{E}_{\boldsymbol{x}}\left[(\|\nabla_{\boldsymbol{x}} f_{\boldsymbol{\theta}}(\boldsymbol{x})\| - 1)^2\right]}_{\text{Eikonal term}}.$$

This makes sense from a "distance" perspective, since we want to increase the distance by 1 when we move 1 unit away. Also, always outputting 0 is no longer optimal.

It is possible to prove that if we parametrize a linear model with random initialization and the points are sampled from a plane, this method will be able to solve the problem by optimizing with gradient descent.

12.3 Training with 2D images

While point clouds are relatively inexpensive compared to watertight meshes, we have an exponentially larger dataset if we only consider 2D images. The idea is to render the model induced by $f_{\boldsymbol{\theta}}$ in the same angle as the image and using a photometric reconstruction loss,

$$\ell(\hat{\mathbf{I}}, \mathbf{I}) = \sum_{\boldsymbol{u}} \|\hat{\mathbf{I}}_{\boldsymbol{u}} - \mathbf{I}_{\boldsymbol{u}}\|.$$

In order to learn from this, the entire computational graph between $f_{\boldsymbol{\theta}}$ and $\ell(\hat{\mathbf{I}}, \mathbf{I})$, including the rendering, must be differentiable. In order to render color images, we introduce a texture field,

$$\boldsymbol{t}_{\boldsymbol{\theta}} : \mathbb{R}^3 \times \mathcal{Z} \to \mathcal{C},$$

where $\mathcal{C} \subset \mathbb{R}^3$ is the color space.

For a camera located at $\boldsymbol{r}_0$, we can predict the color $\hat{\mathbf{I}}_{\boldsymbol{u}}$ at pixel location $\boldsymbol{u}$ by casting a ray from $\boldsymbol{r}_0$ through $\boldsymbol{u}$ and determining the first point of intersection $\hat{\boldsymbol{p}}$ with the isosurface $\{\boldsymbol{p} \in \mathbb{R}^3 \mid f_{\boldsymbol{\theta}}(\boldsymbol{p}) = \tau\}$; see Figure 12.1. The color is then given by the texture field $\hat{\mathbf{I}}_{\boldsymbol{u}} = \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})$ [Niemeyer et al., 2020]. The point $\hat{\boldsymbol{p}}$ is found by the secant method.

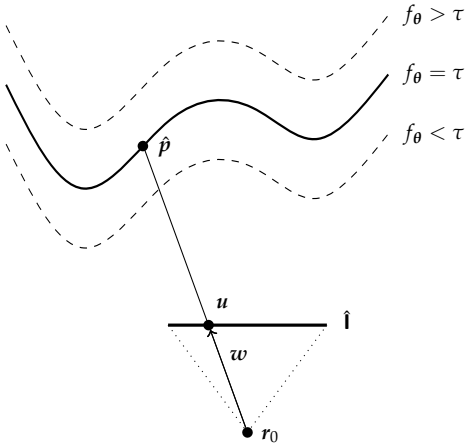

Figure 12.1. To render an object from the occupancy network $f_{\boldsymbol{\theta}}$ and texture field $\boldsymbol{t}_{\boldsymbol{\theta}}$, we cast a ray with direction $\boldsymbol{w}$ through a pixel $\boldsymbol{u}$ and determine the intersection point $\hat{\boldsymbol{p}}$ with the isosurface $f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}}) = \tau$. Afterward, we evaluate the texture field $\boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})$ to obtain the color prediction $\hat{\mathbf{I}}_{\boldsymbol{u}}$.

Secant method. In order to efficiently find points on the surface, we use the secant method; see Algorithm 6. It involves iteratively approximating $f_{\boldsymbol{\theta}}$ as the plane connecting $(\boldsymbol{x}_{t-2}, f_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-2}))$ and $(\boldsymbol{x}_{t-1}, f_{\boldsymbol{\theta}}(\boldsymbol{x}_{t-1}))$, and computing the zero-crossing of that plane. This approximation gets closer and closer to the actual zero-crossing of $f_{\boldsymbol{\theta}}$; see Figure 12.2.

Require: Initial points x_0, x_1
1: **for** $t = 2, \ldots, T$ **do**
2: Compute intersection of line connecting $(x_{t-2}, f(x_{t-2}))$ and $(x_{t-1}, f(x_{t-1}))$, and the x-axis,

$$x_t = x_{t-1} - f(x_{t-1}) \frac{x_{t-1} - x_{t-2}}{f(x_{t-1}) - f(x_{t-2})}.$$

3: **end for**
4: **return** $\boldsymbol{x}_T$

Algorithm 6. Secant method for finding a zero-crossing of f.

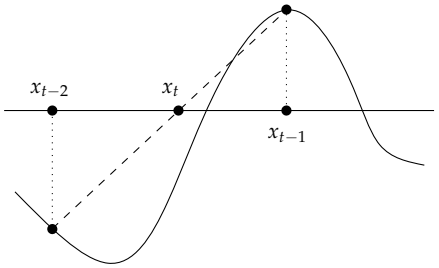

Figure 12.2. Illustration of the secant method.

It is important that we get the first zero-crossing, since that is the point that is seen.

Forward pass. The forward pass is done by querying the occupancy network for all the pixels. This gives us three types of points:

- $f_{\boldsymbol{\theta}}(\boldsymbol{p}) < \tau$: outside surface;
- $f_{\boldsymbol{\theta}}(\boldsymbol{p}) > \tau$: inside surface;
- $f_{\boldsymbol{\theta}}(\boldsymbol{p}) = \tau$: in the surface.

Thus, for all points $\boldsymbol{p}$ with $f_{\boldsymbol{\theta}}(\boldsymbol{p}) = \tau$, we evaluate the texture field $\boldsymbol{t}_{\boldsymbol{\theta}}(\boldsymbol{p})$ and assign the pixel $\boldsymbol{u}$ that color.

Backward pass. To obtain gradients, we use the chain rule,

$$\begin{aligned}
\frac{\partial \ell(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}} &= \sum_{\boldsymbol{u}} \frac{\partial \ell(\boldsymbol{\theta})}{\partial \hat{\mathbf{I}}_{\boldsymbol{u}}} \frac{\partial \hat{\mathbf{I}}_{\boldsymbol{u}}}{\partial \boldsymbol{\theta}} \\
&= \sum_{\boldsymbol{u}} \frac{\partial \ell(\boldsymbol{\theta})}{\partial \hat{\mathbf{I}}_{\boldsymbol{u}}} \left(\frac{\partial^+ \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} + \frac{\partial \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \frac{\partial \hat{\boldsymbol{p}}}{\partial \boldsymbol{\theta}} \right) \\
&= \sum_{\boldsymbol{u}} \frac{\partial \ell(\boldsymbol{\theta})}{\partial \hat{\mathbf{I}}_{\boldsymbol{u}}} \left(\frac{\partial^+ \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} + \frac{\partial \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \frac{\partial \boldsymbol{r}(\hat{d})}{\partial \boldsymbol{\theta}} \right) \\
&= \sum_{\boldsymbol{u}} \frac{\partial \ell(\boldsymbol{\theta})}{\partial \hat{\mathbf{I}}_{\boldsymbol{u}}} \left(\frac{\partial^+ \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} + \frac{\partial \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \boldsymbol{w} \frac{\partial \hat{d}}{\partial \boldsymbol{\theta}} \right)
\end{aligned}$$

The $+$ indicates that we compute the derivative directly w.r.t. $\boldsymbol{t}_{\boldsymbol{\theta}}$ and not $\hat{\boldsymbol{p}}$.

$\boldsymbol{r}(d) \doteq \boldsymbol{r}_0 + d\boldsymbol{w}$ is the ray through pixel $\boldsymbol{u}$.

To compute $\frac{\partial \hat{d}}{\partial \boldsymbol{\theta}}$, we need to use implicit differentiation, meaning that we take the derivative of $f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}}) = \tau$ on both sides,

$$\begin{aligned}
\frac{\partial f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} &= 0 \\
\frac{\partial^+ f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} + \frac{\partial f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \frac{\partial \hat{\boldsymbol{p}}}{\partial \boldsymbol{\theta}} &= 0 \\
\frac{\partial^+ f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} + \frac{\partial f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \boldsymbol{w} \frac{\partial \hat{d}}{\partial \boldsymbol{\theta}} &= 0 \\
\frac{\partial \hat{d}}{\partial \boldsymbol{\theta}} &= -\left(\frac{\partial f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \boldsymbol{w} \right)^{-1} \frac{\partial^+ f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}}.
\end{aligned}$$

Thus, we get the following,

$$= \sum_{\boldsymbol{u}} \frac{\partial \ell(\boldsymbol{\theta})}{\partial \hat{\mathbf{l}}_{\boldsymbol{u}}} \left(\frac{\partial^+ \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} - \frac{\partial \boldsymbol{t}_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \boldsymbol{w} \left(\frac{\partial f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \hat{\boldsymbol{p}}} \boldsymbol{w} \right)^{-1} \frac{\partial^+ f_{\boldsymbol{\theta}}(\hat{\boldsymbol{p}})}{\partial \boldsymbol{\theta}} \right).$$

Thus, we do not need to store intermediate results of $\hat{\boldsymbol{p}}$ to compute the gradient.

12.4 Neural radiance field

Compared to implicit surfaces, the advantage of NERF is that it can model transparency and thin structure, but it generally leads to worse geometry.

The problem with the approach thus far is that it is not great at learning very complex scenes. For example, representing thin structures by a surface is very difficult. Also, transparency is not possible. To solve these problems, *neural radiance fields* (NERF) [Mildenhall et al., 2021] were introduced. It introduces a density σ that can be used to learn difficult structures, such as hair and windows. In particular, our new function takes a 3D position (x, y, z) and the camera parameters (θ, ϕ) as input, and output the color (r, g, b) and density σ; see Figure 12.3. The camera parameters are important to model view-dependent effects, such as glare.

Rendering. Technically, we do not represent surfaces, but rather "volumetric clouds". Instead of looking for the surface, we sample along the whole ray and compute a weighted average to be the color. The density models how much light is propagated to the next point on the ray. Let σ_i be the density of point i along the ray, then we define the probability of light stopping at this point as

$$\alpha_i = 1 - \exp(-\sigma_i \delta_i), \quad \delta_i = t_{i+1} - t_i.$$

Then, we can compute the probability of light reaching point i by

$$T_i = \prod_{j=1}^{i-1} 1 - \alpha_j.$$

In order to compute the final color, we take the weighted average of the colors along the ray, weighted by the probability $T_i \alpha_i$ of light reaching the point and stopping there,

$$\boldsymbol{c} = \sum_{i=1}^{n} T_i \alpha_i \boldsymbol{c}_i.$$

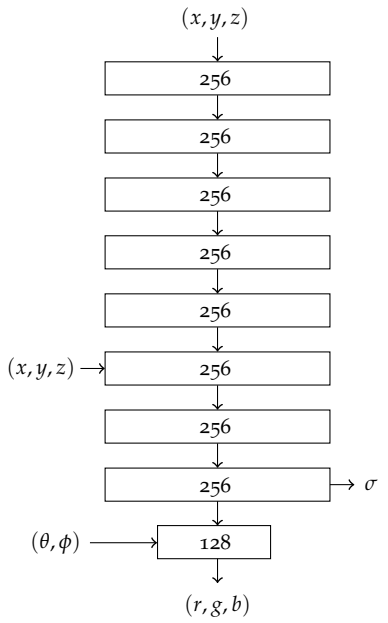

Figure 12.3. NERF architecture. The location information (x, y, z) is introduced twice to make sure that the network does not "forget" it. Furthermore, the view direction (θ, ϕ) is introduced after outputting the density σ, because density does not depend on the view direction physically.

We then do a simple backward pass by backpropagating from the loss of this color, compared to the actual image. The fundamental difference with differentiable rendering is that we backpropagate through multiple points rather than a single point.

Since the sampling is quite expensive, we can try to sample more in positions with higher weights. This can be done by initially sampling a few points uniformly and then sampling more around points with high weight.

Positional encoding. In general, neural networks perform poorly at representing high-frequency variation in color and geometry. This happens because they are biased toward learning lower frequency functions. Thus, the reconstructions using the above architecture result in poor renderings. The solution to this is to introduce positional encodings,[27] mapping the inputs (x, y, z, θ, ϕ) to a higher dimensional space by

$$\gamma(p) = \left[\sin\left(p \cdot 2^0 \pi\right), \cos\left(p \cdot 2^0 \pi\right), \ldots, \sin\left(p \cdot 2^{L-1} \pi\right), \cos\left(p \cdot 2^{L-1} \pi\right)\right].$$

This has the consequence that a low-frequency transformation w.r.t. $\gamma(p)$ is a high-frequency transformation w.r.t. p.

[27] A similar mapping is used in the transformer architecture, but for different reasons.

Limitations. The problems with NERF are that it requires many calibrated views, rendering is slow, and it can only model static scenes.

12.5 *Gaussian splatting*

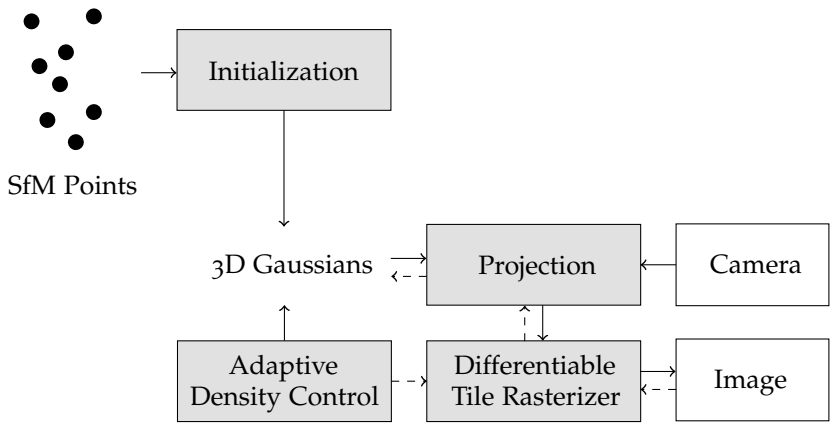

Figure 12.4. Workflow of Gaussian splatting. The dashed lines are gradient flow and solid lines are operation flow.

The problem with NERF is that it is slow, because of the number of samples needed for rendering. We can reduce the amount of samples by estimating a set of primitives around the object boundary and only training/sampling within these objects [Lombardi et al., 2021]. This paper used cubes, but that is hard to optimize. Lassner and Zollhofer [2021] parametrized with spheres, rather than cubes, which can be optimized

from scratch from a randomly sampled initial sphere cloud. However, this approach has the problem that it is very hard to model thin structures with isotropic shapes, such as spheres and cubes. The solution is to model the scene by many 3D Gaussians [Kerbl et al., 2023].

The scene is initialized by a point cloud, obtained through Structure from Motion. Then, iteratively we project the Gaussians onto the camera's image plane and weight them similarly to NERF, compute the loss, and backpropagate to update the Gaussians. Note that we do not have to optimize a neural network, but only need to optimize the Gaussians directly. Furthermore, the model adaptively adds more points as necessary.[28] The weight of each Gaussian at a point $\boldsymbol{x}$ can be computed by

$$\alpha_i(\boldsymbol{x}) = o_i \cdot \exp\left(-\frac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_i')^\top \boldsymbol{\Sigma}_i'^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_i')\right),$$

where o_i is the opacity, and $\boldsymbol{\mu}_i'$ and $\boldsymbol{\Sigma}_i'$ are the parameters of the 2D projection of the i-th 3D Gaussian.

[28] It does so by densifying points every 100 iterations and removing any Gaussians that are essentially transparent, *i.e.*, with $\alpha < \epsilon_\alpha$. It densifies by splitting Gaussians with large variance into two smaller ones. Furthermore, to keep the amount of Gaussians low, it sets all o_i to zero every 300 iterations. The model will then increase the opacity of Gaussians that are needed and the rest are culled by the deletion process.

13 Parametric body models

Being able to represent and track a body is interesting for many applications, such as virtual reality and augmented reality. The easier problem is to estimate the projected pose from 2D images, while the harder problem is to estimate 3D pose from images. For both problems, we need to model the body and learn a feature representation for prediction.

13.1 2D poses

Body modeling. Body modeling entails finding a way to understand how the different parts of the body are linked. The *pictorial structure model* [Yang and Ramanan, 2011] models the body as a graph, where the joints are vertices and they are connected by edges. For example, in a simplified model, the foot vertex might be connected to the knee vertex.

In a 2D image, we indicate the position of vertex i by $\ell_i = [x_i, y_i]$. Then, given an image $\mathbf{I}$ and a configuration $L = [\ell_1, \ldots, \ell_k]$, we can define the score of that configuration by

$$S(\mathbf{I}, L) = \sum_{i \in V} \alpha_i \phi(\mathbf{I}, \ell_i) + \sum_{i,j \in E} \beta_{ij} \psi(\ell_i, \ell_j),$$

where the first term measures the score of placing vertex i at ℓ_i in image $\mathbf{I}$ and the second term measures the deformation between connected vertices. The best configuration is the one that minimizes the score.

We can further generalize this model to a mixture of non-oriented pictorial structures. Let m_i be the mixture component type of vertex i. The mixture component expresses concepts as orientations of a part, such as vertically vs horizontally oriented hand, front-view head vs side-view head, or open versus closed hand. Formally, we compute the score by

$$S(\mathbf{I}, L, M) = \sum_{i \in V} \alpha_i^{m_i} \phi(\mathbf{I}, \ell_i) + \sum_{i,j \in E} \beta_{ij}^{m_i, m_j} \psi(\ell_i, \ell_j) + S(M),$$

where $\alpha_i^{m_i}$ is the "local appearance template" for part i with type m_i, $\beta_{ij}^{m_i, m_j}$ expresses the likelihood of having template m_i for part i and template m_j for part j given the distance between ℓ_i and ℓ_j, and $S(M)$ is the co-occurrence bias defined by

$$S(M) = \sum_{i,j \in E} b_{ij}^{m_i, m_j},$$

where $b_{ij}^{m_i, m_j}$ is the pairwise co-occurrence prior. This allows us to model things such that a vertical lower leg is more likely to be connected to a vertical upper leg.

Feature learning. One architecture that does feature representation learning by direct regression is *DeepPose* [Toshev and Szegedy, 2014], which uses CNNs to directly compute ℓ_i for all i. It does so in multiple stages,

where in the first stage, it only gets the image as input. In the further stages, it gets the image and the previous estimate and input and must refine the estimate by taking only a patch around the previous estimate.

A different way of doing it is through heatmaps. *Convolutional pose machines* [Wei et al., 2016] predicts heatmaps for each vertex. Like DeepPose, it also uses a refinement process with intermediate losses. The key is that the receptive field grows as stages are applied, which allows the model to capture long-range dependencies.

Body modeling and feature learning. We can also combine the two methods by first obtaining the heatmaps and then refining the predictions using body modeling.

13.2 3D poses

Linear blend skinning. Linear blend skinning (LBS) is the simplest method of transforming a rest pose into a specific pose. It does so by transforming vertices as a weighted linear combination of global joint transformations,

$$\boldsymbol{t}_i' = \left(\sum_k w_{ki} \boldsymbol{G}_k(\boldsymbol{\theta}, \boldsymbol{J}) \right) \boldsymbol{t}_i.$$

Each bone-joint pair is given a weight w_{ki}, which determines the influence of the transformation of a bone on the joint. $\boldsymbol{G}_k(\boldsymbol{\theta})$ is the rigid bone transformation for bone k, which transforms the original joint location $\boldsymbol{t}_i$, which is linearly weighted by w_{ki}. Here, $\boldsymbol{\theta}$ is the desired pose and $\boldsymbol{J}$ are the joint locations.

The problem with this approach is that it does not account for any variation in the body shape. Also, often there are problems with unrealistic deformations, caused by the pose.

SMPL. The *skinned multi-person linear* (SMPL) model [Loper et al., 2015] solves the problems of linear blend skinning by allowing the model to learn how to account for body shape variation and deformations caused by the poses. First an overview of how these problems are solved is given, then we will see how they are learned from data.

SMPL encodes human subjects by two parameters: a body shape parameter $\boldsymbol{\beta} \in \mathbb{R}^{10}$ and a pose parameter $\boldsymbol{\theta} \in \mathbb{R}^{9K}$. The template mesh $\bar{\boldsymbol{T}}$ is the starting pose, which transformations are relative to.[29] In the following, we assume access to $\boldsymbol{B}_S(\boldsymbol{\beta})$ and $\boldsymbol{B}_P(\boldsymbol{\theta})$, which we learn from data.

[29] Usually, the template mesh is a T-pose. The matrix $\bar{\boldsymbol{T}} \in \mathbb{R}^{N \times 3}$ contains the location of all vertices.

1. First, translate the template mesh to the identity mesh for the specific body shape, parametrized by $\boldsymbol{\beta}$,

$$\bar{\boldsymbol{T}} + \boldsymbol{B}_S(\boldsymbol{\beta});$$

$\boldsymbol{B}_S(\boldsymbol{\beta})$ encodes the variation in the body shape from the mean shape.

2. Then, translate the identity mesh to correct for deformations caused by linear blend skinning, which is a function of the pose,

$$\boldsymbol{T}_P(\boldsymbol{\beta}, \boldsymbol{\theta}) = \bar{\boldsymbol{T}} + \boldsymbol{B}_S(\boldsymbol{\beta}) + \boldsymbol{B}_P(\boldsymbol{\theta});$$

3. Lastly, perform linear blend skinning on the resulting base mesh,

$$\bar{\boldsymbol{t}}'_i = \left(\sum_k w_{ki} \boldsymbol{G}_k(\boldsymbol{\theta}, \boldsymbol{J}(\boldsymbol{\beta})) \right) (\bar{\boldsymbol{t}}_i + \boldsymbol{b}_{S,i}(\boldsymbol{\beta}) + \boldsymbol{b}_{P,i}(\boldsymbol{\theta})).$$

Learning shape variation. To learn the variations in body shape, we need a dataset of many body shapes. Each body shape is represented by its joint locations. To be able to linearly represent any body shape with a small amount of parameters, we perform PCA on this dataset, and only take the top principal components. Each body shape can then be defined by a linear combination of these components,

$$\boldsymbol{B}_S(\boldsymbol{\beta}; \mathcal{S}) = \sum_{n=1}^{|\boldsymbol{\beta}|} \beta_n \boldsymbol{S}_n.$$

Specifically, the data must be relative to the template mesh joints, such that $\boldsymbol{B}_S(\boldsymbol{\beta})$ is also relative to these joints. Then, we can use them as a measure of variation from the template.

Learning pose-dependent deformations. Let $\boldsymbol{R} : \mathbb{R}^{|\boldsymbol{\theta}|} \to \mathbb{R}^{9K}$ be a function that maps the pose vector $\boldsymbol{\theta}$ to a vector of concatenated relative rotation matrices.[30] The used rig has $K = 23$ joints, thus $\boldsymbol{R}(\boldsymbol{\theta}) \in \mathbb{R}^{207}$. Let $\boldsymbol{\theta}^\star$ be the rest pose, then the vertex deviations from the rest template are given by

$$\boldsymbol{B}_P(\boldsymbol{\theta}; \mathcal{P}) = \sum_{n=1}^{9K} (\boldsymbol{R}_n(\boldsymbol{\theta}) - \boldsymbol{R}_n(\boldsymbol{\theta}^\star)) \boldsymbol{P}_n,$$

where $\boldsymbol{P}_n \in \mathbb{R}^{3N}$ is a vector of vertex displacements.[31] Thus, the pose-dependent translation is a linear combination of vertex displacements $\mathcal{P}$, dependent on the difference in pose. This matrix $\mathcal{P}$ is learned from a dataset of many poses.

[30] Each rotation matrix has dimensionality 3×3.

[31] There are N vertices, each having x, y, z components.

13.3 Learned gradient descent

Learned gradient descent (LGD) [Song et al., 2020] is a method for fitting 3D human shapes to images by combining gradient-based optimization and neural networks. It leverages a neural network to predict the parameter update rule for each optimization iteration. This allows the algorithm to converge in few steps. See Algorithm 7 for the full algorithm.

Sample $\boldsymbol{\theta}, \boldsymbol{\beta}$ from data
Sample $\boldsymbol{R}, \boldsymbol{t}, s$ uniformly from feasible range
$\boldsymbol{X} \leftarrow \boldsymbol{W}\boldsymbol{M}(\boldsymbol{\theta}, \boldsymbol{\beta})$
$\boldsymbol{x} \leftarrow s\Pi(\boldsymbol{R}\boldsymbol{X}) + \boldsymbol{t}$
$\Theta_0 \leftarrow \{\boldsymbol{\theta}_0 = \boldsymbol{0}, \boldsymbol{\beta}_0 = \boldsymbol{0}, \boldsymbol{R}_0 = \boldsymbol{0}, \boldsymbol{t}_0 = \boldsymbol{0}, s_0 = 0\}$
for $n = 0, \ldots, N-1$ **do**
    $\boldsymbol{X}_n \leftarrow \boldsymbol{W}\boldsymbol{M}(\boldsymbol{\theta}_n, \boldsymbol{\beta}_n)$
    $\boldsymbol{x}_n \leftarrow s_n\Pi(\boldsymbol{R}_n\boldsymbol{X}_n) + \boldsymbol{t}_n$
    $\mathcal{L}(\Theta_n) \leftarrow L_{\text{reproj}}(\boldsymbol{x}_n, \boldsymbol{x})$
    $\Delta \leftarrow \mathcal{N}_{\boldsymbol{w}}\left(\frac{\partial \mathcal{L}(\Theta_n)}{\partial \Theta_n}, \Theta_n, \boldsymbol{x}\right)$
    $\Theta_{n+1} \leftarrow \Theta_n + \Delta$
end for

Algorithm 7. Learned gradient descent training scheme. $\mathcal{N}_{\boldsymbol{w}}$ is the neural network that predicts the update rule. $\boldsymbol{x}$ is a 2D projection of the 3D pose $\boldsymbol{X}$ onto the image. $\boldsymbol{M}(\cdot, \cdot)$ computes the body mesh using SMPL. $\boldsymbol{W}$ is a matrix that maps vertices to k joints of interest. The camera model is parametrized by the global rotation $\boldsymbol{R} \in \mathbb{R}^{3\times3}$.

References

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017.

Junyoung Chung, Kyle Kastner, Laurent Dinh, Kratarth Goel, Aaron C Courville, and Yoshua Bengio. A recurrent latent variable model for sequential data. *Advances in neural information processing systems*, 28, 2015.

Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

Laurent Dinh, David Krueger, and Yoshua Bengio. Nice: Non-linear independent components estimation. *arXiv preprint arXiv:1410.8516*, 2014.

Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. Density estimation using real nvp. *arXiv preprint arXiv:1605.08803*, 2016.

Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

Brendan J Frey. *Graphical models for machine learning and digital communication*. MIT press, 1998.

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. Made: Masked autoencoder for distribution estimation. In *International conference on machine learning*, pages 881–889. PMLR, 2015.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

Amos Gropp, Lior Yariv, Niv Haim, Matan Atzmon, and Yaron Lipman. Implicit geometric regularization for learning shapes. *arXiv preprint arXiv:2002.10099*, 2020.

Irina Higgins, Loic Matthey, Arka Pal, Christopher P Burgess, Xavier Glorot, Matthew M Botvinick, Shakir Mohamed, and Alexander Lerchner. beta-vae: Learning basic visual concepts with a constrained variational framework. *ICLR (Poster)*, 3, 2017.

Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance, 2022.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33: 6840–6851, 2020.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5): 359–366, 1989.

Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. 3d gaussian splatting for real-time radiance field rendering. *ACM Transactions on Graphics*, 42(4):1–14, 2023.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Durk P Kingma and Prafulla Dhariwal. Glow: Generative flow with invertible 1x1 convolutions. *Advances in neural information processing systems*, 31, 2018.

Mark A Kramer. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal*, 37(2):233–243, 1991.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.

Christoph Lassner and Michael Zollhofer. Pulsar: Efficient sphere-based neural rendering. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 1440–1449, 2021.

Stephen Lombardi, Tomas Simon, Gabriel Schwartz, Michael Zollhoefer, Yaser Sheikh, and Jason Saragih. Mixture of volumetric primitives for efficient neural rendering. *ACM Transactions on Graphics (ToG)*, 40(4): 1–13, 2021.

Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. Smpl: a skinned multi-person linear model. *ACM Trans. Graph.*, 34(6), oct 2015. ISSN 0730-0301. doi: 10.1145/2816795.2818013. URL `https://doi.org/10.1145/2816795.2818013`.

Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for gans do actually converge? In *International conference on machine learning*, pages 3481–3490. PMLR, 2018.

Lars Mescheder, Michael Oechsle, Michael Niemeyer, Sebastian Nowozin, and Andreas Geiger. Occupancy networks: Learning 3d reconstruction in function space. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 4460–4470, 2019.

Luke Metz, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. Unrolled generative adversarial networks, 2017.

Ben Mildenhall, Pratul P Srinivasan, Matthew Tancik, Jonathan T Barron, Ravi Ramamoorthi, and Ren Ng. Nerf: Representing scenes as neural radiance fields for view synthesis. *Communications of the ACM*, 65(1): 99–106, 2021.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

Michael Niemeyer, Lars Mescheder, Michael Oechsle, and Andreas Geiger. Differentiable volumetric rendering: Learning implicit 3d representations without 3d supervision. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 3504–3515, 2020.

Sebastian Nowozin, Botond Cseke, and Ryota Tomioka. f-gan: Training generative neural samplers using variational divergence minimization. *Advances in neural information processing systems*, 29, 2016.

Christopher Olah. Understanding lstm networks, 2015. URL `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

Jeong Joon Park, Peter Florence, Julian Straub, Richard Newcombe, and Steven Lovegrove. Deepsdf: Learning continuous signed distance functions for shape representation. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 165–174, 2019.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision, 2021.

Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

Evan Shelhamer, Jonathan Long, and Trevor Darrell. Fully convolutional networks for semantic segmentation, 2016.

Jie Song, Xu Chen, and Otmar Hilliges. Human body model fitting by learned gradient descent. In *European Conference on Computer Vision*, pages 744–760. Springer, 2020.

Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

Lucas Theis, Aäron van den Oord, and Matthias Bethge. A note on the evaluation of generative models, 2016.

Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1653–1660, 2014.

Benigno Uria, Marc-Alexandre Côté, Karol Gregor, Iain Murray, and Hugo Larochelle. Neural autoregressive distribution estimation. *Journal of Machine Learning Research*, 17(205):1–37, 2016.

Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. Wavenet: A generative model for raw audio, 2016.

Aaron Van den Oord, Nal Kalchbrenner, Lasse Espeholt, Oriol Vinyals, Alex Graves, et al. Conditional image generation with pixelcnn decoders. *Advances in neural information processing systems*, 29, 2016.

Aäron Van Den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks. In *International conference on machine learning*, pages 1747–1756. PMLR, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Shih-En Wei, Varun Ramakrishna, Takeo Kanade, and Yaser Sheikh. Convolutional pose machines. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pages 4724–4732, 2016.

Yi Yang and Deva Ramanan. Articulated pose estimation with flexible mixtures-of-parts. In *CVPR 2011*, pages 1385–1392. IEEE, 2011.

Fisher Yu and Vladlen Koltun. Multi-scale context aggregation by dilated convolutions, 2016.