Machine Perception

Paul He

June 4, 2024

Notes are based on Machine Perception course notes and lectures at ETH Zürich for the Spring 2024 semester. Some sections may be very similar to lecture notes and the original papers. In addition, I included new concepts introduced in the weekly worksheets that will be relevant for the exam. It also contains my derivation to the exercises. These are simply my notes used for studying the course. I do not guarantee the correctness of this set of notes, please feel free to point out if there are typos or mistakes.

Contents

Part I

Foundations of Deep Learning

1 Neural Networks

1.1 Multi-layer perceptron

Among the early models, we find the **Perceptron**. The classification of a single point is defined as follows

$$\hat{y} = f(\mathbf{x}, \mathbf{w}) = \mathbb{I}\{\mathbf{w}^\top \mathbf{x} + b > 0\} \tag{1}$$

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

$$\mathbf{w}_{k+1} \leftarrow \mathbf{w}_k + \eta \underbrace{(y_i - \hat{y}_i)}_{\text{residual}} \mathbf{x}_i, \tag{2}$$

where η is the learning rate. However, with a single perceptron it could not solve the XOR problem. This can be fixed by adding hidden layers.

$$\hat{y} = \sigma(\boldsymbol{W}_k \sigma(\boldsymbol{W}_{k-1} \cdots \sigma(\boldsymbol{W}_1 \mathbf{x}))). \tag{3}$$

Which becomes a multi-layer perceptron (MLP). We call the parameters $\boldsymbol{\theta} = \{\boldsymbol{W}_1, \ldots, \boldsymbol{W}_k, \mathbf{b}_1, \ldots, \mathbf{b}_k\}$ and estimate them using an optimization algorithm such as gradient descent, which we call is "learning" and we compute the gradient by backpropagation.

1.2 Loss Functions

In order to learn, we need to define a loss function that needs to be optimized. The first technique we are going to use to define a suitable loss function is Maximum Likelihood Estimation.

Suppose we are given a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i\}_{i=1}^N$ with inputs x_i, y_i and a parametric family of probability distributions $p_{\text{model}}(y|\mathbf{X}, \Theta)$ over the output space, indexed by Θ. The dataset is assumed to be drawn from an underlying probability distribution p_{data} which I'll refer to as p_d, which is *not* accessible to us except though the samples in $\mathcal{D}$, which we assume to be i.i.d.. I'll refer p_{model} as p_m.

When defining the loss function, we will always follow these three main steps:

1. Write down the parametric probability distribution of the mode $p_m(\mathbf{y}|\mathbf{X}, \Theta)$
2. Decompose that probability distribution into per sample probability $p_m(y_i|\mathbf{x}_i, \Theta)$
3. Convert everything in log scale and minimize the Negative Log Likelihood (NLLK)

More formally, the conditional maximum likelihood estimator for Θ is given by

$$\Theta^*_{\text{MLE}} = \text{argmax}\, p_m(\mathbf{y}|\mathbf{X}, \boldsymbol{\Theta}) \tag{4}$$

$$= \underset{\Theta}{\text{argmax}} \prod_{i=1}^{N} p_m(y_i|\mathbf{x}_i, \Theta) \tag{5}$$

$$= \underset{\Theta}{\text{argmax}} \sum_{i=1}^{N} \log p_m(y_i|\mathbf{x}_i, \Theta) \tag{6}$$

The negative log-likelihood (NLLK) is the loss function that we will use to optimize out networks. We now see some examples for different types of p_m.

Gaussian Let $X \sim \mathcal{N}(\mu, \sigma^2)$. Then, our model is a 1-dimensional Gaussian distribution and we can model our system using a Gaussian probability density function with μ and σ as our parameters.

In order to minimize NLLK, we can adjust the values of μ and σ such that the Gaussian curve places more probability mass on areas where we expect to see data and less probability mass on areas where we do not expect to see data.

Bernoulli Suppose we are given a dataset $\mathcal{D} = \{(\mathbf{x}_i, y_i\}_{i=1}^N$ with $y_i \in \{0, 1\}$ i.e. we are performing binary classification. Now, a Gaussian would not be a valid p_m anymore. Instead we will model the output variable $\hat{y}_i$ using a Bernoulli distribution $\hat{y}_i \sim \text{Bern}(\sigma(\boldsymbol{\theta}^\top \mathbf{x}_i))$ The parameter of this Bernoulli distribution derives from the model used for binary classification. The sigmoid can be interpreted as a probability distribution as its always positive and in the interval $(0, 1)$. Following the steps

$$p_m(\mathbf{y}|\mathbf{X}, \Theta) = \prod_{i=1}^{N} p(y_i|\mathbf{x}_i, \Theta) \tag{7}$$

$$= \prod_{i=1}^{N} \Big[\frac{1}{1 + \exp\{-\phi\}}\Big]^{y_i} \Big[1 - \frac{1}{1 + \exp\{-\phi\}}\Big]^{1-y_i} \tag{8}$$

where we denote $\phi = \Theta^\top \mathbf{x}_i$. Hence, the NLLK will be our loss function

$$\mathcal{L} = -\log p_m(\mathbf{y}|\mathbf{X}, \Theta) \tag{9}$$

$$= -\frac{1}{N}\sum_{i=1}^{N} y_i \log(\pi_i) + (1 - y_i)\log(1 - \pi_i) \tag{10}$$

1.3 Approximation capabilities of Neural Networks

A Multi-Layer Perceptron (MLP) that solely relies on linear activation functions is mathematically equivalent to a single unit network with a linear activation, which is insufficient for effectively learning numerous types of functions. Therefore, the inclusion of non-linear activation functions between layers is crucial to enable the network to perform effectively.

In particular, the approximation capabilities of an MLP with a non-linear activation function are given by the following theorem.

Universal Approximation Theorem. Let $\sigma : \mathbb{R} \mapsto \mathbb{R}$ 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 functions on I_m denoted by $C(I_m)$. Then, a continuous function $f \in C(I_m)$ can be approximated given any $\epsilon > 0$, $N \in \mathbb{Z}$, real constants $v_i, b_i \in \mathbb{R}$ and real vectors $\mathbf{w}_i \in \mathbb{R}^m$. $\forall i \in \{1, \ldots, N\}$ we have

$$f(\mathbf{x}) \approx g(\mathbf{x}) = \sum_{i=1}^{N} v_i \sigma(\mathbf{w}_i^\top \mathbf{x} + b_i) \tag{11}$$

and $|g(\mathbf{x}) - f(\mathbf{x})| < \epsilon$, $\forall \mathbf{x} \in I_m$

In easy terms, the theorem says that: *A feed-forward neural network with a single hidden layer and continuous non-linear activation function can approximate any continuous function with arbitrary precision.*

2 *Training Neural Networks*

2.1 Regularization

Regularization is any modification to a learning algorithm intended to improve its generalization error, but not its training error.

In the idea scenario, model family being trained includes the data generating process, but also many other possible generating processes. Regularization pushes or restricts the solution space towards the true generating process.

Parameter Norm Penalties The idea is to add a penalty term to the loss function that depends on the weights of the network, usually specific norm of the weights. Results depend on the norm, different norms will lead to different preferences.

$$\tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta; X, y) + \lambda\Omega(\theta) \tag{12}$$

Usually the penalty term does not include the bias parameters.

L2 Regularization
We set $\Omega(\theta) = \frac{1}{2}\|\theta\|_2^2$ such that

$$\nabla\tilde{\mathcal{L}}(\theta) = \nabla\mathcal{L}(\theta) + \lambda\theta \tag{13}$$

and we get

$$\theta \leftarrow \theta - \alpha\nabla\tilde{\mathcal{L}}(\theta) = \underbrace{(1-\alpha\lambda)}_{\text{weight decay}}\ \theta - \underbrace{\alpha\nabla\mathcal{L}(\theta)}_{\text{Parameter Update}} \tag{14}$$

L2 regularization tends to shrink the coefficients towards zero but rarely sets them exactly to zero. This results in models where all features are retained but with smaller weights.

L1 Regularization
We set $\Omega(\theta) = \|\theta\|_1 = \sum_i |\theta_i|$ so that

$$\nabla\tilde{\mathcal{L}}(\theta) = \nabla\mathcal{L}(\theta) + \lambda\text{sign}(\theta) \tag{15}$$

L1 prefers sparse solutions, it is less used than L2 regularization in Neural Networks. L1 regularization can shrink some coefficients to exactly zero, effectively performing feature selection by excluding some features entirely.

Penalties can cause the optimization to get stuck in local minima with small weight values. From a Maximum a Posteriori (MAP) Bayesian

perspective, it is equivalent to specifying a prior distribution on the weights' values. It is also equivalent to imposing (implicit) constraints on the weights i.e. restricting the solution space.

2.2 *Ensemble Methods*

Ensemble methods use finite amount of different machine learning models to obtain better performance than any one of them alone.

We can train different model classes (Linear Regression, Decision Tree, Neural Network) on the same data and then aggregate the predictions.

We can also train the same model class on different data (sampled from the original dataset) and aggregate the predictions.

Bagging

1. We first create k different bootstraps from the training set with replacement.
2. For each bootstrap we train a classifier, which gives us k different classifiers.
3. Our final prediction is the combination of the k different classifiers.

Dropout

Dropout is a computationally inexpensive technique to regularize a broad family of models. In practice, it consists of ignoring a subset of neurons (chosen at random) during each training iteration. It is regarded as a ensemble technique because it is equivalent to creating an ensemble of all sub-networks that can be formed by removing non-output units from an underlying base unit.

At training stage, if we let y_l be the input to the (l+1)th layer in the network, f and activation function and θ_l, b_l be the weights and bias parameters. Then a feed-forward with dropout is first sampling $r_j^{(l)} \sim \text{Bern}(p)$, then compute $\tilde{y}_l = r^{[l]} \odot y_l$ which leads to $z_{l+1} = \theta_{l+1}\tilde{y}_l + b$ and $y_{l+1} = f(z_{l+1})$.

At the test stage, we can:

1. Approximate with sampling i.e. do limited number of forward passes in the network where dropout behaves same as at training time and take the average of the result.
2. (Preferred) **weight scaling inference rule**. The idea is the expected total input to any unit at test time equls the expected

total input at training time, which approximates the true geometric mean of the ensemble. Exactly on for model with non-linearities, but works well in practice.

2.3 *Data Normalization*

Scales matter:

- Scale in the input: large input values could result in large weight values which make the predictions unstable.
- Scale in the output: a target variable with a large spread in its values can make the training process unstable.

Let X be our input data and $x^{(i)}$ be a particular data point. Then, the mean and the standard deviation of the data are $\mu = \frac{1}{n}\sum_{i=1}^{n} x^{(i)}$ and $\sigma = \frac{1}{n-1}\sum_{i=1}^{n}(x^{(i)} - \mu)^2$. So the normalized is $X_N = \frac{X-\mu}{\sigma}$. **During testing, use the same mean and standard deviation**.

Batch Normalization Given some mini-batch of intermediate activations $z_1, z_2, \ldots z_n$, we can compute $\mu = \frac{1}{n}\sum_{i=1}^{n} z_i$, $\sigma^2 = \frac{1}{n-1}\sum_{i=1}^{n}(z_i - \mu)^2$ and $z_i^{\text{norm}} = \frac{z_i - \mu}{\sqrt{\epsilon+\sigma^2}}$ and hence $\tilde{z}_i = \gamma z_i^{\text{norm}} + \beta$

The bias term in a linear (and convolutional) layer becomes redundant if you use batch normalization after it. Batch normalization makes the weights in deeper layers more robust to changes to weights in the shallower layers of the network. Each mini-batch is scaled by the mean/variance computed on just that mini batch. This adds some inherent noise within that mini-batch (similar to dropout) and has a slight regularization effect.

During test time, μ, σ^2 are estimated using exponentially weighted average across mini-batches i.e. keep running average during training time, and use that during test time.

Batch Normalization tries to solve **internal covariate shift:**

- Training deep neural networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers
- The gradient tells us how to update each parameter, under the assumption that other layers do not change. In practice, we update all of the layers simultaneously.

However, recently research has shown it my not even be reducing internal covariate shift, instead, it fundamentally impacts the training process by making the optimization landscape significantly smoother,

thus leading to a more predictive and stable behaviour of the gradients.

2.4 Optimization Algorithms

Keep in mind, optimization and learning are two different problems. In learning, we optimize an objective function J in the hope that doing so will improve some performance measure P that is typically inaccessible.

Gradient Descent $\theta = \theta - \eta \cdot \nabla_\theta J(\theta)$ Follow the direction of the slop of the surface created by the objective function downhill. The learning rate η determines the size of the steps taken.

Stochastic Gradient Descent (SGD) $\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$ Unbiased estimate of the true gradient, high variance, high efficiency per iterations (n times cheaper than GD). Enables jumping to new and potentially better local minima. Risk of overshooting.

Near a smoothened minimum, the SGD step is dominated by stochastic fluctuations. In practice, it is necessary to decrease the learning rate of time for SGD to converge.

Mini-batch Gradient Descent Sample a minibatch of m examples from the training set. Reduces the variance of the gradient estimate which leads to more stable convergence, and allows parallelization over up to m processors.

Polyak's Momentum Accelerate gradient in the face of: high curvature (poor conditioning of Hessian matrix), small but consistent gradients, noisy gradients. $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)}) \right)$ and update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$. The velocity term $\boldsymbol{v}$ accumulates successive gradient elements. The larger α is relative to ε, the more previous gradients affect the current direction. The size of the step depends on how large and how aligned a sequence of the gradient is. The step size is largest when many successive gradients point in the same direction. However, Polyak's momentum has been proved to *not* converge in the very simple case of a strongly-convex and smooth function for carefully chosen α and ε.

Nesterov's Momentum Fixes Polyak's momentum's limitation by adding a correlation factor. Now, $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \varepsilon \nabla_\theta \left(\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)}) \right)$ and update $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

3 Convolutional Neural Network

Many tasks in the field of computer vision can be modelled with Convolutional Neural Network (CNN). *Classification, Classification and Localization, Object Detection, Instance Segmentation, 3D Body Pose Estimation, Eye Gaze Estimation, Dynamic Gesture Recognition*.

HMAX Model The HMAX model is a biologically motivated architecture for computer vision that incorporates these neuroscientific insights. It aligns closely with physiological evidence, particularly in terms of existence and operation of simple (S) and complex (C) cells at different levels of the visual hierarchy. S cells are tuned to specific stimuli and typically have small receptive fields. Given an input x, the response y of a simple cell is computed as

$$y = \exp\left(-\frac{1}{2\sigma^2} \sum_{j=1}^{n_{s_k}} (w_j - x)^2 \right) \tag{16}$$

C cells combine the outputs from multiple simple cells to increase invariance and receptive field size. The output of a complex cell is

$$y = \max_{j=1,\dots n_{c_k}} (x_j) \tag{17}$$

Research has shown that through many iterations of these operations, complex objects can be constructed from low-level features.

3.1 Convolution Operation

Convolutions as linear, shift-equivariant transforms In deep neural networks, our goal is to transform a given input signal f into a more informative representation using an operator T. Among the various operators, convolutions can express any linear, shift equivariant transform.

A transform T is **linear** if:

$$T(\alpha\mathbf{u} + \beta\mathbf{v}) = T(\alpha\mathbf{u}) + T(\beta\mathbf{v}) \tag{18}$$

A transform T is **invariant** to f if:

$$T(f(u)) = T(u) \tag{19}$$

Invariance is a property we want to exploit in classification tasks. If we want to classify cats in an image and we shift every pixel by one unit. The image should still represent a cat. In other words, the classifier f should be invariant to the shift T of one pixel.

A transform T is **equivariant** to f if:

$$T(f(u)) = f(T(u)) \tag{20}$$

We desire equivariance for tasks such as edge detection. When there is an edge present in the input image and we apply the function f to shift the image content, we want the edge detector to also shift its response (the position of the edge) along with the function f.

Linear Filters to Convolution Linear operations can be represented in the form of

$$I'(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} K(m,n) I(i+m, j+n) \tag{21}$$

where I represents the input image, I' is the output of the operation, K is the kernel of the operation, and $(2k+1) \times (2k+1)$ represents the dimension of the filter. Note that K does not depend on (i,j), the position of the current pixel image.

In a linear transform, the value of the kernel K, which is applied to each point of the image depends on both the position and neighboring point (m,n) w.r.t (i,j). This dependency on the specific position (i,j) makes the linear transform non-shift-invariant. To achieve shift invariance, we need to remove the dependency on the position (i,j), which can be done by considering kernels that are constant over the image. From now on, the kernel K will represent a constant matrix $K(m,n)$.

Correlation Correlation is a particular case of shift-invariant linear filtering. In correlation, a fixed spatial pattern is shifted over the image, and the response is recorded as the pattern is applied to different patches. The response is computed by multiplying the pattern with the under-lighted portion of the image. If the elements are similar, the outputs will be high, whereas dissimilar elements will yield low outputs.

The ability to perform pattern matching[1] makes the correlation operation particularly useful in object detection.

[1] finding a pattern when the correlation between the kernel and input pixel is high [object detection]

For instance, given a 3×3 kernel K and an input image I, the output of the correlation operation for each cell (i,j) can be computed as:

$$\begin{aligned} I'(i,j) = {} & c_{11} I(i-1, j-1) + c_{12} I(i-1, j) \\ & + c_{13}(i-1, j+1) + c_{21} I(i, j-1) + c_{22} I(i,j) + c_{23} I(i, j-1) \\ & + c_{31} I(i+1, j-1) + c_{32} I(i+1, j) + c_{33} I(i+1, j+1) \end{aligned} \tag{22}$$

where (i, j) is the center of the image I, and $(i-1, j-1)$ is the top left corner of the 3×3 patch of the image.

So in general, for any $2k \times 2k$ kernel, the correlation operation can be described as:

$$I'(i,j) = \sum_{m=-k}^{k} \sum_{n=-k}^{k} K(m,n) I(i+m, j+n) \tag{23}$$

Convolution vs. Correlation

For any $2k \times 2k$ kernel, the correlation operation can be described as:

$$I'(i,j) = (I * K)(i,j) \sum_{m=-k}^{k} \sum_{n=-k}^{k} I(i-m, j-n) K(m,n) \tag{24}$$

where $*$ denotes the convolution operation. This is basically correlation but with the kernel flipped. However, the difference is that **convolution is commutative**, while correlation is not. This means the above equation is equivalent to

$$I'(i,j) = (K * I)(i,j) \sum_{m=-k}^{k} \sum_{n=-k}^{k} K(m,n) I(i-m, j-n) \tag{25}$$

This formula is preferred for implementation in ML libraries such as PyTorch as it allows for a smaller variation in the range of valid (m, n) values. The commutative property is the primary reason it was commonly used instead of correlation. Convolution and correlation become equivalent when $K(m,n) = K(-m,-n)$.

Discrete convolution as matrix multiplication

$$I * K = \begin{bmatrix} k_1 & 0 & \dots & 0 \\ k_2 & k_1 & & \\ k_3 & k_2 & & \vdots \\ \vdots & k_3 & & \\ 0 & \vdots & \dots & k_m \end{bmatrix} \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ \vdots \\ I_n \end{bmatrix} \tag{26}$$

3.2 *Convolutonal Neural Network*

A CNN is composed of a sequence of convolutional layers interspersed with activation function and pooling layers, followed by a final dense layer (also called fully connected). The dense layer aggregates the features extracted by the convolutional layers and produces the final output of the network.

Convolution Layer In CNNs, the first step of a convolutional layer involves applying convolution to an input image. This is achieved by convolving a kernel[2] with the entire image. In practice, this involves sliding the kernel over the image spatially and computing dot products.

[2] referred to as a filter in deep learning

Note that filters must extend the full depth of the input volume. For example, if we have an RGB image (3 colour channels), we would apply a $k \times k \times 3$ filter to it.

When taking the dot product between the filter and a small $5 \times 5 \times 3$ chunk of the image, resulting in a 75-dimensional dot product + bias, the output is a single number.

Forward Pass

Let $z_{i,j}^{[l-1]}$ be the output of the $(l-1)$-th layer at position (i,j), let $w_{n,m}^{[l]}$ be the weight of the filter in the position (m,n), and let b be the bias parameter (every convolutional layer has one bias parameter per filter). We can write the output of the l-th layer as (ignoring activation functions for now):

$$z_{i,j}^{[l]} = W^{[l]} \cdot z^{[l-1]} + b = \sum_m \sum_n w^{[l]}(m,n) z^{[l-1]}(i-m, j-n) + b \quad (27)$$

Backward Pass

$$\delta_{i,j}^{[l-1]} = \frac{\partial \mathcal{L}}{\partial z_{i,j}^{[l-1]}} \quad (28)$$

$$= \sum_{i'} \sum_{j'} \frac{\partial \mathcal{L}}{\partial z_{i',j'}^{[l]}} \frac{\partial z_{i',j'}^{[l]}}{\partial z_{i,j}^{[l-1]}} \quad (29)$$

$$= \sum_{i'} \sum_{j'} \delta_{i',j'}^{[l]} \frac{\partial \sum_m \sum_n w^{[l]}(m,n) z^{[l-1]}(i'-m, j'-n) + b}{\partial z_{i,j}^{[l-1]}} \quad (30)$$

$$= \sum_{i'} \sum_{j'} \delta_{i',j'}^{[l]} w^{[l]}(m,n) \quad (31)$$

$$= \delta^{[l]} * \mathrm{ROT}_{180}(W^{[l]}) \quad (32)$$

Notice this is just using the definition of forward pass and on the last step the double sum only has a non-zero gradient w.r.t. $z_{i,j}^{[l-1]}$ when $i' - m = i$ and $j' - n = j$.

Parameter Update
Is derived in a similar way as the backward pass

$$\frac{\partial \mathcal{L}}{\partial w^{[l]}(m,n)} = \sum_i \sum_j j \frac{\partial \mathcal{L}}{\partial z_{i,j}^{[l]}} \frac{\partial z_{i,j}^{[l]}}{\partial w_{m,n}^{[l]}} \tag{33}$$

$$= \sum_i \sum_j \delta_{i,j}^{[l]} \frac{\partial z_{i,j}^{[l]}}{\partial w_{m,n}^{[l]}} \tag{34}$$

$$= \sum_i \sum_j \delta_{i,j}^{[l]} \frac{\partial \sum_m \sum_n w^{[l]}(m,n) z^{[l-1]}(i-m, j-n) + b}{\partial w_{m,n}^{[l]}} \tag{35}$$

$$= \sum_{i'} \sum_{j'} \delta_{i',j'}^{[l]} z^{[l-1]}(i-m, j-n) \tag{36}$$

$$= \delta^{[l]} * \mathrm{ROT}_{180}(Z^{[l-1]}) \tag{37}$$

Pooling Layer The pooling layer substitutes the output of the network at a specific position with a condensed representation of the adjacent outputs, typically in the form of a statistical summary. This operation reduces the size of the representations and makes them more manageable. Note that the pooling operation is applied independently to each activation map.

One of the possible pooling operations is **max pooling**, which simply outputs the maximum value from the input within the given region. The **forward** pass is

$$z_{i,j}^{[l]} = \max \left\{ z_i^{[l-1]} \right\} \tag{38}$$

Hence, the **backward pass** is

$$\delta^{[l-1]} = \{\delta^{[l]}\}_{i^*,j^*} \tag{39}$$

since

$$\frac{\partial z^{(l)}}{\partial z_i^{(l-1)}} = \mathbb{I}\{i = i^*\} \tag{40}$$

where (i^*, j^*) correspond to the pixel with the maximum value. Note that the max-polling layer has no learnable parameter, its just a propagation of the error and it is not used for weight update.

Dense Layer The dense layer, also referred to as a fully connected layer, complements the role of convolutional and pooling layers in capturing local features and reducing spatial dimensions.

Its crucial function lies in aggregating the extracted features and generating the final output of the network. In this layer, each neuron

performs a weighted sum of all its inputs and applies a non-linear activation function. By learning complex relationships among the features extracted by preceding layers, the dense layer enables the network to make predictions based on these learned representations.

3.3 Summary of Parameters and Dimensions

Let I be the length of the input length, F be the filter length, P be the amount of zero padding, S be the striding and the output O is:

$$O = \frac{I - F + P_{\text{START}} + P_{\text{END}}}{S} + 1 \tag{41}$$

usually we round down. Usually $P_{\text{START}} = P_{\text{END}} \triangleq P$.

	Input Dim	Output Dim	Parameters
Convolution	$I \times I \times C$	$O \times O \times K$	$(F \times F \times C + 1) \times K$
Pooling	$I \times I \times C$	$O \times O \times C$	ZERO
FCNN	N_{in}	N_{out}	$(N_{\text{in}} + 1) \times N_{\text{out}}$

Table 1. Summary of Operations

3.4 Fully Convolutional Neural Network

Semantic Segmentation Semantic Segmentation is a critical task in computer vision that involves assigning a semantic class to each pixel in an image.[3]

[3] Traditional image classification outputs a single class for the entire image, semantic segmentation classify each pixel individually.

The easiest approach is to classify each pixel individually by extracting features from a patch centered on it. However, this method is inefficient and redundant for processing large images. Instead, practitioners adopt a pipeline that involves using the entire image as input to a CNN. **The final fully connected layer, typically used for image classification, is removed**, and the resulting feature maps are used as segmentation predictions. Due to convolutions and max-pool operations, these predictions have lower resolution than the original image. To obtain the same resolution as the input image, we could keep the same dimensions by using appropriate padding in the convolutions and avoiding pooling layers. However, this method can be computationally expensive.

In practice, the most common approach is to downsample features obtained using convolution and pooling layers and upsample them again. By applying convolution to a smaller object, this method is more computationally efficient while producing output with the same resolution as the input. Downsampling can be achieved with pooling and strided convolution, and we will explore the various techniques for upsampling.

Fixed Upsampling techniques Nearest neighbour, bed of nails and max unpooling.

- Nearest neighbour upsampling involves upsampling features by copying the same value into all corresponding pixels at a higher resolution.
- Best of nails upsampling involves padding zero to the neighbour values.
- Max unpooling uses zero padding as in bed of nails, but it 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.

Learnable Upsampling An example is **transposed convolutions**, they make use of learning.

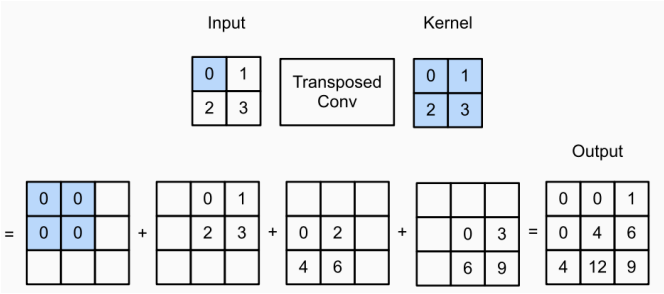

Figure 1. Operation of the transposed convolution on a 2×2 input and kernel.

In practice, given a low-resolution image, we learn a kernel (e.g., 2×2) that is used to produce all the terms whose sum will be the final output. Each term is obtained by multiplying all the elements of the kernel by the 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. Note that each term of the sum is a sparse matrix, potentially with non-zero terms only in a number of pixels equal to the kernel size. Figure 1 provides a visualization of this process.

U-Net U-Net is a popular fully convolutional neural network (FCNN) architecture that has been widely used for semantic segmentation tasks. The main idea behind U-Net is to combine global and local feature maps by copying corresponding tensors from earlier stages in each upsampling stage. This allows the network to capture both local[4] and global context, leading to more accurate semantic segmentation results.

[4] Residual connections help to mantain local features as images are not completely downsampled at every stage.

4 Recurrent Neural Network

Recurrent Neural Networks (RNNs) are a type of neural network that can process sequential data. Unlike traditional feedforward neural networks, which take fixed-length inputs, RNNs can take inputs of variable length and maintain an internal memory of the past inputs that it has seen. At their core, RNNs are a type of dynamical system which is a mathematical concept to describe the behavior of a system over time based on its current state. It context of RNNs, the hidden state of each timestep can be thought of as representation of the current state of the system and the transition function that updates the state can be thought of as the set of rules that govern the behavior of the system over time.

4.1 Vanilla Recurrent Neural Network

The vanilla version of a Recurrent Neural Network is characterized by a single hidden vector dented as $\mathbf{h}_t$, which forms the state of the network.

$$\hat{y} = W_{hy}\mathbf{h}_t \tag{42}$$

$$\mathbf{h}_t = \tanh(W_{hh}\mathbf{h}_{t-1} + W_{xh}\mathbf{x}_t) \tag{43}$$

where $\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \Theta)$ represents the hidden state at time step t.

Compared with an MLP, we notice serval important differences.

- RNN employs the tanh function instead of sigmoid. It is preferred as it is centered in 0 and the norm of its gradients are higher than the norms of the sigmoid gradients which allows for faster convergence.
- The layer at timestep depends on **both** the previous hidden state $\mathbf{h}_{t-1}$ and the input $\mathbf{x}_t$ of that timestep. Importantly, the weights W_{hy}, W_{hh}, W_{xh} are shared among timesteps.[5]

[5] It means that the same weight matrices are used at each time step.

4.2 Backpropagation Through Time

First lets define the loss $\mathcal{L}_t = \|\hat{\mathbf{y}}_t - \mathbf{y}_t\|^2$ so $\mathcal{L} = \sum_t \mathcal{L}_t$. Then, we derive the gradients for each term one by one.

$$\frac{\partial \mathcal{L}}{\partial W} = \sum_{t=1}^{S} \frac{\partial \mathcal{L}_t}{\partial W} \tag{44}$$

To perform this computation, it is crucial to view the unrolled recurrent model as a multi-layer network with a potentially infinite number of layers. We can then apply backpropagation to efficiently compute gradients in this extended network structure. For each timestep t we have

$$\frac{\partial \mathcal{L}_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial \mathcal{L}_t}{\partial \hat{\mathbf{y}}_t} \frac{\partial \hat{\mathbf{y}}_t}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} \frac{\partial^+ \mathbf{h}_k}{\partial W} \tag{45}$$

where ∂^+ denotes the immediate derivative, which in our cases treats $\mathbf{h}_{k-1}$ constant w.r.t. the weight W. For simplicity, denote $W_{hh} = W$ and $W_{xh} = U$ and ignore the tanh for now. Lets first consider the term $\frac{\partial \mathbf{h}_t}{\partial W}$. This may be complicated at first because $\mathbf{h}_t$ depends on W and on $\mathbf{h}_{t-1}$ it depends on W again. Notice that the partial derivatives cancel out in the middle resulting in $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$ directly.

$$\frac{\partial \mathbf{h}_t}{\partial W} = \frac{\partial W\mathbf{h}_{t-1} + U\mathbf{x}_t}{\partial W} \tag{46}$$

$$= \frac{\partial}{\partial W}\Big[\mathbf{h}_t(\mathbf{h}_{t-1}(\mathbf{h}_{t-2}(\dots \mathbf{h}_1(W)))\Big] \tag{47}$$

$$= \frac{\partial^+ \mathbf{h}_t}{\partial W} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial^+ \mathbf{h}_{t-1}}{\partial W} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}}\frac{\partial^+ \mathbf{h}_{t-2}}{\partial W} + \dots \tag{48}$$

$$= \frac{\partial^+ \mathbf{h}_t}{\partial W} + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial^+ \mathbf{h}_{t-2}}{\partial W} + \dots + \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_1}\frac{\partial^+ \mathbf{h}_1}{\partial W} \tag{49}$$

$$= \sum_{k=1}^{t} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}\frac{\partial^+ \mathbf{h}_k}{\partial W} \tag{50}$$

We reintroduce tanh so $\mathbf{h}_t = \tanh(W_{hh}\mathbf{h}_{t-1} + W_{xh}\mathbf{x}_t)$. Let $\mathbf{a}_t = W_{hh}\mathbf{h}_{t-1} + W_{xh}\mathbf{x}_t$ then

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{a}_t} = \operatorname{diag}(1 - \tanh^2(\mathbf{a}_t)) = \operatorname{diag}(\mathbf{I} - \mathbf{h}_t^2) \tag{51}$$

Now lets compute $\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k}$ for $t > k$

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_k} = \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \mathbf{h}_{t-2}} \cdots \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \tag{52}$$

$$= \prod_{i=k+1}^{k} \frac{\partial \mathbf{h}_i}{\mathbf{h}_{i-1}} \tag{53}$$

$$= \prod_{i=k+1}^{t} \Big[\operatorname{diag}(\mathbf{I} - \mathbf{h}_i^2)W\Big] \tag{54}$$

That's a lot of multiplication! Which leads to scaling issues with the diagonal term. This leads to the vanishing and exploding gradient problem in the vanilla RNN.

Assuming the existence of an eigenvalue decomposition of the weight matrix W_{hh} (i.e. W_{hh} is symmetric), we can alternatively express $W_{hh} = Q\Lambda Q^\top$ where Λ is a diagonal matrix containing the eigenvalues of W_{hh} along its diagonal. Rearranging the previous matrix we obtain

$$(W_{hh}^\top)^{t-k-1} = (Q^\top \Lambda Q)^{t-k-1} = Q^\top \Lambda^{t-k-1} Q \tag{55}$$

where the last step is due to the fact that $QQ^\top = I$ as Q is orthogonal. If we consider f to be a sigmoid or tanh which are both upper bounded by 1, we can say there exists a $\gamma \in R$ such that

$$\|\operatorname{diag}(f'(\mathbf{h}_{i-1}))\| < \gamma \tag{56}$$

We show that the gradients could indeed vanish or explode, which will be problematic during training.

We want to show if $\lambda_1 < \frac{1}{\gamma}$ then as $t \to \infty$ the gradient **vanishes** and if $\lambda_1 > \frac{1}{\gamma}$ then as $t \to \infty$ the gradient **explodes**.

Proof. Let λ_1 be the largest singular value of the matrix W_{hh}, we first consider when $\lambda_1 < \frac{1}{\gamma}$. Then for all i we have

$$\left\| \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i-1}} \right\| \leq \|W_{hh}^\top\| \|\text{diag}(f'(\mathbf{h}_{i-1}))\| < \frac{1}{\gamma}\gamma = 1 \tag{57}$$

where $\|\cdot\|$ denotes the spectral norm. Let $\eta \in \mathbb{R}$ be such that for all i, $\left\| \frac{\partial \mathbf{h}_i}{\partial \mathbf{h}_{i=1}} \right\| \leq \eta < 1$. By induction over i we have

$$\left\| \prod_{i=k+1}^{t} \frac{\partial \mathbf{h}_1}{\partial \mathbf{h}_{i=1}} \right\| < \eta^{t-k} \to 0 \text{ as } t \to \infty \tag{58}$$

□

A simple solution is simply truncate the gradients after certain timesteps. Via **gradient clipping** we can ensure the gradient remains in a certain threshold (solves exploding gradient). We now introduce two new RNN architectures that aims to solve vanish gradients.

4.3 LSTM

Long Short Term Memory networks (LSTM) are a special kind of RNN that are designed to stabilize training by mitigating the vanishing gradient problem. The cell of a LSTM consists of four layer. In particular, these layers, also called as gates have the following four functions.

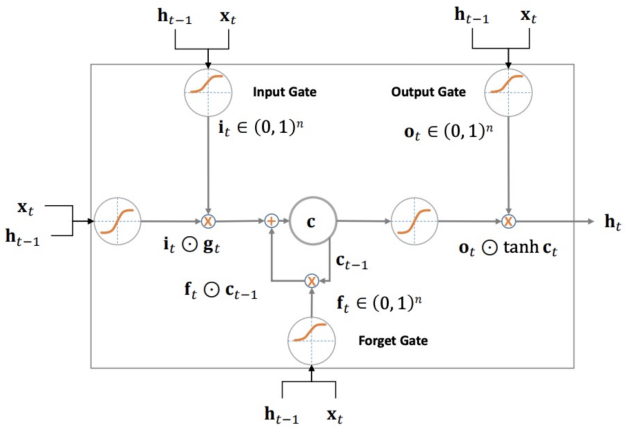

Figure 2. LSTM's gate structure

- f is the **forget gate** and has the role of scaling the old cell state $\mathbf{h}_{t-1}$. Depending on $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$, it decides which information should be forgotten from the previous cell state. The output of it is a sigmoided value, which for each element of the previous cell state $\mathbf{c}_{t-1}$ decides how much of the old state is kept in the current one. (0 delete, 1 keep)
- i is the **input gate** and has the role of deciding which values of the state cell should be updated in the current timestep. Its output is a sigmoided value, which for each element of the cell state decides how much of it should be written in the current cell state $\mathbf{c}_t$. (0 deletes everything, 1 keeps everything)
- o is the **output gate** and has the role of decidng which values of the current cell state should be put in the output of the cell $\mathbf{h}_t$. Like the previous gates, its output is a sigmoided value, which for each element of the current cell state $\mathbf{x}_t$, decides how much of it should flow into the output.
- g is the **gate** that decides what to write in the cell state. It is a tanh layer, which creates a vector of new candidate values.

In practice, the vectors $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$ are stacked and then multiplied with a big weight matrix in order to obtain the four different values $\mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g}$ with the roles described before.

Given the cell state $\mathbf{c}_t$ and the input $\mathbf{x}_t$ and outputs $\mathbf{h}_{t-1}$ have dimensionality n and given $W \in \mathbb{R}^{4n \times 2n}$, we have

$$\begin{bmatrix} \mathbf{i} \\ \mathbf{f} \\ \mathbf{o} \\ \mathbf{g} \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \begin{bmatrix} \mathbf{x}_t \mathbf{h}_{t-1} \end{bmatrix} \tag{59}$$

where σ denotes the sigmoid function. So we can rewrite W as

$$W = \begin{bmatrix} W_{xi} & W_{ci} \\ W_{xf} & W_{cf} \\ W_{xo} & W_{co} \\ W_{xg} & W_{cg} \end{bmatrix} \tag{60}$$

Moreover, in a multi-layer architecture, we can see $\mathbf{x}_t$ as the output of the layer before $\mathbf{h}_t^{[l-1]}$ and we can alternatively write

$$\begin{bmatrix} \mathbf{i} \\ \mathbf{f} \\ \mathbf{o} \\ \mathbf{g} \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W^{[l]} \begin{bmatrix} \mathbf{h}_t^{[l]} \mathbf{h}_{t-1}^{[l]} \end{bmatrix} \tag{61}$$

Recall that in the case of RNN the equation for $\mathbf{h}_t$ was

$$\mathbf{h}_t^{[l]} = \tanh W^{[l]} \begin{bmatrix} \mathbf{h}_t^{[l]} \mathbf{h}_{t-1}^{[l]} \end{bmatrix} \tag{62}$$

where $\mathbf{h} \in \mathbb{R}^n$ and $W \in \mathbb{R}^{n \times 2n}$. Once computed the values for $\mathbf{i}, \mathbf{f}, \mathbf{o}, \mathbf{g}$, we can compute the new cell state $\mathbf{c}_t$ and the new output $\mathbf{h}_t$ as

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \mathbf{g}_t \tag{63}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \tag{64}$$

In vanilla RNNs, the gradient flow relies on matrix multiplication as seen in $\mathbf{h}_t = \tanh(W_{hh}\mathbf{h}_{t-1} + W_{xh}\mathbf{x}_t)$ where the weight matrix W remains constant throughout. However, in LSTMs, the gradient flow takes a different approach. As a matter of fact, the $+$ operator allows the gradient to directly propagate to the element-wise multiplication $\mathbf{c}^{t-1} \odot \mathbf{f}$.

Part II

Generative Models

5 *Autoencoders*

In the field of learning, data is generally represented as measurement vectors, denoted as $\mathbf{x} \in \mathbb{R}^n$. If we select the features carefully, the dimension of these vectors can be low. Modern machine learning applications however involve in high-dimensional data (images, audio, or time-series). In such cases, one crucial objective is to find low-dimensional representations that can effectively compress the data while preserving its essential information. These **representations should be intepretable** and **capable of capturing different modes of variation**.

Autoencoders offer a solution through the use of an encoder-decoder structure. It operates under the assumption that a **compressed but meaningful** representation of the data can be obtained if the decoder is capable of reconstructing the original input solely from that compressed representation [6].

[6] In general it means the intermediate space where the data is projected to. Other literature might refer a *latent space* as *code* or *embedding space*

- The **encoder** f projects the original input space X into a latent space Z.
- The **decoder** g maps samples from the latent space Z back to the input space X.

The objective of the composition $[g \circ f]$ is there for to approximate the identity function on the data for a **low reconstruction error**.

5.1 *Linear Autoencoders: the PCA projection*

Restricting f and g to be linear. Then, the encoder function f of the autoencoder becomes equivalent to the projection performed by Principal Component Analysis (PCA) projection, which is the projection achieving the lost reconstruction loss $\mathcal{L}$.

Given N data points, we have

$$\mathcal{L} = \sum_{n=1}^{N} \|\mathbf{x}_n - \hat{\mathbf{x}}_n\|^2 = \sum_{n=1}^{N} \|\mathbf{x}_n - g(f(\mathbf{x}_n))\|^2 \tag{65}$$

They can be found in a closed form. However, they are not too

powerful.

5.2 Non-Linear Autoencoders

We remove the restriction on f and g to be linear such that the autoencoder becomes a non-linear projection of the data. This means both the encoder and decoder are implemented as neural networks.

To construct such autoencoder, we typically use a feedforward neural network trained to reconstruct its inputs. It optimizes the following objective function w.r.t. the encoder parameters Θ_f, Θ_g.

$$\hat{\Theta}_f, \hat{\Theta}_g = \underset{\hat{\Theta}_f, \hat{\Theta}_g}{\operatorname{argmin}} \sum_{n=1}^{N} \|\mathbf{x}_n - g(f(\mathbf{x}_n))\|^2 \tag{66}$$

In general, $\dim(X) > \dim(Z)$ (*undercomplete hidden representation*). The idea is that Z enables 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. They work well in practice to extract those featurers in training samples, however, it may not generalize effectively to out-of-distribution samples.

There are cases when $\dim(Z) > \dim(X)$ (*overcomplete hidden representation*), this lack of compression potentially allows each hidden unit to simply copy different input components, achieving a perfecting reconstruction loss without extracting any meaningful features. There are two main uses of autoencoders with overcomplete hidden representations: **Denoising** and *Inpainting autoencoders*.

The goal of **Denoising Autoencoders** is reconstructing the original clean image given a noisy image.

1. During training, a clean image is intentionally corrupted by injecting noise (such as Gaussian noise)
2. The noisy image is then provided as input to an Autoencoder with an overcomplete hidden representation.
3. The loss is evaluated based on a comparison with the original (clean) image, so the network is discouraged from simply copying the noisy image.
4. Hence, the network must learn the necessary transformations to remove noise and accurately restore the clean information.

The idea of *inpainting autoencoders* is similar to above.

5.3 *Autoencoder Limitations*

In order to generate new samples, it is desirable for the latent space to exhibit a well-structured nature, **characterized by continuity and interpolation capabilities**.

However, the decoder of classical version of autoencoders struggles to generate high-quality samples. This limitation arises due to the lack of continuity in the latent space. In regions of the latent space where there are discontinuities or gaps between clusters, the decoder has no knowledge and was not exposed to encoded vectors from those regions. They excel at reconstructing input data but face challenges with new samples.

5.4 *Variational Autoencoder*

Variational Autoencoders (VAEs) propose to solve the limitations of vanilla Autoencoders. **VAEs latent space are designed to be continuous**. This means VAEs can easily generate new and diverse samples by smoothly interpolating between different points (explored during training) in the latent space.

The encoder will output two vectors of size $\dim(Z)$: means $\boldsymbol{\mu}$ and standard deviations $\boldsymbol{\sigma}$. This stochastic generation means that even for the same input, although the mean and standard deviation remain the same, the actual encoding will vary on every single pass simply due to sampling from a Gaussian distribution. A problem we would like to avoid is since there are no limits on the values for $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$, the encoder might learn to generate very different $\boldsymbol{\mu}$ for each different class while minimizing $\boldsymbol{\sigma}$ to reobtain a clustered structure which achieves a low reconstruction error. This is a problem because we want the latent space to be continuous, not clustered! This is avoided by KL-divergence[7] between the output distribution and a standard normal distribution. Intuitively, this encourages the encoder to distribute the encodings evenly around the center of the latent space.

[7] $D_{\mathrm{KL}}(p\|q) = \mathbb{E}[\log \frac{p(x)}{q(x)}]$

To train a VAE, we want to **maximize the likelihood of training data**

$$p(x) = \int_{\mathbf{z}} p(x|z)p(z)dz \tag{67}$$

$p(z)$ and $p(x|z)$ are known, but we are not able to compute the integral over all of $\mathbf{z}$. This is intractable! A direct consequence is the posterior distribution $p(z|x) = \frac{p(x|z)p(z)}{p(x)}$ becomes intractable. In order to solve this problem, we define an approximation of the posterior $q_\phi(z|x)$ computed by the encoder to approximate $p_\theta(z|x)$.

$$\log(p_\theta(x)) = \mathbb{E}_{z\sim q_\phi(z|x)}[\log(p(x))] \tag{68}$$

$$= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log \frac{p(x|z)p(z)}{p(z|x)}\right] \tag{69}$$

$$= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log \frac{p(x|z)p(z)}{p(z|x)}\frac{q(z|x)}{q(z|x)}\right] \tag{70}$$

$$= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p(x|z)\right] - \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q(z|x)}{p(z)}\right] + \mathbb{E}_{z|x\sim q_\phi}\left[\log \frac{q(z|x)}{p(z|x)}\right] \tag{71}$$

$$= \underbrace{\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p(x|z)\right]}_{\text{reconstruction error}} - \underbrace{D_{\mathrm{KL}}[q(z|x)\|p(z)]}_{\text{make approx. posterior close to prior}} + \underbrace{D_{\mathrm{KL}}[q(z|x)\|p(z|x)]}_{\text{intractable, but } \geq 0} \tag{72}$$

$$\geq \underbrace{\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p(x|z)\right] - D_{\mathrm{KL}}[q(z|x)\|p(z)]}_{\text{ELBO}} \tag{73}$$

The second term can be computed in a closed form since both arguments are Gaussian.

With the ELBO, we want to joinly maximize the reconstruction error and minimize the KL divergence between the approximate posterior and the prior. The first term encourages the encoder to form clusters where the samples from the same category or with similar properties are closely located in the latent space. The second term encourages the encoder to project latent representations evenly around the center of the latent space.

During training, we must be able to compute the gradients of the ELBO w.r.t to the paremeters of the encoder and the decoder. A minor problem: the process of sampling (in the case of $\mathbf{z}$) from a distribution that is parameterized by our model is not differentiable. For this

reason, we use the **reparametrization trick**. We can now do backprop:

$$\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p(x|z)\right] - D_{\mathrm{KL}}[q(z|x)\|p(z)] = \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log p(x|z)\right] + \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log \frac{p(z)}{q(z|x)}\right] \tag{74}$$

$$= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log \frac{p(x|z)p(z)}{q(z|x)}\right] \tag{75}$$

$$= \mathbb{E}_{z\sim q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] \tag{76}$$

Now we compute the gradients w.r.t θ, ϕ

$$\nabla_{\theta,\phi}\mathbb{E}_{z\sim q_\phi(z|x)}\left[\log \frac{p_\theta(x,z)}{q_\phi(z|x)}\right] = \nabla_{\theta,\phi}\mathbb{E}_{\epsilon\sim\mathcal{N}(0,\mathbf{I})}\left[\log \frac{p_\theta(x,f(x,\epsilon,\theta))}{q_\phi(f(x,\epsilon,\theta)|x)}\right] \tag{77}$$

$$= \mathbb{E}_{\epsilon\sim\mathcal{N}(0,\mathbf{I})}\left[\nabla_{\theta,\phi}\log \frac{p_\theta(x,f(x,\epsilon,\theta))}{q_\phi(f(x,\epsilon,\theta)|x)}\right] \tag{78}$$

$$\approx \frac{1}{K}\sum_{i=1}^{K}\nabla_{\theta,\phi}\log \frac{p_\theta(x,f(x,\epsilon,\theta))}{q_\phi(f(x,\epsilon,\theta)|x)} \tag{79}$$

We can derive an analytic solution for the KL-divergence term. We know $p_\theta(z) \sim \mathcal{N}(0, I)$ and $q_\phi(z|x) \sim \mathcal{N}(\mu, \sigma^2 I)$

$$-D_{\mathrm{KL}}[q_\phi(z|x)\|p_\theta(z)] = \int q_\phi(z|x)\log \frac{p_\theta(z)}{q_\phi(z|x)}dz \tag{80}$$

$$= \int q_\phi(z|x)\log p_\theta(z)dz - \tag{81}$$

$$\int q_\phi(z|x)\log q_\phi(z|x)dz \tag{82}$$

We use the fact that $p(z) = \mathcal{N}(\mu_p, \sigma_p^2 I)$ and $q(z) = \mathcal{N}(\mu_q, \sigma_q^2 I)$. For the first term, we get

$$-\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}(\sigma_j^2 + \mu_j^2) \tag{83}$$

And the second term we get

$$-\frac{J}{2}\log(2\pi) - \frac{1}{2}\sum_{j=1}^{J}(\sigma_j^2 + 1) \tag{84}$$

Adding them, we can conclude

$$-D_{\mathrm{KL}}[q_\phi(z|x)\|p_\theta(z)] = \frac{1}{2}(1 + \log(\sigma_j^2) - \mu_j - \sigma_j^2) \tag{85}$$

5.5 *Monte Carlo Gradient Estimator*

In latent variable models, we often encounter the problem of computing gradient of an expectation

$$\nabla_\psi \mathbb{E}_{p_\psi(z)}[f(z)] = \nabla_\psi \int p_\psi(z) f(z) dz \tag{86}$$

where the random variable z is parameterized by ψ and its samples are fed into another differentiable function f. We need to calculate the gradient of a stochastic process w.r.t. ψ as in the above the equation. Note that this is equivalent to the first term of ELBO where $f = \log p_\theta(x|z)$. We apply the reparameterization trick to compute the gradient in a more amendable way:

$$\mathbb{E}_{p(\epsilon)}[\nabla_\psi f(z)] = \mathbb{E}_{p(\epsilon)}[\nabla_\psi f(g(\epsilon, \psi))] \tag{87}$$

Now we can calculate the expectation of a gradient instead of the gradient of an expectation. We can compute the expectation via Monte Carlo integration

$$\mathbb{E}_{p(\epsilon)}[\nabla_\psi f(g(\epsilon, \psi))] = \frac{1}{S}\sum_{s=1}^{S}[\nabla_\psi f(g(\epsilon^{(s)}, \psi))]_{\epsilon^{(s)} \sim p(\epsilon)} \tag{88}$$

Here, we prove the change of variables step we used above.

Proof. We have $z = g(\epsilon, \psi)$ and $p(z)|dz| = p(\epsilon)|d\epsilon|$ so

$$\int p_\psi(z) f(z) dz = \int p(\epsilon) f(z) d\epsilon \tag{89}$$

$$= \int p(\epsilon) f(g(\epsilon, \psi)) d\epsilon \tag{90}$$

Hence,

$$\mathbb{E}_{p(\epsilon)}[\nabla_\psi f(g(\epsilon, \psi))] = \nabla_\psi \int p(\epsilon) f(g(\epsilon, \psi)) d\epsilon \tag{91}$$

$$= \nabla_\psi \mathbb{E}_{p(\epsilon)}[f(g(\epsilon, \psi))] \tag{92}$$

The gradient is unrelated to the distribution so we obtain the result

$$\mathbb{E}_{p(\epsilon)}[\nabla_\psi f(g(\epsilon, \psi))] \tag{93}$$

□

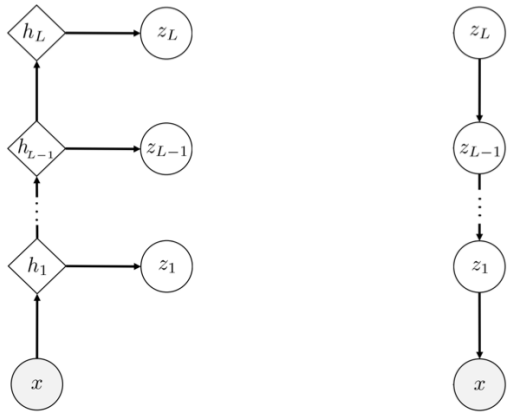

Figure 3. (Left) Inference (encoder/recognition) and (right) generative (decoder) models of a hierarchical VAE. Circles are stochastic variables and diamonds are deterministic variables. This model is equal to a VAE for $L = 1$

5.6 Hierarchical Latent Variable Models

In order to increase the expressiveness of latent variable models, we can form a hierarchy of stochastic latent variables by stacking them. In figure 3, a hierarchical latent variable model is illustrated. Similar to VAEs, we introduce an inference model $q(z|x)$ and optimize a variational lower bound on the log-likelihood.

From figure 3, we can see that $q_\phi(z_1, \dots, z_L|x)$ and generative $p_\theta(z_1, \dots, z_L)$ can be decomposed.

$$q_\phi(z_1, \dots, z_L|x) = q_\phi(z_1|x) \dots q_\phi(z_{L-1}|x) q_\phi(z_L|x) \tag{94}$$

$$p_\theta(z_1, \dots, z_L) = p_\theta(z_L) p(z_{L-1}|z_L) \dots p_\theta(z_1|z_2) \tag{95}$$

We can now derive the ELBO for $z = \{z_1 \dots z_L\}$. We start with

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x) \| p_\theta(z)) \tag{96}$$

The derivation is equivalent to single random variable case. If we expand the random variables:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z_1 \dots z_L|x)}[\log p_\theta(x|z_1 \dots z_L)] - D_{\mathrm{KL}}(q_\phi(z_1 \dots z_L|x) \| p_\theta(z_1 \dots z_L)) \tag{97}$$

Now we use the decomposition we showed earlier, note that x is independent from all $z_{l>1}$, given z_1:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z_1|x)}[\log p_\theta(x|z_1)] - D_{\mathrm{KL}}(q_\phi(z_1 \dots z_L|x) \| p_\theta(z_1 \dots z_L)) \tag{98}$$

Now we expand the KL-divergence term:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z_1|x)}[\log p_\theta(x|z_1)] - \int_{z_1 \dots z_L} q_\phi(z_1 \dots z_L|x) \log \frac{p_\theta(z_1 \dots z_L)}{q_\phi(z_1 \dots z_L|x)} dz_1 \dots z_L \tag{99}$$

Now we can use the decomposition of the inference and generative models:

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z_1|x)}[\log p_\theta(x|z_1)] - \int_{z_1 \dots z_L} \prod_{j=1}^{L} q_\phi(z_j|x) \log \prod_{i=1}^{L} \frac{p_\theta(z_i|z_{i+1})}{q_\phi(z_i|x)} dz_1 \dots z_L \tag{100}$$

where we abusively denote $p_\theta(z_L|z_{L+1}) = p_\theta(z_L)$. Now we take the product out of the log as a sum and move it outside of the integral

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z_1|x)}[\log p_\theta(x|z_1)] - \sum_{i=1}^{L} \int_{z_1 \dots z_L} \prod_{j=1}^{L} q_\phi(z_j|x) \log \frac{p_\theta(z_i|z_{i+1})}{q_\phi(z_i|x)} dz_1 \dots z_L \tag{101}$$

Since we only have z_i and z_{i+1} in the log, the $q_\phi(z_j|x)$ will be marginalized out for $j \notin \{i, i+1\}$

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z_1|x)}[\log p_\theta(x|z_1)] - \sum_{i=1}^{L} \int_{z_1 \dots z_L} q_\phi(z_{i+1}|x) q_\phi(z_i|x) \log \frac{p_\theta(z_i|z_{i+1})}{q_\phi(z_i|x)} dz_i d_{z_{i+1}} \tag{102}$$

Now we can transform the integral into an expectation and KL-divergence term, which gives us the ELBO for hierachical latent variable models.

$$\mathcal{L}(\theta, \phi; x) = \mathbb{E}_{q_\phi(z_1|x)}[\log p_\theta(x|z_1)] - D_{\mathrm{KL}}(q_\phi(z_L|x) \| p_\theta(z_L)) - \sum_{i=1}^{L-1} \mathbb{E}_{z_{i+1} \sim q_\phi(z_{i+1}|x)} D_{\mathrm{KL}}(q_\phi(z_i|x) \| p_\theta(z_i|z_{i+1})) \tag{103}$$

5.7 β-VAE

VAEs still have problems with their latent space: the representations are still entangled. This means that we do not have an explicit way of controlling the output. For example, in the MNIST dataset, we have no way of explicitly sampling a specific number. The β-VAE 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.

In practice, we want to force the KL loss to be under a certain threshold, so we write

$$\max_{\phi, \theta} \mathbb{E}_{x \sim \mathcal{D}} \Big[\log p_\theta(\mathbf{x}|z) \Big] \text{ subject to } D_{\mathrm{KL}} \Big[q_\phi(z|x) \| p_\theta(z) \Big] < \delta \tag{104}$$

Rewriting the constrain optimization problem as a Lagrangian, we ob-

tain:

$$\mathcal{F}(\theta, \phi, \beta) = \mathbb{E}_{z\sim q_\phi(z|x)}\Big[\log p_\theta(\mathbf{x}|x)\Big] - \beta\Big(D_{\mathrm{KL}}\Big[q_\phi(z|x)\|p_\theta(z)\Big] - \delta\Big) \tag{105}$$

$$= \mathbb{E}_{z\sim q_\phi(z|x)}\Big[\log p_\theta(\mathbf{x}|x)\Big] - \beta D_{\mathrm{KL}}\Big[q_\phi(z|x)\|p_\theta(z)\Big] - \underbrace{\beta\delta}_{\geq 0} \tag{106}$$

$$\geq \mathbb{E}_{z\sim q_\phi(z|x)}\Big[\log p_\theta(\mathbf{x}|x)\Big] - \beta D_{\mathrm{KL}}\Big[q_\phi(z|x)\|p_\theta(z)\Big] \tag{107}$$

Thus, the loss (opposite of the objective function) becomes

$$\mathcal{L}_{\beta\text{-VAE}} = -\mathbb{E}_{z\sim q_\phi(z|x)}\Big[\log p_\theta(\mathbf{x}|x)\Big] + \beta D_{\mathrm{KL}}\Big[q_\phi(z|x)\|p_\theta(z)\Big] \tag{108}$$

6 *Autoregressive models*

A regression model, such as linear regression, models an output valued based on a linear combination of input values.

$$\hat{y} = b_0 + b_1 x_1 \tag{109}$$

This technique can be used on time series where input variables are taken as observations at previous time steps, called lag variables. For example, we can predict the value of the current time step t given the observations at the last two time steps $t-1$ and $t-2$. As a regression model, this would look as follows:

$$x_t = b_0 + b_1 x_{t-1} + b_2 x_{t-2} \tag{110}$$

Because the regression model uses data from the same input variable at previous time steps, it is referred to as an **autoregression** (regression of self).

We briefly talk about **Sequence models**. A particular application of autoregressive models is in sequence modeling (i.e. the modeling of sequential data). Some examples of sequence models are language models, which map sequences to scalars or machine translation models, which map sequences to sequences. Sequence models are often considered generative models because they can generate new sequences based on learned patterns.

In particular, autoregressive models generate one element of the sequence at a time, conditioning its generation on previously generated elements. In the typical generation setting, such a model takes a seed of input, such as $x_1, \dots x_k$ and predicts the element in the sequence x_{k+1}. Then we repeat using $x_2, \dots, x_{k+1}$ to predict x_{k+2} and so on. By employing this approach, autoregressive models can effectively capture dependencies between elements in a sequence and build a probability distribution over possible sequences. This distribution can be used to generate new sequences that adhere to the learned patterns. In the subsequent chapter, we will delve into the techniques and strategies for parameterizing sequence models to accomplish this objective.

6.1 *Learning distribution of natural data*

Suppose we have an image consisting of n pixels, and each pixel can take on one of two colours: black or white. We can represent the colour of each pixel using Bernoulli random variables $X_1, X_2, \dots, X_n$ where $X_i = 1$ corresponds to a white pixel, $X_i = 0$ corresponds to a

black pixel. The space of possible configurations consists of 2^n states, as each pixel can independently be black or white. Our goal is to find a parameterization of $p(x_1, x_2, \ldots, x_n)$ that allows us to learn such a distribution from a dataset of images. Once we can define this distribution, we can sample from it to generate new images.

Our first attempt is the **tabular approach**. Via the chain rule of probability we can factorize the joint distribution over the n dimensions:

$$p(\mathbf{x}) = p(x_1, \ldots, x_n) = \prod_{i=1}^{n} p(x_i | x_1, \ldots, x_{i-1}) = \prod_{i=1}^{n} p(x_i | \mathbf{x} < i) \quad (111)$$

Advantage: can represent any possible distribution of n random variables

Limitation: We need $\Theta(\sum_{i=1}^{n} 2^{i-1}) = \mathcal{O}(2^{n-1})$ parameters to parameterize this model.

So a straightforward approach in reducing the number of parameters is to **assume that the variables are independent**. In this case, we obtain the following expression.

$$p(\mathbf{x}) = p(x_1) \times \cdots \times p(x_n) \quad (112)$$

Advantage: we only need n parameters! Feasible fro training.

Limitation: Independence assumption is likely too strong, in practice, it would just be random sampling of pixels unrelated to each other.

So the third attempt is to **specify conditionals with a fixed number of parameters**. In practice, for each position in the sequence, we learn a function $f_i : \{0,1\}^{i-1} \mapsto [0,1]$ governed by Θ_i, which takes the previous pixels as input and outputs the probability of current pixel being white $p(x_i = 1 | x_1, \ldots, x_{i-1})$. The number of total parameters of this model is $\sum_{i=1}^{n} |\Theta_i|$. If we carefully set the dimensions of all $\{\Theta_i\}_{i=1}^{n}$, we obtain a controllable number of parameters.

6.2 *Fully Visible Sigmoid Belief Network*

In a Fully Visible Sigmoid Belief Network, the function f is modelled via logistic regression. Specifically, for $i = 1, \ldots, n$, we have:

$$f_1(x_1, x_2, \ldots, x_{i-1}) = \sigma(\alpha_0^i + \alpha_1^i x_1 + \cdots + \alpha_{i-1}^i x_{i-1}) \quad (113)$$

where σ is the sigmoid function. At the i-th timestep,w e have $i - 1$ parameters denoted as $\Theta = \{\alpha_1, \ldots, \alpha_{i-1}$. Over a horizon of n steps, the number of parameters can be calculated as

$$\sum_{i=1}^{n} |\Theta_i| = \sum_{i=1}^{N} i = \frac{n^2 + n}{2} \in \mathcal{O}(n^2) \quad (114)$$

6.3 Neural Autoregressive Density Estimator (NADE)

The Neural Autoregressive Density Estimator (NADE) is a solution to model binary sequences which offers an alternative parameterization based on Multi-Layer Perceptrons (MLPs). This parameterization offers statistical and computational efficiency compared to the vanilla approach. **Statistical and computational efficiency** compared to the vanilla approach.

$$\mathbf{h}_i = \sigma(\mathbf{b} + W_{:,<i} x_{<i}) \tag{115}$$

$$\hat{x}_i = \sigma(c_i + \mathbf{V}_{i,:}\mathbf{h}_i) \tag{116}$$

The weight matrix $W \in \mathbb{R}^{n \times d}$ and the bias vector $\mathbf{c} \in \mathbb{R}^d$ are shared across the conditionals. The parameter sharing offers two advantages:

1. The total number of parameters gets reduced from $(n^2 d)$ to (nd)
2. The hidden unit activation can be evaluated in $\mathcal{O}(nd)$ doing the following

$$\mathbf{h}_i = \sigma(\mathbf{a}_i) \tag{117}$$

$$\mathbf{a}_{i+1} = \mathbf{a}_i + W_{:,i} x_i \tag{118}$$

where $\mathbf{a}_1 = \mathbf{c}$

NADE is **trained** by **maximizing the average log-likelihood**

$$\frac{1}{n}\sum_{j=1}^{n} \log p(\mathbf{x}^{(j)}) = \frac{1}{n}\sum_{j=1}^{n} \log \left(\prod_{i=1}^{D} p(x_i^{(j)}) | \mathbf{x}_{<i}^{(j)} \right) \tag{119}$$

where j is the sample index. During training ode NADE, **teacher forcing** approach is used: ground truth values (instead of predicted ones) of pixels are used for conditioning when predicting subsequent values. **At inference time predicted values are used.**

6.4 Masked Autoencoder Distribution Estimation (MADE)

The idea behind MADE is to construct an Autoencoder that fulfils the autoregressive such that its outputs can be used as conditionals $p(x_i | \mathbf{x}_{<i})$.

To fulfill the autoregressive property, **no** computational path between output unit x_d and any of its input units $x_d \dots x_D$ must exist (relative to some ordering).

In particular, we can achieve this by defining the mask matrices M^W and M^V as

$$M_{i,j}^W = \mathbb{I}\{m^{[l]}(i) \geq m^{[l-1]}(j)\} \tag{120}$$

$$M_{i,j}^V = \mathbb{I}\{m^{[l]}(i) > m^{[l-1]}(j)\} \tag{121}$$

6.5 Pixel RNN

Pixel RNN generate image starting from the corner and modeling the dependency on previous pixels using a LSTM. In this framework, we condition the probability of the value of a specific pixel both on the two neighboring pixels it is directly connected to and the RNNs hidden state. **Issue**: generation is sequential, thus, slow.

6.6 Pixel CNN

A similar approach is used in PixelCNN with the difference that dependencies are modeled using masked convolutions (To correctly model the conditional probability,one needs to prevent the current and future pixel from contributing to the prediction). This particular type of convolution is needed to ensure that the autoregressive property is satisfied. Specifically, the conditional probability is expressed as:

$$p(\mathbf{x}_i|\mathbf{x}_1,\ldots,\mathbf{x}_{i-1}) = p(x_{i,R}|\mathbf{x}_{<i})p(x_{i,G}|\mathbf{x}_{<i},x_{i,R})p(x_{i,B}|\mathbf{x}_{<i},x_{i,R},x_{i,G}) \tag{122}$$

In order to guarantee this property, we use two types of masks: Mask A and Mask B. Mask A is only applied to the first convolutional layer and restricts connections to those colours in current pixels that have already been predicted. Mask B is applied to higher layers and allows them to be fully connected. During training, we maximize the likelihood of training images. [8]

[8] Lots of tricks can be used to improve PixelCNN such as using gated convolutional layers, short-cut connections, discretized logistic loss, multi-scale, training tricks...

To summarize PixelCNN: Pros

1. Explicit likelihood $p(\mathbf{x})$
2. Likelihood of training data gives good evaluation metric
3. Good samples

Con: Sequential generation: slow at inference time (even though its faster to train than PixelRNN).

6.7 WaveNet: autoregressive generative model for audio data

The concept of WaveNet is aimed to adapt the PixelCNN framework for audio data, which typically involves sequences with a much longer time horizon, such as 16,000 samples per second. To capture long-term dependencies effectively, WaveNet incorporates the concept of dilated convolutions. This type of convolution allows for the exploration of dependencies over larger distances without increasing the number of layers.

6.8 Variational RNNs

The internal transition structure of a standard RNN is entirely deterministic. The only source of randomness or variability in the RNNs can be found in the conditional output probability model. More specifically, $p_\theta(x_t|x_{<t})$ can be represented with Bernoulli or Gaussian distributions for discrete and real-valued data, respectively.

Reasons for augmenting RNNs with random latent variables:

1. To increase the modelling capacity
2. To better capture the uncertainty
3. To infer from the observed variables in the sequence
4. Higher level of abstractions.

Variational RNNs (VRNN) contains a VAE at every time-step. Recall from VAEs that the prior term is a standard Gaussian while inference and generation operations are parameterized by neural networks. In VRNNs we also decompose it into a prior, inference and generation operations but as well as a recurrence step to take the temporal structure into account. The prior is **dynamic** and conditioned on the sate variable h_{t-1}. There is a common subset between the inference ψ and generative θ model parameters. The operations are tied through the RNN hidden state h_t The **prior** term is therefore

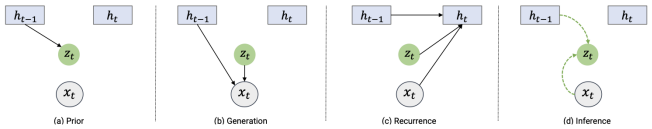

Figure 4. VRNN in graphical model notation decomposed into individual operations. RNN is augmented with a random latent variable z_t. Green dashed connections represent the computation that is part of the inference model.

$$p_\theta(\mathbf{z}|\mathbf{h}) = \prod_{t=1}^{T} p_\theta(z_t|x_{<t}, z_{<t}) \tag{123}$$

where $x_{<t}, z_{<t}$ correspond to the dependencies on the past time-steps through the deterministic hidden state h_{t-1}. The **inference** model $q_\phi(\mathbf{z}|\mathbf{x})$ is

$$q_\phi(\mathbf{z}|\mathbf{x}) = \prod_{t=1}^{T} q_\phi(z_t|z_{<t}, x_{<t}, x_t) \tag{124}$$

The **transition function** is

$$h_t = f_\theta(h_{t-1}, x_t, z_t) \tag{125}$$

The **factorization** over time of the generative model $p_\theta(\mathbf{x}, \mathbf{z})$ is

$$p_\theta(\mathbf{x}, \mathbf{z}) = \prod_{t=1}^{T} p_\theta(x_t|z_t, z_{<t}, x_{<t}) p_\theta(z_t|z_{<t}, x_{<t}) \tag{126}$$

We can now substitute what we decomposed for the prior and inference term into the ELBO. We therefore obtain

$$\log p_\theta(x_t) \geq \mathcal{L}_t(\theta, \phi; x_t) \tag{127}$$

$$= \mathbb{E}_{q_\phi(z_t|x_t, x_{<t}, z_{<t})}\Big[\log p_\theta(x_t|z_t, z_{<t}, x_{<t})\Big] - D_{\mathrm{KL}}(q_\phi(z_t|x_t, x_{<t}, z_{<t}) \| p_\theta(z_t|z_{<t}, x_{<t})) \tag{128}$$

The full training objective ELBO $\mathcal{L}(\theta, \phi; \mathbf{x})$ is simply the sum of single step ELBO $\mathcal{L}_t$ over the entire sequence.

In the standard VAE, the KL-divergence term ensures that the approximate posterior does not deviate too much from the fixed prior. It can be considered as a bound on the approximate posterior, enforcing the model to pack more information, hence resulting in a smoother latent space. The KL-divergence term in the VRNN has a different interpretation. Looking at the ELBO, the first term is the reconstruction of the current step x_t by using the previous hidden state h_{t-1} and latent sample z_t. At training time, the latent variable z_t is computed by using the hidden state h_{t-1} and the current step x_t. Ignoring the reccurence is the same as a VAE.

The inputs of the approximate posterior (inference) and prior models are the same except the additional input x_t to the inference model. Due to the different inputs, by design there is a discrepancy between these two distributions. The KL term aims to minimize the discrepancy between the approximate posterior and the prior. There are two scenarios:

1. The approximate posterior model ignores the additional information from x_t so that minimizing the KL term is straight forward. In this case, the latent variable z_t is no longer informative and generation mostly relies on the hidden state h_{t-1}
2. The approximate posterior model exploits some information from the input x_t, so that the reconstruction term is better optimized while the KL term ensures that the approximate posterior does not memorize the input content x_t and that the prior is enforced to be predictive of the next step t by only using past information h_{t-1}. The latter is achieved since the prior is also parameterized by a network.

6.9 Self-Attention and Transformers

Another approach that can be used for sequence modeling is based on the Transformers architecture. The prediction of the current time step is formed by taking a convex combination of the entire input sequence. The Attention operation, a key component of Transformers, learns to

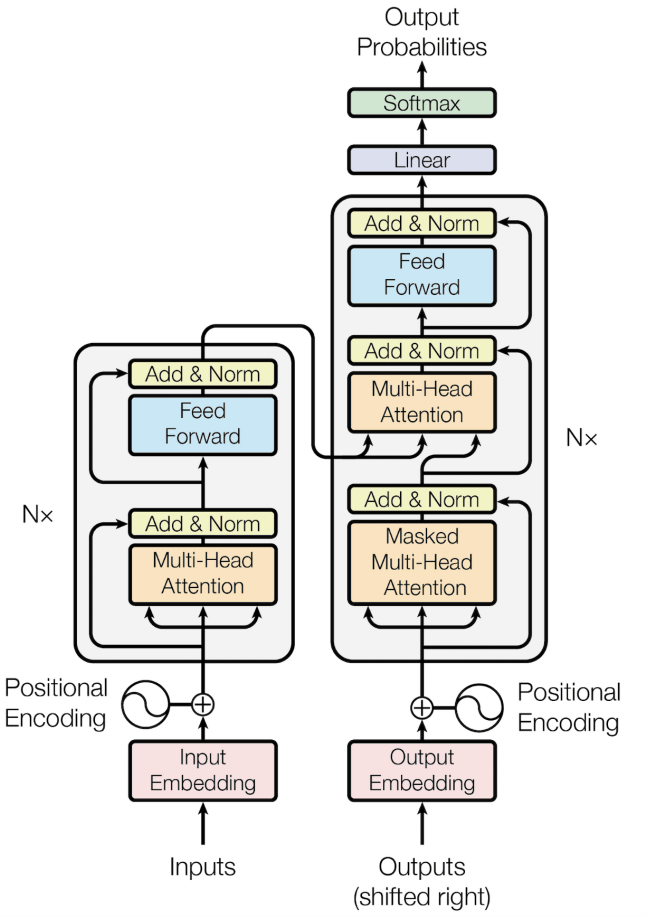

Figure 5. The transformer architecture (left) encoder and (right) decoder.

identiy and select the relevant past information for predicting the next step.

To implement the attention mechanism, given a matrix $\mathbf{x} \in \mathbb{R}^{T \times D}$, we first extract the keys ($\mathbf{K}$), values $\mathbf{V}$ and queries $\mathbf{Q}$ from $\mathbf{X}$ as follows:

$$\mathbf{K} = \mathbf{X}\mathbf{W}_K \tag{129}$$

$$\mathbf{Q} = \mathbf{X}\mathbf{W}_Q \tag{130}$$

$$\mathbf{V} = \mathbf{X}\mathbf{W}_V \tag{131}$$

where $\mathbf{W}_K, \mathbf{W}_Q, \mathbf{W}_V$ are learnable weightes and are generally $D \times D$ matrices (where D is the representation size of the inputs $\mathbf{X}$). We can also choose the queries and keys to have a different dimensionality. Intuitively, the attention mechanism learns a codebook representation from the inputs $\mathbf{X}$ (*we can think of this as a dictionary representation where we map keys* $\mathbf{K}$ *to the values* $\mathbf{V}$ *using the queries* $\mathbf{Q}$). We then identify a set of values to form a new prediction with a weighted, linear combination of these values.

If all keys, values and queries are extracted from the same sequence, we call it **self-attention**. If the queries are generated from a different sequence, we call it **cross-attention**.

Scaled Dot-Product Attention[9] has $\boldsymbol{\alpha} = \text{score}(\mathbf{q}_t, \mathbf{K}_i) = \frac{\mathbf{q}_t \mathbf{K}_i^\top}{\sqrt{d_k}}$. The attention function is then

[9] The following sections are adpated from a research paper I wrote on Transformers

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d_k}})\mathbf{V} \tag{132}$$

where the softmax is applied column wise. The scale $\frac{1}{\sqrt{d_k}}$ was proposed to prevent the softmax from pushing into extremely small gradients, which is caused by the large dot product.

Let $\mathbf{W}_i^Q, \mathbf{W}_i^K \in \mathbb{R}^{d_{\text{model}} \times d_k}, \mathbf{W}_i^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and $\mathbf{W}^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$ where h is the number of heads. Then, we can define each head as:

$$\text{head}_i = \text{Attention}(\mathbf{Q}\mathbf{W}_i^Q, \mathbf{K}\mathbf{W}_i^K, \mathbf{V}\mathbf{W}_i^V) \tag{133}$$

By concatenating each of these heads, we get the Multi-head Attention:

$$\text{MHSA}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{Concat}(\text{head}_1 \dots \text{head}_h)\mathbf{W}^O \tag{134}$$

The inputs, outputs masks have a 1 for non PAD indices and 0 otherwise. Since we also give the outputs during training, we need to correctly apply subsequent masks along with shifting output embedding to the right to ensure the auto-regressive nature of the model. The subsequent mask is a lower triangular matrix filled with 1s. The **masking** for the output is done as taking the & operation with the output mask.

7 *Normalizing Flows and Invertible Neural Networks*

So far we have seen variational autoencoders $p_\theta(x) = \int_\theta(x, z)dz$ and autoregressive models $p_\theta(x) = \prod_{i=1}^{n} p_\theta(x_i|x_{<i})$. Variational autoencoders can learn features/representations via a learned latent variable z. However, they have intractable marginal likelihoods! Autoregressive models have tractable likelihoods but have no latent space concept and therefore does not have a direct mechanism to learn the features.

Some **Desired properties of any model distribution**

1. Analytic model density and easy to sample
2. The distribution should represent for complex data, such as images or videos. (This however is usually muti-modal and complex. Highly **non**-trivial to sample from it)

So can we design a latent variable model with tractable likelihoods?

The answer is YES! All thanks to the "change of variables" technique, which allows us to create a complex distribution from a simple distribution, and a mapping between them.

7.1 *Change of Variables*

Lets quickly review the change of variables technique. In the simple 1D case:

(u-)Substitution: For $x = g(u)$

$$\int_{x_0=g(a)}^{x_1=g(b)} f(x)dx = \int_{u_0=a}^{u_1=b} f(g(u))g'(u)du \tag{135}$$

A similar change of variables can be applied in the case of probability distributions. Suppose we have a random variable $z \sim p_z(z)$ and $x = f(z)$ where f is a monotone, continuous and differentiable function with inverse $z = f^{-1}(x)$. Then the pdf of x is given by

$$p_x(x) = p_z(f^{-1}(x))|f^{-1'}(x)| = p_z(z)|f^{-1'}(x)| \tag{136}$$

Now lets extend this to any dimension.

$$p_X(\mathbf{x}) = p_Z(f^{-1}(x)) \left|\det\Big(\frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}}\Big)\right| \tag{137}$$

which is also $p_Z(f^{-1}(x)) \left|\det\Big(\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}\Big)\right|^{-1}$. Note the Jacobian matrix

is invertible. It is lower triangular to compute the determinant in $\mathcal{O}(d)$ time.

7.2 *Normalizing Flows*

Normalizing flows is an direct application of the change of variables. Consider directed, latent variable model over observed variables X and latent variables Z. The mapping between Z and X is given by deterministic and invertible function $f_\theta : \mathbb{R}^d \mapsto \mathbb{R}^d$, s.t. $X = f_\theta(Z)$ and $Z = f_\theta^{-1}(X)$. The marginal likelihood is

$$p_X(\mathbf{x};\theta) = p_Z(f^{-1}(\mathbf{x})) \left| \det\left(\frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \right) \right| \tag{138}$$

We can parameterize the transformation f with a neural network. However, an aribtary neural network does not work here. From a theoretical perspective, the neural network must be **differentiable, invertible** and **dimension preserving**. For a computational perspective, the Jacobian determinant must be computed **efficiently**. Figure 6 shows a

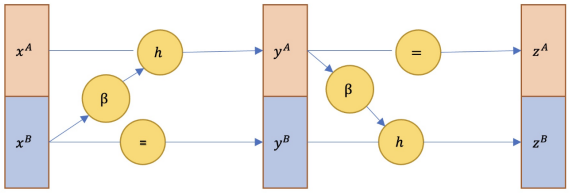

Figure 6. Combined Coupling layers. It's design ensures the efficient computation of the Jacobian determinant. Note that the function β can be arbitrarily complex and does not need to be invertible.

combined coupling layer. If we only consider the mapping between $\mathbf{x}$ and $\mathbf{y}$ we have the **forward pass** as:

$$\begin{bmatrix} y^A \\ y^B \end{bmatrix} = \begin{bmatrix} h(x^A, \beta(x^B)) \\ x^B \end{bmatrix} \tag{139}$$

Inverse pass as:

$$\begin{bmatrix} x^A \\ x^B \end{bmatrix} = \begin{bmatrix} h^{-1}(y^A, \beta(y^B)) \\ y^B \end{bmatrix} \tag{140}$$

And the Jacobian matrix as:

$$\begin{bmatrix} h' & h'\beta' \\ 0 & 1 \end{bmatrix} \tag{141}$$

Where h is an element-wise function and β is arbitarily complex, which could be an MLP, CNN, etc. Notice the bottom part of the input simply gets copied over, which is why we combine coupling layers as shown in 6.

A single nonlinear transform is normally not powerful enough. More complex transformations can be attained via composition. Now we have a flow of transformations, each transform can be a neural network.

$$\mathbf{x} = f(\mathbf{z}) = f_k \circ f_{k-1} \circ \dots f_2 \circ f_1(\mathbf{z}) \tag{142}$$

Based on the generic change of variables we have

$$p_X(\mathbf{x};\theta) = p_Z(f^{-1}(\mathbf{x})) \prod_k \left| \det\Big(\frac{\partial f_k^{-1}(\mathbf{x})}{\partial \mathbf{x}}\Big) \right| \tag{143}$$

Training We train the model via **maximizing the EXACT log likelihood over the dataset** $\mathcal{D}$

$$\log p_{\mathbf{x}}(\mathcal{D}) = \sum_{x \in \mathcal{D}} \left(\log p_Z(f^{-1}(\mathbf{x})) + \sum_k \left| \log \det\Big(\frac{\partial f_k^{-1}(\mathbf{x})}{\partial \mathbf{x}}\Big) \right| \right) \tag{144}$$

we assume that samples are independently and identically distributed.

Inference At test time:

- To generate sample $\mathbf{x}$, we can draw a sample from $\mathbf{z} \sim p_Z$ and transform it via $f : \mathbf{x} = f(\mathbf{z})$
- To evaluate the probability of an observation $\mathbf{x}$, we leverage the inverse transform $\mathbf{z} = f^{-1}(\mathbf{x})$ to calculate its probability $p_Z(\mathbf{Z})$

7.3 Planar Normalizing Flow

Given the latent variable $\mathbf{z} \in \mathbb{R}^d$, the mapping funcion is given by

$$f(\mathbf{z}) = \mathbf{z} + \mathbf{u} h(\mathbf{w}^\top \mathbf{z} + \mathbf{b}) \tag{145}$$

where $\mathbf{u}, \mathbf{w}, \mathbf{b}$ are learnable parameters and h is a nonlinear activation function. f is continuously differentiable and invertible. Assuming $\mathbf{z} \sim \mathcal{N}(0, \mathbf{I})$, i.e., $p_z(\mathbf{z}) = \prod_i \frac{1}{\sqrt{2\pi}} \exp(-\frac{z_i^2}{2})$. The objective function over the learnable parameters can be

$$\operatorname{argmin} \mathcal{L} = \operatorname{argmax} \log p_x(\mathbf{x}) = \operatorname{argmin} \Big\{ |\mathbf{z}|^2 + \log |1 + h' \mathbf{u}^\top \mathbf{w}| \Big\} \tag{146}$$

Proof. By definition of normalizing flows, we obtain

$$p_x(\mathbf{x}) = p_z(f^{-1}(x)) \left| \log \det\Big(\frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}}\Big) \right|^{-1} \tag{147}$$

So we have

$$\frac{\partial f}{\partial z} = \mathbf{I} + h'(\mathbf{w}^\top \mathbf{z} + \mathbf{b})\mathbf{u} \cdot \mathbf{w}^\top \tag{148}$$

Via the matrix determinant lemma, we get

$$\det\Big(\frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}}\Big) = (1 + h'\mathbf{u}^\top \mathbf{I}\mathbf{w})\det(\mathbf{I}) = (1 + h'\mathbf{u}^\top \mathbf{w}) \tag{149}$$

Applying the inverse to above and substituting back to $p_x(\mathbf{x})$ and applying change of variables we get

$$p_x(\mathbf{x}) = p_z(\mathbf{z})(1 + h'\mathbf{u}^\top \mathbf{w}) \tag{150}$$

Applying the definition of gaussians and using log rules, we obtain

$$p_x(\mathbf{x}) = -|\mathbf{z}|^2 - \log|1 + h'\mathbf{u}^\top \mathbf{w}| \tag{151}$$

So

$$\operatorname{argmax} \log p_x(\mathbf{x}) = \operatorname{argmin}\Big\{|\mathbf{z}|^2 + \log|1 + h'\mathbf{u}^\top \mathbf{w}|\Big\} \tag{152}$$

□

7.4 Conditional Coupling Normalizing Flow

In many practical cases, we wish to generate samples based on certain conditions, such as generating images based on the object labels. Here we consider an example of conditional normalizing flow with the following relation:

$$p_x(\mathbf{x}) = p_z(f^{-1}(x))\left|\log\det\Big(\frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}}\Big)\right|^{-1} \tag{153}$$

with $\mathbf{x}, \mathbf{z} \in \mathbb{R}^D$. The function f is given by a coupling layer, i.e.

$$\mathbf{x}_0 = \mathbf{z}_0 \tag{154}$$

$$\mathbf{x}_1 = \tau(\mathbf{z}_0) + \mathbf{z}_1 \odot \sigma(\mathbf{A}\mathbf{z}_0 + \mathbf{b}) \tag{155}$$

in which $\mathbf{x}_0, \mathbf{z}_0 \in \mathbb{R}^{d_0}$, $\mathbf{x}_1, \mathbf{z}_1 \in \mathbb{R}^{D-d_0}$. σ denotes a positive nonlinear activation function, $\tau : \mathbb{R}^{d_0} \mapsto \mathbb{R}^{D-d}$. $\mathbf{A}, \mathbf{b}$ are learnable affine transform parameters and $\odot$ denotes the element-wise multiplications (Hadamard product). The exact likelihood is

$$\log p(\mathbf{x}|c) = \log p(\mathbf{z}|c) - \sum_{j-1}^{D-d_0} \log\Big[\sigma(\mathbf{A}\mathbf{z}_0 + \mathbf{b})\Big]_j \tag{156}$$

where $[]_j$ denotes the j-th entry of the vector.

Proof. First, apply logarithms on the conditional normalizing flow relation (note we took the inverse down and made it a minus due to log rules)

$$\log p_x(\mathbf{x}) = \log p_z(f^{-1}(x)) - \log \left| \log \det \Big(\frac{\partial f^{-1}(\mathbf{x})}{\partial \mathbf{x}} \Big) \right| \tag{157}$$

Now lets find the second term, f is a coupling layer in the form of

$$\begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \end{bmatrix} = \begin{bmatrix} \mathbf{z}_0 \\ \tau(\mathbf{z}_0) + \mathbf{z}_1 \odot \sigma(\mathbf{A}\mathbf{z}_0 + \mathbf{b}) \end{bmatrix} \tag{158}$$

We find the Jacobian, we can think of $\mathbf{f} = [f_1, f_2]$ and $\mathbf{z} = [z_1, z_2]$

$$\det \Big(\frac{\partial \mathbf{f}(\mathbf{z})}{\partial \mathbf{z}} \Big) = \det \left(\begin{bmatrix} \frac{\partial f_1(\mathbf{z})}{\partial z_0} & \frac{\partial f_1(\mathbf{z})}{\partial z_1} \\ \frac{\partial f_2(\mathbf{z})}{\partial z_0} & \frac{\partial f_2(\mathbf{z})}{\partial z_1} \end{bmatrix} \right) \tag{159}$$

$$= \det \left(\begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \frac{\partial f_2(\mathbf{z})}{\partial z_0} & \sigma(\mathbf{A}\mathbf{z}_0 + \mathbf{b}) \end{bmatrix} \right) \tag{160}$$

$$= \left(\prod_{j=1}^{D-d_0} \operatorname{diag}(\sigma(\mathbf{A}\mathbf{z}_0 + \mathbf{b})) \right)^{-1} \tag{161}$$

Via change of variables and logarithm properties, substituting it back to the conditional normalizing flow relation we get

$$\log p(\mathbf{x}|c) = \log p(\mathbf{z}|c) - \sum_{j-1}^{D-d_0} \log \Big[\sigma(\mathbf{A}\mathbf{z}_0 + \mathbf{b}) \Big]_j \tag{162}$$

□

8 Generative Adversarial Networks

All generative models we have seen so far act maximizing the likelihood. However there are cases when high-dimensional data, the log-likelihood can be heavily influenced by the term proportional to d, resulting in an high log-likelihood even when the majority of the generated samples are of poor quality. There are also instances where we can generate high-quality even when the log-likelihood is low when the model simply memorizes the training data, allowing it to replicate the exact samples it has seen before. However, when faced with new, unseen data during testing, the model struggles to assign non-zero probabilities, resulting in a poor log-likelihood.

To address these situations, likelihood-free models (Implicit Models or Neural Samplers) come into play. They are capable of handling highly expressive model classes, often referred to as universal, and can also handle cases where the density function is undefined or intractable. However, it's important to note that likelihood-free models present their own set of challenges. They lack a well-established theory and may require different learning algorithms compared to explicit models.

Lets first define some components behind the GAN model.

Generator

$$G : \mathbb{R}^Q \mapsto \mathbb{R}^D \tag{163}$$

The generator G is a neural network that is trained to ideally map random normal-distributed inputs, drawn from $\mathbb{Z}$, to a sample following the data distribution as output.

Discriminator

$$D : \mathbb{R}^D \mapsto [0, 1] \tag{164}$$

The discriminator D is trained to output a probability. Ideally the discriminator assigns a probability of 1 if the input is a real image and a probability of 0 if the input is a generated (fake) image.

8.1 GAN Objective

Assume G is fixed to train D given a set of real samples $x^n \sim p_{\text{data}}$ where $n = 1, \ldots, N$. We generate an equal number of random samples $z^n \sim \mathcal{N}(0, 1)$ which allows us to form the training dataset $\mathcal{D}$ as

$$\mathcal{D} = \{\underbrace{(x_1, 1), \ldots (x_N, 1)}_{\text{real samples}}, \underbrace{(G(z_1), 0), \ldots, (G(z_N), 0)}_{\text{fake samples}}\} \tag{165}$$

To train D given the data $\mathcal{D}$, we use Binary Cross Entropy (BCE) loss:

$$\mathcal{L}(\mathcal{D}) = -\frac{1}{2N}\left(\sum_{i=1}^{N} y^{(i)} \log(D(x_i)) + \sum_{N+1}^{2N} (1 - y_i) \log(1 - D(x_i))\right) \tag{166}$$

where y_i are the ground truth binary labels. Thus providing us with an objective to train D as a function of G:

$$D^* = \underset{D}{\operatorname{argmin}} -\frac{1}{2}\left(\mathbb{E}_{x\sim p_{\text{data}}}\Big[\log D(x\Big] + \mathbb{E}_{z\sim p_z}\Big[\log(1 - D(x_i))\Big]\right) \tag{167}$$

Then, to find the optimal discriminator G^*, we aim to fool any possible discriminator D. Thus, we minimize the value function $V(D, G) = \mathbb{E}_{x\sim p_d}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(\hat{\mathbf{x}}))]$ computed for the optimal discriminator D^*

$$G^*, D^* = \underset{G}{\operatorname{argmin}}\, \underset{D}{\operatorname{argmax}}\, V(D, G) \tag{168}$$

Then we can derive an optimal discriminator. Note I will be using p_d for the data distribution and p_m for the model distribution.

The **Optimal Discriminator** is, for each generator G

$$D^* = \frac{p_d(x)}{p_d(x) + p_m(x)} \tag{169}$$

Proof.

$$D^* = \underset{D}{\operatorname{argmax}}\, V(D, G) \tag{170}$$

$$= \underset{D}{\operatorname{argmax}}\, \mathbb{E}_{x\sim p_d}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(\hat{\mathbf{x}}))] \tag{171}$$

$$= \underset{D}{\operatorname{argmax}} \int_x p_d(x) \log(D(x))dx + \int_z p_z(z) \log(1 - D(G(z))dz \tag{172}$$

$$= \underset{D}{\operatorname{argmax}} \int_x p_d(x) \log(D(x)) + p_m(x) \log(1 - D(x))dx \tag{173}$$

Note that we used the law of unconscious statistician to change the variables. Considering any $a, b \in \mathbb{R}^2\backslash\{0, 0\}$ and the function $f(y) = a \log(y) + b \log(1 - y)$. f reaches its maximum in the point $\frac{a}{a+b}$ as

$$f'(y) = \frac{a}{y} - \frac{b}{1 - y} = 0 \iff y = \frac{a}{a + b} \tag{174}$$

and checking its second derivative $\forall a, b > 0$

$$f''(y) = -\frac{a}{y^2} - \frac{b}{(1-y)^2} < 0 \tag{175}$$

which is indeed a maximum point. Thus,

$$D^* = \frac{p_d(x)}{p_d(x) + p_m(x)} \tag{176}$$

□

Now we will derive the global optimum of the training criterion. First, we can substitute the D^* which we derived earlier

$$V(G, D^*) = \mathbb{E}_{x \sim p_d}\left[\log\left(\frac{p_d(x)}{p_d(x) + p_m(x)}\right)\right] + \mathbb{E}_{z \sim p_z}\left[\log\left(1 - \frac{p_d(x)}{p_d(x) + p_m(x)}\right)\right] \tag{177}$$

Rewriting the second term

$$V(G, D^*) = \mathbb{E}_{x \sim p_d}\left[\log\left(\frac{p_d(x)}{p_d(x) + p_m(x)}\right)\right] + \mathbb{E}_{z \sim p_z}\left[\log\left(\frac{p_m(x)}{p_d(x) + p_m(x)}\right)\right] \tag{178}$$

Inside each log we multiply and divide by 2

$$V(G, D^*) = \mathbb{E}_{x \sim p_d}\left[\log\left(\frac{2p_d(x)}{2(p_d(x) + p_m(x))}\right)\right] + \mathbb{E}_{z \sim p_z}\left[\log\left(\frac{2p_m(x)}{2(p_d(x) + p_m(x))}\right)\right] \tag{179}$$

$$= \mathbb{E}_{x \sim p_d}\left[\log\left(\frac{2p_d(x)}{p_d(x) + p_m(x)}\right)\right] + \mathbb{E}_{z \sim p_z}\left[\log\left(\frac{2p_m(x)}{p_d(x) + p_m(x)}\right)\right] - \log 4 \tag{180}$$

$$= D_{\mathrm{KL}}\left(p_d \| \frac{p_d(x) + p_m(x)}{2}\right) + D_{\mathrm{KL}}\left(p_m(x) \| \frac{p_d(x) + p_m(x)}{2}\right) - \log 4 \tag{181}$$

$$= 2D_{\mathrm{JS}}(p_d(x) \| p_m(x)) - \log 4 \tag{182}$$

As $G^* = \operatorname{argmin}_G V(D^*, G)$ and $\forall x$ we hahve $D_{\mathrm{JS}} \geq 0$, the optimal value is achieved when $D_{\mathrm{JS}}(p_d(x) \| p_m(x)) = 0$ that specifically when $p_d(x) = p_m(x)$, so

$$V(G^*, D^*) = -\log 4 \tag{183}$$

Global optimum of training criterion

$$\mathbb{E}_{x \sim p_d}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))] \tag{184}$$

is achieved if $p_m(x) = p_d(x)$ and at optimum $V(G^*, D^*) = -\log 4$

8.2 Convergence of training algorithm

Convergence of training algorithm If G and D have enough capacity and at each update step D is allowed to reach $D = D^*$ and p_m is updated to improve

$$V(p_m, D^*) = \mathbb{E}_{x\sim p_d}[\log D^*(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D^*(x))] \quad (185)$$

$$\propto \sup_D \int_x p_m(x) \log(1 - D(x))dx \quad (186)$$

then p_m converges to p_d

Proof. The argument of the supremum is convex in p_m. The supremum operation does not change the convexity, thus $V(G, V^*)$is also convex in p_m with global optimum as in *Theorem 1.4.2* of the course notes. □

The theoretical results make very strong **assumptions**,

- The generator G and discriminator D have enough capacity
- The discriminator D reaches its optimum D^* at very outer iteration
- We directly optimize p_m instead of its parameters Θ

In practice, it happens that G and D have finite capacity, D is optimized for only k steps. Using a neural netowrk for G might mean the function is no longer convex. Thus in practice p_m may not converge to p_d and oscillate. GANs however, still work well in practice by keeping D close to D^* provides meaning gradients for G to improve generation.

8.3 Training

Initialize G and D with random weights Θ_G, Θ_D
while not converged **do**
  for k steps **do**
    Draw n training samples $\{x_1, \dots x_n\}$ from $p_d(x)$
    Draw n training samples $\{z_1, \dots z_n\}$ from $p(z)$
    $\mathcal{L}_D = \frac{1}{n}\sum_{i=1}^{n} \log(D(\boldsymbol{x}_i)) + \log(1 - D(G(\boldsymbol{z}_i)))$
    perform a gradient ascent step on $\nabla_{\Theta_D}\mathcal{L}_D$
  end for
  Draw n training samples $\{z_1, \dots z_n\}$ from $p(z)$
  $\mathcal{L}_G = \frac{1}{n}\sum_{i=1}^{n} \log(D(G(\boldsymbol{z}_i)))$
  perform a gradient ascend step on $\nabla_{\Theta_G}\mathcal{L}_G$
end while

Algorithm 1. GAN training algorithm with gradient ascent

The original equation $\mathcal{L}_G = \frac{1}{n}\sum_{i=1}^{n}\log(1 - D(G(\boldsymbol{z}_i)))$ may not provide sufficient gradient for G to learn well. In early stages, D can reject samples from G because G is poor and thus the generated samples are highly different. In this case $\log(1 - D(G(\boldsymbol{z}_i)))$ saturates. This is why instead of minimizing $\log(1 - D(G(\boldsymbol{z}_i)))$ we maximize $\nabla_{\Theta_G}\mathcal{L}_G$ to perform gradient ascent.

Mode collapse is when the generator learns to produce high-quality samples with very low variability, covering only a fraction of the data distribution.

The most common solution to mode collapse is the *unrolled GAN*. The idea is to (move the generator forward) 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.

GAN objective can be generalized to an entire family of divergences. The Wasserstein GAN 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, we can add a gradient penalty,

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

9 *Diffusion Models*

In Diffusion Models, we split up the generation process into multiple smaller steps. It can be regarded as a chain of steps. We start with an image that purely of noise and, in each step of that chain we want to partially denoise it.

- **Denoising** is the process of going from random noise to an image
- **Diffusion** is the process of going from an image of random noise. The steps are deterministic, and each step adds some noise to the current version of the image.

The notation is to call the real image $\mathbf{x}_0$ and the noisy sample $\mathbf{x}_t$. In particular, we have $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ where $q(\mathbf{x}_0)$ is the original data distribution, and $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$ is a sample from the Gaussian image. We call $\mathbf{x}_t$ the image at the t-th step of the diffusion chain with large t corresponding to noisier images. in practice, we model the chain as a **Markov stochastic process** $\{\mathbf{x}_t\}_0^T$, thus each step only depends on the one immediately before it.

9.1 *Diffusion Step*

To generate training data for our Diffusion Model we need to perform the diffusion process mapping real images to random noise. The process is done in the following way.

Each diffusion step $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ adds Gaussian noise to the previous image $\mathbf{x}_{t_1}$ obtaining $\mathbf{x}_t$. Doing that, we produce a sequence of T noisy samples $\mathbf{x}_1, \ldots, \mathbf{x}_T$. The amount of noise introduced at each step is controlled by a variance schedule $\{\beta \in (0,1)\}_{t=1}^T$ such that $0 < \beta_1 < \beta_T < 1$. In particular, we compute $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ is computed as

$$q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{187}$$

However, this definition is sequential: we have to generate each step on top of the previous one. So, we cannot parallelized it. Conversely, we would like a formulation that, given a time step t, and an original image x_0 that can generate the image $\mathbf{x}_t$ directly.

To derive the closed-form solution of x_t, we first use the reparameterization trick to write the noisy image at time step t as:

$$\mathbf{x}_t = \sqrt{1-\beta_t}\mathbf{x}_{t-1} + \sqrt{\beta_t}\epsilon \tag{188}$$

where $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. Now lets define $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^t \alpha_i$. We

can now obtain

$$\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\epsilon \tag{189}$$
$$= \sqrt{\alpha_t\alpha_{t-1}}\mathbf{x}_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\epsilon \tag{190}$$
$$= \dots \tag{191}$$
$$= \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon \tag{192}$$

As a result, we can directly compute the noise sample as

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{193}$$

which drastically speeds up the process of training data generation.

There are mainly two different ways to define the β scheduler: a linear schedule and a cosine schedule. The problem with the linear scheduler is the image becomes pure noise too quick and makes it hard for the model to learn. For this reason, a cosine schedule is usually preferred.

9.2 Denoising Step

The process we want to learn is **denoising**, which allows us to transform an image sampled from random noise into a realistic one. In practice, we need to learn $p(\mathbf{x}_t|\mathbf{x}_{t-1})$, which perform the denosing step from $\mathbf{x}_t$ to $\mathbf{x}_{t-1}$. As these steps are in practice very small, $p(\mathbf{x}_t|\mathbf{x}_{t-1})$ is a small transformation which can be easily approximated by a neural network $p_\Theta(\mathbf{x}_t|\mathbf{x}_{t-1})$.

By definition, $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ is a Gaussian distribution with known parameters. For small enough forward/diffusion steps, i.e. if β is small enough, $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ will also be Gaussian so we can parameterize it as:

$$p_\Theta(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\mu_\Theta(\mathbf{x}_t, t), \Sigma_\Theta(\mathbf{x}_t, t)) \tag{194}$$

Here similary to VAEs, instead of trying to predict the full distribution, we only need to predict the parameters of the Gaussian distribution. There are only two denoising models, one for the mean $\mu_\Theta(\mathbf{x}_t, t)$ and one for the variance $\Sigma_\Theta(\mathbf{x}_t, t)$. [10] To generate, we

[10] Each of them take in the noisy image $\mathbf{x}_t$ and the timestep t.

1. Randomly sample some Gaussian noise $\mathbf{x}_t$
2. Iteratively denoise it until we get $\mathbf{x}_0$, a sample from the approximated real distribution. Since this is a Markov Chain, we can define $p_\Theta(x_{0:T})$ as

$$p_\Theta(x_{0:T}) = p(\mathbf{x}_0)\prod_{t=1}^{T} p_\Theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \tag{195}$$

where $p(\mathbf{x}_t) \sim \mathcal{N}(0,1)$ and $p_\Theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \sim \mathcal{N}(\mu_\Theta(\mathbf{x}_t, t), \Sigma_\Theta(\mathbf{x}_t, t))$

Reverse Distribution in Diffusion Models We now show that the denoising distribution in diffusion models is Gaussian when conditioned on clean data. More specifically, let $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ be the data and assume we define a Gaussian diffusion process via $q(\mathbf{x}_t|\mathbf{x}_{t-1}) = \mathcal{N}(\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$. Define $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{i=1}^{t} \alpha_i$. Specifically, we want to show $q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)$ is given by $\mathcal{N}(\boldsymbol{\mu}_q(\mathbf{x}_t, \mathbf{x}_0), \sigma_q^2\mathbf{I})$.

Proof. We start by Bayes' theorem for $q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)$ and ignore the terms that do note depend on $\mathbf{x}_{t-1}$

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \frac{q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} \propto q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0)q(\mathbf{x}_{t-1}|\mathbf{x}_0) \tag{196}$$

We notice this is a product of two Gaussians, however note the first one is in terms of $\mathbf{x}_t$ and the second one is in terms of $\mathbf{x}_{t-1}$, so we cannot directly use any formulas for product of Gaussians.

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \cdot \mathcal{N}(\sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t\mathbf{I})) \tag{197}$$

Now, we want to show that the product indeed does take the following form

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}(\boldsymbol{\mu}_q(\mathbf{x}_t, \mathbf{x}_0), \sigma_q^2\mathbf{I}) \propto \exp\{-\frac{\|\mathbf{x}_t - \boldsymbol{\mu}_q(\mathbf{x}_t, \mathbf{x}_0)\|^2}{2\sigma_q^2}\} \tag{198}$$

So we try to match the terms

$$q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \propto \exp\{\frac{\|\mathbf{x}_t - \sqrt{1-\beta_t}\mathbf{x}_{t-1}\|^2}{-2\beta_t}\} \exp\{\frac{\|\mathbf{x}_{t-1} - \sqrt{\bar{\alpha}_t}\mathbf{x}_0\|^2}{-2(1-\bar{\alpha}_t)}\} \tag{199}$$

Expanding the square first term

$$\exp\left\{-\frac{\mathbf{x}_t^2 - 2\sqrt{1-\beta_t}\mathbf{x}_{t-1}\mathbf{x}_t + (1-\beta_t)\mathbf{x}_{t-1}^2}{2\beta_t}\right\} \tag{200}$$

Second Term

$$\exp\left\{-\frac{\mathbf{x}_{t-1}^2 - 2\sqrt{\bar{\alpha}_t}\mathbf{x}_0\mathbf{x}_{t-1} + \bar{\alpha}_t\mathbf{x}_0^2}{2(1-\bar{\alpha}_t)}\right\} \tag{201}$$

So we can match the terms. Notice that $a = 1$, $b = -1$, $c = 1$ of of a quadratic $ax^2 + bxy + cy^2$ and so we can see which terms are "matching" or "left over"

$$\frac{1}{\sigma_q^2} = \frac{1}{1-\bar{\alpha}_{t-1}} + \frac{1-\beta_t}{\beta_t} \tag{202}$$

$$= \frac{1-\bar{\alpha}_t}{\beta_t(1-\bar{\alpha}_{t-1})} \tag{203}$$

So

$$\sigma_q^2 = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} = \frac{(1-\alpha_t)(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} \tag{204}$$

And for the mean we have (matching cy^2)

$$\frac{\boldsymbol{\mu}_q(\mathbf{x}_t, \mathbf{x}_0)}{\sigma_q^2} = \frac{\sqrt{1-\beta_t}\mathbf{x}_t}{\beta_t} + \frac{\sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0}{1-\bar{\alpha}_{t-1}} \tag{205}$$

So

$$\boldsymbol{\mu}_q(\mathbf{x}_t, \mathbf{x}_0) = \frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}\mathbf{x}_t + (1-\alpha_t)\sqrt{\bar{\alpha}_{t-1}}\mathbf{x}_0}{1-\bar{\alpha}_t} \tag{206}$$

□

9.3 ELBO for Diffusion Models

Objective function ELBO Similar to VAE, we can rely on a lower bound (ELBO) to optimize log-likelihood.

$$\log p(\mathbf{x}) = \log \int p(x_{0:T}) d\mathbf{x}_{1:T} \tag{207}$$

$$= \log \int \frac{p(x_{0:T})q(x_{1:T}|\mathbf{x}_0)}{q(x_{1:T}|\mathbf{x}_0)} d\mathbf{x}_{1:T} \tag{208}$$

$$= \log \mathbb{E}_{q(x_{1:T}|\mathbf{x}_0)}\left[\frac{p(x_{0:T})}{q(x_{1:T}|\mathbf{x}_0)}\right] \tag{209}$$

$$\geq \mathbb{E}_{q(x_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(x_{0:T})}{q(x_{1:T}|\mathbf{x}_0)}\right] \tag{210}$$

Where in the last step, we used Jensen's inequality. Now using the chain rule, we notice that $q(\mathbf{x}_{1:T}) = q(\mathbf{x}_1)q(\mathbf{x}_2|\mathbf{x}_1)q(\mathbf{x}_3|\mathbf{x}_2, \mathbf{x}_1)\dots q(\mathbf{x}_T|\mathbf{x}_{T-1}, \dots, \mathbf{x}_1)$. Because the diffusion step is Markovian, each conditional at time t only depends on previous timestep $t-1$, hence, $q(\mathbf{x}_{1:T}) = q(\mathbf{x}_1)q(\mathbf{x}_2|\mathbf{x}_1)q(\mathbf{x}_3|\mathbf{x}_2)\dots q(\mathbf{x}_T|\mathbf{x}_{T-1})$. Thus, the conditional term is

$$q(x_{1:T}|\mathbf{x}_0) = q(\mathbf{x}_1|\mathbf{x}_0)\prod_{t=2}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1}, \mathbf{x}_0) \tag{211}$$

We assume a Markovian structure for the denoising distribution, so

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \tag{212}$$

This results in

$$\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(x_{1:T}|\mathbf{x}_0)} = \log \frac{p(\mathbf{x}_T)\prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_1|\mathbf{x}_0)\prod_{t=2}^{T} q(\mathbf{x}_t|\mathbf{x}_{t-1},\mathbf{x}_0)} \tag{213}$$

$$= \log \left[\frac{p_\theta(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(x_1|\mathbf{x}_0)} \prod_{t=2}^{T} \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1},\mathbf{x}_0)} \right] \tag{214}$$

$$= \log \frac{p_\theta(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(x_1|\mathbf{x}_0)} + \sum_{t=2}^{T} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1},\mathbf{x}_0)} \tag{215}$$

Hence,

$$\mathbb{E}_{q(x_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(x_{0:T})}{q(x_{1:T}|\mathbf{x}_0)}\right] = \mathbb{E}_{q(x_{1:T}|\mathbf{x}_0)}\left[\log \frac{p_\theta(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(x_1|\mathbf{x}_0)} + \sum_{t=2}^{T} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1},\mathbf{x}_0)}\right] \tag{216}$$

Now, we use Bayes' Theorem.

$$q(\mathbf{x}_t|\mathbf{x}_{t-1},\mathbf{x}_0) = \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)} \tag{217}$$

Therefore,

$$\sum_{t=2}^{T} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_t|\mathbf{x}_{t-1},\mathbf{x}_0)} = \sum_{t=2}^{T} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)q(\mathbf{x}_t|\mathbf{x}_0)} \tag{218}$$

$$= \sum_{t=2}^{T} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)} + \log q(\mathbf{x}_{t-1}|\mathbf{x}_0) - \log q(\mathbf{x}_t|\mathbf{x}_0) \tag{219}$$

Notice the last two terms $\log q(\mathbf{x}_{t-1}|\mathbf{x}_0), \log q(\mathbf{x}_t|\mathbf{x}_0)$ periodically cancel each other. This results in

$$\mathbb{E}_{q(x_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(x_{0:T})}{q(x_{1:T}|\mathbf{x}_0)}\right] = \mathbb{E}_{q(x_{1:T}|\mathbf{x}_0)}\left[\log \frac{p_\theta(\mathbf{x}_T)p_\theta(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)} + \sum_{t=2}^{T} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)} + \log q(\mathbf{x}_1|\mathbf{x}_0) - \log q(\mathbf{x}_T|\mathbf{x}_0)\right] \tag{220}$$

$$= \mathbb{E}_{q(x_{1:T}|\mathbf{x}_0)}\left[\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T|\mathbf{x}_0)} + \log p_\theta(\mathbf{x}_0|\mathbf{x}_1) + \sum_{t=2}^{T} \log \frac{p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)}\right] \tag{221}$$

So by definition of KL divergence we have

$$\log p(\mathbf{x}_0) \geq \underbrace{\mathbb{E}_{q(\mathbf{x}_1|\mathbf{x}_0)}[\log p_\theta(\mathbf{x}_0|\mathbf{x}_1)]}_{\text{Reconstruction Term}} - \underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_T|\mathbf{x}_0)||p(\mathbf{x}_T))}_{\text{Prior Matching Term}} - \sum_{t=2}^{T} \underbrace{\mathbb{E}_{q(\mathbf{x}_t|\mathbf{x}_0)}\left[D_{\mathrm{KL}}(q(\mathbf{x}_{t-1}|\mathbf{x}_t,\mathbf{x}_0)||p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t)\right]}_{\text{Denoising Matching Term}} \tag{222}$$

9.4 Training

Algorithm 2. Diffusion Training Algorithm with Gradient Descent

while not converged **do**
  $\mathbf{x}_0 \sim q(\mathbf{x}_0)$
  $t \sim \text{Uniform}(\{1, \dots, T\})$
  perform a gradient descent step on

$$\nabla_\Theta \|\epsilon - \epsilon_\Theta(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon\mathbf{x}_0, t)\|^2$$

end while

Algorithm 3. Diffusion Training Algorithm with Gradient Descent

$\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
for $t = T, \dots, 1$ **do**
  $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ if $t > 1$, else $\mathbf{z} = \mathbf{0}$
  Given $\sigma_t^2 = \beta_t$

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)\right) + \sigma_t\mathbf{z}$$

end for
return $\mathbf{x}_0$

9.5 Guidence

So far we modeled only the data distribution $p(x)$. However, we often also require the conditional distribution $p(x|y)$ which enables the explicit control over the generated data. In practice, we would like to sample from a distribution from

$$p_\Theta(\mathbf{x}_{0:T}|y) = p(\mathbf{x}_T)\sum_{t=1}^{T} p_\Theta(\mathbf{x}_{t-1}|\mathbf{x}_t, y) \tag{223}$$

where y is a conditioning variable such as an image generated with DALL-E 2 giving in input the caption: "An astronaut riding a horse in a photorealistic style." The tool used for conditioning on images is called CLIP.

CLIP CLIP is a large image-language model trained and distributed by OpenAI. It has been trained on a dataset of paris one consisting of an image and its textual description. CLIP consists of two encoder networks, one encodes images while other encodes textual caption. With a contrastive loss, CLIP is encouraged to encode an image and its caption into similar embedding vectors. One of the possible applications of CLIP is Zero-Shot classification, which leverages the CLIP model to predict the class of an image without any training, by predicting the classes that maximizes the cosine similarity between the image and the class name.

Classifier Guidance We can guide a diffusion model via Classifier Guidance. The idea is to guide denoising in a direction favouring images that are more reliably classified. To do that, we need a pre-trained unconditional diffusion model and a classifier trained from scratch on noisy images. Then, we inject gradients of the classifier model into sampling process.

Even if intuitively this direction seems reasonable, in practice it counts some limitations. First, it requires a specific classifier trained on noisy data, as we want to guide the generation during all the levels of noise of the process. Second, it is hard to interpret what the classifier guidance is doing.

Classifier-Free Guidance Jointly train class-conditional and unconditional diffusion models and to guide the generation process in the direction of conditioning

$$\epsilon^*_\Theta(\mathbf{x}, c; t) = (1+\rho)\underbrace{\epsilon_\Theta(\mathbf{x}, c; t)}_{\text{Conditional}} - \underbrace{\rho\epsilon_\Theta(\mathbf{x}; t)}_{\text{Unconditional}} \tag{224}$$

where c is the conditioning variable (usually a text's CLIP encoding) and ρ is a hyperparameter that controls the strength of the guidance. In practice, we usually train a single model $\epsilon_\Theta(\mathbf{x}, c, ; t)$ and just set conditioning variable c to zeros for the unconditional generation.

Classifier-Free Guidance is in practice easier to implement since it does not rely on the training of any additional classification model. However, the generation is twice slower when compared to the Classifier Guidance, at each step it has to run both the conditional and the unconditional generation. From an overall perspective, guidance improves fidelity (how good/convincing the image looks) but reduces the diversity of the generated images.

Part III

Deep Learning for Computer Vision

10 Implicit Surfaces and Neural Radiance Fields

The *universal approximation theorem* tells that neural networks are able to learn an approximation of any continuous function. Thus, as a 3D shape is a continuous function, we should not be surprised that neural networks are able to solve it.

Voxels Voxels are 3D correspondent of pixel, a democratization of 3D space into grid. It occupies cubic memory $\mathcal{O}(n^3)$, thus **resolution is limited**.

Points or Volumetric Primitives These are discretization of surface into 3D points. However, it **does not model connectivity / topology**.

Meshes Discretization into vertices and faces. Requires discretization into vertices or faces. Requires either class-specific templates or the maximum number of vertices to represent it. There will **always be an approximation error** and they lead to self-intersections.

Implicit functions Learns the analytic function which represents the 3D-surface. There are **no approximation error** thus **smooth and continuous surface**. However, a graphical visualization is not directly applicable, unless we convert them to aforementioned explicit representation. In any case, it is **hard to obtain high frequency details**.

10.1 Neural Implicit Surface Representation

Represent surface as the level-set of a continuous function. The form is

$$f(x) = x_1^2 + x_2^2 + x_3^2 - r^2 \tag{225}$$

$$S = \{x | f(x) = 0\} \tag{226}$$

These are two kind of function that can do this work:

- Occupancy Networks: $f_\theta : \mathbb{R}^3 \times \mathcal{X} \mapsto [0, 1]$ outputs the probability of being inside the surface.

- DeepSDF (Signed Distance Field): $f_\theta : \mathbb{R}^3 \times \mathcal{X} \mapsto \mathbb{R}$ output the signed distance from the surface (negative if inside, positive if outside).

Pros

- The representation is continuous
- We obtain an arbitrary topology and resolution
- Low memory footprint

10.2 Implementation of Neural Implicit Surface Representations

The implicit function f_θ is parameterized as an MLP. We condition over the type of shapes we want to obtain via input concatenation. The output can be both occupancy probability or Signed Distance Field. In order to obtain an explicit representation, we then use marching cubes.[11] In general, we can choose one of these three representations as ground truth to learn the function f.

[11] Converts the implicit function representation to a mesh for visualization. Extracts a polygonal mesh from the continuous function by evaluating the function on a grid and connecting points with isosurfaces.

- Watertight Meshes
- Point Clouds
- 2D Images

Watertight Meshes This is the simplest case (they have no holes thus the space is divided in inside and outside): we can uniformly sample points inside the surface and we train the model using BCE:

$$\mathcal{L}(\theta, \phi) = \sum_{j=1}^{K} \text{BCE}(f_\theta(p_{ij}, z_i), o_{ij}) \tag{227}$$

Point Clouds Point clouds are a collection of data points defined in a three-dimensional coordinate system. These points represent the external surface of an object or a scene in 3D space and are commonly used in various fields such as computer graphics, computer vision, and robotics.

- **Representation**: A point cloud is represented by a set of points, each defined by its x, y, z coordinates. Optionally, each point can also carry additional information such as color, intensity, or normal vector (indicating surface orientation).
- **Generation**: Point clouds are often generated by 3D scanning devices such as LiDARs, laser scanners, and depth cameras. They can also be created from photogrammetry, where 3D structure is inferred from multiple 2D images.
- **Data Structure**: Typically stored as lists or arrays of points (x, y, z)

We use this representation sometimes because:

- Many 3D sensors output unordered point clouds
- Generally, they are cheaper to obtain than watertight meshes

Input: points cloud $\mathcal{X} = \{x_i\}_{i \in I} \subset \mathbb{R}^3$. In this case the implicit function represents the **signed distance** function to a plausible surface $\mathcal{M}$ define by $\mathcal{X}$ since learning only from points would be hard. Therefore, the **loss** we wish to minimize is

$$\mathcal{L}(\theta) = \sum_{i \in I} \underbrace{|f_\theta(x_i)|^2}_{\text{Vanish Term}} + \underbrace{\lambda \mathbb{E}_x (\|\nabla_x f_\theta(x)\| - 1)^2}_{\text{Eikonal Term}} \tag{228}$$

Where we want the loss to vanish at training points, hence the *Vanish term*. We want the spacial gradient at surface points to be 1, so we can interpret it as a geometric surface (we do not want sudden changes in the norm of the gradient), encourages smoothness defined by *Eikonal term*.

Convergence and Linear Reproduction
Gradient descent of the linear model with random initialization converges with probability 1 to the reproducing plane.

2D Images Now suppose we want to construct a 3D representation with only 2D images (no more 3D supervision). In order to learn from them we need to render them in a differentiable way. We achieve this via **Differentiable Volumetric Rendering**. We quickly review the Secant Method.

Secant Method
In order to find the points which lay on the surface, we use the secant method. The idea is to start from 2 points x_0, x_1 and connect them with a striaght line. Find the intersection of this line with the x-axis, call this point x_2. Repeat this until convergence using point with opposite signs.

Consider the line $\Big(x_0, f(x_0)\Big) \to \Big(x_1, f(x_1)\Big)$:

$$y_2 = \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x_2 - x_1) + f(x_1) \tag{229}$$

The **root** of the function $y_2 = 0$ is:

$$x_2 = x_1 - f(x_1)\frac{x_1 - x_0}{f(x_1) - f(x_0)} \tag{230}$$

Iteratively applying this rule yields the secant method which is a finite-difference approximation of Newton's method.

Now we can move to the forward pass (rendering).

Differentiable Volumetric Rendering
Our goal is to learn f_θ (occupancy function) and t_θ (texture) from 2D image observations. Consider a single image observation. We define a photometric reconstruction loss

$$\mathcal{L}(I, \hat{I}) = \sum_u \|\hat{I}_u - I_u\| \tag{231}$$

where I is the observed image (Ground Truth) and $\hat{I}$ is rendered by our implicit model. Moreover, I_u denotes the RGB value of the observation I at pixel u and $\|\cdot\|$ is a (robust) photo-consistency measure such as the ℓ_1-norm.

To minimize the reconstruction loss $\mathcal{L}$ w.r.t the network parameters θ using gradient-based optimization techniques, we must be able to

- Render $\hat{I}$ given f_θ ($f_\theta = \tau$ if the point is in the surface, $> \tau$ if the point is behind and $< \tau$ if it is outside) and t_θ
- Compute gradients of gL w.r.t. the network parameters θ

To obtain the **rendering**, we follow this procedure. Given $\mathbf{r}_0$, the position of the camera, for each pixel u:

1. Draw $\mathbf{w}$, a vector connecting $\mathbf{r}_0$ to u
2. Consider the ray $r(d) = \mathbf{r}_0 + d\mathbf{w}$
3. Call $\hat{\mathbf{p}}$ the first point of intersection (where $f_\theta(\hat{\mathbf{p}}) = \tau$, found with Secant Method) with the estimated surface in the direction of $r(d)$. Call $\hat{d}$ the distance of this point from $\mathbf{r}_0$, in particular, we know that $r(\hat{d}) = \hat{\mathbf{p}}$
4. Query the texture network and obtain $t_\theta(\hat{\mathbf{p}})$
5. Colour the pixel u with colour $t_\theta(\hat{\mathbf{p}})$.

Now lets introduce the **backward pass**. Lets call I the real image and $\hat{I}$ the predicted one where we define the loss as $\mathcal{L}(\hat{I}, I) = \sum_u \|\hat{I}_u - \hat{I}\|$. The gradient of the loss w.r.t our parameters will be

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_u \frac{\partial \mathcal{L}}{\partial \hat{I}_u} \frac{\partial \hat{I}_u}{\partial \theta} \tag{232}$$

where

$$\frac{\partial \hat{I}_u}{\partial \theta} = \frac{\partial t_\theta(\hat{\mathbf{p}})}{\partial \hat{\mathbf{p}}} \frac{\partial \hat{\mathbf{p}}}{\partial \theta} \tag{233}$$

In order to evaluate $\frac{\partial \hat{\mathbf{p}}}{\partial \theta}$ we need implicit differentiation.

Consider the ray $\hat{\mathbf{p}} = \mathbf{r}_0 + \hat{d}\mathbf{w}$ and condition for the intersection between the ray and the surface (remember that we evaluate the colour

of a point only for the points on the surface) and take the derivative on both sides.

$$f_\theta(\hat{\mathbf{p}}) = \tau \tag{234}$$

$$\frac{\partial f_\theta(\hat{\mathbf{p}})}{\partial \theta} + \frac{\partial f_\theta(\hat{\mathbf{p}})}{\partial \hat{\mathbf{p}}}\frac{\partial \hat{\mathbf{p}}}{\partial \theta} = 0 \tag{235}$$

$$\frac{\partial f_\theta(\hat{\mathbf{p}})}{\partial \theta} + \frac{\partial f_\theta(\hat{\mathbf{p}})}{\partial \hat{\mathbf{p}}} \cdot \mathbf{w}\frac{\partial \hat{d}}{\partial \theta} = 0 \tag{236}$$

$$\frac{\partial \hat{d}}{\partial \theta} = -\Big(\frac{\partial f_\theta(\hat{\mathbf{p}})}{\partial \hat{\mathbf{p}}} \cdot \mathbf{w}\Big)^{-1}\frac{\partial f_\theta(\hat{\mathbf{p}})}{\partial \theta} \tag{237}$$

As $\hat{\mathbf{p}} = \mathbf{r}_0 + \hat{d}\mathbf{w}$ we have that

$$\frac{\partial \hat{\mathbf{p}}}{\partial \theta} = \mathbf{w}\frac{\partial \hat{d}}{\partial \theta} \tag{238}$$

$$= -\mathbf{w}\Big(\frac{\partial f_\theta(\hat{\mathbf{p}})}{\partial \hat{\mathbf{p}}} \cdot \mathbf{w}\Big)^{-1}\frac{\partial f_\theta(\hat{\mathbf{p}})}{\partial \theta} \tag{239}$$

10.3 NEural Radiance Field (NERF)

So far we have learnt how to represent surfaces, but in some cases this is not enough, scenes are more complex. In particular, we have to learn *thin structures* (e.g. leaves, hair), *transparency* (e.g. glasses, smoke).

> Compared with implicit surfaces, they can model transparency and thin structure, and therefore is a more flexible representation. **However**, it generally leads to worse geometry compared to implicit surface.

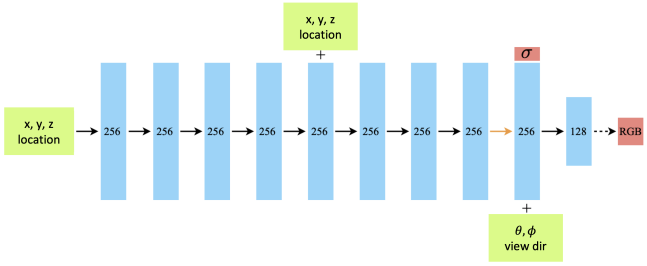

Figure 7. NERF architecture

Architecture Before we were interested in one single output, the RGB value of a pixel. The novelty of NERF is that they introduce the concept of density σ which enables us to learn more about difficult surfaces. In particular, the **input** is x, y, z: The 3D position of the point we are considering and θ, ϕ: the camera parameters. The **output** is σ, the density of the point and c, the RGB value of the point. Note that the view directories θ and ϕ are given to the network only in later layers,

after having predicted σ to enforce this value not to be dependent on ϕ, θ, but just from x, y, z. After some layers we are given again the position x, y, z to the network to make sure it has not been washed out.

Procedure We want to analyze how we can obtain the volume rendering. The first step is to draw a ray connecting the camera position to the point we want to represent. However, we now analyze the whole ray sampling points along it, without stopping at the first intersection with the surface. The parameters we analyze are

- The density σ
- The transmittancy $T_i = \prod_{j=1}^{i-1}(1 - \alpha_i)$

In order to get the colour we apply **alpha compositing**. To better understand the process, consider the following formula:

$$\delta_i = t_{i+1} - t_i \tag{240}$$

$$\alpha_i = 1 - \exp\{-\sigma_i \delta_i\} \tag{241}$$

The final colour will be the weighted average of the colours along the way

$$c = \sum_{i=0}^{N} T_I \alpha_i c_i \tag{242}$$

Since the sampling operation is very expensive, one trick is to sample more n more significant positions (i.e. positions with high weights (high $T_i \alpha_i$)).

Positional Encoding Despite the fact that neural networks are universal function approximators, having the network F_Θ directly operate on x, y, z, θ, ϕ input coordinates result renderings that perform poorly at representing high-frequency variation in colour and geometry. This happens because neural networks are biased towards learning lower frequency function.

Solution to poor high-frequency variation
Introduce positional encoding, mapping the inputs to a higher dimensional space $\mathbb{R}^{2L}$ and then applying the MLP function. Formally, define the encoding function used is

$$\gamma(p) = \Big[\sin(2^0 \pi p), \cos(2^0 \pi p), \cdots, \sin(2^{L-1} \pi p), \cos(2^{L-1} \pi p)\Big] \tag{243}$$

This function $\gamma(\cdot)$ is applied separately to each of the three coordinate values in $\mathbf{x}$ (which are normalized to lie in $[-1, 1]$) and to the three components of the Cartesian viewing direction unit

vector $\mathbf{d}$ (which by construction lie in $[-1, 1]$). In the original NERF paper experiments, we set $L = 10$ for $\gamma(\mathbf{x})$ and $L = 4$ for $\gamma(\mathbf{d})$.

A similar mapping is used in the popular Transformer architecture, where it is referred to as a *positional encoding*. However, Transformers use it for a different goal of providing the discrete positions of tokens in a sequence as input to an architecture that does not contain any notion of order. In contrast, we use these functions to map continuous input coordinates into a higher dimensional space to enable our MLP to more easily approximate a higher frequency function. Concurrent work on a related problem of modeling 3D protein structure from projections also utilizes a similar input coordinate mapping.

Limitations of NERFs

- Requires many (50+) calibrated views
- Slow rendering speed for high-res images
- Only models static scenes

10.4 Gaussian Splatting

The input to this method is a set of images of a static scene together with the corresponding cameras calibrated by SfM which produces a sparse point cloud as a side-effect. From these points, we create a set of 3D Gaussians defined by a position μ, covariance matrix Σ and opacity α that allows a very flexible optimization regime. This results in a reasonably compact representation of the 3D scene.

Optimization starts with the sparse SfM point cloud and creates a set of 3D Gaussians. Then, we optimize and adaptively control the density of this set of Gaussians. During optimization, we use our fast ti-ebased render, allowing competitive training times compared to SOTA fast radiance field methods. Once trained, the renderer allows real-time navigation for a wide variety of scenes.

Differentiable Rasterization of 3D Gaussians (splatting)

- For each pixel (in practice, approximated by tile-based sorting), sort all Gaussians according to their depth.
- Calculate the opacity of each Gaussian

$$\alpha = o \cdot \exp(-0.5(x - \mu')^\top \Sigma'^{-1}(x - \mu')) \tag{244}$$

- Use volume rendering equation, similary to NeRFs

$$C = T_1\alpha_1 c_1 + T_2\alpha_2 c_2 + T_3\alpha_3 c_3 \tag{245}$$

  where $T_1 = 1$ and $T_2 = 1 - \alpha_1$, etc.

11 Parametric Human Body Models

2D human pose representation and estimation consists on two main fields of study that should be combined:

- Body modeling
- Feature representation learning

We will first analyze them separately and then we will understand how to combine them efficiently.

11.1 Body Modeling

Body modeling aims to find a way to understand how the different parts of the body are linked to each other. Pictorial Structure Model, which will be explained below, comes up with a graph based solution.

Pictorial Structure Model We describe the human body model as a graph $G = (V, E)$ where:

- $V = (1, \dots, k)$ are vertices to represent k parts of the human body
- E are edges to specify which pairs of parts are constrained to have consistent relations

Given an 2D image I, we indicated $l_i = (x_i, y_i)$ the estimated position of a vertex i in a 2D plane. Thus, given an image I and a configuration estimate $L = (l_1, \dots, l_k)$, which contains position estimations for each vertex, we can define a score as follows.

$$S(I, L) = \sum_{i \in V} \alpha_i \cdot \phi(I, l_i) + \sum_{i,j \in E} \beta_{ij} \cdot \psi(l_i, l_j) \tag{246}$$

where $\phi(I, l_i)$ is a unary term which measures the mismatch with the original image when the vertex is placed in the location l_i and $\psi(l_i, l_j)$ measures the deformation between the connected points i and j when i is placed in l_i and j is placed in l_j.

Pictorial Structure Model with Flexible Mixtures It has been proved empirically that a **mixture** of non-oriented pictorial structures can outperform **explicitly articulated parts** because mixture models can capture orientation-specific statistics of background features. Thus, we modify the framework previously discussed introducing the concepts of mixture models. Let m_i be the type (mixture component) of the part i. The mixture component can express many concepts as orientations of a part (vertical versus horizontally oriented hand), but types may span out-of-plane rotations (front-view head versus side-view head) or even semantic classes (an open versus closed hand).

Formally, the score becomes

$$S(I, L, M) = \sum_{i \in V} \alpha_i^{\mathbf{m}_i} \cdot \phi(I, l_i) + \sum_{i,j \in E} \beta_{ij}^{\mathbf{m}_i, \mathbf{m}_j} \cdot \psi(l_i, l_j) + \mathbf{S}(\mathbf{M}) \quad (247)$$

where $\alpha_i^{\mathbf{m}_i}$ is the local appearance template for part i with type assignment m_i, $\beta_{ij}^{\mathbf{m}_i, \mathbf{m}_j}$ is the spatial spring parameter for pair of types (m_i, m_j). It 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. $\mathbf{S}(\mathbf{M})$ is the co-occurence bias and is defined as

$$\mathbf{S}(\mathbf{M}) = \sum_{ij \in E} b_{ij}^{m_i m_j} \quad (248)$$

where b_{ij} is the **pairwise co-occurrence prior** between part i with mixture type m_i and j with mixture type m_j and it favours particular co-occurrences of part types.

11.2 Feature Representation Learning

We will explore *Direction Regression* and *Heatmaps* for feature representation learning.

Direct Regression Direct Regression is based on deep convolutional neural networks. The idea is to directly regress x and y coordinates and it involves a refiner.

Heatmaps For heatmaps-based representation learning we refer to Convolutional Pose Machines. The objective is to improve performance of human pose estimation when occlusions are present. The main idea is to create seperate heatmaps (Gaussian distribution around keypoints) for each keypoint, and then combining them only in the last phase. The key is that at every stage, the architecture operates both on image evidence as well as belief maps from preceding stages. In each stage, the computed beliefs provide an increasingly refined estimate for the location of each part.

11.3 3D Human Pose Representation and Estimation

SMPL representation: 3D Mesh In order to represent the body in 3D, we use a 3D mesh that is designed by an artist and contains around 7000 vertices. In order to define a body we need to define its **shape** and **pose**.

Shape In order to define the shape, we do PCA of meshes in canonical pose to estimate the directions of maximal shape variation. Doing that,

we obtain a low-dimensional subspace (10D - 300D) in the canonical pose. Usually 10 dimensions are enough to define a pose.

Pose: Linear blend skinning The linear mesh skinning is the simplest mesh skinning method. Linear blend skinning is the idea of transforming vertices inside a single mesh by a (blend) of multiple transforms. Deformed position of a point (vertex) is a **sum** of the positions determined by each bone's transform alone, **weighted** by that vertex's weight for that bone. In particular, for each vertex i, starting from a rest position t_i, its position in the transformed pose t'_i is

$$t'_i = \sum_k w_{k_i} \mathbf{G}_k(\theta, \mathbf{J}) \mathbf{t}_i \tag{249}$$

where w_{k_i} are the blend skinning weights, created by an artist. $\mathbf{G}_k$ is a rigid bone transformation, θ is the desired pose, $\mathbf{J}$ are the joint locations. Thus, posed vertices are linear combination of transformed template vertices. It is simple fast to compute, but it produces only well known artifacts.

Pose: SMPL A solution to the above problem is SMPL, where t'_i is computed as

$$t'_i = \sum_k w_{ki} \mathbf{G}_k(\theta, J(\beta))(t_i + s_i(\beta) + p_i(\theta)) \tag{250}$$

where $s_i(\beta)$ is the vertex i in $B_s(\beta)$, which represents offset from the template depending on the shape described by β. $p_i(\theta)$ is the vertex i in $B_P(\theta)$, which represents offset from the template depending on the pose described by θ.

Shape blend shapes B_S These are the body shapes of different people represented by a linear function B_S

$$B_S(\beta, \mathcal{S}) = \sum_{n=1}^{|\mathcal{S}|} \beta_n S_n \tag{251}$$

where $\beta = [\beta_1, \ldots, \beta_{|\beta|}]^\top$ are the linear shape coefficients, $S_n \in \mathbb{R}^{3N}$ are the orthonormal principle components of shape displacements, $\mathcal{S} = [S_1, \ldots S_{|\beta|}]$ is the matrix of all shape displacements learned from registered training meshes. Notationally, the values to the right of a semicolon represent learned parameters, while those on the left are parameters set by an animator.

Pose blend shapes B_P We denote $R : \mathbb{R}^{|\theta|} \mapsto \mathbb{R}^{9K}$ a function that maps a pose vector θ to a vector of concatenated part relative rotation relative rotation matrices (each rotation matrix has dimensions 3×3). Given

that our rig has 23 joints, we have that $K = 3$ and thus $R(\theta)$ is a vector of length $23 \times 9 = 207$. Elements of $R(\theta)$ are functions of sines and cosines of joint angles and therefore $R(\theta)$ is non-linear with θ. If we define θ^* as the rest pose, the the vertex deviations from the rest template are

$$B_P(\theta, \mathcal{P}) = \sum_{n=1}^{9K} (R_n(\theta) - R_n(\theta^*))\mathbf{P}_n \tag{252}$$

where $\mathbf{P}_n \in \mathbb{R}^{3N}$ are the vector of vertex displacements. Thus, $\mathcal{P} = [\mathbf{P}_1, \dots \mathbf{P}_{9K}] \in \mathbb{R}^{3N \times 9K}$ is a matrix of all 207 pose blend shape.

As a consequence of this formula, the rotation of a particular joint can influence all the body vertices, not only the local ones.

As a result, we obtain a **mesh** $\mathcal{M}(\beta, \theta, \phi)$ which depends on β (shape), θ (pose), ϕ (gender) and the **joints position** $J(\beta; \mathcal{J}, \bar{T}, \mathcal{S})$.

Part IV

Reinforcement Learning

12 *Reinforcement Learning*

In Reinforcement Learning (RL) models learn how to act interacting with the environment trough some actions. This can be useful in many fields such as games, logistics and operations and Robot Control/Computer Vision. Reinforcement Learning is a problem, not a method. Given an unknown and uncertain environment, it aims to choose the right actions in order to maximize the reward signal in the long-term.

An RL agent may include one or more of these components:

Policy A policy expresses the agent's behaviour, it is a map from states to action. It can be:

- **Deterministic**: $\pi(s)$, it returns the precise action to do given a state $s \in S$
- **Stochastic** $\pi(a|s) = \mathbb{P}(a|S_t = s)$ it returns the probability of doing the action a in the state s

There are three types of RL agents.

Value Based The agent **has access to the value function**. Given a value function, we can derive a greedy policy by reading the value function and maximizing the best action.

Policy Based The agent have just access to a policy and try to adjust this policy directly to get the highest possible reward.

Model-free and model-based agents Model-free directly optimize value/policy function while model-based first builds a model how the enviroment works and then finds the optimal way to behave.

12.1 *Markov Decision Processes*

Markov Decision Processes (MDPs) formally describe an enviroment for reinforcement learning where the enviroment is fully observable.

Markov Property *The future is independent of the past given the present.*

A state S_t is Markov if and only if

$$\mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1, \ldots, S_t] \tag{253}$$

As a consequence, the state captures all relevant information from the history. Once the state is known, the history may be thrown away.

A **Markov Process** (or Markov Chain) is a tuple $\langle \mathcal{S}, \mathcal{P} \rangle$ where

- $\mathcal{S}$ is a (finite) set of states
- $\mathcal{P}$ is a state transition probability matrix, where $\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$

A **Markov Reward Process** is a tuple $\langle \mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where

- $\mathcal{R}$ is a reward function $\mathcal{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$
- $\gamma \in [0, 1]$ is a discount factor, where γ close to 0 leads to myopic evaluation and close to 1 leads to far-sighted evaluation.

Return G_t is the total discounted reward from time-step t.

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k} \tag{254}$$

Markov Decision Process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ where

- $\mathcal{S}$ is the set of states
- $\mathcal{P}$ is the set of actions
- $\mathcal{R}$ is a reward function $r : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}$ (deterministic or distribution)
- $\mathcal{P}$ is a transition function $p : \mathcal{S} \times \mathcal{A} \mapsto \mathcal{S}$ (deterministic or distribution)
- $s_0 \in \mathcal{S}$ is an initial state
- γ is a discount factor

Value Function The value function is a prediction of the expected future reward. It is used to evaluate the goodness/badness of states. It is the bases the agent uses to decide the next action. We now derive the **bellman equation**[12] for $V_\pi(s)$

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{255}$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{256}$$
$$= \sum_a \underbrace{\pi(a|s)}_{*} \sum_{s'} \sum_r \underbrace{p(s', r|s, a)}_{**} \underbrace{(r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'])}_{***} \tag{257}$$
$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)(r + \gamma v_\pi(s')) \tag{258}$$

[12]
* is the probability of taking an action given a state
** is the joint probability of next state and the reward given the current state and action. Transition Matrix.
*** is the expected reward given a new state

Action-value function The action-value function $q_\pi(s, a)$ is the expected return starting from state s, taking action a then following policy π

$$q(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{259}$$
$$= \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_{t=s}, A_t = a] \tag{260}$$

Bellman Optimality Equation With this, we can have the **Bellman Optimality Equation**, which says $v_*(s)$ is the maximum value function over all policies.

$$v_*(s) = \max_\pi v_\pi(s) = \max_a q_*(s, a) = \max_a \sum_{s'} p(s', r|s, a)(r + \gamma v_*(s')) \tag{261}$$

Notice that this equation is *not linear*, has *no closed form solution* but can be solved using many iterative solution methods such as DP, Monte-Carlo Methods, Temporal-Difference Learning (combination of DP and MC).

12.2 Dynamic Programming

Dynamic Programming (DP) is able to compute optimal policies given a perfect model of the world (MDP). Thus, in order to apply DP we need to know the transition probabilities. This has a limited utility, but still has great theoretical importance. The two key ideas to compute the optimal policy

- Value iteration: 1. Compute optimal v_* using the value iteration algorithm, 2. Find a policy π to obtain v_*
- Policy iteration: 1. For any policy π compute $v(\pi)$, 2. Update policy π given $v(\pi)$ and obtain π', 3. Iterate until $\pi \sim \pi'$

Algorithm 4. Value Iteration Algorithm

$i \leftarrow 0$
$V_0 \leftarrow 0$
while $\Delta > \theta$ **do**
    $\Delta \leftarrow 0$
    for $s \in S$ **do**
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_{a \in A} \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|$
    end for
end while
$\pi_i \leftarrow$ greedy policy from V_i

Value Iteration (Algorithm 4)
Pros:

- Exact Methods
- Policy/Value iteration are guaranteed to converge in finite number of iterations
- Value iteration is typically more efficient than policy iteration

Cons:

- Need to know the transition probability matrix
- Need to iterate over the whole state space (very expensive)
- Requires memory proportional to the size of the state space

12.3 Monte Carlo Methods

If the state space is too big to iterate over it and if we don't know the transition probabilities we can use the idea of Monte Carlo methods. Monte Carlo policy evaluation uses empirical mean return instead of expected return.

$$V_\pi(s_t) = \mathbb{E}_{a_t \sim \pi(s_t), s_{t+1} \sim p(s_t, a_t)} \sum_{i=0} \gamma^i r(s_{t+i}, a_{t+i}) \tag{262}$$

We can just run episodes with the given policy and compute samples of the expression in the expectation:

$$\sum_{i=0} \gamma^i r(s_{t+i}, a_{t+i}) \tag{263}$$

12.4 Temporal Difference Learning

Temporal Difference (TD) learning allows learning from incomplete episodes. We don't have anymore to go all the way in a particular

trajectory, TD can learn before knowing the final outcome, it learns online at every step.

Intuitively, the procedure is the following:

1. Guess the reward of a certain trajectory
2. Go one step forward in that trajectory
3. Estimate again the reward from the new state
4. Come back and update the previous estimate of the initial state

More formally, for each step with action a from s to s' that we take with our policy, we compute the difference to our current estimate and update our value function:

$$\Delta V(s) = r(s,a) + \gamma V(s') - V(s) \tag{264}$$

$$V(s) \leftarrow V(s) + \alpha \Delta V(s) \tag{265}$$

here $\alpha > 0$ is the learning rate. The TD(0) learning is guaranteed to converge to $v_\pi(s)$, the real value. Observe that we don't update the whole state space, but only visited states! However, we still need to find a criterion to visit the state space. There are two options:

- **Random Policy**: In each state, choose an action randomly
- **Greedy Policy**: In each state, always choose the best action

At a first glance, the greedy strategy may look better, but in practice it could get us stuck in local minima. We need to find the balance between

- **Exploration**: Gather more data to avoid missing out on a potentially large reward?
- **Exploitation**: Stick with out current knowledge and build an optimal policy for the data we've seen?

A good trade-off is the ϵ-greedy policy, where in each state, with a small probability ϵ we choose greedy. Works well in practice, where we decrease the value of ϵ so we have more exploration in the beginning.

On policy: Computes the Q-Value according to a policy and then the agent follows that policy.

Off-Policy: Computes the Q-Value according to a greedy policy, but the agent follows a different exploration policy.

There are to main major implementations of TDL.

SARSA (On policy) We follow the policy π to obtain a transition s, a, r, s' so we compute the difference to our current estimate and update our value function as

$$\Delta Q(s,a) = R_{t+1} + \gamma Q(s',a') - Q(s,a) \tag{266}$$

$$Q(s,a) \leftarrow Q(S,A) + \alpha \Delta Q(s,a) \tag{267}$$

where α is the learning rate, and action a' is chosen by π in the state s'.

Q-Learning (Off-Policy) For each action A transitioning from S to S', compute the difference to current estimate and update the value function

$$\Delta Q(S,A) = R_{t+1} + \gamma \max_a \{Q(S',a)\} - Q(S,A) \tag{268}$$

$$Q(S,A) \leftarrow Q(S,A) + \alpha \Delta Q(S,A) \tag{269}$$

Pro and Cons of TD Learning Pros

- Less variance than Monte Carlo Sampling due to bootstrapping
- More sample efficient than Dynamic Programming
- Do not need to know the transition probability matrix

Cons

- Biased due to bootstrapping, we use "old" value estimates as labels
- Exploration/Exploitation dilemma

13 Deep Reinforcement Learning

Recall that $\pi : \mathcal{S} \mapsto \mathcal{A}$ and $v_\pi : S_t \mapsto \mathbb{R}$ are functions, and thus can be approximated by a neural network. We use this idea (function approximation) to learn the value function.

13.1 Deep Q-Learning

Recall that in Q-Learning, we assign a value at each pair (a, s), thus, our goal is to use function approximation to learn the value function

$$v_\pi(s) \approx v_\pi(s, \theta) \tag{270}$$

We can use a neural network to learn the mapping between state-action pairs (s, a) and their values. The Q-learning update reduce to SGD on the TD-error (ΔQ)

$$\mathcal{L}(\theta) = (R + \gamma \max_{a'}\{Q_\theta(s', a')\} - Q_\theta(s, a))^2 \tag{271}$$

However, we still have a problem: SGD assumes that our updates are i.i.d.. In RL, states visited in the trajectory are strongly correlated. To address this, we use a **replay buffer** to store the generated samples. The procedure for **training** will be

1. Run some exploration policies and, during them, store all the generated samples there.
2. When we have enough transitions, we sample a random minibatch from the buffer
3. For every transition present in this minimatch, we update loss and parameters
4. Iterate until convergence

13.2 Policy Search Methods

Q-learning is limited to discrete action spaces, for continuous action space the problem is intractable. Policy search methods learn a policy π directly and often much easier. The algorithm directly learns the correct behavior, without exploring the value function.

Policy Gradients We can see the policy from a particular time step t as a normal distribution of mean μ_t and variances σ_t^2 over the possible actions, thus we use a Gaussian parameterization of the policy

$$\pi(a_t|s_t) \sim \mathcal{N}(\mu_t, \sigma_t^2) \tag{272}$$

The advantage of this parametrization is that now we can sample from there and learn the parameters of our network. In particular, if we

want the probability of a particular trajectory τ we have to compute

$$p(\tau) = p(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi(a_t|s_t) p(s_{t+1}|a_t, s_t) \tag{273}$$

Ideally, our objective is to make good trajectories more likely and bad trajectories less likely. In order to do that, we sample from the Gaussian parametrization of the policy and we learn our parameters to obtain $\pi(a_t|s_t, \theta)$.

Training: Exploration and Evaluation The learning phase can be split into two main parts:

- **Exploration**: Get the trajectory data. To do that, we sample action at every time-step from the policy probability distribution (on-policy methods)
- **Evaluation**: Evaluate the policy by computing the expectation of the trajectory reward given the parameters θ

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \Big[\sum_t \gamma^t r(s_t, a_t) \Big] \tag{274}$$

Optimization: Policy Update The goal is to maximize the performance measure

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, J(\theta) \tag{275}$$

In order to do that we update the parameters using gradient ascent:

$$\theta \leftarrow \theta + \nabla_\theta J(\theta) \tag{276}$$

First we rewrite $J(\theta)$ in a compact way

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t \gamma^t r(s_t, a_t) \right] \tag{277}$$

$$= \mathbb{E}_{\tau \sim p_\theta(\tau)} [r(\tau)] \tag{278}$$

$$= \int p(\tau) r(\tau) d\tau \tag{279}$$

Now we can compute the gradient and using the fact $\nabla \log(f(x)) = \frac{\nabla f(x)}{f(x)}$

$$\nabla_\theta J(\theta) = \int \nabla_\theta p(\tau) r(\tau) d\tau \tag{280}$$

$$= \int p(\tau) \nabla_\theta \log p(\tau) r(\tau) d\tau \tag{281}$$

$$= \mathbb{E}_{\tau \sim p_\theta(\tau)} [\nabla_\theta \log p(\tau) r(\tau)] \tag{282}$$

Lets write $\log p(\tau)$ as

$$\log p(\tau) = \log p(s_1, a_1, \ldots, s_T, a_T) \tag{283}$$

$$= \log \left[p(s_1) \prod_{t=0}^{T} \log \pi_\theta(a_t|s_t) p(s_{t+1}|a_t, s_t) \right] \tag{284}$$

$$= \log \Big[p(s_1) \Big] + \Big[\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) \Big] + \Big[\sum_{t=0}^{T} \log p(s_{t+1}|a_t, s_t) \Big] \tag{285}$$

The first and last term do no depend on the policy we choose, and thus do not depend on the parameters θ of our neural network. Thus, when we apply ∇_θ they will be constants and become 0. Hence[13],

[13] We use the fact $r(\tau) = \sum_{t=0}^{T} \gamma^t r(s_t, a_t)$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} [\nabla_\theta \log p(\tau) r(\tau)] \tag{286}$$

$$= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\nabla_\theta \Big[\sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) \Big] r(\tau) \right] \tag{287}$$

$$= \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\Big(\nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) \Big) \Big(\sum_{t=0}^{T} \gamma^t r(s_t, a_t) \Big) \right] \tag{288}$$

To compute the expectation in practice, we use the REINFORCE algorithm.

REINFORCE

1. Initialize the policy parameters θ at random.
2. Use this policy π_θ to collect a trajectory $\tau = (s_0, a_0, r_1, s_1, a_1, \ldots, a_H, r_{H+1}, s_{H+1})$
3. Calculate the discounted reward for each step k by backpropagation

$$G_k = \sum_{t=k+1}^{H+1} \gamma^{t-k-1} R_k = R_k + \gamma G_{k+1} \tag{289}$$

4. Calculate the expected reward J and ∇J_θ using Monte Carlo sampling (we sample N trajectories):

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left[\Big(\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t^i|s_t^i) \Big) G_0^i \right] \tag{290}$$

5. Adjust the weights of the policy to increase J

$$\theta \leftarrow \theta + \nabla_\theta J(\theta) \tag{291}$$

6. Iterate until convergence

REINFORCE with baseline REINFORCE has the problem of its gradient is estimated over only a few samples (we used Monte Carlo sampling), this the obtained policy gradients are very noisy. The solution

is to reduce the variance by introducing a baseline $b(s_t^i)$ in the term related to the trajectory reward.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\nabla_\theta \sum_{t=0}^{T} \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=0}^{T} \gamma^t r(s_t, a_t) - b(s_t^i) \right) \right] \tag{292}$$

Note that the **baseline must be a function that does not depend on the policy**. As a consequence *variance is reduced*, but the policy gradient estimate remains unbiased.