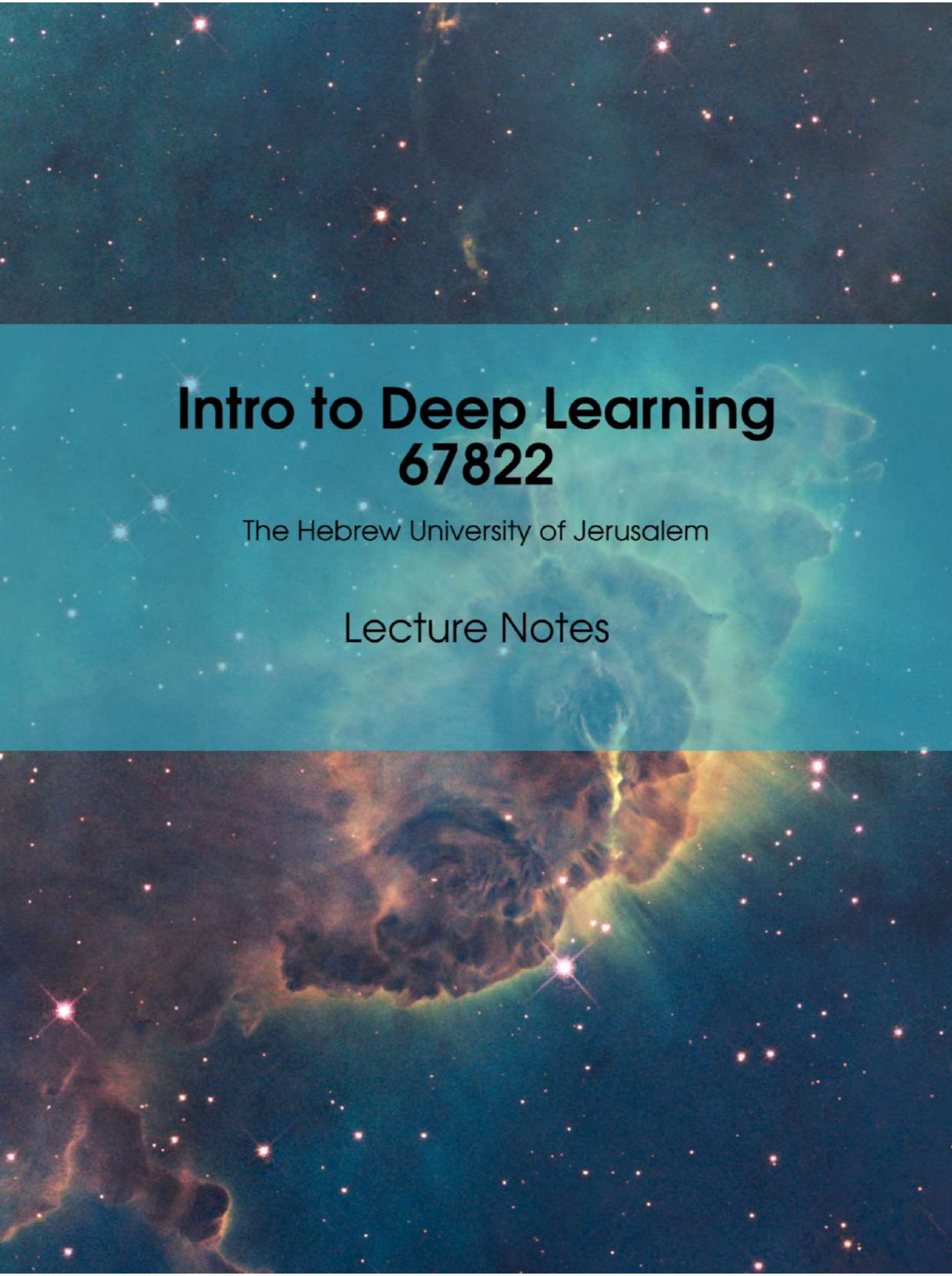


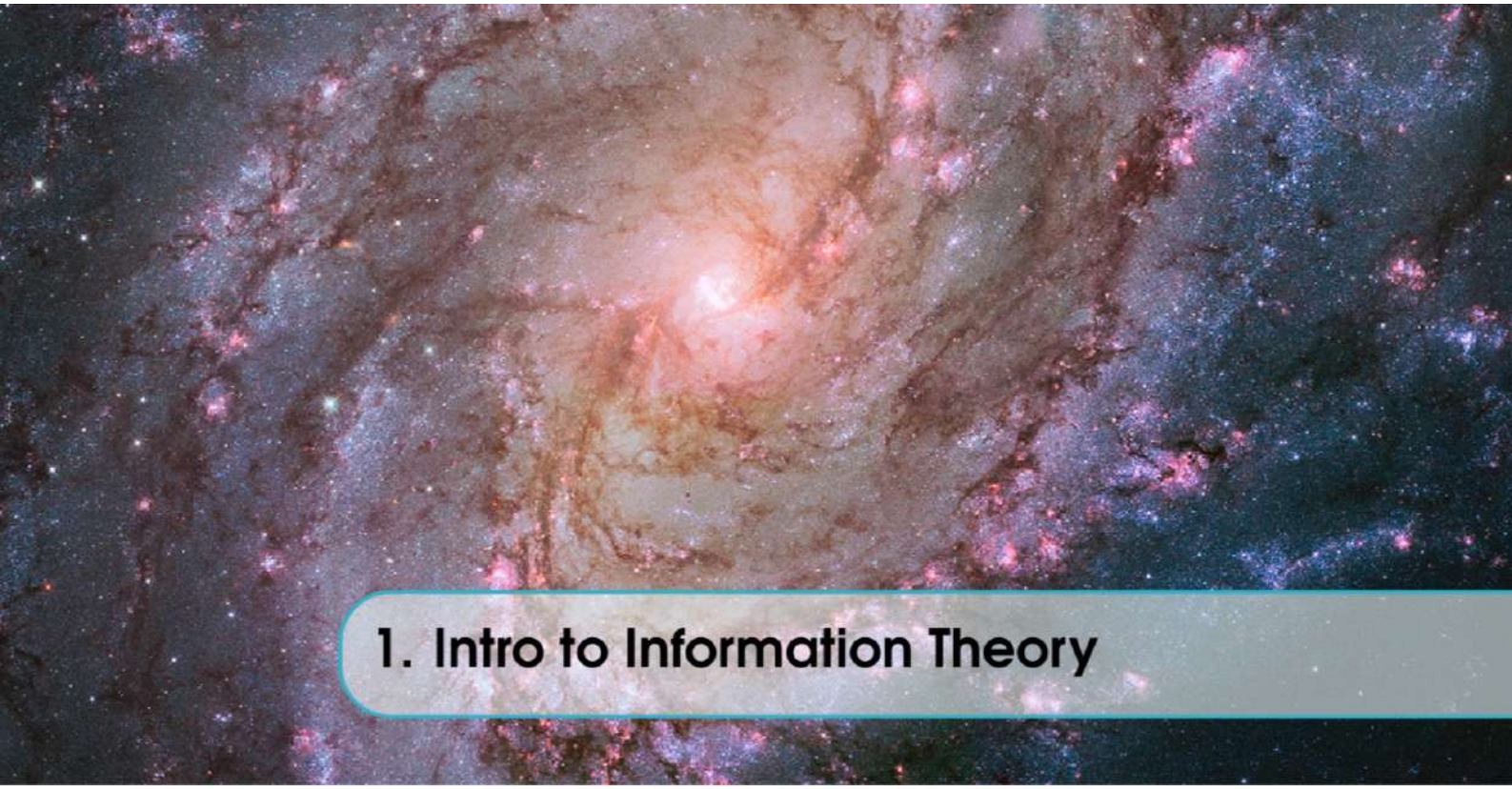
Intro to Deep Learning 67822

The Hebrew University of Jerusalem

Lecture Notes

WRITTEN BY: Hadar Sharvit
ALSO AVAILABLE ON GITHUB:
CONTACT ME AT: Hadar.Sharvit1@mail.Huji.ac.il
BASED ON: Lectures given by Raanan Fattal, and recitations given by Matan Halfon.
DISCLAIMER: the course does not intend to be formal, and nor does this summary. Having said that, I may elaborate on some topics I find important, and those cases will be marked with (†).
LAST UPDATE: January 22, 2022

Contents



1. Intro to Information Theory

Suppose we are given with data points $x \in X$, which are sampled from some unknown probability space P. One might ask - how would I find a somewhat similar probability space Q, that represents P well enough? in other words, what metric should I define to distinguish if Q resembles P? lucky for us, this has been studied long before neural networks walked the earth, and is related to the theory of information.

1.1 Entropy and Cross-Entropy

Definition 1.1.1 — Shannon's Entropy. for a discrete probability distribution p, we define the (Shannon) Entropy as

$$H(P) = -\sum_{x \in X} p(x) log_2(p(x)) = \mathbb{E}_{x \in X} log(1/p(x))$$

A "hand-wavy" definition to Entropy would be the amount of uncertainty a system has, that is - Entropy is an amount that represents how our data is unexpected when sampled.

▪ **Example 1.1** as a simple example, consider three boxes with 4 balls in each. denoting $b_1 = \{R,R,R,R\}, b_2 = \{R,R,R,B\}$ and $b_3 = \{R,R,B,B\}$. in b_1, it is easy to guess the outcome of sampling one ball, and in b_3 - not so much. when it is "easy" to guess the outcome, the entropy is low and vice versa. Notice how this relates to the definition of uncertainty - in b_1, everything is certain as all balls have the same color R (therefore the entropy is low). In $b3$ on the other hand the entropy is high, as the uncertainty is high (maximal in fact, as there is 50% chance for either R or B). ▪

going past the basic definition, the entropy also bounds from below the minimal number of bits that are needed to represent our data. for example, if our data is comprised of images, every image (sample) can only be compressed to a size that is exactly the entropy of our sample. What is the entropy of an image you ask? Well, we can define a probability function on some pixel to provide the chance to be equal to some grey-scale value. the summation of all the pixels would give the entropy of the image (but that's a topic for another time).

it is important to notice that as of now, we haven't answered the question that was presented in

the opening section. More specifically, the entropy is a function of a single probability space, and doesn't provide us with some way to compare two probability spaces as we wished for. For that we will define a new method.

Definition 1.1.2 — Cross Entropy (CE). for two discrete probability distributions p and q, we define the cross Entropy as

$$H(p,q) = -\sum_{x\in X} p(x)log_2(q(x)) = \mathbb{E}_{x\sim p(x))}log(1/q(x))$$

This definition measures the average number of bits needed to identify an event drawn from the set if a coding scheme used for the set is optimized for an estimated probability distribution q, rather than the true distribution p. some notes that are worth mentioning:

- we can refer to p as the target probability (as in the probabilities that are associated with the true labels), and q would be our approximation of p.
- the cross entropy does not obtain the value 0 when $p = q$, like we would have wanted. For that reason (and others) we define the `Kullback Leibler (KL)` divergence $D(p||q)$, that does achieve value 0 when the probabilities are the same:

$$D(p||q) = \sum_{x\in X} p(x)log(p(x)/q(x)) = \mathbb{E}_{x\sim p(x)}log(p(x)/q(x))$$

the cross entropy defines in many cases the loss function which we wish to minimize, and this fact is strongly related the the principle of `maximum likelihood` (denoted $\mathfrak{L}$), where we try to find a set of parameters w that are correlated to the highest probabilities

1.1.1 minimizing the CE = maximizing the Likelihood

this section is denoted $(\dagger)$.

as we clean the dust from our basic ML course, a bird may whisper to you that given parameters w, a sample matrix X and true labels y, the maximum likelihood principle wishes to find the most probable w's for such X, y. in a more formal way, we have

$$\mathfrak{L}(w|X,y) = Pr[y_1,...,y_m|X,w] \overbrace{=}^{x\sim iid} \prod_i Pr[y_i|x_i,w]$$

as we wish to maximize the term, applying `log` wouldn't matter, and so does multiplying by some constant. so we denote $\mathfrak{l} = \frac{1}{|X|}log(\mathfrak{L})$, and from here:

$$\mathfrak{l}(w|X,y) = \frac{1}{m}\sum_{i=1}^{m} logPr(y_i|x_i,w) = \mathbb{E}_{x\sim p(x)}logPr(y_i|x_i,w)$$

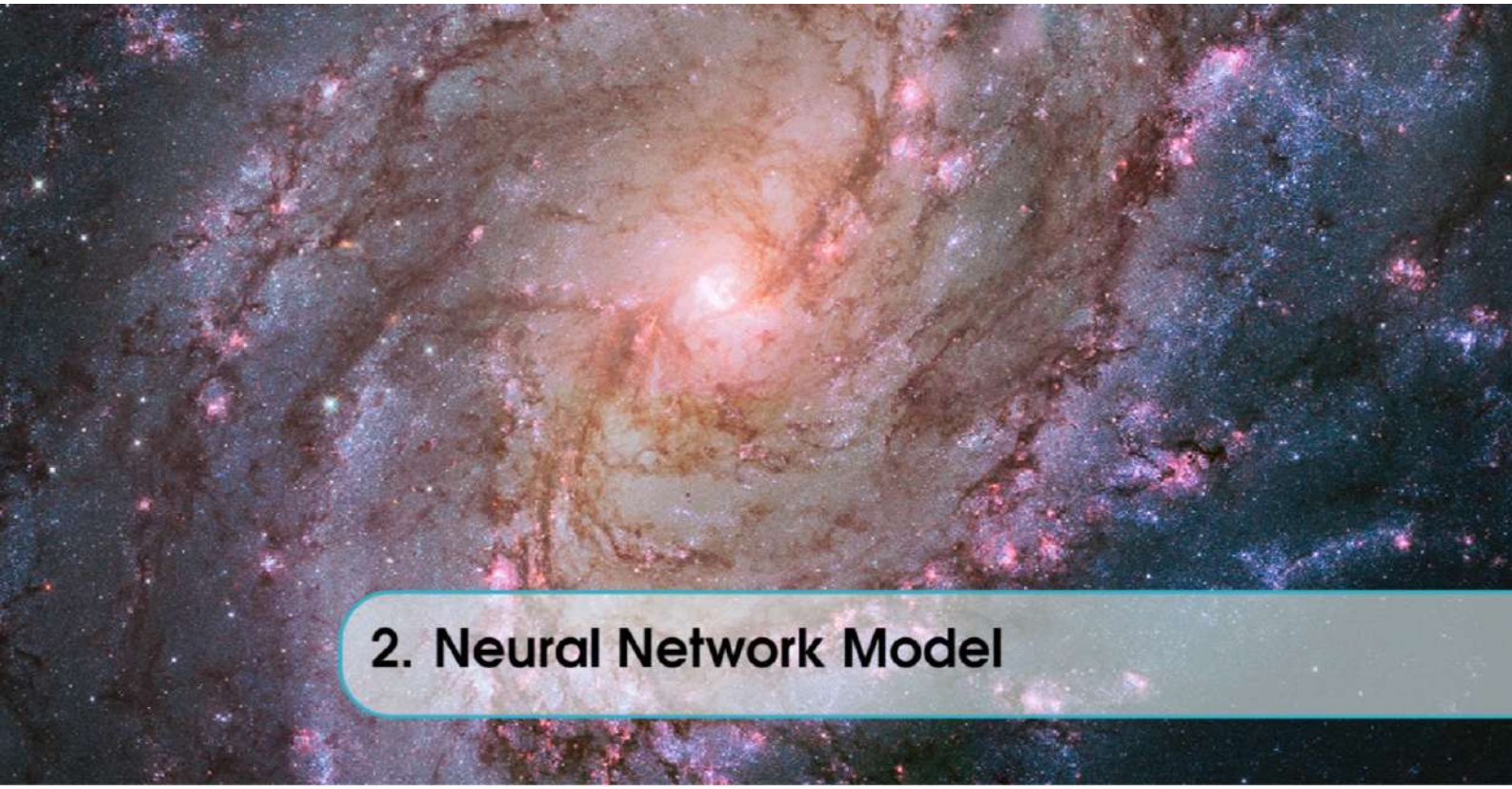
The latter simply states that we aim to find the most probable y, given X and the weights w, and can be rewritten as

$$\mathfrak{l}(w|X,y) = \mathbb{E}_{x\sim p(x)}log(q(x))$$

now our goal is to find some $\hat{w}$ such that $\hat{w} = argmax_w\mathfrak{l}(w|X,y)$, or in other words:

$$\hat{w} = argmin_w - \mathbb{E}_{x\sim p(x)}log(1/q(x)) = H(p,q)$$

which is what we wanted to show.



2. Neural Network Model

before we dive to the model of a neural network (aka a multi layer Perceptron), we will first remind ourselves what the Perceptron exactly is

2.1 Perceptron

this section is denoted (†)

the Perceptron is a supervised learning algorithm for binary classification, that finds a hyper-plane which splits a data-set S into two distinct sections/clusters. to formally describe that approach, let us first define some useful notations

Definition 2.1.1 — Hyper-plane. Is the set of all points x such that

$$\{x : \langle w,x\rangle = b, x \in \mathbb{R}^d\} = H$$

For a half space we write $\langle w,x\rangle \geq b$, which is the same as

$$\{x : sign(\langle w,x\rangle - b) \geq 0), x \in \mathbb{R}^d\}$$

▪ **Example 2.1** consider the weights $w = \{1,2,3\}$ and the bias $b = 0$. Those define the plane $\langle w,x\rangle = x_1 + 2x_2 + 3x_3 = 0$, or perhaps with relation to 3D space $x + 2y + 3z = 0$. This defines a plane that intersects with the point $(0,0,0)$. we can ask ourselves - given the points $a = (1,2,3)$ and $b = (-1,-2,-1)$, how are they positioned in relation to the plane defined by w and b. it is easy to verify that a is "above" the plane $(+1)$, and b is "below" it (-1), as figure 2.1 portrays ▪

let $\mathbb{H}_{half} = \{h_{w,b}(x) = sign(x^T w + b) : w \in \mathbb{R}^n, b \in \mathbb{R}\}$ be an hypothesis class that represents all half-planes. we would like to find some $h \in \mathbb{H}_{half}$ that classifies every sample x from our data-set S to some value $\in \{+1,-1\}$ (note that this might not always be possible). The main approach uses the fact that when ever there's a "mismatch" between the signs of a true label y_i and $h(x_i)$ (for any i), which is the same as saying that $y_i\langle x_i, w\rangle \leq 0$, the weights are altered in a manner that decreases the chance for such mismatch to occur in later iterations. This can be formalized as followed: we initialize the weights to $w_0 = (0,0,...,0)$. Then, if $\exists i$ s.t $y_i\langle x, w_t\rangle \leq 0$ we update $w_{t+1} = w_t + y_i x_i$.

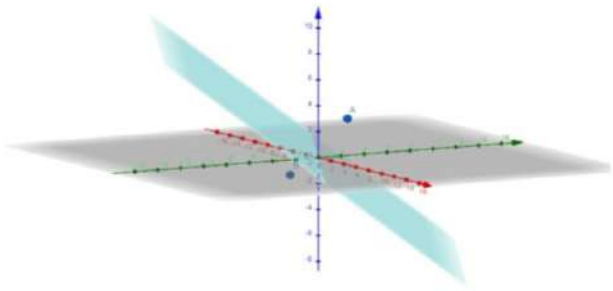


Figure 2.1: A hyper-plane in 3D

notice that $y_i\langle x_i, w_{t+1}\rangle = y_i\langle x_i, w_t + y_i x_i\rangle = y_i\langle x_i, w_t\rangle + y_i^2||x_i||$. that is, in every iteration we add an element $||x_i|| \geq 0$, which guarantees there exists some t for which the if condition doesn't hold, and in that case we return w_t.

2.2 Multi Layer Perceptron (MLP)

The term multi-layer Perceptron refers to a number of single Perceptrons that are organized to a layer. In addition, the MLP has more "freedom" to it, as it is equipped with a non-linearity function that enhances the expressibility of the model, and in general can also be used in regression problems (as opposed the the original Perceptron). In most books, the model of a single "neuron" is represented with the diagram in figure 2.2

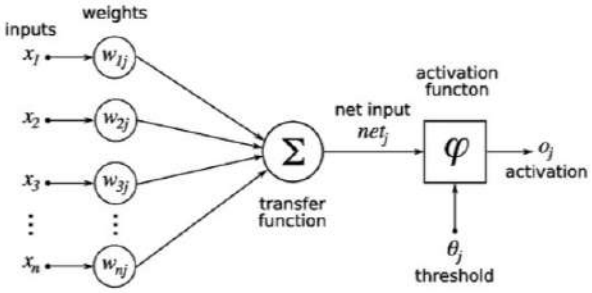


Figure 2.2: A single neuron

which indicates that given some input vector with n features $x = (x_1, x_2, ..., x_n)$, every value x_i is multiplied with some weight w_{ij}, that should in theory represent how valuable the $i'th$ neuron is to the learning process, such that the sum of all of those is the input to some activation function ϕ, to which we also add a threshold θ_j (that somewhat resembles to the activation notion of an actual neuron). This long sentence could be summarized to a single formula for the activation o_j of the $j'th$ neuron (which will resemble the notation for the Perceptron of the previous section:

$$o_j = \phi(\sum_{i=1}^{n} w_{ij}x_i + \theta_j) = \phi(\langle w_j, x\rangle + \theta_j)$$

where w_j in the single neuron model is a vector that represents the weight associated our neuron (j) and all of the other neurons in the next layer, and θ_j is the added bias (scalar).

the generalization of the above to many neurons is straight forward, as the $(\ell+1)'th$ layer $x_{\ell+1}$ is represented using the previous layer x_ℓ and a now weights matrix W^ℓ, such that $w_{ij} = [W^\ell]_{ij}$ indicates the weight associated with the (directed) edge (i, j). formally we can write $x_{\ell+1} = \phi(W^\ell x_\ell + \theta_\ell)$ (note that though ϕ here is the same for every neuron, this is not obligatory).

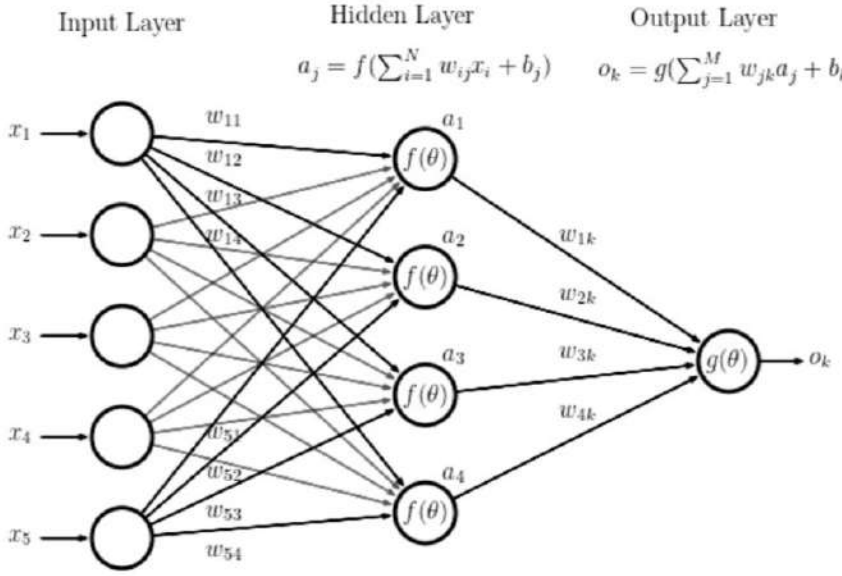


Figure 2.3: A Neural Network diagram - the input x has length n, the first weight matrix is $W \in M_{n\times m}$, the second weight matrix is $W \in M_{k\times 1}$, and we use initial activation f and final activation g.

Before moving on, notice that as much as the NN model diagrams are very cool, all they represent eventually is a series of affine transformations (with the addition of the non-linearity). That is, the output of a MLP is simply a set of affine transformations + non-linearities, and that doesn't require any diagram.

2.3 Activation Function

an activation function ϕ is a function (mostly non linear one) that is given the activation of a neuron as an input, and outputs some number. The fact that those function are mostly non-linear is an important one, as it is what distinguishes the neural network model from a basic model, only capable of predicting linear phenomenons from a much more capable one. In other words, the non-linear activation function allows the model to predict/classify various range of cases, that are not necessarily linear in nature (this will be formulated later on). Lastly, in relation to the previous sentence, notice that when using linear activation functions, the out come is a linear component, and when summed over all the layers network, we are left with an output that can be represented as $x_{out} = \mathfrak{w}x_{in} + \mathfrak{b}$, where $\mathfrak{w}$ and $\mathfrak{b}$ are some combinations of the original $\{W_i, b_i\}_{i=1}^n$. This simply indicates that when using a linear model we lose the significance of the layer formation (as well as restrict ourselves only to linear cases, as stated before).

2.3.1 Types

- `sigmoid`: also called a `logistic` curve, defined as

$$\phi(x) = \frac{1}{1+e^{-x}}$$

The sigmoid maps all values x to some normalized values $\in (0,1)$

- `relu`: which stands for rectified linear unit.

$$\phi(x) = max(0,x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases}$$

- `Softmax`: normalizes the output to probability distribution

$$\phi(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

obviously, many more options exist, but those are the basic (and most known for) ones. Furthermore, every function may be better or worse depending on the task in hand. for example, `sigmoid` will not describe magnitudes, but is valuable when the output should be bounded in $(0,1)$. Furthermore, the Sigmoid function is "flat" for large values (positive or negative), so it may cause problems when differentiating. `relu` is useful in many cases, but may cause problems if we are dealing with negative activations, and in that case the function outputs 0, which may cause problem in `SGD` process (as described for softmax, small or 0 grad may affect convergence)

2.4 Loss Function

The following step to building a NN model would be to come up with a meaningful loss function, that will somehow provide us with a number that represents how "good" our prediction is. This loss is sometimes called the `"Empirical Risk"`, and is defined as followed:

Definition 2.4.1 — Empirical Loss. given algorithm h (in our case, the algorithm is the network) that takes a sample and outputs a prediction, and m samples $x_i \in \mathbb{R}^d$ the `ER` is

$$ER(h) = \frac{1}{m}\sum_{i=1}^{m} \ell(h(x_i), y_i)$$

where $\ell : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ is some known function.

remember that our "goal" is to find $\hat{h}$ such that $\hat{h} = argmin_{h \in H} ER(h)$, and H is the hypothesis class. this is the `empirical risk minimization (ERM)` rule. The method to achieve such minima will be discussed later on.

2.4.1 Regression Loss

when a specific number is to be predicted (for example - predicting housing prices, weight percentage, etc..), a good rule of thumb would be that a small/big deviation from the "true label" would yield small/big loss respectively. One option to do that is using l_2 norm loss. Of course, other forms of functions l could work, depending on the problem. In formal notation, if we denote out hypothesis as $h = N_\theta$, where θ represents the network parameters, and use l_2 norm we have

$$ER(N_\theta) = \frac{1}{m}\sum_{i=1}^{m} ||N_\theta(x_i) - y_i||_2^2$$

2.4.2 Classification Loss

When the task in hand is to predict finite number of classes, a naive approach could use a "misclassification" loss (sometimes called "l_0" norm, though isn't a norm per say, or "$0-1$" loss).

Definition 2.4.2 — Misclassification (0,1) Loss. given algorithm h, and m samples $x_i \in \mathbb{R}^d$ the `(0,1)-loss` is

$$L_{0,1}(h) = \frac{1}{m}\sum_{i=1}^{m} \mathbb{1}(h(x_i) = y_i)$$

where $\mathbb{1}(x) = \begin{cases} 1, x \text{ is true} \\ 0, \text{otherwise} \end{cases}$ is the indicator function

the misclassification loss may be a bad idea in cases where it is important to distinguish `Type1` or `Type2` errors. Another option is to use `Cross Entropy`, which we have already seen before. For this to have any meaning though, we must first "cast" the network output to some bounded values, that would represent some distribution p. Two common metrics are (for a network output vector $\hat{y}_i$):

- `logistic`: $\hat{y}_i \mapsto 1/(1+e^{-\hat{y}_i})$ suited for binary classification
- `softmax`: $\hat{y}_i \mapsto e^{\hat{y}_i}/(\sum_i e^{-\hat{y}_i})$. mostly suited for multi class classification

after this was done, the cross entropy would give an estimation to the network's "capabilities", by comparing the true distribution of the data with the network's output distribution:

Definition 2.4.3 — Cross Entropy in Multi Class Classification. $H(p,q) = -\sum_{c=1}^{C}\sum_{i=1}^{m} \hat{y}_i^c log(h^c(x_i))$ where C is the number of classes.

specifically for binary classification, we have

$$H(p,q) = -\sum_{i=1}^{m}[\hat{y}_i log(h(x_i)) + (1-\hat{y}_i) log((1-h(x_i)))]$$

where $h(x_i)$ are the logistic values of the output labels.

2.5 Network Training

We have structured a neural network model N_θ, and chose a meaningful loss function L. Now its time to train the model, that is, find all the parameters θ that would minimize L. In other words, we wish to calculate $min_\theta\ L$, and a key component to this is the fact that we can find the (local) minimum iteratively, by taking "steps" in the direction of the maximum descent, aka the minus gradient direction $-\nabla L$

2.5.1 Gradient Descent Optimization

We would like to "move" in a direction $\Delta x \in \mathbb{R}^n$ such that $f(x_t + \Delta x) < f(x_t)$ (where t denotes the current iteration and f is the loss). usually we pick $\Delta x = -\eta \nabla f|_{(x_t)}$, where η is a step size (predefined hyper parameter). This choice is not arbitrary - if f is differentiable around x_t (if its not we use the sub-differential, though we will not linger on this) we can look at its Taylor approximation

$$f(x_t + \Delta x) = f(x_t) + \Delta x^T \nabla f|_{(x_t)} + O(||\Delta x||^2)$$

as we wish to maximize the difference in every step, our problem is

$$\max_{||\Delta x||=\varepsilon} f(x_t + \Delta x) - f(x_t) \approx \max_{||\Delta x||=\varepsilon} \Delta x^T \nabla f|_{(x_t)}$$

(for some small ε). using Lagrange multipliers for $h = \Delta x^T \nabla f|_{(x_t)}$ and $g = ||\Delta x|| - \varepsilon = (\Delta x)^T \Delta x - \varepsilon$:

$$\begin{aligned} \nabla_{(\Delta x)^T} h = \lambda \nabla_{(\Delta x)^T} g \rightarrow \nabla_{(\Delta x)^T}(\Delta x^T \nabla f|_{(x_t)}) &= \lambda \nabla_{(\Delta x)^T}((\Delta x)^T \Delta x - \varepsilon) \\ \rightarrow \nabla f|_{(x_t)} &= \lambda \Delta x \end{aligned}$$

What this really tells us, is that if we wish to maximize the objective, we should pick Δx that is in the direction of the gradient ∇f (which in general correlates with that we know about the gradient - direction of steepest ascend). the above can be summarized to an iteration loop which is sometimes referred to as `gradient descend process`:

Definition 2.5.1 — Gradient Descend Process. given a neural network model N_θ with parameters θ (weights,biases,etc..), step size η, true labels y, samples X and a loss function L, a gradient descend process over the parameters θ is

$$\theta_{i+1} = \theta_i - \eta \nabla_\theta L(N_{\theta_i}, X, y)$$

some important things to keep in mind

- the convergence isn't guaranteed to be towards a global minimum.
- the step size should be small enough to justify linear approximation, though not too small so we wont converge

In the final chapter of the course, we will provide more optimization approaches

2.5.2 Mini Batching and SGD

as our data becomes larger, it may be less feasible to calculate the gradient of the loss function with respect to every sample in every iteration. For that reason we may choose a subset of samples randomly, and find the gradients with respect to those samples. Notice that this does not come without risks, as our data may not be consistent, which may cause problems when trying to converge. Nevertheless, if we are sufficiently confident that every random sample can represent out data properly, the calculated gradient is this approach gives an approximation to the actual gradient, which in theory should bring us closer to a (local) minima.

formally speaking, if $B \subset X$ is a mini batch of samples that is randomly generated from our data X, the gradient of the loss function over all X is approximated by the gradient over all B:

$$L = \frac{1}{|X|}\sum_{i\in X} l(N_\theta(x_i), y_i) \Rightarrow \nabla L \approx \frac{1}{|B|}\sum_{i\in B} \nabla l(N_\theta(x_i), y_i)$$

Notice we've used the linearity of the operator ∇. From the section above, we can now formulate an algorithm

Algorithm 1 Stochastic Gradient Descent

Require: learning rate η, number of epochs T, batch size m, NN model N_θ

$\theta \leftarrow (0,0,...,0)$

for $t = 1,2,...,T$ **do**

sample a mini-batch: $B = \{\mathbf{x_i}, y_i\}_{i=1}^m$

$g \leftarrow \frac{1}{m}\sum_{i\in B} \nabla l(N_\theta(\mathbf{x_i}), y_i)$

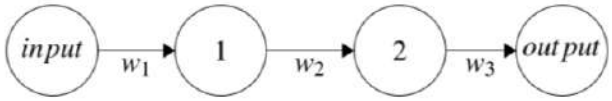
$\theta \leftarrow \theta - \eta g$

end for

2.5.3 Back-propagation

this section is (mostly) denoted (†)

It turns out that the calculation of the gradient is in itself a quite daunting task. In this section we will see how to calculate it efficiently using what is known as the back propagation algorithm. lets start with a basic neural network - input node, two hidden nodes and one output node



in this network the loss is $L = L(w_1, b_1, w_2, b_2, w_3, b_3)$, and we wish to find what are the weights and biases that provide the minimal L. In the section below, we will seclude ourselves to l_2 loss (though this is generally true for every loss), the networks error over the $0'th$ sample (say, the first image out of 20,000) is

$$L_0 = (a^{(L)} - y)^2$$

where $a^{(L)}$ is the activation of the $L'th$ layer (the last one), and y is our true labels vector. more specifically, the activation is (as stated at the beginning of the chapter)

$$a^{(L)} = \sigma(w^{(L)} a^{(L-1)} + b^{(L)}) := \sigma(z^{(L)})$$

For starter, our goal is to understand how the loss L changes with respect to $w^{(L)}$ and $b^{(L)}$. Specifically for $w^{(L)}$ that change can be written as its partial derivative

$$\frac{\partial L_0}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L_0}{\partial a^{(L)}}$$

we know the exact forms of L_0, $a^{(L)}$ and $z^{(L)}$:

$$\begin{aligned} \frac{\partial L_0}{\partial a^{(L)}} &= 2(a^{(L)} - y) \\ \frac{\partial a^{(L)}}{\partial z^{(L)}} &= \sigma'(z^{(L)}) \\ \frac{\partial z^{(L)}}{\partial w^{(L)}} &= a^{(L-1)} \end{aligned}$$

which means

$$\frac{\partial L_0}{\partial w^{(L)}} = a^{(L-1)} \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

note that this is only for the $0'th$ sample, and for the entire data-set (or batch) we take the average:

$$\frac{\partial L}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial L_k}{\partial w^{(L)}}$$

now we repeat the process for the biases b:

$$\frac{\partial L_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L_0}{\partial a^{(L)}} = 1 \sigma'(z^{(L)}) 2(a^{(L)} - y)$$

and when averaging:

$$\frac{\partial L}{\partial b^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial L_k}{\partial b^{(L)}}$$

Great! but those were just two elements in the vector ∇L we wish to calculate, as we remember that

$$\nabla L = \left(\frac{\partial L}{\partial w^{(0)}} \quad \frac{\partial L}{\partial b^{(0)}} \quad \cdots \frac{\partial L}{\partial w^{(L)}} \quad \frac{\partial L}{\partial b^{(L)}} \right)^T$$

so now we have to look at every layer $l \leq L$. we start with $l = L - 1$:

$$\frac{\partial L_0}{\partial w^{(L-1)}} = \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial L_0}{\partial a^{(L-1)}}$$

the first two elements are calculated by hand:

$$\frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} = a^{(L-2)}$$
$$\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} = \sigma'(z^{(L-1)})$$

for the last element though, notice that When using chain rule again we get

$$\frac{\partial L_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial L_0}{\partial a^{(L)}}$$

aside from $\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(L)}$, we already have the two right most elements, as they were calculated in the previous part (where $l = L$), so we could write

$$\frac{\partial L_0}{\partial w^{(L-1)}} = a^{(L-2)}\sigma'(z^{(L-1)})w^{(L)}\sigma'(z^{(L)})2(a^{(L)} - y)$$

in other words, to find $\frac{\partial L_0}{\partial w^{(L-1)}}$ we needed $\frac{\partial L_0}{\partial a^{(L-1)}}$, but the latter is already known from previous iterations. This would give us a recursive definition for $\frac{\partial L_0}{\partial w^{(l)}}$ for any $l \leq L$, but before that - lets generalize for a multi layered network:

we will add a sub-script that indicates which neuron are we referring to in the specific layer. for example: $a_k^{(l)}$ is the activation of the $k'th$ neuron of the $l'th$ layer. Now, L_0 is simply the sum over all neurons. suppose the $L'th$ layer has n_L neurons, so

$$L_0 = \sum_{j=0}^{n_L} (a_j^{(L)} - y_j)^2$$

lets denote the weight between $node_k^{L-1}$ and $node_j^L$ as w_{jk}^L. the activation would now be

$$a_j^{(L)} = \sigma(w_{j,0}^{(L)} a_0^{(L-1)} + ... + w_{j,n_L-1} a_{n_L-1}^{(L-1)} + b_J^{(L)}) = \sigma(\mathbf{w}_j^{(L)} \cdot \mathbf{a}^{(L-1)}) := \sigma(z_j^{(L)})$$

where we defined

$$z_j^{(L)} = \sum_{k=0}^{n_{L-1}-1} w_{jk}^{(L)} a_k^{(L-1)} + b_j^{(L)}$$

we now have everything that is needed to calculate the derivatives

$$\frac{\partial L_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial a_j^{(L)}} = a_k^{(L-1)}\sigma'(z_j^{(L)})2(a_j^{(L)} - y_j)$$

where as usual, the entire loss is an average

$$\frac{\partial L}{\partial w_{jk}^{(L)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial L_i}{\partial w_{jk}^{(L)}}$$

repeating the process for the bias:

$$\frac{\partial L_0}{\partial b_j^{(L)}} = \frac{\partial z_j^{(L)}}{\partial b_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial a_j^{(L)}} = 1\sigma'(z_j^{(L)})2(a_j^{(L)} - y_j)$$

and the average

$$\frac{\partial L}{\partial b_j^{(L)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial L_i}{\partial b_j^{(L)}}$$

following the thread of the single layer network, we would also wish to find the derivatives according to the activations. This is where the multi layer network differs from the single layer one. The main difference is that while for the single layer model there was only one activation, in the multi layer model every node has its own activation, and they all should be summed together (as the loss L is a function of them all). this can be formally written as

$$\frac{\partial L_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial L_0}{\partial a_j^{(L)}}$$

This is pretty much all we need - so we can write it all together:

$$\frac{\partial L_0}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \sigma'(z_j^{(l)}) \frac{\partial L_0}{\partial a_j^{(l)}}$$
$$\frac{\partial L_0}{\partial b_j^{(l)}} = \sigma'(z_j^{(l)}) \frac{\partial L_0}{\partial a_j^{(l)}}$$

where

$$\frac{\partial L_0}{\partial a_j^{(l)}} = \begin{cases} \sum_{j=0}^{n_{l+1}-1} w_{jk}^{l+1} \sigma'(z_j^{l+1}) \frac{\partial L_0}{\partial a_j^{(l+1)}}, & l < L \\ 2(a_j^{(l)} - y_j), & l = L \end{cases}$$

finally, the elements of the vector ∇L are

$$\frac{\partial L}{\partial w_{jk}^{(l)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial L_i}{\partial w_{jk}^{(l)}}$$
$$\frac{\partial L}{\partial b_j^{(l)}} = \frac{1}{n} \sum_{i=0}^{n-1} \frac{\partial L_i}{\partial b_j^{(l)}}$$

2.6 before moving on...

we would like to present some important topics that are relevant, in no particular order

2.6.1 Data pre-processing

let us briefly describe some a data processing methods.

- standard normalizing: given data points $\{x_i\}$, we may normalize it to have zero mean and with variance of size 1. why would this be necessary? recall that generic MLP can be formulated as $f(\sum_i w_i x_i + b)$, therefore when back-propagating, $\frac{dL}{dw} = \frac{df}{dw} \cdot \frac{dL}{df} = x_i \frac{dL}{df}$. this means that, for example, if all x_i are positive, the update rule would be truncated to only some specific direction (as the signs wouldn't change). For this to be fixed, we can subtract the mean value of x. Also, dividing by the standard deviation stretches out data, which can help with getting more suitable gradients.

$$X \mapsto (X - mean(X))/std(X)$$

- BatchNorm: this normalization is done not to the data itself, but as a layer in the architecture. In general terms, we may not always wish to normalize with some constant mean and deviation, but to some learned normalization. Intuitively, theres no reason to believe that after every step the normalization we computed previously is still relevant the the current situation, therefore a learned distribution is more suitable. a BatchNorm layer can be defined as $BN(x) = \gamma\hat{x} + \beta$, where γ, β are some learned parameters, and $\hat{x}$ is some normalization to x. more formally, given some mini-batch B of size m, we write the empirical mean and variance as

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i \text{ and } \sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_B)^2$$

for a layer of the network with dimension d: $x = (x^{(1)}, ..., x^{(d)})$, we define the normalization of x to be

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{(\sigma_B^{(k)})^2}}, \text{ where } i \in [1,m] \text{ and } k \in [1,d]$$

finally, a transformation step would be

$$BN(x^{(k)}) = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}$$

where $\gamma^{(k)}$ and $\beta^{(k)}$ are learned in the learning process

of course, many more options are relevant and extremely important when pre-processing our data, yet most of those are task-specific.

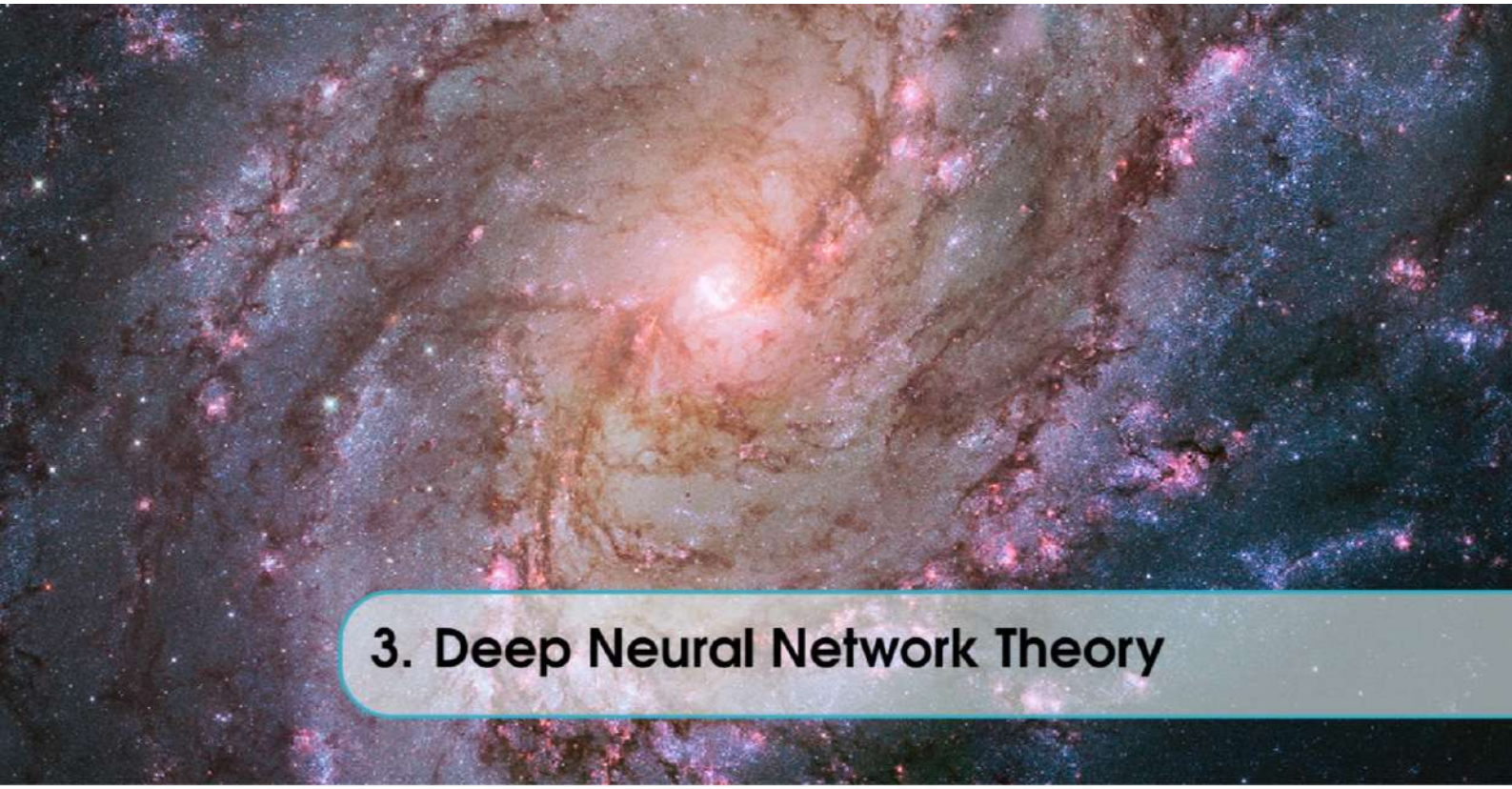
2.6.2 Bias-Variance Trade-off

Though we assume the reader is familiar with the term, it is important to state out some methods of dealing with the trade-off, under the umbrella of deep learning

- Dropout: in every iteration we zero out/normalize some randomly chosen weights. This forces the network to have a redundant representation, and prevents co-adaptation of features. In terms of bias-variance, adding dropout increases the bias (As the model is being simplified) but decreases the variance (as we decrease dependencies)
- Regularization: either ℓ_1 or ℓ_2 are common, where in most cases the regularization term are the weights themselves. For example, we can add a regularization term to the basic ℓ_2 loss of an MLP that is (for $k \in \{1,2\}$)

$$L = ||Xw - y||_2^2 - \lambda \sum_i |W_i|^k$$

- Data augmentation: this also relates to the pre-processing, but the main idea is to augment our data in some way that still preserves the main features, which provides a model that is more robust. (some options are (a) crop (b) resize (c) flip (d) color distort (e) Gaussian noise, etc...)



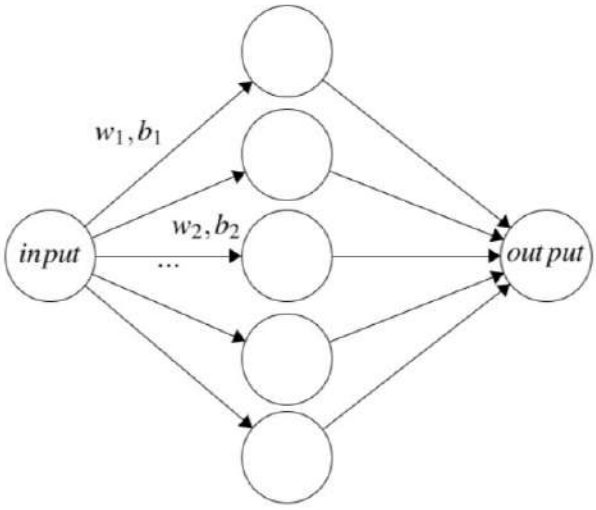
3. Deep Neural Network Theory

3.1 Universal Approximation Theorem

let us deal with the question of how "expressive" a neural network can get. More specifically, a network with only one hidden layer.

Theorem 3.1.1 — Cybenko (89). for a continuous monotone function σ with $\sigma(-\infty) = 0$ and $\sigma(\infty) = 1$, the family of function $f(x) = \sum_i \alpha_i \sigma(w_i x + b_i)$ is dense in $C([0,1])$ w.r.t the supremum norm $d(f,g) = sup|f(x) - g(x)|$

In a less formal form, Cybenko's theorem states that we can approximate any continuous function $g : \mathbb{R} \to [0,1]$ (`sigmoid` for example) with arbitrarily small precision using a function f that represents a single hidden layer neural network. the network itself can be represented as below



notice that both input and output are scalars, and it that sense the network is "as simple as it gets". A proof is not provided here, but intuitively speaking - the model can represent any stretch w_i, shift

b_i and weight α_i to the input, and this is what gives the model its power or expressivity. with respect to Cybenko's theorem, few notes are worth taking:https://www.overleaf.com/project/618c00b45e87e2124e043747

- the work of Cybenko was extended by Hornik 2 years later, and in his theorem the statement was provided for every bounded σ, which is a relaxation to the claim that σ must be monotone.
- the above does not contradict the fact that one can also use $\sigma \equiv relu$ function, even though `relu` is not bounded. this is because the following composition:

  $$f(x) = relu(Wx+b_1) - relu(Wx+b_2)$$

  is bounded, and for a specific choice of b_1, b_2, the function f is sigmoidal ("S" shaped, and bonded)

3.2 Shallow compared to Deep NN

A legitimate question that may arise from Cybenko's Theorem is - if a shallow layer can represent every function with small error, why should we go the extra mile by defining a deep network (with 2 or more hidden layers)? To answer this question, let us begin with a short example

■ **Example 3.1** We define the `t-saw-tooth` function as a piece-wise affine function with t pieces. Also, we define the `hat` function as $hat(x) = relu(2relu(x) - 4relu(x-0.5))$. note that `hat` is a `4-saw-tooth` function by definition. We can use `hat` to concatenate `saw-tooth` functions - note that $t(x) = hat(x) + hat(x-1)$ comprises of two `hats`, and is a `6-saw-tooth` function, and the composition $t(t(x))$ is a `10-saw-tooth`, $t(t(t(x)))$ is a `18-saw-tooth`, and in general, a composition of T functions (T $t(x)$'s) is a $(2+2^{T+1})$-`saw-tooth`, namely, the composition is comprised of 2^T hats.How can we represent this composition using a shallow network? remember that every neuron in the (single) hidden layer "represents" a single `relu`, therefore two neurons can represent a single `hat`. As there are exponentially many hats, a shallow network will have to use exponentially many neurons (2^{T+1} to be exact). On the other hand, a deep neural network would only need $\Theta(T)$ (if, for example, we use T layers and 2 nodes per layer). This is the case because the deep network $i'th$ layer receives as input the values of the $(i-1)'th$ layer, and for that matter is represents the composition of $t(x)$ simply as a product of its architectural design. ■

the example above demonstrates an exponential gap (in terms of size) between the shallow and deep NN architectures, and can be summarized to the following lemma:

Lemma 3.2.1 if f is `a-saw-tooth` and g is `b-saw-tooth`, $f+g$ is at most `(a+b)-saw-tooth` and $f(g)$ is at most `a×b-saw-tooth`

"proof". as for $f+g$, the "worst-case-scenario" is when the edges of f and g doesn't intersect, and in that case $f+g$ will inherit all the discontinuities that f and g had. For $f(g)$, the image of each piece of g can contain the entire range in which f has all its pieces, which means that for every one of the b affine section of g we provide an input (to f) that generates a affine sections, to a total sum of $a \cdot b$. ■

the above statements could also be asked in reverse order - we have claimed that a deep layer with T layers and $O(T)$ neurons can be represented using a shallow layer with 2^T neurons, and now we claim that a shallow network with n neurons could be represented with a deep network, using $\Theta(n)$ neurons. As an example, we go back to our shallow network model $f(x) = \sum_i \alpha_i \sigma(w_i x + b_i)$ under the assumption $\alpha_i > 0$ and construct a deep network with n layers and $O(1)$ neurons per layer as followed: the network is built from three main "branches" h_1, h_2 and h_3. h_1 will be used to "carry"

the input x as it was initially provided, h_2 is for the actual calculation of the affine transformation on the input and h_3 sums the values of h_2.

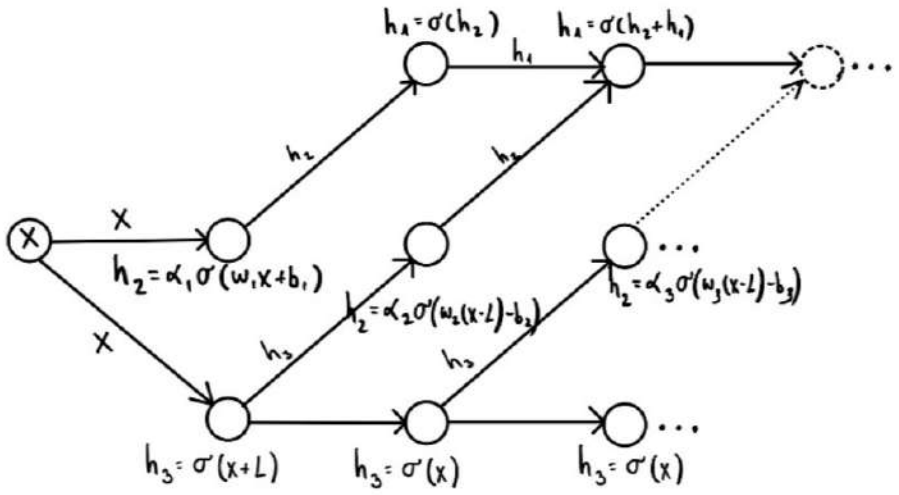


Figure 3.1: Shallow NN to Deep NN conversion.

note that the first value of h_3 is $\sigma(x+L)$, where L is a shift we make to ensure positive values. This is needed so we wont lose any negative value edges under the `relu` function. In the following parts of h_2 we shift back to the original location $x-L$. Also note that as $\alpha_i > 0$, the values of h_3 are always positive, that is $\sigma(h_1+h_2) = h_1+h_2$. the network provided in the figure has $\Theta(n)$ layers with 3 neurons in each, which is what we intended to show ($O(n)$ neurons). All in all, we can summarize the above with the following diagrams

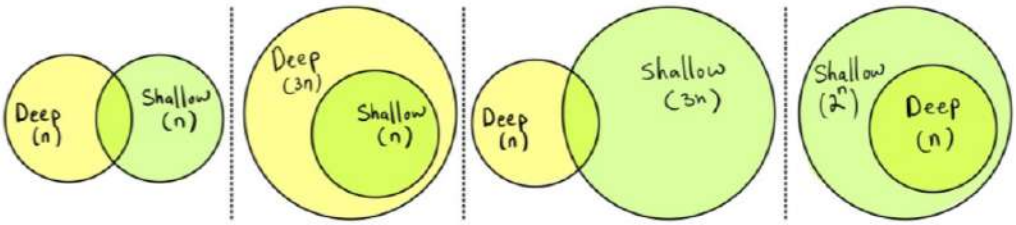
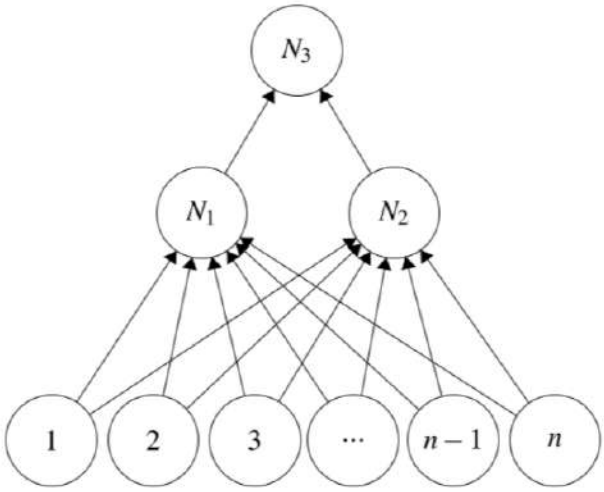


Figure 3.2: Venn Diagram of NN models with $(\cdot)$ neurons. From left to right - (1) n neurons deep network isn't enough to represent every n neurons shallow network, but in some cases it is sufficient. (2) On the other-hand, $3n$ neurons is enough, and the deep model is more expressive, hence contains the shallow models. (3) the reverse statement cant be made, (4) a shallow model with 2^n neurons can represent every n neurons deep model, and then some.

3.3 Training a NN is NP-Complete

In this section we will prove that the training process of a NN is considered a NP-complete problem. The model we will use consists of Three neurons and is fully connected, where the input is of length n.



Moreover, the neurons N_1, N_2 and N_3 performs `Heaviside` activation on its input, as followed - given input $x = (x_1, ..., x_n)$, weights $(a_1^{(1)}, a_2^{(1)}, ..., a_n^{(1)}, a_1^{(2)}, a_2^{(2)}, ..., a_n^{(2)}, a_1^{(3)}, a_2^{(3)})$ and biases (b_1, b_2, b_3)

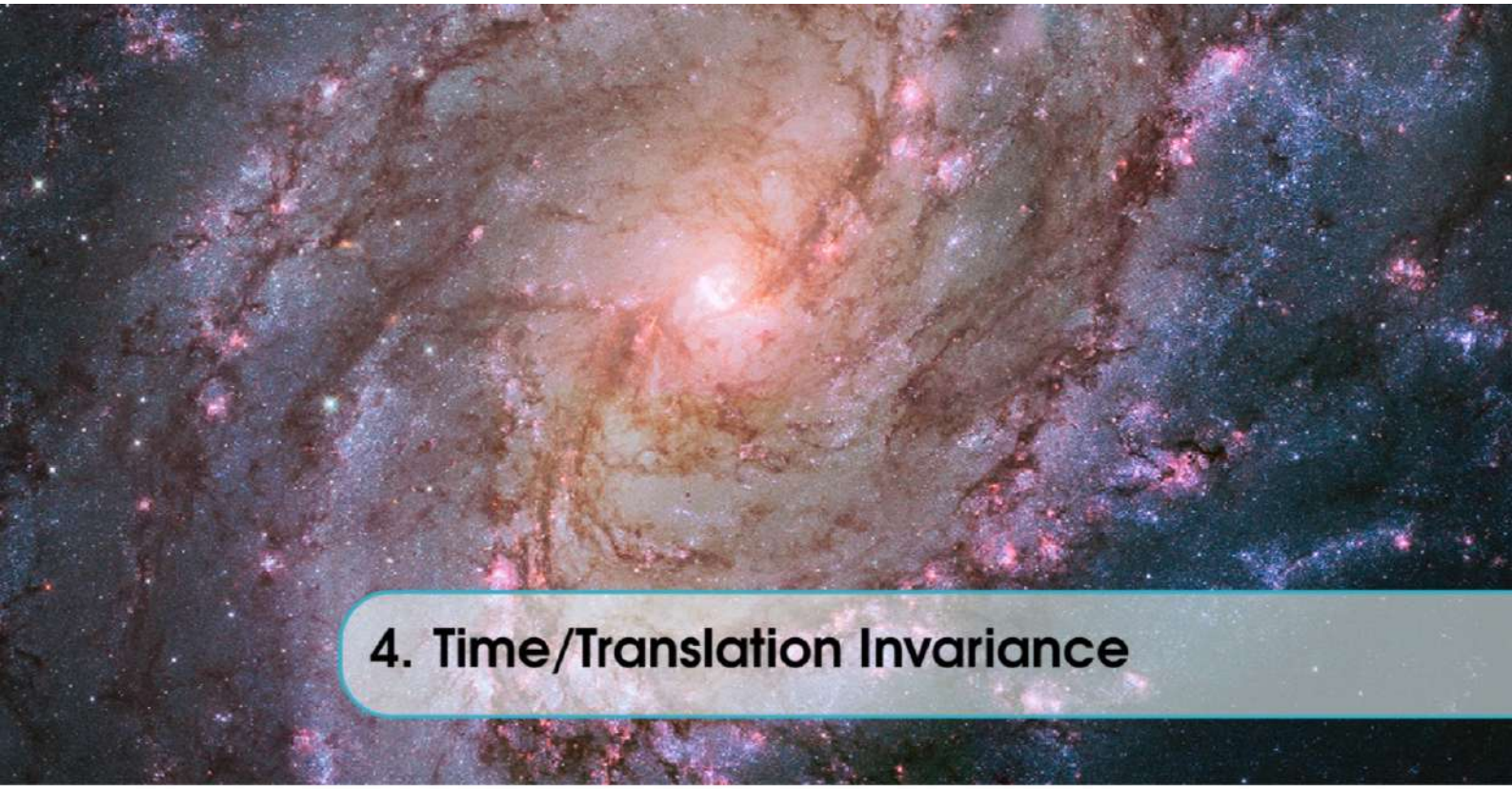
$$f_{i=1,2}(x) = \begin{cases} 1, & \sum_{j=1}^{n} a_j^{(i)} x_i + b_i > 0 \\ -1, & \text{else} \end{cases}$$

$$f_{i=3}(x) = \begin{cases} 1, & a_1^{(3)} N_1 + a_2^{(3)} N_2 + b_3 > 0 \\ -1, & \text{else} \end{cases}$$

where N_1, N_2 in $f_{i=3}$ indicates the outputs of neurons N_1, N_2, which are provided as input to N_3.

Corollary 3.3.1 — The Trainability Problem. given a set of $O(n)$ samples $\in \{0,1\}^n$, finding the Networks' parameters and threshold functions for which it produces outputs consistent with the training set is NP-complete

Proof. soon to be ■



4. Time/Translation Invariance

4.1 Notion of Time-Space

In this section we will discuss systems that are Invariant - either in time or in space translations. Such prior knowledge on our data will allow us to define more suited NN models that could potentially provide precise solutions using less parameters, and with great efficiency.

4.1.1 Translation/Time Invariant Systems

Systems that does not depend on time or spatial coordinated are called Translation/Time Invariant. This formally means that a shift in the input leads to a shift in the output, which for some operator L translates to:

$$L[f(x+t)](y) = L[f(x)](y+t)$$

where f is our input (image/audio for example). for example - our face detector algorithm should operate the same for all faces that are in different offsets in the image.

from here on we will refer to $f(x)$ as discrete samples of our signal, that is

$$f(x) = \sum_s \delta(x-s) f(s)$$

where $f(s)$ are scalars and δ is `Dirac's delta function`

R regarding δ-function, recall that

- $\int_{-\infty}^{\infty} \delta(x)dx = 1$
- $\int_{-\infty}^{\infty} \delta(x-x_0)f(x)dx = f(x_0)$
- a non formal definition would be $\delta(x-x_0)$" = " $\begin{cases} \infty, x = x_0 \\ 0, \text{otherwise} \end{cases}$

4.1.2 Linear Translation/Time Invariant Systems (LTI)

Let us also consider the linear and Time/Translation Invariant (TI) case, in which L performing on the sum of samples is as if we sum the operation of L on every sample by itself. This setup provides a convenient conclusion:

$$\begin{aligned} L[f(x)](y) &= L[\textstyle\sum_s \delta(x-s)f(s)] \overset{linearity}{=} \\ &\textstyle\sum_s f(s)L[\delta(x-s)](y) \overset{TI}{=} \\ &\textstyle\sum_s f(s)L[\delta(x)](y-s) \overset{g(x):=L[\delta(x)]}{=} \\ &\textstyle\sum_s f(s)g(y-s) := f*g \end{aligned}$$

In other words, if L represents a `LTI` system, it can be written as convolution of the input f with some filter/kernel $g = L[\delta]$ (that can be chosen to our likelihood).

R regarding convolution, recall that

- it is commutative: $f*g = g*f$
- it is associative: $f*(g*h) = (f*g)*h$
- it is distributive: $f*(g+h) = f*g+f*h$
- convolution theorem: $f*g = FT^{-1}[FT(f)\cdot FT(g)]$, where $FT(\cdot)$ signifies the `Fourier Transform` of $(\cdot)$

It is also worthy mentioning that every linear transformation can be expressed by a matrix. In our case

$$(f(x)*g(x))(y) = \textstyle\sum_s f(s)g(y-s) = \langle f(x), g(y-x)\rangle = G\hat{f}$$

Where we restricted ourselves to finite space (sum is from $s=0$ to $n-1$) and cyclic convolution, where n signifies the number of elements in the signal, and there are infinitely many cycles of repeating n such elements. This is for

$$G = \begin{pmatrix} g(0) & g(1) & g(2) & \dots & g(n-1) \\ g(n-1) & g(0) & g(1) & \dots & g(n-2) \\ g(n-2) & g(n-1) & \dots & \dots & g(n-3) \\ \vdots & \vdots & \dots & \dots & \vdots \\ g(1) & g(2) & \dots & g(n-1) & g(0) \end{pmatrix} \text{ and } \hat{f} = \begin{pmatrix} f(1) \\ f(2) \\ \vdots \\ f(n) \end{pmatrix}$$

notice that every row in G is a cyclic permutation of the previous row

▪ **Example 4.1 — using G and $\hat{f}$.** denote $h = f*g$ and let f,g be defined in the coordinate range $[0,5]$ as followed: $f = (0,0,0,1,0,0)$ and $g = (0,0,0,1,-1,0)$, for specific $y = 6$

$$h(6) = \textstyle\sum_{s=0}^{5} f(s)g(6-s) = f(0)g(6) + \dots + f(3)g(3) + \dots + f(5)g(1) = f(3)g(3) = -1$$

for every value of h, we can use G and $\hat{f}$:

$$h = \begin{pmatrix} 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 \\ -1 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ -1 \end{pmatrix}$$

■

■ **Example 4.2 — derivative.** consider the filter $g = (-1, 1)$, and some general $f(x)$ notice that $f * g = f(x+1) - f(x)$.

how does this relates to derivatives? from definition

$$\frac{\partial f(x)}{\partial x} = \lim_{\varepsilon \to \infty} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}$$

this can be approximated when f is discrete (for example, if f is some image, we can consider a single pixel as our minimum length, which has size 1). That is

$$\frac{\partial f(x)}{\partial x} \approx f(x+1) - f(x)$$

■

before we move on, let us define 2D convolution, as it is relevant for many convolutional NN architectures

Definition 4.1.1 — discrete 2D convolution. given 2D functions $f(x,y)$ and $g(x,y)$ and two integers $a, b \in \mathbb{N}$, we define the 2D discrete convolution as

$$f * g = \sum_{k=-a}^{a} \sum_{l=-b}^{b} g(k,l) f(x-k, y-l)$$

intuitively, we can think of g as a kernel that we "run-through" our function f and for every k, l, the new value we assign to $f * g$ is the sum of all multiplications of points where g lays on top of f.

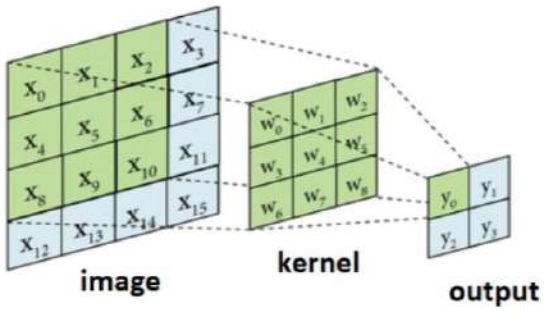


Figure 4.1: 2D convolution diagram

4.2 Convolutional NN

4.2.1 the outline

A convolution layer is a layer in our architecture that operates convolution instead of just summing up weights (like we saw in the MLP architecture), that is - the values of the $i'th$ layer L_i is given by convolving the previous layer L_{i-1} with some chosen kernel function g, i.e $L_i = L_{i-1} * g$ (up to bias addition and non linearity) . our weights would be the values of the kernel, and these would be (along with the biases) the parameters which we aim to learn in the iterative learning process. Generally speaking, the transition between layers in such architecture would be

- apply N dimensional convolution - same as applying dot product to the input with the kernel
- add bias

- apply some nonlinear activation
- Optional: apply pooling - sample every $j'th$ pixel, for some j (see below)

note that compared to multi layer fully connected NN, applying convolution greatly reduces the parameters of the network, as the matrix G possesses "only" n different values, compared to fully connected layer that applies a matrix with n^2 parameters.

In Conv nets, it is not clear a-priori what the neurons really are. Most Literature defines the neurons as the rows of G, which are neurons with shared weights (as $g_1, ..., g_n$ are the same for every neuron, with some permutation).

It is common to think of the convolution layers as layers that extract features that are increasingly concrete. For example, the first layers would be responsible for only edge detection, and deeper layers may focus on features that are less vague, such as specific features of a face if we try to detect faces.

4.2.2 the architecture

for a more rigorous description, lets consider the case of 2D convolutions applied on images. An image is a tensor that has dimensions $w \times h \times c$, where in most cases c represents `Red,Green,Blue`, therefore it satisfies $c = 3$. A convolution layer receives the image as input, and generates an output that is called an `activation map`, which is a tensor with dimensions $w' \times h' \times c'$, which is the product of convolving with a kernel with dimensions $w_g \times h_g \times c \times c'$. What this actually mean is that we map every cube with dimensions $h_g \times w_g \times c$ to a single pixel in our output, and we perform this c' times, for c' kernels. The following figure illustrates a concrete example

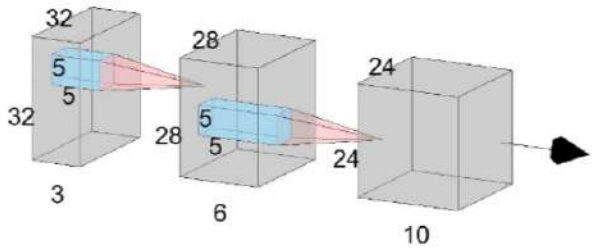


Figure 4.2: left to right - $layer_1$ ($32 \times 32 \times 3$) convolved with $kernel_1$ ($5 \times 5 \times 3 \times 6$) outputs $layer_2$ ($28 \times 28 \times 6$) convolved with $kernel_2$ ($5 \times 5 \times 6 \times 10$) outputs $layer_3$ ($24 \times 24 \times 10$). notice there is no padding, so the width and height decreases, and between each layer we may perform some non linearity σ

note that when no padding is being made and the $stride = s$, the dimensions of the $i'th$ tensor are fully determined by the dimensions of the kernel and the previous tensor $i-1$:

$$w_i = (w_{i-1} - w_{g,i-1})/s + 1, h_i = (h_{i-1} - h_{g,i-1})/s + 1 \text{ and } c_i = c_{g,i-1}$$

4.2.3 pooling layer and strides

A pooling layer reduces the size of the image, which may be relevant for example if our input is too large to be fed through our network efficiently. Say we wish to scale down our input by a factor of 2 - few variations exist

- max pooling - take a 2×2 section in the input (for example grid of 2×2 pixels) and choose only the pixel that has maximum grey-scale. One downside to this is that we lose the information of which pixel was indeed the one with the max grey-scale.
- average pooling - take the average of your input points in a given window

stride, on the other-hand is the process of sub-sampling our input - instead of shifting our kernel one step at a time, we shift it two or more steps.

4.3 Other Invariances

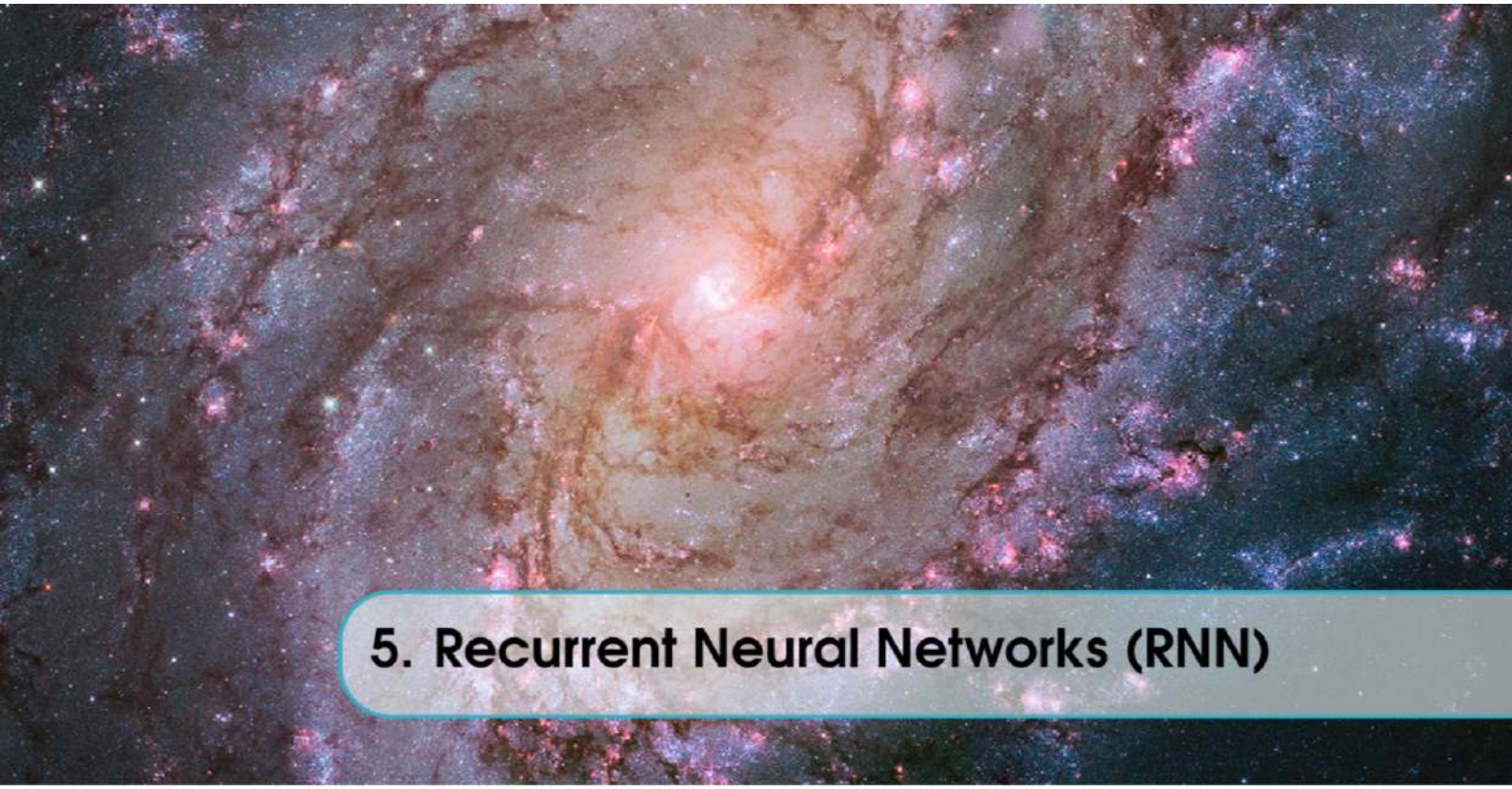
Setting aside time and space, it is easy to come up with more cases for which our model should be invariant to. For example, two images of a cat - one that is dark and one that is bright, should always be classified the same way. This means that I want my system to be invariant to changes in lighting. A good network might classify such cases properly without apriori intervention, but we can make it's life easier if we apply some smart preprocessing to our data, say normalize each pixel beforehand. As this may indeed work, it is also common to normalize one of the layers and not the images (or add a new layer that simply normalized its input). This process/layer is known as `Local Response Normalization Layer`, and is formally defined as the following mapping:

$$R_i(x) \rightarrow \frac{R_i(x)}{(c+\alpha \sum_j R_j(x))^\beta}$$

where c, α, β are some predefined parameters, and $R_i(x)$ represents some response of sample i

another example is when we wish that our results would satisfy some relation - say we output (x,y), but the only relevance is that (x,y) would be positioned on a circle defined by $r = \sqrt{x^2+y^2}$. What we can do is map the predicted position (x,y) to $\frac{(x,y)}{\sqrt{x^2+y^2}}$, which formally means (in terms of invariance) that all dots (x,y) on the same ray should land on the same point where the ray intersects the circle, and for that reason we add the normalization.

Lastly, there's also the case of projecting our data to values bigger than 0 that sums to one, this could be done using the mapping of our output vector $x = (x_1, ..., x_n)^T$ to $softmax(x)$



5. Recurrent Neural Networks (RNN)

As of now, we only considered inputs that were fixed and predefined, that had similar dimension to the matrices that made up our architecture (either in a fully connected MLP or in Conv NN). Let us relax that assumption, and discuss new types of inputs.

5.1 Sequence signals

we consider Sequence signal as any signal that has the following properties

- the prediction at some time t depends on history (say, time $t - \tau$)
- the input may have varying length

for example, Google Translate receives its input as a sequence signal, as the translation of some sentence relays on the words that were previously inserted, and the sentence provided could be of any length.

5.2 Recurrent NN

the main idea would be that we can receive an input, perform some algorithm to it and then output it to a future version of our model, that would also be able to receive future input, as well as the prediction of the previous model. That is, for every time stamp t, we

- receive an input x_t
- receive a hidden state h_{t-1} that represents an output of the previous time stamp
- produce a prediction y_t
- produce a memory hidden state h_t that can be transferred to future states

generally speaking, we can think of this model as some generic predictor h that is constantly receiving inputs and constantly predicts outputs, or (unfolding) as a sequence of predictors, as the below figure describes. This model is known as `Elman network` (See figure below). notice that every input x_i and every output o_i are of constant size, yet the number of such inputs and outputs is unknown, and isn't necessarily constant. usually we will look only at the output at the very end, and back to

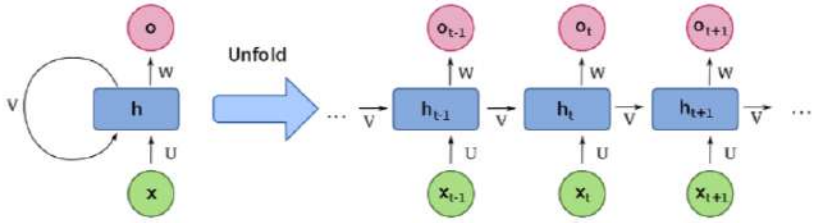


Figure 5.1: Elman network diagram. U, V, W are some matrices that we multiply x_i, o_i and h_i with (aka the network parameters).

our Google Translate example, this is intuitively clear as a portion of a sentence may not have any meaning up until it is finished.

we can formally write h_t as

$$h_t = \sigma_h(U_h x_t + V_h h_{t-1} + b_h)$$

where W_h and U_h are some matrices that we learn in the iterative learning process (they are not different per t, as we do not know how big t is), and b_h is the bias. our prediction o_t would be

$$o_t = \sigma_y(W_o h_t + b_o)$$

In general notation we could also write $h_t, o_t = N_\theta(h_{t-1}, x_t)$

R another common approach is to concatenate x_t and h_{t-1}, i.e $h_t = \sigma_h(U[x_t, h_{t-1}] + b_h)$

5.2.1 RNN architectures

One key feature of Elman's network model is that we have the freedom to play with the number of inputs/outputs. In most cases we distinguish between the following designs

- `one-to-one`: the input and the output are of fixes size, this is similar to models we already saw before.
- `one-to-many`: fixed size input, and arbitrary size output. for example we input an image and output its textual representation
- `many-to-one`: exactly the opposite. for example - trying to predict stock price given many future values
- `many-to-many`: for example, translating Hebrew to English.

5.3 Back Propagation Through Time (BPTT)

In regular back-propagation, The loss was only a function of the weights and biases, and therefore the derivatives of the loss were with respect to the weights and biases. In RNN, on the other hand, there are more inputs to the loss, so we have to be careful when deriving with respect to the weights. Lets denote the model prediction and hidden state as

$$h_t, o_t = N(\theta, h_{t-1}, x_t)$$

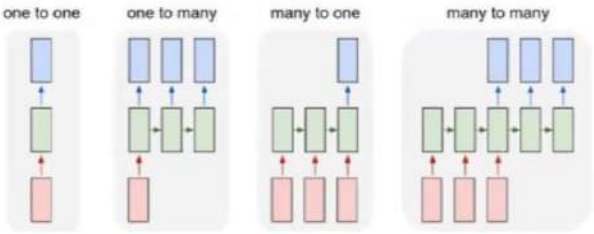


Figure 5.2: Various RNN architectures.

that is, the network outputs both h_t and o_t. we also denote $N_h = h$, $N_o = o$ (as N outputs a tuple). from here, we may choose some lose function L (that could, for example take the squared differences between N and some labels y), but regardless of how we chose L, there would be a term in which we specifically derive the network's prediction, so let us focus on that term

$$\frac{dh_t}{d\theta} = \frac{d}{d\theta}N_h(\theta, h_{t-1}, x_t) = \frac{\partial N_h}{\partial \theta}\frac{\partial \theta}{\partial \theta} + \frac{\partial N_h}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial \theta} + \frac{\partial N_h}{\partial x_t}\frac{\partial x_t}{\partial \theta} = \frac{\partial}{\partial \theta}N_h(\theta, h_{t-1}, x_t) + \frac{\partial}{\partial h_{t-1}}N_h(\theta, h_{t-1}, x_t)\frac{\partial h_{t-1}}{\partial \theta}$$

where $\frac{\partial x_t}{\partial \theta} = 0$ because the input is not a function of the network parameters. Now comes the tricky part, as we recursively expand the values of $\frac{\partial h_t}{\partial \theta}$ for every t to achieve the final results

$$\frac{\partial h_t}{\partial \theta} = \frac{\partial}{\partial \theta}N_h(\theta, h_{t-1}, x_t) + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t}\frac{\partial}{\partial h_{j-1}}N_h(\theta, h_{j-1}, x_j)\right)\frac{\partial}{\partial \theta}N_h(\theta, h_{i-1}, x_i)$$

which can be written in shorter notation as

$$a_t = b_t + \sum_{i=1}^{t-1}\left(\prod_{j=i+1}^{t} c_j\right)b_i$$

where

$$a_t = \frac{\partial h_t}{\partial \theta},\ b_t = \frac{\partial}{\partial \theta}N_h(\theta, h_{t-1}, x_t) \text{ and } c_t = \frac{\partial}{\partial h_{j-1}}N_h(\theta, h_{j-1}, x_j)$$

this gives us a way to derive the loss - suppose $L = \frac{1}{T}\sum_{t=1}^{T} l(y_t, o_t)$, for some loss function l (say, $l = l_2$) and T samples. The gradient is given by

$$\frac{\partial L}{\partial \theta} = \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial \theta} = \frac{1}{T}\sum_{t=1}^{T}\frac{\partial l(y_t, o_t)}{\partial o_t}\frac{\partial o_t}{\partial h_t}\frac{\partial h_t}{\partial \theta}$$

note that l is some loss function over o_t, so we can easily derive it with respect to o_t. Furthermore, $o_t = \sigma_y(W_y h_t + b_y)$ so we can derive it with respect to h as well. the third factor is given from the calculation above, and this concluded what we wished to achieve.

R Though the derivation of $\frac{\partial h_t}{\partial \theta}$ from above can be computed fully, for large T this gets extremely long. There are some strategies for dealing with such problem, but we will not discuss them in the course

5.4 Vanishing/Exploding gradient problem

when we look at the term $\prod_{j=i+1}^{t} c_j$, it is reasonable to think that if c_j is relatively big, the product would converge exponentially. This will obviously cause a problem, as our gradient iteration will not bring us towards the solution (in fact, it will probably won't provide any solution). On the contrary, if the term c_j is sufficiently small, we will face the opposite problem of vanishing gradient, and the network will work hard to converge. the problem described above isn't solely related to RNNs - For example, a very deep NN could encounter vanishing gradients too. To minimize the effect, `skip connection` was introduced. This means, in general, that some input x (of some layer) would not be multiplied (or convolved) with the layers that follows him , and instead would be transferred directly deeper down the network blocks.

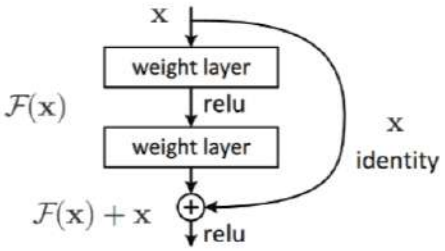


Figure 5.3: skip connection diagram. for some predefined weight layer that takes x and outputs $F(x)$, instead of using $F(F(x))$ we have $F(x)+x$

5.4.1 First solution: Long-Short-Term-Memory (LSTM)

LSTM is an architecture that was introduced to try and reproduce the concept of skip connections in RNNs. the model itself is comprised of a concatenation of identical "cells", and each cell has a few components that are in charge of various applications

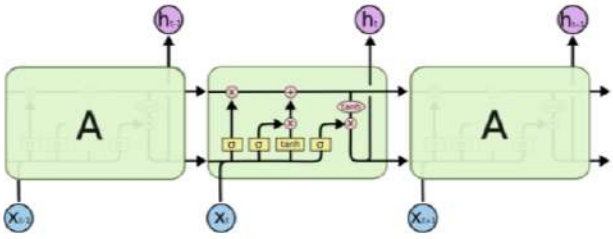
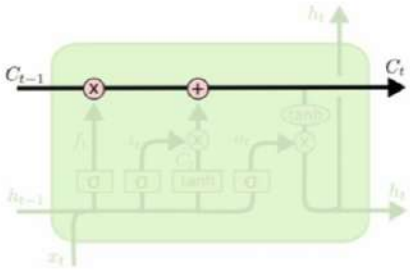


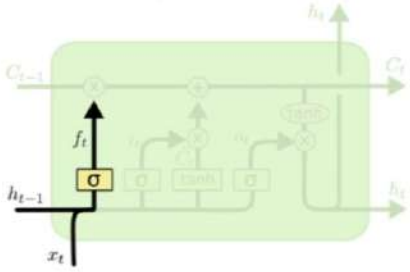
Figure 5.4: LSTM is a concatenation of cells

let us break the cell to its main components:

1. `Cell State`: represented with c_t, is the analog of the `skip connection`. The cell state is to be thought of as the "memory" that it passed along from previous cells. With that intuition in mind, notice that c_t can be multiplied by a number $\in (0,1)$ (output of some `sigmoid`), and this will allow us to control how much from past iterations we wish to "remember" (0 - forget the history and 1 let it pass fully). Finally, our c_t is added with "+" to our processing of h_{t-1} and x_t, which is important as we remember that when deriving with respect to our network parameters θ, $\frac{\partial c_t}{\partial \theta} = 0$, and therefore c_t will not contribute to vanishing gradients.

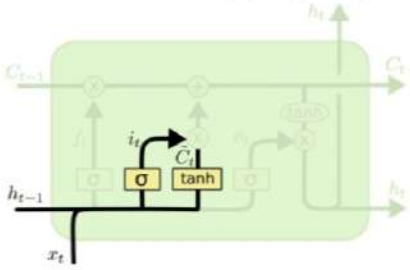


2. `Forget State`: both the current input x_t and the output from the previous cell h_{t-1} are inserted as input to a layer that outputs some value $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \in (0,1) vec$ ($[\cdot,\cdot]$ means concatenation).This value, as described before, is what chooses whether to "forget" or "remember" c_t.



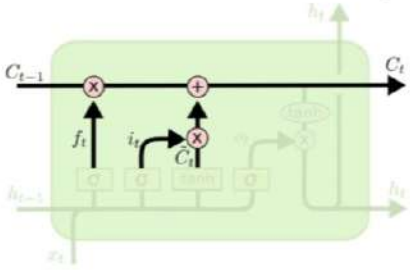
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

3. `Update (Input) State`: the current input and previous output can update the previous memory by adding to it new memory $\tilde{C} = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$. Before $\tilde{C}$ is added to the previous c, we also calculate $i = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$ which will provide a percentage for how much of $\tilde{C}$ we wish to transfer. We use tanh as it allows us an update that is both addition and subtraction (as $\tanh(x) \in [-1,1]$)



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

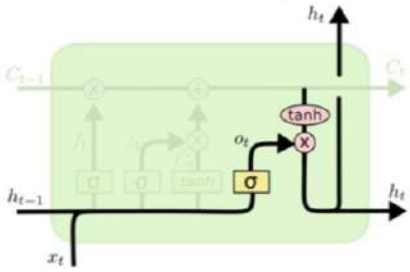
4. `Forgetting and Updating Cell State`: This state is responsible of updating the memory c given the outputs of the Update state and Forget state, that is it sets $c_t = f_t \times c_{t-1} + i_t \times \tilde{C}_t$, where $\times$ is `element-wise` multiplication.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

5. `Output State`: The last state is responsible for generating the output of the current cell, which is the hidden state h_t, as well as propagating h_t to the text cell alongside C_t. Specifically we have $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$ and $h_t = o_t \times \tanh(c_t)$, which indicates that the hidden state

is some factor of the output with the previous layers memory.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

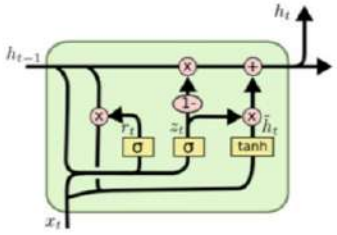
—

Provided an LSTM architecture, the problem of vanishing gradients greatly reduces, as the cell state can pass either a long or short memories without changing them, which is what skip connection is all about. This allows us to build a deep model with elongated time dependencies. Do notice though that there are still assignments to the parameters of the LSTM that will result in vanishing gradient, and therefore the method isn't bullet proof.
the above could be concisely summarized to the following set of instructions, where notice we've added matrices U, which won't change the architecture (you could either use $W \cdot [h_{t-1}, x_t]$ or $Wh_{t-1} + Ux_t$, only notice that the number of learned variables may change):

`forget gate` - controls what is kept vs forgotten from previous cell states:
$$f_t = \sigma(W_f h_{t-1} + U_f x_t + b_f)$$
`input gate` - controls what parts of the new cell content are written to cell:
$$i_t = \sigma(W_i h_{t-1} + U_i x_t + b_i)$$
`output gate` - controls what parts of cells are output to hidden state:
$$o_t = \sigma(W_o h_{t-1} + U_o x_t + b_o)$$
`new cell content` - is the new content to be written to the cell:
$$\tilde{c}_t = \tanh(W_c h_{t-1} + U_c x_t + b_c)$$
`cell state` - forgets some content from last cell state and writes some new cell content
$$c_t = f_t \times c_{t-1} + i_t \times \tilde{c}_t$$
`hidden state` - outputs some content from the cell
$$h_t = o_t \times \tanh(c_t)$$

5.4.2 Second solution: Gated Recurrent Unit (GRU)

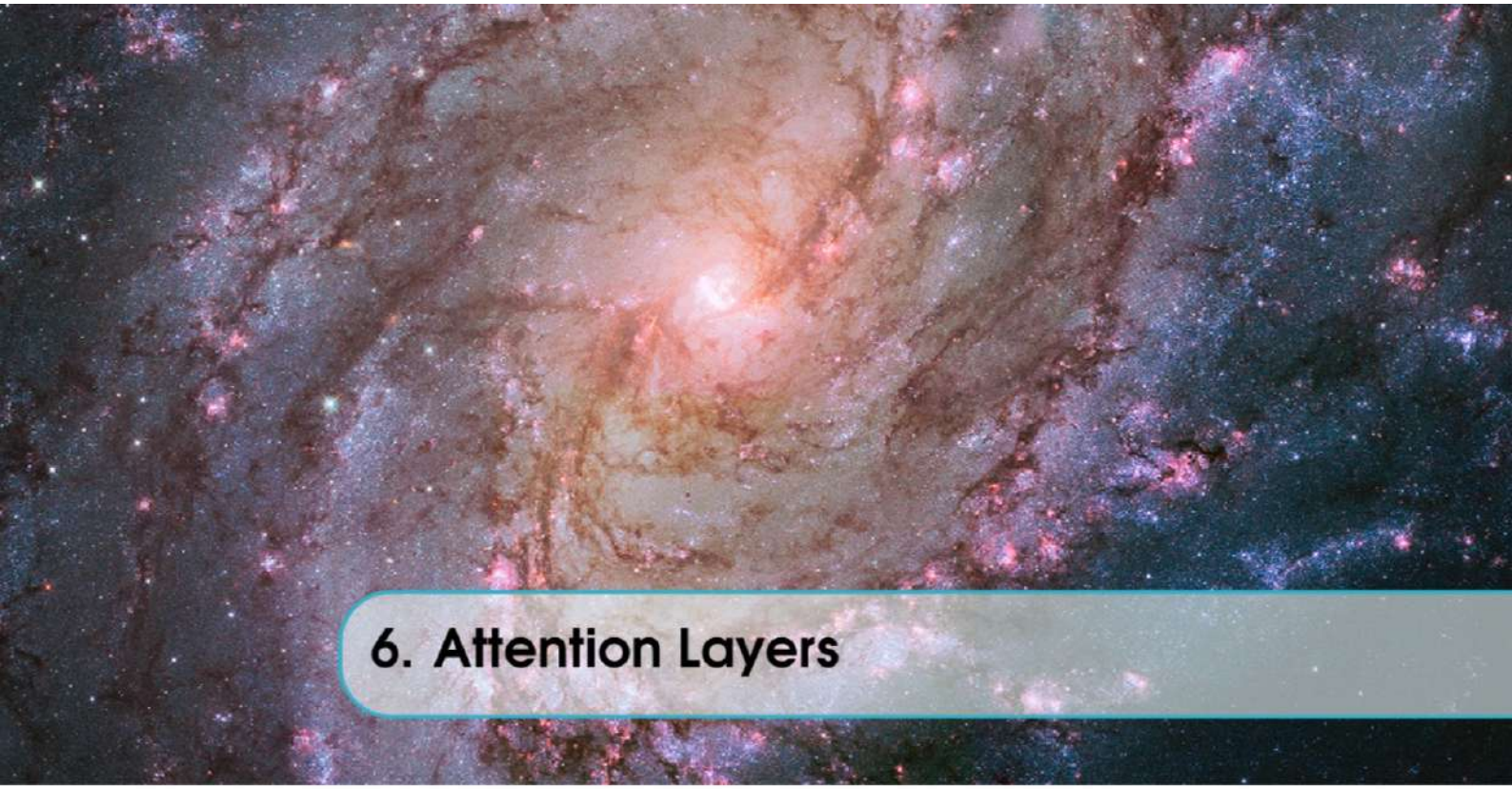
The GRU is a recurrent unit that was introduces by `Cho.et al(2014)`, that has similar performance as the basic `LSTM`, though is more compact and has less inner parameters. let us briefly go over its main components:



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

Figure 5.5: GRU cell

The first thing to notice is we do not use c_t any more - the hidden state h_t will hold the information that was previously attributed to c_t. More formally, h_t can be either updated (using z_t) or rebooted (using r_t) in the following manner: if we choose to reset, r_t will be set to $(0,0,...,0)$ and z_t would be set to $(1,1,...,1)$. This will result in an updated hidden state h_t that is only affected by the input of the current time stamp, x_t. eventually the hidden state will be a factor of both the updated state $\tilde{h}_t$ and the previous state h_{t-1}, as a sum with factoring by z_t - if z_t is closer to 1, h_t will be less similar to the previous h_{t-1}, and vice versa (see figure). Also notice that the update is performed using tanh, as to incorporate both addition and subtraction.



6. Attention Layers

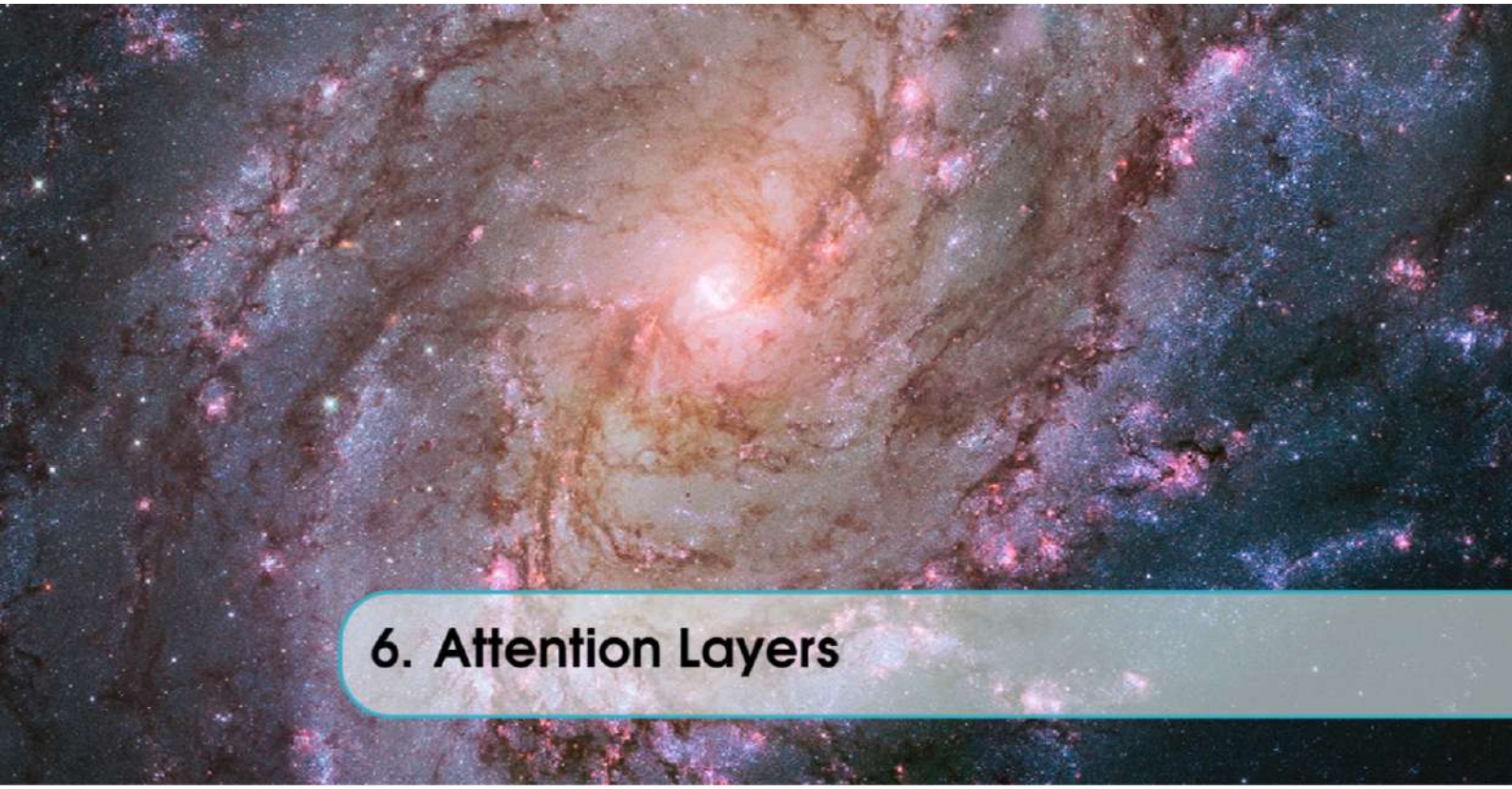
As we dive deeper to the realm of deep neural networks, we are facing the realization that getting acquainted with the inductive bias of our network, that is, finding how differently our network tends to learn the data based on its specific architecture, is indeed a key component to the success of the network. A great example to that is the usage of CNNs instead of MLPs, as we had some prior knowledge of LTI, and this allowed us to specifically build our model to use that prior knowledge. We can take this general concept to the next level, as in broader terms, we would also like our network to distinguish what is essential and what is bland, that is, equip our network with some way to divide its "attention" between samples that have significant meaning, and samples that does not. In this chapter we will see how the intuition above was translated to what is called an `Attention Layer`

6.1 The problem with encoder-decoder models

encoder-decoder are a general term to models that have (no surprise here) two phases

- `encoding` - we receive an input (say, some sentence), and encode it to hidden states. This phase is finished when the input was fully processed (for example, when we have reached the end our the sentence)
- `decoding` - we decode the data possessed in the hidden layers to some meaningful output (for example, some translation of text)

The problem is that our input may very well be extremely long, with some complex dependencies inside of it, but still, at some point t in time we expect our encoder to give us some output vector o_t of constant length that should contain all that information. The problem here is evident, and indeed it was shown that large and complex inputs weren't successfully decoded to a meaningful output.
another example of the complexity of the problem arises when discussing the following learning task: given some image input - provide a textual representation to it. Why is this so difficult? consider an image of a girl throwing Frisbee. Even if the image is very clear, it may consist of various other parts - for example, there ought to be some background (trees maybe?), etc.. Long story short, as we feed our input through some fancy deep vision CNN concatenated with an LSTM cell, we still



6. Attention Layers

As we dive deeper to the realm of deep neural networks, we are facing the realization that getting acquainted with the inductive bias of our network, that is, finding how differently our network tends to learn the data based on its specific architecture, is indeed a key component to the success of the network. A great example to that is the usage of CNNs instead of MLPs, as we had some prior knowledge of LTI, and this allowed us to specifically build our model to use that prior knowledge. We can take this general concept to the next level, as in broader terms, we would also like our network to distinguish what is essential and what is bland, that is, equip our network with some way to divide its "attention" between samples that have significant meaning, and samples that does not. In this chapter we will see how the intuition above was translated to what is called an `Attention Layer`

6.1 The problem with encoder-decoder models

encoder-decoder are a general term to models that have (no surprise here) two phases

- `encoding` - we receive an input (say, some sentence), and encode it to hidden states. This phase is finished when the input was fully processed (for example, when we have reached the end our the sentence)
- `decoding` - we decode the data possessed in the hidden layers to some meaningful output (for example, some translation of text)

The problem is that our input may very well be extremely long, with some complex dependencies inside of it, but still, at some point t in time we expect our encoder to give us some output vector o_t of constant length that should contain all that information. The problem here is evident, and indeed it was shown that large and complex inputs weren't successfully decoded to a meaningful output.
another example of the complexity of the problem arises when discussing the following learning task: given some image input - provide a textual representation to it. Why is this so difficult? consider an image of a girl throwing Frisbee. Even if the image is very clear, it may consist of various other parts - for example, there ought to be some background (trees maybe?), etc.. Long story short, as we feed our input through some fancy deep vision CNN concatenated with an LSTM cell, we still

output some vector that should in theory represent the entire image (and not only the girl throwing Frisbee), What we would have wanted is some way to tell the network which parts of the image are more relevant than others, aka, to which pixels should the network pay more attention

6.2 The solution - Attention Layer

(based on the paper `Attention Is All You Need`)
Attention Layer is essentially a mapping/table, that given keys $\{k_i\}$, values $\{v_i\}$ and a query q

- compute the proximity[1] d_i between q and every key k_i
- apply `softmax` to define a probability α_i that signifies the relevance of each value
- produce some soft prediction, mean value with respect to the distribution

several variants exists, and the simplest one is

$$d_i = \frac{\langle k_i, q \rangle}{\sqrt{N}},\ \alpha_i = softmax(d_i) \text{ and } v_{out} = \sum_i \alpha_i v_i$$

where N is the size of the table
or shortly in matrix form, where q is a query vector, K is a matrix where each row is a vector of keys and V is a matrix where each row is a set of values:

$$v_{out} = V^T softmax(Kq)/\sqrt{N}$$

another option is to concatenate k_i and q, and add some learnable weights (a vector w, and a matrix W). In that case $d_i = w^T \tanh(W[k_i, q])$, where we use tanh to get both positive and negative values. lastly, there's also the variant of

$$d_i = \frac{\langle W_K k_i, W_Q q \rangle}{\sqrt{N}},\ \alpha_i = softmax(d_i) \text{ and } v_{out} = \sum_i \alpha_i W_V v_i$$

which is a common variant, that allows k and q to have different dimensions.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. This provides us with a concise formula for the attention mechanism

$$Att(Q, K, W) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

lets see how this layer can be added to the task of image captioning to provide significantly improved results

6.2.1 Image Captioning

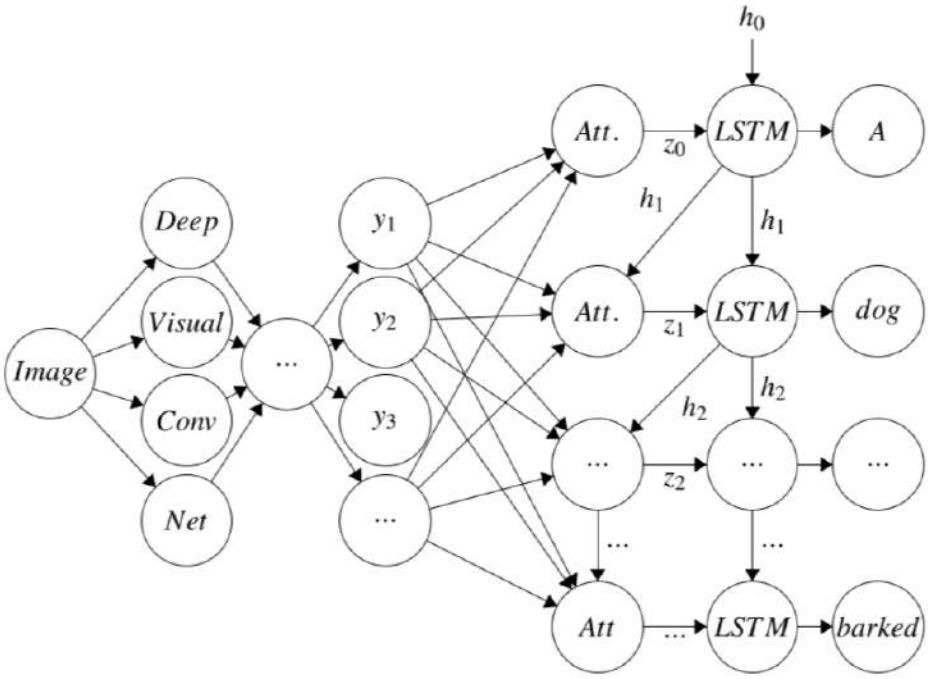
As a reminder, the task in hand is to provide meaningful caption to a given image, and this should be done by taking into consideration different regions that may be more relevant. In this approach, we will define attention layers that will do just that. Consider the following architecture (See figure)

- the input is fed to a Deep Vision CNN
- we zoom in on some (non-output) layer that outputs a vector y
- every element y_i is to be considered both as key and as value, and is connected to Attention models that outputs some z_j

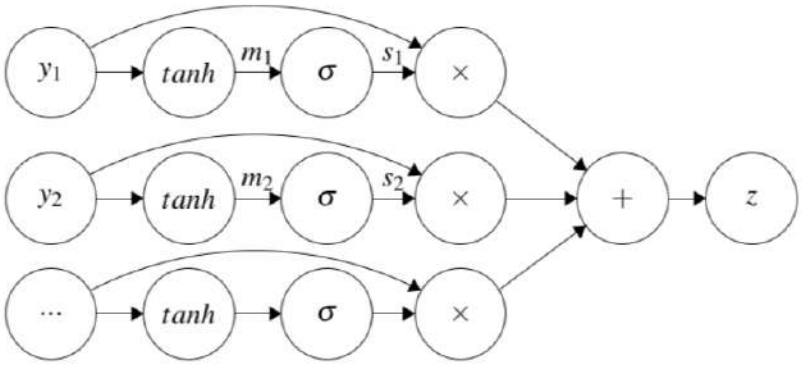
[1] Proximity could be calculated in various ways. two common ones are inner product and tanh

- every z_j is fed as sequential input to an LSTM cell
- the LSTM cell outputs a hidden state h_i and some output (for example, a word), where h_i is inserted back to the Attention layer as a query q ,

this can be visualized as



where the Attention layer is as followed ($\sigma \equiv softmax$):



In fact, the input to tanh is both y_i and the output of the *LSTM* cells, which is the hidden state (omitted from the diagram to avoid cluttering). Notice that every attention layer is provided with all $\{y_i\}$. under these notations, we have $m_i = \tanh(y_i W_{y_i} + cW_c)$, where c is really the hidden states memory h. we applied tanh so that very high/small values would have small differences. This is not obligatory, and we could also use inner product, as we have described in the previous section. these m's are then fed to a softmax function $s_i = e^{m_i} / \sum e^{m_i}$, and finally, the output z is the expectation of y under the probabilities provided as s_i (was denoted α_i before), i.e $z = \sum s_i y_i$. It is important to understand that the output of the CNN is provided without the notion of a future sentence that is to be constructed - it only gives us the information for the various key features of an image. the

attention layer takes that data and combined with the LSTM cell provides a mapping between the features in the image and their appropriate positioning in a sentence.

R taking $v_{out} = \sum \alpha_i v_i$ is known as `soft` Attention. on the other hand, `hard` Attention samples one of the v_i according to α_i. This may be relevant when taking a single value makes more sense than averaging

■ **Example 6.1 — Neural Machine Translation (NMT).** The general topic of translation from one language to another is known as Neural Machine Translation (NMT). as this was previously discussed, let us tackle the task of translating sentence in which the words may be related to previous or succeeding words. with RNNs, one approach would be to parse our input sentence both from beginning to end and vice versa, and for every time step generate a feature vector. From here, all feature vector would be fed to an attention mechanism, where it will help with deciding which words are the most relevant based on the words that have already been read.
such mechanism generates different weights to the attention layers in every time stamp, and a nice representation to this would be to look at a plot in which the vertical axis represents the translated sentence, the horizontal axis represents the given sentence, and the color is associated with every pixel represents how the weights distribute among the input words. consider a trivial translation

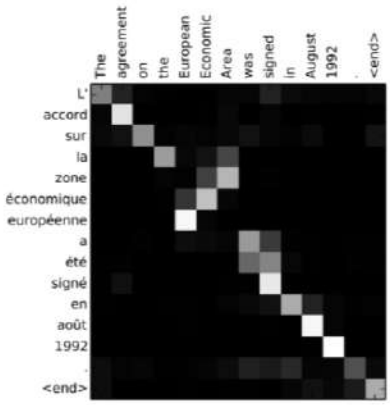


Figure 6.1: Alignment matrix - from English to French

mechanism, for which every word in the input is directly translated to a word in the output. the alignment matrix would be strictly diagonal, as there is one to one correspondence between every input word to a single output word. figure 6.1 demonstrates that using attention layers this is not the case - and some words were translated with respect to two or more words, so the matrix isn't diagonal. ■

6.2.2 Multi-Headed Attention

Though a word may be associated to various other words in the sentence (for example, in "I gave my dog Charlie some food", the word "gave" relates to "my dog", "Charlie" and "some food"), the association might be different, and there is no apparent reason for why we should sum their affect in a single linear combination (as we would using a single attention layer). To account for those different relations, we can apply Multiple attentions and concatenate their results

$$MultiHead(Q,K,V) = Concat(head_1,...,head_n)W_O$$

$$head_i = Attention(QW_{i,Q}, KW_{i,k}, VW_{i,V})$$

It is important to notice that each head is independent and has different learnable parameters that are (in most cases) randomly initialized. This also indicates that they will contain different values throughout the learning process.

6.2.3 Self Attention

What is common among the following sentences?

- She loves eating dates
- She took him on a date
- What is your date of birth
- Not to date myself, but...

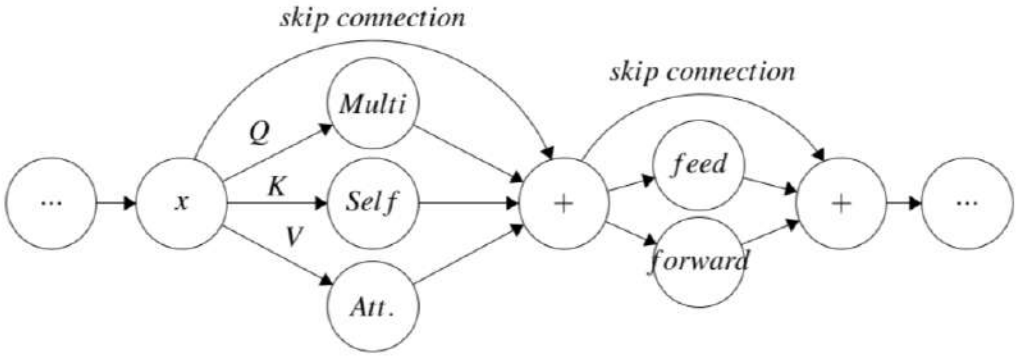
the same word is being used, but its meaning is completely different. Thus, if we wish to translate the sentence, we should relate to the word `date` as if it is a completely new word every time. In other words, to properly translate the sentences above, it would be best if we first understand its contextualization. Self Attention does just that - for a given input x, we define the all the inputs (the query, the key and the value) as x, and output $y = Attention(xW_{query}, xW_{key}, xW_{value})$, where $Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$. The output of the self attention layer reflects how each words related to every other word in the sentence.

6.3 Transformer Network

In our path of describing more and more suitable network architectures, we keep in mind that

- if our input is of varying length: MLP (and CNN) will fail
- if we deal with input that has inner relations with itself (for example, the first word of a sentence is related to the last): CNN would probably fail as well
- if there's no progression axis, that is, only in some time in the future enough input would be given to extract a legitimate and understandable output: RNNs might fail

To provide a solution to all the cases described above, Transformer networks were introduced. Those are MLP networks with the addition of a Multi-head self-attention layer. more specifically, a transformer networks is comprised of blocks, where each block can be represented as followed



that is, we concatenate blocks where each block in itself contains a Multi-head Self-Attention and a feed-forward fully connected network, where the output of the first is fed as an input to the latter (with skip connections as well that should prevent vanishing gradients). notice that in order to add x with the output of a multi headed self attention, the attention mechanism should output a tensor of

same dimensions as x. The diagram should be read as followed - given some input x (say, a sentence), the multi-head self attention layer generates an embedding for each word, and the feed forward network works on each word embedding independently (This allows us to work on inputs of varying length).
The provided architecture can handle inputs of varying length, and it considers every connection between every segment in out input. furthermore, changing the order representation of our input has no effect (it is invariant), as the self attention layer handles that. Lastly, as the fully connected feed forward layer parses every segment separately, and the heads in the attention layer are independent, we are able to use parallel computing to speed up the training process.

as a side note, a performance comparison for various layers is provided
complexity per layer:

- self attention layer: runs in $O(n^2 d)$ as the attention mechanism compares every word with every word, where n is the number of words in the input and d the dimension of every word
- RNNs: $O(nd^2)$, as we parse every word and whenever we get a feature vector for it, matrix multiplication is performed
- Conv NN: for n pixels, d channels and k kernel size, we have $O(kd^2 n)$, as for every pixel we perform convolution which takes the sum of every pixel in the kernel that is both in the image and in the kernel, that is for every channel (d for the kernel and d for the image, though not necessarily equal)

sequential operations

- self attention layer: $O(1)$, as it is automatically aligning every word pair
- RNN layer: $O(n)$ because in every step we perform a calculation on a single sequential input
- CNN layer: $O(1)$ as all the computation is done using single step of convolution

max path length (after how many steps every would would relate to every word)

- self attention layer: $O(1)$ because we match every pair from the beginning
- RNN layer: $O(n)$ because the first and last words demands n steps
- CNN layer: $O(log_k(n))$ as the two most distant pixels will be narrowed down under the same kernel frame only after the image was convolved with the kernel $log_k(n)$ times

6.3.1 Encoder-Decoder Transformer for NMT

As an example to the encoder-decoder architecture, a transformer is presented in figure 6.2. the block to the left is one segment (the encoder), which in itself is a transformer that is very deep (n such blocks). the right segment is somewhat different, though is still considered a transformer (that uses two attention mechanisms). The encoder receives the inputs, and its output is some coding of the input sentence (more specifically, the input was encoded using n Transformers). Following the encoder comes the decoder - the coded data which was the output of the encoder is inserted both as keys and as values to a multi head attention layer within the decoder. regardless of that input, every word in the output of the decoder itself is fed right back to the decoder, and is set as query input to the multi head attention layer. This allowed every word in the output to relate in some manner to the entire sentence (and in every such relation, new information may be generated). The masked multi head attention is an attention mechanism that only matches words with words that was already provided in the sentence (only words that came prior to the currently processed word). the above could be summarized to the following steps:

1. the input sentence is inserted to the encoder (all words simultaneously)
2. the input is embedded to some vector representation (using GloVe/Word2Vec for example)

3. the input is fed forward through n Transformers (which are what the encoder is comprised of)
4. the output of the encoder is fed both as keys and as values to a multi head attention in the decoder
5. in an iterative process, the decoder synthesises the output sentence as followed
   (a) every single word that was the output of the decoder is first embedded and encoded (in the first iteration, a randomly generated word is initialized)
   (b) we perform masked multi head attention on the word with all the words that preceded it
   (c) the output of the previous layer is inserted as query to a multi head attention layer (with keys and values provided from the encoder as previously described)
   (d) the output is both: multiplied with a matrix that has dimensions which is the size of the output language (a very big number) and normalized with softmax to achieve output probability AND returns as input back to the decoder, to parse the following words

the loss is defined per word in the target language, and is mostly cross entropy

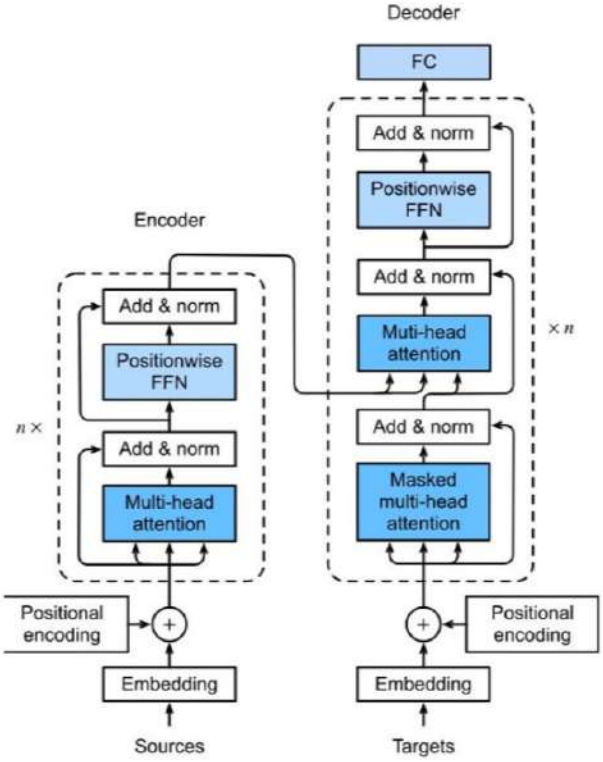


Figure 6.2: Encoder-Decoder Transformer, where the position-wise FFN is defined as $FNN(x) = relu(xW_1 + b_1)W_2 + b_2$ and the positional encoding (which tries to map between every word in the input to some position) was defined as a mapping from the position of a word pos (with corresponding index $2i$ of $2i+1$) to sine and cosine functions - $PE(pos, 2i) = \sin(\frac{pos}{(10^5)^{2i/d_{model}}})$ and $PE(pos, 2i+1) = \cos(\frac{pos}{(10^5)^{2i/d_{model}}})$

6.3.2 Bidirectional Encoding Representation Transformer (BERT)

The above architecture brought to life the notion that one can freely concatenate a transformer to provide general purpose network for which the order of the input doesn't matter (it can though, if we add positional encoding), all the elements interact with each other, and the input itself can be of varying length. One such example was BERT model, introduced by Google AI in 2019. Lets take a glimpse to the world of BERT, providing a high level description two language models:

- `Answering Questions`: consider a data-set of the following form - there are questions, and there are references to material that should contain the answers/relevant material to answer these questions. The labeling to the data is a `start` and `end` indexes, such that for every question the label is the references from `start` to `end`, and in that segment the solution should be written. The output of the $i'th$ transformer layer provides the probability (softmax-ing the tokens) that a given token represents `start` or `end`, or neither.
- `Translation`: we use a pre-trained BERT, trained both on next sentence prediction (NSP) and on filling missing words (masked language model - MLM). as for the latter, the notion is to filter out some percentage of the words in the sentence (i.e replacing say 15% of the words with some arbitrary token) and let the BERT model fill in the masked words. with respect to the missing words, a loss function is defined. The NSP is being performed while the words are still masked (from the MLM process), and its general process is to choose between two sentence options which is more suitable to positioned before the other. The pre trained models were trained mostly in an unsupervised framework, as most of its data wasn't labeled (at least for the masking task, no labeling is needed as one could simply compare the masked version with the input). Now, with the trained model in hand, one could, for example, use it as a method of embedding the input (instead of using GloVe or word2vec) in a fashion that also provides context (the word date would be embedded differently with respect to the sentence in which it was given). Another option is to train BERT on our own data, as a some what continuation to the training that was already been done, and provided the already chosen weights. This would provide a great starting point, and would tune the BERT model to suit better to our specific task.

Two familiar models that utilize the concepts introduced above are `GPT-2` and `GPT-3`. These models architecture is based on stacking decoders in consecutive order, and they are using enormous data-sets (large subset of all text available online) and parameters (in the billions) to achieve

$$head_i = Attention(QW_{i,Q}, KW_{i,k}, VW_{i,V})$$

It is important to notice that each head is independent and has different learnable parameters that are (in most cases) randomly initialized. This also indicates that they will contain different values throughout the learning process.

6.2.3 Self Attention

What is common among the following sentences?

- She loves eating dates
- She took him on a date
- What is your date of birth
- Not to date myself, but...

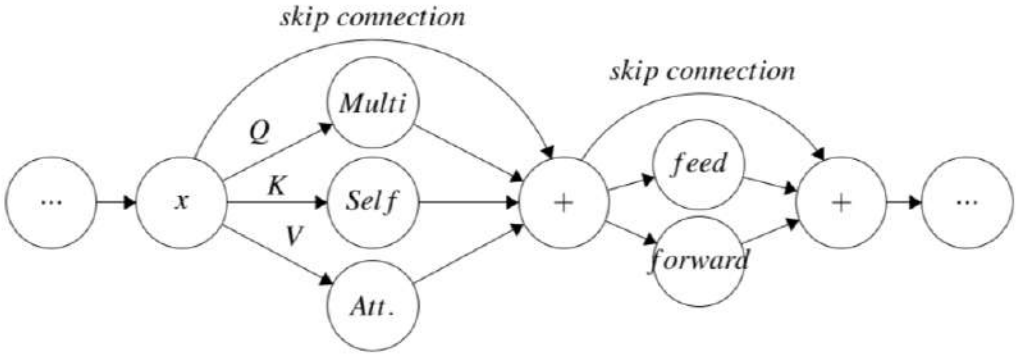
the same word is being used, but its meaning is completely different. Thus, if we wish to translate the sentence, we should relate to the word `date` as if it is a completely new word every time. In other words, to properly translate the sentences above, it would be best if we first understand its contextualization. Self Attention does just that - for a given input x, we define the all the inputs (the query, the key and the value) as x, and output $y = Attention(xW_{query}, xW_{key}, xW_{value})$, where $Attention(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$. The output of the self attention layer reflects how each words related to every other word in the sentence.

6.3 Transformer Network

In our path of describing more and more suitable network architectures, we keep in mind that

- if our input is of varying length: MLP (and CNN) will fail
- if we deal with input that has inner relations with itself (for example, the first word of a sentence is related to the last): CNN would probably fail as well
- if there's no progression axis, that is, only in some time in the future enough input would be given to extract a legitimate and understandable output: RNNs might fail

To provide a solution to all the cases described above, Transformer networks were introduced. Those are MLP networks with the addition of a Multi-head self-attention layer. more specifically, a transformer networks is comprised of blocks, where each block can be represented as followed



that is, we concatenate blocks where each block in itself contains a Multi-head Self-Attention and a feed-forward fully connected network, where the output of the first is fed as an input to the latter (with skip connections as well that should prevent vanishing gradients). notice that in order to add x with the output of a multi headed self attention, the attention mechanism should output a tensor of

same dimensions as x. The diagram should be read as followed - given some input x (say, a sentence), the multi-head self attention layer generates an embedding for each word, and the feed forward network works on each word embedding independently (This allows us to work on inputs of varying length).
The provided architecture can handle inputs of varying length, and it considers every connection between every segment in out input. furthermore, changing the order representation of our input has no effect (it is invariant), as the self attention layer handles that. Lastly, as the fully connected feed forward layer parses every segment separately, and the heads in the attention layer are independent, we are able to use parallel computing to speed up the training process.

as a side note, a performance comparison for various layers is provided
complexity per layer:

- self attention layer: runs in $O(n^2 d)$ as the attention mechanism compares every word with every word, where n is the number of words in the input and d the dimension of every word
- RNNs: $O(nd^2)$, as we parse every word and whenever we get a feature vector for it, matrix multiplication is performed
- Conv NN: for n pixels, d channels and k kernel size, we have $O(kd^2 n)$, as for every pixel we perform convolution which takes the sum of every pixel in the kernel that is both in the image and in the kernel, that is for every channel (d for the kernel and d for the image, though not necessarily equal)

sequential operations

- self attention layer: $O(1)$, as it is automatically aligning every word pair
- RNN layer: $O(n)$ because in every step we perform a calculation on a single sequential input
- CNN layer: $O(1)$ as all the computation is done using single step of convolution

max path length (after how many steps every would would relate to every word)

- self attention layer: $O(1)$ because we match every pair from the beginning
- RNN layer: $O(n)$ because the first and last words demands n steps
- CNN layer: $O(log_k(n))$ as the two most distant pixels will be narrowed down under the same kernel frame only after the image was convolved with the kernel $log_k(n)$ times

6.3.1 Encoder-Decoder Transformer for NMT

As an example to the encoder-decoder architecture, a transformer is presented in figure 6.2. the block to the left is one segment (the encoder), which in itself is a transformer that is very deep (n such blocks). the right segment is somewhat different, though is still considered a transformer (that uses two attention mechanisms). The encoder receives the inputs, and its output is some coding of the input sentence (more specifically, the input was encoded using n Transformers). Following the encoder comes the decoder - the coded data which was the output of the encoder is inserted both as keys and as values to a multi head attention layer within the decoder. regardless of that input, every word in the output of the decoder itself is fed right back to the decoder, and is set as query input to the multi head attention layer. This allowed every word in the output to relate in some manner to the entire sentence (and in every such relation, new information may be generated). The masked multi head attention is an attention mechanism that only matches words with words that was already provided in the sentence (only words that came prior to the currently processed word). the above could be summarized to the following steps:

1. the input sentence is inserted to the encoder (all words simultaneously)
2. the input is embedded to some vector representation (using GloVe/Word2Vec for example)

3. the input is fed forward through n Transformers (which are what the encoder is comprised of)
4. the output of the encoder is fed both as keys and as values to a multi head attention in the decoder
5. in an iterative process, the decoder synthesises the output sentence as followed
   (a) every single word that was the output of the decoder is first embedded and encoded (in the first iteration, a randomly generated word is initialized)
   (b) we perform masked multi head attention on the word with all the words that preceded it
   (c) the output of the previous layer is inserted as query to a multi head attention layer (with keys and values provided from the encoder as previously described)
   (d) the output is both: multiplied with a matrix that has dimensions which is the size of the output language (a very big number) and normalized with softmax to achieve output probability AND returns as input back to the decoder, to parse the following words

the loss is defined per word in the target language, and is mostly cross entropy

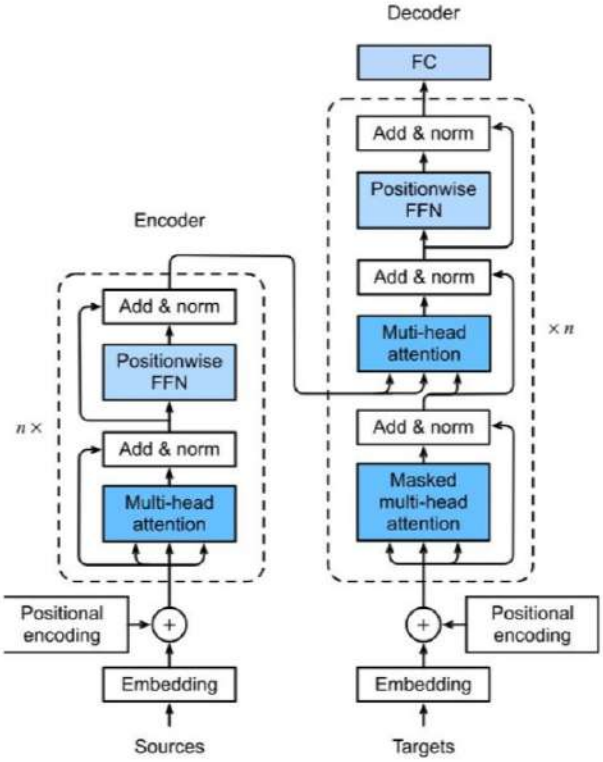


Figure 6.2: Encoder-Decoder Transformer, where the position-wise FFN is defined as $FNN(x) = relu(xW_1 + b_1)W_2 + b_2$ and the positional encoding (which tries to map between every word in the input to some position) was defined as a mapping from the position of a word pos (with corresponding index $2i$ of $2i+1$) to sine and cosine functions - $PE(pos, 2i) = \sin(\frac{pos}{(10^5)^{2i/d_{model}}})$ and $PE(pos, 2i+1) = \cos(\frac{pos}{(10^5)^{2i/d_{model}}})$

6.3.2 Bidirectional Encoding Representation Transformer (BERT)

The above architecture brought to life the notion that one can freely concatenate a transformer to provide general purpose network for which the order of the input doesn't matter (it can though, if we add positional encoding), all the elements interact with each other, and the input itself can be of varying length. One such example was BERT model, introduced by Google AI in 2019. Lets take a glimpse to the world of BERT, providing a high level description two language models:

- `Answering Questions`: consider a data-set of the following form - there are questions, and there are references to material that should contain the answers/relevant material to answer these questions. The labeling to the data is a `start` and `end` indexes, such that for every question the label is the references from `start` to `end`, and in that segment the solution should be written. The output of the $i'th$ transformer layer provides the probability (softmax-ing the tokens) that a given token represents `start` or `end`, or neither.
- `Translation`: we use a pre-trained BERT, trained both on next sentence prediction (NSP) and on filling missing words (masked language model - MLM). as for the latter, the notion is to filter out some percentage of the words in the sentence (i.e replacing say 15% of the words with some arbitrary token) and let the BERT model fill in the masked words. with respect to the missing words, a loss function is defined. The NSP is being performed while the words are still masked (from the MLM process), and its general process is to choose between two sentence options which is more suitable to positioned before the other. The pre trained models were trained mostly in an unsupervised framework, as most of its data wasn't labeled (at least for the masking task, no labeling is needed as one could simply compare the masked version with the input). Now, with the trained model in hand, one could, for example, use it as a method of embedding the input (instead of using GloVe or word2vec) in a fashion that also provides context (the word date would be embedded differently with respect to the sentence in which it was given). Another option is to train BERT on our own data, as a some what continuation to the training that was already been done, and provided the already chosen weights. This would provide a great starting point, and would tune the BERT model to suit better to our specific task.

Two familiar models that utilize the concepts introduced above are `GPT-2` and `GPT-3`. These models architecture is based on stacking decoders in consecutive order, and they are using enormous data-sets (large subset of all text available online) and parameters (in the billions) to achieve

7. Auto-Encoders

The following topic shifts our focus from classifiers and supervised framework to models that are generative, which can, for example, synthesize new samples (and are in general unsupervised).

7.1 What are Auto-Encoders

An Auto-Encoder is a learning model used to learn efficient codings of unlabeled data. The encoding is iteratively validated and constantly refined in order to resemble the provided input. In other words, an Auto-Encoder learns some representation of the input (in most cases, the representation is of lower dimension) by training a network to ignore insignificant data. Mathematically speaking, given some input I, an encoder encodes I to some chosen representation $E[I]$ and then decodes the encoding back to an output that is approximately I.

$$AE[I] = Decode[Encode[I]] \text{ such that } AE[I] \approx I$$

This is done by minimizing the `Reconstruction loss`, defined below.

Definition 7.1.1 — Reconstruction Optimization. let $D = D(\theta_D)$ and $E = E[\theta_E]$ denote the decoder and the encoder (that is, these are models with learned parameters θ_D, θ_E), and let Ω denote the sample space. For some norm $||\cdot||$, the reconstruction loss is $||AE[I] - I||$ and the objective is

$$\min_{\theta_E, \theta_D, I \in \Omega} ||AE[I] - I||$$

R notice that $AE[I] = I$ solves the minimization problem, but of course this is not what we wish to achieve, and some method exist to avoid such output.

Few variations exist when defining efficient encoding:

- `dimension reduction`: As noted before, the encoding is, in most cases, to a smaller dimension space. In those cases we can represent the model as a bottle neck - the input I had some

dimension d_I, yet $E[I]$ has some dimension d_E, where in theory $d_E \ll d_I$. finally $AE[I]$ has dimension d_{AE} which is close to d_I. As an example, one could think of d_I as the dimension of an image in MNIST (28×28) and d_E the dimension of an encoding of an MNIST image, which may vary to our liking (and if you've completed exercise 3, you'd know that even for $d_E \approx 10$, most of the image's prominent features were preserved)

- `sparse latent vectors`: another option (which in most cases comes as an addition to the dimension reduction) for efficient coding is that the input's representation in the encoded space will be sparse. This can be done by adding an ℓ_1 regularization term to the reconstruction loss $||AE[I]-I|| + \Phi[E[I]]$, where $\Phi[E[I]] = ||E[I]||_1$. This provides sparser solutions as the ℓ_1 loss tends to converge towards sparse solutions.
- `Contractive AEs`: in this case we add a regularization term which is the gradients of $E[I]$ with respect to I, that is $||AE[I]-I|| + \Phi[E[I]]$, where $\Phi[E[I]] = ||\nabla_I E[I]||$. This regularization term ensures that small changes in the input wouldn't propagate to the latent space (of the encoded input).
- `Denoising AEs`: In this version we consider the case where the input I is to be constructed from some noisy version $I+\eta$, where $\eta \sim N(0,\sigma^2)$. To allow that we define out loss as $||AE[I+\eta]-I||$. Notice that the "bottle-neck" effect doesn't occur in this version, it might even be the contrary - dealing with noise may very well result in a space larger then what was provided ($d_E > d_I$).

7.2 Variational Auto-Encoders (VAE)

Though the main built is the same (that is, we start with input, followed by its representation in the latent space, following by its reconstruction by the decoder), in VAE we give specific meaning to the encoding of I. Instead of treating the encoded version of I as some arbitrary vector in the latent space, we refer to it as a vector for which half of it is the `Mean` of some Gaussian distribution and the other half is the `Variance` of some Gaussian (Multivariate distribution). In other words, starting with a single input I, the encoder provides us with some Gaussian distribution (which also introduces noise!), for which the mean and variance are defined by $E[I]$ (and notice that we assume $\forall i \neq j$ $cov(x_i, x_j) = 0$). Then, the decoder would "sample" from that distribution to construct the decoded version.

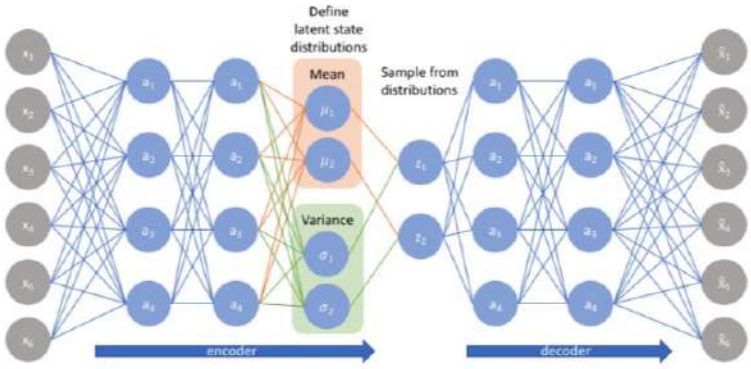


Figure 7.1: VAE diagram - In this example, the latent space is defined as both the mean $\mu = (\mu_1, .\mu_2)$ and the variance $\sigma = (\sigma_1, \sigma_2)$. z_1 is sampled from $N(\mu_1, \sigma_1)$ and z_2 from $N(\mu_2, \sigma_2)$

7.2.1 Training VAEs

Notice that the following problem arises: Though the weights θ_E define the latent space distribution, the decoder would use that distribution only to sample points, and from that point on θ_E are irrelevant, and wouldn't be considered when the sampled points are fed forwards through the weights θ_D. In other words, the weights of the encoder only implicitly effect the mean and the distribution in the latent space, but does not affect the loss, which is defined on $AE[I]$. This should be addressed, as we would like all of our weights to explicitly affect the loss, and one way to solve this is by analytically defining a parameter z that is a function of σ, μ and is also a representation of a sample from a gaussian distribution $N(\mu, \sigma^2)$: we sample some $\varepsilon \sim N(0,1)$ and then decode based on the value $z = \mu + \sigma \times \varepsilon$ (where $\times$ is element wise). as σ and μ are functions of θ_E, we have that z is a function of θ_E as well.
the above could be summarized to the following steps:

- given input I, encode it to $E[I]$
- based on $E[I]$, extract μ and σ
- sample some z from $N(\mu, \sigma^2)$ as $z = \mu + \sigma \times \varepsilon$, given $\varepsilon \sim N(0,1)$
- decode $D[z]$

now, how would our loss look like? the first element would be, as mentioned before, the reconstruction loss. That loss will allow the model to learn its weights such that $VAE(I) \approx I$. what this loss also tells us is that classes remain separable in the latent space - this means, for example, that the encoded version of the digit 4 and 8 would be distinguishable in the latent space (otherwise we wouldn't be able to decode them to different values). This can be understood by looking at the minimization objective - let I_4, I_8 represent images of $4, 8$, if $E[I_4] \approx E[I_8]$, $AE[4] - I_4$ and $AE[8] - I_8$ would be large, and the model would punish accordingly. Aside from the reconstruction loss, there's another term that is introduced to our loss - a term that will provide us with a latent distribution which is close to $N(0,1)$. You may recall that a good loss for measuring distances between distributions is the D_{KL} loss. In our case, that loss should represent the distance between $N(0,1)$ and the generated distribution of the encoder. Enforcing such probabilities means that our latent space would be compact (centered and not too smeared - just like $N(0,1)$). In the following calculations, notice that though the Gaussians in hand are multivariate, the co-variance is zero throughout, therefore one could consider the 1D-Gaussian distribution, and account for every μ_i and σ_i by themselves, and finally sum the results.

If so, lets build our term for the D_{KL} divergence (for continuous parameter) - denoting the input as x and the latent space parameters as z, we define two Gaussian distribution $p_{\theta_D}(z|x_i) = \frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(-\frac{(x-\mu_p)^2}{2\sigma_p^2}\right)$, known as our latent prior (associated with our decoder), and $q_{\theta_E}(z|x_i) = \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(-\frac{(x-\mu_q)^2}{2\sigma_q^2}\right)$, known as our approximated posterior (associated with the encoder), we have

$$D_{KL}(q(z|x_i)||p(x_i|z)) = -\int_z q(z|x_i) \log\left(\frac{p(x_i|z}{q(z|x_i)}\right) dz$$

substituting p, q: (notice the slight abusive notation in which we use x instead of x_i - x and x_i here represents a scalar where before x was a vector - we will get back to the notation of x as a vector at the end, when summing over all x_i's)

$$-D_{KL}(q_\theta(z|x_i) = \int_z \frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(-\frac{(x-\mu_q)^2}{2\sigma_q^2}\right) \log\left(\frac{\frac{1}{\sqrt{2\pi\sigma_p^2}} \exp\left(-\frac{(x-\mu_p)^2}{2\sigma_p^2}\right)}{\frac{1}{\sqrt{2\pi\sigma_q^2}} \exp\left(-\frac{(x-\mu_q)^2}{2\sigma_q^2}\right)}\right) dz$$

the above can be simplified to

$$\frac{1}{\sqrt{2\pi\sigma_q^2}}\int_z \exp\left(-\frac{(x-\mu_q)^2}{2\sigma_q^2}\right)\left(\log\left(\frac{\sigma_p}{\sigma_q}\right)-\frac{(x-\mu_p)^2}{2\sigma_p^2}+\frac{(x-\mu_q)^2}{2\sigma_q^2}\right)dz$$

notice that for a continuous Gaussian distribution, what we have is exactly the expected value under q, that is

$$=\mathbb{E}_q\left(\left(\log\left(\frac{\sigma_p}{\sigma_q}\right)-\frac{(x-\mu_p)^2}{2\sigma_p^2}+\frac{(x-\mu_q)^2}{2\sigma_q^2}\right)\right)$$

from linearity of $\mathbb{E}$, and since $\log\left(\frac{\sigma_p}{\sigma_q}\right)$ is nothing but a number:

$$=\log\left(\frac{\sigma_p}{\sigma_q}\right)-\frac{1}{2\sigma_p^2}\mathbb{E}_q((x-\mu_p)^2)+\frac{1}{2\sigma_q^2}\mathbb{E}_q((x-\mu_q)^2)$$

using definition, $\sigma=\mathbb{E}((x-\mu)^2)$, so

$$=\log\left(\frac{\sigma_p}{\sigma_q}\right)-\frac{1}{2\sigma_p^2}\mathbb{E}_q((x-\mu_p)^2)+\frac{1}{2}$$

we now use a trick:

$$(x-\mu_p)^2=(x-\mu_q+\mu_q-\mu_p)^2=(x-\mu_q)^2+2(x-\mu_q)(\mu_q-\mu_p)+(\mu_q-\mu_p)^2$$

under the expected value $\mathbb{E}_q$ we get

$$-D_{KL}=\log\left(\frac{\sigma_p}{\sigma_q}\right)-\frac{1}{2\sigma_p^2}\mathbb{E}_q((x-\mu_q)^2+2(x-\mu_q)(\mu_q-\mu_p)+(\mu_q-\mu_p)^2)+\frac{1}{2}$$

which can be simplified as

$$=\log\left(\frac{\sigma_p}{\sigma_q}\right)-\frac{\sigma_q^2+(\mu_q-\mu_p)^2}{2\sigma_p^2}+\frac{1}{2}$$

specifically, for $\sigma_p=1$ and $\mu_p=0$ we have

$$-D_{KL}=\frac{1}{2}(1+log(\sigma_q^2)-\sigma_q^2-\mu_q^2)$$

consequently, to account for the reconstruction loss and for all the elements in the multivariate gaussian, our loss for the $i'th$ sample would be

$$L_i(\theta_D,\theta_E)=||D(E(I_i))-I_i||+\sum_{j=1}^{J}\frac{1}{2}(1+log(\sigma_j^2)-\sigma_j^2-\mu_j^2)$$

where J is the dimension of our multivariate Gaussian (the latent space distribution) and i is the $i'th$ sample. notice that L_i is a function of θ_D,θ_E as σ_i,μ_i are function of θ_D,θ_E. finally, our objective would be

$$\theta_D^*,\theta_E^*=arg\min_{(\theta_D,\theta_E)}\sum_i L_i(\theta_D,\theta_E)$$

7.3 Wasserstein Autoencoders (WAE)

(was given as part of GAN, but is more suitable here) Recall that when using VAE, every sample is mapped to some distribution, which also means that every two points map to the same distribution. This causes some problems, as we wouldn't want to associate different values with the same distribution. In WAE on the other hand, we map ALL the points together with our encoder to some distribution. In other words, the distribution of all the points combined defines the latent space distribution, which in turn allows every point to map to different "place" in the latent space.
Like VAEs, two losses are used - one is the reconstruction loss and the other provides us with a target of similar distributions. This can be written as followed

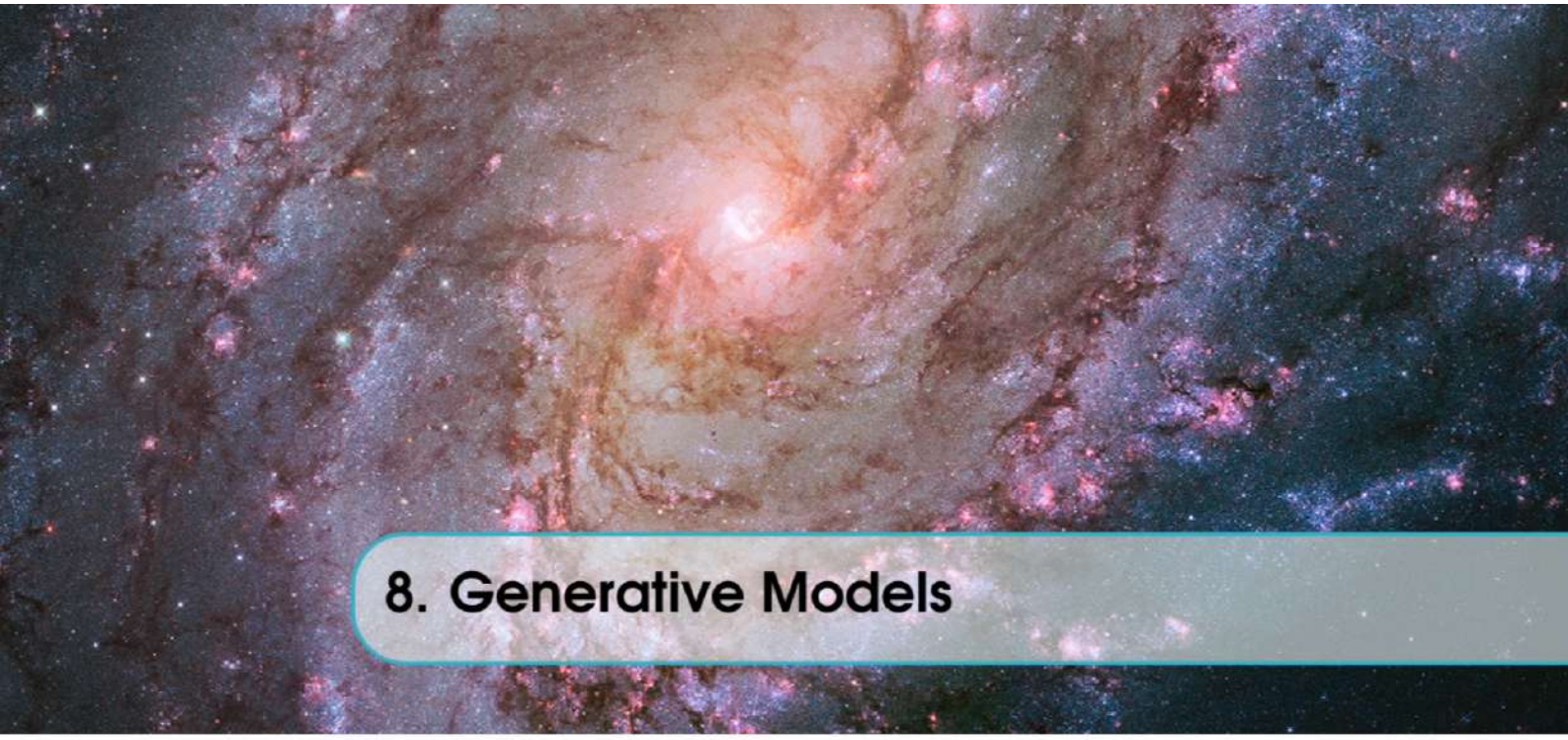
$$min_\theta ||D(E(I)) - I|| + \alpha \sum_k |\int_z \Phi_k(z) P_E(z) dz - \int_z \Phi_k(z) P_{gauss}(z) dz|$$

where $P_{gauss}(z) \propto \exp(-z^2/2)$ and $P_E(z)$ is the probabilities induced by our encoder. The distance between $P_{gauss}(z)$ and $P_E(z)$ is calculated using "Tester" functions Φ_k that are carefully chosen. those represent random variables that were predefined, and for which the expectation of both $P_{gauss}(z)$ and $P_E(z)$ is being calculated. We expect that if $P_{gauss}(z)$ and $P_E(z)$ are similar, the resulted integrals would be the same, which will minimize the objective. Notice though that the integrals may cancel each other even when $P_{gauss}(z)$ and $P_E(z)$ are different, and this is the reason why we use many Φ_k.

the second loss term is equivalent to the following

$$\alpha \sum_k |\sum_i (\Phi(E(I_i)) - \Phi(g_i))|$$

where $g_i \sim N(0,1)$. this is a more practical approach to actually calculate this term.



8. Generative Models

Generative models are models that try and learn the distribution of some sample space, in order to generate new samples as output. given an output distribution G and a data-set X, a good rule of thumb is that these models should be

- Sufficient: $\forall x \in X : \exists z$ s.t $G(z)$=x
- Compact: $\forall z$, $G(z)$ should be a valid sample in X

where G is our generated distribution.
As an introduction, we will first discuss a non parametric method for estimating the PDF of a random variable

8.1 Kernel Density Estimation

let $x_1, ..., x_n$ be n iid samples drawn from an unknown distribution, associated with a density function f. We would like to estimate f's "shape", And an approach to achieve this goal (among many others) would be to define an estimator to f, as followed [`Parzen Windows`]:

$$\hat{f}_h(x) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - x_i) = \frac{1}{nh}\sum_{i=1}^{n} K(\frac{x-x_i}{h})$$

this is for a predefined non negative kernel function K (some window, for example a rectangular one or a triangular one), such that $k_h(x) = \frac{1}{h}K(\frac{x}{h})$, and h is a smoothing parameter called the bandwidth (h should be as small as the data allows). Intuitively, we can think of $\hat{f}$ as the summation of n close neighbors $\{x_i\}_{i=1}^{n}$ to some point x, similar to `K-nearest-neighbors`.
Choosing h is most commonly achieved when optimizing the `MSE` loss, which is (for continuous parameter x):

$$MSE(h) = \mathbb{E}[\int_x (\hat{f}_h(x) - f(x))^2 dx]$$

notice though that even for $\hat{f}_h$ that achieves small MSE error, $\hat{f}$ is not necessarily f, that is, KDE method does not provide us with the actual f.

8.2 Transfer Learning

(may change the order later) How can we use some pre-trained model that classifies 1000 classes, if we want to classify some new class, that isn't part of what the model was trained on?
One option would be to add new layers to our model and then, based on the already trained weights from the previous layers, conduct training iterations on the new layers. Notice that this is not "Fine-Tuning", which is the process of conducting small changes and re-training the entire model, as we only train new layers based on weights provided from the already trained initial model. This can be visualized as followed:
given input I, original layers $\{\ell_i\}_{i=1}^{n}$ (the layers of the original model), and new layers $\{\tilde{\ell}_i\}_{i=1}^{m}$ (the new layers we add):

$$I \to \underbrace{\ell_1 \to ... \to \ell_n}_{\text{freeze these}} \to \underbrace{\tilde{\ell}_1 \to ... \to \tilde{\ell}_m}_{\text{train these}}$$

m may very based on how many more classes we wish to classify, but generally speaking is $< n$. Remember that under the scope of feature extraction, for small i's the layer ℓ_i represent extraction of more generic features, therefore when adding layers deeper in the architecture we account for more specific changes.

R Transfer learning could be used not only to classify more classes, but for various reasons. Today it is very common to add layers on top of well known architectures, such as VGG, to provide it with more task specific results.

8.3 Knowledge Distillation

(may change the order later) Sometimes we do not need/cannot contain in our memory all the power of a very big model ("Teacher" model). It may be sufficient using a more simple model that achieves similar results, and is smaller ("Student" model). To do that, we parse our data through a pre-learned teacher model and a (currently training) student model. those would output what is known as logits (for example, the confidence percentage of both models), and our student model would try and replicate the logits that were the output of the teacher model. In a `soft`-distillation framework, the student would try and output similar logit (replicate the distribution results), and in `hard`-distillation we wish our student model to classify and misclassify exactly like the teacher did.

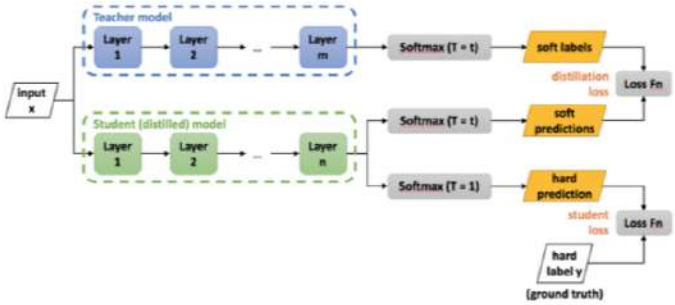


Figure 8.1: Knowledge distillation - the loss is a combination of the distillation loss, which accounts for the loss based on the logits, and the student loss which is based on the ground truth

8.4 Generative Networks

As already mentioned, generative models model the distributions of the individual classes and can, for example, generate novel signals by sampling that distribution. Such task immediately provides two difficulties

- Deriving the distribution: Traditionally, one may pick a family of distributions P_θ and find the most probable one using MLE principle $\max_\theta \prod_i P_\theta(x_i)$. As the P_θ's represent probabilities, they should be normalized, i.e $\int_x P_\theta(x)dx = 1$. This is where a problem may arise - as P_θ can represent a distribution of enormous classes (all images of some type for example), finding such function with the property or normalization is a difficult, or rather impossible, task.
- Sampling from a distribution: as we cannot derive the distribution's form, we cannot sample from it. Also, when we do have the distribution, how does the process of sampling from it should be implemented? Having said that, there are few distribution that we can, in fact, sample from (see remark)

R [Sampling Distribution]: What are some distributions we actually can sample from?

- Uniform: the main assumption is that we (aka our computer) can sample from uniform distribution $x \sim U([0,1])$. This can be done, for example, using a proprietary software that enhances hardware randomness, which is in some cases Uniform. Under this assumption, we can also sample from multi dimensional uniform distribution $P(x,y,z) = U(x)U(y)U(z)$, as those are just independent samples drawn from a one dimensional Uniform distribution
- Gaussian: using the central limit theorem, starting from some initial distribution, its sum is a random variable that approaches normal distribution. (notice this is also the case for multi-dimensional Gaussian where the covariance matrix is diagonal)
- any general 1D distribution: since $F(x) = \int_{-\infty}^{\infty} p(s)ds \iff F'(x) = p(x)$, the probability that $F^{-1}(u)$ is in some interval $\pm\varepsilon$ is

$$P(z < F^{-1}(u) < z+\varepsilon) = P(F(z) < u < F(z+\varepsilon)) \approx P(F(z) < u < F(z) + \varepsilon F'(z)) =$$
$$P(F(z) < u < F(z) + \varepsilon p(z)) \overbrace{=}^{\varepsilon \to 0} \varepsilon p(z)$$

this means that if we wish to sample from p we can perform the following
  - find $F(z)$
  - find $F^{-1}(z)$
  - take some $u \sim U([0,1])$
  - the sample $F^{-1}(u)$ represents sampling from p

8.5 Modeling by change of variable

We start with some z that is distributed from a simple, known, distribution $z \sim p_z(z)$ (for example $z \sim N(0,I)$), and define a change of variable - a function with parameters θ that will be learned $M_\theta : \mathbb{R}^n \to \mathbb{R}^n$, $M_\theta(z) = x$, such that $x \sim p_x(x)$, and p_x is to be found. $p_x(x)$ is given by

$$p_x(x) = |\det J_x[M_\theta^{-1}(x)]| p_z(M_\theta^{-1}(x))$$

where the first multiplier is the density expansion/contraction term (J_x is the Jacobian with respect to x). So we can generally say that our paradigm is to start from some simple probability function, and then, in an iterative learning process, find the mapping between our simple distribution to the actual distribution we wish to learn. In future section we will describe the GLOW model, that finds $p_x(x)$ specifically. Prior to GLOW, we will describe the famous GAN model, that does not provide p_x, but does provide a method of sampling novel samples from p_x.

8.6 Generative Adversarial Networks

Consider the following iterative process: we are given with R - our real data, G, some generator that provides samples from some naive distribution, I our input to the generator and D - a discriminator network which learns to compare R with $G(I)$. G would be trained to mislead the Detective D (if D fails, and G does a good job, the data $G(I)$ resembles R).
lets formalize this process by defining our models mathematically:

- Generator G: receives input $z \sim p_z(z)$ and outputs some $G(z) = x \sim p_g(x)$. The distribution $p_z(z)$ is a very simple one, and p_g is the distribution that G generated. G would try, to the best of his capabilities, to bring p_g as close to p_{data} as it can.
- Discriminator D: classifies between $x \sim p_g$ and $x \sim p_{data}$. D outputs a scalar that is 0 if D "thinks" that the input was provided from G_θ, and 1 if it thinks that the input was provided from the original data $x \sim p_{data}$

The training process is done simultaneously (both on D and G), and the loss function is

$$\min_G \max_D V(D,G) = \min_G \max_D \mathbb{E}_{p_{data}}[log(D(x))] + \mathbb{E}_{p_z}[log(1 - D(G(z))]$$

the term $\max_D(...)$ is made up of two segments - the first one involves maximizing $\mathbb{E}_{p_{data}}[log(D(x))]$, that is, given samples from our original data $x \sim p_{data}$, we max $log(D(x))$. notice that this value approaches maximal value (0) when $D(x) \to 1$, so D learns to return 1 for those samples. the second item is $\mathbb{E}_{p_z}[log(1 - D(G(z))]$, that is, given samples from our naive distribution $z \sim p_z$, we want to maximize $log(1 - D(G(z))$. also notice that this value approaches maximal value 0 when $D(G(z)) \to 0$. Simply put, that max term tells us that D wants to classify "good" examples as 1 and "bad" examples (that were forged by G) as 0. The outer term $\min_G$ tries to make D's life harder - as G outputs data that is more and more similar to the original data, the less D would be able to distinguish between the two. In other words, G works in reverse order with respect to D, and instead of maximizing the term, tries to minimize it.

The discriminator maximizes the binary cross entropy loss:

$$BCE\ Loss(y, D(x)) = \sum_i y_i log(D(x_i)) + (1 - y_i) log(1 - D(x_i))$$

the sum over i takes both the samples from p_{data} and from p_z, so we can write

$$= \sum_{x_{data}} 1 \cdot log(D(x_{data})) + 0 \cdot log(1 - D(x_{data})) + \sum_{x_g} 0 \cdot log(D(x_g)) + 1 \cdot log(1 - D(x_g))$$

which gives us what we were working with initially

$$\sum_{x_{data}} log(D(x_{data})) + \sum_{x_g} log(1 - D(x_i))$$

given this Framework, it is interesting to see that (at least in the theoretical realm), p_g eventually does equal to p_{data}, or in other words - our generator is able to produce the true distribution. the following provides a theoretical analysis assuming infinite capacity networks and convergence to global optimal

Corollary 8.6.1 Assuming G is fixed, the optimal D obeys $D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x) + p_g(x)}$

"proof". working with continuous parameters:

$$V(G,D) = \int_x p_{data}(x) log(D(x)) dx + \int_z p_z(z) log(1 - D(G(z))) dz$$

as $G(z)$ outputs some $x \sim p_g(x)$, we can combine those two integrals

$$= \int_x p_{data}(x)log(D(x)) + p_g(x)log(1 - D(x))dx$$

an important concept here is that as G has infinite capacity, therefore the resulted output of $G(x)$ can be of any form, and is independent for every x (for a bounded capacity G in terms of its inner parameters, its learned parameters may be based on 100 samples, but the results for the 101'th example is, to some extent, a results of the finite number or parameters, and isn't independent). This means that minimizing the integral is the same as minimizing the intergant for every x, which means that we are now facing a much simpler problem: minimize the following 1D problem

$$a\log(y) + b\log(1 - y)$$

with respect to y. This gives us $y = \frac{a}{a+b}$, or in our case, $D^*_G(x) = \frac{p_{data}(x)}{p_{data}(x)+p_g(x)}$ ■

Corollary 8.6.2 assuming $D = D^*_G$, the min-max problem is solved for $p_g = p_{data}$

"proof". substituting D^*_G to $\max_D V(G,D)$, we have

$$V(G,D^*_G) = \max_D \mathbb{E}_{p_{data}}[log(\frac{p_{data}(x)}{p_{data}(x)+p_g(x)})] + \mathbb{E}_{p_z}[log(1 - \frac{p_{data}(G(z))}{p_{data}(G(z))+p_g(G(z))})]$$

which is the same as

$$V(G,D^*_G) = \max_D \mathbb{E}_{p_{data}}[log(\frac{p_{data}(x)}{p_{data}(x)+p_g(x)})] + \mathbb{E}_{p_g}[log(\frac{p_g(x)}{p_{data}(x)+p_g(x)})]$$

lets focus on the right-most term:

$$\mathbb{E}_{p_g}[log(\frac{p_g(x)}{p_{data}(x)+p_g(x)})] = \int_x p_g(g)\log(\frac{p_g(x)}{p_{data}(x)+p_g(x)})dx$$

this can be re-written as

$$\begin{aligned} &= \int_x p_g(x)\log(\frac{p_g(x)}{(p_{data}(x)+p_g(x))/2}) - \log(2)dx \\ &= D_{KL}(p_g||(p_{data}+p_g)/2) - \log(2) \end{aligned}$$

the same can be said for the first term, giving us

$$\max_D V(G,D) = D_{KL}(p_g||(p_{data}+p_g)/2) + D_{KL}(p_{data}||(p_{data}+p_g)/2) - \log(4)$$

the sum of two such Kullback-Leiber divergences is called the `Jensen-Shannon Divergence` (which is symmetric, unlike D_{KL})

$$JSD(P||Q) = \tfrac{1}{2}D_{KL}(P||M) + \tfrac{1}{2}D_{KL}(Q||M)$$

which gives us

$$\max_D V(G,D) = 2JSD(p_g||p_data) - \log(4)$$

this is very convenient, as the *JSD* function is bounded by zero, and zero is achieved when $p_g = p_{data}$ (this could have been stated without introducing the JS divergence, as the KL divergence achieves zero under the same condition). In other words:

$$\min_G \max_D V(G,D) = \min_G 2JSD(p_g||p_{data}) - \log(4)$$

which is obtained for $p_g = p_{data}$. (This also means that if $p_g = p_{data}$, $D^*_G = \frac{p_g}{p_g+p_g} = \frac{1}{2}$, for every x) ■

R Unrelated to the theoretical part, it is worthwhile stating that in most cases, our z's are sampled from a distribution from some small dimension, and is then up-sampled to an appropriate dimension (for example, the dimension of an image). Following this, the discriminator will do its job in the appropriate dimension. This is in contrast to what we have initially stated (that we start and end in the same dimension).
Upsampling is usually done using the "Transpose convolution" operation. lets see a quick example:

$$input = kernel = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}$$

it is common to use $stride = 2$ (usually the stride is 1 in regular conv operations.), as to not give more weight to specific pixels that overlap. denoting the input as I, the kernel as K and the operator as $C^T(...)$, we have

$$C^T(I,K) = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 4 & 6 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 \\ 0 & 0 & 6 & 9 \end{pmatrix} =$$
$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 3 \\ 0 & 2 & 0 & 3 \\ 4 & 6 & 6 & 9 \end{pmatrix}$$

note that in the framework of NN, k is a learned kernel, just like in regular convolution.

8.6.1 Interpolation in "Z-space"

recall that our generator G mostly works on z_i that are from a trivial distribution. One characteristic of such distributions is that they represent a convex set. If so, for two samples z_1, z_2, the line that connects them should be contained in that set (by definition of convex sets), so we can ask what would be the output of $G(\alpha z_1 + (1-\alpha)z_2)$. for varying $\alpha \in [0,1]$, this would give us the transition from one image I_1, mapped using $G(z_1)$ to another image I_2, mapped using $G(z_2)$. This is known as interpolating the z-space, and does provide interesting results, as can be seen in the figure below

8.6.2 Multi-Modal Interpolation

diverging a bit to problems of reconstruction, consider your data has some inherent degradation (for example - noised audio, or low resolution images) and our goal is to reconstruct that data. In some of those cases, the solution isn't unique (think for example of eyes dataset where to colors are missing), namely, the reconstruction could output various results that are all somewhat plausible, therefore we associate some distribution to the output (say, blue eyes are rare, and brown are more common). With this framework in mind, using regression trained to minimize ℓ_2 will predict the average results (some color between brown and blue), but a GAN would generate outputs that represent all plausible options, as it mimics the underlying distribution.

8.7 The pitfalls of GAN

the usage of both a discriminator and a generator results in many difficulties. lets see some of the more known ones

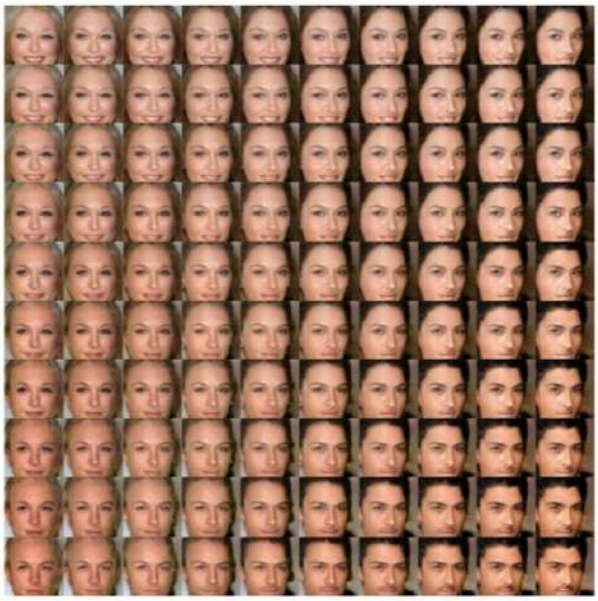


Figure 8.2: Interpolation obtained from the CelebA data-set, where only the images in the 4 corners are real (from Nvidia Devblogs). notice that every entry in the interpolation matrix represents a reasonable face (i.e every entry is a valid sample), and that is due to the fact that the generator successfully learns the entire probability space

8.7.1 Discriminator Saturation

consider the case that G wasn't trained already (or was for only a short time), therefore its results are very bad. as D performs its training for increasingly large iterations, it would "beat" G with no effort, and its output classification would be closer and closer to the binary values $\{0,1\}$ (saturation). this means that when its G's "turn" to train, as G is based on the values of D, the derivatives of the loss would tend to zero (as the loss has both the term $\log(D(x))$ and $\log(1-D(G(z)))$, whose sum, if D classifies correctly, approaches 0). In other words, training D for too many iteration would cause G to suffer from `Vanishing Gradients`.
This can also be understood from the following - the last layer in D is usually a linear one, followed by a softmax. therefore, as D wants to maximize the term $\log(1-D(G(z))$, it will tend to provide values that approach $-\infty$ (which will cause the log to explode). i.e. $D_{linear}(G(z)) \to -\infty$, and $softmax(D_{linear}(G(z)) \to 0$ therefore the derivative tends to 0 as well.
one way to solve this problem is using a different loss function, that doesn't saturate. the loss is of the form

$$\max_D \left(-\mathbb{E}_{p_{data}}(D_{linear}(x)-1)^2 - \mathbb{E}_{p_z}(D_{linear}(G(z))-0)^2\right)$$

such loss function enhances the fact that D should output the value $0,1$, namely, D providing values approaching $-\infty$ isn't a satisfactory condition no more. in other words, if for the initial loss D could have achieved good loss by providing large values (or small values), it now needs to learn in an iterative process how to return a binary result (while still remaining a continuous function). we hope that as D learns to return such values, G would train enough to be non-trivial (with respect to D)

8.7.2 Mode Collapse

It may be reasonable to think that changing our minimization problem from $\min_G \max_D V(D,G)$ to $\max_D \min_G V(D,G)$ would give G the chance to train "first", achieving some non trivial output, and only then maximize D for that G.
Here is the problem though - in this scenario, G only focuses on some minima of D, and will generate its examples based on such minima. As G starts producing the same output over and over again, D's best strategy is to simply reject those outputs. from here, if G would find itself stuck in some local minima, both D and G would rotate through a small set of output types, and wouldn't be able to provide results that may achieve the actual minima, or in fact wouldn't be able to provide various different results (for example, G would only generate some subset of faces)
an example of Mode collapse was given in the paper `What GAN Cannot See`, in which it was demonstrated that G has inherent expressibility limitation, in the sense that it cannot reconstruct in full images that it has already encountered
the following process provides an outline to how, given some image x^*, one can look for z such that $G(z) = x^*$

- step 0: we assume there exists some generator G, and our data is $G(z)$, for various zs
- step I: we train an encoder E that given an image $G(z)$, a recreation of z is being made, i.e $E[G(z)] \to z$. we can think of E as a model that is trained to be G^{-1}
- step II: given some image x^*, we create $z_0 = E[x]$. we think of this z_0 as the best z from which G will generate the image x (i.e $G(z_0) \approx x^*$)
- step III: we construct a training process, where our loss can be written as $\min_z ||G(z) - x^*||$, where z was initially defined as $z = z_0$. (to some extent, the previous steps provided us only with the initialization of $z = z_0$, which is helpful as a non random starting point)

the paper shows that the reconstructed image weren't very similar, even when considering x^* as an image from our training set (G already seen x^*, and was trained based on it). This may be logical, as the GAN model was never required to reproduce the images from the training set, only generate new ones. In terms of mode collapse, we can think of G as a model that can extract **some** image x from the data's distribution, but cannot necessarily generate **every** image x from that distribution

8.8 Unpaired Image-To-Image Translation (Cycle-GAN)

(This section provides a high level description of a specific GAN network)
GANs can also be used to convert one input to another. As an example, we can take an image of zebras and convert them to images of horses, while preserving the orientation and outline of the original image. How would such GAN work? we use 2 GAN models, G and F. G would "take" us from an image of a zebra (denoted X) to an image of a horse (denoted Y), and F vice versa. for every sample space (zebras/horses) we associate a discriminator D_X and D_Y that is trained to distinguish between original zebras and constructed zebras (or original horses and constructed ones). When constructing the loss of G, for example, we should take into consideration two factors - the first one is $D_Y(G(X))$, which is the loss that accounts for how good can D distinguish the output of the GAN (as usual), and the second one would be $||F(G(X)) - X||$, which accounts for minimizing the difference between the original input (zebra) to the zebra that was reconstructed from the horse (that was generated from the original zebra). those would be summed, preferably with some hyper parameter α that governs the weight each loss should have:

$$Loss[G] = D_Y(G(X)) + \alpha||F(G(X)) - X||$$

$$Loss[F] = D_X(F(Y)) + \alpha||G(F(Y)) - Y||$$

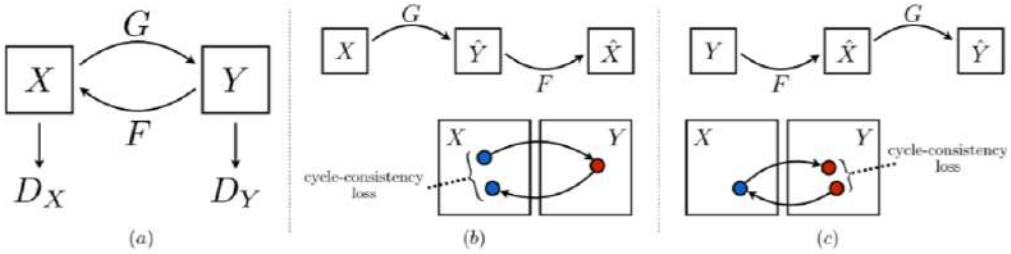


Figure 8.3: Cycle GAN diagram

8.9 Generative Model Evaluation

It is sometimes unclear how can we evaluate the quality of our GAN models. Taking reconstructed images of dogs for example, how could we say something about "how dog, is our dog"?. In a broader sense, it may be difficult to provide concrete conclusions based merely on our own eyes. This has resulted in defining some metrics, that should provide a more mathematical way to describe GAN performances.

8.9.1 Inception Score

The Inception Score (sometimes IS), is an objective metric for evaluating the quality of generated images, specifically synthetic images output by generative adversarial network models. In order to calculate the score, a large number of generated images are classified using some strong classifier model (more specifically, the induced probability space is being constructed based on classifying large quantities of photos), and those are compared to the original distribution using KL-divergence. more formally, we define the IS as

$$IS(G) = \exp(\mathbb{E}_{x\sim p_g} D_{KL}(p(y|x)||p(y)))$$

where $p(y|x)$ is the conditional probability function provided from our classifier, $p(y)$ is the actual distribution, and $x \sim p_g$ represents that we sample from the generative model output probability space.

Though very popular, Inception score has some problems. one such problem is the fact that a good score requires having a good distribution of generated images across the possible objects supported by the classifier, but this is hard to control as GAN doesn't necessarily provide that feature (in most cases, we can only generate new images but not control how frequent they are).

8.9.2 Frechet Inception Distance (FID)

In this method we measure distances between two Gaussians that were fitted from extracted features that were obtained from our original and our fake data.

- compute a set of features ($\sim 10k$ real and fake images)
- fit a multivariate Gaussian for the real data and the fake data. denoted r and g

- measure the Frechet Inception Distance:
  - for multidimensional Gaussian: $FID = ||\mu_r - \mu_g||^2 + Tr(\Sigma_r + \Sigma_g - 2\sqrt{\Sigma_r^{1/2}\Sigma_g\Sigma_r^{1/2}})$
  - for one dimensional Gaussian $FID = ||\mu_r - \mu_g||^2 - (\sigma_r - \sigma_g)^2$

in other words, we compare the first and second moment between two Gaussians that represent an approximated distribution of our original data and our fake data

8.10 Conditional GAN generator (cGAN)

We've mentioned that we cannot (as of now) determine from which class our GAN output would be, but what if we did want that feature? In other words, our goal is that our generator output would be of some chosen class. In theory, we want to create a dependency between some wanted class and our loss function, but it turns out that this is not necessary. One approach of doing so is providing our generator with a class vector (for example, one hot encoding of the classes, and then using e_i for some class i), along side the vector provided from our discriminator. the same trick would be applied to our discriminator - we will provide it with the original image as well as the class vector. by concatenating the original input with some class vector we train our network to output only outputs of some specific class.

R the class representation can be generated using any reasonable embedding, and not necessarily one hot encoding

8.11 GLOW: generative Flow

The problems that were shown in the previous section regarding GAN provided some motivation to describe generative models that does not use a `min-max` loss approach. Lets describe some of those methods

recall our initial premise - given some probability function P_z, we would like to transition to another probability space, associated with the probability function P_θ. The first thing we did was defining a method of transition between those two function, which was

$$P_x(x) = |det(J_x(M_\theta^{-1}(x))|P_z(M_\theta^{-1}(x))$$

this was when z is sampled from some easy distribution (for example $z \sim N(0,I)$), $M_\theta(z) = x$ was our change of variable function (with trained parameters θ), and $x \sim P_x(x)$ our target distribution. In the following, we assume that M_θ is a function that represents some multi-layered network, therefore can be written as the composition of k functions (for some k)

$$M_\theta = h_k \circ h_{k-1} \circ ... \circ h_1$$

Now, the Jacobian of a composition is the product of the Jacobian on every composed function. as we can think of our function as matrices, we are left with a determinant of some matrix product, which is the same as the product of matrix determinants. Lets combine all of those while also applying log on both sides of the formula (also note were using z instead of $M^{-1}(x)$, and the abusive notation $J_x(z) = \frac{dz}{dx}$):

by the inverse function theorem:

$$P_x(x) = |\det(\tfrac{M_\theta(x)}{dz})^{-1}|P_z(M_\theta^{-1}(x)) = |\det(\tfrac{M_\theta(x)}{dz})^{-1}|P_z(z)$$

as M is invertible, we can also use the fact that $\det(M^{-1}) = \det(M)^{-1}$, which gives

$$P_x(x) = |\det(\frac{M_\theta(x)}{dz})|^{-1} P_z(z)$$

applying log on both sides:

$$\log p_x(x) = \log |\det(\frac{M_\theta(x)}{dz})|^{-1} P_z(z)$$
$$\log p_x(x) = \log P_z(z) - \log |\det(\frac{M_\theta(x)}{dz})|$$

now we can write $\frac{M_\theta(x)}{dz}$ as a product of Jacobians, and use $det(\prod_i x_i) = \prod_i det(x_i)$:

$$= \log P_z(z) - \log \prod_k \det(\frac{dh_{k-1}}{dh_k})$$
$$= \log P_z(z) - \sum_k \log \det(\frac{dh_{k-1}}{dh_k})$$

The above has given us a way to apply the maximum likelihood principle, as p_x represents our learned target distribution (and can be denoted as p_θ):

$$\log \prod_i P_\theta(x_i) = \sum_i \sum_k \log |\det(dh_{k-1}/dh_k)| + \log p(M^{-1}(x_i))$$

note that M must be analytically invertible. this means, for example, that we cannot use `relu` activations in our architecture

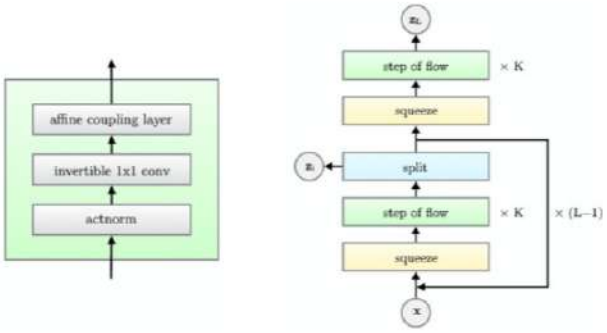
But nothing comes without sacrifices, which in our case is embedded into the calculation of the determinant. When using basic Laplacian expansion, this is an $O(n!)$ operation, this is obviously no good. Another approach uses lower-upper (LU) decomposition, which converts a square matrix A into two triangular matrices L, U, such that $A = LU$, L is lower triangular and U is upper triangular. This reconstruction costs $O(n^3)$, but the determinant of L and U takes $O(n)$ to calculate. For such matrices our loss can be simplified even further, as

$$\log |\det(dh_{k-1}/dh_k)| = \sum_j (\log |(dh_{k-1}/dh_k)_{jj}|)$$

In the original paper, new layers were defined, and those represented triangular matrices. More specifically, they have constructed a network that is a concatenation of blocks, where in each block there were three main components

- `actnorm`: point-wise multiplication of the input pixel with some learned scalar s which has dimensions $1 \times c$, where c is the number of channels in the input, and an addition of bias b (this resembles BatchNorm)
- `invertible` 1×1 `conv`: this layer is a convolution layer that only takes into consideration the channels (doesn't "mix" adjacent pixels). Furthermore, the output channels is the same as the input channel (a necessary condition for invertibility). such structure means that we if we use c channels, the conv operator can be represented using a $c \times c$ matrix, for which we can invert and calculate the determinant on.
- `affine coupling layer`: we split our input x to x_a, x_b. x_b is passed through a general NN, and the (two) outputs of the network are *logs* and t (*logs* and not s as we will take the exponent later on) finally we stack our output y as an act-norm of x_a with x_b. This layer outputs y's that are a function of subsection of the input (say, only even pixeled xs), and the stacking provides coupling between the two subsets of the splitting of x.

The specific formulation of each layer in the network is provided in the following figure



(a) One step of our flow. (b) Multi-scale architecture (Dinh et al., 2016).

Figure 8.4: (left) is the architecture of each block. (right) is the flow combined with a multi scale architecture (each green block represents the entire block on the left)

Description	Function	Reverse Function	Log-determinant
Actnorm. See Section 3.1.	$\forall i,j : \mathbf{y}_{i,j} = \mathbf{s} \odot \mathbf{x}_{i,j} + \mathbf{b}$	$\forall i,j : \mathbf{x}_{i,j} = (\mathbf{y}_{i,j} - \mathbf{b})/\mathbf{s}$	$h \cdot w \cdot \mathtt{sum}(\log \lvert\mathbf{s}\rvert)$
Invertible 1×1 convolution. $\mathbf{W} : [c \times c]$. See Section 3.2.	$\forall i,j : \mathbf{y}_{i,j} = \mathbf{W}\mathbf{x}_{i,j}$	$\forall i,j : \mathbf{x}_{i,j} = \mathbf{W}^{-1}\mathbf{y}_{i,j}$	$h \cdot w \cdot \log \lvert\det(\mathbf{W})\rvert$ or $h \cdot w \cdot \mathtt{sum}(\log \lvert\mathbf{s}\rvert)$ (see eq. (10))
Affine coupling layer. See Section 3.3 and (Dinh et al., 2014)	$\mathbf{x}_a, \mathbf{x}_b = \mathtt{split}(\mathbf{x})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{x}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{y}_a = \mathbf{s} \odot \mathbf{x}_a + \mathbf{t}$ $\mathbf{y}_b = \mathbf{x}_b$ $\mathbf{y} = \mathtt{concat}(\mathbf{y}_a, \mathbf{y}_b)$	$\mathbf{y}_a, \mathbf{y}_b = \mathtt{split}(\mathbf{y})$ $(\log \mathbf{s}, \mathbf{t}) = \mathtt{NN}(\mathbf{y}_b)$ $\mathbf{s} = \exp(\log \mathbf{s})$ $\mathbf{x}_a = (\mathbf{y}_a - \mathbf{t})/\mathbf{s}$ $\mathbf{x}_b = \mathbf{y}_b$ $\mathbf{x} = \mathtt{concat}(\mathbf{x}_a, \mathbf{x}_b)$	$\mathtt{sum}(\log(\lvert\mathbf{s}\rvert))$

Figure 8.5: Mathematical expressions for each layer

8.12 Generative Latent Optimization (GLO)

GLO is yet another variant of generative learning (that is not GAN), that can be described in one sentence as an Encoder-less auto-encoder. The main premise is that instead of using an encoder E which will provide us with latent codes z_i, we initialize learnable vectors z_i (that are not a product of some Encoder) and minimize $||G(z_i) - x_i||$ over G's weights AND z_i. As z_i are initialized to our desire, this is equivalent to the process of training an auto-encoder with unbounded encoding capabilities (in fact, the latent space dimension is the only restriction we have on our latent codes). In the original paper, the initialization of z_i was from a simple Gaussian distribution where there has been an assumption that z_i changed only slightly in the learning process (this was justified using smaller learning rates for the z_is). Under those assumptions, sampling the slightly altered z_i was a matter of sampling from a slightly deformed Gaussian distribution.

8.13 Implicit Maximum-Likelihood Estimation (IMLE)

We might as well provide another variant of generative learning, as if we haven't seen enough! IMLE is a non parametric model (using Parzen windows that was introduced at the beginning of the chapter), that is used to create a generator in the following manner

- we sample $z_i \sim N(0, I)$
- Given the sampled z_i's, we perform nearest-neighbour optimization: given some initialized mapper M (our soon to be generator), we find some j_i such that $j_i = argmin_{(i,j_i)}||M(z_{j_i}) - x_i||$. j can be thought of as an index of some group of neighbours, and i is the $i'th$ sample (in the $j'th$ group).
- once the minimzer j_i was found, we perform regular SGD minimization $min_{\theta_M}||M(z_{j_i}) - x_i||$
- now we repeat the process from the beginning

such method is very costly, mostly due to the second step - in high dimensions the minimization of $||M(z_{i_j}) - x_i||$ under some metric $||\cdot||$ takes many calculations.

[GLANN] A method to improving the results of IMLE is to combine it with the GLO method from the previous section. The process would be to reduce our input x dimension using GLO and then use IMLE (in the smaller dimension) given z_i that are sampled from a simple Gaussian distribution. This solves the costly metrics calculation in the higher dimensions, and is known as Generative Latent Nearest Neighbors (GLANN).

8.14 Restoring signals using Bayes rule

In the opening section of this chapter, we shortly described that generative networks can also help when our goal is restoring signals. For that we introduced the Bayes rule:

$$P(I|I_c) = \frac{P(I_c|I)P(I)}{P(I_c)} \propto P(I_c|I)P(I)$$

We call $P(I|I_c)$ the posterior, $P(I_c|I)$ the likelihood and $P(I)$ the prior (how plausible is I). our goal is to maximize the posterior for all I given I_c, as we are looking for the I that is most probable given I_c. this translates to the following maximization problem

$$argmax_I \log P(I|I_c) = argmax_I \log P(I_c|I) + \log P(I)$$

notice we discarded the $-\log P(I_c)$ as this is just some constant

8.14.1 The Likelihood

The likelihood tells us how probable some noised image I_C, if we have initially started with I. This term is usually easy to model, as the following examples show:

- De-noising: assume $I_c = I + \eta$, where $\eta \sim N(0, \sigma^2)$. In this case, our likelihood is exactly the probability that given I, a given corruption would be made on it, which is the same as sampling $\eta = I_c - I$. in other words, as η is sampled from a Normal distribution, we have

$$likelihood = P(I_c|I) \propto \exp(-\eta^2/2\sigma^2) = \exp(-(I_c - I)^2/2\sigma^2)$$

- De-blurring + noise: consider $I_c = k * I + \eta$, where $*$ is the convolution operator. The same steps would be taken, to achieve

$$likelihood = P(I_c|I) \propto \exp(-\eta^2/2\sigma^2) = \exp(-(I_c - k * I)^2/2\sigma^2)$$

8.14.2 The Prior

The prior $P(I)$ tells how probable is some input I, and this is where the generative models comes to place. We will describe a method that was introduced in the paper `Deep Image Prior (DIP)`, for which some change of notation would be made: the likelihood would be $\log P(I_c|I) \mapsto E(x, x_0)$ and the prior $\log P(I) \mapsto R(x)$. under those new notations, we describe our minimization problem as

$$x^* = argmin_x E(x, x_0) + R(x)$$

From here we assume that x^* can be achieved using a Generator `CNN` that takes as input a randomized yet constant latent vector z and will be denoted as $f_{\theta^*}(z)$. In other words, we state that $R(x)$ is centered around $f_{\theta^*}(z)$, which in turn tells us that it can be written as $R(x) = \delta(x - f_\theta(z))$, as the probability to find x that is not some output of $f_\theta(z)$ is strictly 0. We can write x only in terms of $f_\theta(z)$, and our minimization problem would be

$$\theta^* = argmin_\theta E(f_\theta(z), x_0)$$

In the original paper, they have plotted an MSE loss vs iterations for the following images $\{image, image + noise, shuffle(image), Uniform([0,1])\}$ (see figure below) and showed that the prior is minimized much faster for clean images, compared to corrupted ones. This means that Generator CNN can easily describe clean images using small number of iterations, but cannot describe noisier images (at least, it would take more iterations). This is an associated bias that such CNN network inherently posses. Using this knowledge, we can start the training process and stop it before reaching large number of iterations. this in turn will provide us with a clean image, as the parts that were clean were more easily constructed, compared to the noisy parts.

R a drawbacks of that method is that the optimization is made for every image

8.15 Plug-and-Play Priors, Regularization by Denoising (RED)

For this paper, we would have to change our notations once more - starting with $\hat{x} = argmin_x - \log(p(y|x)) - \log(p(x))$ as our initial objective, we write instead $\hat{x} = argmin_x \ell(y, x) + s(x)$. This means that $\ell(y, x)$ denotes the likelihood of getting y - the corrupted version given x - the original image, and $s(x)$ is the prior. The next step is to add an auxiliary variable that does not change the objective: $\hat{x}, \hat{v} = argmin_{x,v} \ell(y, x) + \beta s(v)$, subject to $x = v$. The last and final step would be to add a regularization term, and yet another auxiliary term

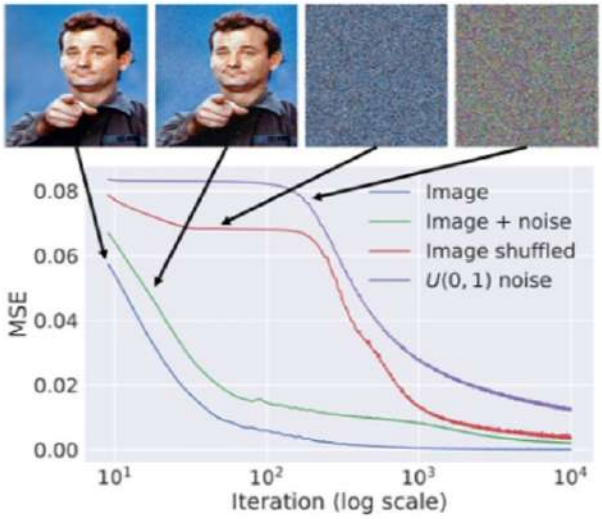


Figure 8.6: MSE loss plots

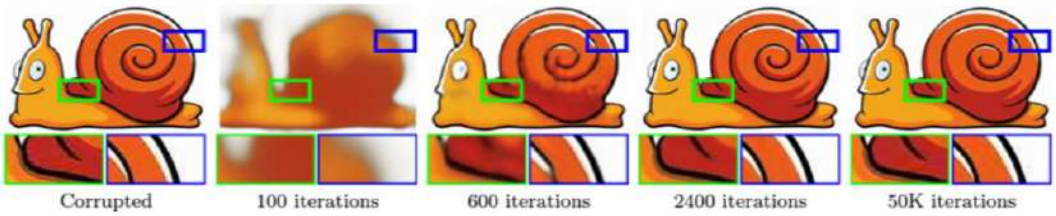


Figure 8.7: for large number of iterations, the cleaned noise returns to the image (slightly visible when comparing $2.4k$ to $50k$

$$L_\lambda(x,v,u) = \ell(y,x) + \beta s(v) + \frac{\lambda}{2}||x - v + u||_2^2 - \frac{\lambda}{2}||u||_2^2$$

Where in total - $\ell(x,y)$ is the likelihood, $s(v)$ is the prior, x is the input, y is the deformed input (the noised version of x for example), u,v are new auxiliary terms and λ, β are new hyper parameters.

R The process that was described here is known as ADMM, which in simple words is a method to break the initial objective into smaller pieces, all of which are a slightly easier version of the initial objective. The new objective should in theory converges better, and can be solved in an iterative manner.

The solution to the problem $argmin L_\lambda(x,v,u)$ is done using the following steps

- $\hat{x} = argmin_x \ell(y,x) + \frac{\lambda}{2}||x - \tilde{x}||_2^2$
- $\hat{v} = argmin_v \frac{\lambda}{2}||\tilde{v} - v||_2^2 + \beta s(v)$
- $u = u + (\hat{x} - \hat{v})$

where in the first term, $\tilde{x} = v - u$ and in the second term $\tilde{v} = x + u$.
Looking at the second minimization problem (for $\hat{v}$), the first term ($\beta s(v)$) is the prior, and we can think of the first term $\frac{\lambda}{2}||\tilde{v} - v||_2^2$ as our likelihood . recall that when we discussed variations of likelihood terms, such term (squared differences between the input and its corrupted version) was attributed to the problem of denoising an image. In other words, the solution of $\hat{v}$ is in itself, a denoising problem.

8.16 Deep Generative Priors

Deep Generative Priors is a method of image restorations that assumes a linear degradation was made on the image. In other words, the corrupted image y is the result of some matrix A that was performed on the input x, so $y = Ax$ (Note that A could also be considered an operator, and for that we denote $y = A(x)$ instead). Now, given a Generator G, recall that one assumption we had was that for every input x there exists some z that was sampled from a basic distribution such that $G(z) = x$. We will use this assumption when building the loss function - our goal would be to find z such that $A(G(z))$ would be as close to the corrupted image y as possible. This is because given such z, $G(z)$ will represent the clean image x. If so, we can write

$$L_{obj} = ||y - A(G(z))||$$

another term we need to consider in our loss is the GAN loss, which is always necessary when we wish G would output sample from our data's distribution. This term should be familiar, except the fact that we use $G(z)$ instead of x, as x is not given:

$$L_{gan} = \log(1 - D(G(z))) - \log(D(G(z)))$$

One last element we add to the objective signifies that we wish to ensure that the z we choose are plausible, namely that they are given from the simple probability space we initialized. such term will be mathematically defined as

$$L_z = \log(p_z(z))$$

the last two terms are added as a regularization term, so in total our minimization problem is

$$\hat{z} = argmin_z[L_{obj}(z) + \lambda(L_{gan}(z) + L_z(z))]$$

R note that A should be known, or at least easily extracted, as it denotes our likelihood term

8.17 wGAN

wGAN is yet another GAN model that changes the loss function, in order to avoid mode collapse and vanishing gradients. before we explain that model, some preliminaries

8.17.1 Earth mover's distance (EMD)

EMD (also known as the Wasserstein metric) is a measure of the distance between two probability distributions over some region D. Intuitively, we can think of those two distributions as two different ways of piling dirt (=earth) in the region D, and the EMD is the minimum cost of turning one pile into the other (where the "cost" is the amount of dirt moved $\times$ the distance by which it is moved). this intuition can be then formulated as follows

Definition 8.17.1 — Earth mover's distance. given two probability distributions P and Q, the EMD is defined as

$$EMD(P,Q) = \inf_{\gamma \in \prod(P,Q)} \mathbb{E}_{(x,y)\sim\gamma}[||x-y||]$$

where $\prod(P,Q)$ is the set of all joint distributions whose margins are P and Q.

In most cases though, the EMD is calculated using the following duality

Definition 8.17.2 — Kantorovich-Rubinstein duality. By K-R duality, the following is an equivalent expression the the earth mover's distance:

$$EMD(P,Q) = \sup_{||f||_L \leq 1} \mathbb{E}_{x\sim P}[f(x)] - \mathbb{E}_{x\sim Q}[f(x)]$$

where $||f||_L$ is a 1-Lipschitz continuous function, that is,$\forall x \; ||\nabla f(x)|| \leq 1$

R in simpler notation, given generator g and a critic c (which is a different name to the discriminator), we calculate $\min_g \max_c \mathbb{E}[c(x)] - \mathbb{E}[c(g(x))]$

8.17.2 wGAN model

If so, wGAN is a generative model that uses the EMD as its loss function. this loss was proved to avoid mode collapse, and was less prone to vanishing gradients. On the other-hand, the algorithm implemented in the paper was slow, and its basic requirements are hard to initialize (for example, it is difficult to attain the property of 1-Lipschitz of f).
but what is so different, if we only change the loss? in the previous remark we've denoted the discriminator with c instead of d - this was to emphasise that instead of using a discriminator to distinguish real and fake images, the critic provides a score to the "realness" and "fakeness" of a given image. The motivation to this change is the fact that we rather seek distance between the fake generated distribution and the real distribution, instead of simply classifiying right or wrong, as this provies a more detailed measure of our model precision

8.18 Translation (Pix2Pix)

As a cool last topic, GAN can also be used in translation tasks. For example - translate a GTA5 footage (say, first person driver mode) to real life driving footage, or edges image to regular one.
Briefly going over the details, in the paper they've used the regular cGAN loss, with additional ℓ_1 regularization

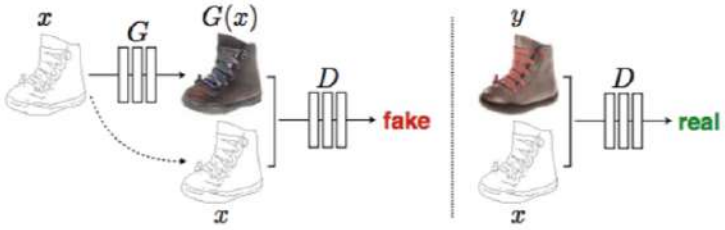


Figure 8.8: high level description of Edges-Image-to-Image translation, that uses a conditional-GAN framework. in the example, the conditioning image x is applied as the input to G and to D. Notice that both x and y are needed, where the latter are not-at-all-trivial labels

$$L(G,D) = \mathbb{E}_{x,y}[log(D(x,y))] + \mathbb{E}_{x,z}[log(1 - D(x,G(x,z)))] + \lambda \mathbb{E}_{x,y,z}[||y - G(x,z)||]$$

where x is an input from the data, y is an encoding used by D, z is an encoding used by G and λ is a regularization parameter, that was reported to provide the best results when $\lambda \approx 100$. finally, the objective was $arg\min_G \max_D L(G,D)$.
The Generator G resembled an auto-encoder (compressed the input intro low dimension and then learns how to up-sample in into the output image), and the Discriminator was changed such that it works by classifying $N \times N$ patches in the image as real vs fake (instead of looking at the entire image). This was justified by the authors of the paper by saying that such discrimination provided sharper details (also, it performed faster than regular discriminator)

9. Optimization

In our last (and short) chapter, we will discuss various approaches to minimizing our loss, and how various approaches to that minimization process could affect the underline results.

9.1 Initial settings

We have mostly discussed minimization problems in terms of gradient descent, and for that matter we introduced the gradient descent process, which updates the learned weights based on the direction of the loss's gradient

$$\theta_{i+1} = \theta_i - \eta \nabla L(N_\theta(x), y)$$

Furthermore, we stated that we typically calculate the gradient only in terms of batches that are chosen randomly. This is known as Stochastic Gradient descent.

$$\nabla L \approx \frac{1}{|B|} \sum_{i \in B} \nabla l(N_\theta(x_i), y_i)$$

such process can also be modeled by `Langevin process`[1], up to some constant

$$\dot{x}(t) = -\nabla E(x(t)) + \eta(t)$$

which is solved in terms of the probabilities of x

$$\Pr[x] = \exp(-E(x)/\sigma^2)$$

where we can think of $E(x)$ as our loss function, and σ as some deviation associated with $\Pr[x]$. This should mostly appeal to your intuition, as we understand that in an SGD process, some random output is always possible, though such output is still governed by a known function that is also exponentially decreasing.

[1] This is deeply related to the Langevin equation, defining the movement of a random particle (Brownian motion) $m\ddot{x} = -\alpha\dot{x} + \delta F(t)$, which has decaying average solution $\langle x^2(t) \rangle = \frac{mk_B T}{\alpha^2}(1 - e^{-t\frac{\alpha}{m}})$

9.2 Sharp/smooth minima

(based on the paper on `large batch training for deep learning: generalization gap and sharp minima`). How changing our batch size affect the results? starting with the paper's conclusions, we know that large batches tend to

- Explore less the space
- converge to sharper minima
- the resulted neural network tend to overfit more often

R [Sharp Minima] A sharp minima is a minima point for which the functions values change close the that minima. Compared to a flat mimima, points close to a sharp minima are very different in value - say $f(x)$ achieves minimum for x_{sharp} and x_{flat}. so $f(x_{sharp} \pm \varepsilon) \gg f(x_{sharp})$ (or $\ll$ for that matter), yet $f(x_{flat} \pm \varepsilon) \approx f(x_{flat})$

We now compare the loss function in terms of both the training and test data In the paper, that

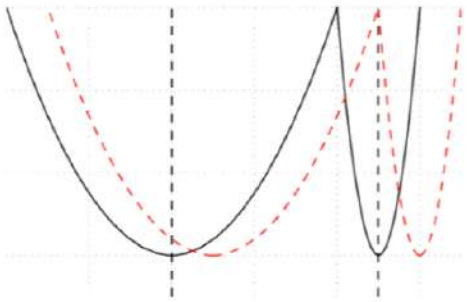


Figure 9.1: Train (black plot) and test (red plot) loss achieves similar minima values when said minima is flat, but very different values when that minima is sharp

differences between sharp and flat minima was experimentally shown to be the result of choosing large or small batch size respectively. More specifically, they have defined the sharpness as `sharp`$= \max \frac{L(x+y)}{1+L(x)}$ where y were small values in some range $[-\varepsilon, \varepsilon]$, and various options for ε were tested.
Intuitively speaking, as our minimization process involves random perturbations, and such perturbations may cause different results depending on where in the loss landscape they occur - smaller batch admits more noise, which in turn result in large randomized steps in arbitrary direction in our loss landscape. such large randomized steps could result in moving outside some local minima, to which we will not be able to return in the iterative learning process (it is easier to "escape" sharp minima, compared to flat minima).

9.3 Convergence Acceleration Strategies

Over the years, many versions that claim to out-perform the basic SGD process were introduced. Lets go over some of the more famous ones

9.3.1 Momentum

Stochastic gradient descent with momentum "remembers" the update $\Delta\theta$ at each iteration, and determines the next update as a linear combination of the gradient and the previous update:

$$\Delta\theta = \alpha\Delta\theta - \eta\nabla L(\theta)$$
$$\theta = \theta + \Delta\theta$$

that is, $\theta = \theta + \alpha\Delta\theta - \eta\nabla L(\theta)$.
The name `momentum` stems from the concept of momentum in mechanics (physics) - Think of θ as the momentum function of a particle, that is, $\theta(t)$ is the momentum (mass$\times$velocity) of our particle at time t. Also, think of the loss function L as some potential (say, gravitation), and ∇L as the force that governs how the momentum $\theta(t)$ behaves. Our particle tends to keep traveling in the same direction, and therefore we add the term that indicates that tendency of the particle - which is exactly $\alpha\Delta\theta$, where α tells us how that tendency is strong

9.3.2 Averaging

Averaged SGD is pretty much the same as regular SGD but with another feature - in every iteration we save in our memory the average weights of the previous iterations, namely, aside from the SGD iteration, we also calculate $\bar{\theta} = \frac{1}{t}\sum_{i=1}^{t}\theta_i$, where t is the current iteration. Finally, when the optimization is done, $\bar{\theta}$ takes the place of the weight vector θ. This method on one hand reduces noise (as it resembles a committee), as the results are always averaged out, but does require more memory, as $\bar{\theta}$ should always be saved.

9.3.3 AdaGrad

for this method, denote subscript i as the iteration index and upper-script k the index of the vector. In AdaGrad we perform regular SGD with an additional normalization to the weights, that is done per entry as followed -

$$\theta_k^{i+1} = \theta_k^i - \eta\nabla L_k^i / \sqrt{\varepsilon + \sum_{j=0}^{t}\nabla(L_k^{i-j})^2}$$

Notice that $\eta\nabla L_k^i$ is a regular SGD iteration, so lets focus on the denominator - for some time window of t steps, we look at the norm value of ∇L and normalize by that factor (up to an ε). If, for example, that norm value was small, then the gradients were weak and normalizing by a small factor would increase the overall gradient. Same goes for very big gradients, in which the overall gradient will decrease. The ε term inside the square root ensures that when the sum value is extremely small, it will be redundant compared to ε, and the update rule wouldn't explode.

9.3.4 Adaptive Moment Estimation (Adam)

Combines both the momentum and the AdaGrad. In this algorithm, we use running averages of both the gradients and the second moments of the gradients. Formally we define

$$m_\theta^{t+1} = \beta_1 m_\theta^t + (1-\beta_1)\nabla_\theta L^t$$
$$v_\theta^{t+1} = \beta_2 v_\theta^t + (1-\beta_2)(\nabla_\theta L^t)^2$$
$$\hat{m}_\theta = \frac{m_\theta^{t+1}}{1-\beta_1}$$
$$\hat{v}_\theta = \frac{v_\theta^{t+1}}{1-\beta_2}$$

and those will be used in the update rule

$$\theta^{t+1} = \theta^t - \eta\frac{\hat{m}_\theta}{\sqrt{\hat{v}_\theta + \varepsilon}}$$