Deep Learning

Cristian Perez Jensen

February 4, 2025

Contents

List of symbols

$\doteq$	Equality by definition
$\overset{!}{=}$	Conditional equality
$\approx$	Approximate equality
$\propto$	Proportional to
$\mathbb{N}$	Set of natural numbers
$\mathbb{R}$	Set of real numbers
$i : j$	Set of natural numbers between i and j. *I.e.*, $\{i, i+1, \dots, j\}$
$f : A \to B$	Function f that maps elements of set A to elements of set B
$\mathbb{1}\{\text{predicate}\}$	Indicator function (1 if predicate is true, otherwise 0)
$\boldsymbol{v} \in \mathbb{R}^n$	n-dimensional vector
$\boldsymbol{M} \in \mathbb{R}^{m \times n}$	$m \times n$ matrix
$\boldsymbol{M}^\top$	Transpose of matrix $\boldsymbol{M}$
$\boldsymbol{M}^{-1}$	Inverse of matrix $\boldsymbol{M}$
$\det(\boldsymbol{M})$	Determinant of $\boldsymbol{M}$
$\frac{\mathrm{d}}{\mathrm{d}x} f(x)$	Ordinary derivative of $f(x)$ w.r.t. x at point $x \in \mathbb{R}$
$\frac{\partial}{\partial x} f(\boldsymbol{x})$	Partial derivative of $f(\boldsymbol{x})$ w.r.t. x at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^n$	Gradient of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$
$\boldsymbol{\nabla}^2_{\boldsymbol{x}} f(\boldsymbol{x}) \in \mathbb{R}^{n \times n}$	Hessian of $f : \mathbb{R}^n \to \mathbb{R}$ at point $\boldsymbol{x} \in \mathbb{R}^n$

1 Connectionism

1.1 McCulloch-Pitts neuron

One of the first approaches to modeling functions of nervous functions with an abstract mathematical model is the McCulloch-Pitts neuron [McCulloch and Pitts, 1943]. It treats neurons as linear threshold elements, which receive and integrate a large number of inputs and produce a Boolean output. More specifically, it receives $\boldsymbol{x} \in \{0,1\}^n$ as input and has $\boldsymbol{\sigma} \in \{-1,1\}^n, \theta \in \mathbb{R}$ as parameters. Its transfer function is formalized as

$$f[\boldsymbol{\sigma},\theta](\boldsymbol{x}) = \mathbb{1}\{\boldsymbol{\sigma}^\top \boldsymbol{x} \geq \theta\}.$$

The synapses $\boldsymbol{\sigma}$ are inhibitory if -1 and excitatory if $+1$. However, the problem with this model is that it does not specify how to set or adjust its parameters.

1.2 Perceptron

The perceptron [Rosenblatt, 1958] is the first model to perform supervised learning, where patterns are represented as feature vectors $\boldsymbol{x} \in \mathbb{R}^d$ and have binary class memberships $y \in \{-1,+1\}$. Rosenblatt [1958] proposed to use a linear threshold unit with synaptic weights $\boldsymbol{w} \in \mathbb{R}^d$ and threshold $b \in \mathbb{R}$,

$$f[\boldsymbol{w},b](\boldsymbol{x}) = \operatorname{sgn}\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right),$$

where

$$\operatorname{sgn}(z) \doteq \begin{cases} +1 & z > 0 \\ 0 & z = 0 \\ -1 & z < 0. \end{cases}$$

This model implicitly induces a decision boundary, where

$$\boldsymbol{w}^\top \boldsymbol{x} + b \overset{!}{=} 0 \iff \frac{\boldsymbol{w}^\top \boldsymbol{x}}{\|\boldsymbol{w}\|} + \frac{b}{\|\boldsymbol{w}\|} \overset{!}{=} 0.$$

The perceptron thus models the decision boundary as a hyperplane in $\mathbb{R}^n$ with normal vector $\boldsymbol{w}/\|\boldsymbol{w}\|$ and $-b/\|\boldsymbol{w}\|$ is the signed distance of the hyperplane to the origin.[1] Furthermore, we can formalize how bad/good the model is for a data point by the signed distance function,

$$\gamma[\boldsymbol{w},b](\boldsymbol{x},y) = \frac{y(\boldsymbol{w}^\top \boldsymbol{x} + b)}{\|\boldsymbol{w}\|}.$$

The sign of $\gamma(\cdot,\cdot)$ encodes the correctness of the classification. The following is a short proof of this fact,

$$\begin{aligned} f[\boldsymbol{w},b](\boldsymbol{x}) = y &\iff \operatorname{sgn}\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) = y \\ &\iff \operatorname{sgn}\left(y\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right)\right) = 1 \\ &\iff \operatorname{sgn}(\gamma[\boldsymbol{w},b](\boldsymbol{x},y)) = 1 \\ &\iff \gamma[\boldsymbol{w},b](\boldsymbol{x},y) > 0. \end{aligned}$$

[1] In Hesse normal form, a hyperplane is formulated by

$$\boldsymbol{n}^\top \boldsymbol{x} - d = 0,$$

where $\boldsymbol{n}$ is a unit vector and d is the shortest distance of the hyperplane to the origin. The distance of any point $\boldsymbol{x}$ to the hyperplane is given by $|\boldsymbol{n}^\top \boldsymbol{x} - d|$.

We define the margin of a classifier on training data $\mathcal{S}$ as the minimum signed distance,

$$\gamma[\boldsymbol{w}, b](\mathcal{S}) = \min_{(\boldsymbol{x},y)\in\mathcal{S}} \gamma[\boldsymbol{w}, b](\boldsymbol{x}, y).$$

If $\gamma[\boldsymbol{w}, b](\mathcal{S}) > 0$, then the dataset has been linearly separated by a hyperplane, formed by the parameters, *i.e.*, all classifications are correct. Moreover, we say that $f[\boldsymbol{w}, b]$ γ-separates the dataset $\mathcal{S}$ if $\gamma[\boldsymbol{w}, b](\mathcal{S}) \geq \gamma$, where γ is also called the margin.[2]

[2] Finding the maximum margin classifier is known as support vector machines.

The version space—see Figure 1.2—is defined as the set of all model parametrizations that correctly classify the data,

$$\mathcal{V}(\mathcal{S}) \doteq \{(\boldsymbol{w}, b) \mid \gamma[\boldsymbol{w}, b](\mathcal{S}) > 0\} \subseteq \mathbb{R}^{d+1}.$$

Geometrically, the version space of a linear threshold classifier is the intersection of n half-spaces in $\mathbb{R}^{d+1}$ as each data point imposes a linear constraint on the parameters. Hence, $\mathcal{S}$ is linearly separable if and only if $\mathcal{V}(\mathcal{S}) \neq \emptyset$. Adding data points to the dataset can only shrink the version space.

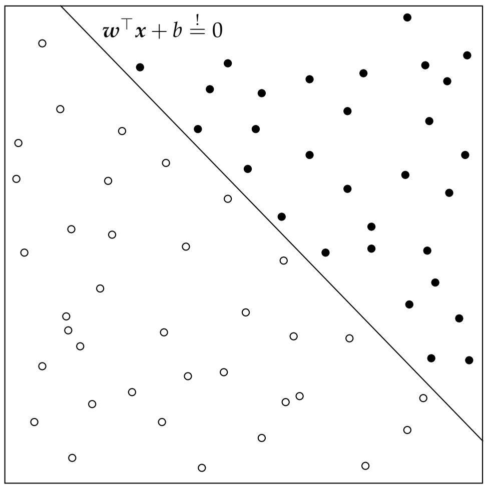


Figure 1.1. Linear separability of negative and positive data points.

The perceptron algorithm. The groundbreaking aspect of [Rosenblatt, 1958] is that it specified how to iteratively adjust the weights to provably find a solution for a linearly separable dataset.[3] (However, it is not guaranteed that it will find a classifier with a small error if the data is not linearly separable.) Given a dataset $\mathcal{S} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^n$, the perceptron algorithm aims to find some solution $(\boldsymbol{w}, b) \in \mathcal{V}(\mathcal{S})$. Note that this means that it does not aim to find classifiers with small error if $\mathcal{V}(\mathcal{S}) = \emptyset$.

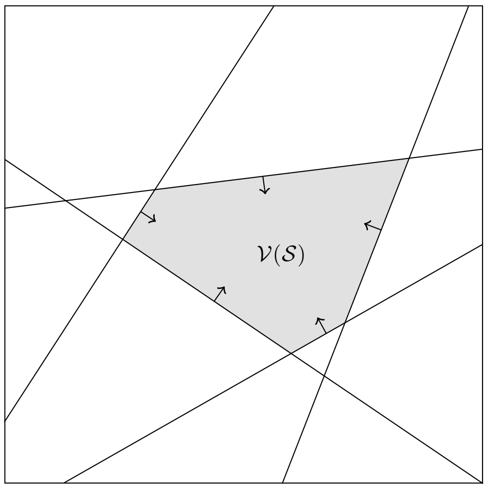


Figure 1.2. In this version space, every line represents a data point's halfspace in which it is correctly classified. As can be seen, adding data points can only shrink the version space.

[3] A solution is defined as any parameters that correctly classify all data points.

The perceptron algorithm is a mistake-driven algorithm, meaning that it will only consider data points that are misclassified by the current parameters. Given a misclassified data point $(\boldsymbol{x}, y) \in \mathcal{S}$, *i.e.*, $f[\boldsymbol{w}, b](\boldsymbol{x}) \neq y$, it has the following update rule,

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + y\boldsymbol{x}$$
$$b \leftarrow b + y.$$

We keep going through the dataset, presenting data points in an arbitrary order, until every data point is correctly classified—see Algorithm 1. Note that this algorithm will never converge if $\mathcal{S}$ is not linearly separable. If we start from $\boldsymbol{w}_0 \in \operatorname{span}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, then all weight vectors will remain in the span of the data,

$$\boldsymbol{w}_t \in \operatorname{span}(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n), \quad \forall t.$$

```
$w \leftarrow \mathbf{0}$
$b \leftarrow 0$
mistake $\leftarrow$ true
**while** mistake $=$ true **do**
    mistake $\leftarrow$ false
    **for** $(x, y) \in \mathcal{S}$ **do**
        **if** $f[w, b](x) \neq y$ **then**
            $w \leftarrow w + yx$
            $b \leftarrow b + y$
            mistake $\leftarrow$ true
        **end if**
    **end for**
**end while**
**return** $(w, b)$
```

Algorithm 1. The perceptron algorithm.

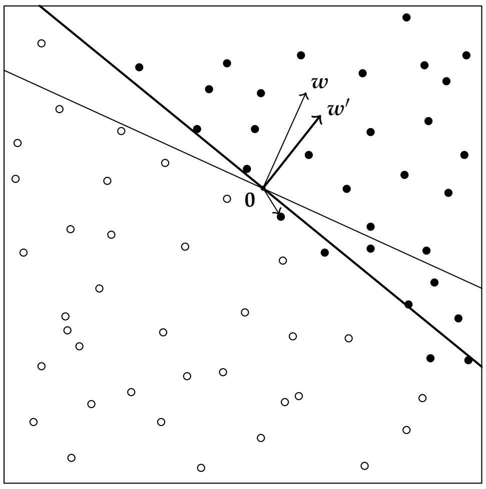


Figure 1.3. One update of the perceptron algorithm without a bias.

Proof of convergence. In order to prove convergence of the perceptron algorithm for linearly separable data, we will restrict ourselves to the case where no bias is necessary and $b = 0$. We denote the weights after t updates of the perceptron algorithm (ignoring correctly classified samples) as w_t and we initialize $w_0 = \mathbf{0}$.

Lemma 1.1. Let $w \in \mathbb{R}^d$ with $\|w\| = 1$ and $\gamma \doteq \gamma[w](\mathcal{S}) > 0$. (*I.e.*, $\mathcal{S}$ is γ-separable with w.) Then,

$$w^\top w_t \geq t\gamma, \forall t.$$

Proof. This can easily be shown by a recursion,

$$\begin{aligned} w^\top w_{t+1} &= w^\top (w_t + yx) && \text{Perceptron update.} \\ &= w^\top w_t + yw^\top x && \text{Linearity.} \\ &= w^\top w_t + \gamma[w](x, y) && \|w\| = 1. \\ &\geq w^\top w_t + \gamma. && \gamma = \min_{x,y} \gamma[w](x,y) \leq \gamma[w](x,y). \end{aligned}$$

Now, it is easy to show the result by induction, starting from $w_0 = \mathbf{0}$. ∎

Lemma 1.2. Let $R \doteq \max_{x \in \mathcal{S}} \|x\|$, then

$$\|w_t\| \leq R\sqrt{t}.$$

Proof. This can easily be shown by a recursion,

$$\begin{aligned}
\|\boldsymbol{w}_{t+1}\|^2 &= \|\boldsymbol{w}_t + y\boldsymbol{x}\|^2 \\
&= \|\boldsymbol{w}_t\|^2 + \|y\boldsymbol{x}\|^2 + 2y\boldsymbol{w}_t^\top \boldsymbol{x} \\
&= \|\boldsymbol{w}_t\|^2 + \|\boldsymbol{x}\|^2 + 2\|\boldsymbol{w}_t\|\gamma[\boldsymbol{w}_t](\boldsymbol{x}, y) \\
&\leq \|\boldsymbol{w}_t\|^2 + \|\boldsymbol{x}\|^2 \\
&\leq \|\boldsymbol{w}_t\|^2 + R^2.
\end{aligned}$$

Perceptron update.

Cosine theorem.

The perceptron update condition is $\gamma[\boldsymbol{w}_t](\boldsymbol{x}, y) \leq 0$.

The claim follows by induction, starting from $\boldsymbol{w}_0 = \boldsymbol{0}$, and taking the square root. ∎

Theorem 1.3 ([Novikoff, 1962]). Let the dataset $\mathcal{S}$ be γ-separable and $R \doteq \max_{\boldsymbol{x} \in \mathcal{S}} \|\boldsymbol{x}\|$, then the perceptron algorithm converges in less than $\lfloor R^2/\gamma^2 \rfloor$ steps.

Proof. By Lemmas 1.1 and 1.2, we have the following inequality,

$$1 \geq \cos\angle(\boldsymbol{w}, \boldsymbol{w}_t) = \frac{\boldsymbol{w}^\top \boldsymbol{w}_t}{\|\boldsymbol{w}\|\|\boldsymbol{w}_t\|} = \frac{\boldsymbol{w}^\top \boldsymbol{w}_t}{\|\boldsymbol{w}_t\|} \geq \frac{t\gamma}{R\sqrt{t}} = \frac{\sqrt{t}\gamma}{R}.$$

When $\sqrt{t}\gamma/R = 1$, then $\cos\angle(\boldsymbol{w}, \boldsymbol{w}_t) = 1$, which means that $\boldsymbol{w}_t$ has converged, since $\boldsymbol{w} \in \mathcal{V}(\mathcal{S})$. Hence, we have the following sufficient condition for convergence,

$$t = \frac{R^2}{\gamma^2}.$$

It might converge earlier, so we have an upper bound on the number of timesteps until convergence. Since t is integer, this bound is $\lfloor R^2/\gamma^2 \rfloor$. ∎

This theorem does not only guarantee convergence of the perceptron algorithm, but also relates the separation margin γ to the number of steps necessary for convergence. If γ is large, it should be easier to find parameters that classify all data points correctly than if γ is small, because then you have to be very precise—see Figure 1.1.

However, the problem with this approach is that it requires linear separability of the data, which is not fulfilled for simple problems like the XOR [Minsky and Papert, 1969],

$$\mathcal{S} = \left\{ \left(\begin{bmatrix} 0 \\ 0 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}, 1 \right), \left(\begin{bmatrix} 0 \\ 1 \end{bmatrix}, -1 \right), \left(\begin{bmatrix} 1 \\ 0 \end{bmatrix}, -1 \right) \right\}.$$

Number of unique linear classifications. Assume that we are given a dataset $\mathcal{S} \subset \mathbb{R}^d$ of n points, then we define the set of possible linear classifications of this dataset as,

$$\mathcal{C}(\mathcal{S}, d) \doteq \left| \left\{ \boldsymbol{y} \in \{-1, +1\}^n \;\middle|\; \exists \boldsymbol{w} \in \mathbb{R}^d : \forall i \in [n] : y_i\left(\boldsymbol{w}^\top \boldsymbol{x}_i\right) > 0 \right\} \right|.$$

We assume points to be in general position, which means that any subset $\Xi \subseteq \mathcal{S}$ with $|\Xi| \leq d$ is linearly independent.[4]

[4] This is a very weak condition.

Theorem 1.4 ([Cover, 1965]). Given n d-dimensional points in general position,

$$\mathcal{C}(n+1, d) = 2\sum_{i=0}^{d-1} \binom{n}{i}.$$

Proof. It is easy to show that the initial values are

$$\mathcal{C}(1, d) = 2, \quad \mathcal{C}(n, 1) = 2, \quad \forall n, d \in \mathbb{N}.$$

An important insight for the second case is that $\boldsymbol{w}$ cannot move from the origin, because there is no bias.

Consider a realizable classification of n points. *I.e.*, any classification of all $\boldsymbol{x} \in \mathcal{S}$ that is linearly separable. This classification has a non-empty version space $\mathcal{V}$. Let $\boldsymbol{x}_{n+1}$ be a pattern that we add to $\mathcal{S}$. This gives us two new version spaces,

$$\mathcal{V}^+ \doteq \mathcal{V} \cap \left\{\boldsymbol{w} \;\middle|\; \boldsymbol{w}^\top \boldsymbol{x}_{n+1} > 0\right\}, \quad \mathcal{V}^- \doteq \mathcal{V} \cap \left\{\boldsymbol{w} \;\middle|\; \boldsymbol{w}^\top \boldsymbol{x}_{n+1} < 0\right\},$$

Per existing classification, there are two possible situations,

1. *$\mathcal{V}^+$ and $\mathcal{V}^-$ are non-empty.* Hence, $\boldsymbol{x}_{n+1}$ can be classified as either $+1$ or -1. This is the case if and only if there is a $\boldsymbol{w} \in \mathcal{V}$ such that $\boldsymbol{w}^\top \boldsymbol{x}_{n+1} = 0$.[5] Recall that we want to know the number of classifications of this new dataset $\mathcal{S} \cup \{\boldsymbol{x}_{n+1}\}$. For any classification of $\mathcal{S}$ that is in this situation, we can make two new classifications: $\boldsymbol{x}_{n+1}$ is classified $+1$ or -1. There are $\mathcal{C}(n, d-1)$ such classifications, because the constraint on $\boldsymbol{w}$ makes the problem effectively $(d-1)$-dimensional with n data points. Hence, we gain $2 \cdot \mathcal{C}(n, d-1)$ classifications;

2. *$\mathcal{V}^+$ is non-empty and $\mathcal{V}^-$ is empty or $\mathcal{V}^+$ is empty and $\mathcal{V}^-$ is non-empty.* In this case, we would only be able to create one new classification for each existing classification, and there are $\mathcal{C}(n, d) - \mathcal{C}(n, d-1)$ such original classifications. Hence, we gain $\mathcal{C}(n, d) - \mathcal{C}(n, d-1)$ classifications.

[5] Because then we would be able to shift the hyperplane, formed by $\boldsymbol{w}$, infinitesimally to allow arbitrary classification of $\boldsymbol{x}_{s+1}$ while keeping all other classifications the same.

In conclusion, in total we can create

$$\begin{aligned}\mathcal{C}(n+1, d) &= \mathcal{C}(n, d) - \mathcal{C}(n, d-1) + 2 \cdot \mathcal{C}(n, d-1) \\ &= \mathcal{C}(n, d) + \mathcal{C}(n, d-1)\end{aligned}$$

classifications of $n+1$ data points. The claim follows by induction using Pascal's identity. ■

Corollary. For $n \leq d$, we have $\mathcal{C}(n, d) = 2^n$.

It turns out that after $n = 2d$, there is a steep decrease in number of linear classifications, quickly moving toward 0.

1.3 *Parallel distributed processing*

The philosophy behind modern machine learning comes from PDP (***P**arallel **D**istributed **P**rocessing*) [Rumelhart et al., 1986, McClelland et al., 1987]. The elements of PDP are the following,

1. *A set of processing units with states of activation*. The basic building blocks of models;
2. *Output functions for each unit*. They define how the output of the units are computed;
3. *A pattern of connectivity between units*. They define how the units interact with each other;
4. *Propagation rules for propagating patterns of activity*. Makes the dependence of the units explicit;
5. *Activation functions for units*. These make the model expressive beyond linearity;
6. *A learning rule to modify connectivity based on experience*. This is what the training data is used for;
7. *An environment within which the system must operate*. Formalized by a loss function.

One can change and experiment with each of these design elements. The fact that we still use this wording says much about the impact that PDP has had on modern deep learning.

1.4 *Hopfield networks*

The Hopfield network [Hopfield, 1982] models an associative memory, which aims to reconstruct a "memory" from an input that has been subjected to noise. It does so by defining a parameterized energy function via second-order interactions between n binary neurons,

$$H(\boldsymbol{X}) \doteq -\frac{1}{2}\sum_{i=1}^{d}\sum_{j=1}^{d} w_{ij}X_iX_j + \sum_{i=1}^{n} b_iX_i, \quad \boldsymbol{X} \in \{-1,+1\}^d.$$

The couplings w_{ij} quantify the interaction strength between neurons and the biases b_i act as thresholds. We constrain the weights such that

$$w_{ii} = 0, w_{ij} = w_{ji}, \quad \forall i,j \in [d].$$

Hopfield networks follow a simple dynamic starting from some input state. It iteratively updates the neurons as follows,

$$X_i \leftarrow \begin{cases} +1 & H([\ldots, X_{i-1}, +1, X_{i+1}, \ldots]) \leq H([\ldots, X_{i-1}, -1, X_{i+1}, \ldots]) \\ -1 & \text{otherwise.} \end{cases}$$

Hence, X_i becomes the value that minimizes the energy function, given the rest of the state. In practice, we do not need to evaluate the full energy function for the update—we only need the effective field per neuron,

$$H_i \doteq \sum_{j=1}^{n} w_{ij} X_j - b_i.$$

Then, updates can equivalently be expressed as

$$X_i \leftarrow \operatorname{sgn}(H_i), \quad \operatorname{sgn}(z) = \begin{cases} +1 & z \geq 0 \\ -1 & z < 0. \end{cases}$$

Hebbian learning. The goal of Hopfield networks is to use the update dynamics to evolve noisy stimulus toward a target pattern. For example, we might want noisy greyscale images to converge to images of numbers 0–9. Given a set of patterns that we wish to memorize, $\mathcal{S} \subseteq \{-1, +1\}^d$, Hebbian learning involves setting the weights as outer products,

$$w_{ij} = \frac{1}{d} \sum_{t=1}^{n} x_{t,i} x_{t,j} \implies \boldsymbol{W} = \frac{1}{d} \sum_{t=1}^{n} \boldsymbol{x}_t \boldsymbol{x}_t^\top.$$

Intuitively, neurons that are frequently in the same state reinforce each other (positive coupling), whereas neurons that are frequently in opposite states repel each other (negative coupling).

The minimal requirement of considering a pattern $\boldsymbol{x}_t$ as memorized is that it is meta-stable, *i.e.*, when in the state of a pattern, the update rule will not make any updates,

$$\begin{aligned} x_{t,i} &\overset{!}{=} \operatorname{sgn}\left(\sum_{j=1}^{d} w_{ij} x_{t,j} \right), \quad \forall i \in [d]. \\ &= \operatorname{sgn}\left(\sum_{j=1}^{d} \frac{1}{d} \sum_{r=1}^{n} x_{r,i} x_{r,j} x_{t,j} \right) \\ &= \operatorname{sgn}\left(x_{ti} + \underbrace{\frac{1}{d} \sum_{j=1}^{d} \sum_{r \neq t}^{n} x_{r,i} x_{r,j} x_{t,j}}_{\doteq C_{t,i}} \right). \end{aligned}$$

We ignore biases from now on.

Hebbian learning.

Here, $C_{t,i}$ is the cross-talk between patterns, and ideally $|C_{t,i}| < 1$, for all patterns $t \in [n]$ and indices $i \in [d]$, because then the minimal requirement is fulfilled.

If we assume that the patterns have *i.i.d.* random signs and we look at the limit $d \to \infty$, then we have

$$C_{t,i} \sim \mathcal{N}\left(0, \frac{n}{d}\right).$$

The probability of a single sign being flipped is then

$$P[-x_{t,i} C_{t,i} \geq 1] \approx \int_{1}^{\infty} \exp\left(-\frac{dz^2}{2n}\right) \mathrm{d}z = \frac{1}{2}\left(1 - \operatorname{erf}\left(\sqrt{d/2n}\right)\right).$$

Hence, the ratio n/d controls the asymptotic error rate. At $d/n \approx 0.138$, a phase transition occurs, beyond which an avalanche of errors occur. Requiring a pattern to be retrieved with high probability, one gets a sublinear capacity bound of

$$s \leq \frac{d}{2\log_2(d)}.$$

Recently, research has been done on increasing the capacity of Hopfield networks by making use of higher-order energy functions [Krotov and Hopfield, 2016, Demircigil et al., 2017]. The increased capacity is the consequence of increased number of local minima in complex cost functions. Furthermore, Ramsauer et al. [2020] have investigated a connection between Hopfield networks and transformers.

2 *Feedforward networks*

2.1 *Regression models*

In least squares, we attempt to fit a linear model, $f[\boldsymbol{w}](\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$, to data points with an MSE (***M**ean **S**quared **E**rror*) loss,

$$\ell[\boldsymbol{w}](\mathcal{S}) = \frac{1}{2}\sum_{i=1}^{n}\left(\boldsymbol{w}^\top \boldsymbol{x}_i - y_i\right)^2.$$

Summarizing the patterns into a design matrix $\boldsymbol{X} \in \mathbb{R}^{d\times n}$ and output vector $\boldsymbol{y} \in \mathbb{R}^n$, we get the following loss,

$$\ell[\boldsymbol{w}](\mathcal{S}) = \frac{1}{2}\|\boldsymbol{X}^\top \boldsymbol{w} - \boldsymbol{y}\|^2.$$

This loss function is convex, so we can find the minimizer by setting the gradient to zero,

$$\boldsymbol{\nabla}_{\boldsymbol{w}}\ell[\boldsymbol{w}](\mathcal{S}) = \boldsymbol{X}^\top\boldsymbol{X}\boldsymbol{w} - \boldsymbol{X}^\top\boldsymbol{y} \overset{!}{=} \boldsymbol{0}.$$

This gives the OLSE (***O**rdinary **L**east **S**quares **E**stimator*),

$$\boldsymbol{w}^\star = (\boldsymbol{X}^\top\boldsymbol{X})^{-1}\boldsymbol{X}^\top\boldsymbol{y}.$$

In logistic regression, the outputs are binary. Hence, we make use of the sigmoid function $\sigma : \mathbb{R} \to (0,1)$,

$$\sigma(z) \doteq \frac{1}{1+\exp(-z)}.$$

And the model is linear with this activation function, $f[\boldsymbol{w}](\boldsymbol{x}) = \sigma\big(\boldsymbol{w}^\top\boldsymbol{x}\big)$. This outputs the probability of the label of $\boldsymbol{x}$ being 1. We train this model to optimize the cross-entropy loss,

$$\ell[\boldsymbol{w}](\mathcal{S}) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log\sigma(\boldsymbol{w}^\top\boldsymbol{x}_i) + (1-y_i)\log(1-\sigma(\boldsymbol{w}^\top\boldsymbol{x}_i)).$$

This problem does not have a closed-form solution, but we can optimize the weights by SGD (***S**tochastic **G**radient **D**ecent*) with the following gradient,

$$\boldsymbol{\nabla}_{\boldsymbol{w}}\ell[\boldsymbol{w}](\boldsymbol{x}_i, y_i) = \left(\sigma\left(\boldsymbol{w}^\top\boldsymbol{x}_i\right) - y_i\right)\boldsymbol{x}_i.$$

2.2 *Layers and units*

A mapping is a function with vectors as input and output. The following function is an example of a mapping,

$$f[\boldsymbol{W},\boldsymbol{b}](\boldsymbol{x}) = \phi(\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}), \quad \boldsymbol{W} \in \mathbb{R}^{m\times n}, \boldsymbol{b} \in \mathbb{R}^n,$$

where ϕ is a pointwise activation function and m is the width of the layer.

Deep neural networks compose maps in sequence,

$$G = F_L[\boldsymbol{\theta}_L] \circ \cdots \circ F_1[\boldsymbol{\theta}_1],$$

where $\boldsymbol{\theta}_\ell$ are the (adjustable) weights of layer ℓ. Intuitively, models with higher depth are able to extract features with increasing complexity. Such networks induce intermediate results (or layer activations),

$$\boldsymbol{x}^\ell \doteq (F_\ell \circ \cdots \circ F_1)(\boldsymbol{x}) = F_\ell\left(\boldsymbol{x}^{\ell-1}\right), \quad \boldsymbol{x}^0 = \boldsymbol{x}.$$

The intermediate layers are permutation symmetric, meaning that the units within a hidden layer are interchangeable if we change the order of the weights accordingly,

$$F[\boldsymbol{W}, \boldsymbol{b}](\boldsymbol{x}) = \boldsymbol{P}^{-1}\phi(\boldsymbol{PWx} + \boldsymbol{Pb}) = \boldsymbol{P}^{-1}F[\boldsymbol{PW}, \boldsymbol{Pb}](\boldsymbol{x}),$$

where $\boldsymbol{P}$ is a permutation matrix. Hence, the parameters are not unique in feedforward networks.

The presented layers differ only in their choice of activation function,

- Linear activation,
$$\phi = \mathrm{Id};$$
- Sigmoid activation,
$$\phi = \sigma;$$
- ReLU (***Re****ctifier* ***L****inear* ***U****nit*) activation,
$$\phi = (z)_+ \doteq \max\{0, z\}.$$

An essential part of training neural networks is constructing the loss function. For a regression problem, a simple—and popular—choice is the squared error loss,

$$\ell[\boldsymbol{\theta}](\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2}\|\boldsymbol{y} - f[\boldsymbol{\theta}](\boldsymbol{x})\|^2.$$

For a multi-class classification problem, the final layer must be the soft-max, which outputs a categorical probability distribution over classes,

$$\mathrm{softmax}_i(\boldsymbol{z}) = \frac{\exp(z_i)}{\sum_{j=1}^n \exp(z_j)}.$$

Usually, this type of model optimizes the cross-entropy loss,

$$\ell[\boldsymbol{\theta}](\boldsymbol{x}, y) = -\log \frac{\exp(z_y)}{\sum_{v \in \mathcal{Y}} \exp(z_v)} = -z_y + \log \sum_{v \in \mathcal{Y}} \exp(z_v),$$

where $\boldsymbol{z}$ is the pre-activation of the softmax function and $\mathcal{Y}$ is the set of classes of the problem.

In a perfect world, we would want to minimize the expected risk,

$$\mathbb{E}[\ell(y, f[\boldsymbol{\theta}](\boldsymbol{x}))].$$

However, since we do not have access to the underlying probability distribution of the data, this is intractable. Hence, we minimize the empirical risk,

$$\frac{1}{n}\sum_{i=1}^{n}\ell(y_i, f[\boldsymbol{\theta}](\boldsymbol{x}_i)).$$

In practice, we partition the dataset into training and validation sets. Then, we directly minimize the empirical risk of the training set, and approximate the expected risk with the validation set.

2.3 *Linear networks*

Linear layers are closed under composition, meaning that we do not gain any representational power by increasing the depth. However, linear layers are nice to work with for theoretical analysis.

2.4 *Residual networks*

Residual layers are formalized as follows,

$$F[\boldsymbol{W}, \boldsymbol{b}](\boldsymbol{x}) = \boldsymbol{x} + \phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) - \phi(\boldsymbol{0}).$$

They have the following property,

$$F[\boldsymbol{0}, \boldsymbol{0}] = \mathrm{Id}.$$

In most architectures, learning the identity map is non-trivial. However, it is desirable to incrementally learn a better representation, rather than having to learn it at every layer. Intuitively, the residual layer learns how to change its input representation, instead of learning how to construct a new one.

A problem with the above formalization is that the input and output must have the same dimensionality. This is solved by a projection,

$$F[\boldsymbol{V}, \boldsymbol{W}, \boldsymbol{b}](\boldsymbol{x}) = \boldsymbol{V}\boldsymbol{x} + (\phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) - \phi(\boldsymbol{0})), \quad \boldsymbol{V}, \boldsymbol{W} \in \mathbb{R}^{m\times n}.$$

He et al. [2016] showed that increasing model depth with residual layers leads to better performance than when using normal layers. This small change allows model depths of up to 100—200 layers. DenseNet [Zhu and Newsam, 2017] makes use of a similar idea of shortcutting information by feeding the output of all upstream layer activations to every layer,

$$\boldsymbol{x}^{\ell+1} = F^{\ell+1}\left(\boldsymbol{x}^{\ell}, \ldots, \boldsymbol{x}^{1}, \boldsymbol{x}\right).$$

2.5 *Sigmoid networks*

We will now look at which functions an MLP (***M****ulti-****L****ayer* ***P****erceptron*) with sigmoid activation function,

$$g[\boldsymbol{v}, \boldsymbol{W}, \boldsymbol{b}](\boldsymbol{x}) \doteq \boldsymbol{v}^{\top}\sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}), \quad \boldsymbol{v}, \boldsymbol{b} \in \mathbb{R}^{m}, \boldsymbol{W} \in \mathbb{R}^{m\times n},$$

The sigmoid function and hyperbolic tangent,

$$\sigma(z) \doteq \frac{1}{1+\exp(-z)}, \quad \tanh(z) \doteq \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)},$$

are representationally equivalent, because you can always obtain the one from the other by the following identity,

$$\tanh(z) = 2\sigma(2z) - 1.$$

are able to express. The function class of MLPs is formalized by

$$\begin{aligned}\mathcal{G}_n &\doteq \bigcup_{m=1}^{\infty} \mathcal{G}_{n,m} \\ \mathcal{G}_{n,m} &\doteq \left\{g \;\middle|\; g(\boldsymbol{x}) = \boldsymbol{v}^\top \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}), \boldsymbol{v}, \boldsymbol{b} \in \mathbb{R}^m, \boldsymbol{W} \in \mathbb{R}^{m \times n}\right\}.\end{aligned}$$

An alternative way of expressing this is as a linear span of units,

$$\mathcal{G}_n = \operatorname{span}\left\{\sigma\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) \;\middle|\; \boldsymbol{w} \in \mathbb{R}^n, b \in \mathbb{R}\right\}.$$

Definition 2.1 (Function distance metric)**.** $d_{\mathcal{K}}$ is a distance metric over a compact set $\mathcal{K}$ induced by the uniform norm,

$$d_{\mathcal{K}}(f, g) \doteq \|f - g\|_{\infty,\mathcal{K}}, \quad \|f\|_{\infty,\mathcal{K}} \doteq \sup_{\boldsymbol{x} \in \mathcal{K}} |f(\boldsymbol{x})|.$$

Definition 2.2 (Function class distance metric)**.** Let f be a function and $\mathcal{G}$ a function class, then their distance is computed as

$$d_{\mathcal{K}}(f, \mathcal{G}) \doteq \inf_{g \in \mathcal{G}} d_{\mathcal{K}}(f, g).$$

Definition 2.3 (Universal function approximator)**.** A function class $\mathcal{F}$ is approximated by function class $\mathcal{G}$ on $\mathcal{K}$ if, and only if,

$$d_{\mathcal{K}}(f, \mathcal{G}) = 0, \quad \forall f \in \mathcal{F}.$$

If this holds for all compact sets $\mathcal{K}$, then $\mathcal{G}$ is a universal approximator of $\mathcal{F}$.

Theorem 2.4 (Weierstrass theorem)**.** Polynomials are universal approximators of $\mathcal{C}(\mathbb{R})$, where $\mathcal{C}(\mathbb{R})$ is the set of all continuous functions over $\mathbb{R}$.

Theorem 2.5 ([Leshno et al., 1993])**.** Let $\phi \in \mathcal{C}^\infty(\mathbb{R})$, but not a polynomial, then

$$\operatorname{span}(\{\phi(ax + b) \mid a, b \in \mathbb{R}\})$$

universally approximates $\mathcal{C}(\mathbb{R})$.

Hence, an MLP with 1-dimensional input and output is a universal function approximator, if the activation function is not a polynomial.

Lemma 2.6 (Lifting lemma [Pinkus, 1999])**.** Let ϕ be such that

$$\operatorname{span}(\{\phi(ax+b) \mid a, b \in \mathbb{R}\})$$

universally approximates $\mathcal{C}(\mathbb{R})$, then

$$\operatorname{span}\left(\left\{\phi\left(\boldsymbol{w}^\top \boldsymbol{x} + b\right) \mid \boldsymbol{w} \in \mathbb{R}^n, b \in \mathbb{R}\right\}\right)$$

universally approximates $\mathcal{C}(\mathbb{R}^n)$.

Thus, we can lift the previous result into n dimensions, making MLPs universal approximators of continuous functions of any dimensionality. Moreover, this does not only hold for the sigmoid function, but for any smooth activation function that is not a polynomial.

However, this does not give us any insights into how depth affects performance, because the theorem assumes a single hidden layer of arbitrary width. Also, it does not provide a bound on the width of the hidden layer in order to achieve some desired error.

Theorem 2.7 ([Barron, 1993])**.** For every $f : \mathbb{R}^n \to \mathbb{R}$ with finite $\mathcal{C}_f$ and any $r > 0$, there is a sequence of one hidden layer MLPs $(g_m)_{m \in \mathbb{N}}$ such that

$$\int_{r\mathcal{B}} (f(\boldsymbol{x}) - g_m(\boldsymbol{x}))^2 \mu(d\boldsymbol{x}) \leq \mathcal{O}\left(\frac{1}{m}\right),$$

where $r\mathbb{B} \doteq \{\boldsymbol{x} \in \mathbb{R}^n \mid \|\boldsymbol{x}\| \leq r\}$ and μ is any probability measure on $r\mathbb{B}$.

Hence, if we relax the notion of approximation to squared error over a ball with radius r, we gain a decay of $1/m$ for the approximation error. Further, the approximation error bound does not depend on the input dimensionality n.

2.6 *ReLU networks*

The ReLU activation function is defined as

$$(z)_+ \doteq \max\{0, z\}.$$

Consider a layer of m ReLU units on a fixed input $\boldsymbol{x}$. In this situation, each unit is either active or inactive, where active means that its input is positive,

$$\mathbb{1}\{\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} > 0\} \in \{0, 1\}^m.$$

In this way, we can partition the input space into cells that have the same activation pattern,

$$\mathcal{X}_{\boldsymbol{\kappa}} \doteq \{\boldsymbol{x} \mid \mathbb{1}\{\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} > 0\} = \boldsymbol{\kappa}\}.$$

We can measure the complexity of a network as the amount of these cells it has. Firstly, we have the trivial upper bound $|\{\mathbb{1}\{\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}>0\} \mid \boldsymbol{x}\in\mathbb{R}^n\}| \leq 2^m$. However, we would like to obtain a stricter bound. We can represent each hidden unit as a hyperplane $\boldsymbol{w}_i^\top \boldsymbol{x} + b_i$. On one side the unit would be active and inactive on the other. Geometrically, we can thus think of it as a space, where each hidden unit represents a hyperplane—see Figure 2.1. The connected regions of these hyperplanes are the activation patterns.

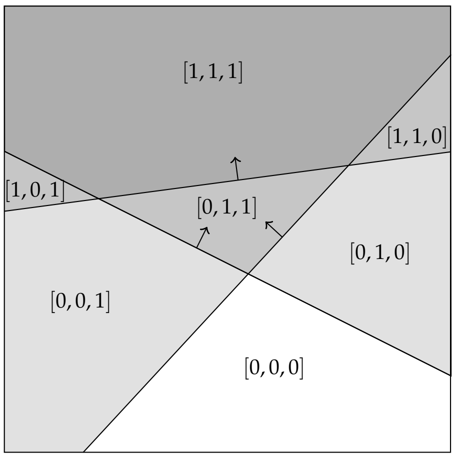


Figure 2.1. Connected regions, partitioned according to activation pattern. Each hyperplane represents a hidden unit. This shows an MLP with 2-dimensional input and 3-dimensional hidden layer. So, we are in 2 dimensions and have 3 hyperplanes.

Theorem 2.8 ([Zaslavsky, 1975]). Let $\mathcal{H}$ be a set of m hyperplanes in $\mathbb{R}^n$. Denote by $R(\mathcal{H})$ the number of connected regions of $\mathbb{R}^n \backslash \mathcal{H}$, then

$$R(\mathcal{H}) \leq \sum_{i=0}^{\min\{n,m\}} \binom{m}{i} \doteq R(m).$$

This upper bound is attained by hyperplanes in general position.

This gives us a tighter bound on the number of activation patterns.

Theorem 2.9 ([Montufar et al., 2014]). Consider a ReLU network with L layers of width $m > n$. The number of linear regions is lower bounded by

$$R(m, L) \geq R(m) \left\lfloor \frac{m}{n} \right\rfloor^{n(L-1)}.$$

Finally we have a result that relates model complexity to layer depth. By letting the amount of possible activation patterns represent complexity, this is a good argument for why deep networks tend to perform well.

Theorem 2.10 ([Shekhtman, 1982]). Piecewise linear functions are universal approximators of $\mathcal{C}(\mathbb{R})$.

Theorem 2.11 (Lebesgue). A piecewise linear function with m pieces can be written as

$$g(x) = ax + b + \sum_{i=1}^{m-1} c_i (x - x_i)_+.$$

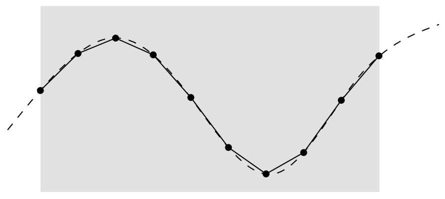


Figure 2.2. Piecewise linear approximation of a continuous function.

Hence, we can rewrite any piecewise linear function with m pieces as a sum of $m-1$ ReLUs. In 1 dimension, we can approximate any function by uniformly spacing out points on the function and connecting them as a piecewise linear function—see Figure 2.2. We can lower approximation error by increasing the number of units, approaching 0 as $m \to \infty$. Using the lifting lemma, we get the following result.

Theorem 2.12 (ReLU universality)**.** Networks with one hidden layer of ReLU units are universal function approximators.

3 Gradient-based learning

3.1 Backpropagation

In order to make use of gradient-based learning, we first need to compute the gradient. Backpropagation is an algorithm that allows the computation of any function, if we know the gradient of all basic blocks of the function.

We assume that we are differentiating the following function,

$$F[\boldsymbol{\theta}](\boldsymbol{x}) \doteq (F_L \circ \cdots \circ F_1)(\boldsymbol{x}),$$

with the following hidden layer,

$$\boldsymbol{h}_\ell \doteq F_\ell[\boldsymbol{\theta}_\ell](\boldsymbol{h}_{\ell-1}), \quad \boldsymbol{h}^0 = \boldsymbol{x}.$$

The following intermediate gradient is essential for computing the gradients of the parameters,

$$\delta_\ell = \frac{\partial \ell(\boldsymbol{y}, F[\boldsymbol{\theta}](\boldsymbol{x}))}{\partial \boldsymbol{h}_\ell}.$$

It has the following recurrence relationship (and base case),

$$\delta_L = \frac{\partial \ell(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}}, \quad \hat{\boldsymbol{y}} = F[\boldsymbol{\theta}](\boldsymbol{x})$$
$$\delta_\ell = \left[\frac{\boldsymbol{h}_{\ell+1}}{\boldsymbol{h}_\ell}\right]^\top \delta_{\ell+1}.$$

These can thus be computed efficiently in linear time with dynamic programming. Then, to compute the parameter gradient of the ℓ-th layer, we use the chain rule,

$$\frac{\partial \ell(\boldsymbol{y}, F[\boldsymbol{\theta}](\boldsymbol{x}))}{\partial \boldsymbol{\theta}_\ell} = \delta_\ell \frac{\partial \boldsymbol{h}_\ell}{\partial \boldsymbol{\theta}_\ell}.$$

3.2 Gradient descent

Gradient descent is a gradient-based learning algorithm with the following update rule,

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta \nabla h(\boldsymbol{\theta}^t), \quad \eta > 0$$

$h \doteq \ell \circ F.$

One can think of this as a discretization of the following ODE,

$$\mathrm{d}\boldsymbol{\theta} = -\nabla h(\boldsymbol{\theta})\mathrm{d}t.$$

The solution to this ODE is called the gradient flow. The gradient flow is—in a local view— the ideal trajectory to follow. However, there is a dependency on the initial condition, which results in different trajectories with different stationary points, where $\nabla h(\boldsymbol{\theta}) = \mathbf{0}$.

A key insight of analysis into the behavior of gradient descent is that it can only be successful if the gradients change slowly. This is formalized by smoothness.

Definition 3.1 (Smoothness)**.** A function h is L-smooth if there exists $L > 0$ such that

$$\|\nabla h(\boldsymbol{\theta}_1) - \nabla h(\boldsymbol{\theta}_2)\| \leq L\|\boldsymbol{\theta}_1 - \boldsymbol{\theta}_2\|, \quad \forall \boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \in \Theta.$$

This is equivalent to the following condition,

$$\|\nabla^2 h(\boldsymbol{\theta})\|_2 \leq L, \quad \forall \boldsymbol{\theta} \in \Theta.$$

From the Taylor series expansion, we have

$$\begin{aligned} h(\boldsymbol{\theta}_2) - h(\boldsymbol{\theta}_1) &= \nabla h(\boldsymbol{\theta}_1)^\top (\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1) + \frac{1}{2}(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1)^\top \nabla^2 h(\boldsymbol{\theta}_1)(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1) \\ &= -\eta\|\nabla h(\boldsymbol{\theta}_1)\|^2 + \frac{1}{2}(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1)^\top \nabla^2 h(\boldsymbol{\theta}_1)(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1) \\ &\leq -\eta\|\nabla h(\boldsymbol{\theta}_1)\|^2 + \frac{L}{2}\|\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1\|^2 \\ &= -\eta\|\nabla h(\boldsymbol{\theta})_1\|^2 + \frac{L\eta^2}{2}\|h(\boldsymbol{\theta})_1\|^2 \\ &= -\eta\left(1 - \frac{L\eta}{2}\right)\|\nabla h(\boldsymbol{\theta}_1)\|^2. \end{aligned}$$

Gradient descent update rule.

Spectral norm condition of smoothness.

Update rule of gradient descent.

A strict decrease in h is guaranteed if $\eta < 2/L$, hence we choose $\eta = 1/L$,

$$= -\frac{1}{2L}\|\nabla h(\boldsymbol{\theta}_1)\|^2.$$

As a result, we obtain sufficient decrease,

$$h(\boldsymbol{\theta}_2) \leq h(\boldsymbol{\theta}_1) - \frac{1}{2L}\|\nabla h(\boldsymbol{\theta}_1)\|^2.$$

Lemma 3.2 (Convergence of gradient descent on smooth functions)**.** Let h be L-smooth, then gradient descent with stepsize $\eta = 1/L$ will find an ϵ-critical point ($\|\nabla h(\boldsymbol{\theta})\| \leq \epsilon$) in at most

$$T = \frac{2L}{\epsilon^2}(h(\boldsymbol{\theta}_0) - h(\boldsymbol{\theta}^\star)).$$

Proof. As shown, we have sufficient decrease,

$$h(\boldsymbol{\theta}_t) - h(\boldsymbol{\theta}_{t+1}) \geq \frac{1}{2L}\|\nabla h(\boldsymbol{\theta}_t)\|^2.$$

Summing up both sides and telescoping the sum, we get

$$h(\boldsymbol{\theta}_0) - h(\boldsymbol{\theta}^\star) \geq h(\boldsymbol{\theta}_0) - h(\boldsymbol{\theta}_T) \geq \frac{1}{2L}\sum_{t=0}^{T-1}\|\nabla h(\boldsymbol{\theta}_t)\|^2 \geq \frac{1}{2L}\min_{t=0}^{T-1}\|\nabla h(\boldsymbol{\theta}_t)\|^2.$$

Thus, we get

$$\min_{t=0}^{T-1}\|\nabla h(\boldsymbol{\theta}_t)\| \leq \sqrt{\frac{2L(h(\boldsymbol{\theta}_0) - h(\boldsymbol{\theta}^\star))}{T}} \overset{!}{\leq} \epsilon.$$

The claim follows by solving for T with this condition. ■

Definition 3.3 (PL-inequality)**.** A function h satisfies the PL-inequality with $\mu > 0$ if

$$\frac{1}{2}\|\nabla h(\boldsymbol{\theta})\|^2 \geq \mu(h(\boldsymbol{\theta}) - h(\boldsymbol{\theta}^\star)), \quad \forall \boldsymbol{\theta} \in \Theta.$$

Intuitively, the PL-inequality means that if $\boldsymbol{\theta}$ has a small gradient, then $\boldsymbol{\theta}$ is nearly optimal.

Lemma 3.4. Let h be differentiable, L-smooth, and μ-PL. Then, gradient descent with stepsize $\eta = 1/L$ converges at a geometric rate,

$$h(\boldsymbol{\theta}_T) - h(\boldsymbol{\theta}^\star) \leq \left(1 - \frac{\mu}{L}\right)^T (h(\boldsymbol{\theta}_0) - h(\boldsymbol{\theta}^\star)).$$

Proof. We have

$$\begin{aligned} h(\boldsymbol{\theta}_{t+1}) &\leq h(\boldsymbol{\theta}_t) - \frac{1}{2L}\|\nabla h(\boldsymbol{\theta}_t)\|^2 \\ &\leq h(\boldsymbol{\theta}_t) - \frac{\mu}{L}(h(\boldsymbol{\theta}_t) - h(\boldsymbol{\theta}^\star)). \end{aligned}$$

Sufficient decrease.

PL-inequality.

Subtracting $h(\boldsymbol{\theta}^\star)$ from both sides yields

$$h(\boldsymbol{\theta}_t) - h(\boldsymbol{\theta}^\star) \leq \left(1 - \frac{\mu}{L}\right)(h(\boldsymbol{\theta}_t) - h(\boldsymbol{\theta}^\star)).$$

The result follows from a trivial induction. ∎

3.3 *Acceleration, adaptivity, and momentum*

Acceleration. Nesterov acceleration is a method that achieves better theoretical guarantees than vanilla gradient descent,

$$\begin{aligned} \chi^{t+1} &= \boldsymbol{\theta}^t + \beta\left(\boldsymbol{\theta}^t - \boldsymbol{\theta}^{t-1}\right) \\ \boldsymbol{\theta}^{t+1} &= \chi^{t+1} - \eta\nabla h\left(\chi^{t+1}\right). \end{aligned}$$

Extrapolation step.

Gradient descent step.

Momentum. The intuition behind momentum is that if the gradient is stable, gradient descent can make bolder steps. A simple method making use of this is the Heavy Ball method,

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta\nabla h(\boldsymbol{\theta}^t) + \beta\left(\boldsymbol{\theta}^t - \boldsymbol{\theta}^{t-1}\right), \quad \beta \in [0, 1].$$

Assuming a constant gradient δ, we have the following update,

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \eta\left(\sum_{\tau=1}^{t-1} \beta^\tau\right)\delta.$$

Thus, we see that that the learning rate increases in the case of a constant gradient.

Adaptivity. In adaptivity, we realize that we want parameter-specific learning rates, since different parameters behave differently. It defines the following,

$$\gamma_i^t \doteq \gamma_i^{t-1} + \left[\partial_i h(\boldsymbol{\theta}^t)\right]^2.$$

We then have a parameter-specific update rule,

$$\theta_i^{t+1} = \theta_i^t - \eta_i^t \partial_i h(\boldsymbol{\theta}^t), \quad \eta_i^t \doteq \frac{\eta}{\sqrt{\gamma_i^t + \delta}}.$$

Here, we see that the gradients of previous iterates influence the effective stepsize. Parameters with small gradient magnitudes will be updated with an effectively larger step size.

Adam. Adam (***Ada****ptive* ***M****oment Estimation*) [Kingma, 2014] combines adaptivity and momentum,

$$\begin{aligned} \boldsymbol{g}_t &= \beta \boldsymbol{g}_{t-1} + (1-\beta)\boldsymbol{\nabla} h(\boldsymbol{\theta}_t), \quad \beta \in [0,1] \\ \gamma_t &= \alpha \gamma_{t-1} + (1-\alpha)\boldsymbol{\nabla} h(\boldsymbol{\theta}_t)^{\odot 2}, \quad \alpha \in [0,1]. \end{aligned}$$

Moving average (smooth gradient estimator).

Exponential averaging (measure of stability in the optimization landscape).

The update rule is then

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \boldsymbol{\eta}_t \odot \boldsymbol{g}_t, \quad \boldsymbol{\eta}_t = \frac{1}{\sqrt{\gamma_t} + \delta}.$$

3.4 *Stochastic gradient descent*

When the dataset is too large, computing the full gradient is infeasible. Stochastic gradient descent solves this by computing the gradient only w.r.t. a small batch of data points at each timestep. One can prove that such an algorithm converges under certain conditions. In general, we are forced to use SGD due to the size of modern datasets. However, practically SGD outperforms GD, because it has a lower chance of getting stuck in a local optimum due to variance in the gradient estimator.

4 *Convolutional networks*

4.1 *Convolutions*

An operator T maps a function $f : \mathbb{R} \to \mathbb{R}^m$ to another function $Tf : \mathbb{R} \to \mathbb{R}^m$.

Definition 4.1 (Integral operator)**.** The integral operator T uses a kernel $G : \mathbb{R}^2 \to \mathbb{R}$ and maps a function $f : \mathbb{R} \to \mathbb{R}$ to another as follows,

$$(Tf)(u) \doteq \int_{t_1}^{t_2} H(u,t)f(t)\mathrm{d}t, \quad -\infty \leq t_1 < t_2 \leq \infty.$$

Definition 4.2 (Fourier transform)**.** The Fourier transform is an example of an integral operator with $H(u,t) = \exp(-2\pi itu)$, $t_1 = -\infty$, and $t_2 = +\infty$,

$$(\mathcal{F}f)(u) \doteq \int_{-\infty}^{\infty} e^{-2\pi itu} f(t)\mathrm{d}t.$$

Definition 4.3 (Convolution)**.** The convolutional operator is a special case of the integral operator with $H(u,t) = h(u-t)$ with $h : \mathbb{R} \to \mathbb{R}$, $t_1 = -\infty$, and $t_2 = +\infty$,

$$(f * h)(u) \doteq \int_{-\infty}^{\infty} h(u-t)f(t)\mathrm{d}t.$$

Lemma 4.4 (Convolution is commutative)**.**

$$f * h = h * f, \quad \forall f, h : \mathbb{R} \to \mathbb{R}.$$

Proof. Let $u \in \mathbb{R}$, then

$$\begin{aligned}
(h * f)(u) &\doteq \int_{-\infty}^{\infty} h(u-t)f(t)\mathrm{d}t \\
&= \int_{\infty}^{-\infty} h(v)f(u-v)(-\mathrm{d}v) && v = u - t. \\
&= \int_{-\infty}^{\infty} h(v)f(u-v)\mathrm{d}v \\
&= (f * h)(u).
\end{aligned}$$

■

Lemma 4.5 (Convolution is shift-equivariant)**.** Let f_Δ denote a shifted function,

$$f_\Delta(t) \doteq f(t+\Delta).$$

The convolution is shift-equivariant,

$$f_\Delta * h = (f * h)_\Delta.$$

So, it does not matter if we first shift and then apply convolution or the other way around.

Proof. Let $u, \Delta \in \mathbb{R}$, then

$$\begin{aligned}(f_\Delta * h)(u) &= \int_{-\infty}^{\infty} h(u-t) f(t-\Delta) \mathrm{d}t \\ &= \int_{-\infty}^{\infty} h(u+\Delta-v) f(v) \mathrm{d}v && v = t - \Delta. \\ &= (f*h)(u+\Delta) \\ &= (f*h)_\Delta(u).\end{aligned}$$

■

Lemma 4.6 (Convolution with Fourier transform)**.** The convolutional operator can be computed via the Fourier transform,

$$\mathcal{F}(f*h) = \mathcal{F}f \cdot \mathcal{F}h.$$

Then, the convolution can be obtained by

$$f * h = \mathcal{F}^{-1}(\mathcal{F}f \cdot \mathcal{F}h).$$

Proof. Note the definition of the Fourier transform in Definition 4.2,

$$\begin{aligned}\mathcal{F}(f*h)(u) &= \int_{-\infty}^{\infty} e^{-2\pi i u t} (f*h)(t) \mathrm{d}t \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-2\pi i u t} h(t-s) f(s) \mathrm{d}s \mathrm{d}t \\ &= \int_{-\infty}^{\infty}\int_{\infty}^{-\infty} e^{-2\pi i u(v+s)} h(v) f(s) \mathrm{d}s(-\mathrm{d}v) && v = t - s. \\ &= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} e^{-2\pi i u(v+s)} h(v) f(s) \mathrm{d}s \mathrm{d}v \\ &= \int_{-\infty}^{\infty} e^{-2\pi i u v} h(v) \cdot \int_{-\infty}^{\infty} e^{-2\pi i u s} f(s) \mathrm{d}s \mathrm{d}v \\ &= \mathcal{F}f \cdot \mathcal{F}h.\end{aligned}$$

■

In the discrete case, this allows computing the convolution in $\mathcal{O}(n \log n)$ with the Fast Fourier Transform algorithm and its inverse—however, this is not very useful for machine learning.

Theorem 4.7. Let T be a linear shift-equivariant operator, then there exists a kernel $h : \mathbb{R} \to \mathbb{R}$ such that T can be written as a convolution,

$$Tf = f * h.$$

Proof. A linear shift-equivariant operator T is characterized by its linearity,

$$T(\alpha f + \beta g) = \alpha T f + \beta T g, \quad \forall f, g : \mathbb{R} \to \mathbb{R}, \alpha, \beta \in \mathbb{R},$$

and its shift-equivariance,

$$T f_\Delta = (Tf)_\Delta.$$

Let δ be the Dirac delta function,

$$\delta(u) \doteq \begin{cases} \infty & u = 0 \\ 0 & u \neq 0, \end{cases} \qquad \int_{-\infty}^{\infty} \delta(u) \mathrm{d}u = 1.$$

We can write the function as an integral over delta functions,

$$f(u) = \int_{-\infty}^{\infty} f(t)\delta(t-u)\mathrm{d}t.$$

Applying T yields

$$\begin{aligned} Tf(u) &= T\left(\int_{-\infty}^{\infty} f(t)\delta(t-u)\mathrm{d}t\right) \\ &= \int_{-\infty}^{\infty} T(f(t)\delta(t-u))\mathrm{d}t \\ &= \int_{-\infty}^{\infty} f(t)T\delta(t-u)\mathrm{d}t \\ &= \int_{-\infty}^{\infty} f(t)h(u-t)\mathrm{d}t \\ &= (f * h)(u). \end{aligned}$$

T is linear.

T acts w.r.t. u, so $f(t)$ is a constant that we can pull out due to linearity.

$h(u) = T\delta(-u)$. This holds because of shift-equivariance.

■

Definition 4.8 (Discrete convolution)**.** Let $f, h : \mathbb{Z} \to \mathbb{R}$, then

$$(f * h)[u] \doteq \sum_{t=-\infty}^{\infty} f[t]h[u-t].$$

Typically, the kernel h has support over a finite window, such that $h[t] = 0, \forall t \notin [t_{\text{min}}, t_{\text{max}}]$. Then, the sum can be truncated,

$$(f * h)[u] \doteq \sum_{t=t_{\text{min}}}^{t_{\text{max}}} f[t]h[u-t].$$

Definition 4.9 (Cross-correlation)**.** Let $f, h : \mathbb{Z} \to \mathbb{R}$, then

$$(f \star h)[u] \doteq \sum_{t=-\infty}^{\infty} f[t]h[u+t].$$

Remark. This is equivalent to convolution with a flipped kernel,

$$(f \star h) = (f * \bar{h}), \quad \bar{h}[t] \doteq h[-t].$$

Let $f : [n] \to \mathbb{R}$ and $h : [n] \to \mathbb{R}$, then the Toeplitz matrix $\boldsymbol{H}_n^h \in \mathbb{R}^{(n+m-1)\times n}$ is a matrix, where h_i is on the i-th diagonal,

$$\boldsymbol{H}_n^h \doteq \begin{bmatrix} h_1 & 0 & 0 & 0 & \cdots & 0 & 0 \\ h_2 & h_1 & 0 & 0 & \cdots & 0 & 0 \\ h_3 & h_2 & h_1 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & h_m & h_{m-1} \\ 0 & 0 & 0 & 0 & \cdots & 0 & h_m \end{bmatrix}.$$

Convolution is equivalent to applying this matrix to a vectorized $\boldsymbol{f} \in \mathbb{R}^n$,

$$f * h = \boldsymbol{H}_n^h \begin{bmatrix} f_1 \\ \vdots \\ f_n \end{bmatrix}.$$

This highlights the fact that a convolutional operator is a linear layer with shared parameters within the matrix—it has increased statistical efficiency over a linear layer.

4.2 *Convolutional networks*

The goal of convolutional layers is to exploit translational equivariance of data, such as images. Furthermore, convolutional layers have higher statistical efficiency than fully connected layers, because of weight sharing. In order to achieve this, we can learn the parameters of the kernel.

In order to apply convolutions to images, we need to define the operation on 2-dimensional data,

$$(\boldsymbol{I} * \boldsymbol{W})[i,j] = \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} \boldsymbol{I}[i-k, j-\ell]\boldsymbol{W}[k,\ell].$$

In general, the data has channels. So, in practice, we learn multiple convolutional filters—one for every pair of input-output channel. The output channel is then computed as the sum over its corresponding kernels with all input channels.

Let $\boldsymbol{X}^\ell$ be the ℓ-th layer of a convolutional network. Further, let $\boldsymbol{\Delta}^\ell \doteq$

$\frac{\partial h}{\partial \boldsymbol{X}^{\ell}}$, where $h = \ell \circ f$ then we have

$$\begin{aligned}
\delta_{ij}^{\ell-1} &= \frac{\partial h}{\partial x_{ij}^{\ell-1}} \\
&= \sum_m \sum_n \frac{\partial h}{\partial x_{mn}^{\ell}} \frac{\partial x_{mn}^{\ell}}{\partial x_{ij}^{\ell-1}} \\
&= \sum_m \sum_n \delta_{mn}^{\ell} \frac{\partial}{\partial x_{ij}^{\ell-1}} \sum_k \sum_l x_{m-k,n-l}^{\ell-1} w_{kl}^{\ell} \\
&= \sum_m \sum_n \delta_{mn}^{\ell} w_{m-i,n-l}^{\ell}.
\end{aligned}$$

Thus, we can compute $\boldsymbol{\Delta}^{\ell-1}$ by a cross-correlation,

$$\boldsymbol{\Delta}^{\ell-1} = \boldsymbol{\Delta}^{\ell} \star \boldsymbol{W}^{\ell}.$$

Furthermore, we can compute the gradient w.r.t. the weights,

$$\begin{aligned}
\frac{\partial h}{\partial w_{mn}^{\ell}} &= \sum_i \sum_j \frac{\partial h}{\partial x_{ij}^{\ell}} \frac{\partial x_{ij}^{\ell}}{\partial w_{mn}^{\ell}} \\
&= \sum_i \sum_j \delta_{ij}^{\ell} \frac{\partial}{\partial w_{mn}^{\ell}} \sum_k \sum_l x_{i-k,j-l}^{\ell-1} w_{kl}^{\ell} \\
&= \sum_i \sum_j \delta_{ij}^{\ell} x_{i-m,j-n}^{\ell-1}.
\end{aligned}$$

We can compute this with a convolution,

$$\frac{\partial h}{\partial \boldsymbol{W}^{\ell}} = \boldsymbol{\Delta}^{\ell} * \boldsymbol{X}^{\ell-1}.$$

In conclusion, we can compute both the forward and backward pass using only convolutional operations.

Non-linearities and pooling. We interleave convolutional layers with non-linearities and pooling layers, which downsample the input,

$$\boldsymbol{I}'[i,j] = \max\{\boldsymbol{I}[i+k, j+\ell] \mid k, \ell \in [0, r)\},$$

where r is the window size. In general, convolutional networks have a pyramid structure, where the data gets iteratively downsampled.

Let the ℓ-th layer be a max-pooling layer and $(i^\star, j^\star)$ the indices of the maximum values in the forward pass, then we can compute the gradient by

$$\frac{\partial x_{ij}^{\ell}}{\partial x_{mn}^{\ell-1}} = \mathbb{1}\{(m,n) = (i^\star, j^\star)\}.$$

Note that max pooling has no parameters, so the backward pass only propagates the gradient to further layers.

Channels. We can extend the convolutional layer to multiple channels as follows,

$$(\mathbf{X} * \mathbf{W})[c, i, j] \doteq \sum_{r=1}^{C_{\text{in}}} \sum_{k=-\infty}^{\infty} \sum_{\ell=-\infty}^{\infty} w[c, r, k, \ell] x[r, i-k, j-\ell].$$

Here, $\mathbf{W} \in \mathbb{R}^{C_{\text{in}} \times C_{\text{out}} \times K \times K}$ is a tensor of parameters, where K is the kernel size. In general, such layers are used to sequentially increase the channel dimension, while the max pool layers are used to decrease the spatial dimension. In the end, a classification or regression network produces a $C \times 1 \times 1$ image for some dimensionality C, which is cast to be a C-dimensional feature vector. This is then given to a fully connected network to make the final prediction.

5 Recurrent neural networks

Typically, networks cannot process variable-sized data, such as sequences. Further, convolutional networks constrain the range of the dependencies between timesteps of a sequence, and linear layers would explode in the number of parameters. RNNs (***R**ecurrent **N**eural **N**etworks*) process the data sequentially, where each timestep depends on its entire history. Let $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T$ denote the observed input sequence, RNNs compute the sequence of activations recursively with shared parameters across timesteps,

$$\boldsymbol{z}_t \doteq F[\boldsymbol{\theta}](\boldsymbol{z}_{t-1}, \boldsymbol{x}_t), \quad \boldsymbol{z}_0 = \mathbf{0}.$$

Dependent on the application, we can compute output variables from these activations,

$$\boldsymbol{y}_t \doteq G[\boldsymbol{\varphi}](\boldsymbol{z}_t).$$

For example, in same length sequence-to-sequence prediction, y_t will denote the output token at the t-th timestep; in autoregressive modeling, y_t predicts the next input token x_{t+1}; and in sequence classification, the final output y_T predicts the classification of the entire sequence.

RNNs can be extended by bidirectional RNNs [Schuster and Paliwal, 1997], which apply two RNNs—forward and backward. The outputs of the two RNNs are concatenated, such that every hidden state captures the full sequence.

Furthermore, stacked RNNs [Joulin and Mikolov, 2015] increases modeling power by connecting layers horizontally,

$$\boldsymbol{z}_{t,\ell} = \phi(\boldsymbol{U}_\ell \boldsymbol{z}_{t-1,\ell} + \boldsymbol{V}_\ell \boldsymbol{z}_{t,\ell-1}), \quad \boldsymbol{z}_{t,0} = \boldsymbol{x}_t.$$

Alternatively, the recurrence function F can be replaced by a deep MLP.

The simplest RNN architecture is the Elman RNN [Elman, 1990],

$$\begin{aligned} F[\boldsymbol{U}, \boldsymbol{V}](\boldsymbol{z}, \boldsymbol{x}) &= \phi(\boldsymbol{U}\boldsymbol{z} + \boldsymbol{V}\boldsymbol{x}), \quad \boldsymbol{U} \in \mathbb{R}^{m \times m}, \boldsymbol{V} \in \mathbb{R}^{m \times n} \\ G[\boldsymbol{W}](\boldsymbol{z}) &= \psi(\boldsymbol{W}\boldsymbol{z}), \quad \boldsymbol{W} \in \mathbb{R}^{q \times m}. \end{aligned}$$

However, this model has difficulties modeling large-range dependencies, as will become apparent from the gradients. Let

$$L \doteq \sum_{t=1}^{T} \ell(\hat{\boldsymbol{y}}_t, \boldsymbol{y}_t).$$

Then, we have the following gradients w.r.t. the recurrence weights,

$$\begin{aligned} \frac{\partial L}{\partial \boldsymbol{U}} &= \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{z}_t} \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{U}} \\ \frac{\partial L}{\partial \boldsymbol{V}} &= \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{z}_t} \frac{\partial \boldsymbol{z}_t}{\partial \boldsymbol{V}}. \end{aligned}$$

We can compute the gradients w.r.t. the hidden states as follows,

$$\begin{aligned}\frac{\partial L}{\partial z_t} &= \sum_{i=1}^{T} \frac{\partial \ell(\hat{y}_i, y_i)}{\partial z_t} \\ &= \sum_{i=t}^{T} \frac{\partial \ell(\hat{y}_i, y_i)}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_t} \\ &= \sum_{i=t}^{T} \frac{\partial \ell(\hat{y}_i, y_i)}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i} \frac{\partial z_i}{\partial z_t} \\ &= \sum_{i=t}^{T} \frac{\partial \ell(\hat{y}_i, y_i)}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i} \prod_{j=t+1}^{i} \frac{\partial z_j}{\partial z_{j-1}} \\ &= \sum_{i=t}^{T} \frac{\partial \ell(\hat{y}_i, y_i)}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial z_i} \prod_{j=t+1}^{i} \dot{\mathbf{\Phi}}_j \mathbf{U},\end{aligned}$$

where

$$\dot{\mathbf{\Phi}}_j \doteq \operatorname{diag}\big(\phi'(\mathbf{U} z_{j-1} + \mathbf{V} x_j)\big).$$

This gradient is only stable if

$$\left\| \frac{\partial z_j}{\partial z_{j-1}} \right\|_2 = \left\| \dot{\mathbf{\Phi}}_j \mathbf{U} \right\|_2 = 1,$$

which is almost never the case. Assuming bounded gradient norm $\|\dot{\mathbf{\Phi}}_j\|_2 \leq \alpha$—which holds for most activation functions,[6]

$$\left\| \frac{\partial z_i}{\partial z_t} \right\|_2 \leq (\alpha \|\mathbf{U}\|_2)^{i-t} = (\alpha \sigma_1(\mathbf{U}))^{i-t}.$$

[6] *E.g.*, $\sigma'(z) \leq 1/4$.

So, the gradient will vanish if $\sigma_1(\mathbf{U}) \leq 1/\alpha$. An analogous argument can be made for exploding gradients, resulting in numerical instabilities. This is a big problem.

5.1 *Gated memory*

Long-range dependencies are hard to memorize for the Elman RNN due to the instability of the gradient. LSTM (***L**ong **S**hort-**T**erm **M**emory*) [Schmidhuber et al., 1997] and GRU (***G**ated **R**ecurrent **U**nit*) [Cho et al., 2014] avoid short-term fluctuations by more directly controlling when memory is kept and when it is overwritten. It does so by making use of gating,

$$\mathbf{z} = \boldsymbol{\sigma} \odot \mathbf{z}, \quad \boldsymbol{\sigma} \in (0,1)^m, \mathbf{z} \in \mathbb{R}^m.$$

When $\sigma_i \to 0$, z_i is forgotten and when $\sigma_i \to 1$, z_i is preserved. By combining gates in smart ways, learning involves understanding what new information is relevant and trading off its relevance with the stored long-term information. The LSTM works as follows,

$$\begin{aligned} z_t &= \sigma(\mathbf{F}\tilde{x}_t) \odot z_{t-1} + \sigma(\mathbf{G}\tilde{x}_t) \odot \tanh(\mathbf{V}\tilde{x}_t), \quad \tilde{x}_t = [\zeta_{t-1}, x_t] \\ \zeta_t &= \sigma(\mathbf{H}\tilde{x}_t) \odot \tanh(\mathbf{U} z_t). \end{aligned}$$

Here, z_t is called the cell state and ζ_t is the hidden state. This mechanism has the following components,

- $\sigma(\boldsymbol{F}\tilde{\boldsymbol{x}}_t)$ is the forget gate and computes what information should be discarded from the previous cell state;
- $\tanh(\boldsymbol{V}\tilde{\boldsymbol{x}}_t)$ is the input gate and computes new information;
- $\sigma(\boldsymbol{G}\tilde{\boldsymbol{x}}_t)$ is the gate gate and computes what of the new information should be stored;
- $\sigma(\boldsymbol{H}\tilde{\boldsymbol{x}}_t)$ is the output gate and has the role of determining what information from the cell state should be put in the hidden state;
- $\tanh(\boldsymbol{U}\boldsymbol{z}_t)$ computes what information should be given to the hidden state.

One can show that with this mechanism, the gradient cannot vanish because the gradients can take a short cut through the cell state.

The GRU simplifies the above by combining the forget and input gates as a convex combination,

$$\boldsymbol{z}_t = \boldsymbol{\sigma} \odot \boldsymbol{z}_{t-1} + (1 - \boldsymbol{\sigma}) \odot \tilde{\boldsymbol{z}}_t, \quad \boldsymbol{\sigma} = \sigma(\boldsymbol{G}\tilde{\boldsymbol{x}}_t), \quad \tilde{\boldsymbol{x}}_t = [\boldsymbol{z}_{t-1}, \boldsymbol{x}_t].$$

However, the computation of new storage remains complex,

$$\tilde{\boldsymbol{z}}_t = \tanh(\boldsymbol{V}[\boldsymbol{\zeta}_t \odot \boldsymbol{z}_{t-1}, \boldsymbol{x}_t]), \quad \boldsymbol{\zeta}_t = \sigma(\boldsymbol{H}[\boldsymbol{z}_{t-1}, \boldsymbol{x}_t]).$$

In practice, $\boldsymbol{\zeta}_t$ can be computed implicitly without any additional recursion. The advantage of this over LSTM is that it only has 3 weight matrices, instead of 5.

5.2 *Linear recurrent models*

The LRU (***L****inear* ***R****ecurrent* ***M****odel*) [Feng et al., 2024] further simplifies the GRU recurrence function to be linear, such that it can exploit fast parallel sequence processing for training,

$$\boldsymbol{z}_t = \boldsymbol{\sigma} \odot \boldsymbol{z}_{t-1} + (1 - \boldsymbol{\sigma}) \odot \tilde{\boldsymbol{z}}_t, \quad \boldsymbol{\sigma} = \sigma(\boldsymbol{G}\boldsymbol{x}_t), \quad \tilde{\boldsymbol{z}}_t = \boldsymbol{V}\boldsymbol{x}_t.$$

Now we can use prefix scan parallelism, which allows for an $\mathcal{O}(\log T)$ runtime during training, instead of $\mathcal{O}(T)$. This might bridge the gap to the performance of transformers.[7]

[7] In general, transformers perform better than RNNs because the training of transformers can be parallelized. On the other hand, RNNs could be preferable, because transformers have runtime quadratic in the context length at every step, whereas RNNs have already encoded the full history in the hidden state. As a result, RNNs are faster during inference.

We will now look at how we can ensure that gradients do not explode in linear systems [Orvieto et al., 2023]. The LRU hidden state evolution is a discrete time linear system,

$$\boldsymbol{z}_{t+1} = \boldsymbol{A}\boldsymbol{z}_t + \boldsymbol{B}\boldsymbol{x}_t, \quad \boldsymbol{A} \in \mathbb{R}^{m \times m}, \boldsymbol{B} \in \mathbb{R}^{m \times n}.$$

Let the following be the diagonalization of $\boldsymbol{A}$ over the complex numbers,[8]

[8] Most matrices can be diagonalized over the complex numbers.

$$\boldsymbol{A} = \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{-1}, \quad \boldsymbol{\Lambda} = \operatorname{diag}(\lambda_1, \dots, \lambda_m), \quad \lambda_i \in \mathbb{C}.$$

We can then perform a change of basis,

$$\boldsymbol{\zeta}_{t+1} = \boldsymbol{\Lambda}\boldsymbol{\zeta}_t + \boldsymbol{C}\boldsymbol{x}_t, \quad \boldsymbol{\zeta}_t = \boldsymbol{P}^{-1}\boldsymbol{z}_t, \boldsymbol{C} = \boldsymbol{P}^{-1}\boldsymbol{B}.$$

The stability of this linear system requires the modulus of the eigenvalues to be bounded,

$$\max_j |\lambda_j| \leq 1, \quad |a + bi| \doteq \sqrt{a^2 + b^2}.$$

One can represent any complex number in polar coordinates form via modulus r and phase ϕ,

$$z = r(\cos(\phi) + \sin(\phi)i), \quad r = |z| \geq 0, \phi \in [0, 2\pi).$$

Thus, we want to parametrize λ_i in such a way that their moduli can only exist within $(0, 1)$. We can do this by parametrizing λ_i with two numbers $\nu_i, \phi_i \in \mathbb{R}$ in the following way,

$$\begin{aligned}\lambda_i &= \exp(-\exp(\nu_i) + \phi_i i) \\ &= \exp(-\exp(\nu_i)) \exp(\phi_i i) \\ &= \exp(-\exp(\nu_i))(\cos(\phi_i) + \sin(\phi_i)i).\end{aligned}$$

$\exp(\theta i) = \cos(\theta) + \sin(\theta)i.$

So, we have $r_i = \exp(-\exp(\nu_i)) \in (0, 1)$. At initialization, we can then sample

$$\phi_i \sim \text{Unif}([0, 2\pi]), \quad r_i \sim \text{Unif}(I), \quad I \subseteq [0, 1].$$

We can compute $\nu_i = \log(-\log r_i)$. When we update the parameters ν_i and ϕ_i, the modulus will always remain less than 1.

The advantage of such a simple recurrence unit is that it provides a clean understanding of long range and short range dependencies, there is no requirement for mixing of channels, and parallelization during training. Furthermore, we do not lose any representational power, because we can move all power to the output map. The resulting model is provably universal as a sequence-to-sequence map [Feng et al., 2024], because we can put all the representational power into the output map,

$$\boldsymbol{y}_t = \text{MLP}(\text{Re}(\boldsymbol{G}\boldsymbol{\zeta}_t)), \quad \boldsymbol{G} \in \mathbb{C}^{k \times m},$$

where Re is a real projection.

5.3 *Sequence learning*

In sequence learning, we want to generate a sequence step-by-step, given another sequence. This induces the following probability distribution,

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

Sequence-to-sequence mapping [Sutskever, 2014] is generally done by mapping the input sequence to a latent representation,

$$x_1, \ldots, x_m \mapsto \boldsymbol{\zeta},$$

which can be computed by an encoder RNN. Then, at every timestep, we compute a latent representation of everything generated until now, which can be computed by a decoder RNN with $z_0 = \boldsymbol{\zeta}$,

$$\boldsymbol{\zeta}, y_1, \ldots, y_{t-1} \mapsto z_{t-1}.$$

These are then combined to compute a distribution over next tokens,

$$\boldsymbol{z}_{t-1} \mapsto \boldsymbol{\mu}_t, \quad y_t \sim p(\boldsymbol{\mu}_t).$$

Usually, $\boldsymbol{\mu}_t$ is a categorical distribution over tokens, computed by a softmax.

The problem with this approach is that $\boldsymbol{\zeta}$ will be a lossily compressed version of the input sequence. We would want the decoder to be able to look back at the input sequence while generating the output sequence. Bahdanau [2014] solved this by introducing a cross-attention mechanism into this framework, where attention is used on top of the RNN encoder,

$$a_{ij} = \text{softmax}_j(\text{MLP}(\boldsymbol{z}_{i-1}, \boldsymbol{\zeta}_j)).$$

Then, for timestep t we replace $\boldsymbol{\zeta}$ with a context vector,

$$\boldsymbol{c}_t = \sum_{i=1}^{T} a_{ti} \boldsymbol{\zeta}_i.$$

This mechanism makes intuitive sense, because it allows for alignment between source and target sequence.

6 Transformers

6.1 Self-attention

Let $\boldsymbol{X} \in \mathbb{R}^{T\times d}$ denote the input embeddings and $\Xi \in \mathbb{R}^{T\times d_v}$ the output embeddings. The problem with $\boldsymbol{X}$ is that the embeddings are non-contextual—each embedding has no information about its neighbors. Self-attention aims to contextualize the embeddings in Ξ.

It does so by computing queries, keys, and values by linear projections of the input,

$$\boldsymbol{Q} = \boldsymbol{X}\boldsymbol{W}_Q, \quad \boldsymbol{K} = \boldsymbol{X}\boldsymbol{W}_K, \quad \boldsymbol{V} = \boldsymbol{X}\boldsymbol{W}_V,$$

where $\boldsymbol{W}_Q, \boldsymbol{W}_K \in \mathbb{R}^{d\times d_k}$ and $\boldsymbol{W}_V \in \mathbb{R}^{d\times d_v}$. Intuitively, for each timestep, the queries represent what information is missing, the keys represent the information that is offered, and the values are the actual information.

The attention mechanism is computed as follows,

$$\boldsymbol{A} = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^\top}{\sqrt{d_k}}\right), \quad \Xi = \boldsymbol{A}\boldsymbol{V}.$$

Softmax is performed row-wise. The division by $\sqrt{d_k}$ is necessary because $\text{Var}[\boldsymbol{x}\cdot\boldsymbol{y}] = d$, where $\boldsymbol{x}, \boldsymbol{y} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_d)$. We want to recover unit variance.

Here, $\boldsymbol{A} \in \mathbb{R}^{T\times T}$ is the attention matrix—$\boldsymbol{a}_i$ is a convex combination that tells us how much attention the i-th timestep pays to each other timestep. The rank of this matrix is bounded by d_k.

The contextualized outputs are convex combinations of values,

$$\boldsymbol{\xi}_i = \sum_{t=1}^{T} \text{softmax}_t(\boldsymbol{a}_i)\boldsymbol{v}_i, \quad a_{it} \propto \boldsymbol{q}_i^\top \boldsymbol{k}_t.$$

This makes intuitive sense, because the weight of the t-th timestep for timestep i depends on the alignment between $\boldsymbol{q}_i$ and $\boldsymbol{k}_t$. Furthermore, $\boldsymbol{\xi}_i$ depends only on its corresponding query and all other key-value pairs. In a sense, the attention mechanism is a soft-dictionary lookup.

MHSA (***M**ulti-**H**eaded **S**elf-**A**ttention*) computes multiple attention mechanism in parallel—see Figure 6.1. Let h be the number of heads, then we first compute h queries, keys, and values for each input token.[9] Then, we apply attention h times using these representations and concatenate the outputs into a single vector. Lastly, we perform a linear layer to combine the outputs of the heads.

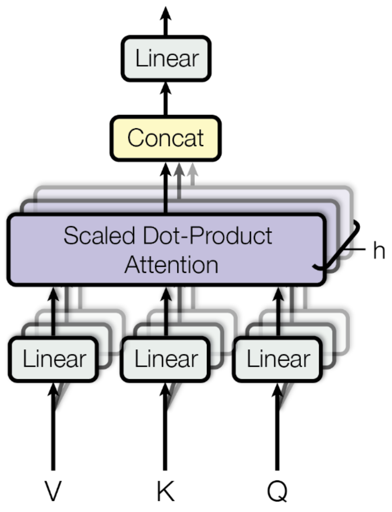


Figure 6.1. Multi-headed self-attention [Vaswani, 2017].

[9] In practice, we use three—not $3\cdot h$—linear layers for the query, key, and value representations, where the output is $h\cdot d_k$-dimensional. We can chunk this output to get the corresponding representations for each head. This makes PyTorch—or any other library—compute the heads in parallel.

6.2 Cross-attention

A cross-attention layer takes two sequences as inputs,

$$\boldsymbol{A} \in \mathbb{R}^{T_a\times d_a}, \quad \boldsymbol{B} \in \mathbb{R}^{T_b\times d_b}.$$

Then, it computes the queries from $\boldsymbol{A}$ and the keys and values from $\boldsymbol{B}$,

$$\boldsymbol{Q} = \boldsymbol{A}\boldsymbol{W}_Q, \quad \boldsymbol{K} = \boldsymbol{B}\boldsymbol{W}_K, \quad \boldsymbol{V} = \boldsymbol{B}\boldsymbol{W}_V,$$

where $\boldsymbol{W}_Q \in \mathbb{R}^{d_a \times d_k}$, $\boldsymbol{W}_K \in \mathbb{R}^{d_b \times d_k}$, and $\boldsymbol{W}_V \in \mathbb{R}^{d_b \times d_v}$. Then, we can apply (multi-headed) attention to these representations. This is an effective way of giving additional sequence data $\boldsymbol{B}$ to a sequence $\boldsymbol{A}$.

6.3 *Positional encoding*

The attention mechanism is permutation equivariant, which means that the order of input tokens does not influence the output. So, we need a way of reintroducing the sequence structure to this mechanism. We do this by defining a positional encoding matrix $\boldsymbol{P} \in \mathbb{R}^{T \times d}$ and adding it to the input sequence, $\boldsymbol{X} + \boldsymbol{P}$. A common way of defining this matrix is as follows,

$$p_{tk} = \begin{cases} \sin(t\omega_k) & k \mod 2 = 0 \\ \cos(t\omega_k) & k \mod 2 = 1, \end{cases} \qquad \omega_k \doteq C^{k/d}.$$

A heatmap representation of this matrix can be seen in Figure 6.2.

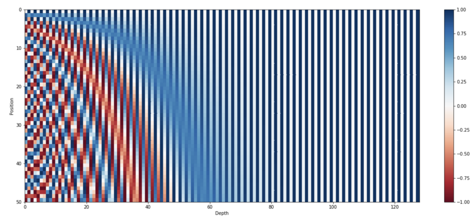


Figure 6.2. Positional encoding matrix, represented as a heatmap.

6.4 *Machine translation*

The transformer [Vaswani, 2017] was the first architecture to show that attention can be used effectively in machine learning. Vaswani [2017] designed an encoder and an autoregressive decoder for machine translation—see Figure 6.3. The encoder works by applying an MHSA layer and a pointwise MLP layer N times in an alternating fashion. In addition, it also employs residual connections [He et al., 2016] and layer normalization [Lei Ba et al., 2016]. These are essential for effectively backpropagating gradients and ensuring stability. Furthermore, it also makes use of positional encoding to preserve order information. Let $\boldsymbol{X} \in \mathbb{R}^{T \times d}$ denote the input of the encoder and $\Xi \in \mathbb{R}^{T \times d}$ its output.

Furthermore, the decoder works in an autoregressive manner, which means that it computes the output tokens one-by-one. The decoder first receives the history of previously generated tokens $\boldsymbol{Y}_{1:t-1}$ and contextualizes it using an MHSA layer. Let $\mathbf{Y}_{1:t-1}$ denote the output of the MHSA layer. Then, a multi-headed cross-attention layer receives Ξ as input and aligns $\mathbf{Y}_{1:t-1}$ with it. Lastly, a pointwise MLP is applied. It performs these steps N times. Again, the decoder makes use of residual connections and layer normalization to ensure stability of the gradient and output.

The advantage of this architecture is that—unlike RNNs—we do not need to memorize tokens in the hidden state, because we can look back at the full sequence at every step. However, this has the disadvantage that we need to look at the full sequence at every step, instead of having all information encoded in a precomputed hidden state. Furthermore, transformers allow for easy scaling up by simply increasing the number of heads, hidden dimensionality, or the number of encoders/decoders.

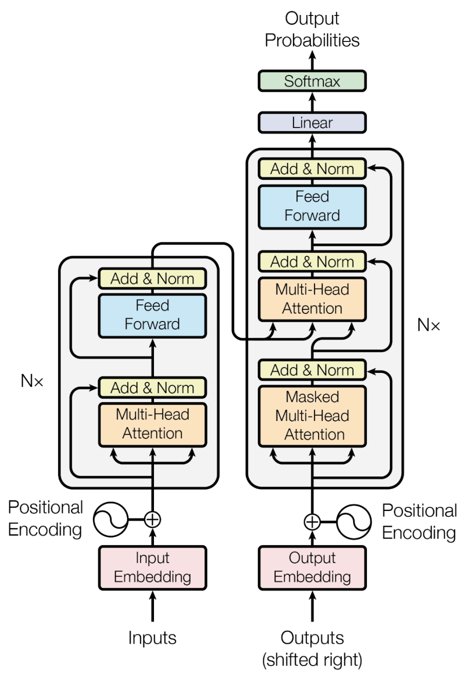


Figure 6.3. Architecture of the transformer [Vaswani, 2017]. The left side is called the encoder, which encodes the input sentence. The right side is called the decoder, which uses cross-attention to incorporate information from the source sequence in the prediction of the target sequence. The model works in an autoregressive manner, which means that the next token is predicted from the history until an end-of-sentence token is predicted.

6.5 BERT

BERT (***B**idirectional **E**ncoder **R**epresentations from **T**ransformers*) [Devlin, 2018] is a transformer-based pretrained LM that can be used for finetuning on downstream natural language processing tasks. BERT first tokenizes its input sequence using WordPiece tokenization [Wu, 2016] and prepends it with a `[CLS]` token. Further, it makes use of the encoder blocks from the transformer architecture to contextualize its input tokens. When the weights of these encoders are pretrained, we can place additional layers on top of the encoders that operate on BERT's contextualized tokens. We then finetune the weights of the full model on the specific task we are interested in.

BERT's pre-training consists of two stages,

1. *Predicting masked out tokens using its left and right context as input.*[10,11] The task is performed by passing a masked out text to BERT to retrieve its representations. Then, the representation embedding of the masked tokens are passed to a model that predicts which token was originally there. This was previously not easy to do with RNNs, because they can only process sequences left-to-right or right-to-left;

2. *Binary next sentence classification, where the model must classify two sentences as being consecutive or not.* Here, the two sentences are separated by a `[SEP]`-token as input to BERT. In order to make classification possible, the `[CLS]`-token is appended to the input tokens and its representation is used for the final prediction network.

[10] This is also known as the Cloze test [Taylor, 1953]. Cloze tests require the ability to understand the context and vocabulary in order to identify the correct language or part of speech that belongs in the deleted passages.

[11] BERT masks 15% of tokens, of which 80% is replaced by `[MASK]`, 10% is replaced by a random token, and 10% is left unchanged.

The first stage trains BERT's understanding of language, whereas the second stage enables BERT to infer relationships between sentences, which is important for tasks like question-answering.

Lastly, we can finetune the parameters of BERT with a small ground truth dataset to a large variety of tasks. *E.g.*,

- *Question answering*, where a question and a context passage is provided with a `[SEP]`-token separating them. Each token is passed to start and end token classifiers, which predict how likely each token is to be the start and end of the answer. Using these classifiers, we can extract the answer from the passage;

- *Part-of-speech tagging*, where each token embedding is passed to a classification model, which outputs a distribution over part-of-speech tags;

- *Sentiment classification*, where the representation of the `[CLS]`-token is passed to a binary classification model that predicts whether the sentence is positive or negative.

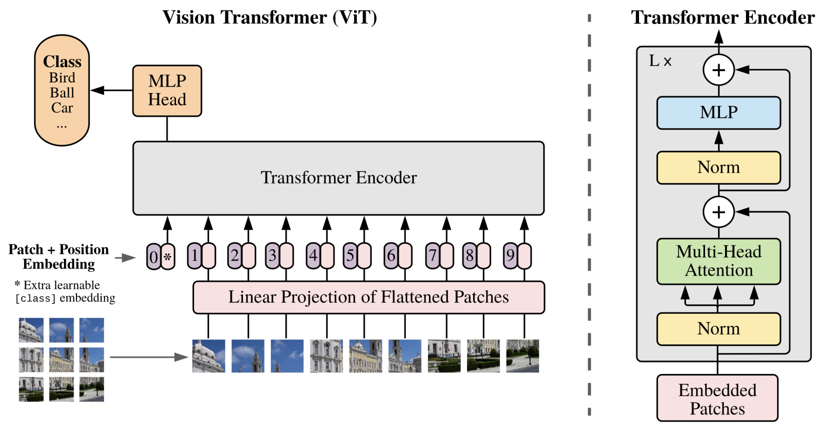


Figure 6.4. Architecture of the vision transformer [Dosovitskiy, 2020].

6.6 *Vision transformer*

ViT (***V**ision* ***T**ransformer*) [Dosovitskiy, 2020] adapts the transformer architecture to images by treating patches of an input image as its tokens—see Figure 6.4. Dosovitskiy [2020] showed the effectiveness of this approach by adapting the transformer architecture to an image classification task. ViT computes the input tokens by vectorizing 16×16 patches of the input image and linearly projecting them to a token space. Further, a `[CLS]`-token is prepended to the sequence of tokens. Then, ViT employs the encoder architecture of the transformer to contextualize its input representations. Finally, the contextualized embedding of the `[CLS]`-token is passed to a classification network that predicts the class of the input image.

A possible reason for this model's effectiveness is that this architecture carries less inductive bias than CNN-based models. In general, this seems to be beneficial for very large datasets.

7 Geometric deep learning

GDL (***G**eometric **D**eep **L**earning*) is involved with modeling neural networks that satisfy invariances by design. Assume we have a set of feature vectors $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M\} \subset R$ over which we want to realize a function $f : \mathcal{P}(R) \to \mathcal{Y}$. Naively, we could concatenate the set into a single feature vector and apply a standard multi-layer perceptron,

$$\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M\} \mapsto [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M].$$

However, this has two problems: (1) M is not fixed, so the inputs have variable length and (2) the order in which we concatenate the feature vectors is arbitrary. We need to model an architecture that can take a variable-length input and does not depend on the ordering of the feature vectors.

7.1 Invariance and equivariance in neural networks

In order to formally design such functions, we need the following two definitions.

Definition 7.1 (Order-invariance). A function f that takes an arbitrary number of inputs is order-invariant if and only if

$$f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M) = f(\boldsymbol{x}_{\pi_1}, \ldots, \boldsymbol{x}_{\pi_M}), \quad \forall \boldsymbol{\pi} \in \Pi(M),$$

where $\boldsymbol{\pi}$ is a permutation. Or, in matrix notation,

$$f(\boldsymbol{X}) = f(\boldsymbol{P}\boldsymbol{X}),$$

where $\boldsymbol{X} \in \mathbb{R}^{M \times d}$ contains the feature vectors and $\boldsymbol{P} \in \mathbb{R}^{M \times M}$ is a permutation matrix.

Definition 7.2 (Equivariance). A function f that takes an arbitrary number of inputs and has outputs of the same length is equivariant if and only if

$$\begin{aligned} & f(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_M) = (y_1, \ldots, y_M) \\ \implies & f(\boldsymbol{x}_{\pi_1}, \ldots, \boldsymbol{x}_{\pi_M}) = (y_{\pi_1}, \ldots, y_{\pi_M}), \quad \forall \boldsymbol{\pi} \in \Pi(M). \end{aligned}$$

Or, in matrix notation,

$$f(\boldsymbol{X}) = \boldsymbol{P} f(\boldsymbol{P}\boldsymbol{X}),$$

where $\boldsymbol{X} \in \mathbb{R}^{M \times d}$ contains the feature vectors and $\boldsymbol{P} \in \mathbb{R}^{M \times M}$ is a permutation matrix.

The question thus becomes how we can design model architectures that are order-invariant or equivariant.

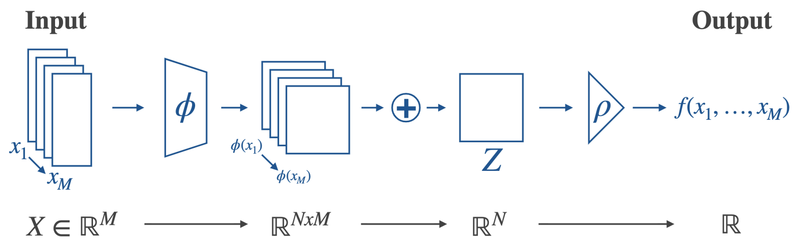


Figure 7.1. Model structure of Deep Sets [Zaheer et al., 2017].

7.2 *Deep sets*

Let $\phi : R \to \mathbb{R}^d$ be a pointwise feature extractor neural network. The Deep Sets architecture [Zaheer et al., 2017] obtains an order-invariant representation of the input set by summing their features up, because the sum operation enforces order-invariance,

$$\sum_{m=1}^{M} \phi(x_m).$$

We can then use this representation with any type of neural network $\rho : \mathbb{R}^d \to \mathcal{Y}$ to get an order-invariant model,

$$f(x_1, \ldots, x_M) = \rho\left(\sum_{m=1}^{M} \phi(x_m)\right).$$

Similarly, we can use other aggregation functions, such as the maximum or minimum.

Once we have an order-invariant feature extractor, we can easily turn it into an equivariant map by additionally providing x_m to $\rho : R \times \mathbb{R}^d \to \mathcal{Y}$ and applying ρ pointwise,

$$f(x_1, \ldots, x_M) = \left(\rho\left(x_1, \sum_{m=1}^{M} \phi(x_m)\right), \ldots, \rho\left(x_M, \sum_{m=1}^{M} \phi(x_m)\right)\right).$$

This architecture is universal for a fixed d, but it requires mappings that are highly discontinuous as $M \to \infty$, which makes its usefulness limited in practice [Wagstaff et al., 2019]. More realistic mappings require $d \geq M$.

7.3 *PointNet*

The PointNet model [Qi et al., 2017] is a specific use case of the Deep Sets architecture. The model receives a set of three-dimensional points as input—a point cloud—and must classify the object or segment its parts. The former use case requires an order-invariant model, while the latter requires an equivariant model, because the order that the points are presented in does not carry meaning.

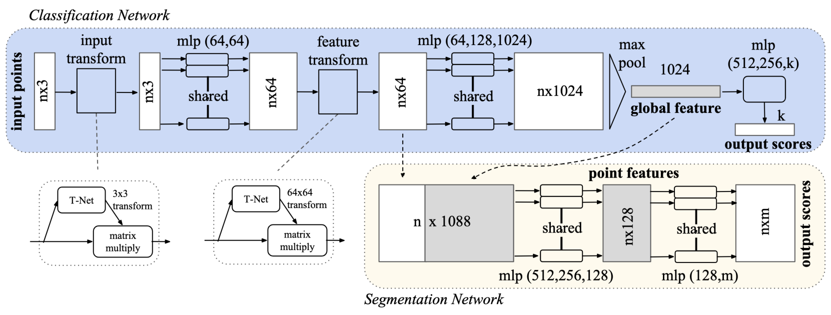


Figure 7.2. Model architecture of PointNet [Qi et al., 2017].

This model employs T-net blocks, which apply rigid transformations to the input point cloud, which is permutation invariant. These are applied alternatingly with multi-layer perceptrons to form a permutation invariant feature extractor ϕ. ϕ applies two stages of this, which result in a 64-dimensional intermediate feature vector and a 1024-dimensional final feature vector. The features are aggregated by a max-pool operator to obtain an order-invariant 1024-dimensional global feature vector.

For object classification, ρ is implemented as a multi-layer perceptron with a softmax head that takes the global feature vector as input. For object segmentation, ρ concatenates the intermediate local 64-dimensional feature vector with the global 1024-dimensional vector, which is given to a multi-layer perceptron with a softmax head.

7.4 *Graph neural networks*

Definition 7.3 (Graph)**.** An undirected graph $G = (V, E)$ consists of vertices $V = \{v_1, \dots, v_M\}$ and edges $E = \{e_1, \dots, e_K\} \subseteq \{\{v, v'\} \mid v, v' \in V\}$.

In GNNs (*Graph Neural Networks*), we associate a feature vector $\boldsymbol{x}_m \in \mathbb{R}^d$ with each node $v_m \in V$. Let $\boldsymbol{X} \in \mathbb{R}^{M \times d}$ contain all vertex feature vectors and $\boldsymbol{A} \in \mathbb{R}^{M \times M}$ be the adjacency matrix, where

$$a_{ij} = \begin{cases} 1 & \{v_i, v_j\} \in E \\ 0 & \{v_i, v_j\} \notin E. \end{cases}$$

Definition 7.4. A function f on a graph with adjacency matrix $\boldsymbol{A}$ is order-invariant if and only if

$$f(\boldsymbol{X}, \boldsymbol{A}) = f(\boldsymbol{P}\boldsymbol{X}, \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\top), \quad \forall \boldsymbol{P} \in \Pi(M).$$

Definition 7.5. A function f on a graph with adjacency matrix $\boldsymbol{A}$ is equivariant if and only if

$$f(\boldsymbol{X}, \boldsymbol{A}) = \boldsymbol{P} f(\boldsymbol{P}\boldsymbol{X}, \boldsymbol{P}\boldsymbol{A}\boldsymbol{P}^\top), \quad \forall \boldsymbol{P} \in \Pi(M).$$

We now want to design a model on graphs that is in- or equivariant. A common way to achieve this is by parametrizing a local function that only depends on the neighbors of each vertex. Let $\boldsymbol{X}_m \doteq \{\{\boldsymbol{x}_n \mid \{v_n, v_m\} \in E\}\}$, which denotes the multiset of feature vectors of the neighbors of v_m. We then parametrize a feature function ϕ that takes $\boldsymbol{x}_m$ and $\boldsymbol{X}_m$ as input. (As a consequence, any pair of isomorphic graphs result in the same feature representations.)[12] This function must also be order-invariant to the neighbors, so we need to additionally aggregate the neighbor feature vectors, which are processed by a separate network ψ,

[12] Two graphs are isomorphic if the edges are preserved under a bijection of the vertices.

$$\phi(\boldsymbol{x}_m, \boldsymbol{X}_m) = \phi\left(\boldsymbol{x}_m, \bigoplus_{\boldsymbol{x} \in \boldsymbol{X}_m} \psi(\boldsymbol{x})\right),$$

where $\bigoplus$ is an invariant aggregation function. This is sometimes called a message-passing scheme in the sense that a vertex receives messages from its neighbors via a messaging function ψ and uses an update function ϕ to update its representation.

Effectively, we are constructing an equivariant function that makes additional use of provided information in the form of a graph. We may want to use this to do equivariant node classification or invariant graph classification.

Coupling matrix. In GCNs (***G**raph **C**onvolutional **N**etworks*), the aggregation over local neighborhoods is performed with a fixed set of weights, known as the coupling matrix,

$$\bar{\boldsymbol{A}} \doteq \boldsymbol{D}^{-1/2}(\boldsymbol{A} + \boldsymbol{I})\boldsymbol{D}^{-1/2}, \quad \boldsymbol{D} = \operatorname{diag}(\boldsymbol{d}), \quad d_m = 1 + \sum_{n=1}^{M} a_{nm}.$$

We multiply by $\boldsymbol{D}^{-1/2}$ on both sides, and not $\boldsymbol{D}^{-1}$ on one side, because we want both the rows and columns to be normalized.

Here, $\boldsymbol{D}$ is the degree matrix and $\bar{\boldsymbol{A}}$ is a normalized version of $\boldsymbol{A}$ with self-loops as a result. As such, if we compute $\boldsymbol{Y} = \bar{\boldsymbol{A}}\boldsymbol{X}$, where $\boldsymbol{X} \in \mathbb{R}^{M \times d}$ contains the features, then $\boldsymbol{y}_i$ is an average of the features of the neighbors and node i itself.

Furthermore, we introduce learnable parameters $\boldsymbol{W}$ that linearly transforms the vertex feature vectors. Let σ be an activation function, then the following is one step of propagation in GCNs,

$$\Xi = \sigma(\bar{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}), \quad \boldsymbol{W} \in \mathbb{R}^{d \times d'}.$$

Note that $\bar{\boldsymbol{A}}$ operates on the node-edge structure and $\boldsymbol{W}$ operates in the feature space. This layer can be stacked as in normal neural networks to introduce depth. A simple two-layer GCN for node classification looks as follows,

$$\boldsymbol{Y} = \operatorname{softmax}\big(\bar{\boldsymbol{A}}(\bar{\boldsymbol{A}}\boldsymbol{X}\boldsymbol{W}_0)_+ \boldsymbol{W}_1\big).$$

As the depth increases, it is important to note that $\|\bar{\boldsymbol{A}}\|_2 \leq 1$, which ensures that activations do not grow out out of control.

A limitation of GCNs is that it requires a depth equal to the diameter of the graph to exchange information between all nodes. However, the problem with very deep GCNs is that feature vectors between nodes become indistinguishable due to the smoothing that $\bar{\boldsymbol{A}}$ introduces [Chen et al., 2020]. Further, there is a bottleneck effect of how much information can be stored in fixed-size representations [Alon and Yahav, 2020]. There is no canonical solution to these problems.

Attention. As we have already seen in transformers, the attention mechanism is permutation equivariant w.r.t. the sequence order.[13] GATs (***G****raph* ***A****ttention* ***N****etworks*) [Veličković et al., 2017] define the coupling matrix $\boldsymbol{Q}$ using attention,

[13] This is due to the softmax operator being equivariant.

$$q_{ij} = \operatorname{softmax}_j \Big(\rho\Big(\boldsymbol{u}^\top [\boldsymbol{V}\boldsymbol{x}_i, \boldsymbol{V}\boldsymbol{x}_j, \boldsymbol{x}_{ij}]\Big)\Big), \quad \sum_{j=1}^{M} a_{ij} q_{ij} = 1, \forall i \in [M].$$

where $\boldsymbol{V}$ projects the node features and $\boldsymbol{x}_{ij}$ is a feature vector representing the edge between v_i and v_j. These are concatenated and projected to a learnable direction $\boldsymbol{u}$. The advantage of this method is that the aggregation coefficients are now learnable, instead of fixed equal weights.

Despite having a higher degree of adaptivity, a GAT is still a message-passing algorithm. Such models have inherent limitations in the type of graphs that they can distinguish. The Weisfeiler-Lehman graph isomorphism test computes whether there exists an isomorphism between two graphs. Morris et al. [2019] show that many message-passing algorithms—such as GCNs and GATs—cannot distinguish graphs beyond the WL-test. Hence, there is a clear need for higher order GNNs.

7.5 *Spectral graph theory*

Definition 7.6 (Laplacian operator)**.** The Laplacian is defined as

$$\Delta f \doteq \sum_{i=1}^{d} \frac{\partial^2 f}{\partial x_i^2}, \quad f : \mathbb{R}^d \to \mathbb{R}.$$

Intuitively, the Laplacian measures the local deviation from the mean of f in vanishingly small neighborhoods.

Definition 7.7 (Graph Laplacian)**.** The graph Laplacian is defined as

$$L \doteq D - A,$$

where $D \in \mathbb{R}^{M \times M}$ is the (diagonal) degree matrix and $A \in \mathbb{R}^{M \times M}$ is the adjacency matrix. Alternatively, the symmetric degree-normalized Laplacian can be used,

$$\tilde{L} \doteq I - D^{-1/2} A D^{-1/2} = D^{-1/2}(D - A)D^{-1/2}.$$

They are both positive semidefinite.

One can generalize the Fourier transform to graphs by making use of the diagonalization of the Laplacian,

$$L = U \Lambda U^\top.$$

The columns of the orthogonal matrix U can be seen as the graph Fourier basis and the eigenvalues as frequencies. By making use of Lemma 4.6, a graph convolution can be defined as pointwise multiplication in the Fourier domain,

$$X * Y = U\Big(\Big(U^\top X\Big) \odot \Big(U^\top Y\Big)\Big).$$

The learned convolution operation from one-dimensional signals is generalized as follows,

$$G_\theta(L)x = U G_\theta(\Lambda) U^\top X.$$

The problem with this approach is that computing the eigendecomposition of L is done in $\mathcal{O}(M^3)$. A trick to circumvent this problem is to use polynomial kernels,

$$U\left(\sum_{k=0}^{K} \alpha_k \Lambda^k\right) U^\top X = \sum_{k=0}^{K} \alpha_k L^k X.$$

Here, the polynomial order K defines the size of the neighborhood, *i.e.*, the kernel size. The parameters of this model are $\alpha \in \mathbb{R}^K$, so the number of parameters—and hence the expressivity—of this layer is much smaller than in traditional one-, two-, or three-dimensional convolutions.

Motivated by spectral graph theory, we can define a graph-convolutional layer as

$$\xi_m = \sum_{n=1}^{M} p_{mn}(L) x_n + b_n, \quad p_{mn}(L) \doteq \sum_{k=0}^{K} \alpha_{mnk} L^k.$$

As before, K defines the "kernel size" and $\alpha \in \mathbb{R}^{M \times M \times K}$ are the parameters, which is used to compute the coefficients of the neighbors. As in traditional convolutions, this can be expressed as an affine transformation.

8 Tricks of the trade

8.1 Parameter initialization

After defining a model, we have to choose how to initialize the parameters of that model. We could initialize it by a Gaussian or a uniform distribution with a fixed variance,

$$\boldsymbol{\theta} \sim \mathcal{N}(0, \sigma^2), \quad \boldsymbol{\theta} \sim \mathrm{Unif}\left(\left[-\sqrt{3}\sigma, \sqrt{3}\sigma\right]\right).$$

The variance of the uniform distribution $\mathrm{Unif}([a,b])$ is $\frac{1}{12}(b-a)^2$.

There are many initialization schemes that aim to set the weights in a smarter way—they turn out to be crucial to the convergence of the model. Consider a linear layer with parameters $\boldsymbol{W} \in \mathbb{R}^{m \times n}$,

$$f(\boldsymbol{x}) = \boldsymbol{W}\boldsymbol{x}.$$

The following schemes depend on the number of in- and output elements to initialize $\boldsymbol{W}$,[14] generally assume that the elements of $\boldsymbol{x}$ are uncorrelated and have standard deviation γ, and take the form above with a set σ,

[14] In this case, there are n input elements and m output elements.

- LeCun initialization [LeCun et al., 2002] aims to preserve the input variance,

  $$\sigma = \frac{1}{\sqrt{n}}.$$

  Let γ be the input variance and $\mathbb{E}[\boldsymbol{x}] = \boldsymbol{0}$, then

  $$\begin{aligned}\mathrm{Var}[(\boldsymbol{W}\boldsymbol{x})_i] &= \mathbb{E}\left[\left(\boldsymbol{w}_i^\top \boldsymbol{x}\right)^2\right] \\ &= \mathbb{E}\left[\boldsymbol{w}_i^\top \boldsymbol{x}\boldsymbol{x}^\top \boldsymbol{w}_i\right] \\ &= \gamma\mathbb{E}\left[\|\boldsymbol{w}_i\|^2\right] \\ &= \gamma\sum_{j=1}^{n}\mathbb{E}\left[w_{ij}^2\right] \\ &= \gamma.\end{aligned}$$

  Thus, input variance is preserved;

- Xavier—or Glorot—initialization [Glorot and Bengio, 2010] aims to normalize the magnitude of the gradient,

  $$\sigma = \sqrt{\frac{2}{n+m}}.$$

  Intuitively, the reason for the definition of σ is that backpropagation combines an upstream n-dimensional input vector and backpropagated downstream m-dimensional output vector;

- Kaiming—or He—initialization [He et al., 2015] is designed to be used together with the ReLU activation function by observing that only half of the units are activated in expectation,

  $$\sigma = \sqrt{\frac{2}{n}};$$

- Orthogonal initialization [Saxe et al., 2013, Hu et al., 2020] does not assume the weights to be *i.i.d.*, but instead considers the weights holistically per layer. It initializes $\boldsymbol{W}$ to be an orthogonal matrix. This offers benefits to forward and backpropagation, because the eigenvalues are equal to ± 1.

8.2 Weight decay

Weight decay introduces a term to gradient descent that moves $\boldsymbol{\theta}$ toward the origin, favoring smaller weights,

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta(\boldsymbol{\nabla}\ell(\boldsymbol{\theta}_t) - \mu\boldsymbol{\theta}_t) \\ &= (1-\eta\mu)\boldsymbol{\theta}_t - \boldsymbol{\nabla}\ell(\boldsymbol{\theta}_t).\end{aligned}$$

This is equivalent to traditional gradient descent with ℓ_2-regularization,

$$\ell_\mu(\boldsymbol{\theta}) = \ell(\boldsymbol{\theta}) + \frac{\mu}{2}\|\boldsymbol{\theta}\|^2, \quad \|\boldsymbol{\theta}\|^2 = \sum_{l=1}^{L} \|\boldsymbol{W}_l\|_F^2.$$

From a Bayesian perspective, this is equivalent to the introduction of a prior for the weights to have a small absolute value and helps combat overfitting.[15] It can also be viewed as the Lagrangian of a convex program minimizing $\ell(\boldsymbol{\theta})$ with constraint $\|\boldsymbol{\theta}\| \leq \mu$.

[15] In combination with linear regression, this is called Ridge regression.

Let $\boldsymbol{\theta}^\star \in \operatorname{argmin}_{\boldsymbol{\theta}} \ell(\boldsymbol{\theta})$, it is interesting to look at how the optimum changes when we instead optimize $\ell(\boldsymbol{\theta}) + \frac{\mu}{2}\|\boldsymbol{\theta}\|^2$. To answer this, we first make a second-order Taylor approximation around the optimum,

$$\ell(\boldsymbol{\theta}) \approx \ell(\boldsymbol{\theta}^\star) + (\boldsymbol{\theta} - \boldsymbol{\theta}^\star)^\top \boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star)(\boldsymbol{\theta} - \boldsymbol{\theta}^\star).$$

The first-order term disappears because the gradient at an optimum is zero.

The gradient of the ℓ_2-regularized ℓ_μ—using the above approximation—is written as

$$\begin{aligned}\boldsymbol{\nabla}\ell_\mu(\boldsymbol{\theta}) &= \boldsymbol{\nabla}\ell(\boldsymbol{\theta}) + \mu\boldsymbol{\theta} \\ &\approx \boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star)(\boldsymbol{\theta} - \boldsymbol{\theta}^\star) + \mu\boldsymbol{\theta} \\ &= \Big(\boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star) + \mu\boldsymbol{I}\Big)\boldsymbol{\theta} - \boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star)\boldsymbol{\theta}^\star.\end{aligned}$$

A necessary property of the optimum of ℓ_μ is $\boldsymbol{\nabla}\ell_\mu(\boldsymbol{\theta}_\mu^\star) \overset{!}{=} \boldsymbol{0}$,

$$\Big(\boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star) + \mu\boldsymbol{I}\Big)\boldsymbol{\theta}_\mu^\star = \boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star)\boldsymbol{\theta}^\star.$$

Hence,

$$\boldsymbol{\theta}_\mu^\star = \Big(\boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star) + \mu\boldsymbol{I}\Big)^{-1}\boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star)\boldsymbol{\theta}^\star.$$

Let $\boldsymbol{\nabla}^2\ell(\boldsymbol{\theta}^\star) = \boldsymbol{Q}^\top\boldsymbol{\Lambda}\boldsymbol{Q}$, then

$$\begin{aligned}\boldsymbol{\theta}_\mu^\star &= \Big(\boldsymbol{Q}^\top\boldsymbol{\Lambda}\boldsymbol{Q} + \mu\boldsymbol{Q}^\top\boldsymbol{Q}\Big)^{-1}\boldsymbol{Q}^\top\boldsymbol{\Lambda}\boldsymbol{Q}\boldsymbol{\theta}^\star \\ &= \boldsymbol{Q}^\top(\boldsymbol{\Lambda} + \mu\boldsymbol{I})^{-1}\boldsymbol{Q}\boldsymbol{Q}^\top\boldsymbol{\Lambda}\boldsymbol{Q}\boldsymbol{\theta}^\star \\ &= \boldsymbol{Q}^\top(\boldsymbol{\Lambda} + \mu\boldsymbol{I})^{-1}\boldsymbol{\Lambda}(\boldsymbol{Q}\boldsymbol{\theta}^\star) \\ \boldsymbol{Q}\boldsymbol{\theta}_\mu^\star &= (\boldsymbol{\Lambda} + \mu\boldsymbol{I})^{-1}\boldsymbol{\Lambda}(\boldsymbol{Q}\boldsymbol{\theta}^\star).\end{aligned}$$

So, under basis $\boldsymbol{Q}$, the optimum of ℓ_μ is

$$\boldsymbol{\theta}^\star_\mu = \operatorname{diag}\left(\frac{\lambda_i}{\lambda_i + \mu}\right)\boldsymbol{\theta}^\star.$$

I.e., the new solution scales each axis based on its sensitivity. If $\lambda_i \gg \mu$, then $\lambda_i/\lambda_i+\mu \approx 1$, so the solution does not change much in that direction—this matches the intuition that if the loss function is sensitive in a direction, the new solution will not change much in that direction.

8.3 *Early stopping*

In early stopping, we hold out a validation dataset, which is used for assessing generalization during training. If the validation error has not decreased for the past p checks, we stop training. Generally, the validation error is computed after every training epoch. This helps combat overfitting, because it does not allow the model to fit on the noise in the training data. Here, we make the assumption that the signal in the data is learned first and then the noise, because the signal contributes more to decreasing the loss function than the noise.

As in the analysis of weight decay, we make a second-order Taylor approximation at the optimum $\boldsymbol{\theta}^\star$,

$$\ell(\boldsymbol{\theta}) \approx \ell(\boldsymbol{\theta}^\star) + (\boldsymbol{\theta} - \boldsymbol{\theta}^\star)^\top \boldsymbol{\nabla}^2 \ell(\boldsymbol{\theta}^\star)(\boldsymbol{\theta} - \boldsymbol{\theta}^\star).$$

This has the following gradient,

$$\boldsymbol{\nabla}\ell(\boldsymbol{\theta}) \approx \boldsymbol{\nabla}^2 \ell(\boldsymbol{\theta}^\star)(\boldsymbol{\theta} - \boldsymbol{\theta}^\star).$$

Thus, gradient descent approximately results in

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \boldsymbol{\nabla}\ell(\boldsymbol{\theta}_t) \approx \boldsymbol{\theta}_t - \eta \boldsymbol{\nabla}^2 \ell(\boldsymbol{\theta}^\star)(\boldsymbol{\theta}_t - \boldsymbol{\theta}^\star).$$

Subtracting $\boldsymbol{\theta}^\star$ from both sides yields

$$\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}^\star \approx \left(\boldsymbol{I} - \eta \boldsymbol{\nabla}^2 \ell(\boldsymbol{\theta}^\star)\right)(\boldsymbol{\theta}_t - \boldsymbol{\theta}^\star).$$

Further, we diagonalize the Hessian $\boldsymbol{\nabla}^2 \ell(\boldsymbol{\theta}^\star) = \boldsymbol{Q}^\top \boldsymbol{\Lambda} \boldsymbol{Q}$, which gives

$$\begin{aligned}\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}^\star &\approx \left(\boldsymbol{Q}^\top \boldsymbol{Q} - \eta \boldsymbol{Q}^\top \boldsymbol{\Lambda} \boldsymbol{Q}\right)(\boldsymbol{\theta}_t - \boldsymbol{\theta}^\star) \\ &= \boldsymbol{Q}^\top (\boldsymbol{I} - \eta \boldsymbol{\Lambda}) \boldsymbol{Q} (\boldsymbol{\theta}_t - \boldsymbol{\theta}^\star).\end{aligned}$$

Under the basis $\boldsymbol{Q}$, we have

$$\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}^\star = (\boldsymbol{I} - \eta \boldsymbol{\Lambda})(\boldsymbol{\theta}_t - \boldsymbol{\theta}^\star).$$

Assuming $\boldsymbol{\theta}_0 = \boldsymbol{0}$, we get

$$\boldsymbol{\theta}_T = \boldsymbol{I} - (\boldsymbol{I} - \eta \boldsymbol{\Lambda})^T \boldsymbol{\theta}^\star = \operatorname{diag}\left(1 - (1 - \eta \lambda_i)^T\right).$$

We want to find the timestep such that early stopping is equivalent to weight decay. We can do this by equating it to the solution of weight decay under the same basis,

$$\frac{\lambda_i}{\lambda_i + \mu} \overset{!}{=} 1 - (1 - \eta \lambda_i)^T.$$

This is the case when $T \approx 1/\eta\mu$ where we assume $\mu \gg \max_i \lambda_i$.

8.4 Dropout

Dropout [Srivastava et al., 2014] randomly disables a subset of the model's weights during training—as a result, units become less dependent on one another. Instead of units being specialized and the model being highly dependent on specific units, the units stabilize to being generally useful to the task.

There are two views in which one can see dropout: (1) a regularization method or (2) an ensemble of networks defined by a binary mask $\boldsymbol{b} \in \{0,1\}^n$ of whether a weight is activated or not,

$$p[\boldsymbol{w}](\boldsymbol{y} \mid \boldsymbol{x}) = \sum_{\boldsymbol{b} \in \{0,1\}^n} p(\boldsymbol{b}) p[\boldsymbol{w} \odot \boldsymbol{b}](\boldsymbol{y} \mid \boldsymbol{x}), \quad p(\boldsymbol{b}) = \prod_{i=1}^{n} \pi_i^{b_i} (1 - \pi_i)^{1-b_i},$$

where π_i is the probability of weight i being activated. In order to prevent having to evaluate hundreds of sampled networks, we can use the heuristic of scaling the weights by their dropout probability,

$$\tilde{\theta}_i \leftarrow \pi_i \theta_i.$$

In practice, it has consistently been found that dropout results in greater generalization.

8.5 Normalization

The goal of normalization is to make all units more similar, such that optimization is easier. Let $f : \mathbb{R}^d \to \mathbb{R}$ be some layer in a model. This layer can be normalized by

$$\bar{f} = \frac{f - \mathbb{E}[f(\boldsymbol{x})]}{\sqrt{\operatorname{Var}[f(\boldsymbol{x})]}}.$$

As a result, $\mathbb{E}[\bar{f}(\boldsymbol{x})] = 0$ and $\operatorname{Var}[\bar{f}(\boldsymbol{x})] = 1$. However, this removes 2 degrees of freedom—bias and variance—which might be important to the model. To introduce bias and variance back, we explicitly parametrize them,

$$\bar{f}[\mu, \gamma](\boldsymbol{x}) = \mu + \gamma \bar{f}(\boldsymbol{x}).$$

In general, $\mathbb{E}[f]$ and $\operatorname{Var}[f]$ are expensive to compute due to a large amount of data. Let $\mathcal{B}$ be a mini-batch, then a BN (***B**atch **N**ormalization*) [Ioffe, 2015] layer estimates them by

$$\mathbb{E}[f(\boldsymbol{x})] \approx \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x} \in \mathcal{B}} f(\boldsymbol{x})$$
$$\operatorname{Var}[f(\boldsymbol{x})] \approx \frac{1}{|\mathcal{B}|} \sum_{\boldsymbol{x} \in \mathcal{B}} (f(\boldsymbol{x}) - \mathbb{E}[f(\boldsymbol{x})])^2.$$

Then, we re-introduce bias and variance by adding μ and γ parameters as above.

Normalization is very effective and even essential in some model types—the question is why it is so effective. Historically, many believed that it helps to combat covariance shift [Ioffe, 2015], however, this is no longer believed to be true. Santurkar et al. [2018] presented a modern motivation as follows. Let $g[\mu, \gamma]$ be a BN layer, $h[\boldsymbol{w}]$ a linear layer, and ϕ a non-linearity and compose a block as $f = \phi \circ g[\mu, \gamma] \circ h[\boldsymbol{w}]$. Then,

$$\begin{aligned} f(\boldsymbol{x}) &= \phi\left(\mu + \gamma \frac{\boldsymbol{w}^\top \boldsymbol{x} - \mathbb{E}\left[\boldsymbol{w}^\top \boldsymbol{x}\right]}{\sqrt{\operatorname{Var}\left[\boldsymbol{w}^\top \boldsymbol{x}\right]}}\right) \\ &= \phi\left(\mu + \gamma \frac{\boldsymbol{w}^\top (\boldsymbol{x} - \mathbb{E}[\boldsymbol{x}])}{\|\boldsymbol{w}\|_{\boldsymbol{\Sigma}}}\right), \quad \boldsymbol{\Sigma} = \mathbb{E}\left[\boldsymbol{x}\boldsymbol{x}^\top\right]. \end{aligned}$$

Assume $\mathbb{E}[\boldsymbol{x}] = \boldsymbol{0}$,

$$\begin{aligned} &= \phi\left(\mu + \gamma \frac{\boldsymbol{w}^\top \boldsymbol{x}}{\|\boldsymbol{w}\|_2} \frac{\|\boldsymbol{w}\|_2}{\|\boldsymbol{w}\|_{\boldsymbol{\Sigma}}}\right) \\ &= \phi\left(\mu + \gamma \left(\frac{\boldsymbol{w}}{\|\boldsymbol{w}\|_2}\right)^\top \boldsymbol{x} \frac{\|\boldsymbol{w}\|_{\boldsymbol{I}}}{\|\boldsymbol{w}\|_{\boldsymbol{\Sigma}}}\right). \end{aligned}$$

Effectively, the weight vector is normalized and the result is scaled by the discrepancy between $\|\boldsymbol{w}\|_{\boldsymbol{I}}$ and $\|\boldsymbol{w}\|_{\boldsymbol{\Sigma}}$. Practically, it has been found that μ is not as important as γ.

LN (***L**ayer **N**ormalization*) [Lei Ba et al., 2016] differs from BN in the way that it estimates the mean and variance. Instead of computing them over the batch dimension, it does so over the feature dimension,

$$\mathbb{E}[\boldsymbol{x}] \approx \frac{1}{d} \sum_{j=1}^{d} x_j, \quad \operatorname{Var}[\boldsymbol{x}] \approx \frac{1}{d} \sum_{j=1}^{d} (x_j - \mathbb{E}[\boldsymbol{x}])^2.$$

Each sample is thus normalized with different means and variances. The advantage is that the normalization is defined independently from the mini-batch size. Generally, BN is used in computer vision, whereas LN is used in natural language processing.

8.6 Weight normalization

In weight normalization, the weights are normalized before applying them,

$$f[\boldsymbol{v}, \gamma](\boldsymbol{x}) = \phi\left(\boldsymbol{w}^\top \boldsymbol{x}\right), \quad \boldsymbol{w} = \frac{\gamma}{\|\boldsymbol{v}\|_2} \boldsymbol{v}.$$

As we have seen before, this is equivalent to applying normalization if $\frac{\|\boldsymbol{w}\|_{\boldsymbol{I}}}{\|\boldsymbol{w}\|_{\boldsymbol{\Sigma}}} = 1$.

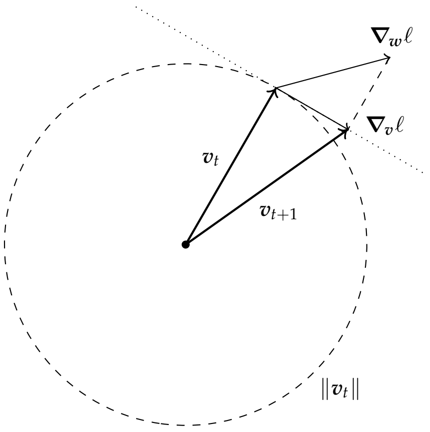


Figure 8.1. When using weight normalization, the direction of $\boldsymbol{w}$ is projected out at every update step.

The gradients of the parameters are computed as follows,

$$\begin{aligned}
\frac{\partial \ell}{\partial \gamma} &= \frac{\partial \ell}{\partial \boldsymbol{w}} \frac{\partial \boldsymbol{w}}{\partial \gamma} \\
&= \frac{\partial \ell}{\partial \boldsymbol{w}} \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|}. \\
\frac{\partial \ell}{\partial \boldsymbol{v}} &= \frac{\partial \ell}{\partial \boldsymbol{w}} \frac{\partial \boldsymbol{w}}{\partial \boldsymbol{v}} \\
&= \gamma \frac{\partial \ell}{\partial \boldsymbol{w}} \left(\frac{1}{\|\boldsymbol{v}\|} \boldsymbol{I} + \left(\frac{\partial}{\partial \boldsymbol{v}} \frac{1}{\|\boldsymbol{v}\|} \right) \boldsymbol{v}^\top \right) \\
&= \gamma \frac{\partial \ell}{\partial \boldsymbol{w}} \left(\frac{1}{\|\boldsymbol{v}\|} \boldsymbol{I} - \frac{\boldsymbol{v}\boldsymbol{v}^\top}{\|\boldsymbol{v}\|^3} \right) \\
&= \frac{\gamma}{\|\boldsymbol{v}\|} \frac{\partial \ell}{\partial \boldsymbol{w}} \left(\boldsymbol{I} - \frac{\boldsymbol{w}\boldsymbol{w}^\top}{\gamma^2} \right) && \frac{\boldsymbol{v}}{\|\boldsymbol{v}\|} = \frac{\boldsymbol{w}}{\gamma}. \\
&= \frac{\gamma}{\|\boldsymbol{v}\|} \frac{\partial \ell}{\partial \boldsymbol{w}} \left(\boldsymbol{I} - \frac{\boldsymbol{w}\boldsymbol{w}^\top}{\|\boldsymbol{w}\|^2} \right). && \|\boldsymbol{w}\| = \gamma.
\end{aligned}$$

Here, $\boldsymbol{I} - \frac{\boldsymbol{w}\boldsymbol{w}^\top}{\|\boldsymbol{w}\|_2^2}$ is a projection matrix onto the complement of $\boldsymbol{w}$—the direction of $\boldsymbol{w}$ is projected out at every update step, as shown in Figure 8.1.

8.7 Data augmentation

Data augmentation involves transforming the data with transformations that the model should be invariant to and pass it to the model during training—the model learns the invariances of the data rather than that we have to design the architecture as such. Let $\{\tau_k\}_{k=1}^K$ be transformations, then the dataset is extended in the following way,

$$\{(\boldsymbol{x}_i, y_i)\}_{i=1}^n \mapsto \bigcup_{k=1}^K \{(\tau_k(\boldsymbol{x}_i), y_i)\}_{i=1}^n.$$

In practice during training, the transformations are done on the fly, because they are usually cheap.

8.8 Label smoothing

Classifiers are generally not good at dealing with mislabeled data, so we smooth the labels out by replacing them with noisy probability distributions,

$$y \mapsto \boldsymbol{\Pi} \boldsymbol{e}_y \in \Delta^{k-1},$$

where k is the number of output classes. Here, $\boldsymbol{\Pi}$ is a confusion matrix that defines how the "smoothed" distribution of each label is defined. This distribution is then used in the cross-entropy loss. This concept can be generalized to any type of label by simply adding noise.

8.9 Distillation

In model distillation [Hinton, 2015], we have a teacher and a student model, where the student attempts to match the teacher's outputs. The idea is that a teacher's knowledge lies in its outputs, so we should be able to condense this information into a shallower model if we make use of its outputs. In practice, we generate a lot of data with the teacher model and train the student on it.

In a classification setting, we train the student to match the probability distribution of the teacher. Let F be the teacher, G its student and denote by F_y the logit of the teacher of class y, then we use the cross-entropy loss between the teacher's and student's output logits,

$$\ell(\boldsymbol{x}) = \sum_{y \in \mathcal{Y}} \frac{\exp(F_y(\boldsymbol{x}))}{\sum_{y' \in \mathcal{Y}} \exp(F_{y'}(\boldsymbol{x}))} \log\left(\frac{\exp(G_y(\boldsymbol{x}))}{\sum_{y' \in \mathcal{Y}} \exp(G_{y'}(\boldsymbol{x}))} \right).$$

Hinton [2015] suggested using a "tempered" cross-entropy loss,

$$\ell(\boldsymbol{x}) = \sum_{y \in \mathcal{Y}} \frac{\exp\left({}^{F_y(x)}/_T\right)}{\sum_{y' \in \mathcal{Y}} \exp\left({}^{F_{y'}(x)}/_T\right)} \left(\frac{1}{T} G_y(\boldsymbol{x}) - \log \sum_{y' \in \mathcal{Y}} \exp\left({}^{G_{y'}(x)}/_T\right) \right).$$

Here, the distillation loss is "tempered" by a temperature parameter $T > 0$. Typically, the teacher is trained with $T = 1$. However, often the teacher gets overconfident in its predictions, so we can set $T > 1$ to soften its outputs. Deriving the gradient is trivial,

$$\frac{\partial \ell}{\partial G_y} = \frac{1}{T} \left(\frac{\exp\left({}^{F_y(x)}/_T\right)}{\sum_{y' \in \mathcal{Y}} \exp\left({}^{F_{y'}(x)}/_T\right)} - \frac{\exp\left({}^{G_y(x)}/_T\right)}{\sum_{y' \in \mathcal{Y}} \exp\left({}^{G_{y'}(x)}/_T\right)} \right).$$

Notice that the gradient is a difference of tempered logits.

9 Neural tangent kernel

9.1 Linearized models

The non-linearities of neural networks make them hard to analyze. We can linearize a model $f[\boldsymbol{\theta}]$ by a first-order Taylor approximation over fixed parameters $\boldsymbol{\theta}_0$,

$$f[\boldsymbol{\theta}] \approx f[\boldsymbol{\theta}_0] + \langle \nabla f[\boldsymbol{\theta}_0], \boldsymbol{\theta} - \boldsymbol{\theta}_0 \rangle, \quad \boldsymbol{\theta} \in \mathbb{R}^p.$$

Here, f is still a non-linear function, but it is linear w.r.t. $\boldsymbol{\theta}$. In this way, we can define a linear model with parameters $\boldsymbol{\beta}$,

$$h[\boldsymbol{\beta}](\boldsymbol{x}) \doteq f[\boldsymbol{\theta}_0](\boldsymbol{x}) + \boldsymbol{\beta}^\top \nabla f[\boldsymbol{\theta}_0](\boldsymbol{x}).$$

Here, $\nabla f[\boldsymbol{\theta}] : \mathbb{R}^d \to \mathbb{R}^p$ can be seen as constructing a p-dimensional feature vector and $h[\boldsymbol{\beta}]$ a linear model over those features—we have a kernel method with the following kernel,

$$k(\boldsymbol{x}, \boldsymbol{x}') \doteq \nabla f[\boldsymbol{\theta}_0](\boldsymbol{x})^\top \nabla f[\boldsymbol{\theta}_0](\boldsymbol{x}').$$

Assuming that we want to minimize the mean-squared error,

$$\ell(\boldsymbol{\beta}) \doteq \frac{1}{2}\|\boldsymbol{f} + \boldsymbol{\Phi}\boldsymbol{\beta} - \boldsymbol{y}\|^2, \quad \boldsymbol{\Phi} \doteq [\nabla f[\boldsymbol{\theta}_0](\boldsymbol{x}_1), \dots, \nabla f[\boldsymbol{\theta}_0](\boldsymbol{x}_n)] \in \mathbb{R}^{n \times p},$$

we get the following unique solution,

$$\boldsymbol{\beta}^\star = \boldsymbol{\Phi}^\top \boldsymbol{K}^{-1}(\boldsymbol{y} - \boldsymbol{f}), \quad \boldsymbol{K} \doteq \boldsymbol{\Phi}\boldsymbol{\Phi}^\top.$$

And, we make predictions by

$$h^\star(\boldsymbol{x}) = \boldsymbol{k}(\boldsymbol{x})^\top \boldsymbol{K}^{-1}(\boldsymbol{y} - \boldsymbol{f}), \quad \boldsymbol{k}(\boldsymbol{x}) \doteq [k(\boldsymbol{x}, \boldsymbol{x}_1), \dots, k(\boldsymbol{x}, \boldsymbol{x}_n)] \in \mathbb{R}^n.$$

We now have a linearized network—along with a way of evaluating it—which is simply an approximation of a model with parameters $\boldsymbol{\theta}_0$. Note that computing $\boldsymbol{K}$ may be intractable due to the number of samples and parameters. Furthermore, Lee et al. [2019] showed that linearized networks are non-competitive with full networks. However, the benefit of this derivation is that we can look at neural networks through the lens of kernel methods, as long as the parameters do not evolve far away from $\boldsymbol{\theta}_0$ such that the linear approximation achieves a negligible loss.

9.2 Training dynamics

Consider the case where we wish to minimize the mean-squared error,

$$\ell(\boldsymbol{\theta}) = \frac{1}{2}\|\boldsymbol{f}[\boldsymbol{\theta}] - \boldsymbol{y}\|^2, \quad \boldsymbol{f}[\boldsymbol{\theta}] \doteq [f[\boldsymbol{\theta}](\boldsymbol{x}_1), \dots, f[\boldsymbol{\theta}](\boldsymbol{x}_n)].$$

We optimize the model parameters by gradient descent,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \nabla \ell(\boldsymbol{\theta}_t).$$

We can derive the gradient flow ODE of this process by rearranging the above,

$$\begin{aligned}\frac{\boldsymbol{\theta}_{t+1} - \boldsymbol{\theta}_t}{\eta} &= -\boldsymbol{\nabla}\ell(\boldsymbol{\theta}_t) \\ \frac{\mathrm{d}\boldsymbol{\theta}_t}{\mathrm{d}t} &= -\boldsymbol{\nabla}\ell(\boldsymbol{\theta}_t) \\ &= \sum_{i=1}^{n}(y_i - f[\boldsymbol{\theta}_t](\boldsymbol{x}_i))\boldsymbol{\nabla} f[\boldsymbol{\theta}_t](\boldsymbol{x}_i) \\ &\doteq \dot{\boldsymbol{\theta}}_t.\end{aligned}$$

η is the stepsize and we turn it into continuous time.

We can also consider the functional gradient flow of the function $f_j[\boldsymbol{\theta}]$ for a data point $\boldsymbol{x}_j$,

$$\begin{aligned}\dot{f}_j[\boldsymbol{\theta}_t] &\doteq \frac{\mathrm{d}f[\boldsymbol{\theta}_t](\boldsymbol{x}_j)}{\mathrm{d}t} \\ &= \dot{\boldsymbol{\theta}}_t^{\top}\boldsymbol{\nabla} f[\boldsymbol{\theta}_t](\boldsymbol{x}_j) \\ &= \left(\sum_{i=1}^{n}(y_i - f[\boldsymbol{\theta}](\boldsymbol{x}_i))\boldsymbol{\nabla} f[\boldsymbol{\theta}_t](\boldsymbol{x}_i)\right)^{\top}\boldsymbol{\nabla} f[\boldsymbol{\theta}_t](\boldsymbol{x}_j) \\ &= \sum_{i=1}^{n}(y_i - f[\boldsymbol{\theta}_t](\boldsymbol{x}_i))\boldsymbol{\nabla} f[\boldsymbol{\theta}_t](\boldsymbol{x}_i)^{\top}\boldsymbol{\nabla} f[\boldsymbol{\theta}_t](\boldsymbol{x}_j) \\ &= \sum_{i=1}^{n}(y_i - f[\boldsymbol{\theta}_t](\boldsymbol{x}_i))k[\boldsymbol{\theta}_t](\boldsymbol{x}_i, \boldsymbol{x}_j).\end{aligned}$$

Chain rule.

This can also be written in matrix form as

$$\dot{\boldsymbol{f}}[\boldsymbol{\theta}_t] = \boldsymbol{K}[\boldsymbol{\theta}_t](\boldsymbol{y} - \boldsymbol{f}[\boldsymbol{\theta}_t]).$$

We can generalize this to any loss function by

$$\dot{\boldsymbol{f}}[\boldsymbol{\theta}_t] = -\boldsymbol{K}[\boldsymbol{\theta}_t]\boldsymbol{\nabla}_{\boldsymbol{f}[\boldsymbol{\theta}]}\ell(\boldsymbol{\theta}_t).$$

But, for our purposes, we only consider the MSE loss.

This shows that the kernel governs the evolution of the joint sample predictions—the kernel is called the NTK (***N**eural **T**angent **K**ernel*).

Now we can analyze how gradient descent behaves—it is entirely dependent on the kernel. However, the NTK has a dependence on the parameters, so it is practically not very interesting. We will next see why the NTK is interesting as a theoretical tool.

9.3 Infinite width

In practice, it has been found that as the width of a model is scaled, the parameters stay close to their initialization during gradient descent. One can prove, under basic assumptions, that if the model is scaled to infinite width, then the kernel converges in probability to a deterministic limit,

$$k[\boldsymbol{\theta}] \xrightarrow{p\to\infty} k_\infty, \quad \boldsymbol{\theta} \in \mathbb{R}^p.$$

The limit k_∞ only depends on the law of initialization, not the initial random parameters themselves. Effectively, there is only one infinite-width network at initialization. Under these training dynamics, minimizing the mean-squared error equates to solving a kernel regression problem with

k_∞. This provides analytical insight into why overparametrization works so well in practice and why such models generalize, despite having the obvious ability to overfit.

9.4 NTK constancy

Consider an MLP with L layers and m_l denoting the number of parameters in layer $l \in [L]$, where we initialize the parameters by

$$w_{ij}^l \sim \mathcal{N}\left(0, \frac{\sigma_w}{\sqrt{m_l}}\right), \quad b_i^l \sim \mathcal{N}\left(0, \frac{\sigma_b}{\sqrt{m_l}}\right).$$

LeCun initialization.

Now consider the case where $m_l \to \infty$ for all $l \in [L]$. Under suitable conditions, it can be shown that with the above scaling, the initial NTK converges to a deterministic limit k_∞. The kernel limit depends only on the law of initialization and not the actual initialized values. Effectively, there is only one possible initial infinite-width network. With a few additional assumptions, it can also be shown that $\boldsymbol{\theta}_t$ converges uniformly to k_∞.

In other words, the NTK remains constant under gradient flow—NTK constancy,

$$\frac{\partial k[\boldsymbol{\theta}_t]}{\partial t} = 0.$$

When NTK constancy holds, learning in the infinite width limit works the same as for the linearized model—kernel methods. As a result, the solution to NTK learning can be expressed as

$$f_\infty(\boldsymbol{x}) = \boldsymbol{k}_\infty(\boldsymbol{x})^\top \boldsymbol{K}_\infty^{-1}(\boldsymbol{y} - \boldsymbol{f}).$$

Bietti and Mairal [2019] showed that the NTK of a two-layer MLP with a ReLU activation function can analytically be written as

$$k_\infty(\boldsymbol{x}, \boldsymbol{x}') = \|\boldsymbol{x}\|\|\boldsymbol{x}'\|\kappa\left(\frac{\boldsymbol{x}^\top \boldsymbol{x}'}{\|\boldsymbol{x}\|\|\boldsymbol{x}'\|}\right),$$

where

$$\kappa(u) \doteq \frac{2u}{\pi}(\pi - \arccos(u)) + \frac{\sqrt{1-u^2}}{\pi}.$$

10 Bayesian learning

Typically in deep learning, we start from initial parameters and evolve them through gradient descent to obtain near-optimal parameters. In Bayesian learning, we take a more general point of view. Starting from a prior $p(\boldsymbol{\theta})$, we want to compute or approximate the posterior $p(\boldsymbol{\theta} \mid \mathcal{S})$. The ultimate goal is the Bayesian predictive distribution,

$$f(\boldsymbol{x}) = \int p(\boldsymbol{\theta} \mid \mathcal{S}) f[\boldsymbol{\theta}](\boldsymbol{x}) \mathrm{d}\boldsymbol{\theta},$$

We weight predictions by the posterior probability of its parameters.

where the posterior can be defined via Bayes' rule,

$$p(\boldsymbol{\theta} \mid \mathcal{S}) = \frac{p(\mathcal{S} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta})}{p(\mathcal{S})}, \quad p(\mathcal{S}) = \int p(\boldsymbol{\theta}) p(\mathcal{S} \mid \boldsymbol{\theta}) \mathrm{d}\boldsymbol{\theta}.$$

The evidence $p(\mathcal{S})$ is often intractable, but we often do not need it when unnormalized probabilities are sufficient.

The isotropic Gaussian is a common prior,

$$p(\boldsymbol{\theta}) = \mathcal{N}(\mathbf{0}, \sigma^2 \boldsymbol{I}).$$

Optimizing this prior leads to a weight decay term as we have seen before,

$$\begin{aligned}
\boldsymbol{\theta}^\star &= \operatorname*{argmax}_{\boldsymbol{\theta} \in \Theta} p(\boldsymbol{\theta} \mid \mathcal{S}) \\
&= \operatorname*{argmax}_{\boldsymbol{\theta} \in \Theta} p(\mathcal{S} \mid \boldsymbol{\theta}) p(\boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta} \in \Theta} -\log p(\mathcal{S} \mid \boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta} \in \Theta} -\log p(\mathcal{S} \mid \boldsymbol{\theta}) - \log p(\boldsymbol{\theta}) \\
&= \operatorname*{argmin}_{\boldsymbol{\theta} \in \Theta} -\log p(\mathcal{S} \mid \boldsymbol{\theta}) + \frac{1}{2\sigma^2} \|\boldsymbol{\theta}\|^2 \\
&= \circledast
\end{aligned}$$

$p(\mathcal{S})$ is a constant w.r.t. $\boldsymbol{\theta}$.

The logarithm is increasing.

Plug in the definition of the Gaussian and remove all terms which are constant w.r.t. $\boldsymbol{\theta}$.

Thus, a priori, the model favors small parameters.

Further assume that we have data that is described by a function $f^\star : \mathcal{X} \to \mathcal{Y}$ with normal noise,

$$y_i = f^\star(\boldsymbol{x}_i) + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, \gamma^2).$$

We get the following negative log-likelihood,

$$\begin{aligned}
-\log p(\mathcal{S} \mid \boldsymbol{\theta}) &= -\sum_{i=1}^{n} \log p(y_i \mid \boldsymbol{x}_i, \boldsymbol{\theta}) \\
&\propto \sum_{i=1}^{n} \frac{1}{2\gamma^2} (y_i - f[\boldsymbol{\theta}](\boldsymbol{x}_i))^2 \\
&= \frac{1}{2\gamma^2} \|\boldsymbol{y} - \boldsymbol{f}[\boldsymbol{\theta}]\|^2.
\end{aligned}$$

We assume that the data is sampled *i.i.d.*

We modeled $y_i \sim \mathcal{N}(f^\star(\boldsymbol{x}_i), \gamma^2)$ and $f[\boldsymbol{\theta}]$ must approximate $f^\star$.

So, the final optimization problem becomes

$$\circledast = \operatorname*{argmin}_{\boldsymbol{\theta}\in\Theta} \frac{1}{2\gamma^2}\|\boldsymbol{y} - \boldsymbol{f}[\boldsymbol{\theta}]\|^2 + \frac{1}{2\sigma^2}\|\boldsymbol{\theta}\|^2.$$

This only gives us the best parameters, however, we want the posterior distribution of parameters, such that we can compute the Bayesian predictive distribution. Thus, the question becomes how we sample parameters from the posterior $p(\boldsymbol{\theta} \mid \mathcal{S})$ to approximate the predictive distribution,

$$f(\boldsymbol{x}) \approx \sum_{i=1}^{m} \frac{p(\boldsymbol{\theta}^{(i)} \mid \mathcal{S})}{\sum_{j=1}^{m} p(\boldsymbol{\theta}^{(j)} \mid \mathcal{S})} f[\boldsymbol{\theta}^{(i)}](\boldsymbol{x})$$

The next sections will consider common methods to sample the unknown posterior distribution.

10.1 Markov chain Monte Carlo

MCMC (***M**arkov **C**hain **M**onte **C**arlo*) is the standard method of sampling from a high-dimensional posterior distribution. It does so by constructing a Markov chain in the parameter space, where the stationary distribution is equal to the posterior—when sampling a random sequence of parameters, we converge to the posterior distribution. If we can construct such a Markov chain, we can sample the posterior by running the Markov chain for long enough—this period is known as the burn-in period. Further note that close parameters in the Markov chain are highly correlated, so we cannot take nearby samples as independent draws from the posterior.

Lemma 10.1. If a Markov chain, described by its kernel $\Pi : \Theta \to \Delta(\Theta)$, satisfies the DBE (***D**etailed **B**alance **E**quation*),

$$q(\boldsymbol{\theta})\Pi(\boldsymbol{\theta}' \mid \boldsymbol{\theta}) = q(\boldsymbol{\theta}')\Pi(\boldsymbol{\theta} \mid \boldsymbol{\theta}'), \quad \forall \boldsymbol{\theta}, \boldsymbol{\theta}' \in \Theta,$$

then the Markov chain is time reversible and has the unique stationary distribution q.

Metropolis-Hastings. Using Lemma 10.1, we can guarantee that the stationary distribution of the Markov chain is the posterior if we have

$$p(\boldsymbol{\theta} \mid \mathcal{S})\Pi(\boldsymbol{\theta}' \mid \boldsymbol{\theta}) = p(\boldsymbol{\theta}' \mid \mathcal{S})\Pi(\boldsymbol{\theta} \mid \boldsymbol{\theta}'), \quad \forall \boldsymbol{\theta}, \boldsymbol{\theta}' \in \Theta.$$

MH (***M**etropolis-**H**astings*) starts with sampling from an arbitrary Markov kernel $\tilde{\Pi}$ and modifies the transition probability with an acceptance (or rejection) step to achieve an effective kernel Π that satisfies the DBE. Let $\alpha(\cdot \mid \cdot)$ be the acceptance function, and construct Π as

$$\Pi(\boldsymbol{\theta}' \mid \boldsymbol{\theta}) = \tilde{\Pi}(\boldsymbol{\theta}' \mid \boldsymbol{\theta})\alpha(\boldsymbol{\theta}' \mid \boldsymbol{\theta}).$$

Intuitively, $\tilde{\Pi}$ makes a suggestion and α accepts or rejects it, probabilistically. Then, we need to construct α such that it satisfies the DBE,

$$p(\boldsymbol{\theta} \mid \mathcal{S})\tilde{\Pi}(\boldsymbol{\theta}' \mid \boldsymbol{\theta})\alpha(\boldsymbol{\theta}' \mid \boldsymbol{\theta}) = p(\boldsymbol{\theta}' \mid \mathcal{S})\tilde{\Pi}(\boldsymbol{\theta} \mid \boldsymbol{\theta}')\alpha(\boldsymbol{\theta} \mid \boldsymbol{\theta}')$$

If we enforce the acceptance function to satisfy a one-sided structure,

$$\alpha(\boldsymbol{\theta}' \mid \boldsymbol{\theta}) = 1 \vee \alpha(\boldsymbol{\theta} \mid \boldsymbol{\theta}') = 1,$$

the following is the only choice of α,

$$\alpha(\boldsymbol{\theta} \mid \boldsymbol{\theta}') = \min\left\{1, \frac{p(\boldsymbol{\theta} \mid \mathcal{S})\tilde{\Pi}(\boldsymbol{\theta}' \mid \boldsymbol{\theta})}{p(\boldsymbol{\theta}' \mid \mathcal{S})\tilde{\Pi}(\boldsymbol{\theta} \mid \boldsymbol{\theta}')}\right\}.$$

If $\tilde{\Pi}$ is symmetric, then the acceptance probability is simply the ratio of posteriors. Note that we do not need to compute the normalizer $p(\mathcal{S})$ for this method.

A potential problem with this approach is that while the Markov chain is guaranteed to converge to the posterior as its stationary distribution, this might take arbitrarily long—the burn-in period can be impractically costly. This is due to poor initial kernels $\tilde{\Pi}$ leading to very high rejection probabilities.

Hamiltonian Monte Carlo. HMC (***H**amiltonian **M**onte **C**arlo*) is an MCMC method for obtaining posterior averages. Consider an energy function—or loss function—equal to the negative log posterior,

$$E(\boldsymbol{\theta}) \doteq -\sum_{x,y} \log p[\boldsymbol{\theta}](y \mid \boldsymbol{x}) - \log p(\boldsymbol{\theta}).$$

The Hamiltonian is defined as the energy function, augmented with a momentum vector $\boldsymbol{v}$ and a corresponding energy term,

$$H(\boldsymbol{\theta}, \boldsymbol{v}) \doteq E(\boldsymbol{\theta}) + \frac{1}{2}\boldsymbol{v}^\top \boldsymbol{M}^{-1}\boldsymbol{v}.$$

The joint probability of $\boldsymbol{\theta}$ and $\boldsymbol{v}$ is given by a Gibbs distribution,

$$p(\boldsymbol{\theta}, \boldsymbol{v}) \propto \exp(-H(\boldsymbol{\theta}, \boldsymbol{v})).$$

We get the following two coupled differential equations—Hamiltonian dynamics,

$$\dot{\boldsymbol{v}} = -\boldsymbol{\nabla} E(\boldsymbol{\theta}), \quad \dot{\boldsymbol{\theta}} = \boldsymbol{v}.$$

HMC discretizes this dynamic with a stepsize η,

$$\begin{aligned} \boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t + \eta \boldsymbol{v}_t \\ \boldsymbol{v}_{t+1} &= \boldsymbol{v}_t - \eta \boldsymbol{\nabla} E(\boldsymbol{\theta}_t). \end{aligned}$$

Although very slowly, HMC samples from the posterior by following these dynamics. Note that it is very similar to gradient descent with momentum—we essentially sample the posterior by following momentum-based gradient descent dynamics.[16] However, this approach requires the full gradient, which is often intractable in practice.

[16] As a result, optimization with momentum gradient descent results in a single sample approximation of the predictive distribution.

Langevin dynamics. Langevin dynamics extends HMC by introducing friction,

See the following video for a visualization of the sampling process with Langevin dynamics: `https://www.youtube.com/watch?v=cVn0kru3hL8`.

$$\dot{\boldsymbol{\theta}} = \boldsymbol{v}$$
$$\mathrm{d}\boldsymbol{v} = -\boldsymbol{\nabla} E(\boldsymbol{\theta})\mathrm{d}t - \boldsymbol{B}\boldsymbol{v}\mathrm{d}t + \mathcal{N}(\boldsymbol{0}, 2\boldsymbol{B}\mathrm{d}t).$$

Intuitively, the friction reduces the momentum and "dissipates" kinetic energy and the Wiener noise process injects stochasticity. As with HMC, we can discretize the above process,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \eta \boldsymbol{v}_t$$
$$\boldsymbol{v}_{t+1} = (1 - \eta\gamma)\boldsymbol{v}_t - \eta s \boldsymbol{\nabla}\tilde{E}(\boldsymbol{\theta}) + \sqrt{2\gamma\eta}\mathcal{N}(\boldsymbol{0}, \boldsymbol{I}).$$

Here, $\tilde{E}$ is a stochastic potential function, which includes an empirical loss over a random mini-batch of the data. The first term introduces friction, which leads to an exponential damping with time.

10.2 Gaussian processes

GPs (***G**aussian **P**rocesses*) are one of the few fully tractable Bayesian methods. It starts from a continuous stochastic process over the input domain $\mathcal{X}$,

$$\{f(\boldsymbol{x}) \mid \boldsymbol{x} \in \mathcal{X}\},$$

where each $f(\boldsymbol{x})$ is a real random variable. f is a GP if for every finite subset $\{\boldsymbol{x}_1, \dots \boldsymbol{x}_n\} \subset \mathcal{X}$, the resulting finite marginal is jointly normally distributed,

$$\begin{bmatrix} f(\boldsymbol{x}_1) \\ \vdots \\ f(\boldsymbol{x}_n) \end{bmatrix} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}).$$

The mean $\boldsymbol{\mu}$ can be computed by a deterministic regression, whereas the covariance matrix $\boldsymbol{\Sigma}$ introduces stochasticity to the prediction. When given a finite dataset, the covariance matrix can be fully evaluated using a kernel function,

$$\sigma_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j), \quad k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}.$$

The kernel function can be seen as a prior over function space that describes how related the output values corresponding to two input values should be. *E.g.*, we might want to encode that close input values should result in close output values—then you might want to use the RBF kernel,

$$k(\boldsymbol{x}, \boldsymbol{x}') = \exp\Big(-\gamma \|\boldsymbol{x} - \boldsymbol{x}'\|^2\Big).$$

Linear networks. Assume we have n d-dimensional inputs. Consider a single linear unit $\boldsymbol{w} \in \mathbb{R}^d$ with a random Gaussian weight vector,

$$\boldsymbol{w} \sim \mathcal{N}\Big(\boldsymbol{0}, \frac{\sigma^2}{d}\boldsymbol{I}_d\Big).$$

The outputs can be written as $y_i = \boldsymbol{w}^\top \boldsymbol{x}_i$ for all $i \in [n]$, or in a vectorized form,

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w}, \quad \boldsymbol{X} \in \mathbb{R}^{n \times d}.$$

Note that this is a Gaussian vector,

$$\boldsymbol{y} \sim \mathcal{N}\left(\boldsymbol{0}, \frac{\sigma^2}{d}\boldsymbol{X}^\top \boldsymbol{X}\right).$$

Hence, this is a Gaussian process with the following kernel,

$$k(\boldsymbol{x}, \boldsymbol{x}') = \frac{\sigma^2}{n}\boldsymbol{x}^\top \boldsymbol{x}'.$$

We can do this for multiple units, because the preactivations of units in the same layer are independent, conditioned on the input.

If we increase the depth of this network, we do not get the same effect in general, because there is randomness not only in the weights, but also in the preactivations. However, a deep preactivation process is "near normal" for high-dimensional inputs. This can be made rigorous with a multivariate version of the central limit theorem.

Non-linear networks. By introducing non-linear activation functions into the network, the activations are no longer Gaussian. However, due to the central limit theorem, they are effectively shaped back into Gaussians when they propagate to the next layer. The mean function can be computed by

$$\mu\left(\boldsymbol{x}^{\ell+1}\right) = \mathbb{E}\left[\phi\left(\boldsymbol{W}^\ell \boldsymbol{x}^\ell\right)\right].$$

This might need to be computed using numerical integration. The kernel can be defined recursively,

$$k_{ij}^\ell = \mathbb{E}\left[\phi\left(x_i^{\ell-1}\right)\phi\left(x_j^{\ell-1}\right)\right].$$

We can now use kernel regression,

$$f^\star(\boldsymbol{x}) = \boldsymbol{k}(\boldsymbol{x})^\top \boldsymbol{K}^{-1}\boldsymbol{y}, \quad \boldsymbol{K} = \boldsymbol{K}^L.$$

In conclusion, deep neural networks can be thought of as GPs in the infinite-width limit.[17] The advantage is that we can use wide random layers without the need for training, we can quantify uncertainty, and we can leverage techniques from kernel machines. However, in general, it is not feasible to compute $f^\star$ and store $\boldsymbol{K}^\ell$. Furthermore, the expectations need to be computed, which is much less efficient than optimizing weights with gradient descent.

[17] In the infinite-width limit, all preactivations can be viewed as incoming Gaussians.

11 Statistical learning theory

11.1 Vapnik-Chervonenkis theory of machine learning

Let $\mathcal{F}$ be a function class and consider its functions $f \in \mathcal{F}$ to be binary classifiers. Then, the following denotes the set of possible classification outcomes over a dataset $\mathcal{S} \subseteq \mathcal{X} \times \mathcal{Y}$ with n datapoints,

$$\mathcal{F}(\mathcal{S}) \doteq \{[f(\boldsymbol{x}_1), \dots, f(\boldsymbol{x}_n)] \in \{0,1\}^n \mid f \in \mathcal{F}\}, \quad f : \mathcal{X} \to \{0,1\}.$$

Furthermore, the following denotes the maximal number of different classifications of a dataset of size n that can be realized by functions in $\mathcal{F}$,

$$\mathcal{F}(n) \doteq \sup_{|\mathcal{S}|=n} |\mathcal{F}(\mathcal{S})|.$$

One says that $\mathcal{F}$ shatters $\mathcal{S}$ if $|\mathcal{F}(\mathcal{S})| = 2^n$, *i.e.*, every possible labeling is realized by some function $f \in \mathcal{F}$. The VC dimensionality of a function class $\mathcal{F}$ is the maximum number of data points such that a dataset of that size is shattered by $\mathcal{F}$,

$$\mathrm{VC}(\mathcal{F}) \doteq \max_{n \in \mathbb{N}} \sup_{|\mathcal{S}|=n} \mathbb{1}\{\mathcal{F}(n) = 2^n\}.$$

Let $\hat{\ell} : \mathcal{F} \to \mathbb{R}$ be the empirical loss function and $\ell : \mathcal{F} \to \mathbb{R}$ the expected loss function. Under uniform convergence, one can prove the VC inequality,

$$\mathcal{P}\left(\sup_{f \in \mathcal{F}} |\hat{\ell}(f) - \ell(f)| > \epsilon\right) \leq 8|\mathcal{F}(n)| \exp\left(-\frac{n\epsilon^2}{32}\right).$$

Here, $|\hat{\ell} - \ell|$ is the generalization error. Note that if the VC dimensionality of $\mathcal{F}$ is bounded, then in the large sample size limit, the generalization error is bounded. In words, using this theorem, no generalization guarantees can be given if $\mathcal{F}$ can be fit to any labeling.

Randomization experiments. Zhang et al. [2021] experimented with the CIFAR-10 dataset, which is a classification dataset that labels images as one-of-ten classes. They observed the following

1. Deep neural networks can perfectly fit the training data and obtain a competitive test error;
2. When randomly replacing training labels, the models can still perfectly fit the data. This shows that the training data has been memorized perfectly;
3. The training time does not increase much when the labels are randomized, so it is not hard to find memorization solutions;
4. When randomly shuffling pixels, the models can also perfectly fit the data. Hence, the inductive bias of convolutional networks does not provide much benefit in this regard.

These findings are unexplainable by learning theory. Since perfect classification is possible on random labelings, the model must have infinite capacity and hence the VC dimensionality is unbounded. As a result, the VC inequality cannot be applied to explain the generalization of the first observation.

Double descent. In recent years, the double descent phenomenon has been observed in overparameterized models. At first, the model will overfit at some point when it memorizes the training data. However, beyond that point, large models will eventually start getting even better test results than before the model overfit—see Figure 11.1.

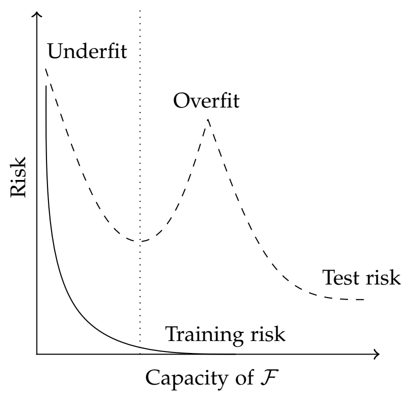


Figure 11.1. Double descent curves.

Flat minima. The flatness of local minima are linked to their generalization ability [Hochreiter and Schmidhuber, 1997]. If a minima is flat, small perturbations in the parameters will only have a small effect on performance—see Figure 11.2. Keskar et al. [2016] showed that small-batch stochastic gradient descent leads to flatter minima, because of the introduced stochasticity. Similarly, weight averaging also improves flatness of found optima [Izmailov et al., 2018]. Furthermore, entropy stochastic gradient descent is specifically designed to find flat minima by introducing Langevin dynamics to favor optima with high entropy [Chaudhari et al., 2019].

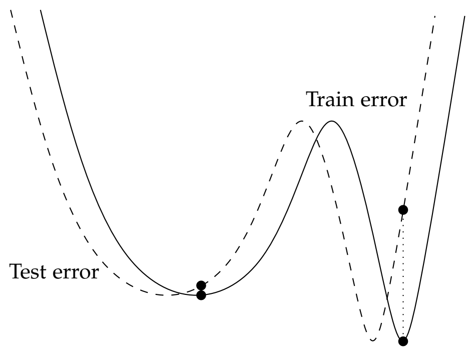


Figure 11.2. Flat minima lead to better generalization than sharp minima.

11.2 PAC Bayesian

Consider the 0/1 loss of a function f on a sample x with binary label $y \in \{-1, 1\}$,

$$\mathbb{1}\{f(x) \neq y\} = \frac{1 - yf(x)}{2}.$$

Lemma 11.1. For any $p \gg q$ and p-measurable X,

$$\mathbb{E}_q[X] \leq D_{\mathrm{KL}}(q \| p) + \log \mathbb{E}_p[\exp(X)].$$

Proof.

$$\begin{aligned}
\log \mathbb{E}_p[\exp(X)] &\geq \log \mathbb{E}_q\left[\exp(X)\frac{p(X)}{q(X)}\right] && p \geq q. \\
&\geq \mathbb{E}_q\left[\log\left(\exp(X)\frac{p(X)}{q(X)}\right)\right] && \text{Jensen's inequality} \\
&= \mathbb{E}_q[X] - \mathbb{E}_q\left[\log\frac{p(X)}{q(X)}\right] \\
&= \mathbb{E}_q[X] - D_{\mathrm{KL}}(q \| p).
\end{aligned}$$

■

Theorem 11.2. For a fixed p, any q, and $\epsilon \in (0,1)$, we have the following with probability greater than or equal to ϵ,

$$\mathbb{E}_q[\ell(f)] - \mathbb{E}_q[\hat{\ell}(f)] \leq \sqrt{\frac{2}{n}\left(D_{\mathrm{KL}}(q\|p) + \log\frac{2\sqrt{n}}{\epsilon}\right)}.$$

Here, p is a prior over the parameters, q is the posterior computed from n datapoints, and we bound the expected generalization gap over stochastic classifiers. This ensures a rate of $\mathcal{O}(1/\sqrt{n})$ on the generalization error without any hidden constants. However, this bound only applies to stochastic classifiers, not single classifiers.

PAC Bayesian learning. Let the prior be a Gaussian over parameters,

$$p = \mathcal{N}(\mathbf{0}, \lambda^2 \boldsymbol{I}).$$

Then, parameterize a Gaussian model distribution,

$$q = \mathcal{N}(\boldsymbol{\theta}, \sigma^2).$$

Since the prior and posterior are Gaussian, we can compute their KL divergence in closed form,

$$D_{\mathrm{KL}}(q\|p) = \sum_{i=1}^{p} \log\frac{\lambda}{\sigma_i} + \frac{\sigma_i^2 + \theta_i^2}{2\lambda^2} - \frac{1}{2}.$$

We can then optimize the PAC Bayes bound to optimize the expected risk,

$$\ell_{\mathrm{PAC}}(q) \doteq \mathbb{E}_q[\hat{\ell}] + \sqrt{\frac{2}{n}\left(D_{\mathrm{KL}}(q\|p) + \log\frac{2\sqrt{n}}{\epsilon}\right)}.$$

A good choice of q involves one that achieves small empirical error $\hat{\ell}$ in expectation and is close to the prior. So, the PAC Bayes loss function effectively adds a regularization term. Generally, the prior has large variance, so q must balance good empirical performance and retaining high variance such that small parameter perturbations do not influence performance.

Dziugaite and Roy [2017] proposed applying stochastic gradient descent to the posterior distribution q over parameters,

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \eta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \hat{\ell}(\tilde{\boldsymbol{\theta}}), \quad \boldsymbol{\sigma} = \boldsymbol{\sigma} - \eta \boldsymbol{\nabla}_{\boldsymbol{\sigma}} \hat{\ell}(\tilde{\boldsymbol{\theta}}), \quad \tilde{\boldsymbol{\theta}} \sim \mathcal{N}(\boldsymbol{\theta}, \boldsymbol{\sigma}^2).$$

Backpropagation can be done by making use of the reparameterization trick,

$$\tilde{\boldsymbol{\theta}} = \boldsymbol{\theta} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}).$$

12 *Generative models*

12.1 *Autoencoders*

Autoencoders map datapoints to latents with an encoder $\boldsymbol{\mathcal{E}}$ and latents to their respective datapoint with a decoder $\boldsymbol{\mathcal{D}}$,

$$\boldsymbol{\mathcal{E}} : \mathcal{X} \to \mathcal{Z}, \quad \boldsymbol{\mathcal{D}} : \mathcal{Z} \to \mathcal{X}.$$

We want these to satisfy $\boldsymbol{\mathcal{D}}(\boldsymbol{\mathcal{E}}(\boldsymbol{x})) = \boldsymbol{x}$. Generally, the latent space is smaller than the data space. As a result, new data points can be sampled by sampling the latent space and using the decoder.

Linear autoencoder. The simplest autoencoder is linear,

$$\boldsymbol{\mathcal{E}}(\boldsymbol{x}) = \boldsymbol{E}\boldsymbol{x}, \quad \boldsymbol{\mathcal{D}}(\boldsymbol{z}) = \boldsymbol{D}\boldsymbol{z}, \quad \boldsymbol{E} \in \mathbb{R}^{m \times d}, \boldsymbol{D} \in \mathbb{R}^{d \times m},$$

where generally $m \ll d$. The reconstruction loss is

$$\ell[\boldsymbol{E}, \boldsymbol{D}](\boldsymbol{x}) = \frac{1}{2}\|\boldsymbol{x} - \hat{\boldsymbol{x}}\|^2 = \frac{1}{2}\|(\boldsymbol{I} - \boldsymbol{D}\boldsymbol{E})\boldsymbol{x}\|^2.$$

It can be shown that the solution to this loss is performing PCA (***P****rincipal* ***C****omponent* ***A****nalysis*) on the covariance matrix $\frac{1}{n}\boldsymbol{X}^\top\boldsymbol{X} \in \mathbb{R}^{d \times d}$ and taking the m principal eigenvectors. The intuition behind this is that we want to retain as much variance in the data as possible.

Variational autoencoder. VAEs (***V****ariational* ***A****uto****e****ncoders*) [Kingma, 2013] perform inference by sampling a latent from a prior and decoding it,

$$\boldsymbol{x} = \boldsymbol{\mathcal{D}}[\boldsymbol{\theta}](\boldsymbol{z}), \quad \boldsymbol{z} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}_m).$$

However, the question is how to optimize the decoder. In generative modeling, we want to optimize the likelihood of the data. However, this is not possible for the VAE, so we must bound it using the ELBO (***E****vidence* ***L****ower* ***Bo****und*),

$$\begin{aligned}
\log p[\boldsymbol{\theta}](\boldsymbol{x}) &= \log \int p[\boldsymbol{\theta}](\boldsymbol{x}, \boldsymbol{z})\mathrm{d}\boldsymbol{z} \\
&= \log \int p[\boldsymbol{\theta}](\boldsymbol{x} \mid \boldsymbol{z})p(\boldsymbol{z})\mathrm{d}\boldsymbol{z} \\
&= \log \int q(\boldsymbol{z})\left(p[\boldsymbol{\theta}](\boldsymbol{x} \mid \boldsymbol{z})\frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\right)\mathrm{d}\boldsymbol{z} \\
&= \log \mathbb{E}_q\left[p[\boldsymbol{\theta}](\boldsymbol{x} \mid \boldsymbol{z})\frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\right] \\
&\geq \mathbb{E}_q\left[\log\left(p[\boldsymbol{\theta}](\boldsymbol{x} \mid \boldsymbol{z})\frac{p(\boldsymbol{z})}{q(\boldsymbol{z})}\right)\right] \\
&= \mathbb{E}_q[\log p[\boldsymbol{\theta}](\boldsymbol{x} \mid \boldsymbol{z})] - \mathbb{E}_q\left[\log \frac{q(\boldsymbol{z})}{p(\boldsymbol{z})}\right] \\
&= \mathbb{E}_q[\log p[\boldsymbol{\theta}](\boldsymbol{x} \mid \boldsymbol{z})] - D_{\mathrm{KL}}(q\|p) \\
&= \mathbb{E}_{p[\boldsymbol{\vartheta}](\boldsymbol{z}|\boldsymbol{x})}[\log p[\boldsymbol{\theta}](\boldsymbol{x} \mid \boldsymbol{z})] - D_{\mathrm{KL}}(p(\boldsymbol{z} \mid \boldsymbol{x})\|p(\boldsymbol{z})).
\end{aligned}$$

Sum rule, where $\boldsymbol{z}$ could be anything.

Product rule. The conditional distribution is induced by the decoder with parameters $\boldsymbol{\theta}$.

q can be any distribution over $\boldsymbol{z}$, which we will make use of later.

Jensen's inequality.

Since this holds for any distribution over $\boldsymbol{z}$, we can replace q by $p[\boldsymbol{\vartheta}](\boldsymbol{z} \mid \boldsymbol{x})$—this distribution is induced by the encoder with parameters $\boldsymbol{\vartheta}$.

Thus, to maximize the log-likelihood of the data, we maximize the ELBO. The ELBO can be seen as a reconstruction loss with a regularization term—the KL divergence makes sure that the predicted distributions over the latent variables stays close to the prior. The advantage of this is that we get a well-behaving latent space, where the distribution over latents for x is an area rather than a single point. Then, the decoder will see a greater variety of latents during training, making it perform well on all latents around the prior. As a result, when we want to sample from the autoencoder, we can sample z from the prior and get a good reconstruction $x = \mathcal{D}(z)$. If every training data point instead only had to cover a single point in the latent space, a sampled latent $z \sim p$ would likely not have been seen before by the decoder and thus give a bad sample.

In general, the posterior $p[\boldsymbol{\vartheta}](z \mid x)$ is intractable, so we restrict it to the Gaussian family of distributions,

$$z \mid x \sim \mathcal{N}(\boldsymbol{\mu}[\boldsymbol{\vartheta}](x), \boldsymbol{\Sigma}[\boldsymbol{\vartheta}](x))$$

Generally, the prior is given as

$$z \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I}).$$

Consequently, the KL divergence can be computed in closed form,

$$D_{\mathrm{KL}}(p[\boldsymbol{\vartheta}](z \mid x) \| p) = \frac{1}{2}\Big(\|\boldsymbol{\mu}[\boldsymbol{\vartheta}](x)\|^2 + \operatorname{tr}(\boldsymbol{\Sigma}[\boldsymbol{\vartheta}](x)) - \log\det(\boldsymbol{\Sigma}[\boldsymbol{\vartheta}](x)) - m\Big).$$

The parameters of the encoder $\boldsymbol{\vartheta}$ can be optimized by making use of the reparameterization trick.

12.2 *Generative adversarial networks*

Optimizing the likelihood of the data is not the only way to train a generative model—there are many ways to provide a training signal to a generative model. In GANs (*G*enerative *A*dversarial *N*etworks) [Goodfellow et al., 2020], we introduce a classifier that distinguishes between samples from "nature" and the generator—see Figure 12.1. The high-level goal of the generator is to "fool" the discriminator into "thinking" that its generations come from "nature". The training signal thus comes from an adversarial model.

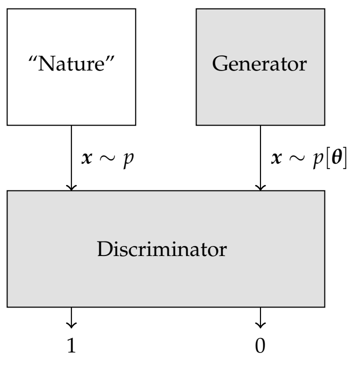


Figure 12.1. The loss function of the generator is adversarial in GANs—the generator wants the discriminator to "think" that its generations come from "nature".

Formally, the discriminator is a binary classifier where 1 means that the sample is real and 0 that it is not. We have the following augmented distribution over samples and their label,

$$\tilde{p}(x, y) = \frac{1}{2}(yp(x) + (1-y)p[\boldsymbol{\theta}](x)), \quad y \in \{0, 1\},$$

where p is the true probability distribution and $p[\boldsymbol{\theta}]$ denotes the model's implicit distribution over samples. The theoretical Bayes-optimal classifier is

$$\mathbb{P}(y = 1 \mid x) = \frac{\mathbb{P}(x \mid y = 1)\mathbb{P}(y = 1)}{\mathbb{P}(x)} = \frac{p(x)}{p(x) + p[\boldsymbol{\theta}](x)}.$$

Assume a 1-to-1 ratio of samples, so the prior is 1/2.

We can then train the generator to minimize the logistic log-likelihood of the Bayes-optimal discriminator,

$$\begin{aligned}
\ell^\star(\boldsymbol{\theta}) &= \mathbb{E}_{\boldsymbol{x},y\sim\tilde{p}}[y\log\mathbb{P}(y=1\mid\boldsymbol{x}) + (1-y)\log(1-\mathbb{P}(y=1\mid\boldsymbol{x}))] \\
&= \mathbb{E}_{\boldsymbol{x},y\sim\tilde{p}}\left[y\log\frac{p(\boldsymbol{x})}{p(\boldsymbol{x})+p[\boldsymbol{\theta}](\boldsymbol{x})} + (1-y)\log\frac{p[\boldsymbol{\theta}](\boldsymbol{x})}{p(\boldsymbol{x})+p[\boldsymbol{\theta}](\boldsymbol{x})}\right] \\
&= \mathbb{E}_p\left[\log\frac{p(\boldsymbol{x})}{p(\boldsymbol{x})+p[\boldsymbol{\theta}](\boldsymbol{x})}\right] + \mathbb{E}_{p[\boldsymbol{\theta}]}\left[\log\frac{p[\boldsymbol{\theta}](\boldsymbol{x})}{p(\boldsymbol{x})+p[\boldsymbol{\theta}](\boldsymbol{x})}\right] \\
&= \mathbb{E}_p[\log p(\boldsymbol{x})] - \mathbb{E}_p[\log(p(\boldsymbol{x})+p[\boldsymbol{\theta}](\boldsymbol{x}))] \\
&\quad + \mathbb{E}_{p[\boldsymbol{\theta}]}[\log p[\boldsymbol{\theta}](\boldsymbol{x})] - \mathbb{E}_{p[\boldsymbol{\theta}]}[\log(p(\boldsymbol{x})+p[\boldsymbol{\theta}](\boldsymbol{x}))] \\
&= -\frac{1}{2}H(p) - \frac{1}{2}H(p[\boldsymbol{\theta}]) + H\left(\frac{1}{2}(p+p[\boldsymbol{\theta}])\right) - \log 2 \\
&= D_{\mathrm{JS}}(p\|p[\boldsymbol{\theta}]) - \log 2.
\end{aligned}$$

Combine the two negative terms into an expectation w.r.t. $\tilde{p}$. Then, get the argument of the logarithm to be $\tilde{p}$ to get the entropy term of $\tilde{p} = \frac{1}{2}(p+p[\boldsymbol{\theta}])$.

In conclusion, minimizing the logistic log-likelihood of the optimal classifier leads to a loss function that is the discrepancy between the nature distribution and the model distribution.

Generally, the Bayes-optimal classifier is intractable. Instead, we parameterize a discriminator,

$$q[\boldsymbol{\varphi}] : \mathcal{X} \to [0,1], \quad \boldsymbol{\varphi} \sim \Phi.$$

Since this model cannot be better than the Bayes-optimal classifier, we have the following bound,

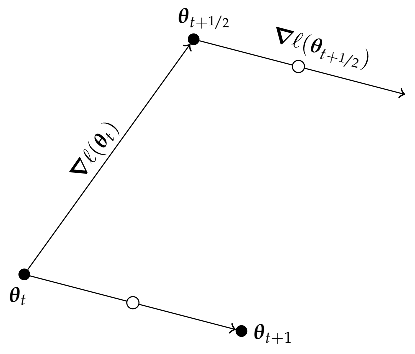
$$\ell^\star(\boldsymbol{\theta}) \geq \sup_{\boldsymbol{\varphi}\in\Phi} \ell(\boldsymbol{\theta},\boldsymbol{\varphi}),$$

where

$$\ell(\boldsymbol{\theta},\boldsymbol{\varphi}) \doteq \mathbb{E}_{\tilde{p}}[y\log q[\boldsymbol{\varphi}](\boldsymbol{x}) + (1-y)\log(1-q[\boldsymbol{\varphi}](\boldsymbol{x}))].$$

In practice, we thus first optimize the discriminator and then the generator,

$$\boldsymbol{\theta}^\star,\boldsymbol{\varphi}^\star \in \operatorname*{argmin}_{\boldsymbol{\theta}\in\Theta}\operatorname*{argmax}_{\boldsymbol{\varphi}\in\Phi} \ell(\boldsymbol{\theta},\boldsymbol{\varphi}).$$



$\boldsymbol{\theta}_{t+1/2}$
$\nabla\ell(\boldsymbol{\theta}_{t+1/2})$
$\nabla\ell(\boldsymbol{\theta}_t)$
$\boldsymbol{\theta}_t$
$\boldsymbol{\theta}_{t+1}$

Figure 12.2. Illustration of the extragradient algorithm for updating $\boldsymbol{\theta}$.

This can also be interpreted as a two-player zero-sum game—we need the extragradient optimization algorithm,

$$\begin{aligned}
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \eta\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta}_{t+1/2},\boldsymbol{\varphi}_t), \quad \boldsymbol{\theta}_{t+1/2} \doteq \boldsymbol{\theta}_t - \eta\nabla_{\boldsymbol{\theta}}\ell(\boldsymbol{\theta}_t,\boldsymbol{\varphi}_t) \\
\boldsymbol{\varphi}_{t+1} &= \boldsymbol{\varphi}_t + \eta\nabla_{\boldsymbol{\varphi}}\ell(\boldsymbol{\theta}_t,\boldsymbol{\varphi}_{t+1/2}), \quad \boldsymbol{\varphi}_{t+1/2} \doteq \boldsymbol{\varphi}_t + \eta\nabla_{\boldsymbol{\varphi}}\ell(\boldsymbol{\theta}_t,\boldsymbol{\varphi}_t).
\end{aligned}$$

This is necessary, because alternating gradient descent/ascent is not guaranteed to converge to local optima.

In practice, it is also necessary to set the loss function of the generator to be the following,

$$\ell(\boldsymbol{\theta}\mid\boldsymbol{\varphi}) = \mathbb{E}_{p[\boldsymbol{\theta}]}[-\log q[\boldsymbol{\varphi}](\boldsymbol{x})].$$

Initially, it was $\mathbb{E}_{p[\boldsymbol{\theta}]}[\log(1-q_{\boldsymbol{\varphi}}(\boldsymbol{x}))]$.

As a result, the generator does not saturate when its performance is poor, because the gradient is bounded as $q_{\boldsymbol{\varphi}}(\boldsymbol{x}) \to 1$.

12.3 *Diffusion models*

As in VAEs, we want to map a simple distribution to a target distribution over the data,

$$\pi \mapsto p[\boldsymbol{\theta}] \approx p,$$

where π is the simple distribution. Whereas VAEs do this in a single pass, diffusion models do it incrementally,

$$\pi = \pi_T \mapsto \pi_{T-1} \mapsto \cdots \mapsto \pi_0 \approx p.$$

Stochastic differential equation view. One can view the diffusion process in a continuous time by an SDE (*Stochastic **D**ifferential **E**quation*) [Song and Ermon, 2019, Song et al., 2020],

$$\mathrm{d}\boldsymbol{x}_t = -\frac{1}{2}\beta_t \boldsymbol{x}_t \mathrm{d}t + \sqrt{\beta_t}\mathrm{d}\mathbb{W}_t,$$

where $\mathrm{d}\mathbb{W}_t$ is a Wiener process.[18] The time-reversed SDE can be computed by the following,

[18] The Wiener process $\mathrm{d}\mathbb{W}_t$ is a Gaussian distribution with variance $\mathrm{d}t$.

$$\mathrm{d}\boldsymbol{x}_t = \left(-\frac{1}{2}\beta_t \boldsymbol{x}_t - \beta_t \boldsymbol{\nabla}_{\boldsymbol{x}_t} \log q_t(\boldsymbol{x}_t)\right)\mathrm{d}t + \sqrt{\beta_t}\mathrm{d}\tilde{\mathbb{W}}_t,$$

where $\mathrm{d}\tilde{\mathbb{W}}_t$ is the reversed Wiener process. Intuitively, denoising amounts to approximating a vector field over the gradient of the probability distribution—moving towards areas with high probability density. Effectively, we are performing gradient ascent on the log-probability perturbed by a Wiener noise process. Hence, we can think of diffusion models as approximating the score function $\boldsymbol{\nabla} \log p(\boldsymbol{x})$.

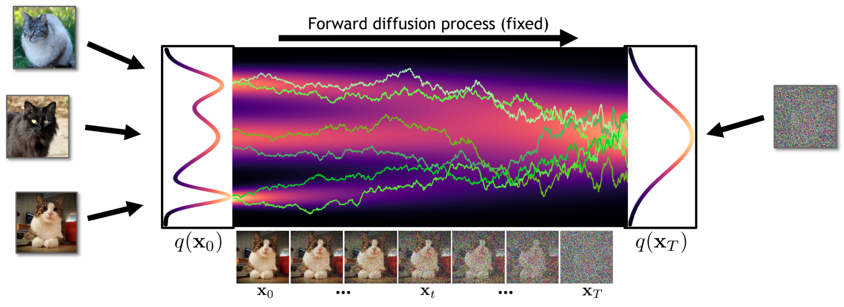


Figure 12.3. Diffusion models through the lens of SDEs [Song et al., 2020].

Evidence lower bound view. One can also present diffusion models in a less involved way by deriving an ELBO. Let the following be the forward process,

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

Note that the energy of the stochastic process evolves as follows,

$$\mathbb{E}\left[\|\boldsymbol{x}_t\|^2 \;\middle|\; \boldsymbol{x}_{t-1}\right] = (1-\beta_t)\|\boldsymbol{x}_{t-1}\|^2 + \beta_t \mathrm{tr}(\boldsymbol{I}).$$

Fixing the scale of the data such that $\mathbb{E}\left[\|\boldsymbol{x}_0\|^2\right] = \operatorname{tr}(\boldsymbol{I}) = \dim(\boldsymbol{x}_0)$ results in energy conservation.

Let q be the kernel of the forward diffusion Markov chain and $p[\boldsymbol{\theta}]$ the learned kernel for the time-reversed one. Furthermore, let p be the actual distribution over data points. Then, the log-likelihood of a sample $\boldsymbol{x}_0 \sim p$ can be lower bounded as follows,

$$\begin{aligned}
\log p[\boldsymbol{\theta}](\boldsymbol{x}_0) &= \log \int p[\boldsymbol{\theta}](\boldsymbol{x}_{0:T}) \mathrm{d}\boldsymbol{x}_{1:T} && \text{Sum rule.} \\
&= \log \int q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0) \frac{p[\boldsymbol{\theta}](\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)} \mathrm{d}\boldsymbol{x}_{1:T} \\
&\geq \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\log \frac{p[\boldsymbol{\theta}](\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T} \mid \boldsymbol{x}_0)}\right] && \text{Jensen's inequality.} \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\sum_{t=0}^{T} \log p[\boldsymbol{\theta}](\boldsymbol{x}_t \mid \boldsymbol{x}_{t+1:T}) - \sum_{t=1}^{T} \log q(\boldsymbol{x}_t \mid \boldsymbol{x}_0, \boldsymbol{x}_{1:t-1})\right] && \text{Product rule.} \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\sum_{t=0}^{T} \log p[\boldsymbol{\theta}](\boldsymbol{x}_t \mid \boldsymbol{x}_{t+1}) - \sum_{t=1}^{T} \log q(\boldsymbol{x}_t \mid \boldsymbol{x}_0, \boldsymbol{x}_{t-1})\right] && \text{Markov property.} \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\Bigg[\log p[\boldsymbol{\theta}](\boldsymbol{x}_0 \mid \boldsymbol{x}_1) + \log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T \mid \boldsymbol{x}_0)} \\
&\qquad + \sum_{t=1}^{T-1} \log \frac{p[\boldsymbol{\theta}](\boldsymbol{x}_t \mid \boldsymbol{x}_{t+1})}{q(\boldsymbol{x}_t \mid \boldsymbol{x}_0, \boldsymbol{x}_{t-1})}\Bigg] \\
&= \mathbb{E}_q[\log p[\boldsymbol{\theta}](\boldsymbol{x}_0 \mid \boldsymbol{x}_1)] + \sum_{t=1}^{T-1} \mathbb{E}_q\left[\log \frac{p[\boldsymbol{\theta}](\boldsymbol{x}_t \mid \boldsymbol{x}_{t+1})}{q(\boldsymbol{x}_t \mid \boldsymbol{x}_0, \boldsymbol{x}_{t-1})}\right] \\
&\qquad + \mathbb{E}_q\left[\log \frac{p(\boldsymbol{x}_T)}{q(\boldsymbol{x}_T \mid \boldsymbol{x}_0)}\right] \\
&= \mathbb{E}_q[\log p[\boldsymbol{\theta}](\boldsymbol{x}_0 \mid \boldsymbol{x}_1)] - \sum_{t=1}^{T-1} D_{\mathrm{KL}}(q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{x}_0) \| p[\boldsymbol{\theta}](\boldsymbol{x}_t \mid \boldsymbol{x}_{t+1})) \\
&\qquad - D_{\mathrm{KL}}(q(\boldsymbol{x}_T \mid \boldsymbol{x}_0) \mid \pi).
\end{aligned}$$

We can divide this up into loss terms (that we want to maximize) per timestep,

$$\ell_t = \begin{cases} \mathbb{E}_q[\log p[\boldsymbol{\theta}](\boldsymbol{x}_0 \mid \boldsymbol{x}_1)] & t = 0 \\ -D_{\mathrm{KL}}(q(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}, \boldsymbol{x}_0) \| p[\boldsymbol{\theta}](\boldsymbol{x}_t \mid \boldsymbol{x}_{t+1})) & 0 < t < T \\ -D_{\mathrm{KL}}(q(\boldsymbol{x}_T \mid \boldsymbol{x}_0) \mid \pi) & t = T. \end{cases}$$

Here, the KL divergences can analytically be computed, because all q are Gaussians and if the steps β_t are small enough, the reverse distributions $p[\boldsymbol{\theta}]$ can be accurately approximated by Gaussians—this is generally how they are parameterized,

$$\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t \sim \mathcal{N}(\boldsymbol{\mu}[\boldsymbol{\theta}](\boldsymbol{x}_t, t), \boldsymbol{\Sigma}[\boldsymbol{\theta}](\boldsymbol{x}_t, t)).$$

Often, the covariance matrix is fixed and only the mean is predicted.

Entropy bounds. Using conditional entropy, we can derive the following,

$$H(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t) = H(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}) + H(\boldsymbol{x}_{t-1}) - H(\boldsymbol{x}_t).$$

Since the unit distribution is the maximum entropy distribution, we have the following entropy bounds between timesteps,

$$H(\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t) \leq H(\boldsymbol{x}_t \mid \boldsymbol{x}_{t-1}).$$

As such, the entropy of the reverse process is bounded by the entropy of the forward process.

Simplified model. Consider a noise schedule $\{\beta_t\}_{t=1}^T$ and define

$$\bar{\alpha}_t \doteq \prod_{\tau=1}^{t} (1 - \beta_\tau), \quad \bar{\beta}_t \doteq 1 - \bar{\alpha}_t.$$

Using these, we can compute the forward process in closed form at any timestep,

$$\boldsymbol{x}_t \sim \mathcal{N}(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, \bar{\beta}_t \boldsymbol{I}).$$

Furthermore, the q targets in the ELBO can be derived to be the following,

$$\boldsymbol{x}_{t-1} \mid \boldsymbol{x}_t, \boldsymbol{x}_0 \sim \mathcal{N}(\boldsymbol{\mu}(\boldsymbol{x}_t, \boldsymbol{x}_0, t), \tilde{\beta}_t \boldsymbol{I}).$$

where

$$\boldsymbol{\mu}(\boldsymbol{x}_t, \boldsymbol{x}_0, t) \doteq \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1 - \bar{\alpha}_t}\boldsymbol{x}_0 + \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t}\sqrt{1 - \beta_t}\boldsymbol{x}_t, \quad \tilde{\beta}_t \doteq \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t}\beta_t.$$

Now, the divergences for $0 < t < T$ in the ELBO simplify to

$$\ell_t = -\frac{1}{2\sigma_t^2}\|\boldsymbol{\mu}(\boldsymbol{x}_t, \boldsymbol{x}_0, t) - \boldsymbol{\mu}[\boldsymbol{\theta}](\boldsymbol{x}_t, t)\|^2,$$

where $\sigma_t^2 \in [\beta_t, \tilde{\beta}_t]$ is the chosen fixed variance of the backward process.

We can also use a different definition of $\boldsymbol{\mu}(\boldsymbol{x}_t, \boldsymbol{x}_0, t)$ by noting the forward process,

$$\boldsymbol{x}_t = \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \sqrt{1 - \bar{\alpha}_t}\boldsymbol{\epsilon}, \quad \boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I})$$

Rewriting yields

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

As such, we can write $\boldsymbol{\mu}(\boldsymbol{x}_t, \boldsymbol{x}_0, t)$ as

$$\boldsymbol{\mu}(\boldsymbol{x}_t, \boldsymbol{x}_0, t) = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t - \frac{\beta_t}{\sqrt{1 - \bar{\alpha}_t}}\boldsymbol{\epsilon}\right).$$

Note that $\boldsymbol{\epsilon}$ fully determines $\boldsymbol{x}_t$ and $\boldsymbol{x}_0$ is constant. Hence, the backward process actually only needs to predict $\boldsymbol{\epsilon}$ by a network $\boldsymbol{\epsilon}[\boldsymbol{\theta}](\boldsymbol{x}_t, t)$. Using this observation a new loss function can be derived,

$$\mathbb{E}_q[\mathcal{L}_t \mid \boldsymbol{x}_0] = \mathbb{E}_\epsilon\left[\lambda(t)\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}[\boldsymbol{\theta}](\boldsymbol{x}_t, t)\|^2\right], \quad \lambda(t) \doteq \frac{\beta_t^2}{2\sigma_t^2\alpha_t(1 - \bar{\alpha}_t)}.$$

In practice, the full loss is often approximated by

$$\ell(\boldsymbol{\theta} \mid \boldsymbol{x}_0) = \frac{1}{T}\sum_{t=1}^{T} \mathbb{E}_{\boldsymbol{\epsilon}}\left[\|\boldsymbol{\epsilon} - \boldsymbol{\epsilon}[\boldsymbol{\theta}](\boldsymbol{x}_t, t)\|^2 \;\middle|\; \boldsymbol{x}_0\right].$$

We make a Monte Carlo approximation of this loss by uniformly sampling a timestep t and sampling noise $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$.

13 Adversarial attacks

In adversarial attacks, the attacker wants to make small changes to the input of the model such that the model gives different results. This can have negative consequences in use cases such as automated driving, where the traffic signs may be perturbed to give the wrong signals to the car. Furthermore, it could also be used in medical classification and segmentation. Such attacks hint at fundamental differences in human and machine vision.

13.1 p-norm robustness

Consider a multi-class classifier,

$$f : \mathbb{R}^d \to \{1, \dots, m\}.$$

The goal of an adversarial attack is to find a perturbation $\boldsymbol{\eta}$ such that

$$f(\boldsymbol{x}+\boldsymbol{\eta}) \neq f(\boldsymbol{x}), \quad \|\boldsymbol{\eta}\|_p \leq \epsilon,$$

where

$$\|\boldsymbol{x}\|_p \doteq \left(\sum_{i=1}^d x_i^p\right)^{1/p}, \quad \|\boldsymbol{x}\|_\infty \doteq \max_{i=1}^d |x_i|, \quad \|\boldsymbol{x}\|_0 \doteq |\{i \mid x_i \neq 0\}|.$$

Using ∞-norm, we get perturbation everywhere, whereas we only change a few pixels if we use 0-norm.

We will focus on $p = 2$ and an affine classifier that we wish to attack,

$$f(\boldsymbol{x}) = \operatorname*{argmax}_{i=1}^m f_i(\boldsymbol{x}), \quad f_i(\boldsymbol{x}) = \boldsymbol{w}_i^\top \boldsymbol{x} + b_i.$$

Further consider a binary classifier—$m = 2$. Assume $\boldsymbol{x}$ is currently classified as the first class, then we want $\boldsymbol{x}+\boldsymbol{\eta}$ to be classified as the second class. We need $f_2(\boldsymbol{x}+\boldsymbol{\eta}) > f_1(\boldsymbol{x}+\boldsymbol{\eta})$, so we have the following convex program,

$$\begin{aligned} \min \quad & \frac{1}{2}\|\boldsymbol{\eta}\|_2^2 \\ \text{subject to} \quad & f_1(\boldsymbol{x}+\boldsymbol{\eta}) \leq f_2(\boldsymbol{x}+\boldsymbol{\eta}). \end{aligned}$$

This can be rewritten to

$$\begin{aligned} \min \quad & \frac{1}{2}\|\boldsymbol{\eta}\|_2^2 \\ \text{subject to} \quad & (\boldsymbol{w}_1 - \boldsymbol{w}_2)^\top(\boldsymbol{x}+\boldsymbol{\eta}) + b_1 - b_2 \leq 0. \end{aligned}$$

The Lagrangian is written as

$$\mathcal{L}(\boldsymbol{\eta}, \lambda) = \frac{1}{2}\|\boldsymbol{\eta}\|_2^2 + \lambda\Big((\boldsymbol{w}_1 - \boldsymbol{w}_2)^\top(\boldsymbol{x}+\boldsymbol{\eta}) + b_1 - b_2\Big).$$

By the KKT conditions, we have

$$\nabla_{\boldsymbol{\eta}}\mathcal{L}(\boldsymbol{\eta}, \lambda) = \boldsymbol{\eta} + \lambda(\boldsymbol{w}_1 - \boldsymbol{w}_2) \overset{!}{=} \mathbf{0}.$$

Hence,

$$\boldsymbol{\eta} = \lambda(\boldsymbol{w}_2 - \boldsymbol{w}_1).$$

We only need to find the minimum $\lambda > 0$ such that

$$\begin{aligned}
& 0 > f_1(\boldsymbol{x} + \lambda(\boldsymbol{w}_2 - \boldsymbol{w}_1)) - f_2(\boldsymbol{x} + \lambda(\boldsymbol{w}_2 - \boldsymbol{w}_1)) \\
\iff \quad & 0 > (\boldsymbol{w}_1 - \boldsymbol{w}_2)^\top (\boldsymbol{x} + \lambda(\boldsymbol{w}_2 - \boldsymbol{w}_1)) + b_1 - b_2 \\
\iff \quad & 0 > f_1(\boldsymbol{x}) - f_2(\boldsymbol{x}) - \lambda \|\boldsymbol{w}_1 - \boldsymbol{w}_2\|_2^2 \\
\iff \quad & \lambda > \frac{f_1(\boldsymbol{x}) - f_2(\boldsymbol{x})}{\|\boldsymbol{w}_1 - \boldsymbol{w}_2\|_2^2}.
\end{aligned}$$

In conclusion the optimal $\boldsymbol{\eta}$ is the following,

$$\boldsymbol{\eta} = \frac{f_1(\boldsymbol{x}) - f_2(\boldsymbol{x})}{\|\boldsymbol{w}_2 - \boldsymbol{w}_1\|_2^2}(\boldsymbol{w}_2 - \boldsymbol{w}_1).$$

This can be generalized to any source class i and target class j—if the target class does not matter, you take the most easily confusable class. In the general case, we can linearize the model,

$$f_i(\boldsymbol{x}) \approx f_i(\tilde{\boldsymbol{x}}) + \langle \boldsymbol{\nabla} f_i(\tilde{\boldsymbol{x}}), \boldsymbol{x} - \tilde{\boldsymbol{x}} \rangle.$$

We choose $\tilde{\boldsymbol{x}}$ to be the current iterate, since that results in the best approximation. Now, $\boldsymbol{\nabla} f_i(\tilde{\boldsymbol{x}})$ acts as $\boldsymbol{w}_i$ and $f_i(\tilde{\boldsymbol{x}})$ as b_i in the previous example.

and iteratively find the optimal adversarial perturbation [Moosavi-Dezfooli et al., 2016]. This is done by iteratively solving the following convex program,

$$\begin{aligned}
\min \quad & \|\boldsymbol{\delta}\|_p \\
\text{subject to} \quad & \big\langle \boldsymbol{\nabla} f_i(\boldsymbol{x}) - \boldsymbol{\nabla} f_j(\boldsymbol{x}), (\boldsymbol{x} + \boldsymbol{\delta}) - \boldsymbol{x} \big\rangle + f_i(\boldsymbol{x}) - f_j(\boldsymbol{x}) \leq 0.
\end{aligned}$$

Then, update the noise,

$$\boldsymbol{\eta}_t = \boldsymbol{\eta}_{t-1} + \boldsymbol{\delta}, \quad \boldsymbol{\eta}_0 = \boldsymbol{0},$$

until $f_j(\boldsymbol{x} + \boldsymbol{\eta}_t) > f_i(\boldsymbol{x} + \boldsymbol{\eta}_t)$.

13.2 Robust training

Robust training is a systemic approach to making models robust to adversarial attacks. It works by extending the loss function to neighborhoods of training points,

$$\ell(\boldsymbol{x}) \mapsto \max_{\boldsymbol{\eta}:\|\boldsymbol{\eta}\|_p \leq \epsilon} \ell(\boldsymbol{x} + \boldsymbol{\eta}).$$

This yields a two-player minimax game, where the adversary picks the worst perturbation and the learner picks the best parameters in response,

$$\operatorname*{argmin}_{\boldsymbol{\theta} \in \Theta} \max_{\boldsymbol{\eta}:\|\boldsymbol{\eta}\|_p \leq \epsilon} \ell(\boldsymbol{x} + \boldsymbol{\eta}).$$

The adversarial task can be solved with projected gradient ascent, *e.g.* when $p = 2$, we get

$$\boldsymbol{\eta}_{t+1} = \epsilon \cdot \Pi(\boldsymbol{\eta}_t + \alpha \cdot \boldsymbol{\nabla}_{\boldsymbol{x}} \ell(\boldsymbol{x} + \boldsymbol{\eta}_t)), \quad \Pi(\boldsymbol{z}) \doteq \frac{\boldsymbol{z}}{\|\boldsymbol{z}\|_2}.$$

And with $p = \infty$, we get

$$\boldsymbol{\eta}_{t+1} = \epsilon \cdot \Pi(\boldsymbol{\eta}_t + \alpha \cdot \operatorname{sgn}(\nabla_{\boldsymbol{x}} \ell(\boldsymbol{x} + \boldsymbol{\eta}_t))), \quad \Pi(\boldsymbol{z}) \doteq \frac{\boldsymbol{z}}{\|\boldsymbol{z}\|_\infty}.$$

The fast gradient sign method [Goodfellow et al., 2014] performs one iteration of projected gradient descent with $p = \infty$, resulting in the following adversarial choice,

$$\boldsymbol{\eta} = \epsilon \cdot \operatorname{sgn}(\nabla_{\boldsymbol{x}} \ell(\boldsymbol{x})).$$

References

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. *arXiv preprint arXiv:2006.05205*, 2020.

Dzmitry Bahdanau. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Andrew R Barron. Universal approximation bounds for superpositions of a sigmoidal function. *IEEE Transactions on Information theory*, 39(3): 930–945, 1993.

Alberto Bietti and Julien Mairal. On the inductive bias of neural tangent kernels. *Advances in Neural Information Processing Systems*, 32, 2019.

Pratik Chaudhari, Anna Choromanska, Stefano Soatto, Yann LeCun, Carlo Baldassi, Christian Borgs, Jennifer Chayes, Levent Sagun, and Riccardo Zecchina. Entropy-sgd: Biasing gradient descent into wide valleys. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12): 124018, 2019.

Deli Chen, Yankai Lin, Wei Li, Peng Li, Jie Zhou, and Xu Sun. Measuring and relieving the over-smoothing problem for graph neural networks from the topological view. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 3438–3445, 2020.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. 2014. URL `https://arxiv.org/abs/1406.1078`.

Thomas M Cover. Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE transactions on electronic computers*, (3):326–334, 1965.

Mete Demircigil, Judith Heusel, Matthias Löwe, Sven Upgang, and Franck Vermet. On a model of associative memory with huge storage capacity. *Journal of Statistical Physics*, 168:288–299, 2017.

Jacob Devlin. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Alexey Dosovitskiy. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.

Gintare Karolina Dziugaite and Daniel M Roy. Computing nonvacuous generalization bounds for deep (stochastic) neural networks with many more parameters than training data. *arXiv preprint arXiv:1703.11008*, 2017.

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

Leo Feng, Frederick Tung, Mohamed Osama Ahmed, Yoshua Bengio, and Hossein Hajimirsadegh. Were rnns all we needed? *arXiv preprint arXiv:2410.01201*, 2024.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

Geoffrey Hinton. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Sepp Hochreiter and Jürgen Schmidhuber. Flat minima. *Neural computation*, 9(1):1–42, 1997.

John J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558, 1982.

Wei Hu, Lechao Xiao, and Jeffrey Pennington. Provable benefit of orthogonal initialization in optimizing deep linear networks. *arXiv preprint arXiv:2001.05992*, 2020.

Sergey Ioffe. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. *arXiv preprint arXiv:1803.05407*, 2018.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in neural information processing systems*, 28, 2015.

Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. *arXiv preprint arXiv:1609.04836*, 2016.

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

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Dmitry Krotov and John J Hopfield. Dense associative memory for pattern recognition. *Advances in neural information processing systems*, 29, 2016.

Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–50. Springer, 2002.

Jaehoon Lee, Lechao Xiao, Samuel Schoenholz, Yasaman Bahri, Roman Novak, Jascha Sohl-Dickstein, and Jeffrey Pennington. Wide neural networks of any depth evolve as linear models under gradient descent. *Advances in neural information processing systems*, 32, 2019.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *ArXiv e-prints*, pages arXiv–1607, 2016.

Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural networks*, 6(6):861–867, 1993.

James L McClelland, David E Rumelhart, PDP Research Group, et al. *Parallel distributed processing, volume 2: Explorations in the microstructure of cognition: Psychological and biological models*, volume 2. MIT press, 1987.

Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5: 115–133, 1943.

Marvin Minsky and Seymour Papert. An introduction to computational geometry. *Cambridge tiass., HIT*, 479(480):104, 1969.

Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the number of linear regions of deep neural networks. *Advances in neural information processing systems*, 27, 2014.

Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2574–2582, 2016.

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 4602–4609, 2019.

Albert BJ Novikoff. On convergence proofs on perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622. New York, NY, 1962.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *International Conference on Machine Learning*, pages 26670–26698. PMLR, 2023.

Allan Pinkus. Approximation theory of the mlp model in neural networks. *Acta numerica*, 8:143–195, 1999.

Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

Hubert Ramsauer, Bernhard Schäfl, Johannes Lehner, Philipp Seidl, Michael Widrich, Thomas Adler, Lukas Gruber, Markus Holzleitner, Milena Pavlović, Geir Kjetil Sandve, et al. Hopfield networks is all you need. *arXiv preprint arXiv:2008.02217*, 2020.

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

David E Rumelhart, Geoffrey E Hinton, James L McClelland, et al. A general framework for parallel distributed processing. *Parallel distributed processing: Explorations in the microstructure of cognition*, 1(45-76): 26, 1986.

Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization? *Advances in neural information processing systems*, 31, 2018.

Andrew M Saxe, James L McClelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural networks. *arXiv preprint arXiv:1312.6120*, 2013.

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

Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing*, 45(11):2673–2681, 1997.

Boris Shekhtman. Why piecewise linear functions are dense in cio, 1. *Journal of Approximation Theory*, 36:265–267, 1982.

Yang Song and Stefano Ermon. Generative modeling by estimating gradients of the data distribution. *Advances in neural information processing systems*, 32, 2019.

Yang Song, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. *arXiv preprint arXiv:2011.13456*, 2020.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1): 1929–1958, 2014.

I Sutskever. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

Wilson L Taylor. "cloze procedure": A new tool for measuring readability. *Journalism quarterly*, 30(4):415–433, 1953.

A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Edward Wagstaff, Fabian Fuchs, Martin Engelcke, Ingmar Posner, and Michael A Osborne. On the limitations of representing functions on sets. In *International Conference on Machine Learning*, pages 6487–6494. PMLR, 2019.

Yonghui Wu. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Russ R Salakhutdinov, and Alexander J Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.

Thomas Zaslavsky. *Facing up to arrangements: Face-count formulas for partitions of space by hyperplanes: Face-count formulas for partitions of space by hyperplanes*, volume 154. American Mathematical Soc., 1975.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning (still) requires rethinking generalization. *Communications of the ACM*, 64(3):107–115, 2021.

Yi Zhu and Shawn Newsam. Densenet for dense flow. In *2017 IEEE* *international conference on image processing (ICIP)*, pages 790–794. IEEE, 2017.