Deep Learning

Cristian Perez Jensen

December 5, 2024

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

Contents

List of symbols

$\doteq$	Equality by definition
$\overset{!}{=}$	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.*, $\{i, i+1, \ldots, j\}$
$f : A \to B$	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 $b \in \mathbb{R}$,

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

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

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

The perceptron thus models the decision boundary as a hyperplane in $\mathbb{R}^n$ with normal vector $w/\|w\|$ and $-b/\|w\|$ 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,

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

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

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

[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.

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.

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

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

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.

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.[2] Given a dataset $\mathcal{S} = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^{s}$, the perceptron algorithms 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 $\mathcal{V}(\mathcal{S}) = \emptyset$.

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 $(\boldsymbol{x}, y) \in \mathcal{S}$, it has the following update rule,

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

We keep going through the dataset until every data point is correctly classified—see Algorithm 1. Note that this algorithm will never converge if $\mathcal{S}$ is not linearly separable.

Proof of convergence. In order to prove convergence of the perceptron algorithm for linearly separable data, we will assume that there is no bias. We denote the weights after t updates of the perceptron algorithm (ignoring correctly classified samples) as $\boldsymbol{w}_t$.

We will first need the following two lemmas,

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

$$\boldsymbol{w}^\top \boldsymbol{w}_t \geq t\gamma.$$

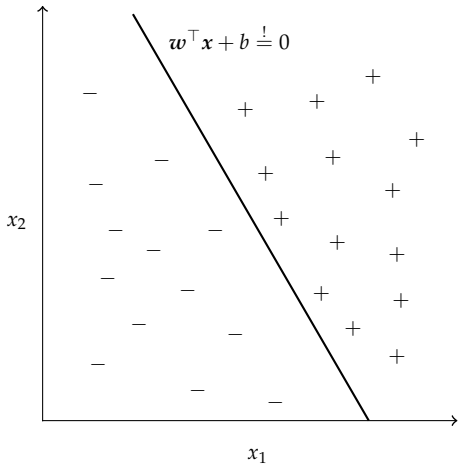

Figure 1.1. Linear separability of negative and positive data points.

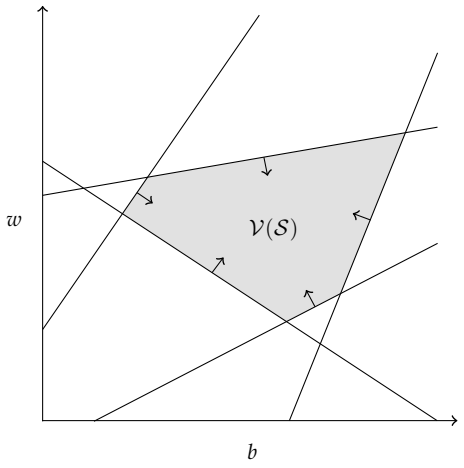

Figure 1.2. In this version space, every line represents a data point's halfspace in which it is correctly classified. As can be seen, adding data points can only shrink the version space.

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

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

Algorithm 1. The perceptron algorithm.

Proof. This can easily be shown by a recursion,

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

Now, it is easy to show the result by induction, starting from $\boldsymbol{w}_0 = \mathbf{0}$. ■

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

$$\|\boldsymbol{w}_t\| \leq R\sqrt{t}.$$

Proof. This can easily be shown by a recursion,

$$\begin{aligned}
\|\boldsymbol{w}_{t+1}\|^2 &= \|\boldsymbol{w}_t + y\boldsymbol{x}\|^2 && \text{Perceptron update.} \\
&= \|\boldsymbol{w}_t\|^2 + \|y\boldsymbol{x}\|^2 + 2y\boldsymbol{w}_t^\top \boldsymbol{x} && \text{Cosine theorem.} \\
&\leq \|\boldsymbol{w}_t\|^2 + \|\boldsymbol{x}\|^2 && \text{The perceptron update condition is } \gamma[\boldsymbol{w}](\boldsymbol{x}, y) \leq 0. \\
&\leq \|\boldsymbol{w}_t\|^2 + R^2.
\end{aligned}$$

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

Theorem 1.3 ([Novikoff, 1962])**.** Let $\mathcal{S}$ 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,

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

where $\boldsymbol{w} \in \mathcal{V}(\mathcal{S})$. Hence,

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

Thus, the number of updates is upper bounded. When there are no more updates, there are no more mistakes—we only make updates when we find a mistake. Hence, $\boldsymbol{w}_t$ will have converged. 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,

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

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

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

We assume points to be in general position, which means that any subset $\Xi \subseteq \mathcal{S}$ with $|\Xi| \leq n$ is linearly independent.[3]

[3] This is a very weak condition.

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

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

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

$$\mathcal{C}(1, n) = 2, \quad \mathcal{C}(s, 1) = 2.$$

Consider a realizable classification of s 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 $\boldsymbol{x}_{s+1}$ be a pattern that we add to $\mathcal{S}$. This gives us two new version spaces,

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

There are two situations,

1. $\mathcal{V}^+$ and $\mathcal{V}^-$ are non-empty. Hence, $\boldsymbol{x}_{s+1}$ 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 $\boldsymbol{w}^\top \boldsymbol{x}_{s+1} = 0$.[4] Recall that we want to know the number of classifications of this new dataset $\mathcal{S} \cup \{\boldsymbol{x}_{s+1}\}$. For any classification of $\mathcal{S}$ that is in this situation, we can make two new classifications; one where $\boldsymbol{x}_{s+1}$ is classified $+1$ or -1. There are $\mathcal{C}(s, n-1)$ such that classifications, because the constraint on $\boldsymbol{w}$ makes the problem effectively $(n-1)$-dimensional with s data points. Hence, we gain $2\mathcal{C}(s, n-1)$ 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 $\mathcal{C}(s, n) - \mathcal{C}(s, n-1)$ such original classifications. Hence, we gain $\mathcal{C}(s, n) - \mathcal{C}(s, n-1)$ classifications.

[4] 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

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

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

It turns out that after $s = 2n$, 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]. The elements of PDP are the following,

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

All of these elements are design choices that can be changed and experimented with. The fact that we still use this wording says much about the impact of PDP.

1.4 Hopfield networks

The Hopfield model [Hopfield, 1982] defines a parameterized energy function via second-order interactions between n binary neurons,

$$H(\boldsymbol{X}) \doteq -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_i j X_i X_j + \sum_{i=1}^{n} b_i X_i, \quad \boldsymbol{X} \in \{-1,+1\}^n.$$

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

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

Hopfield networks follow a simple dynamic,

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

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

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

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,

$$\mathcal{S} \subseteq \{-1,+1\}^n,$$

Hebbian learning involves setting the weights as outer products,

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

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 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,

$$x_{t,i} \stackrel{!}{=} \operatorname{sgn}\left(\sum_{j=1}^{n} w_{ij} x_{t,j}\right).$$

Expanding this with the couplings from Hebbian learning, we get

$$\begin{aligned} x_{t,i} &\stackrel{!}{=} \operatorname{sgn}\left(\frac{1}{n}\sum_{j=1}^{n}\sum_{r=1}^{s} x_{r,i} x_{r,j} x_{t,j}\right) \\ &= \operatorname{sgn}\left(x_{t,i} + \underbrace{\frac{1}{n}\sum_{j=1}^{n}\sum_{r\neq t}^{s} x_{r,i} x_{r,j} x_{t,j}}_{\doteq C_{t,i}}\right). \end{aligned}$$

$C_{t,i}$ is the cross-talk between patterns, and ideally $|C_{t,i}| < 1$, for all patterns $t \in [s]$ and indices $i \in [n]$, 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 $n \to \infty$, then we have

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

The probability of a single sign being flipped is then

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

Hence, the ratio s/n controls the asymptotic error rate. At $s/n \approx 0.138$, 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

$$s \leq \frac{n}{2 \log_2 n}.$$

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 a MSE (***M**ean **S**quared **E**rror*) loss,

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

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

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

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 $\sigma : \mathbb{R} \to (0,1)$,

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

Hence, the model has the following form,

$$f[\boldsymbol{w}](\boldsymbol{x}) = \sigma\Big(\boldsymbol{w}^\top \boldsymbol{x}\Big).$$

This outputs the probability of the label of $\boldsymbol{x}$ being 1. We train this model to optimize the cross-entropy loss,

$$\ell[\boldsymbol{w}](\mathcal{S}) = \frac{1}{s}\sum_{i=1}^{s} -\log\sigma\Big((2y_i - 1)\boldsymbol{w}^\top \boldsymbol{x}_i\Big).$$

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,

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

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,

$$G = F^L\left[\boldsymbol{\theta}^L\right] \circ \cdots \circ F^1\left[\boldsymbol{\theta}^1\right],$$

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),

$$\boldsymbol{x}^\ell \doteq \left(F^\ell \circ \cdots \circ F^1\right)(\boldsymbol{x}) = F^\ell\left(\boldsymbol{x}^{\ell-1}\right).$$

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,

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

where $\boldsymbol{P}$ is a permutation matrix.[5] Hence, the parameters are not unique in feedforward networks.

[5] A permutation matrix $\boldsymbol{P} \in \mathbb{R}^{n\times n}$ satisfies the following condition,

$$\sum_{i=1}^{n} p_{ij} = \sum_{j=1}^{n} p_{ij} = 1, \quad \forall i, j \in [n].$$

The layers—as presented—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,
$$\phi = (z)_+ = \max\{0, z\}.$$

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 softmax, 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.

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,

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

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 and residual networks*

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

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.

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,

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

2.4 *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

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

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

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

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

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

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

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

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.

2.5 *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. The connected regions of these hyperplanes are the activation patterns.

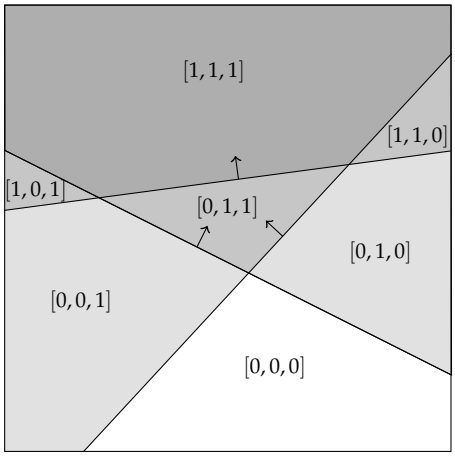

Figure 2.1. Connected regions, partitioned according to activation pattern. Each hyperplane represents a hidden unit. This shows an MLP with 2-dimensional input and 3-dimensional hidden layer.

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 dense in $\mathcal{C}([0,1])$.

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

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

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.

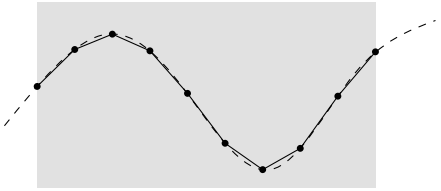

Figure 2.2. Piecewise linear approximation of a continuous function.

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,

$$F[\boldsymbol{\theta}](\boldsymbol{x}) \doteq \left(F^L \circ \cdots \circ F^1\right)(\boldsymbol{x}),$$

with the following hidden layer,

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

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

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

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

$$\boldsymbol{\delta}^L = \frac{\partial \ell(\boldsymbol{y}, \hat{\boldsymbol{y}})}{\partial \hat{\boldsymbol{y}}}, \quad \hat{\boldsymbol{y}} = F[\boldsymbol{\theta}](\boldsymbol{x})$$

$$\boldsymbol{\delta}^\ell = \left[\frac{\boldsymbol{h}^{\ell+1}}{\boldsymbol{h}^\ell}\right]^\top \boldsymbol{\delta}^{\ell+1}.$$

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,

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

3.2 Gradient descent

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

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

$h \doteq \ell \circ F.$

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)**.** h is L-smooth if there exists $L > 0$ such that

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

This is equivalent to the following condition,

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

From the Taylor series expansion, we have

$$
\begin{aligned}
h(\boldsymbol{\theta}_2) - h(\boldsymbol{\theta}_1) &= \boldsymbol{\nabla} h(\boldsymbol{\theta}_1)^\top (\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1) + \frac{1}{2}(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1)^\top \boldsymbol{\nabla}^2 h(\boldsymbol{\theta}_1)(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1) \\
&= -\eta \|\boldsymbol{\nabla} h(\boldsymbol{\theta}_1)\|^2 + \frac{1}{2}(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1)^\top \boldsymbol{\nabla}^2 h(\boldsymbol{\theta}_1)(\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1) && \text{Gradient descent update rule.} \\
&\leq -\eta \|\boldsymbol{\nabla} h(\boldsymbol{\theta}_1)\|^2 + \frac{L}{2}\|\boldsymbol{\theta}_2 - \boldsymbol{\theta}_1\|^2 && \text{Spectral norm condition of smoothness.} \\
&= -\eta \|\boldsymbol{\nabla} h(\boldsymbol{\theta})_1\|^2 + \frac{L\eta^2}{2}\|h(\boldsymbol{\theta})_1\|^2 && \text{Update rule of gradient descent.} \\
&= -\eta\left(1 - \frac{L\eta}{2}\right)\|\boldsymbol{\nabla} h(\boldsymbol{\theta}_1)\|^2.
\end{aligned}
$$

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

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

As a result, we obtain sufficient decrease,

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

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

$$T = \frac{2L}{\epsilon^2}\left(h\left(\boldsymbol{\theta}^0\right) - h(\boldsymbol{\theta}^\star)\right).$$

Proof. TODO ■

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

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

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,

$$h\left(\boldsymbol{\theta}^T\right) - h(\boldsymbol{\theta}^\star) \leq \left(1 - \frac{\mu}{L}\right)^T \left(h\left(\boldsymbol{\theta}^0\right) - h(\boldsymbol{\theta}^\star)\right).$$

Proof.

$$
\begin{aligned}
h\left(\boldsymbol{\theta}^T\right) - h\left(\boldsymbol{\theta}^{T-1}\right) &\leq -\frac{1}{2L}\left\|\boldsymbol{\nabla} h\left(\boldsymbol{\theta}^T\right)\right\|^2 && \text{Sufficient decrease.} \\
&\leq -\frac{\mu}{L}\left(h\left(\boldsymbol{\theta}^T\right) - h(\boldsymbol{\theta}^\star)\right) && \text{PL-inequality.}
\end{aligned}
$$

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

$$h\Big(\boldsymbol{\theta}^T\Big) - h(\boldsymbol{\theta}^\star) \leq \Big(1 - \frac{\mu}{L}\Big)\Big(h\Big(\boldsymbol{\theta}^T\Big) - h(\boldsymbol{\theta}^\star)\Big).$$

The result follows from a trivial induction. ■

3.3 *Acceleration and adaptivity*

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

$$\boldsymbol{\chi}^{t+1} = \boldsymbol{\theta}^t + \beta\Big(\boldsymbol{\theta}^t - \boldsymbol{\theta}^{t-1}\Big) \qquad \text{Extrapolation step.}$$

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\chi}^{t+1} - \eta\boldsymbol{\nabla}h\Big(\boldsymbol{\chi}^{t+1}\Big). \qquad \text{Gradient descent step.}$$

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,

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

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.

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

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

We then have a parameter-specific update rule,

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

Adam (***Ada**ptive **M**oment Estimation*) [Kingma, 2014] combines these two,

$$\boldsymbol{g}_t = \beta\boldsymbol{g}_{t-1} + (1-\beta)\boldsymbol{\nabla}h(\boldsymbol{\theta}_t), \quad \beta \in [0,1] \qquad \text{Moving average (smooth gradient estimator).}$$

$$\boldsymbol{\gamma}_t = \alpha\boldsymbol{\gamma}_{t-1} + (1-\alpha)\boldsymbol{\nabla}h(\boldsymbol{\theta}_t)^{\odot 2}, \quad \alpha \in [0,1]. \qquad \text{Exponential averaging (measure of stability in the optimization landscape).}$$

The update rule is then

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

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 single data point at each timestep.

4 Convolutional networks

4.1 Convolutions

Definition 4.1 (Integral operator)**.**

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

Definition 4.2 (Fourier transform)**.**

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

Special case of integral operator with $H(u,t) = e^{-2\pi i t u}, t_1 = -\infty, t_2 = \infty$.

Definition 4.3 (Convolution)**.**

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

Special case of integral operator with $H(u,t) = h(u-t), t_1 = -\infty, t_2 = \infty$.

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

$$f * h = h * f, \quad \forall f, h.$$

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

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

■

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,

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

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

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

■

The convolutional operator can be computed via the Fourier transform,

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

In the discrete case, this allows computing the convolution with the Fast Fourier Transform algorithm—however, this is not very useful for machine learning.

Theorem 4.6. Any linear shift-equivariant transformation can be written as a convolution with a suitable kernel.

Proof. TODO ■

Definition 4.7 (Discrete convolution)**.** Let $f, h : \mathbb{Z} \to \mathbb{R}$, then

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

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

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

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

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

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

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

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 is effectively a proof that the convolutional operator is linear.

4.2 Convolutional layers

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.

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.

TODO: Gradients.

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 $\boldsymbol{x}^1, \dots, \boldsymbol{x}^T$ denote the observed input sequence, RNNs compute the sequence of activations recursively,

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

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.

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

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

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,

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

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.

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

$$\begin{aligned}\frac{\partial L}{\partial \boldsymbol{z}_t} &= \sum_{i=1}^{T} \frac{\partial \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)}{\partial \boldsymbol{z}_t} \\ &= \sum_{i=t}^{T} \frac{\partial \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)}{\partial \hat{\boldsymbol{y}}_i} \frac{\partial \hat{\boldsymbol{y}}_i}{\partial \boldsymbol{z}_t} \\ &= \sum_{i=t}^{T} \frac{\partial \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)}{\partial \hat{\boldsymbol{y}}_i} \frac{\partial \hat{\boldsymbol{y}}_i}{\partial \boldsymbol{z}_i} \frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{z}_t} \\ &= \sum_{i=t}^{T} \frac{\partial \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)}{\partial \hat{\boldsymbol{y}}_i} \frac{\partial \hat{\boldsymbol{y}}_i}{\partial \boldsymbol{z}_i} \prod_{j=t+1}^{i} \frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{z}_{j-1}} \\ &= \sum_{i=t}^{T} \frac{\partial \ell(\hat{\boldsymbol{y}}_i, \boldsymbol{y}_i)}{\partial \hat{\boldsymbol{y}}_i} \frac{\partial \hat{\boldsymbol{y}}_i}{\partial \boldsymbol{z}_i} \prod_{j=t+1}^{i} \dot{\boldsymbol{\Phi}}_j \boldsymbol{U},\end{aligned}$$

where

$$\dot{\boldsymbol{\Phi}}_j = \operatorname{diag}\big(\phi'(\boldsymbol{U}\boldsymbol{z}_{j-1} + \boldsymbol{V}\boldsymbol{x}_j)\big).$$

This gradient is only stable if

$$\left\| \frac{\partial \boldsymbol{z}_j}{\partial \boldsymbol{z}_{j-1}} \right\|_2 = \left\| \dot{\boldsymbol{\Phi}}_j \boldsymbol{U} \right\|_2 = 1,$$

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

$$\left\| \frac{\partial \boldsymbol{z}_i}{\partial \boldsymbol{z}_t} \right\|_2 \leq (\alpha \|\boldsymbol{U}\|_2)^{i-t} = (\alpha \sigma_1(\boldsymbol{U}))^{i-t}.$$

So, the gradient will vanish if $\sigma_1(\boldsymbol{U}) \geq 1/\alpha$. An analogous argument can be made for exploding gradients.

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

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,

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

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 store information. The LSTM works as follows,

$$\begin{aligned}\boldsymbol{z}_t &= \sigma(\boldsymbol{F}\tilde{\boldsymbol{x}}_t) \odot \boldsymbol{z}_{t-1} + \sigma(\boldsymbol{G}\tilde{\boldsymbol{x}}_t) \odot \tanh(\boldsymbol{V}\tilde{\boldsymbol{x}}_t), \quad \tilde{\boldsymbol{x}}_t = [\boldsymbol{\zeta}_{t-1}, \boldsymbol{x}_t] \\ \boldsymbol{\zeta}_t &= \sigma(\boldsymbol{H}\tilde{\boldsymbol{x}}_t) \odot \tanh(\boldsymbol{U}\boldsymbol{z}_t).\end{aligned}$$

Here, $\boldsymbol{z}_t$ is called the cell state and $\boldsymbol{\zeta}_t$ 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.

The GRU combines the forget and input gates as a convex combination,

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

However, the computation of new storage remains complex,

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

$\boldsymbol{\zeta}_t$ 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] simplifies the LSTM and GRU recurrence functions to be linear, such that it can exploit fast parallel sequence processing for training,

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

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

We will now look at how we can ensure that gradients do not vanish 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]

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

We can then perform a change of basis,

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

[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.

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

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, such 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 $\nu_i = \log(-\log r_i)$.

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].

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 $\boldsymbol{z}_0 = \boldsymbol{\zeta}$,

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

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 the attention mechanism into this framework, where attention is used on top of the RNN encoder,

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

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 $\boldsymbol{\Xi} \in \mathbb{R}^{T \times d_v}$ 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 $\boldsymbol{\Xi}$.

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,

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

Softmax is performed row-wise. The division by $\sqrt{d_k}$ is necessary because $\operatorname{Var}[\boldsymbol{x} \cdot \boldsymbol{y}] = d$, 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 contextualized outputs are convex combinations of values,

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

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.

TODO: Multi-headed self-attention.

6.2 Cross-attention

TODO

6.3 Positional encoding

TODO

6.4 Layer normalization

TODO

6.5 Residual layers

TODO

6.6 Architecture

TODO: Encoder/decoder.

TODO: Figure.

6.7 BERT

TODO

6.8 Vision transformers

TODO: Image patches as tokens...

References

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.

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.

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.

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.

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.

Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. *Advances in neural information processing systems*, 28, 2015.

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.

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.

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.

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.

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.

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.

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.

I Sutskever. Sequence to sequence learning with neural networks. *arXiv preprint arXiv:1409.3215*, 2014.

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.

Yi Zhu and Shawn Newsam. Densenet for dense flow. In *2017 IEEE international conference on image processing (ICIP)*, pages 790–794. IEEE, 2017.