

# PROBABILISTIC MODELS IN MODERN AI

## 1. FOUNDATIONS: WHAT AI MEANS

- What is the goal of AI? To construct dynamical systems that will process information, appropriately reacting to inputs, and to have some meta-program explain how this dynamical system will be shaped from an environment (either static, with data, or dynamic, or both). They will construct a representation of the data that allows for some downstream tasks that are not describable using usual programs
- Modern AI is about a search in the space of programs, and the identification of those that will do well.

### 1.1. Goals of Modern AI.

- A central long-term goal for AI is to create programs that perform tasks that require intelligence, following instructions we cannot explicitly write down.
- Modern AI seeks to build programs that autonomously construct representations of data and act on them under uncertainty.
- These programs must be able to play games, interact with streams of information and do clever things.
- In the end, these programs are just a family of functions. The thing is that there are quite many functions out there; how do we find the right ones is not completely trivial.
- To find the right functions, we need an environment: either static, or dynamic; the earlier is already the source of many exciting things, while the latter is obviously at the heart of current developments.
- It is not very important to define intelligence, but for us intelligence is the ability to treat streams of information and to find interesting things/patterns in them, with regards to some goals, which can be explicit or implicit, known in advance or unforeseen.
- Some central tasks that drive the construction of intelligent structures are prediction, compression, denoising; to perform those, a training algorithm will typically need to find patterns in data.

### 1.2. Branches of Modern AI.

- What are the branches of modern AI?
  - Symbolic/Knowledge-Based
    - \* Search, planning
    - \* Kernel methods
  - Machine Learning
    - \* Supervised: we have a training dataset of data points with labels, and we try to fit the labels
    - \* Unsupervised: we have a training dataset made of samples, and we try to model the dataset

- \* Self-Supervised: we have a training set where data points can be used as labels for others
- \* Reinforcement Learning: we have an environment that responds to actions
- \* Miscellaneous: semi-supervised learning, transfer learning, on-line learning, meta-learning
- Some other (more or less useful) dichotomies:
  - Deterministic vs probabilistic algorithms: is the output of the AI system deterministic or random? Most neural networks output deterministic outputs, but these can be fed into (or be fed to) some random variables, to generate random outputs.
  - Generative vs discriminative algorithms: we can either sample from some data, or try to infer parameters behind the data.
  - Static versus dynamic environment: a dataset is e.g. a static environment (we don't choose what we see), while a game is e.g. a dynamic environment
- Information theory is the foundation of what we do in modern AI and how we work with this course. Information theory is about what can be done theoretically (as opposed to algorithmically or practically) with information at our disposal. Information theory gives a baseline.
- Modern AI is in my view at the intersection of information theory and practical optimization. Information theory is in some sense the limit of what we can do, and it also suggests some general optimization tasks we can perform. From there, we obtain objects that can be transformed into e.g. intelligent agents.

### 1.3. Modern AI in a Nutshell.

- The most exciting general tasks devised by information theory are prediction, compression, denoising; and they turn out to be the most exciting ones to train AI models
- These three tasks allow one to transform to formulate optimization problem to train e.g. neural networks: we optimize on a space of functions in regard of such tasks. (generally speaking, optimization problems can be applied to labeled or unlabeled data or to environments). And this optimization process yields models with some capabilities. Then there is the question of how we leverage these capabilities to do things.

### 1.4. What this course will be about.

- We will start with the study of prediction and lossless compression as information-theoretic tasks.
- Then we will study neural networks, which provide a means to optimize.
- Later, we will discuss lossy compression and denoising. This will lead us to diffusion models.
- We will finish by discussing reinforcement learning, causality, and games.
- This will lead us to the 'universal AGI' ideas that are at the foundation of the current wave of the field.

## 2. PREDICTION

- We will start with prediction and lossless compression, two fundamentally related information-theoretic tasks. These lead (once we can optimize properly) to self-supervised learning, including LLMs, other things.

### 2.1. Machine Learning and Prediction.

- Intelligence has often been likened to the ability to predict, not only for machine learning, but also for animal and human intelligence. For instance, it is clearly one of the mechanism by which neurons learn, and ‘predicting the next symbol in a sequence’ is emblematic of IQ tests.
- In general, if we are to optimize something (as we will discuss with neural networks) we need a clear, universal objective, that can be attached to data. If the data has a time structure or a space structure, we can naturally formulate prediction tasks. The prediction task is extremely deep, as performing it well entails a good model of the world that generates the data. Next-token prediction, infilling, backward prediction are all part of that galaxy of tasks.
- Maybe we can start with an example: if we compute the first terms of a mathematical sequence... if we give 6.2831853 and we ask what is the next digit, how do we answer this? It would make sense to say that the answer is 0... why? This is not necessarily the simplest task, but an explanation here would be that  $2\pi$  is a good summary of the data, and it is a low-complexity one; again, as we will see, it corresponds to compressing the data.
- For textual data, supervised learning can be viewed as particular case of prediction, if we make the labels follow the data, e.g. if we write “ $x_1 :: y_1$   
 $x_2 :: y_2$   $x_3 :: y_3$  ...”, and then we provide a  $x ::$  and ask the model to predict after, it should predict  $y$ . This is in fact how we would use a foundation LLM for many prediction tasks.
- There are many closely related tasks, in particular denoising, which is also at the heart of information theory: if we were to noise some digits of  $2\pi$ , an intelligent algorithm should still be able to denoise them, based on compression ideas, for instance. But for now, it is good to start with prediction, as it is conceptually simpler and it yields some of the very best results in practice.

### 2.2. Scoring Rules.

- Note: the introduction of scoring rules is a bit unusual as a treatment of AI. We follow this road as a means to get into information theory without knowing information theory a priori. This is based on questions and results that were already available at least four hundreds years ago (Pascal and Huygens talked about it, the Bernoulli worked on it), but were maybe not pushed later. The theory of probability was in large part developed in the context of gambling, and prediction and gambling are very obviously related.
- What is a prediction for a random variable? Assume for simplicity and concreteness that it can only take a finite number of values. Predicting consists in delivering probabilities for the various outcomes of that variable, as available to an agent based on their information. The goal of a scoring

rule is to reward an agent outputting predictions based on their predictions and on the outcome.

- Of course, the outcome of a random variable is a single value, and this is the only feedback about how good a prediction was going to be. We can still try to make the rewards so as to *elicit* the right reward, i.e. to make it so that the agent maximizes their reward when they give the correct probabilities.
- So, let us formalize this: an agent outputs probabilities  $\vec{\pi} = (\pi_1, \dots, \pi_n)$  for the possible values  $\{1, \dots, n\}$  that a random variable could take. The space of possible outputs is the simplex  $\Delta_n \subset \mathbb{R}^n$  defined as the set of  $n$ -dimensional vectors with nonnegative entries summing up to 1.
- Then a scoring rule  $s$  takes as input  $\vec{\pi}$  and the actual (random) outcome of the observed variable  $i$  to yield a reward  $s(\vec{\pi}, i)$ .
- A scoring rule is called *proper* if the only maximizer of the expectation

$$\mathbb{E}_{\vec{p}}[s(\vec{\pi}, \cdot)] = \sum_{i=1}^n p_i s(\pi_i, i),$$

assuming ‘true’ probabilities  $p_1, \dots, p_n$  for the outcomes, is when  $\vec{\pi} = \vec{p}$ . The ‘subjective’ view of this question is that as much as agents don’t know the true probabilities, they are incentivized to disclose their own ‘beliefs’: their goal is to maximize their (perceived) expectation.

- Note that this point of view is very much related to a deep belief at the core of modern AI: subjectivism/bayesianism, which itself is at the heart of information theory. The idea is not really that there is mathematical model of the world that we can define and study mathematically, just that there are models of reality, that there is a feedback, and that we should just update the beliefs based on feedback. Of course, this is more a ‘general vision’ than anything else: it is not forbidden to assume that there is a more or less correct vision of the world; it is just not to be assumed that it can be defined accurately or accessed in any way.
- Ok... so how do we design proper scoring rules? A naive idea that is bad (but probably used at places) is to reward  $s(\vec{\pi}, i) = \pi_i$ : this seems reasonable, because if the agent gives higher probabilities to more likely events, their expected score increases... the problem is that they increase beyond the true probabilities: it is easy to see that the optimal strategy is in fact to output  $\pi_i = \mathbf{1}_{i=j}$  if  $p_j > p_k$  for all  $k \neq j$  (exercise: what is the solution if there is no such  $p_j$ ?).
- What are some examples of proper scoring rules?
  - The quadratic scoring rule, for instance  $-(1 - \pi_i)^2 - \sum_{k \neq i} \pi_k^2 = -\|\vec{\pi} - \delta_i\|^2$ , where  $\delta_i$  is the  $i$ -th canonical basis vector (or ‘one-hot’ vector) with zero entries at  $k \neq j$  and entry 1 at  $k = j$ .
  - The quartic scoring rule:  $4\pi_i^3 - 3\sum_k \pi_k^4$
  - The logarithmic scoring rule (the most beautiful of all):  $\log \pi_i$ .

### 2.3. Classification of Proper Scoring Rules.

- In the exercises, we saw the beautiful characterization of the proper scoring rules. Let us do this is slightly differently.

- Let us first state or remind a simple fact about convex functions: a function is convex if and only if it is the sup of all the affine functions dominated by it.
- An important (though obvious) trick: all expectations are linear as a function of the probability measure upon which they are based.
- Given a smooth strictly convex function  $G : \Delta_n \rightarrow \mathbb{R}$ , we can define a proper scoring rule by

$$s(\vec{\pi}, i) = G(\vec{\pi}) + \langle \delta_i - \vec{\pi}, \nabla G(\vec{\pi}) \rangle$$

- First note that when taking expectation, for fixed  $\vec{p}$ , the only ‘random quantity’ in this expectation that we need to average is  $\sum_i p_i \langle \delta_i, \nabla G(\vec{\pi}) \rangle = \langle \vec{p}, \nabla G(\vec{\pi}) \rangle$ .
- Define the Savage representation  $S(\vec{p}, \vec{\pi})$  as the expected return  $\mathbb{E}_{\vec{p}}[s(\vec{\pi}, \cdot)] = G(\vec{\pi}) + \langle \vec{p} - \vec{\pi}, \nabla G(\vec{\pi}) \rangle$ .
- Why is our scoring rule proper, now? A small cool twist (related to convex duality): for a fixed  $\vec{\pi}$ , as a function of  $\vec{p}$ , we have an affine function that is tangent to the graph of  $G$ , and so, since convex functions are always the sup of the affine functions that are tangent to their graphs,  $S(\vec{p}, \vec{\pi}) \leq G(\vec{p})$  for all  $\vec{p} \in \Delta_n$ , and in fact we have strict inequality if (and only if)  $\vec{p} \neq \vec{\pi}$ .
- So, now if we fix  $\vec{p}$  and optimize  $\vec{\pi}$  instead, we find that the best expected return is by taking  $\vec{\pi} = \vec{p}$ .
- Conversely, if we have an expected return function that derives from a proper scoring rule, we have, at every fixed  $\vec{p}$ , some Savage function  $S(\vec{p}, \vec{\pi})$ ; we can define the ‘entropy’  $H(\vec{p})$  as  $\sup_{\vec{\pi}} S(\vec{p}, \vec{\pi})$ .
- Note that  $H(\vec{p}) = \langle \vec{p}, s(\vec{p}, \cdot) \rangle$ .
- For every fixed  $\vec{\pi}$ , we have  $S(\vec{p}, \vec{\pi}) \leq H(\vec{p})$ , and hence (for fixed  $\vec{\pi}$  again), the affine function  $S(\cdot, \vec{\pi})$  lies below  $H(\cdot)$ , and is tangent at  $\vec{p} = \vec{\pi}$ . Hence,  $H$  is convex function of  $\vec{p}$  (it is the sup of functions depending on  $\vec{\pi}$  lying below it).
- Hence (assuming differentiability and strict properness), we get that the gradient at  $\vec{p} = \vec{\pi}$  of the affine function  $S(\cdot, \vec{\pi})$  matches that of  $H(\cdot)$ .
- Now, if we evaluate how much we get rewarded in case of outcome  $i$ , we collapse the probability to  $\vec{p} = \delta_i$ , and we get  $S(\delta_i, \vec{\pi}) = H(\vec{\pi}) + \langle \nabla H(\vec{\pi}), \delta_i - \vec{\pi} \rangle$ .
- So, we have found that the scoring rule should be  $s(\vec{\pi}, i) = H(\vec{\pi}) + \langle \delta_i - \vec{\pi}, \nabla H(\vec{\pi}) \rangle$
- So (modulo differentiability assumptions), we have found a complete characterization of scoring rules!
- Now, if we ask for locality, i.e. that  $s(\vec{\pi}, i)$  only depends on  $\pi_i$  (and not the other  $\pi_j$ ’s for  $j \neq i$ ), we find that the only solution is  $s(\vec{\pi}, i) = \alpha \log \pi_i + \beta$  for some  $\alpha > 0, \beta \in \mathbb{R}$ .
- As we will see, this leads us to information theory, a couple hundred years earlier than introduced by Shannon!

## 3. COMPRESSION

Note: much of this is taken from the the wonderful classic ‘Elements of Information Theory’ by Cover and Thomas.

## 3.1. Overview.

- In the modern era, if there is a single task that is now understood to be associated with intelligence/understanding, it is compression. This was mostly initiated by Claude Shannon, although the problem was somehow considered before, with e.g. the Morse code representing some attempt to make communication more efficient.
- The ability to synthesize observations about the world down to a few principles is what physics is (abstractly) about. Note that for physics and the natural sciences, they largely deal with a continuous world, and hence arguably about lossy compression. There are some theoretical ideas as to why the shortest explanation is the best: Occam’s razor is the name under which this falls.
- Here is a puzzle to think a bit: assume we throw a coin a 100 times and that we encode the result and encode in via the binary sequence 0101010101...0101. Intuitively, this suggests there is something wrong with the coin. We are tempted to say: this sequence is super unlikely, i.e. has a chance of  $2^{-100}$  to be observed. But can’t this be said about *any sequence of observations*?
- Obviously, this sequence is easy to compress, and that’s something that should tell us something!

## 3.2. Entropy.

- The most important information-theoretic quantity associated with a discrete random  $X$  variable taking values in a set  $\mathcal{S}$  is its entropy  $H(X)$  defined as  $-\sum_{i \in \mathcal{S}} p_i \log p_i$ , with the usual convention that  $0 \log 0 = 0$  (by continuous extension of  $x \mapsto x \log x$ ); when the logarithm is the natural one (basis  $e$ ), the entropy is measured in so-called nats, and when we divide by  $\log(2)$ , we get a  $H_2(X) = -\sum_{i \in \mathcal{S}} p_i \log_2 p_i$ , measured in bits.
- Going back to the prediction question, the entropy is the loss (negative reward) incurred by an optimal predictor for the cross-entropy loss.
- Now, what is interesting about the entropy  $H_2(X)$  is that this corresponds to the optimal compression rate if we are to store outcomes of i.i.d. (independent identically distributed) samples  $x_1, \dots, x_n$  sampled from  $X$ : in other words we should be able to compress (in a lossless fashion) those with  $\approx nH_2(X)$  bits as  $n \rightarrow \infty$ .
- The next most useful form is Gibbs inequality: the Kullback-Leibler divergence between two probability measures is always nonnegative, i.e.  $D_{KL}(p||q) = \sum p_i \log \frac{p_i}{q_i} \geq 0$
- The most direct consequence that it is maximized when all probabilities are equal: take  $q_i$  to be uniform, we get

$$D_{KL}(p||q) = \sum p_i \log p_i + \sum p_i \log n \geq 0$$

and so we get  $\sum p_i \log p_i \geq -\log n$ , and hence  $-\sum p_i \log p_i \leq \log n$ .

### 3.3. Encoding.

- Let us remind why this is the case: we need to show that we *can* encode  $x_1, \dots, x_n$  with roughly  $nH_2(X)$  bits and that we can't encode with less with high probability.
- Let's start with this direction: why do we need *at least*  $nH_2(X)$  bits to encode  $x_1, \dots, x_n$ ?
- The simple idea is that if not outcomes of  $X$  are equally likely, we should use shorter messages to encode outcomes that are more likely.
- As a first tool, we need to understand what 'encoding' means: it means that we have an encoding and decoding function  $\mathcal{S}$  to a vocabulary of codewords  $\Gamma$  sequences of 0 and 1s that is unambiguous, i.e. such that we know 'when to stop' when reading.
- Of course, the simplest would be to have a 'stop' character, but that means we have to reserve a character which could perhaps be better used.
- The notion we need is that of prefix code (also called instantaneous code): no codeword  $C \in \Gamma$  must be the prefix of another codeword  $C' \in \Gamma$ : in other words, when we see  $C$ , we know that 'this is it, another codeword starts', so that if we concatenate  $C_1C_2 \dots C_n$ , we get a uniquely decodable sequence.
- A useful, simple observation is Kraft's inequality: it asserts that for any prefix code  $\Gamma$ , we have  $\sum_{C \in \Gamma} 2^{-\ell(C)} \leq 1$ , where  $\ell(C)$  denotes the length in bits of the codeword  $C$  (you will see this in the homework, but it should not come as a surprise).

### 3.4. Lossless Compression Upper Bound via Arithmetic Coding.

- Below, we will show that we need *at least*  $nH_2(X)$  bits (in expectation) to encode  $n$  i.i.d. samples of a random variable  $X$ .
- How do we see that we don't need more than  $nH_2(X)$  to encode  $n$  samples? The simplest answer comes in the form of arithmetic coding.
- The idea is very simple: just split the interval  $[0, 1]$  into 'buckets' of sizes  $p_i$  for all  $i \in \mathcal{S}$ .
- Now, to encode  $i$ , just write down a real number in binary in the form  $0.b_1b_2b_3 \dots$  until we land in one interval corresponding to a bucket.
- The number of bits required to write down that real number (assuming we know in advance its length) is obviously  $\leq \lceil -\log_2 p_i \rceil$ : adding one bit splits in two the range that the number could take (if we have written  $b_1b_2 \dots b_n$ , the possible range is  $[0.b_1 \dots b_n, 0.b_1 \dots b_n 1)$ ), so we can point that in  $\log_2 p_i$ .
- So, in expectation we need  $H_2(X) + 1 + C$  as an upper bound, where  $C$  is what is needed to represent the number of bits.
- Now, if we have  $n$  i.i.d. samples of  $X$  (for  $n$  large), it is fairly easy to see that with very high probability we can amortize the  $1 + C$ .
- Indeed, we have that the log likelihood of  $X$  will accumulate around  $-nH_2(X)$ : i.e. with high probability we will be in a 'typical set' where each element has roughly probability  $2^{-nH_2(X)}$ : this is a simple consequence of the law of large numbers.

- So, with overwhelmingly large probability, we have a sample of probability  $\approx 2^{-nH_2(X)}$ , requiring  $nH_2(X) + 1 + C \approx nH_2(X)$  (technically  $C$  can grow with  $n$  but not so fast).
- More generally speaking, i.i.d. is not very important: the important thing is that the probabilities are all small enough that the  $1 + C$  can be ignored.
- In the homeworks, you will see more detail about this, and also about the other codings.

### 3.5. Lossless Compression Lower Bound.

- So, the key result for lossless encoding is: why can't we use less than  $nH_2(X)$  bits to encode  $n$  i.i.d. samples of  $X$  as  $n \rightarrow \infty$ ?
- Let us start with the obvious statement that if we have uniform probability for a random variable  $X$  taking  $2^n$  values, we can't do better than encoding with  $n$  bits.
- It is obvious, but how do we prove that? The idea is that if we use a shorter codeword for some value, we must get rid of some code words, and increase somewhere else.
- Things 'feel convex', i.e. balancing lengths consistently improves things... as a result, all lengths must be equal, and hence be equal to  $n$ ... How do we formalize this?
- This is ultimately Gibbs' inequality: comparing the expected length of encoding to the entropy, we have

$$\begin{aligned} \sum p_i \ell_i - p_i \log(1/p_i) &= - \sum p_i (\log(2^{-\ell_i}) - \log(p_i)) \\ &= \sum p_i \log\left(\frac{p_i}{r_i}\right) - \log(Z) \\ &\geq D_{KL}(p||r) \end{aligned}$$

where  $r_i = 2^{-\ell_i}/Z$ , with  $Z = \sum_j 2^{-\ell_j}$ , where we used Kraft inequality to say  $\log(Z) < 0$ .

- The key insight is that 'most of the mass' over  $n$  samples of  $X$  will be concentrated in a 'typical region' of samples that all have roughly the same probability
- You will see in the homework that this is the region where  $-\sum_{k=1}^n p_{x_k} \log(p_{x_k})$  is close to  $nH(X)$ .

### 3.6. Towards Algorithmic Information Theory.

- From the above, we learn that compressing and predicting are very closely related: if we know the probabilities of sequences, we can compress.
- The question of compression can be asked outside of a probabilistic framing a priori (sometimes it is hard to know probabilities a priori): how to compress some piece of data that we have in front of us?
- For instance, if we see 10101010101010, we may want to say '10 repeated 8 times': at least for most humans that would be easier to remember.
- For instance, if we see the first 100 bits of the expansion of the number  $\pi$ , we would like to say 'First 100 bits of  $\pi$ ', instead of giving the 100 bits.
- If we find that a sequence of bits can be computed by a relatively short program, we could also give the source code of that program instead of the sequence of bits.

- Obviously, if we had a choice of several programs yielding the same number, we should give the shortest program.
- Of course, all of this hinges a priori on a choice of computer and of programming language, but if the data we want to carry is sufficiently large, these choices may only contribute ‘by a constant’: we can write the emulator of one computer in terms of another computer.
- Kolmogorov, Solomonoff, and Chaitin all reached this conclusion independently (for all we know) in the late 1960s: we should follow the ‘minimum description length principle’: find the shortest program that describes the data we want to compress (for a fixed Turing-complete computation model).
- Definition: the Kolmogorov Complexity of a string  $s$  of length  $\ell$  with respect to a universal computer  $\Upsilon$  is equal to the minimal length of the program  $p$  computing  $s$  (that halts afterwards):  $\mathcal{K}_\ell(s) = \min_{p:\Upsilon(p)} \text{length}(p)$ , where length is measured in bits.
- Note: in the worst case,  $\mathcal{K}_\ell(s)$  will typically be equal to  $\ell + c$  for some small constant  $c$ : you can always write a program that outputs  $s$  verbatim.
- It has a small problem: it is uncomputable, as you will see in the homeworks (but this is not as probably big a problem as people make it to be).

### 3.7. Probabilistic Meaning of Compressibility.

- Theorem: if we consider a sequence  $x_1, x_2, \dots$  of i.i.d. samples of a random variable  $X$ , then as  $n \rightarrow \infty$ ,  $\mathbb{E}[\mathcal{K}_n(x_1 \cdots x_n)] \rightarrow nH_2(X)$ .
- We start with a lower bound: encoding with programs is a special case encoding, and we know that we need at least  $\approx nH_2(X)$  bits to encode, regardless of the method.
- For the upper bound, let us first think of a simple example, with i.i.d. samples from a Bernoulli variable  $B(p)$  with  $p$  very small (i.e. there are very few 1s): in this case, it is probably worth just noting where the 1s lie (everything else should be 0).
- If  $p = 2^{-32}$ , say, and we have  $2^{32}$  samples, we should just encode how many 1s there are, and if there is one, we need 32 bits to say where, if there is two we need  $\log_2 \binom{2^{32}}{2}$ , if there are three, we need  $\log_2 \binom{2^{32}}{3}$ , etc.
- So, if we choose to describe things this way (regardless of whether there is a lot of 1s or not), in case there are  $k$  1s, we need to describe the scheme, to describe  $k$  and to give the index of the  $k$  bits, i.e.  $c + \log n + \log \binom{n}{k}$ .
- Now,  $\log \binom{n}{k} \leq -n(k/n) \log(k/n)$ , and so if we average over  $k$ , and we have  $n$  large, the number  $k$  concentrates around  $pn$ , so we end up with  $\leq c + \log n + nH_2(B(p))$ .
- And hence, we find that this specific algorithm achieves the required compression rate. Of course, this general argument does not require us to work with binary sequences.
- This can even be refined into something sharper:  $\mathcal{K}_n(x_1 \cdots x_n) \leq nH_2(B(\frac{1}{n} \sum_{i=1}^n x_i)) + \frac{1}{2} \log n + c$  (we can shave  $\frac{1}{2} \log n$  by being finer about  $\log \binom{n}{k}$ ).
- Anyway... the result is that the Kolmogorov complexity achieves exactly the compressibility in the i.i.d. case.
- The important insight here is a bit beyond the theorem: Kolmogorov’s complexity can be defined *outside of a probabilistic framework*... we just showed show it coincides with entropy in the i.i.d. context.

### 3.8. Algorithmic Randomness.

- Here is a simple idea: a sequence of bits  $x_1 \cdots x_n$  is *algorithmically random* if  $\mathcal{K}_n(x_1 \cdots x_n) \geq n$ ; we call an infinite string  $(x_j)_{j \geq 1}$  *incompressible* if  $\lim_{n \rightarrow \infty} \mathcal{K}_n(x_1 \cdots x_n) / n = 1$ .
- Why is it called random? It satisfies the law of large numbers (the proportion of 0's and 1's will both tend to 1/2), as well as *any computable statistical test*.
- Why? We leave it as an exercise, but the idea for the law of large numbers is that if the proportion of 0's was not going to 1, we could exploit this to compress.
- This is an interesting, complementary view on probability, which does not involve sigma algebras or measure theory: this is sometimes called the Martin-Löf theory of randomness (also emerged in 1966, like all the great things described above).
- It feels much more concrete than probability theory, though its uncomputability related to the halting problems don't make it very practical a priori; still, as we will see, if we don't think of this too much, we can end up with something that works reasonably well in practice.

### 3.9. Universal Prediction Prior.

- From the of ideas associated with compression and algorithmic randomness, we can try to answer the question: if we see a string of words, and try to predict the next bit, how to predict the next word?
- Intuitively, if the string has a very simple description relative to its a priori length (e.g. the word 'cow repeated 128 times'), then we would want to use a minimal description length approach to say that it is likely the next word is 'cow'.
- Of course, it would be foolish to ascribe 100% of chance that the next word is cow, but it is more likely than any word.
- The natural thing to do is to try to see what are the possible programs constructing the sentence, to ask ourselves how likely these are, and to give a weight to each 'hypothesis'.
- Now, the weight we would want to give to a hypothesis should depend on the length of the program, and the natural weight would be  $2^{-\ell(p)}$ , where  $\ell(p)$  is the length of the program  $p$ .
- Now, the total mass of this is obviously finite, since not all programs terminate the probabilities are  $2^{-\ell(p)} / \Omega$ , where  $\Omega = \sum_{p:p \text{ terminates}} 2^{-\ell(p)}$  is the so-called Chaitin (or sometimes Kolmogorov-Chaitin) constant.
- But now, since the determining which program terminates or doesn't is itself not algorithmically computable,  $\Omega$  is not computable (there is no Turing machine that outputs its expansion in any basis).
- In fact if we could compute the first bits of  $\Omega$  as a number, this would give us a very substantial insight into how many programs of a given length terminate.
- An interesting property about  $\Omega$  is that it is algorithmically random in our definition.
- If we are not worried about computability, then  $2^{-\ell(p)} / \Omega$  is a reasonable prediction prior: if we want to predict the next word in a sequence, to

compute the likelihood of any possible next word, just integrate against each terminating program outputting the first part of the sequence, and sum ‘what they say’.

- Generally speaking, one prove that the probability ascribed by the universal prior to a sequence  $x$  is comparable to  $2^{-\mathcal{K}(x)}$  (the ratio of both is bounded away from 0 and  $\infty$ ).
- This estimation mechanism has been proven to be asymptotically optimal in the case the sequence is itself generated by a Turing machine.

### 3.10. Kolmogorov’s Sufficient Statistic.

- Given a data sequence  $x$ , we can think of how close we get to compressing it with  $\ell$  bits (rather than what is the shortest description).
- This is a subtle (but philosophically important) difference with the previous setting: we are ready to give up on some description, in exchange for concision.
- The Kolmogorov’s structure function  $\mathcal{K}_n^\ell$  is defined as the log size of the smallest set containing the beginning

$$\mathcal{K}_n^\ell(x) := \min_{p: \ell(p) \leq \ell, x_{1:n} \in S \subset \{0,1\}^n, \mathcal{U}(p,n)=S} \log_2 |S|$$

- We have then that for any  $n \geq 1$  and some constant  $c > 0$ , there will be a first  $\ell_*$  such that  $\mathcal{K}_n^{\ell_*}(x) + \ell_* \leq \mathcal{K}_n(x) + c$ .
- The idea is that as we allow  $\ell$  to grow by 1 bit, we should *at least* reduce  $\log_2 |S|$  by 1, but typically more; we stop increasing  $\ell$  as soon as  $\log_2 |S|$  does not decrease *faster* than  $\ell$  grows.
- We define the Kolmogorov minimal sufficient statistic for  $x$  as the program  $p_{**}$  and/or the set  $S_{**}$  achieving the minimum in  $\ell_*$ .
- The idea is that if we find this program for the corresponding  $\ell_*$ , we have that  $p_{**}$  is literally all we can say about the data using  $\ell_*$  bits.
- From the point of view of that description, all else is noise.

### 3.11. Minimum Description Length Principle.

- More generally, if we have data from a distribution and we have i.i.d. samples, the Kolmogorov’s statistics provides a unified vision to describe data: the minimum description length (MDL principle).
- For a distribution  $X$  and i.i.d. samples  $x_1, \dots, x_n$ , we should find the probability distribution  $\pi$  that minimizes

$$L_p(x_1, \dots, x_n) = \mathcal{K}(\pi) + \log \frac{1}{\pi(x_1, \dots, x_n)},$$

where  $\mathcal{K}(\pi)$  is the Kolmogorov complexity of the distribution (i.e. how many bits are required to describe it).

- So the idea here is that there is a trade-off between fitting and complexity of fitting: this is a simple universal idea.
- Note: this is quite reminiscent of what is happening with LLMs and the various model sizes.
- Also: an idea (upon which we will elaborate), that seems somehow important to understand LLMs is to think of circuits rather than Turing machines (this is at the same time more tractable, and closer to e.g. transformer models).

### 3.12. Where this will go (later in the course).

- Anyway, the compression ideas are very philosophical (rather than practical), but they led to the empirical revolution we see now.
- Upon these foundations, some universal AGI models (i.e. agents acting in an environment, not only making predictions) were made: AiXi and the Gödel machines in particular come to mind.
- All that was needed in some sense for this to bloom into what we see today was the advent of neural networks: they allowed to work with viable models of computations that could be optimized, and would naturally be robust to a lot of details.
- So, what we will do next is to focus on neural networks, understand something about them, and then return to the key applications: LLMs (direct incarnation of the ideas of Turing, Shannon, Kolmogorov, Chaitin, Solomonoff), diffusion models (a different take also based on information theory, with an algorithmic component), and reinforcement learning.

## 4. NEURAL NETWORKS

Neural networks have made many things once deemed impossible work quite well in the last decade. If we stick with the idea of optimization applied to compression, the only thing that is missing is a proper family of programs that predict the next words based on the previous ones. The general vision of a universal predictor has become approximately possible with e.g. the advent of transformer-based models applied to text data.

So we will first discuss neural networks: the major insight of the years 2007-2017 was that neural networks, when given enough parameters and data can indeed be *trained* (i.e. optimized) to yield good functions that can serve as building blocks to powerful predictors. This has first shined in the context of supervised learning, but the idea works for semi-supervised and unsupervised problems as well.

### 4.1. Setting.

- In order to explain the ideas and challenges associated with neural networks, it is useful to take a concrete framework.
- As said before, the simplest framework (though by no means the only one) to discuss the success of neural networks is in the context of supervised learning; historically, it is also the framework where their power first became apparent.
- Imagine hence that we have a dataset  $x_1, \dots, x_N \in \mathbb{R}^{d_{in}}$  and that the data points have 'labels'  $y_1, \dots, y_N \in \mathbb{R}^{d_{out}}$ ; we would like to learn a function  $f : \mathbb{R}^{d_{in}} \rightarrow \mathbb{R}^{d_{out}}$  such that  $f(x_i) \approx y_i$  and hope that it generalizes well to unseen data points.
- Typically an underlying hypothesis is that  $x_1, \dots, x_N$  are i.i.d. from an unknown distribution, and that we want for instance minimize the expected error (measured in a certain way, with e.g. the least-squares or MSE or  $L^2$  norm) for a point  $x$  sampled according to that distribution.
- The idea is as a result that we should try to pick the 'best'  $f$  that we can find; while this is a fitting problem, we should be mindful of the classical overfitting problem: if e.g.  $d_{in} = d_{out} = 1$  and we took a polynomial of

degree  $N$  to fit  $x_1, \dots, x_N$ , it would probably be a very bad idea when  $N$  is large (why?).

#### 4.2. Architecture.

- The simplest and most natural space of functions over which to optimize is the space of linear or of affine maps; many things will be stable, and we have a good control over the behavior/uniqueness of the maps we find, and a good understanding of their sensitivity to noise in the data, etc. (Note: it is a good exercise to go over this case, where we have  $(d_{in} + 1) d_{out}$  parameters).
- It was realized fairly early on that there are simple functions (like the XOR function) that cannot be realized by linear networks, leading to a first neural network winter.
- The problem is obviously that in reality the true map  $f$  (if it exists) is often nonlinear and it is not really clear a priori how to extend from the linear case; in many ways the quadratic case is not (or at least not obviously) the right thing.
- Neural networks were initially designed after some simplified description of the working of biological neurons.
- So, neural networks just compose affine maps with pointwise nonlinearities; basically fix a nonlinear map  $\sigma : \mathbb{R} \rightarrow \mathbb{R}$  and compose ‘layers’.
- This is the simple example of the multilayer perceptron: we have  $\ell$  layers,  $\alpha_0(x) = x$ , and  $\tilde{\alpha}_1(x) = W^{(0)}x + b^{(1)}$ , with  $W^{(0)}$  being a  $d_1 \times d_0$  (with  $d_0 = d_{in}$ ) matrix and  $b^{(1)} \in \mathbb{R}^{d_1}$ ,  $\alpha_1(x) = \sigma(\tilde{\alpha}_1(x))$  (where  $\sigma$  is applied pointwise),  $\tilde{\alpha}_2(x) = W^{(1)}\alpha_1(x) + b^{(2)}$ ,  $\alpha_2(x) = \sigma(\tilde{\alpha}_2(x))$ , etc. and we output  $\tilde{\alpha}_\ell(x) = W^{(\ell-1)}\alpha_{\ell-1}(x) + b^{(\ell)}$ , where  $W^{(\ell-1)}$  is a  $d_\ell \times d_{\ell-1}$  (with  $d_\ell = d_{out}$ ) matrix. and  $b^{(\ell)} \in \mathbb{R}^{d_\ell}$ .
- So, we have  $\ell - 1$  hyperparameters  $d_1, \dots, d_{\ell-1}$  and  $P = \sum_{i=0}^{\ell-1} (d_i + 1) d_{i+1}$  parameters, which we typically write  $\theta \in \mathbb{R}^P$ .
- We denote by  $f_\theta : \mathbb{R}^{d_{in}} \rightarrow \mathbb{R}^{d_{out}}$  the resulting function  $\alpha_\ell$  function.
- Now the the question is: what do we do with this?
- There were fairly early on some universality results saying that if we took the number of parameters large enough, we could represent any function  $\mathbb{R}^{d_{in}} \rightarrow \mathbb{R}^{d_{out}}$  in an efficient way, with as little as one hidden layer.
- This led the community to (incorrectly) believe that one layer was the way to go, and to (incorrectly) believe that the correct thing to do was to minimize the number of parameters while being able to fit the data.

#### 4.3. Optimization.

- The modern wisdom is that specifying an architecture is only half of the story; specifying a training process is as much part of defining an AI model as specifying the number of layers and their sizes.
- So the first thing that should be said is that training means optimizing, i.e. trying to minimize a score function, for instance the mean-square error:  $L(\theta) = \frac{1}{N} \sum_{n=1}^N \|f_\theta(x_i) - y_i\|^2$ .
- It can be that the true objective may be different, but we need to select a decent objective to train our model.
- Now, we typically minimize this by using a first-order method, like gradient descent, i.e. we consider  $\dot{\theta}(t) = -\nabla L(\theta)$  and we let the system converge.

- Note: there was initially no theoretical justification for why this is a good idea, it just so happened that with a large number of parameters, this is the only thing that can be done.
- To initialize the parameters of the network, it is generally not a good idea to start from zero... it is much better (in practice) to initialize the parameters randomly; this much was understood for a while ago. Typically, the thing that is done is to initialize the parameters with iid centered Gaussians.
- Then, even if one buys that thesis (which was not the case initially), there are some extra hyperparameters:
  - What is the variance of the Gaussians?
  - How long do we run the optimization?
  - Do we penalize the weights of the parameters?
  - Do we use momentum for the optimization?

#### 4.4. A Brief History of Successes and Misconceptions.

- The ideas of neural networks came to be in the 1940s, but the first serious works with the idea of applying neural networks to concrete AI tasks came in 1958, with Rosenblatt's linear perceptron (linear 1-layer neural network).
- Linear perceptrons (trained by an ad-hoc algorithm) gave promising results in the beginning, and initially people thought one could do anything with them.
- Around the late 1960s, people became aware of their limitations via theoretical results: they could not represent naturally certain functions, such as parity and xor. This led to a first 'neural network winter' from the late 1960s to the mid-1980s (which strictly contained a more general AI winter from the mid-1970s to the early 1980s).
- The introduction in the mid-1980s of the back-propagation algorithm showed gradient descent could be used for fairly large networks, and these results showed that the functions not learnable by the linear perceptron could in fact be learned, which gave hope that neural networks could indeed be trained.
- A bit later, a line of results (in the late 1980s) showed that large nonlinear neural networks with  $\ell \geq 2$  layers could theoretically approximate any function, which gave further hope.
- Despite some successes (AI playing backgammon in 1993, and ConvNets), the community remained largely skeptical of neural networks because of the challenges associated with training a (possibly very) non-convex function in high-dimensional, and the problems of exploding and vanishing gradients in the case of deep or recurrent neural networks.
- The absence of a theory guaranteeing convergence of gradient descent was largely viewed as being a good reason to steer away from such models from the mid-1990s to around sometime between 2006 and 2012 (i.e. a second neural network winter).
- Other methods, such as kernel methods worked from alternate, mathematically sophisticated postulate, like kernel methods.
- Interestingly, there were a large number of misconceptions even among the believers of the theory, in particular about e.g. the fact that the size of neural networks should stay small to prevent over-fitting, because of two concurrent reasons:
  - Some wrongly applied theoretical tools (e.g. VC dimension).

- Some misleading early empirical results.
- It was only because of hardcore believers and empirical successes that neural networks came back to life.
- Then people started making (again wrong) hypotheses of why deep learning was working in practice:
  - There was the (somehow strange, but interesting) theory that neural networks would get stuck in local minima, but that there was an ocean of local minima that were all equivalent.
  - This hypothesis became very popular, and stimulated a lot of research, in particular research that would correct it.
- For a while, theory was just saying the opposite to practice, until two (linked) key insights corrected the early misconceptions:
  - The theory of infinitely-wide neural networks (in particular the NTK (invented at EPFL) and the mean-field approach (invented at Stanford but also developed by EPFL people)): the idea is that fixed-depth neural networks that are large enough, behave, *when trained with gradient descent*, quite well. This in particular disproved the local minima theory, as well as the ideas that non-convexity would get worse with an increasing number of parameters.
  - The double-descent curve in deep learning (also discovered at EPFL, though other people are usually credited): if we look at the generalization curve of a fixed-depth neural network (trained for some time with gradient descent) as a function of the number of parameters, then the generalization error first goes down (unsurprisingly), then goes up again (as some would expect), and then (perhaps most unexpectedly) goes down below the first local minimum... so very large neural networks (when trained with gradient descent) don't overfit after all.
- Now, though the theory doesn't predict everything about what we see, there are no fundamental disagreements anymore... and it so appears that certain neural networks (like transformers) can really implement the ideas of Kolmogorov, Solomonoff, and Chaitin in an approximate form, to the point that computers will soon be able to be AI researchers!

#### 4.5. Infinite-Width Limit.

- As a means to understanding what large neural networks do, we should first understand what is meant large neural networks in a mathematical setup.
- The simplest framework to understand these is to fix a dataset  $x_1, \dots, x_N \in \mathbb{R}^{d_{\text{in}}}$  with labels  $y_1, \dots, y_N \in \mathbb{R}^{d_{\text{out}}}$ , fix a number of layers  $\ell \geq 2$ , take a neural network with hidden layer widths  $n_1, \dots, n_{\ell-1}$  and let  $n_1, \dots, n_{\ell-1} \rightarrow \infty$  (simultaneously or sequentially): this is what we will mean by a large neural network.
- Such a limit was first considered in the case of one hidden layer by Neal in the 1990s:
  - he found that the variances of the weights  $W^{(1)}$  at initialization of the layer should be scaled like  $1/n_1$ , i.e. like  $1/\sqrt{n_1}Z$  where  $Z$  is fixed-variance Gaussian) in order to get a non-trivial limit (we get 0 if we scale smaller and infinity if we scale bigger);
  - the weights  $W^{(0)}$  don't need any rescaling: the size  $n_0 = d_{\text{in}}$  is fixed, and hence the values taken there are fixed.

- the bias weights  $b^{(1)}$  don't need any scaling (and they can be initialized to zero, in fact) either; let's say that their variance is  $\beta^2$ .
- To be a little more precise, the question of scaling was so that the first and second moments of the random function ( $f_\theta : \mathbb{R}^{d_{\text{in}}} \rightarrow \mathbb{R}^{d_{\text{out}}}$ ) are under control as  $n_1 \rightarrow \infty$ :
  - Conditionally on the values of  $(\alpha_1)_{x \in \mathbb{R}^{n_1}}$ , the output function  $f_\theta = \tilde{\alpha}_2$  is a centered Gaussian with covariance
 
$$\mathbb{E} [f_\theta(x) f_\theta(x') | \alpha_1(x), \alpha_1(x')] = \sigma_2^2 (\alpha_1(x))^T (\alpha_1(x')) + \beta^2.$$
  - Note: as a result, this is a 'random Gaussian', i.e. it has a random covariance, but that doesn't pose any problem.
  - Now, the right scaling so that  $(\alpha_1(x))^T (\alpha_1(x'))$  doesn't blow up as  $n_1 \rightarrow \infty$  is that we have a rescaling by  $1/\sqrt{n_1}$  (otherwise, we are just summing  $n_1$  i.i.d. terms (for fixed  $x$ ), and the variances just add up).
- This way, we find that at initialization, a neural network  $f_\theta$  converges as  $n_1 \rightarrow \infty$  to a random Gaussian function; the covariance function is a kernel.
- And this was often used as a reason to work with kernel methods instead: we can think that we have a random function prior and then for each  $x \in \mathbb{R}^{d_{\text{in}}}$ , we can compute a Bayesian posterior for what the function  $f_\theta$  is at an unknown data point, given  $f(x_i) = y_i$  (with some possible uncertainty).
- If you didn't know what kernel methods, this is exactly what kernel regression is: finding the expected value of  $f(x)$  knowing  $f(x_i) = y_i$  (if you add the possibility that there is an iid noise around each label, this becomes kernel ridge regression); this will be discussed further below.
- This was then extended by Cho and Saul: for multilayer neural networks, we find that if you scale the standard deviations like  $1/\sqrt{n_1}, 1/\sqrt{n_2}, \dots, 1/\sqrt{n_{L-1}}$ , then the resulting  $f_\theta$  (at random initialization) tends as  $n_1, \dots, n_{\ell-1} \rightarrow \infty$  to a random Gaussian function (with non-zero and non-infinite variance).
- To make this more simpler to study, a simple thing to say that all matrices  $W^{(0)}, \dots, W^{(L-1)}$  and biases  $b^{(0)}, \dots, b^{(L-1)}$  will be i.i.d. standard Gaussians (variance 1), and that we will just re-parametrize the neural network using the following parametrization for each layer:

$$\tilde{\alpha}_\ell(x) = \frac{1}{\sqrt{n_\ell}} W^{(\ell-1)} \alpha_{\ell-1}(x) + \beta b^{(\ell-1)}$$

- As we will discuss later, this is in fact less innocuous than it seems; it is now known as the NTK (Neural Tangent Kernel) parametrization.

#### 4.6. Neural Tangent Kernel.

- The idea of the Neural Tangent Kernel is to study what happens to the *training dynamics* of an infinitely-wide neural network, when the weight matrices are scaled according like  $1/\sqrt{n_1}, 1/\sqrt{n_2}, \dots$
- In other words: one can describe the trajectory in function space of the function  $f_{\theta(t)}$  as  $\dot{\theta}(t) = -\nabla C(\theta(t))$ , starting with  $\theta(0)$  as above, where  $C(\theta(t))$  is the cost function e.g. defined by  $\sum_{i=1}^N \|f_\theta(x_i) - y_i\|^2$ ; perhaps surprisingly, this dynamics becomes particularly simple when  $n_1, \dots, n_{\ell-1} \rightarrow \infty$ .
- Let us first focus on the scalar output case  $d_{\text{out}} = 1$ . The vector case is not very different (just with worse notation!).

- The idea is in fact quite simple: if we study what happens in function space, we have a (convex) cost function  $\mathcal{C}$  on the space of all functions  $\mathcal{F}$  from  $\mathbb{R}^{d_{\text{in}}} \rightarrow \mathbb{R}$  (a fairly strange thing in functional analysis, as we don't put any topology, but we are topology-agnostic here), composed with a nonlinear parametrization  $\mathbb{R}^P \rightarrow \mathcal{F}$  defined by  $F : \theta \mapsto f_\theta$ .
  - Let us say that we perform a small step  $\epsilon > 0$  of gradient descent: we update the parameters  $\theta(t)$  by

$$\begin{aligned} \theta(t + \epsilon) &= \theta(t) + \epsilon \nabla \mathcal{C}(\theta(t)) + o(\epsilon) \\ &= \theta + \epsilon D\mathcal{C}|_{f_{\theta(t)}} \circ DF(\theta(t)) + o(\epsilon), \\ &= \theta + \epsilon \sum_{n=1}^N (D_{f_\theta(x_n)} \mathcal{C}|_{f_{\theta(t)}}) \partial_\theta F(x_n) + o(\epsilon) \end{aligned}$$

where  $(D_{f_\theta(x_n)} \mathcal{C}|_{f_{\theta(t)}})$  represents the derivative of the cost functional with respect to a modification of the value of its (functional) argument  $f$  at  $x_n$ .

- Now, if we look at how we move in function space, we have that

$$\begin{aligned} f_{\theta(t+\epsilon)}(x) &= f_{\theta(t)}(x) - \epsilon \sum_{p=1}^P \partial_{\theta_p} F|_{\theta(t)}(x) \sum_{n=1}^N (D_{f_\theta(x_n)} \mathcal{C}|_{f_{\theta(t)}}) \partial_{\theta_p} F(x_n) + o(\epsilon) \\ &= f_{\theta(t)} - \epsilon \sum_{p=1}^P (\partial_{\theta_p} F|_{\theta(t)}(x)) \partial_{\theta_p} F|_{\theta(t)}^T(x_n) D_{f_\theta(x_n)} \mathcal{C}|_{f_{\theta(t)}} + o(\epsilon) \\ &= f_{\theta(t)} - \epsilon \sum_{n=1}^N \left( \sum_{p=1}^P (\partial_{\theta_p} F|_{\theta(t)}(x)) \partial_{\theta_p} F|_{\theta(t)}^T(x_n) \right) D_{f_\theta(x_n)} \mathcal{C}|_{f_{\theta(t)}} + o(\epsilon) \\ &= f_{\theta(t)} - \epsilon \left( \sum_{n=1}^N \Theta(x, x_n) \right) D_{f_\theta(x_n)} \mathcal{C}|_{f_{\theta(t)}} + o(\epsilon) \end{aligned}$$

- We introduced the NTK  $\Theta(x, x') = \sum_{p=1}^P \partial_{\theta_p} F(x) \partial_{\theta_p} F(x')$ : it informally represents how sensitive the value of  $f_\theta(x')$  is to a change of label  $y$  of a point at  $x$  in a gradient step.
- Conceptually, the NTK represents the following picture: in functional space, we have a convex cost functional, and a manifold parametrized by  $F$ , and we project the gradient to the tangent space to that manifold (hence the name ‘tangent’).
- In functional analysis, this corresponds to the notion of kernel gradient: we have  $\partial_t f_{\theta(t)} = -\nabla_{\Theta(t)} \mathcal{C}|_{f_{\theta(t)}}$ , which is a pretty-looking equation; the idea is that  $\mathcal{C}$  depends (a priori) on the values of  $f_{\theta(t)}$  at all the points  $x'$ , and to understand what happens at  $x$ , we should ‘sum over all the derivatives, and aggregate these dependences via  $\Theta(x, x')$ ’ (like we would sum over indices if we were in finite dimension).
- Of course, the equation is a priori just hiding the complexity in what  $\Theta(t)$  contains.
- Let us briefly discuss the vector output case  $d_{\text{out}} > 1$ :

- The kernel becomes a  $d_{\text{out}} \times d_{\text{out}}$ -valued kernel:  $\Theta_{ij}(x, x_n)$  then becomes  $\sum_{p=1}^P \partial_{\theta_p} F_i(x) \partial_{\theta_p} F_j(x')$ , and it can similarly represent a gradient descent step.
- What is interesting about this? As it will turn out, the key statements are that as  $n_1, \dots, n_{\ell-1} \rightarrow \infty$ , we have the following:
  - The NTK at initialization has a deterministic, analytically computable limit.
  - This limit is constant throughout training (i.e. for any time interval  $[0, T]$ ,  $\Theta_{\theta(t)} \rightarrow \Theta_{\theta(0)}$  as  $n_1, \dots, n_{\ell-1} \rightarrow \infty$ ), allowing one to derive an explicit form for the training dynamics.
- This will have many consequences, in particular when the cost function is quadratic  $\mathcal{C}(f) = \sum_{n=1}^N |f(x_n) - y_n|^2$ :
  - We will have that we start at a random Gaussian point  $f_{\theta(0)}$  given by the ‘activation kernel’.
  - The convergence speed will be governed by the eigenvalues of the kernel matrix  $(\Theta(x_m, x_n))_{1 \leq m, n \leq N}$ .
  - As we will see below, we just perform kernel ridge regression with respect to  $\Theta$  as the training time goes to infinity.
  - If the NTK is positive-definite, then we will converge to a global minimum, not a local one.
  - The double-descent phenomenon for neural networks can be explained (semi-rigorously) using the NTK picture.
  - The NTK picture has some weaknesses, in particular with regards to its lack of feature learning, leading to the question of how good an approximation it is of the whole learning process, and that has led to some (semi-rigorous) descriptions which are very important in practice.
- Where did the NTK come from?
  - If one focuses on quadratic loss scalar case, and one the tries to understand the spectrum of the Fisher information matrix (as a means to understand the Hessian of the cost, to clarify the question of whether one encounters local minima, as was initially postulated), we have that it is

$$\left( \sum_{n=1}^N \partial_{\theta_p} F(x_n) \partial_{\theta_q} F(x_n) \right)_{p,q}$$

and if one wonders a possible  $P \rightarrow \infty$  infinite, we obtain that it is the same (except for zero eigenvalues) as that of

$$\left( \sum_{p=1}^P \partial_{\theta_p} F(x_m) \partial_{\theta_p} F(x_n) \right)_{m,n}$$

which is the NTK kernel matrix (the idea is simply that for any matrix,  $A$ , the matrices  $AA^T$  and  $A^T A$  have the same spectrum, except for zero eigenvalues).

- Then realizing its connection with the question of activation kernels and the question of understanding the training dynamics led to the latter presentation.

#### 4.7. Proof Ideas.

- Let us focus on the scalar output case, as it is much easier in terms of notation, and as the vector output case is conceptually not different.
- The easiest regime to prove the existence of a limit is the sequential limit: first let  $n_1 \rightarrow \infty$ , then  $n_2 \rightarrow \infty$ , etc., up to  $n_{\ell-1} \rightarrow \infty$ ; the ‘more ideal’ regime is  $\min(n_1, \dots, n_{\ell-1}) \rightarrow \infty$  (i.e. as long as each layer is large enough, we are close enough to the desired limit); this is however a little bit more complicated.
- The idea is that because we have the right scaling, we have laws of large numbers guaranteeing that we converge to the right things: the pre-activations  $(\tilde{\alpha}_{\ell,k})_{k=1, \dots, n_k}$  converge to Gaussians at each layer, the activations converge to  $\sigma$  of a Gaussian, and then when we bring them to the next layer, we take a sum divided by the square root the number of activations we sum, leading to a Gaussian again (by the central limit theorem).
- First thing to write down: the useful (putative) limits (their justification will directly come from the proof)

$$\Sigma^{(1)}(x, x') = \frac{1}{n_0} x^T x + \beta^2,$$

$$\Sigma^{(\ell+1)}(x, x') = \mathbb{E}_{f \sim \mathcal{N}(0, \Sigma^{(\ell)})} [\sigma(f(x)) \sigma(f(x'))],$$

then

$$\dot{\Sigma}^{(\ell+1)}(x, x') = \mathbb{E}_{f \sim \mathcal{N}(0, \Sigma^{(\ell)})} [\dot{\sigma}(f(x)) \dot{\sigma}(f(x'))],$$

and

$$\Theta_{\infty}^{(1)} = \Sigma^{(1)},$$

and

$$\Theta_{\infty}^{(\ell+1)} = \Theta_{\infty}^{(\ell)} \dot{\Sigma}^{(\ell+1)} + \Sigma^{(\ell+1)}.$$

- Let us prove by induction on the number of layers  $L$  that the activations  $(\alpha_i^{(L)})_{i=1, \dots, n_L}$  converge to  $n_L$  i.i.d. Gaussian random functions, where each component is a centered Gaussian with activation kernel  $\Sigma^{(L)}$ .
  - For  $L = 1$ , we just have that  $f_{\theta}$  is the random Gaussian function  $\frac{1}{\sqrt{n_0}} W^{(0)} x + \beta b^{(0)}$  (again think of  $x$  as just fixed). The covariance  $\mathbb{E}[f_{\theta}(x) f_{\theta}(x')] = \Sigma^{(1)}(x, x')$  as needed.
  - Assuming the result true for  $L$ , to prove it for  $L + 1$ , we have that, conditionally on  $\alpha^{(L)}$ ,  $f_{\theta}$  is also a random Gaussian function

$$\frac{1}{\sqrt{n_L}} \alpha^{(L)}(x) W^{(L)} + \beta b^{(L)},$$

so it is centered and of covariance between  $f_{\theta}(x)$  and  $f_{\theta}(x')$  is given by

$$\frac{1}{n_L} \alpha^{(L)}(x) \alpha^{(L)}(x') + \beta^2$$

and now as  $n_L \rightarrow \infty$ , we have that this converges (by the induction assumption and the law of large numbers) to the (deterministic) expectation

$$\mathbb{E}_{f \sim \mathcal{N}(0, \Sigma^{(L)})} [\sigma(f(x)) \sigma(f(x'))] + \beta^2,$$

so this corresponds to the induction formula for  $\Sigma^{(L)}$ .

- If we have several outputs (i.e.  $n_{L+1} > 1$ ), for any  $1 \leq i < j \leq n_{L+1}$ , we have that the functions  $\alpha_i^{(L+1)}$  and  $\alpha_j^{(L+1)}$  are independent.
- The proof of convergence at initialization of the NTK is exactly the same. We just split the terms in the dot product  $\nabla_{\theta} f_{\theta}(x) \nabla_{\theta} f_{\theta}(x')$  across layers, and we perform the same induction to go from  $L$  to  $L + 1$  ( $L = 1$  is easy):
  - For each parameter  $\theta_p$  of the first  $L$  layers, we have that the derivatives of  $f_{\theta}$  with respect to  $\theta_p$  is

$$\sum_{k=1}^{n_L} \frac{1}{\sqrt{n_L}} \partial_{\theta_p} \tilde{\alpha}_k^{(L)}(\theta) \frac{1}{\sqrt{n_L}} \dot{\sigma}(\tilde{\alpha}_k^{(L)}(\theta)) = \frac{1}{n_L} \sum_{k=1}^{n_L} \partial_{\theta_p} \tilde{\alpha}_k^{(L)}(\theta) \dot{\sigma}(\tilde{\alpha}_k^{(L)}(\theta)).$$

- Summing these up we find  $\Theta_{\infty}^{(L)} \dot{\Sigma}$
- Similarly, the components of the NTK are independent.
- Now, if we sum over the parameters of the last layer, we easily find  $\Sigma^{(\ell+1)}$ : we have the  $L$  layers.
- Then there is the question of stability during training which is much more delicate: why does the neural network not evolve during training?
- The case  $L = 1$  is trivial: the NTK is independent of the parameters.
- The simplest is to first look at one hidden layer ( $L = 2$ ) follow this heuristic: we can examine how much each neuron activation is expected to change during training with our regime, as  $n_1 \rightarrow \infty$ .
- Since the output is rescaled by  $1/\sqrt{n_1}$ , the change that the  $n$  neurons need to incur is  $\mathcal{O}(\epsilon/\sqrt{n_1})$  in a gradient step of size  $\epsilon$ . Hence, the change of the NTK is of order  $n_1 \mathcal{O}(\epsilon/n_1 \cdot n_1) = \mathcal{O}(\epsilon/n_1)$  (since we have  $n_1$  neurons, when we compute the derivative of the output function  $f_{\theta}$  with respect to the parameters, we get a factor  $1/\sqrt{n_1}$ ), and so it doesn't change very much at all!
- For more layers, we have the same result: the change of the last layer is the same, and the first  $L$  layers can be viewed as a neural network being trained through a lens that doesn't change and is a bounded deformation of the cost.
- To formalize this, we need something like Grönwall's inequality: the amount by which the NTK changes is determined in terms of the NTK itself, and so if the NTK is not exploding, the amount of change can be made arbitrarily small for large enough neural networks.

#### 4.8. About Positive-Definiteness and Kernel Methods.

- So, the NTK  $\Theta_{\theta}(x, x')$  is a kernel, which can be (naively, but fruitfully) just be thought as a (positive semi-definite) symmetric matrix indexed by pairs of 'indices'  $x, x'$ .
- This has some important consequences for the convergence of neural networks
  - From the NTK picture above, if  $\Theta_{\infty}^{(L)}$  is a positive-definite kernel and  $\mathcal{C}$  is a convex cost functional (as is customarily the case), then the cost will keep going down until we hit a global minimum.
  - Now, how can we guarantee that  $\Theta_{\infty}^{(L)}$  is positive-definite?
  - Since  $\Theta_{\infty}^{(L)}$  is a sum of kernels including  $\Sigma_{\infty}^{(L)}$ , it is enough to show that the latter is positive-definite.

- In fact, to show that  $\Sigma_\infty^{(L)}$  is positive-definite, it is enough to show that  $\Sigma_\infty^{(2)}$  is positive-definite: if we assume  $\Sigma_\infty^{(L-1)}$  is positive-definite, then  $\Sigma_\infty^{(L)}$  is, as well, as for any  $c_1, \dots, c_n \neq 0$ , we have  $\sum_{i,j} c_i c_j \Sigma_\infty^{(L)}(x_i, x_j) = \mathbb{E} \left[ \sum (c_i f(x_i))^2 \right] + \beta^2$ , which is nonzero if  $(f(x_i)_{i=1, \dots, n})$  is a non-degenerate Gaussian, by the induction assumption.
- One can show that  $\Sigma_\infty^{(2)}$  is positive-definite when restricted to the unit sphere  $\mathbb{S}^{\text{din}} = \{x \in \mathbb{R}^{\text{din}} : \|x\| = 1\}$  and  $\sigma$  is not a polynomial.
- If  $\sigma$  is not a polynomial, it has an expansion into Hermite polynomials that contains infinitely-many terms.
- Now there is (somehow miraculously) a theorem saying that an isotropic kernel on the sphere  $\mathbb{S}^{\text{din}}$ , i.e. of the form  $K(x, x') = \nu(x^T x')$  is positive-definite if and only if  $\nu : [-1, 1] \rightarrow \mathbb{R}$  has an infinite number of nonzero even and odd modes in its power series expansion.
- How do we bridge things?
- We can then see that  $\Sigma_\infty^{(2)}$  has an explicit form:  $\hat{\mu} \left( \frac{n_0 \beta^2 + x^T x'}{n_0 \beta^2 + 1} \right) + \beta^2$ , where  $\hat{\mu}$  is the ‘Gaussian dual of  $\mu(x) = \sigma(x\sqrt{1/n_0 + \beta^2})$ .
- The Gaussian dual  $\hat{\mu}$  of a Lipschitz function  $\mu$  is given by

$$\hat{\mu}(\rho) = \mathbb{E}_{(x, x') \sim \mathcal{N} \left( 0, \begin{pmatrix} 1 & \rho \\ \rho & 1 \end{pmatrix} \right)} [\mu(x) \mu(x')].$$

- A small computation shows that if  $\mu$  has a Hermite expansion  $\sum a_i h_i$ , then  $\hat{\mu}(\rho) = \sum_{i=0}^{\infty} a_i^2 \rho^i$ . So  $\hat{\mu}$  has an infinite number of even coefficients in its expansion.
- Now we have  $\Sigma_\infty^{(2)}(x, x') = \nu(x^T x')$  with

$$\nu(\rho) = \beta^2 + \sum_{i=0}^{\infty} a_i^2 \left( \frac{n_0 \beta^2 + \rho}{n_0 \beta^2 + 1} \right)^i$$

which contains an infinite number of odd and even terms in  $\rho$  (since the sum contains a infinite number of nonzero terms, each of which contains even and odd powers of  $\rho$ ).

- Kernel methods form a deep subject, and there are many things that can be said about them, but perhaps the simplest is to say that conceptually they correspond to a Gaussian a posteriori given function: imagine that we want to define a Gaussian on a space of functions  $\mathbb{R}^{\text{din}} \rightarrow \mathbb{R}$  (don’t worry too much about the topology), we need a mean (which is easy, we can take 0) and a covariance: we need as a result a kernel  $\mathcal{K}$ .
- Then what we want to do is Bayesian inference:
  - Assuming that we have as an a priori that our function  $f$  is sampled according to  $\mathcal{N}(0, \mathcal{K})$
  - Assume that we have observed  $f(x_i) = y_i$ .
  - We can ask, for any  $x$  what is the expectation  $\mathbb{E}_{f \in \mathcal{N}(0, \mathcal{K})} [f(x) | f(x_i) = y_i]$ : this is the kernel ridgeless regression.
  - If we allow the observations to have iid Gaussian noises  $\lambda z_i > 0$  with  $z_i \sim \mathcal{N}(0, 1)$ , we obtain the kernel ridge regression

$$\mathbb{E}_{f \sim \mathcal{N}(0, \mathcal{K})} [f(x) | f(x_i) = y_i + \lambda z_i]$$

- This is ridge regression with a regularization parameter  $\lambda > 0$  (when  $\lambda = 0$ , we speak of ‘ridgeless regression’): it represents how much we are ready to go in the ‘unlikely’ region to fit the  $(x_i, y_i)$  data points.
- Kernel methods have a long history:
  - For a long time (between the mid 1990s and ~2010), kernel methods were viewed as generally superior to neural networks because of their easier-to-interpret probabilistic formulation, their relative flexibility (you can engineer a lot of kernels) and (perhaps) because people seemed attracted by their mathematical sophistication.
  - They then fell out of favor from the applied machine learning world (perhaps due to the fact that their inference doesn’t scale well when the number of data samples grows); the random features
  - They regained some popularity in the 2018-2020 period in part thanks to the NTK.
  - Generally speaking, they remain a very powerful tool to understand various problems mathematically, in particular neural networks, thanks to the NTK connection.
  - The cleanest derivation of the double-descent phenomenon is in the context of random feature approximation of kernel methods.

#### 4.9. Random Features.

- The NTK results suggest the following naive (and partly right, but still useful) view: in some regime, neural networks are a kind of a random features model.
  - This allows one to give an understanding the double-descent curve for neural networks.
- What is the random features model?
  - Take a kernel  $\mathcal{K}$  and sample random functions  $f_1, \dots, f_P : \mathbb{R}^{d_{in}} \rightarrow \mathbb{R}$  in a centered way with a covariance kernel  $\mathcal{K}$  (again, don’t mind too much the topology).
  - Now, given data points  $x_1, \dots, x_N \in \mathbb{R}^{d_{in}}$  and labels  $y_1, \dots, y_P$ , consider the following model:

$$f_{\theta}^{RF} = \frac{1}{\sqrt{P}} (\theta_1 f_1 + \dots + \theta_P f_P) = \frac{1}{\sqrt{P}} \theta^T F^{(P)},$$

where  $F^{(P)}$  is the vector of functions

$$F^{(P)} = (f_1, \dots, f_P),$$

so that  $\theta^T F^{(P)}(x_i) \in \mathbb{R}$  for all  $i$ , and hence if we write  $Z = (F^{(P)}x)^T$ , we have  $Z \in \mathbb{R}^{N \times P}$ ,  $\theta \in \mathbb{R}^P$ ,  $Y \in \mathbb{R}^N$ , and we are looking at trying to fit  $Y$  with  $Z\theta$ .

- We then perform gradient descent (starting from  $\theta = 0$ ) to optimize  $f_{\theta}^{RF}$  in the same ways as a neural network.
- We have that as  $P \rightarrow \infty$ ,  $\left(f_{\theta(t)}^{RF}\right)_{t \geq 0}$  also converges to the kernel spanned by the features.
- The evolution in time hence yields the minimal norm solution that minimizes  $\|Z\theta - Y\|^2$ .

- So, in some sense, if we can show the double descent curve for this random features model, this will suggest that we can see a similar phenomenon for neural networks (assuming neural networks are close enough to random features models).

#### 4.10. Double Descent in Random Features.

- Where does the double-descent curve come from? We are looking at a kernel method with the random kernel  $(F^{(P)})^T F^{(P)}$ ; there is a small phase transition when we go from under-parametrized  $P < N$  to over-parametrized  $P > N$ .
- If  $P < N$  (the number of parameters is less than the number of examples), we will generally under-fit, and find the ordinary least-squares solution:
  - $\hat{\theta} = (Z^T Z)^{-1} Z^T Y$ .
  - How to remember this formula? A simple way to see this is by writing the SVD and seeing that all we want is to invert the nonzero eigenvalues... another way is just by ‘dimensional analysis’, seeing that we need something  $\mathbb{R}^N \rightarrow \mathbb{R}^P$  from  $Z : \mathbb{R}^P \rightarrow \mathbb{R}^N$  and if  $P > N$ , it is good to start from  $Z^T$  and correct that with what would make sense if we were in the square case.
- If  $P > N$ , we can ‘generically’ (assuming all  $x_i$ ’s are distinct, and that the features restricted to the  $x_i$ ’s are linearly independent) fit and the minimal norm solution will read:
  - $\hat{\theta} = Z^T (Z Z^T)^{-1} Y$
  - Again, this can be seen from SVD, and it is the only other choice that makes sense in terms of dimensionality (if  $N > P$ , we want to rely on the ‘small’ square matrix  $Z Z^T : \mathbb{R}^{N \times N}$ , rather than its non-invertible ‘dual’  $Z^T Z : \mathbb{R}^{P \times P}$ ).
- Intuitively, the idea is that when  $P$  is very small, we project, so we are not too sensitive to the noise of the labels; when  $P$  is very large, we pick among all the solutions the one with the smallest norm, so we are not too sensitive to the noise either, while when  $N = P$ , we can exactly fit, and hence overfit.
- One still needs to explain the double-descent curve per se: why does the error initially decrease, and then rebound?

#### 4.11. An analytical treatment.

- Here is a simple model (due to Mei and Montanari) that shows how the error can increase and decrease again
- We assume that we have a ‘truth’ model  $Y = X\theta_* + \epsilon$  for some fixed  $\theta_* \in \mathbb{R}^P$ , for some random Gaussian matrix  $X \in \mathbb{R}^{N \times P}$ , and where the noise is Gaussian i.i.d. of variance  $\sigma^2$ .
- If we look at the test error of the least-squares solution  $\hat{\theta}$ , we have that

$$\mathcal{E}(P) = \mathbb{E}_{x \in \mathcal{N}(0, \text{Id}_P)} \left[ \left( \hat{\theta}^T x - \theta_*^T x \right)^2 \right].$$

- Since  $x \in \mathcal{N}(0, \text{Id}_P)$ , we have that  $\mathbb{E}[xx^T] = \text{Id}_P$ , and by independence of  $x$  from  $\hat{\theta}$ , we have that this simplifies to

$$\mathcal{E}(P) = \mathbb{E} \left[ \|\hat{\theta} - \theta_*\|^2 \right].$$

- Now, the statement we will show is that

$$\mathcal{E}(P) = \begin{cases} \sigma^2 \frac{P}{N-P-1} & \text{if } P < N-1 \\ \|\theta_*\|^2 \left(1 - \frac{N}{P}\right) + \sigma^2 \frac{N}{P-N-1} & \text{if } P > N+1 \end{cases}$$

- For  $P < N$ , we have  $\hat{\theta} - \theta_* = (X^T X)^{-1} X^T \varepsilon$  and

$$\mathbb{E} \left[ \|\hat{\theta} - \theta_*\|^2 \right] = \sigma^2 \mathbb{E} \left[ \text{tr} \left( X (X^T X)^{-2} X^T \right) \right] = \sigma^2 \mathbb{E} \left[ \text{tr} (X^T X)^{-1} \right].$$

- For  $P > N$ , we have

$$\hat{\theta} - \theta_* = \left( X^T (X X^T)^{-1} X - \text{Id} \right) \theta_* + X^T (X X^T)^{-1} \varepsilon$$

- Now, we need to show that if  $X$  is a standard Gaussian matrix of size  $N \times P$ , we have:

$$\mathbb{E} \left[ \text{tr} (X^T X)^{-1} \right] = \frac{P}{N - P - 1}.$$

- Here is a statement: let  $X$  be a standard Gaussian matrix of size  $N \times P$ , and let  $W = X X^T$ , then if  $N > P$ , we have

$$\begin{aligned} \mathbb{E} [W^{-1}] &= \frac{1}{P - N - 1} \text{Id} \\ \mathbb{E} [\text{tr} (W^{-1})] &= \frac{N}{P - N - 1} \end{aligned}$$

- Proof:

- by symmetry,  $\mathbb{E} [W^{-1}]$  commutes with any matrix, so it must be a multiple of the identity;
- we just need to compute the top-left entry  $\mathbb{E} [(W^{-1})_{11}]$  as a result.
- write  $X$  into blocks  $\begin{pmatrix} x \\ Z \end{pmatrix}$ , where  $x$  is the first row and  $Z$  the rest.

Then

$$W = \begin{pmatrix} x x^T & x Z^T \\ Z^T x & Z Z^T \end{pmatrix},$$

and the block-inverse formula yields  $\left( x x^T - x Z^T (Z Z^T)^{-1} Z x^T \right)^{-1}$ , which is  $(x P_{\text{null}} x^T)^{-1}$ , where  $P_{\text{null}} = \text{Id}_N - P_{\text{row}}$  and  $P_{\text{row}}$  is the projection on the span of the rows of  $Z$ .

- We find that  $x P_{\text{null}} x^T$  is a sum of  $P - N + 1$  squares of Gaussians, so a chi-square  $\chi_\nu^2$  with  $\theta = P + N - 1$  degrees of freedom.
- We have that  $\mathbb{E} [(\chi_\nu^2)^{-1}] = 1/(\nu - 2)$ :

\* The density of a chi square is  $\frac{1}{2^{\nu/2} \Gamma(\nu/2)} x^{\nu/2-1} e^{-x} \mathbf{1}_{\mathbb{R}_+}$

\* Now, if we compute the expectation of the inverse, we have

$$\frac{1}{2^{\nu/2} \Gamma(\nu/2)} \int_0^\infty x^{-\nu/2-2} e^{-x} dx = \frac{1}{2} \frac{\Gamma(\nu/2 - 1)}{\Gamma(\nu/2)},$$

and thanks to the Gamma function recursion  $\Gamma(\alpha + 1) = \alpha \Gamma(\alpha)$ , we obtain the result.

- Now, to prove the theorem:

– For  $P < N$ , we have  $\hat{\theta} - \theta_* = (X^T X)^{-1} X^T \epsilon$  and

$$\begin{aligned} \mathbb{E} \left[ \|\hat{\theta} - \theta_*\|^2 \right] &= \sigma^2 \mathbb{E} \left[ \text{tr} \left( X (X^T X)^{-2} X^T \right) \right] \\ &= \sigma^2 \mathbb{E} \left[ \text{tr} (X^T X)^{-1} \right] \\ &= \sigma^2 \frac{P}{N - P - 1} \end{aligned}$$

– For  $P > N$ , we have

$$\hat{\theta} - \theta_* = \left( X^T (X X^T)^{-1} X - \text{Id} \right) \theta_* + X^T (X X^T)^{-1} \epsilon$$

– Now  $\left( X^T (X X^T)^{-1} X - \text{Id} \right) = -P_{\text{null}(X)}$

– Since the two parts  $P_{\text{null}} \theta_*$  and  $X^T (X X^T)^{-1} \epsilon$  are independent and orthogonal in expectation and since  $\mathbb{E} \left[ \|P_{\text{null}} \theta_*\|^2 \right] = \left( 1 - \frac{N}{P} \right) \|\theta_*\|^2$ , we get the desired result.

- This shows the phenomenon of ‘no overfitting as we over-parametrize’ in this simple model of random features.
- Something that it lacks is a true ‘double-descent’ curve: in this model, when the number of parameters is very small there is no error.
- To obtain a ‘true’ double-descent curve, one needs
- The connection with neural networks (for now) is typically understood heuristically by comparing neural networks to a random features model.

#### 4.12. Neural Network Hessian and the Origin of the NTK.

- The context in which the NTK came was a belief around that time that neural network optimization with a large number of parameters would lead to local minima.
- A parallel (that is now believed to be wrong) had been made with spherical spin glass models, which are models of random functions that are analytically tractable: due to remarkable properties, the spectrum of their Hessian at critical points can be studied explicitly.
- From this came the idea to study the Hessian of the cost function of a neural network and its spectrum: this is what ought to give an idea about the structure of (what was then thought to be) the family of critical points of the network loss function.
- If we want to study the Hessian of a neural network cost, we get something like

$$H_{ij} = \partial^2 \mathcal{C} (\partial_{\theta_i} F, \partial_{\theta_j} F) + \partial \mathcal{C} (\partial_{\theta_i} \partial_{\theta_j} F),$$

where we should understand  $\partial^2 \mathcal{C}$  as a quadratic form on the functional space (‘eating’ the two ‘directions’  $\partial F$ ) and  $\partial \mathcal{C}$  as a linear form (‘eating’ the functional direction  $\partial_{\theta_i} \partial_{\theta_j} F$ ).

- In the least-squares case, we we get two terms:

$$H = A + B,$$

where  $A_{ij} = \sum \partial_{\theta_i} f_{\theta} \partial_{\theta_j} f_{\theta}$  and  $B_{ij} = \sum_{n=1}^N \sum_{i=1}^{d_1} e_{n,j} \partial_{\theta_i}^2 f_{\theta}$ , where  $e_{n,i} = y_{n,i} - f_{\theta,i}(x_n)$ .

- The idea is that as the network gets closer to optimality, the latter term vanishes, making the first relevant to understand the shape around the global minimum.
- The first matrix  $A$  is in fact the Fisher information matrix.
- Since it turns out that the Fisher matrix grows with the number of parameters, the idea came to study its ‘dual’, which shares the same nonzero eigenvalues, but doesn’t grow: this is how the NTK was born.
- Then the regime dictated by the activation kernel suggested to study the NTK limit at initialization.
- And finally, the interpretation of the role of each data point on the prediction of a test data point came.
- And then it was realized (first empirically, and then mathematically) that (in that regime) the NTK stays stable during training.

#### 4.13. Feature Learning and Neural Networks.

- In the end, the NTK comes from the parametrization that was chosen so that the activation kernel would be finite with ‘macroscopic initialization’ of the random.
- This leads to the training of neural networks with no ‘feature learning’: the dimensions along which we learn are fixed at initialization, and cannot depend on the data.
- It is in fact not clear that this is exactly what we want to study... what if we scaled the neural network differently?
- For instance, if we parametrized the neural network with  $1/n_\ell$  rather than  $1/\sqrt{n_\ell}$  to go from layer  $\ell$  to  $\ell + 1$ ?
- At first glance, that looks a bit problematic: the NTK will go to zero...
- But if we train the neural network for a very long time at finite size, it is reasonable to expect that something must happen.
- This is called sometimes the ‘active’ or ‘feature-learning’ regime, in opposition to the ‘lazy’ or ‘kernel’ regime defined by the NTK.
- Mathematically, this regime is much harder to study, even though there has been some progress on several fronts:
- For a two-layer neural network (i.e. only the layer  $\ell = 1$  is hidden) with fixed parameters between  $\ell = 1$  and the output, we can leverage the permutation symmetry to describe the dynamics of parameters and functions in terms of a ‘population’ of parameters.
- This regime can be understood for two-layered neural networks, using the so-called mean-field approach.
- This was also developed by people at EPFL (like Lénaïc Chizat).
- The key idea is that one should normalize things differently: instead of dividing by  $1/\sqrt{n_1}$  the outputs of the neurons in the last layer, one divides them by  $1/n_1$ .
- As a result, the neuron activations ‘move’ by quite a lot during training: this corresponds to a so-called ‘feature-learning’ regime.
- Typically assuming the output dimension  $n_2 = 1$ , one write this as

$$f_\theta(x) = \frac{1}{n_1} \sum_{i=1}^{n_1} \sigma_*(x; \theta_i),$$

where  $\sigma_* = a_i \sigma(\langle w_i, x \rangle + b_i)$ , where  $\sigma$  is the usual nonlinearity (note that this is just a reformulation of the neural net parametrization that we had before).

- In this formulation, we see a permutation symmetry emerge: each of the  $\sigma_*(x; \theta_i)$  plays the same role.
- Now, assuming we are in a least-squares scenario, we want to minimize

$$\mathbb{E}_{x \sim \mathbb{P}_{\text{data}}} \left[ (f_*(x) - f_\theta(x))^2 \right].$$

- This can be rewritten as

$$C + \frac{2}{n_1} \sum_{i=1}^{n_1} V(\theta_i) + \frac{1}{n_1^2} \sum_{i,j=1}^{n_1} U(\theta_i, \theta_j),$$

where

$$\begin{aligned} V(\theta) &= -\mathbb{E}[f_*(x) \sigma_*(x; \theta)], \\ U(\theta_1, \theta_2) &= \mathbb{E}[\sigma_*(x; \theta_1) \sigma_*(x; \theta_2)]. \end{aligned}$$

- If we think of the parameters as being described by a measure

$$\rho = \frac{1}{n_1} \left( \sum_{i=1}^{n_1} \delta_{\theta_i} \right)$$

Then we can rewrite things as

$$C + 2 \int V(\theta) \rho(d\theta) + \int U(\theta_1, \theta_2) \rho(d\theta).$$

- Then, when we do gradient descent, we get a nice equation for the evolution of the measure  $\rho$ :

$$\begin{aligned} \partial_t \rho_t &= 2 \operatorname{div}_\theta (\rho_t \nabla_\theta \Psi(\theta, \rho_t)) \\ \Psi(\theta; \rho) &= V(\theta) + \int U(\theta, \theta') \rho(d\theta'). \end{aligned}$$

- This scales nicely as  $n_1 \rightarrow \infty$  to a corresponding equation on the space of measures.
- For two layers, one can prove that the neural network converges to a global optimum as well, using this formulation.

#### 4.14. Neural Network Hyperparameter Scaling.

- Ultimately, an important role of the deep learning theory that has been developed in the recent years is to understand the role of discrete hyper-parameters:
  - The width
  - The embedding dimensions
  - The depth has recently appeared
- And of continuous hyper-parameters:
  - The parametrization rescalings (*a*)
  - The initialization variances (*b*)
  - The learning rates of the various layers (*c*)
- The scaling of these continuous hyper-parameters what is sometimes now called the *a, b, c*-parametrization: we rescale the initialized variables by  $n^{-a_\ell}$ ,  $n^{-b_\ell}$ ,  $\eta n^{-c_\ell}$  for some constant  $\eta$ .

- The question of how to scale these parameters together to converge to a meaningful limit, if we want to grow a neural network, has been a particularly important one. .
- A simple attractive approach to scale hyper-parameters in the recent years has been provided by the idea of  $\mu$ -parametrization: the idea is that we should make each parameter move as much as possible as the sizes go to infinity, without exploding.
- This rescaling will typically reduce the size of the parameters and learning rates in the last layer, increase it in the intermediate layers (compared to the NTK regime, to something like the mean-field limit), and even increase it more in the first layer.
- This gives a family of various scaling regimes as the sizes go to infinity, along which one can expect that (for given hyperparameters), things behave roughly the same; this is now used as the basis for hyper-parameter optimization (where we optimize at one scale and then transfers to a larger scale).

#### 4.15. Sparsity.

- An alternative route that has given interesting results is the study of linear networks, i.e. networks without nonlinearities
- While on the surface this is a trivial problem (we will output a linear function), to study the training dynamics of a linear neural network near initialization is far from trivial.
- If we start very close to the origin (even closer than what we would have from  $\mu$ -parametrization), we can see a dynamics that suggests that we can discover sparse representations of functions: we move from saddlepoint to saddlepoint of the loss, where the saddlepoints correspond to low-rank approximations.
- Similarly, if we add an  $L^2$  regularization on the weights (which is something that e.g. AdamW would implicitly correspond to), we find that this biases things towards low-rank solution
- While this picture is not completely clear, it provides a compelling picture of what we want, beyond the NTK regime: sparsity, i.e. the presence of simple descriptors behind the functions at hand.
- There has been some recent progress on making sense of this: neural networks optimize in some sense the circuit size of a description of the data.
- This echoes the ideas around Kolmogorov complexity at the beginning of the course; the interest of the circuits is that things are a bit better behaved than Turing machines.

## 5. REINFORCEMENT LEARNING: TOWARDS AGI

### 5.1. Towards Reinforcement Learning.

- We started this course with a simple objective: prediction, which led us to information theory
- From information theory, we formulated the question of the optimal compression
- From optimal compression, we obtained the ideas of Kolmogorov complexity

- From there we departed to ‘feasible skies’ with neural networks playing the role of the computational class
- We studied the theory of neural network optimization and saw what we obtain from that
- We presented LLMs as approximations of the universal prediction/compression algorithms
- But intelligence is usually not understood only in terms of ‘predicting’: there is the question of how we act in an environment.

## 5.2. Reinforcement Learning (RL).

- Reinforcement learning (RL) is first about agents that act in an environment.
- The focus of RL is on the process whereby an agent learns to optimize (via its actions) a quantitatively-measurable objective.
- Let us discuss only the discrete-time and discrete-state and discrete-action case; this is the case that we see for LLMs.
- In RL:
  - the agent is at a state  $s_t$ , where multiple actions are possible; the agent picks an action  $a_t$ , gets a reward (possibly zero)  $r_t$ , and arrives at a new state  $s_{t+1}$ .
  - the environment rules dictate which states the agent lands on and the reward; the rules may be deterministic or random
- An agent’s behavior is modeled by a policy  $\pi$ ; in the context of LLMs, the policy is a probability measure  $\pi_\theta$  outputted by the transformer architecture (we use  $\theta$  to represent the parameters of the network).
- We have a discount factor  $\gamma \leq 1$ , a return function  $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$ . Typically we would want to maximize the expected returns.
- Reinforcement is not unsupervised learning, as it assumes a reward is given by the environment; note that we can do somehow unsupervised learning if we pick rewards for ourselves (this is sometimes called self-supervised learning).

## 5.3. AiXi: Solonomoff-Bayesian-Environment.

- The idea of AiXi is that if we assume we have infinite compute power (or more precisely don’t care about non-computability), there is a universally optimal algorithm to achieve an optimum.
- If we know our environment, we know we should do optimal control, it is ‘just’ an optimization problem (it can be infinite-dimensional, it doesn’t matter if we have infinite compute).
- If we have a prior on environment models, we can do expected Bayesian control: we can just average over all environment models.
- The idea is the simply: take as a priori on the models  $\mathcal{M}$  to be proportional to  $2^{-K(\mathcal{M})}$ , where  $K$  is the Kolmogorov complexity, and do Bayesian updates on that prior according to the observations, and maximize the expected return according to the posterior.
- There are asymptotic optimality results: if the environment is run by a finite-size Turing machine, AiXi will asymptotically prevail.

- This idea, developed in Lugano, was called ‘Universal AGI’ and was deemed an important part of the reasoning that launched DeepMind (one of the co-founders did his PhD on AiXi).

#### 5.4. RL in Practice.

- Obviously, things are not computable in the AiXi world, so in practice life is quite different. Typically, it will be important to have methods that don’t assume we really have a known model of the environment.
- A first classical dichotomy made by Sutton and Barto is about the size of the state-space: small state-space vs large state-space. This leads to tabular methods (we have a vector indexed by all action-state pairs) vs approximate methods (we have a function of states that outputs actions).

#### 5.5. Tabular methods:

- Assuming we have a model of the environment:
  - If the state space contains only one state (and the environment is not deterministic/known), we are facing a bandit problem: we should do a balance between exploration and exploitation.
    - \* If the state space contains several states and we can observe in what space we are, we are in a Markov Decision Process (MDP) scenario, and we can in principle use dynamic programming techniques, provided the computational cost doesn’t blow up.
    - \* If the state space contains several states and there are some hidden states, we have a Partially Observable MDP (POMDP), so we should do inference on top of that.
- Assuming we don’t have a model of the environment:
  - We can implement value-based or action-value-based methods: if we can estimate the value of a state-action pair (how much we expect to get from now on, if we are at a state  $s$  and pick an action  $a$ ), we can just optimize over the action; we don’t need a model of the environment.
    - \* We can estimate the values  $Q_t$  using Monte Carlo methods.
    - \* We can update the values  $Q_t$  using temporal-difference methods: this leads to SARSA and Q-Learning.
      - SARSA is about evaluating how good the value of a state is in expectation by taking the current policy
      - Sarsa learning update (with learning rate  $\alpha$ )

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha (R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, a)),$$

where  $a'$  is the action that was taken next.

- Q-learning is about evaluating how good the value of a state if we take the best move. The update reads

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left( r + \gamma \max_{a'} Q(S_t, a') - Q(S_t, A_t) \right).$$

- We can also have policy-only methods, like policy-gradient methods:
  - \* We don’t try to estimate the value of a state, we just try to optimize the policy:

### 5.6. Approximate methods:

- Typically, what we will end up is parametrizing (in a way or another) some policy  $\pi_\theta$  that will depend on parameters and be a function of the state and output some action (typically by sampling:  $\pi_\theta(s, \cdot)$  will be a probability measure on actions).
- Policy-Gradient Methods:
  - The idea is here to just update the weights to modify the probabilities to maximize the expected reward.
  - Assume that we have actions sampled by softmax from logits parametrized by a neural network (like for LLMs).
  - The REINFORCE algorithm: for each episode, for each step of the episode  $t = 0, \dots, T - 1$ , compute  $G_t$  the returns from  $t$  to  $T$ , and update

$$\theta_{t+1} \leftarrow \theta_t + \alpha G_t \frac{\nabla_\theta \pi(A_t | S_t, \theta_t)}{\pi(A_t | S_t, \theta_t)}.$$

- Actor-Critic Methods
  - The idea here is that we train simultaneously a critic neural network to evaluate the *advantage* of a move over a baseline average
  - The policy gradient becomes  $\nabla_\theta J = \mathbb{E}[\nabla_\theta \log \pi_\theta(a|s) A(s, a)]$

### 5.7. Policy-Gradient Theorem.

- There is a formula:

$$\nabla_\theta J(\theta) = \frac{1}{1-\gamma} \mathbb{E}_{S \sim d_\gamma^{\pi_\theta}, A \sim \pi_\theta(\cdot|S)} [\nabla_\theta \log \pi_\theta(A|S) Q^{\pi_\theta}(S, A)],$$

where

$$Q^{\pi_\theta}(s, a) = \mathbb{E}_{\pi_\theta} \left[ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s, A_t = a \right]$$

$$d_\gamma^{\pi_\theta}(s) = (1-\gamma) \sum_{t=0}^{\infty} \gamma^t \mathbb{P}_{\pi_\theta} \{S_t = s\},$$

where  $d_\gamma^{\pi_\theta}(s)$  is the (normalized) discounted state-visitation probability measure.

- What is it saying intuitively? That how much our value changes if we change the parameters is just given by how much the log-probabilities of our actions
- Proof:
  - Interchanging differentiation and expectations, we get

$$\nabla_\theta J(\theta) = \mathbb{E}[R(\tau) \nabla_\theta \log P_\theta(\tau)]$$

- The trajectory density is written as the product

$$\rho_0(s_0) \prod_{t=0}^{\infty} \pi_\theta(a_t | s_t) \mathbb{P}\{s_{t+1}, r_{t+1} | s_t, a_t\}$$

and so if we take the logs we have

$$\log \rho_0(s_0) + \sum_{t=0}^{\infty} \log \pi_\theta(a_t | s_t) + \sum_{t=0}^{\infty} \log \mathbb{P}\{s_{t+1}, r_{t+1} | s_t, a_t\}$$

- Since the model doesn't depend on  $\theta$ , we have that the derivative with respect to  $\theta$  will only capture

$$\sum_{t=0}^{\infty} \nabla_{\theta} \log \pi_{\theta}(a_t | s_t)$$

- $G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$  ; decompose  $R(\tau)$  into 'past'+ 'future'
- And then

$$\mathbb{E}[R(\tau) \nabla_{\theta} \log \pi_{\theta}(A_t | S_t)] = \mathbb{E}[\nabla_{\theta} \text{past}] + \mathbb{E}[\nabla_{\theta} \text{future}]$$

- Then  $\mathbb{E}[\nabla_{\theta} \text{past}] = 0$  and then we can replace the future returns by the action-state value
- Then we obtain  $\nabla_{\theta} J(\theta) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t \nabla_v \log \pi_{\theta}(A_t | S_t) Q^{\pi_{\theta}}(S_t, A_t)]$

### 5.8. Xent Games.

- One criticism of saying that AGI is about RL (e.g. AiXi) is the choice of the objective: what do we even want to optimize, in practice?
- In real life, we may have an objective on a given day, but how do we prepare for that objective, until you know it?
- Ideally, one should try to anticipate what that objective is and prepare for it, using a model of the world.
- In some sense, short of anything, we will just try to summarize what we have.
- It is a bit like in a class: you get exercises that are supposed to make you ready for the exam... how do we design these exercises? Of course, if we know the exam, it is clear how you train... but the exam is itself supposed to be a sketch of the true objective, so it is random. So how do you prepare for a random exam?
- There is the idea of transfer value: how useful a game is in terms of teaching you another game?
- Then there is the idea of cover: given a 'scope', how do we cover this optimally with some games, so that some level of skill is guaranteed.
- But then there is the deeper question of where the true objective comes from: for e.g. the course material it came from research.
- What does research mean? It means growing one's set of skills, discovering new games, that bring not-yet-covered skills, while being maximally relevant to old skills
- The idea of xent games is to do this with LLMs, by leveraging the implicit vs explicit knowledge gap: the space of games that tap that measure seems very large and exciting.
- Then we have a space of games that we can explore in a suitable fashion: that's the pre-training we should be hoping for!