Neural Networks and Machine Learning

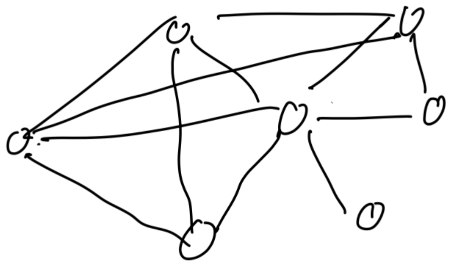
Fabian Ruehle

May 1, 2025

Contents

1 Intro to NNs

Biology: Neurons connected by synapses. Neurons activate other neurons once their activation threshold is passed.

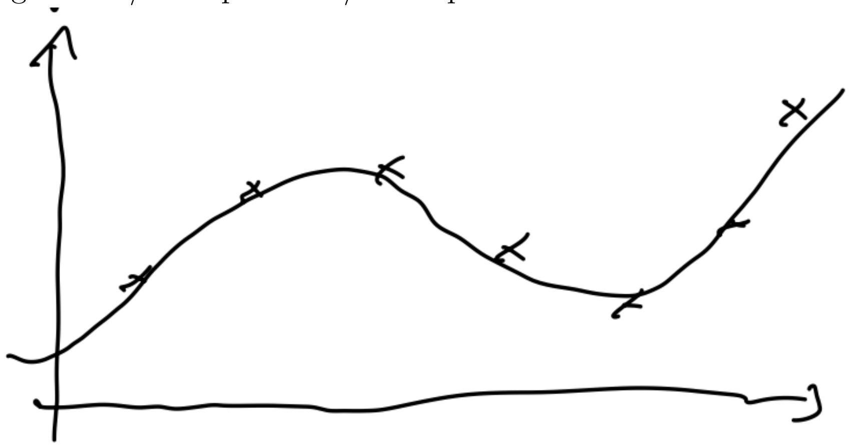


Math: A NN is a parameterized map $f_\theta : \mathbb{R}^{n_{\text{in}}} \to \mathbb{R}^{n_{\text{out}}}$.

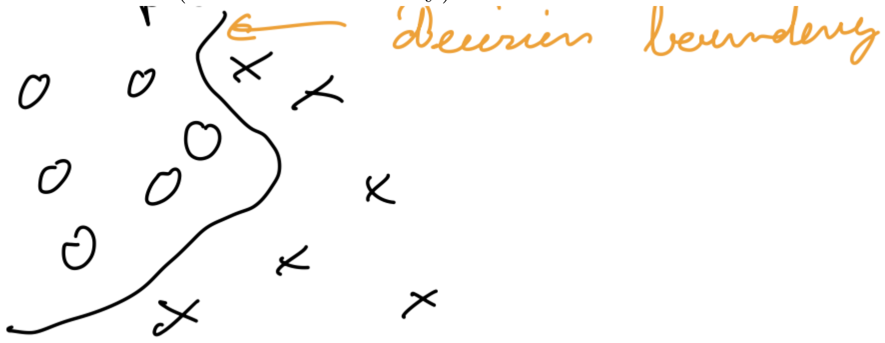
Goal: Adjust the parameters θ s.t. the map does what you want.

Applications of maps:

1. Regression / Interpolation / Extrapolation

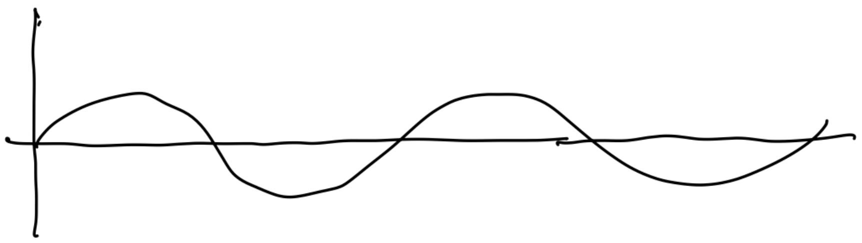


2. Classification (Decision boundary)



3. PDEs

$$\ddot{y}(t) = -\omega^2 y(t), \quad y(0) = 0, \quad \dot{y}(0) = 1$$



4. Games:

$$f : \text{Chess board positions} \to \text{Next moves}$$

5. Chat GPT:

$$\begin{aligned} f : \text{Collections of words} &\to \text{Next word} \\ \text{OED}^N &\mapsto \text{OED} \end{aligned}$$

How do we choose the map $\mathbb{R}^{n_\text{in}} \to \mathbb{R}^{n_\text{out}}$?

Multiply by $n_\text{out} \times n_\text{in}$ matrix W and add a vector $\vec{b}$ of dim n_out:

$$\vec{x}_\text{out} = W \cdot \vec{x}_\text{in} + \vec{b}$$

These are linear (more precisely affine) maps w/ parameters $\theta := \{W_{ij}\} \cup \{b_i\}$

Note: Making it more complicated by making it deeper (nesting multiple affine maps) does not help: Affine maps are closed under composition:

$$W_2(W_1\vec{x} + \vec{b}_1) + \vec{b}_2 = W\vec{x} + \vec{b}\,, \qquad W = W_2 W_1\,, \qquad \vec{b} = W_2\vec{b}_1 + \vec{b}_2\,.$$

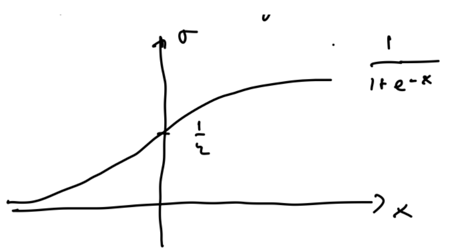
To make the NN more powerful/expressive, we need to break this by inserting non-linearities between the affine maps. Usually one takes unary non-linearities:

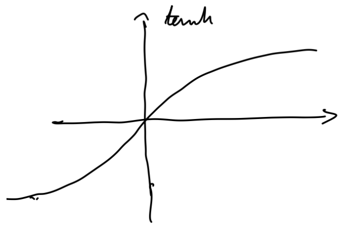
$$\sigma(x) = \frac{1}{1+e^{-x}} \qquad \text{(sigmoid)} \tag{1}$$

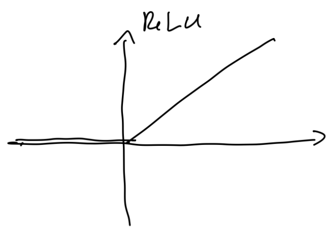
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \text{(hyperbolic tangent)} \tag{2}$$

$$\text{ReLU}(x) = \max(0, x) \qquad \text{(rectified linear unit)} \tag{3}$$

$$\text{softmax}_i(\vec{x}) = \frac{e^{x_i}}{\sum_j e^{x_j}}\,, \qquad 0 < \text{softmax}_i < 1\,, \qquad \sum_i \text{softmax}_i = 1\,. \tag{4}$$

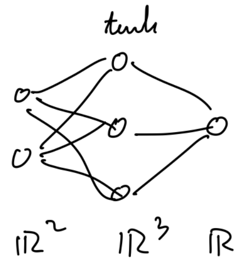






These are approximately linear at the origin and have derivatives $\sim$ original function (important in training).

Example:



$$\begin{array}{rcl} \vec{x}^{\,\text{in}} & \mapsto & \vec{x}^{(1)} = W_1\vec{x}^{\,\text{in}} + \vec{b}_1 \\ \vec{x}^{(1)} & \mapsto & \vec{\ell}^{(1)} = \tanh(\vec{x}^{(1)}) \\ \vec{\ell}^{(1)} & \mapsto & \vec{x}^{\,\text{out}} = W_2\vec{\ell}^{(1)} + \vec{b}_2 \\ \vec{x}^{\,\text{out}} & = & W_2 \cdot \tanh(W_1\vec{x}^{\,\text{in}} + \vec{b}_1) + \vec{b}_2 \end{array}$$

Why this parameterization?

Compare with Taylor expansion

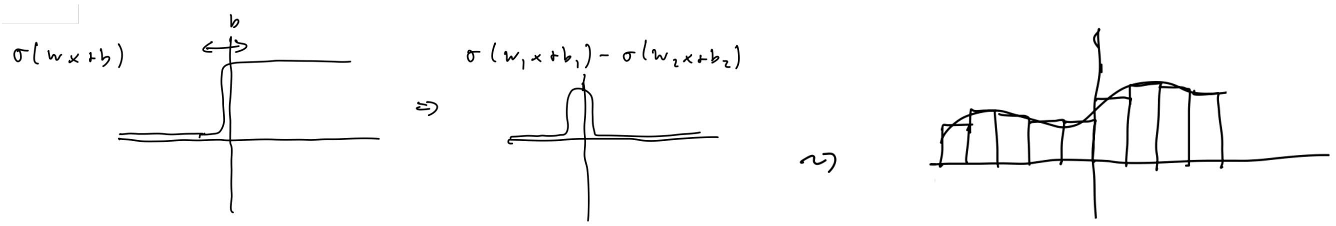
$$f(x) = \sum_i b_0 + w_i x^i\,, \qquad b_0 = f(x_0)\,, \qquad w_i = \frac{f^{(n)}(x_0)}{n!}$$

or with Fourier decomposition

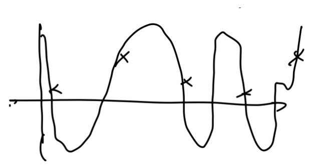
$$f(x) = \sum_i w_i^{(2)} \cos(w_i^{(1)} x + b_i)$$

Theorem (Universal Approximation Theorem): Can write any function that way (universally approximate), i.e., w/ a 1-layer NN w/ σ activation in the limit $n_{\text{hidden}} \to \infty$.

Proof sketch: The function $\sigma(wx + b)$ can create localized "bumps" when properly parameterized. These bumps can be combined to approximate arbitrary functions.

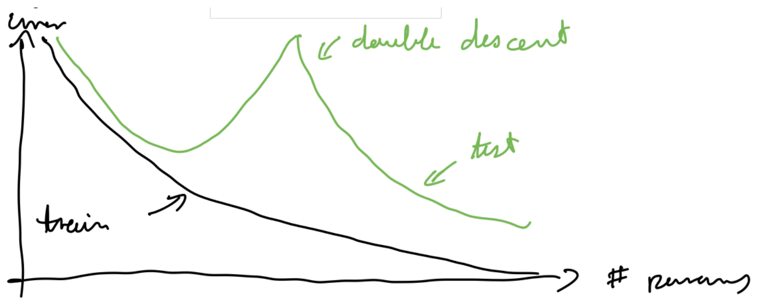


Note: This is not what a NN does in practice (constant splines).



Why this parameterization is a good idea nobody knows. Empirically, huge NNs generalize well and do not overfit. A polynomial of deg N can fit $N + 1$ pts, but it also has $N - 1$ extrema, so the function oscillates wildly outside training set.

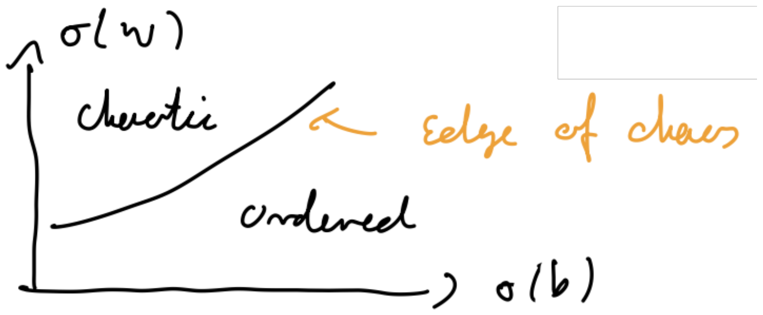
Bias-Variance Tradeoff



More ways on how to detect overfitting and prevent it in Section 2.

What do (outer) W, $\vec{b}$ do?

- $W \cdot x + \vec{b}$: normal operation
- $W \cdot x \gg \vec{b}$: Small changes in $x \Rightarrow$ huge changes in out, "Chaotic Phase"
- $W \cdot x \ll \vec{b}$: x is overridden by bias $\Rightarrow$ const function, "Ordered Phase" (important in training)



2 Training

Modern NNs have billions of parameters. Finding the "right ones" is a billion-dim optimization problem.

We will focus on training in supervised ML, where one has labeled data:

$$D = \{(\vec{x}^0, \vec{y}^0), (\vec{x}^1, \vec{y}^1), ...\}$$

How far are we from solution? Need a measure (loss function) for how far the NN $f_\theta(\vec{x}^A) = \hat{y}^A$ is away from the true value $\vec{y}^A$.

Example:

$$1)\text{ MAE:}\quad \mathcal{L} = \frac{1}{N}\sum_A |\vec{y}^A - \hat{y}^A| \tag{5}$$

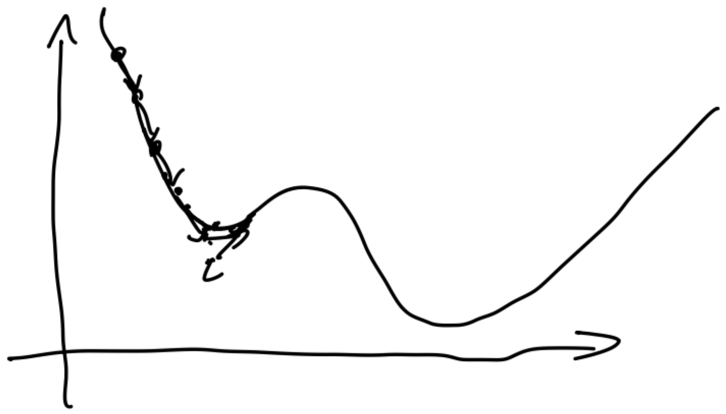
$$2)\text{ MSE:}\quad \mathcal{L} = \frac{1}{N}\sum_A (\vec{y}^A - \hat{y}^A)^2 \tag{6}$$

$$3)\text{ Binary Cross Entropy:}\quad \mathcal{L} = -\frac{1}{N}\sum_A [y^A \ln \hat{y}^A + (1-y^A)\ln(1-\hat{y}^A)] \tag{7}$$

These have the property that $\mathcal{L} \geq 0$ and $\mathcal{L} = 0 \Leftrightarrow y = \hat{y}$.

MAE tends to give sparser NNs (feature selection).

Training with GD and Backprop: To optimize a billion-dim parameter space, can only do simplest (1st order) method: Gradient Descent.



Forward pass:

$$\begin{aligned}
\vec{x}^{(0)} &= \vec{x}_{\text{in}} \\
\vec{x}^{(1)} &= W^1\vec{x}^{(0)} + \vec{b}^1 \\
\vec{\ell}^{(1)} &= \sigma(\vec{x}^{(1)}) \\
\vec{x}^{(2)} &= W^2\vec{\ell}^{(1)} + \vec{b}^2 \\
\hat{\vec{y}} &= \vec{x}^{(2)} \\
\mathcal{L} &= \frac{1}{2}(\vec{y} - \hat{\vec{y}})^2
\end{aligned}$$

Backward pass: Calculate difference at output and propagate error backwards.

$$\begin{aligned}
\frac{\partial \mathcal{L}}{\partial W^2} &= \underbrace{\frac{\partial \mathcal{L}}{\partial \vec{x}^{(2)}}}_{:=\vec{\delta}^{(2)}} \frac{\partial \vec{x}^{(2)}}{\partial W^2} = \underbrace{(\vec{y} - \hat{\vec{y}})}_{=\vec{\delta}^{(2)}} \vec{\ell}^{(1)} \\
\frac{\partial \mathcal{L}}{\partial \vec{b}^2} &= \frac{\partial \mathcal{L}}{\partial \vec{x}^{(2)}} \frac{\partial \vec{x}^{(2)}}{\partial \vec{b}^2} = \vec{\delta}^{(2)} \cdot 1 \\
\frac{\partial \mathcal{L}}{\partial W^1} &= \underbrace{\frac{\partial \mathcal{L}}{\partial \vec{x}^{(2)}}}_{=\vec{\delta}^{(2)}} \underbrace{\frac{\partial \vec{x}^{(2)}}{\partial \vec{\ell}^{(1)}}}_{=W^2} \underbrace{\frac{\partial \vec{\ell}^{(1)}}{\partial \vec{x}^{(1)}}}_{\sigma'(x^{(1)})} \underbrace{\frac{\partial \vec{x}^{(1)}}{\partial W^1}}_{\vec{x}^{(0)}} = \underbrace{(\vec{\delta}^{(2)} W^2) \cdot \sigma'(x^{(1)})}_{:=\vec{\delta}^{(1)}} \cdot \vec{x}^{(0)} \\
\frac{\partial \mathcal{L}}{\partial \vec{b}^1} &= \vec{\delta}^{(1)}
\end{aligned}$$

Then

$$W^{(i)} \mapsto W^{(i)} - \alpha \vec{\nabla}_{W^{(i)}} \mathcal{L} \,,$$

where $\alpha \approx 10^{-4}$ is the learning rate/step size (a so-called hyperparameter of the ML algorithm).

Note:

- $\vec{\delta}^{(n)} \sim \vec{\delta}^{(n+1)} W^{n+1} \cdot \sigma' \Rightarrow$ If W has small singular values and/or $\sigma' \ll 1$, gradients vanish and if W has large singular values and or $\sigma' >> 1$, gradients explode in deep NN ($n \gg 1$). Gradients will be large in chaotic phase and small in ordered phase $\Rightarrow$ Initialize at edge of chaos or use U-Nets/skip connections or layer norm.

- $\vec{\delta}$ contains σ and σ'. Thus if $\sigma = \sigma' = 0$, $\vec{\delta} = 0$. So $\sigma = 0$ means that the node is inactive, and since updates vanish, it will stay deactivated forever. Note that $\sigma =$ ReLU has $\sigma = \sigma' = 0$ for $x \leq 0$ and $\sigma =$ sigmoid has $\sigma \sim \sigma' \sim 0$ for $x \ll 0$. To combat this can use leaky ReLU (or elu, silu, gelu), and tanh instead of sigmoid. $\tanh(0) = 0$ but $\tanh'(0) = 1$, and $\tanh'(\pm\infty) = 0$ but $\tanh(\pm\infty) = \pm 1$

- Since GD is a first-order method, it gets stuck at (local) extrema. To prevent this, batch the training data into mini-batches (of size 8, 32, 64) and perform GD for this randomly selected sub-sample only. Chances are that the loss landscape has extrema at different loci for different batches. Thus even if the gradient vanishes for some batch, it won't for the next. Processing all batches is called an epoch. Typically, one trains for $N \approx 100$ epochs.
- Other possibilities are to use momentum, etc. Nowadays, ADAM plus batching is typically used.
- Typically one does not want to find a global min, since the model is likely overfitting. So one way to prevent this is **early stopping**. Other methods include:
  - **Dropout:** During training, randomly deactivate some nodes in a layer. Since the NN does not know which ones it will lose, it cannot fine-tune weights against one another.
  - **Weight regularization:** Add a term $\lambda \sum_i |\theta_i|$ or $\lambda \sum_i \theta_i^2$ known as L1 (or lasso) or L2 weight regularization, and $\lambda \in \mathbb{R}$ is a hyperparameter. This means the NN wants all parameters ~ 0. Remember that σ, ReLU, tanh are (approx) linear around 0. But linear act. functions mean that we're composing affine maps, which lead to an affine map, so a simple function, so no overfitting. Also, small weights means that the output is small (non-chaotic), so no wild oscillations.

  To detect overfitting, one should split the dataset D into a train set and a test set (and maybe also a validation set). Typical splits are 90:10. One then trains on the train set and uses the test set to evaluate the performance of the model on unseen data. The validation set can be used to detect overfitting for hyperparameter tuning.

How can I improve the accuracy of my NN?

Usually there are 3 axes to optimize on:

(i) More training time

(ii) More parameters

(iii) More training data

Empirical observations show that when changing one while holding the other 2 fixed, the error falls with a power law:

$$E \sim E_0 + N_i^{-\alpha_i}$$

α_i is architecture dependent (want α_i as large as possible). E_0 is an irreducible error and conjecturally architecture independent. This behavior is called Neural Scaling Laws.

Other training setups

The loss function need not be computed from labels. One can use the techniques described above to extremize any objective function. For example, in PDEs, one lets the NN be the function that solves the PDE. One then takes derivatives of the NN (with respect to its input) and for training,with respect to its parameters:

$$\ddot{y} = -\omega^2 y \quad \rightarrow \quad \text{let } y = f_\theta(t) \text{ and } \mathcal{L} = (\ddot{y} + \omega^2 y)^2$$

To impose initial or boundary conditions, add a term, say:

$$\mathcal{L} = (\ddot{y} + \omega^2 y)^2 + (y(0) - y_0)^2 + (y(1) - y_1)^2$$

This is much easier than a shooting method.

3 NN Architectures

Over the years, the most important NN architectures were: MLP, CNN, RNN/LSTM, Graph NN, Transformers. We already discussed MLPs. CNNs were developed for vision tasks. RNNs/LSTMs were developed for Natural Language Processing (NLP) or time series. Graph NNs were developed for graphs/point clouds and Transformers again for NLP. We won't discuss RNNs because they have been replaced by transformers, and we will skip Graph NNs for time reasons.

Equivariant NNs: Often, input data is invariant under some group operation. We then want the output to be invariant or equivariant as well. Let G be a group and R a representation under which the input transforms. A function is equivariant if:

$$f(R \circ \vec{x}) = g \circ f(\vec{x})$$

If $g = \mathrm{id}_Y$, then f is invariant.

Ex:

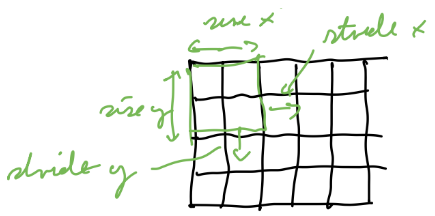
$$\Psi_D(x) \mapsto \underbrace{\Lambda_{1/2}}_{\text{spinor irrep}} \Psi_D(\underbrace{\Lambda^{-1}}_{\text{vector irrep}} x)$$

This problem is ubiquitous in math and physics. To obtain equivariant or invariant NNs, we can either:

- build the symmetry into the architecture
- data-augment by adding multiple symmetry-transformed input-output pairs

In real life, images are often translation and rotation invariant. Graphs are invariant under relabeling nodes, etc.

CNNs: To classify images into cats or avocados, we need to detect the object or defining features of the object (like whiskers) somewhere in the image (translation inv). This is what CNNs do. They pass filter/kernels of size $N \times M$ over the image and convolve the image pixel values w/ the trainable weights in the filter:

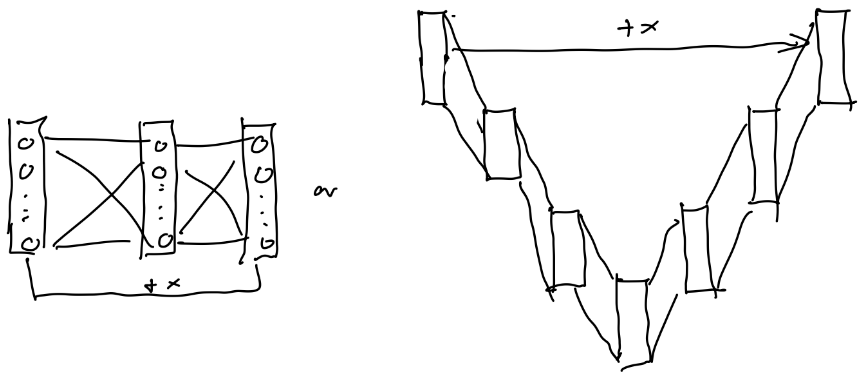


The convolution operation is:

$\sum_{i,j} \underbrace{p_{a+i,b+j}}_{\text{pixel value}} \cdot \underbrace{w_{i,j}}_{\text{kernel weights}} = y_{a,b}$

Typically one uses multiple filters (64), and they learn border detection of salient features (like whiskers). Often, Conv layers are followed up by max-pooling layers (2×2 kernel which just selects the max value). These are stacked a few times, fed into a fully connected layer, and then into the output layer.

The filter should be of the typical size of the salient feature it should detect. Since this is not known a priori, inception networks use filters of different sizes that are combined with fully-connected layers. If one stacks too many, the NN becomes too deep and one needs skip connections with U-Nets:



This is of course not exclusive to CNNs but can be used in other architectures as well.

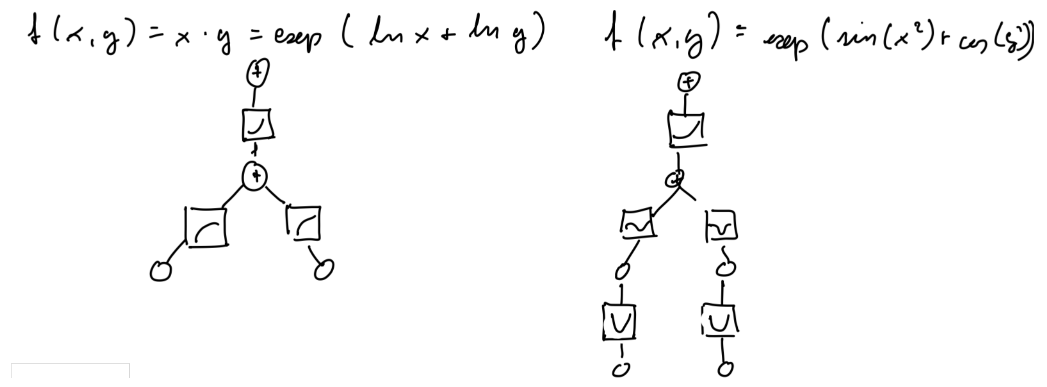
Note that weight sharing (i.e., conv) is also necessary for practical reasons: A 16 Megapixel image has $3 \cdot 16 \cdot 10^6$ inputs, so if this is put into an MLP with 100 hidden nodes, already the first layer has close to 1 billion parameters.

KANs:

I want to briefly introduce KANs, which are geared toward interpretability and scientific discovery.

The idea is to put learnable functions (cubic splines) on the edges of the computational graph and just sum over incoming edges at each node. The Kolmogorov-Arnold representation theorem states that any function can be written as a sum of unary functions.

Examples:

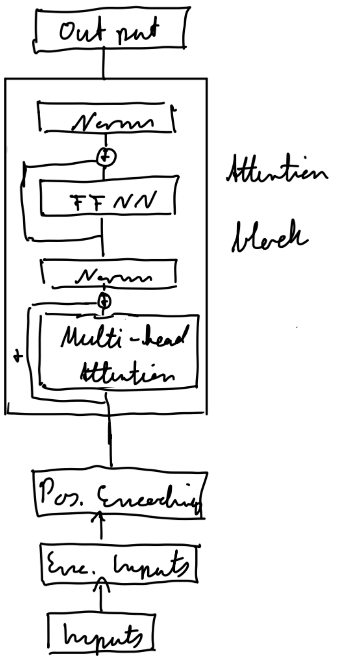

$f(x,y) = x \cdot y = \exp(\ln x + \ln y)$
$f(x,y) = \exp(\sin(x^2) + \cos(y^2))$
+
+
+
+

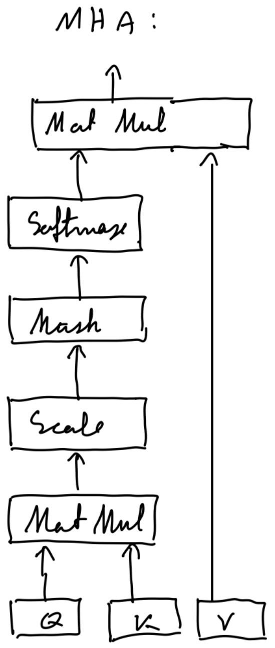
Advantages:

- Approximation theory of splines + KA says that α in the neural scaling laws only depends on the spline order, whereas for NN it depends on the "intrinsic dimensionality of the input data manifold". So for a function $\mathbb{R}^{100} \to \mathbb{R}$, $\alpha \sim \frac{1}{100}$ for NNs and $\alpha \sim \frac{1}{4}$ for cubic splines.
- Splines have compact support $\to$ combat catastrophic forgetting.
- In NNs, neurons are multi-purpose and parameters non-local, which makes them difficult to interpret. In KANs, one can look at the splines and just write down a symbolic expression.

4 Generative Models

Transformers: Transformers were initially developed for NLP. They introduce a new layer called the attention layer. The rest are just FC layers as in an MLP. Schematically, the architecture is





Encoded Inputs: To feed words into a NN, we need to map them into $\mathbb{R}^N$ first. We do this in a 2-step procedure:

(i) One-hot encode the OED:

$$\text{-a} \mapsto 10...0 \tag{8}$$

$$\text{-Aardvark} \mapsto 010...0 \tag{9}$$

$$\text{-Zygzygna} \mapsto 00...1 \tag{10}$$

(ii) Multiply w/ a $512 \times 300\text{k}$ matrix to get an embedding into $\mathbb{R}^{512}$

Positional encoding: Since most languages are not permutation invariant ("The scientist eats the chicken" $\neq$ "The chicken eats the scientist"), we add a positional encoding, i.e., some deterministic sequence of numbers that the NN can easily pick up on and that tells the NN where in the sentence the word was. The original paper uses alternating $\sin(\frac{2i \cdot t}{1000})$ and $\cos(\frac{2i \cdot t}{1000})$.

Attention block: The embedded + pos-encoded vector $\vec{w}$ is fed into the attention head. Each head copies it 3 times and multiplied by three different weight matrices called Query (Q), Key (K), Vector (V) which contain the trainable params and are 64×512 mat.

$$\vec{q_i} = Q \cdot \vec{w}_i \qquad \vec{k}_i = K \cdot \vec{w}_i \qquad \vec{v}_i = V \cdot \vec{w}_i$$

The naming is taken from databases. There, a key is used to retrieve the result of a query by comparing all keys in the database w/ the object in the query. In databases, this is done via hash functions.

In transformers, it is done via cosine similarity (aka) scalar product: $r_{ij} = \vec{q_i} \cdot \vec{k}_j$. Optionally, at this point, one can mask out some $\vec{k}_j$. Also, the authors observe that the algorithm is numerically more stable if we scale $r_{ij} \to \frac{r_{ij}}{\sqrt{64}}$ where 64 is the length of $\vec{q}$ or $\vec{k}$.

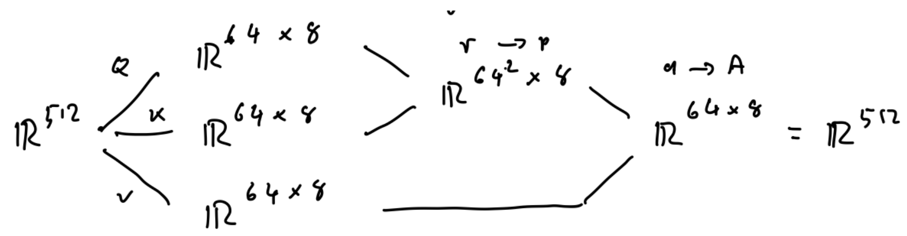
Next, the author convert the "retrieved rows j for query i", r_{ij}, into a probability with a softmax:

$$P_{ij} = \text{softmax}(r_{ij})$$

Then, finally, these probabilities are multiplied with the value vector, which assigns a value to each position:

$$\vec{a}_i = \sum_j P_{ij} \vec{v}_j$$

In this way, each 512D input word $\vec{w}_i$ is mapped to the 64D attention vector $\vec{a}_i$. The paper uses 8 attention heads (just like CNNs use multiple filters), and it then concatenates the eight 64D attention vectors $\vec{a}_i$ into one 512D vector $\vec{A}_i$. In that way, the MHA maps from $\mathbb{R}^{512} \to \mathbb{R}^{512}$:



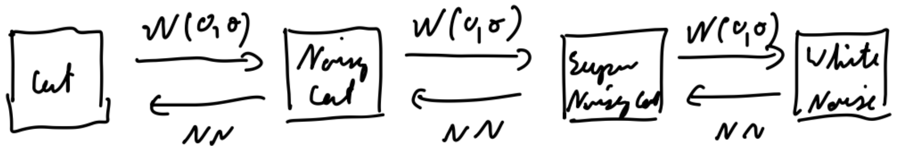
$\mathbb{R}^{512}$
Q
K
V
$\mathbb{R}^{64\times 8}$
$\mathbb{R}^{64\times 8}$
$\mathbb{R}^{64\times 8}$
$r \to P$
$\mathbb{R}^{64^2 \times 8}$
$a \to A$
$\mathbb{R}^{64\times 8}$
$= \mathbb{R}^{512}$

Skip connection + Normalization: The original input $\vec{w}$ also skips the attention blocks and is added to the attention vector $\vec{A}_i$. Then a layer norm is applied again for numerical stability.

FFNN + Skip connection + Normalization: The result is then shoved through a FFNN, which is also bypassed with a skip connection. The result is normalized and presents the output of the NN.

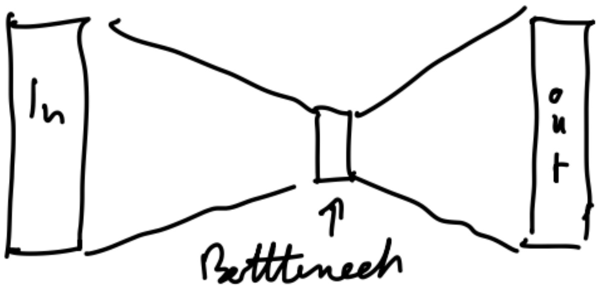
Auto-regressive generation: Models like Chat GPT generate sentences auto-regressively. They consist of an encoder-decoder pair. What we just described is the encoder architecture. The decoder is almost the same. It is fed its own output as its input, plus the output of the encoder. The decoder output is querying the encoder outputs (which mimic the DB, i.e., the key and value). The final layer of the decoder is a 512×300k layer with softmax activation. In this way, we get a (one-hot encoded) probability vector pointing to the next word in the OED. This is then appended to the output and the process is repeated.

Diffusion Models: These are mostly used for image generation nowadays. The idea is that one starts with a bunch of cat images and successively makes them more and more noisy (pixel diffusion). One then trains NNs to undo the noising steps. Then, one can simply feed white (Gaussian) noise to the model and it will successively undo the diffusion until it becomes a cat. Since the starting point was random, the resulting image is random (and not real), but since all the model knows are cat images, the resulting image will be a cat image.



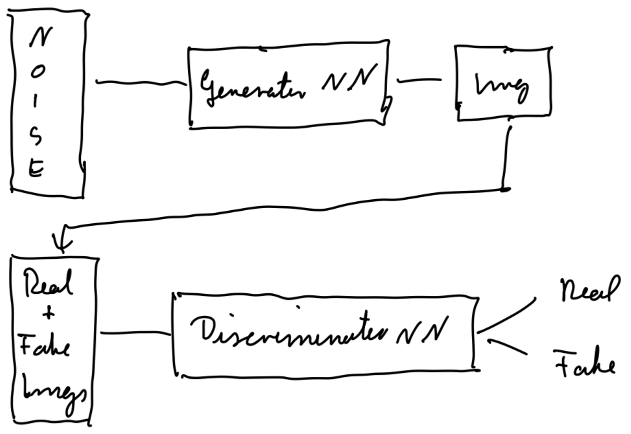
One can inject some signal into the noise (e.g., the output of a transformer that processed a human input) to steer whether the resulting image will be a cat or an avocado.

VAEs: Variational Auto-Encoders learn the identity function, but do so by going through a bottleneck layer of small dimension. The encoder learns to compress (auto-encode) the input to a low-dim representative, the decoder learns to reproduce the original input from this compressed representation.



By adding random noise (and using dirty tricks to backprop through this), one can then move around on the submanifold in embedding space where all cat images lie and get a new cat image.

GANs: In Generative Adversarial Networks, two NNs are competing. A generator is trying to generate images from noise, and a discriminator NN is trained as a binary classifier that classifies images into real versus generated. So the adversarial generator is trained to fool the discriminator. The losses of the two NNs are inverses of each other. In the end, one will say the generator will have learned to generate images from white noise that cannot be distinguished from real images anymore.



5 Reinforcement Learning

RL is based on behavioral psychology where an agent explores an environment and receives pos/neg rewards for its actions. To talk about RL, we need to introduce some vocabulary. We illustrate it using chess along the way.

Agent: The player(s) of the game (black and white).

State space: The set of all states in the game (all $\sim 10^{120}$ inequivalent chess boards). Can also be infinite, but often discrete and huge. We denote state space by S and (a sequence of) states by $s_t \in S$.

Action space: The set of all actions that transition from one state to the next (all legal chess moves). Can be continuous or discrete, but often discrete and small-ish. We denote action space by

$$\begin{aligned} A:&\quad S \to S \\ a:&\quad s_t \mapsto s_{t+1} \end{aligned}$$

Often, the set of legal actions depends on the state one is in. Also, actions can be stochastic (e.g., a drone is steered to the left, but a gust of wind pushes it nevertheless to the right). This does usually not occur in games (like chess) or in pure math.

Terminal states: The set of states $T \subset S$ on which the game ends, i.e., where no legal actions are possible (check-mate, stale-mate, draw).

Reward: The reward R is defined as:

$$\begin{aligned} R: \quad & S \times A && \to \mathbb{R} \\ & (s_t, a(s_t)) && \mapsto \mathbb{R}\,. \end{aligned}$$

It is a function into the real numbers that reinforces the behavior of the agent. Negative rewards correspond to punishments (e.g., (-1,0,+1) for loss, draw, win).

Policy: The function that chooses the next action, given a state:

$$\begin{aligned} \pi: \quad & S \to A \\ & s_t \to a_t \end{aligned}$$

This will be outsourced to a NN.

Return: The (expected) cumulative reward given the current policy π:

$$G_{s_0,\pi,\gamma} = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} R(s_t, a_\pi(s_t))$$

Here $\gamma \in (0, 1]$ is called the discount factor. It's a percentage by how much future rewards and punishments are reduced (discounted). (This is what we will extremize via SGD).

From a math perspective, RL solves a Markov Decision process (MDP): The agent decides on the next action, and its decision only depends on the current state s_t, not the history of how it got there (although RL might also work for non-Markov processes, in which case we can define a state to be the set of all previous states $\tilde{S} = S^* = S \times S \times ...$, so $\tilde{s} = (s_0, s_1, ..., s_t)$).

Note:

- Stable (multi-agent) RL policy optimization ends in a Nash equilibrium.
- RL can play many types of games:

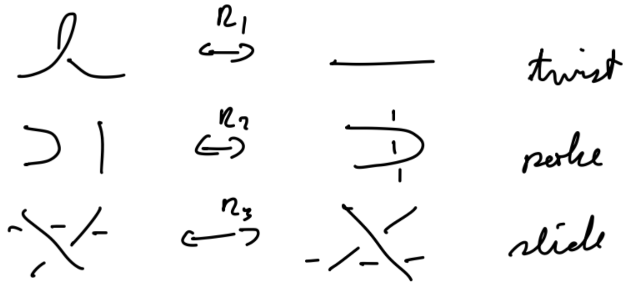
  - perfect/imperfect information
  - balanced/imbalanced
  - single/multi-agent
  - co-operative/competitive

- Rewards are often sparse (e.g. in chess one needs to finish an entire game before a single reward (0, ± 1) is awarded). This is ok for a chess game which has O(50-100) moves. In "needle-in-the-haystack" type of search problems, however, you could play forever w/o reaching a terminal state. In such cases it is important to assign intermediate rewards (if possible), and/or apply curriculum learning where one starts w/ a terminal state and moves 1 action away from it, then 2, 3, ...

- When choosing the next action, one has to ask whether to just use the best currently known one or whether to keep exploring the state space. Typically initially it is better to explore and then transition towards exploiting learned knowledge. Ways to do this are to not greedily choose the best action next (which would be pure exploitation) but to sample actions stochastically from a prob. distribution, e.g., ϵ-greedy, where we choose the best next action with prop. $1-\epsilon$ and a random action w/ probability ϵ. We could then reduce ϵ as training progresses. Another possibility is to assign a prob. to each action and sample from this distribution w/ some temp T which is reduced as training progresses.

Example: Unknotting a knot

A knot is an embedded $S^1 \hookrightarrow S^3$. Two knots are equivalent if their knot projections are related by Reidemeister moves:



The goal of the game is to find a sequence of R_i that untangles a knot (if possible). This is a single player, perfect information, "needle in the haystack" kind of game.

States: $S = \{\text{All knot diagrams}\}$, $|S| = \infty$

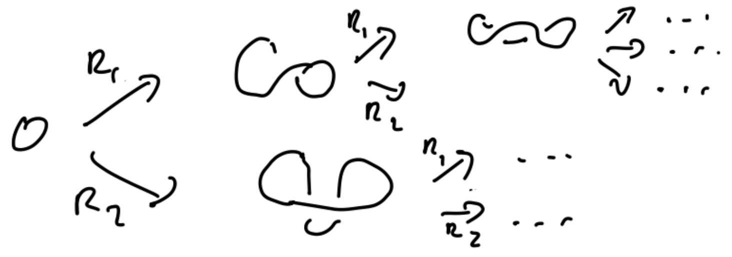
Actions: $A = \{R_1(x_j), R_2(x_j), R_3(x_j) \mid x_j \text{ a crossing in K}\}$. Note: R_i can be impossible/illegal in some states s_t.

Terminal states: $T = \{\bigcirc\}$

Reward: Only punish, $R(s) = -\#$ crossings in K.
Note, R_1 and R_2 can increase the # of crossings by 1 or 2. Sometimes, it is necessary to increase them before we can simplify further. This is why it is important to extremize G and not R. Now, since we only punish by the number of crossings, the agent will minimize its losses (i.e., max. its return) by getting to a 0-crossing state as quickly as possible. We can safely set $\gamma = 1$ in this case. If we chose a sparse reward:

$$R = \begin{cases} 0 & \text{if } K \neq \bigcirc \\ 1 & \text{if } K = \bigcirc \end{cases} \tag{11}$$

and set $\gamma = 1$, the agent would have no incentive to finish the game as quickly as possible. We could punish each step α: R = -1 if $K \neq \bigcirc$ and R = +1 if $K = \bigcirc$ (which is essentially what we did, except that the punishment was proportional to the distance from the solution), or we could just reward and use $\gamma < 1$ to ensure that the agent tries to finish the game as fast as possible. But then we still have the sparse reward problem. We can do curriculum learning by building an "inverse search tree":



RL and deep learning:

In practice, one typically has a policy network $\pi_{NN} : S \to A$ that is given a state (represented as a vector in $\mathbb{R}^m$, as usual) and $n_{\text{out}} = |A|$ with a softmax as the last layer, i.e., the output is a probability vector over all actions (illegal actions can be masked out later, or the NN can be punished for proposing illegal actions, or both).

In order to train the policy, we need to compute (or estimate) the (expected) return. For this, we define the state value function

$$v(s) = \mathbb{E}_\pi(G_t | s_t = s)\,.$$

We could obtain this by just calculating $G_{s,\pi}$ for all possible outcomes and then averaging (called MC),[1] but there are typically too many possibilities for this to be feasible. Another possibility is to truncate the computation of G after T steps:

$$\tilde{G}_t(s_t) = \sum_{k=t_0}^{t+T} \gamma^{k-t_0} R(s_k)$$

and use $\tilde{G}$ as an estimate for G (which converges better for small γ), or to just define a state-value function network V_{NN} that estimates the state value function. Or one could combine the two steps and have it estimate

$$G_t - \tilde{G}_t = \sum_{k=t_0+T}^{\infty} \gamma^{k-t_0} R(s_k)\,.$$

Also note that given a value function, we could use it as a policy: given a state s_t, carry out all possible actions and take the one with the largest $v(s_{t+1})$ (or sample with weights $\sim v(s_{t+1})$). Similarly, some RL algorithms use an action-value function

$$q(s,a) = \mathbb{E}(G_t | s = s_t, a_t = a)\,,$$

which can lead to more stable alg. Given $v(s)$ or $q(s,a)$, we can use it to train the policy NN π_{NN} by gradient ascending on $v(s)$: $\theta_{\pi_{NN}} \to \theta_{\pi_{NN}} + \alpha \nabla v$

Note that now π changed and hence the expected return, since this depends on the policy π. So now we need to retrain $v(s)$ (e.g., by truncating G to $\tilde{G}$) and then optimize π with this new value function, etc. We are thus interleaving the control problem (optimizing v) and the decision problem (optimizing π).

There are many RL algorithms:

A^{2}C (Advantage actor-critic): The actor is the policy NN π_{NN}, the critic is the value NN v_{NN}. They use the advantage

$$A(s_t, a_t) = r_t + [\gamma v(s_{t+1}) - V(s_t)]$$

to estimate the full return where:

- r_t is the reward of action a_t
- $\gamma V(s_{t+1}) - V(s_t)$ is the reward as calculated from the value function

[1]This is for stochastic policies. For greedy policies, the outcome is fixed (deterministic) and there is only one return.

If $A > 0$: Action better than expected $\to$ encourage
If $A < 0$: Action worse $\to$ discourage

The critic is trained to minimize the MSE of the advantage

$$\mathcal{L}_{\text{critic}} = (\underbrace{r_t}_{\text{target}} + \underbrace{[\gamma V(s_{t+1}) - V(s_t)]}_{\text{Estimate}})^2$$

The actor is trained to maximize the expected advantage

$$\mathcal{L}_{\text{actor}} = -\ln \pi(a_t|s_t) \cdot A(s_t, a_t)$$

A^{3}C: Asynchronous version of A^{2}C where multiple agents explore the environment and update the same NN. Better for "needle-in-a-haystack" searches, but can be less stable than A^{2}C.

Deep Q-learning: Classical Q-learning just maintains a table for the action-value function, where it estimates the return similarly to A^{2}C:

$$Q(s_t, a_t) \to Q(s_t, a_t) + \underbrace{\alpha}_{\text{LR}} \left(r_t + \gamma \underbrace{\max_a Q(s_{t+1}, a)}_{\text{maximize over all legal actions}} - Q(s_t, a_t) \right)$$

In DQL, we replace this table by a NN w/ loss:

$$\mathcal{L}_{\text{DQN}} = \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]^2$$

The policy is then defined (implicitly) by choosing $\max_a Q(s, a)$ (ϵ)-greedily.

Tricks to make this more stable:

- Copy the Q-network, freeze its weights for a while and use this to calculate $\max_a Q(s_{t+1}, a)$
- Experience replay buffer: store (s_t, a_t, s_{t+1}, r) in a buffer and train the NN to predict Q for this tuple (w/o actually playing the game at this point)

PPO (Proximal Policy Optimization): Like A^{2}C, but clip gradients (updates) to policy NN to keep training more stable.

TRPO (Trust Region Policy Optimization): Like PPO, but instead of clipping the policy gradient updates, one computes a trust region (along which the policy does not change too much).

This is done by optimizing a surrogate loss (KL divergence of the old and new policy):

$$\max_\theta \mathbb{E}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \cdot A(s,a)\right]$$

subject to

$$\mathbb{E}[D_{KL}(\pi_{\theta_{old}}(\cdot \mid s) \;\; \| \;\; \pi_\theta(\cdot \mid s))] \leq \delta\,,$$

where δ is the trust region size. To do that, the argument of $\max_\theta$ is approximated by a second-order Taylor series. One then uses conjugate gradients to find good update directions and performs a line search to ensure that the constraint is $\leq \delta$.

SAC (Soft Actor-Critic): Like DQN, but it adds an entropy term to the return, i.e., it tries to maximize

$$G + \alpha\,\mathcal{H}(\pi(\cdot \mid s_t))\,,$$

with α the temperature and $\mathcal{H}$ the entropy of the policy at state s_t.

MCTS: Build a search tree, expand only nodes with the highest PUCT score:

$$\text{PUCT} = \underbrace{q(s_t,a_t)}_{\text{action value}} + \underbrace{c_{\text{PUCT}}}_{\text{const}} \times \underbrace{\pi(s_t,a_t)}_{\substack{\text{prob of choosing } a_t \\ \text{(from policy NN)}}} \times \frac{\sqrt{\sum_b N(s_t,b)}}{1+N(s_t,a_t)}\,,$$

where $N(s_t,a_t)$ is the number of times action a_t was chosen in state s_t.

It is usually best to leave the RL algorithm implementation to the pros (stable baselines 3), and just implement the environment. RL uses the Gym environment, which exposes the state space, action space, a step function which carries out an action and returns the reward + new state (among other things), and a reset function to reset the environment after reaching a terminal state.