Neural networks and Deep learning

1 Theory

1.1 Introduction

1.1.1 Introduction to Machine Learning

1.1.1.1 Supervised learning

- Where an algorithm learns from **labeled data**.
- More formally:
  - Given a training set $\mathcal{X} = \{(\mathbf{x}_i, y_i)\}_{i \in |N|}$.
  - Learn a function $g(\mathbf{x}; \theta)$ that approximates the *true* function $f(\mathbf{x})$ that generated the data.
    - $g(\circ)$ is the model, θ are its parameters.
- g is assumed to **generalize** to unseen data.

1.1.1.2 Tasks

- Tasks in ML specify the nature of the problem to be solved.
  - Classification, regression, clustering, association rules extraction, etc.
- **Classification**:
  - A **supervised learning** task.
  - *Goal*: build a model to **classify** instances of an underlying concept using given **labels**.
  - eg Spam detection, sentiment analysis, object detection, LLM, etc.
- **Regression**:
  - $y \in \mathbb{R}$ wile in classification y is a categorical value.
  - eg stock price prediction, weather forecasting, image generation, etc.

1.1.1.3 Induction

Induction: inference of a generalized conclusion from particular instances.

- ML algorithms **optimizes** the parameters, θ, in such a way a **loss function** L is minimized.
  - $\theta^\star = \arg\min_\theta L(g(\mathbf{X}; \theta), \mathbf{y})$.
  - This assumes that the **induction** process hold in the applicative settings and generalizes to **unseen data**.
- **Induction** is precisely what every algorithms aims to achieve when learning a classification function.
- **No Free Lunch Theorem**: «all algorithms are equivalent, on average, by any of the following measures of risk».
  - Or, «all models are wrong, but some models are useful».
  - A learning algorithms is only as good as its **inductive bias**.
    - How well the inductive bias of the algorithms fits the problem at hand must be considered.
- eg Examples of inductive bias:
  - A **linear classifier** assumes that:
    - Examples lie in a metric space.
    - The underlying concept is linear in nature.
  - A **convolutional neural network** assumes that:
    - The underlying concept is translation invariant.
    - The input is an image (a 2D grid of pixels where adjacent pixels are correlated).
  - A **recurrent neural network** assumes that:
    - The underlying concept is sequential in nature.

1.1.1.4 Training a neural network

- NN are a incredible flexible family of models that can be used to approximate any function.
  - This flexibility comes at a cost: they depend on a vast number of **hyperparameters**.
    - Like the number of layers, neurons in each layer, learning rate, optimizer, etc.
    - These hyperparameters need to be set correctly in order to achieve good performance.
  - It is then important to understand how to **set these hyperparameters correctly**.
    - And how to validate the quality of the inferred model **without overestimating its performance**.
- Problem statement:
  - Define the correct procedure to:
    - Find the best possible way to **set the hyperparameters**.
    - Get an **accurate estimation of the generalization error** at the end of the process.

1.1.1.5 Errors

Generalization error

- `def` **Generalization errors**: $R = \mathbb{E}_{(\mathbf{x},y)\sim p^\star}[L(y, g(\mathbf{x}, \theta))]$.
- The error that the acquired classifier g commit, on average, on examples drawn from the same distribution used for sampling the examples in the training set.
- $p^\star$ is the true distribution of the data.
- L is a loss function used to measure how bad the error is when y is predicted as $g(\mathbf{x}, \theta)$.
- Problems:
  - No actual access to $p^\star$.
  - Even with access to $p^\star$, it's not possible to compute an average on an infinite number of samples.

Loss functions

- `def` $0-1$ **loss**: $L(y, y') = \mathcal{J}_{y\neq y'} = 1$ if $y \neq y'$, 0 otherwise.
- `def` **Quadratic loss**: $L(y, y') = (y - y')^2$.
- But many others are possible (`eg` hinge, exponential, logistic, etc).

Empirical error

- `def` **Empirical error**: $\hat{R}_T = \frac{1}{|T|}\sum_{(\mathbf{x},y)\in T} L(y, g(\mathbf{x}; \theta))$.
- It should be evident that (in general) R cannot be computed.
  - To overcome this problem in most cases in R is approximated with the empirical error.
- T is a finite sample drawn from $p^\star$.

Training and test errors

- **Training error** ($\hat{R}_{Tr}$): when T is the **training set** ($T = Tr$).
  - Since it's implicitly (sometimes explicitly) optimized by the learning algorithm, it has an **optimistic bias**.
- **Test error** ($\hat{R}_{Te}$): when T is the **test set** ($T = Te$).
  - It serves as an unbiased estimator of the generalization error R.
  - It may exhibit a **pessimistic bias** because retraining the model on the entire dataset is likely to produce a model with a lower error.

1.1.1.6 Overfitting

- When $\hat{R}_{Te} - \hat{R}_{Tr} > 0$, the algorithm is said to be **overfitting**.
- Overfitting is common problem for learning algorithms.
  - And it is usually necessary to counter it by some method.
- But minimizing the generalization error on the test set is a sort of tuning.
  - This makes the test set error an **optimistic estimator** of the generalization error.
  - And therefore **overfitting the test set**.
- To overcome this problem, an extra set, the **validation set**, is needed.
  - Used to solely evaluate model performance under a given choice of hyperparameters.
  - Once the hyperparameters are set, Te is used to asses the final model's quality.
  - When data is scarce, choices have been made and the final quality is fine, retrain the classifier on $Tr + Va$.

    - When the test set is no longer needed, it can be merged to the training set.
- Selecting best hyperparameters for a model (?):
    - Both validation and test sets are **finite samples**, measuring the error on them will introduce some noise in the estimation of the generalization error.
        - $\hat{R}_{Te} = R + \epsilon_{Te}$ and $\hat{R}_{Val} = R + \epsilon_{Val}$.
    - The model is chose according to the hyperparameters that minimize the error on the validation set.
    - Determine if at the end of the process $\hat{R}_{Te}$ and $\hat{R}_{Val}$ are **unbiased estimators of the GE**.
        - Which is, $\mathbb{E}[\hat{R}_{Te}] = \mathbb{E}[\hat{R}_{Val}] = R$.
        - The distribution of Test errors is uniform and has mean μ_0.
        - The distribution of Validation errors is skewed to the left, resulting in an under-estimation.
- Training, validation and test sets:
    - **Training set**: used during learning.
    - **Validation set**: used to assess the quality of the current choice of hyperparameters.
    - **Test set**: used to asses the quality of the final classifier.

1.2 Mathematical foundations

1.2.1 Matrices

- In ANN, vectors and matrices are everywhere.
    - Inputs are vectors, weights are matrices.
    - Vectors and matrices must be multiplied to calculate network output.
- A **scalar** is just a single number x.
- A **vector** is an array of numbers.
- A **matrix** is a $2D$ array of numbers.
    - Indices: i for the row and j for the column.
- A **tensor** is an array with more than two dimensions.
    - Indices: i for the batch (slice), j for the row and k for the column.
- The **transpose** $\mathbf{A}^T$ of a matrix $\mathbf{A}$ is a mirror image, $(\mathbf{A}^T)_{i,j} = \mathbf{A}_{j,i}$.
- If matrices have the same shape, they can be **added**, $C_{i,j} = A_{i,j} + B_{i,j}$.
    - A scalar can be added or multiplied to, $D_{i,j} = a \cdot B_{i,j} + c$.
- **Matrix multiplication**: if A has shape $m \times n$ and has shape B $n \times p$, $\mathbf{A} \cdot \mathbf{B}$.
    - $\mathbf{A} \cdot \mathbf{B} \neq \mathbf{B} \cdot \mathbf{A}$.
- **Vector multiplication**: $\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T\mathbf{y}$.

1.2.1.1 Geometric interpretation

- Vector can be represented geometrically.
    - $\mathbf{x} = [x_1, x_2]$ will be a vector originating at $(0, 0)$ and arriving at (x_1, x_2).
- `def` **Euclidean norm** (or L^2 norm): $\|\mathbf{x}\| = \sqrt{x_1^2 + x_2^2} = \sqrt{x \cdot x}$.
    - The euclidean distance from the origin to the point identified by $\mathbf{x}$.
    - `def` **Euclidean norm**: $\|\mathbf{x}\| = \sqrt{\sum_i x_i^2} = \sqrt{x \cdot x}$.
    - The squared euclidean norm is also used.
- `def` L^1 **norm**: $\|\mathbf{x}\|_1 = \sum_i |x_i|$.
- `def` **Dot product**: $\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\| \cos(\theta)$.

1.2.1.2 Linear transformation

- Matrices are a tool to **linearly transform** vectors (and vector space).
- **Determinant** of a squared matrix, $det(\mathbf{A})$, is a function mapping matrices to real scalar.
    - $det(\mathbf{A})$ gives a measure of how multiplication by the matrix expands or contract space.
        - If $|\det(\mathbf{A})| = 1$ the volume remains unchanged.
        - If $\det(\mathbf{A}) = 0$ the space is contracted along at least one dimension.
    - For $\mathbf{A}$ with dimension 2, $det(A) = a_1 \cdot a_4 - a_2 \cdot a_3$.

1.2.1.3 Special matrices

- **Identity matrix**: it does not change any vector when multiplied the matrix by the vector.
- **Inverse matrix** (A^{-1}): the matrix such as $A^{-1}A = I_n$.
    - Matrices for which A^{-1} exists are **invertible** (only squared matrices).

    - `prop` $\mathbf{A}$ is invertible iff $det(\mathbf{A}) \neq 0$.
    - When $\mathbf{A}^{-1}$ exists, there are several algorithms for finding it.
- **Symmetric matrix**: any matrix equal to its transpose ($\mathbf{A} = \mathbf{A}^T$).
- **Unit vector**: a vector with unit norm ($\|\mathbf{x}\|_2 = 1$).
- **Orthogonal vectors**: $\mathbf{x}$ and $\mathbf{y}$ to each other if $\mathbf{x}^T\mathbf{y} = 0$.
    - Vectors orthogonal to each other and with unit norm are **orthonormal**.
- **Orthogonal matrix**: a square matrix whose rows are mutually orthonormal and whose columns are mutually orthonormal.
    - `prop` For orthogonal matrices $\mathbf{A}^T\mathbf{A} = \mathbf{A}\mathbf{A}^T = \mathbf{I}$.
        - Hence $\mathbf{A}^{-1} = \mathbf{A}^T$.

1.2.2 Calculus

1.2.2.1 Derivatives

- The slope of the tangent line to f at point x (denoted as $f'(x)$ or $\frac{df}{dx}(x)$).
- It specifies how to scale a small change in input in order to obtain the corresponding change in output.
    - $f(x + \epsilon) \approx f(x) + \epsilon f'(x)$.
- The secant line becomes the tangent line when $\epsilon \to 0, f'(x) = \lim_{h\to 0} \frac{f(x+\epsilon)-f(x)}{\epsilon}$.
- `prop` Properties of derivatives:
    - Linearity: $(\alpha f(x) + \beta g(x))' \equiv \alpha f'(x) + \beta g'(x)$.
    - **Chain rule**: $(f(g(x)))' \equiv f'(g(x))g'(x)$.
    - Product rule: $(g(x)h(x))' \equiv g'(x)h(x) + g(x)h'(x)$.
    - Quotient rule: $(\frac{f(x)}{g(x)})' \equiv \frac{f(x)'g(x)-f(x)g'(x)}{(g(x))^2}$.
    - Power rule: $(x^r)' \equiv rx^{r-1}$.

1.2.2.2 Integrals

- The integral value of f between a and b is the area under f in the given region.
    - When the function is below 0, the area contributes negatively.
- `def` **Fundamental Theorem of Calculus**:
    - If f admits an antiderivative F (if it exists F such that $F'(x) = f(x)$) then:
        - $\int f(x)\,dx = F(x) + C$.
        - $\int_a^b f(x)\,dx = F(x)|_a^b = F(b) - F(a)$.
- Approximation integral area with rectangles: $\int_a^b f(x)\,dx \approx \sum_i f(x)\,dx$.
- `prop` Properties of integrals:
    - Linearity: $\int \alpha f(x) + \beta g(x)\,dx \equiv \alpha \int f(x)\,dx + \beta \int g(x)\,dx$.
    - Constant rule: $\int k\,dx \equiv kx + C$.
    - Power rule: $\int x^n\,dx \equiv \frac{x^{n+1}}{n+1} + C\ (n \neq -1)$.
    - Log rule: $\int \frac{1}{x}\,dx \equiv \ln(|x|) + C$.
    - Exponential rule: $\int a^{kx}\,dx \equiv \frac{a^{kx}}{k\ln a}\ (a > 0, a \neq 1)$.
    - Sine rule: $\int \sin(x)\,dx \equiv -\cos(x) + C$.
    - Cosine rule: $\int \cos(x)\,dx \equiv \sin(x) + C$.
    - Derivatives can be easily computed automatically, but for antiderivatives this is not true.
        - But those properties can be applied.

1.2.2.3 Partial derivatives

- A function $y = f(x_1, \dots, x_n) = f(\mathbf{x})$ is given, where $y \in \mathbb{R}$ and $\mathbf{x} \in \mathbb{R}^n$.
- $\frac{\partial}{\partial x_j} f(\mathbf{x})$ measures how f changes as only the x_j variable increases at point $\mathbf{x}$.
    - $\frac{\partial}{\partial x_j} f(\mathbf{x}) = \lim_{h\to 0} \frac{f(\mathbf{x}+h\hat{i}_j)-f(\mathbf{x})}{h} = \lim_{h\to 0} \frac{f(x_1,\dots,x_j+h,\dots,x_n)-f(x_1,\dots,x_n)}{h}$.
        - h multiplied by the versor $\hat{i}_j$ is a *nudge* in the direction given by the current versor.
- `def` **Gradient** (of f): $\nabla_x f$ (or ∇f) $= [\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n}]^T$.
    - The vector collecting all partial derivatives.
    - When $\nabla f(\mathbf{x}_0) = \mathbf{0}$, the tangent plane is horizontal.
        - The function doesn't grow in any direction along that point.

- `prop` **Chain rule for multivariate calculus**: $\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt}$.
  - $z = f(x, y)$ is assumed and let x, y depend on an additional variable t (z can be seen as $f(x(t), y(t))$).
  - `prop` **Chain rule for multivariate calculus**: $\frac{df}{dt} = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}\frac{dx_i}{dt}$.
    - More in general for $f\colon \mathbb{R}^n \to \mathbb{R}$ when $x_1, \dots, x_n$ depend on a variable t.

1.2.2.4 Directional derivatives

- `def` **Directional derivative**: $D_{\mathbf{u}} f(\mathbf{x}) = \lim_{h\to 0} \frac{f(\mathbf{x}+h\mathbf{u}) - f(\mathbf{x})}{h}$.
- DD of f at $\mathbf{x}$ in $\mathbf{u}$ direction: the rate of change in the direction given by the unit vector $\mathbf{u}$.
- The derivative of $f(\mathbf{x} + \alpha\mathbf{u})$ w.r.t. α evaluated at $\alpha = 0$.
- Using the chain rule the expression for $D_{\mathbf{u}} f(\mathbf{x})$ can be easily computed.
  - $D_{\mathbf{u}} f(\mathbf{x}) = \frac{d}{d\alpha} f(\mathbf{x} + \alpha\mathbf{u})|_{\alpha=0} = \sum_{i=1}^{n} \frac{\partial f(\mathbf{x}+\alpha\mathbf{u})}{\partial x_i}|_{\alpha=0} \frac{dx_i}{d\alpha} = \nabla f(\mathbf{x}) \cdot \mathbf{u} = \mathbf{u}^T \nabla f(\mathbf{x})$.

1.2.2.5 Gradient and optimization

- Goal: to find the direction in which the function increases the most.
  - To find $\mathbf{u}$ such that $\nabla_{\mathbf{u}} f$ is largest.
- $\max_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} D_{\mathbf{u}} f(\mathbf{x}) = \max_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} \mathbf{u}^T \nabla f(\mathbf{x}) = \max_{\mathbf{u},\mathbf{u}^T\mathbf{u}=1} |\mathbf{u}||\nabla f(\mathbf{x})| \cos(\theta)$.
  - $\mathbf{u}^T\mathbf{u} = 1$ is used to ensure that the $\mathbf{u}$ is still a unit vector.
    - $\|\mathbf{u}\|_2 = \sqrt{\sum_i u_i^2} = \sqrt{u \cdot u} = \mathbf{u}^T\mathbf{u}$.
  - $|\mathbf{u}| = 1$ and since $\nabla f(\mathbf{x})$ doesn't depend on $\mathbf{u}$.
    - Therefore, only to find $\mathbf{u}$ that maximises $\cos\theta$ is needed.
  - Therefore the maximum is attained when $\mathbf{u}$ is in the same direction as $\nabla f(\mathbf{x})$.
- `!` The **gradient** points in the direction in which f **increases the most**.

1.2.2.6 Jacobian Matrix

- `def` **Jacobian matrix**: $\mathbf{J}_{i,j} = \frac{\partial}{\partial x_j} f(\mathbf{x})_i$ or $\mathbf{J} = [\nabla[f(\mathbf{x})_i)^T]_{i=1}^m$.
- A multi-valued, multi-variable function $f\colon \mathbb{R}^n \to \mathbb{R}^m, f(\mathbf{x}) = [f(\mathbf{x}_1), \dots, f(\mathbf{x}_n)]$ is considered.
- The Jacobian matrix $\mathbf{J} \in \mathbb{R}^{m\times n}$ contains the partial derivative of all $f(\mathbf{x})_i$ for all variables x_j.
  - With $1 \le i \le m$ and $1 \le j \le n$.

1.2.2.7 Hessian Matrix

- A **second derivatives** is a derivative of a derivative.
  - Specify how the first derivative will change as the input is varied (it measures **curvature**).
  - With $f\colon \mathbb{R}^n \to \mathbb{R}$, n^2 second derivatives can be computed as $\frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x})$.
    - There is a second derivative for each partial derivative.
- `def` **Hessian matrix** ($H(f)$): $H(f) = \mathbf{J}(\nabla f)$.
  - It contains all these partial derivatives.
- `prop` Properties of Hessian matrices:
  - The Hessian Matrix is **symmetric** where the second partial derivates are continuous (Schwarz's Theorem).
    - Anywhere that the second partial derivatives are continues, differential operators are commutative.
    - Therefore their order can be swapped: $\frac{\partial^2}{\partial x_i \partial x_j} f(\mathbf{x}) = \frac{\partial^2}{\partial x_j \partial x_i} f(\mathbf{x})$.
  - When $\nabla f(\mathbf{x}_0) = \mathbf{0}$, the Hessian helps detect **minimum**, **maximum**, or **saddle points**.
    - **Maximum**: if the Hessian is negative definite (i.e. all eigenvalues are < 0).
    - **Minimum**: if the Hessian is positive definite (i.e. all eigenvalues are > 0).
    - If the Hessian is neither positive nor negative definite (at least one zero eigenvalue exists):
      - **Saddle point**: if there is at least one positive eigenvalue and one negative.
      - **Inconclusive test**: otherwise.

1.2.3 Probability theory

- Can be seen as the extension of logic to deal with uncertainty.
- Logic provides a set of formal rules for determining what proposition are implied to be true.
  - Given the assumption that some other set of propositions is true or false.
- PT provides a set of formal rules for determining the likelihood of a proposition being true.

- Given the likelihood of other propositions.

1.2.3.1 Probability distributions

- **Random variable**: a variable that can take on different values randomly.
  - Random variables may be **discrete** or **continuous**.
  - The set of all possible values taken by a random variable x is denoted Ω_{x}.

Discrete probability distributions

- Described using a **probability mass function** (PMF, $P(\mathbf{x})$):
  - It maps from a state of a random variable to the probability of that random variable taking on that state.
  - $x \sim P$: the random variable x follows the $P(\mathrm{x})$ distribution.
- `prop` Properties of a PMF (not to be confused with axioms of probabilities):
  - To be on a PMF on a random variable x, a function P must:
    - The domain of P must be the set of all possible states of x.
    - $\forall x \in \mathrm{x}\colon 0 \leq P(x) \leq 1$.
    - $\sum_{x \in \mathrm{x}} P(x) = 1$.
  - With S, S_1 and S_2 being sets of possible outcomes:
    - $P(S) = \sum_{x \in S} P(x)$ ($P(S)$ is a shorthand for $P(\mathrm{x} \in S)$).
    - $P(S_1 \cup S_2) = P(S_1) + P(S_2) - P(S_1 \cap S_2)$.
    - $P(\Omega - S) = 1 - P(S)$.

Continuous probability distributions

- When working with continuous random variables, a **probability density function** (PDF, $p(\mathbf{x})$) is used.
- `prop` Properties of a PDF:
  - To be on a PDF on a random variable x, a function p must:
    - The domain of p must be the set of all possible states of $\mathbf{x}$.
    - $\forall x \in \mathrm{x}\colon p(x) \geq 0$ ($p(x) \leq 1$ is not required).
    - $\int p(x)\, dx = 1$.

Marginal probability

- The probability distribution over a subset of the set of variables.
- Marginal probabilities are computed by summing over all values of other variables.
- `eg` With discrete variables x and y with a joint distribution $P(\mathrm{x}, \mathrm{y})$:
  - The marginal distribution $P(x)$ is $\forall x \in \mathrm{x}\colon P(x) = \sum_y P(\mathrm{x} = x, \mathrm{y} = y)$.
- `eg` With continuous variables x and y with a joint distribution $p(\mathrm{x}, \mathrm{y})$:
  - The marginal distribution $p(x)$ is $\int p(x, y)\, dy$.

1.2.3.2 Conditional probability

- **Conditional probability**: $P(\mathrm{y} = y \mid \mathrm{x} = x) = \frac{P(\mathrm{y}=y, \mathrm{x}=x)}{P(\mathrm{x}=x)}$.
- In many cases, it's useful to have the probability of some event given that some other has happened.
- `def` **Chain rule of Conditional probabilities**: $P(\mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(n)}) = P(\mathrm{x}^{(1)}) \prod_{i=2}^{n} P(\mathrm{x}^{(i)} \mid \mathrm{x}^{(1)}, \ldots, \mathrm{x}^{(i-1)})$.
  - Any joint probability distribution over many random variables may be decomposed.
    - Into products of conditional distribution over only one variable.
- `def` **Bayes rule**: $P(\mathrm{x} \mid \mathrm{y}) = \frac{P(\mathrm{y} \mid \mathrm{x})}{P(\mathrm{y})}$.
  - Used to find where $P(\mathrm{x} \mid \mathrm{y})$ where $P(\mathrm{y} \mid \mathrm{x})$ is known.

Independence

- **Independent variables** ($\mathrm{x} \perp \mathrm{y}$): x and y if their probability distribution can be expressed as a product of two factors, one involving only x and one involving only y.
  - $\forall x \in \mathrm{x}, y \in \mathrm{y}\colon p(\mathrm{x} = x, \mathrm{y} = y) = p(\mathrm{x} = x)p(\mathrm{y} = y)$.
  - $\mathrm{x} \perp \mathrm{y} \iff \forall x \in \mathrm{x}, y \in \mathrm{y} : p(x \mid y) = p(x) \wedge p(y \mid x) = p(y)$.
- **Conditional independent variables** ($\mathrm{x} \perp \mathrm{y} \mid \mathrm{z}$, given a random variable z): x and y if the conditional probability over x and y factorizes in this way for every value of z.
  - $\forall x \in \mathrm{x}, y \in \mathrm{y}, z \in \mathrm{z}\colon p(\mathrm{x} = x, \mathrm{y} = y \mid \mathrm{z} = z) = p(\mathrm{x} = x \mid \mathrm{z} = z)p(\mathrm{y} = y \mid \mathrm{z} = z)$.

- x and y may not be independent per se, but they may be come it when another value is known.

1.2.3.3 Core statistical measures

- **Expectation** (μ):
  - The expected value of $f(x)$ with respect to $P(x)$ is the mean value that f takes on where x is drawn from P.
  - Discrete variables: $\mathbb{E}_{x\sim P}[f(x)] = \sum_x P(x)f(x)$.
  - Continuous variables: $\mathbb{E}_{x\sim p}[f(x)] = \int p(x)f(x)\,dx$.
  - `prop` **Linearity of the expectation operator**: $\mathbb{E}[\alpha f(x) + \beta g(x)] = \alpha\mathbb{E}[f(x)] + \beta\mathbb{E}[g(x)]$.
- `def` **Variance** (σ^2): $Var[f(\mathrm{x})] = \mathbb{E}[(f(\mathrm{x}) - \mathbb{E}[f(\mathrm{x})])^2] = \mathbb{E}[f(\mathrm{x})^2] - \mathbb{E}[f(\mathrm{x})]^2$.
  - It gives a measure of how much the values of a function of x vary as a different values of x from its probability distribution is sampled.
- **Standard deviation** (σ): the square root of the variance.
  - For a normally distributed variable, about 95% of the points fall into the range $\mu \pm 2\sigma$.

Covariance

- `def` **Covariance**: $Cov(\mathrm{x}, \mathrm{y}) = \mathbb{E}_{x,y\sim P(\mathrm{x},\mathrm{y})}[(x - \mathbb{E}[\mathrm{x}])(y - \mathbb{E}[\mathrm{y}])] = \mathbb{E}_{x,y\sim P(\mathrm{x},\mathrm{y})}[(x - \mu_\mathrm{x})(y - \mu_\mathrm{y})]$.
- It gives a sense of how much two random variables are related to each other.
- It is affected by the scale of variables.
- `def` **Correlation**: $Corr(x, y) = \frac{Cov(x,y)}{\sigma_x\sigma_y}$.
  - Adjust the scale of each variable, ensuring that the relation between the variables is measured without being influenced by their individual magnitude.
  - A correlation close to ± 1 indicates a strong relationship between the variables.
  - A correlation close to 0 suggests that the variables may be independent.
- **Covariance** and **dependence** are related, but are in fact **distinct** concepts.
  - Two variables that are independent have zero covariance.
  - Two variables that have non-zero covariance are dependent.
  - Two variables can have zero covariance and be dependent nonetheless.

1.2.3.4 Common probability distribution

Bernoulli distribution

- **Bernoulli distribution**: a distribution over a single **binary** random variable.
  - Controlled by a single parameter $\phi \in [0, 1]$, which gives the probability of $\mathrm{x} = 1$.
  - `prop` Properties of Bernoulli distributions:
    - $P(\mathrm{x} = 1) = \phi$ and $P(\mathrm{x} = 0) = 1 - \phi$.
    - $P(\mathrm{x} = x) = \phi^x(1 - \phi)^{1-x}$.
    - $\mathbb{E}[\mathrm{x}] = \phi$ and $Var(\mathrm{x}) = \phi(1 - \phi)$.
- **Multinulli distribution** (or categorical): a distribution over a single discrete variable with k (finite) different states.
  - It is parametrized by a vector $\mathbf{p} \in [0, 1]^k$, with $\mathbf{1}^T\mathbf{p} = 1$.
    - Where p_i gives the probability of the i-th state.
  - Often used to refer to distributions over **categories** of objects.
    - The state 1 having numerical value 1, etc, is not assumed.
    - usually no expectation or variance is computed.

Binomial distribution

- It gives the probability of observing a given number of success in a **repeated Bernoulli experiment**.
- It is parametrized by:
  - p: the probability of success of the Bernoulli experiment.
  - N: the number of total repetitions of the Bernoulli experiment.
- If $\mathrm{x} \sim Bi(p, N)$ then:
  - $P(\mathrm{x} = k) = \binom{N}{k}p^k(1 - p)^{N-k}$ ($\binom{N}{k} = \frac{N!}{k!(N-k)!}$).
  - $\mathbb{E}[\mathrm{x}] = Np$ and $Var[\mathrm{x}] = Np(1 - p)$.

Gaussian (or normal) distribution

- Most commonly used distribution over real numbers.
- It is parametrized by mean μ and variance σ^2.
- If x $\sim \mathcal{N}(\mu, \sigma^2)$ then:
  - $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{(x-\mu)^2}{2\sigma^2})$.
    - $\frac{1}{\sqrt{2\pi\sigma^2}}$ to ensure that its integral will be 1.
    - It goes to 0 exponentially fast as x goes far from μ.
- Many distribution in the real world are truly close to being normal distribution.
  - `th` **Central limit theorem of PT**:
    - Let $X_1, \dots, X_n$ be a sequence of iid random variable with finite mean μ and variance σ^2.
    - Then the distribution of $S = \sum_{i=1}^{n} X_i$ approaches $\mathcal{N}(n\mu, n\sigma^2)$ as n approaches infinity.
    - The average $\frac{1}{n}S$ also approaches $\mathcal{N}(n\mu, n\sigma^2)$ as n approaches infinity.
- Out of all possible probability distribution with the same mean and variance, $\mathcal{N}$ **encodes the maximum amount of uncertainty**.
  - It is the distribution having the **maximum entropy**.
  - With an unknown phenomenon, to assume that it behaves as $\mathcal{N}$ takes the least amount of assumptions.
- $\mathcal{N}$ generalizes to $\mathbb{R}^n$: $\mathcal{N}(\mathbf{x}, \mu, \Sigma) = \frac{1}{\sqrt{(2\pi)^n \det(\Sigma)}} \exp(-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu))$.
  - μ: a vector denoting the mean of the distribution.
  - Σ: the covariance matrix of the distribution.
    - Covariance matrices are **symmetric** and **positive semi-definite**.
    - Their main diagonal contains variances.
    - If $\Sigma = \sigma^2\mathbf{I}$ the variance is the same in every direction: the distribution is said to be **isotropic**.

Exponential and Laplace distributions

- `def` **Exponential distribution**: $p(x; \lambda) = \lambda \exp(-\lambda x)$, $x \geq 0$.
  - In DL, a distribution with a sharp point at $x = 0$ is needed.
- `def` **Laplace distribution**: Laplace$(x; \mu, \gamma) = \frac{1}{2\gamma} \exp(-\frac{|x-\mu|}{\gamma})$.
  - Closely related to the exponential one.
  - It allows to place a sharp peak of probability mass at an arbitrary point μ.

Mixtures of distributions

- It's common to define probability distributions by combining other simpler distributions.
- One common way is to construct a **mixture distribution**: $P(\mathrm{x}) = \sum_i P(\mathrm{c} = i) P(\mathrm{x} \mid \mathrm{c} = i)$.
  - Where $p(c)$ is a multinulli distribution over the components of the mixture.
- **Latent variables**:
  - Random variables that **cannot be observed directly**.
    - The component identity variable c of the mixture model provides an example.
  - Latent variables may be related to x through the joint distribution.
    - Then $P(\mathrm{x}, \mathrm{c}) = P(\mathrm{x} \mid \mathrm{c}) P(\mathrm{c})$.

1.2.3.5 Useful properties of common functions

- `def` **Sigmoid**: $\sigma(x) = \frac{1}{1+\exp(-x)}$.
  - Often used to produce the ϕ paramter of a Bernoulli distribution.
  - It **saturates** for large (negative/positive) values of x.
- `def` **Softplus**: $\zeta(x) = \log(1 + exp(x))$.
  - A smoothed version of $x^+ = \max(0, x)$.

1.2.3.6 Continuous random variables

Measure theory

- A proper formal understanding of Continuous random variables and PDF requires **measure theory**.
  - Without it, one might encounter paradoxical situation.
    - `eg` To construct two set S_1 and S_2 where $S_1 \cap S_2 = \emptyset$ such that $p(x \in S_1) + p(x \in S_2) > 1$.

        - These paradoxes usually involve constructing very exotic sets, but the possibility exists.
    - One of MT key contributions is that it provides a framework to characterize sets where probabilities can be consistently computed, thus avoiding paradoxes.
- **Negligibly small** set of points:
    - It has **measure zero** (eg a line in $\mathbb{R}^2$ has measure zero).
    - Any **union of countably many sets** having measure zero has measure zero.
        - So the set of all rational numbers has measure zero.
        - A property that **holds almost anywhere**:
            - When a property holds throughout all of space except for points in a set of measure zero.

Continuous variables that are deterministic functions of one another

- Two random variables $\mathbf{x}$ and $\mathbf{y}$, such that $\mathbf{y} = g(\mathbf{x})$, are given.
    - Where g is an invertible, continuous differentiable transformation.
    - Since g is invertible, $\mathbf{x} = g^{-1}(\mathbf{y})$.
        - By inverting the transformation, the probability in the initial space should be computable.
    - Unfortunately: $p_y(\mathbf{y}) \neq p_x(g^{-1}(\mathbf{y}))$.
        - Since the **transformation changes the space**.
- This approach fails to account for the **distortion of space** introduced by g.
    - The probability of x lying in an infinitesimally small region with volume δx is given by $p(x)\delta x$.
    - Since g can expand or contract space, the infinitesimal volume surround x in x space may have different volume in y space.
- To correct the problem the following property must be preserved: $|p_y(g(x))dy| = |p_x(x)dx|$.
    - Which yields $p_y(g(x))|dy| = p_x(x)|dx| \implies p_x(x) = p_y(g(x))|\frac{d}{dx}g(x)|$.
    - Or equivalently, $p_y(y)|dy| = p_x(g^{-1}(y))|dx| \implies p_y(y) = p_x(g^{-1}(y))|\frac{d}{dy}g^{-1}(y)|$.
- In **higher dimension** $g\colon \mathbb{R}^m \to \mathbb{R}^n$ the derivative generalizes to the Jacobian matrix.
    - And the absolute value to the absolute value of the determinant.
    - $P_x(\mathbf{x}) = p_y(g(\mathbf{x}))|det(J)|$.

1.2.3.7 Graphical models

- Often probability distribution can be split into many factors.
    - Like when a random variable influences another, and the latter influences another one, etc.
    - eg $P(a, b, c) = p(a)p(b \mid a)p(c \mid b)$ (the second expression is way simpler).
        - With 10 values for each variable, $10 + 10^2 + 10^2 = 210$ instead of $10^3 = 1000$.
    - These factorizations can greatly reduce the number of parameters needed to describe the distribution.
- Factorization over distributions can be visually described using **graphs** (directed or undirected).

Directed models

- **Directed models**: use graphs with directed edges.
- They represent factorizations into conditional probabilities distributions.
- One factor for every random variable x_i.
- An edge from a to b represent the dependency $P(b \mid a)$.
- `def` **Directed model factorization**: $p(\mathbf{x}) = \prod_i p(x_i \mid P_{a_\mathcal{G}}(\mathrm{x}_i))$.
    - Where $P_{a_\mathcal{G}}(\mathrm{x}_i)$ is the set of parents of x_i.
    - It consists of the conditional distribution of x_i given its parameters.

Undirected models

- **Undirected models**: use graphs with undirected edges.
- Variables influence each others.
- They represent factorizations using a set of functions.
    - Not necessary, nor common that they are probabilistic distributions.
- One factor $\phi^{(i)}$ per clique $C^{(i)}$ in the graph.
- `def` **Undirected models factorization**: $p(\mathbf{x}) = \frac{1}{Z}\prod_i \phi^{(i)}(C^{(i)})$.
    - Where $Z = \sum_{x \in \mathbf{x}} \prod_i \phi^{(i)}(C^{(i)})$ (very problematic to compute).
    - i ranges between all the cliques (fully-connected graph subsets) in the graph.
    - The normalized product of all factors.

1.2.4 Information Theory

- Revolves around quantifying how much information is present in a signal.
- Useful applications of IT for DL:
  - To characterize probability distributions.
  - To quantify similarity between probability distributions.
    - `eg` Minimize the distance to a goal distribution (the one that generate Tr data).
- **Quantifying information** (main intuition behind IT):
  - The quantity of information carried by a message depends on **how likely it is**.
    - Learning that an **unlikely event** has occurred is **more informative**.
  - The **less probable** an even is, the more surprising it is.
    - The more surprising it is the **more information** it yields.
    - An event with probability 100% is **perfectly unsurprising and yields no information**.
  - **Independent events** should have **additive information**.

1.2.4.1 Shannon's Entropy Measure

- `def` **Shannon's Entropy Measure**: $H(\text{x}) = \mathbb{E}_{x\sim P}[I(x)] = -\sum_x P(x)\log P(x)$.
- It captures the average amount of *information* across all possible outcomes of a random variable.
- $\sum$ if the probability is discrete, otherwise $\int$.
- `def` **Self-information**: $I(x) = -\log P(x)$.
  - Used to quantify the uncertainty of an event.
  - $-\log$ used to guarantee that the more surprising an event is the **more information** it yields.
- `th` **Shannon's Source Coding Theorem**: $H(x)$ provides a lower bound for the average length of code-words in an optimal encoding of the possible values of x.
  - An optimal encoding for messages (with different probabilities) is needed.
  - Ideal: if a message is very frequent, a smaller code is assign to it.
    - Since it's very frequent, a smaller code is better to minimize communication.
- The entropy of a random variable $\text{x} \sim \text{Bernoulli}(\phi)$ as ϕ varies from 0 to 1.
  - 0 nats (**natural unit of information**) at $\phi = 0$ and $\phi = 1$, 0.7 nats at $\phi = 0.5$ (upward parabola).
  - The quantity of information send with a message with $\phi = 0$ or $\phi = 1$ is 0 (no information gained).
  - The entropy of a distribution is maximized for a uniform distribution.

1.2.4.2 Kullback-Leibler divergence

- `def` **Kullback-Leibler divergence**: $D_{KL}(P\|Q) = \mathbb{E}_{x\sim P}[\log \frac{P(x)}{Q(x)}] = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)]$.
- $P(\text{x})$ and $Q(\text{x})$ are two separate distribution over the same random variable x.
- The KL is used to measure how much different are these distributions.
  - If two distributions are the same, there is no information loss $\implies D_{KL} = 0$.
  - If two distributions are very dissimilar, there will be a substantial information loss.
- In the case of discrete variables:
  - It is the **extra amount of information** needed to send a message containing symbols drawn from P.
  - When it is used a code that was designed to minimize the length of message drawn from Q.
- `prop` Properties of the KL divergence:
  - KL divergence is always **non-negative**.
  - KL divergence is $0 \iff P$ and Q are the same distribution.
    - Or are equal *almost anywhere* in the case of continuous variables.
  - KL **is not a distance**.
    - A distance should also be symmetric and satisfy the triangle inequality.
    - The fact that KL is not symmetric has important consequences when the distance between P and Q has to be minimized.
    - Most of the times, minimizing $D_{KL}(p\|q)$ yields different output than minimizing $D_{KL}(q\|p)$.
- `eg` An optimization problem is used to approximate a bimodal distribution with a unimodal gaussian distribution.
  - By optimizing $\min_q[D_{KL}(q\|p) = \min_q[\mathbb{E}_{x\sim q}[\log \frac{q(x)}{p(x)}]$:
    - p is given and can't be modified, instead q is *tuned* to approximate the target distribution.
    - The result is a narrow distribution that approximate only one of the twos modes.
  - By optimizing $\min_q[D_{KL}(p\|q) = \min_q[\mathbb{E}_{x\sim p}[\log \frac{p(x)}{q(x)}]$:
    - The result is a shallow distribution that approximate both of the two modes, but in a more inaccurate way.
  - It is possible to approximate by composing the best parts of both results.

- By just chaning the order of KL argument, there are two very different solution to the optimization problem.

1.2.4.3 Cross-Entropy

- `def` **Cross-Entropy**: $H(P, Q) = H(P) + D_{KL}(P\|Q) = -\mathbb{E}_{x\sim P}[\log Q(x)]$.
- The average number of bits needed to encode message for code Q with a code designed for P.
- Similar to $D_{KL}(P\|Q) = \mathbb{E}_{x\sim P}[\log P(x) - \log Q(x)]$.
  - Similar expression, but it lacks the term $\log P(x)$.
  - D_{KL} measures the expected **extra** number of bits.
    - While H measures the total number of bits.
- Minimizing $H(P, Q)$ wrt Q is the same as minimizing $D_{KL}(P\|Q)$.
  - $\min_q D_{KL}(P\|Q) = \min_q \mathbb{E}_{x\sim P}[\log P(x)] - \mathbb{E}_{x\sim P}[\log Q(x)]$.
  - Since minimization is wrt Q, the first element term doesn't contribute to anything.
  - The resulting term is exactly the cross-entropy.

1.3 Introduction to Neural networks

1.3.1 Perceptrons

- Introduced by Frank Rosenblatt in 1958.
  - Presented as a pattern recognition device (although is used for more general problems).
  - It mimics a part of what is known of the mammalian visual system.
- Simple form of neural network used to classify **linearly separable** examples.
- Basic structure of a perceptron:
  - *Input*: a vector $\mathbf{x}$ (eg pixel values, phonemes, outputs from previous neurons, etc).
  - *Output*: a vector $\mathbf{y}$ (single value if only a neuron).
  - *Weight vector*: $\mathbf{w}$.
  - Training set: set of tuples (input, desired output).
    - A **pattern** is an element of the training set.
- **Biological neural networks**:
  - In the brain the basic computational unit is the **neuron**.
    - A neuron collect, elaborate and propagate electric signals.
    - The output electric signal is proportional to the received input.
    - Neurons collect and transmit informations **in parallel** (as in ANN).
    - There are 10 billion neurons and 60 millions of millions of millions of **synapses**.
  - In the same way, the basic computational unit of NN is the neuron.
    - Several neurons are connected to each other by synapses.
    - Very simple operations: deciding if the **total input is higher than a threshold**.

1.3.1.1 McCulloch & Pitts' model of a neuron [1943]

- `def` **Network input** (*netinput*) to j: $\mathbf{x} \cdot \mathbf{w} = \mathbf{x}^T\mathbf{w} = \sum_{i=1}^{p} x_i w_i$.
  - For each input value x_i what reaches the neuron j is $w_{ji}x_i$.
    - w_{ji} is the weight of the synapse from i to j.
      - A synapse can be **excitatory** or **inhibitory**, therefore the weight can be $<, >, = 0$.
- The output of the neuron will depend on the netinput and the bias.
  - The **bias** represents the natural predisposition of the neuron to activate.
    - If the bias < 0, for the neuron to be activated, incoming input values must be high.
    - If the bias > 0, the neuron is often activated, also with low inputs.
    - To simplify the computation, the bias is represented as an extra input element $x_0 = 1$ with $w_0 = b$.
      - In this case, $v + j = \sum_{j=0}^{p} w_{ji}x_i$.
  - `def` **Perceptron output**: $y_j = \varphi(uj + b)$.
    - φ is the **activation function** (eg sigmoid).
      - **Perceptron activation function**:
        - 1 for all input vectors $\mathbf{x}$ s.t. $\mathbf{x} \cdot \mathbf{w} > 0$.
        - -1 for all input vectors $\mathbf{x}$ s.t. $\mathbf{x} \cdot \mathbf{w} \leq 0$.
    - $uj + b$ is the **activation potential** or local field of j.

1.3.1.2 Perceptron Learning

- The **decision boundary** is made by all $\mathbf{x}$ s.t. $\mathbf{x} \cdot \mathbf{w} = 0$.
  - In 3-D with $w_0 = 0$, $w_1x_1 + w_2x_2 = 0$ is the equation of the DB line.
- The perceptron **learn** to classify examples linearly separable by **adjusting the weights**.

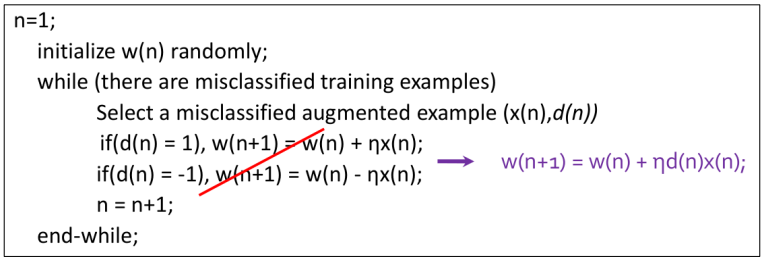

Figure 1: Perceptron Learning Algorithm.

Algorithm: Perceptron Learning

- Adjust the weights to obtain the desired classification.
  - The activation function cannot be modified, only the weights.
  - Learning based on **correction of the error** (for each misclassified pattern).
- **Weights update**:
  - Updates are performed **pattern by pattern** (and not by **epoch**, as in other neural networks).
    - An **epoch** is a iteration over all elements of the training set.
    - Instead, here weights are updated for each pattern misclassified.
  - Correct output: $w(n+1) = w(n)$.
  - Incorrect output:
    - $w(n+1) = w(n) - \eta(n) \times (s)$ if $x(n) \cdot w(n) > 0$ and $x(n) \in C_2$ (output too high).
    - $w(n+1) = w(n) + \eta(n) \times (s)$ if $x(n) \cdot w(n) \leq 0$ and $x(n) \in C_1$ (output too low).
    - η is the **learning rate** (with high η updates will be abrupt).
- th **Convergence theorem**: if the problem is **linearly separable** the learning algorithm will find a solution.
  - At each iteration, the weight vector is modified and, as a consequence, the decision boundary.
    - This theorem guarantees that **weight adjustment terminates**.
  - The proof is based on a geometric interpretation about how the weights vector is updated.
    - A: $\|w(k+1)\|^2 \geq \frac{k^2\alpha^2}{\|w^*\|^2}$ (lower bound).
    - B: $\|w(k+1)\|^2 \leq k\beta$ (upper bound).
    - A and B are compatible only if $\frac{k^2\alpha^2}{\|w^*\|^2} \leq k\beta \rightarrow k \leq \frac{\beta\|w^*\|^2}{\alpha^2}$.
      - k can't be larger than this **finite** quantity, the algorithm **terminates** (QED).
  - It converges for **linearly separable problems**.
    - But simple (eg XOR) and non-linearly solvable problems cannot be solved [Minsky & Papert, 1969].
    - To **test if a dataset is linearly separable**:
      - The model should be trained on the whole dataset (train and test set).
      - Its training accuracy should be 100% (if not, then the dataset is not linearly separable).

1.3.1.3 Multilayer networks

- To solve more complex problems **hidden units** are introduced.
  - Each unit solves part of the problem, then solutions are combined.
  - But a different learning algorithm (**backpropagation**) is introduced.
- With enough hidden units, the network can **discriminate a convex zone**.

1.3.2 Multilayer neural networks

- In ML, a task is approached by trying to create a system that can learn from examples.
- By combining several perceptrons into a **complex networks**, a very large quantity of function can be modelled.
  - A perceptron can easily represent a NAND gate (eg with $x_1, x_2 \in \{0, 1\}$, $(-2, -2) \cdot x + 3$).
    - Any other logic operator can be built in terms of NAND operators only.
    - Any NAND-only complex circuit (eg binary adder with carry) can be translated into a perceptron NN.
- Networks as computational device:
  - Perceptron networks can perform all sorts of computation.
  - Algorithms that **learns those computations** can be made.

1.3.2.1 Universal Approximation Theorem

th **Universal Approximation Theorem**: a **feedforward network** with a **linear output layer** and **at least one hidden layer** with **any *squashing* activation function** (eg logistic sigmoid) can **approximate any Borel measurable function** from one finite-dimensional space to another with **any desired non-zero amount of error**, provided that the network is given **enough hidden units**. The derivatives of the feedforward network can also approximate the derivatives of the function arbitrarily well.

- A **linear output layer** computes simply $\mathbf{w} \cdot \mathbf{x} + b$ (no > 0).
- Any continuous function on a closed and bounded subset of $\mathbb{R}^n$ is *Borel measurable*.
  - And therefore may be approximated by a neural network.
- With a big enough NN, any arbitrarily-complex practical problem can be approximated.
  - Empirical observations proved that with enough data, any problem of overfitting can be overcome.

- Unfortunately, in the worst case, an **exponential number of hidden units** may be required.
  - Possibly with one hidden unit for each input configuration that needs to be distinguished.
  - This is easiest to see in the binary case:
    - The number of possible binary functions on vectors $\mathbf{v} \in \{0,1\}$ is 2^{2^n}.
    - Selecting one such function requires 2^n bits, which will in general require $O(2^n)$ degrees of freedom.

1.3.2.2 Sigmoid neurons

- It is hard to learn anything useful using the perceptrons.
- *Desiderata*: *small* changes in any weight (or bias) causes a *small* changes in the output.
  - $\mathbf{w} + \Delta\mathbf{w} \rightarrow output + \Delta output$.
- `def` **Sigmoid neuron**: a neuron where the output is computed as $\sigma(\mathbf{w} \cdot \mathbf{x} + b)$, where $\sigma(z) = \frac{1}{1+e^{-z}}$.
  - The output approximate the Step function when $\mathbf{w} \cdot \mathbf{x} + b$ is very large or small.
    - But it does not change abruptly in proximity of 0.
  - Since the output is in $[0,1]$, each internal node input ranges in $[0,1]$ as well.
- The information relating to changes can be used to **updates the weights**.
  - A Step function doesn't provide any meaningful information about those changes.
  - When the activation function has a larger slope, it's easier to **guide the learning**.
- It is important that the **activation function** is **smooth** in order to control how varies $\Delta output$.
  - Calculus implies that $\Delta output \approx \sum_j \frac{\partial output}{\partial w_j}\Delta w_j + \frac{\partial output}{\partial b}\Delta b$.
    - The *output* must be **differentiable w.r.t. the weights and the biases**.
    - The Step function is not differentiable since it's not continuous (it also has derivatives $= 0$, bad).
  - It is not the exact shape of σ to be crucial, but rather its smoothness.
    - In other situations, the sigmoid activation function is not the best choice.
      - Everything in NN is **situational**, and to develop a proper model is *more an art than a science*.

1.3.2.3 Architecture of Neural networks

- A typical neural network is composed by: an **input layer**, 1-n **hidden layers**, an **output layer**.
  - This architecture is called **multilayer perceptrons** (despite not being built out of perceptrons).
  - The input layer is not a proper *layer*, it only represents the inputs.
    - While all other nodes in other layers are sigmoid neurons.
  - In the output vector, an output neuron can be used to represent each of the possible labels.
    - The final output will be the output unit with the **largest activation**.
  - In a **feedforward network**, a node is connected only to *successive* nodes in the following layer.
    - In a **fully connected FN**, a node is connected to **every** nodes in the following layer.
- The design of input and output layers is application-dependent.
  - `eg` An output neuron can be used to represent each one of the possible labels.
    - The final output is pick up as the output unit with the largest activation.
  - `eg` Each element of the output vector can be modeled as an output neuron.
- Hidden layers design is not as straight forward as the one for input or output layers.
  - NN researchers have developed over time many design heuristics for it.
    - `eg` Sharing weights between neurons allow to learn a single feature detector and apply it in many different locations.
    - `eg` Stacking layers may allow to learn complex features from simpler ones.
    - `eg` Using a large number of hidden units is powerful but it's prone to overfitting or require more data.

1.3.3 Gradient descent

- A learning algorithm to find the weights for a NN.
  - A **cost function** to measure how well a network is doing is defined.
    - $C(\mathbf{w}, \mathbf{b})$ measures how well the network performs on Tr data $(\mathbf{X}, \mathbf{y})$.
  - A way to **minimize this cost function** needs to be found.
    - To solve the optimization problem $\text{minimize}_{\mathbf{w},\mathbf{b}} C(\mathbf{w}, \mathbf{b})$.
  - `def` **Cost function**: $C(\mathbf{w}, \mathbf{b}) = \frac{1}{2n}\sum_{\mathbf{x}} \|y(\mathbf{x} - \mathbf{a})\|^2$.
    - $\mathbf{a}$ is the vector of outputs with $\mathbf{x}$ has an input.
      - It depends on $\mathbf{w}$, $\mathbf{b}$ and $\mathbf{a}$, more properly denoted as $\mathbf{a}_{\mathbf{w},\mathbf{b}}(\mathbf{x})$.
  - Accuracy is a natural evaluation measure, but it is **not smooth**.

1.3.3.1 Gradient descent derivation

! GD is an **iterative optimization algorithm** that at each step move the current position in the **opposite direction** w.r.t. the gradient, $\mathbf{v}' \leftarrow \mathbf{v} - \eta \nabla C$. The error function C is descended following the **direction of the steepest descent**.

- Consider a function $C(\mathbf{v})$ defined as $C : \mathbb{R}^n \to \mathbb{R}$.
- In general it might be very difficult to pinpoint the global minimum.
  - Using Calculus to find **closed form solution** is not feasible.
- The variables are moved by a little $\Delta\mathbf{v} = (\Delta v_1, \Delta v_2)^T$:
  - The vector C changes as $\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 = \nabla C \cdot \Delta v$.
    - $-\nabla C$ is the direction where C decrease the most.
    - $-\nabla C \cdot \nabla C = -\|\nabla C\|_2^2$ (the norm is a positive number).
  - The Taylor expansion is **truncated at the first derivative to use a plane** to approximate the problem.
    - Then the variables are moved along this plane.
- By choosing $\Delta\mathbf{v} = -\eta\nabla C$, it is guaranteed to make the **cost function to decrease**.
  - The step η, called **learning rate**, should be not big nor too small (typically of the order 0.1-0.001).
- The GD is applied until the found derivatives are very small.
- **Second order approximation**:
  - The Taylor expansion can be truncated not at the first but at the second derivative.
    - But with n parameters, there are n^2 derivatives (not feasible with large n).
  - Methods to mitigate this have been developed, but GD still remains the most common implementation.
- **Applying Gradient descent in a NN**:
  - Applying GD to weights and biases allows for training a NN.
  - Start from a fixed point and iterate through the data updating at each step the weights as:
    - $w_k' \leftarrow w_k - \eta\frac{\partial C}{\partial w_k}$.
    - $b_l' \leftarrow b_l - \eta\frac{\partial C}{\partial b_l}$.

1.3.3.2 Stochastic gradient descent (SGD)

- Consider the cost function $C = \frac{1}{n}\sum_{\mathbf{x}} C_{\mathbf{x}}$ (generic averaged CF).
  - The gradient of C is then $\nabla C = \frac{1}{n}\sum_{\mathbf{x}} \nabla C_{\mathbf{x}}$ (∇ is a **linear operator**).
  - To compute the gradient ∇C:
    - The gradients $\nabla C_{\mathbf{x}}$ has to be computed separately for each training input $\mathbf{x}$.
    - And then average them.
  - When the dataset is very big, this becomes too slow.
- In **mini-batch** SDG, a **small subsample** of the dataset is used to approximate the gradient over the entire dataset.
  - Let $x_1, \dots, x_m$ be m randomly chosen training inputs.
  - If m is large enough, it should be apparent that $\sum_{j=1}^{m}\frac{\nabla C_{x_j}}{m} \approx \frac{\sum_{\mathbf{x}}\nabla C_{\mathbf{x}}}{n} = \nabla C$.
- Procedure:
  - A mini-batch is extracted and the weights are updated using the following rules:
    - $w_k' \leftarrow w_k - \frac{\eta}{m}\frac{\partial C x_j}{\partial w_k}$.
    - $b_l' \leftarrow b_l - \frac{\eta}{m}\frac{\partial C x_j}{\partial b_l}$.
  - Then a **new mini-batch** is randomly chosen and the model is trained with those.
  - When all examples have been used an **epoch of training** is completed.
    - At that point a new training epoch is started over.
- The estimate won't be perfect (there will be statistical fluctuations), but it isn't needed.
  - The point is to **move in a general direction** that will help decrease C.

1.3.4 Backpropagation

- Gradient descent is a general procedure for optimizing differentiable functions.
  - Backpropagation is its instantiation in the context of NN.

1.3.4.1 Mathematical preamble to backpropagation

- Notation:

- w^l_{jk}: the weight from the k-th neuron in level $l-1$ to the j-th neuron in the level l.
- b^l_j: the bias of the j-th neuron in the l-th layer.
- z^l_j: the activation value of the j-th neuron in the l-th layer.
- **Activation**: $z^l_j = \sigma(\sum_k w^l_{jk} z^{l-1}_k + b^l_j)$.
  - The sum is on all neurons k in the $(l-1)$-th layer.
- $\mathbf{W}^l$: the matrix having element w^l_{jk} at row j and column k.
  - The x-th row of $\mathbf{W}^l$ corresponds to the weights entering the x-th neuron in the l layer.
- $\mathbf{b}^l$: the column vector having b^l_j as its j-th element.
- $\mathbf{z}^l$: the column vector having z^l_j as its j-th element.
- $f(\mathbf{v})$ is the element-wise application of f to a matrix (output: a matrix).

- `def` **Activation**: $\mathbf{z}^l = \sigma(\mathbf{W}^l \mathbf{z}^{l-1} + \mathbf{b}^l)$.
  - Faster to compute thanks to matrix computation libraries.
  - The **weighting input** to a neuron as level l is: $\mathbf{a}^l \equiv \mathbf{W}^l \mathbf{z}^{l-1} + \mathbf{b}^l$.
    - The component j of a vector $\mathbf{a}^l$ is $a^l_j = \sum_k w^l_{jk} z^{l-1}_k + b^l_j$.
- Working assumptions about the **cost function**:
  - Can be written as an average $C = \frac{1}{n} \sum_{\mathbf{x}} C_{\mathbf{x}}$ over $C_{\mathbf{x}}$ for individual Tr examples.
  - It is a function of the outputs of the neural network (**self-contained**).
  - `eg` Those assumptions are satisfied by the quadratic cost function.
- **Hadamard Product** ($\odot$): the element-wise product of two vector.

- **(BP1)**: $\boldsymbol{\delta}^L = \nabla_{\mathbf{z}} C \odot \sigma'(\mathbf{a}^L)$
- **(BP2)**: $\boldsymbol{\delta}^l = ((\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot \sigma'(\mathbf{a}^l)$
- **(BP3)**: $\frac{\partial C}{\partial b^l_j} = \delta^l_j$
- **(BP4)**: $\frac{\partial C}{\partial w^l_{jk}} = z^{l-1}_k \delta^l_j$

Figure 2: Backpropagation four fundamental equations.

1.3.4.2 Backpropagation four fundamental equations

- The **gradient of the cost function** w.r.t. weights and biases is $\nabla_{\mathbf{w},\mathbf{b}} C = [\frac{\partial C}{\partial w^l_{jk}}, \dots, \frac{\partial C}{\partial b^l_j}, \dots]$.
- `def` **Neuron error**: $\delta^l_j = \frac{\partial C}{\partial a^l_j}$.
  - The error at level l and neuron j.
  - BP provides a procedure to compute it for each neuron in each layer.
    - And then will relate δ_j to the quantities of real interest.
  - Measures how much the CF varies when the inputs of the neurons at that level are slightly perturbed.
  - The term *error* derives from the idea of introducing a *small* error in the inputs of the layer.
    - And then observe how it propagates to the cost function.

First equation of BP

- `def` **First equation of BP** (BP1): $\delta^L_j = \frac{\partial C}{\partial z^L_j} \sigma'(a^L_j)$.
- Specifies how to compute the *error* δ^L at the output layer (L).
- How much the cost function C changes when there is a small change in the input of σ.
- **Easily computable**:
  - a^L_j is computed by letting the inputs to flow through the network up to neuron j the level L.
  - The exact form of $\frac{\partial C}{\partial z^L_j}$ will depend on the form of the cost function.
    - `eg` For the quadratic cost: $C = \frac{1}{2} \sum_j (y_j - z^L_j)^2 \implies \frac{\partial C}{\partial z^L_j} = (z^L_j - y_j)$.
- **First equation of BP** (BP1): $\delta^L = \nabla_{z^L} C \odot \sigma'(\mathbf{a}^L)$.

- The gradient is by definition the collection of the partial derivatives.

Second equation of BP

- `def` **Second equation of BP** (BP2): $\delta^l = ((\mathbf{W}^{l+1})^T \delta^{l+1}) \odot \sigma'(\mathbf{a}^l)$.
- Allows to compute the error for non-output layer.
- Multiplication between:
  - $\sigma'(\mathbf{a}^l)$: measures how fast the output at this level varies in response to a variation of its input.
  - $(\mathbf{W}^{l+1})^T$: measures how that changes propagate through the NN via the connecting weights.
  - δ^{l+1}: how fast that level will change in response to changes in its input.
- Ignore anything before l and after $l+1$ (already accounted by δ^{l+1}).
- **Easily computable**:
  - The only non-trivial term, δ^{l+1} can be computed via BP1 applied to the last layer.
    - And then apply BP2 to the other layers up to the current layer.
  - Information is computed from the back (output) and then **propagated backwards**.

Third and fourth equations of BP

- Those equations are the ones directly related to **applying gradient descent to a NN**.
  - How to compute the cost function given the biases and weights terms.
- `def` **Third equation of BP** (BP3): $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.
  - That is, the error δ_j^l is *exactly equal* to the rate of change $\frac{\partial C}{\partial b_j^l}$.
- `def` **Fourth equation of BP** (BP4): $\frac{\partial C}{\partial w_{jk}^l} = z_k^{l-1}\delta_j^l = z_{in}\delta_{out}$.
  - How to compute the rate of change of the cost w.r.t. any of the weights in the network.
  - It is intended that the weight w.r.t. of which the derivative is taken determines the levels and the indices of z_{in} and δ_{out}.

Consequences of the four equations of BP

- Considering BP4:
  - Whenever z_{in} is small, the gradient will also tend to be small and the learning rate will be slow.
  - Weights originating from **low activation** units will **evolve slowly**.
- Considering BP1:
  - The sigmoid derivative is almost zero when its argument is either very large or very small.
  - The learning will proceed slowly for an output neuron if it's either *low activation* (≈ 0) or *high activation* (≈ 1).
  - When the derivative of the activation function is almost zero, a neuron is **saturated**.
- Same considerations can be made about the implications of BP2.
  - If the neuron saturates the learning will be slow.
- These observations do not rely on the activation function being the sigmoid.
  - They do **apply to any activation function**.
- These observations can be applied to design novel activation functions.
  - `eg` An activation function σ so that σ' is always positive (to avoid saturation).

1.3.4.3 Algorithm: Gradient descent + Backpropagation

- Weights and biases are initialized randomly.
  - And then (hopefully) they will converge to useful values.
- Backpropagation:
  - *Input*: $\mathbf{x}$ will be used to set the corresponding activation $\mathbf{z}^1$ for the input layer.
  - *Output*: the gradient of the cost function given by $\frac{\partial C}{\partial w_{jk}^l} = z_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$.
- Backpropagation provides only the gradient of the error committed on a fixed input.
  - To get (S)GD, iterate over all examples and average the results.
- `eg` $\mathbf{a}^{\mathbf{x},l}$: $\mathbf{a}$ computed at the level l on the example $\mathbf{x}$.
- To fully implement SGD, one needs:
  - An additional loop that iterates over the **minibatches**;
  - An additional loop to iterate through the **epochs**.

Input a minibatch of training examples of length m.

1. **For each training example $\mathbf{x}$**: Set the corresponding input activation $\mathbf{z}^{\mathbf{x},1}$, and perform the following steps:
   - **Feedforward**: For each l=2,3,...,L compute $\mathbf{a}^{\mathbf{x},l} = \mathbf{W}^l \mathbf{z}^{\mathbf{x},l-1} + \mathbf{b}^l$ and $\mathbf{z}^{\mathbf{x},l} = \sigma(\mathbf{a}^{\mathbf{x},l})$.
   - **Output error $\boldsymbol{\delta}^{\mathbf{x},L}$**: Compute the vector $\boldsymbol{\delta}^{\mathbf{x},L} = \nabla_{\mathbf{z}} C_{\mathbf{x}} \odot \sigma'(\mathbf{a}^{\mathbf{x},l})$.
   - **Backpropagate the error**: For each $l = L-1, L-2, \ldots, 2$ compute $\boldsymbol{\delta}^{\mathbf{x},l} = ((\mathbf{W}^{l+1})^T \boldsymbol{\delta}^{\mathbf{x},l+1}) \odot \sigma'(\mathbf{a}^{\mathbf{x},l})$.
2. **Gradient Descent**: For each $l = L, L-1, \ldots, 2$ update the weights according to the rule $\mathbf{W}^l \leftarrow \mathbf{W}^l - \frac{\eta}{m} \sum_{\mathbf{x}} \boldsymbol{\delta}^{\mathbf{x},l} (\mathbf{z}^{\mathbf{x},l-1})^T$, and the biases according to the rule $\mathbf{b}^l \leftarrow \mathbf{b}^l - \frac{\eta}{m} \sum_{\mathbf{x}} \boldsymbol{\delta}^{\mathbf{x},l}$.

Figure 3: Gradient descent + Backpropagation algorithm.

Algorithm efficiency

- To understand how efficient this algorithm it has to be considered:
  - A numerical approximation of the derivative based on computing $\frac{\partial C}{\partial w_j} \approx \frac{C(w+\epsilon e_j) - C(w)}{\epsilon}$:
    - Would require to compute the approximation:
      - For each possible weight (they can be millions).
      - Each time performing a forward pass to compute C with the new weights.
  - The computation of the exact derivative performed by BP requires a **single forward pass** + a **single backward pass**.
- It is trivial to exploit **dynamic programming** to avoid exponential computation.
  - The influence of nodes on later layers onto nodes of the l-th layers is totally captured by δ^{l+1} vector.
  - If this was not the case, to compute the derivative of the cost w.r.t. a weight w^l_{jk} one would need:
    - To integrate the effects over the exponential # of paths that connect that node k at level $l-1$ to the output node.

1.3.5 Best practices in neural networks

1.3.5.1 Slow learning

- Networks based on sigmoid units with a quadratic loss function often learn **very slowly**.
  - When a sigmoid is combined with a squared loss, GD requires to compute the CF derivative wrt the weights.
  - The obtained expression is the sigmoid derivative (flat in most of its domain).
  - Therefore the derivative is almost 0 in most of its domain (**slow learning**).

Cross-entropy

- The **cross-entropy** CF solves this problem without changing the activation function.
- σ will be still defined to be the sigmoid function.
- `def` **Cross-entropy**: $C = -\frac{1}{n} \sum_{\mathbf{x}} [y \ln z + (1-y) \ln(1-z)]$.
  - Where y is the desired (binary) output and $z \in [0,1]$ is the activation of the neuron.
  - A single neuron with multiple inputs is considered.
  - It still behaves as a cost function:
    - If $y = 1$ and $z \approx 1$, then $-(y \ln z + (1-y) \ln(1-z)) \approx 0$.
    - If $y = 1$ and $z \approx 0$, then $-(y \ln z + (1-y) \ln(1-z)) \approx \infty$.
    - The same goes for $y = 0$ and $z \approx 0$ and $y = 0$ and $z \approx 1$.
  - Always positive and tend to 0 as the computed outputs tend to the desired outputs.
- The sigmoid derivative is $\sigma'(a) = \frac{d}{da}(\frac{1}{1+e^{-a}}) = \frac{e^{-a}}{(1+e^{-a})^2} = \sigma(a)(1-\sigma(a))$.
  - Which yields $\frac{\partial C}{\partial w_j} = \frac{1}{n} \sum_{\mathbf{x}} x_j (\sigma(a) - y)$.
  - The rate of learning depends only on how well the output unit is approximating the desired output.
- **Learning fast when in wrong** and **learning slow when correct** (desirable).
  - And by just changing the CF and not the activation function.
- `def` **Multi-layer Cross-entropy**: $C = -\frac{1}{n} \sum_x \sum_j [y_j \ln z^L_j + (1-y_j) \ln(1-z^L_j)]$.
  - Generalization of CE to many-neurons, multi-layers NN.

- $y = y_1, y_t, ...$ are the desired values at the output neurons.
- Each neuron is treated as a binary problem where the usual CE is applied.
  - And then all these cross-entropies are summed up.
- For numerical reasons, ML libraries doesn't include the sigmoid in z_j^L.

Soft-Max and Log-Likelihood

- When the task involves the prediction of a probability distribution, Soft-Max activation is useful.
  - It shares the benefit of using a cross-entropy.
- `def` **Soft-Max activation**: $z_j^L = \frac{e^{a_j^L}}{\sum_k e^{a_k^L}}$.
  - $\sum_k$ considers the other neurons in the layer.
  - By using the exponential, the highest number get the most of the weight.
    - Usually the maximal number results in almost 1, while the others are much below.
    - A *soft* version of evaluating the *maximum* of a function.
  - $\sum_j z_j^L = \sum_j \frac{e^{a_j^L}}{\sum_k e^{a_k^L}} = \frac{\sum_j e^{a_j^L}}{\sum_k e^{a_k^L}} = 1$ (required in a probability distribution).
  - It guarantees that each activation $z_j^L \in (0, 1)$ (not included) and $\sum_j z_j^L = 1$.
  - The collective behavior of the output units can be seen predicting a distribution of probabilities.
    - A distribution of probabilities Where $z_j^L = P(j \mid x)$.
- `def` **Log-likelihood cost function**: $C \equiv -\ln z_y^L$.
  - y is not the one-hot encoding representing the correct class, y is the number correct class.
    - z_y^L is the output of the **neuron corresponding to the correct class**.
  - It behaves as a cost function:
    - If the network compute the correct output, then $z_y^L \approx 1$ and $C = -\ln z_y^L \approx 0$.
    - If the network compute the *wrong* output, then $z_y^L \approx 0$ and C will assume a large value.
  - Only a single neuron is considered since the others are already **taken into account in the Soft-Max**.
- Using Soft-Max + Log-Likelihood architecture **does not suffer of slow learning**.
  - $\frac{\partial C}{\partial w_{jk}^L} = z_k^{L-1}(z_j^L - y_j)$ (it depends on how far the output is from the desired result).
  - $\frac{\partial C}{\partial b_k} = z_j^L - y_j$.

1.3.5.2 Parameter initialization

- It can have a significant impact on generalization performance and convergence speed.
  - Unfortunately there is very little theory suggesting how to properly initialize the parameters.
  - Initializing them to large values is not ideal according to seen activation functions.
- Main approaches involve:
  - Symmetry breaking:
  - Keeping the variance of the parameters constant.

Symmetry breaking

- By initializing all parameters to the same value, all neurons will compute the same function.
- A common approach is to **initialize the parameters randomly**.
  - Using either a uniform distribution in the range $[-\epsilon, +\epsilon]$ or using $\mathcal{N}(0, \epsilon^2)$.
- The choice of ϵ is very important too.
  - Too big or too small weight will contribute to very large (**gradient explosion**) or very small gradients (**gradient implosion**).

He initialization

- `def` **He initialization**: $\epsilon = \sqrt{\frac{2}{M}}$ (where M is the number of inputs of the neuron being initialized).
- A few attempts have been made to find good initialization values.
  - For ReLU activation units the *He initialization* should make the gradient approximately equal to 1.
- The main idea is to **keep the variance constant between layers**.
  - *Assumptions*:
    - Each layer l of the network evaluates to: $a_i^l = \sum_{j=1}^M w_{ij} z_j^{l-1}$ and $z_j^l = ReLU(a_i^l)$.
    - The weights are initialized drawing from a Gaussian $\mathcal{N}(0, \epsilon^2)$.

    - The outputs of units at layer $l-1$ have zero mean and variance λ^2.
  - It can be shows that $var[z_j^l] = \frac{M}{2}e^2\lambda^2$.
  - Then to keep the variance constant between layers, $\epsilon = \sqrt{\frac{2}{M}}$.

Xavier initialization

- `def` **Xavier initialization**: $\epsilon = \frac{1}{\sqrt{M}}$.
- Designed for activation functions **symmetric** around 0 (sigmoid and tanh).
- It samples initialization weights from $U[-\epsilon, \epsilon]$.
- `def` **Normalized Xavier initialization**: $\epsilon = \frac{6}{\sqrt{N+M}}$.
  - Where N is the number of units in the layer.

1.3.5.3 Convergence

- Interesting facts can be proven about the convergence of GD method.
- The components of **w evolve independently**.
  - After T steps, it can be shown that $\alpha^{(T)} = (1-\eta\lambda_i)^T\alpha_i^{(0)}$.
  - Provided that $|1-\eta\lambda_i| < 1$, the limit as $T \to \infty$ leads to $\alpha_i = 0$.
    - Which implies that the minimum of the error function has been reached.
- GD leads to a **linear convergence** in the neighborhood of a minimum.
  - The **order of convergence is linear** with rate $1-\eta\lambda_i$.
  - Since $\lim_{T\to\infty} \frac{\alpha_i^{(T)}-0}{(\alpha_i^{T-1}-0)^1} = 1-\eta\lambda_i$.
- The **rate of convergence** is governed by $1-(\frac{2\lambda_{min}}{\lambda_{max}})$.
  - Where λ_{max} and λ_{min} are the maximal and minimal eigen-values.
  - By increasing η the speed of convergence can be improved.
  - But $|1-\eta\lambda_i| < 1$ must be ensured, implying $\eta < 2/\lambda_{max}$.
  - The **fastest convergence** is obtained when $\eta = 1/\lambda_{max}$.
    - In this case the rate of convergence in the direction of λ_{max} is 0.
    - Which implies the minimum will be reached in a single step.
  - Assuming to set $\eta = \frac{1}{\lambda_{max}}$, the direction in which convergence is the slowest is λ_{min}.
    - In this case the rate of convergence is $1-\frac{\lambda_{min}}{\lambda_{max}}$.
    - The worst case scenario is when λ_{min} is small compared to λ_{max} (convergence rate ≈ 1).
      - Convergence will be then very slow.
  - The reciprocal of $\frac{\lambda_{min}}{\lambda_{max}}$ is the **condition number** of the Hessian matrix.
    - The larger this number, the slower will be the convergence of the GD.
  - With very dis-equal λ_i, the corresponding hyper-paraboloid will be very irregular.
    - Taking a point on this surface, its gradient won't point exactly to the minimum (**slower convergence**).
      - The gradient of a point is orthogonally directed to the tangent of the level line.
    - While on a perfect hyper-sphere, its gradient will point exactly to the minimum (**faster convergence**).

Momentum

- When convergence is slow, a momentum term can be added to the GD formula.
  - This adds inertia to the motion and smooths out oscillations.
- `def` **Network update with momentum**: $\Delta\mathbf{w}^{(\tau-1)} = -\eta\nabla E(\mathbf{w}^{(\tau-1)}) + \mu\Delta\mathbf{w}^{(\tau-2)}$.
  - Where μ is the **momentum parameter** (typical value: 0.9).
  - In region of low curvature, the momentum is increasing the learning rate by a factor of $\frac{1}{1-\mu}$.
  - In region of high curvature, the effective learning rate will be close to η.
    - In those regions the GD is oscillatory, successive contributions of the momentum tend to cancel.

Learning rate scheduling

- It is advantageous to change the learning rate η during learning.
- In practices, best results are obtained using a larger value for η at the start of training.
  - And then reducing the learning rate over time.

- $\mathbf{w}^{(\tau)} = \mathbf{w}^{(\tau-1)} - \eta^{(\tau-1)} \nabla E(\mathbf{w}^{(\tau-1)})$.
- ex Common learning schedules:
  - **Linear**: $\eta^{(\tau)} = (1 - \tau/K)\eta_0 + (\tau/K)\eta_K$.
    - η_0, η_K and K are three parameters and the formula is applied up to K steps.
    - After these K steps, η is kept fixed at η_K.
  - **Power law**: $\eta^{(\tau)} = \eta_0(1 + \tau/s)^c$.
    - The scaling factor s determines how fast the learning rate decreases.
    - $c < 0$ determines the shape of the descrease (usually $c = -0.5$).
  - **Exponential decay**: $\eta^{(\tau)} = \eta_0 c^{\tau/s}$.
    - s is the scaling factor and $c < 1$ is the decay factor (usually $c \in [0.95, 1)$).
- Extra refinements by optimizing separately η for each direction in parameter space (for each w_i):
  - **AdaGrad** (*Adaptive Gradient*):
    - Reduce each learning rate parameter using the accumulated sum of squares of all derivatives calculated for that parameter.
    - The idea is to **reduce learning rate for parameters** that have received **large updates in the past**.
  - **RMSProp** (*Root Mean Square Propagation*):
    - Replace the sum of squared gradients of AdaGrad with an exponentially weighted average.
    - Fixes a problem with AdaGrad where the weight updates tend to become too small.
  - **Adam** (*Adaptive moments*):
    - Combines RMSProp with momentum.
    - Probably the most used optimization method in DL.

1.3.5.4 Normalization

- Two main problem in DL:
  - Coping with values that vary in very different ranges.
  - Having to deal with **vanishing** and **exploding** gradients.
- Weight normalization try to keep the values computed by the network in a reasonable range.
- Three main approaches: **data** normalization, **batch** normalization, **layer** normalization.

Data normalization

- If the dataset has input variables that span very different ranges.
  - Then a change in one dimension will produce a much larger change in the output wrt a change in another dimension.
- Mean and variance for each dimension is computed and then all data points are rescaled.
  - $\mu_i = \frac{1}{N}\sum_{n=1}^{N} x_{ni}$, $\sigma_i^2 = \frac{1}{N}\sum_{n=1}^{N}(x_{ni} - \mu_i)^2$.
  - $\hat{x}_{ni} = \frac{x_{ni} - \mu_i}{\sigma_i}$.
  - Examples distribution will behave as a Gaussian with $\mu = 0$ and $\sigma^2 = 1$ (nice for learning).
  - And applying this to normally distributed examples won't change anything.

Batch normalization

- The same reasoning can be applied to variables (weights) at each hidden layer.
  - Unfortunately, normalization for those values **cannot be done once for all**.
  - The computation need to be performed every time the variables are updated.
- Batch normalization works by normalizing, across the examples of the mini-batch.
  - Normalizing the values computed at each layer of the network by each unit i.
- One can normalize the pre or the post activations (both work well in practice).
  - Pre-activation normalization:
    - $\mu_i = \frac{1}{K}\sum_{n=1}^{K} a_{ni}$, $\sigma_i^2 = \frac{1}{K}\sum_{n=1}^{K}(a_{ni} - \mu_i)^2$.
    - $\hat{a}_{ni} = \frac{a_{ni} - \mu_i}{\sqrt{\sigma_i^2 + \delta}}$.
      - K is the size of the mini-batch.
      - δ avoids numerical issues when σ_i^2 is small (eg when using small batches).
- This kind of normalization **reduces the representational capability** of the hidden units.
  - To compensate for this, one can **rescale the pre-activation values** to have $\mu = \beta_i$ and $\sigma = \gamma_i$.
    - $\tilde{a}_{ni} = \gamma_i \hat{a}_{ni} + \beta_i$ (with γ_i and β_i learnable parameters).
    - While originally mean and variance across a minibatch were computed by a complex function of all weights and biases.
      - Now they are determined by two simple independent parameters.

        - Which turn out to be much easier to learn by GD.
    - By doing this, normalization is not undone.
        - Before, it was hard to find the right compromise between μ and σ^2 just by acting on hundreds of weights.
        - With γ_i and β_i, only those two weights govern this (lines act way better).
- Once training completes, mini-batches to compute the normalization factors won't be available.
    - A moving average of those factors is kept during the training and used at **inference time**.
    - $\bar{\mu}_i^{(\tau)} = \alpha \bar{\mu}_i^{(\tau-1)} + (1-\alpha)\mu_i$ (with $0 \leq \alpha \leq 1$).
    - $\bar{\sigma}_i^{(\tau)} = \alpha \bar{\sigma}_i^{(\tau-1)} + (1-\alpha)\sigma_i$ (with $0 \leq \alpha \leq 1$).

Layer normalization

- Normalization across the **hidden unit values** for each **data point separately**.
    - Instead of normalizing across examples within a mini-batch for each hidden unit separately.
- It updates the pre-activation values as it follows:
    - $\mu_n = \frac{1}{M}\sum_{i=1}^{M} a_{ni}$, $\sigma_n^2 = \frac{1}{M}\sum_{i=1}^{M}(a_{ni}-\mu_i)^2$.
    - $\hat{a}_{ni} = \frac{a_{ni}-\mu_i}{\sqrt{\sigma_i^2+\delta}}$.
        - n ranges over the examples and M is the number of hidden units in the layer.
- **Additional learnable parameters** β_i and λ_i are introduced (similar to batch normalization).
    - In this case there is no need to keep moving averages to normalize data at inference time.

1.4 Hopfield networks and restricted Boltzmann Machines

1.4.1 Hopfield networks

- Introduced by the physicist John Hopfield in 1982 (2024 Nobel in Physics).
- Used to represent **content addressable memory**:
  - Given a reasonable portion of the memorized information and a partial corrupted portion.
  - It can be completed by recovering other associated information (**pattern completion**).
- The system preserves this property also when some components are broken (**fault tolerance**).
- *Attractor networks*: useful to memorize fundamental memories which attract other inputs.
  - Given a partial (and/or noisy) figure it retrieves the original figure.
- **Pattern completion** characterizes a lot of human memory.
  - These network (and the attractor network in general) are used to model several cognitive phenomena.
- Network of McCulloch-Pitts neurons interconnected, where each neuron can have two states (1 and -1).
  - Given N units:
    - Each neuron is connected with all the other neurons but itself (**fully connected**).
    - Weights are symmetrical $w_{ij} = w_{ji}$.

1. Input. To the network is presented x*: configuration +1 and -1 of length N. For each neuron j, $y_j(0) = x^*(j)$
2. Iteration until convergence. Update the elements of the network in an asyncronous way by choosing a unit at random. The rule to update the elements is

$$y_j(n) = \begin{cases} 1 \text{ if } \sum_{i=0}^{N} w_{ji} y_i(n) > 0 \\ -1 \text{ if } \sum_{i=0}^{N} w_{ji} y_i(n) < 0 \\ y_j(n-1) \text{ if } v_j(n) = 0 \end{cases}$$

3. Convergence. Repeat the operation until we find a **stable state**, s.t. for each unit j $y_j(n+1) = y_j(n)$.
4. Output. This stable state is the output of the network.

Figure 4: Information withdrawal phase in Hopfield networks.

1.4.1.1 Information withdrawal phase

- The **pattern completion** happens during the **withdrawal/retrieval phase**.
  - *State* of a network with N neurons at a given iteration: activations of the N neurons $(1, -1)$.
  - *Input*: vectors of 1 and -1 of dimension N.
    - Each neuron represent an element of the input.
  - Then calculate new activation of a neuron at a time, randomly chosen, until there is no change.
- `def` **Activation of** j **at iteration** n: $y_j(n) = \varphi(v_j(n))$.
  - $v_j =$ netinput to $j = \sum_{i=0}^{N} w_{ji} y_i(n)$.
  - $\varphi(v_j)(n) = -1$ if $v_j(n) < 0$, 1 if $v_j(n) > 0$, and $\varphi(v_j)(n-1)$ if $v_j(n) = 0$.
  - Activations are calculated in an **asynchronous way**.
- A **stable state** (where $\forall i, y_i(n+1) = y_i(n)$) will be the output of the network.
  - **Stable states are not unique**, they depends on the initial input and/or evaluation order.
    - `prop` The opposite of a stable state is a stable state.
  - A initial input (`eg` noisy image) is perturbed to a stable state (`eg` actual image).
  - Stable states act as **attractors**: given any state, it will converge in one of the stored states.

1.4.1.2 Storage phase

- The information withdrawal phase assumes weights are already computed.
  - Those weights are computed in the **storage phase**.
  - Once the memories are stored, the pattern completion can start.
- **Memorized information works as attractor**.
- `def` **Weights computation**: $w_{ji} = \frac{1}{M} \sum_{k=1}^{M} f_k(i) \cdot f_k(j)$ where $j \neq i$.
  - Given M fundamental memories and $f_1, \dots, f_M$ vectors of dimension N.
  - For each fundamental memory the product between the i-th and the j-th element.

    - If the have different sign, $f_k(i) \cdot f_k(j) < 0$ contributes to weaken w_{ji}.
    - If the have the same sign, $f_k(i) \cdot f_k(j) > 0$ contributes to strengthen w_{ji}.
- The learning algorithm is based on **Hebb's principle**:
  - Strengthen the connections between units with the same activation.
  - Weaken the connections between units with opposite activation.
  - Hebb's principle has a neurobiological counterpart:
    - In the synapses between units which are often active at the same time are strengthened.
    - While synapses between neurons which are not simultaneously active are weakened.
    - *Neurons that fire together wire together*.

1.4.1.3 Convergence of Hopfield networks

`th` **Convergence**: given a starting state the network will always arrive to a stable state.

- The activities cannot be changed forever.
- *Proof*:
  - Given N units, there are 2^N possible states of the network.
  - To each of these states is associated an **energy** value.
    - `def` **Energy function**: $E = -\frac{1}{2} \sum_i \sum_j w_{ij} y_i y_j$.
    - E measures the predisposition of the network to change state.
      - Opposite of **harmony** (bad).
      - Harmony increases with the number of positive $w_{ij} y_i y_j$.
        - Higher harmony → less predisposition to change state.
        - Lower harmony → more predisposition to change state.
  - Each change of state of the network leads to a lowering of the energy.
    - Each $w_{ij} y_i y_j$ is positive if y_i and y_j have the same sign and w_{ij} is positive.
      - Or they have opposite sign, and w_{ij} is negative.
      - The activations enforce each other (**low energy → low predisposition to change**).
    - Each $w_{ij} y_i y_j$ is negative if y_i and y_j have the same sign and w_{ij} is negative.
      - Or they have opposite sign, and w_{ij} is positive.
      - Instable (**high energy → high predisposition to change**).
    - Consider how E changes with the activation of the k-th unit.
      - E and E' denote energy before and after the change of y_k.
      - $E = \frac{1}{2} \sum_i \sum_j w_{ij} y_i y_j = -\frac{1}{2} (\sum_{i \neq k} \sum_{j \neq k} w_{ij} y_i y_j + 2 \sum_{j \neq k} w_{kj} y_k y_j)$.
      - $E' = -\frac{1}{2} (\sum_{i \neq k} \sum_{j \neq k} w_{ij} y_i y_j + 2 \sum_{j \neq k} w_{kj} y'_k y_j)$.
      - $E - E' = -\frac{1}{2} (2 \sum_{j \neq k} w_{kj} y_k y_j - 2 \sum_{j \neq k} w_{kj} y'_k y_j) = - \sum_{j \neq k} w_{kj} y_j (y_k - y'_k)$.
        - $\sum_{j \neq k} w_{kj} y_j$ is v_k.
      - There are two cases:
        - y_k passes from $+1$ to -1:
          - Hence $y_k - y'_k > 0$ and $\sum_j w_{kj} y_j < 0$.
          - Then $- \sum_{j \neq k} w_{kj} y_j (y_k - y'_k) > 0$ and $E > E'$.
        - y_k passes from -1 to $+1$:
          - Hence $y_k - y'_k < 0$ and $\sum_j w_{kj} y_j > 0$.
          - Then $- \sum_{j \neq k} w_{kj} y_j (y_k - y'_k) > 0$ and $E > E'$.
      - At each change E lowers.
  - Therefore, there is a moment in which there is **no reachable state with lower energy**.
    - Since the states are finite, at a given point there's a state that doesn't change.
      - This state is a **stable state**.

1.4.1.4 Qualities and drawback of Hopfield networks

- Qualities:
  - They **complete partial patterns**.
  - They **generalize**.
    - Given an input similar to what has been memorized, they recover the corresponding information.
  - They are **fault tolerant**.
    - If some synapses get broken (*brain damage*) the output is still reasonable.
  - They allow the **extraction of prototypes**.

        - If the NN learns several similar informations, it creates their prototype and this is the explicitly presented state.
    - The **learning rule is Hebb-like**.
        - Which is biologically plausible and there is evidence that it exists in the brain.
    - The networks can account for **context effects**.
        - Humans remember better what is learned if they are put in the same context.
- Drawbacks:
    - Not all the stable states are fundamental memories memorized during storage.
        - There are **spurious states**:
            - The opposite of a stable state is a stable state.
            - Combinations of stable states are stable states.
        - Not all fundamental memories are stable states.
        - Storage capacity with few errors given N units $= 0.14N$ (*rule of thumb*).
    - Limited storage capacity.
    - Errors.
    - In the brain synapses are not symmetrical.
    - In the brain there are no stable states but states transitorily stable, which evolve in **successive states**.
        - There are extensions of Hopfield networks that learn sequences of states.
- Hybrid models are being used.
    - Standard convolutional model are poor at pattern completion.
    - `eg` Convolutional + HN for pattern completion.

1.4.2 Restricted Boltzmann machines

- **Spurious states** are the main limitation of Hopfield networks.
    - The capacity of a totally connected net with N units is only about $0.15N$ memories.
        - An inefficient use of the bits required to store the weights.
    - A phase of **unlearning** can be used after learning to get rid of spurious states.
        - Hinton suggests this is the function of dreams.
- Three main idea behind Restricted Boltzmann machine:
    - Visible vs. **Hidden units**.
    - **Stochastic** units.
    - New learning algorithm to correct errors (*fantasies*).
        - HN use *one-shot learning* during the storage phase.

1.4.2.1 Hidden states

- Main features:
    - Two levels: visible and hidden neurons.
    - Connections only between visible and hidden units.
        - All visible units are connected to all hidden units.
    - Weights are symmetrical: $w_{ij} = w_{ji}$.
    - Possible neurons' activation states: 0 or 1 (and not -1 and -1).
    - Neurons are **stochastic**.
    - Input and output are computed on the visible layer.
- Instead of using the net to store memories, use it to construct **interpretations** of sensory input.
    - The input is represented by the visible units.
    - The interpretation is represented by the states of the hidden units.
    - Construct interpretations: detect and **represent regularities** in visual activations.
        - Hidden units specialise in recognising (and generating) **portions of visual patterns**.

1.4.2.2 Stochastic units

- Noisy networks find better energy minima.
- A HN always makes decisions that reduce the energy.
    - This makes it impossible to escape from local minima.
- Stochastic binary units:
    - Replace the binary threshold units by binary stochastic units that make biased random decision.
    - `def` **Activation probability**: $P(s_i = 1) = \frac{1}{1+e^{-\Delta E_i/T}}$ (sigmoid).
        - `def` **Energy gap**: $\Delta E_i = E(S_i = 0) - E(S_i = 1)$.
            - Objective: minimize the energy.

        - $E = -\sum_k \sum_j w_{kj} v_k h_j$.
      - $\Delta E_i = \sum w_{ij} s_j$ for all j connected to i.
      - If $\Delta E_i < 0$, $p(s_i = 1) < 0.5$.
      - If $\Delta E_i = 0$, $p(s_i = 1) = 0.5$ (no difference by assigning 1 or 0).
      - If $\Delta E_i > 0$, $p(s_i = 1) > 0.5$.
    - The *temperature* T controls the amount of noise.
      - $T = 1$ is assumed (can be ignored it).
- **Sampling**:
  - Sampling is used then to attribute a state (0 or 1) using the corresponding probability.
  - The same VL activation is presented multiple times and corresponding HL activation is collected.

1.4.2.3 Execution of a RBM

- Procedure:
  - Present a visual input.
  - Calculate activation at the hidden level.
  - Recalculate activation at the visual level.
  - Possible, start again.
  - Continue until *convergence* (no significant change).
- By evaluating the hidden layer only, the most probable activation at the visible level is provided.
  - An activation of the HL represents a **codified version** of the corresponding VL activation.
    - Experimentally it has been observed each HL neuron *specialises* in a sub-set of VL neurons.
  - It also works with different (or corrupted) patterns.
- RBM are **generative models**: they are used to generate visual states.
  - The process of generating them usually requires several steps.
    - Calculate visual states, then hidden, then visual again, etc.
    - Until equilibrium (a state in which there are no big changes in the probability distribution).
  - They learn to **reproduce** at the visual level the training set.

1.4.2.4 Weights learning in RBM

- Goal: maximize the probability of obtaining at the VL the vectors of Tr.
  - Tr: set of visible vectors to be learned by the RBM (in order to reconstruct or generate).
- A **fantasy** is the reproduction (at the VL) of an original pattern after a very high number of steps.
  - The weights are learned in order to minimize difference between pattern and fantasies.
  - Actually, Hinton discovered that just **two steps** are needed («a very surprising short-cut»).
  - Importance of weights in RBM:
  - Hidden units specialise in recognising (and generating) **portions of patterns**.
  - The probability of unit i to get state 1 depends on the weights to i.
    - Roughly, positive weight w_{ij} augments the impact of neuron j to $p(s_i = 1)$.
  - By observing weights matrices to HU, it can be understood which inputs values augment their probability to activate.
- After learning the weights starting from Tr, RBM is ready to be used:
  - **Generation**: generate a visual activation pattern from a random hidden activation pattern.
  - **Reconstruction**:
    - Start with a visible input.
    - See what visual activation the RBM reproduces after a few passage V-H-V, etc.

Contrastive divergence (learning algorithm)

- Execution:
  - Start with a training vector on the visible units.
  - Update all the hidden units in parallel (since they're independent).
  - Update all the visible units in parallel to get a *reconstruction*.
  - Update the hidden units again.
- `def` **Weights update**: $\Delta w_{ij} = \varepsilon(\langle v_i h_j \rangle^0 - \langle v_i h_j \rangle^1)$.
  - v_i and h_j are the activations of visual unit i and hidden unit j.
  - ε is the learning rate.
  - Just **two step** ($\langle\rangle^0$ and $\langle\rangle^1$) are needed.
- Everything must be iterated for several epochs.
- Overall algorithm:

- Weights are initialised randomly.
- At each epoch n:
  - An element of Tr is chosen.
  - For each weight, Δw_{ij} is computed, and $w_{ij}(n+1) = w_{ij}(n) + \Delta w_{ij}$.
- Until the maximum number of epochs is reached.

1.4.2.5 Deep (or stacked) RBM

- Idea:
  - First train a layer of features that receives input directly from the pixels.
  - This layer discovers patterns of activities, regularities, in the layer below.
  - Then treat the activations of the trained features as if they were pixels and learn features of features in a second HL.
  - This second HL discovers patterns of activities in the layer below correlations between features discovered at the previous level.
  - Each level models the correlation between activities in the level below.
  - Then do it again.
- It's proven that with each additional layer of features the probability for generation Tr is **improved**.
- SBM can be **fine tuned** according to the desired applications.
  - If used for *classification*, by adding a classification layer.
    - The first really successful deep autoencoders [Hinton & Salakhutdinov, 2006].
  - If used for *generation*, with **Deep belief nets**.
  - If used for finding few very descriptive features with **autoencoders**.
- As in CNN, the more deep RBM are specialized on more complex features.

1.5 Convolutional neural networks

1.5.1 Convolutional neural networks

- Designed to solve computer **vision problem** (eg image classification, object detection, etc).
  - Traditionally, computer vision was on *hand-crafted features*.
  - Today features are no longer hand-craft but **learned** by CNN architectures.
    - Usually, those features cannot be described in intelligible terms.
  - Traditional NN don't scale well for image data.
- Key advantages of CNN: local connections, shared weights, pooling, hierarchy of features.
  - Passage through all these phases is necessary since it's not possible to pass from raw pixel to abstract concepts.
  - The focus of DNN is to go through all these levels of abstraction.
  - Designed to encode invariances and equivariances **specific** to image data.
- Multilayer perceptrons don't scale well on big size images.
  - eg With a 1000px square image, the corresponding network should have 3M weights.
  - They also loose the **spatial information** of an image.

1.5.1.1 CNN basic elements

- **Convolution** layers, **pooling** layers, **fully-connected** layers, activation function.
- Two terminological variants:
  - In the complex layer terminology, the convolutional layer is composed by:
    - Convolution stage → Detector stage (eg ReLU) → Pooling stage.
  - In the simple layer terminology, the convolutional layer is the convolution stage only.
- CNN contain multiple such layers of convolution and pooling in succession.
  - In which the output channels of a particular layer form the input channel of the next layer.
  - After convolution blocks there are one or more **fully connected layers**.

1.5.1.2 CNNs applications

- CNNs have state-of-the-art accuracy for **image classification tasks**.
  - Their development was accelerated through the introduction of **ImageNet**.
    - Large-scale benchmark dataset: 14M natural images, hand labelled into 22K categories.
  - Image Classification, object detection, image segmentation, neural style transfer.
  - Used also for other tasks, as analysis of sequential data.
- Recent alternative architectures based on Transformers have become competitive with CNN.

ImageNet Large Scale Visual Recognition Challenge (ILSVRC, 2010-2017)

- ImageNet subset with 1K non-overlapping categories, 1.28M training, 50K validation and 100K test images.
- Key milestones:
  - AlexNet (2012, deep breakthrough).
  - VGG, GoogleLeNet, ResNet (advanced depth and accuracy).
- The model outputs a ranked list of 1K class probabilities for each image (output of softmax layer).
  - If the correct label is within the top-5 predictions, it counts as correct.
  - Top-5 accuracy = $\frac{\text{\# images where true label} \in \text{top-5 predictions}}{\text{total images}}$.
  - Top-5 error rate = 1 − Top-5 accuracy, 5% for human, 2.3% in 2017 for a CV system.
    - Superhuman performance maybe because humans cannot distinguish specific classes [Bishop].
- Further developments:
  - Models have been evaluated on noisy and blurred images [2017].
  - Slightly different dataset has been proposed (eg ImageNetV2) [2019].
  - Top-1 accuracy = $\frac{\text{\# number of correct predictions}}{\text{total images}}$ [2023].
    - ImageNet still drives progress to date, but Top-1 accuracy is stagnating.

Notable CNNs

- LeNet (one of earliest CNN models) [LeCun et al., 1989].
  - Two convolutional layers, pooling and fully connected layers.
  - Used for digit recognition tasks (MNIST dataset).
- **AlexNet** [Krizhevsky, Sutskever & Hinton, 2021]:

    - Won the 2012 competition with top-5 error rate of 15.3%.
    - Use of the ReLU activation function.
    - Application of GPUs to train the network.
    - Use of dropout regularization.
- VGG-16 model [2014].
- ResNet family.
- Inception Family (GoogLeNet, V2, V3).

1.5.1.3 Convolution layers

- Hidden units are connected to a **portion** of the image, the unit's **receptive field**.
    - Instead of linearizing the image, and loosing spatial information.
- Weight vector: **filter** (or **kernel**) that determines the kind of features for which the unit activates the most.
    - Unit activation: $z = \text{ReLU}(\mathbf{w}^T\mathbf{x} + w_0)$.
        - **w** and **x** are vector's representation of pixel values and weights, while w_0 the bias.
- **Feature maps**:
    - Nearby units are connected to adjacent regions of the image with the same weight vector.
        - Detect the **same feature** in a different area of the image.
        - **Weight sharing** (all units share the same weights).
    - Units of the hidden layer form a **feature map**.
        - All units of a feature map detect the **same feature** in a different area of the image.
    - The filter goes over the image and determines the activation of the associated units in the FM.
- Convolutional layers contain a **set of feature maps**, each with a different filter.
    - eg Examples of convolution filters: identity, edge detection, sharpen, Gaussian blur.

Properties of convolution layers

- **Sparse weights**.
- **Parameter (weight) sharing**.
- **Locality-based analysis**.
- **Equivaraince to translation**.
    - A feature detected in a position can be detected in another position never seen before by means of the same filter.
    - It will activate a unit in the same layer but possibly in a different location.
        - It will activate a nearby unit in the same feature map.
    - **Translation equivariance ≠ transaltion invariance**.
        - At classification level, particular object should be assigned the same classification irrespective of position.
        - **Scale invariance**: changes to object size should also leave its classification unchanged.
        - For MLP to learn invariance one would need to consider huge datasets (like by using data augmentation).

Convolution layers hyperparameters

- **Kernel size** (eg 3×3, usually odd numbers).
    - Larger kernel size → smaller feature map.
- **Stride**: determines how far the filter moves across the input.
    - The filter is moved in steps of size S.
    - Larger stride → smaller feature map.
- **Padding**: add extra pixels around the edge of the input to control output size.
    - So the feature map can have the same dimensions as the original image.
    - Larger strides reduce the output size, while padding preserves more information.
- **Convolutional filters number** (of feature maps in a convolutional layer).

Filters

- In classical image processing, kernels' weights were **handcrafted** by experts.
    - Instead, CNNs **learn the best weights** (via backpropagation).
- Learned filters are much less neat and interpretable than the examples made by humans.
- **Multidimensional convolution**:
    - A multi-dimensional filter takes input from across the R, G and B channels.
        - eg A 3×3 filter has 27 weights (plus a bias) and can be visualized as a 3×3 tensor.

- This holds for the first level of the kernels.
  - For higher level filters the number of input channel is in general higher than 3.
- Filters in deeper levels cannot be visualized directly.
  - The input that maximizes activation for all neurons of a given FM can be visualized.
  - **Saliency maps**:
    - Used to identify those regions of an image most significant in determining the class label.
- Multiple independent filter lead to multiple feature maps, forming a single **convolutional layer**.
  - The filter operates on a portion of the input volume.
  - Each slice of neurons (same depth) denotes a feature map.
    - Weights are shared in a feature map.
    - Neurons in the same FM process different portions of the input volume in the **same way**.
  - Each feature map can be seen as the result of a specific filtering of input.

1.5.1.4 Rectified Linear Unit

- MLP historically used the sigmoid activation function.
  - In deep network, its use is problematic for back propagation (**vanishing gradient**).
  - Sigmoid for large and small values derivatives is close to 0.
  - In weight update: $\delta^l = ((\mathbf{W}^{l+1})^T \delta^{l+1}) \odot \sigma'(\mathbf{a}^l)$.
    - If σ' is close to 0 the product is very small → vanishing gradient.
- `def` **Rectified linear unit activation function** ($ReLU$): $f(u) = \max(0, u)$.
  - The derivative is 0 for negative values, and 1 for positive values.
  - It introduces non-linearity to the model.
  - This leads to sparse activations (part of the neurons are off).
    - Which leads to simpler activation patterns, which makes it **more generalizable**.
  - Unit acts as a **feature detector** that signals when it finds a sufficiently good match to its kernel.

1.5.1.5 Pooling

- A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby output.
  - The **max pooling** operation reports the maximum output within a rectangular neighborhood.
  - The **average pooling** operation reports the average output within a rectangular neighborhood.
- It **reduces dimensionality** (downsampling).
- It selects the **most informative values**.
- It implements **invariance wrt (small) translation**.
- Pooling layers have no weights.
  - Pooling does not have learnable parameters but only hyperparameters (fixed with the architecture).
  - `eg` A standard choice is max-pooling over 2×2 kernels with stride 2.

1.5.1.6 Cost functions

- For classification with CNN, common CFs are: **Soft-Max** and **Cross-Entropy**.
- `def` **Soft-Max**: $z_k = f(v_k) = \frac{e^{v_k}}{\sum_{c=1,\dots,s} e^{v_c}}$.
  - It consist of a final layer of s neurons (one for each class) fully connected to the previous layer.
  - The neuron inputs v_k are calculated as usual, but the activation function for the k-th neuron is the one above.
  - Values z_k can be viewed as probabilities: they belong to $[0, 1]$ and their sum is 1.
- `def` **Cross-Entropy loss**: $-\sum_{i=1}^{S} t_i \cdot \log(p_i)$.
  - It measures how much the predicted distribution p **differs** from the desired output.
  - t_i is the desired output for neuron i, while p_i is the predicted value.
  - Functioning:
    - If the model predicts $p_j \approx 1$ for the correct class, the prediction is very close to the truth.
      - $\log(1) = 0 \rightarrow$ low cross-entropy.
    - If the model predicts $p_j \ll 1$ (possibly $p_j = 0$) for the correct class, the prediction is very wrong.
      - $\log(0) = -\infty \rightarrow$ high cross-entropy.
  - Backpropagation is then applied with this loss function.
  - Cross entropy between two discrete distributions: $H(p, q) = -\sum_v p(v) \cdot \log(q(v))$.
    - It measures how much q **differs** from p.
- By using Soft-Max and Cross-entropy, BP involves calculating gradients for both the output and hidden units.

1.5.1.7 Avoiding overfitting

- **Dropout regularization**:
  - During training, some neurons are randomly ignored (*dropped out*) with a given probability p.
  - It speeds up the training process.
    - More training iterations needed, but those will be faster.
  - Encourages **sparsity**, thus enhances generalization and avoids overfitting.
  - Typically implemented as a *layer* that zeroes the previous neurons' output with probability p.
    - Placed after **activation functions**.
- **Early stopping**:
  - Monitors the model's performance on a **validation set** during training.
    - Stop the training when performance starts to degrade (indicating overfitting).
  - A way to keep the model from continuing to learn noise in the Tr after the optimal point.
- **Data augmentation**:
  - Not a regularization technique in the traditional sense.
    - It artificially increases the size of the training dataset by applying random transformation to Tr images.
  - It helps the model generalize better by exposing it to variation to be found in the real world.
- **Batch normalization**:
  - It normalizes the output of each layer.
  - Which can reduce overfitting by introducing noise to the training process.
- **Weight decay**:
  - Adds a penalty to the loss function that encourages weights to be small, thus preventing overfitting.

1.5.1.8 Neuroscientific basis of CNN

- In the mammalian visual cortex, there are simple neurons that detect simple features.
  - Hubel & Wiesel paved the way in this research by analyzing cat's striate cortex.
- Several studies tried to establish commonalities between CNN and brain processing.

1.5.2 Transfer learning

- The training of complex CNN on large dataset can require a lot of machine time (even on GPUs).
  - One trained, the classification of a new pattern is generally fast (eg 10-100ms).
- **Fine-tuning**:
  - Start with a pre-trained network trained on a similar problem.
  - The output layer is replaced with a new layer of Soft-Max neurons (adapting classes number).
  - As initial values for weights those of the pre-trained network are used.
    - Except for connections between penultimate and final layers whose weights are initialized random.
  - New training iteration with SDG are carried out to optimize weights.
    - To respect to the peculiarities of the new dataset (not necessary to be of large size).
  - eg Examples of fine-tuning:
    - A network trained on object classification → lesion classification (**scarcer dataset**).
- **Feature reusing**:
  - A pre-trained NN is used without further fine-tuning.
  - Generated features by the NN during the forward step on the new dataset are extracted (at intermediate layers).
  - These features are used to train an external classifier.
    - To classify the patterns belonging to the new application domain.

1.6 Recurrent neural networks

1.6.1 Recurrent neural networks

- Specialized to work on **sequences**:
  - Speech recognition, music generation, DNA analysis, sentiment classification, translation, NER, LM, etc.
  - The input and the output can have same or different dimensions.
  - Standard NN aren't well suited for seq-to-seq tasks.

        - Inputs and outputs can be different lengths in different examples.
        - Doesn't share features learned across different positions of text.
        - **Context** must be taken into account.
- **Elman networks** [1990]:
    - Precursor of simple RNN, built for cognitive purposes.
    - Introduced to explain how infants could learn to identify words from a continuous stream of phonemes.
        - `eg` «manyyearsagoaboyandgirllivedbytheseatheyplayedhappily».
    - It showed how this network can induce categorical grammar (noun/verb) of input sentences with no previous knowledge.
        - Just by learning it from example.
    - Architecture:
        - Input units.
        - Hidden units.
            - The output of HU is fed to output units but also to **context units**.
            - At the next iteration, the new input (`eg` next word in the sentence) and previously stored context units are fed to the current hidden units (**recursion**).
        - Output units.
- At the $i+1$-th time step, the i-th activation is taken into account.
    - Unrolled vs Rolled representation (the latter represent the recurrent component).
        - The rolled representation highlights that the **weights are shared**.
    - `def` **RNN Activation**: $a^{\langle i \rangle} = g(W_{ax} * x^{\langle i \rangle} + b_a + W_{aa} * a^{\langle i-1 \rangle})$ and $\hat{y}^{\langle i \rangle} = g'(W_{ya}\, a^{\langle i \rangle} + b_y)$
        - $a^{\langle 0 \rangle}$: zero vector.

1.6.1.1 Training a RNN

- Training is performed using **Backpropagation Through Time** (BPTT).
    - BPTT begins by unfolding a recurrent neural network in time.
        - The unfolded network contains k inputs and outputs.
        - But every copy of the network shares the same parameters.
    - Then, BP is used to find the gradient of the loss function wrt all the network parameters.
- Problems:
    - **Vanishing** or **exploding gradient** (technical problem).
    - Not very good at capturing **long distance dependencies**.

1.6.2 Long Short Term Memory

- To capture long distance dependencies, a sort of **memory** is needed.
- A **LSTM cell** substitutes the hidden layer of a RNN.
    - Each cell receive an input from the example and the activation of the previous LSTM cell (a la RNN).
    - Units have **gates** that decide when to update memory cell states, what to keep, what to forget.
- Three type of gate: **forget**, **input** and **output gate**.
- Applications:
    - Speech recognition, machine translation, syntactic parsing, handwriting recognition, image captioning.
- LSTM can be **stacked** too (deep LSTM).
    - The output of the LSTM below is used for the upper one.

1.6.2.1 LSTM architecture

- **Cell state**:
    - The horizontal line running throught the top of the diagram.
    - It transports informations down the entire chain, with only some minor linear interactions.
    - LSTM have the ability to remove or add information to the cell state, carefully regulated by **gates**.
- `def` **LSTM Forget gate**: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$.
    - What information will be eliminated from the cell state.
    - Given h_{t-1} and x_t, f_t is a vector of values between 1 and 0 (with same dimension as C_{t-1}).
    - Values of f_t are then multiplied element-wise with C_{t-1}.
        - Thus gating what values of C_{t-1} are kept, what are thrown away.
- `def` **LSTM Input gate**: $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$.
    - What information should be added (`eg` information on new subject).

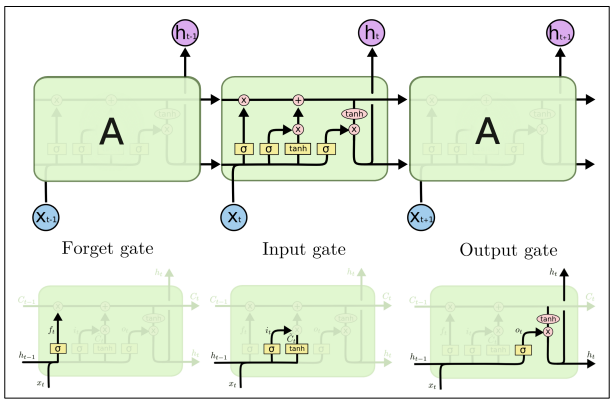

Figure 5: LSTM architecture with corresponding gates.

- Input gate is multiplied with **candidate cell state** $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$.
  - And the result is then added to the previous cell state.
- As a result of previous operations by forget and input gate, a **new cell state** is computed.
  - $C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$.
- `def` **LSTM Output gate**: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$.
  - Which part of the cell state will be kept.
  - From the new cell state C_t, the **output** h_t is computed.
    - $h_t = o_t \cdot \tanh(C_t)$.
    - o_t is applied to cell state C_t to really compute (filter) what to output.

1.7 Cognitive Assessment

1.7.1 NN and cognition

- Three different approaches:
  - **ANN as cognitive models**.
    - Cognitive models of how humans might learn language or analyze visual information.
    - First NN for language aimed to be cognitively plausible models of language learning in infants.
      - `eg` Elman's first RNN, Rumelhart and McCelland's PDP model.
      - Tasks: learn word-meaning associations, learn past-tense, etc.
    - Perceptron aimed to model simple visual cells in the mammalian cortex.
      - CNN are considered as good models of visual information is processed in the brain.
  - **Cognitive assessment of NN models**.
    - Use tools of CogSci and linguistics to probe models' performance and compare to human performance.
    - Contemporary NN are focuses to application with looser relation to cognitive plausibility.
      - However, cognitive tools ave been used to assess NN performance.
  - **ANN as models to understand the brain**.
    - Several works to predict and describe activity in areas of the brain using CNNs (and family).
      - Di Carlo's claim: CNN are a good model of how visual information is processed in the brain ($V1 \rightarrow V2 \rightarrow$ PIT ...).
    - Learning how the brain processes visual information can also improve CNNs.
      - `eg` CNN models with a neural hidden layer that better match primary visual cortex (V1) are also more robust to adversarial attack.

1.7.2 ANN as cognitive models

1.7.2.1 Interactive Activation model [Rumelhart & MCCelland, 1981-82]

- A possible mechanistic explanation of **context effect**.
  - People are faster and more accurate at identifying the O in real word *SPOT* vs. *TKOR*.
  - This suggest that humans use orthographic rules (not just familiarity).
- Question: do human represent these rules in their minds and use them in the service of perceiving letters.
  - No, humans are not using orthographic rules when reading.

- People simply process visual input using the **interactive activation model**, which explains the context effect.
  - Pipeline: Visual input → feature level ↔ letter level ↔ word level.
  - Levels have self-loops and feed information back to the level underneath.

1.7.2.2 Debate between nativism and empiricism

- **Nativism** (Chomsky, Fodor):
  - The mind contains **innate**, **domain-specific structures** (eg Universal grammar).
  - Learning is impossible without rich built-in constraints.
  - *Poverty of stimulus*: input is insufficient, innate knowledge must fill the gaps.
- **Empiricism** (English tradition as in Lock and Hume, now the *USA East coast*):
  - The mind begins with **few domain-specific structures**.
  - Rich learning mechanisms extract structure from experience.
  - Statistical regularities in input are sufficient for acquisition.
- **Connectionism** (Rumelhart, McCelland, Elman):
  - **No built-in rules** or domain-specific constraints.
  - Structures **emerge** from exposure to data (eg past tense learning, word segmentation, phonotactics).
  - Connectionist models were historically developed *as a challenge* to nativist claims.
  - eg **Elman's (recurrent) neural network**:
    - Task: predict the next letter in a sequence of letters.
      - The network was presented with one letter at a time.
      - It takes into account the context in order to make a prediction via the **context layer**.
      - The network learns alone word boundaries.
        - Prediction error accuracy is somehow a **proxy of word boundaries**.
        - Inside a single word the error becomes lower, and it increases when a new word begin.
    - Task: predict the next word in a sequence of words.
      - By analysing inner representation, words can be clustered according to their similarities into **syntactic categories**.
      - The network learns to form a inner representation for verbs and one for nouns.
    - This is a contribution to the empiricist view.
      - No previous knowledge of syntactic structures needed.

1.7.3 Cognitive assessment of NN models

1.7.3.1 Assessing the Ability of LSTM to Learn Syntax-Sensitive Dependencies [Linzen, Dupoux, Goldberd, 2016]

- **Cognitive science as a tool to evaluate NN models**.
- Objectives of the paper:
  - Stress test for LSTM (performance side):
    - Success in language processing is typically attributed to theirs ability to capture **long-distance statistical regularities**.
  - On the cognitive side:
    - Limits of RNN as model of (child) language learning (as in Elman, 1990-91).
      - **There are rules that infants learn but LSTM don't learn**.
      - Therefore either LSTM is not the right model, or there is innate knowledge (authors more on the nativist side).

Structured representations

- Arguments for **structured representations**:
  - Many word co-occurring statistics can be captured by treating sentences as an unstructured list of words.
    - Most naturally occurring agreement cases in the Wikipedia (training corpus) are easy.
      - Solvable without syntactic information, based only on the sequence of nouns preceding the verb.
    - Other dependencies however are **sensitive to the syntactic structure** of a sentence.
    - Given the difficulty in identifying the subject from the linear sequence of the sentence.
      - Dependencies such as subject-verb agreement serve as an argument for structured syntactic representations in human.
  - But LSTM don't use any type of structured representations.

- Sadly (compared to Elman's results), LSTMs don't behave well on complex sentence with several and far apart attractors.

Task considered

- **Task considered**: **subject-verb agreement**.
- Training set: derived from Wikipedia.
- Three different training conditions:
  - Most favorable condition:
    - Training with explicit supervision directly on the task of guessing the number of a verb based on the words that preceded it.
    - Error rates of LSTM evolve as a **function of distance** (with no intervening nouns).
  - Grammatically judgment training objective.
    - Full sentences annotated as to whether or not they violate subject-verb number agreement.
  - Training of a model without any grammatical supervision.
    - Instead using a *language modeling* objective (next word prediction).

Results and improvements

- Results (number prediction task):
  - Most naturally occurring agreement cases in Wikipedia are easy.
    - And can be solved without syntactic information.
    - This lead to high overall accuracy in all models.
  - The accuracy of supervised model is lower on **harder cases**, even if it managed to recover.
    - More mistakes when no overt cues to syntactic structure (function words) are available.
- Related tasks:
  - Verb inflection (eg write/writes, access to semantics too).
  - Grammaticality judgments (weaker supervision, no syntactic clues boundaries).
  - Language model (worst performer of all).
- Main theoretical results:
  - LSTM make errors on **harder sentences** (mostly if trained in the language model setting).
    - It highlights the limitation of LSTMs that wouldn't be apparent when **analysing average performance**.
  - Therefore, **explicit supervision** is necessary for learning the agreement dependency using LSTM.
    - **Limiting its plausibility** as a model of child language acquisition (similarly to Elman, 1990, for RNNs).
- Some (not general) single hidden units were found to be specialized in detecting singular or plural.
- Possible improvements:
  - LSTM can be encouraged to develop more sophisticated generalizations by **oversampling** grammatically challenging training sentences.
  - Language modeling objective is not by itself sufficient for learning structure-sensitive dependencies.
    - **Joint training objective** (with supervisions as in SVA task) can be used.
      - To supplements LM on tasks for which syntax-sensitive dependencies are important.

Other neural architectures

- In following papers by different authors, newer neural architectures were tested.
- NN based on Transformers perform well on syntactic tasks (eg subject-verb agreement).
  - However, they may be *right for the wrong reasons*.
    - Learning wrong heuristics, learning statistical approximations, etc.
- *Embers of autoregression* [McCoy et al., 2024]:
  - LLM learn to predict the next words, and this has its advantages but also disadvantages.
    - They displays sensitivity to **output probability**, **input probability** and **task frequency**.
  - Even state-of-the-art models tends to fail on **lower probability sentences**.
  - If the rule is well defined, they don't infer the underlying rule, they learn a **statistical approximation**.
- **Argument of the Poverty of the stimulus** re-adapted:
  - Today's best LLMs are trained on vastly more data than a child is exposed to.
    - Estimation: less than $100M$ words for a 12yo human.
  - Hence, the models will necessarily be un-human-like (a la Chomsky).
  - *BabyLM challenge*: to train language models in low-resource data settings.
    - Where amount of linguistic input resembles the amount received by human language learners.

- Find the best architecture who perform best given the same initial conditions.
- Another objective of the challenge is to democratize research.

1.7.3.2 Humans learn proactively in ways language don't [Han & McCelland, 2025]

- LLMs now match or exceed average human performance.
  - Maybe LLMs may soon become genuine examples of general intelligence or unified theories of human cognition.
  - This paper tries to answer these grandiose claims that appear in the literature.
- Task considered (for humans and LLMs): **chain rank task**.
  - **Train phase**:
    - Participants learn about this chain from individual sentences describing the relations of neighboring pairs of entities.
    - The participant's goal is to learn whether the correct completion for each sentence is *true* or *false*.
    - This allows the CRT to be presented to humans and LLMs with maximally similar inputs and outputs.
    - eg Five entities (persons: Kennedy, Roosevelt, Lincoln, Einstein and Presley), considered as pairs.
      - The relation is *better at chess, worse at chess*.
  - **Test phase**:
    - Participants are required to make judgments of entity pairs from the underlying chain, including:
      - The original learning items.
      - New items that requires participants to make either:
        - **Reverse generalizations**: $A > B \rightarrow B < A$.
        - **Transitive generalization**: $A > B \wedge B > C \rightarrow A > C$.
- Performed on LoRA, ICL and ICL+CoT (aligned or unaligned).

Results

- LLM **learns to generalize much more slowly** than humans.
- While on learned relationships LLMs are optimal, they are worse in reverse and n-hop generalization.
- Via a post-experimentation form provided to all participants, it emerges:
  - Humans rely on one of two strategies for representing and reasoning about the learning items:
    - *Ranker strategy*:
      - Actively recording the learning items into a single representation of entity ranking.
      - Ranking that can be recalled for any test item.
    - *Non-ranker strategy*:
      - Memorizing each learning item independently.
      - Recalling only relevant learning items for each item.
  - During the testing stage, rankers achieved higher accuracy but also make faster judgments.
- → Stronger performance is characterized by the **proactive formation of efficient representations that aid learning and generalization**.
  - Thesis: human intelligence is explained by **proactivity** and **curiosity** (and not only memorization).

1.7.3.3 Atoms of recognition in human and computer vision [Ullman, Assif, Fataya, Harari, 2016]

- CNN have been proposed several times as a very close model to how human brain process visual information.
  - Goal: see how far this analogy can go.
- Central problem in the study of human vision:
  - Discovering **visual features** and **representation** used by the brain to recognize object.
  - NN models rival human performance on image classification.
    - Investigates commonalities in representation and learning process between current NN and human visual system.
- **Minimal recognizable configurations** (MIRC).
  - Small portion of images from which a human can recognize the subject of the picture.
  - By rendering smaller (sub-MIRCs), no recognition is possible.
  - Since they are small, they **reveal features** that are important in recognition.
    - The role of features is revealed uniquely at the minimal level (essential contribution).
- Human and networks models are tested to check if they **perform in the same way**.
  - If the human vision uses the same features to understand the content of an image.

Results

- In humans, better recognition of MIRC than in network models.
- In humans, **sharp decrease between MIRC and sub-MIRC recognition**.
  - Not in neural networks models.
  - All MIRC features are necessary for human recognition.
- Networks models:
  - They are not good at recognizing minimal images.
  - No decrease between MIRC and sub-MIRC.
- Therefore **different features are used (in different ways)** by humans and NN.

Internal interpretation

- Additional limitation is the ability to perform a detail internal interpretation of MIRC.
- Human can consistently recognize multiple components internal to the MIRC.
- Such internal interpretation is beyond the capacities of current NN.
  - It can contribute to accurate recognition.
  - Because a false detection could be rejected if it does not have **expected internal interpretation**.

Feed forward and top-down processes

- Features are used in the visual system as a part of the cortical feed-forward process or by a top-down process.
  - Top-down process is currently missing from the purely feed-forward computational model.
  - While in human, there is a **combination of bottom-up and top-down processes**.
- Top-down processes are likely involved:
  - Detailed interpretation appears to require features and interrelations that are class-specific.
- Two main stages:
  - Initial activation of class candidates, which is incomplete and with limited accuracy.
  - The activated representations then trigger the application of class-specific interpretation and validation processes.
    - Which recover richer and more accurate interpretation of the visible scene.

1.8 Representation learning

1.8.1 Representations

- **Representation learning** underpins all current developments in DL.
  - It enables the sharing of statistical power across tasks.
  - Particularly beneficial when dealing with multiple modalities or domains.
  - Also facilitate **knowledge transfer** to tasks with few/zero examples are available, but a task representation exists.
  - Changing representation can make a problem very difficult or very easy.
- A **good representation** is one that makes a subsequent learning task easy.
  - FFN training by supervised learning can be seen as performing a sort of representation learning.
    - Most part of the network is used to **learn better representations** (more and more abstract).
    - And then a simple classifier is used at the top of the network.
- Learning objective can be tailored to force the representation to have some nice properties.
  - Most representation learning problems face a trade-off between:
    - Preserving as much information about the input as possible.
    - Attaining nice properties (eg *independence* of the features).

1.8.1.1 Properties of good representations

- An ideal representation is one in which the **features** within the representation correspond to the **underlying causes** of the observed data.
- Two key properties: they are often **distributed** and **disentangled**.
- Local vs. distributed representations:
  - A **local representation** uses a single, discrete value for each concept (eg one-hot encoding).
    - Problem: all concepts are equally distant (no similarity is learnt).
      - This representation is **sparse** and **inefficient**.
  - A **distributed presentation** describes each concept using multiple and continuous features.
    - Each feature is involved in representing many concept.
    - It creates a **rich similarity space** (semantically close concepts are close in distance).
    - -ex DR help in **representing an exponential number of regions using** a linear number of parameters.
      - This doesn't automatically translate in an exponential advantage for the classification algorithm.
      - The VC dimension of a NN of linear threshold units is only $O(w \log w)$ (w number of weights).
        - The **Vapnik-Chervonenkis** dimension is a measure of the capacity of a model (hypothesis) space.
        - It is used to deduce bounds on the generalization capabilities of ML.
        - Keeping the VC dimension of the hypothesis space small is usually a sensible goal.
      - Despite the representation being able to distinguish between many zones, not all zones can be represented.
    - DR can be **interpretable**, but this is **not the norm**.
- In a **disentangled representation** single features (or single features directions in the feature space) correspond to **single distinct factors** of variation in the data.
  - It allows to reason about and manipulate high-level concepts independently.

1.8.2 Transfer Learning and Domain Adaptation

- TL: Where what has been learned in one setting (distribution P_1) is **reused** to improve generalization in another setting (P_2).
  - In domain adaptation, the task is the same but, eg, the data distribution may differ.
- Generalization of the idea of greedy PT.
  - Where the transferred representations were between unsupervised and supervised tasks.

1.8.2.1 Transfer Learning

- The learner must perform two or more different tasks.
  - But it is assumed that many of the factors that explain variations in P_1 are relevant in P_2.
- **Depth** seems to be crucial in this process.

    - As an architecture uses deeper representations, the learning curve on new categories of transfer setting P_2 becomes much better.
- eg Image recognition on cancerous cells.
    - Strategy: fine-tune a general model (like ImageNet) on smaller medical dataset.
- TL and DA assume there is a common **(shared) representation** that explain variations in the input data.
    - These variation are then adapted to the various tasks by different output layers.
        - eg Image recognition.
    - The shared part is the initial part of the network.
- But there are situations, where what is shared between different tasks is not the semantics of the input.
    - But the semantics of the output and inputs need to be adapted to be compatible with that.
    - The shared part is the last part of the network.
    - eg Speech recognition.

1.8.2.2 Domain adaptation

- The task remains the same between each setting, but the input distribution is slightly different.
    - eg Sentiment analysis (from movie reviews to product reviews).
- Create a representation that captures the underlying patterns in the data while **ignoring domain-specific nuances**.
- *Unsupervised Domain Adaptation by Backpropagation* [Ganin & Lempitsky, 2015]:
    - The source is MNIST while the target is MNIST-M (MNIST with different background).
    - A part of the network is used to learn a shared representation.
    - To this shared representation two classification heads are attached:
        - The first one classify digits.
        - The second one classify domains.
    - During training, each time the second head succeeds, the shared network is penalized.
        - The gradient is inverted (**gradient ascent**).

1.8.2.3 Concept drift

- When the concept underlying the data distribution shifts in time.
    - A model learnt at a given point in time need, then, to be updated to take into account the drift.
- In this case too, data from a given setting (previous than the shift) is exploited to get an advantage in the new setting.
- The representation part is not changed, the classification part is.

1.8.2.4 One-shot and zero-shot learning

- **One-shot learning**:
    - An extreme case of transfer learning, where only a single labeled example is given for the new setting.
    - An already-learnt representation (eg a manifold of known class) is used.
- **Zero-shot learning**:
    - One needs to adapt to a new setting without seeing any labelled example.
    - It can be seen as including three random variables:
        - The traditional input $\mathbf{x}$.
        - The traditional outputs $\mathbf{y}$.
        - An additional random variable describing the task T.
    - The model is trained to model $p(\mathbf{y} \mid \mathbf{x}, T)$.

1.8.3 Greedy layer-wise Unsupervised Pre-training

- Layers are trained in an **unsupervised** way (supervised ways also available).
- Supposed to be **only a first step before** joint **training** starts.
- Instrumental to the revival of deep NN.
    - Classical deep nets had trouble in propagating information.
    - Specifically, deep networks had problems with **vanishing** and **exploding** gradients.
    - *Solution*: break up the training into the training of smaller networks.
- Main idea:
    - Each layer is trained using **unsupervised** learning, taking the output of the previous layers.
    - And producing as output a new representation of the data, whose distribution is hopefully simpler/better.
    - Most often the procedure relies on a single-layer representation learning.

```
def greedy_layer_wise_unsupervised_pretraining(X,y)

    f = lambda z: z        # identity function
    data = X               # repr 0

    for k in range(m):
        f_k = UnsupervisedLearn(data)
        f = lambda x: f_k(f(x))
        data = f_k(data)   # repr k+1

    if fine_tuning:
        f = FineTune(f, X, y)

    return f
```

Figure 6: Greedy layer-wise Unsupervised Pre-training pseudo-code.

    - eg RBM, single layer autoencoders, other models that learn latent representations.
  - y is used only in the **fine-tuning** phase.
- In 2006 it was shown that it could be used to find **good initialization** to train deep architectures.
  - Nowadays, other techniques are components are used (ReLU, He, Xavier initialization, Adam dropouts, etc).
  - Nowadays, unsupervised is largely abandoned, except in the field of NLP.
    - DL techniques based on supervised learning, regularized with dropout or batch normalization usually outperform UPT.
    - In many fields (NLP and CV), **self-supervised learning** (kind of UPT) is widely used.
  - But this approach was the initial demonstration that such training was indeed achievable.
- Variants:
  - Can also be used as initialization for deep unsupervised models.
  - Greedy layer-wise Supervised Pre-training.
  - Simultaneous supervised and unsupervised learning.
    - Allows incorporating the constraints imposed by the output layer from the outset.

1.8.3.1 Main ideas beneath Unsupervised Pre-training

- UPT acted as a **powerful regularizer** and provided much better **initialization** for network weights.
- Two main ideas are important to understand why and when unsupervised pre-training work:
  - The **choice of initial parameters** can have a significant **regularizing effect** on the model.
    - Initially it was assumed that this would result in approaching a local minimum.
    - Training is longer without pre-training.
    - PT and non-PT models start and stay in different regions of function spaces.
    - All trajectories of a given type initially move together and then diverge.
      - This suggests that each trajectory moves into a different apparent local minimum.
    - Nowadays **local minima** are no longer considered a serious problem.
      - Critical points are more likely **saddle points** rather than spurious local minima.
      - Local minima **concentrate near the global optimum**.
  - **Learning about input distribution** can help to learn the mapping from inputs to outputs.
    - In many tasks, features learned in unsupervised stage can be useful also in supervised stage.
    - Unsupervised pre-training often helps when the **original representation is poor**.
      - eg Learning word embeddings instead of using one-hot vector.

1.8.3.2 Advantages and disadvantages of Unsupervised Pre-training

- Advantages:
  - It is useful to view unsupervised PT as *regularizer*:
    - When the number of examples is small.
    - When the number of unlabeled examples is large.
    - When the **function to be learned is extremely complicated**.
  - In contrast to other regularizers, UPT **doesn't force the learnt function to be simple**.
    - Rather it helps in discovering feature functions that are helpful in the unsupervised task.
    - UPT can be a more appropriate regularizer if the true underlying functions are:
      - *Complicated* **and** *shaped by regularities* of the input distribution.
- Disadvantages:
  - As a regularization technique, UPT has the problem that it is **difficult to calibrate**.

        - In most regularization techniques, there is a single parameter to set regularization strength.
    - With unsupervised training, either the network is initialized using PT or it is not.
    - The hyper-parameters of the PT phase needs to be adjusted and this can be extremely slow.

1.8.4 Self-supervised learning (SSL)

- The modern successor to UPT, it learns from unlabeled data by creating a **pretext** task.
    - Pretext: a reason given in justification of a course of action that is not the real reason.
- Core idea: hide/change **some part of the input** and train the network to predict the original input.
    - The supervision signal comes from the data itself, not from human labels.
    - This forces the model to learn a rich semantic representation of the data.
- `eg` Pretext tasks in Vision:
    - Image inpainting: predict a missing patch of an image.
    - Colorization: predict the color version of a greyscale image.
    - Jigsaw puzzle: learn to assemble shuffled image patches correctly.
- `eg` Pretext tasks in Language:
    - **Masked Language modeling (MLM)**: predict a randomly masked word in a sentence.
        - The core idea behind models like BERT.

1.8.4.1 SSL via contrastive learning

- A dominant family of SSL methods where the goal is to learn an embedding space where:
    - Semantically **similar** examples are pulled **together**.
    - Semantically **dissimilar** examples are pushed **apart**.
- Procedure:
    - Take an example (the *anchor*).
    - Create two different random augmentation of it, resulting in a **positive pair**.
    - Take examples from the rest of the batch, the **negative examples**.
    - Train the model to:
        - Maximize the similarity of the positive pair's representations.
        - Minimize the similarity to all negative examples.
- `def` **Information Noise Contrastive Estimation Losss**: $\mathcal{L}_{\text{InfoNCE}} = -\log \frac{\exp(\text{sim}(z_i, z_i^+)/\tau)}{\sum_j \exp(\text{sim}(z_i, z_j)/\tau)}$.
    - Key contrastive objective where:
        - sim is cosine similarity, with temperature τ which sharpens similarity scores.
        - (z_i, z_i^+) is the positive pair, all the other z_j in batch are negatives.
    - Goal: maximize similarity of positives, minimize similarity to negatives.
- CL requires **negative samples** to avoid the trivial collapse where all representation become identical.
    - Positive pairs suffice to introduce a notion of similarity.
        - But **many negatives** are crucial to **define dissimilarity and to prevent collapse**.
    - Negatives encourage the model to **spread representations apart**.
        - So that different inputs occupy distinct regions of the embedding space.

CL systems

- **SimCLR**:
    - It popularized contrastive learning in vision by demonstrating that with the right choices, simple CL can rival supervised learning.
    - Key ideas:
        - **Strong data augmentation** (random crop + resize, color jitter, gaussian blur, solarization).
        - **Projection head**: a small MLP head improves representation quality.
        - **Large batch size** (4096-8192): more negatives → better **contrastive signal**.
    - It showed that very **large batch sizes** (i.e. negative samples) allows for the **best results**.
- **Momentum Contrast** (MoCo):
    - It removes SimCLR's dependency on massive batch sizes.
    - Key ideas:
        - Maintain a **queue** of thousands of negative embeddings.
        - Use a **momentum encoder** to produce consistent keys.
        - Queue provides many negatives without huge GPU memory.
    - Effect: Moco allows contrastive learning with batch size ≈ 256.
    - **Momentum encoder** (ME):

- MoCo requires the ME to be updated slowly so that the produced **feature representations** (keys) **change gradually** over time.
- This stability is crucial because MoCo maintains a queue of **cached key embeddings** generated from past batches.
- If the ME changed too quickly, the embeddings stored in the queue would **become stale**.
  - And no longer consistent with the current model, which would weaken or even break the CL objective.
- The slowly updated ME ensures:
  - Cached key remain compatible with current queries.
  - The queue provides a large, consistent set of negatives.
  - Training remains stable and effective.

1.8.5 Semi Supervised Learning (SemiSL)

- A possibly large amount of examples is given, but only few of them are labelled.
  - The task is to exploit the unlabelled examples to better perform on the supervised task.
- **Causality Hypothesis**: an **ideal representation** is one in which the **features** within the representation correspond to the **underlying causes** of the observed data.
  - It underlies a large deal of research motivated by the idea that disentangling the causal factors in $p(\mathbf{x})$ could be a good step for learning $p(\mathbf{y} \mid \mathbf{x})$.
  - This motivated SemiSL approach.
  - But sometimes this fails since unsupervised learning of $p(\mathbf{x})$ is of no help to learn $p(\mathbf{y} \mid \mathbf{x})$.
    - eg When there is no regularities in the data.
  - Other times $\mathbf{y}$ is among the salient causes of $p(\mathbf{y})$.
    - In these cases learning $p(\mathbf{x})$ can be very useful.
    - eg When there is some regularities underlying the data.

1.8.5.1 Causal factors

- If $\mathbf{h}$ represents all **factors causing** $\mathbf{x}$, and $\mathbf{y}$ is assumed to be related to one of them:
  - The generative process can be conceived as $p(\mathbf{h}, \mathbf{x}) = p(\mathbf{x} \mid \mathbf{h})p(\mathbf{h})$.
    - The data has marginal probability: $p(\mathbf{x}) = \sum_{\mathbf{h}} p(\mathbf{h}, \mathbf{x}) = \sum_{\mathbf{h}} p(\mathbf{x} \mid \mathbf{h})p(\mathbf{h}) = \mathbb{E}_{\mathbf{h}}[p(\mathbf{x} \mid \mathbf{h})]$.
  - The best possible model of $\mathbf{x}$ is the one that uncovers the above *true* structure.
    - With $\mathbf{h}$ as a latent variable that explains the observed variations in $\mathbf{x}$.
- **Most observations are formed by an extremely large number of underlying causes**.
  - The brute force approach of encoding *all* possible factors of variations does not work.
- It is then necessary to decide what to encode into $\mathbf{h}$.
  - To find a strategy to guide the network to keep only the **relevant** part of $\mathbf{h}$.
  - Two main strategy:
    - Use a supervised signal to guide the unsupervised process.
    - Use a much larger $\mathbf{h}$ when using only unsupervised learning.
- Another emerging strategy is to **change the definition of what is salient**.
  - Historically, one would optimize against a fixed criterion (often similar to MSE), but this is problematic.
    - eg MSE applied to pixels of images implicitly specifies that a cause is only relevant if affect these brightness of a large number of pixels.
  - Other definitions of salience are possible.
    - In GAN, it's defined implicitly by a game played by an encoder trying to fool a discriminator.
- **Causal factors are robust**.
  - With different domains or task natures, or temporal non-stationarity, the **causal mechanisms remain invariant**.
  - While the marginal distribution over the underlying causes can change.

1.9 Autoencoders

1.9.1 Autoencoders

- Unsupervised learning tools used to **improve supervised networks**.
  - `eg` Image colorization, increase resolution, image inpainting, machine translation, etc.
- A NN trained to attempt to copy its input to its output via a representation **h** built by its hidden layers.
  - Encoder: $\mathbf{h} = f(\mathbf{x})$.
  - Decoder: $\mathbf{r} = g(\mathbf{h})$.
  - Not expected to faithfully copy every input to its output.
    - Instead, they are forced to prioritize which aspects of the input should be preserved.
- AE are a classic example of **self-supervised learning**, where the **supervised signal** is the input itself.
- Traditionally used for dimensionality reduction or feature learning.
  - Today they are used for **generative models** due to connections established with **latent variable models**.
    - **Latent variable model**:
      - The model has both observed **x** and latent variables **h**.
      - The goal is to learn the distribution $p(\mathbf{x}, \mathbf{h})$.
      - Similar to what happens in representation learning.
- Goal:
  - Only copying the input to the output is not useful.
  - Instead, training the AE should result in **h** taking on useful properties.
  - Approach: force **h** to have a smaller dimension than **x**.

1.9.1.1 Dimensionality reduction

- Dimensionality reduction has been the first motivation to study autoencoders.
- Learning AE that map input to lower dimensional output.
  - Many tasks can be more easily solved, and models of smaller spaces consume less memory and runtime.
  - It often places semantically related examples nearby, helping **generalization**.
- **Semantic hashing**:
  - Store all database entires in a hash table mapping binary code vectors to entry.
  - The hash table allows retrieval of similar elements (they share the same binary code).
    - Swap single bits in the code to search for slightly less similar elements.
  - To produce these binary codes, one typically use sigmoid units in the final layer and train them to saturation.
    - By injecting additive noise just before the sigmoid.

1.9.1.2 Undercomplete Autoencoders

- An autoencoder whose code dimension is less than the input dimension ($len(\mathbf{x}) \gg len(\mathbf{h})$).
- The learning process for UAE is usually the minimization of a loss function, $L(\mathbf{x}, g(f(\mathbf{x}))$.
  - Where L is a loss function penalizing $g(f(\mathbf{x}))$ for being dissimilar from **x** (`eg` MSE).
  - When the decoder is linear and L is MSE, a UAE learn to span the same subspace as PCA.
    - PCA finds the mapping $f(\mathbf{x}) = \arg\min_{\mathbf{h}} \|\mathbf{x} - g(\mathbf{h})\|_2$, imposing g as a linear model.
    - When $L = \|\cdot\|_2$ and the reconstruction layer is modeled by linear units, the two approaches can solve the same problem.
  - Data to be reconstructed is assumed to lives on a linear manifold.
    - The best approach must be determined and why.
      - PCA is easier so when its requirements are granted.
      - An AE is better with a highly non-linear.

1.9.2 Regularized Autoencoders

- The difference between $len(\mathbf{x})$ and $len(\mathbf{h})$ determines how much the AE is forced to learn only the *most relevant* part to the input.
  - If the size of **h** is too big, the AE could simply learn the identity function.
- RAE uses a loss function that encourages the model to have other properties besides I/O copying ability.
- Properties exploited for regularization:
  - Sparsity of the representation → **Sparse autoencoders**.
  - Robustness to noise or missing inputs → **Denoising autoencoders**.

- Smallness of the derivative → **Contractive autoencoders**.

1.9.2.1 Sparse Autoencoders

- The training criterion involves a sparsity penalty $\Omega(\mathbf{h})$ on the code layer $\mathbf{h}$, plus the reconstruction error.
  - $\Omega(\mathbf{h}) = \lambda\|\mathbf{h}\|_1 = \lambda \sum_i |h_i|$.
- `def` **Sparse Autoencoders Loss**: $L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h})$.

Generative models

- Training a SAE, by minimizing reconstruction error plus a sparsity penalty, is approximately equivalent to maximizing the model's likelihood.
  - $\min L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}) \approx \min -\log p_{model}(\mathbf{x})$.
  - Then **sparsity penalty is not just a regularizer**.
    - It naturally arises from **modeling the joint distribution** over data and latent variables when a **Laplace prior** is assumed on the latent factors.
  - Adopting a probabilistic view reveals that **SAE are effectively learning a generative model**.
- **Generative models**: a model that learns the joint distribution $p(\mathbf{x}, \mathbf{y})$ of the data.
  - After the model is trained, it can generate new data points of a given class by sampling from it.
  - Sometimes $\mathbf{y}$ is not an observable variable.
    - $\mathbf{y}$ is then called a **latent variable** and usually denoted by $\mathbf{h}$.
  - GM differ from a **discriminative model**.
    - Since DM learn to approximate the conditional probability $p(\mathbf{y} \mid \mathbf{x})$.
    - A DM cannot therefore generate new data points.
- Given a GM with visible variable $\mathbf{x}$ and latent variables $\mathbf{h}$.
  - $p_{model}(\mathbf{h}, \mathbf{x}) = p_{model}(\mathbf{h})\, p_{model}(\mathbf{x} \mid \mathbf{y})$.
  - The **likelihood** of $\mathbf{x}$ given the model can be computed as the marginalization of the latent variables $\mathbf{h}$.
    - **Likelihood**: $p_{model}(\mathbf{x}) = \sum_{\mathbf{h}} p_{model}(\mathbf{h}, \mathbf{x})$.
    - **Log-Likelihood**: $\log p_{model}(\mathbf{x}) = \log \sum_{\mathbf{h}} p_{model}(\mathbf{h}, \mathbf{x})$.
  - AE can be seen as approximating this sum with a **point estimate for just one highly likely value** for $\mathbf{h}$.
    - By taking the logarithm of $p_{model}(\mathbf{x})$ the log-likelihood of $\mathbf{x}$ under the model is obtained.
      - $\log p_{model}(\mathbf{x}) = \log \sum_{\mathbf{x}} p_{model}(\mathbf{h}, \mathbf{x})$.
    - Given an $\tilde{\mathbf{h}}$ generate by the decoder:
      - $\log p_{mod}(\mathbf{x}) = \log \sum_{\mathbf{h}} p_{mod}(\mathbf{h}, \mathbf{x}) \approx \log p_{mod}(\tilde{\mathbf{h}}, \mathbf{x}) = \log p_{mod}(\tilde{\mathbf{h}}) + \log p_{mod}(\mathbf{x} \mid \tilde{\mathbf{h}})$.

Loss decomposition

- The Loss can be divided in a **reconstruction loss** $(L(\mathbf{x}, g(f(\mathbf{x}))))$ and a **sparsity loss** $(\Omega(\mathbf{h}))$.
- **Reconstruction loss**:
  - By minimizing the reconstruction loss, the AE is learning to approximate the log-likelihood of the data.
    - By minimizing it, the AE is learning to approximate the conditional distribution $p_{model}(\mathbf{x} \mid \mathbf{h})$.
    - It is trying to find the $\mathbf{x}$ that is the best reconstruction given $\mathbf{h}$.
      - That is, solving $\arg\max_{\mathbf{x}} p_{model}(\mathbf{x} \mid \mathbf{h}) = \arg\min_{\mathbf{x}} -\log p_{model}(\mathbf{x} \mid \mathbf{h})$.
        - A minimization problem is preferred for optimization.
        - While logarithms are preferred to avoid numerical issues thanks to their additivity.
- **Sparsity loss**:
  - By setting $\Omega(\mathbf{h}) = \lambda \sum_i |h_i|$ minimizing the sparsity term correspond to maximizing the log-likelihood of the $p(\mathbf{h})$ term assuming a Laplace prior over each component of $\mathbf{h}$ independently.
    - Using the L_1 norm of $\mathbf{h}$ is known to **encourage sparsity**.
    - Since h_i are independent, $p(\mathbf{h}) = \prod P_{model}(h_i)$.
      - By taking the logarithms of it, $\prod$ turns into a $\sum$.
    - If $p_{mod}(h_i) = \frac{\lambda}{2} e^{-\lambda|h_i|}$ then $-\log p_{mod}(\mathbf{h}) = \sum_i (\lambda|h_i| - \log\frac{\lambda}{2}) = \Omega(\mathbf{h}) + \text{const}$.
      - λ can be treated as an hyper-parameter.
      - const, depending only on λ, is not optimized during learning and can be ignored.

Training a SAE

- Training the network correspond to minimize the **negative log-likelihood of the data under the model**.
  - Obtained by the results related to the loss decomposition.

- `def` **Sparse Autoencoder Training**: $\arg\min_\theta[-\log p_{model}(\mathbf{x})] \approx \arg\min[-\log p(\mathbf{x} \mid \tilde{\mathbf{h}}) - \log p(\tilde{\mathbf{h}})] \approx \arg\min_\theta[L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h})]$.
  - The sparsity penalty then is not a regularization term.
    - It's just the consequence of modeling the joint distribution taking into account the latent variables and assuming them to have a Laplace prior.
- This view provides different motivations for training a SAE:
  - It is a way of approximately training a generative model (`eg` to generate new examples).
  - Learnt features are useful since they describe the latent variables that explain the input.

Generating new data with SAE

- In principle, a learned model can be used to generate new data.
  - By sampling from the prior $p_{model}(\mathbf{h})$ and then reconstructing $\mathbf{x}$ using the decoder $g(\mathbf{h})$.
- During the training process, the decoder has only been trained to reconstruct data points close to Tr examples.
  - In practice, by naively sampling from prior, it is likely to get $\mathbf{h}$ values very far from the ones seen during training.
    - Yielding very poor samples.
- SAE can be used to **generate new data points**, but one should **calibrate the sampling process**.
  - To ensure that the sampled $\mathbf{h}$ values are close to the ones seen during training.

1.9.2.2 Denoising Autoencoders

- More in general, one can view both the encoder and the decoder as modeling some distribution.
  - $p_{encoder}(\mathbf{h} \mid \mathbf{x}) = p_{model}(\mathbf{h} \mid \mathbf{x})$.
  - $p_{decoder}(\mathbf{x} \mid \mathbf{h}) = p_{model}(\mathbf{x} \mid \mathbf{h})$.
  - In general, the E and D distribution are not necessarily conditional distributions compatible with a unique joint distribution $p_{model}(\mathbf{x}, \mathbf{h})$.
    - Training E and D as a **denoising autoencoder** will tend to make them compatible asymptotically.
      - With enough capacity and examples.
      - DAE guarantees that those p will be compatible, when previously it wasn't guaranteed.
- Rather than adding a penalty Ω to the cost function, they changes the reconstruction error of the CF.
- `def` **Denoising Autoencoders Loss**: $L(\mathbf{x}, g(f(\tilde{\mathbf{x}}))$.
  - Where $\tilde{\mathbf{x}}$ is a copy of $\mathbf{x}$ that has been corrupted by some form of noise.
  - DAE must therefore undo this corruption rather than simply copying their input.
- It has been shown that denoising training forces f and g to implicitly learn the structure of $p_{data}(\mathbf{x})$.

DAE Stochastic view

- A corruption process $C(\tilde{\mathbf{x}} \mid \mathbf{x})$ that produces corrupted samples is introduced.
- The AE then learn a **reconstruction distribution**.
  - `def` **Reconstruction distribution**: $p_{reconst}(\mathbf{x} \mid \tilde{\mathbf{x}}) = p_{decoder}(\mathbf{x} \mid \mathbf{h}) = g(\mathbf{h})$ and $\mathbf{h} = f(\tilde{\mathbf{x}})$.
- Typically, GD minimization on the negative log likelihood $-\log p_{decoder}(\mathbf{x} \mid \mathbf{h})$ can be performed.
  - As long as the encoder is deterministic, the whole AE can be **trained end-to-end using SGD**.
  - The DAE can be seen as performing SGD on $-\mathbb{E}_{\mathbf{x} \sim p_{data}(\mathbf{x})} \mathbb{E}_{\tilde{\mathbf{x}} \sim C(\tilde{\mathbf{x}} \mid \mathbf{x})} \log p_{decoder}(\mathbf{x} \mid \mathbf{h} = f(\tilde{\mathbf{x}}))$.

Manifolds

- Many ML algorithms exploit the idea that data concentrates around **low-dimensional manifolds**.
  - Or a small set of such manifolds.
  - AE take this idea further and aim to learn the structure of the manifold.
- DAE can be seen as **approximating a vector field**.
  - The corruption process moves the examples away from the lower dimensional manifold.
  - The AE is **learning to project back** these examples to the manifold.
    - The corrupted example is projected to the manifold, obtaining $\mathbf{h}$.
    - h can then be used to obtain the original example.
  - To assume that the lower dimensional manifold is *smooth* has been proved empirically right.
  - The DAE learns a **vector field that provides the fastest path to the manifold**.
- **Manifold tangent planes**:
  - An important characterization of a manifold is the set of its tangent planes.
  - At a point $\mathbf{x}$ on a d-dimensional manifold:

        - The tangent plane is given by d basis vectors that span the local directions of variation allowed on the manifold.
    - These local directions specify how one can **change x infinitesimally while staying on the manifold**.
- AE and manifolds:
    - If the data generating distribution concentrates near a low-dimensional manifold, this yields representation that implicitly capture a local coordinate system for this manifold.
    - Only the variations tangent to the manifold around $\mathbf{x}$ need to correspond to changes in $\mathbf{h} = f(\mathbf{x})$.
    - Hence the encoder learns a mapping from the input space $\mathbf{x}$ to a representation space.
        - A mapping that is **only sensitive to changes along the manifold directions**.
        - But that is **insensitive to changes orthogonal to the manifold**.

1.9.2.3 Contractive Autoencoders

- `def` **Contractive Autoencoders Loss**: $L(\mathbf{x}, g(f(\mathbf{x}))) + \lambda \sum_i \|\nabla_{\mathbf{x}} h_i\|^2$.
    - As in SAE, optimize a regularized objective $L(\mathbf{x}, g(f(\mathbf{x}))) + \Omega(\mathbf{h}, \mathbf{x})$, but with a different form of Ω.
    - This force the model to learn a function that does not change much when $\mathbf{x}$ changes only slightly.
    - With a very high λ, a constant plane is obtained by minimizing the derivative.
- As a side effect of the contractive penalty, the encoder learn to map a **neighborhood of inputs** onto a **smaller neighborhood of outputs**.
    - Without any other competing force (or with a very high λ) the penalty would drive f to be learnt as a constant function.
        - Which maps the whole input space onto a single point.
- CAE is **contractive only locally**.
    - If $\mathbf{x}$ and $\mathbf{x}'$ are different enough, they can be mapped to point very far apart than the original points.
- In the limit of small Gaussian input noise, the **denoising reconstruction** error is equivalent to a **contractive penalty** on the **reconstruction function**.
    - DAE make the **reconstruction function** to resist to small errors in the input (decoder part).
    - CAE make the **feature extraction function** to resist small perturbations of the input (encoder part).

Learning manifolds

- Regularized AE learns manifolds by balancing two opposing forces.
    - In case of CAE, these two forces are reconstruction error and the contractive penalty $\Omega(\mathbf{h})$.
- The compromise between these two forces yields an AE whose derivatives $\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$ are mostly tiny.
    - Only a **small number of hidden units** (corresponding to a small number of directions in the input) may have **significant derivatives**.
- Directions $\mathbf{x}$ with large $\mathbf{J}\mathbf{x}$ rapidly change $\mathbf{h}$.
    - **Such directions are penalized during learning** (due to contractive penalty).
- So the ones *surviving* are those really necessary to model the data.
    - And hence they are likely to to be directions which **approximate the tangent planes of the manifold**.
        - By loosing those directions, the input cannot be reconstructed, so those are kept.

1.9.3 Variational Autoencoders

- An AE whose **encodings distribution is regularised** during the training.
    - In order to ensure that its latent space has **good properties allowing to generate** some new data.
    - The regularization method is based on *variational inference*, an inference method popular in statistics.
- AE are not good generative models.
    - To do generation:
        - After training the encoder is discarded.
        - A random $\mathbf{h}$ is picked and served to the decoder.
        - The decoder will then generate an output.
    - While this simple schema is intuitive, it does not work well in practice.
- In seen AE, encoding networks are not regularized.
    - The semantic *areas* are concentrated and well-spaced.
    - When an example (`eg` the randomized one) that is not nearby any of those area, will return a meaningless output.
    - Encoding networks need to be **regularized**.
        - Larger semantic *areas* all nearby between each other.
- To impose this kind of regularization, images are **encoded into a distribution**.
    - During training, the decoder is trained to decode points sampled from these distributions.

1.9.3.1 Probabilistic model

- A probabilistic model where a latent variable $\mathbf{h}$ is sampled from a distribution $p(\mathbf{h})$.
  - And then $\mathbf{x}$ is sampled from $p(\mathbf{x} \mid \mathbf{h})$.
- It is also assumed:
  - $p(\mathbf{h}) \sim N(\mathbf{0}, \mathbf{I})$.
  - $p(\mathbf{x} \mid \mathbf{h}) \sim N(g(\mathbf{h}), c\mathbf{I})$ (centered around the reconstruction, isotropic distribution).
- Even under these assumptions, the inference problem is **intractable** due to the normalization factor in the denominator.
  - $p(\mathbf{h} \mid \mathbf{x}) = \frac{p(\mathbf{x}|\mathbf{h})p(\mathbf{x})}{p(\mathbf{x})} = \frac{p(\mathbf{x}|\mathbf{h})p(\mathbf{x})}{\int p(\mathbf{x}|\mathbf{u})p(\mathbf{u})\,d\mathbf{u}}$.
    - $p(\mathbf{x})$ is unknown, it's therefore approximated with an integral (that is also unknown, intractable).
  - Variational inference is a technique allowing to solve this problem.
    - By **approximating the hard to compute probability with a simpler one**.
    - Specifically, a distribution $q_{\mathbf{x}}(\mathbf{h}) \sim N(f_1(\mathbf{x}), f_2(\mathbf{x})) \approx p(\mathbf{h} \mid \mathbf{x})$ is used.
      - $p(\mathbf{h} \mid \mathbf{x})$ is approximated using a Gaussian.
- *Goal*: to find f_1 and f_2 allowing the best possible approximation of $p(\mathbf{h} \mid \mathbf{x})$.
  - $\arg\min_{f_1,f_2} KL(q_{\mathbf{x}}(\mathbf{h}) \parallel p(\mathbf{h} \mid \mathbf{x})) = \cdots = \arg\max_{f_1,f_2} E_{\mathbf{h}\sim q_{\mathbf{x}}}[-\frac{\|\mathbf{x}-g(\mathbf{h})\|^2}{2c}] - KL(q_{\mathbf{x}}(\mathbf{h}) \parallel p(\mathbf{h}))$.
- *Goal*: to find the decoder function g that maximises the likelihood of $p(\mathbf{x} \mid \mathbf{h})$ under assumption $\mathbf{h} \sim q_{\mathbf{x}}(\mathbf{h})$.
  - `def` **VAE optimization problem**: $\arg\min_{f_1,f_2,g} E_{\mathbf{h}\sim q_{\mathbf{x}}}[\frac{\|\mathbf{x}-g(\mathbf{h})\|^2}{2c}] + KL(q_{\mathbf{x}}(\mathbf{h}) \parallel p(\mathbf{h}))$.
    - With $q_{\mathbf{x}}(\mathbf{h}) \sim \mathcal{N}(f_1(\mathbf{x}), f_2(\mathbf{x})) \equiv \mathcal{N}(\mu_{\mathbf{x}}, \sigma_{\mathbf{x}})$ and $p(\mathbf{h}) \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.
    - It has the form a **regularized network** (with a reconstruction term and a regularization term).
    - Easily solvable via GD.

Reparametrization trick

- While optimization is easily done via GD, the extraction of a point in the embedding space is **not differentiable**.
  - Randomly picking a point from a distribution is not a differentiable operation.
- Sampling from $N(\mu_{\mathbf{x}}, \sigma_{\mathbf{x}})$ is equivalent to sampling $\zeta \sim N(\mathbf{0}, \mathbf{I})$ and then $\mathbf{h} = \sigma_{\mathbf{x}} + \zeta\mu_{\mathbf{x}}$.
  - Points are rescaled and shifted (standard translation from standard Gaussian to other normal distributions).
  - A point is not extracted from $N(\mu_{\mathbf{x}}, \sigma_{\mathbf{x}})$, which depends on $\mu_{\mathbf{x}}$ an $\sigma_{\mathbf{x}}$, which are part of the network.
    - But is extracted from $N(\mathbf{0}, \mathbf{I})$, which contains only constant terms.
    - Then $\mathbf{h}$ is computed as described.
- During BP, $\mu_{\mathbf{x}}$ and $\sigma_{\mathbf{x}}$ are then easily reachable from $\mathbf{h}$.
  - Derivatives to ζ (where sampling is performed) are not needed.
  - Optimization is only needed for part of the NN which construct $\mu_{\mathbf{x}}$ and $\sigma_{\mathbf{x}}$.

1.9.3.2 VAE Loss

- `def` **VAE Loss**: $C\|\mathbf{x} - g(\mathbf{h})\|^2 + KL(N(\mu_{\mathbf{x}}, \sigma_{\mathbf{x}}) \parallel N(\mathbf{0}, \mathbf{I}))$.
- It can be shown that $KL(N(\mu_{\mathbf{x}}, \sigma_{\mathbf{x}}) \parallel\parallel N(\mathbf{0}, \mathbf{I})) = \frac{1}{2}[-1 - \log(\sigma_{\mathbf{x}}^2) + \sigma_{\mathbf{x}}^2 + \mu_{\mathbf{x}}^2]$.
  - Where $\mu_{\mathbf{x}} = f_1(\mathbf{x})$ and $\sigma_{\mathbf{x}} = f_2(\mathbf{x})$.

1.10 Generative Adversarial Networks

1.10.1 Generative models

- A systems that:
  - Take a training set of samples drawn from a distribution p_{data}.
  - Learn to represent and estimate of that distribution.
- To learn $p(\mathbf{x})$ is way more difficult that to learn $p(y \mid \mathbf{x})$ (discriminative models).
- The distribution p_{model} can be estimated explicitly.
  - Or the model can only give the possibility to draw samples from it.
  - GANs are usually used to draw examples even if they can be designed to do both.
- Advantages of studying generative models:
  - The ability to **represent high-dimensionality** is important in many domains.
  - GM can be incorporated in (model based) **reinforcement learning**.
    - The GM model can be *queried* by the RL system to validate assumptions.
  - They can be used as the **basis of Semi Supervised Learning** systems, etc.

1.10.1.1 Maximum likelihood models

- GANs are compared to **maximum likelihood models** (a type of generative models).
- `def` **Maximum Likelihood Optimization**: $\theta^\star = arg\ max_\theta\ p_{model}(\{\mathbf{x}^{(i)}\}_{i=1}^m; \theta)$.
  - Parameters θ that maximize the likelihood of the training data given the model.
  - Almost always that optimization is made in **log-space**.
    - $\theta^\star = arg\ max_\theta\ p_{model}(\{\mathbf{x}^{(i)}\}_{i=1}^m; \theta) \approx arg\ max_\theta\ \mathbb{E}_{\mathbf{x}\sim p_{data}}[\log p_{model}(\mathbf{x}; \theta)]$.
  - **Maximizing** the ML w.r.t. θ is the same as **minimizing** the KL divergence of p_{data} and p_{model}.
    - $KL(p_{data} \| p_{model}) = \cdots = \mathbb{E}_{\mathbf{x}\sim p_{data}}[\log p_{data}(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim p_{data}}[\log p_{model}(\mathbf{x}; \theta)]$.
      - $\mathbb{E}_{\mathbf{x}\sim p_{data}}[\log p_{data}(\mathbf{x})]$ is the log-likelihood of data.
      - $\mathbb{E}_{\mathbf{x}\sim p_{data}}[\log p_{model}(\mathbf{x}; \theta)]$ is the log-likelihood of the model.
    - Since the minimization is w.r.t. to θ, only the second element must be maximized.
- Maximum likelihood models can be subdivided into ones with *explicit density* and ones with *implicit density*.
  - Explicit density can be subdivided into tractable density and approximate.

Explicit Density models with tractable density

- The distribution is fully modelled.
- **Fully Visible Belief Nets**:
  - Strong assumption: the distribution is tractable.
  - Chain rule of probability to factor the probability of model $\mathbf{x}$ into a product of $1D$ probabilities.
    - $p_{model}(\mathbf{x}) = \prod_{i=1}^n p_{model}(x_i \mid x_1, \ldots, x_{i-1})$.
  - They are the basis of sophisticated generative models.
  - *Problem*: sample must be generated one entry at a time → generating a new sample is $O(n)$.
- **Nonlinear Independent Component Analysis**:
  - A **simple distribution** over $\mathbf{z}$ coupled with a **non-linear transformation** g that warps space in complicated ways can yield a **complicated distribution** over $\mathbf{x}$.
  - If there is a vector of latent variables $\mathbf{z}$ and a continuous differentiable, invertible transformation g such that $g(\mathbf{z})$ yields a sample from the model in $\mathbf{x}$ space, then:
    - $p_{\mathbf{x}}(x) = p_{\mathbf{z}}(g^{-1}(x)) |\det(\frac{\partial g^{-1}(x)}{\partial \mathbf{x}})| = p_{\mathbf{z}}(g^{-1}(x)) |\det(\mathbf{J}_{\mathbf{x}}(g^{-1}(x)))|$.
  - The density $p_{\mathbf{x}}$ is tractable if the density of $p_{\mathbf{z}}$ is tractable and the Jacobian of g^{-1} is tractable.
  - *Problem*: these models impose constaints on the choice of g.
    - The invertibility restriction impose that the number dimensions of $\mathbf{x}$ is equals to the ones of $\mathbf{z}$.

Explicit Density models with approximate density

- The distribution is approximated.
- **Variational approximations**:
  - Use a deterministic approximation to overcome the problems of having to deal with an intractable distribution.
  - The main idea is to define and maximize a lower bound on the intractable distribution.
    - $\mathcal{L}(\mathbf{x}; \theta) \leq \log p_{model}(\mathbf{x}; \theta)$.
    - This is the kind of approach taken by **variational autoencoder**.
  - Very often the approximation is based on multivariate gaussians.

        - VAE learns make several approximation to the true likelihood using Gaussian distribution.
    - *Drawback*:
        - When the approximation is too crude even with a perfect optimization and infinite data the gap between $\mathcal{L}$ and the true likelihood can make the results poor.
        - VAE often obtain very good likelihood, but produce lower quality samples (w.r.t. GANs).
        - If compared to FVBNs, VAE are hard to train (but GANs are even harder to train).
- **Markov Chain Approximations**:
    - Use some form of stochastic approximation.
    - Goal: to find a way to efficiently draw samples from $p_{model}(\mathbf{x})$ when the distribution is intractable.
    - **Markov Chain** method (*MC Monte Carlo*) draw examples by repeatedly sampling from simpler distributions.
        - According to a transition operator: $\mathbf{x}' \sim q(\mathbf{x}' \mid \mathbf{x})$.
    - Convergence:
        - By repeating updating $\mathbf{x}$ according to the transition operator q, MC methods can sometimes guarantee that $\mathbf{x}$ will eventually converge to sampling from $p_{model}(\mathbf{x})$.
    - *Challenge*: **slow convergence**.
        - *Learning phase*: drawing of examples via MCMC can be too costly.
        - *Inference phase*: inefficient if compared to GANs.

Implicit Density models

- Some models model the distribution only implicitly.
- **Generative stochastic models**:
    - Learns a Markov transition operator that allows to draw from the implicit model.
    - *Problem*:
        - By approximating examples using a MC, these models have the same problems as MCMC.

1.10.2 Generative Adversarial Networks

- GANs model the distribution only implicitly.
    - The distribution is implicit but can be used to generates sample.
    - Offer a single step sample generation method.
- Features:
    - GANs can generate samples in parallel.
    - The generator function has very few restrictions.
    - No costly Markov chain approximations.
    - No variational bound is needed.
    - Subjectively GANS are regarded to produce better samples.

1.10.2.1 GANs functionality

- The idea is to set up a game between two players.
    - **Generator** (G): create samples intended to come from the same distribution as the training data.
    - **Discriminator** (D): examines samples to determine whether they are real or false.
- Definitions:
    - G and D are two differentiable functions (i.e. two NNs).
    - G takes an input $\mathbf{z}$, is defined in terms of parameters θ_G and outputs a value $\tilde{x}$ from same space of $\mathbf{x}$.
    - D takes an input $\mathbf{x}$, is defined in terms of parameters θ_D and outputs a value $y \in \{False, Real\}$.
- Assumption: the discriminator tries to predict the probability that the input is real (regression).
    - $D(\mathbf{x}) \approx 1$ if the discriminator believes that the $P(y = Real \mid \mathbf{x})$ is high.
- **Cost functions**: $L_G(\theta_G, \theta_D)$ and $L_D(\theta_G, \theta_D)$.
    - ! They are both defined with respect to both θ_G and θ_D.
- Each player want to minimize its cost function but can only do so by **acting on its own parameters**.
    - The optimization can be seen as a zero-sum game between two players.
        - Not always true, depending on the definition of L_G and L_D.
        - The solution of a zero-sum is a **Nash equilibrium**.
            - A point in the (θ_G, θ_D) space such that the point is local minimum of L_D w.r.t. θ_D and a local minimum of L_G w.r.t. θ_G.
        - Nowadays, constraints for having a zero-sum game are abandoned for easier training.

1.10.2.2 GANs training

- The training process consists of simultaneous SGD.
  - Sample two minibatches (one from real data, one from generated data).
  - Evaluate the two losses and update θ_D using the gradients from L_D and θ_G using the ones from L_G.
- Sometimes more steps are performed on the discriminator before going back to the generator.

Discriminator's Cost function

- `def` **Discriminator's Cost function**: $L_D(\theta_D, \theta_G) = -\frac{1}{2}\mathbb{E}_{\mathbf{x}\sim p_{data}}[\log D(\mathbf{x})] - \frac{1}{2}\mathbb{E}_{\mathbf{z}}[\log(1 - D(G(\mathbf{z})))]$.
- The usual cross-entropy used when minimizing a binary classifier with sigmoid output units.
- Initially, all variants of GANs used this exact cost for the discriminator.
  - While, costs for the generator changes from model to model.
- $\frac{1}{2}$ is used since usually 50% of examples are real and 50% examples are not.
  - But since it is an optimization problem, constants are not impactful.
- The distribution of $\mathbf{z}$ is usually a Gaussian distribution.

Generator's Cost function

- `def` **Generator's Cost function (zero-sum game)**: $L_G = -L_D$.
  - Each time the discriminator lower its loss, the generator is penalized by the same amount.
  - The players are just trying to minimize/maximize (**minmax**) the same function.
    - $\theta_{G^*}, \theta_{D^*} = arg\ min_{\theta_G}\ max_{\theta_D} V(\theta_D, \theta_G)$, with $V(\theta_D, \theta_G) = -L_D(\theta_D, \theta_G)$.
    - A **saddle (or minmax) point** is searched in the loss surface.
      - The same loss surface is evaluated, but from two different perspectives.
      - At a relative minimum along one axial direction and at a relative maximum along the crossing axis.
  - Theoretical advantages of this setup:
    - This game corresponds to minimizing the **Jensen-Shannon divergence** between the data and the model distribution.
    - `def` **Jensen-Shannon divergece**: $JSD(P\|Q) = \frac{1}{2}KL(P\|M) + \frac{1}{2}KL(Q\|M)$.
      - Where M is the median distribution $M = \frac{1}{2}(P + Q)$.
      - The JSD is a symmetrical type of KL.
    - The game **converge to equilibrium** if both players' policies can be updated in **function space**.
      - This is not possible in practice though.
  - When the **discriminator minimizes** the cross-entropy, the generator **maximizes** it.
    - The gradient of the loss **vanishes** when the discriminator rejects the generator examples with high confidence.
- `def` **Generator's Cost function (non-saturating game)**: $L_G = \frac{1}{2}\mathbb{E}_{\mathbf{z}} \log D(G(\mathbf{z}))$.
  - The generator minimize a cross-entropy term tailored to its view of the problem.
    - The generator maximizes the probability that the discriminator is mistaken.
  - It lacks the theoretical properties of the minmax loss, but it doesn't suffer from gradient vanishing.
    - The generator loss has been chosen **heuristically** (not theoretically sound, it *just works*).
    - The game is no longer a zero-sum one.

1.10.2.3 Why do GANs work

- GANs performances were attributed to the minimization of JSD instead of KL.
  - KL is not symmetric and MLE minimizes $KL(P_{data}\|p_{model})$.
  - Minimizing the JSD is more akin to minimizing $KL(p_{model}\|p_{data})$.
- More recent results suggest that JSD doesn't explain why GANs make sharper samples.
  - Using ML to optimize GANs doesn't show problems in selecting a small number of modes and generating sharp images.
    - By selecting a single mode (corresponding to an image) the result won't be the average of several modes.
      - Averaging several modes results in a *blurrier* image.
  - GANs often choose to generate from very few modes.
    - Fewer than the number allowed by the model capacity.
    - **Model collapse problem**: generating always the same image.
    - Reverse KL would select as many modes as allowed by the model.
- This suggests GANs choose to generate a small number of modes due to a **defect in the training procedure**.
  - Rather than due to the divergence they aim to minimize.

- The reason why this happens is still not clear.
  - Maybe it **makes different approximations** than other models.
  - Maybe it **optimizes a different family of functions**.

1.10.2.4 Deep Convolutional GANs

- Nowadays, most GANS are loosely based on DCGANs.
- Main insights for this architecture are:
  - **Batch normalization**:
    - Most layers for both D and G, batches for G and D are normalized separately.
    - The last layer of G and the first layer of D are not batch normalized.
  - **No pooling nor unpooling layers**:
    - When G needs to increase the spatial dimension of the representation it used *transposed convolution* (*deconvolution*) with a stride greater than 1.
  - **Adam instead of SGD with momentum**.

1.10.3 Wasserstein GANs

- GANs are very difficult to train.
  - The game between G and D doesn't converge easily.
- `def` **Wasserstein distance** (*earth-mover distance*): $W(p,q) = \inf_{\gamma \in \Pi(p,q)} \mathbb{E}_{(x,y)\sim\gamma}\|x-y\|$.
  - A distance between two probability distributions p and q.
  - $\Pi(P,Q)$: the set of all joint distribution $\gamma(x,y)$ whose marginals are respectively p and q.
  - $(x,y) \sim \gamma$: (x,y) distributes according to γ.
  - Interpretable as the **minimum amount of *work*** required to transform the probability mass p to q.
- W has better properties than other distances and divergences used for GANs.
  - It allows **learning to convergence** in many situations when other measures fail.
  - But the infimum in the definition of W is **intractable**.

1.10.3.1 Lipschitz continuity

- A *solution* to the intractability of W.
- **WGANs**: the family of functions considered $\{f_{\theta_D}(x)\}$ is **assumed to be Lipschitz continuous**.
  - `def` **Lipschitz continuous function**: f if $\exists c$: $\|f(x) - f(y)\| \leq c\|x-y\|$ for all x and y in $dom(f)$.
    - The smaller the constant, the smoother the function.
    - The function cannot change too much in a small region of the domain.
- Under this assumption:
  - **Discriminator training**: maximize W by maximizing $\mathbb{E}_{\mathbf{x}\sim p_{data}}[D_{\theta_D}(\mathbf{x})] - \mathbb{E}_{\mathbf{z}}[D_{\theta_D}(G_{\theta_G}(\mathbf{z}))]$.
    - $\mathbb{E}_{\mathbf{x}\sim p_{data}}[D_{\theta_D}(\mathbf{x})]$: the average score for real examples.
    - $\mathbb{E}_{\mathbf{z}}[D_{\theta_D}(G_{\theta_G}(\mathbf{z}))]$: the average score for fake examples.
  - **Generator training**: minimize $L_G(\theta_G, \theta_D) = -\mathbb{E}_{\mathbf{z}}[D_{\theta_D}(G_{\theta_G}(\mathbf{z}))]$.
  - Those quantities are easy to compute (therefore W **intractability is avoided**).
- Everything can be trained by BP with a small trick to ensure that the learned function is Lipschitz continuous.
  - Every time θ_D parameters are updated, they are **clipped** in $[-c, c]$ (where c is a user-defined constant).
    - `eg` With $[-5, 5]$, both 14 and 16 are clipped to 5, while -4.2 is left as it is (no normalization).
- Another difference is the activation function of the last layer of D is **linear** (instead of sigmoid).
  - D is no longer meant to model a probability distribution (there D is often referred as the **critic**).

1.10.3.2 WGANs advantages

- The loss is **meaningful** and **correlates** with G **convergence** and sample quality.
  - Other losses do not correlate with sample quality.
  - In these cases, with a smaller loss the sample quality is still low or even worse than before.
- The discriminator f is usually trained near optimality.
  - Way more epochs are assigned to the discriminator.
    - When D is properly trained, then G is trained further (for a small amount of epochs).
  - It can be shown that the loss is an **estimate** of W **given a factor** (determined by the constant c).
  - Empirical evidence shows that this correlates with the **quality of generated samples**.
  - The discriminator **should** be trained till optimality.

    - When the critic is trained to completion, it simply provides a loss to the generator that can be trained as any other NN.
- Empirical evidence shows that WGANs are **much more robust** than GANs when one varies the generator.

1.11 Pruning

[Notes from a.a. 2024/2025, Pruning is not part of the exam curriculum in a.a. 2025/2026]

1.11.1 Deep networks

- Features of deep networks:
  - High number of hidden networks.
  - Parameters number is a proxy of complexity, but it's not its only source.
- Methods to both boost perform and make ANN robust to noise are needed.
- Recent trends in ANNs:
  - **Handcrafted** (eg VGG, ResNet).
  - **Efficient hand-crafted** (eg Inception).
    - Variation of handcrafted models with variations to make them more efficient.
  - **Neural-architecture search**.
    - A technique for automating the design of ANN.
    - Learning the architecture is adding one layer of complexity.
    - Other optimization metrics (eg best accuracy while minimizing number of parameters).
  - **Hardware-aware Neural-architecture search**.
    - In the optimization phase, the hardware is considered.
    - Information from the HW after a deploy of the model is collected.
      - Information like memory footprint, latency, energy consumption, etc.
      - And then this information is incorporated in the model.
    - A new optimization phase is launched after the information has been integrated.

1.11.1.1 Deep models outside the datacenters

- Communication has a cost, to have computation *on the edge* is desirable.
- Mobiles phones, DL accelerators, FPGAs have little memory.
- Methods: quantization (avoid FP-unit), knowledge distillation, pruning.
  - eg 8-bit quantization bit is good enough for most image classification.
  - **Knowledge distillation**:
    - A large *teacher* model and a shallow *student* model.
    - The *teacher* has learnt input-output mapping.
    - This knowledge is tried to be distilled in the *student*.
    - *Issue*: how *shallow* the model should be is not known to store this mapping.

1.11.2 Pruning

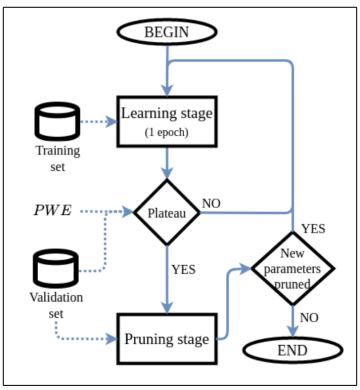

Figure 7: General Pruning scheme.

- **Sparsification**: to take some model parameters and set them to 0.
- **Pruning**: removing parameters or entire units from the DL model.
  - Pruning relates with sparsification.
    - The weight matrix (representing a layer) becomes sparse.
    - Removed units are set to 0, introducing nonetheless some representation overhead.
- Advantages:
  - The number of parameters is reduced (for special design like ASICs the gain is real).

        - Modern GPUs implement functions to completely avoid operations involving zeros.
    - If they are removed in a *structured* way (entire blocks), there is no representation overhead.
        - The gain is real even in general frameworks.
- Pruning is also biological plausible.
    - The peak synaptical connectivity is at 7 years of age, then the neuronal density decrease.

1.11.2.1 Traditional pruning methods

- Limits of traditional pruning methods:
    - Computationally extremely expensive.
    - They do not work so well in multi-layer architectures.
    - Slow (a lot of iterations to converge).

Skeletonization [Mozer & Smolensky, 1988]

- Motivation: computing was extremely expensive at the time.
- Principal idea:
    - Iteratively train the network to a certain performance criterion (until convergence).
    - Compute a measure of relevance: $\rho_i = E_{without\ uniti} - E_{with\ unit\ i}$.
        - A single unit i is removed and the error is computed without it.
        - If ρ_i is very high, the unit i is fundamental to mode performance.
    - Trim the least relevant units (once they are ranked).
- This scheme is the most common in several pruning approaches.

Optimal brain damage [Le Cun et al., 1989]

- Use of **second-order derivative information** to find a trade-off between **complexity** and **training error**.
    - Going therefore beyond gradient.
- The focus here switches from units to **single parameters**.
- MSE is used (CrossEntropy was not popular at the time).
- Principal idea: $\delta E = \sum_i g_i \delta u_i + \frac{1}{2} h_{ii} \delta u_i^2 + \frac{1}{2} \sum_{i \neq j} h_{ij} \delta u_i \delta u_j + O(\|\delta U\|^3)$.
    - With $g_i = \frac{\partial E}{\partial u_i}$ the gradient and $h_{ij} = \frac{\partial^2 E}{\partial u_i \partial u_j}$ the Hessian.
    - Let the error function being approximated through Taylor expansion.
    - A perturbation over the parameter vector (δU) will produce a perturbation on the error.
    - Goal: to find the largest subset of parameters whose pruning will cause the least increase of E.
    - **Intractable** in this form (eg for 2.6K parameters network, the Hessian would be 6.5×10^6).
- $\delta E = \frac{1}{2} \sum_i h_{ii} \delta u_i^2$ ($h_{ii} \delta u_i^2$ is the **neuron saliency**).
    - To approximate a result, the **diagonal** is used.
    - Deleting one parameter will not cause impact on the others (in general not true).
    - Assumption: the model has been already trained and a local minimum is found.
        - Therefore the gradient term of the error should be closed to zero (ignorable).
    - All the h_{ii} are non-negative (hence every perturbation will cause the error to go up or stay the same).
    - Estimating the diagonal of deviation nowadays can be done with the two back propagation passes.
        - It increases the complexity but acceptably.
- Algorithm:
    - Choose a reasonable network architecture.
    - Train the network unit a reasonable solution is obtained (train until convergence).
    - Compute the second derivatives h_u for each parameters.
    - Compute the saliencies for each parameter.
    - Sort the parameters by saliency and delete some low-saliency parameters (*thresholding*).
    - Iterate to training step.
        - First work to propose **retraining**.
        - The number of parameters to be pruned is not known in advance.
- With retraining the MSE remains small even with fewer parameters (bigger pruning).
    - Without retraining, the MSE grows with fewer parameters.

1.11.2.2 Newer pruning methods

- **Newer pruning methods**:
    - A revival of pruning methods developed after 2015.
        - The number of parameters of modern architectures became huge.

- A *slow process* of parameters pruning is still used as traditionally was.

Learning both weights and connections [Han et al., 2015]

- **Magnitude pruning** (previously proposed).
  - Simpler idea than from the past.
  - If a parameter has a very small value (its **magnitude**), it is pruned from the network.
  - No concept of saliency or other estimator.
    - But the model is **regularized**.
- Principal idea:
  - Parameters are randomly initialized (nowadays, pre-trained models are used).
  - **Training stage**: parameters are updated then trained with standard GD until performance is achieved.
  - **Parameter sparsification**: parameters below T (hyperparameter) are removed, pruning connections.
    - Drop-out stochastically disable entire units, but they don't disappear from the net.
      - The underline parameter is still in the model.
    - In parameter sparsification, the parameter is **permanently removed** after training.
  - **Neuron sparsification**: neurons without input arcs input are pruned from the network.
    - This can led to a degradation in network performance.
  - **Fine-tuning**: fine-tune the model, recovering the performance and iteratively prune again.
    - Called also re-training (by LeCun) or regularization.
- Assumption: parameters having low magnitude are also **less important**.
  - A regularization function that is trying to push as much as possible value of the parameters of the NN as close as possible to 0 is desirable.
- **Regularization**:
  - From optimization theory, LASSO (L1) is the most common choice.
  - L0 is non-differentiable (there are proxies).
  - Empirically, **L2 regularization** leads to **sparser models** (under same performance constraints).
    - Fine-tuning is the key.
    - **L2 regularization with iterative prune and retrain** works better than other variants.
      - Parameters used to store *noise* are pruned away.
      - Pruning help in the **signal-to-noise** ratio.
      - It maximizes the storage capacity of the network.
      - L2 is very cheap to implement inside an optimizer.
- Unstructured pruning can also unveil some structure of input data (even in deeper architectures).
- Network pruning can save 9× to 13× parameters with no drop in predictive performance.
- In order to prune the network, **various re-training** are needed (**extremely expensive**).
  - *Goal*: reach the highest sparsity with no task-related performance degradation.
    - The most effective approaches are also the ones **more computationally intensive**.
    - One-shot (or few-shot) pruning approaches are in general worse.
      - One-shot pruning: trained → prune → fine-tune and then stop.
      - Not as effective as re-training approach.
  - New challenge: achieve sparsity with less computation at training time.

Lottery ticket hypothesis [Frankle & Carbin, 2019]

A randomly-initialized, dense neural network contains a **sub-network** that is initialized such that, **when trained in isolation**, it can **match the test accuracy** of the original network after training for at most the same number of iterations.

- **Pruning at initialization** (even before training).
- The sub-network exists already at initialization.
  - If it can be found, a lot of computation can be saved.
    - Computation for training the model (less parameters to train).
    - Computation for pruning (no iterative pruning anymore, just pruning at initialization, **zero-shot pruning**).
- **Identification of *lottery winners***:
  - A dense architecture is taken and training and following pruning is performed.
    - **Rewind process**: just the parameters which have not been pruned will be rolled-back to the value they had at initialization (before training).
  - A smaller model is obtained, onto which the training is performed.
    - The **same performance** is obtained training the model with **less parameters**.

    - But the transition from **dense to small model** shouldn't need **full training information**.
- So there are parameters *winning at the lottery of initialization* in deep models.
    - It is possible to successfully train a model from the initialization phase.
    - Removing the largest part of the parameters and **training just the remaining fraction**.
- In This work, just the **existence of lottery winners** has been shown.
    - How to identify them at initialization (or in the first learning stages) was not proposed.
- Several methods for pruning models at initialization were then proposed.

Rigging the lottery [Evci et al., 2019]

- Principal idea:
    - Start from a random, sparse configuration of the model.
    - Then, some connections can be either chopped (pruning) or re-instated (**growing**).
- **Pruning during training**.
    - Instead of waiting to reach convergence.
    - Pruning is performed during a certain amount of iterations (eg after every epoch).
    - The network therefore became sparser and sparser.
- But maybe a parameter has low magnitude at training just because convergence is not reached.
    - No certainty that only *useless* parameters has been removed.
- Instead of only removing some parameters (**drop**), some parameters are **reintroduced** (**growing**).
    - **Growing**: connections are created in a sparse model.
    - *Proposal*: grow connection based on the **gradient** of the units.
        - A high gradient for a missing parameter is a signal to reintroduce the parameter.
        - An iterative scheme that is locally reducing the parameters number and that might converge.
- **Massive disoptimalities**:
    - Tremendously sensitive to hyper-parameters choice (eg pruning or growing rate at each iteration, etc).
    - Tremendously sensitive to dataset, architecture, etc.
- First try at solving the lottery ticket problem, but not an effective on.

SNIP [Lee et al., 2019]

- *SNIP: Single-shot network pruning based on connection sensitivity*.
- **Prune just-once**, instead of running an iterative algorithm involving regularization+pruning.
- A randomly initialized network (not even trained) is considered:
    - The **gradient** is computed for each parameters (like *gradient accumulation*).
    - *Assumption*: if the gradient is low at initialization, the parameter will not evolve much during training.
    - These parameters can be removed from the network.
        - The ranking of parameters to remove is not magnitude-based, but **sensitivity-based**.
    - This is performed right after initialization, hence the main direction of the gradient is preserved.
- Similar to *Optimal Brain Damage* (but with gradient instead of second derivatives).
    - In OBD: the network is converged $\implies$ the gradient is $\approx 0 \rightarrow$ second order information is needed.
    - In SNIP: since not at convergence, gradient information is enough.
        - Loss variation is assumed constant during training (*bold assumption*).
- Pruning at initialization using a **gradient informing the strategies** could have been a good choice to reduce the complexity.
    - This is true for MNIST, but for bigger datasets the reality is different.
- Iterative approach are generally better than zero or few-shot approaches.

Lottery winners identification

- To identify winners at initialization time is not feasible [Frankle et al, 2020 2021]:
    - Few **iterations of warm-up** is needed.
        - eg 2K for ResNet-20 on CIFAR-10, 8 out of 90 epochs for ImageNet).
        - Before there are huge (but decreasing) gradients and rapid motions in weight space.
    - Without warm-up, all lottery winners found are *unstable*.
    - This *resizes* the original lottery winner hypothesis.
- NN are initialized to maximize FP and BP signal, to find the local minimum as fast as possible.
    - Adam optimizer is typically used to converge quickly.
    - SDG is typically used for robustness.

1.11.2.3 Structured and unstructured sparsity

- Unstructured sparsification focuses on the connection.
  - Pruning connections.
  - Each neuron is still part of the network (possible bottleneck).
- Structured sparsification focuses on the neurons.
  - Pruning neurons.
  - Pruned neurons can be ignored.
- Testing on embedded devices.
  - Compression with NN (eg MPEG-7):
    - Pruning → simplification → entropy coding → bit stream → decompression.
- Unstructured strategies prune way more parameters than the structured counterpart.
  - eg Even with 99% reduction in parameters, the memory footprint can goes from 47MB to 39MB.
    - Even with entropy coding, the memory footprint saving is not much.
  - While structured approach, the parameter reduction is less but the memory footprint saving is bigger.
- There are hybrid approaches that uses both structured and unstructured sparsity.

1.11.2.4 Pruning and GPUs

- With parallel computation, the bottleneck is in either caching or the critical path (maximum model's path).
  - If caching is sufficiently fast, the bottleneck is the critical path.
- The possibility of pruning entire layers must be established.

Layer collapse

- When a specific layer in a NN fails to effectively differentiate the input features.
  - Causing all outputs from the layer to converge to similar or identical values regardless of input.
  - Seen as a downside of poor initialization or hyper-parameter fine-tuning (i.e. the model does not learn).
  - But if the model learns while some layers are collapsed, the computation of this layers can be **skipped**.
- Layer collapse **without skip connections**:
  - One way to reduce model's depth is to **remove non-linearities**.
    - Layer fold is one approach parametrizing the negative slope of a PReLU.
      - α is a parameter: when $\alpha = 0 \rightarrow$ ReLU, when $\alpha = 1 \rightarrow$ identify activation.
      - With an identity activation, the layer is successfully **linearized**.
- With fully-connected layers:
  - The solution is straight-forward using simple linear algebra.
- With convolutional layers (stride = 1, no padding):
  - The solution is straight-forward using simple linear algebra.
- With convolutional layers (stride = 1, with padding = 1):
  - Up to 2024, no closed-form solutions has been found.
  - Most architectures using skip connections make use of padding to maintain same dimensionality of the output.

1.12 Transformers

1.12.1 Introduction to transformers

- The main idea behind LLM is:
  - To have a **large model**.
  - Trained on a **large dataset**.
  - To **predict the next token in a sequence**.
- They unlocked **efficient parallel processing** and the ability to **model long-range dependencies**.
- Before Transformers, the main tools were RNNs and LSTMs.
  - These models have some limitations.
    - They are **sequential** and cannot be easily parallelized.
    - They have difficulties in capturing **long-range dependencies**.

1.12.1.1 Foundation models

Scaling hypothesis: performance improves smoothly as we increase the **model** size, **dataset** size, and amount of **compute** used for training. For optimal performance **all three factors must be scaled up in tandem**.

- Large models that are trained on a wide range of tasks.
  - And can be **fine-tuned** for specific tasks.
- Transformers make them possible because of the following reasons:
  - They are **scalable** and can be trained on very large datasets.
    - Exploiting **parallelism** and distributed computing.
  - They can be trained in a **self-supervised** way (no need for labeled data).
  - The **scaling hypothesis** asserts that:
    - Simply by increasing the scale of the model (as measured by the number of parameters)
    - And training on a commensurately large dataset.
    - Significant improvements can be achieved, even with no architectural changes.
  - Performance has a power-law relationship with each of the three scale factors.

1.12.2 Attention

- A mechanism that allows a model to focus on different parts of the input when making predictions.
- Originally introduced as an enhancement to RNNs for machine translation.
  - Later showed that significantly better results can be achieved using attention mechanism alone.
    - **Eliminating the recurrence mechanism** completely.
- A transformer can be viewed as a way to build a **richer form of embedding**.
  - In which a given vector is mapped to a location that depends on the other vectors in the sequence.
  - While previous NN architectures (once trained) are fixed on word orders.
- Why attention work better than recurrence:
  - Attention layers let **every token interact with every other token in one step**.
    - Whereas recurrent models process tokens **one at a time**.
  - This remove the sequential bottleneck, enabling **massive parallelization**.
    - And helping capture **long-range dependencies** more effectively.

1.12.2.1 Transformer processing

- Input data to a transformer is a sequence of vector $[x_n^T]_{n\in[N]}$ of dimensionality D.
  - Each vector is called a **token** (eg a word in a sentence, a patch within an image, etc).
  - Tokens are collected in a matrix $\mathbf{X} \in \mathbb{R}^{N\times D}$, which is the input to the transformer.
- The fundamental building block of a transformer is the **transformer layer**.
  - A function that takes $\mathbf{X} \in \mathbb{R}^{N\times D}$ as input and produces a matrix $\tilde{\mathbf{X}} \in \mathbb{R}^{N\times D}$ as output.
    - $\tilde{\mathbf{X}} = \text{TransformerLayer}[\mathbf{X}]$.
  - The transformer layer is composed by two blocks: the **attention** block and the **transform** block.

1.12.2.2 Attention weights

- Assume to want to compute new embeddings $\mathbf{y}_1, \dots, \mathbf{y}_N$ for tokens $\mathbf{x}_1, \dots, \mathbf{x}_N$.
  - In such a way that the embedding for $\mathbf{y}_n$ depends on the embeddings of all other token.
    - Instead of moving information *left-to-right* (as in RNNs), everything is connected.
  - $\mathbf{y}_n = \sum_{m=1}^{N} a_{nm}\mathbf{x}_m$ (where a_{nm} are the **attention weights**).

- Requirements for attention weights:
  - Capture the similarity between the tokens $\mathbf{x}_n$ and $\mathbf{x}_m$.
  - $a_{nm} \geq 0$.
  - $\sum_{m=1}^{N} a_{nm} = 1$ (to form a distribution).
- `def` **Dot-product Self-attention**: $a_{nm} = \frac{\exp(\mathbf{x}_n^T \mathbf{x}_m)}{\sum_{m'=1}^{N} \exp(\mathbf{x}_n^T \mathbf{x}_m)}$.
  - **Similarity** is computed via **dot-product** (common method).
  - It computes the a between the tokens in the same sequence using dot-product.
  - It uses **softmax** to form a distribution.
    - It takes the exponential and then normalize the result to sum to one.
  - With unrelated $\mathbf{x}_n$ and $\mathbf{x}_m$ (i.e. orthogonal), the attention weights will be ≈ 0.
- `def` **Dot-product Self-attention** (matrix version): $\mathbf{Y} = Softmax[\mathbf{X}\mathbf{X}^T]\mathbf{X}$.
  - Fast way to compute all the attention weights at once.

1.12.2.3 Self-attention

- **Query, key and value**:
  - **Query**: the *user* request to get the value.
  - **Key**: the information that the system uses to accompany the value and that should be match with query.
  - **Value**: the information should be returned when the query match the key.
- **Soft-attention**:
  - Continuous variables are used to measure the degree of match between the query and the keys.
  - These variables are used to weight the influence of the value vectors on the output.
- Question: given an embedding of a token $\mathbf{x}_n$, to which information should this token attend to compute its new embedding and how much.
  - $\mathbf{x}_n$ can be used to produce a **query**, and each $\mathbf{x}_m$ in the sequence can be seen as giving rise to:
    - A **key** than can be matched against the **query** $\mathbf{x}_n$, to get a sense of how similar they are.
    - A **value** that can be used to compute how much influence token $\mathbf{x}_m$ should have on the new embedding of $\mathbf{x}_n$.
- `def` **Self-attention** (matrix version): $\mathbf{Y} = Softmax[\mathbf{X}\mathbf{X}^T]\mathbf{X} = Softmax[\mathbf{Q}\mathbf{K}^T]\mathbf{V}$.
  - **Value**: the token $\mathbf{x}_n$ that will be used to compute output tokens.
  - **Key**: the token $\mathbf{x}_n$ for value $\mathbf{x}_n$.
  - **Query**: the token $\mathbf{x}_n$ that will be used to compute the attention weights for output $\mathbf{y}_n$.
  - Self-attention since the **same sequence** is used to determine **all three components**.
  - $\mathbf{Q}$, $\mathbf{K}$ and $\mathbf{V}$ are actually three different linear projections of $\mathbf{X}$.
    - These projections allow the model to learn **separate similarity spaces** for queries, keys and values.

Trainable parameters

- The given formula is **deterministic** and doesn't depend on the parameters model (not trainable).
- Each feature with token $\mathbf{x}_n$ contributes equally to the attention weights.
  - Whereas the flexibility to focus more on some features than on others is desirable.
- **Trainable parameters** are introduced to compute the attention weights, $\tilde{\mathbf{X}} = \mathbf{X}\mathbf{U}$.
  - Where $\mathbf{U} \in \mathbb{R}^{D \times D}$ is a matrix of trainable parameters.
    - Analogous to a layer in a standard NN.
  - Therefore the formula for new embeddings is $\mathbf{Y} = Softmax[\mathbf{X}\mathbf{U}\mathbf{U}^T\mathbf{X}^T]\mathbf{X}\mathbf{U}$.
    - But the matrix $\mathbf{X}\mathbf{U}\mathbf{U}^T\mathbf{X}^T$ is **symmetric** (while significant asymmetries should be supported).
      - `eg` *Chisel* should be strongly associated with *tool* (since every chisel is a tool).
        - But *tool* should only be weakly associated with *chisel* (not every tool is a chisel).
  - Also, the **same matrix** $\mathbf{U}$ is used to define both value vectors and attention coefficients (not ideal).
    - Therefore different separate matrices are defined for queries, keys and value.
- **Trainable parameters**:
  - $\mathbf{Q} = \mathbf{X}\mathbf{W}^{(q)}$.
    - With dimensionality $D \times D_k$ (where D_k is the dimensionality of the **key** vectors).
  - $\mathbf{K} = \mathbf{X}\mathbf{W}^{(k)}$.
    - With dimensionality $D \times D_k$, so that the dot product with **query** vectors $\mathbf{Q}\mathbf{K}^T$ is well defined.
  - $\mathbf{V} = \mathbf{X}\mathbf{W}^{(v)}$.
    - With dimensionality $D \times D_v$ (where D_v is the dimensionality of the output **value** vectors).

- `def` **Self-attention**: $\mathbf{Y} = Softmax[\mathbf{Q}\mathbf{K}^T]\mathbf{V}$.

Scaled self-attention

- The gradient of the softmax function can become exponentially small for inputs of high magnitude.
  - Which can lead to **vanishing gradients** during training.
- The values of the dot product can be **scaled** by a factor $\sqrt{D}$.
  - Where D is the dimensionality of the key vectors.
- `def` **Scaled self-attention**: $\mathbf{Y} = Attention(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = Softmax[\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{D_k}}]\mathbf{V}$.
- **Choice of the scaling factor** $\sqrt{D_k}$:
  - $\mathbf{q}$ and $\mathbf{k}$ vectors are assumed to be independent and have zero mean and unit variance.
  - D_k is then exactly the **standard deviation** of each one of the dot products in $\mathbf{Q}\mathbf{K}^T$.
  - The standard deviation of $\mathbf{q}^T\mathbf{k}$ matters since:
    - **Softmax saturates for large-magnitude inputs**.
      - Very large or very small logits entering the softmax leads to **vanishing gradients**.
    - The **typical magnitude** of $\mathbf{q}^T\mathbf{k}$ **grows as** $O(\sqrt{D_k})$.
      - Since $\text{std}(\mathbf{q}^T\mathbf{k}) = O(\sqrt{D_k})$, increasing the dimensionality of key/query vectors makes the dot-product logits naturally grown in magnitude.
      - Larger logits make the softmax more likely to saturate, causing the above gradient issue.

1.12.2.4 Multi-head self-attention

- Used to handle when there might be **multiple patterns of attention** that are relevant at the same time.
  - `eg` IN NL, some patterns might be relevant to tense whereas other might be relevant to subject-verb agreement.
- Given H heads indexed by $h \in [H]$, for each h:
  - $\mathbf{H}_h = Attention(\mathbf{Q}_h, \mathbf{K}_h, \mathbf{V}_h)$.
    - Where $\mathbf{Q}_h = \mathbf{X}\mathbf{W}_h^{(q)}$, $\mathbf{k}_h = \mathbf{X}\mathbf{W}_h^{(k)}$ and $\mathbf{V}_h = \mathbf{X}\mathbf{W}_h^{(v)}$.
  - The heads are first concatenated and then multipled by a matrix $\mathbf{W}^{(o)}$.
- `def` **Multi-head self-attention**: $\mathbf{Y}(\mathbf{X}) = \text{Concat}[\mathbf{H}_1, \dots, \mathbf{H}_H]\mathbf{W}^{(o)}$.
  - Dimensionality:
    - $\mathbf{Y}(\mathbf{X})$: $N \times D$.
    - $\text{Concat}[\mathbf{H}_1, \dots, \mathbf{H}_H]$: $N \times HD_v$.
    - $\mathbf{W}^{(o)}$: $HD_v \times D$.
  - $\mathbf{W}^{(o)}$ is both used:
    - To **reduce dimensionality** (to have $N \times D$).
    - To **learn to best combine attention heads to produce a useful output**.
  - Typically $D_v = \frac{D}{H}$ so that the concatenated matrix has the same dimensionality as the input matrix.
    - So that $N \times D$ for the concat, and $D \times D$ for $\mathbf{W}^{(o)}$.
    - In this case $\mathbf{W}^{(o)}$ is not needed for dimensionality reduction, but for the other reason.

1.12.3 Transform block

- A few additional improvements can be made to self-attention to make it more expressive and easier to train.
- To improve training efficiency:
  - A **residual connection** around self-attention is added.
  - Followed or proceeded by **layer normalization**.
  - $\mathbf{Z} = \text{LayerNorm}[\mathbf{Y}(\mathbf{X}) + \mathbf{X}]$ or $\mathbf{Z} = \mathbf{Y}(\text{LayerNorm}[\mathbf{X}]) + \mathbf{X}$.
- The output is then passed through a non-linear NN with D input units and D output units (MLP).
  - This can be a two-layer feedforward NN with ReLU activations.
  - $\tilde{\mathbf{X}} = \text{LayerNorm}[\text{MLP}[\mathbf{Z}] + \mathbf{Z}]$ or $\tilde{\mathbf{X}} = \text{MLP}[\text{LayerNorm}[\mathbf{Z}]] + \mathbf{Z}$.

1.12.4 Positional encoding

- **Permutation equivariant of self-attention**:
  - Consider $\mathbf{X} \in \mathbb{R}^{N \times d}$ a sequence of token embeddings and $\mathbf{P}$ a permutation matrix of the rows of $\mathbf{X}$.
    - $\mathbf{P}$ is a square binary matrix obtained by permuting the rows of an identity matrix.
    - $\mathbf{PX}$ (left multiplication) $\rightarrow$ permuting the rows i and j of $\mathbf{X}$ iff $P_{ij} = 1$.

    - $\mathbf{P}^T = \mathbf{P}^{-1} \implies \mathbf{P}^T\mathbf{P} = \mathbf{P}\mathbf{P}^T = \mathbf{I}$.
  - The main matrix operation in the transformer are $\mathbf{X}\mathbf{X}^T\mathbf{X}$.
    - Note that $\mathbf{P}\mathbf{X}(\mathbf{P}\mathbf{X})^T\mathbf{P}\mathbf{X} = \mathbf{P}\mathbf{X}\mathbf{X}^T\mathbf{P}^T\mathbf{P}\mathbf{X} = \mathbf{P}(\mathbf{X}\mathbf{X}^T\mathbf{X})$.
    - $\implies$ **Self-attention is equivariant** to the permutations of the input tokens.
  - This is a problem, since **order** of tokens is usually important (eg in text sentences).
- The goal is not to change the transformer architecture.
  - Instead, to **encode the position of the tokens in the input sequence**.
  - Idea: to encode the position n of the n-th tokens as an additional input vector $\mathbf{r}_n$.
    - And combine it with the token vector $\mathbf{x}_n$ before passing it to the transformer.

1.12.4.1 Appending or adding positional information

- **Appending** the positional encoding to the token vector is not desirable.
  - Since the dimensionality of the input space and subsequent layers increase.
- The positional encoding can be simply **added** to the token vector.
  - This can seems dangerous since it induces a corruption or loss of information.
  - But two randomly chosen uncorrelated vectors tend to be nearly **orthogonal in high-dimensional spaces**.
    - Allowing the network to process them separately.
  - The residual connection allows the positional information to not get lost too.
- Due to linear processing in the transformer, **a concatenated representation exhibits properties to an additive one**.

1.12.4.2 Ideal and bad positional encodings

- An ideal encoding should:
  - Be **unique** for each position.
  - Be **bounded** (each element of the encoding representation should have a **finite range**).
  - Generalize to **sequences of arbitrary length**.
  - Have a consistent way to express **relative positions**.
- eg Bad positional encodings:
  - **One-hot encoding**:
    - It's unique and bounded.
    - But it doesn't generalize to sequences of arbitrary length.
    - And not make it easy to reason about relative positions.
  - **Assigning an integer to each position**:
    - It's unique, but not bounded.
    - It may start to corrupt the vector significantly as the sequence length increases.
  - **Assigning a real number in** $[0, 1]$ **to each position**:
    - It's bounded, but not unique since it depends on the length of the sequence.

1.12.4.3 Sinusoidal positional encoding

- There are many approach to define positional encodings.
  - One of the most popular is the **sinusoidal positional encoding**.
- def **Sinusoidal positional encoding**: $\mathbf{r}_n = [\sin(w_1 \cdot n),\ \cos(w_1 \cdot n), \dots, \sin(w_{D/2} \cdot n), \cos(w_{D/2} \cdot n)]$.
  - Where n is the token position, $w_i = \frac{1}{10000^{2i/D}}$ (frequency) and D the size of the representation.
- prop This encoding makes it easy to reason about **relative positions**.
  - Two reasoning sustains this assertion:
    - The dot product between two positional encoding $\mathbf{r}_n$ and $\mathbf{r}_m$ depends only on $n - m$.
      - And not on the absolute positions n and m.
    - Encoding of $n + m$ can be expressed as a **linear combination of the encodings** of n and m.
      - It is always possible to find $\mathbf{M}$ that depends only on k, such that $\mathbf{r}_{n+k} = \mathbf{M}\mathbf{r}_n$.

1.12.5 Applications of Transformers

- Three modes:
  - **Prediction** (**encoders**)
  - **Generation** (**decoders**)
  - **Translation** or **seq2seq** (**encoder-decoders**).

1.12.5.1 Tokenization

- Tokens are generally small groups of characters.
  - They might include words in their entirety.
  - Created in a pre-processing step that convert a string of words and punctuation into a string of tokens.
- Tokenization can be applied to various types of data (images, etc).
- **Byte pair encoding**:
  - One of the most common tokenization method.
  - It initially treats the set of unique characters as 1-character-long n-grams (the initial tokens).
  - Then, successively, the **most frequent pair of adjacent tokens** is merged into a new, longer n-gram.
    - All instances of the pair are replaced by this new token.
  - This is repeated until a vocabulary of prescribed size is obtained.
- Rule of thumb: one token generally corresponds to ~ 4 character of text for common English.
  - This translates to roughly $\frac{3}{4}$ of a word (`eg` 100 tokens $\approx$ 75 words).

Detokenization

- Modern tokenizer must take *special care* to ensure that text can be reconstructed **exactly**.
  - With a naive concatenation, the original spacing may be lost.
  - A proper **detokenization** step is therefore required to recover the original text.
- Modern LLM tokenizers **encode** whether a **space precedes each token directly inside the token itself**.
  - During tokenization a special prefix character is added whenever a token begins a new word.
  - Detokenization simply consists of joining the tokens **exactly as the tokenizer produced them**.
    - After which the special space-prefix markers are converted back into normal spaces.

1.12.5.2 Decoder transformers (*generation*)

- Goal: to use a transformer architecture to construct a **autoregressive model**.
  - `def` **Autoregressive model**: $p(\mathbf{x}_1, \dots, \mathbf{x}_N) = \prod_{n=1}^{N} p(\mathbf{x}_n \mid \mathbf{x}_1, \dots, \mathbf{x}_{n-1})$.
    - But in the attention weights, everything is connected, is not sequential.
  - $p(\mathbf{x}_n \mid \mathbf{x}_1, \dots, \mathbf{x}_{n-1})$ are expressed by a transformer network learned from data.
- The architecture consists of a **stack of transformers**:
  - That a sequence of tokens.
  - And produce a sequence $\tilde{\mathbf{x}}_1, \dots, \tilde{\mathbf{x}}_n$ of dimensionality D as output.
  - Then a linear transformation followed by a softmax to compute the **distribution** over the K output tokens.
  - $\mathbf{Y} = \text{Softmax}[\tilde{\mathbf{X}}\mathbf{W}^{(p)}]$.

Masking

- Decoder models generate text by sampling from $p(\mathbf{x}_n \mid \mathbf{x}_1, \dots, \mathbf{x}_{n-1})$ at each step n.
  - Older predictions are **used as input** to the model to generate the next token.
- With other NN architecture for processing sequences it would be nice to **process entire sequences at once**.
  - But the model would be able to see the future tokens (making the task trivial).
- The attention mechanism gives a way to **mask out the future tokens** when computing the attention weights.
  - Thus allowing parallel processing of whole sequence.
  - The idea consists of two steps:
    - Add a special token to the input sequence that represents the start of the sequence.
    - Mask out (set to zero) the attention weights for the future tokens.
  - The attention weights are computed as $\mathbf{QK}^T$ but with attention set to 0 for **future tokens**.
    - A **mask matrix M** has $-\infty$ in the upper triangular part.
    - $\mathbf{Y} = \text{Softmax}[\frac{QK^T}{\sqrt{D_k}} \circ \mathbf{M}]\mathbf{V}$.
    - Therefore $\mathbf{y}_i$ will depends only on $\mathbf{x}_1$ to $\mathbf{x}_{i-1}$.

Padding

- To allow processing multiple sequences at once, it is desirable to collect them in a sequence.
  - But this would require that all sequences have the same length.
- To solve this issue, sequences are **padded** to the same length.
  - Using a special token to represent the padding.

- A **mask matrix M** that has $-\infty$ in the positions of the **padding tokens** is used.
  - So that the attention weights for the padding tokens are zero.

Sampling strategies

- Once the distribution $p(\mathbf{x}_n \mid \mathbf{x}_1, \dots, \mathbf{x}_{n-1})$ is computed, the next token can be sampled from it.
- The **greedy strategy** simply chooses the token with the highest probability.
- To find the most probable sequence, the joint distribution over all tokens must be maximized.
  - $p(\mathbf{y}_1, \cdots, \mathbf{y}_N) = \prod_{n=1}^{N} p(\mathbf{y}_n \mid \mathbf{y}_1, \cdots, \mathbf{y}_{n-1})$
  - The higher probability sequences can be generated using a **beam search** strategy.
    - Which keeps track of the k most probable sequences at each step.
- However, the most likely sequences are not necessarily the most human-like sequences.
  - By sampling on the full distribution, sequences that are nonsensical or grammatically incorrect can be generated.
  - This arises from the typically very large size of the token dictionary.
    - Which has **long tail of tokens** with very **low probability**.
- **Top-K sampling** mitigates this issue by sampling from a **truncated distribution**.
  - A distribution that includes the top-K most probable tokens.
- **Top-p sampling** (**nucleus sampling**) samples from a truncated distribution that:
  - Includes the smallest set of tokens whose cumulative probability exceeds a threshold p.
- `def` **Temperature scaling**: $y_i = \frac{\exp(a_i/T)}{\sum_j \exp(a_j/T)}$.
  - A soft version of Top-K sampling.
  - Scales the pre-activation values by a temperature parameter T before applying the softmax function.
  - When:
    - $T \to 0$ the distribution becomes more peaked around the most probable tokens.
    - $T = 1$ the distribution is the same as the original softmax distribution.
    - $T \to \infty$ the distribution becomes uniform across all states.

1.12.5.3 Encoder transformers (*prediction*)

- Used to process sequences of tokens and produce contextual embeddings that can be used to produce a **fixed-size representation of the sequence**.
- **Masked language model**:
  - The $\langle class \rangle$ is put at the beginning of the input sequence.
    - During pretraining, **class** token **is not itself predicted**, since only masked tokens contribute directly to the loss.
    - But it still participates fully in the transformer layers.
    - It attends to all other tokens (and vice versa), so it representation becomes a **contextual summary of the whole sequence**.
    - It is never masked during pretraining.
  - A randomly chosen subset (`eg` 15%) of the tokens are replaced by the **mask token**.
  - The model is then trained to predict the original tokens from the masked tokens.
    - **Self-supervised learning task**.
- For sequence classification task, the **class token** is used to produce a fixed-size presentation.
  - Which is then passed through a linear layer to produce the output.
- For token classification tasks, the output of the transformer is passed through a linear layer (or more complex classifier) for each token to produce the output.
  - The output of the class token is ignored.

1.12.5.4 Encoder-Decoder transformers (*seq2seq*)

- To perform translation, the transformer architecture uses a **cross-attention mechanism**.
  - It allows the **decoder to mix information from the encoder** with the information generated so far.
  - The output of the E **Z** is fed with entire input sequence in the **multi-head cross attention** in D.
    - **Z** will be the representation of the **class token**.
- At training time, the input of D will be the *translated sequence*.
  - It is therefore a **supervised learning task**.

1.13 Graph neural networks

1.13.1 Graph neural networks

- NNs tailored to process **graphs** (their architecture are not *graph-like*).
- Graphs are a general language for **describing entities with relations/interactions**.
  - eg Social networks, biological networks, chemical compounds, etc.

1.13.1.1 Graphs

- def **Graph**: $G = (V, E)$, a tuple of a set of **nodes** and a set of **edges**.
  - Each edge is a pair of two vertices.
- **Adjacency matrix**:
  - An undirected graph G with n nodes is assumed.
  - A (binary) **adjacency matrix** A^G:
    - A square matrix of size $n \times n$.
    - Each entry $\mathbf{A}_{ij}^G$ is 1 it there is an edge between the node i and j.
- Setup:
  - A graph $G = (V, E)$ is given.
  - V is the set of nodes and E is the set of edges.
  - $\mathbf{A}$ is the adjacency matrix.
  - $N(v)$ is the set of neighbors of node $v \in V$.
  - Each node v has an associated **feature vector** $\mathbf{x}_v \in \mathbb{R}^d$:
    - If not, $\mathbf{x}_v = 1$ can be set.
    - Or an indicator vector to each node (one-hot encoding) can be assigned.

Tasks involving graphs

- **Node-level** prediction (eg drug-drug interaction).
  - The training can be performed on a single graph.
  - The test is performed on *unlabelled* nodes.
- **Link-level** prediction (eg recommendation).
  - The training can be performed on a single graph.
  - The model can learn patterns and relationships that indicate preferences.
  - The model can use interactions and node features to predict the likelihood of preferences.
- **Graph-level** prediction (eg time of arrival).

1.13.2 Graphs and Machine learning

- The traditional method to handle graphs using ML:
  - Extract features from the graph (eg node degree, centrality, subgraph patterns, etc).
  - Use these features as input to a ML model (eg SVM, NN, RF, etc).
- **Graph representation learning**:
  - DNN have changed the way data is learnt and represented.
  - Feature engineering is replaced by **learning representation**:
    - Learn a function that maps a graph to a vector representation.
    - Use the learned representation for downstream tasks (eg node classification, etc).

1.13.2.1 Node embeddings

- Early approach in GNN.
- Goal: define $ENC()$ so that $sim(u, v) \approx \mathbf{z}_u^T \mathbf{z}_v$.
  - Where sim is a similarity measure between nodes u and v.
  - A simple way to get graph embeddings is to aggregate (eg average) the node embeddings.

Encoder-decoder framework

- **Encoder**: $\mathbf{z}_v = ENC(v)$.
  - A function that maps nodes to **vector representations**.
- **Similarity**: $sim(v, u)$.
  - A function that measures the similarity between two nodes in the **graph space** (**domain knowledge**).
  - **Random walk**: a sequence of steps where at each step, a node is chosen randomly from the neighboring nodes of the current node.

    - **Random Walk similarity** (RW): a node similarity based on visiting node u on a random walk starting from node v.
    - RW is a flexible stochastic definition that incorporates both **local and high-order neighborhood information**.
    - `eg` Expected commute time between two nodes, etc.
- **Decoder** $y_{vu} = DEC(\mathbf{z}_v, \mathbf{z}_u)$.
  - A function that compute the similarity between two nodes in the **embedding space**.
  - Usually, a function of the dot product between embeddings of two nodes.
    - $y_{vu} = DEC(\mathbf{z}_v, \mathbf{u}) = f(\mathbf{z}_v^T \mathbf{z}_u)$.
- Once fixed DEC, the goal is to learn the function ENC such that the similairty in the embedding space y_{vu} **approximates the similarity in the graph space**.

Learning

- *Goal*: to learn the parametrized function ENC_θ such that $\mathbf{z}_v^T \mathbf{z}_y = ENC_\theta(v)^T ENC_\theta(u)$.
  - $\approx$ probability that u and v co-occur on a random walk over the graph.
- `def` **Node embeddings loss (for RW)**: $\arg\min_\theta \sum_{v \in V} \sum_{u \in N_r(v)} -\log(\frac{\exp(\mathbf{z}_v^T \mathbf{z}_u)}{\sum_{w \in V} \exp(\mathbf{z}_v^T \mathbf{z}_w)})$.
  - A heuristic that tries to approximate the above probability.
  - Where $N_R(v)$ is the set of neighbors of v according to the random-walk strategy R.
  - This expression is **easily optimized using SGD**.
  - Derivation:
    - $-\log(\frac{\exp(\mathbf{z}_v^T \mathbf{z}_u)}{\sum_{w \in V} \exp(\mathbf{z}_v^T \mathbf{z}_w)}) = \cdots = C_v \mathbf{z}_v^T \mathbf{z}_w$.
    - v is assumed fixed, the expression is minimized when $\mathbf{z}_v^T \mathbf{z}_w$ is maximized.
    - Which is, when the similarity between v and u is maximized.
    - Since u is drawn from v neighbors, the **similarity** between v and its neighbors is **effectively maximized**.
  - f is assumed to be a monotonic function (`eg` a normalization of the dot product).
  - This loss is **quadratic**, in practice **negative sampling optimization** is used.
    - The denominator becomes $\sum_{w \notin N_R(v)} \exp(\mathbf{z}_v^T \mathbf{z}_w)$.

1.13.2.2 Graph embeddings

- The node embeddings can be aggregated into a graph embedding $\mathbf{z}_G$.
  - Using different strategies to obtain a graph embedding.
  - The most common strategies are:
    - Sum/average of the node embeddings.
    - Create a super-node that represents the entire graph.
- Early attempts tried to directly learn the embeddings of the nodes.
  - Instead of learning a function that maps the graph to a vector representation.
  - Idea:
    - Learn the embedding $\mathbf{z}_v$ for all nodes $v \in V$.
    - Use the embeddings for downstream tasks.
- Downstream tasks:
  - **Node clustering**: cluster points $\{\mathbf{z}_u\}$ in the embedding space.
  - **Node prediction**: predict the label of a node u based on $\mathbf{z}_u$.
  - **Link prediction**: predict the existence of edge (u, v) based on $\{\mathbf{z}_u, \mathbf{z}_v\}$.
    - Combining the embeddings: concatenate, Hadamard, sum/avg, distance, etc.
  - **Graph classification**: predict the label of the entire graph based on $\mathbf{z}_G$ (hardest task).

1.13.2.3 Limitations of earlier approaches

- Why not use the predefined similarity (which node embeddings try to reproduce) on graph nodes.
  - Those similarity are usually:
    - Implicit or expensive to compute.
    - Not suitable as ML inputs.
    - Tied to a fixed graph.
    - Hard to generalize or scale.
  - Node embeddings act as a **low-dimensional**, **learnable surrogate** for graph similarity.
- Limitations of earlier approaches:
  - **Transductive method**.

        - Cannot obtain embeddings for nodes not in the training set.
            - Therefore cannot be applied to new graphs.
        - Inductive methods: a model is built and used to predict on new data.
    - Cannot capture structural similarity.
    - Cannot utilize node, edge and graph features.

1.13.3 Modern approaches to Graph neural network

- Assumptions:
    - Arbitrary size and complex topological structure (no spatial locality like grids).
    - No fixed node ordering or reference point.
    - Often dynamic (graph changes over time) and have multimodal features.
- Using an adjacency matrix with extra information for features is problematic.
    - By swapping the labelling of two nodes, the input matrix changes (not desirable).
    - Main issues:
        - **Sensitive to node ordering**.
        - Not applicable to graphs of different sizes (the input layer of the DNN wouldn't match).

1.13.3.1 Invariance and equivariance

- **Permutation invariance**:
    - Graph does not have a canonical order of the nodes.
    - Graph and node **representations should be the same** for different ordering.
- **Permutation matrix**:
    - A square binary matrix that has:
        - Exactly one entry of 1 in each row and each column.
        - 0 elsewhere.
    - Multiplication:
        - If multiplied on the left of a matrix, it permutes the rows of the matrix.
        - If multiplied on the right of a matrix, it permutes the columns of the matrix.
    - The structure of the corresponding graph is the same, but the nodes are permuted.
        - The roles played by the nodes is no more the same, since their connectivity is different.
    - To restore the correct connectivity, the adjacency matrix must be permuted as well.
        - The permutation matrix is applied to swap the rows and columns of the adjacency matrix.
            - $\mathbf{PAP}^T$ is computed.
        - Then the graph is the same as the original one, but with a different labelling.
- `def` **Permutation invariance**: the graph function f if $f(\mathbf{A}, \mathbf{X}) = f(\mathbf{PAP}^T, \mathbf{PX})$.
    - For any graph function $f : \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times m} \to \mathbb{R}^d$ and for any permutation matrix $\mathbf{P}$.
        - $\mathbb{R}^{n\times n} \times \mathbb{R}^{n\times m}$, $\mathbb{R}^{n\times n}$ is for the adjacency matrix, $\mathbb{R}^{n\times m}$ for the feature matrix.
        - n is the number of nodes, m the number of features of a node.
- `def` **Permutation equivariance**: the node function f if $\mathbf{P}f(\mathbf{A}, \mathbf{X}) = f(\mathbf{PAP}^T, \mathbf{PX})$.
    - For any node function $f : \mathbb{R}^{n\times n} \times \mathbb{R}^{n\times m} \to \mathbb{R}^{n\times d}$ and for any permutation matrix $\mathbf{P}$.
        - n is the number of nodes, m the number of features of a node.
    - The output is permuted as is the input (it changes but it changes in the same manner).
- `eg` Permutation invariant and equivariant functions:
    - $f(\mathbf{A}, \mathbf{X}) = 1^T\mathbf{X} = \sum_i \mathbf{x}_i$ is permutation-invariant.
        - Proof: $f(\mathbf{PAP}^T, \mathbf{PX}) = \mathbf{1}^T\mathbf{PX} = \sum_i x_i = f(\mathbf{A}, \mathbf{X})$.
    - $f(\mathbf{A}, \mathbf{X}) = \mathbf{X}$ is permutation-equivariant.
        - Proof: $f(\mathbf{PAP}^T, \mathbf{PX}) = \mathbf{PX} = \mathbf{P}f(\mathbf{A}, \mathbf{X})$.
    - $f(\mathbf{A}, \mathbf{X}) = \mathbf{AX}$ is permutation-equivariant.
        - Proof: $f(\mathbf{PAP}^T, \mathbf{PX}) = \mathbf{PAP}^T\mathbf{PX} = \mathbf{PAX} = \mathbf{P}f(\mathbf{A}, \mathbf{X})$.
    - **A MLP is neither permutation-invariant nor permutation-equivariant**.

1.13.3.2 GNN design principles

- Desiderata:
    - Ensure **permutation invariance** and **equivariance**.
        - By requiring PI and PE, the type of computation that the NN can performed is hugely restricted.
    - Use **layers** as reusable, permutation-equivariant units.
        - Stacking layers will not break permutation invariance.

        - **Node-level predictions** will be automatically permutation-invariant.
    - Design layers as **flexible differentiable functions**.
    - Enable scalability with **parameter sharing** (to enable scalability to big data).
- A GNN is a **optimizable transformation** on all attributes of the graph (nodes, edges, global-context) **that preservers graph symmetries** (permutation invariances).
- *Goal*: design graph neural networks using permutation invariant/equivariant transformations and passing and aggregating information from neighbors.

1.13.4 Message Passing Framework

- Neighbour aggregation: generate node embeddings based on local network neighborhoods.
    - The neighborhood can be enlarged by apply the same produces for several steps.
    - NA can be described as a computational graph.
        - Computations can be done iteratively in steps (with parameter k for the neighbor depth).
        - Key distinctions are in how different approaches aggregate information across the layers.
- The MPF is a general way to define the update rule for each node.
    - A single GNN layer is considered.
- MPF consists of two main steps:
    - **Message computation**:
        - Each node v sends a message to its neighbors.
        - Based on its own features and the features of its neighbors.
    - **Message aggregation**:
        - Each node v aggregates messages received from its neighbors.
        - And updates its own features.
- For node v, $\mathbf{x}_v = \mathbf{h}_v^{(0)} \rightarrow \mathbf{h}_v^{(1)} \rightarrow ... \rightarrow \mathbf{h}_v^{(L)}$.

1.13.4.1 Message computation

- `def` **Message computation**: $m_u^{(l)} = MSG^{(l)}(\mathbf{h}_u^{(l)})$.
- The message computed at level v by the node u.
- $MSG^{(l)}$ is usually a (simple, `eg` one-layer) NN that:
    - Takes as input the features of the node u at level $l-1$.
    - Returns a message $\mathbf{m}_u^{(l)}$.
    - `eg` $\mathbf{m}_u^{(l)} = f_{\mathbf{W}^{(l)}}(\mathbf{h}_u^{(l-1)}) = \text{ReLU}(\mathbf{W}^{(l)}\mathbf{h}_u^{(l-1)})$.
- `prop` Properties of message computation:
    - The **parameters** of the message computation function $MSG^{(l)}$ are **shared across the nodes**.
    - The function is **permutation-equivariant**.
        - Because the same function is applied to all nodes in the graph independently.
    - Both properties are crucial to ensure that the GNN can be applied to graphs of different sizes.

1.13.4.2 Message aggregation

- `def` **Message aggregation**: $\mathbf{h}_v^{(l)} = \text{Agg}^{(l)}(\{\mathbf{m}_u^{(l)}, \forall u \in N(v)\})$.
- Node v will aggregate the messages from its neighbors.
    - The aggregation function must be **permutation-invariant** (`eg` sum, mean, max).
    - Potential parameters associated with $\text{Agg}^{(l)}$ are shared across the nodes.
- Issue: information from node v itself could **get lost**.
    - Right now, the computation of $\mathbf{h}_v^{(l)}$ does not **directly** depend on $\mathbf{h}_v^{(l-1)}$.
    - Possible fix (not all GNN fix this):
        - Message: $\mathbf{m}_v^{(l)} = \mathbf{B}^{(l)}\mathbf{h}_v^{(l-1)}$.
            - Compute message from node v itself.
        - Aggregation: $\mathbf{h}_v^{(l)} = \text{Agg}^{(l)}(\{\mathbf{m}_u^{(l)}, \forall u \in N(v)\}, \mathbf{m}_v^{(l)})$.
            - Aggregates the message from node v itself after aggregating from neighbors.
            - `eg` Sum aggregation: $\mathbf{h}_v^{(l)} = \mathbf{m}_v^{(l)} + \sum_{u \in N(v)} \mathbf{m}_u^{(l)}$.

1.13.5 Graph Convolutional Networks

- A type of NN designed to operate on graph-structured data.
    - It generalizes the concept of **convolution** from traditional grid data to graph data.

- It **aggregates feature information** from a node's **local neighborhood**.
  - Each layer updates the feature representation of each node.
    - By combining its own features with the features of its neighbors.
  - It does so by **multiplying** the **adjacency matrix** $\mathbf{A}$ $(n \times n)$ with the **feature matrix** $\mathbf{H}^{(l)}$ $(n \times m)$.
    - For each node u and feature f the value $(\mathbf{AH})_{uf} = \sum_{k \in N(u)} A_{uk} H_{kf}$ contains:
      - The sum of f of all the nodes in the graph, weighted by edges connecting u to those nodes.
      - Not connected nodes won't contribute to the sum via multiplication via $\mathbf{A}$ (zero values).
    - With weights 0 or 1, $(\mathbf{AH})_{uf} = \sum_{k \in N(u)} A_{uk} H_{kf} \rightarrow \sum_{k \in N(u)} \mathbf{H}_k = \sum_{k \in N(u)} \mathbf{h}_k$.

1.13.5.1 Normalization

- To **avoid numerical instability**, the adjacency matrix $\mathbf{A}$ can be **normalized**.
  - By dividing each row by the sum of its elements.
- `def` **Normalized adjacency matrix**: $\tilde{\mathbf{A}} = \mathbf{D}^{-1}\mathbf{A}$.
  - $\tilde{A}_{uv} = \frac{A_{uv}}{deg(u)} = \frac{A_{uv}}{|N(u)|}$.
  - $\mathbf{D}$: the diagonal degree matrix, $D_{uu} = \sum_v A_{uv}$.
- `def` **Symmetric normalization**: $\tilde{\mathbf{A}} = \mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$.
  - $\tilde{A}_{uv} = \frac{A_{uv}}{\sqrt{D_{uu}D_{vv}}} = \frac{A_{uv}}{\sqrt{deg(u)deg(v)}} = \frac{A_{uv}}{\sqrt{|N(u)||N(v)|}}$.
  - $\mathbf{D}^{-1/2}$: the **diagonal matrix** with the inverse of the square root of the diagonal elements of $\mathbf{D}$.
  - Enhancement over the previous solution, standard nowadays for GCN.

1.13.5.2 GCN layers

- `def` **GCN layer**: $\mathbf{H}^{(l+1)} = \sigma(\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}\mathbf{H}^{(l)}\mathbf{W}^{(l)})$.
  - $\tilde{\mathbf{A}} = \mathbf{A} + \mathbf{I}$: adjacency matrix with added **self-loops**.
    - Self-loops are used to store information from the node itself (desiderata in message aggregation).
  - $\tilde{\mathbf{D}}$: diagonal degree matrix of $\tilde{\mathbf{A}}$, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$.
  - $\tilde{\mathbf{D}}^{-1/2}\tilde{\mathbf{A}}\tilde{\mathbf{D}}^{-1/2}$: normalized $\tilde{\mathbf{A}}$.
  - $\mathbf{W}^{(l)}$: learnable weight matrix.
  - σ: activation function (`eg` ReLU).
- Benefits:
  - **Locality**: leveraging local graph structure makes them effective for tasks like node classification and link prediction.
  - **Scalability**: by stacking multiple GCN layers, information from larger neighborhoods can be captured.
- Everything can be described with the **MPF**:
  - $\mathbf{h}_u^{(l)} = \sigma(\sum_{v \in N(u) \cup \{u\}} \mathbf{W}^{(l)^T} \frac{\mathbf{h}_v^{(l-1)}}{\sqrt{|N(v)||N(u)|}})$.
  - $\sum_{v \in N(u) \cup \{u\}}$ performs **aggregation**, while $\mathbf{W}^{(l)^T} \frac{\mathbf{h}_v^{(l-1)}}{\sqrt{|N(v)||N(u)|}}$ performs **message computation**.

1.13.6 Training a GNN

- `def` **Last layer node embeddings**: $\mathbf{z}_v = \mathbf{h}_v^L = ENC_\theta(\mathbf{A}, \mathbf{x}_v)$.
  - Given a GNN with L layers and a GNN model ENC_θ (`eg` GCN, GraphSAGE, GAT).
- On top of the GNN, **further layers** (`eg` fully connected layers) are defined to perform **downstream tasks**.
  - $\mathbf{y}_v = DEC_{\theta'}(\mathbf{z}_v)$.
- To train the GNN, a **loss function** is defined and optimized via gradient-based methods.

1.13.6.1 Supervised learning with GNNs

- A **node-level task** is assumed.
- The goal is to minimize the following generic loss function: $\min_{\theta,\theta'} \sum_v \mathcal{L}(\mathbf{y}_v, DEC_{\theta'}(ENC_\theta(\mathbf{A}, \mathbf{x}_v))$.
  - This loss can be optimized using gradient-based methods.
- The examples are the **nodes of the graph**.
  - To estimate the generalization error, a separate test set of nodes is kept.
  - The model is trained on the remaining nodes.

1.13.6.2 Model parameters

- In a GNN model the parameters are:
  - The parameters of the message computation and aggregation function.
  - The parameters for the downstream task (eg fully connected layers).
- In **Graph convolutional networks** the parameters are:
  - The weight matrices $\mathbf{W}^{(l)}$.
  - The parameters for solving the downstream task.

1.13.6.3 Stacking GNN layers

- In practice, multiple GNN layers can be stacked to **enhance model expressivity**.
  - But GNNs suffer from over-smoothing problem.
- **Over-smoothing problem**: all the node embeddings converge to the **same value**.
  - It's not possible to capture the structural properties of the graph.
  - The model is not able to distinguish between different nodes.
- Stacking multiple layers corresponds **enlarging the receptive field of a node**.
  - The set of nodes that influence the node's embedding.
  - Shared neighbors quickly grows when the number of hops.
    - Since the node embeddings is determined by its receptive field.
      - If two nodes have highly-overlapped receptive fields, then their **embeddings are highly similar**.
  - Stack **many** GNN layers $\implies$ nodes will have **highly overlapped** receptive fields.
    - $\implies$ node embeddings will be **highly similar** $\implies$ suffer from the **over-smoothing problem**.

Enhancing GNN expressivity

- **Enhancing GNN expressivity** (main strategies):
  - **Increase the expressive power within each GNN layer**.
    - Message computation and aggregation become more complex.
  - **Add layers that do not pass messages**.
    - Add more layers before and after the GNN layers.
  - **Skip connections**.
    - Node embeddings in earlier GNN layers can sometimes better differentiate nodes.
    - Shortcuts are therefore added in the GNN.
    - Usually skip connection is implemented by summing the previous embeddings to the current ones.

2 Exercises

2.1 Mathematical foundations

2.1.1 Example 1 - Linear transformation

- **Linear transformation**:
  - $det(A) = a_1 \cdot a_4 - a_2 \cdot a_3 = 2 * 2 - 4 * 1 = 0$.
  - How does it affects the vector spaced composed by $V_1 = [1, 0]$ and $V_2 = [0, 1]$?
    - $(A \cdot V_1)_{1,1} = 2$
    - $(A \cdot V_1)_{2,1} = 1$
    - $(A \cdot V_2)_{1,1} = 4$.
    - $(A \cdot V_2)_{2,1} = 2$
    - Both vectors lies on the same line (a dimension is lost).

2.1.2 Example 2 - Properties of derivatives

- Properties of derivaties:
  - Power rule: $(x^4)' = 4x^3$.
  - Linearity: $(3\sin(x) + x^2)' = 3(\sin(x))' + (x^2) = 3\cos(x) + 2x$.
  - Chain rule: $(\sin(x^2))' = \cos(x^2)(x^2)'2\cos(x^2)x$.
  - Product rule: $(x^2x^3)' = 2x(x^3) + x^2(3x^2) = 5x^4 = (x^5)'$.
  - Quotient rule: $(\frac{x^5}{x^2})' = \frac{5x^4(x^2) - x^5(2x)}{x^4} = \frac{3x^6}{x^4} = 3x^2 = (x^3)'$.

2.1.3 Example 3 - Partial derivatives

- Evaluate the derivative of z w.r.t. t:
  - Where $(x, y) = (t^2, t)$ and $z = f(x, y) = x^2y^2$.
  - Then $\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{dx}{dt} + \frac{\partial z}{\partial y}\frac{dy}{dt} = 2xy^2 \cdot 2t + 2x^2t \cdot 1 = 4t^5 + 2t^5 = 6t^5$.
  - The same results could have been obtained simply evaluating $\frac{d}{dt}t^6 = 6t^5$.
    - By noticing that $f(x, y) = (x(t))^2 \cdot (y(t))^2 = (t^2)^2 \cdot (t)^2 = t^4t^2 = t^6$.

2.1.4 Example 4 - Discrete probability distributions

- **Discrete probability distributions**:
  - A random variable x taking k possible values $x_1, \dots, x_k$ is considered.
  - The PMF $P(\text{x}) = \frac{1}{k}$ (**uniform probability**) is a valid PMF if:
    - It is defined over all possible states of x.
    - $\forall x \in \text{x} : 0 \leq P(x) = \frac{1}{k} \leq 1$.
    - $P(\Omega) = \sum_{x \in \{x_1, \dots, x_k\}} P(x) = \sum_{x \in \{x_1, \dots, x_k\}} \frac{1}{k} = 1$.
  - Thanks to other properties it can be evaluated:
    - $P(\{x_1, x_2, x_3\} \cup \{x_4, x_5\}) = P(\{x_1, x_2, x_3\}) + P(\{x_4, x_5\}) = \frac{3}{k} + \frac{2}{k} = \frac{5}{k}$.
    - $P(\Omega - \{x_1, x_2\}) = 1 - \frac{2}{k} = \frac{k-2}{k} = P(\{x_3, \dots, x_k\})$.

2.1.5 Example 5 - Continuous probability distributions

- **Continuous probability distributions**:
  - Consider a uniform distribution on an interval of the real numbers: $\text{x} \sim U(a, b)$.
    - Where a and b are the endpoints of the interval, with $b > a$.
  - The PDF $u(x; a, b)$ of the uniform distribution is:
    - $\frac{1}{b-a}$ if $x \in [a, b]$.
    - 0, otherwise.
  - The PDF $u(x; a, b)$ is a valid PDF if:
    - It is defined over all possible states of x.
    - $\forall x \in \text{x} : p(x) \geq 0$.
    - $\int_{-\infty}^{\infty} u(x; a, b)\, dx = \int_a^b \frac{1}{b-a} dx = \frac{1}{b-a}\int_a^b 1\, dx = \frac{1}{b-a} \cdot (x|_a^b) = 1$.
      - $(x|_a^b) = (b - a)$.

2.2 Introduction to neural networks

2.2.1 Example 1 - Perceptron

- **Perceptron**:
  - Data: $[1,1,1]$, $[1,1,0]$, $[1,0,1]$, $[1,0,0]$ (each x_0 is the bias).
  - Given the following weights, the OR function is computed: $w_0 = 0, w_1 = 0.5, w_2 = 0.5$.
    - Expected output: -1 with $(0,0)$, 1 for $(0,1)$, $(1,0)$, $(1,1)$.
    - $x_0 \cdot w_0 + x_1 \cdot w_1 + x_2 \cdot w_2$:
      - $[1,1,1]$: $1 \cdot 0 + 1 \cdot 0.5 + 1 \cdot 0.5 = 1 > 0 \rightarrow y = 1$.
      - $[1,1,0]$: $1 \cdot 0 + 1 \cdot 0.5 + 0 \cdot 0.5 = 0.5 > 0 \rightarrow y = 1$.
      - $[1,0,1]$: $1 \cdot 0 + 0 \cdot 0.5 + 1 \cdot 0.5 = 0.5 > 0 \rightarrow y = 1$.
      - $[1,0,0]$: $1 \cdot 0 + 0 \cdot 0.5 + 0 \cdot 0.5 = 0 \leq 0 \rightarrow y = -1$.
  - Given the following weights, the AND function is computed: $w_0 = -1, w_1 = 1, w_2 = 1$.
    - Expected output: 1 with $(1,1)$, -1 for $(0,0)$, $(0,1)$, $(1,0)$.
    - $x_0 \cdot w_0 + x_1 \cdot w_1 + x_2 \cdot w_2$:
      - $[1,1,1]$: $1 \cdot (-1) + 1 \cdot 1 + 1 \cdot 1 = 1 > 0 \rightarrow y = 1$.
      - $[1,1,0]$: $1 \cdot (-1) + 1 \cdot 1 + 0 \cdot 1 = 0 \leq 0 \rightarrow y = -1$.
      - $[1,0,1]$: $1 \cdot (-1) + 0 \cdot 1 + 1 \cdot 1 = 0 \leq 0 \rightarrow y = -1$.
      - $[1,0,0]$: $1 \cdot (-1) + 0 \cdot 1 + 0 \cdot 1 = -1 \leq 0 \rightarrow y = -1$.
    - Another set of weights which compute the AND function is: $w_0 = -0.8, w_1 = 0.5, w_2 = 0.5$.
  - Just by modifying the weights, the computed function is completely different.

2.2.2 Example 2 - Perceptron Learning Algorithm

- Perceptron Learning Algorithm:
  - Input data: $([1,1,1],1)$, $([1,1,0],-1)$, $([1,0,1],-1)$, $([1,0,0],-1)$.
  - Execution:
    - $w(0) = [0,0,0]$ (given this time, but usually randomized).
    - $\eta = 0.5$.
    - Examples can be evaluated in any orders.
    - $n = 0$, with $([1,1,1],1)$, netinput 0, net output -1, but desired output 1 (misclassification).
      - $w(n+1) = w(n) + \eta d(n) x(n)$.
      - $w(1) = [0,0,0] + 0.5 \times [1,1,1] = [0.5, 0.5, 0.5]$.
      - Consider each pattern and consider if it is correctly predicted.
        - $([1,1,1],1)$ is okay, $([1,1,0],-1)$ is not.
    - $n = 1$, with $([1,1,0],-1)$, netinput 1, net output 1, but desired output -1 (misclassification).
      - $w(2) = [0.5,0.5,0.5] - 0.5 \times [1,1,0] = [0,0,0.5]$.
      - Consider each pattern and consider if it is correctly predicted.
        - $([1,1,1],1)$ and $([1,1,0],-1)$ are okay, $([1,0,1],-1)$ is not.
    - $w(2) = [0.5,0.5,0.5] - 0.5[1,0,0] = [0,0.5,0.5]$.
    - $w(3) = [0,0.5,0.5] - 0.5[1,1,0] = [-0.5,0,0.5]$.
    - $w(4) = [-0.5,0,0.5] + 0.5[1,1,1] = [0,0.5,1]$.
    - $w(5) = [0,0.5,1] - 0.5[1,0,1] = [-0.5,0.5,0.5]$ (solution).
    - The AND problem is **linearly separable**.
      - The convergence theorem provides that a solution will be found.
      - In this example and given the patterns evaluation order, in $n = 5$ iterations.

2.2.3 Exercise 1 - Sigmoid neurons

- **Sigmoid neurons simulating perceptrons**:
  - Suppose to multiply all the weights and biases in a network of perceptrons by a positive constant $c > 0$.
    - Show that the behavior of the network doesn't change.
  - Suppose that the overall input to the same network has been chosen (fixed input).
    - Suppose that $\mathbf{w} \cdot \mathbf{x} + b \neq 0$ to any particular perceptron in the network.
    - Then replace all the perceptrons in the network by **sigmoid neurons**.
      - And multiply the weights and biases by a positive constant $c > 0$.
    - Show that in the limit as $c \rightarrow \infty$, this new network behaves the same as the original one.
      - How can this fail when $\mathbf{w} \cdot \mathbf{x} + b = 0$ for one of the perceptrons?

2.2.3.1 Solution

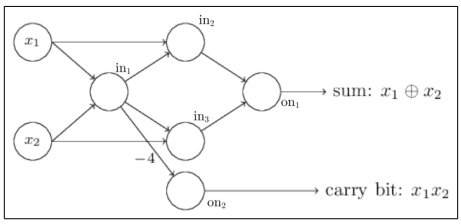

Figure 8: NN-based binary adder with carry (default weight: -2).

- Perceptron network:
  - After the multiplication the equation is $\mathbf{w}c \cdot \mathbf{x} + bc$.
  - The *binary adder with carry* perceptron network is taken as reference.
  - The chosen positive constant is $c = 4$.
    - With $x_1 = 0$ and $x_2 = 1$:
      - in_1: with $\mathbf{x} = [x_1, x_2] = [0, 1]$, $0 \cdot (-2 \cdot 4) + 1 \cdot (-2 \cdot 4) + (3 \cdot 4) = 4 > 0 \rightarrow y_{in_1} = 1$.
      - in_2: with $\mathbf{x} = [x_1, y_{in_1}] = [0, 1]$, $0 \cdot (-2 \cdot 4) + 1 \cdot (-2 \cdot 4) + (3 \cdot 4) = 4 > 0 \rightarrow y_{in_2} = 1$.
      - in_3: with $\mathbf{x} = [x_2, y_{in_1}] = [1, 1]$, $1 \cdot (-2 \cdot 4) + 1 \cdot (-2 \cdot 4) + (3 \cdot 4) = -4 \leq 0 \rightarrow y_{in_3} = 0$.
      - on_1: with $\mathbf{x} = [y_{in_2}, y_{in_3}] = [1, 0]$, $1 \cdot (-2 \cdot 4) + 0 \cdot (-2 \cdot 4) + (3 \cdot 4) = 4 > 0 \rightarrow y_{on_1} = 1$.
      - on_2: with $\mathbf{x} = [y_{in_1}] = [1]$, $1 \cdot (-4 \cdot 4) + (3 \cdot 4) = -4 \leq 0 \rightarrow y_{on_2} = 0$.
      - *Output*: $\mathbf{y} = [1, 0]$, where $0 \oplus 1 = 1$ with carry $= 0$ (correct).
    - With $x_1 = 1$ and $x_2 = 1$:
      - in_1: with $\mathbf{x} = [x_1, x_2] = [1, 1]$, $y_{in_1} = 0$.
      - in_2: with $\mathbf{x} = [x_1, y_{in_1}] = [1, 0]$, $y_{in_2} = 1$.
      - in_3: with $\mathbf{x} = [x_2, y_{in_1}] = [1, 0]$, $y_{in_3} = 1$.
      - on_1: with $\mathbf{x} = [y_{in_2}, y_{in_3}] = [1, 1]$, $y_{on_1} = 0$.
      - on_2: with $\mathbf{x} = [y_{in_1}] = [0]$, $y_{on_2} = 1$
      - *Output*: $\mathbf{y} = [0, 1]$, where $1 \oplus 1 = 0$ with carry $= 1$ (correct).
    - With $x_1 = 1$ and $x_2 = 0$:
      - in_1: with $\mathbf{x} = [x_1, x_2] = [1, 0]$, $y_{in_1} = 1$.
      - in_2: with $\mathbf{x} = [x_1, y_{in_1}] = [1, 1]$, $y_{in_2} = 0$.
      - in_3: with $\mathbf{x} = [x_2, y_{in_1}] = [0, 1]$, $y_{in_3} = 1$.
      - on_1: with $\mathbf{x} = [y_{in_2}, y_{in_3}] = [0, 1]$, $y_{on_1} = 1$.
      - on_2: with $\mathbf{x} = [y_{in_1}] = [1]$, $y_{on_2} = 0$
      - *Output*: $\mathbf{y} = [1, 0]$, where $0 \oplus 1 = 1$ with carry $= 0$ (correct).
    - With $x_1 = 0$ and $x_2 = 0$:
      - in_1: with $\mathbf{x} = [x_1, x_2] = [0, 0]$, $y_{in_1} = 1$.
      - in_2: with $\mathbf{x} = [x_1, y_{in_1}] = [0, 1]$, $y_{in_2} = 1$.
      - in_3: with $\mathbf{x} = [x_2, y_{in_1}] = [0, 1]$, $y_{in_3} = 1$.
      - on_1: with $\mathbf{x} = [y_{in_2}, y_{in_3}] = [1, 1]$, $y_{on_1} = 0$.
      - on_2: with $\mathbf{x} = [y_{in_1}] = [1]$, $y_{on_2} = 0$
      - *Output*: $\mathbf{y} = [1, 0]$, where $0 \oplus 0 = 0$ with carry $= 0$ (correct).
    - All four cases match the original network and the corresponding *binary adder with carry* function.
  - When **multiplied by a positive constant**, the perceptron NN behaves the same as the original one.
- Sigmoid network:
  - The *binary adder with carry* perceptron network is taken as reference.
  - The chosen positive constant is $c = 1$ (same as perceptron NN, but with sigmoid neurons):
    - With $x_1 = 0$ and $x_2 = 1$:
      - in_1: with $\mathbf{x} = [x_1, x_2] = [0, 1]$, $0 \cdot -2 + 1 \cdot -2 + 3 = 1 \rightarrow y_{in_1} = \sigma(1) = 0.731$.
      - in_2: with $\mathbf{x} = [x_1, y_{in_1}] = [0, 0.731]$, $0 \cdot -2 + 0.731 \cdot -2 + 3 = 1.538 \rightarrow y_{in_2} = \sigma(1.538) = 0.823$.
      - in_3: with $\mathbf{x} = [x_2, y_{in_1}] = [1, 0.731]$, $1 \cdot -2 + 0.731 \cdot -2 + 3 = -0.462 \rightarrow y_{in_3} = \sigma(-0.462) = 0.386$.
      - on_1: with $\mathbf{x} = [y_{in_2}, y_{in_3}] = [0.823, 0.386]$, $0.823 \cdot -2 + 0.386 \cdot -2 + 3 = 0.582 \rightarrow y_{on_1} = \sigma(0.582) = 0.641$.
      - on_2: with $\mathbf{x} = [y_{in_1}] = [0.731]$, $0.731 \cdot -4 + 3 = 0.076 \rightarrow y_{on_2} = \sigma(0.076) = 0.518$.
      - *Output*: $\mathbf{y} = [0.641, 0.518]$, very distant from the actual solution $[1, 0]$.
  - The chosen positive constant is $c = 4$.
    - With $x_1 = 0$ and $x_2 = 1$:
      - in_1: with $\mathbf{x} = [x_1, x_2] = [0, 1]$, $0 \cdot (-2 \cdot 4) + 1 \cdot (-2 \cdot 4) + (3 \cdot 4) = 4 \rightarrow y_{in_1} = \sigma(4) = 0.982$.
      - in_2: with $\mathbf{x} = [x_1, y_{in_1}] = [0, 0.982]$, $0 \cdot (-2 \cdot 4) + 0.982 \cdot (-2 \cdot 4) + (3 \cdot 4) = 4.143 \rightarrow y_{in_2} =$

$\sigma(4.134) = 0.984$.
- in_3: with $\mathbf{x} = [x_2, y_{in_1}] = [1, 0.982]$, $1 \cdot (-2 \cdot 4) + 0.982 \cdot (-2 \cdot 4) + (3 \cdot 4) = -3.872 \rightarrow y_{in_3} = \sigma(-3.872) = 0.020$.
- on_1: with $\mathbf{x} = [y_{in_2}, y_{in_3}] = [0.984, 0.020]$, $0.984 \cdot (-2 \cdot 4) + 0.020 \cdot (-2 \cdot 4) + (3 \cdot 4) = 3.968 \rightarrow y_{on_1} = \sigma(3.968) = 0.981$.
- on_2: with $\mathbf{x} = [y_{in_1}] = [0.982]$, $0.982 \cdot (-4 \cdot 4) + (3 \cdot 4) = -3.712 \rightarrow y_{on_2} = \sigma(-3.712) = 0.024$.
- *Output*: $\mathbf{y} = [0.981, 0.024]$, closer but still not exact from the actual solution $[1, 0]$.

- The larger chosen positive constant is $c = 123456$.
  - With $x_1 = 0$ and $x_2 = 1$:
    - in_1: with $\mathbf{x} = [x_1, x_2] = [0, 1]$, $y_{in_1} = \sigma(123456) = 1.\bar{0}$.
    - in_2: with $\mathbf{x} = [x_1, y_{in_1}] = [0, 1]$, $y_{in_2} = \sigma(123456) = 1.\bar{0}$.
    - in_3: with $\mathbf{x} = [x_2, y_{in_1}] = [1, 1]$, $y_{in_3} = \sigma(-123456) \approx 0$.
    - on_1: with $\mathbf{x} = [y_{in_2}, y_{in_3}] = [1, 0]$, $y_{on_1} = \sigma(123456) = 1.\bar{0}$.
    - on_2: with $\mathbf{x} = [y_{in_1}] = [1]$, $y_{on_2} = \sigma(-123456) \approx 0$.
    - *Output*: $\mathbf{y} = [1, 0]$, where $0 \oplus 1 = 1$ with carry $= 0$ (correct).
  - Even though this is a partial solution, it's clear that with $c \rightarrow \infty$ the **original function is approximated**.

2.2.4 Example 3 - Weights representation

- **Weights representation**:
  - The third (and output) L of a NN has two neurons.
  - Its weights vector is then (with a previous layer with three neurons): $\mathbf{W}^3 = \begin{pmatrix} w^3_{11} & w^3_{12} & w^3_{13} \\ w^3_{21} & w^3_{22} & w^3_{23} \end{pmatrix}$.
  - $\mathbf{W}^3 \mathbf{z}^{l-1} = \begin{pmatrix} w^3_{11} & w^3_{12} & w^3_{13} \\ w^3_{21} & w^3_{22} & w^3_{23} \end{pmatrix} \cdot \begin{bmatrix} z^2_1 \\ z^2_2 \\ z^3_2 \end{bmatrix} = \begin{bmatrix} w^3_{1\cdot} \cdot \mathbf{z}^2 \\ w^3_{2\cdot} \cdot \mathbf{z}^2 \end{bmatrix}$.

2.2.5 Example 4 - First equation of BP

- Computing BP1 using the **quadratic cost**:
  - $C = \frac{1}{2} \sum_j (y_j - z^L_j) \implies \frac{\partial C}{\partial z^L_j} = (z^L_j - y_i)$.
    - C is actually $C_{\mathbf{x}}$.
    - $\frac{1}{2} \|y(x) - z^L(x)\|^2_2 = [y(x) - z^L(x)]^T [y(x) - z^L(x)] = \sum_j (y_j(x) - z^L_j(x))(y_j(x) - z^L_j(x))$.
      - The goal is to compute the derivative of C w.r.t. z^L_j (consider everything else as a constant).
      - In $\frac{1}{2}((y_1 - z^L_1)^2 + \cdots + (y_j - z^L_j)^2 + \dots)$ the only term dependent on z^L_j is $(y_j - z^L_j)^2$.
    - $\frac{\partial C}{\partial z^L_j}(y_j - z^L_j)^2 = 2(y_j - z^L_j) \cdot -1 = 2(z^L_j - y_j)$.
      - -1 is the derivative of $y_j - z^L_j$ w.r.t. z^L_j.
      - The $\frac{1}{2}$ in C cancels out with the 2 in $2(z^L_j - y_j)$.
    - $\frac{\partial C}{\partial z^L_j} = (z^L_j - y_i)$.
      - To compute z^L_j, the information is conveyed through the NN up to layer L.
      - While y_j is a given constant (a value, 0 or 1).
      - Therefore $\frac{\partial C}{\partial z^L_j} = (z^L_j - y_i)$ is a simple value.
  - As it is $\delta^L_j = \frac{\partial C}{\partial z^L_j} \sigma'(a^L_j)$, therefore BP1 is easily computable.

2.3 Hopfield networks

2.3.1 Exercise 1 - Hopfield networks

2.3.1.1 Solution

Information withdrawal phase

- Compute the stable state of the above network with the initial state $[-1, -1, 1]$:
  - Initial state:

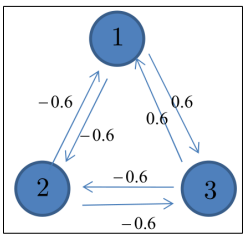

Figure 9: Hopfield networks for the example.

- $y_1(0) = -1, y_2(0) = -1, y_3(0) = 1$.
- The neuron are picked randomly:
  - $v_1(1) = -0.6 \cdot -1 + 0.6 \cdot 1 = 1.2 \rightarrow \varphi(v_1(1)) = 1 \rightarrow [1, -1, 1]$.
  - $v_2(2) = -0.6 \cdot 1 + -0.6 \cdot 1 = -1.2 \rightarrow \varphi(v_2(2)) = -1 \rightarrow [1, -1, 1]$.
  - $v_3(3) = 0.6 \cdot 1 + -0.6 \cdot -1 = 0 \rightarrow \varphi(v_3(3)) = -1 \rightarrow [1, -1, 1]$.
  - $v_1(4) = -0.6 \cdot -1 + 0.6 \cdot 1 = 1.2 \rightarrow \varphi(v_1(4)) = 1 \rightarrow [1, -1, 1]$.
- Stable state: $[1, -1, 1]$ (output of the network).
  - With a different evaluation order, the stable state can be different.
  - The other stable state is $[-1, 1, -1]$ (opposite of the one found, property of HN).

Storage phase

- Compute the weights with the initial memory $f_1 = [1, -1, 1]$:
  - $w_{12} = 1 \cdot (-1) = -1 = w_{21}$.
  - $w_{23} = (-1) \cdot 1 = -1 = w_{32}$.
  - $w_{31} = 1 \cdot 1 = 1 = w_{13}$.
- Compute the weights with the initial memories $f_1 = [1, -1, 1]$ and $f_2 = [1, 1, 1]$:
  - $w_{12} = \frac{1 \cdot -1 + 1 \cdot 1}{2} = 0 = w_{21}$.
  - $w_{23} = \frac{-1 \cdot 1 + 1 \cdot 1}{2} = 0 = w_{32}$.
  - $w_{31} = \frac{1 \cdot 1 + 1 \cdot 1}{2} = 1 = w_{13}$.
- Apply the memorization rule so that $[1, -1, 1]$ and $[-1, 1, -1]$ are stable states.
  - $w_{12} = \frac{1 \cdot -1 + -1 \cdot 1}{2} = -1 = w_{13}$.
  - $w_{23} = \frac{1 \cdot -1 + -1 \cdot 1}{2} = -1 = w_{32}$.
  - $w_{31} = \frac{1 \cdot 1 + -1 \cdot -1}{2} = 1 = w_{13}$.

2.3.2 Exercise 2 - Restricted Boltzmann machines

- Calculate $p(h_i = 1)$:
  - Visible layer composed by v_1 and v_2 of values: 1 and 0.
  - Hidden layer composed by a single node h_1 with $w_{11} = 0.01$ and $w_{21} = 0.02$.
    - $\Delta E_i = 1 \cdot 0.01 + 0 \cdot 0.02 = 0.01$.

2.3.2.1 Solution

- Two ways of computing ΔE_i:
  - Computing $E(h_1 = 1)$ and $E(h_1 = 0)$ and then $\Delta E_i = E(s_i = 0) - E(s_i = 1)$.
  - Or Computing $\Delta E_i = \sum w_{ij} S_j = 1 \cdot 0.01 + 0 \cdot 0.02 = 0.01$.
  - Then $p(h_1 = 1)$ can be computed: $p(h_1 = 1) = 0.50251$.
- Sampling is used to attribute a state (0 and 1) using the probability.

2.3.3 Exercise 3 - Restricted Boltzmann machines

- Apply contrastive divergence by learning $[1, 0]$:
  - Given:
    - Visible layer composed by v_1 and v_2.
    - Hidden layer composed by a single node h_1 with $w_{11} = 1$ and $w_{21} = 1$.
    - $\varepsilon = 0.1$ (learning rate).
  - Assume that sampling will output 1 in case $p(s = 1) \geq 0.5$, 0 otherwise.
    - In general, sampling will return 1 with probability p, hence not always, unless $p(s = 1) = 1$.

2.3.3.1 Solution

- Execution of contrastive divergence:
  - Epoch #1:
    - $t = 0$:
      - v_1 and v_2 are the ones in input.
      - h_1: $\Delta E_{h_1} = \sum_j w_{h_1 v_j} v_j = 1 * 1 + 1 * 0 = 1 \rightarrow p(h_1 = 1) = \frac{1}{1+e^{-\Delta E_{h_1}/T}} = 0.73$.
        - By sampling, with probability $0.73 \geq 0.5$, $h_1 = 1$.
      - $\langle v_1 h_1 \rangle^0 = 1 \cdot 1 = 1$ and $\langle v_2 h_1 \rangle^0 = 0 \cdot 1 = 0$.
    - $t = 1$:
      - To reconstruct v_1 and v_2, h_1 is the one computed at $t = 0$.
      - v_1: $\Delta E_{v_1} = \sum_j w_{v_1 h_j} h_j = 1 * 1 = 1 \rightarrow p(v_1 = 1) = \frac{1}{1+e^{-\Delta E_{v_1}/T}} = 0.73$.
        - By sampling, with probability $0.73 \geq 0.5$, $v_1 = 1$.
      - v_2: $\Delta E_{v_2} = \sum_j w_{v_2 h_j} h_j = 1 * 1 = 1 \rightarrow p(v_2 = 1) = \frac{1}{1+e^{-\Delta E_{v_2}/T}} = 0.73$.
        - By sampling, with probability $0.73 \geq 0.5$, $v_2 = 1$.
      - h_1: $\Delta E_{h_1} = \sum_j w_{h_1 v_j} v_j = 1 * 1 + 1 * 1 = 2 \rightarrow p(h_1 = 1) = \frac{1}{1+e^{-\Delta E_{h_1}/T}} = 0.88$.
        - By sampling, with probability $0.88 \geq 0.5$, $h_1 = 1$.
      - $\langle v_1 h_1 \rangle^1 = 1 \cdot 1 = 1$ and $\langle v_2 h_1 \rangle^1 = 1 \cdot 1 = 1$.
    - Weights update:
      - $\Delta w_{h_1 v_1} = \varepsilon(\langle v_1 h_1 \rangle^0 - \langle v_1 h_1 \rangle^1) = 0.1((1 \cdot 1) - (1 \cdot 1)) = 0$.
        - $w_{h_1 v_1}(2) = w_{h_1 v_1}(1) + \Delta w_{h_1 v_1} = 1 + 0 = 1$ (unchanged).
      - $\Delta w_{h_1 v_2} = \varepsilon(\langle v_2 h_1 \rangle^0 - \langle v_2 h_1 \rangle^1) = 0.1((0 \cdot 1) - (1 \cdot 1)) = -0.1$.
        - $w_{h_1 v_2}(2) = w_{h_1 v_2}(1) + \Delta w_{h_1 v_2} = 1 + (-0.1) = 0.9$.

2.4 Recurrent neural networks

2.4.1 Example 1 - Long Short Term Memory

- LSTM forget memory:
  - Let $\sigma = 1$ for arguments > 0, 0 otherwise.
  - Let's $W_f = [1, 1, 1, -10; 1, 1, 2, -10]$ and $b_f = 0$.
  - $[h_{t-1}, x_t] = [1, 1, 0, 1]$ and suppose $x_t = [0, 1]$ codifies ".", the dot character.
  - Then $f_t = [0, 0]$.
    - Intuitively (and informally) erases everything from C_{t-1}.
  - Since weights are learned via BP, gates are not as interpretable.