1 Connectionism

1.1 Background

Connectionism aims to build intelligent machines inspired by systems of interconnected neurons.

McChulloch & Pitts Neuron $\sim$ **1940:**

- A neuron with n inputs can be characterized as a threshold element with Boolean signature $\sigma \in \{-1,1\}^n$ and threshold θ.
- A neuron computes its action or output as a Boolean function of its input $x \in \{0,1\}^n$ according to $output = \begin{cases} 1 & \text{if } \sum_{i=1}^n \sigma_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$

Issue: The Boolean logic model of neural networks fails to capture biological systems' adaptive and learning capabilities.

Solution: Subsequent contributions (the Theseus maze-navigating machine and Minsky's SNARC), demonstrated early concepts of learning and adaptability but achieved limited practical impact.

1.2 Perceptron, 1958 and 1969

The perceptron marked a milestone in machine learning by introducing supervised learning for binary classification.

Setting: Feature vector $x \in \mathbb{R}^n$, class membership $y \in \{-1,+1\}$, sample of s training examples $\mathcal{S} = \{(x_i, y_i) : 1 \leq i \leq s\} \subset \mathbb{R}^n \times \{\pm 1\}$

Goal: Learn discriminant function f that classifies examples in $\mathcal{S}$

Model: Linear threshold unit with synaptic weights $w \in \mathbb{R}^n$ and threshold $b \in \mathbb{R}$. Formally, $f[w,b](x) = \text{sign}(x \cdot w + b)$, where $x \cdot w = \sum_{i=1}^n x_i w_i$. Here, $\text{sign}(0) = 0$

Decision Boundary: The decision boundary is the set of all x for which $\frac{x \cdot w}{\|w\|} + \frac{b}{\|w\|} = 0$. Note that this is the Hesse normal form of a hyperplane in $\mathbb{R}^n$ with normal vector $n = \frac{w}{\|w\|}$.

Distances:

- Signed distance from any point x_0 to hyperplane: $\frac{x_0 \cdot w + b}{\|w\|}$

Geometric Margin:

- **Definition:** $\gamma[w,b](x,y) = \frac{y(x \cdot w + b)}{\|w\|}$ (the sign of γ encodes the correctness of the classification + correct,- wrong)
- **γ-Separation:** A classifier $f[w,b]$ is said to γ-separate the sample $\mathcal{S}$, if $\forall (x,y) \in \mathcal{S} : \gamma[w,b](x,y) \geq \gamma$
- **Maximum Margin (MM) Classifier:** $f[w^*, b^*]$ is a classifier with maximal separation margin: $(w^*, b^*) \in \text{argmax}_{(w,b)} \gamma[w,b](\mathcal{S})$, where we define $\gamma[w,b](\mathcal{S}) = \min_{(x,y)\in\mathcal{S}} \gamma[w,b](x,y)$
- **Uniqueness of MM Classifier:** The MM classifier is unique, but its parameters are only unique up to positive scaling

Version Space: The version space is the set of all parameters of a classifier $f[\theta]$ that are compatible with the training sample (predict all labels correctly): $\mathcal{V}(\mathcal{S}) = \{\theta \mid \forall (x,y) \in \mathcal{S} : f[\theta](x) = y\}$.

Linear Separability: The version space of a linear threshold classifier is $\mathcal{V}(\mathcal{S}) = \{(w,b) : \gamma[w,b](\mathcal{S}) > 0\} \subseteq \mathbb{R}^{n+1}$. We call $\mathcal{S}$ linearly separable if and only if $\mathcal{V}(\mathcal{S}) \neq \emptyset$ ($\implies$ perceptron learning aims to find a perfect linear separator, and does not explicitly aim (and is not guaranteed) to find classifier with low training error).

Perceptron Learning:

- Present training examples repeatedly in some arbitrary order, making sure every example is repeated after finite number of steps.
- Follow the update rule: $f[w,b](x) \neq y \implies w \leftarrow w + yx, \quad b \leftarrow b + y$. Note how the perceptron only learns from mistakes!

Perceptron weights: $w^0 \in span(x_1, \ldots, x_s) \implies w^t \in span(x_1, \ldots, x_s), \forall t$

Convergence: General setting: Set $b = 0$, initialize $w^0 = 0$.

- **Lemma** Let w, $\|w\| = 1$, such that $\gamma[w](\mathcal{S}) = \gamma > 0$. Then $w^t \cdot w \geq t\gamma$.
- **Lemma** Let $R = max_{x\in\mathcal{S}} \|x\|$, then $\|w^t\| \leq R\sqrt{t}$
- **Theorem** Let $\mathcal{S}$ be γ-separable, $R = max_{x\in\mathcal{S}} \|x\|$. The perceptron converges in at most $\lfloor R^2/\gamma^2 \rfloor$ steps.
- **Exercise** Let (x^t, y^t) be sequence of mistakes, inducing updates $\Delta\theta^t$, $\theta^s = \sum_{t=1}^s \Delta\theta^t$. Then $\|\theta^s\|^2 \leq \sum_{t=1}^s \|x^t\|^2$.

Combinatorics and Geometry: General setting: Set $b = 0$.

- **Number of ways** to classify s points in n-space with linear classifier: $C(\mathcal{S}, n) = |\{y \in \{-1,1\}^s \mid \exists w \in \mathbb{R}^n : \forall (x,y) \in \mathcal{S} : y(x \cdot w) > 0\}|$
- **General position:** $\mathcal{X} = \{x_1, \ldots, x_s\} \subset \mathcal{R}^n$ are in general position if and only if $\forall \Xi \subseteq \mathcal{X}$ with $|\Xi| \leq n$: Ξ is linearly independent.
- **Cover's Theorem:** Let $\mathcal{S} \subset \mathbb{R}^n$ be a set of s points in general position. Then $C(s+1, n) = 2\sum_{i=0}^{n-1} \binom{s}{i}$
- **Corollary:** $C(s,n) = 2^s$ for $s \leq n$.
- **Asymptotic Shattering:** $\frac{C(s,n)}{2^s} = \begin{cases} 1 & \text{if } s \leq n, \\ 1 - \mathcal{O}(e^{-n}) & \text{if } n < s < 2n, \\ \frac{1}{2} & \text{if } s = 2n, \\ \mathcal{O}(e^{-n}) & \text{otherwise.} \end{cases}$
- **Interpretation:** There two growth regimes. One where the number of realizable labelings grows exponentially, and one where the growth function of realizable labelings starts go get constrained by geometry.

1.3 Parallel Distributed Processing

Layers and Activation Functions: The layer transfer function is $f(W,B)(x) = \Phi(Wx + B)$, $W = (w_1, \ldots, w_m)^\top$, $\Phi(z) = (\phi(z_1), \ldots, \phi(z_m))^\top$. Note how the linear threshold unit is a special case with $\phi = \text{sign}$.

Delta Rule:

- The original perceptron update rule has disadvantages:
  - $0/1$ classification error not differentiable at decision boundary and has zero derivatives elsewhere.
  - We do not learn anything from correctly classified points.
- The delta learning rule can be formulated as follows: $\Delta w_{i,j} = \eta \delta_i \phi'(w_i \cdot x) x_j$, $\delta_i = (y_i - f_i)$, $\eta > 0$

Remarks:

- Update of a weight vec. for a unit is always in dir. of a training point
- The update is scaled by the residual (or delta) δ_i of the unit
- The *sensitivity* of the unit factors in (derivative of activation)
- η is a design choice
- The delta rule is nothing but a gradient step with step size η, i.e., $\Delta w_{i,j} = -\eta \frac{\partial \ell}{\partial w_{i,j}}$ with squared loss $\ell = \frac{1}{2}\sum_{i=1}^m (\phi_i - y_i)^2$
- Generalized delta rule for deep architectures: $\delta_i = \frac{\partial \ell}{\partial \phi_i}$, derived by propagating error terms from the outputs back towards the inputs

Multi-Layer Perceptron: Particular neural network architecture that can be written as (for n inputs, m hidden units, and one real output): $f[v,W](x) = \sum_{i=1}^m v_i \phi_i$, $\phi_i = \phi(w_i \cdot x)$, $\phi(z) = \frac{1}{1+e^{-z}}$.

1.4 Hopfield Networks: Classical model of an associate memory

Idea: Define a parameterized energy function via second order interaction of n binary neurons:

$E(X) = -\frac{1}{2}\sum_{i\neq j} w_{i,j} X_i X_j + \sum_i b_i X_i$, $X_i \in \{-1,+1\}$, $1 \leq i \leq n$.

Where we constrain the weights as follows: $w_{i,j} = w_{j,i}$ $(\forall i,j)$, $w_{i,i} = 0$ $(\forall i)$. Think of the weights as quantifying the interaction strength between neurons, and of the biases as thresholds (set to zero in the following).

Dynamics: After initialization, the state of each neuron is iteratively (in some fixed order) updated as follows: $X_i \leftarrow \begin{cases} +1 \text{ if } E(\ldots, X_{i-1}, +1, X_{i+1}, \ldots) \leq E(\ldots, X_{i-1}, -1, X_{i+1}, \ldots) \\ -1 \text{ else} \end{cases}$,

where updates can be performed synchronous or asynchronous. **Note** that asynchronous updates guarantee that the energy function will never increase. **Note** after a finite number of steps, a fixed point is reached, which is the pattern of the Hopfield network.

Simplified update Rule: Instead of evaluating the $n(n+1)/2$ summands in E, we can just calculate n terms: $H_i = \sum_j w_{i,j} X_j$ and follow the update rule $X_i = \text{sign}(H_i)$. Here $\text{sign}(0) = 1$.

Hebbian Learning: Given a set $\mathcal{S}$ with patterns $x^t \in \{\pm 1\}^n$, $(1 \leq t \leq s)$. Then choose the weights as $w_{i,j} = \frac{1}{n}\sum_{t=1}^s x_i^t x_j^t$, which in matrix notation can be written as $W = \frac{1}{n}\sum_{t=1}^s x^t x^{t\top}$ ($\implies$ neurons that fire together wire together)

Memorization: For a pattern to be included in associative memory, the pattern must define a meta-stable state: $x_i^t = \text{sign}\left(\sum_{j=1}^n w_{i,j} x_j^t\right)$ (the dynamics will not diverge from the pattern).

Cross-talk term between patterns: Using the couplings learned with Hebbian learning, we get: $\sum_{j=1}^n w_{i,j} x_j^t = x_i^t + \frac{1}{n}\sum_{j=1}^n \sum_{r\neq t} x_i^r x_j^r x_j^t$, where $C_i^t = \frac{1}{n}\sum_{j=1}^n \sum_{r\neq t} x_i^r x_j^r x_j^t$ is called the cross-talk between patterns. **For x^t to be stable, we require this term to be strictly less than 1 in magnitude for all neurons.**

Capacity: Assume patterns to be random signs (probability of $\frac{1}{2}$ for -1 and $+1$). Assume that $n \to \infty$. Then $C_i^t \overset{\text{approx.}}{\sim} \mathcal{N}(0, \frac{s}{n})$. Then the error probability of any sign to be incorrectly flipped becomes: $P_e = \text{Prob}\{-x_i^t C_i^t\} \approx \int_1^\infty e^{-\frac{nz^2}{2s}} dz = \frac{1}{2}\left[1 - \text{erf}(\sqrt{n/2s})\right]$ ($\implies$ ratio s/n controls asympt. err. rate).
$\implies$ It can be shown that a phase transition occurs at $s/n \approx 0.138$. Beyond that limit many errors occur. If one requires that a pattern is retrieved with high probability, then one gets a sub-linear capacity bound of $s \leq \frac{n}{2\log_2 n}$

Spurious patterns: Simplest spurious pattern: sign inverted version of stable pattern is also stable as the network is sign symmetric.

2 Linear Networks

2.1 Regression Models

Least Squares Assume: $f[w](x) = w^\top x$

- Criterion (MSE): $h[w] = \frac{1}{2s}\sum_{i=1}^s (w^\top x_i - y_i)^2$
- Matrix form $h[w](\mathcal{S}) = \frac{1}{2s}\|Xw - y\|^2$, Normal eq.: $\nabla h = 2X^\top X - 2X^\top y$
- $w^* = X^* y \in argmin_w\, h[w]$ where X^* is the Moore-Penrose inverse of X $\equiv lim_{\delta\to 0}(X^\top X + \delta I)^{-1} X^\top$
- For very large datasets, SGD can be used to train: $w_{t+1} = w_t + \eta(y_{i_t} - w_t^\top x_{i_t}) x_{i_t}$ where $i_t \overset{iid}{\sim} Uniform(1, \ldots, s)$
- LS model $\equiv$ neg-log-likelihood function corresponds to Gaussian homoscedastic noise model: $y_i = w^\top x_i + \epsilon_i$, $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$
- LS assumes iid Gaussian noise on targets
- Regularized: $h_\lambda[w] \equiv h_{old}[w] + \frac{\lambda}{2}\|w\|^2$ s.t. $w^* = (X^\top X + \lambda I)^{-1} X^\top y$
- $\lambda \to 0 \equiv$ LS and l1 = Lasso, l2 = Ridge

Logistic function $\sigma(z) = \frac{1}{1+e^{-z}}$ $\sigma(z) + \sigma(-z) = 1$, as $\sigma(-z) = \frac{\exp^{-z}}{1+\exp^{-z}}$.

It means $\sigma - \frac{1}{2}$ is an odd function. Derivatives are polynomials in σ. $\sigma' = \sigma(1-\sigma)$, $\sigma'' = \sigma(1-\sigma)(1-2\sigma)$

σ is monotonically increasing, convex in $\mathbb{R}_-$ and concave in $\mathbb{R}_+$

- σ and $1 - \sigma$ are often interpreted as probabilities of Bernoulli event
- (Negative) logistic log likelihood with respect to binary outcome is equivalent to cross entropy with respect to degenerate target distribution: $\ell(y,z) = -ylog(\sigma(z)) - (1-y)log(1-\sigma(z))$
- $ln(\sigma(z)) = -ln(1+e^{-z})$, $(ln(\sigma(z)))' = 1 - \sigma$, $(ln(\sigma(z)))'' = -\sigma(1-\sigma) < 0$
- Implying: (1) $-log(\sigma)$ strictly convex (2) upper-bounds the heavy-side function (0-1 error)

Logistic Regression

- Minimizes logistic loss over $\mathcal{S} : h[w] = \frac{1}{s}\sum_{i=1}^s \ell(y_i, w^\top x_i)$
- No closed form solution, but can be optimized with SGD
- Iterates remain in the span of $\{w_0, x_1 \ldots, x_s\}$, and $\nabla \ell_i = (\sigma(w^\top x_i) - y_i) x_i$. **Note:** predicted probability minus 0/1 target appears in the gradient.

2.2 Layers & Units

- Layer F is a parametrized map (affine transformation + nonlinearity)
- Width m: $F : \underbrace{\mathbb{R}^{m(n+1)}}_{parameters} \times \underbrace{\mathbb{R}^n}_{input} \to \underbrace{\mathbb{R}^m}_{output}$; $F[\theta \equiv vec(W,b)](x) \equiv \phi(Wx + b)$
- **Composition** $G = F^L[\theta^L] \circ \ldots F^1[\theta^1]$, $F^l[\theta^l] = \phi^l(W^l x + b^l)$
- Computing $x^l = F^l(x^{l-1})$ is dominated by matrix-vector product
- **Units**: component functions of F parametrized by w and b defining family of functions $f[w,b](x) = \phi(w^\top x + b)$
- **Invar. of units**: $f(x) = f(x + \delta x)$ $\forall \delta x \perp w$. Hence, level sets of a unit are orthogonal to w: For $L_f(z) \equiv \{x : \phi(w^\top x + b) = z\}$ we have $w \perp L_f$ (nonlinearity only affects the rate of change in the direction of w)
- **Symmetry**: Units can be arbitrarily numbered: for Permutation matrix P, we have $F(x) = P^{-1}\phi(P(Wx + b)) \implies$ map invariant under simultaneous permutation of in- and out-weights of a layer.
- Also implies parametrization are never unique in feedforward network
- **Nesting**: Layer can be nested within layer of width $m+1$ by adding a unit with weight ($v = 0$) zero (barren), by having the same unit twice where the weights sum to old weight, or $w = b = 0$

Activation functions

- Linear: $\phi \equiv id$, Sigmoid: $\phi \equiv \sigma = \frac{1}{1+e^{-z}} \in (0;1)$, ReLU: $\phi \equiv max\{0, z\}$
- **Loss functions**: Squared $\frac{1}{2}\|\hat{y} - y\|^2$, binary CE loss (see below)
- CE between PMFs p and q is $-\sum_i p_i log q_i \to$ Define CE for $p = y \in$: $\{1, \ldots, m\}$ $\ell(y,q) = -log(q_y)$
- q_y outputs of Softmax $= \frac{e^{z_i}}{\sum_j e^{z_j}} \simeq$ probability of class i

CE loss can be written in terms of logits $z : \ell(y;z) = [-z_y + log\sum_j e^{z_j}]\frac{1}{ln2}$

- Train risk: $h[\theta](\mathcal{S})$. Population risk $E_P(h[\theta])$ where $(x,y) \sim P$: can be estimated with held-out data or CV.

2.3 Theorems and Results:

- Lin. networks *don't gain represnt. power from depth* $(W^2 W^1 := W)$
- Lin. layers can reduce rep. power: Project to a low-dim rep $(m_2 < m_1)$: $\to \text{rank}(F) = \text{rank}(W^L \ldots W^1) \leq \min_{l=0}^{L} m_l \leq n \to$ reduction in representational power is limited by the layer of smallest width
- **Linear. AE** attempt to learn identity to find an embedding by learning linear map $G(x) = VWx \to h(x) = \frac{1}{2}\|VWx - x\|^2$; $W, V^\top \in \mathbb{R}^{m\times n}$
- $rank(VW) \leq m$
- Any optimal W will map data n-space to subspace spanned by m-PCs of sample patterns

Residual Layers: Often, however non-trivial to learn identity. So, initialize layers around identity so re-learning not necessary $\to$ residual layer

- Add x to F output directly to ease learning of identity (Skip connection), often hard in DNNs $F[W,b](x) = x + [\phi(Wx + b) - \phi(0)]$, s.t. $F[0,0] = id$

Compositions with residual layers cause number of paths to grow exponentially: (2 layers) $F[\theta](x) = x + \phi(W^1 x) + \phi(W^2 x) + \phi(W^2\phi(W^1 x))$

- doubles to 8 summands for 3 layers and so on
- Finally note that as x does not depend on $[W,b]$: gradient map remains unchanged $\nabla F = \nabla\phi(Wx + b)$. Activations change though, naturally.
- Linear Projection can more easily preserve existing features. However, Residual layers do not change dimensionality as W is square matrix, so one often projects to allow flexibility of residual layers wrt to dimensions: $F[V,W,b] = \phi(Wx + b) + Vx$ where $V, W \in \mathbb{R}^{m\times n}$
- Normal res layer recovered for $n = m$ and $V = I$
- DenseNet $x^{l+1} = F^{l+1}(x^l, \ldots, x^0 = x)$. Issue: dimensionality $\uparrow$
- **Sigmoid Networks**: $\phi(z) = \sigma(z)$ (tanh(z)= $2\sigma(2z) - 1 \to$ tanh and σ lead to equally expressive networks, tanh preferred for sign symmetry)
- **Universal Approx:** Let $\mathcal{G}_n = span\{\sigma(w^\top x + b) | w \in \mathbb{R}^n, b \in \mathbb{R}\}$

$\to$ can one approximate any continuous functions arbitrarily well with sufficiently wide, single hidden layer MLPs? YES We want to obtain uniform guarantees for all points in a compact domain K. So, we use *uniform norm*. Let:

- $\|f\|_{\infty,K} \equiv \sup_{x\in K}|f(x)|$ and $d_K(f,g) := \|f - g\|_{\infty,K}$
- Assume we have a function class $\mathcal{G}$ with wich we want to approx. f over K. Then
- Approximation error of function class $\mathcal{G} : d_K(f,\mathcal{G}) \equiv \inf_{g\in\mathcal{G}} d_K(f,g)$
- If $d_K(f,\mathcal{G}) = 0 \implies f$ is approximated by $\mathcal{G}$ on K
- Alternatively, f approximated by $\mathcal{G}$ for $\exists\{g_1, \ldots\} \in \mathcal{G}$ then $\forall \epsilon > 0 : \exists M \geq 1 : \forall m \geq M : \|g_m - f\|_{\infty,K} < \epsilon$
- Func. class is approx by $\mathcal{G}$ over K if $\forall f \in \mathcal{F} : d_K(f,\mathcal{G}) = 0$

$\to$ If this holds for all compact sets K, then $\mathcal{G}$ is a univ. approxim. for $\mathcal{F}$

- A point x is in $cl(G)$, if for every neighborhood, there is a point G within an ϵ-distance of x $\to$ It must hold that $\mathcal{F} \subseteq cl(\mathcal{G})$ i.e. the largest function class that can be approximated by G is its closure
- $\to$ For every f and m there is a simple MLP of width m which will achieve error $\epsilon = \epsilon_m$ which can be made arbitrarily small by $\uparrow m$
- Def: Let $C(\mathbb{R}^n) = \{f : \mathbb{R}^n \to \mathbb{R} \mid f \text{ is continuous on } \mathbb{R}^n\}$.
- **Weierstrass Thrm**: Polynomials are univers. approxim. of $C(\mathbb{R})$
- **Smooth func. Approx:** Let $\phi \in C^\infty(\mathbb{R}) \equiv$ all ∞ differentiable functions, but not a polynomial $\to \text{span}(\{\phi(ax + b) | a, b \in \mathbb{R}\})$ univ. approx. $C(\mathbb{R})$
  - For n=1, an MLP with smooth activ. σ which **isn't a polynomial** is a univ. approx.
- **Lifting Lemma:** Let ϕ be s.t. $\text{span}(\{\phi(ax + b) | a, b \in \mathbb{R}\})$ univer. approx. $C(\mathbb{R})$ then $\text{span}(\{\phi(w^\top x + b) | w \in \mathbb{R}^n, b \in \mathbb{R}\})$ univers. approx. $C(\mathbb{R}^n)$
- **For any smooth act. functions except polynomials, MLPs are universal approxim.** (no upper bounds on necessary width m though)
  - Single hidden layer needed, addit. layers could reduce error quicker
  - Why not polynomials? span of ϕ : degree k polynomial is also degree k polynomials. In other words: Polynomial Activation $\implies$ Polynomial Network. With L layers, entire network is polynomial in $\mathbf{x}$ of finite degree k^L and uni-variate finite degree polynomials are not dense in $C(\mathbb{R}^n)$ en compacta!
- *How many units needed for approx acc.* ? Answer: **Barron's Thrm** for any f s.t. : $\int_{\mathbb{R}^n} |f(x)| dx < \infty$ let us define Fourier transform: $\hat{f}(\omega) = \int_{\mathbb{R}^n} e^{-2\pi i\omega' x} f(x) dx$, $\hat{f} : \mathbb{R}^n \to \mathbb{C}$. Barron Thrm applies for f fulfilling regularity condition $C_f := \int \|\omega\| |\hat{f}(\omega)| d\omega < \infty$ and r>0
- Then $\exists$ sequence of one hidden layer MLPs $(g_m)_{m\in\mathbb{N}}$ s.t. $\int_{r\mathbb{B}}(f(x) - g_m(x))^2 \mu(dx) \leq \mathcal{O}(\frac{1}{m})$; $r\mathbb{B} := \{x \in \mathbb{R}^n : \|x\| \leq r\}$ and μ is a meas. on $r\mathbb{B}$
- So, a single hidden layer MLP can approximate functions with finite C_f at order $\frac{1}{m}$ and independently of input dimensions n and data distribution!!
- Approximation error of MLPs: $\propto \frac{1}{m}$
- MLPs avoid the curse of dimensionality (if $C_f < \infty$)and can increase convergence by increasing width!
  - However, methods that use fixed set of m basis fct. $\to$ best approx. order is $(\frac{1}{m})^{\frac{2}{n}}$

2.4 ReLU Networks $\phi(z) = max\{0, z\}$

- Leaky relu to obtain gradient info in zero-branch: $\varepsilon w'x$ for $x \leq 0$ for $\varepsilon \in \mathbb{R}$
- Let $\Theta(Wx + b) \in \{0,1\}^m$, be the binary activation pattern of ReLu units for $\Theta :=$ Heavyside function
- Space can be partitioned into $X_0 \equiv \{x : \Theta(Wx + b) = 0\}$ and $X_1 \equiv \cdots = 1 \to$ restricted to any $X_{0/1}$, ReLU layer map is affine
- How expressive are ReLU Layers? Answer:**Thrm (Zaslavsky):** Let $\mathcal{H}$ be a set of m hyperplanes in $\mathbb{R}^n$ and $R(\mathcal{H})$: number of connected regions of $\mathbb{R}^n - \mathcal{H}$. Then $R(\mathcal{H}) \leq \sum_{i=0}^{min\{n,m\}} \binom{m}{i} := R(m)$. The inequality is an equality if hyperplanes in $\mathcal{H}$ are in general position
- Number of linear cells is a measure of **expressiv.** of ReLU layers. **Complexity grows subexponen. only once** $m > n \to$ ***innefficiency of shallow ReLU Networks.***
- Rep Power of deep ReLU Network: **Thrm (Motufar)**: Consider ReLU network with L layers of width $m > n \to R(m,L) \geq R(m)\left\lfloor \frac{m}{n} \right\rfloor^{n(L-1)} \to$ *exponential growth for* $m \geq 2n$. *So represent. advant. for deep ReLUs*
- *One of very few theoretical results showing advantage of depth in DNNs!!*
- Universality of ReLU?: **Thrm (Shektman):** Piecewise linear fcts. are universal approximators of $C(\mathbb{R})$, then sufficient to show ReLU can represent arbitrary piecewise functions which is a classic result of **Thrm (Lebesgue)**
- One then obtains: **Thrm (ReLU Universality):** Networks with one hidden layer of ReLU units are universal function approximators
- What about minimal non-linearity required for univ. approximators? We need functions that generlize ReLUs
- m-Hinge fct. $g[W,b](x) = \max_{i=1}^{m}\{w_i^\top x + b_i\}$ (Maxout units)
- **Thrm:** Every continuous piecewise linear function $g : \mathbb{R}^n \to \mathbb{R}$ can be written as a signed sum of m-Hinges with $m \leq n + 1$
- **Thrm**: Every continuous piecewise linear function f can be written as the difference of two polyhedral functions as follows $f(x) = \max_{(w,b)\in\mathcal{A}^+}\{w^\top x + b\} - \max_{(w,b)\in\mathcal{A}^-}\{w^\top x + b\}$ where $\mathcal{A}^+$ and $\mathcal{A}^- < \infty$
- **Minimality result**: Two max operations are a sufficient element of non-linearity to obtain universal approximators

2.5 Exercises

Sigmoid vs Tanh: $F_1(z) = \boldsymbol{W}_2 \tanh(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) + \boldsymbol{b}_2 = \boldsymbol{W}_2[2\sigma(2\boldsymbol{W}_1\boldsymbol{x}2\boldsymbol{b}_1) - \boldsymbol{1}] + \boldsymbol{b}_2 = \boldsymbol{W}_2'\sigma(\boldsymbol{W}_1'\boldsymbol{x} + \boldsymbol{b}_1') + \boldsymbol{b}_2'$ with $\boldsymbol{W}_1' = 2\boldsymbol{W}_1$, $\boldsymbol{W}_2' = 2\boldsymbol{W}_2$, $\boldsymbol{b}_1' = 2\boldsymbol{b}_1$, $\boldsymbol{b}_2' = \boldsymbol{b}_2 - \boldsymbol{W}_2\boldsymbol{1}$

Linear Auto-encoders: Let $\boldsymbol{X} = [\boldsymbol{x_1} \ldots \boldsymbol{x}_n] \in \mathbb{R}^{d\times n}$ be the data matrix and $\boldsymbol{A} = \boldsymbol{DE}$ be a linear autoencoder with $\boldsymbol{E} \in \mathbb{R}^{h\times d}$, $\boldsymbol{D} \in \mathbb{R}^{d\times h}$ and $\hat{\boldsymbol{X}} = \boldsymbol{AX}$ be the reconstruction matrix. Then

- $\text{rank}(\boldsymbol{A}) = \text{rank}(\boldsymbol{DE}) \leq \min\{\text{rank}(\boldsymbol{D}), \text{rank}(\boldsymbol{E})\} \leq \min\{d,h\}$ as $\text{rank}(\mathbf{E}) \leq \min\{d,h\}$ and $\text{rank}(\boldsymbol{D}) \leq \min\{d,h\}$
- By Eckart-Young-Mirsky Thm.: $\hat{\boldsymbol{X}}^* = \text{argmin}_{\hat{\boldsymbol{X}}:\text{rank}(\hat{\boldsymbol{X}})=k}\|\boldsymbol{X} - \hat{\boldsymbol{X}}\|_F^2 = \boldsymbol{U}_k\Sigma_k\boldsymbol{V}_k^\top$, where $\boldsymbol{X} = \boldsymbol{U}_k\Sigma_k\boldsymbol{V}_k^\top$ is the truncated SVD of $\boldsymbol{X}$
- The optimal weight matrices are given by the first k columns (ranked according to dominant singular values) of $\boldsymbol{U}$ combined with any invertible $\boldsymbol{A} \in \mathbb{R}^{k\times k}$: $\boldsymbol{D}^* = \boldsymbol{U}_k(A)$, $\quad \boldsymbol{E}^* = (\boldsymbol{A}^{-1})\boldsymbol{U}_k^\top$
- $\|\boldsymbol{X} - \boldsymbol{DEX}\|_F^2$ is convex in $\boldsymbol{E}$ and $\boldsymbol{D}$ but not jointly

Thm. (Weierstrass): If $f(x)$ is a *continuous* function for $a \leq x \leq b$ and if ϵ is an arbitrary positive quantity, it is possible to construct an approximating polynomial $P(x)$ s.t. $|f(x) - P(x)| \leq \epsilon$, $\quad a \leq x \leq b$ where w.l.o.g. one can assume $0 < a < b < 1$ and $f(x) = 0$ outside of (a,b)

3 Gradient-based learning

3.1 Back-propagation

Denominator Layout: $F : \mathbb{R}^d \to \mathbb{R}^m \implies \nabla_x F(x) \in \mathbb{R}^{d\times m}$

$\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial h_1}\text{diag}(f'(z_1)) = \frac{\partial L}{\partial h_1} \odot f'(z_1)$, f element-wise activation function

Per-layer Gradients in DNN: $\frac{\partial x_i^l}{\partial w_{ij}^l} = \frac{\partial \phi^l(W^l x^{l-1} + b^l)}{\partial w_{ij}^l} = \dot{\phi}_i^l x_j^{l-1}$ $\frac{\partial x_i^l}{\partial b_i^l} = \dot{\phi}_i^l$;

Where $\dot{\phi}_i^l \equiv$ sensitivity of i-th unit in l-th layer $= \dot{\phi}^l((w_i^l)^\top x^{l-1} + b^l)$

We observe that gradients only flow through $x^{l-1} \to x^l$ which reflects **locality** of backprop in terms of units and layers.

Downpropagating the changes to the loss with chain rule gives:

$\frac{\partial h[\theta](x,y)}{\partial w_{ij}^l} = \frac{\partial h^l[\theta](x^l,y)}{\partial x_i^l}\frac{\partial x_i^l}{\partial w_{ij}^l} = \delta_i^l \dot{\phi}_i^l x_j^{l-1}$ and $\frac{\partial h[\theta](x,y)}{\partial b_i^l} = \delta_i^l \dot{\phi}_i^l$

note: $\delta_i^l \equiv \frac{\partial h^l[\theta](x^l,y)}{\partial x_i^l}$ and $h^l[\theta](x^l, y) \equiv \ell(y, (F^L \circ \cdots \circ F^{l+1})(x^l))$

- δ describes the sensitivity of the loss to the i-th unit from the l-th layer
- h^l is the subsequent loss after forward-prop
- **Linear layers** sensitivity $\dot{\phi} = 1$ and $\frac{\partial h[\theta](x^l,y)}{\partial w_{ij}^l} = \delta_i^l \times x_j^{l-1}$
- **Logistic layers**: $\dot{\sigma}(z) = \sigma(z)(1 - \sigma(z))$. Maximal sensitivity $(\frac{1}{4})$ at $z = 0$
  - Large $|z|$ cause small sensitivities, complicating optimization
- **ReLU**: Deriv. is 1 f $z > 0$ and 0 if $z < 0$. Subderiv. $\in [0;1]$ for $z = 0$

Computing δ^l :

1. $\delta^L = \frac{\partial \ell(\hat{y},y)}{\partial \hat{y}}$ e.g., for $\ell = 1/2(y - \hat{y})^2$, $\delta^L = \hat{y} - y$
2. $\delta^l = [\frac{\partial F^{l+1}}{\delta x^l}]^\top \delta^{l+1}$, where $\frac{\partial F}{\partial x} = (\frac{\partial F_i}{\partial x_j})_{ij}$
   (a) So, all δ^l terms can be obtained by propagating the initial loss through the transposed Jacobi matrices
   (b) For $F(x) = \phi(Wx + b)$. $\frac{\partial F}{\partial x} = diag(\dot{\phi}(Wx + b)))W$

Backpropagation can be applied to any DAG

Deep Linear Networks

- $\frac{\partial \frac{1}{2}\|W^L \ldots W^1 x - y\|^2}{\partial W^l} = [W^L \ldots W^{l+1}]^\top \delta x^\top [W^{l-1} \ldots W^1]^\top$
  - $\delta \equiv W^L \ldots W^1 x - y$
- Aggregating gradients in linear (LS) networks over sets of examples yields a dependence on a set of sufficient statistics
  - $\Sigma \equiv \mathbf{E}(xx^\top)$, $\Gamma \equiv \mathbf{E}(xy^\top)$
  - Where $\mathbf{E}$ is the empirical average over a training set
- Computing full gradient in forward prop is $\mathcal{O}$(#paramaters) One backprop is $\mathcal{O}$(#outputs) Backprop in practice (assume L=3)

In general for a W^L, $\frac{\partial L}{\partial \mathbf{W^L}} = \frac{\partial L}{\partial \hat{y}}\mathbf{h}^\top$. For a **generic layer** $z = \mathbf{W}x$: $\frac{\partial L}{\partial z}x^\top$

Thus: $\frac{\partial L}{\partial \mathbf{W_2}} = \frac{\partial L}{\partial \mathbf{z_2}}h_1^\top$ and $\frac{\partial L}{\partial \mathbf{W_1}} = \frac{\partial L}{\partial \mathbf{z_1}}x^\top$ $\quad \frac{\partial L}{\partial \mathbf{z_1}} = \frac{\partial L}{\partial \mathbf{h_1}} \odot \dot{\phi}(z_1)$

3.2 Gradient Descent $\theta^{t+1} = \theta^t - \eta\nabla h(\theta^t)$

Gradient Flow: $\frac{d\theta}{dt} = -\nabla h(\theta)$. Gradient flow is ideal trajectory to follow.

Remark: Gradient descent is a discrete approximation of gradient flow. GD only successful if gradient change slowly in paramater space.

Smoothness: Function h is called L-smooth, if for $L > 0$ s.t.: $\|\nabla h(\theta_1) - \nabla h(\theta_2)\| \leq L\|\theta_1 - \theta_2\|$ $\quad (\forall \theta_1, \theta_2)$. In terms of gradient descent: the euc norm of gradient differences cannot grow more than linearly in the euc distance in parameter space this is $\equiv \lambda_{max}(\nabla^2 h) \leq L$ or $\nabla^2 h \leq LI$ $\forall\theta$

$\to$ Both conditions imply that Hessian is bounded above by L and that Principal eigenvalues correspond to maximum curvature of h.

From the Taylor series expansion for some θ being on the path from θ_1 to θ_2, we get for gradient descent:

$h(\theta_2) - h(\theta_1) \leq -\eta\|\nabla h(\theta_1)\|^2 + \frac{L\eta^2}{2}\|\nabla h(\theta_1)\|^2 = -\eta\left(1 - \frac{L\eta}{2}\right)\|\nabla h(\theta_1)\|^2$

$\to$Strict decrease in h guaranteed for $\eta < \frac{2}{L}$; optimal choice is $\eta = \frac{1}{L}$.

$\to$Smoothness and $\eta = \frac{1}{L}$ sufficient to find ϵ-critical $\equiv \|\nabla h\| \leq \epsilon$ point with $O(\epsilon^{-2})$ steps of GD specifically $t = \frac{2L}{\epsilon^2}(h(\theta^0) - \min h)$

PL-Condition: A differentiable function h obeys PL condition with parameter $\mu > 0$, iff $\frac{1}{2}\|\nabla h(\theta)\|^2 \geq \mu(h(\theta) - \min h)$ $(\forall\theta)$.

A small gradient norm at param. θ implies its near optimality. This allows us to use the suboptim. of the solution to ensure fast(er) convergence.

Proposition. Let h be differentiable, L-smooth and μ-PL. Then gradient descent w/ step size $\eta = 1/L$ converges at geometric rate

$h(\theta^t) - \min h \leq \left(1 - \frac{\mu}{L}\right)^t (h(\theta^0) - \min h)$.

In DNNs, PL holds not globally, but around a local minimum. Ensures fast convergence, without sub-optimality claims. Use noisy GD to avoid slow-down close to saddle points.

3.3 Acceleration & Adaptivity

Momentum: Heavy ball overcomes small gradient norms by increasing effective step size; $\theta^{t+1} = \theta^t - \eta\nabla h(\theta^t) + \beta(\theta^t - \theta^{t-1})$, $\quad \beta \in (0,1)$.

If the gradient is constant over many steps, then in the limit $\lim_{t\to\infty}(\theta^t - \theta^{t-1}) = -\eta\sum_{i=0}^\infty \beta^i \nabla h = -\left[\frac{\eta}{1-\beta}\right]\nabla h$. Large momentum boosts the effective step size by an arbitrarily large factor.

Nesterov: Evaluates the gradient at the extrapolated point $\nu^{t+1} = \theta^t + \beta(\theta^t - \theta^{t-1})$, $\quad \theta^{t+1} = \nu^{t+1} - \eta\nabla h(\nu^{t+1})$.

Guarantees convergence for convex and strongly-convex functions; optimal in the convex case and accelerated in the strongly-convex case.

Adaptivity: Adjusts learning rates per parameter using gradient history, as in AdaGrad, which gradually reduces step sizes over time.

Adam & RMSprop: Adam uses an exponentially weighted average to estimate the mean and variance of each partial derivative.

$g_i^t = \beta g_i^{t-1} + (1-\beta)\partial_i h(\theta^t)$, $\beta \in [0;1]$, $g_i^0 := \partial_i h(\theta^0)$

$\gamma_i^t = \alpha\gamma_i^{t-1} + (1-\alpha)[\partial_i h(\theta^t)]^2$, $\alpha \in [0;1]$, $\gamma_i^0 := [\partial_i h(\theta^0)]^2$ with iterate sequence: $\theta_i^{t+1} = \theta_i^t - \eta_i^t g_i^t$, $\eta_i^t = \frac{\eta}{\sqrt{\gamma_i^t + \delta}}$

Adam without the use of momentum $(\beta = 0)$ is RMSprop. Gradient in Adam is typically stochastic based on minibatches.

3.4 Stochastic Gradient Descent

Minibatches: Accelerate learning in DNNs using minibatches. For size $r = 1$, SGD writes as follows: $\theta^{t+1} = \theta^t - \eta\nabla h(\theta^t)(x_i, y_i)$, $i_t \sim U[1,s]$

Bias & Variance: Unbiasedness of update direction (i.e. in expectation the update directon equal the full batch gradient) is important in SGD.

$\mathbf{V}[\theta](\mathcal{S}) = \frac{1}{s}\sum_{i=1}^s \|\nabla h[\theta](\mathcal{S}) - \nabla h[\theta](x_i, y_i)\|^2$

Of importance for convergence is variance around a global/local minima, $\nabla h(\theta^*) = 0$

4 Convolutional Networks

4.1 Convolution

Integral Operators: $(Tf)(u) = \int_{t_1}^{t_2} G(u,t) f(t) dt$ with a kernel $G : \mathbb{R}^2 \to \mathbb{R}$

Convolution: $(f * g)(u) := \int_{-\infty}^\infty g(u-t) f(t) dt = \int_{-\infty}^\infty f(u-t) g(t) dt = (g * f)(u)$ integral operator with kernel $G(u,t) = g(u-t)$

Shift equivariance: Let $f_\Delta(t) = f(t + \Delta)$ be a shifted fct. Convolution is shift-equivariant: $\forall \Delta \in \mathbb{R} : f_\Delta * g = (f * g)_\Delta$ because of the shift invariance of the kernel: $G(u - \Delta, t - \Delta) = g(u - t) = G(u,t)$ $\forall\Delta$

Any translation equivariant operator is a convolution Fourier transform: $\mathcal{F}(f)(u) := \int_{-\infty}^\infty e^{-2\pi itu} f(t) dt$

Convolutional operators can be computed via pointwise multiplications in Fourier space: $\mathcal{F}(f * g) = \mathcal{F}(f) \cdot \mathcal{F}(g)$ and the **Inverse Fourier transform** maps it back to the input space $f * g = \mathcal{F}^{-1}((\mathcal{F}(f) \cdot \mathcal{F}(g)))$. Computations of convolutions: Naive $\mathbf{O}(n^2)$, FFT $\mathbf{O}(n\log n)$

Linear shift equivariant transforms: A convolution can also be characterized as a linear shift invariant operator (1)$T(\alpha f + \beta g) = \alpha Tf + \beta Tg$, (2) $(Tf_\Delta)(t) = (Tf)(t + \Delta)$. Any linear, translation equivariant transformation T can be written as a convolution with a suitable kernel.

Discrete Convolution: $(f * g)[u] = \sum_{t=-\infty}^\infty f[t]g[u-t] \to$*commutative*

Cross-correlation: $(g \star f)[u] := \sum_{t=-\infty}^\infty g[t]f[u+t] = (\overline{g} * f)[t]$ where $\overline{g}[t] = g[-t] \to$*non-commutative*

Toeplitz matrix: if the f,g have finite support (f[t] = 0 $\forall t \notin [1:n]$, $g[t] = 0$ $\forall t \notin [1:m]$ and $m \leq n$ we have $(f * g) =$

$$\begin{pmatrix} g_1 & 0 & 0 & \cdots & 0 & 0 \\ g_2 & g_1 & 0 & \cdots & 0 & 0 \\ \cdot & \cdot & \cdot & \cdots & \cdot & \cdot \\ 0 & 0 & 0 & \cdots & g_m & g_{m-1} \\ 0 & 0 & 0 & \cdots & 0 & g_m \end{pmatrix} \cdot \begin{pmatrix} f_1 \\ f_2 \\ \cdot \\ \cdot \\ \cdot \\ f_n \end{pmatrix}$$

number of free parameters is dim(g) = m

4.2 Convolutional Networks

Goal: Define layered arch. for signal processing that exploits translation equivariance, locality and scale => learn kernel fct/filters from data.

2D discrete convolution: Extend conv. to higher dimensions by using matrices/tensors. $(F * G)[i,j] = \sum_{k=-\infty}^\infty \sum_{l=-\infty}^\infty F[i-k, j-l]G[k,l]$. Convolved signal inherit topology from the original signal so units in a conv layer are usually arranged on the same grid as the input (1d, 2d,..)

Pooling: Let r be window size, then $x_{ij}^{\max} = \max\{x_{i+k,j+l} : 0 \leq k, l < r\}$ (also avg, min...) is max pooling. Using strides when pooling downsizes.

Channels: Learn multiple filters(width) per layer. All channels use the same window size and number of channels is a design parameter. Channels are permutation invariant and connections across channels are dense.

4.3 Backpropogation of CNN

Receptive Fields: Receptive field of x_i^l: $\mathcal{I}_i^l := \{j : w_{ij} \neq 0\}$ where W^l is the toeplitz matrix of the convolution (includes all the elements in the previous layer that effect the calculation of x_i^l). Then $\frac{\partial x_i^l}{\partial x_j^{l-1}} = 0$ for $j \notin \mathcal{I}_i^l$.

Con: local receptive field make it hard to connect distant features.

Weight sharing: When computing the loss gradients $\partial h / \partial g_j^l$ (g_j^l is a kernel weight) we have to respect the weight sharing: $\frac{\partial h}{\partial g_j^l} = \sum_i \frac{\partial h}{\partial x_i^l} \frac{\partial x_i^l}{\partial g_j^l}$. The sum over units appears as the same weight is re-used for every unit within the target layer.

- $\nabla_v (v^T \text{vec}(\sigma(x * w))) = \text{vec}(\sigma(x * w))$
- $\nabla_A v^T \text{vec}(A) = mat(v)$
- $\nabla_w (x * w) \in \mathbb{R}^{q \times q \times r \times r}$ if $(x * w) \in \mathbb{R}^{r \times r}$, $w \in \mathbb{R}^{q \times q}$
- $\nabla_w (v^T \text{vec}(\sigma(x * w))) = \text{rot}_\pi(x) * B$ where $B = \text{mat}(v) \odot \sigma'(x * w)$

```
                      class Net(nn.Module):
                          def __init__(self):
                              super(Net, self).__init__()
      5x5x3x6 + 6 +           self.conv1 = nn.Conv2d(3, 6, 5)
                0 +           self.pool = nn.MaxPool2d(2, 2)
    5x5x6x16 + 16 +           self.conv2 = nn.Conv2d(6, 16, 5)
 16x5x5x120 + 120 +           self.fc1 = nn.Linear(16 * 5 * 5, 120)
     120x84 + 84 +            self.fc2 = nn.Linear(120, 84)
      84x10 + 10 =            self.fc3 = nn.Linear(84, 10)

           62'006
```

4.4 Inception Network

Deep stacking, uses many channels for accuracy. Dimension reduction in between convolutions (compression of m channels) using a $1 \times 1 \times m$ convolution with $k \leq m$: $x_{ij}^+ = \sigma(W x_{ij})$; $W \in \mathbb{R}^{k \times m}$

- 1×1 part means there is no spatial conv. performed
- Not committing to a certain window size, but solve trade-off problem a part of the learning by multiple processing paths
- Softmax layers to improve learning dynamics (shortcut error backprop)

4.5 Embeddings

Enable processing of non-numeric sequences via DNN's by embedding symbols in vector space. Ω is the alphabet of symbols; $\Omega \ni \omega \mapsto x_\omega \in \mathbb{R}^n$. Embedding $\{x_\omega\}$ are learnable parameters.

Word2vec: Learn input and output embeddings: $\Omega_{in} \ni \omega \mapsto x_\omega \in \mathbb{R}^n$ and $\Omega_{out} \ni \nu \mapsto y_\nu \in \mathbb{R}^n$.

Combine: $\mathbb{P}(\nu|\omega) = \frac{\exp[x_\omega^T y_\nu]}{\sum_\mu \exp[x_\omega^T y_\mu]}$ and use to predict an output word ν in neighbourhood of an input word ω

5 Recurrent Neural Networks

Idea: Provide more flexibility than CNNs to model temporal/sequential data (must specify filter width in CNNs).

5.1 Setup

Observation sequence: $x^1, x^2, x^3, \dots$, with $x^t \in \mathbb{R}^n$ and **Hidden states**: $z^1, z^2, z^3, \dots$, with $z^t \in \mathbb{R}^m$. There are two operators:

- **Connecting x to hidden states z**: $F[U,V](z,x) \equiv \phi(Uz + Vx)$, with $U \in \mathbb{R}^{m \times m}, V \in \mathbb{R}^{m \times n}$.
- **Output map (optional)**: $G[W](z) \equiv \psi(Wz)$, where $W \in \mathbb{R}^{q \times m}$.

Rem. 1: Both F and G are *time-invariant*, (i.e., shared parameters).
Rem. 2: With a fixed time horizon $t \leq T$, the unrolled RNN is equivalent to a feedforward network with T hidden layers. The main difference is the sharing of parameters between layers.
Rem. 3: (**Bi-directional RNN**) Hidden state evolution does not have to follow direction of time. One can define a reverse order sequence $\tilde{z}^t = \phi(\tilde{U} z^{t+1} + \tilde{V} x^t)$ and modify the output map to be $\hat{y}^t = \psi(W z^t + \tilde{W} \tilde{z}^t)$
Rem. 4: (**Stacked RNN**) Possible to add depth by connecting layers horizontally: $z^{t,l} = \phi(U^l z^{t-1,l} + V^l z^{t,l-1})$, where $z^{t,0} \equiv x^t$. Outputs are generated based on the last hidden activations $z^{t,L}$

5.2 Backpropagation through time

- Error signals at the outputs: $\delta_k^t = \frac{\partial h}{\partial \hat{y}_k^t}$
- Train the **output weights**: $\frac{\partial h}{\partial w_{k,i}} = \sum_{t=1}^T \delta_k^t \dot{\psi}_k^t z_i^t$, with $\dot{\psi}_k^t \equiv \dot{\psi}(w_k^\top z^t)$
- Train the **input weights**: $\frac{\partial h}{\partial v_{i,j}} = \sum_{t=1}^T \frac{\partial h}{\partial z_i^t} \dot{\phi}_i^t x_j^t$, with $\dot{\phi}_i^t \equiv \dot{\phi}(v_i^\top x^t)$
- which needs... $\frac{\partial h}{\partial z_i^t} = \sum_{k=1}^K \sum_{s=t}^T \delta_k^s \sum_{j=1}^m \frac{\partial \hat{y}_k^s}{\partial z_j^s} \frac{\partial z_j^s}{\partial z_i^t}$, with $\frac{\partial \hat{y}_k^s}{\partial z_j^s} = \dot{\psi}_k^s w_{k,j}$
- Train the **state-to-state weights**: $\frac{\partial h}{\partial u_{i,j}} = \sum_{t=1}^T \frac{\partial h}{\partial z_i^t} \dot{\phi}_i^t z_j^{t-1}$

5.3 (In)stability of RNNs

In backprop. through time, the norm of the gradients is difficult to control. We have: $\frac{\partial z^T}{\partial z^0} = \dot{\Phi}^T U \dots \dot{\Phi}^1 U$, with $\dot{\Phi}^t \equiv \text{diag}\left(\dot{\phi}(u_j^\top z^{t-1} + v_j^\top x^t)\right)$.

For typical activations, diagonal sensitivity matrices are bounded $\dot{\Phi}^t \leq \alpha \mathbb{I}$ ($\alpha = 1$ for ReLUs, $\alpha = 1/4$ for logistic activations). We have: $\left\| \frac{\partial z^T}{\partial z^0} \right\|_2 \leq (\alpha \sigma_1(U))^T$, which ...

... $\to 0$ as $T \to \infty$ if we assume $\sigma_1(U) < \frac{1}{\alpha}$ (**Vanishing gradients**)

... $\to \infty$ as $T \to \infty$ if we assume $\sigma_1(U)$ to be large (**Exploding gradients**)

$\to$**Makes it difficult to model long-term dependencies**

5.4 Gated Memory

$\to$*Avoid short-term fluct. by controlling when memory is kept or overwritten*

A gating unit takes the following form: $z^+ \equiv \sigma(G\zeta) * z$, where z^+ and $z \in \mathbb{R}^m$, $\sigma(G\zeta) \in (0;1)^m$ and $*$ denotes pointwise multiplication. σ is the vector-valued logistic function, and ζ is a control input determining the relative preservation of information contained in z. In the limit $\sigma_i \to 0$, z_i is forgotten, while the limit $\sigma_i \to 1$ preserves z_i

1. **Long Short-Term Memory (LSTM)**:
   Uses two gating units, a forget gate and a storage gate:
   $z^t \equiv \sigma(F\tilde{x}^t) * z^{t-1} + \sigma(G\tilde{x}^t) * \tanh(V\tilde{x}^t)$. Here, $\tilde{x}^t \equiv [x^t, \zeta^t]$ (concatenated), and $\zeta^{t+1} = \sigma(H\tilde{x}^t) * \tanh(Uz^t)$ is a control signal sequence.
   $\to$An LSTM has **5 weight matrices in total**: 3 gating matrices (F,G,H), 2 propagation matrices (V,U) (6 incl. output weight matrix: W).
2. **Gated Recurrent Unit (GRU)**:
   The GRU combines the forget and storage gates of an LSTM into a convex combination: $z^t = (1-\sigma) * z^{t-1} + \sigma * \tilde{z}^t$, with $\sigma \equiv \sigma(G[x^t, z^{t-1}])$, and $\tilde{z}^t \equiv \tanh(V[\zeta^t * z^{t-1}, x^t])$, where $\zeta^t \equiv \sigma(H[z^{t-1}, x^t])$.
   Adv.: ζ^t can be computed implicitly without additional recursion.
   $\to$Only **3 weight matrices in total** (4 incl. output weight matrix).
3. **Minimal Gated Units**:
   Idea: Remove dependencies on previous hidden states from gatin mechanism: $\sigma \equiv \sigma(Gx^t)$, and $\tilde{z}^t = Vx^t$, which also removes the squeezing of the dynamical range via tanh

5.5 Linear Recurrent Models

Background: Classical RNNs lack sufficient parallelization during learning. Transformers, on the other hand, scale quadratically and not just linearly in context length. $\to$Linear recurrent models like S4 or Mamba avoid non-linear dependencies along the chain of hidden states.

- **Linear Recurrent Unit (LRU)**:
  $z^{t+1} = Az^t + Bx^t$. The idea is that we can parametrize and run this recurrence over the complex numbers in a way that the architecture & parametrization guarantee stability:
  (1) Diagonalize the evolution matrix A (assuming it is non-defective) over $\mathbb{C}$ as follows: $A = P\Lambda P^{-1}$, where $\Lambda \equiv \text{diag}(\lambda_1, \dots, \lambda_m)$, and $\lambda_i \in \mathbb{C}$.
  (2) Perform a change of basis: $\zeta^{t+1} = \Lambda \zeta^t + Cx^t$, with $\zeta^t \equiv P^{-1} z^t$, and $C \in \mathbb{C}^{m \times n}$. We make this equation our recurrence.
  (3) Stability of this (linear) system requires the modulus of all eigenvalues to be bounded: $\max_j |\lambda_j| \leq 1$, where $|a + ib| \equiv \sqrt{a^2 + b^2}$ (modulus) $\to$spectral radius of A (or Λ)
  (4) Control the spectral radius: Use the polar form to represent complex number as $z = r(\cos(\phi) + \sin(\phi))$, with modulus $r = |z| \geq 0$, and phase $\phi \in [0; 2\pi)$. Stable parametrization with double exp. for the λ's: $\lambda_i = \exp(-\exp(\nu_i) + i\phi_i) \in (0;1)$. This can be done by sampling at initialization from the ring defined by I:
  $\phi_i \sim \text{Uni}[0;2\pi], \quad r_i \sim \text{Uni}[I], \quad I \subseteq [0;1], \quad \exp(\nu_i) = -\ln(r_i)$
  (5) Move the non-linearities into the output map to compensate for the simpler linear recurrence: $y^t = \text{MLP}(\text{RE}(Gz^t)), \quad G \in \mathbb{C}^{k \times m}$.
  $\to$The resulting model is universal as a seq-2-seq map

5.6 RNN Models for Sequences

Generative model $p\left(y^{1:T} | x^{1:T}\right) \approx \prod_{t=1}^T p\left(y^t | x^{1:t}, y^{1:t-1}\right)$. Then we have $x^{1:t} \overset{F}{\mapsto} z^t$ and to get a distribution and not a deterministic output, the deterministic computation of the RNN is augmented with a final probabilistic step with a parametric family $z^t \mapsto \mu^t, \quad \mu^t \mapsto p(y^t; \mu^t)$. This could be a softmax: $p(y; \mu) = \frac{\exp(\mu_y)}{\sum_\nu \exp(\mu_\nu)}, \quad y \in \{1, \dots, k\}, \quad \mu \in \mathbb{R}^k$.

Note. We need to feed back outcome of stochastic generation because otherwise, y^t will depend on $y^{1:t-1}$ only through z^t which is too strong an independence assumption to make.

5.7 Teacher Forcing

Fixing the problem that predicted sequences may quickly diverge from the target one for long-range dependencies, resulting in error signal of little value. Teacher forcing feeds back the target sequence y^* during training, instead of the generated outputs. **Adv.**: No lasting effect of preds. during training as they get overwritten by ground-truth. **Disadv.**: Creates divergence between training & testing because it can only be applied during training (**Exposure Bias**).

5.8 *seq-2-seq*

Encoder-Decoder based architecture. The encoder maps input sequence to fixed dimensional activation vector via encoder RNN without outputs: $z^0 := 0, \quad (x^t, z^{t-1}) \mapsto z^t \quad \forall 1 \leq t \leq T$. Then, a decoder RNN without inputs is used: $\zeta^0 := z^T, \quad (y^{s-1}, \zeta^{s-1}) \mapsto \zeta^s, \quad \zeta^s \overset{\sim}{\mapsto} y^s$.
As before, outputs are fed back and next output is generated stochastically based on the hidden state of the RNN.

6 Transformers

6.1 Token Paradigm

Given is a sequence of tokens in the form of their respective embeddings $X = [x_1, \dots, x_T] \in \mathbb{R}^{n \times T}$. The fundamental idea is to map the non-contextualized embeddings X to contextualized representations $\Xi = [\zeta_1, \dots, \zeta_T] \in \mathbb{R}^{m \times T}$.

6.2 Attention Mixing

An attention mechanism produces numbers $\alpha_{s,t}$ that are then used to convexly combine the input representations into new representations $\zeta_s = \Sigma_t \alpha_{s,t} W x_t$, where $\alpha_{s,t} \geq 0$ and $\Sigma_t \alpha_{s,t} = 1$.

Remarks:

- The attention weights $\alpha_{s,t}$ have a **source** (where attention emerges, s) and a **target** (where attention extracts information, t)
- In matrix notation, we write $A = (a_{st}) \in \mathbb{R}^{T \times T}$, such that $\Xi = WXA^\top$.
- Attention mechanism mixes information across columns of X via $A^\top$, and mixes dimensions linearly via W.

6.3 Query-Key Matching

Question: How to learn A?

1. Project X to query matrix $Q = U_Q X$ and key matrix $K = U_K X$, where $U_Q, U_K \in \mathbb{R}^{q \times n}$
2. Produce matching matrix by matching keys and queries via inner products: $Q^\top K = X^\top U_Q^\top U_K X \in \mathbb{R}^{T \times T}$ (note that $rank(U_Q^\top U_K) \leq q$)
3. Soft-max transformation: $A = \text{softmax}(\beta Q^\top K)$ where $a_{st} = \frac{e^{\beta[Q^\top K]_{st}}}{\sum_r e^{\beta[Q^\top K]_{sr}}}$

Inverse temperature β**:** Controls the entropy of the softmax and is not learned, but usually chosen as $\beta = 1/\sqrt{q}$ (to standardize the size of $Q^\top K$ dot products to be independent of q). **Higher** β leads to **lower entropy** of the softmax and the **attention** becomes **more selective/focused** on the highest-scoring queries/keys (and vice-versa for lower β)

6.4 Multi-Headed Attention

Idea: Run multiple attention models in parallel.

- Replicate the matrices U_K, U_Q, W a total of r times
- Perform attention-based propagation $X \mapsto \Xi_i$ for $i = 1, \dots, r$
- Concatenate the matrices Ξ_i along the feature dimension

Concrete numbers from *Attention Is All You Need* are: feature space dimension $n = 512$, number of heads $r = 8$, query/key space dimension $q = 64$.

Remark: Originally proposed values for r, q and n ensure that after concatinating the different heads, the output dimension stays the same ($8 \cdot 64 = 512$). The reasons for this *trick*:

- Residual connections: Would not be possible with different dimensions.
- Efficiency costs: Less parameters to estimate.
- Scalability: The output dimension would increase with every layer by a factor of r, if we would not use this trick.

6.5 Feature Transformation

Issue: Attention mechanism **only** performs convex combination of columns of X, which is insufficient (even if the relative weights are non-linear functions of X).

Solution: Gain representational power by **applying a per-token** transformation with a feedforward network. Formally, $X \mapsto \Xi \mapsto F(\Xi)$, where $F(\Xi) = (F(\zeta_1), \dots, F(\zeta_T))$ operates on the columns of Ξ.

6.6 Positional Encoding

Issue: Attention mechanism operates on (unordered) set of tokens ($\rightsquigarrow$ value of token position is lost)

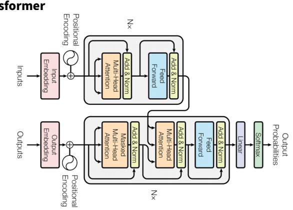
Solution: Assign each non-negative integer t to a low-dimensional vector and add this vector to the respective token embedding x_t.

Sinusoidal Encoding: $p_{tk} = \begin{cases} sin(t\omega_k) & k \text{ even} \\ cos(t\omega_k) & k \text{ odd} \end{cases}$, where $\omega_k = C^{n/k}$ and n is the feature space dimension ($= 512$ in the original paper)

6.7 Masked and Cross-Attention

Masked Attention: In cases where transformers operate in coupled encoder-decoder pairs, the decoder typically operates in autoregressive fashion (output at t depends on encoder, but also on produced token representations for $s < t$). Masked attention restricts the attention to the past. **Cross-Attention:** Attention to representations produced by the encoder. Cross-Attention is implemented across layers of the same depth. Note that in cross-attention the queries emerge from the decoder, whereas the keys and values are derived from the encoder states.

6.8 Transformer



6.9 Large Language Models (BERT)

- **Acronym:** Bidirectional Encoder Representations from Transformers.
- **Bidirectional:** BERT processes text in both directions simultaneously.
- **Pre-Training Tasks:**
  - Masked Language Modeling (Cloze test-like training where 15% of words are masked for prediction). The to be predicted words are replaced by `MASK` token in 80% of cases, by random token in 10% of cases, and left unchanged for the remaining 10% of cases.
  - Sentence pair classification. Specifically, classifying if two sentences are consecutive or not.
- **Tokenization:** BERT operates on tokenized text. Specifically, the WordPiece tokenizer was used. Additionally, a `CLS` token (and its final embedding) was used for classification problems. Also, a `SEP` token was used for marking the sepeartion of two senteces.

6.10 Vision Transformers

Idea: Decompose image into non-overlapping patches as the tokens of an image (typically into patches of size 16×16 pixels). Note how usually we flatten the tokens into vectors. Formally,

$\mathbb{R}^{p \times p \times q} \ni \text{patch}^t \mapsto x^t \equiv V \text{vec}(\text{patch}^t) \in \mathbb{R}^n, \quad V \in \mathbb{R}^{n \times (qp^2)}$

Issues: This pre-processing ignores the 2D structure of pixel arrangements within a patch. If enough data, this is not really an issue!

Remark: Early VITs have used two layer GELU (Gaussian error linear unit) MLP. The GELU activation is given by, $\phi(z) = z\text{Prob}(Z \leq z), \quad Z \sim \mathcal{N}(0,1)$

CNN or VIT: CNN-based models have the advantage of built-in translation equivariance as an inductive bias. If this property was strictly to hold for all images, one would expect it to be an advantage. Vision Transformers have less of an inductive bias and generally little spatial awareness.

6.11 Complexity

Let n be the sequence length, d the representation dimension, k the kernel size of convolutions and r the size of the neighborhood in restricted self-attention

Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

7 Geometric Deep Learning

7.1 Sets & Points

Goal: realize some function $\{x_1, \dots, x_M\} \subset R, \quad f : 2^R \to \mathcal{Y}$. Naively, we could turn the set into an ordered N-tupel $\{x_1, \dots, x_M\} \mapsto (x_1, \dots, x_M) \in R^M$ and then apply standard DNN $\rightsquigarrow$ Problems: variable-length inputs if sets have different cardinality & arbitrariness of ordering of elements

Invariance and Equivariance:

1. f is *order-invariant* iff $f(x_1, \dots, x_M) = f(x_{\pi_1}, \dots, x_{\pi_M}) \boldsymbol{x} \in R^M, \forall \pi \in S_M$, π : permutation, S_M: symmetric group
2. f is *order-equivariant* iff $f(x_1, \dots x_M) = (y_1, \dots, y_M) \implies f(x_{\pi_1}, \dots, x_{\pi_M}) = (y_{\pi_1}, \dots, y_{\pi_M}) \boldsymbol{x} \in R^M, \forall \pi \in S_M$

More general, for some transformation $T \in \mathcal{T}$, where $\mathcal{T}$ is some group of transformations, and a function f, we call f

1. *invariant* iff $f(T(x_1, \dots, x_M)) = f(x, \dots, x_M) \forall \boldsymbol{x} \in R^M, \forall T \in \mathcal{T}$
2. *equivariant* iff $f(T(x_1, \dots, x_M)) = T(f(x_1, \dots, x_M)) \forall \boldsymbol{x} \in R^M, \forall T \in \mathcal{T}$

Examples: conv. operator is equivariant to translations; att. mechanism permutation equivariant with regard to token sequence since softmax is equivariant operation; Euclidean norm is invariant under rotations

Deep Sets:

- Sum is invariant under permutations (commutativity): $\sum_{m=1}^M x_m = \sum_{m=1}^M x_{\pi_m}, \forall M, \forall \pi \in S_M \rightsquigarrow$ by allowing for other (fixed or DNN) maps $\phi : R \to \mathbb{R}^N$ and $\rho : \mathbb{R}^N \to \mathbb{R}$ we can construct a larger class of invariant function as follows: $f(x_1, \dots x_M) = \rho\left(\sum_{m=1}^M \phi(x_m)\right)$, where ϕ, ρ are independent of M, so f is defined over an arbitrary number of inputs.
- Can construct invariant mappings using different agg. functions such as (point-wise) max.& min., etc.: $f(x_1, \dots, x_M) = \rho\left(\max_{m=1}^M \phi(x_m)\right)$
- Once we have invariant f, this can be turned into an equiv. map using $\rho : R \times \mathbb{R}^N \to \mathcal{Y}, \left(x_m, \sum_{k=1}^M \phi(x_k)\right) \mapsto y_m$ and apply pointwise for each x_m. Hence, the down-stream processing ρ depends on an invar. argument (here: aggregated as sum) and x_m itself (via shortcut connection)
- fixed N (width of hidden representations) is sufficient if one allows for discontinuous mappings in the limit of $M \to \infty$, but more practically realizable mappings may require $N \geq M$

PointNet: Sparse measurements at specific 3D points i.e. point cloud $\rightsquigarrow$ desirable that a DNN obeys permutation in-/equivariance

7.2 Graph Convolutional Networks

Basics: Set of Vertices $\mathcal{V} = \{v_1, \dots, v_M\}$. We associate a feature vector x_m with every node and consider undirected edges $\mathcal{E} = \{e_1, \dots, e_K\} \subseteq \{e \in 2^\mathcal{V} : |e| = 2\}$. Idea: graph may encode similarities between nodes and corresponding dependencies between their outputs.

Adj. Matrix: $A = (a_{nm}), \quad a_{nm} = \mathbb{I}_{\{v_n, v_m\} \in \mathcal{E}}$. A is symmetric and has 0 diagonal (i.e. no self-loops)

Permut. Matrix: $P \in \{0,1\}^{M \times M}$ s.t. $\sum_{n=1}^M p_{nm} = \sum_{n=1}^M p_{mn} = 1 \, \forall m$ Cauchy's two line notation for permutations: $P = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \leftrightarrow \pi = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 4 & 1 \end{pmatrix}; \quad \pi^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix} \quad \pi \& \pi^{-1}$ are always inverse elements in the group S_M

Permutation matrices are orthogonal i.e. $P^{-1} = P'$ or $PP' = P'P = I$ as $(PP')_{nm} = \sum_k p_{nk} p_{mk} = \delta_{nm}$ Let Π_M be the group of permutation matrices, then invar. of a function of a graph can be defined as $f(X,A) \overset{!}{=} f(PX, PAP'), \forall P \in \Pi_M$ and equiv. as $Pf(X,A) \overset{!}{=} f(PX, PAP'), \forall P \in \Pi_M$

Invariant Core: Multiset of features in neighborhood of node i: $X_m = \{\{x_n : \{v_n, v_m\} \in \mathcal{E}\}\}$, $\{\{\}\}$ = multiset i.e. for given node $v_m$it's the multiset of feature vectors from its neighbors. Choose any $\phi : (x_m, X_m) \mapsto \phi(x_m, X_m) \in \mathbb{R}^N$. We can then apply this function at every node to arrive at $X \mapsto \phi(X) := \begin{pmatrix} \phi(x_1, X_1)' \\ \vdots \\ \phi(x_M, X_M)' \end{pmatrix} \in \mathbb{R}^{M \times N}$

Note that any pair of isomorphic graphs (i.e. graphs that are bijective under edge re-numbering) are processed in the same manner. ϕ can be designed e.g. by transforming node features and then aggregating by summation: $\phi(x_m, X_m) = \phi(x_m, \sum_{x \in X_m} \psi(x))$. If globally invariant function is desired, one can add a final invariant aggregation step. **Graph-Learning Scenarios:** *node classification*: make a discrete/real-valued prediction at every node; *graph classification*: classify the entire graph; *link prediction*: predict missing edges in graph

Coupling Matrix: In convolutional graph NNs, the aggregation over neighborhoods is performed with a fixed sets of weights known as coupling matrix (typically derived from adj. matrix A). Often we add self-loops to A and normalize with the degree matrix as follows: $\overline{A} = D^{-1/2}(A + I)D^{-1/2}$, $D = \text{diag}(d_1, \dots, d_M)$, $d_m = 1 + \sum_{n=1}^M a_{nm}$. Rmk: The normalization balances out degree variability between the nodes but the activation of a node propagates to its neighbors. E.g. for a 1-dim (cyclic) chain, a regular graph of degree 2, then $\overline{A} = 1/3A$

GCN Layers: Let W be weight matrix and σ an activation function. Then, one step of propagation in a convolutional GNN: $Z = \sigma(\overline{A}XW)$, $W \in \mathbb{R}^{N \times d}$.

Note: Multiplication by $\overline{A}$ from the left sums over nodes, based on the edge structure of the graph, whereas multiplication by W from the right sums over features. Iteratively, we have for two-layer GCN: $Z = \text{softmax}(\overline{A}\,\text{ReLU}(\overline{A}XW)W_1)$

! Control the spectrum of the coupling matrix $\overline{A}$: want $|\lambda(\overline{A})| \leq 1$, $\to$guarantees stability in terms of repeated propagation in a deep GCN

Limitations of GCN: GCNs require that depth is equal to diameter of graph to propagate between all pairs of nodes $\rightsquigarrow$ deep networks $\rightsquigarrow$ issues :

- **Oversmoothing:** Repr. at nodes may become indistinguishable
- **Oversquashing:** Bottleneck effect of how much long-range information can be encoded in fixed-size representation $\rightsquigarrow$ can improve learning from long-range dependencies by modyfiying the graph structure to allow for smaller diameters

7.3 Spectral Graph Theory:

Laplacian Operator: is $\Delta f := \sum_{n=1}^N \frac{\partial f^2}{\partial x_n^2}$, $f : \mathbb{R}^N \to \mathbb{R}$. Think of Laplace operator at x as measuring local variation from f-average in a vanishingly small neighborhood of x: for balls of radius $h \to 0 \quad \overline{f}(x,h) - f(x) = \frac{h^2}{2n} \Delta f(x) + o(h^2)$

Graph Laplacian: with adjacency matrix A & degree matrix D: $L := D - A$, $(Lx)_n = \sum_{m=1}^M a_{nm}(x_n - x_m)$; but in practice often better to use (symmetric) degree-normalized Laplacian: $\tilde{L} = I - D^{-1/2} A D^{-1/2} = D^{-1/2}(D - A)D^{-1/2}$. Note: $L \& \tilde{L}$ are symmetric, positive semi-definite and weakly diagonally dominant i.e. $d_n = \sum_m a_{nm} = \sum_m |a_{nm}|$

Graph Fourier Trafo: $L = D - A = U\Lambda U'$, $\Lambda := \text{diag}(\lambda_1, \dots, \lambda_M)$, $\lambda_i \geq \lambda_{i+1}$ with U orthogonal. The columns of U can be considered to be the graph Fourier basis. **Graph Convolution:** Pointwise multiplication in the Fourier domain: $x * y = U((U'x) \odot (U'y))$. The filtering operation from 1d-signals can be generalized: $G_\theta(L)x = UG_\theta(\Lambda)U'x \rightsquigarrow \mathcal{O}(M^3)$ since full eigendecomposition of L. **TRICK: Polynomial Kernels** i.e. $U\left(\sum_{k=0}^K \alpha_k \Lambda^k\right) U' = \sum_{k=0}^K \alpha_k L^k$. If A sparse ($\implies L$ sparse), then highly efficient scaling of arithmetic operations with $|\mathcal{E}|$ as only powers of Laplacian have to be calculated. Spectral conv. can then be expressed as multiple equivariant local layers in the vertex domain. Polynomial order K defines size of neighborhood (typically 2-5). These filters are isotropic and have fewer params. and lower express. than in traditional convs.

Polynomial Kernel Networks: Channel indices denoted by i, j, then pre-activation of graph-convolutional layer can be written as $x_i^{l+1} = \sum_j p_{ij}(L) x_j^l + b_i$, $p_{ij}(L) = \sum_{k=0}^K \alpha_{ijk} L^k$. The number of parameters is essentially (ignoring b_i) the product of the channel dimensions times $K + 1$.

7.4 Attention GNNs

Coupling with Attention: Define a coupling matrix via attention: $Q = (q_{ij})$, $q_{ij} = \text{softmax}(\rho(u'(Vx_i; Vx_j; x_{ij}))$, $\quad$ s.t. $\sum_j A_{ij} q_{ij} = 1$; V projection matrix for node features; x_{ij} possible edge features i.e. the projected node and the edge features are concatenated and then projected to a tunable direction u. $\rho(\cdot)$ is a non-linearity (leaky ReLU recommended) Can combine results of multiple attention heads with independent parameters in a structured coupling matrix Q: $X^+ = \sigma(QXW)$

Limitations: Attention GNNs have higher degree of model adaptivity compared to fixed coupling matrix in GCN's but still are in class of neighborhood-based message passing models: *inherent limitations that certain non-isomorphic graphs cannot be distinguished by network.*

Graph Isomorphism: Two graphs $\mathcal{G}$ and $\mathcal{H}$ are isomorphic if there exists a bijection between their vertex sets s.t. for any pair of vertices (u,v) in $\mathcal{G}$, there is an edge between u and v iff there is an edge between the corresponding vertices in $\mathcal{H}$.

Weisfeiler-Lehmann (WL) graph isomorphism test: based on node coloring:

1. Init all nodes with same color
2. Two nodes u, v get a different color if
   (a) they had different color before
   (b) there is a color c s.t. u and v have a different number of c-colored neighbors
3. if obtained color histogram differs, the graphs cannot be isomorphic

It has been shown that many message-passing algorithms including GCNs and GATs cannot distinguish beyond the WL test
1-WL cannot distinguish these two graphs and neither can GNNs

7.5 Exercises

Let $\mathcal{G}$ denote the group acting on the doman Ω and let $\chi(\Omega)$ denote the signals defined on Ω.

$\mathcal{G}$-**equivariance:** A function f is $\mathcal{G}$-equivariant if $f(g \cdot x) = g \cdot f(x) \quad \forall x \in \chi(\Omega), \forall g \in \mathcal{G}$, where $g \cdot x$ denotes the action of the group element g on the signal $x \in \chi(\Omega)$. The action g on x can be written as $\rho(g)x$, where $\rho(g)$ is referred to as the group representation of g (typically $\rho(g) \in \mathbb{R}^{n \times n}$ if $x \in \mathbb{R}^n$. For any $h, g \in \mathcal{G}, \rho(hg) = \rho(h)\rho(g)$

$\mathcal{G}$-**convolution:** The $\mathcal{G}$-convolution operator between two signals $x, \theta \in \chi(\Omega)$ is a function of the group elements defined as $(x * \theta)(g) = \langle x, \rho(g)\theta \rangle_{\chi(\Omega)} = \int_\Omega x(u)\theta(g^{-1}u)du$, where $[\rho(g)\theta](u) = \theta(g^{-1}u)$. Substituting in $\mathcal{G}$ the cyclic shift group $\mathcal{G} = Z_n = \{0, \dots, n-1\}$ and $\Omega = \mathbb{Z}_n$, we get back the 1D convolution.

Theorem The $\mathcal{G}$-convolution is $\mathcal{G}$-equivariant: $(\rho(h)x * \theta)(g) = \langle \rho(h)x, \rho(g)\theta \rangle \rangle_{\chi(\Omega)} = \int_\Omega x(h^{-1}u)\theta(g^{-1}u)du \overset{h^{-1}u=z}{=} \int_\Omega x(z)\theta((h^{-1}g)^{-1}z)dz = (x * \theta)(h^{-1}g) = [\rho(h)(x * \theta)](g)$

Permutation Equivariance: Let $\phi : \mathbb{R}^D \to \mathbb{R}^{D'}$ be a function of a node and its neighbourhood. A permutation *equivariant* function can be cast in the form $f(X,A) = [\phi(x_1, X_{\mathcal{N}_1}), \dots, \phi(x_N, X_{\mathcal{N}_N})]$ as long as ϕ is permutation *invariant* (e.g. sum, mean, maximum). This generally can be proven by letting $\{i_1, i_2, \dots i_N\}$ be the set of permuted indices $[i_1, i_2, \dots, i_N] = [1, 2, \dots N]P$: $f(XP, PAP^\top) = [\phi(x_{i_1}, X_{\mathcal{N}_{i_1}}), \dots, \phi(x_{i_N}, X_{\mathcal{N}_{i_N}})] = ([\phi(x_{i_1}, X_{\mathcal{N}_{i_1}}), \dots, \phi(x_{i_N}, X_{\mathcal{N}_{i_N}})]P^\top)P = [\phi(x_1, X_{\mathcal{N}_1}), \dots, \phi(x_N, X_{\mathcal{N}_N})]P = f(X,A)P$

Dirichlet Energy: The Dirichlet energy E of a signal x on a graph is defined as $E(x) = \frac{1}{2} \sum_{u,v} A_{uv}(x_u - x_v)^2 = x^\top L x$, where L is the graph Laplacian and hence $\frac{\partial^2 E(x)}{\partial x^2} = L$ (also implies L is psd)

8 Optimization

8.1 Initialization

Parameters are sampled independently. Most common are:
$\theta_i^0 \sim \mathcal{N}(0, \sigma_i^2)$ or $\theta_i^0 \sim \text{Unif}[-\sqrt{3}\sigma_i, \sqrt{3}\sigma_i]$ but how to choose σ_i?

LeCun: $w_{ij} \overset{iid}{\sim} \text{Unif}[-a, a]$, $a := 1/\sqrt{n}$, $b_i = 0$ for i-th unit in n-inputs layer

- if the n inputs to a unit are iid zero mean RV's with variance γ^2 their sum will have Var $n\gamma^2$ so dividing by $\sqrt{n}$ counteracts this growth and stabalizes the variance
- zero mean assump. violated if activation fct not symmetric around 0
- More compatible with tahn than logistic
- While variance of activation reproduce the shape of the distribution will change: $Z = \prod_{i=1}^D Z_i$; $Z_i \overset{iid}{\sim} \text{Unif}[0,1]$ then $p_Z(z) = -\log(z)^{D-1}/(D-1)!$

Glorot initialization Initialize weights in layer with m units and n inputs as $w_{ij} \overset{iid}{\sim} \text{Unif}[-\sqrt{3\gamma}, \sqrt{3\gamma}]$ where $\gamma := 2/(n+m)$
To help with exploding/vanishing gradients (derived from backprop)

He init: Approach for ReLU activations. When weights in relu layer are init. symmetrically, number of active units is 1/2 in exp. so only n/2 of n inputs are propogated forward => increase var. in the init by factor of 2. $w_{ij} \overset{iid}{\sim} \mathcal{N}(0, \gamma)$ or $w_{ij} \overset{iid}{\sim} \text{Unif}[-\sqrt{3\gamma}, \sqrt{3\gamma}]$ where $\gamma := 2/n$

Orthogonal init: Non-iid init scheme. $\frac{1}{\sqrt{m}} W \sim \text{Unif}(O(m))$ s.t. $W^T W = W W^T = mI$ where m is the indentical dimension of inputs and outputs. Offers benefits in forward/backward -prop in DNN as eigenvalues of an orthogonal matrix are $\{\pm 1\}$

8.2 Weigth Decay

L_2 **regularizer:** $\Omega_\mu(\theta) = \frac{\mu}{2} \|\theta\|_2^2$, $\mu \geq 0$

Adding L_2 regularizer to emperical loss E will include weight decay in GD: $\Delta\theta = -\eta \nabla E(\theta) - \eta \nabla \Omega_\mu(\theta) = -\eta \nabla E(\theta) - \eta \mu \theta$ => params are pulled back to the origin by $-\nabla \Omega_\mu$ (strength controlled by μ).

Local loss landscape

- $\theta_\mu^* = (H + \mu I)^{-1} H \theta^*$ where $H := \nabla^2 E(\theta^*)$ and θ^* is min of $E(\theta)$
- $H = Q'\Lambda Q$ => $(H + \mu I)^{-1} H = Q'(\Lambda + \mu I)^{-1} \Lambda Q = Q' \text{diag}(\frac{\lambda_i}{\lambda_i + \mu})Q$. So along direction in param space with large eigenvalues $\lambda_i >> \mu$ weigth decay has vanishing effect while directions with $\lambda_i << \mu$ the min θ^* is shrunk

Generalization:

- Minima with smaller weight norms are **not** corr. with better generaliz.
- Weight decay projects out irrelevant directions in ill-posed problems
- Selectively shrinks weights and suppresses noise

8.3 Dropout

Co-adoptation: One unit depends on the presence of other units which leads to overfitting => dropout unit i with prob $(1-\pi_i)$

Ensemble Each instant of drop-out defines a sub-network. $p(y|x) = \sum_{b\in\{0,1\}^R} p(b)p(y|x;b)$ where $R := \#$units and $p(b) = \prod_{i=1}^R \pi_i^{b_i}(1-\pi_i)^{1-b_i}$
=> Dropout is like SGD on ensemble, pick a random sub-network (of 2^R possible) and perform a gradient step.

Test time: scale $\tilde{w}_{ij} \leftarrow \pi_j w_{ij}$ by prob of upstream unit being active

8.4 Convexity, Smoothness and GD

- $\ell : \mathbb{R}^d \to \mathbb{R}$ is convex if $\forall w, w' \in \mathbb{R}^d, \forall \lambda \in [0,1] : \ell(\lambda w + (1-\lambda)w') \leq \lambda\ell(w) + (1-\lambda)\ell(w')$
- For a diff. fct l: $\ell(w) \geq \ell(w') + \nabla\ell(w')^T(w-w') \Leftrightarrow \ell$ is convex
- If ℓ is diff. & L-smooth, then $\ell(w) \leq \ell(w') + \nabla\ell(w')^\top(w-w') + \frac{L}{2}\|w-w'\|$
- Strongly convex: $\ell(w) \geq \ell(w') + \nabla\ell(w')^T(w-w') + \frac{\mu}{2}\|w-w'\|_2^2$
- GD step $s_t := w_{t+1} - w_t = -\frac{1}{\lambda}\nabla\ell(w_t)$ (with $\lambda > 0$) minimizes a quadr. reg. 1st-order Taylor e.: $s_t = \underset{s}{\text{argmin}} \quad \ell(w_t) + s^\top\nabla\ell(w_t) + \frac{\lambda}{2}\|s\|_2^2$
  $\implies$ If ℓ is L-smooth, and $\lambda > \frac{L}{2}$, then s_t is guaranteed to decrease the objective, i.e., $\ell(w_{t+1}) \leq \ell(w_t)$
- GD strictly follows the gradient in negative direction, so its trajectory is always perpendicular to the level set

9 Regularization & Propagation

9.1 Neural Network with Skip connections

Let $x \in \mathbb{R}^d$ be the input to the network, $y_i \in \mathbb{R}^d$ the intermediate representations, $\Theta_i \in \mathbb{R}^{d\times d}$ the trainable parameters, $f : \mathbb{R}^d \to \mathbb{R}^d$ a smooth activation function, and $L : \mathbb{R}^d \to \mathbb{R}^+$ the (smooth) loss function. Also, let $\alpha > 0$ be a real parameter that controls the strength of the residual branch. Consider the residual network:
$y_1 = \alpha f(x,\Theta_1) + x, \quad y_2 = \alpha f(y_1,\Theta_2) + y_1 \quad \ldots \quad y_n = \alpha f(y_{n-1},\Theta_n) + y_{n-1}$
with loss $l = L(y_n)$

For $\mathbf{n = 1}$, we have $\frac{\partial l}{\partial x} = \frac{\partial L(y_1)}{\partial y_1}\frac{\partial y_1}{\partial x} = \frac{\partial L(y_1)}{\partial y_1}\left(\alpha\frac{\partial f(x,\Theta_1)}{\partial x} + \mathbb{I}_d\right) \in \mathbb{R}^{1\times d}$.

For general $\mathbf{n > 1}$, we have for some $1 \leq k < n$: $\frac{\partial l}{\partial \Theta_k} = \frac{\partial l}{\partial y_n}\frac{\partial y_n}{\partial y_{n-1}} \cdots \frac{\partial y_k}{\partial \Theta_k} = \frac{\partial L(y_n)}{\partial y_n}\left(\frac{\alpha\partial f(y_{n-1},\Theta_n)}{\partial y_{n-1}} + \mathbb{I}_d\right)\cdots\left(\frac{\alpha\partial f(y_k,\Theta_{k+1})}{\partial y_k} + \mathbb{I}_d\right)\cdot\frac{\alpha\partial f(y_{k-1},\Theta_k)}{\partial\Theta_k}$

Note: We add 1s on the "diagonal"terms, which helps for optimization.

9.2 Ridge Regression

The ridge regression solution for a linear model is given by
$\hat{w} = \text{argmin}_{w\in\mathbb{R}^d}\|Xw-y\|_2^2 + \frac{\lambda}{2}\|w\|_2^2 = (X^\top X + \lambda I)^{-1}X^\top y$. Using the SVD $X = UDV^\top$, we get for the predictions $\hat{y} = X\hat{w} = \sum_{i=1}^d u_i\frac{d_i^2}{d_i^2+\lambda}u_i^\top y$, where u_i are the left-singular vectors of X.

For OLS, we get predictions $\tilde{y} = X\tilde{w} = X(X^\top X)^{-1}X^\top y = \sum_{i=1}^d u_i u_i^\top y$. $\to$In ridge regression, the contributions of the left-singular vectors of X associated with small singular values are shrunk. In OLS we just project onto the column space of X.

Remark (Connection to MAP inference): Consider the model $y = Xw + \epsilon$, with $\epsilon \sim \mathcal{N}(0,\sigma^2\mathbb{I})$ and $w \sim \mathcal{N}(0,\tau\mathbb{I})$. Our objective using MAP is $\hat{w} = \text{argmax}_{w\in\mathbb{R}^d} p(w|X,y) = \cdots = \text{argmin}_{w\in\mathbb{R}^d}\|Xw-y\|_2^2 + \frac{\sigma^2}{\tau}\|w\|_2^2$, i.e., with normal noise and an isotropic normal prior on the parameters, we recover ridge regression. Note that: $\tau\downarrow \implies \lambda\uparrow$ (a strong prior belief that w is close to 0 implies a strong regularization)

9.3 Lasso Regression

The lasso regression solution for a linear model is given by
$\hat{w} = \text{argmin}_{w\in\mathbb{R}^d}\|Xw-y\|_2^2 + \frac{\lambda}{2}\|w\|_1$. Consider the L^1-regularized second-order approx. of an arbitrary loss fct. around an optimal point θ^*: $\mathcal{R}_{L^1}(\theta) \approx \mathcal{R}(\theta^*) + \frac{1}{2}(\theta-\theta^*)^\top\mathbf{H}(\theta-\theta^*) + \lambda\|\theta\|_1$.

Assuming that $\mathbf{H} = \text{diag}(h_1,...,h_d)$, with $h_i > 0$, this is equivalent to $\mathcal{R}_{L^1}(\theta) \approx \sum_{i=1}^d\left[\frac{h_i}{2}(\theta_i-\theta_i^*)^2 + \lambda|\theta_i|\right] + \text{const.}$ We find the following expression for those $\{\theta_i\}_{i=1}^d$ that minimize the approximation:
$\theta_i = \text{sign}(\theta_i^*)\cdot max\left(0, |\theta_i^*| - \frac{\lambda}{h_i}\right)$. We distinguish the following cases:

1. $0 < \theta_i^* \leq \lambda/h_i \implies$ optimal θ_i is 0: Regularization term overwhelms loss functions, pushing the value of θ_i to zero
2. $0 < \lambda/h_i < \theta_i^*$: Regularization shifts the value of θ_i by a distance of λ/h_i
3. $-\lambda/h_i \leq \theta_i^*$ and $\theta_i^* < -\lambda/h_i < 0$ are analogous

Remark: L^2-regularization with a diagonal Hessian results in **scaling** the optimal parameters, i.e., $\theta_i = \frac{h_i}{hi+\lambda}\theta_i^*$, meaning that when θ_i^* is nonzero, the optimal θ_i will be nonzero as well.

9.4 Connection between early stopping and L^2-regularization

Let λ be the weight decay factor, η the learning rate, and τ the (early) stopping time. Then, for a linear model with a quadratic approximation of the loss function, and assuming $\lambda_i/\lambda \ll 1$ and $\eta\lambda_i \ll 1$, where λ_i's are the eigenvalues of the Hessian of the quadratic approximation, it holds that: $\tau \approx \frac{1}{\eta\lambda}$.

$\to$The number τ of training iterations is **inversely proportional** to the L^2-regularization parameter λ. Thus, optimizing for many epochs is similar to enforcing a regularization with a relatively low coefficient and vice versa.

9.5 Batch and Weight Normalization

Batch normalization normalizes the data projected on the weight vector before feeding it into the (sufficiently smooth) function $l : \mathbb{R}\times\mathbb{R} \to \mathbb{R}^+$ as follows: $f(w;g,\gamma) = \mathbb{E}_{x,y}[l(\text{BN}(x^\top w;g,\gamma),y)]$, where
$\text{BN}(x^\top w;g,\gamma) = g\frac{x^\top w - \mathbb{E}[x^\top w]}{\text{Var}[x^\top w]^{1/2}} + \gamma$. We omit γ w.l.o.g. in the following.

With x being zero-mean, we have $\text{Var}[x^\top w] = w^\top Sw = \|w\|_S^2$, where $S = \text{Var}(x)$. Then, $\text{BN}(x^\top w;g) = g\frac{x^\top w}{\|w\|_S}$, and thus $\text{BN}(x^\top w;g) = x^\top\tilde{w}(w,g)$, with $\tilde{w}(w,g) := g\frac{w}{\|w\|_S}$. This means that under certain assumptions, batch normalization corresponds to a **re-parametrization of the weight space**. On the other hand, **weight** normalization parametrizes the weights as $w = g\frac{v}{\|v\|_2}$. Hence, WN is the same as BN (under certain assumptions) with the covariance matrix replaced by the identity matrix.

Remark: Both BN and WN are linear scale invariant because both perform a scaling of the weights by a norm.

Remark: Benefits of WN over BN include: (i) it is easier to use in inference mode; (ii) it permits distributed optimization because the normalization does not include dependencies between batch samples.

10 Theory

10.1 Bayesian Linear Regression:

Let $y = f(x) + \epsilon, f(x) = \phi(x)^\top w, \quad \Phi = [\phi(x_1),\ldots,\phi(x_n)], \quad y|\Phi,w \sim \mathcal{N}(\Phi^T w,\sigma^2 I)$, and $w \sim \mathcal{N}(0,\Sigma)$. Then,

- posterior: $w|y,X \sim \mathcal{N}\left(Q^{-1}\frac{\Phi y}{\sigma^2}, Q^{-1}\right)$ with $Q = \left(\frac{\Phi\Phi^\top}{\sigma^2} + \Sigma^{-1}\right)$
- posterior pred.: $\phi_\star^\top w \sim \mathcal{N}\left(\phi_\star^T Q^{-1}\frac{\Phi y}{\sigma^2}, \phi_\star^\top Q^{-1}\phi_\star\right)$, where $\phi(x^\star) = \phi_\star$

Note that for $\Phi \in \mathbb{R}^{N\times n}, N >> n$ the computation of the matrix inverse above is of order $\mathcal{O}(N^3)$. Hence, using Woodburry's identity and the pushthrough identity we can reduce the complexity to $\mathcal{O}(n^3)$: $f_\star(x_\star) \sim \mathcal{N}(\phi_\star^\top\Sigma\Phi(\Phi^T\Sigma\Phi + \sigma^2 I_n)^{-1}y, \phi_\star^\top\Phi\phi_\star - \phi_\star^\top\Sigma\Phi(\sigma^2 I_n + \Phi^\top\Sigma\Phi)^{-1}\Phi^\top\Sigma\phi_\star)$, where the feature representation only enters in form of a kernel $\mathcal{K}(x,x') = \psi(x)^\top\psi(x') \in \mathbb{R}$ with $\psi(x) = \Sigma^{1/2}\phi(x)$

Bayesian Ensembling: We use relative posterior weighting to average outputs of set of DNNs.
$f(\Theta)(x) = \sum_{i=1}^n\frac{p(\theta^i|S)}{\sum_{j=1}^n p(\theta^j|S)}f(\theta^i)(x) = \sum_{i=1}^n\frac{\exp[-E(\theta^i)]}{\sum_{j=1}^n\exp[-E(\theta^j)]}f(\theta^i)(x)$
The above is feasible even if we do not have direct access to the normalized posterior, we can still use unnormalized posterior via energy function E.

Markov Chain Monte Carlo: It is a standard approach to sampling from high-dimensional posterior distributions. We define a Markov chain (random sequence) in parameter space $\theta^0,\theta^1,\theta^2,...$, s.t. $\theta^{t+1}|\theta^t \sim \Pi$.We choose next parameter based on previous one w.r.t. Π, which is Markov kernel of the chain. Posterior is the stationary distribution of the Markov chain. Lastly, one usually needs burn-in period for better estimates.

Metropolis-Hastings: This is a way of specifying the kernel. We define detailed balance condition and Markov kernel s.t.
$p(\theta_1 \mid S)\Pi(\theta_2 \mid \theta_1) = p(\theta_2 \mid S)\Pi(\theta_1 \mid \theta_2)$.
If this condition holds, the resulting Markov chain is time-reversible and has the desired posterior as its unique stationary distribution

Remark: Markov chains can have stationary distributions without fulfilling detailed balance.

In MH sampling, we start from initial $\tilde{\Pi}$ but then modify the transition probability via an acceptance/rejection step to achieve an effective Π meeting detailed balance. Assume that we always want to accept the transition in one of the directions (increasing probability usually), but possibly reject it with some probability in the opposite direction. Thus, we get

$$p(\theta_1|\mathcal{S})\underbrace{\tilde{\Pi}(\theta_2|\theta_1)A(\theta_2|\theta_1)}_{=:\Pi(\theta_2|\theta_1)} \stackrel{!}{=} p(\theta_2|\mathcal{S})\underbrace{\tilde{\Pi}(\theta_1|\theta_2)A(\theta_1|\theta_2)}_{=:\Pi(\theta_1|\theta_2)}.$$

With the desired one-sided structure of acceptance probabilities $A(\theta_2 \mid \theta_1) = 1 \vee A(\theta_1 \mid \theta_2) = 1$ which leads to the unique choice
$A(\theta_1 \mid \theta_2) = \min\left\{1, \frac{p(\theta_1|S)\tilde{\Pi}(\theta_2|\theta_1)}{p(\theta_2|S)\tilde{\Pi}(\theta_1|\theta_2)}\right\}$.

Hamiltonian Monte Carlo: Consider an energy function equalt to negative log posterior $E(\theta) = -\sum_{x,y}\log p(y \mid x;\theta) - \log p(\theta)$. This potential function is augmented by momentum v and kinetic energy term s.t. $H(\theta,v) = E(\theta) + \frac{1}{2}v'M^{-1}v$. Then the joint probability distribution is given by Gibbs distribution $p(\theta,v) \propto \exp[-H(\theta,v)]$. Hamiltonian dynamics reformulate the Euler-Lagrange equations as$\dot{v} = -\nabla E(\theta), \quad \dot{\theta} = v$. In Hamiltonian Monte Carlo (HMC), these are discretized using a step size η: $\theta^{t+1} = \theta^t + \eta v^t, \quad v^{t+1} = v^t - \eta\nabla E(\theta^t)$ HMC relates to gradient descent with momentum and samples the posterior correctly, but it is slow. To scale HMC for stochastic gradients, Langevin Dynamics is used to make the posterior invariant under stochastic updates.

Langevin Dynamics: An important contribution was the introduction of Langevin dynamics with friction and noise process
$\dot{\theta} = v, \quad dv = -\nabla E(\theta)dt - Bv\,dt + \mathcal{N}(0,2B\,dt)$. Friction reduces the momentum and dissipates kinetic energy. Wiener noise process injects stochasticity. This SDE can be discretized via symplectic Euler, resulting in (friction, stochastic, extra noise):
$\theta^{t+1} = \theta^t + \eta v^t, \quad v^{t+1} = (1-\eta\gamma)v^t - \eta s\nabla\tilde{E}(\theta) + \sqrt{2\gamma\eta}\,\mathcal{N}(0,I)$,
where $\tilde{E}$ is a stochastic potential function, which includes an empirical loss over a random minibatch of data. The friction leads to an exponential damping with time.

10.2 Gaussian Processes

A Gaussian Process is a collection of random variables, any finite number of which have a joint Gaussian distribution. It is completely specified by its mean function $m(x) = \mathbb{E}[f(x)]$ and covariance function $k(x,x') = \mathbb{E}[(f(x)-m(x))(f(x')-m(x'))]$ and we write $f(x) \sim \mathcal{GP}(m(x),k(x,x'))$.

Exa: A Bayesian linear model $f(x) = \phi(x)^\top w, w \sim \mathcal{N}(0,\Sigma)$ is a Gaussian Process since for any $m, \quad (f(x_1),\ldots,f(x_m))$ it holds that $\sum_{i=1}^m\alpha_i f(x_i) = \sum_{i=1}^m\alpha_i\phi(x_i)^\top w = (\sum_{i=1}^m\alpha_i\phi(x_i))^\top w$ which, as a linear combination of Gaussians, is Gaussian. Hence, $f(x) \sim \mathcal{GP}(0,\phi(x)^\top\Sigma\phi(x'))$.

Linear Unit & Layer: Consider a linear unit with n inputs and a random Gaussian weight vector $w \sim \mathcal{N}\left(0, \frac{\sigma^2}{n}I_{n\times n}\right)$. The units outputs for a vector of inputs can be written as $y_i = w'x_i$ and any linear combination of y writes as $\sum_{i=1}^s\alpha_i y_i = w'\bar{x}, \quad \bar{x} := \sum_{i=1}^s\alpha_i x_i$. The pre-activation of a single linear unit follows a Gaussian process due to the properties of normal distributions under linear combinations. Its covariance function is the scaled inner product kernel, given as
$\mathbb{E}[y_i y_j] = \mathbb{E}[(w\cdot x_i)(w\cdot x_j)] = x_i'\mathbb{E}[ww']x_j = x_i'\left(\frac{\sigma^2}{n}I_{n\times n}\right)x_j = \frac{\sigma^2}{n}x_i'x_j$.
This concept applies to single layers of linear units where activations are independent, conditioned on the inputs.

Deep Layers: In the next layers there is randomness in weights and also in activations. We observe products between random matrices of the type $W^{l+1}X^l, \ l \geq 1$, where $X^l = X^l(W^l,...,W^1)$ is the activation matrix of layer l and is random. Thus, such products are not normal, although a deep pre-activation process yields normality due to CLT.

Non-linear Layer Maps: A non-linear activation function ϕ reshapes pre-activations which are no longer Gaussian. In the GP limit of pre-activations, the mean and covariance of units are sufficient to work with, since activations of layer l are summed over to form pre-activations of $l+1$. CLT reshapes them back to Gaussian such that the mean functions writes as $\mu(x^{l+1}) = \mathbb{E}[\phi(W^l x^l)]$. DNNs can be interpreted as Gaussian Processes (GPs) in the infinite width limit, using kernel recursion. Then $K_{\mu\nu}^l = \mathbb{E}[\phi(x_{i\mu}^{l-1})\phi(x_{i\nu}^{l-1})] = \sigma^2\mathbb{E}[\phi(f_\mu)\phi(f_\nu)], \quad f \sim GP(0,K^{l-1})$.

As for NTK, we hereby use kernel regression and Bayesian predictive mean looks like $f^*(x) = \mathbf{k}(x)'K^+\mathbf{y}$. Adv. of GP view: We can also quantify the conditional variance: $\mathbb{E}[(f(x)-f^*(x))^2] = K(x,x) - \mathbf{k}(x)'K^+\mathbf{k}(x)$.

10.3 Empirical Neural Tangent Kernel

$K(x,x';\theta) = \nabla_\theta f(x;\theta)^\top\nabla_\theta f(x';\theta) = \sum_{p=1}^P\frac{\partial f(x;\theta)}{\partial\theta_p}\frac{\partial f(x';\theta)}{\partial\theta_p}$

Using the NTK, we can track how our function changes in the sample during training (functional gradient flow): $\dot{\mathbf{f}} = K(\theta)(\mathbf{y} - \mathbf{f})$.

Exa: $f(x) = w^\top Vx, w \in \mathbb{R}^m, V \in \mathbb{R}^{m\times d}$
$\rightsquigarrow \nabla_\theta f(x)^\top\nabla_\theta f(x') = (x^\top V^\top w^\top \otimes x^\top)\begin{pmatrix}Vx'\\ w\otimes x'\end{pmatrix} = \langle Vx,Vx'\rangle + (w^\top\otimes x^\top)(w\otimes x') = \langle Vx,Vx'\rangle + (w^\top w)\otimes(x^\top x') = \langle Vx,Vx'\rangle + \|w\|^2\langle x,x'\rangle$

Infinite Width Limit Consider an MLP with width sequence m_l and weights initialized as
$w_{ij}^l = \frac{\sigma_w}{\sqrt{m_l}}\omega_{ij}^l, \quad b_i^l = \frac{\sigma_b}{\sqrt{m_l}}\beta_i^l, \quad \omega_{ij}^l,\beta_i^l \sim \mathcal{N}(0,1)$.
This scaling normalizes weight magnitudes by the square root of layer width (similar to LeCun initialization). In the infinite width limit where $m_l \to \infty$ for all hidden layers, under suitable technical conditions (including Lipschitz activation functions), the initial Neural Tangent Kernel (NTK) converges in probability to a deterministic limit $k(\theta) \xrightarrow{P} k^\infty$. The limiting kernel k^∞ depends only on the initialization distribution, not the specific random parameters. This effectively means there is only one infinite width network at initialization. This result can be intuitively understood through the law of large numbers: as width approaches infinity, the sampled population of units in each layer approaches the population distribution.

NTK Constancy Convergence to deterministic NTK k^∞ cannot only be established pointwise, but with a few additional technical assumptions, convergence also holds uniformly over training trajectories s.t. $k(\theta(t)) \to k^\infty \quad \text{uniformly over} \quad t \in [0;T]$.

Note: Activation functions need to have bounded second derivatives and the time-integral of the norms of update directions needs to remain stochastically bounded. The latter assumption can be verified for gradient flow trajectories.

This remarkable property can also be stated (non-rigorously) as follows: the initial NTK stays constant under gradient flow $\frac{dk(\theta(t))}{dt} = 0$.

Under NTK constancy, learning in the infinite width limit becomes equivalent to the linearized model, with the solution s.t. $f^\infty(x) = \mathbf{k}(x)K^+(\mathbf{y} - \mathbf{f}), \quad k = k^\infty$

The emergence of NTK constancy can be explained through vanishing curvature in the infinite width limit, where the Hessian operator norm scales as $\frac{|\nabla^2 f(\theta_0)|_2}{|\nabla f(\theta_0)|_2^2} \ll 1$. This property relates to the lazy training regime, where DNN training becomes a convex problem in the infinite width limit. For empirical NTK (gram matrices), near-constancy can be established as $|k(\theta_0) - k(\theta_t)|_F^2 \in \mathcal{O}(1/m), \quad m = m_1 = \cdots = m_L$. This holds under technical assumptions including bounded input domain $\mathcal{X} = x : |x|_2 \leq 1$ and appropriate learning rate conditions.

10.4 Statistical Learning Theory

VC Learning Theory: Possible classif. outomes of a fct. class on sample $S = \{x_1,\ldots,x_s\}$: $\mathcal{F}(S) = \{(f(x_1),\ldots,f(x_s)) \in \{-1,1\}^s : f \in \mathcal{F}\}$. **VC dimension** is the largest possible sample size for which the function class $\mathcal{F}$ is still able to shatter a sample. Let $\hat{E}(f)$ be the empirical error and $E(f)$ expected error. The uniform convergence yields VC inequality $\mathbb{P}\{\sup_\mathcal{F}|\hat{E}(f) - E(f)| > \epsilon\} \leq 8|\mathcal{F}(s)|e^{-s\epsilon^2/32}$, yielding a non-vacuous bound on generalization error in the limit, provided that the VC dimension of $\mathcal{F}$ is finite, which means that the shattering coefficient grows subexponentially.

Randomization Experiments: Observations from Zhang et al.:

1. DNNs can perfectly fit training data
2. DNNs can possibly memorize training data perfectly (even when true labels are replaced with random labels), and thus learn nothing and cannot generalize.
3. Transition from true to random labels increase train time only by small factor. Gradient-based methods easily yields memorization solutions.
4. These exist, even when randomly shuffling the pixels of a DNN.

DNNs generalize despite infinite capacity, which defies traditional learning theory. They can memorize random data, and ConvNets' inductive bias, like shift invariance, offers little advantage, showing that these phenomena are not data-dependent.

Double Descent: There is an interpolation threshold at which the DNN "memorizes"training examples with little generaliz. But beyond this point, even larger models start to learn and may eventually reach lower risk. This is called the *over-parametrized regime*, which is not harmed by over-fitting.

PAC Bayesian Bounds: We consider a stochastic classifier f and aim to bound the expected generalization gap. The generalization gap is defined as $max(0, E - \hat{E})$, where $\hat{E}$ is the empirical error w.r.t. the training sample, and E is expected error w.r.t. the data distribution. We make use of the change of measure inequality.

Theorem: For any $P \gg Q$ and P-measurable ϕ (i.e., a random variable), $\mathbb{E}_Q[\phi] \leq \text{KL}(Q\|P) + \ln\mathbb{E}_P\left[e^\phi\right]$. Proof: Via Jensen's inequality,
$\ln\mathbb{E}_P\left[e^{\phi(f)}\right] \geq \ln\mathbb{E}_Q\left[e^{\phi(f)}\frac{P(f)}{Q(f)}\right] \geq \mathbb{E}_Q\ln\left[e^{\phi(f)}\frac{P(f)}{Q(f)}\right] = \mathbb{E}_Q[\phi(f)] - \mathbb{E}_Q[\ln Q(f) - \ln P(f)]$, where the last term is equivalent to $\text{KL}(Q\|P)$.

We state famous PAC-Bayesian Theorem, ensuring a general rate of $\mathcal{O}(1/\sqrt{s})$ and there are no hidden constants. This bound is as simple as it gets, but it only bounds stochastic classifier, not a single classifier.

PAC-Bayesian for DNNs: Define a Gaussian over parameters $P := \mathcal{N}(\theta_0,\lambda^2 I)$. The setting $\theta_0 = 0$ and $\lambda = 1$ yields a standard Gaussian s.t. $Q := \mathcal{N}(\theta,\text{diag}(\sigma_i^2))$. With P and Q being Gaussians with diagonal covariance, one can derive $\text{KL}(Q|P)$: $\text{KL}(Q|P) = \sum_i\log\frac{\lambda}{\sigma_i} + \frac{\sigma_i^2+\theta_i^2}{2\lambda^2} - \frac{1}{2}$
In the limit $\lambda \to \infty$, $\text{KL}(Q|P) \to -\sum_i\log\sigma_i$, favoring large variances for the Q-ensemble. The PAC-Bayes bound to be minimized is:
$E_{\text{PAC}}(Q) := \mathbf{E}_Q[\hat{E}] + \sqrt{\frac{2}{s}\left[\text{KL}(Q|P) + \ln\left(\frac{2\sqrt{s}}{\epsilon}\right)\right]}$

A practical implementation involves generating parameter sequence θ^t via SGD, evaluating gradients on perturbed parameters sampled from the Q-ensemble: $\theta = \theta - \eta\nabla E_Q[\hat{E}], \quad \tilde{\theta} \sim Q(\theta,\sigma)$ The reparameterization trick enables backpropagation: $\tilde{\theta} = \theta + \text{diag}(\sigma_i)\eta, \quad \eta \sim \mathcal{N}(0,I)$ This approach balances between finding parameters θ that achieve low expected loss and maintaining robustness through the variance in Q, which affects the KL-divergence term in the bound.

Wide Local Minima: Flatness of local minima with better generalization. $\to$Argued that SGD with small mini-batches leads to flatter minima comp. to full batch gradient descent. Also, weight avg. found to be favorably towards finding flatter mimima. $\to$entropy SGD uses Langevin dynamics to favor minima with higher entropy.

11 Generative Models

11.1 VAEs

- AE: $x \mapsto z = Cx, C \in \mathbb{R}^{m\times n}$, used with $E(C,D)(x) = \frac{1}{2}\|x - DCx\|^2$
  - Goal: find good embedding z : bottleneck layer
- Solution can be obtained using red. SVD, leading to optimality condition: $DCX = \hat{X} = U\Sigma_m V^\top, \Sigma_m = diag(\sigma_1,\ldots,\sigma_m,0,\ldots)$
- For centered data, equiv. to PCA and optimal C, is any matrix that projects pattens to subspace by m-principal eigenvectors of $XX^\top$ $(D = C^+)$
- Lin. AE objective has **no** non-global minima

Linear Factor Analysis

- Latent variable model known as *factor analysis*, (1) define a pdf p_Z over latents, (2) define $p_{X|Z}$, (3) obtain $p_X(x) = \int p_Z(z)p_{X|Z}(x|z)dz$
- Gaussian Prior: $z \sim \mathcal{N}(0,I), z \in \mathbb{R}^m$ and lin. obs model: $x = \mu + Wz + \eta$, $\eta \sim \mathcal{N}(0,\Sigma), \Sigma = diag(\sigma_1^2,\ldots,\sigma_n^2)$
- Assume $\eta \perp z$, typically: $m << n$, $\hat{\mu} = \frac{1}{s}\sum_{i=1}^s x_i$ (data centering) induces $x \sim \mathcal{N}(\mu, WW^\top + \Sigma)$
- For W and any orthog. $m\times m$ matrix $Q \to (WQ)(WQ)^\top = WW^\top \to$ factors only identif. upto orthog. transf. (rot, refl., permutat.)
- $\to$In factor analysis, you *cannot* identify a unique set of factors or loadings W; you can only identify them up to this "orthogonal freedom"
- Encoder implicitly defines gauss. *posterior* via Bayes' rule: $\mu_{z|x} = W^\top(WW^\top + \Sigma)^{-1}(x-\mu)$ and $\Sigma_{z|x} = I - W^\top(WW^\top + \Sigma)^{-1}W$
- If $\Sigma = \sigma^2 I$ and $\sigma \to 0 \to \mu_{z|x} = W^\top(WW^\top + \sigma^2 I)^{-1} \to W^+ \in \mathbb{R}^{m\times n}$ (pseudoinv.)
  - $\mu_{z|x} \to W^+(x-\mu), \Sigma_{z|x} \to 0$
- Use MLE to find parms.: $\mu, W \overset{max}{\leftarrow} logp(\mu,W)(\mathcal{S})$ (No closed form sol.)
- For $\Sigma = \sigma^2 I$ (probabilistic PCA) optim. cond. for i-th column of W: $w_i = \rho_i u_i, \rho_i = max\{0, \sqrt{\lambda_i - \sigma^2}\}$
  - (λ_i, u_i) is i-th Princ. eigenvalue/-vector of covariance matrix.
  - $\sigma \to 0 \equiv$ PCA

Variational Autoencoder: (factor analysis+depth+non-linear activations)

- $z \overset{iid}{\sim} \mathcal{N}(0,I)$ and propag. through NN $x = F(\theta)(z) = (F^L \circ\cdots\circ F^1)(z)$

Derive ELBO: $\log p_\theta(x) = \log\int p_\theta(x \mid z)p(z)dz = \log\int q(z)\frac{p_\theta(x|z)p(z)}{q(z)}dz$
$\geq \int q(z)\log\big(p_\theta(x \mid z)\big)dz - \int q(z)\log\left(\frac{q(z)}{p(z)}\right)dz := \mathcal{L}(\theta,q)(x)$, with $D_{\text{KL}}(q\|p)$

- $\theta \overset{max}{\to} \mathcal{L}(\theta,q)(\mathcal{S}) = \sum_{i=1}^s\mathcal{L}(\theta,q)(x_i)$. $q(.|x)$ can be any distrib. over latent vars|x, optimal $q(z|x) = p(z|x) \equiv$ EM
- Practice: restrict q to var. family (e.g., iid normal) $z \sim \mathcal{N}(\mu(x),\Sigma(x))$, $\Sigma(x) = diag(\sigma_1^2(x),...,\sigma_n^2(x))$
- **Optimizing** θ: (1) sample from z (2) perform backprop and SGD step for θ treating sampled z as fixed input (no MCMC needed) $\to$ unbiased estimate of gradient and small $\uparrow$ var
- **Optimizing** q: Use stoch. backprop (1) reparam. parametrized distr. by fixed unpara. randomness e.g., $z \sim \mathcal{N}(\mu,\Sigma) \Leftrightarrow z = \mu + \Sigma^{\frac{1}{2}}\eta$, $\eta \sim N(0,I)$ (2) parameters no longer hidden $\to$ use backprop on μ and $\Sigma^{\frac{1}{2}}$
- Approach generiz. for any smooth and integrable f $\nabla_\mu\mathbf{E}[f(z)] = \mathbf{E}[\nabla_z f(z)]$, $\nabla_\Sigma\mathbf{E}[f(z)] = \frac{1}{2}\mathbf{E}[\nabla_z^2 f(z)]$

Reparametrizations:

- $\epsilon \sim \mathcal{U}[0,1] \rightsquigarrow (b-a)\epsilon + a \sim \mathcal{U}[a,b]$
- $\epsilon \sim \mathcal{N}(0,1) \rightsquigarrow \mu + \sigma\epsilon \sim \mathcal{N}(\mu,\sigma^2)$
- $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0},\boldsymbol{I}) \rightsquigarrow \boldsymbol{\mu} + \Sigma^{1/2}\boldsymbol{\epsilon} \sim \mathcal{N}(\boldsymbol{\mu},\Sigma)$
- $\epsilon \sim \mathcal{N}(0,1) \rightsquigarrow \exp(\mu + \sigma\epsilon) \sim \text{LogNormal}(\mu,\sigma^2)$

11.2 GANs

(derive training signal without likelihood or ELBO)

- Goal: Derive a Training signal from classifier that discrimin. true data samples from model-generated samples (discriminator). $\tilde{p}_\theta(x,y) = \frac{1}{2}(yp(x) + (1-y)p_\theta(x)$. $y \in \{0,1\}$
- p: true probability, p_θ: model
- Bayes optim. class.: $q_\theta(x) := \mathbb{P}\{y = 1|x\} = \frac{p(x)}{p(x)+p_\theta(x)} \to$ posterior prob. of x being real
- Now, Train generator by **minimizing** the logistic likelihood $\theta \overset{min}{\to} \ell^*(\theta) := \mathbf{E}_{\tilde{p}_\theta}[yln(q_\theta(x)) + (1-y)ln(1-q_\theta(x))]$
- JS-Divg. obj. gets minim. when minim.: $\mathbf{E}_{\tilde{p}_\theta}[\ldots] = JS(p,p_\theta) - ln2$
- JS-Divg. is a symmetrized version of KL-Divg. $\to$ instead of max likeli., minimiz. JS-Divg.
- JS-Divg: $D_{\text{JS}}(P\|Q) = \frac{1}{2}D_{\text{KL}}\left(P\middle\|\frac{P+Q}{2}\right) + \frac{1}{2}D_{\text{KL}}\left(Q\middle\|\frac{P+Q}{2}\right)$.

Discriminators

- Define parametrized classif. instead of bayes opt. $q_\phi : x \mapsto [0;1], \phi \in \Phi$
- Obj. w/ bound:$\ell^*(\theta) \geq \sup_{\phi\in\Phi}\ell(\theta,\phi) := \mathbf{E}_{\tilde{p}_\theta}[y\ln q_\phi(x) + (1-y)\ln(1-q_\phi(x))]$

GAN Training (Saddle Pt problem) $\theta^* := \text{argmin}_{\theta\in\Theta}\{\sup_{\phi\in\Phi}\ell(\theta,\phi)\}$.
Impractical to perform inner sup, so use backprop with extra gradient step (to stabilize training): $\theta^{t+1} = \theta^t - \eta\nabla_\theta\ell\left(\theta^{t+\frac{1}{2}},\phi^t\right), \quad \theta^{t+\frac{1}{2}} := \theta^t - \eta\nabla_\theta\ell\left(\theta^t,\phi^t\right)$, $\phi^{t+1} = \phi^t + \eta\nabla_\phi\ell\left(\theta^t,\phi^{t+\frac{1}{2}}\right), \quad \phi^{t+\frac{1}{2}} := \phi^t + \eta\nabla_\phi\ell\left(\theta^t,\phi^t\right)$.
$\to$Make virtual step forward to eval. the gradient from which to calculate the actual update direction. Req. two grad.u steps per parameters change

11.3 Exercises

- Mode collapse of GANs: the failure of the generator to produce variety / diversity of samples, by collapsing many values of z to the same sample x (partial collapse occurs often)
- GANs are typically trained by taking a single (or a few) update steps for the discriminator in the inner loop and a single (or a few) update steps for the generator in the outer loop, in order for the discriminator not to become too strong (gradient vanishes)
- Backpropagation of GANs doesn't work on discrete data
- VAE: well defined objective, but no discrete latent variable and vice versa for GANs

We obs. that GAN training approx. minimizes the JS-div. GANs however can even be extended to arbitrary f-diverges. $D_f(P\|Q) = \int_X q(x)f\left(\frac{p(x)}{q(x)}\right)dx$

- $f : \mathbb{R}_+ \to \mathbb{R}$ is a convex, lower-semicontinous fct & $f(1) = 0$
- $f(u) = -(u+1)\log\frac{1+u}{2} + u\log(u) \to$ JS-Div
- For $T : X \to \mathbb{R}$, T is an arbitrary function within some fam of fcts.
- Using $f(u) = \underset{t\in\mathbf{dom}_{f^*}}{sup}\{tu - f^*(t)\}$ (bi-conjugate, convex conj, fenchel conj.) one can write $D_f(P\|Q) \geq \underset{T\in}{sup}(\mathbb{E}_{x\sim D}[T(x)] - \mathbb{E}_{x\sim Q}[f^*(T(x))])$

We train NNs Q_θ (gen) and T_ω (variation fct, takes sample and outputs scalar; disc). Now we write the var. Low.-bound as $F(\theta,\omega) = \mathbb{E}_{x\sim P}[T_\omega(x)] - \mathbb{E}_{x\sim Q_\theta}[f^*(T_\omega(x))]$

- $T_\omega = g_f(V_\omega(x))$ where $V_\omega : X \to \mathbb{R}$ and $g_f : \mathbb{R} \to \underset{f^*}{dom}$
- Using $g_f(v) = -log(1 + exp(-v))$ and $f^*(t) = -log(1 - exp(t)) \to$ we retrieve the GAN obj: $\mathbb{E}_{x\sim P}[log(D_\omega(x))] + \mathbb{E}_{x\sim Q_\theta}[log(1 - D_\omega(x)))]$
- Benefits of optimizing JS divergence over standard MLE training: In standard MLE, we optimize forward KL $= \int_x p(x)\frac{p(x)}{q(x)}dx$. Thus, when $p(x) = 0$, $q(x)$ is ignored. Thus, model tends to spread its mass everywhere where $p(x) > 0$ and is zero otherwise ($\to$blurry samples). Reverse KL has opposite problem ($\to$mode collapse as model only focuses on certain modes of the true distribution). JS mitigates this problem by combining the two: $JS(Q\|P) = \frac{1}{2}KL(P\|\frac{P+Q}{2}) + \frac{1}{2}KL(Q\|\frac{P+Q}{2})$

Assume: $\mathbf{z} \sim N(\mathbf{0},\mathbf{1})$, $z \in \mathbb{R}^m$ and $\mathbf{x} = \mathbf{x_0} + \mathbf{W} + \eta$, $\eta \sim \mathcal{N}(\mathbf{0},\mathbf{\Psi})$

- $\Psi := diag(\sigma_1^2,\ldots,\sigma_n^2)$
- $\to \binom{z}{x} \sim \mathcal{N}\left(\binom{\mathbf{0}}{\mathbf{x_0}}, \begin{pmatrix}\mathbf{I} & \mathbf{W}^\top\\ \mathbf{W} & \mathbf{WW}^\top + \mathbf{\Psi}\end{pmatrix}\right)$
- No closed form solution to perform MLE, need to use EM-algorithm

11.4 Diffusion models

Stochastic process $x_{0:\infty} := (x_0,x_1,...)$ is called a **Markov chain** if future and past are cond. ind. given present: $x_{0:t-1} \perp x_{t+1:\infty}|x_t, \forall t$. This implies marginal over temporal slice $p(x_{s:t}) = p(x_s)\prod_{\tau=s+1}^t p(x_\tau|x_{\tau-1})$ (**Markov kernel**). Markov chain can equivalently be expressed in **time-reversed form**: $p(x_{s:t}) = p(x_t)\prod_{\tau=s+1}^t p(x_{\tau-1}|x_\tau)$. Markov chain $x_{0:\infty}$ is called **time-homogeneous** if $p(x_t|x_{t-1}) = p(x_1|x_0), \forall t$, otherwise **time-inhomogeneous**. A **state distr.** π is called **invariant** if $\pi(x_{t+1}) = \int\pi(x_t)p(x_{t+1}|x_t)dx_t$.

Denoising diffusion: Goal is to find flex. class of models to approx. high-dim. empirical data distr. via parametric transfo. of a simple source distr.: $\pi \mapsto \mu(\theta) \stackrel{!}{\simeq} \pi_* \overset{iid}{\sim} \{x^1,...,x^s\}$. Denoising diff. learns such a transfo. in multi-step process, aiming to generate a trajectory of probability measures $\pi = \mu_T \overset{\theta}{\mapsto} \mu_{T-1} \overset{\theta}{\mapsto} \ldots \overset{\theta}{\mapsto} \mu_1 \overset{\theta}{\mapsto} \mu_0 \stackrel{!}{\simeq} \pi_*$. This is thought of as an inversion of a typically **fixed forward trajectory** $\pi_* = \nu_0 \mapsto \nu_1 \mapsto \cdots \mapsto \nu_{T-1} \mapsto \nu_T = \pi$ (Markov chain with π as its invariant measure; will gradually forget initial π_* and approach invariant distr. in limit). *Ex.:* $\pi \simeq \mathcal{N}(0,\mathbb{I}), x_t|x_{t-1} \sim \mathcal{N}(\sqrt{1-\beta_t}x_{t-1},\beta_t\mathbb{I})$ w/ noise schedule $\beta_1,\ldots,\beta_T$

SDE view of denoising diff.: $dx_t = -\frac{1}{2}\beta_t x_t dt + \sqrt{\beta_t}dw_t$ (forward proc.).
$dx_t = [-\frac{1}{2}\beta_t x_t - \beta_t\nabla_{x_t}\log q_t(x_t)]dt + \sqrt{\beta_t}d\overline{w}_t$ (backward proc. = denoising)

ELBO view of denoising diff.: $x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon_t$, $\epsilon_t \sim \mathcal{N}(0,\mathbb{I})$.
q: kernel & marginal probs. of fwd. Markov chain; p_θ: time-rev. chain.
$\to\ln p_\theta(x_0) = \ln\int q(x_{1:T}|x_0)\frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}dx_{1:T} \geq \mathbb{E}_q[\ln\frac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}|x_0]$.
Exploiting Markov property, can break up ELBO into log-ratios per step: $\ln p_\theta(x_0) \geq \sum_{t=0}^T L_t$, which is defined as: $\mathbb{E}[\ln p_\theta(x_0|x_1)]$ if $t = 0$, as $-D(q(x_T|x_0)\|\pi)$ if $t = T$, and as $-D(q(x_{t-1}|x_t,x_0)\|p_\theta(x_{t-1}|x_t))$ otherwise.

Note: $x_T \sim \pi$ (noise distr.).

Now, define $\alpha_t := 1 - \beta_t$, $\overline{\alpha}_t := \prod_{i=1}^t\alpha_i$, $\overline{\beta_t} := 1 - \overline{\alpha}_t$
Then $x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}z_t$, $z_t \sim \mathcal{N}(0,\mathbb{I}) \implies x_t \sim \mathcal{N}(\sqrt{\overline{\alpha}_t}x_0,\overline{\beta}_t\mathbb{I})$
Using Bayes' rule on $x_{t-1}|x_t,x_0$: $q(x_{t-1}|x_t,x_0) = \frac{1}{Z}q(x_t|x_{t-1},x_0)q(x_{t-1}|x_0) = \frac{1}{Z}\exp\left(-\frac{1}{2}\left[\left(\frac{1}{1-\overline{\alpha}_{t-1}} + \frac{\alpha_t}{\beta_t}\right)x_{t-1}^2 - 2\left(\frac{\sqrt{\overline{\alpha}_t}x_t}{\beta_t} + \frac{\sqrt{\overline{\alpha}_{t-1}}x_0}{1-\overline{\alpha}_{t-1}}\right)x_{t-1} + \cdots\right]\right)$. Bringing this into the form $\frac{1}{Z}\exp\left(-\frac{1}{2}a\left[x_{t-1}^2 - 2\frac{b}{a}x_{t-1} + \cdots\right]\right)$ and completing the square gives $\propto \exp\left(-\frac{1}{2}a\left(x_{t-1} - \frac{b}{a}\right)^2\right)$.

Hence, $q(x_{t-1}|x_t,x_0) = \mathcal{N}(m(x_t,x_0,t),\tilde{\beta}_t\mathbb{I})$, with $m(x_t,x_0,t) = \frac{\sqrt{\alpha_t}(1-\overline{\alpha}_{t-1})}{1-\overline{\alpha}_t}x_t + \frac{\sqrt{\overline{\alpha}_{t-1}}\beta_t}{1-\overline{\alpha}_t}x_0$, and $\tilde{\beta}_t = \frac{1-\overline{\alpha}_{t-1}}{1-\overline{\alpha}_t}\beta_t$.

Rewrite x_0 in terms of x_t and $\epsilon \sim \mathcal{N}(0,\mathbb{I})$: We know $x_t \sim \mathcal{N}(\sqrt{\overline{\alpha}_t}x_0,\overline{\beta}_t\mathbb{I})$; thus $x_t = \sqrt{\overline{\alpha}_t}x_0 + \overline{\beta}_t\epsilon = \cdots \implies x_0 = \frac{1}{\sqrt{\overline{\alpha}_t}}x_t - \frac{\sqrt{1-\overline{\alpha}_t}}{\sqrt{\overline{\alpha}_t}}\epsilon$.

Plug this into $m(x_t,x_0,t)$ to get $m(x_t,\epsilon,t) = \frac{1}{\sqrt{\alpha_t}}(x_t - \frac{\beta_t}{\sqrt{1-\overline{\alpha}_t}}\epsilon)$. Hence, $q(x_{t-1}|x_t,\epsilon) = \mathcal{N}(m(x_t,\epsilon,t),\tilde{\beta}_t\mathbb{I})$.

Now, fix $p_\theta(x_{t-1}|x_t)$ to be Gaussian with variance $\tilde{\beta}_t$ and mean defined as above, **but** we predict ϵ by a NN $\epsilon_\theta(x_t,t)$! I.e., $p_\theta(x_{t-1}|x_t) = \mathcal{N}(m(x_t,\epsilon_\theta,t),\tilde{\beta}_t\mathbb{I})$. To generate samples similar to training data, we would like to max. $\ln p_\theta(x_0)$ for training sample x_0.
$\to$Max. lower bound from above (while enforcing $p_\theta(x_0|x_1) = q(x_0|x_1)$):
$L_t = \cdots = \rho_t\|\epsilon - \epsilon_\theta\|_2^2$ with $\rho_t := \frac{\beta_t^2}{2\tilde{\beta}_t\alpha_t(1-\overline{\alpha}_t)}$. Thus, max. ELBO on $\ln p_\theta(x_0)$ is equivalent to min. $\rho_t\|\epsilon - \epsilon_\theta(x_t,t)\|_2^2$

Using further simplification, we get $h(\theta)(x) = \frac{1}{T}\sum_{t=1}^T\mathbf{E}[\|\epsilon - \epsilon_\theta(\sqrt{\overline{\alpha}_t}x_0 + \sqrt{1-\overline{\alpha}_t}\epsilon,t)\|^2]$. For training, this means: randomly pick data point, time index t and noise realization ϵ to predict ϵ_θ and minimize MSE-loss.

Rem. 1: Possible to perform diffusion in latent space for better performance and increased generality

Rem. 2: Noise sched. dictates smooth trans. between noise and output

GANS vs. Diffusion

- **Training Obj**: Diffusion: minimize reconstr. loss for noise pred. or data reconstr. GANs: Advers. setup
- **Training Setup:** Diffusion: Stable due to non-advers. setup (requires many training steps, hyperparam tuning), GANS: prone to instability
- **Output Quality:** Diversity may suffer in GANs due to mode collapse
- **Gen Process:** Iterative process starting from noise to output, GAN: Single forward pass to create sample
- **Comp cost:** Computationally expensive due to iterative sampling, GANs: Efficient during inference

12 Advanced Topics

12.1 Theory

p-Norm Robustness: For a multi-class classifier f, the goal of adversarial example perturbation is to find a perturbation η given a pattern x s.t. $f(x+\eta) \neq f(x)$ s.t. $\|\eta\|_p \leq \epsilon$

For $p = 2$ and a linear model with weight vectors w_i, we have $f(x) = \arg\max_i f_i(x)$, $f_i = w_i^T x + b_i$. For binary classification, the optimal perturbation is then given by $\eta \propto \text{sign}(f_1(x) - f_2(x))(w_2 - w_1)$. This generalizes to $m > 2$ classes ($f(x) = 1$ w.l.o.g.) by finding most easily confusable class: $\eta = \arg\min_{i>1} \frac{f_i(x)-f_i(x)}{\|w_1 - w_i\|_2^2}(w_i - w_1)$

Linearization: Suggested in DeepFool: leverage the linear case to *linearize* a nonlinear model and iteratively find an adversarial perturbation. For $m = 2$ one iterates solving the problem $\arg\min_{\Delta\eta} \|\Delta\eta\|_2$, s.t. $(\nabla f_1(x) - \nabla f_2(x))^\top \Delta\eta < f_1(x) - f_2(x)$, with $f_1(x) - f_2(x)$ being the margin. This can be generalized to p-norms.

Robust Training: Loss is extended to neighborhoods of training points: $\ell(f(x), y) \mapsto \max_{\eta:\|\eta\|_p \leq \epsilon} \ell(f(x+\eta), y)$ which yields a minimax problem, where adversary picks worst perturbation η an learner picks best parameter θ in response. One can solve inner loop maximization via *projected gradient ascent*. For $p = 2$: $\eta^{t+1} = \epsilon\Pi\Big[\eta^t + \alpha\nabla_x \ell(f(x+\eta^t), y)\Big]$, $\Pi[z] \equiv \frac{z}{\|z\|_2}$,

for $p = \infty$: $\eta^{t+1} = \epsilon\Pi\Big[\eta^t + \alpha\,\text{sign}(\nabla_x \ell(f(x+\eta^t), y))\Big]$, $\Pi[z] \equiv \frac{z}{\|z\|_\infty}$.

A practical method is Fast Gradient Sign Method (FGSM), which performs one iteration of $\|\cdot\|_\infty$-projected gradient descent: $\eta = \epsilon\,\text{sign}(\nabla_x \ell(f(x), y)$, modifying the pixels by $\pm\epsilon$ depending on the sign of $\frac{\partial \ell}{\partial x_i}$ per pixel. FGSM results in $\approx 2\times$ overhead compared to standard training.

12.2 Adversarial Attacks

Aspect	White-Box Attack	Black-Box Attack
Access to model	Full access (parameters, gradients)	Limited access (input-output only)
Gradient information	Direct access	Estimated or approximated
Attack methods	FGSM, PGD, C&W, etc.	Transferability, query-based, score-based
Computational cost	Lower (direct gradient usage)	Higher (requires queries or surrogate training)
Strength	Very strong	Relatively weaker, but still effective

Grey-box: partial knowledge of target model, which could include some knowledge such as model architecture, training data, setup and ouputs, but does not have access to the exact model parameters (weights and biases) or gradients.

Targeted vs Untargeted:

- **Untargeted**: aim to fool model such that output is anything different from the correct label
- **Targeted**: aim to fool model to return target label of the attacker's interest

Types of Attacks:

- **Gradient-based:** for white-box setting by solving either targeted or untargeted adversarial objective (e.g. FGSM and PGD)
- **Optimization-based:** achieve adversarial objective while minimize perturbation size (e.g. Carlini& Wagner attack)
- **Transferability-Based:** usually for black-box and grey-box; trains surrogate model $\implies$ generate adversarial examples based on the surrogate using optimization-based methods $\implies$ applied to target model
- **Query-based:** Usually for the black-box; if attacker can query the black-box model and observe its outputs (predictions or confidence scores), they can iteratively approximate gradients using optimization-based methods. Gradients can be estimated using *finite differences*: $\frac{\partial J}{\partial x_i} \approx \frac{J(x + \delta e_i) - J(x)}{\delta}$

Aspect	FGSM	PGD	C&W
Attack type	Single-step	Iterative (multi-step)	Optimization-based
Perturb. constraint	L_∞-bounded	L_∞-bounded (often L_∞)	Flexible (L_0, L_2, L_∞)
Objective	Maximize loss (fast)	Maximize loss iteratively	Minimize pertur.+ misclass.
Strength	Weak	Stronger than FGSM	Very strong
Comp. Cost	Low	Medium (multiple iters)	High (complex optim. Problem)
Use case	Quick evaluation	Strong adversarial training	Advanced adversarial attacks

FGSM: $x_{adv} = x + \epsilon \cdot \text{sign}(\nabla_x J(\theta, x, y))$, where x : original input, ϵ : perturbation magnitude (small scalar value), $\nabla_x J(\theta, x, y)$ gradient of loss; FGSM is a special case of PGD with a single step

PGD: $x_{adv}^{(t+1)} = \text{Proj}_{\mathcal{B}_\epsilon(x)}(x_{adv}^{(t)} + \alpha \cdot \text{sign}(\nabla_x J(\theta, x_{adv}^{(t)}, y)))$, where $\text{Proj}_{\mathcal{B}_\epsilon(x)}$ projects adversarial examples back into the ϵ-ball centered at x.

Carlini & Wagner: minimize $\|x_{adv} - x\|_p + c \cdot f(x_{adv})$, where $\|x_{adv} - x\|_p$ is perturbation size e.g. L_2 norm, c: regularization constant balancing perturbation size and attack success, $f(x_{adv})$: function that ensures x_{adv} is misclassified (often $f(x_{adv}) \leq 0$ if attack succeeds). Then use optimization algorithm such as SGD or Adam to search for optimal x_{adv}

12.3 Other Adversarial Attacks

- **Backdoor Attacks:** *training-time* attack where an adversary injects hidden "backdoor"patterns into the training dataset. The model behaves normally on clean inputs but produces targeted outputs when presented with inputs containing the backdoor trigger
- **Privacy Inference Attacks:** *test-time* attack where an adversary exploits the model's predictions to infer sensitive information about the training data e.g. *Membership Inference* (determining whether data point was used during model training), *Attribute Inference* (recovering specific attributes of the input data e.g. gender, age, private features)
- **Model Extraction Attacks:** adversary attempts to replicate or approximate the target model by querying it repeatedly and analyzing the outputs to reconstruct the model's behavior
- **Model Inversion Attacks:** *test-time* attack where adversary uses the model's output predictions to reconstruct the input features, potentially revealing private or sensitive data

12.4 Adversarial Defense

- **Denoising based methods:** attempt to use conventional image/signal processing techniques, GANs, autoencoders, or some generative approaches to erase the adversarial perturbations placed on the input samples
- **Randomization based methods:** based on random transformation or regularization with motivation that randomizing adversarial effects introduced in adversarial examples could potentially make DNNs more robust to them
- **Adversarial training based methods:** aim to improve robustness of DNNs by training them with adversarial examples. Motivation: adversarial examples can be considered scarce data residing near the true decision boundary. Including such samples in the training process will enable the models to learn more specific decision boundaries that take these outliers into consideration

Denoising-based easily bypassed by optim.-based approaches; randomization-based have been proven to be exploiting *obfuscated gradient* which gives a false sense of security and can be easily compromised by adv. examples generated through Expectation over Transformation (EoT).

Adversarial training has been the most robust defense method

13 Miscellaneous

13.1 Activation Functions and Important Derivatives

- $\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$; $\sigma(x) = \frac{1}{1+e^{-x}} = \frac{1+\tanh(x/2)}{2}$
- $\tanh'(x) = 1 - \tanh^2(x)$, $\sigma'(x) = \sigma(x)(1-\sigma(x))$
- $\text{GELU}(x) = x\Phi(\alpha x) \implies \text{GELU}'(x) = \frac{\alpha}{\sqrt{2\pi}}\exp(-\frac{(\alpha x)^2}{2}) + \Phi(\alpha x)$

13.2 Trigonometric Functions

- $\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$; $\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$
- $\tan(x) = \frac{\sin(x)}{\cos(x)}$; $\frac{\partial \tan(x)}{\partial x} = \frac{1}{\cos^2(x)} = \sec^2(x)$
- $\sinh(x) = \frac{e^x - e^{-x}}{2}$; $\frac{\partial \sinh(x)}{\partial x} = \cosh(x)$
- $\cosh(x) = \frac{e^x + e^{-x}}{2}$; $\frac{\partial \cosh(x)}{\partial x} = \sinh(x)$
- $\tanh(x) = \frac{\sinh(x)}{\cosh(x)}$; $1 = \sinh^2(x) + \cosh^2(x)$

13.3 Analysis

- $\frac{\partial \|x-b\|_2}{\partial x} = \frac{x-b}{\|x-b\|_2}$; $\frac{\partial |x-b|}{\partial x} = \frac{x-b}{|x-b|}$; $\frac{\partial b^\top x}{\partial x} = \frac{\partial x^\top b}{\partial x} = b$
- $\frac{\partial b^\top A x}{\partial x} = A^\top b$; $\frac{\partial x^\top x}{\partial x} = 2x$; $\frac{\partial x^\top A x}{\partial x} = (A^\top + A)x \overset{A \text{ symm.}}{=} 2Ax$
- $\frac{\partial \log|X|}{\partial X} = X^{-\top}$; $\frac{\partial \text{tr}(X^\top A)}{\partial X} = A$; $\frac{\partial \text{tr}(X^\top X)}{\partial X} = \frac{\partial \text{tr}(XX^\top)}{\partial X} = 2X$
- $\frac{\partial x^\top A x}{\partial A} = xx^\top$; $\frac{\partial \text{ReLU}(b^\top x)}{\partial x} = \mathbb{I}_{[b^\top x > 0]} b$; $\frac{\partial \text{softmax}(z_i)}{\partial z_j} = \text{softmax}(z_i)(\mathbb{I}_{[i=j]} - \text{softmax}(z_j)) = \text{softmax}(z_i)(\delta_{ij} - \text{softmax}(z_j))$
- $\frac{\partial A_1 B A_2}{\partial X} = (A_1 \otimes A_2^\top)\frac{\partial B}{\partial X}$; $(A_1 \otimes B_1)(A_2 \otimes B_2) = A_1 A_2 \otimes B_1 B_2$
- $f(x) = w^\top V x$, $\frac{\partial f(x)}{\partial V} = w^\top \otimes x^\top$; $\nabla_V f(x) = w \otimes x$
- $\frac{\partial x \odot y}{\partial x} = \text{diag}(y)$; $\frac{\partial x \odot y}{\partial y} = \text{diag}(x)$
- $\sum_{i=0}^{n} \binom{n}{i} = 2^n$

13.4 Proability Theory

- Markov: $\phi(\cdot)$ non-decr.& non-neg. $\rightsquigarrow \mathbb{P}(X \geq \epsilon) \leq \frac{\mathbb{E}[\phi(X)]}{\phi(\epsilon)}$
- Chebyshev: $\phi(\cdot)$ non-decr.& non-neg. $\rightsquigarrow \mathbb{P}(f(X) \geq \epsilon) \leq \frac{\mathbb{E}[\phi(f(X))]}{\phi(\epsilon)}$

Normal Distribution:

- $p(z) \propto \exp\left(-\frac{1}{2} z^\top Q z + z^\top m\right) \rightsquigarrow p(z) = \mathcal{N}(z; Q^{-1}m, Q^{-1})$
- $X_1 | X_2 = a \sim \mathcal{N}(\mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(a - \mu_2), \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{21})$

13.5 Linear algebra

Cauchy-Schwarz inequality: $|\langle x, y\rangle|^2 \leq \langle x, x\rangle\langle y, y\rangle$. With dot prod. as inner prod., and $\|x\| := \sqrt{\langle x, x\rangle}$, this means $|x \cdot y|^2 \leq \|x\|^2 \cdot \|y\|^2 \iff |x \cdot y| \leq \|x\| \cdot \|y\|$

Spectral norm definition & properties:

- $\|A\|_2 = \max_{x:\|x\|=1} \|Ax\| = \sigma_1(A)$, where σ_1 is A's largest singular value
- $\|AB\|_2 \leq \|A\|_2 \|B\|_2$. Also extends to higher order products

Woodbury identity:

$(I + UCV)^{-1} = I - U(C^{-1} + VU)^{-1}V$, C invertible

$(A + UCV)^{-1} = A^{-1} - A^{-1}U(C^{-1} + VA^{-1}U)^{-1}VA^{-1}$, A, C invertible

Pseudoinverse:

$W_{\text{left}}^+ = \lim_{\sigma^2 \to 0} W^\top (WW^\top + \sigma^2 I)^{-1}$ and $WW_{\text{left}}^+ = I$

$W_{\text{right}} = \lim_{\sigma^2 \to 0} (W^\top W + \sigma^2 I)^{-1} W^\top$ and $W_{\text{right}}^+ W = I$

Note that the identities depending on left or right inverse will have different dimensions!!

Other Matrix identities:

$(I + AB)^{-1} A = A(I + BA)^{-1}$ and $(I + AB)^{-1} = I - A(I + BA)^{-1}B$

13.6 Group Theory

A group is a non-empty set G with a binary operation on G, denoted "$\cdot$", such that the following group axioms are satisfied:

- **(Closure)**: $\forall a, b \in G : a \cdot b \in G$
- **(Associativity)**: $\forall a, b, c \in G : (a \cdot b) \cdot c = a \cdot (b \cdot c)$
- **(Identity Element)**: $\exists e \in G \quad \forall a \in G : e \cdot a = a \quad \text{and} \quad a \cdot e = a$. The element e is called the *identity element*
- **(Inverse Element)**: $\forall a \in G \quad \exists b \in G : a \cdot b = e \quad \text{and} \quad b \cdot a = e$, where e is the identity element. Note that for all a, the **inv. element** b is unique.

The group is called **Abelian** if on top of the group axioms, commutativity holds as well, i.e., $\forall a, b \in G : a \cdot b = b \cdot a$

14 Coding

Series 1

```
# np.sign maps 0 to 0 and is therefore incorrect
y_hat=2*np.int32(np.inner(X_train[idx, :],theta)>=0)-1
# perform Perceptron update only if wrongly classified
if y_hat  * y_train[idx] < 0:
    theta += self.lr * y_train[idx] * X_train[idx, :]
# compute accuracy
np.mean(np.sign(np.inner(X_train, theta)) == y_train)
```

Series 6

```
class PositionalEncoding(nn.Module):
    def __init__(self,d_model,dropout,max_len=5000):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        position = torch.arange(max_len).unsqueeze(1)
        div_term=torch.exp(torch.arange(0,d_model,2)*
                (-math.log(10000.0) / d_model))
        pe = torch.zeros(max_len, 1, d_model)
        pe[:, 0, 0::2]=torch.sin(position * div_term)
        pe[:, 0, 1::2]=torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x: Tensor) -> Tensor:
        """x:[seq_len, batch_size, embedding_dim]"""
        x = x + self.pe[:x.size(0)]
        return self.dropout(x)

class SelfAttention(nn.Module):
    def __init__(self, d: int, heads: int=8):
        super().__init__()
        self.k, self.h = d, heads

        self.Wq = nn.Linear(d, d * heads, bias=False)
        self.Wk = nn.Linear(d, d * heads, bias=False)
        self.Wv = nn.Linear(d, d * heads, bias=False)

        self.unifyheads = nn.Linear(heads * d, d)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        b, l, d = x.size()
        h = self.h

        q=self.Wq(x).view(b, l, h, d).transpose(1, 2)
            .contiguous().view(b * h, l, d)
        k=self.Wk(x).view(b, l, h, d).transpose(1, 2)
            .contiguous().view(b * h, l, d)
        v=self.Wv(x).view(b, l, h, d).transpose(1, 2)
            .contiguous().view(b * h, l, d)
        # (b*h, l, l)
        w_prime = torch.bmm(q, k.transpose(1, 2)) \
                / np.sqrt(d)
        w = F.softmax(w_prime, dim=-1)
        out = torch.bmm(w, v).view(b, h, l, d)
        out = out.transpose(1, 2).contiguous()
        return self.unifyheads(out.view(b, l, h * d))

class TransformerBlock(nn.Module):
    def __init__(self, d, heads= 8, n_mlp = 4):
        super().__init__()
        self.attention = SelfAttention(d, heads)

        self.norm1 = nn.LayerNorm(d)
        self.norm2 = nn.LayerNorm(d)

        self.ff = nn.Sequential(
            nn.Linear(d, n_mlp * d), nn.ReLU(),
            nn.Linear(n_mlp * d, d)
        )
### Transformer forward: ###
self.token_emb = nn.Embedding(num_tokens, embed_dim)
self.pos_emb = nn.Embedding(max_seq_len, embed_dim)
token_embs = self.token_emb(x)
pos_embs = self.pos_emb(torch.arange(l).to(DEVICE))
    .expand(b, l, d)
final_embs = token_embs + pos_embs
out = self.transformer_blocks(final_embs)
# Compute the mean latent vector for each sequence.
# The mean is applied over dim=1 (time). Shape: [b, d]
out = out.mean(dim=1)
# Classify. Shape: [b, num_classes].
out = self.classification(out)

### RNN forward: ###
token_embs = self.token_emb(x)
# Feed the embedding into the GRU. Shape: [b, l, d]
# Use output of the last token as the encoding.
out, final_state = self.rnn(token_embs)
out = out[:, -1]
out = self.classification(out)
```

Series 11

```
class GAN(nn.Module):
    def __init__():
        super().__init__()
        # Generator
        self.generator = nn.Sequential(
            nn.ConvTranspose2d(in_channels=latent_dim,
                out_channels=num_maps_gen*8,
                kernel_size=4, stride=1, padding=0,
                bias=False),
            nn.Batchnorm2d(num_maps_gen * 8),
            nn.ReLU(True),
            nn.ConvTranspose2d(
                in_channels = num_maps_gen*8,
                out_channels = num_maps_gen*4,
                kernels_size=4, stride = 2,
                padding = 1, bias = False),
            nn.Batchnorm2d(num_maps_gen*4),
            nn.ReLU(True),
            [...]
            nn.ConvTranspose2d(
                in_channels = num_maps_gen,
                out_channels = 1,
                kernels_size=4, stride = 2,
                padding = 1, bias = False),
            nn.Tanh()
        )
        # Discriminator
        self.discriminator = nn.Sequential(
        nn.Conv2d(in_channels = 1,
            out_channels = num_maps_gen,
                kernels_size = 4, stride = 2,
                padding= 1, bias = False),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Conv2d(
                in_channels = num_maps_gen,
                out_channels = num_maps_gen*2,
                kernel_size= 4, stride = 2,
                padding = 1, bias = False),
            nn.BatchNorm2d(num_maps_gen*2),
            nn.LeakyReLU(0.2,inplace=True),
            [...]
            nn.Conv2d(
                in_channels = num_maps_gen*8,
                out_channels = 1,
                kernels_size = 3, stride = 1,
                padding= 0, bias = False),
            nn.Sigmoid()
        )
#GAN forward
    def gen_forward(self, z):
        img = self.generator(z)
        return self.generator(z)
    def disc_forward(self, img):
        pred = self.discriminator(img)
        return pred.view(-1)
gen_optim = torch.optim.Adam( #Adam Gen
    params=model.generator.parameters(),
    lr=generator_lr, betas=(0.5, 0.999))
disc_optim = torch.optim.Adam( #Adam Disc
    params=model.discriminator.parameters(),
    lr=discriminator_lr, betas=(0.5, 0.999))
disc_loss_func = nn.BCELoss()
gen_loss_func = nn.BCELoss()
#GAN Training
for epoch in range(num_epochs):
    model = model.train()
    for batch_idx, (real_data, targets)
    in enumerate(train_loader):
      batch_size = targets.size(0)
      real_data = real_data.to(device)
      targets = targets.to(device)
      # CREATING GROUND-TRUTH
      y_real=torch.ones(batch_size).float().to(d)
      y_fake=torch.zeros(batch_size).float().to(d)
      # TRAIN DISCRIMINATOR
      disc_optim.zero_grad()
      # train the disc.to classify real images
      disc_pred_real=model.disc_forward(real_data)
        .view(-1)
      real_loss = disc_loss_func(disc_pred_real,
        y_real)
      # gen imgs frm smpls of the ltnt prior
      z = torch.randn(batch_size, latent_dim, 1, 1,
        device=device )
      fake_data = model.gen_forward(z)
      # train the disc. to classify fake images
      # detach the comp. graph of the generator
      # s.t. grads are not backprop into the gen)
      disc_pred_fake = model.disc_forward(
        fake_data.detach()).view(-1)
      fake_loss = disc_loss_func(disc_pred_fake,
        y_fake)
      disc_loss = 0.5*(real_loss + fake_loss)# avg
      disc_loss.backward()
      disc_optim.step()
      # TRAIN GENERATOR
      gen_optim.zero_grad()
      # train gen s.t. image is class. as real
      disc_pred_fake = model.disc_forward(
        fake_data).view(-1)
      gen_loss = gen_loss_func(disc_pred_fake, y_real)
      gen_loss.backward()
      gen_optim.step()
      [...]
```

Series 12

```
class ForwardProcess: # FIND X_t and noise
    def __init__(self, betas: torch.Tensor):
        self.beta = betas
        self.alphas = 1. - betas
        self.alpha_bar=torch.cumprod(self.alphas,
        dim=-1)

    def get_x_t(self, x_0: torch.Tensor, t:
    torch.LongTensor) -> Tuple[torch.Tensor, torch.Tensc
        eps_0 = torch.randn_like(x_0).to(x_0)
        alpha_bar = self.alpha_bar[t-1, None]
        mean = (alpha_bar ** 0.5) * x_0
        std = ((1. - alpha\_bar) ** 0.5)
        return (eps_0, mean + std * eps_0)

class NoiseNetwork((nn.Module)):
    def __init__(self, T):
    super().__init__()
    self.T = T
    self.t_encoder = nn.Linear(T,1)
    self.model = nn.Sequential (
        nn.Linear(2 + 1, 100),
        nn.LeakyReLU(inplace=True),
        nn.Linear(100, 2),
    )

def forward(self, x_t, t):
    t_embedding = self.t_encoder(
        F.one_hot(t - 1, num_classes=self.T
            ).to(torch.float)
    )
    input = torch.cat([x_t, t_embedding], dim=1)
    return self.model(input)
T = 100
beta_start = 0.004
beta_end = 0.02
betas = torch.linspace(beta_start,beta_end,T)
#Training
fp = ForwardProcess(betas)
model = NoiseNetwork(T=T)
optimizer = torch.optim.AdamW(params=model.parameters(),
lr=1e-2, betas=(0.9, 0.999), weight_decay=1e-4)

N = X.shape[0]
for epoch in trange(5000):
    with torch.no_grad():
        t = torch.randint(low=1, high=T, size=(N,))
        eps_0, x_t = fp.get_x_t(X, t=t)
    pred_eps = model(x_t, t)
    loss = torch.nn.functional.mse_loss(pred_eps,eps_0)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
#Define reverse process
class ReverseProcess(ForwardProcess):
    def __init__(self, betas: torch.Tensor,
    model: nn.Module):
        super().__init__(betas)
        self.model = model
        self.T = len(betas)

        self.sigma = (
        betas * (1 - torch.roll(self.alpha_bar, 1))
        / (1 - self.alpha_bar)) ** 0.5
        self.sigma[0] = 0
    def get_x_t_minus_one(self, x_t: torch.Tensor,
    t: int) -> torch.Tensor:
        with torch.no_grad():
            t_vector = torch.full(size=(x_t.shape[0],),
            fill_value=t, dtype=torch.long)
            eps = self.model(x_t, t=t_vector)
        eps *= (1 - self.alphas[t-1])
            / ((1 - self.alpha_bar[t-1]) ** 0.5)
        mean = 1 / (self.alphas[t-1] ** 0.5)
            * (x_t - eps)
        return mean + self.sigma[t-1]
            * torch.randn_like(x_t)

    def sample(self, n_samples=1, full_trajectory=False
        x_t = torch.randn(n_samples, 2)
        trajectory = [x_t.clone()]
        for t in range(self.T, 0, -1):
            x_t = self.get_x_t_minus_one(x_t, t=t)
            if full_trajectory:
                trajectory.append(x_t.clone())
        return torch.stack(trajectory, dim=0) if
        full_trajectory else x_t
```

Series 13

```
def fgsm_(model, x, target, eps, targeted=True,
    clip_min=None, clip_max=None):
    # copy of the input tensor
    input_ = x.clone().detach_()
    # we have to be able to differentiate
    input_.requires_grad_()

    logits = model(input_)
    target = torch.LongTensor([target])
    model.zero_grad()
    loss = nn.CrossEntropyLoss()(logits, target)
    loss.backward()
    #perfrom either targeted or untargeted attack
    if targeted:
        out = input_ - eps * input_.grad.sign()
    else:
        out = input_ + eps * input_.grad.sign()
    if (clip_min is not None)or(clip_max is not None):
        out.clamp_(min=clip_min, max=clip_max)
    return out

def fgsm_targeted(model, x, target, eps,**kwargs):
    return fgsm_(model, x, target, eps, targeted=True,
    **kwargs)
def fgsm_untargeted(model, x, label, eps, **kwargs):
    return fgsm_(model, x, label, eps, targeted=False,
    **kwargs)

# x: input image
# label: current label of x
# k: number of FGSM iterations
# eps: size of l-infinity ball
# eps_step: step size of FGSM iterations
# Each FGSM iteration is projected back to the
#eps-sized l_inf ball around x
def pgd(model, x, label, k, eps, eps_step, targeted,
    clip_min, clip_max):
    x_min = x - eps
    x_max = x + eps

    # Randomize the starting point x.
    x_adv = x + eps * (2 * torch.rand_like(x) - 1)
    # Clamp back
    if (clip_min is not None)or(clip_max is not None):
        x_adv.clamp_(min=clip_min, max=clip_max)
    for i in range(k):
        # FGSM step
        # We don't clamp here (clip_min=clip_max=None)
        # as we want to apply the attack as defined
        x_adv = fgsm_(model, x_adv, label, eps_step,
            targeted)
        # Projection Step
        x_adv =torch.min(x_max,torch.max(x_min, x_adv))
    #if desired: clip the ouput back to image domain
    if (clip_min is not None)or(clip_max is not None):
        x_adv.clamp_(min=clip_min, max=clip_max)
    return x_adv
```