Classical ML

- **Training Paradigms:** *Supervised learning*, with labelled data pairs $(\mathbf{x}, y)$, *Unsupervised learning* with unlabeled data $\mathbf{x}$, *semi-supervised learning*, with some labelled and some unlabeled data. Often models predict $f : \mathcal{X} \to \mathcal{Y}$ such that $f(\mathbf{x}_i) \approx y_i$, when y_i is continuous this is *regression* and when y_i is discrete this is *classification*. Datasets are often represented in matrix form, where a column is a data point and a row is a feature. E.g., $X \in \mathbb{R}^{n\times d}$ and $Y \in \mathbb{R}^{n\times t}$ (often $t = 1$). Note that for finite datasets, there exist infinitely many functions for which $f(\mathbf{x}_i) = y_i$ for all i, so we want to leverage prior information and select simple f's. The optimal f is $f^*(\mathbf{x}) = \mathbb{E}[Y|X = \mathbf{x}]$.
- **Bias Variance Decomposition:**

$$\underbrace{\mathbb{E}_{D,X,Y}\|f_D(X) - Y\|_2^2}_{\text{test error}} = \underbrace{\mathbb{E}_X\|\mathbb{E}_D[f_D(X)] - f^*(X)\|_2^2}_{\text{bias}^2} + \underbrace{\mathbb{E}_{D,X}\|f_D(X) - \mathbb{E}_D[f_D(X)]\|_2^2}_{\text{variance}} + \underbrace{\mathbb{E}_{X,Y}\|f^*(X) - Y\|_2^2}_{\text{hardness/noise}}$$

- **Linear Function:** A function is linear if $f(\alpha\mathbf{x} + \beta\mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y})$ or equivalently if it can be written as $f(\mathbf{x}) = \langle \mathbf{x}, \mathbf{w}\rangle$. If $f(\mathbf{x}) = \langle \mathbf{x}, \mathbf{w}\rangle + b$ then f is said to be *affine*. Note for an affine function, $\mathbf{w}$ is orthogonal to the decision boundary $H = \{\mathbf{x} : \langle \mathbf{x}, \mathbf{w}\rangle + b = 0\}$.
- **Perceptron Algorithm:** The goal is to find $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$ such that $y_i\hat{y}_i = y_i(\langle \mathbf{x}, \mathbf{w}\rangle + b) > 0$ for all i (i.e., no mistakes). When the model makes a mistake, we update it by $\mathbf{w} \leftarrow \mathbf{w} + y\mathbf{x}$ and $b \leftarrow b + y$, this always increases the confidence $y\hat{y}$; repeat until convergence. Note that perceptrons don't have unique solutions and will cycle infinitely if there is no perfectly separating hyperplane. We can extend to a multiclass case by either training C one vs all perceptron, or $\binom{C}{2}$ one vs one perceptron according to which classes are negative.
- **Perceptron Convergence:** Suppose $\exists \mathbf{w}^*$ such that $\forall i,\ y_i\langle \mathbf{x}_i, \mathbf{w}^*\rangle > 0$. Let C be such that $\|\mathbf{x}_i\|_2 \le C$ for all i and let $\mathbf{w}^*$ be such that $\|\mathbf{w}^*\|_2 = 1$. Let $\gamma = \min_i |\langle \mathbf{x}_i, \mathbf{w}^*\rangle|$ be the margin of error. Then the Perceptron Algorithm will converge after C^2/γ^2 iterations. The idea is to show that after updating $\mathbf{w}$, $\langle \mathbf{w}, \mathbf{w}^*\rangle$ increases by $\ge \gamma$ and $\langle \mathbf{w}, \mathbf{w}\rangle$ increases by $\le C^2$ and consider $\cos(\mathbf{w}, \mathbf{w}^*)$ after converging.
- **Perceptron Loss:** For a linear classifier $\hat{y} = \text{sign}(\langle \mathbf{x}, \mathbf{w}\rangle)$ (can use bias hiding trick), we minimize the perceptron loss $l(\mathbf{w}, \mathbf{x}_i, y_i) = -y_i\langle \mathbf{x}_i, \mathbf{w}\rangle \mathbb{I}[\text{mistake on } \mathbf{x}_i]$. Learning with SGD, we get $\mathbf{w}_{i+1} = \mathbf{w}_i - \eta_i \nabla_{\mathbf{w}} l(\mathbf{w}_i, \mathbf{x}_i, y_i) = \mathbf{w}_i + \eta_i y_i \mathbf{x}_i \mathbb{I}[\text{mistake on } \mathbf{x}_i]$, which is the same as the perceptron algorithm when $\eta = 1$.
- **Linear Regression:** Consider a 1-padded dataset $X \in \mathbb{R}^{n\times(d+1)}$. Then our class of functions is $f(x) = Wx$ for $W \in \mathbb{R}^{(d+1)\times t}$ and we minimize the least squares loss $L(W) = \hat{\mathbb{E}}\|f(x_i) - y_i\|_2^2 = \frac{1}{n}\|XW - Y\|_F^2$ where $\|A\|_F^2 = \sum_{ij} A_{ij}^2$. Using the *optimality condition* and setting the gradient w.r.t. W to 0, we get $\nabla_W L(W) = \frac{2}{n}X^T(XW - Y) = 0$ to get $W = (X^TX)^{-1}X^TY$. Note when X is (close to) rank-deficient two rows are almost linearly dependent but may have different y's which leads to unstable W.
- **Ridge Regression:** We add a regularization term to the loss: $L(W) = \hat{\mathbb{E}}[\|XW - Y\|_F^2 + \lambda\|W\|_F^2]$ which yields a solution $W = (X^TX + n\lambda I)^{-1}X^TY$. The matrix $X^TX + n\lambda I$ is far from rank-deficient for large λ, which makes the solution more stable. Note, $\lambda = 0$ reduces to ordinary linear regression and $\lambda \to \infty$ reduces to $W \to 0$. Ridge regression is equivalent to augmenting our dataset with points $\mathbf{x}_j = \sqrt{n\lambda}\mathbf{e}_j$ for $j = 1, \dots, d$ and $y_j = 0$.
- **Logistic Regression:** For classification data $\mathcal{Y} = \{0, 1\}$, we directly model the probability $f(\mathbf{x}; \mathbf{w}) = P(Y = 1|X = \mathbf{x})$ by modeling the log-odds with linear regression, i.e., $\log\frac{P(Y=1|X=\mathbf{x})}{P(Y=0|X=\mathbf{x})} = \langle \mathbf{x}, \mathbf{w}\rangle$. In particular, for $\sigma(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}$ the sigmoid function, we let $f(\mathbf{x}; \mathbf{w}) = \sigma(\langle \mathbf{x}, \mathbf{w}\rangle)$. We must learn $\mathbf{w}$ with MLE by minimizing the NLL $\min_{\mathbf{w}} \sum_{i=1}^n \big[-y_i \log f(\mathbf{x}_i; \mathbf{w}) - (1 - y_i)\log(1 - f(\mathbf{x}_i; \mathbf{w}))\big]$ using SGD. Note $\hat{y} = \arg\max_k P(Y = k|X = \mathbf{x})$ so that $\hat{y} = 1$ iff $f(\mathbf{x}; \mathbf{w}) > \frac{1}{2}$, which gives us a decision boundary $H = \{\mathbf{x} : \langle \mathbf{x}, \mathbf{w}\rangle = 0\}$. A multiclass extension exists by learning separate $\mathbf{w}_1, \dots, \mathbf{w}_C$ and using the *softmax function* $\bar{\sigma}_k(a_1, \dots, a_C) = \frac{\exp(a_k)}{\sum_{\ell=1}^C \exp(a_\ell)}$ for $k = 1, \dots, C$. Then $P(Y = k|X = \mathbf{x}; W = [\mathbf{w}_1, \dots, \mathbf{w}_C]) = \bar{\sigma}_k(\langle \mathbf{x}, \mathbf{w}_1\rangle, \dots, \langle \mathbf{x}, \mathbf{w}_C\rangle) = \frac{\exp(\langle \mathbf{x}, \mathbf{w}_k\rangle)}{\sum_{\ell=1}^C \exp(\langle \mathbf{x}, \mathbf{w}_\ell\rangle)}$, we still use $\hat{y} = \arg\max_k P(Y = k|X = \mathbf{x}; W)$.

- **Hard-Margin SVM:** The aim is to maximize the *margin* of a perceptron, where the margin is the smallest distance from a point to the decision boundary, $\gamma = \min_i \frac{y_i \hat{y}_i}{\|\mathbf{w}\|_2}$. I.e., want $\max_{\mathbf{w},b} \gamma$. By scaling, we set the numerator to 1, leaving us the optimization problem $\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|_2^2$ such that $\forall i,\ y_i\hat{y}_i \geq 1$ (note perceptron satisfies this but doesn't minimize $\frac{1}{2}\|\mathbf{w}\|_2^2$). The Lagrangian dual problem is $\min_{\mathbf{w},b} \max_{\alpha \geq 0} \frac{1}{2}\|\mathbf{w}\|_2^2 - \sum_i \alpha_i[y_i(\langle \mathbf{x}_i, \mathbf{w}\rangle + b) - 1]$ (since $\alpha_i = \infty$ whenever $y_i\hat{y}_i < 1$). Setting the derivative w.r.t. $\mathbf{w}$ and b to 0 and swapping max to min we get $\min_{\alpha \geq 0} -\sum_i \alpha_i + \frac{1}{2}\|\sum_i \alpha_i y_i \mathbf{x}_i\|_2^2 = \min_{\alpha \geq 0} -\sum_i \alpha_i + \frac{1}{2}\sum_{ij} \alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j\rangle$ such that $\sum_i \alpha_i y_i = 0$. *Support vectors* (points on the boundary of the margin: $H_\pm = \{\mathbf{x} : \langle \mathbf{x}, \mathbf{w}\rangle + b = \pm 1\}$) have $\alpha_i \in \mathbb{R}^+$ and non-support vectors have $\alpha_i = 0$. We prefer the dual form since we can apply kernel tricks to $\langle \mathbf{x}_i, \mathbf{x}_j\rangle$.
- **Soft-Margin SVM:** The aim to allow an SVM to make mistakes by using a hinge loss $\ell_{hinge}(y\hat{y}) = (1 - y\hat{y})^+$ which is 0 when $y\hat{y} \geq 1$. We now solve $\min_{\mathbf{w},b} \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_i (1 - y_i\hat{y}_i)^+$. Optimizing the Lagrangian dual problem yields $\min_{0 \leq \alpha \leq C} -\sum_i \alpha_i + \frac{1}{2}\alpha_i \alpha_j y_i y_j \langle \mathbf{x}_i, \mathbf{x}_j\rangle$ such that $\sum_i \alpha_i y_i = 0$. We see that hard-margin SVM is simply the case where $C = \infty$. Correct points have $\alpha_i = 0$, incorrect points have $\alpha_i = C$, and support vectors (on the boundary) have $0 \leq \alpha_i \leq C$. We can recover $\mathbf{w}^* = \sum_i \alpha_i^* y_i \mathbf{x}_i$ and b by using a point on $H_{\pm 1}$.
- **On the Hinge Loss:** The optimal SVM classifier minimizes ℓ_{0-1}, but ℓ_{hinge} and ℓ_{0-1} are *classification-calibrated* since their optimal classifiers always have the same sign. So minimizing the hinge loss maximizes $P(Y = \text{sign}(\hat{Y}))$ and minimizes the expected ℓ_{0-1} loss.
- **Kernel Trick:** Data may not be linearly separable, but it might be in a higher dimension, e.g., in $\phi(\mathbf{x}) = (\mathbf{x}\mathbf{x}^T, \sqrt{2}\mathbf{x}, 1)$. This blows up the dimension, but we only need $\langle \phi(\mathbf{x}), \phi(\mathbf{z})\rangle = (\langle \mathbf{x}, \mathbf{z}\rangle + 1)^2$ for the dual form. $k : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *(reproducing) kernel* iff there is a $\phi : \mathcal{X} \to \mathcal{H}$ such that $\langle \phi(\mathbf{x}), \phi(\mathbf{z})\rangle = k(\mathbf{x}, \mathbf{z})$ (note ϕ may not be unique) iff the kernel matrix $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ is symmetric and positive semidefinite for any $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathcal{X}$. At inference time, compute $f(\mathbf{x}) = \langle \phi(\mathbf{x}), \mathbf{w}^*\rangle = \sum_{i=1}^n \alpha_i^* y_i k(\mathbf{x}, \mathbf{x}_i)$.
- **Gradient Descent:** To solve $\min_x f(x)$ choose initial $x^{(0)} \in \mathbb{R}^d$ and repeat $x^{(k)} = x^{(k-1)} - \eta \nabla_x f(x^{(k-1)})$ until convergence (η is the step size) since the gradient points in the direction of steepest ascent. We can derive gradient descent as a local minimization of the Taylor expansion of $f(x)$ at $x^{(k-1)}$, $\min_y f(y) \approx \min_y [f(x) + \nabla f(x)^T(y - x) + \frac{1}{2\eta}\|y - x\|_2^2]$, which is minimized by $y = x - \eta \nabla f(x)$.
- **Gradient Descent for Convex Functions:** When f is convex (i.e., $f(y) \geq f(x) + \nabla f(x)^T(y - x)$, so $f(y)$ is always above the tangent line at x) and L-smooth (i.e., $LI - \nabla^2 f(x)$ is positive semi-definite), then gradient descent has $f(x^{(k)}) - f(x^*) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2\eta k}$ for $\eta \leq \frac{1}{L}$. This means we achieve $f(x) - f(x^*) \leq \varepsilon$ in $O(1/\varepsilon)$ iterations. The proof follows by using a quadratic expansion and L-smooth to show $f(x^+) \leq f(x) - \frac{1}{2}\eta\|\nabla f(x)\|_2^2$ (using $\eta \leq \frac{1}{L}$). Then using convexity we can show $f(x^+) - f(x^*) \leq \frac{1}{2\eta}(\|x - x^*\|_2^2 - \|x^+ - x^*\|_2^2)$. Summing over iterations, we get a telescoping sum to prove the result. When f is m-strong convex (i.e., $f(x) - m\|x\|_2^2$ is convex) and L-smooth, we get $f(x^{(k)}) - f(x^*) \leq \gamma^k \frac{L}{2}\|x^{(0)} - x^*\|_2^2$ for $\eta \leq \frac{2}{m+L}$ and some $\gamma \in (0, 1)$. This means we achieve $f^{(x)} - f(x^*) \leq \varepsilon$ exponentially fast, in $O(\log_{1/\gamma}(1/\varepsilon))$ iterations. When f is nonconvex, but differentiable and L-smooth, we find stationary points at a rate of $O(1/\sqrt{k})$, which cannot be improved by deterministic algorithms.
- **Stochastic Gradient Descent** approximates $\nabla f(x)$ by taking the mean over a subset of the data. Each step of gradient descent takes $O(\frac{n}{\varepsilon})$ time whereas each SGD step takes $O(\frac{1}{\varepsilon^2})$ time, but SGD needs more steps.

Neural Networks

- **Multi-Layer Perceptron:** Linear transformation followed by non-linear activation, repeated over multiple layers. E.g., a 2 layer MLP can be $\mathbf{z} = U\mathbf{x} + c$, $\mathbf{h} = \sigma(\mathbf{z})$, $\hat{y} = W\mathbf{h} + b$ where we learn U, W, c, b. We train by using a loss ℓ to measure the difference between $\hat{y}$ and y and use SGD to minimize the loss.
- **Back-Propagation:** To compute the gradients of our network for SGD, we need to use the chain rule to propagate the gradient from one layer to the previous layer. This involves matrix calculus; it's often easiest

to try and guess a solution based on scalar calculus. Modern frameworks compute this using automatic differentiation by constructing computational graphs of the forward process.

- **Universal Approximation Theorem:** For any continuous function $f : \mathbb{R}^d \to \mathbb{R}^c$ and any $\varepsilon > 0$, there exists a $k \in \mathbb{N}$ and $W \in \mathbb{R}^{k\times d}$, $b \in \mathbb{R}^k$, and $U \in \mathbb{R}^{c\times k}$ so that $\sup_x \|f(\mathbf{x}) - g(\mathbf{x})\|_2 < \varepsilon$ where $g(\mathbf{x}) = U(\sigma(W\mathbf{x}+b))$. However, there are functions where a 2-layer MLP needs exponentially many nodes but a 3-layer MLP only needs polynomially many. Deep neural networks are more parameter efficient.
- **Dropout:** For each training minibatch, use a different and random network using only a fraction q of all nodes. This way nodes are less likely to collude to overfit. At inference time, use the full network and multiply each node by $\frac{1}{q}$ to have the same expected magnitude.
- **Normalization:** In *batch normalization* and *layer normalization* we normalize input features (subtract by mean and divide by variance). In batch norm, the mean and variance are computed across features (nodes) over the whole batch whereas in layer norm they're computed across each input of the batch. In both cases, we apply an affine transformation afterward and at inference time use a learned running mean/variance.
- **Convolutional Neural Networks:** Instead of applying an MLP to all the pixels of an image, CNNs learn a convolution kernel and apply it locally to each patch of the image. By stacking multiple kernels rich and locally informed features are learned. Kernels can also be applied to padded images and strided to change the output size. CNNs can also be viewed as an MLP with weight sharing by reshaping the kernel as a circulant matrix. CNNs also commonly use [max, average] pooling to reduce feature map sizes. Inception networks use multiple convolutions of different sizes and ResNets introduce skip connections to improve gradient flow to early layers.
- **Transformers:** Inputs are first tokenized and then embedded to get a sequence of input feature vectors, often a positional embedding is added to inform where in the sequence each token is from (since attention is permutation equivariant). Attention takes in values Q, K, V (in self-attention $Q = K = V$, in cross-attention only $K = V$) and returns a convex combination of rows of V for each row of Q, computed as $\bar{\sigma}(\frac{QK^T}{\sqrt{d}})V$. In multi-head attention, we apply multiple linear layers W^q, W^k, W^v to the input and perform attention on each of the transformed inputs, concatenating their outputs at the end. Inputs can also be optionally masked, e.g., for training auto-regressive language modelling. A transformer encoder block is made up of a self-attention layer, followed by a 2-layer MLP with a $4\times$ hidden dimension. In a transformer decoder block, we instead have masked self-attention, followed by cross-attention (keys and values from the last output of the encoder), and finally an MLP as usual. For both encoders and decoders, a layer norm and skip connection are applied after each stage. In language modelling, the outputs of the decoder are passed to a linear layer and softmax to predict words and trained to minimize the NLL $\hat{\mathbb{E}} - \langle Y, \log \hat{Y}\rangle$ where Y is the ground-truth output as a one-hot vector.

Modern Machine Learning Paradigms

- **Large Language Models:** Large pre-trained models, BERT is an encoder-only architecture trained to predict a randomly sampled masked word (and binary classification next sentence prediction task) and GPT is a decoder-only architecture trained to predict the next word. These architectures are meant to be fine-tuned on specific tasks afterwards, such as classification, entailment (text logically follows), similarity, question answering, etc. Later, RoBERTa is trained for longer on more data with longer sequences and bigger batches. Sentence-Transformer uses a Siamese network to compute sentence similarity (which is better than the previous training of passing sentence pairs to BERT). GPT-2 is $10\times$ larger than GPT and trained on a huge dataset, it is comparable to BERT in performance but is good at zero-shot learning. GPT-3 is $100\times$ larger than GPT-2 and starts showing chain-of-thought and in-context learning. GPT-3.5 then trains GPT-3 using RLHF, by first fine-tuning on desired human outputs, then training a reward model to predict human preference on various model outputs, and finally this reward model is used to supervise the language model.
- **Fenchel Conjuugate:** The conjugate of $f : \mathbb{R} \to \mathbb{R}$ is $f^*(s) = \max_t st - f(t)$ and is convex. If f is convex and continuous then $(f^*)^* = f$.

- **Generative Adversarial Networks:** Consider training data $\mathbf{x}_1, \dots, \mathbf{x}_n \sim q(\mathbf{x})$, we wish to find a model to learn the data density q by $\min_\theta \mathsf{KL}(q\|p_\theta) = \int -q(\mathbf{x}) \log \frac{q(\mathbf{x})}{p_\theta(\mathbf{x})} d\mathbf{x} \approx -\frac{1}{n}\sum_{i=1}^n \log \frac{q(\mathbf{x}_i)}{p_\theta(\mathbf{x}_i)}$. But it's difficult to model p_θ as an explicit p.d.f., instead we learn a push-forward map from standard Gaussian noise to our data density, i.e., $T_\theta(Z) \sim p_\theta$ and minimize the divergence to our data. We also don't know $q(\mathbf{x})$ explicitly, so we will find a formulation to remove it. Consider $f(t) = t\log t - t$, take the conjugate and solve the maximum to get $f^*(s) = \exp(s)$, and so $f(t) = f^{**}(t) = \max_s st - \exp(s)$. Notice then $\mathsf{KL}(q\|p_\theta) = \int \frac{q}{p_\theta}\big(\log \frac{q}{p_\theta} - 1\big)p_\theta d\mathbf{x} = \int f(\frac{q(\mathbf{x})}{p_\theta(\mathbf{x})})p_\theta(\mathbf{x})d\mathbf{x} = \int \big(\max_s s\frac{q(\mathbf{x})}{p_\theta(\mathbf{x})} - \exp(s)\big)p_\theta(\mathbf{x})d\mathbf{x} = \max_{S:\mathbb{R}^d\to\mathbb{R}} \int \big(S(\mathbf{x})q(\mathbf{x}) - \exp(S(\mathbf{x}))p_\theta(\mathbf{x})\big)d\mathbf{x} = \mathbb{E}_{\mathbf{x}\sim q}[S(\mathbf{x})] - \mathbb{E}_{\mathbf{x}\sim p_\theta}[\exp(S(\mathbf{x}))]$. Then, parameterizing S as a neural net, we get our objective $\min_\theta \mathsf{KL}(q\|p_\theta) \approx \min_\theta \max_\phi \frac{1}{n}\sum_{i=1}^n S_\phi(\mathbf{x}_i) - \frac{1}{m}\sum_{j=1}^m \exp(S_\phi(T_\theta(\mathbf{z}_j)))$. T_θ is our generator from Gaussian noise to data and S_ϕ is our discriminator distinguishing real and generated data. In practice, we use the JS divergence $\mathrm{JS}(q\|p_\theta) = \mathsf{KL}(q\|\frac{q+p_\theta}{2}) + \mathsf{KL}(p_\theta\|\frac{q+p_\theta}{2})$ to avoid issues where $p_\theta(\mathbf{x}) \approx 0$. Through a similar derivation, we get the objective $\min_\theta \mathrm{JS}(q\|p_\theta) \approx \min_\theta \max_\phi \frac{1}{n}\sum_{i=1}^n S_\phi(\mathbf{x}_i) + \frac{1}{m}\sum_{j=1}^m \log(1 - \exp(S_\phi(T_\theta(\mathbf{z}_j))))$, applying a change of variable $S_\phi \leftarrow \log S_\phi$, we can change the objective to minimizing binary cross-entropy loss where real data is positive. The two networks play a minimax game, at equilibrium the discriminator cannot distinguish real from fake.
- **Flow Models:** We again learn the data density q, but this time explicitly parameterize our model as a push-forward map from Gaussian noise to our data with the objective $\min_\theta \mathsf{KL}(q\|p_\theta)$. Let $r(\mathbf{z})$ be our noise density and T be the push-forward, then a theorem tells us $p(\mathbf{x}) = (T_\# r)(\mathbf{x}) = r(T^{-1}\mathbf{x})|\det(\nabla T^{-1}\mathbf{x})| = r(T^{-1}\mathbf{x})/|\det(\nabla T(T^{-1}\mathbf{x}))|$. To train our model we need T^{-1} (which is slow) and to sample we need T, so modelling the inverse doesn't help. To solve this, we enforce that T is an increasing triangular map, so that $x_1 = T_1(z_1), x_2 = T_2(z_1, z_2), \dots, x_d = T_d(z_1, \dots, z_d)$ and so that $T_j(z_1, \dots, z_j)$ is increasing w.r.t. z_j. Enforcing this makes ∇T a lower triangular matrix with a positive diagonal, making it easy to compute the determinant $\det(\nabla T)$ and since T_j is increasing in z_j, we can find T_j^{-1} by a binary search. Moreover, a theorem tells us there is a unique increase triangular push-forward map from r to q (and so T characterizes q). To ensure that early outputs are not disadvantaged, we compose multiple triangular maps, permuting the outputs of one to get the input of the next. The most common flow model, real-NVP, simplifies the model further by using only $x_i = T(z_i)$ for $i = 1, \dots, L$ and $x_j = T(z_1, \dots, z_L, z_j)$ for $j = L+1, \dots, D$, alternating which partition of the outputs are transformed univariately and multivariately after each layer. Neural Autoregressive Flow uses a neural network to model the pushforward and Polynomial Flow uses polynomials to model the pushforward. We can view T as a multivariate generalization of the quantile function.
- **Neural ODE:** In a normal model, we have $\mathbf{x}_{t+1} = \mathbf{x}_t + \eta_t f_t(\mathbf{x}_t) = T_t(\mathbf{x}_t)$, which is equivalent to $d\mathbf{x}_{t+1} = f_t(\mathbf{x}_t)dt$. By a change of variables we get $\log p_{t+1}(\mathbf{x}_{t+1}) = \log p_t(\mathbf{x}_t) - \log|\det(I + \eta_t \cdot \partial_{\mathbf{x}} f_t(\mathbf{x}_t))| \approx \log p_t - \eta_t\langle\partial_x, f_t(\mathbf{x}_t)\rangle$ since off-diagonal elements in the determinant are relatively small when η_t is small. This yields $\frac{d\log p_t(\mathbf{x}_t)}{dt} = -\langle\partial_{\mathbf{x}}, f_t(\mathbf{x}_t)\rangle$.
- **Diffusion Models:** We again learn the data density q, but this time model an SDE transforming our data to noise and reverse the SDE. Consider the SDE $d\mathbf{x}_{t+1} = f_t(\mathbf{x}_t)dt + G_t(\mathbf{x}_t)dW_t$, discretized as $\mathbf{x}_{t+1} \approx \mathbf{x}_t + \eta_t f_t(\mathbf{x}_t) + g_t(\mathbf{x}_t)$ where $g_t(\mathbf{x}_t) \sim \mathcal{N}(0, \eta_t^2 G_t(\mathbf{x}_t)G_t(\mathbf{x}_t)^T)$ so that $\mathbf{x}_{t+1}$ is a noisy version $\mathbf{x}_t$. The Kolmogorov equations tell us $\partial_t p_t = -\langle\partial_{\mathbf{x}}, p_t f_t\rangle + \frac{1}{2}\langle\partial_{\mathbf{x}}\partial_{\mathbf{x}}^T, p_t G_t G_t^T\rangle$ (and the backward time equation $-\partial_s p_s = \langle f_s, \partial_x p_s\rangle + \frac{1}{2}\langle G_s, G_s^T, \partial_x\partial_x^T p_s\rangle$). Note any SDE can be transformed to an ODE by using $f_t - \frac{1}{2}(G_t G_T^T)\partial_x - \frac{1}{2}(G_t G_t^T)\partial_x \log p_t$. Moreover, any SDE can be reversed as $d\bar{\mathbf{x}}_{t+1} = \bar{f}_t(\bar{\mathbf{x}}_t)dt + G_t(\bar{\mathbf{x}}_t)dW_t$ where $\bar{f}_t = -f_t + (G_t G_t^T)\partial_{\mathbf{x}} + (G_t G_t^T)\partial_x \log p_t$. Note we choose f and G, and a certain choice allows us to model SGD using an SDE, but we often choose f and G to be simple, and when G doesn't depend on $\mathbf{x}$, the $(G_t G_t^T)\partial_{\mathbf{x}}$ term disappears. The only piece we don't know is the score function $s_p = \partial_{\mathbf{x}} \log p_t$, so we model it using a neural net. We optimize it by $\min_\theta \mathbb{F}(p_\theta\|q) = \frac{1}{2}\mathbb{E}_{X\sim q}\|\partial_{\mathbf{x}} \log p_\theta(X) - \partial_{\mathbf{x}} \log q(X)\|_2^2 = \mathbb{E}_{X\sim q}[\frac{1}{2}\|s_p(X)\|_2^2 + \langle\partial_{\mathbf{x}}, s_p(X)\rangle + \frac{1}{2}\|s_q(X)\|_2^2]$. We can also use a denoising autoencoder when we have a joint density $q(\mathbf{x}, \mathbf{z})$ for a latent variable Z, this is especially useful when it easy to find $q(\mathbf{x}|\mathbf{z})$ (e.g., from the forward SDE when f_t is affine). In this case we get $\mathbb{F}(p\|q) = \frac{1}{2}\mathbb{E}_{(\mathbf{x},\mathbf{z})\sim q}[\|s_p(\mathbf{x}) -$

$\partial_x \log q(\mathbf{x}|\mathbf{z})\|^2 + \|s_q(\mathbf{x})\|_2^2 - \|\partial_x \log q(\mathbf{x}|\mathbf{z})\|_2^2]$ since the cross-product terms are the same. In diffusion models, we only need to learn the score function with the objective $\min_\theta \hat{\mathbb{E}}_{t\sim\mu,(\mathbf{x}_t,\mathbf{x}_0)\sim q} \lambda_t \|s_t(\mathbf{x}_t;\theta) - \partial_x \log q(\mathbf{x}_t|\mathbf{x}_0)\|_2^2$, where λ_t weights certain timesteps more heavily. At inference time, we run either the reverse SDE $d\bar{\mathbf{x}}_{t+1} = -f_t + (G_t G_t^T)\partial_x + (G_t G_t^T) s_t(\bar{\mathbf{x}}_t;\theta)dt + G_t(\bar{\mathbf{x}}_t)dW_t$ or ODE $d\bar{\mathbf{x}}_{t+1} = f_t - \frac{1}{2}(G_t G_t^T)\partial_{\mathbf{x}} - \frac{1}{2}(G_t G_t^T) s_t(\bar{\mathbf{x}}_t;\theta)dt$ using a solver (e.g., Euler-Maruyama). Modern models (e.g., stable diffusion) also use an auto-encoder to reduce images to lower dimensional latent features and add conditioning (e.g., text) either by concatenating it to the latent features or adding it to the attention mechanism of the score function (modelled using a U-Net).

- **Contrastive Learning:** Contrastive learning is a self-supervised way of pre-training (especially vision) models. One of the first popular methods is *SimCLR*, where images are passed through stochastic data augmentations to generate two views, and then the model followed by a projection head produces features for all augmented views. A contrastive loss encourages all features from the same image (positive pairs) to be close and features from other images (negative pairs) to be far from one another, as measured by the cosine similarity $\frac{z_i \cdot z_j}{\|z_i\|\|z_j\|}$. While this is a simple and effective method, it requires strong augmentations and large batch sizes. *MoCo* addresses the memory issue by introducing a memory bank. In particular, of the two views, one of them goes through the current encoder (query) and the other through a momentum-moving average of the encoder (key) to form positive pairs. Negative pairs are taken from a queue of previous batches, and the current keys are then added to the queue. *BYOL* simplifies the idea of an online and momentum network further by having the features passed to the online network go through a prediction head and then directly minimizing the ℓ_2 distance between the momentum network's features and the predicted features (still with two augmented views). The currently favoured method, *CLIP*, uses natural language to help supervise the contrastive task using pairs of images and text captions, producing both a pre-trained image and text encoder. The text caption goes through a text encoder and the image goes through an image encoder, they are then both trained to use a contrastive loss to maximize the similarity of positive pairs and minimize the similarity of negative pairs (caption from a different image). Zero-shot prediction can then be achieved by producing a set of labels from the text encoder and selecting the label which has the highest similarity with the encoded image.

Trustworthy Machine Learning

- **Adversarial Attacks:** Deep models are sensitive to inputs, and adding carefully constructed noise to an input (e.g., image) can cause unexpected behaviour. In particular, there exist small Δx such that $f(x + \Delta x) \neq f(x)$ due to f not being sufficiently smooth, or *robust*. The issue arises from the existence of hard boundaries in our dataset, as such there exist adversarial examples for any (non-constant) classifier, and so we have an accuracy-robustness trade-off. An adversarial example is found as $\max_{\|\mathbf{x}-\mathbf{x}_0\|\leq\varepsilon} f(\mathbf{x}, y; \mathbf{w})$, e.g., where $f(\mathbf{x}, y; \mathbf{w}) = -\log p_y(\mathbf{x}; \mathbf{w})$. Note $\mathbf{w}$ is constant, we optimize for $\mathbf{x}$. In a targeted attack we try to make the model predict a specific output, e.g., maximize $f(\mathbf{x}, y; \mathbf{w}) = \log p_{\bar{y}}(\mathbf{x}; \mathbf{w})$ for $\bar{y} \neq y$. Two common methods are the Fast Gradient Sign Method where we find examples by repeatedly doing $\mathbf{x} \leftarrow \mathbf{x} + \varepsilon \operatorname{sign}(\nabla_{\mathbf{x}} f(\mathbf{x}, y; \mathbf{w}))$ and Project Gradient Method $\mathbf{x} \leftarrow +\eta \nabla_{\mathbf{x}} f(\mathbf{x}, y; \mathbf{w})$, after each step in both we must project $\mathbf{x} \leftarrow \text{Proj}(\mathbf{x})$ so that the example is within ε distance. Usual backprop can compute gradients w.r.t. input $\mathbf{x}$ when we have the model (white box), but if we don't have the model (black box) then we must approximate this gradient as $\lim_{\delta\downarrow 0} \frac{f(\mathbf{x}+\delta) - f(\mathbf{x})}{\delta}$. To improve model robustness, it is common to train using adversarial attacks in a sort of minimax between the model and attacker. Data augmentation/regularization and robustness are closely related, for instance, adversarial training on a linear regression model is equivalent to adding lasso regression. Another way to improve robustness is with robust losses, such as Huber's loss (mix of MSE and MAD, equivalent to gradient clipping), or variational losses which let the model ignore certain outliers from the loss.
- **Data Poisoning:** Given a training distribution μ, the goal is to produce a poison distribution ν such that a model trained the mixed distribution $\chi \propto \mu + \varepsilon_d \nu$ (where ε_d is the poisoning fraction) performs worse than it would when trained only on μ. This is formulated as $\max_{\nu\in\Gamma} L(\tilde{\mu}; \mathbf{w}^*)$ where $\mathbf{w}^* = \arg\min_{\mathbf{w}} L(\chi; \mathbf{w})$

where $\tilde{\mu}$ is the validation set. Unlike adversarial attacks where the model is already trained and examples are found to break the model, in data poisoning the examples must be selected to break the model before it is even trained. We say that the parameter $\mathbf{w}$ is ε_d-poisoning reachable if there is a poisoning distribution ν such that $g(\chi;\mathbf{w}) = g(\mu;\mathbf{w}) + \varepsilon_d g(\nu;\mathbf{w}) = 0$, i.e., $\mathbf{w}$'s gradients vanish w.r.t. loss ℓ over over the mixed distribution. This can be implemented by a gradient canceling attack $\min_\nu \frac{1}{2}\|g(\mu) + \varepsilon_d g(\nu)\|_2^2$. E.g., in logistic regression, we see that $-0.28 \approx= \inf_t \frac{-t}{1+\exp(t)} \leq \langle \mathbf{w}, g(\nu)\rangle \leq \sup_t \frac{-t}{1+\exp(t)} = \infty$, thus $\mathbf{w}$ is ε_d-poisoning reachable iff $\langle \mathbf{w}, g(\mu)\rangle + \varepsilon_d \langle \mathbf{w}, g(\nu)\rangle = 0$ iff $\varepsilon_d \geq \max\{\frac{\langle \mathbf{w}, g(\mu)\rangle}{0.28}, 0\}$.

- **Differential Privacy:** Even after anonymizing data, it's still often possible to identify people by the details of their responses. As a result, people may not want to divulge true information due to the risk of being identified. A way around this is by randomizing responses, e.g., with a 50% chance select a random answer and with a 50% chance answer honestly, this gives plausible deniability to everyone. Formally, let $M : \mathcal{D} \to \mathcal{Z}$ be a randomized mechanism, then M is (ε, δ)-DP if for any $D, D' \in \mathcal{D}$ differing by one point, then $P(M(D) \in E) \leq \exp(\varepsilon) P(M(D') \in E) + \delta$. The idea is that the log odds of an event E happening in either dataset is less than ε, then differencing will be harder since any event is (close to) equally likely in each dataset. Alternatively, we can view DP as $\text{FPR} \leq \exp(\varepsilon) \cdot \text{TPR} + \delta$ for $H_0 : D$ and $H_A : D'$, i.e., how hard it is to distinguish one dataset from the other. Alternatively, M is (α, ε)-RDP if $\mathbb{D}_\alpha(M(D)\|M(D')) = \frac{1}{\alpha-1} \log \mathbb{E}_{x\sim q} \left(\frac{p(x)}{q(x)}\right)^\alpha \leq \varepsilon$ where p and q are the densities of $M(D)$ and $M(D')$ respectively. Note $\lim_{\alpha\downarrow 1} \mathbb{D}_\alpha = \mathsf{KL}$, and if M is (α, ε)-RDP then M is $(\varepsilon + \frac{1}{\alpha-1}\log\frac{1}{\delta}, \frac{\delta}{\alpha})$-DP. Properties of DP: (1) *post-processing*: if M is DP, then $T \circ M$ is DP for any T, (2) *parallel composition:* if each M_k is DP, then $M(D) = (M_1(D_1), \ldots, M_K(D_K))$ is DP, (3) *sequential composition:* $(M(D), N(D, M(D)))$ is $(\alpha, \varepsilon_M + \varepsilon_N)$-RDP, (4) *group of k:* if D and D' differ by k elements, then M is $(k\varepsilon, \delta)$-DP, (5) *subsampling:* subsampling D from a larger dataset $\mathcal{D}$ amplifies privacy. The Gaussian mechanism, $M(D) = f(D) + \varepsilon$ for $\varepsilon \sim \mathcal{N}(0, \Sigma)$, has sensitivity $\Delta_2 f = \sup_{D\sim D'} \|(f(D) - f(D'))^T \Sigma^{-1} (f(D) - f(D'))\|_2^2$ and is $(\alpha, \frac{\alpha}{2}\Delta_2 f)$-RDP. This leads us to DP-SGD, where you train as usual, but clip gradients to C (to ensure that $\Delta_2 f$ is bounded C) and then add Gaussian noise $\sigma C \varepsilon$ to make your model $(\alpha, \frac{\alpha}{2}\Delta_2 f)$-RDP.
- **Neuron Activation Modeling:** Ways to understand neuron activations including fixing the network and finding the test set examples that maximize the activation, or performing gradient ascent on the input. Another way is gradient saliency, where the gradients of the input w.r.t. the neuron are visualized. However, these don't always work, e.g., for the OR problem consider fixing $x_1 = x_2 = 1$ and varying only one input concludes that $y = 1$ is constant and x_1, x_2 don't matter.
- **Attribution:** Suppose we have a reward function $u : 2^{[n]} \to \mathbb{R}$ (with $u(\emptyset) = 0$) based on what subset of features are present and want to determine marginal contribution ϕ_i of each feature. *Probabilistic values:* we can find an additive approximation, e.g., by $\phi_i = u(S \cup i) - u(S \setminus i)$ or $\phi_i = u([n]) - u([n] \setminus i)$ but these don't take into account other features. A better approximation would make use of averaging: $\phi_i = \sum_{S \not\ni i} p_s (u(S \cup i) - u(S))$ where p_s is the probability of a set of size $|S| = s$. The choice p_s is important, the choice $p_s = \frac{1}{2^{n-1}}$ leads to the *Banzhaf value*, but there is no guarantee $\sum_{i=1}^n \phi_i = 1$. We also can't weigh each feature equally, since similar features will collude to get similar results. *Shapley's value:* with $p_s = \frac{s!(n-s-1)!}{n!}$ (almost reciprocal of binomial coefficient) it is the only choice that satisfies (1) *linear:* $\phi_i(u + v) = \phi_i(u) + \phi_i(v)$ for separate games u, v (2) *symmetry:* if $u(S \cup i) = u(S \cup j)$ for all $S \not\ni i, j$, then $\phi_i = \phi_j$, (3) *null:* if $u(S \cup i) = u(S)$ for all $S \not\ni i$, then $\phi_i = 0$, and (4) *efficient:* $\sum_i \phi_i = u([n])$. Using Monte Carlo estimation, we can get $\hat{\phi}_i$ by sampling m subsets $S \not\ni i$ with probability $\propto \binom{n-1}{s} p_s$ and average $u(S \cup i) - u(S)$ over all samples. This estimation yields $P(|\hat{\phi}_i - \phi_i| \geq \varepsilon) \leq 2\exp(-m\varepsilon^2/2)$, so to get $\|\hat{\phi} - \phi\|_\infty \leq \varepsilon$ with probability $1 - \delta$, we need $m \in O(\frac{n}{\varepsilon^2}\log\frac{n}{\delta})$ samples. A similar method can get an estimation of Banzhaf value within (ε, δ)-ℓ_2 error or $(\frac{\varepsilon}{\sqrt{n}}, \delta)$-$\ell_\infty$ error in $m \in O(\frac{1}{\varepsilon^2}\log\frac{n}{\delta})$. *Random order value:* where π is a permutation $[n]$, we define $\psi_i(u, \pi) = u(\{\pi(1), \ldots, \pi(k)\}) - u(\{\pi(1), \ldots, \pi(k-1)\})$ where $\pi(k) = i$ and let $\phi(u) = \mathbb{E}_\pi \psi_i(u, \pi)$. *Least-square value:* $\hat{\phi} = \arg\min_{\phi\in\mathbb{R}^n} \sum_{S\subseteq[n]} q_s (u(S) - \phi(S))^2$ such that $u([n]) = \sum_i \phi_i$ where $q_s = p_s + p_{s-1}$ and can be approximated with $m \in O(\frac{n}{\varepsilon^2}\log\frac{n}{\delta})$ samples. Note that Shapley value $\subseteq$ random order value $\subseteq$ probabilistic value $\subseteq$ least-square value.