Decision Trees

Splitting criterion is the function that measures how good or useful splitting on a particular feature is for a specified dataset: training error rate (minimize), gini impurity (minimize; CART), mutual information (maximize; ID3)

$$\text{Entropy } H(S) = -\sum_{v \in V(S)} \frac{|S_v|}{|S|} \log_2 \frac{|S_v|}{|S|}$$

Where $V(S)$ is the set of unique values in S and S_v is the collection of elements in S with value v. $H(pure) = 0$ and $H(50/50) = 1$.

$$\text{Mutual Information } I(Y; x_d) = H(Y) - \sum_{v \in V(x_d)} f_v \cdot H(Y_{x_d=v})$$

Where x_d is a feature, Y is the collection of all labels, $v(x_d)$ is the set of unique values x_d, f_v is the fraction of inputs where $x_d = v$, $Y_{x_d=v}$ is the collection of labels where $x_d = v$.

Termination criteria: all labels in D are the same, tree is too deep, label set has low enough entropy, D is empty, all feature vectors in D are identical.

Inductive bias: the principle by which an ML algorithm generalizes to unseen samples (ID3: shortest tree; zero training error; high MI features on top)

Pros: interpretable, efficient, can be used for classification/regression, compatible with categorical/real-valued; **Cons:** learned greedily (only considers immediate impact on splitting criterion, not guaranteed to find optimal tree); liable to overfit

Overfitting happens when model is too complex; fit noise/outliers in train; not enough inductive bias (true error ¿ train error) (prevention: do not split past a fixed depth; do not split leaves with fewer data points; do not split leaves where MI is less than threshold; take majority vote in impure leaves; do reduced-error pruning)

Underfitting model too simple; does not capture pattern; too much inductive bias (regularization increases bias)

Splitting on real-valued features can split on same feature more than once based on values; max leaves $= N$; max depth $= N$

Binary att.: max leaves $= 2^{(k-1)}$; max depth $= O(k)$; split once

kNN

kNN: classify a point as the most common label among the labels of the k-nearest training points; 1-NN always have zero training error

Cover and Hart: "half the classification information in an infinite sample is contained in the nearest neighbor"

$$err(h) = (1 - \pi(x^{(\hat{i}(x'))}))\pi(x') + \pi(x^{(\hat{i}(x'))})(1 - \pi(x'))$$

$$N \to \infty, \pi(x^{(\hat{i}(x'))}) \to \pi(x')$$

$$err(h) \to 2(1 - \pi(x'))\pi(x') \leq 2\min(1 - \pi(x'), \pi(x')) = 2err(h)$$

Tie-breaking: look at next NN, 1-NN, majority vote, distance-weighted votes, change distance metric

If k=1, overfitting (complicated DB); if k=N (no DB); sweet spot in the middle; increasing k and decreasing depth in a DT have similar regularizing effects

Inductive bias: data points near each other have the same label; classes are piecewise linearly separable; features are scaled (features have equal weight)

As $N \to \infty$ and k follows N, then the true error of a kNN model -¿ Bayes error rate (Bayes classifier: counts; majority vote); more training data, can use more complex models w/o overfitting

Model selection: model (set of classifiers a learning algorithm searches through to find the best one); model parameters (numeric values or structures that are selected by the learning algorithm; kNN is non-parametric); hyperparameters (tunable aspects of the model that are not selected by the learning algorithm; must be pre-specified; do not use test set to find; K-fold cross-validation: use one dataset fold as a validation set once, average the error)

Linear Regression

Linear Regression: assume that there is a linear relationship between the data and its labels (linear dep); objective function: minimize the mean square error; solve in closed-form

$$\ell_D(w) = \frac{1}{N}\sum_{n=1}^{N}(X^{(n)T}w - y^{(n)})^2 = \frac{1}{N}(w^T X^T X w - 2w^T X^T y + y^T y)$$

$$\nabla_w \ell_D(w) = \frac{1}{N}(2X^T X w - 2X^T y + 0) \to \hat{w} = (X^T X)^{-1} X^T y$$

$$H_w \ell_D(w) = \frac{1}{N}(2X^T X) \geq 0 \text{ which is positive semi-definite}$$

$X^T X$ **invertibility:** when $N >> D + 1$, $X^T X$ is almost always full-rank and therefore invertible, as long as features are not linearly dependent (then infinitely many solutions); computational cost is $O(D^3)$

Gradient Descent

Gradient descent: move some distance in the most downhill direction; iterative method for minimizing functions; requires the gradient to exist everywhere

$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} + \eta^{(0)} \|\nabla_{\mathbf{w}} \ell_D(\mathbf{w}^{(t)})\| \left(-\frac{\nabla_{\mathbf{w}} \ell_D(\mathbf{w}^{(t)})}{\|\nabla_{\mathbf{w}} \ell_D(\mathbf{w}^{(t)})\|}\right)$$

Termination criteria: set max timesteps; small gradient; no change in output

Convexity: a function $f : \mathbb{R}^D \to \mathbb{R}$ is convex if $\forall \mathbf{x}^{(1)} \in \mathbb{R}^D, \mathbf{x}^{(2)} \in \mathbb{R}^D$ and $0 \leq c \leq 1$, $f(c\mathbf{x}^{(1)} + (1-c)\mathbf{x}^{(2)}) \leq cf(\mathbf{x}^{(1)}) + (1-c)f(\mathbf{x}^{(2)})$ (if the $\leq$ sign is $<$ then it is strictly convex); In a convex function, every local minimum is a global minimum; GD is a local optimization algorithm and will converge to local minimum, ideal for convex obj. functions.

MLE/MAP

Probabilistic learning: unknown target distribution, want to find a distribution $p(Y|x)$ that best approximates the target distribution $y \sim p*(Y|X)$

Deterministic learning: unknown deterministic function $c* : X \to y$, find a classifier $h : X \to y$ that best approximates $c*$

Maximum Likelihood: every valid probability has a finite amount of probability mass as it must sum/integrate to 1, and we ant to set the parameter such that the likelihood of the samples is maximized; maximizes the log likelihood of the observations

$$LL(\theta) = \sum_{n=1}^{N} \log p(x^{(n)}|\theta), \text{ we use } f \text{ if X is continuous w/ PDF}$$

$$\hat{\theta}_{MLE} = \text{argmax}_\theta P(D|\theta)$$

Maximum a Priori: sometimes, we have prior information we want to incorporate into parameter estimation, so we use Bayes rule to reason about the posterior distribution over the parameters (when we have uniform prior, MAP=MLE); maximizes the log posterior of the parameters conditioned on the observations

$$\hat{\theta}_{MAP} = \text{argmax}_\theta P(\theta|D) = \text{argmax}_\theta \frac{P(D|\theta)P(\theta)}{P(D)} \propto \text{argmax}_\theta P(D|\theta)P(\theta)$$

Conjugate priors: for a given likelihood function $p(D|\theta)$, a prior $p(\theta)$ is called a conjugate prior if the resulting posterior distribution $p(\theta|D)$ is in the same family as $p(\theta)$, i.e. $p(\theta|d)$ and $p(\theta)$ are the same type of random variable with different parameters

$p(x|\theta) = \phi^x(1-\phi)^{1-x}$; $f(\phi|\alpha,\beta) = \frac{\phi^{\alpha-1}(1-\phi)^{\beta-1}}{B(\alpha,\beta)}$; $B(\alpha,\beta) = \int_0^1 \phi^{\alpha-1}(1-\phi)^{\beta-1}d\phi$

$$f(\phi|x,\alpha,\beta) = \frac{p(x|\phi)f(\phi|\alpha,\beta)}{p(x|\alpha,\beta)} = \frac{\text{Bernoulli likelihood} \cdot \text{Beta prior}}{\text{evidence}}$$

$$p(x|\alpha,\beta) = \int_0^1 P(x|\phi)f(\phi|\alpha,\beta)d\phi = \int_0^1 \phi^x(1-\phi)^{1-x}\frac{\phi^{\alpha-1}(1-\phi)^{\beta-1}}{B(\alpha,\beta)}$$

Bernoulli: α and β dictates roughly how much data you need to observe to take you off your prior belief

Naive Bayes

BOW model: easy to compute, interpretable, does not take into account word frequency or relative co-occurence, vocabulary is small

Naive Bayes: make NB assumption (conditional independence: features used in the model are considered independent given the class variable, assume IID data points, parameters are $P(x|y)P(y)$, maximize LL, solve using MLE

$P(X|Y) = P(X_1 \cap X_2 ... \cap X_D|Y) = \prod_{d=1} P(X_d|Y)$; $P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)} \propto P(X|Y)P(Y)$

Pros: significantly reduces the no. of parameters, reduces overfitting; **Cons:** terrible assumption (features are almost always not conditionally independent)

$\ell_D(\pi,\theta) = \log\prod_{n=1}^{N} P(x^{(n)}, y^{(n)}|\pi,\theta) = \log\prod_{n=1}^{N}\left(\prod_{d=1}^{D} P(x^{(n)}|y^{(n)},\theta_{d,1},\theta_{d,0})\right)P(y^{(n)},\pi)$

Bernoulli NB: $Y \sim \text{Bernoulli}(\pi); \hat{\pi} = \frac{N_{Y=1}}{N}$; $X_d|Y \sim \text{Bernoulli}(\theta_{d,y}); \hat{\theta}_{d,y} = \frac{N_{Y=y,x_d=1}}{N_{Y=y}}$; binary labels and features

$P(y=1|x') \propto P(x'|y=1)P(y=1) = \hat{\pi}(\prod_{d=1}^{D}\hat{\theta}_{d,1}^{x'_d}(1-\hat{\theta}_{d,1})^{1-x'_d})$

$P(y=0|x') \propto (1-\hat{\pi})(\prod_{d=1}^{D}\hat{\theta}_{d,0}^{x'_d}(1-\hat{\theta}_{d,0})^{1-x'_d}$

Gaussian NB:

$Y \sim \text{Bernoulli}(\pi); \hat{\pi} = \frac{N_{Y=1}}{N}; X_d|Y \sim \text{Gaussian}(\mu_{d,y}, \sigma^2_{d,y}); \hat{\mu}_{d,y} = \frac{1}{N_{Y=y}}\sum_{n:y^{(n)}=y} x_d^{(n)}$;

$\hat{\sigma}^2_{d,y} = \frac{1}{N_{Y=y}}\sum_{n:y^{(n)}=y}(x_d^{(n)} - \hat{\mu}_{d,y})^2$; $G(x,\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma})$

Unseen word-label pairs: add a prior over the parameters

No. of parameters: d features x_i takes in K values, y takes in M labels; $\neg NB = P(X|y) = P(x_1|y)P(x_2|y,x_1)P(x_3|x_2,x_1,y)...$

NB $= M \times (K-1) \times d + (M-1)$; $\neg$**NB** $= M \times (K^d - 1) + (M-1)$

Logistic Regression

Modeling the posterior and logistic decision boundary

$P(Y=1) = \text{logit}(\mathbf{w}^T\mathbf{x}) = \frac{1}{1+\exp(-\mathbf{w}^T\mathbf{x})} = \frac{\exp(\mathbf{w}^T\mathbf{x})}{\exp(\mathbf{w}^T\mathbf{x})+1}$; $P(Y=0|\mathbf{x}) = 1 - P(Y=1|\mathbf{x}) = \frac{1}{\exp(\mathbf{w}^T\mathbf{x}+1)}$

$\frac{P(Y=1|\mathbf{x})}{P(Y=0|\mathbf{x})} = \exp(\mathbf{w}^T\mathbf{x}) \to$ log odds $= \mathbf{w}^T\mathbf{x}$; $P(Y=1|\mathbf{x}') = \text{logit}(\mathbf{w}^T\mathbf{x}') \geq \frac{1}{2} \to \mathbf{w}^T\mathbf{x}' \geq 0$

Logistic function is preferable because it is a valid probability, differentiable everywhere, monotonically increasing, linear DB

Logistic Regression: assume IID data, assume $P(y=1|x) = \text{logit}(w^T x)$, parameters are w; obj. fn. is to max. cond. LL est. = min. neg. cond. LL est.; conditional because we only model the posterior, we do not model distribution over X, but $P(Y|X)$; $\ell_d(\mathbf{w})$ is concave, i.e. $f''(x) < 0$

$\ell_d(\mathbf{w}) = -\log\prod_{n=1}^{N} P(y^{(n)}|\mathbf{x}^{(n)},\mathbf{w}) = -\sum_{n=1}^{N} y^{(n)}\log(P(Y=1|\mathbf{x}^{(n)},\mathbf{w})) + (1-y^{(n)})\log(P(Y=0|\mathbf{x}^{(n)},\mathbf{w}))$

$\ell_d(\mathbf{w}) = -\sum_{n=1}^{N} y^{(n)}\log\left(\frac{P(Y=1|\mathbf{x}^{(n)},\mathbf{w})}{P(Y=0|\mathbf{x}^{(n)},\mathbf{w})}\right) + \log(P(Y=0|\mathbf{x}^{(n)},\mathbf{w})) = -\sum_{n=1}^{N} y^{(n)}\mathbf{w}^T\mathbf{x}^{(n)} - \log(1+\exp(\mathbf{w}^T\mathbf{x}^{(n)}))$

Stochastic GD: if the datapoint is sampled uniformly at random, then the expected value of the pointwise gradient is proportional to the full gradient ($O(D)$); **Mini-batch GD:** random batches instead of single points ($O(BD) << O(ND)$)

Regularization

Regularization: constrain models to prevent them from overfitting; learning algorithms are optimization problems and regularization imposes constraints on the optimization

Hard constraints: shrink the hypothesis space to weed out complex models prone to overfitting; **Soft constraints:** use L2-norm and impose a max. value on the weights

Lagrange multipliers allows for unconstrained opt. from constraints

$\ell_d(\mathbf{w}) = (X\mathbf{w}-\mathbf{y})^T(X\mathbf{w}-\mathbf{y})[\mathbf{w}^T\mathbf{w}] \to \nabla_{\mathbf{w}}\ell_D(\hat{\mathbf{w}}_{MAP}) = -2\lambda_c\hat{\mathbf{w}}_{MAP} \to \nabla_{\mathbf{w}}\left[\ell_D(\hat{\mathbf{w}}_{MAP}) + \lambda_c\hat{\mathbf{w}}^T_{MAP}\hat{\mathbf{w}}_{MAP}\right]$

$2(X^T X\hat{\mathbf{w}}_{MAP} + X^T X + \lambda_c\hat{\mathbf{w}}_{MAP}) = 0 \to \hat{\mathbf{w}}_{MAP} = (X^T X + \lambda_c I_{D+1})^{-1}X^T y$ (L2 regularized LinReg)

Ridge/L2 encourages small weights $r(w) = \|w\|_2^2 = \sum_{d=0}^{D} w_d^2$;

Lasso/L1 encourages sparsity $r(w) = \|w\|_1 = \sum_{d=0}^{D}|w_d|$; **L0** encourages sparsity (intractable) $r(w) = \|w\|_0 = \sum_{d=0}^{D}\mathbb{1}(w_d \neq 0)$

MAP with Gaussian prior

$w_d \sim N(0, \sigma^2/\lambda)$; $\hat{\mathbf{w}} = (X^T X + \lambda_c I_{D+1})^{-1}X^T\mathbf{y}$

Neural Networks

Efficient because we can reuse outputs from intermediate parameters

LinReg and LogReg as 1-NN: the hidden layer is not required (will be a hidden layer of identity transformations)

Feed-forward model contains layers (input, hidden, output), activation function, weight matrices, signals and outputs

Architecture Layer $l-1$ has $D^(l-1)$ nodes with their corresponding weights $w_{j,0}^{(l)}$, we have $\mathbf{s}^{(l)} = W^{(l)}\mathbf{o}^{(l-1)}$ with shape $W^{(l)} \in \mathbb{R}^{D^l \times (D^{(l-1)}+1)}$

Regression: squared error
$\ell^{(i)}(W_{(t)}^{(1)}, ..., W_{(t)}^{(L)}) = (h_{W^{(1)},...,W^{(L)}}(\mathbf{x}^{(i)}) - y^{(i)})^2$

Binary classification: cross-entropy loss
$\ell^{(i)}(W_{(t)}^{(1)}, ..., W_{(t)}^{(L)}) = -y^{(i)} \log P(y^{(i)} = 1|\mathbf{x}^{(i)}, \mathbf{w}^{(l)}, ..., \mathbf{w}^{(L)}) + (1-y^{(i)}) \log P(y^{(i)} = 0|\mathbf{x}^{(i)}, \mathbf{w}^{(l)}, ..., \mathbf{w}^{(L)})$

Multi-class classification: cross-entropy loss
$\ell^{(i)}(W_{(t)}^{(1)}, ..., W_{(t)}^{(L)}) = -\sum_{c=1}^{C} y[c] \log h_{W^{(1)},...,W^{(L)}}(\mathbf{x}^{(\mathbf{i})})[c]$; we are computing the entropy of dist. over labels using 2 diff. prob. dists, the one that network returns (A) and the empirical one (corresponds to true label) (B)

Parameters: $(\text{input}+1) \times \text{n}_{\text{hidden}} + (\text{n}_{\text{hidden}}+1) \times \text{out}$

Backpropagation

Computing gradients: a weight affects the prediction through downstream signals/outputs; compute derivatives starting from the last layer and move 'backwards'; derive a recursive definition for the relevant partial derivatives; autodiff is storing the intermediate values and reuse for efficiency (DP)

Partial derivatives
$y = f(z_1, z_2); z_1 = g_1(x); z_2 = g_2(x); \frac{\partial y}{\partial x} = \frac{\partial y}{\partial z_1}\frac{\partial z_1}{\partial x} + \frac{\partial y}{\partial z_2}\frac{\partial z_2}{\partial x}$
$y = f(\mathbf{z}); \mathbf{z} = g(x); \frac{\partial y}{\partial x} = \sum_{d=1}^{D} \frac{\partial y}{\partial z_d}\frac{\partial z_d}{\partial x}$

Solve local minima problem for non-compkex functions: introduce randomness by shuffling or random restarts; momentum learning

SGD+momentum: $W^{(l)} : W_{t+1}^{(l)} \leftarrow W_t^{(l)} - \eta_{MB}^{(0)}(\beta G_{t-1}^{(l)} + G_t^{(l)}) \forall l$

RMSProp:
$S_t = \beta S_{t-1} + (1-\beta)(G_t \odot G_t); W^{(l)} : W_{t+1}^{(l)} \leftarrow W_t^{(l)} - \frac{\gamma}{\sqrt{S_t}} \odot G_t$

Adam:
$M_t = \beta_1 M_{t-1} + (1-\beta_1)G_t; S_t = \beta_2 S_{t-1} + (1-\beta_2)(G_t \odot G_t); W^{(l)} : W_{t+1}^{(l)} \leftarrow W_t^{(l)} - \frac{\gamma}{\sqrt{S_t/(1-\beta_2^t)}} \odot (M_t/(1-\beta_1^t))$

Sigmoid der.: $\sigma'(x) = \sigma(x) \odot (1-\sigma(x))$

Softmax der.: $p(y=j) = \frac{\exp(xj)}{\sum_k \exp(xk)}; \frac{\partial L}{\partial x_j} = \sum_k \frac{\partial L}{\partial p_k}\frac{\partial p_k}{\partial x_j}; \frac{\partial p_k}{\partial x_j} = -p_k p_j$ if $k \neq j, p_j(1-p_j)$ if $k = j$

Gradient's magnitude is not a good metric for proximity to minimum, solutions: terminating gradient early (reg. by limiting the hypothesis set); jitter (random noise to each training datapoint); dropout (randomly remove some of the nodes in the network); weight decay (regularize the weights of the NN directly); these all act like regularization

MLPs are universal approximations: any function that can be decomposed into perceptions can be modeled exactly using a 3-layer MLP → "any bounded, continuous function can be approximated to an arbitrary precision using a 2-layer (1 hidden) feed-forward NN if the activation function is continuous, bounded, and non-constant (hidden layers may need to be of infinite depth if activation functions do not fit constraints)"

CNN

CNN: learn a filter for macro-feature detection in a small window and apply it over the entire image (**filter:** weight matrices that are smaller than my original image applied pixel or element-wise to compute some measure of the pixels in the region)

CNN as FFNN: nodes in the input layer are only connected to some nodes in the next layer but not all nodes, and many of the weights have the same value (constrained to be equal)

Padding: used to preserve image size after convolution (usually pad with 0s bc efficient)

Stride: used to downsample, only apply the conv. to some subset of the image; output is much smaller and greatly reduces the no. of weights to be learned in subsequent layers (many relevant macro-features will tend to span large portions of the image, so taking strides with the convolution tends to not miss out on too much, i.e. 'salient features')

Pooling: combine multiple adjacent nodes into a single model reduces the dimensionality of the input to subsequent layers and thus, the no. of weights learned; prevents the network from slightly noisy inputs

Output: $n_{out} = \lfloor \frac{n_{in}+2p-k}{s} \rfloor + 1$; no. of output channels = no. of filters

Parameters:
$(\text{channels}_{in} \times \text{channels}_{out} \times K_H \times K_W) + \text{channels}_{out}$

Multiple input channels: we can specify a filter for each one and sum the results to get a 2D output tensor; we might want a different filter for each input: different macro-feature detection in later stages (face detection), combine macro-features to build different macro-features

1x1 convolutions: layers of size $c_o \times c_i \times 1 \times 1$ can condense many input channels into fewer output channels c_o if $c_o < c_i$; arbitrarily shrink channel dimensions to lin. comb. of channels to scalar values; typically followed by non-linear act. fn. otherwise they could simply be folded into other convolutions

RNN

RNN: use information from previous parts of the input of the input to inform subsequent predictions; hidden layers learn a useful representation; incorporate the output from earlier hidden layers into later ones; can reuse the weight matrices, so not a lot of parameters
$\mathbf{h}_t = [1, \theta(W^{(1)}\mathbf{x}_t^{(i)} + W_h\mathbf{h}_{t-1})]^T$ and $\mathbf{o}_t = \hat{y}_t^{(i)} = \theta(W^{(2)}\mathbf{h}_t)$

Bidirectional RNN: $\mathbf{h}_t^{(f)} = [1, \theta(W_f^{(1)}\mathbf{x}_t^{(i)} + W_f\mathbf{h}_{t-1}]^T$ and $\mathbf{h}_t^{(b)} = [1, \theta(W_b^{(1)}\mathbf{x}_t^{(i)} + W_b\mathbf{h}_{t+1}]^T; \mathbf{o}_t = \hat{y}_t^{(i)} = \theta(W_f^{(2)}\mathbf{h}_t^{(f)} + W_b^{(2)}\mathbf{h}_t^{(b)})$

Pros: can handle arbitrary seq. len w/o having to increase model size (no. of learnable parameters), trainable via BPTT; **Cons:** vanishing/exploding gradients, does not consider info. from later timesteps (addressed by bi-RNNs), seq. computation, entire seq up to some t is represented using just one vector

Prevent exploding gradients: gradient clipping by scaling gradient to a certain threshold (use L2-norm; more stable); shrink arrow but still point at right direction; can also truncate BPTT by limiting the no. of steps to backdrop through (solves van. too)

LSTM: replace hidden layers with memory cells; each cell computes a hidden representation and a separate internal state C_t which allows information to move through cells w/o affecting h_t

Input gate I_t controls how much the state looks like the normal RNN hidden layer ($I_t = \sigma(W_{ix}\mathbf{x}_t^{(i)} + W_{ih}\mathbf{h}_{t-1})$); **Output gate** O_t releases the hidden representation to later timesteps ($O_t = \sigma(W_{ox}\mathbf{x}_t^{(i)} + W_{oh}\mathbf{h}_{t-1})$); **Forget gate** F_t det. if C_{t-1} affects the current internal state ($F_t = \sigma(W_{fx}\mathbf{x}_t^{(i)} + W_{fh}\mathbf{h}_{t-1})$); **Memory cell** C_t ($C_t = F_t \odot C_{t-1} + J_t \odot \theta(W^{(i)}\mathbf{x}_t^{(i)} + W_h h_{t-1})$)

Language models: convert raw text into sequence data (split raw text into smaller units (tokens) then learn a dense numerical vector representation (embedding; learned through 1-FFNN) for each token; learn/approximate a joint prob. dist. over sequences (use chain rule to predict next word based on prev. words); sample from the implied conditional distribution to generate new sentences

Attention and Transformer

Attention: compute a representation of the input sequence for each token x' in the decoder

Scaled dot-product self-attention: compute a representation for each token in the input sequence by attending to all the input tokens; complexity of $O(n^2)$; can only deal with fixed-length inputs; parallelizable and efficient
$H = \text{softmax}(S)V \in \mathbb{R}^{N \times d_v}$; $S = \frac{QK^T}{\sqrt{(d_k)}} \in \mathbb{R}^{N \times N}$
$Q = XW_Q \in \mathbb{R}^{N \times d_v}$; $K = XW_K \in \mathbb{R}^{N \times d_k}$
$V = XW_V \in \mathbb{R}^{N \times d_v}$; $X \in \mathbb{R}^{N \times D}$
Batch+multihead $= B \times N \times d_v \times H$

Multihead scaled dot product self-attention: we want multiple attention weights to learn different relationships between tokens (like using multiple conv. filters in CNNs); concat. together to get final rep.; common arch. choice is no. of heads = input embedding size

Positional encodings: model needs to infer the order of tokens by adding a position-specific embedding p_t to the token embedding x_t, $x_t' = x_t + p_t$; can be fixed (predetermined function t or learned alongside embeddings) or absolute (only dependent on token's location in sequence/relative to the query token's location)

Layer normalization: small change in weight in an earlier layer can have huge downstream effects; norm. output of layer to always have same (learnable) mean and variance, $H' = \gamma(\frac{H-\mu}{\sigma}) + \beta$

Residual connections: solve the 'degradation' problem where adding more layers worsen performance; add input embedding back to the output of a layer $H' = H(x^{(i)}) + x^{(i)}$; hidden layer needs to learn the residual $r = f(x^{(i)}) - x^{(i)}$

Math Properties

Variance: $V(c) = 0; V(cX) = c^2V(X); V(c+X) = V(X); V(a+bX) = b^2V(X); V(X+Y) = V(X)+V(Y); V(X-Y) = V(X)-V(Y); V(X) = E[(X-E[X])^2] = E[X^2-2XE[X]+E[X]^2] = E[X^2]-2E[X]E[X]+E[X]^2 = E[X^2]-E[X]^2$

Expectation: $E(c) = c; E(cX) = cE(X); E(c'+cX) = c'+cE(X); E(X+Y) = E(X)+E(Y); E(X-Y) = E(X)-E(Y); E(XY) = E(X)E(Y), X \perp\!\!\!\perp Y$

Log rules: $\log_a xy = \log_a x + \log_a y; \log_a(x/y) = \log_a x - \log_a y; \log_a x^r = r\log_a x; \log_a a = 1; \log_a 1 = 0; \log_a a^r = r$

Exp rules:
$a^x a^y = a^{x+y}; a^x/a^y = a^{x-y}; (a^x)^r = a^{rx}; a^1 = a; a^0 = 1; a^{\log_a^r} = r$

Matrices: $Var(AX) = AVar(X)A^T; AA^{-1} = A^{-1} = I; (A^{-1})^{-1} = A; (AB)^{-1} = B^{-1}A^{-1}; (AB)^T = B^TA^T; (A^{-1})^T = (A^T)^{-1}; (A^T)^T = A; (A+B)^T = A^T + B^T; (K*A)^T = kA^T; (A^k)^T = (A^T)^k$

Conditional expectation: $E[X|Y] = \sum_{x_j} x_j p_x(x_j|y)$

Law of total expectation: $E_Y(E_{X|Y}(X|Y)) = E_Y[\sum_x xP(X=x|Y)] = \sum_y[\sum_x P(X=x|Y=y)]]P(Y=y) = \sum_y \sum_x xP(X=x|Y=y)P(Y=y) = \sum_x x\sum_y P(X=x|Y=y)P(Y=y) = \sum_x x\sum_y P(X=x,Y=y) = \sum_x xP(X=x) = E(X)$

Derivatives: $\frac{d}{dx}f[g(x)] = f'[g(x)]g'(x); \frac{d}{dx}[f(x)g(x)] = f(x)g(x)' + g(x)f'(x); \frac{d}{dx}\frac{f(x)}{g(x)} = \frac{g(x)f'(x)-f(x)g'(x)}{[g(x)]^2}; \frac{d}{dx}x^n = nx^{n-1}$

Prob:
$P(X) = \sum_y P(X,Y=y) = \sum_y P(X|Y=y)P(y=y); P(X=1|Y=0) + P(X=0|Y=0) = 1; D_{KL}(P||Q) = \sum_i P(i)\log\frac{P(i)}{Q(i)}$