KNN algorithm

- Learning: storing training examples — instance based learning, memorize train data
- Find the K-most similar examples to x_new
- Training is fast, but prediction is slow
- High K-value = less complex, less sensitive to noise, smoother, low variance, less expressive, high bias
- Low K-value = more complex, sensitive to noise, more expressive, high variance, non-smooth

Measures of location

- Mean: $\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x(i)$
- Median: middle value
- Mode: most common value
- Quartiles: values w/ 25%, 75% above/below
- Covariance: $\frac{1}{n}\sum_{i=1}^{n}(x(i)-\bar{x})(y(i)-\bar{y})$
- Correlation: $\rho(x,y) = \frac{\frac{1}{n}\sum_{i=1}^{n}(x(i)-\bar{x})(y(i)-\bar{y})}{\sigma_x \sigma_y}$

Measures of Dispersion / Variability

- Variance: $\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x(i)-\mu)^2$
- Standard dev: $\sqrt{\text{variance}}$
- Range
- Interquartile range
- Manhattan distance (L_1): $d_M(x,y) = \sum_{i=1}^{n}|x_i - y_i|$
- Euclidean distance (L_2): $d_E(x,y) = \sqrt{\sum_{k=1}^{n}(x_k - y_k)^2}$
- Mahalanobis distance: $d_{MH}(x,y) = \sqrt{(x-y)^T \Sigma^{-1}(x-y)}$

Parametric:
- function form is known
- # of parameter fixed in advance
- Ex: Bayes, perceptron

Non-parametric:
- model structure is known from data
- Ex: classification tree, KNN

1. positivity, symmetry, triangular inequality
- $d(i,j) \geq 0$ for all i,j but $d(i,i)=0$ when i=j
- $d(i,j) = d(j,i)$
- $d(i,j) \leq d(i,k) + d(k,j)$

Probabilistic DT models model P(Y|X) directly, discriminative model

Bayes: generative model = model P(X|Y), P(Y) and use Bayes Rule for P(Y|X)

Bayes Rule:

$P(Y|X) = \frac{P(X|Y)P(Y)}{P(X)}$ (likelihood, prior); P(X) } denominator not important; $\propto P(X|Y)P(Y)$

Bayes classifier = argmax P(X|Y)P(Y)

Assumption for Bayes = conditional independence; $P(Y|X) = \prod_{i=1}^{n} P(x_i|Y)P(Y)$ — controls complexity, prevents overfitting

$P(X_i|Y) = P(X_i|X_k;Y)$

$P(X_i \cap X_k|Y) = P(X_i|Y)P(X_k|Y)$

Likelihood:

- $L(\theta|D) = L(\theta|x_1,\dots,x_n) = P(x_1,\dots,x_n|\theta) = \prod_{i=1}^{n} P(x_i|\theta)$
- NBC likelihood: $L(\theta|D) = \prod_{i=1}^{n} P(y_i|x_i;\theta) \propto \prod_{i=1}^{n} P(x_i|y_i;\theta)P(y_i) = \prod_{i=1}^{n}\prod_{j=1}^{m} P(x_{ij}|y_i;\theta)P(y_i|\theta)$
- For log: $L(\theta|D) = \sum_{i=1}^{n}\log P(x_i|\theta)$
- $\hat{\theta}_{MLE} = \arg\max L(\theta) = \arg\max\left[\sum_{i=1}^{n}\log P(y_i) + \sum_{i=1}^{n}\sum_{j=1}^{m}\log P(x_{ij}|y_i)\right]$
- $P(y) = \frac{\text{count}(y)}{n}$; $P(x|y) = \frac{\text{count}(x,y)}{\text{count}(y)}$

Decision Trees:

- Read from root [top-down]
- pruning helps with noise
- can represent any boolean function
- Most important: choosing root node to minimize tree (attributes that are relatively pure in one label)
  - use information gain
- Shorter trees = more generalization, prediction is easy
- batch algorithm
- classification error

Entropy: impurity, disorder

Entropy(S) = $H(S) = -P_+\log(P_+) - P_-\log(P_-)$; P_+: proportion of positive ex., P_-: proportion of neg examples

$E(S) = -\sum_{i=1}^{k} P_i \log(P_i)$ → info gain = 0, children have same labels as parent → highest IG = between ex. w/ diff labels

Information gain: predicting based on majority vote

$\text{Gain}(S,A) = \text{Entropy}(S) - \sum_{v \in \text{vals}(A)} \frac{|S_A|}{|S|}\cdot\text{Entropy}(S_A)$

If data is skewed heavily, accuracy not best way to evaluate

Precision = $\frac{\text{True Pos}}{\text{Pred Pos}} = \frac{\text{True Pos}}{\text{True Pos} + \text{False Pos}}$

Recall: $\frac{\text{True Pos}}{\text{Actual Pos}} = \frac{\text{True Pos}}{\text{True Pos} + \text{False Neg}}$

Score: 1) Avg: $\frac{P+R}{2}$ 2) F-score: $2\frac{PR}{P+R}$

	True label	True label
Prediction	True Pos	False Pos
Prediction	False Neg	True Neg

Perceptron: search for a separating hyperplane

- updates weights only when mistake is made: $w = w + ryx$, $b = b + ry$
- Convergence: stopping criteria = no more mistakes in training data
  - relies on margin; max mistakes = $\frac{R^2}{\gamma^2}$
- Margin: hyperparameter = $\frac{(wx_i+b)y_i}{\|w\|}$
  - of hyperplane: distance between hyperplane and the closest data point
  - γ of dataset: max margin for that dataset using any weight vector
  - larger margin: more confident in generalization

Regularization: weights w/ better margin = generalize better, prevents overfitting by adding penalty term

- thick separator
- impact of margin regularization ↓ as w ↑
- Perceptron not great, what if mistake made at 1000th ex?
  - → use averaged perceptron

Averaged Perceptron:
- Training: maintain running weighted avg of survived hypotheses
- $\hat{y} = \text{sign}\left(\sum_{k=1}^{K} c_k(w_k\cdot\hat{x} + b_k)\right)$; $\bar{\omega} = \frac{1}{T}\sum_{t=1}^{T}\omega_t$

Summary:
- Easy learning fast
- dependences cannot be modeled, class conditional independence assumption can produce skewed prob
- Model space: parametric model, Bayes Rule, conditional independence assumption
- Scoring: likelihood of data given NBC model form
- Search = MLE optimization - Expressive: KNN > DT > NBC linear model ax+b
- $C_{NB} = \arg\max_{y_j \in Y} \log P(y_j) + \sum_{i \in \text{positions}} \log P(x_i|y_j)$

Perceptron is a simpler version of gradient descent that doesn't include regularization

Update rule:
- Perceptron: $\omega = \omega + \eta(y\cdot x)$
- Hinge-GD: $w = w + \eta(y\cdot x - \lambda\omega)$ → penalizes large weights

Convergence:
- perceptron: only if data is linearly separable
- GD: converge for linear + non-linear

Ensemble Learning Algorithms

- combine predictions of multiple classifiers
  - wisdom of crowds
  - Ex: majority vote, weighted vote, confidence-based, etc

train data → train sample 1 → model 1; train sample 2 → model 2; train sample 3 → model 3 → test data → prediction 1, prediction 2, prediction 3 → test data

- 2 ensembles: bagging and boosting improve performance
  - combine same type classifier multiple times by modifying training set
  - help control bias and variance

Bagging: Bootstrap aggregation, trains models independently (tree 1, tree 2, tree 3, tree 4 → Avg prediction)
- combining many unstable predictors in an ensemble produces a stable predictor
  - reduces variance
- unstable predictor: small changes in train data produces large changes in models
  - Ex: decision tree

Random Forest: in addition to bagging, use random feature selection when determining which attribute to split on.

Hyperparameter = T

Boosting: trains models sequentially (each model corrects errs of prev ones)
- combining many weak (but stable) predictors produces a strong predictor
  - reduces bias without increasing variance
- weak predictor: only weakly predicts correct classifier
- strong learner: we can reduce diff between train and test error by ↑ training set
- weak learner: only guarantees to do better than random
- Boosting allows a weak learner → strong learner [change data dist to focus on "difficult parts" of the data]
  - α_m: weight of the m-th weak model: $\log\left(\frac{1-\text{err}_m}{\text{err}_m}\right)$
  - 1) start with weighing all train examples equally
  - 2) train model and compute error of model
  - 3) increase weights on train cases where model gets wrong so next classifier focuses on difficult examples
  - slow convergence

Ada boosting doesn't overfit

But normal boosting: overfit if too complex, underfit if too weak

Ensemble summary:
- boosting may hurt performance but typically helps more than bagging

Gradient Descent / Ascent:

- Descent: minimize a loss function (move downhill)
  - $\theta_t = \theta_{t-1} - \eta\cdot\nabla f(\theta)$; $\nabla f(\theta) = \frac{d}{d\theta}L(\theta|D)$
- Ascent: maximize scoring, likelihood func (uphill)
  - $\theta_t = \theta_{t-1} + \eta\cdot\nabla f(\theta)$; $\frac{d}{d\theta}L(\theta|D)$

Negative log-likelihood:

$NLL(D|\theta) = -\log L(D|\theta) = -\sum_{i=1}^{n}\log P(x_i|\theta)$

$\nabla Err(w)$ = gradient vector of partial derivatives $\left(\frac{d\,Err(w)}{dw_0},\dots,\frac{d\,Err(w)}{dw_n}\right)$

$w_{i+1} = w_i - \alpha\nabla Err(w_i)$

Ex: for squared loss: $\nabla Err(w_i) = -\sum_{d\in D}(y_d - f(x_d))x_{di}$

Update rule: $\Delta w_i = \alpha\sum_{d=1}^{n}(y_d - w_i\cdot x_d)x_{di}$; $w_{i+1} = w_i + \Delta w_i$

GD: updates after going through whole data set

Stochastic GD: updates after each example

Polynomial Regression: non linear data

Danger: overfitting
- prefer simpler model; sometimes willing to trade ↑ error rate for simpler model

Regularization: $\min_w \sum \text{loss}(y_n, w_n) + \lambda R(w)$
- to avoid overfitting by penalties; Ex: $R(\omega) = \frac{1}{2}\|w\|^2$, $\|w\| = \sqrt{\sum \omega_d^2}$

More examples:
- Squared loss: $L = (\hat{y}-y)^2 = (wx+b-y)^2$
- L1 reg: $L = (wx+b-y)^2 + \lambda|w|$; L2 reg: $L = (wx+b-y)^2 + \lambda w^2$
- update rule: $w - \eta\frac{dL}{dw}$
  - $w_{new} = w - \eta[2x(wx+b-y) \pm \lambda]$ ← derivative; $w_{new} = w - \eta[2x(wx+b-y) + 2\lambda w]$

Smooth approx for logistic regression:
- Hard threshold: 0, $w^Tx < 0$; 1, $w^Tx \geq 0$
- Soft threshold: $g(z) = \frac{1}{1+e^{-z}}$, $z = \omega^Tx$

For likelihood example:

$\log L(\omega) = \sum_m y_i\log h_\omega(x_i) + (1-y_i)\log(1-h_\omega(x_i))$; $h_\omega(x) = P(y=1|x;\omega)$

$w_j = w_j + (y_i - h_\omega(x_i))x_{ij}$

Optimization:
- convenient analysis framework: local/global optimal results
- convenient algorithmic framework: we know how to optimize
- convenient modeling framework: capture relevant properties w/o worrying about procedure for getting it

Non-smooth functions: combinatorial optimization

Smooth functions: differentiable, derivatives are continuous
- gradient based optimization
- if convex: all local optimal = globally optimal
  - solve using convex optimization - single minimum
- if not: use iterative approach to find local minimum

loss function: $\text{Loss}(\hat{y}, y) = (\hat{y}-y)^2$; $\text{Loss}(\hat{y}, y) = \{0: y = \hat{y};\ 1: \text{o.w.}\}$

if regression: $y = \omega^Tx$; classification: $y = \text{sign}(\omega^Tx)$

$Err(\omega) = \frac{1}{2}\sum_{d\in D}(\hat{y}_d - y_d)^2$ → convex, squared loss function

Boosting: use when data has complex patterns, structured/tabular data, handles non-linear, when accuracy is critical

Bagging: if overfitting is a concern

Bias/Variance

Variance: how susceptible learner is to noise
- overfitting
- increases with model complexity

Bias: how expressive, identify target hypothesis
- underfitting
- high if simple model

HB: cannot represent target; LV: doesn't overfit

HB: cannot represent target; HV: overfits

LB: powerful enough to learn; LV: doesn't overfit

LB: powerful enough to learn; HV: overfits

Expected error = bias + variance

If high bias: add more features, reduce regularization tradeoff

if high variance: simplify model, add more examples

Plot learning curve by # of examples to understand performance

Simple model: high validation and train error, high bias and low variance

Complex model: high train and low validation error, high variance & low bias

Perceptron finds any separating hyperplane, we need to find the maximizing margin

SVM:

- find max margin classifier: Margin = $\frac{y(\omega^Tx)}{\|w\|}$, either maximize numerator or minimize denominator
  - $\arg\max_w[y(\omega^Tx)/\|w\|]$
  - 1) fix $\|w\|=1 \Rightarrow w^* = \arg\max_{\|w\|=1}\min_{(x,y)} y(w^Tx)$
  - 2) fix $y(w^Tx) \geq 1 \Rightarrow w^* = \arg\min\|w\|$; Minimize: $\frac{1}{2}\|w\|^2$
- good for classifying extreme cases by creating support vector margins
- linearly separable: Hard SVMs
- if not separable: Soft SVMs
  - use slack variable
  - $y_i(\omega\cdot x_i + b) \geq 1 - \varepsilon_i$, where $\varepsilon_i \geq 0$, an error occurs when $\varepsilon_i > 0$
  - Minimize: $f(w,b,\varepsilon) = \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n}\varepsilon_i$

Soft SVM: $\min \frac{1}{2}\|w\|^2 + C\sum\varepsilon_i$, s.t. $y_i(\omega_i^Tx_i) \geq 1 - \varepsilon_i$

if hard SVM: $\min \frac{1}{2}\|w\|^2$ s.t. $y_i(\omega_i^Tx_i) \geq 1$

Hinge loss = $\min \frac{1}{2}\|w\|^2 + C\sum\max(0, 1 - y_i\omega_i^Tx_i)$ (reg term; empirical loss)

Hinge loss: penalizes incorrectly classified examples
- not differentiable at x=1 ⇒ sub gradient
- $\forall x$: $C(x) - C(x_0) \geq V\cdot(x - x_0)$
- For hinge loss: $\{0,\ y_n(\omega\cdot x_n + b) > 1;\ y_nx_n,\ \text{o.w.}\}$

Gradient descent vs. Perceptron:

- Both aim to find optimal parameters for a model by minimizing misclassified data points
  - GD minimizes cost function
  - Perceptron updates weights for misclassified pts
- Both are iterative learning, supervised

Linearly separable data:
- GD:
  - converges smoothly
  - slower than perceptron
- Perceptron:
  - converges faster
  - requires fewer computations

Non-linearly separable data:
- GD:
  - can be done with SVM, hingloss
- Perceptron:
  - doesn't converge and oscillates

Multi class classification:

- So far, only binary predictions, but most scenarios irl are multiclass
- can reduce multiclass to binary

2 approaches:
1. combining multiple binary classifiers
   - one-vs-all
   - all-vs-all
2. Training a single classifier
   - extending SVM

b. Will the Classifier Fail if the Margin is Less than 1?

Yes, the classifier will fail if there is **no solution to the optimization problem** when the datas not linearly separable or the margin is less than 1. Here's why:

- The hard-margin formulation assumes **perfect linear separability**.
- If even a single data point violates the constraint $y_i(w^Tx_i + b) \geq 1$, the optimization problem becomes infeasible, and the model cannot be trained.

One-vs-all:
- each class can be separated from the rest using binary classifier
- K independent binary classifiers, one for each class

Multi-class perceptron:
- $\hat{y} = \arg\max_k w\cdot\Phi(x,k)$
- Update: $w \leftarrow w + \Phi(x,y) - \Phi(x,\hat{y})$

Multiclass SVM: minimize $\|w\|$ such that closest points to hyperplane have score ±1
- Binary SVM: minimize multiclass margin
- Multiclass SVM: each label has diff weight vector, maximize multiclass margin

Hard multiclass SVM:

$\min \frac{1}{2}\sum_k \omega_k^T\omega_k$, s.t. $\omega_{y_i}^Tx - \omega_k^Tx \geq 1$ (score of true label)

optimization problem: $f_k(\omega) = \omega_k\cdot x + b$, $f_k(x) > f_j(x)$ (correct classification)

$\min \frac{1}{2}\sum_k\|w_k\|^2$, s.t. $f_k(x_i) - f_j(x_i) \geq 1$

Soft multiclass SVM: $\min \frac{1}{2}\sum_k\omega_k^T\omega_k + C\sum\varepsilon_i$, s.t. $\omega_{y_i}^Tx - \omega_k^Tx \geq 1 - \varepsilon_i$

optimization problem: $\min \frac{1}{2}\sum_k\|\omega_k\|^2 + C\sum_i\sum_k\varepsilon_{i,k}$, s.t. $f_k(x_i) - f_j(x_i) \geq 1 - \varepsilon_{i,k}$

Cost-sensitive multiclass classification:
- hierarchy of costs/mistakes, some more tolerable than others
- incorporate distance metric into our learning model

$= \min \frac{1}{2}\|w\|^2 + \frac{C}{N}\sum_{n=1}^{N}\varepsilon_n$, s.t. $\langle\omega, \Phi(x_n, y_n)\rangle - \langle\omega\cdot\Phi(x_n, \hat{y})\rangle \geq \Delta(y_n, \hat{y}) - \varepsilon_n$ (True, pred, distance metric)

unconstrained version: $\min \frac{1}{2}\|w\|^2 + \frac{C}{N}\sum_{n=1}^{N} l(\omega; (x^n, y^n))$

$l(\omega; (x^n, y^n)) = \max\Delta(y_n, y) - \langle\omega, \Phi(x_n, y_n)\rangle + \langle\omega, \Phi(x_n, y)\rangle$

Neural Networks:
- useful for non-linear, complex classifiers
- jointly learn representation {multiple layers of linear functions are still linear}
  - neuron = linear classifier; merging multiple perceptrons
- very expressive (can represent any function); hidden layers = non-linear activation functions
- Sigmoid unit: $\frac{1}{1+e^{-z}} = \sigma(z)$
- ReLU: relu(z) = max(z, 0)
- 2-layered NN can represent any boolean function + bounded continuous functions
- learn parameters through back propagation

Forward computation:
- for output unit: $\hat{y}_k = \sigma(z_k) = \sigma(\sum\omega_j^kh_j)$
- for hidden unit: $h_j = \sigma(z_j) = \sigma(\sum\omega_{ij}x_i)$, multiply each input by weight and sum all
- we use low dim real vector for input. Values capture similarity in vector space
- hidden layers capture latent properties by combining multiple attributes of the prev layer

Building Neural Network:

Decide:
- How many input units, hidden layers, output units?

Named entity recognition: NER
- recognizes names, organizations, locations or none

- BOW: bag of words is aka 1-hot
  - each word represented by 1 in the corresponding index and 0 elsewhere
- Dense representations: each word represented by density/prob, helps reduce feature space

2. Overfitting and Underfitting in NN

- **Overfitting:**
  - Happens when the network is too large (too many layers or neurons) relative to the data.
  - The model memorizes training data, including noise, leading to poor performance on unseen data.
  - Common signs include a large gap between training and validation accuracy.
  - **Mitigation Strategies:**
    - Use regularization techniques (e.g., L1/L2 penalties, dropout).
    - Employ data augmentation.
    - Reduce network complexity or collect more data.
- **Underfitting:**
  - Happens when the network is too small or insufficiently trained.
  - The model cannot capture the complexity of the data, leading to poor performance even on the training set.
  - **Mitigation Strategies:**
    - Increase network capacity (more neurons/layers).
    - Train for more epochs.
    - Use advanced architect[illegible] or feature engineering.

1. Expressivity of Classifiers, Decision Boundaries, and Linear Separability

Concepts

- **Expressivity of Classifiers:**
  - The expressivity of a classifier refers to its ability to model and distinguish between different classes of data points.
  - **Linear Classifiers:** Represent decision boundaries as straight lines (or hyperplanes in higher dimensions). They work well for **linearly separable** data but fail for complex patterns.
  - **Non-Linear Classifiers:** Use transformations or more flexible models (e.g., kernels, deep networks) to handle **non-linearly separable** data.
- **Decision Boundaries:**
  - A decision boundary separates the feature space into regions corresponding to different predicted classes.
  - Linear classifiers (e.g., Logistic Regression, SVM with linear kernel) produce straight decision boundaries.
  - Non-linear classifiers (e.g., SVM with RBF kernel, neural networks) produce complex decision boundaries.
- **Linear Separability:**
  - A dataset is linearly separable if there exists a hyperplane that perfectly divides all data points into their correct classes.
  - In higher dimensions, datasets become increasingly likely to be linearly separable.

Leave-One-Out Cross Validation (L-O-O CV)

Definition:

L-O-O CV is a validation technique where:

1. A model is trained on $n-1$ data points, leaving one data point out as the test set.
2. This process is repeated for each of the n data points, and the model's performance is evaluated on the test point each time.
3. The overall accuracy is the average performance across all n test evaluations.

What is the leave 1 out CV accuracy of linear functions on the data set?
- 0

Define a new class of functions called hinges. A combination of two linear function that are connected at a point. Anything with their area is class1, outside class 2.
What is the leave 1 out CV accuracy of hinge functions on this data set?
- You can use a hinge to fit according to the data set, so you can't really be certain

$$\text{Regularization Term: } \lambda\sum_{n=1}^{N}\max(0, 1 - y_n\cdot f(x_n)),$$

Back Propagation: training neural network

= gradient descent + chain rule

- optimize weights to minimize objective function (→ loss function)

Chain Rule: $\frac{d}{dV_{wi}} f(z(V_{wi})) = \frac{df}{dz} \cdot \frac{dz}{dV_{wi}}$

- prediction no longer linear
- early stopping, regularization = prevent overfitting
- No guarantee of convergence; many reach local minimum
  - to avoid local minima: several trials w/ diff random initial weights
- many epochs for adequate training (10ks)
- Terminative criteria = # of epochs, threshold on train set error, no decrease in error
  - too much train = overfit; test valid accuracy after every epoch

Overfitting: non-linear error surface
- regularization, adding noise, data augmentation, dropout

Many hidden units = overfitting: cross validation technique for # of hidden units

Dropout: randomly deactivates neurons during training

NN can learn feature representation

Recurrent neural network:
- NN version of a language model
- unlike n-gram, RNN conditions current word on all previous words
- Efficient time + space

Input vector: $x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n$

$h_i = \sigma(W_{hh} h_{i-1} + W_{hx} x_i)$

$\hat{y} = \text{softmax}(W_s h_i)$

$P(x_{i+1} = V_j \mid x_i, \ldots, x_1) = \hat{y}_{i,j}$

- We define prob. dist. over V

Loss: $J_{(t)}(\theta) = -\sum_{j=1}^{|V|} y_{t,j} \log \hat{y}_{t,j}$

- **Representation Learning:**
  - Enables models to automatically extract useful features from raw data.
  - Examples: Embeddings in natural language processing (e.g., Word2Vec, BERT) or convolutional layers in image classification.

Mathematics:

- Loss Function for Binary Classification:

$$L = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log \hat{y}_i + (1-y_i)\log(1-\hat{y}_i)]$$

where $\hat{y}_i = \sigma(\mathbf{w}^T\mathbf{x}_i + b)$, and σ is the sigmoid function.

- Chain Rule in Backpropagation:
  - For a neural network layer:

$$\frac{\partial L}{\partial \mathbf{w}_l} = \frac{\partial L}{\partial \mathbf{z}_l}\frac{\partial \mathbf{z}_l}{\partial \mathbf{w}_l}$$

3. Deep Learning, Backpropagation, and Representation Learning

Concepts

- **Deep Learning:**
  - A subfield of machine learning focused on learning hierarchical representations of data using deep neural networks (DNNs).
- **Backpropagation:**
  - A gradient-based optimization algorithm for training neural networks.
  - **Steps:**
    1. **Forward Pass:** Compute predictions and loss.
    2. **Backward Pass:** Compute gradients of the loss with respect to parameters using the chain rule.
    3. **Update Parameters:** Use gradient descent to update weights:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\frac{\partial L}{\partial \mathbf{w}}$$

where η is the learning rate and L is the loss function.

General Guide for Forward Pass

1. Inputs to the Network

- Identify the input features $(x_1, x_2, \ldots, x_n)$.
- If there are multiple layers, treat the outputs of the previous layer as inputs for the nex

2. Weighted Sum for Each Neuron

For a given neuron j in a layer:

$$z_j = \sum_{i=1}^{n} w_{ij}x_i + b_j$$

- w_{ij}: Weight connecting input x_i to neuron j.
- b_j: Bias of the neuron.
- x_i: Input to the neuron (can be raw inputs or outputs from the previous layer).

3. Activation Function

Apply the chosen activation function $f(z_j)$ to the weighted sum:

$$a_j = f(z_j)$$

- Common activation functions:
  - **Sigmoid:** $f(z) = \frac{1}{1+e^{-z}}$
  - **ReLU:** $f(z) = \max(0, z)$
  - **Tanh:** $f(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

4. Output Layer

- Use the same procedure as above to compute the final outputs. If it's a regression problem, the output might be a raw value. For classification:
  - **Binary Classification:** Use sigmoid activation.
  - **Multiclass Classification:** Use softmax activation:

$$\text{Softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

5. Loss Function

Compute the loss for a given target y:

- Common loss functions:
  - **Mean Squared Error (MSE):**

$$L = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

  - **Binary Cross-Entropy:**

$$L = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$$

General Guide for Backward Pass

1. Start with the Loss

Compute the derivative of the loss with respect to the output ($\hat{y}$):

$$\frac{\partial L}{\partial \hat{y}}$$

2. Backpropagate Through the Output Layer

For the output layer:

1. Compute the gradient of the activation function:

$$\frac{\partial \hat{y}}{\partial z_j} = f'(z_j)$$

- For sigmoid:

$$f'(z) = \sigma(z)(1 - \sigma(z))$$

- For ReLU:

$$f'(z) = 1 \text{ if } z > 0, \text{ else } 0$$

2. Combine with the gradient from the loss:

$$\frac{\partial L}{\partial z_j} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_j}$$

3. Compute gradients for weights and biases:

- Weight gradient:

$$\frac{\partial L}{\partial w_{ij}} = \frac{\partial L}{\partial z_j} \cdot a_i$$

- Bias gradient:

$$\frac{\partial L}{\partial b_j} = \frac{\partial L}{\partial z_j}$$

3. Backpropagate Through Hidden Layers

For each hidden layer:

1. Compute the gradient for the layer's output (a_j) using the chain rule:

$$\frac{\partial L}{\partial a_j} = \sum_k \frac{\partial L}{\partial z_k} \cdot w_{jk}$$

where w_{jk} are the weights connecting neuron j in the hidden layer to neuron k in the next layer.

2. Compute the gradient for the weighted sum:

$$\frac{\partial L}{\partial z_j} = \frac{\partial L}{\partial a_j} \cdot f'(z_j)$$

3. Compute gradients for weights and biases as in the output layer.

4. Update Weights and Biases

Using gradient descent:

$$w \leftarrow w - \eta \cdot \frac{\partial L}{\partial w}$$

$$b \leftarrow b - \eta \cdot \frac{\partial L}{\partial b}$$

- η: Learning rate (controls step size).

Step 1: Compute Hidden Layer Activations (z_1, z_2)

For each hidden unit z_1, z_2:

- $z_1 = \sigma(\alpha_{11}x_1 + \alpha_{12}x_2 + \alpha_{13}x_3)$:

$$z_1 = \sigma((0.5)(1) + (-0.4)(2) + (0.3)(3))$$

$$z_1 = \sigma(0.5 - 0.8 + 0.9) = \sigma(0.6) = \frac{1}{1+\exp(-0.6)} \approx 0.645$$

- $z_2 = \sigma(\alpha_{21}x_1 + \alpha_{22}x_2 + \alpha_{23}x_3)$:

$$z_2 = \sigma((-0.2)(1) + (0.1)(2) + (0.6)(3))$$

$$z_2 = \sigma(-0.2 + 0.2 + 1.8) = \sigma(1.8) = \frac{1}{1+\exp(-1.8)} \approx 0.858$$

Thus:

$$z_1 \approx 0.645, \quad z_2 \approx 0.858$$

Step 2: Compute Output (y)

Using the output weights:

- $y = \sigma(\beta_1 z_1 + \beta_2 z_2)$:

$$y = \sigma((0.7)(0.645) + (-0.5)(0.858))$$

$$y = \sigma(0.4515 - 0.429) = \sigma(0.0225)$$

$$y = \frac{1}{1+\exp(-0.0225)} \approx 0.506$$

Example General Procedure

1. **Forward Pass:**
   - Compute z_j for each neuron using inputs, weights, and biases.
   - Apply activation functions to get a_j.
   - Compute final output ($\hat{y}$).
   - Compute loss using the loss function.
2. **Backward Pass:**
   - Start with the gradient of the loss w.r.t output ($\hat{y}$).
   - Backpropagate through the output layer to compute gradients for z_j, w_{ij}, and b_j.
   - Backpropagate through all hidden layers in reverse order.
   - Update weights and biases using the computed gradients.

Representation Learning

What is Representation Learning?

- Representation learning refers to **automatically learning features** or representations from raw data that are useful for downstream tasks.
- It reduces the need for manual feature engineering and finds patterns in data that are often not apparent.

Key Concepts

1. **Hierarchical Features:**
   - Learns simple features first (e.g., edges in images) and combines them to form complex representations (e.g., faces).
   - **Deep learning models (e.g., CNNs, RNNs)** excel in hierarchical representation learning.
2. **Latent Representations:**
   - Encodes raw data into a lower-dimensional space while retaining meaningful information (e.g., embeddings in NLP).
3. **Transferability:**
   - Learned representations can be used for different tasks (e.g., pre-trained BERT for various NLP tasks).

What You Should Know

1. **Common Methods:**
   - Supervised: CNNs, RNNs, Transformers.
   - Unsupervised: Autoencoders, PCA, t-SNE.
   - Self-Supervised: Contrastive Learning (SimCLR, BYOL).
2. **Applications:**
   - NLP (word/sentence embeddings), computer vision (object detection), audio processing, and more.
3. **Strengths:**
   - Reduces reliance on domain expertise.
   - Handles high-dimensional data effectively.
4. **Challenges:**
   - Interpretability of learned features.
   - Computationally intensive for deep learning models.

Probabilistic clustering:
- model provides full distributional description
- soft clustering (k-means = hard)
- key cost: assumption of parametric model

Mixture models:
- knowledge: parametric
- score = likelihood
- search: EM

Key Notes

- **Activation Function and Derivative:** Always know the activation function for each layer and its derivative.
- **Chain Rule:** Backpropagation heavily relies on applying the chain rule to compute gradients layer by layer.
- **Learning Rate (η):** Ensure a proper learning rate to avoid overshooting or slow convergence.

Unsupervised Learning
- simplify data, identify patterns, learn meaningful representations, identify outliers?
- Descriptive models summarize; cluster analysis, density estimation

Cluster Analysis:
- partition data into groups
- inter-group similarity low, intra-group similarity high

Good cluster = well-separated clusters, center based (closer to own center) clusters, contiguity based (closer to own pts than in other clusters) clusters, density based, conceptual

Clustering Algorithms:
- partition based models
- Hierarchical clustering: agglomerative (bottom up) (merge clusters), decisive (top-down) (split clusters)

Partition-based clustering: Ex: K-means
- data → K clusters; Evaluation: score(C, D) maximized/minimized
- most approaches used iterative improvement algorithms

Cluster score functions: score(C, D) = measures quality of clustering C for dataset D $= f(wc(C), bc(C))$
- compact clusters: minimize within cluster distance (wc)
- separated clusters: maximize between cluster distance (bc)

Cluster centroid: $r_k = \frac{1}{n_k}\sum_{x(i)} x(i)$

Between-cluster dist: $bc(C) = \sum d(r_j, r_k)^2$

within-cluster dist: $wc(C) = \sum_k wc(C_k) = \sum_k \sum_{x(i)} d(x(i), r_k)^2$

K-means:
1) select K centroids randomly
2) Assign each data point to nearest centroid
3) update centroids by calc mean of all points in each cluster
4) Repeat steps 2-3 till centroids are stable

Terminates? Yes, objective func converges. Converges to optimal? No, terminates at local optima. $O(k \cdot n \cdot i)$; i = iterations

- knowledge rep = k clusters are defined by canonical members (centroids)
- Model space = all possible partitions of the examples into K groups
- score: minimize within cluster distance; search: iterative greedy

- strengths: efficient, finds spherical clusters
- weakness: terminates at local minima, sensitive to outliers, specify k, applicable only when mean defined
- Variations: selection of initial centroids; algo modifications allow for merge, split of centroids
  - run w/ multiple random selections, pick one w/ best score

Probability mixture model:

$f(x) = \sum_k w_k f_k(x; \theta)$ → prob of observing x

Gaussian-Mixture Models: assume data is generated from mixture of gaussian d
- uses expectation-maximization (EM) to estimate parameters

$p(x) = \sum_k p(k)P(x|k) = \sum P(k)P(x \mid x \sim N(\mu_k, \Sigma_k))$

Pros: soft clustering; cons: computationally intensive

Expectation-maximization (EM) algorithm: algo for parameter estimation in data with hidden/unobserved values.
- hidden vars = cluster membership; generalization of k-means for probabilistic models

Algorithm:
1) Initialize hidden variables and parameters
2) E-step: predict values for each hidden variable given current parameters
   - compute probs of each datapoint belonging to each cluster
3) M-step: estimate parameters given current prediction for hidden vars.
   - update model parameters (eg. cluster means) to maximize likelihood.

Hidden cluster membership values:
- for dataset n, each example x_n is assumed to belong to one of K clusters
- Cluster indicator vars: $z_{n1}, \ldots, z_{nK}$ where $z_{nk} = 1$ if x_n belongs to cluster k and 0 o.w.
  - these vars are hidden because true cluster assignments are unknown during training.

if z was known: $\log P(X, Z|\theta) = \sum_n^N \sum_k^K z_{nk}\log P(x_n|\theta_k)$

Solution: use expected values of hidden vars.

$\log P(X, Z|\theta) = \sum^N \sum^K E[z_{nk}]\log P(X_n|\theta_k)$ = M-step ②

$E[z_{nk}] = P(z_{nk} = 1 | x_n, \theta) = \frac{P(X_n|\theta_k)P(\theta_k)}{\sum_j P(X_n|\theta_j)P(\theta_j)}$ = E-step ①

if Gaussian: $\sum^N \sum^K E[z_{nk}][\log w_k + \log N(x_n|\theta_k, \Sigma_k)]$

EM for GMM:
- E-step: $\Gamma(x_n) = [E[z_{1n}], \ldots, E[z_{Kn}]]$
- M-step: $\log P(x, z|\theta) = \sum^n \sum^k E[z_{nk}][\log w_k + \log N(x_n|\theta_k, \Sigma_k)]$

$N_k = \sum_i E[z_{ik}]$: total weight assigned to cluster k

$w_k = \frac{N_k}{N}$: normalize, $\Sigma_k = \frac{1}{N_k}\sum E[z_{ik}](x_i - \theta_k)(x_i - \theta_k)^T$: weighted covariance

$\theta_k = \frac{1}{N_k}\sum_i E[x_{ik}]x_i$ → weighted mean

- Often both E and M can be solved in closed form
- Neither E or M can decrease likelihood
- Algo converges local max of likelihood
- Must specify initialization, stopping criteria

Hierarchical Methods:
- constructs a hierarchy of nested clusters rather than picking k beforehand
- Approaches: agglomerative, divisive

Agglomerative:
```
for i to n:
  C_i = {x(i)}
while |C| > 1:
  let C_i and C_j be pair of clusters with min D(C_i, C_j)
  C_i = C_i ∪ C_j
  remove C_j
```

Brown clusters:
- hierarchical clustering of words based on their context
- clusters correspond to topically related words
- Bit-string encoding: capture tree path from root to word

Distance between clusters:
- Single-link / nearest neighbor:
  $D(C_i, C_j) = \min\{d(x,y) \mid x \in C_i, y \in C_j\}$
  → can produce long, thin clusters
- complete link / furthest neighbor:
  $D(C_i, C_j) = \max\{d(x,y) \mid x \in C_i, y \in C_j\}$
  → sensitive to outliers
- Average link:
  $D(C_i, C_j) = \text{avg}\{d(x,y) \mid x \in C_i, y \in C_j\}$
  → compromise between the 2

Model Evaluation:

cluster validity:
- For prediction tasks = accuracy, squared loss, etc
- For cluster analysis: "goodness" of resulting clusters

Approaches:
- determine clustering tendency of the data
- Evaluate clustering using known class labels
- evaluate how well clusters fit data
- determine which of 2 diff cluster results better
- determine correct # of clusters

Clustering tendency:
- evaluate if dataset has clusters before clustering
- Most common approach:
  - statistical test for randomness
  - Hopkins statistic: Sample 20 pts from dataset; Generate 20 random pts in same space

w_i = dist from random pt to NN in data

u_i = dist from sample pt to NN in data

$H = \frac{\sum w_i}{\sum w_i + \sum u_i}$ — 0.5 = random data, 1.0 = high cluster, 0 = unifo

c. DBSCAN (Density-Based Spatial Clustering of Applications with Noise)
- How It Works:
  - Groups points that are close together based on a density threshold.
  - Identifies noise and outliers as standalone points.
- **Pros:** Handles noise and works well with non-spherical clusters.
- **Cons:** Requires tuning density parameters (e.g., ϵ).

Supervised clustering evaluation:
- measures extent to which clusters match label values

① Classified-oriented:
- Purity: measure of the extent to which clusters (G) contain objects of a particular class (C)

  $\text{purity}(C, G) = \frac{1}{N}\sum_i^K \max_j N_j \in G_i$

  - High purity: easy to achieve if # of clusters is large
- Entropy: degree to which each cluster contains objects of a single class.

  $\text{entropy}(C, G) = \sum_{i=1}^{K} - \sum_{j=1}^{C} P_{i,j}\log(P_{i,j})$

- Normalized mutual info gain (NMI): measures amount of info by which our knowledge about the classes (C) ↑ when we are told what the clusters are

  $NMI(C, G) = \frac{I(C,G)}{H(C)+H(G)} = \frac{\sum_c \sum_g P(c,g)\log\frac{P(c,g)}{P(c)P(g)}}{-\sum_c P(c)\log P(c) - \sum_g P(g)\log P(g)}$

  - between 0 (min) and 1 (max)
  - denominator (normalization) adjusts for prob that entropy tends to increase w/ # of clusters

② Similarity-oriented: based that any pair of object in same cluster should have same class.
- construct similarity matrix based on cluster membership and class values. Then compare 2 matrices.

  cluster matrix: 1 if i, j are in same cluster, 0 if o.w.
  class matrix: 1 if i, j are in same class, 0 if o.w. } ideal matrices
- compare correlation between the 2 matrices

Rand $= \frac{f_{00} + f_{11}}{f_{00} + f_{01} + f_{10} + f_{11}}$ Jaccard $= \frac{f_{11}}{f_{01} + f_{10} + f_{11}}$

Score functions for K:
- plot objective func (within cluster SSE) and look for elbow
- choose k to maximize likelihood
  - As k increases, value of max likelihood cannot decrease
  - ∴ more complex models = higher k
- describe data as precisely as possible + generalize to new data

Penalized score functions: include term that reflects how well the model fits and the data and another (penalty) term to value the simplicity of the model.
- score(θ, M) = error(M) + penalty(M)
  - penalty depends on # of parameters in model (p) and # of data points
  - error based on likelihood

AIC: Akaike info criterion $= -2\log L + 2p$

BIC: Bayesian info criterion $= -2\log L + p\log n$

Unsupervised clustering evaluation:
- visual inspection; order proximity matrix w.r.t class labels
- correlation: ideal similarity cluster matrix
  - correlation between initial similarity matrix and ideal
    High: points in same cluster are close to each other
    NB: assumes prox. values are [0,1]

Cohesion: measures how closely related objects are within cluster
- within cluster sum of squares (SSE)
  - for each pt, the error is the dist. to the centroid
- within cluster pairwise weighting
  - Sum dist between all pairs of pts. in same cluster

Separation: measures how distinct a cluster is from other clusters
  Between cluster SSE
  - error is dist from centroids c to c'

Between cluster pairwise weighting
- sum distance between all pair of pts. in diff clusters

Silhouette coefficient: cohesion + separation

A: avg dist of i to points in same cluster

B: avg dist of i to points in other clusters

$S = \frac{B - A}{\max(A, B)}$; closer to 1 is better

2. Clustering Evaluation

- **Internal Metrics:**
  - **Silhouette Score:** Measures how similar a point is to its cluster compared to others. Range: -1 (bad clustering) to 1 (perfect clustering).
  - **Inertia (SSE):** Sum of squared distances of points to their cluster center. Lower is better.
- **External Metrics (if ground truth exists):**
  - **Adjusted Rand Index (ARI):** Measures the similarity between predicted clusters and true labels.
  - **Normalized Mutual Information (NMI):** Measures shared information between clusters and ground truth.

3. Clustering Analysis

- **Visualization:** Use t-SNE or PCA to reduce dimensions and visualize clusters.
- **Cluster Validation:** Compare results across multiple algorithms and hyperparameter settings.