CSI 4106

Artificial Intelligence

Study Guide

Fall 2024

Types of Learning

- Supervised Learning: Each example is accompanied by a label
- Unsupervised Learning: No feedback is provided to the algorithm
- Reinforcement Learning: The algorithm receives a reward or a punishment following each action

Learning Phases

- Learning (building a model)
- Inference (using the model)

For learning, we first obtain training data, then we convert the data into features vectors. Then we use an algorithm to train a model.

For inference, we use new data to make new predictions with the model.

Supervised Learning

The dataset is a collection of labelled examples

- $\{(x_i, y_i)\}_{i=1}^{N}$
  - Each x_i is a feature vector with D dimensions
  - $x_i^{(j)}$ is the value of the feature j of the example i, for $j \in 1,\dots,D$ and $i \in 1,\dots,N$
  - The label y_i is either a class, taken from a finite list of classes, $\{1,2,\dots,C\}$, or a real number, or a complex object
- When the label y_i is a class, taken from a finite list of classes, $\{1,2,\dots,C\}$, we call the task a classification task
- When the label y_i is a real number, we call the task a regression task

Regression

- Training data is a collection of labelled examples
  - $\{(x_i, y_i)\}_{i=1}^{N}$
    - Each x_i is a feature vector with D dimensions
    - $x_i^{(j)}$ value of feature j of example i, for $j \in 1,\dots,D$ and $i \in 1,\dots,N$
  - Label y_i is a real number
- Problem: Given the data set as input, create a model that can be used to predict the value of y for an unseen x

Linear Regression

- A linear model assumes that the value of the label, $\hat{y}_i$, can be expressed as a linear combination of the feature values $x_i^{(j)}$:

$$\hat{y}_i = h(x_i) = \theta_0 + \theta_1 x_i^{(1)} + \theta_2 x_i^{(2)} + \dots + \theta_D x_i^{(D)}$$

- Here, θ_j is the j^{th} parameter of the (linear) model, with θ_0 being the bias term/parameter, and $\theta_1,\dots,\theta_D$ being the feature weights
- Problem: find values for all the model parameters so that the model "best fits" the training data
- Root mean square error is a common performance measure for regression problems

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(h(x_i)-y_i)^2}$$

Building Blocks of Learning Algorithms

- A typical learning algorithm comprises the following components:
  - A model, often consisting of a set of weights whose values will be "learnt"
  - An objective function
  - Optimization algorithm

Optimization

- Until some termination criteria is met:
  - Evaluate the loss function, comparing $h(x_i)$ to y_i
  - Make small changes to the weights, in a way that reduces the value of the loss function

Gradient Descent (Single value)

- A common loss function for regression problems is the root mean squared error

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(h(x_i)-y_i)^2}$$

- In practice, minimizing the mean squared error is easier and gives the same result

$$\frac{1}{N}\sum_{i=1}^{N}(h(x_i)-y_i)^2$$

- Suppose we have a linear model

$$h(x_i) = \theta_0 + \theta_1 x_i$$

and the loss function

$$J(\theta_0,\theta_1) = \frac{1}{N}\sum_{i=1}^{N}(h(x_i)-y_i)^2$$

We want to find θ_0 and θ_1 that minimizes J

- We run gradient descent
  - Initialization: θ_0 and θ_1 with random values or zeros
  - Loop:
    - Repeat until convergence:

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0,\theta_1)$$

for $j = 0, 1$

  - α is called the learning rate - size of each step
  - $\frac{\partial}{\partial\theta_j}J(\theta_0,\theta_1)$ is partial derivative w.r.t. θ_j

We would have

$$J(\theta_0,\theta_1) = \frac{1}{N}\sum_{i=1}^{N}(\theta_0+\theta_1 x_i - y_i)^2$$

$$\frac{\partial}{\partial\theta_0}J(\theta_0,\theta_1) = \frac{2}{N}\sum_{i=1}^{N}(\theta_0+\theta_1 x_i - y_i)$$

$$\frac{\partial}{\partial\theta_1}J(\theta_0,\theta_1) = \frac{2}{N}\sum_{i=1}^{N}x_i(\theta_0+\theta_1 x_i - y_i)$$

Multivariate Gradient Descent

- A multivariate linear regression model is defined as

$$h(x_i) = \theta_0 + \theta_1 x_i^{(1)} + \dots + \theta_D x_i^{(D)}$$

$x_i^{(j)}$ = value of feature j in i^{th} example

D = number of features

- The new loss function is

$$J(\theta_0,\theta_1,\dots,\theta_D) = \frac{1}{N}\sum_{i=1}^{N}(h(x_i)-y_i)^2$$

Its partial derivative

$$\frac{\partial}{\partial\theta_j}J(\theta) = \frac{2}{N}\sum_{i=1}^{N}x_i^{(j)}(\theta x_i - y_i)$$

where θ, x_i and y_i are vectors, and θx_i is a vector operation

- The vector containing the partial derivatives of J is called the gradient vector

$$\nabla_\theta J(\theta) = \begin{pmatrix}\frac{\partial}{\partial\theta_0}J(\theta)\\ \frac{\partial}{\partial\theta_1}J(\theta)\\ \vdots \\ \frac{\partial}{\partial\theta_D}J(\theta)\end{pmatrix}$$

- This vector gives the direction of the steepest ascent
- It gives its name to the gradient descent algorithm

$$\theta' = \theta - \alpha\nabla_\theta J(\theta)$$

- The algorithm is as follows:

Repeat until convergence:

{

$$\theta_j := \theta_j - \alpha\frac{\partial}{\partial\theta_j}J(\theta_0,\theta_1,\dots,\theta_D)$$

for $j \in [0,\dots,D]$ (update simultaneously)

}

- We assume that the objective function is differentiable

Local vs. Global minimum

- A function is convex if for every pair of points on the graph of the function, the line connecting these two points lies above or on the graph
  - A convex function has a single minimum
  - The loss function for the linear regression (MSE) is convex
- For functions that are not convex, the gradient descent algorithm converges to a local minimum
- The loss function generally used with linear or logistic regressions, and SVMs are convex, but not the ones for ANNs

Learning Rate

- Small steps, low values of α, will make the algorithm converge slowly
- Large steps might cause the algorithm to diverge
- Notice how the algorithm slows down naturally when approaching a minimum

Batch Gradient Descent

- This algorithm is known as batch gradient descent since for each iteration, it processes the "whole batch" of training examples
- This algorithm might take more time to converge if the features are on different scales
- The batch gradient descent becomes very slow as the number of training examples increases
- This is because all the training data is seen at each iteration

Stochastic Gradient Descent

- The stochastic gradient descent algorithm randomly selects one training instance to calculate its gradient
- This allows it to work with large training sets
- Its trajectory is not as regular as the batch algorithm
  - Because of its bumpy trajectory, it is often better at finding the global minima when compared to batch
  - Its bumpy trajectory makes it bounce around the local minimum

Mini-batch Gradient Descent

- At each step, rather than selecting one training example as SGD does, mini-batch gradient descent randomly selects a small number of training examples to compute the gradients
- Its trajectory is more regular compared to SGD
  - As the size of the mini-batches increases, the algorithm becomes increasingly similar to batch gradient descent, which uses all the examples at each step
- It can take advantage of the hardware acceleration of matrix operation, particularly with GPUs

Classification

- Binary classification is a supervised learning task where the objective is to categorize instances (examples) into one of two discrete classes
- A multi-class classification task is a task of supervised learning problem where the objective is to categorize instances into one of three or more discrete classes
- Some machine learning algorithms are specifically designed to solve binary classification problems
  - Logistic regression and SVM are such examples
- Any multi-class classification problem can be transformed into a binary classification problem
- One-vs-All
  - A separate binary classifier is trained for each class
  - For each classifier, one class is treated as the positive class, and all other classes are treated as the negative class
  - The final assignment of a class label is made based on the classifier that outputs the highest confidence score for a given input

Decision Boundary

- A decision boundary is a boundary that partitions the underlying feature space into regions corresponding to different class labels
- The data is linearly separable when two classes of data can be perfectly separated by a single linear boundary, such as a line in two-dimensional space or a hyperplane in higher dimensions
- 2 attributes, linear decision boundary would be a line
- 2 attributes, non-linear decision boundary would be a non-linear curve
- 3 attributes, linear decision boundary would be a plane
- 3 attributes, non-linear decision boundary would be a non-linear surface
- >3 attributes, linear decision boundary would be a hyperplane
- >3 attributes, non-linear decision boundary would be a hypersurface
- Revised definition: a decision boundary is a hypersurface that partitions the underlying feature space into regions corresponding to different class labels

Logistic Regression

- Despite its name, logistic regression serves as a classification algorithm rather than a regression technique
- The labels in logistic regression are binary values, denoted as $y_i \in \{0,1\}$, making it a binary classification task
- The primary objective of logistic regression is to determine the probability that a given instance x_i belongs to the positive class, i.e. $y_i = 1$
- The standard logistic function is defined as $\sigma: \mathbb{R} \to (0,1)$

$$\sigma(t) = \frac{1}{1+e^{-t}}$$

- An S-shaped curve, such as the standard logistic function, is termed a squashing function because it maps a wide input domain to a constrained output range.
- Analogous to linear regression, logistic regression computes a weighted sum of the weighted features, expressed as:

$$\theta_0 + \theta_1 x_i^{(1)} + \theta_2 x_i^{(2)} + \dots + \theta_D x_i^{(D)}$$

- However, using the standard logistic function limits its output to the range (0,1):

$$\sigma(\theta_0 + \theta_1 x_i^{(1)} + \theta_2 x_i^{(2)} + \dots + \theta_D x_i^{(D)})$$

- The logistic regression model, in its vectorized form, is defined as:

$$h_\theta(x_i) = \sigma(\theta x_i) = \frac{1}{1+e^{-\theta x_i}}$$

- In logistic regression, the probability of correctly classifying an example increases as its distance from the decision boundary increases
- The principle holds for both positive and negative classes
- An example lying on the decision boundary has a 50% probability of belonging to either class

- Predictions are made as follows:
  - $y_i = 0$, if $h_\theta(x_i) < 0.5$
  - $y_i = 1$, if $h_\theta(x_i) \geq 0.5$
- The values of θ are learned using gradient descent

Feature Engineering

- The process of creating, transforming, and selecting variables from raw data to improve the performance of machine learning models

Underfitting and Overfitting

- Underfitting
  - Model is too simple
  - Uninformative features
  - Poor performance on both train and test data
- Overfitting
  - Model is too complex
  - Too many features
  - Excellent performance on training set, but poor performance on test set

Learning Curves

- We can visualize learning curves to assess our models
  - A learning curve shows the performance of the model on both training and test set
  - Multiple measurements are obtained by repeatedly training the model on larger and larger subsets of the data

Bias / Variance Tradeoff

- Bias is the error from overly simplistic models. High bias can lead to underfitting
- Variance is the error from overly complex models. High variance can lead to overfitting
- Tradeoff: aim for a model that generalizes well to new data

Confusion Matrix

	Positive (predicted)	Negative (predicted)
Positive (Actual)	True Positive (TP)	False Negative (FN)
Negative (Actual)	False Positive (FP)	True Negative (TN)

Accuracy

- The accuracy of a model measures how accurate the result is
- Ratio of correct number of predictions to total number of predictions

$$\text{accuracy} = \frac{TP+TN}{TP+TN+FP+FN} = \frac{TP+TN}{N}$$

- Accuracy can be misleading in the case of class imbalance, as it disproportionally reflects the performance on the majority class, masking poor performance on the minority class

Precision

- Also known as positive predictive value
- Proportion of true positive predictions among all positive predictions

$$\text{precision} = \frac{TP}{TP+FP}$$

- An algorithm that makes a small number of high-confidence predictions might achieve a high precision score, but may not be helpful

Recall

- Also known as sensitivity or true positive rate (TPR)

$$\text{recall} = \frac{TP}{TP+FN}$$

- Proportion of true positive instances correctly identified among all actual positive instances

F_1 Score

- Harmonic mean of precision and recall

$$F_1 \text{ score} = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}} = 2 \times \frac{\text{Precision} \cdot \text{recall}}{\text{Precision} + \text{recall}} = \frac{TP}{FP + \frac{FN+FP}{2}}$$

Micro Performance Metrics

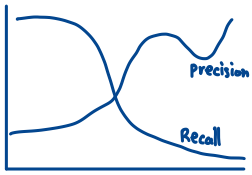
- Micro performance metrics aggregate the contributions of all classes to compute the average performance metric, such as precision, recall, or F_1 score.
- This approach treats each individual prediction equally, providing a balanced evaluation by emphasizing the performance on frequent classes

Macro Performance Metrics

- Compute the performance metric independently for each class and then average these metrics.
- Treats each class equally, regardless of frequency

Precision-Recall Tradeoff

- As the decision threshold decreases, a higher number of examples are predicted positive, potentially leading the classifier to eventually label all instances as positive
- Conversely, as decision threshold increases, fewer examples are classified as positive, which may result in the classifier predicting no positive instances at all



Threshold

ROC Curve

Receiver Operating Characteristic curve

- Curve shows the true positive rate against false positive rate
- An ideal classifier has TPR close to 1.0 and FPR close to 0.0
- $TPR = \frac{TP}{TP+FN}$
- TPR approaches one when the number of false negative prediction is low
- $FPR = \frac{FP}{FP+TN}$
- FPR approaches zero when the number of false positive is low

Training and Test Set Evaluation

- Guideline: Typically, allocate 80% of dataset for training and reserve 20% for testing
- Training Set: Utilized to train model
- Test Set: Independent subset used exclusively at the final stage to assess the model's performance

Errors

- Training Error:
  - Generally tends to be low
  - Achieved by optimizing learning algorithms to minimize errors through parameter adjustments
- Generalization Error: Error rate observed when the model is evaluated on new, unseen data
- Underfitting:
  - High training error
  - Model is too simple to capture underlying patterns
  - Poor performance on both training and new data
- Overfitting:
  - Low training error, but high generalization error
  - Model captures noise or irrelevant patterns
  - Poor performance on new, unseen data

Cross-Validation

- A method used to evaluate and improve the performance of machine learning models
- It involves partitioning the dataset into multiple subsets, training the model on some subsets while validating it on the remaining ones

k-Fold Cross-Validation

1. Divide the dataset into k equally sized parts (folds)
2. Training and validation
   - For each iteration, one fold is used as the validation set, the remaining k-1 folds are used as the training set
3. Evaluation: The model's performance is evaluated in each iteration, resulting in k performance measures
4. Aggregation: Statistics are calculated based on k performance measures

Properties of Cross-Validation

- More reliable:
  - More reliable estimate of model performance compared to a single train-test split
  - Reduces the variability associated with a single split, leading to a more stable and unbiased evaluation
  - For large values of k, consider the average, variance, and confidence interval
- Better Generalized
  - Helps in assessing how the model generalizes to an independent dataset
  - It ensures that the model's performance is not overly optimistic or pessimistic by averaging results over multiple folds
- Efficient Use of Data
  - Particularly beneficial for small datasets, cross-validation ensures that every data point is used for both training and validation
  - This maximizes the use of available data, leading to more accurate and reliable model training
- Hyperparameter Tuning
  - Commonly used during hyperparameter tuning, allowing for the selection of the best model parameters based on their performance across multiple folds
  - This helps in identifying the optimal configuration that balances bias and variance
- Challenges
  - Computational Cost: Requires multiple model trainings
    - Leave-One-Out (LOO): Extreme case k=N
  - Class Imbalance: Folds may not represent minority classes
    - Use Stratified Cross-Validation to maintain class proportions
  - Complexity: Error-prone implementation, especially for nested cross-validation, bootstraps, or integration into larger pipelines

Hyperparameter Tuning

- A hyperparameter is a configuration external to the model that is set prior to the training process and governs the learning process, influencing model performance and complexity
- Hyperparameters: Decision Tree
  - criterion: gini, entropy, log_loss, measure the quality of a split
  - max_depth: limits the number of levels in the tree to prevent overfitting
- Hyperparameters: Logistic Regression
  - penalty: l1 or l2, helps in preventing overfitting
  - solver: liblinear, newton-cg, lbfgs, sag, saga
  - max_iter: maximum number of iterations taken for the solver to converge
  - tol: stopping criteria, smaller values mean higher precision
- Hyperparameters: kNN
  - n_neighbors: number of neighbors to use for k-neighbors queries
  - weights: uniform or distance, equal weight or distance-based weight

Hyperparameter Tuning: Grid Search

- The major disadvantage of ML algorithms is that many hyperparameters need tuning
- Manual exploration of combinations is tedious
- Grid search is more systematic
  1. Enumerate all possible hyperparameter combinations
  2. Train on training set, evaluate on validation set

Randomized Search

- Large number of combinations (many hyperparameters, many values)
- Randomized Search CV:
  - Supply list of values or probability distribution for hyperparameters
  - Specify number of iterations
  - Predictable execution time

Data Augmentation

- A technique used to increase the diversity of a dataset by applying various transformations to the existing data
- Purpose: Enhance the robustness and generalization capability of machine learning models

Machine Learning Engineering

1. Gather adequate data
2. Extract features from the raw data
   - This process is labor-intensive
   - It necessitates creativity
   - Domain knowledge is highly beneficial

One-Hot Encoding

- Some learning algorithms require data to be in numerical form
- We want to find the best encoding method
- One-Hot Encoding: should be preferred for categorical data
  - Increases Dimensionality: one-hot encoding increases the dimensionality of feature vectors
  - Avoids Bias: Other encoding methods can introduce biases
- One-Hot Encoding is a technique that converts categorical variables into a binary vector representation where each category is represented with a single 1 and all other elements as 0

Ordinal Encoding

- A technique that assigns numerical values to categorical attributes based on their inherent order or rank
- The desired order of categories must be provided to the encoder, otherwise, it defaults to alphabetical order

Discretization

- Grouping ordinal values into discrete categories
- Binning, bucketing, quantization
- Advantage:
  - Enables the algorithm to learn effectively with fewer training examples
- Disadvantages:
  - Requires domain expertise to define meaningful categories
  - May lack generalizability

Normalization Scaling

- Learning algorithms perform optimally when feature values have similar ranges
- For every sample, we do

$$\frac{x_i^{(i)} - \min^{(i)}}{\max^{(i)} - \min^{(i)}}$$

Standardization Scaling

- Transforms each feature to have mean 0 and standard deviation 1
- For every sample, we do

$$\frac{x_i^{(i)} - \mu^{(i)}}{\sigma^{(i)}}$$

- Range of values are not bounded

Standardization vs. Normalization

- Treat scaling as a hyperparameter and evaluate both normalization and standardization
- Standardization is generally more robust to outliers than normalization
- Some guidelines:
  - Use standardization for unsupervised learning tasks
  - Use standardization if features are approximately normally distributed
  - Prefer standardization in the presence of outliers
  - Otherwise, use normalization

Handling Missing Values

- Missing values refer to the absence of data points or entries in a dataset where a value is expected
- Handling missing values:
  - Drop Examples
    - Feasible if the dataset is large and outcome is unaffected
  - Drop Features
    - Suitable if it does not impact the project's outcome
  - Use algorithms handling missing data
  - Data Imputation
    - Replace missing values with computed values

Data Imputation

- Process of replacing missing values in a dataset with substituted values, typically using statistical or machine learning methods
- One strategy can be replacing missing values with mean or median of the attribute
- Cons: ignores feature correlations and complex relationships
- Mode imputation: replace missing values with the most frequent value
- Data imputation relies on several assumptions
  - Randomness: many methods assume missingness is unrelated to any data
  - Model bias: Incorrect randomness assumptions can lead to biased estimates and flawed conclusions
  - Information Loss: Imputation can obscure patterns, leading to loss of valuable information for advanced models
- Special Value Method: Replace missing values with a value outside the normal range
  - Objective: Enable the learning algorithm to recognize and appropriately handle missing values
- Middle-Range Imputation: Replace missing values with a value in the middle of the normal range

Using ML for Imputation

- Predict unknown labels for given examples
- Can be framed as a supervised learning problem
- X_i new example

$$X_i^{(1)}, X_i^{(2)}, \ldots, X_i^{(j-1)}, X_i^{(j+1)}, \ldots, X_i^{(D)}$$

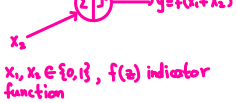
- Let $\hat{y}_i = X_i^{(j)}$
- Training set: use examples where $X_i^{(j)}$ is not missing
- Method: Train a classifier on this set to predict the missing values
- Advantages:
  - Effectively handle complex relationships and correlations between features
- Disadvantages:
  - Cost-intensive in terms of labor, CPU time, and memory resources

Class Imbalance

- A scenario where the number of instances in one class significantly outnumbers the instances in other classes
- Models tend to be biased towards the majority class, leading to poor performance on minority class
- Solutions:
  - Resampling: Techniques such as oversampling the minority class or undersampling the majority class
  - Algorithmic Adjustments: Using cost-sensitive learning or modifying decision thresholds
  - Synthetic Data: Generating synthetic samples for the minority class using methods like SMOTE
- Apply solutions only to the training set to prevent data leakage
- Oversampling:
  - Can lead to overfitting, especially if the synthetic samples are very similar to the existing ones
  - Impact: The model may perform well on training data but generalize poorly to unseen data
- Undersampling
  - Loss of information reduces the number of instances in the majority class
  - Impact: potentially discards valuable information and can lead to underfitting
  - Reduced Model Performance: Smaller training dataset may not capture the complexity of the problem
  - Impact: Can result in a less accurate and less robust model

Neural Networks

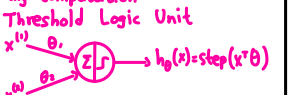
- We now investigate a family of machine learning models that draw inspiration from the structure and function of biological neural networks found in animals
- Computation with neurodes



$x_1, x_2 \in \{0, 1\}$, $f(z)$ indicator function

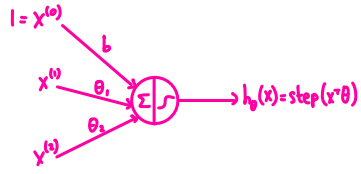
$$f(z) = \begin{cases} 0, & z < \theta \\ 1, & z \geq \theta \end{cases}$$

- Digital computations can be broken down into a sequence of logical operations, enabling neurode networks to execute any computation
- Threshold Logic Unit



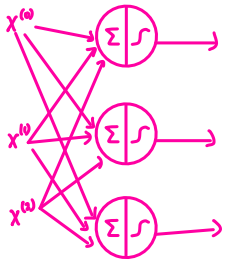
Weighted sum: $z = x^T\theta$

- We should add an extra feature with a fixed value of 1 to the input. Associate it with weight $b = \theta_0$, b bias term



A perceptron consists of one or more threshold logic units arranged in a single layer, with each unit connected to all inputs. This configuration is referred to as fully connected or dense

- As the perceptron generates multiple outputs simultaneously, it performs multiple binary predictions, making it a multilabel classifier

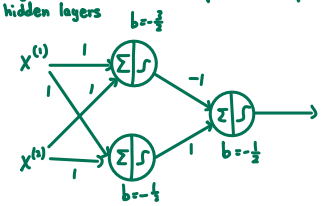


Some Notations

- X is the input data matrix where each row corresponds to an example and each column represents one of the D features
- W is the weight matrix, structured with one row per input (feature) and one column per neuron
- Bias terms can be represented separately. Here, b is a vector with a length equal to the number of neurons

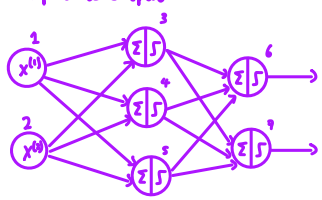
Multilayer Perceptron (MLP)

- Includes an input layer and one or more layers of threshold logic units
- Layers that are neither input nor output are termed hidden layers



Feedforward Neural Network (FNN)

Information in this architecture flows unidirectionally - from left to right, moving from input to output



Activation Functions

- The training algorithm, known as backpropagation, employs gradient descent, necessitating the calculation of the partial derivatives of the loss function
- The step function in the multilayer perceptron had to be replaced, as it consist only of flat surfaces. Gradient descent cannot progress on flat surfaces due to their zero derivative
- Nonlinear activation functions are paramount because, without them, multiple layers in the network would only compute a linear function of the inputs

Universal Approximation Theorem

- A feedforward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on a compact subset of $\mathbb{R}^n$, given appropriate weights and activation functions

Deep Learning

- A machine learning technique that can be applied to supervised learning, unsupervised learning, and reinforcement learning
- Inspired from the structure and function of biological neural networks found in animals
- Comprises interconnected neurons arranged into layers

Backpropagation

- The procedure repeatedly readjusts weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector
- The model employs mean squared error as its loss function
- Gradient descent is used to minimize loss
- A sigmoid activation function is used instead of a step function, as its derivative provides valuable information for gradient descent
- Shows how updating internal weights using a two-pass algorithm consisting of a forward pass and a backward pass
- Enables training multi-layer perceptrons
- Step 1. Initialization
  - Initialize the weights and biases of the neural network
  - Zero Initialization
    - All weights initialized to zero

    Symmetry problems, all neurons produce identical outputs, preventing effective learning
  - Random Initialization
    - Weights are initialized randomly, often using a uniform or normal distribution
    - Breaks the symmetry between neurons, allowing them to learn
    - If not scaled properly, leads to slow convergence or vanishing/exploding gradients
- Step 2: Forward Pass
  - For each example in the training set
    - Input Layer: Pass input features to first layer
    - Hidden Layer: For each hidden layer, compute the activations (output) by applying the weighted sum of inputs plus bias, followed by an activation function
    - Output Layer: Same process as hidden layers. Output layer activations represent the predicted values
- Step 3: Compute Loss
  - Calculate the loss (error) using a suitable loss function by comparing the predicted values to the actual target values
- Step 4: Backward Pass
  - Output Layer: Compute the gradient of the loss with respect to the output layer's weights and biases using the chain rule of calculus
  - Hidden Layers: Propagate the error backward through the network, layer by layer. For each layer, compute the gradient of the loss with respect to the weights and biases. Use the derivative of the activation function to help calculate the gradients
  - Update Weights and Biases: Adjust the weights and biases using the calculated gradients and a learning rate, which determines the step size for each update
- Activation Function: Functions like sigmoid, ReLU, and tanh introduce non-linearity, which allows the network to learn complex patterns
- Learning Rate: A hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated

- Gradient Descent: An optimization algorithm used to minimize the loss function by iteratively moving towards the steepest descent as defined by the negative of the gradient

Vanishing Gradients

- Gradients become too small, hindering weight updates
- Stalled neural networks research in early 2000s
- Sigmoid and its derivative were key factors
- Common initialization: Weights/biases from N(0,1) contributed to the issue
- Solutions:
  - Alternative activation functions
  - Weight initialization

Output Layer

- For regression tasks:
  - 1 output neuron per dimension
  - Activation function
    - ReLU/softplus if positive
    - Sigmoid/tanh if bounded
  - Loss: mean squared error
- For classification tasks
  - 1 output neuron if binary, otherwise 1 per class for multiclass
  - Activation function
    - Sigmoid if binary or multi-label
    - Softmax if multi-class
  - Loss: cross-entropy

Softmax

- An activation function used in multi-class classification problems to convert a vector of raw scores into probabilities that sum to 1
- Given a vector $z = [z_1, z_2, \ldots, z_n]$

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}}$$

$\sigma(z)_i$: probability of i^{th} class, n is number of classes

Cross-Entropy Loss Function

- For one example

$$J(W) = -\sum_{k=1}^{k} y_k \log(\hat{y}_k)$$

Where:

- K number of classes
- y_i: true prediction for class k
- $\hat{y}_i$: predicted probability of class k

- For a dataset with N examples, the average cross-entropy loss over all examples is computed as

$$L = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{k} y_{i,k} \log(\hat{y}_{i,k})$$

Where:

- i indexes over the different examples in the dataset
- $y_{i,k}$ and $\hat{y}_{i,k}$ are the true and predicted probabilities for class k of example i, respectively

Regularization

- Comprises a set of techniques designed to enhance a model's ability to generalize by mitigating overfitting. By discouraging excessive model complexity, these methods improve the model's robustness and performance on unseen data
- In numerical optimization, it is standard practice to incorporate additional terms into the objective function to deter undesirable model characteristics
- For a minimization problem, the optimization process aims to circumvent the substantial costs associated with these penalty terms

Norm

- Assigns a non-negative length to a vector
- l_p norm of a vector $z = [z_1, z_2, \ldots, z_n]$ is defined as

$$\|z\|_p = \left(\sum_{i=1}^{n} |z_i|^p\right)^{1/p}$$

- l_1 norm:

$$\|z\|_1 = \sum_{i=1}^{n} |z_i|$$

- l_2 norm:

$$\|z\|_2 = \sqrt{\sum_{i=1}^{n} z_i^2}$$

l_1 and l_2 Regularization

$$MAE(X,W) = \frac{1}{N}\sum_{i=1}^{N} |h_w(x^{(i)}) - y^{(i)}| + \alpha l_1 + \beta l_2$$

- α and β determine the degree of regularization applied
- l_1 regularization:
  - Promotes sparsity, setting many weights to zero
  - Useful for feature selection by reducing feature reliance
- l_2 regularization:
  - Promotes small, distributed weights for stability
  - Ideal when all features contribute and reducing complexity is key

Dropout

- Regularization techniques in neural networks where randomly selected neurons are ignored during training, reducing overfitting by preventing co-adaptation of features
- During each training step, each neuron in a dropout layer has a probability p of being excluded from the computation
- While seemingly counterintuitive, this approach prevents the network from depending on specific neurons, promoting the distribution of learned representations across multiple neurons
- One of the most popular and effective methods for reducing overfitting
- Typical improvement in performance is modest

Early Stopping

- A regularization technique that halts training once the model's performance on a validation set begins to degrade, preventing overfitting by stopping before the model learns noise

Hierarchy of Concepts

- Each layer detects patterns from the output of the layer preceding it
  - In other words, proceeding from the input to the output of the network, the network uncovers "patterns of patterns"
- Start with one layer, then increase the number of layers until the model starts overfitting the training data
- Finetune the model adding regularization
- With a feed-forward network, the number of parameters grows rapidly with more layers

Convolutional Neural Network

- Specializes in pattern recognition
- Use filters to recognize patterns
- Crucial pattern information is often local
- Convolutional layers reduce parameters significantly
  - Unlike dense layers, neurons in a convolutional layer are not fully connected to the preceding layer
  - Neurons connect only within their receptive fields

Kernel

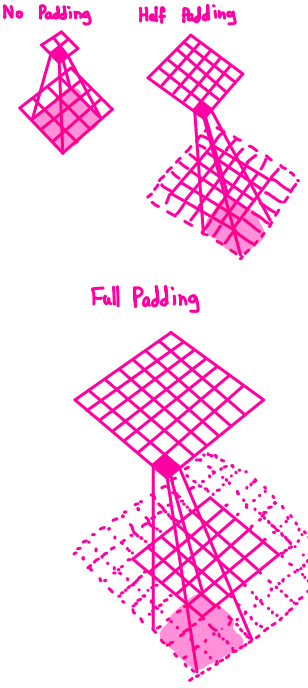
- A small matrix that slides over the input data to perform convolution
- With the kernel placed over a specific region of the input matrix, the convolution is element-wise multiplication followed by a summation of the results to produce a single scalar value
- In convolutional networks, the kernels are automatically learned by the network

Receptive Field

- Each unit is connected to neurons in its receptive fields
  - Unit i,j in layer l is connected to the units i to $i+f_h-1$, j to $j+f_w-1$ of the layer $l-1$, where f_h and f_w are respectively the height and width of the receptive field.

Padding

- Zero padding: In order to have layers of the same size, the grid can be padded with zeros



Stride

It is possible to connect a larger layer ($l-1$) to a smaller one (l) by skipping units. The number of units skipped is called stride, S_h and S_w

- Unit i,j in layer l is connected to the units $i \times S_h$ to $i \times S_h + f_h - 1$, $j \times S_w$ to $j \times S_w + f_w - 1$ of the layer $l-1$, where f_h and f_w are respectively the height and width of the receptive field, S_h and S_w are respectively the height and weight strides

Filters

- A window of size $f_h \times f_w$ is moved over the output of layers $l-1$, referred to as the input feature map, position by position
- For each location, the product is calculated between the extracted patch and a matrix of the same size, known as a convolutional kernel or filter. The sum of the values in the resulting matrix constitutes the output for that location

Model

For every node, we have that its output is

$$Z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1}\sum_{v=0}^{f_w-1}\sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k}$$

$i' = i \times S_h + u \qquad j' = j \times S_w + v$

Convolutional Layer

- A layer full of neurons using the same filter outputs a feature map
- During training the convolutional layer will automatically learn the most useful filters for its task
- The layers above will learn to combine them into more complex patterns
- The fact that all neurons in a feature map share the same parameters dramatically reduces the number of parameters in the model

Summary of CNNs

- Feature Map: In convolutional neural networks, the output of a convolution operation is known as a feature map. It captures the features of the input data as processed by a specific kernel
- Kernel Parameters: The parameters of the kernels are learned through the backpropagation process, allowing the network to optimize its feature extraction capabilities based on the training data
- Bias Term: A single bias term is added uniformly to all entries of the feature map. This bias helps adjust the activation level, providing additional flexibility for the network to better fit the data
- Activation Function: Following the addition of the bias, the feature map values are typically passed through an activation function

Pooling

- A pooling layer exhibits similarities to a convolutional layer
  - Each neuron in a pooling layer is connected to a set of neurons within a receptive field
- However, unlike convolutional layers, pooling layers do not possess weights
  - Instead, they produce an output by applying an aggregating function, commonly max or mean
- This subsampling process leads to a reduction in network size; each window of dimensions $f_h \times f_w$ is condensed to a single value, typically the maximum or mean of that window
- Dimensionality Reduction: Pooling layers reduce the spatial dimensions (width and height) of the input feature maps. This reduction decreases the number of parameters and computational load in the network, which can help prevent overfitting
- Feature Extraction: By summarizing the presence of features in a region, pooling layers help retain the most critical information while discarding less important details. This process enables the network to focus on the most salient features
- Translation Invariance: Pooling introduces a degree of invariance to translations and distortions in the input
- Noise Reduction: Pooling can help smooth out noise in the input by aggregating information over a region, thus emphasizing consistent features over random variations
- Hierarchical Feature Learning: By reducing the spatial dimensions progressively through the layers, pooling layers allow the network to build a hierarchical representation of the input data, capturing increasingly abstract and complex features at deeper layers

Search

When the correct action to take is not immediately obvious, an agent may need to plan ahead: to consider a sequence of actions that form a path to a goal state. Such an agent is called a problem-solving agent, and the computational process it undertakes is called search

An agent is an entity that performs actions. A rational agent is one that acts to achieve the "best" outcome. Conceptually, an agent perceives its environment through sensors and interacts with it using actuators

Search Environment Characteristics

- Observability: Partially observable, or fully observable
- Agent Composition: Single or multiple agents
- Predictability: Deterministic or non-deterministic
- State Dependency: Stateless or stateful
- Temporal Dynamics: Static or Dynamic
- State Representation: Discrete or Continuous

Search Problem Definition

- A collection of states, referred to as the state space
- An initial state where the agent begins
- One or more goal states that define successful outcomes
- A set of actions available in a given state s
- A transition model that determines the next state based on the current state and selected action
- An action cost function that specifies the cost of performing action a in state s to reach state s'

Paths and Solutions

- A path is defined as a sequence of actions
- A solution is a path that connects the initial state to the goal state
- An optimal solution is the path with the lowest cost among all possible solutions
- We assume that the path cost is the sum of the individual action costs, and all costs are positive. The state space can be conceptualized as a graph, where the nodes represent the states and the edges correspond to the actions

Search Tree

- A conceptual tree structure where nodes represent states in a state space, and edges represent possible actions, facilitating systematic exploration to find a path from an initial state to a goal state
- The root of the search tree represents the initial state of the problem
- Expanding a node involves evaluating all possible actions available from that state
- The result of an action is the new state achieved after applying that action to the current state
- Similar to other tree structures, each node (except root and leaf nodes) has a parent and may have children
- Any state corresponding to a node in the search tree is considered reached
- Frontier nodes are those that have been reached but have not yet been expanded

Uninformed Search

- A search strategy that explores the search space using only the information available in the problem definition, without any domain-specific knowledge, evaluating nodes based solely on their inherent properties rather than estimated costs or heuristics

Breadth-First Search

- Employs a queue to manage the frontier nodes, which are also known as the open list

It finds the shortest path from the initial state to the goal state

- The Algorithm:

Input: Initial and goal states
we initialize a queue that stores the frontier nodes
we also initialize a set that stores explored states
Add the initial state to both
While the queue is not empty

Current state and path is first element of queue
if the current state is the goal state, then return the successful path

Otherwise, we expand the current state and we add the neighbor states to the set of explored states and the queue

return None if no solution found

Depth-First Search

- Uses a stack to keep track of states
- The Algorithm

Input: Initial and Goal States
Initialize a stack that stores the frontier states
Initialize a set that stores the explored nodes
while the stack is not empty:

Top state of stack is current state
If the current state is the goal state then return the path

Otherwise, add all its neighbours to the explored set and the stack

return None if no solution found

Heuristic Search

- Use domain-specific knowledge regarding the goal's state location
- Let f(n) be a heuristic function that estimates the cost of the cheapest path from the current state or node n to the goal
- In root finding problems, one might employ the straight-line distance from the current node to the destination as a heuristic. Although an actual path may not exist along that straight line, the algorithm will prioritize expanding the node closest to the destination based on this straight-line measurement
- This approach is known as best-first search
- To implement this, we can employ a priority queue, which is sorted according to the values of the heuristic function h(n)

A* Search

- The most common informed search

$$f(n) = g(n) + h(n)$$

where

- g(n) is the path cost from the initial state to n
- h(n) is an estimate of the cost of the shortest path from n to the goal state

- A* guarantees a solution if one exists
- Cost optimality requires an admissible heuristic
- An admissible heuristic never overestimates the goal cost

Best-First Search

- The Algorithm:

First initialize a priority queue frontier
Then we initialize the heuristic to the distance between start and end state
Push the initial state with the heuristic value into the queue
Create a set that stores explored states
While the queue is not empty
if the current state is the goal state, then return the successful path

Otherwise, we add the current state to the explored set

We expand the current state and go through each neighbor

For every unexplored neighbor, we increment the path cost, calculate the cost to the goal, update total cost, then add it to the priority queue and the set of explored states

return None if no solution found

Measuring Performance

- Completeness: Does the algorithm ensure that a solution will be found if one exists, and accurately indicate failure when no solution exists?
- Cost Optimality: Does the algorithm identify the solution with the lowest path count among all possible solutions?
- Time Complexity: How does the time required by the algorithm scale with respect to the number of states and actions?
- Space Complexity: How does the space required by the algorithm scale with respect to the number of states and actions?

Local Search

- Local search algorithms operate by searching from a start state to neighboring states, without keeping track of the paths, nor the set of states that have been reached

Optimizes memory utilization while effectively solving problems in extensive or infinite state spaces

- Find the "best" state according to an objective function, thereby locating the global maximum
- Commonly referred to as hill climbing

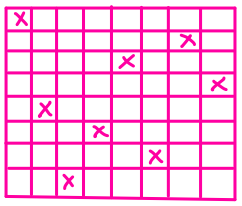
Hill-Climbing

Given as in input a problem

- Current is the initial state of problem
- while not done do
  - neighbour is the highest-valued successor state of current
  - if value(neighbor) ≤ value(current) then return current
  - set current to neighbour

8-Queens Problem

- Involves placing eight queens on an 8×8 chessboard such that no two queens threaten each other, meaning no two queens share the same row, column, or diagonal
- A grid representation of the current state permits the illegal placement of two queens in the same column
- Instead, we can represent the state as a list (state), where each element corresponds to the row position of the queen in its respective column
- state[i] is the row of the queen in column i



State = [0, 4, 7, 5, 2, 6, 1, 3]

- We can randomly create an initial state
- Representations
  - Unconstrained Placement: $\binom{64}{8} = 4,426,165,368$ possible configurations
  - Column Constraint: Uses a list of length 8, with each entry indicating the row of a queen in its respective column, resulting in $8^8 = 16,777,216$ configurations
  - Row and Column Constraints: Model board states as permutations of the 8 row indices, reducing configurations to $8! = 40,320$
- We can write a function that calculates the number of conflicts
  - It iterates through the list and compares if the queens are in the same row, same diagonal, or same anti-diagonal
  - If any of this happens, increase the number of conflicts
- We can also write a function that generates neighboring states by moving one queen at a time to a new row
- We can also write a function that generates neighboring states by swapping two rows
- To define the hill-climbing method, we have the current number of conflicts. Then we go into a loop. We return the current state if there are no conflicts. Otherwise, we get its neighbors and their number of conflicts. We get stuck at the local minimum if we cannot find an improvement. Otherwise, we move the current state to the state with fewest number of conflicts
- We can also define a variable that keeps track of side moves when the objective hits a plateau

Simulated Annealing

- An optimization algorithm inspired by the annealing process in metallurgy.
- It probabilistically explores the solution space by allowing occasional uphill moves, which helps escape local optima
- The algorithm gradually reduces the probability of accepting worse solutions by lowering a "temperature" parameter, ultimately converging towards an optimal or near-optimal solution
- Annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them
- The Algorithm

function SimulatedAnnealing(problem, schedule) returns a solution state
  current ← problem.INITIAL
  for t=1 to ∞ do
    T ← schedule(t)
    if T=0 then return current
    next ← a randomly selected successor of current
    ΔE ← VALUE(current) − VALUE(next)
    if ΔE > 0 then current ← next
    else current ← next only with prob $e^{-\Delta E/T}$

- Moves resulting in significant negative changes to the objective function are less likely to be accepted
- For a fixed ΔE, changes are more likely to be accepted when T is high at the start of the algorithm
- If the schedule lowers T to 0 slowly enough, then a property of the Boltzmann distribution, $e^{\Delta E/T}$, is that all the probability is concentrated on the global maxima, which the algorithm will find with probability approaching 1.

Travelling Salesman Problem

A classic optimization problem that seeks the shortest possible route for a salesman to visit a set of cities, returning to the origin city, while visiting each city exactly once

- Use a list where each element represents the index of a city, and the order of elements indicates the sequence of city visits
- Ways to generating a neighboring solution
  - Swap Two Cities: Select two cities at random and swap their positions
    - Pros: Simple and effective for exploring nearby solutions
    - Cons: Change may be too small, potentially slowing down convergence
  - Reverse Segment: Select two indices and reverse the segment between them
    - Pros: More effective at finding shorter paths compared to simple swaps
    - Cons: Can still be computationally expensive as the number of cities increases
  - Remove and Reconnect: Removes three edges from the route and reconnects the segments in the best possible way. This can generate up to 7 different routes

- Pros: Provides more extensive changes and can escape local optima more effectively than 2-opt
- Cons: More complex and computationally expensive to implement

- Insertion Move: Select a city and move it to a different position in the route
  - Pros: Offers a balance between small and large changes, making it useful for fine-tuning solutions
  - Cons: May require more iterations to converge to an optimal solution
- Shuffle Subset: Select a subset of cities in the route and randomly shuffle their order
  - Pros: Introduces larger changes and can help escape local minima
  - Cons: Can lead to less efficient routes if not handled carefully

Simulated Annealing and TSP

- We can use simulated annealing to solve the travelling salesman problem
- The inputs are: a distance matrix containing the distances between cities, the initial temperature, the cooling rate, and maximum number of iterations
- We first determine the number of cities, the current route, and the current cost
- We initialize the best route and best cost to the current route and current cost
- We initialize the temperature to the initial temperature
- We iterate for a while. For each iteration, we do
  - Pick a neighbor and its cost
  - If the neighbor costs less, then we move to it. Otherwise, we move to it with a probability
  - We then cool down the temperature
- As $t \to \infty$, this algorithm exhibits behaviour characteristic of a random walk. During this phase, any neighboring state, regardless of whether it improves the objective function, is accepted. This facilitates exploration and occurs at the start of the algorithm's execution
- As $t \to 0$, the algorithm behaves like hill climbing. In this phase, only those states that enhance the objective function's value are accepted, ensuring that the algorithm consistently moves towards optimal solutions - specifically, towards lower values in minimization problems. This phase emphasizes the exploitation of promising solutions and occurs towards the algorithm's conclusion