CS2109S Intro. to AI & Machine Learning

AY 25/26 Sem 1 — github/omgeta

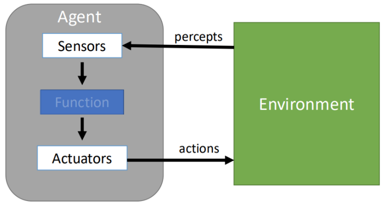
1. Intelligent Agents

Agents receive precepts from the environment through sensors and perform actions through actuators.

PEAS Framework

PEAS defines problems in terms of Performance Measure, Environment, Actuators and Sensors:

i. Rational Agent: chooses actions that maximise performance measure



Properties of Task Environment:

i. Fully Observable (vs. Partially Observable): sensors give access to the complete state of the environment at each point in time.
ii. Single-agent (vs. Multi-agent): agent operates by itself in an environment.
iii. Deterministic (vs. Stochastic): next state of the environment is completely determined by current state and action by agent.
   - Strategic: if environment is dependent on actions of other unpredictable (not "dumb") agents
iv. Episodic (vs. Sequential): agent's experience is divided into atomic "episodes", and choice of action in each episode depends only on the episode itself.
v. Static (vs. dynamic): environment is unchanged while agent deliberates.
   - Semi-dynamic: if environment does not change with time, but agent's performance score does.
vi. Discrete (vs. Continuous): a limited number of distinct percepts and actions.

Agent Structures

Agents are completely specified by the agent function:

i. Agent Function: $f : \mathcal{P} \to \mathcal{A}$ maps from precept histories $\mathcal{P}$ to actions $\mathcal{A}$
ii. Agent Program: implements the agent function

Common Agent Structures:

i. Simple Reflex: chooses action only based on current percept, ignoring precept history (follow if-then rules).
ii. Goal-based: select actions to achieve a given tracked goal.
iii. Utility-based: selects actions to maximise a utility function which assigns a score to any precept sequence. If the utility function aligns with performance measure, the agent is rational.
iv. Learning: improves performance over time with experience.
   - Performance Element: selects actions.
   - Learning Element: updates knowledge from feedback.
   - Critic: provides feedback on performance relative to a fixed performance standard.
   - Problem Generator: suggest exploratory actions

Exploration-Exploitation Dilemma:

i. Explore: learn more about the world.
ii. Exploit: maximize gain from current knowledge.

2. Systematic Search

Systematic search problems are fully-observable, deterministic, static, discrete and defined by:

i. States: representation of problem state
ii. Initial State
iii. Goal State/Test
iv. Actions: possible operations on a state
v. Transition Model: function of action on a state
vi. Action Cost Function

Uninformed Search

Uninformed search explores a problem space without domain-specific knowledge or heuristics to guide searching.

i. Branching factor b: max successors of any node
ii. Optimal solution depth d, Maximum depth m

Breadth-First Search (BFS) expands nodes level-by-level using a queue:

i. Time: $O(b^{d+1})$
ii. Space: $O(b^d)$
iii. Complete: Yes (if b is finite)
iv. Optimal: Yes (if all same step cost)

Uniform-Cost Search (UCS) expands least-cost node using a priority queue, for optimal solution cost C^*, levels C^*/ϵ:

i. Time: $O(b^{C^*/\epsilon})$
ii. Space: $O(b^{C^*/\epsilon})$
iii. Complete: Yes (if step costs $> 0 \wedge$ finite total cost)
iv. Optimal: Yes (if step costs > 0)

Depth-First Search (DFS) expands deepest unexpanded nodes using a stack:

i. Time: $O(b^m)$
ii. Space: $O(bm)$
iii. Complete: No (if infinite depth $\vee$ cyclic)
iv. Optimal: No

Depth-Limited Search (DLS) limits search depth to ℓ:

i. Time: $O(b^\ell)$
ii. Space: $O(b\ell)$ (with DFS)
iii. Complete: No (if $\ell < d$)
iv. Optimal: No (with DFS)

Iterative Deepening Search (IDS) runs DLS with increasing depth limit until solution found:

i. Time: $O(b^d)$ (with additional overhead)
ii. Space: $O(bd)$ (with DFS)
iii. Complete: Yes (if b finite)
iv. Optimal: Yes (if all same step cost)

Informed Search

Informed search uses heuristics to guide searching, where heuristic $h(n)$ estimates optimal cost from a state to goal:

i. All heuristics must be non-negative $\land$ $h(goal) = 0$

Usefulness Properties:

i. Admissible: $\forall$ node n, $h(n) \leq h^*(n)$ where $h^*(n)$ is optimal path cost to reach goal state from n ($h(n)$ never over-estimates true cost to goal)
   - Theorem: if $h(n)$ is admissible, A^* search (without visited memory) is optimal

ii. Consistent: $\forall$ node n, $h(n) \leq c(n, a, n') + h(n')$ (triangle inequality)
   - Theorem: if $h(n)$ is consistent, A^* search (with visited memory) is optimal
   - Theorem: if $h(n)$ is consistent, $h(n)$ is admissible

iii. Dominance: $\forall$ nodes n, $h_1(n) \geq h_2(n)$ implies that h_1 dominates h_2 (more informed)
   - Theorem: if h_1 is admissible, h_1 is better for search, expanding no more nodes than h_2

Admissible heuristics can be found using optimal cost of a relaxed problem (fewer restrictions on actions):

Ex. h_{SLD} straight-line distance if agent can fly
Ex. $h = 0$ if agent can teleport

Best-First Search expands nodes with evaluation $f(n) = h(n)$ using a priority queue:

i. Time: $O(b^m)$
ii. Space: $O(b^m)$
iii. Complete: No (may get stuck in loops).
iv. Optimal: No (heuristic-only may mislead).

A* expands nodes with evaluation $f(n) = g(n) + h(n)$, where $g(n)$ is step cost, using a priority queue:

i. Time: $O(b^d)$ (good heuristic can improve)
ii. Space: $O(b^d)$ (keeps all nodes in memory).
iii. Complete: Yes (if step costs > 0 $\land$ finite b).
iv. Optimal: Depends (if h admissible in tree search, or consistent in graph search).

3. Local Search

Local search problems are defined by:

i. States: representation of candidate solution, may not map to actual problem state
ii. Initial State
iii. Goal State/Test (optional)
iv. Successor Function: generate neighbour states by applying modifications to current state

Local search explores large, otherwise intractable problem spaces by considering only locally reachable states, guided by an evaluation function:

i. Evaluation Function: assesses quality of a state; we either minimize or maximise this function

State Space Landscape:

i. Global Maximum: optimal solution
ii. Local Maximum: local optimum solution
iii. Shoulder: region with flat evaluation function, difficult for algorithm to move past

Hill-Climbing searches for local optimum, generating successors from current state and picking the best using heuristic:

i. Any-time: More time gives better solutions
ii. Space: $O(b)$
iii. Complete: No
iv. Optimal: Not guaranteed
v. Variants:
   - Simulated Annealing: allow some bad moves, gradually decreasing frequency
   - Beam Search: perform parallel k hill-climbs
   - Genetic Algorithm: successor generated by combining 2 parent states
   - Random-Restart: escape local optimum by restarting search

4. Adversarial Search

Adversarial search problems are fully-observable, deterministic-strategic, static, discrete and defined by:

i. States: representation of candidate solution, may not map to actual problem state
ii. Initial State
iii. Terminal States: where game ends (e.g. win)
iv. Actions: possible operations on a state
v. Transition Model: function of action on a state
vi. Utility Function: output value of state from the perspective of our agent

Minimax assumes both players play optimally, expanding game tree in DFS with alternating MAX and MIN levels:

i. Time: $O(b^m)$
ii. Space: $O(bm)$
iii. Complete: Yes (for finite games)
iv. Optimal: Yes (for both, if both play optimally)
   - Theorem: if opponent plays sub-optimally, utility obtained by agent is never less than utility obtained against an optimal opponent

Alpha-Beta Pruning reduces nodes evaluated without altering minimax value and optimal move for root node:

i. Maintains α = best value , β = worst value and pruning subtree if $\alpha \geq \beta$
ii. Time: $O(b^{m/2})$ (with perfect ordering)
iii. Space: $O(bm)$ (same as minimax)
iv. Complete & Optimal: same as minimax

Cutoff Strategy halts search in the middle and estimates value of mid-game states with an evaluation function:

i. Evaluation Function: if terminal, use utility function, else use a heuristic
ii. Heuristic Function: estimates utility of a state
   - A heuristic must be between max and min utility
   - Admissibility and consistency are not needed
iii. Theorem: returns a move that is optimal with respect to the evaluation function at the cutoff; may be suboptimal with respect to true minimax

5. Machine Learning

Machine learning develops learning agents with data.

i. Data, $D = \{1 \le i \le n : (x^{(i)}, y^{(i)})\}$ of n samples, for $x^{(i)} \in \mathbb{R}^d$ feature vector, $y^{(i)}$ label of i^{th} data point.
ii. Model/Hypothesis, $h : X \to Y$ is a predictor of outputs which is learned by a learning algorithm.
iii. Performance Measure evaluates h map $x^{(i)} \to y^{(i)}$.
   - $MAE = \frac{1}{n}\sum_{i=1}^{n} \left|\hat{y}^{(i)} - y^{(i)}\right|$ (outlier robust)
   - $Accuracy = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{\hat{y}^{(i)}=y^{(i)}}$ (classification)
iv. Loss Function measures $\hat{y}^{(i)}$ error for optimization
   - $MSE = \frac{1}{n}\sum_{i=1}^{n} \left(\hat{y}^{(i)} - y^{(i)}\right)^2$ (outlier sensitive)

Binary Metrics:

i. Accuracy $= \frac{\text{TP + TN}}{\text{TP + FN + FP + TN}}$
ii. Precision, $P = \frac{\text{TP}}{\text{TP + FP}}$ (if FP are costly)
iii. Recall, $R = \frac{\text{TP}}{\text{TP + FN}}$ (if FN are costly)
iv. F1 Score, $F1 = \frac{2}{\frac{1}{P}+\frac{1}{R}}$ (balance precision & recall)

Decision Trees

Decision trees split data using features to predict outputs.

i. Representational Completeness: any decision tree can fit any finite consistent labelled data exactly

Entropy: $H(Y) = -\sum_i P(y_i)\log_2 P(y_i)$

Cond. Entropy: $H(Y \mid X) = \sum_x P(X=x)H(Y \mid X=x)$

Information Gain: $IG(Y;X) = H(Y) - H(Y \mid X)$

DTL grows tree top-down, greedily choosing feature X with highest IG, and recurses. At leaves:

i. If no more data, return default
ii. If data has same classification, return classification
iii. If no more features, return majority class

Pruning reduces overfitting, giving simpler hypothesis by limiting representational capacity, removing outliers:

i. Max-depth: limit max depth of decision tree.
ii. Min-sample leaves: set min samples for leaf nodes.

6. Supervised Learning

Supervised Learning learns from labelled data to learn a mapping from inputs to outputs.

Linear Regression

Linear regression creates a linear model to predict $y \in \mathbb{R}$:

$$h_w(x) = w^T x = w_0 + w_1 x_1 + \cdots + w_d x_d$$

where $w_0, \cdots, w_d$ are weights, using MSE loss function:

$$J_{MSE}(w) = \frac{1}{2n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})^2$$

Learning uses $\frac{\partial J(w)}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$ via:

i. Normal Equation: $w = (X^\top X)^{-1}X^\top Y$
   - Closed form, assumes invertible, $O(d^3)$ inversion
ii. Gradient Descent: $w_j \leftarrow w_j - \gamma\frac{\partial J(w)}{\partial w_j}$ repeated
   - Converge on global min., for appropriate $\gamma > 0$
   - If features linearly indep., global min is unique
   - May need feature scaling/ different γ_j for weights
   - Stochastic/ Mini-batch faster by using less data

Logistic Regression

Logistic regression creates a model to classify $y \in \{0, 1\}$:

$$h_w(x) = \sigma(w^T x), \quad \sigma(z) = \frac{1}{1+e^{-z}}$$

where w is the weight vector and $\sigma(z)$ is sigmoid function with $\sigma'(z) = \sigma(z)(1-\sigma(z))$, using BCE loss function:

$$J_{BCE} = \frac{1}{n}\sum_{i=1}^{n} BCE(y^{(i)}, h_w(x^{(i)}))$$

$$BCE(y,p) = -y\log(p) - (1-y)\log(1-p)$$

Model $h_w(x)$ can be interpreted as $P(y=1 \mid x)$, where we classify $y=1$ if $h_w(x) \ge \tau$, decision boundary $h_w(x) = \tau$

Learning only via Gradient Descent with same properties.

Feature Transformation $x \in \mathbb{R}^d \to \phi(x) \in \mathbb{R}^M$ allows linear and logistic regression on non-linear relationships.

Regularisation

Regularisation augments constraints to prevent overfitting:

$$J_{reg}(w) = J(w) + \lambda R(w), \quad \text{where } \lambda > 0$$

i. Generally, L_p-norm, $R_{L_p} = \|w\|_p = (\sum_{j=0}^{d}|w_j|^p)^{\frac{1}{p}}$
ii. L_1-norm, $R_{L_1} = \|w\|_1 = \sum_{j=0}^{d}|w_j|$
   - $\frac{\partial J_{L_1}^{MSE}(w)}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}(w^\top x^{(i)} - y^{(i)})x_j^{(i)} + \lambda\frac{\partial |w_j|}{\partial w_j}$
   - Theorem: features with non-zero weight will have constant penalty of λ
   - Theorem: L_1 induces sparsity, selecting features with $J_{MSE}(w) = \lambda$
iii. L_2-norm, $R_{L_2} = \|w\|_2 = \sqrt{\sum_{j=0}^{d} w_j^2}$ or $\frac{1}{2}\sum_{j=0}^{d} w_j^2$
   - $\frac{\partial J_{L_2}^{MSE}(w)}{\partial w_j} = \frac{1}{n}\sum_{i=1}^{n}(w^\top x^{(i)} - y^{(i)})x_j^{(i)} + \lambda w_j$
   - Theorem: $X^\top X + \lambda I$ is invertible with unique, closed-form $w = (X^\top X + \lambda I)^{-1}X^\top Y$
   - Theorem: L_2 induces weights shrinkage
iv. L_∞-norm, $R_{L_\infty} = \|w\|_\infty = \max_j |w_j|$

Dual Formulation and Kernel Method

Dual formulation re-parameterises w using training data. Representer Theorem for optimal w of $J_{L_2}^{MSE}(w)$:

$$w = \sum_{j=1}^{n}\alpha_j x^{(j)}, \quad h_\alpha(x) = \sum_{j=1}^{n}\alpha_j x^{(j)\top}x = \sum_{j=1}^{n}\alpha_j\langle x^{(j)}, x\rangle$$

$$J_{L_2}^{MSE}(\alpha) = \frac{1}{2}\sum_{i=1}^{n}(\sum_{j=1}^{n}\alpha_j x^{(j)\top}x^{(i)} - y^{(i)})^2 + \lambda\sum_{i=0}^{n}(\sum_{j=1}^{n}\alpha_j x_i^{(j)})^2$$

Kernel function $k_\phi(u,v) = \langle\phi(u), \phi(v)\rangle$ exists for any feature mapping ϕ, giving cheaper computation:

$$h_\alpha^\phi(x) = \sum_{j=1}^{n}\alpha_j k_\phi(x^j, x)$$

Common kernels:

i. Polynomial deg. k: $k_{pk}(u,v) = (u^\top v)^k$
ii. Gaussian var. s^2: $k_{RBF}(u,v) = \exp(-\frac{\|u-v\|^2}{2s^2})$

Support Vector Machines (SVMs)

SVMs are models to classify $y \in \{-1, 1\}$:

$$h_w(x) = \text{sign}(w^T x), \quad h_\alpha(x) \quad = \text{sign}(\sum_{x^{(i)} \in S} \alpha_i k(x^{(i)}, x))$$

where w is normal vector of the zero-offset hyperplane decision boundary $w^\top x_{\in S} = 0$, maximising objective:

$$\max_w \frac{1}{\|w\|} = \min_w \frac{\|w\|^2}{2}, \quad y^{(i)}(w^\top x^{(i)}) \geq 1$$

$$\max_\alpha \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=j=1}^n \alpha_i \alpha_j y^{(i)} y^{(j)} k_\phi(x^{(i)}, x^{(j)}), \sum_{i=1}^n \alpha_i y^{(i)} = 0$$

i. Distance to point x: $\frac{|w^\top x|}{\|w\|}$, Margin: $\frac{1}{\|w\|}$

ii. Support vectors S lie exactly on margin boundary
- $\alpha_i > 0 \iff x^{(i)} \in S$, $\alpha_i = 0 \iff x^{(i)} \notin S$
- Theorem: for linearly separable data of $r \leq d$ effective dimension, $|S| \geq r + 1$ (i.e. support vectors much less than data points)

Applications

Multi-class Classification ($y \in \{1, \ldots, K\}$):

i. One-One: $\binom{K}{2}$ classifiers vote between I, J

ii. One-Rest: K classifiers choose between I, I^c

Multi-label Classification ($y \in \{0, 1\}^K$):

i. Binary Relevance: indep. binary classifiers
- ignores label correlations and constraints

ii. Classifier Chains: chain pred. of binary classifiers
- captures dependencies; order-sensitive

iii. Label Powerset
- preserve correlations; classes explode; data-sparse

Generalizability:

i. Dataset: relevance, noise, balance (classification)

ii. Model Complexity:
- Low: underfits for complex data, high bias, low variance on retrains on different data
- High: overfits for scarce data, low bias with enough data, high variance on retrains on different data

7. Unsupervised Learning

Unsupervised Learning finds patterns in unlabelled data.

K-Means Clustering

K-means Clustering groups n data points into K groups with centroids $\mu_1, \ldots, \mu_K$, where centroid μ_j of n_j points is the average of points $x^{(i)}$ in the cluster:

$$\mu_j = \frac{1}{n_j} \sum_{i=1}^{n_j} x^{(i)}$$

with evaluation function:

$$J(c^{(1)}, \ldots, c^{(n)}, \mu_1, \ldots, \mu_K) = \frac{1}{n} \sum_{i=1}^n \|x^{(i)} - \mu_{c^i}\|^2$$

1. Randomly initialize K centroids $\mu_1, \ldots, \mu_K$
2. Repeat until convergence:
   2.1. For $i = 1, \ldots, n$: $c^{(i)} \leftarrow k$ of closest μ_k
   2.2. For $k = 1, \ldots, K$: $\mu_k \leftarrow$ centroid of data points $x^{(i)}$ assigned to cluster k

i. Theorem: each iteration never increases distortion; need deterministic tie-break for conv. to local opt.

ii. Choose K by business need or using elbow method (stop after decrease in J with K slows suddenly)

iii. Can be used for classification.

Principal Component Analysis (PCA)

PCA preserves variance while reducing features to $r < d$:

i. SVD of $d \times n$ $X^\top$: $X^\top = U \Sigma V^\top$
- U is $d \times d$ orthonormal columns and rows
- Σ is $d \times n$ diagonal with ordered $\sigma_j \geq 0$
- V is $n \times n$ orthonormal columns and rows

ii. Truncating to σ_r yields $d \times r$ $\tilde{U}$, $r \times n$ $\tilde{\Sigma}$

iii. Compression of $X^\top$: $r \times n$ $Z = \tilde{U}^\top X^\top$

iv. Reconstruction: $\hat{X}^\top \approx \tilde{U} \tilde{U}^\top X^\top$ since $\tilde{U} \tilde{U}^T \approx I$ with approximation error dependent on r

v. Theorem: in mean-centered $\hat{X}^\top$, $\frac{\sigma_j^2}{n-1}$ are variance of data in the basis of $u^{(j)}$

vi. Var: $\frac{\sum_{i=1}^r \sigma_i^2}{\sum_{i=1}^n \sigma_i^2} \geq 0.99 \iff \frac{\sum_{i=1}^n \|\hat{x}^{(i)} - \tilde{x}^{(i)}\|^2}{\sigma_{i=1}^n \|\hat{x}^{(i)}\|^2} \leq 0.01$

8. Neural Networks

Neural networks are layered models of neurons that learn feature transformations for prediction tasks.

Neuron computes activation function on a weighted sum:

$$\hat{y} = g(z), \quad z = \sum_{j=0}^d w_j x_j = w^\top x$$

i. Common activation functions:
- Identity: $g(z) = z$
- Sigmoid: $g(z) = \sigma(z) = \frac{1}{1+e^{-z}}$
- Tanh: $g(z) = \tanh(z) = 2\sigma(2x) - 1 = \frac{e^z - e^{-z}}{e^z + e^{-z}}$
- Relu: $g(z) = max(0, z)$
- Leaky Relu: $g(z) = max(az, z)$
- Softmax: $g(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \in [0, 1]$

ii. Output Layer: chosen to match task target

iii. Hidden Layer: any layer between input and output

Forward Propagation is the process of input data passing through neural network to output layer:

i. Weights of layer i, $W^{[i]}$, where $W_{jk}^{[i]}$ are weights from previous neuron j to layer neuron k

ii. Activation function of layer i, $g^{[i]}$

iii. Output of layer i, $\hat{y}^{[i]} = g^{[i]}(W^{[i]\top} y^{[i-1]})$

Backpropagation is gradient descent in the network:

i. Compute loss $L(\hat{y}, y)$ at output layer

ii. Apply chain rule layer-by-layer backwards for $\frac{\partial L}{\partial w_j} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dz} \frac{\partial z}{\partial w_j}$ and $\frac{dL}{dx} = \frac{\partial L}{\partial z_1} \frac{\partial z_1}{dx} + \frac{\partial L}{\partial z_2} \frac{dz_2}{dx}$

iii. Algorithm: Compute $\frac{dL}{d\hat{y}}$, convert $g(z)$ to $g'(z)$, reverse graph and compute gradients

Tasks:

i. Binary Classification: 1 sigmoid output neuron

ii. Multi-class Classification: K softmax output neurons for $\hat{y} = [\hat{y}_1, \ldots, \hat{y}_K]^\top$ as one-hot vector

iii. Single Regression: 1 any output neuron

iv. Multi Regression: K any output neurons for $\hat{y} = [\hat{y}_1, \ldots, \hat{y}_K]^\top$

Convolutional Neural Networks (CNNs)

CNNs apply shared filters over local regions of inputs:

i. Convolution Layer: slide kernel over input and compute weighted sums to produce feature maps
   - Kernel/Filter: matrix of weights
   - Stride: step size
   - Padding: add k padding around border of input
   - Output: $N' = \lfloor \frac{N-K+2P}{S} \rfloor + 1$

ii. Pooling Layer: downsample feature maps to reduce resolution and parameters
   - Max-Pool: keep max value in window
   - Average-Pool: compute average value in window

iii. Architecture:
   - Feature Transformation: stack of Conv and Pools
   - Prediction: fully-connected output layer

usually using CE loss between predicted $\hat{y}$ and actual y:

$$J_{CE} = \frac{1}{n}\sum_{i=1}^{n} CE(y^{(i)}, \hat{y}^{(i)})$$

$$CE(y^{(i)}, \hat{y}^{(i)}) = -\sum_{j=1}^{K} y_j^{(i)} \log(\hat{y}_j^{(i)})$$

Recurrent Neural Networks (RNNs)

RNNs model sequences using hidden state, where for layer j with hidden state $h^{[t]}$ at time t:

$$h^{[t]} = \hat{y}^{[j]} = g^{[j]}(W^{[ij]\top}x^{[t]} + W^{[hj]\top}h^{[t-1]})$$

Tasks:

i. Many-Many (e.g. sequence tagging, translation): 1 input and 1 output neuron for $t \le T_x = T_y$

ii. Many-Many (e.g. QnA): 1 input for $t \le T_x$, "BEGIN" at $t = T_x + 1$, and 1 output neuron for $T_x < t \le T_y$

iii. Many-One (e.g. sentiment analysis): 1 input for $t \le T_x$, 1 output at final $t = T_y = T_x$

iv. One-Many (e.g. image captioning): 1 input at start $t = T_x = 1$, 1 output for $t \le T_y$

Attention Neural Networks (ANNs)

Attention lets a model selectively focus on relevant parts of a sequence using vectors derived from $d \times T$ input X:

i. Query, $d_q \times T$ Q: represent current element query

ii. Key, $d_k \times T$ K: contain keys to search information

iii. Value, $d_v \times T$ V: actual element information

Attention Layers:

i. Self-Attention: Q, K, V from same sequence
   - $q^{[t]} = W^q x^{[t]} \iff Q = W^q X, \qquad W^q$ is $d_q \times d$
   - $k^{[t]} = W^k x^{[t]} \iff K = W^k X, \qquad W^k$ is $d_k \times d$
   - $v^{[t]} = W_v x^{[t]} \iff V = W^v X, \qquad W^v$ is $d_v \times d$
   - $\alpha_{it} = \dfrac{(k^{[i]})^\top q^{[t]}}{\sqrt{d_k}} \iff A = \dfrac{K^\top Q}{\sqrt{d_k}}, \qquad A$ is $T \times T$
   - $\alpha'_{it} = \text{softmax}(a_{it}) \iff A' = \text{softmax(A)}$ col-w
   - $h^{[t]} = \sum_{i=1}^{T} a'_{it} v^{[i]} \iff H = VA', \qquad H$ is $d_v \times T$

ii. Masked Self-Attention: restrict to only $\le t$
   - $a_{it} = \begin{cases} a_{it}, & \text{if } i \le t \\ -\infty, & \text{if } i > t \end{cases}$

iii. Cross-Attention: Q from decoder sequence, K, V from encoder sequence
   - $k^{[t]} = W^k z^{[t]}$ i.e. $K = W^k Z$
   - $v^{[t]} = W_v z^{[t]}$ i.e. $V = W^v Z$

Tasks:

i. Many-Many: self-attention layer with BCE loss
   - Positional Encoding (distinct, consistent with T, bounded): $x'^{[t]} = x^{[t]} + PE^{[t]}$ where for pos. k:

     $$PE_k^{[t]} = \begin{cases} \sin(t/C^{\frac{k}{d}}), & \text{if } k\%2 == 0 \\ \cos(t/C^{\frac{k-1}{d}}), & \text{if } k\%2 == 1 \end{cases}$$

ii. Many-Many (e.g. QnA): self-attention for question, masked self-attention and cross-attention for answer

iii. Many-One (e.g. sentiment): self-attention layer with additional "CLS" input for $h^{[CLS]}$ to capture summary information of input sequence

iv. One-Many (e.g. image caption): masked self-attention layer with additional "BEGIN" input

Teacher Forcing is used in training for sequence prediction:

i. Input for next step is true target, not models own prediction from current step

+ All output can be generated simultaneously, no need to wait for previous word

+ In early training, predictions are nearly random, which may make the model lost

– During testing, teacher forcing is not available, leading to train-test mismatch

– Mitigation: sometimes feed true word, sometimes prediction, gradually reducing teacher forcing

Transformers

Transformers are deep attention neural networks:

i. Stack of Encoder Blocks; self-attention → feed-forward

ii. Stack of Decoder Blocks; masked self-attention → cross-attention → feed-forward

Unsupervised Learning with Neural Networks

Autoencoders use neural networks for dimensionality reduction and representation learning:

i. Encoder: converts high-dimensional d input to $k < d$ dimensional compressed data.

ii. Decoder: reconstructs original data from compressed data.

iii. Reconstruction Loss: measures difference between reconstructed output and original input.

iv. After Training: discard decoder, use encoder to generate compressed representation for each input.

Model Pre-training initialises feature transformation weights based on tasks which are solvable with unlabelled data, challenging enough to force understanding of task, and yield useful learned understanding:

i. Image Rotation Prediction

ii. Contrastive Learning (positive, negative pairs)

iii. Image Inpainting

iv. Next-word Prediction

Problems with Neural Networks

Common problems when training deep neural networks:

i. Overfitting:
   - Model fits training data too well and performs poorly on unseen data.
   - Early Stopping: stop training when performance on validation set begins to worsen.
   - Dropout: during training, randomly set some neurons' output to 0; prevents overfitting by making the network less reliant on specific neurons.

ii. Vanishing / Exploding Gradient:
   - Vanishing gradient: gradients become very small, causing very slow learning in deep networks.
   - Exploding gradient: gradients become very large, causing unstable updates.
   - Gradient Clipping: clip gradients within range $[-\text{clip\_value}, \text{clip\_value}]$ to control exploding gradients.

Perceptrons

Perceptron is a single neuron for binary classification:

$$\hat{y} = \text{sign}(w^\top x + b)$$

i. Represents a linear decision boundary (hyperplane)

ii. Perceptron update rule for misclassified i: $w' \leftarrow w + \lambda(y^{(i)} - \hat{y}^{(i)})x^{(i)}$

iii. Multi-Layer Perceptron (MLP): stack of fully-connected layers with non-linear activations.

Receptive Fields in CNNs

Receptive field of a neuron is the region of input that can affect its activation.

Formualae for receptive field r_i for a neuron in layer i:

$$r_i = r_{i-1} + (K_i - 1) \times j_{i-1}$$
$$j_i = j_{i-1} \times S_i$$

where K_i is kernel size at i, S_i is stride at i, $r_0 = j_0 = 1$

```
def alpha_beta_search(state):
    v = max_value(state, -∞, ∞)
    return action in successors(state) with value v

def max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for next_state in expand(state):
        v = max(v, min_value(next_state, α, β))
        α = max(α, v)
        if v >= β: return v
    return v

def min_value(state , α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for next_state in expand(state):
        v = min(v, max_value(next_state, α, β))
        β = min(β, v)
        if v <= α: return v
    return v
```

a-B pruning Steps:

i. Start at root $\alpha = -\infty, \beta = \infty$

ii. When going down, pass down the values of α, β

iii. When going up, pass up the value of α if MAX, or β if MIN

iv. When taking a value up, store $\alpha = \max(\alpha, value)$ if MAX, or $\beta = \min(\beta, value)$ if MIN

v. At any point, if there is $\alpha \geq \beta$ prune all other children

Worst-case

Name	Time Complexity[1]	Space Complexity[1]	Complete?	Optimal?
Breadth-first Search	Exponential	Exponential	Yes	Yes
Uniform-cost Search	Exponential	Exponential	Yes[3]	Yes[3]
Depth-first Search	Exponential	Polynomial	No | Yes[4]	No
Depth-limited Search	Exponential	Polynomial[2]	No	No[2]
Iterative Deepening Search	Exponential	Polynomial[2]	Yes	Yes
A* Search	Exponential	Exponential	Yes[3]	Yes $^{\text{if admissible}}$
A* Search with visited memory	Exponential	Exponential	Yes[3]	Yes $^{\text{if consistent}}$

1) In terms of some notion of depth/tier
2) If used with DFS
3) If edge costs are positive
4) With visited memory