1 Intelligent Agents

PEAS: Performance Measure, Environ- ment, Actuator, Sensors
Properties of Task Environment: Fully vs partially observable, Deterministic vs stochastic, Strategic (other intelligent agents), Episodic vs Sequential, Static vs dynamic (environment changes over time), Discrete vs continuous, Single vs multi-agent
Agent Structures: Simple Reflex (if-else), Goal based, Utility Based, Learning Agent
Exploration vs Exploitation: Learn about the world vs maximize gain based on current knowledge

2 Uninformed/Informed Search

Complete: Search always find a solution. **Optimal**: If the search produces a solution, it is the best one
Uninformed Search: No clue how good a state is i.e. how close to goal. Algorithm: BFS – FIFO queue, Uniform Cost Search (UCS) – priority queue, DFS – stack
Search with Visited Memory: Maintain set of visited states.
Depth Limited Search (DLS): Limit depth to l, backtrack when hit.
Iterative Deepening Search (IDS): Search with depth = $0, 1, \dots$
Informed Search: has a clue how good the state is using a heuristic function
Best-first search: Priority queue but using $f(n)$ to evaluate cost
A* search: $f(n) = g(n) + h(n)$ where $g(n)$ is the cost to reach n and $h(n)$ is the estimate distance from n to the goal.
Admissible: iff for every node n, $h(n) \leq h^*(n)$ where $h^*(n)$ is the optimal path cost to reach the goal state from n. If $h(n)$ is admissible, A* search without visited memory is optimal.
Consistent: iff for every node n, every successor n' generated by any action a, $h(n) \leq c(n, a, n') + h(n')$ and $h(G) = 0$. If $h(n)$ is consistent, A* search with visited memory is optimal
Dominance: if $h_1(n) \geq h_2(n)$ for all n, then h_1 dominates h_2. h_1 is better for search if it is admissible.

Name	Time	Space	Complete	Opt
BFS	Exp	Exp	Y	Y
UCS	Exp	Exp	Y	Y
DFS	Exp	Poly	N	N
DLS	Exp	Poly*	N*	N*
IDS	Exp	Poly*	Y	Y

* If used with DFS

3 Local/Adversarial Search

Local Search: For problems with too big search space. Search local neighbour states using perturbation or construction. Typically incomplete and suboptimal. **Anytime property**: longer runtime, better solution. Can obtain "good enough" solution
Pertubative search: Search space = complete solutions, search step = modification of solution
Constructive search: search space = partial solutions, search step = extend solution
Hill Climbing: Generate neighbours and pick the one with highest score.
Adversarial Search: Cooperative (same goal), Competitive (opposing goals), Mixed-motive
Adversarial Games: Players compete against each other. Two player zero-sum games: only one winner
Minimax: Max player picks biggest value out of all states returned by min (vice versa)
AB Pruning: α = largest value so far, β = smallest value so far. Pruning doesn't affect the final result, but brings the TC down to $O(b^{m/2})$.

4 Machine Learning

Mean Squared Error: $\frac{1}{N}\sum_{i=1}^{N}(\hat{y}^{(i)} - y^{(i)})^2$
Mean Absolute Error: $\frac{1}{N}\sum_{i=1}^{N}|\hat{y}^{(i)} - y^{(i)}|$
Entropy: $I(P(pos), P(neg)) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$
Information Gain: $IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A), remainder(A) = \sum_{i=1}^{v}\frac{p_i+n_i}{p+n} I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$
Decision Tree: Pick attribute with maximal IG at each node. Occam's Razor: prefer simpler hypothesis over complex one. Pruning: Min-sample leaf, Max-depth
Data Preprocessing: partition continuous values into intervals, assign most common values/assign probabilities to missing values or drop rows/attributes

5 Linear Regression

Regression: Predict a value for a data point
Hypothesis Class: The set of functions $h_w(x) = w_0x_0 + \cdots + w_dx_d = w^Tx$, $x_0 = 1$
Normal Equation: $w = (X^TX)^{-1}X^TY$, assuming invertible. If not invertible, regularize, or compute approximation using Moore-Penrose (least squares) inverse
Problems with Normal Equation: Cost of inverting matrix is high $O(d^3)$: does not scale well for large matrices + Will not work for non-linear models + Assumes invertibility.
Derivative: $\frac{\partial J_{MSE}}{\partial w_j} = \frac{1}{2N}\sum_{i=1}^{N}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$
Theorem: MSE loss function is convex for linear regression (and polynomial regression, with feature transformations)
Features of Different Scales: Normalize $x_j \leftarrow \frac{x_j - \mu_j}{\sigma}$. Alternatives: min-max scaling, robust scaling, different learning rates γ_j for each parameter
Variants: *Mini-batch*: use subset of training data, *Stochastic*: use a random point. Faster, more randomness, may escape local minima.
GD vs NE: GD - Need to choose learning rate γ, many iterations, performs well for large d, may need feature scaling. NE - slow for large d, X^TX must be invertible, no iterations or γ, no feature scaling needed.

6 Logistic Regression

Sigmoid: $\sigma(x) = \frac{1}{1+e^{-x}}$, $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. Squashes values between 0 and 1. $\sigma(x + c)$ shifts the curve left.
Logistic Regression Model: $h_w(x) = \sigma(w^Tx) = \sigma(w_0x_0 + \cdots + w_dx_d)$, $w_0 = 1$. Decision threshold: whether to predict class 0 or 1.
Theorem: MSE loss function is non-convex for logistic regression so it can't be used for gradient descent
Binary Cross Entropy (BCE): i.e. log loss. $BCE(y, \hat{y}) = -y\log(\hat{y}) - (1 - y)\log(1 - \hat{y})$. If y and $\hat{y}$ are close, BCE is small (and vice versa), hence it is a good loss function.
Mean BCE: $J_{BCE}(w) = \frac{1}{N}\sum_{i=1}^{N} BCE(y^{(i)}, h_w(x^{(i)}))$
Theorem: BCE loss is convex for logistic regression. There is only one minimum (global minimum)
Weight Update: $w_j \leftarrow w_j + \gamma\frac{\partial J_{BCE}(w)}{\partial w_j}$
Derivative: $\frac{\partial J_{BCE}(w)}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$
Multi-class Classification: Out of C classes, predict which one it is.
One-to-one: Binary classifier for each pair of classes. At the end, the class with most votes is selected. Results in $\frac{C(C-1)}{2}$ classifiers
One-vs-rest: Binary classifier for one class vs all others, for each class, resulting in C classifiers. The classifier with highest confidence (probability output) determines the class.
Generalization: ability to perform on unseen data. Generalization error: error on unseen data. Generalization is affected by *dataset quality and quantity*, and *model complexity*.
Dataset Quality: Relevance, Noise, Balance (equal repr). **Dataset Quantity**: Generally more is better, except when it contains all possible points, then model just memorizes the answer
Model Complexity: Size + Expressiveness of the hypothesis. Higher complexity = higher polynomial. Low complexity model may underfit, high complexity model may overfit.
Bias: High bias = more data points does not lead to a different model. Low bias = more data points leads to a more complex model
Variance: Low variance = retraining model with different training set based on the same ground truth leads to the same model. High variance = retraining a complex model with a different training set based on the same ground truth leads to a very different model
Model Complexity vs Error: High complexity: Error on unseen data decreases with more data points. Low complexity: error on unseen data remains roughly the same regardless of data quantity. *Overparameterized* (deep neural networks): Error on unseen data decreases initially, peaks once before decreasing again
Hyperparameters: Settings that control the behaviour of the model. They are not learned; need to be set before the training process. E.g. learning rate, feature transformations, batch size and no. of iterations.
Parameters: learned during training e.g. weights

7 Regularization, Kernels, SVM

Overfitting: Complex model includes noise, performs poorly on unseen data. Often associated with large weights. To prevent, keep weights small by putting a cost (penalty) on large weights. Given a loss function $J(w)$, add a penalty function/regularizer $P(w)$ with a penalty strength $\lambda \geq 0$: $J_{reg}(w) = J(w) + \lambda P(w)$. Optimization goal: $\min_w J_{reg}(w)$.
Penalty Functions: Square $P(w) = \sum_{j=0}^{d}\frac{1}{2}w_j^2$ (L2/Ridge, make all weights small). Absolute $P(w) = \sum_{j=0}^{d}|w_j|$ (L1/Lasso, zero out some weights). Max-norm $P(w) = \max(w_0, w_1, w_2, \dots)$
Gradient descent: $\frac{\partial}{\partial w_j}J_{reg}(w) = \frac{\partial}{\partial w_j}J(w) + \lambda\frac{\partial}{\partial w_j}P(w)$
Normal Equation with Reg: $\frac{\partial J_{reg}(w)}{\partial w_j} = \frac{1}{N}\sum_{i=1}^{N}(w^Tx^{(i)} - y^{(i)})x_j^{(i)} + \lambda w_j = 0$, $w = (X^TX + \lambda I)^{-1}X^TY$. NE with regularization works no matter if there is linear dependency or insufficient observations.
Theorem: For all $\lambda > 0$, the matrix $X^TX + \lambda I$ is invertible.
Weights and Training Data: $w = (X^TX)^{-1}X^TY = (X^TX)^{-1}\sum_{j=1}^{n}y^{(j)}x^{(j)} = \sum_{j=1}^{n}\alpha_jx^{(j)}$. α is a n-dimension vector.
Linear Model Dual Formulation: $h_w(x) = w^Tx = \sum_{j=1}^{n}\alpha_j{x^{(j)}}^Tx = h_\alpha(x)$. $h_\alpha(x)$ is a **dual hypothesis** which contains a sum of dot products between all $x^{(j)}$ and x, and weights α_j.
Dual Formulation and Kernel: Let $k(u, v) = u^Tv$. Then $h_\alpha(x) = \sum_{j=1}^{N}\alpha_jk(x^{(j)}, x)$. Other functions can be used to obtain different linear models.
Kernel Trick: Replace dot product with a similarity function $k(u, v)$ i.e. a kernel.
Kernel Machine: A hypothesis that uses kernel trick. *Valid* Kernel: continuous symmetric positive-definite kernel
Single variable transformed features: $x_j \rightarrow x_j$ and x_j^5, $x_j \rightarrow \log(x_j)$ and x_j, $x_j \rightarrow e^{(x_j)}$ and x_j.
Multi-variable transformed features: d-dimension to M-dimension feature vector. In general, $x \in \mathbb{R}^d \rightarrow \phi(x) \in \mathbb{R}^M$. Usually $M \geq d$. ϕ is a *feature map*, and $\phi(x)$ is called *transformed features*. E.g. $x = \begin{bmatrix}x_1, x_2, x_3\end{bmatrix}^T, \phi_{M2}(x) = \begin{bmatrix}x_1, x_2, x_3, x_1^2, x_2^2, x_3^2, x_1, x_2, x_1x_3, x_2x_3\end{bmatrix}^T$ where $M2$ transforms all monomials of degree up to 2.
Hypothesis Class: Given a d dimension input vector x and a M dimension feature map $\phi(x)$, the hypothesis class of linear models with transformed features is the set of functions $h_w^\phi(x) = w_0\phi(x)_0 + w_1\phi(x)_1 + \cdots + w_M\phi(x)_M$, where $w_0, \dots, w_M$ are weights, with dummy feature $\phi(x)_0 = 1$. Shorthand: $h_w(x) = w^T\phi(x)$.
Dual Hypothesis with Transformed Features: $h_\alpha^\phi(x) = \sum_{j=1}^{N}\alpha_j\phi^T(x^{(j)})\phi(x)$. The dot product between $\phi^T(x^{(j)})\phi(x)$ defines a new valid kernel function $k_\phi(u, v) := \phi^T(u)\phi(v)$.
Dual Hypothesis with Kernel: $h_\alpha^\phi(x) = \sum_{j=1}^{N}\alpha_jk_\phi(x^{(j)}, x)$. Don't have to compute $\phi(x)$ explicitly. $k_{Ps}(u, v) = \phi_{Ps}(u)^T\phi_{Ps}(v) = (u^Tv)^s$ for polynomial degree s with d^s terms
Theorem: For every valid kernel $k(u, v)$ there exists a corresponding feature transformation ϕ (which may be infinite-dimensional) where $k(u, v) = \phi^T(u)\phi(v)$. Conversely, every ϕ induces a valid kernel k.
Support Vector Machine: a classifier (can also be used for regression). Construct an 'optimal' separating decision boundary. Fat margin maximizes performance on unseen data and is robust to noise. Kernels can be applied naturally to lead to non-linear boundaries.
Hyperplane: defined by w, contains all vectors u for which $w^Tu = 0$. Sign of w^Tu is the side a point lies. Distance of a point: $|w^Tu|/||w||$. Support vectors lie on the edge of the margin.
$SVM_\alpha(x) = \text{sign}(\sum_{i=1}^{N}a^{(i)}k(x^{(i)}, x))$
Sparsity: Many $\alpha^{(i)}$ will be zero, only few are support vectors, others can be thrown out.

8 Unsupervised Learning

Learn patterns in the data.
Types: Clustering (identify clusters), Dimensionality reduction (find a lower dimensional representation of the data)
Clustering: group similar data points together. The no. of clusters is not predefined by the data. Common applications: data segmentation, anomaly detection.
Dimensionality Reduction: Reduce the no. of features while retaining as much relevant information as possible. Common applications: visualizations, feature extraction
K-Means Clustering: E.g. produce clothes of K distinct sizes such that the fit for everyone is pretty good.
Centroid: The average of a set of points $x^{(i)}$, $\mu = \frac{1}{N}\sum_{i=1}^{N}x^{(i)}$. A centroid defines a cluster.
Algorithm: 1. Randomly initialize K centroids $\mu_1, \dots, \mu_k$. 2. Repeat until convergence: 2.1 For $i = 1, \dots, N$, set $c^{(i)} \leftarrow$ index of centroid closest to $x^{(i)}$, 2.2 For $k = 1, \dots, K$, $\mu_k \leftarrow$ centroid of $x^{(i)}$ assigned to cluster k. Convergence = no more changes

Measuring Goodness of Clusters: Distortion (average distance of each sample to its centroid) $J(c^{(1)},\dots,c^{(N)},\mu_1,\dots,\mu_k)=\frac{1}{N}\sum_{i=1}^{N}||x^{(i)}-\mu_{c^{(i)}}||^2$
Theorem: Every step in K-Means never increases distortion.
Picking No. of Clusters: Elbow method – pick k where the graph of J against K forms an elbow.
Variants of K-Means: K-Medoids – initialize K random centroids, then pick data points closest to centroids and use them as centroids ('snap' centroids to nearest data point)
Clustering vs Classification: Classification: supervised, uses input-output pairs, hyperplane based (SVM), probability based (logistic model), no. of classes is defined by the dataset. Clustering: unsupervised, input only, distance based, no. of classes is set by us
Curse of dimensionality: No. of samples N needed to learn a hypothesis class increases *exponentially* with no. of features d, $N=O(2^d)$. Models that take in high dimensional features may suffer.
Singular Value Decomposition: Redundant features can be removed. Features can be made redundant by a change of basis. $A=U\Sigma V^T$
Theorem: WLOG, let $d>N$. For any $d\times N$ matrix A, there exists a factorization $A=X^T=U\Sigma V^T$ such that U is $d\times d$ and has d orthonormal columns, Σ is $N\times N$ and is a diagonal matrix with $\sigma_j\geq 0$, V is $N\times N$ and has N orthonormal columns and rows. U can be thought of as the new basis, Σ as the importance, and V as the linear combination of coefficients. Another way to think of it is, U contains templates (as columns), Σ is the template importance, and V is the combination coefficients.
Dimensionality Reduction: The N singular values σ tell us about the importance of the new basis vectors. Remove less important basis vectors by setting all singular values except the first r to 0. The new basis matrix $\tilde{U}$ is $N\times r$.
Compression: $Z=\tilde{U}^TX^T$ is $r\times N$ (project data points to the lower dimensional basis)
Theorem - Reconstruction: For a fixed r, $\tilde{U}\tilde{U}^T\approx I$, with approximation error dependent on r. $\tilde{X}^T=\tilde{U}\tilde{U}X^T\approx X^T$, $X^T=\tilde{U}Z$
Mean-centered data: Compute $\bar{x}$, then compute mean-centered data points $\hat{x}^{(i)}=x^{(i)}-\bar{x}$ for all i
Theorem: Given a mean-centered data matrix $\hat{X}^T$, the values $\frac{\sigma_j^2}{N-1}$ are variance of the data in the basis defined by the vectors $u^{(j)}$
Theorem: Task: Retain at least 99% of variance in the data. Solution: choose minimum r s.t. $\frac{\sum_{i=1}^{r}\sigma_i^2}{\sum_{i=1}^{N}\sigma_i^2}\geq 0.99$. Using this r, the original mean-centered data points and the reconstructed data points are close. $\frac{\sum_{i=1}^{N}||\hat{x}^{(i)}-\tilde{x}^{(i)}||^2}{\sum_{i=1}^{N}||\hat{x}^{(i)}||^2}\leq 0.01$
Principal Component Analysis: A statistical application of SVD. Capture components that maximize the statistical variations of the data

9 Neural Networks

Perceptron: Given n data points each with d features and target $\{-1,1\}$, $h_w(x)=g(w^Tx)=g(\sum_{j=0}^{d}w_jx_j)=g(w_0x_0+\cdots+w_dx_d)$ where $w_0,,\dots,w_d$ are weights and $w_0=1$ is a dummy variable. $g(z)=\begin{cases}+1 & z\geq 0\\ -1 & z<0\end{cases}$ is a sign function
Perceptron Learning Algorithm: Select one misclassified instance $(x^{(k)},y^{(k)})$, set $w\leftarrow w+\gamma(y^{(k)}-\hat{y}^{(k)})x^{(k)}$ until it converges. Different from SVM, margin is not maximized. If data is not linearly separable, will not converge. Why it works: $w^Tx=|w||x|\cos\theta$, adding/subtracting x from w will decrease/increase θ to the desired angle. γ affects how much the angle changes in one iteration.
Neuron: basic building block of a NN. A generalized version of perceptron. $\hat{y}=h_w(x)=g(\sum_{j=0}^{d}w_jx_j)$. Activation function can be anything: sigmoid $\sigma(x)=1/(1+e^{-x})$, tanh $\tanh(x)$, ReLU $\max(0,x)$, Leaky ReLU $\max(0.1x,x)$, $ELU(x)=\begin{cases}x & x\geq 0\\ \alpha(e^x-1) & x<0\end{cases}$, Maxout $\max(w_1^Tx+b_1,w_2^Tx+b_2)$
Neuron vs Linear/Logistic Regression: If activation function is the identity function $g(z)=z$, neuron becomes the linear regression model. If $g(z)=\sigma(z)$ then it becomes the logistic regression model.
Modelling OR/AND Gate with Logistic Regression Model: Plot truth table entries as data points and find a decision boundary
XNOR Gate: Data not linearly separable. Feature engineering is required to transform existing features. Can be done manually, or by using neurons to transform them (pass features into two neurons with $g(z)=\sigma(z)$.
NN vs Linear/Logistic Model: Linear/Logistic model requires manual feature engineering while NN learns its own features.
Matrix Multiplication: no. of rows = no. of inputs, no. of columns = no. of outputs. Each column of W is the weight vector for one neuron. $\hat{y}=g(W^Tx)=g(\begin{bmatrix}W_{11}&W_{12}\\W_{21}&W_{22}\\W_{31}&W_{32}\end{bmatrix}^T\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix})=\begin{bmatrix}\hat{y}_1\\\hat{y}_2\end{bmatrix}$
Multi-layer: $\hat{y}=g^{[2]}(W^{[2]^T}\cdot g^{[1]}(W^{[1]^T}x))=\begin{bmatrix}\hat{y}_1\\\hat{y}_2\end{bmatrix}$
Forward Propagation: The process by which input data passes through the NN to generate a prediction.
$\hat{y}=g^{[L]}(W^{[L]^T}\cdot g^{[L-1]}(\dots))=\begin{bmatrix}\hat{y}_1\\\dots\\\hat{y}_2\end{bmatrix}$
Multi-class Classification: c-class classification – set the no. of neurons in the last layer as c. Predict the probability of each class and pick the highest probability.
Softmax: Given $z=[z_1,z_2,\dots,z_c]$, for each z_i, $g(z_i)=e^{z_i}/(\sum_{j=1}^{c}e^{z_j})\in[0,1]$. $\sum g(z_i)=1$.

10 NN Training and CNN

Gradient Computation: $\hat{y}=h_w(x)=g(\sum_{j=0}^{d}w_jx_j)=g(z)$, $L=\frac{1}{2}(\hat{y}-y)^2$ (MSE)
$\frac{dL}{d\hat{y}}=\frac{d(\frac{1}{2}(\hat{y}-y)^2)}{d\hat{y}}=(\hat{y}-y)$, $\frac{d\hat{y}}{dz}=\frac{d(g(z))}{dz}=g'(z)$
$\frac{\partial z}{\partial w_j}=\frac{\partial(\sum_{j=0}^{d}w_jx_j)}{\partial w_j}=x_j$
$\frac{dL}{dw_j}=\frac{dL}{d\hat{y}}\frac{d\hat{y}}{dz}\frac{\partial z}{\partial w_j}=(\hat{y}-y)g'(z)x_j$
If w_j is used to generate u_j from v_j, then $\frac{\partial L}{\partial w_j}=\frac{\partial L}{\partial u_j}v_j$. $\frac{\partial L}{\partial u_j}$ can be computed in one backward pass. v_j can be computed in one forward pass.
One training loop: Forward pass to compute loss → zero the gradients with an optimizer → backpropagation → update weights
Convolution Layer: Naive computer vision – 800x800 image with 100 neurons in the first layer = 64 million weights. Not all pixels (features) are required for one neuron (e.g. detect ear). Can view a small region at a time and apply over all regions of the image.
Kernel/Filter: A matrix used to detect features from input, applied to all regions. For each region, multiply element-wise, then sum. The result is a *feature map*.
Common Practices: Add a padding of 1 pixel to the image border. Can also set the stride (no. of pixels to jump to next region). Both practices control the size of the feature map.
Multiple Input Channels: E.g. 3-channel RGB input. For each channel, use a separate weight matrix and compute the feature map, then sum all feature maps to create the output channel. The kernel is also multi-channeled, composing of the different weight matrices.
Multiple Output Channels: Use more kernels to create more output channels from one input channel. Concatenate the output channels at the end.
Pooling Layer: Downsample feature maps. Aggregation methods: Max-Pool, Average-Pool, Sum-Pool. For each $n\times n$ window, take the max/average/sum. Bigger pooling windows reduce the size of the feature map more.
Typical CNN: Input → Feature learning (convolution + ReLu + Pooling) → Prediction (flatten feature map + fully connected layer4 + softmax).
Popular CNN Architectures: VGG, Inception, ResNet, DenseNet
Applications of CNN: Image classification (e.g. happy/sad), Object detection (e.g. self-driving cars), Image segmentation (e.g. cancer cell detection; more fine-grained classification task)

11 RNN on Sequential Data

Recurrent Neural Network: Handles sequential data e.g. text ("we saw this saw"), predict label (verb/noun) for each word.
One-hot encoding: For each word, create a vector with length equal to the vocabulary (set of all words) size. Each word is assigned a unique index and its vector is all zeros except for a 1 in the position of that index.
Naive NN: One-hot vectors of words in different positions in a sentence will give the same $\hat{y}$; we should not predict each label independently → obtain contextual information from RNN
Hidden State: No. of elements in hidden vector = no. of desired outputs. At time step t, $h^t=\begin{bmatrix}h_1^t\\h_2^t\end{bmatrix}$, $h^t=g^{[h]}((W^{[xh]})^Tx^t+(W^{[hh]})^Th^{t-1})$, $\hat{y}=g^{[y]}((W^{[hy]})^Th^t)$. Hidden state stores information extracted from $x^1,\dots,x^t$ where x^i is the data at time i. One column in $W^{[xh]}$ is used to generate one node.
Many-to-many: $T_x=T_y>1$. Many RNN layers across time steps to generate many outputs.
Many-to-one: $T_x>1,T_y=1$. Only last layer used to generate one output.
One-to-many: $T_x=1,T_y>1$. Output prediction is passed into RNN layer of next time step.
Many-to-many: $T_x\neq T_y,T_x>1,T_y>1$. Use $<begin>$ vector
Deep RNN: Multiple RNN layers for each time step instead of just one
Applications of RNN: Sentiment analysis, speech recognition, video captioning
Properties of RNN: Captures contextual information, but each prediction must wait for all previous steps to be complete (not parallelism friendly)
Self-Attention Layer: Handles sequential data in parallel. Need to identify if previous elements are needed for generating the current output
Attention Score: Determines how much focus (attention) each part of the sequence (x^j) should receive when processing a specific element (x^i). Query $q^i=W^qx^i$, Key $k^j=W^kx^j$. W^q and W^k are trainable matrices used to extract useful features to calculate attention score. $a_{i\cdot j}=k^j\cdot q^i=(k^j)^Tq^i$. Value $v^i=W^vx^i$. Step 1: Linear Projection - transform input vectors into Query, Key and Value using weight matrices (shared across input) $Q=W^qX$, $K=W^kX$, $V=W^vX$. Step 2: Compute attention scores – $a_{ij}=k^j\cdot q^i=(k^j)^Tq^i$. $A=K^TQ$. Step 3: Apply softmax: $a'_{1j}=e^{a_{1j}}/\sum_j e^{a_{1j}}$, $A\rightarrow A'$. Step 4: Aggregate information - multiply values by attention score $h^i=\sum_j a'_{ij}v^j$, $VA'=H$. $h_1,\dots,h_n$ can be generated together, need not wait for previous steps
Positional Encoding: $x''^i=x^i+PE^i$. PE^i is unique for each position. sin and cos can be used to generate position encoding: $PE(i,2k)=\sin(\frac{i}{10000^{\frac{2k}{d}}})$, $PE(i,2k+1)=\cos(\frac{i}{10000^{\frac{2k}{d}}})$. $2k$ and $2k+1$ are are dimension indices of the positional encoding vector. d is the input feature dimension. $0\leq k\leq\frac{d}{2}-1$
Issues with Deep Learning: Overfitting, Vanishing/Exploding Gradient
Prevent Overfitting: Droput – During training, randomly set some neuron's output to 0. Early Stopping – stop before the loss increases
Vanishing/Exploding Gradient: Small gradients get multiplied again and again until it reaches almost zero. Mitigation: change activation functions. Large gradients get multiplied until it overflows. Mitigation: gradient clipping (clip within range [-clip value, clip value])

12 Attention Neural Networks

Many-to-one: $T_x>1,T_y=1$. h^{SUM} is used to capture relevant summary information throughout the entire input sequence to predict the output, thereby minimizing training loss
One-to-many: $T_x=1,T_y>1$. Output is fed as input to next time step generate next hidden vector. $x^1,\hat{y}^1,\dots,\hat{y}^{T_y-1}$ used to generate h^{T_y}.
Masked Self-Attention Layer: In step 3, add mask $[0,-inf,-inf]$ to attention scores before applying softmax.
Many-to-many: $T_x\neq T_y,T_x>1,T_y>1$. Use encoder model to generate a representation for input sequence and decoder to use encoded vectors to generate the output sequence. Self-attention layer used in encoder. Masked self-attention layer + Cross-attention layer for one-to-many decoder.
Cross-Attention Layer: Key $k_x^i=W^kh_x^i$, Query $q_y^i=W^qh_y^i$, Value $v_x^i=W^vh_x^i$. Attention score $a_{ij}=k_x^j\cdot q_y^i=(k_x^j)^Tq_y^i$.
Transformer: a deep attention NN. Basic building blocks: encoder block (Self attention layer + Feed forward NN), decoder block (Masked SA Layer + CA Layer + Feed forward NN). Feed forward NN: Linear layer + ReLu + Linear layer. Encoder: stack of encoder blocks, decoder: stack of decoder blocks. Generative Pretrained Transformer (GPT) are built on the transformer architecture to produce new text based on input prompts.
Vision Transformer: convert image into sequence of patches. Flatten each patch into a vector, multiply it by a weight matrix to project it into a patch vector, then add positional encoding vector and add a learnable summary vector.
AI Ethics: AI can generate biased output. AI can be used to generate fake content to manipulate human behaviour. Autonomous systems can make decisions independently, but 1. how should they behave, 2. who is accountable for harmful actions