Search	Time	Space	Complete?	Optimal?
BFS	Exp	Exp	Yes	Yes
UCS	Exp	Exp	Yes	Yes
DFS	Exp	Poly	No	No
DLS(D)	Exp	Poly	No	No
DLS(B)	Exp	Exp	No	Yes
IDS(D)	Exp	Exp	Yes	Yes
IDS(B)	Exp	Exp	Yes	Yes

Informed Search: Uses heuristics.

- **A***: $f(n) = g(n) + h(n)$.
- **Hill Climbing**: Greedy local search.
- **Admissibility:** $h(n) \leq h^*(n)$. Never overestimates cost.
- **Consistency:** $h(n) \leq h(n') + c(n, n')$. Guarantees optimality. Implies admissibility
- **Dominance:** $\forall n, h_1(n) \geq h_2(n) \implies h_1$ dominates h_2

Adversarial Search: Minimax Algorithm:

$$\text{max-value}(s) = \max \text{min-value}(s')$$

$$\text{min-value}(s) = \min \text{max-value}(s')$$

Alpha-beta pruning:

```
max_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = -∞
    for next_state in expand(state):
        v = max(v, min(v, α, β))
    return v
min_value(state, α, β):
    if is_terminal(state): return utility(state)
    v = ∞
    for next_state in expand(state):
        v = min(v, max(v, α, β))
    return v
alpha-beta search(state):
    v = max_value(state, -∞, ∞)
    // initialized α to be -∞, β to ∞
    return action in expand(state) with value v
```

Supervised Learning: Learns from labeled data.
Unsupervised Learning: Finds patterns in unlabeled data.
Reinforcement Learning: Learns via rewards. **Regression**
Linear Model: $h_w(x) = w^T x$.

- **MSE (L_2)**
  - Definition: $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
  - Sensitivity to Outliers: squares residuals → heavily penalizes large errors; very sensitive
  - Differentiability: smooth everywhere; gradient $= -2(y_i - \hat{y}_i)$
  - Closed-form Solution: yes (normal equations)
  - Statistical Interpretation: estimates the conditional *mean* under Gaussian-noise assumption
  - Typical Use Cases: Gaussian error models, smooth fast-converging gradient methods
- **MAE (L_1)**
  - Definition: $\frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
  - Sensitivity to Outliers: linear growth → penalizes proportionally; more robust
  - Differentiability: not differentiable at zero residual; subgradient $= \pm 1$ elsewhere
  - Closed-form Solution: no (requires iterative optimization, e.g. subgradient methods)
  - Statistical Interpretation: estimates the conditional *median*, robust to heavy tails
  - Typical Use Cases: heavy-tailed or outlier-prone data, when robustness is critical

Logistic Regression
Sigmoid Function: $\sigma(x) = \frac{1}{1+e^{-x}}$.
(BCE) Binary Cross Entropy Loss:

$$BCE(y, \hat{y}) = -y\log(\hat{y}) - (1-y)\log(1-\hat{y}) \tag{1}$$

Loss function:

$$\begin{cases} -log(h_w(x)), & \text{if y = 1} \\ -log(1-h_w(x)), & \text{if y = 0} \end{cases} \tag{2}$$

Gradient Descent:

$$w_j \leftarrow w_j - \gamma\frac{\partial J}{\partial w_j} \tag{3}$$

$$\frac{\partial}{\partial w_0}J_{MSE}(w) = \frac{1}{N}\sum_{i=1}^{N}((w_0 + w_1 x^{(i)}) - y^{(i)}) \quad (x_0 = 1)$$

$$\frac{\partial}{\partial w_1}J_{MSE}(w) = \frac{1}{N}\sum_{i=1}^{N}((w_0 + w_1 x^{(i)}) - y^{(i)})(x^{(i)})$$

Normal Equation: $w = (X^T X)^{-1} X^T y$.
not sensitive to the scale of input features
Gradient Descent

```
def gradient_descent_multi_variable(X, y, lr = 1e-5,
    number_of_epochs = 250):
    bias:number = 0
    weights = np.full((X.shape[1], 1), 0).astype(float)
    loss:List[number] = []
    N:number = X.shape[0]
    pred = X @ weights + bias
# pred:ℝ^{m×1} = X:ℝ^{m×n} × w:ℝ^{n×1} + bias:number
    for e in range(number_of_epochs):
        pred = X @ weights + bias
        g_w = (1 / N) * (X.T @ (pred - y))
# ∂/∂w1 J_MSE(w) = 1/N X^T((Xw + bias) − y)
        g_b = (1 / N) * np.sum(pred - y)
# ∂/∂w0 J_MSE(w) = 1/N Σ_{i=1}^{N}((Xw + bias) − y)
        bias -= lr * g_b
# w0 ← w0 − γ ∂J_MSE(w)/∂w0
        weights -= lr * g_w
# w1 ← w1 − γ ∂J_MSE(w)/∂w1
        loss.append(mean_squared_error(y, pred))
    return bias, weights, loss
```

- **L2 Regularization (Ridge)**
  - Penalty: $\lambda\sum_{i=1}^{d} w_i^2$
  - Effect: Shrinks all weights toward zero but rarely to exactly zero
  - Geometry: Elliptical constraint region
  - Closed-form Solution: Yes, normal equation
  - Differentiability: Smooth everywhere → easy gradient-based optimization
  - Use Cases: Multicollinearity handling; when you want small weights without feature elimination
- **L1 Regularization (Lasso)**
  - Penalty: $\lambda\sum_{i=1}^{d} |w_i|$
  - Effect: Encourages sparsity; many weights become exactly zero
  - Geometry: Diamond-shaped constraint region
  - Closed-form Solution: No; solved via iterative methods (e.g. coordinate descent)
  - Differentiability: Non-smooth at zero
  - Use Cases: Automatic feature selection; sparse model

Model evaluation Metrics

- **Accuracy**: $\frac{TP+TN}{TP+FN+FP+TN}$. Suitable for balanced data
- **Precision**: $\frac{TP}{TP+FP}$. maximize if FP is costly
- **Recall**: $\frac{TP}{TP+FN}$. maximize recall if FN is dangerous
- **F1-score**: $\frac{2}{\frac{1}{P}+\frac{1}{R}}$. when care about both precision and recall

Decision Trees: Classification

```
DTL(examples, attributes, default):
    if (examples = ∅): return default
    if (∀e ∈ example, e has the same classification c):
        return c
    if (attributes = ∅): return mode(examples)
    best = choose_attribute(attributes, examples)
    tree = new decision tree with root test best
    for v_i of best do:
        examples_i = {e|e ∈ examples, e.best = v_i}
        subtree = DTL(examples, attributes \ best,
                      mode(examples))
        tree.add(v_i: subtree)
choose_attribute(attributes, examples):
    best_gain = -∞
    best_attr = None
    for attr in attributes:
        gain = information_gain(attr, examples)
        if gain > best_gain:
            best_gain = gain
            best_attr = attr
    return best_attr
```

For data set contains boolean outputs,

$$I(P(+), P(-)) = -\frac{p}{p+n}log_2\frac{p}{p+n} - \frac{n}{p+n}log_2\frac{n}{p+n}$$

where $0 \leq \mathbb{R}_I \leq 1$. However for non-binary variables the entropy can be greater than 1
Information gain Information gain = entropy of this node - entropy of children nodes

$$IG(A) = I(\frac{p}{p+n}, \frac{n}{p+n}) - remainder(A)$$

Initial $I = 1$

$$remainder(A) = \sum_{i=1}^{v}\frac{p_i + n_i}{p+n}I(\frac{p_i}{p_i+n_i}, \frac{n_i}{p_i+n_i})$$

Multiclass Classification:
One-vs-One: Train $C(C-1)/2$ classifiers.
One-vs-Rest: Train C classifiers.
Binary Cross Entropy

$$BCE(y, \hat{y}) = -ylog(\hat{y}) - (1-y)log(1-\hat{y})$$

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

$\sigma(x) = \frac{1}{1+e^{-x}}$ is Not convex, but $-log(\sigma(x))$ is convex
Hypothesis function

$$h_w(x) = \sigma(w_0, w_1x_1 + w_2x_2)$$

Weight Update

$$w_j \leftarrow w_j - \gamma\frac{\partial J_{BCE}(w_0, w_1, \dots)}{\partial w_j}$$

Loss function derivative

$$\frac{\partial J_{BCE}(w_j)}{\partial w} = \frac{1}{N}\sum_{i=1}^{N}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)}$$

Feature Transformation Modify the original features of a dataset to make them more suitable for modeling.

- **Feature engineering**
  - **Polynomial features**: $z = x^k, k$ is the polynomial degree.
  - **log feature**: $z = \log(x)$:compress large values into a smaller range while preserving the relative order of the data, reducing the skew of the data distribution
  - **Exp. feature**: $z = e^x$:emphasize large values, or to Generating convex basis expansions
- **Feature scale** may improve the classifier and hence the test accuracy
  - Min-max scaling: $z_i = \frac{x_i - min(x_i)}{max(x_i) - min(x-i)}$, to [0, 1]
  - mean normalization $z_i = \frac{x_i - \mu_i}{\max(x_i) - \min(x_i)}$, roughly maps to $[-0.5, 0.5]$.
    * improve distance-based model performance
  - standardization: $z_i = \frac{x_i - \mu_i}{\sigma_i}$, transformed data has mean of 0 and SD of 1

SVM: transformed feature vector: $\mathbb{R}^d \mapsto \mathbb{R}^M$. allows for the parameters to be represented as a linear combination of the training data.
Maximize the margin. Vectors that are on the margin become supporting vectors
linearly separable data: hard-margin SVM handles noise in the data better than logistic classifiers

$$x = [x_1, x_2, x_3]^\top \mapsto \phi_{M_2}(x) = [x_1, x_2, x_3, x_1^2, x_2^2, x_3^2, x_1x_2, x_1x_3, x_2x_3]^\top.$$

$$h_w^\phi(x) = w_0\phi(x)_0 + w_1\phi(x)_1 + \dots + w_M\phi(x)_M, \quad \phi(x)_0 = 1$$
$$h_w^\phi(x) = w^T\phi(x)$$

Dual hypothesis:
$h_\alpha^\phi(x) = \sum_{j=1}^{N}\alpha_j\phi^T(x^{(j)})\phi(x)$, let $k_\phi(u,v) = \phi^T(x^{(j)})\phi(x) \implies h_\alpha^\phi(x) = \sum_{j=1}^{N}\alpha_j k_\phi(u,v)$
For n degree polynomial, $k_{Pn}(u,v) = \phi_{Pn}^T(x^{(j)})\phi_{Pn}(x) = (u^Tv)^n$

Gaussian kernel: $k_{RBF}(u,v) := e^{-\frac{||u-v||^2}{2\sigma^2}}$
feature map ϕ_{RBF} can map to infinite-dim space, cannot be computed explicitly.
Small σ^2: pointy; large σ^2: flat
High dimensionality/Highly correlated features: Use PCA to reduce the dimensionality of the data by finding a new set of orthogonal axes that capture the maximum variance.
K-means clustering: K-means relies on Euclidean distance and assumeslinearly separable clusters

```
# Initialize K cluster centroids randomly from the dataset
    for k = 1 in K clusters:
        μ_k = random(x ∈ D) // Pick a random data point x
              from dataset D as the initial centroid
# Repeat until convergence (e.g., centroids stop moving)
    while (! convergence):
# Assignment step: assign each point to the nearest centroid
        for i = 1 in m:
            c_i = argmin_k ||x^(i) − μ_k||^2
            # Assign point x^(i) to the closest cluster
                (minimizing distance, Euclidean / Manhattan)
# Update step: move centroids to the mean of assigned points
        for k = 1 in K:
            μ_k = 1/|{x^(i)|c^(i) = k}| Σ_{x∈{x^(i)|c^(i)=k}} x
# Recompute centroid μ_k as the mean of all points assigned
    to cluster k
```

Distortion:
$J(c^{(1)}, c^{(2)}, \dots, c^{(N)}, \boldsymbol{\mu_1}, \boldsymbol{\mu_2}, \dots, \boldsymbol{\mu_K}) = \frac{1}{N}\sum_{i=1}^{N}||x^{(i)} - \boldsymbol{\mu_{c^{(i)}}}||$
Each step in the K-Means algorithm never increases distortion
Mean-centred Data
mean feature vector over $\{x^{(i)}\}$ **Dimension reducation:**
SVD decomp: $A^{d\times N} = U^{d\times d}\Sigma^{d\times N}V^{T(N\times N)}$

Theorem 0.1 *given a mean-centred data matrix $\hat{X}^T$, the values $\frac{\sigma_j^2}{N-1}$ is the variance of the data in the basis defined by the vectors $u^{(j)}$*

Variance: $V_p[X] = E_p[(X - E_p(X))^2]$
Choose minimum r, to explaine a variance

$$\rightarrow \frac{\sum_{i=1}^{r}\sigma_i^2}{\sum_{i=1}^{N}\sigma_i^2} \geq a \rightarrow \dots = \frac{\sum_{i=1}^{N}||\hat{x_i}^{(i)} - \tilde{x_i}^{(i)}||^2}{\sum_{i=1}^{N}||\hat{x}^{(i)}||^2} \leq 1 - a$$

Principal Component Analysis: Statistics application of SVD. Capture components that maximize the statistical variations of the data. Same idea, but uses sample covariance matrix as an input

- **Classification**
  - **Type of Feedback:** Supervised learning
  - **Input:** Input–output pairs $\{(x^{(i)}, y^{(i)})\}$
  - **Output:** Model $h(x) \rightarrow \hat{y}$
  - **Methods:** Hyperplane-based (e.g. SVM), probability-based (e.g. logistic regression)
  - **Number of Classes:** Defined by the dataset
- **Clustering**
  - **Type of Feedback:** Unsupervised learning
  - **Input:** Data points only $\{x^{(i)}\}$
  - **Output:** Group assignments (e.g. $x^{(1)} \rightarrow$ cluster 5)
  - **Methods:** Distance-based algorithms
  - **Number of Clusters:** Chosen by practitioner*

Generalization
- In supervised learning, and machine learning in general, the model's performance on unseen data is all we care about. This ability to perform well on new, unseen data is known as the model's generalization capability.
- There are two factors that affect generalization:
  - Dataset quality
    * Relevance
    * Noise (irrelevant or incorrect data)
    * Balance (for classification): Balanced datasets ensure that all classes are adequately represented e.g. ratio
  - Data quantity
    * In general, having more data typically leads to better model performance
    * Extreme case: if the dataset contains every possible data point, memorize
  - Model complexity
    * Size and expressiveness of the hypothesis class.
    * Intricacy of the relationships between input and output variables that the model can capture.

Hyperparameter Hyperparameters are settings that control the behavior of the training algorithm and model but are not learned from the data. They need to be set before the training process begins. Such as
- Learning rate
- Feature transformations
- Batch size and iterations in mini-batch gradient descent

Perceptron Learning algorithm:

```
initialize w
while (i < max_iteration | !convergence):
    for (x_i, y_i) in D:
        ŷ = h_w(x) # W^{N+1×1} · [1, x_i^T]^T
        if x_i is misclassified:
            learning_error = (y_i − ŷ_i)
            w += γ(y_i − ŷ_i)x_i
```

On data not linearly separable The algorithm will not converge

$$\hat{y} = sgn(\mathbf{w}^T x) \quad (Range \in -1, 1)$$

Case 1: $y = +1, \hat{y} = -1$: $\hat{y} < 1 \implies \mathbf{w}^T x < 0 \implies \mathbf{w}x < 0 \implies |\mathbf{w}||x|\cos\theta < 0 \implies \theta \in (\frac{\pi}{2}, \pi)$
But we want the opposite, ie $\hat{y} = +1 \implies \theta \in [0, \frac{\pi}{2}] \implies$ Reduce θ
$\mathbf{w}' = \mathbf{w} + x \implies \theta' < \theta$
Case 2: $y = -1, \hat{y} = +1$ Similarly, $\mathbf{w}' = \mathbf{w} - x \implies \theta' > \theta$

$$\mathbf{w}' \leftarrow \mathbf{w} + 2\gamma x, \mathbf{w}' \leftarrow \mathbf{w} + \gamma(+1 - (-1))x$$
$$\mathbf{w}' \leftarrow \mathbf{w} + \gamma(y - \hat{y})x$$

Trivial to show, both cases arrive at the same update rule.
Multilayer neural network: Single-layer NN is equivalent to a linear classifier
- **Depth (number of hidden layers L)** complexity grows roughly $\mathcal{O}(L)$ in layers.

- **Width (number of input features d)** per-layer cost grows roughly $\mathcal{O}(d)$ (or $\mathcal{O}(d \cdot n)$ if hidden size is n).

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \mathbf{W}^{[1]} = \begin{bmatrix} W_{11}^{[1]} & W_{12}^{[1]} \\ W_{21}^{[1]} & W_{22}^{[1]} \\ W_{31}^{[1]} & W_{32}^{[1]} \end{bmatrix}$$

$$\hat{y}^{[1]} = g^{[1]}(\mathbf{W}^{[1]T} x)$$

Forward Propagation: For n layers,

$$\hat{y}^{[n]} = g^{[n]}(W^{[n]^T} \hat{y}^{[n-1]})$$

$$\hat{y} = g^{[n]}(W^{[n]^T}(g^{[n-1]}(W^{[n-1]^T} ... g^{[2]}(W^{[2]^T} g^{[1]}(W^{[1]^T} \mathbf{x})))))$$

Example:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \mathbf{W}^{[1]} = \begin{bmatrix} W_{11}^{[1]} & W_{12}^{[1]} & W_{13}^{[1]} \\ W_{21}^{[1]} & W_{22}^{[1]} & W_{23}^{[1]} \\ W_{31}^{[1]} & W_{32}^{[1]} & W_{33}^{[1]} \end{bmatrix}, \mathbf{W}^{[2]} = \begin{bmatrix} W_{11}^{[2]} & W_{12}^{[2]} \\ W_{21}^{[2]} & W_{22}^{[2]} \\ W_{31}^{[2]} & W_{32}^{[2]} \end{bmatrix}$$

$$\hat{\mathbf{y}}^{[1]} = g^{[1]}(\mathbf{W}^{[1]T} \mathbf{x}), \qquad \hat{\mathbf{y}} = g^{[2]}(\mathbf{W}^{[2]T} \hat{\mathbf{y}}^{[1]}) = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix}.$$

Multi class classification Softmax function

$$\mathbf{x}^{N\times 1} \to \mathbf{z}^{C\times 1}$$

$\forall z_i \in \mathbf{z}, i \in [1, C], z_i \in [0, 1]$, as z_i indicates the probability of $\mathbf{x}$ belonging to class C_i. To map the neuron network output from $\mathbb{R}$ to $[0, 1]$, we use softmax function:

$$g(z_i) = \frac{e^{z_i}}{\sum_{i=1}^{C} z_i} \quad \text{(Non-convex)}$$

where $\mathbb{R}_g = [0, 1]$, and $\sum_{i=1}^{C} g(z_i) = 1$, indicating a complete probability distributon.

Chain rule:

$$l = h(g(f(x))) \xrightarrow{\text{deriv.}} \frac{\partial l}{\partial x} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial x}$$

where $z = f(x), y = g(z), l = h(y)$

Backpropagation:

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]T} \mathbf{x}, \qquad \mathbf{a}^{[1]} = g^{[1]}(\mathbf{z}^{[1]}),$$

$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]T} \mathbf{a}^{[1]}, \qquad \hat{\mathbf{y}} = g^{[2]}(\mathbf{z}^{[2]}).$$

$$\hat{\mathbf{y}} = g^{[2]}\Big(\mathbf{W}^{[2]T}\, g^{[1]}(\mathbf{W}^{[1]T} \mathbf{x})\Big).$$

Given $L = \frac{1}{2}(\hat{y} - y)^2$,

- $\frac{dL}{d\hat{y}} = \hat{y} - y$
- $\frac{dL}{dz_3} = (\hat{y} - y)\, g'(z_3)$
- $\frac{dL}{da_1} = (\hat{y} - y)\, g'(z_3)\, w_5, \quad \frac{dL}{da_2} = (\hat{y} - y)\, g'(z_3)\, w_6$
- $\frac{dL}{dz_1} = (\hat{y} - y)\, g'(z_3)\, w_5\, g'(z_1), \quad \frac{dL}{dz_2} = (\hat{y} - y)\, g'(z_3)\, w_6\, g'(z_2)$
- $\frac{dL}{dx_1} = (\hat{y} - y)\big[g'(z_3)w_5 g'(z_1)w_1 + g'(z_3)w_6 g'(z_2)w_2\big]$
- $\frac{dL}{dx_2} = (\hat{y} - y)\big[g'(z_3)w_5 g'(z_1)w_3 + g'(z_3)w_6 g'(z_2)w_4\big]$

Pytorch: Loss function:

```
loss_function_MSE = torch.nn.MSELoss()
loss_function_BCE = torch.nn.BCELoss()
loss_function_CE = torch.nn.CrossEntropyLoss()
```

- **Optimizers (torch.optim)**
  - `torch.optim.SGD` (Stochastic Gradient Descent)
  - `torch.optim.Adam`
- **Important functions**
  - `optimizer.zero_grad()`: clear all gradients before computing new ones
  - `optimizer.step()`: update weights and notify optimizer that one step is done
- **Activation functions:**
  - `ReLU(x)` = $x \geq 0?x : 0$ — default hidden-layer activation; sparse output, alleviates vanishing gradients. Polygon Classification boundary
  - `LeakyReLU(x)` = $x \geq 0?x : \alpha x$ — similar to ReLU but with small slope α for $x < 0$; avoids "dead" neurons
  - $\sigma(x) = \frac{1}{1+e^{-x}}$ — maps to $(0, 1)$; common for binary outputs or gating. Smooth classification boundary
  - `tanh(x)` = $2\sigma(2x) - 1$ — zero-centered in $(-1, 1)$; useful in RNNs and when zero mean is preferred

```
import torch, torch.nn as nn, torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
# 1) Minimal CNN definition
class CNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Conv2d(12, 16, kernel_size=3, padding=1),
            nn.GELU(),
            nn.AdaptiveAvgPool2d((3,3)),
            nn.Flatten(),
            nn.Linear(16*3*3, 1)
        )
    def forward(self, x):
        return self.net(x)
# 2) DataLoader, model, loss, optimizer
train_ds  = TensorDataset(X_train_tensor, y_train_tensor)
train_ld  = DataLoader(train_ds, batch_size=64,
    shuffle=True)
model     = CNN().to(device)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-4)
# 3) One training epoch
model.train()
for xb, yb in train_ld:
    xb, yb = xb.to(device), yb.to(device)
    optimizer.zero_grad()
    loss = criterion(model(xb), yb)
    loss.backward()
    optimizer.step()
```

Convolution and pooling:

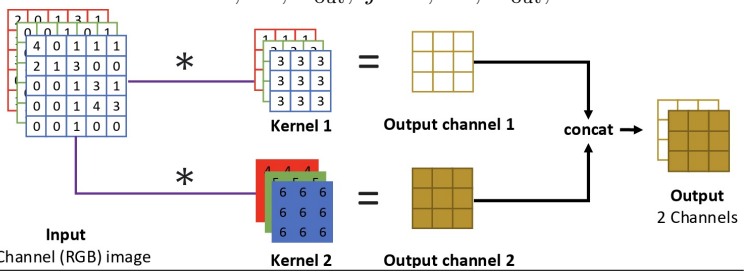
$$X_{in}^{H_{in}\times W_{in}} \xrightarrow{\text{padding}} \tilde{X}^{(H_{in}+2P)\times(W_{in}+2P)}$$

$$H_{\text{out}} = \left\lfloor \frac{H_{\text{in}} + 2P - H_{\text{ker}}}{S} \right\rfloor + 1, \quad W_{\text{out}} = \left\lfloor \frac{W_{\text{in}} + 2P - W_{\text{ker}}}{S} \right\rfloor + 1.$$

Let $\tilde{X} \in \mathbb{R}^{H_{out}\times W_{out}}$ be the zero-padded input. Then

$$Y_{i,j} = \sum_{u=1}^{H_{\text{ker}}} \sum_{v=1}^{W_{\text{ker}}} K_{u,v}\, \tilde{X}_{(i-1)S+u,\,(j-1)S+v},$$

$$i = 1, \ldots, H_{\text{out}},\ j = 1, \ldots, W_{\text{out}},$$



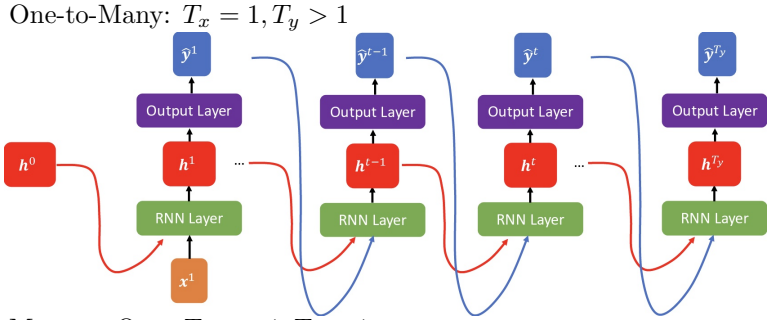

```
conv_layer = torch.nn.Conv2d(in_channels=3,out_channels=2,
                kernel_size=3,stride=1,padding=0)
```

- **Vanishing gradient**
  - Small gradients get repeatedly multiplied → approach zero
  - Mitigation: change activation functions (e.g. use ReLU variants)
- **Exploding gradient**
  - Large gradients get repeatedly multiplied → overflow
  - Mitigation: gradient clipping (clip gradients to $[-c, c]$)
- **Dropout**
  - Randomly zero out activations during training
  - Mitigates overfitting by preventing co-adaptation of neurons
- **Early stopping**
  - Monitor validation loss and stop when it ceases to improve
  - Prevents overfitting by halting before the model fits noise

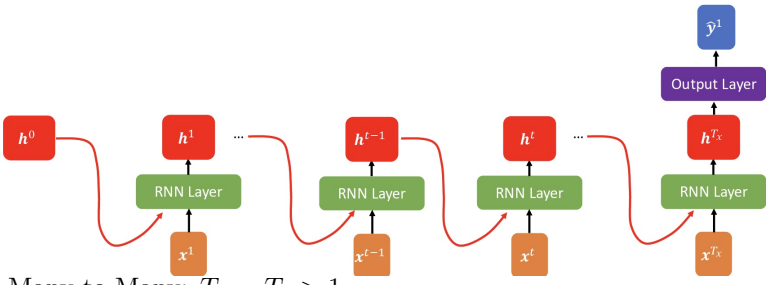
RNN:

- **Capture contextual information** — RNNs can aggregate information over the sequence.
- **Sequential dependency** — the prediction at time step t must wait until all previous steps have been completed (not parallelism-friendly).

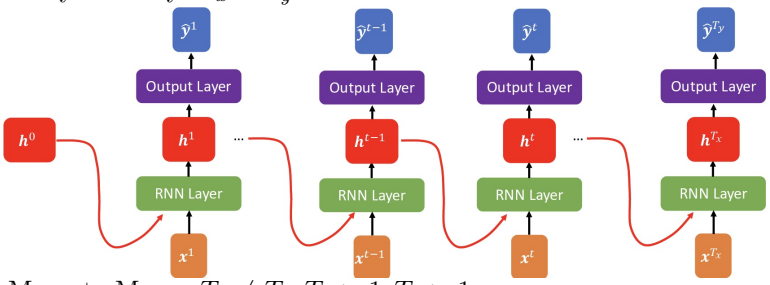
One-to-Many: $T_x = 1, T_y > 1$



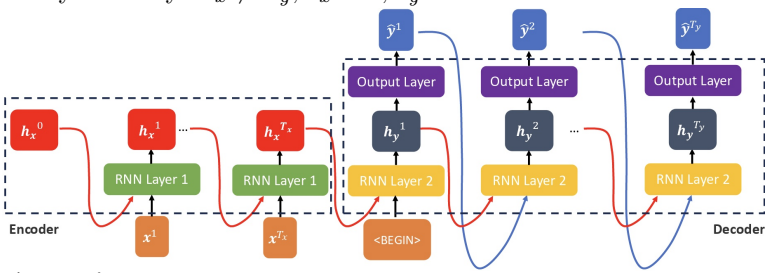
Many-to-One: $T_x => 1, T_y = 1$



Many-to-Many: $T_x = T_y > 1$



Many-to-Many: $T_x \neq T_y, T_x > 1, T_y > 1$



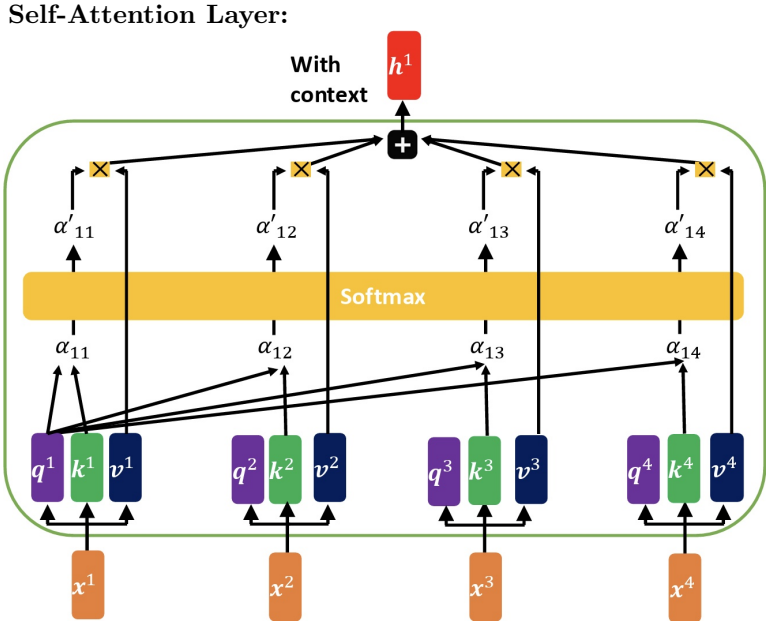
Attention:

$$s_{ij} = \text{score}(x_i, x_j) = \frac{(W^Q x_i)^\top (W^K x_j)}{\sqrt{d_k}} \quad \text{(raw attention score)}$$

$$a_{ij} = \mathbf{k}^j \cdot \mathbf{q}^i = (\mathbf{k}^j)^T \mathbf{q}^i = \frac{e^{s_{ij}}}{\sum^k e^{s_{ik}}}$$

$$q_i = W^Q x_i, \quad k_j = W^K x_j, \quad v_j = W^V x_j$$

Self-Attention Layer:



- **Raw scores**

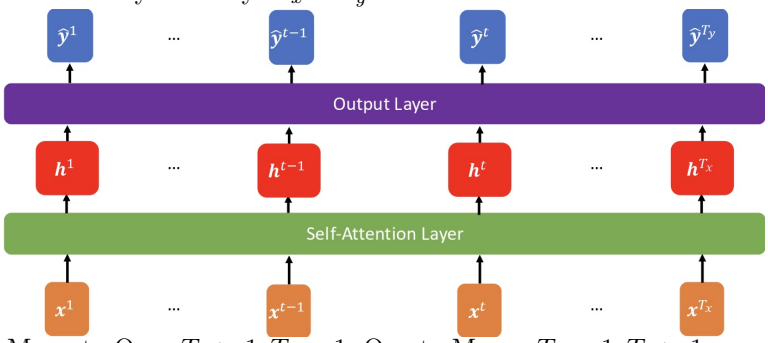
$$s_{ij} = q_i^\top k_j = (k_j)^\top q_i$$

- **Attention weights (softmax)** $a_{ij} = \frac{\exp(e_{ij})}{\sum_l \exp(e_{il})}$
- **Positional encoding**

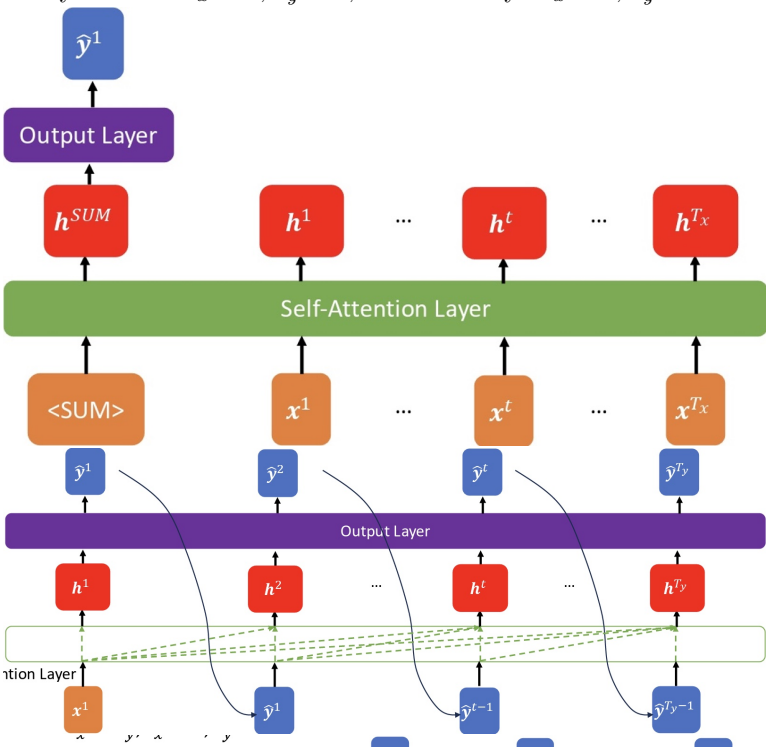
$$x_i' = x_i + PE_i, \quad PE_{i,2k} = \sin\left(i/10000^{2k/d}\right),$$
$$PE_{i,2k+1} = \cos\left(i/10000^{2k/d}\right)$$

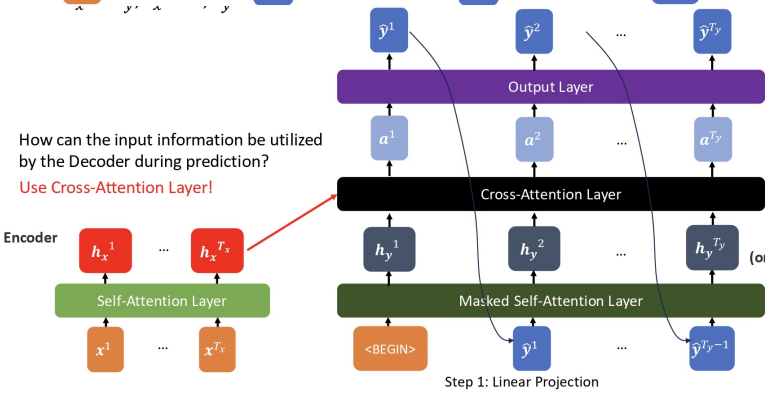
- Attention Neural Networks
  - Masked Self-Attention Linear projections to queries/keys/values, scaled dot-product with causal mask to prevent "seeing" future tokens.
  - Cross-Attention Same mechanism, but queries from decoder attend over encoder's key/value projections.
  - Transformer Architecture Stacks of encoder/decoder blocks combining (masked) self-attention, cross-attention, and feed-forward sublayers.

ANN: Many-to-Many: $T_x = T_y > 1$

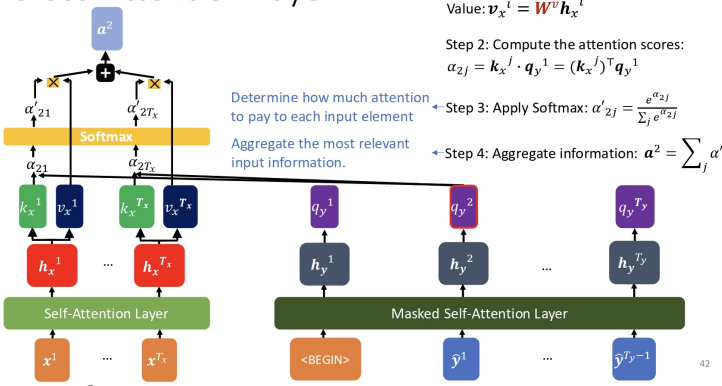


Many-to-One: $T_x > 1, T_y = 1$; One-to-Many: $T_x = 1, T_y > 1$





Cross-Attention Layer



Step 1: Linear Projection
Transform input vectors into Query, Key, and Value
Key: $\boldsymbol{k}_x{}^i = \boldsymbol{W}^k \boldsymbol{h}_x{}^i$ Query: $\boldsymbol{q}_y{}^i = \boldsymbol{W}^q \boldsymbol{h}_y{}^i$
Value: $\boldsymbol{v}_x{}^i = \boldsymbol{W}^v \boldsymbol{h}_x{}^i$

Step 2: Compute the attention scores:
$\alpha_{2j} = \boldsymbol{k}_x{}^j \cdot \boldsymbol{q}_y{}^1 = (\boldsymbol{k}_x{}^j)^\top \boldsymbol{q}_y{}^1$

Determine how much attention to pay to each input element

Step 3: Apply Softmax: $\alpha'_{2j} = \frac{e^{\alpha_{2j}}}{\sum_j e^{\alpha_{2j}}}$

Aggregate the most relevant input information.

Step 4: Aggregate information: $\boldsymbol{a}^2 = \sum_j \alpha'_{2j} \boldsymbol{v}_x{}^j$

Transformer: