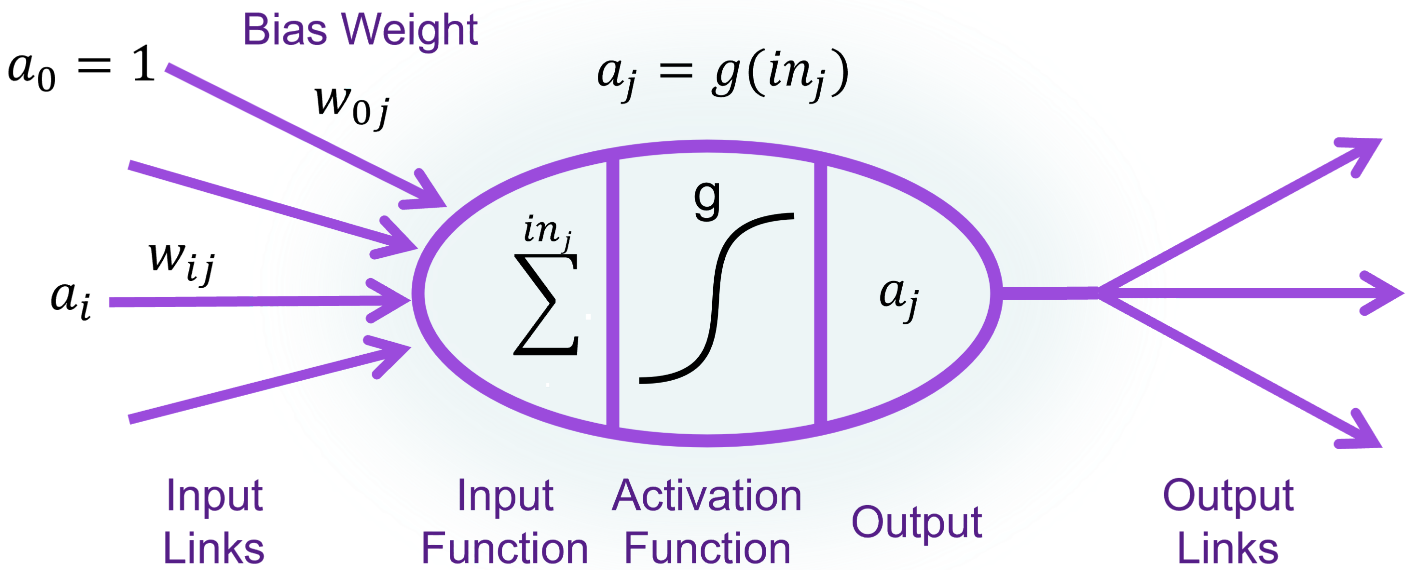
Neural Network Development History

- 1950s-1960s: Early Foundations
  - McCulloch & Pitts (1943): mathematical neuron model
  - Rosenblatt's Perceptron (1958): first trainable network
  - Minsky & Papert (1969): limitations (XOR problem) → AI Winter
- 1970s–1980s: First Revival
  - Werbos (1974); Rumelhart, Hinton, Williams (1986): Backpropagation
  - Hopfield Networks (1982): associative memory
  - Renewed optimism but limited by hardware
- 1990s: Consolidation
  - LeCun's CNN (LeNet, 1989): digit recognition
  - Elman, Jordan: Recurrent Neural Networks
  - Symbolic AI still dominated mainstream
- 2000s: Deep Learning Foundations
  - Better hardware (GPUs) + large datasets
  - Hinton (2006): Deep Belief Networks (unsupervised pretraining)
  - Connectionism regains attention
- 2010s: Deep Learning Boom
  - ImageNet (2012): AlexNet breakthrough
  - RNNs, LSTMs, GRUs → speech & translation
  - Transformers (2017): revolutionized NLP
- 2020s: Scaling & Foundation Models
  - Large Language Models (GPT, BERT, etc.)
  - Multimodal AI: vision, text, speech integration
  - Connectionism dominates AI research & industry

Neural Network Models

- a collection of units (neurons) connected together
- The properties of the network are determined by its topology and the properties of the neurons.
- Roughly speaking, the neuron fires when a linear combination of its inputs exceeds some (hard or soft) threshold.



- $in_j = \sum_{i=0}^{n} w_{ij} a_i$
- $out_j = g(in_j)$
- $a_j = g(\sum_{i=0}^{n} w_{ij} a_i)$

Activation function

ReLU function

$$ReLU(x) = max(0, x)$$

- an abbreviation for rectified linear unit
- Commonly used

Softplus function

$$Softplus(x) = \log(1 + e^x)$$

- A smooth version of the ReLU function

Logistic or Sigmoid function

$$Logistic(x) = \frac{1}{1 + e^{-x}}$$

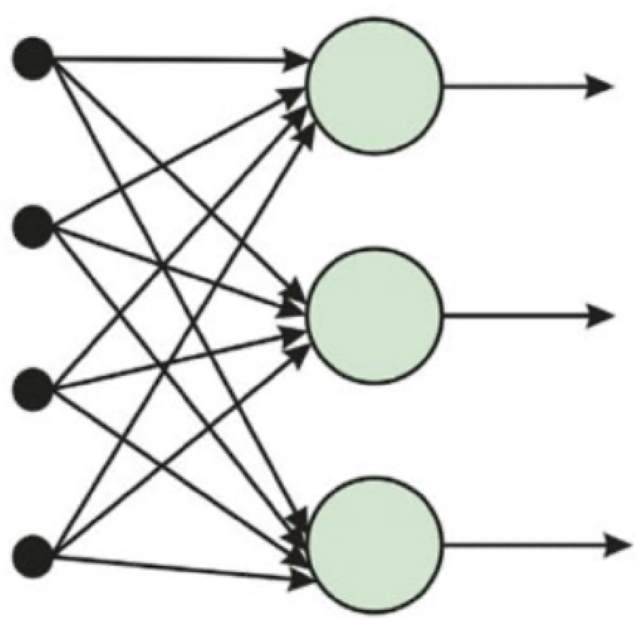
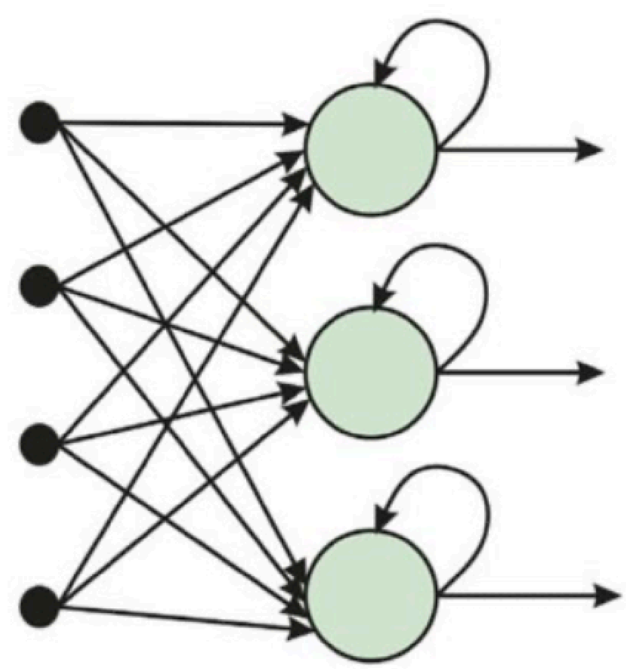
- Non-linear, can represent a nonlinear function

Tanh function

$$tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Topology of a neural network

- Feed-forward network (FFN):
    - Every node receives inputs from "upstream" nodes and delivers output to "downstream" nodes.
    - There are no loops.
    - FFN represents a function of its current inputs, thus it has no internal state other than the weights themselves.
- Recurrent Network (RNN):
    - A recurrent network feeds its outputs back into its own inputs.
    - In a recurrent network, the neuron values can eventually settle down, keep cycling, or behave unpredictably.
    - can support short-term memory

FFN	RNN
	

Training Process

- Go through each training sample.
- If correctly classified → do nothing.
- If misclassified → update the weights:
- $w_i \leftarrow w_i + \alpha(y - \hat{y})x_i$

Perceptron for Binary Classification

- A perceptron separates data into two classes with a hyperplane.
- if $w \cdot x \geq 0 \rightarrow 1$
- if $w \cdot x \leq 0 \rightarrow 0$

Learning Rules

Aspect	Perceptron Learning Rule	Gradient Descent (with Sigmoid)
Activation function	Hard threshold (계단 함수) $Threshold(z) = 1 \text{ if } z \geq 0,\ 0 \text{ otherwise}$	Sigmoid (연속 함수) $h_w(x) = \frac{1}{1+e^{-w \cdot x}}$
Output	0 또는 1	0과 1 사이의 실수 값
Loss function	없음 (틀리면 조정, 맞으면 유지) 규칙 기반 학습	$L = (y - h_w(x))^2$ (L2 loss) 또는 Cross-Entropy (실무에서 자주 사용)
Update rule	틀렸을 때만: $w \leftarrow w + \alpha(y - h_w(x))x$	경사하강법: $w \leftarrow w + \alpha(y - h_w(x)) \cdot h_w(x)(1 - h_w(x)) \cdot x$
Why derivative?	Hard threshold는 미분 불가능 → 단순 규칙 사용	Sigmoid는 연속적이고 미분 가능 → Loss 함수의 기울기(gradient)를 따라 업데이트. 여기서 $h_w(x)(1 - h_w(x))$ 항은 sigmoid의 도함수에서 나온 것.
Interpretation	틀리면 정답 방향으로 한 걸음 이동	Loss가 줄어드는 방향으로 점진적으로 이동

Feadforward NN, FNN

- a multilayer perceptron network

- one input layer, N hidden layers, `N >= 1`, and one output layer.
- Except for the input layer, each layer has a same activation function `g`.
- The final output is represented by a vector function of inputs and weights.
- If it has three layers, Shallow Neural Network, otherwise Deep Neural Network.

Traning a FNN

- Forward
  - Activation passing from the input layer to the output layer
  - **Calculate the output**
- Backward
  - Errors propagating backward from the output layer to the input layer
  - **Update weights**

Forward phase

- **Activation of each node is computed in two steps:**
  i. **Weighted sum (in):** sum of activations from the previous layer, multiplied by weights.
  ii. **Apply activation function g:** pass the weighted sum through g to produce the node's activation.
- **Process:** propagate activations layer by layer towards the output layer.
- **Output value (example with 2 layers):**
  - $h_w(x) = g^{(2)}\big(W^{(2)}g^{(1)}(W^{(1)}x)\big)$

Backward phase

- **Loss function:** choose squared error loss (L2)
  - $L_2(y, \hat{y}) = (y - \hat{y})^2$
- **Prediction:**
  - $\hat{y} = h_w(x)$
- **Gradient descent:** compute gradient of the loss with respect to weights, then update weights along the negative gradient direction.
  - $w_{i,j} \leftarrow w_{i,j} - \alpha \cdot gradient_{w_{i,j}}$
- **Example: sigmoid activation:**
  - $\hat{y} = \frac{1}{1+e^{-w\cdot x}}$
  - Gradient of the loss:
    - $gradient_{w_{i,j}} = \frac{\partial}{\partial w_{i,j}} Loss(h_w) = 2(y - h_w(x)) \cdot \Big(-\frac{\partial}{\partial w_{i,j}} h_w(x)\Big)$
  - **Chain rule applied:**

        - $\frac{\partial g(f(x))}{\partial x} = g'(f(x)) \cdot f'(x)$
- **Example: Gradient derivation for sigmoid**
    - **Weighted input**
        - $W \cdot X = w_{1,3}x_1 + w_{2,3}x_2 + w_{0,3}x_0$
        - (where $x_0 = 1$ for the bias)
    - **Gradient of the loss**
        - $gradient_{w_{i,j}} = \frac{\partial}{\partial w_{i,j}} Loss(h_w) = 2(y - h_w(x)) \cdot \left(- \frac{\partial}{\partial w_{i,j}} h_w(x) \right)$
    - **Derivative of sigmoid output**
        - $\frac{\partial}{\partial w_{i,j}} h_w(x) = h_w(x)(1 - h_w(x)) \cdot \frac{\partial}{\partial w_{i,j}} (W \cdot X)$
            - $\frac{\partial}{\partial w_{i,j}} \left(\frac{1}{1+e^{-WX}} \right)$
            - $= \left(\frac{1}{1+e^{-WX}} \right) \left(1 - \frac{1}{1+e^{-WX}} \right) \cdot \frac{\partial}{\partial w_{i,j}} (WX)$
            - $= h_w(x)\big(1 - h_w(x)\big) \cdot \frac{\partial}{\partial w_{i,j}} (WX)$
    - **Derivative of weighted input**
        - $\frac{\partial}{\partial w_{0,3}} (W \cdot X) = x_0 = 1$
        - $\frac{\partial}{\partial w_{1,3}} (W \cdot X) = x_1$
        - $\frac{\partial}{\partial w_{2,3}} (W \cdot X) = x_2$
    - **Weight update rule**
        - General form: $w_{i,j} \leftarrow w_{i,j} - \alpha \cdot gradient_{w_{i,j}}$
        - $w_{0,3} \leftarrow w_{0,3} + \alpha(y - h_w(x))h_w(x)(1 - h_w(x))$
        - $w_{1,3} \leftarrow w_{1,3} + \alpha(y - h_w(x))h_w(x)(1 - h_w(x))x_1$
        - $w_{2,3} \leftarrow w_{2,3} + \alpha(y - h_w(x))h_w(x)(1 - h_w(x))x_2$

Backward phase Steps

1. **Select a loss function**
    - For example, squared error loss:
    - $L(y, \hat{y}) = (y - \hat{y})^2, \quad \hat{y} = h_w(x)$
2. **Choose an activation function**
    - Suppose we use a sigmoid:
    - $h_w(x) = \frac{1}{1+e^{-W \cdot X}}$
3. **Calculate the error at the output node**
    - The delta (error term) at the output is
    - $\Delta_{out} = 2(\hat{y} - y) \cdot g'(in_{out})$
4. **Calculate the error at hidden nodes**
    - A hidden unit may connect to multiple nodes in the next layer.
    - Therefore, its error is the weighted sum of all deltas it feeds into, scaled by its own derivative:

- $\Delta_i = g'(in_i) \sum_j w_{i,j} \Delta_j$
- The summation appears because the hidden node's output influences several downstream nodes, and all those error signals must be aggregated.

5. **Update the weights with gradient descent**
    - The gradient with respect to weight $w_{i,j}$ is simply the input times the delta:
    - $\frac{\partial L}{\partial w_{i,j}} = a_i \Delta_j$
    - Update rule:
    - $w_{i,j} \leftarrow w_{i,j} - \alpha \, a_i \Delta_j$

Vanishing gradient

- The error signal are extinguished altogher as they are propagated back through the network
- In deep feedforward networks with `sigmoid/tanh` , repeated multiplication of small derivatives ($0 < g'(z) < 1$) during backpropagation causes the gradient to vanish.

Optimizer

- Training a neural network consists of modifying the network's parameters, minimizing the loss function on the training set.
- any kind of optimization algorithm could be used.
- modern neural networks are almost always trained with some variant of stochastic gradient descent (SGD). **Adam Optimizer**
- The optimiser is specified in the compilation step with tensorflow.

Recurrent NN, RNN

- units may take as input a value computed from their own output at an earlier step in the computation.
- have internal state, or memory: inputs received at earlier time steps affect the RNN's response to the current input.
- be used to perform more general computations.
    - to analyze sequential data in which a new input vector x_t arrives at each time step
- **Markov assumption**: the hidden state z_t of the network suffices to capture the information from all previous inputs.
    - $z_t = f(z_{t-1}, x_t)$
    - Once trained, this function represents a time-homogeneous process
    - The same update rule f_w applies at every time step, regardless of whether it’s the first input or the hundredth.

- RNNs are designed for sequential data.
- a hidden state that captures information from previous steps.
- suffer from vanishing/exploding gradients.
- Good for short-term dependencies.

Backpropagtion Through Time, BPTT

- gradient expression is recursive.
  - $\frac{\partial z_t}{\partial w_{z,z}}$
  - $= \frac{\partial}{\partial w_{z,z}} g_z(in_{z,t})$
  - $= g'_z(in_{z,t}) \frac{\partial in_{z,t}}{\partial w_{z,z}}$
  - $= g'_z(in_{z,t}) \frac{\partial}{\partial w_{z,z}} (w_{z,z} z_{t-1} + w_{x,z} x_t + w_{0,z})$
  - $= g'_z(in_{z,t}) \left(z_{t-1} + w_{z,z} \frac{\partial z_{t-1}}{\partial w_{z,z}} \right)$
    - $\frac{\partial z_t}{\partial W_{z,z}}$ includes $\frac{\partial z_{t-1}}{\partial W_{z,z}}$
- the gradient with run time being linear in the size of the network
- handled automatically by deep learning software systems.
- Iterating the recursion shows that the gradient at time T includes a term proportional to:
  - $w_{z,z} \prod_{t=1}^{T} g'_z(in_{z,t})$
- Since for sigmoid, tanh, and ReLU we have $g' \leq 1$, if $w_{z,z} < 1$ the RNN will suffer from the **vanishing gradient problem**.
- If $w_{z,z} > 1$, we may encounter the **exploding gradient problem**.

Long Short-Term Memory, LSTM

- **memory cell** is essentially copied from time step to time step.
- New information enters the memory by adding updates.
  - the gradient expressions do not accumulate multiplicatively over time.
- include **gating units**: vectors control the flow of information in the LSTM, elementwise multiplication of the corresponding information vector.
- a type of RNN designed to overcome vanishing gradient.
- use gates (input, forget, output) to control information flow.
- Capable of learning long-term dependencies.
- Widely used in NLP, speech recognition, and time series forecasting.

Gates in LSTM

- **Forget gate**: decides what information to discard from the cell state.

- **Input gate**: decides what new information to store in the cell state.
- **Output gate**: decides what information to output from the cell state.
  - similar role to the hidden state in basic RNNs.
- Update equations:
  - $f_t = \sigma(W_{x,f}x_t + W_{z,f}z_{t-1})$
    - Decides which parts of the previous cell state c_{t-1} should be kept or discarded.
  - $i_t = \sigma(W_{x,i}x_t + W_{z,i}z_{t-1})$
    - Determines how much of the new information from the current input x_t and the previous hidden state z_{t-1} should be added.
  - $o_t = \sigma(W_{x,o}x_t + W_{z,o}z_{t-1})$
    - Controls which parts of the current cell state c_t are exposed as the hidden state z_t.
  - $c_t = c_{t-1} \odot f_t + i_t \odot \tanh(W_{x,C}x_t + W_{z,C}z_{t-1})$
    - Cell state update
    - Past information (c_{t-1}) is partially retained through the forget gate.
    - New information is added through the input gate and $\tanh$.
    - Thus, c_t serves as the long-term memory of the LSTM.
  - $z_t = o_t \odot \tanh(c_t)$
    - Hidden state update
    - The cell state is normalized with $\tanh(c_t)$ and filtered by the output gate.
    - z_t is the hidden state passed forward to the next time step.

Gated Recurrent Unit, GRU

- Variant of RNN with gating mechanisms.
- Designed to capture long-term dependencies without complex architecture.
- a simpler alternative to LSTMs. (lightweight, effective RNN variant)
- Captures temporal dependencies (short & long).
- Combine input and forget gates into a single update gate.
- Require fewer parameters than LSTM, making them faster to train.
- Perform comparably to LSTMs in many tasks.
- Prevents vanishing gradient.
- Good balance between complexity & performance.
- Excels in time series forecasting tasks.
- Widely used in finance, energy and IoT.

Gates in GRU

- **Update gate (z)**: decides how much past information to keep

- **Reset Gate (r)**: decides how much past information to forget
- **Candidate hidden state ($\tilde{h}$)**: potential new memory
- **Final hidden state (h)**: weighted combination of old and new information.

GRU Workflow

- Reset gate (r)
  - Controls how much of the previous hidden state should be "forgotten."
  - A small value means most of the past memory is erased, while a large value means much of it is retained.
- Update gate (z)
  - Acts as a switch to decide whether to keep the previous state h_{prev} or replace it with the new candidate $\tilde{h}$.
  - If $z = 1$, the past is fully kept; if $z = 0$, it is completely replaced by the new candidate.
- Candidate state ($\tilde{h}$)
  - Combines the current input x_t with the reset-gated previous hidden state to generate the "candidate" new information.
- Final hidden state (h)
  - Blends the past and the candidate using the update gate z.
  - If z is large → the past memory dominates.
  - If z is small → the new candidate dominates.
- $h = (1 - z)\tilde{h} + zh_{prev}$

Comparison: RNN vs LSTM vs GRU

Attribute	RNN	LSTM	GRU
Architecture	Simple, hidden state	Complex, memory cell + 3 gates	Simplified, 2 gates (update/reset)
Information Flow	Stored in hidden state	Controlled by gates	Controlled by merged gates
Long-term Dependency	Weak (vanishing gradient)	Strong (gates solve vanishing gradient)	Strong (gates solve vanishing gradient)
Short-term Dependency	Strong	Strong	Strong

Attribute	RNN	LSTM	GRU
Number of Parameters	Few	Many	Fewer than LSTM
Training Speed	Fast	Slow	Fast
Performance	Good for short-term	Good for long-term	Efficient, similar to LSTM
Application Areas	Simple time series, basic NLP	NLP, speech, time series forecasting	Finance, IoT, energy, time series
Vanishing Gradient	Yes	No	No
Typical Use Cases	Text generation, simple prediction	Translation, speech recognition	Time series prediction, sensor data
Simplicity	Very simple, rarely used	More complex, expressive (3 gates)	Simpler than LSTM, fewer parameters
Expressiveness	Limited, struggles with long-term	High, handles very complex sequences	Moderate, good for moderate data size
Training Efficiency	Fast, but limited	Slower, better for complex data	Fast, efficient, similar performance
Trade-off	Simple but weak for long-term	Capacity for complex, long sequences	Simplicity vs. capacity