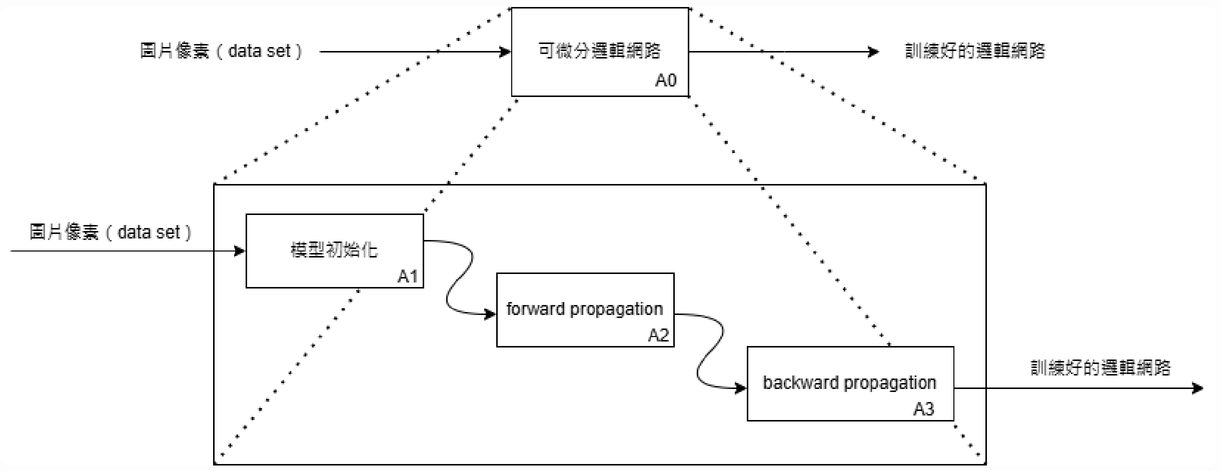
Differentiable Logic Network

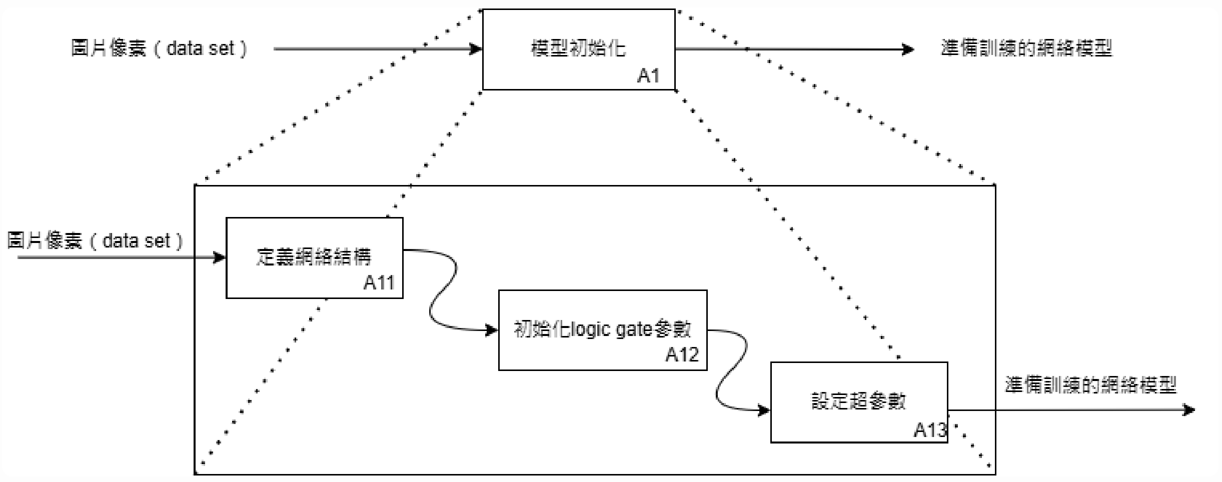
Differentiable Logic Network 深入研究: hackmd

IDEF0 可微分邏輯網路 系統階層式模組化架構

基本設計：一共包含三大部分，分別是「模型初始化」、「fordward propagation」和「backward propagation」，以圖片分類為例，
輸入：圖片像素；輸出：訓練好的邏輯網路
注意，A2和A3為訓練過程，通常會經過多次的迭代，而非單一過程



模型初始化(A1):



定義網路結構(A11):

- 確定網絡的總層數 L（例如 4 到 8 層）
- 固定每層的神經元數量，通常相等
- 每層的每個神經元與前一層的**兩個輸入**隨機連接

初始化Logic gate 參數(A12):

- 每個神經元都對應 16 種邏輯操作（如 AND、OR、XOR 等）
- 使用 Softmax 對 w_i 初始化每個logic gate被選擇機率(p_i)：

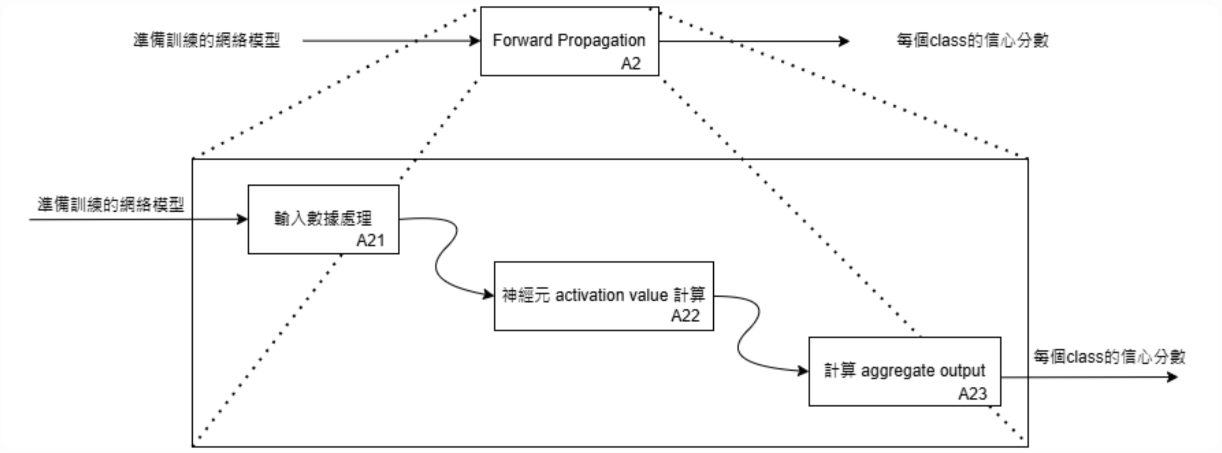
$$p_i = \frac{e^{w_i}}{\sum_{j=0}^{15} e^{w_j}}$$

w_i 是每個logic operation被選擇的優先程度，初始為常態分布隨機抽樣，代表每個logic gate的機率分布是均勻的，w_i 會在Backward Propagation中被更新，目的就是要改變每個神經元對Logic ate的選擇機率

定義超參數(A13):

- 初始化output layer之分組：將輸出層神經元分為 k 組(if k classes)，會在forward propagation 的aggregate output中使用到
- 定義learning rate, loss function (Softmax Cross-Entropy Loss)

Forward Propagation(A2):



輸入數據處理(A21):
若輸入是連續數據（如圖像像素值 $[0, 255]$），則進行nomilization，使a $\in [0, 1]$:

$$a = \frac{\text{輸入像素值}}{255}$$

神經元activation value 計算(A22):
每個神經元接受兩個輸入，假設為 a_1, a_2 (如上提到的a)，並計算所有logic gate的加權期望值：

$$a' = \sum_{i=0}^{15} \boldsymbol{p}_i \cdot f_i(a_1, a_2) = \sum_{i=0}^{15} \frac{e^{\boldsymbol{w}_i}}{\sum_j e^{\boldsymbol{w}_j}} \cdot f_i(a_1, a_2) \, .$$

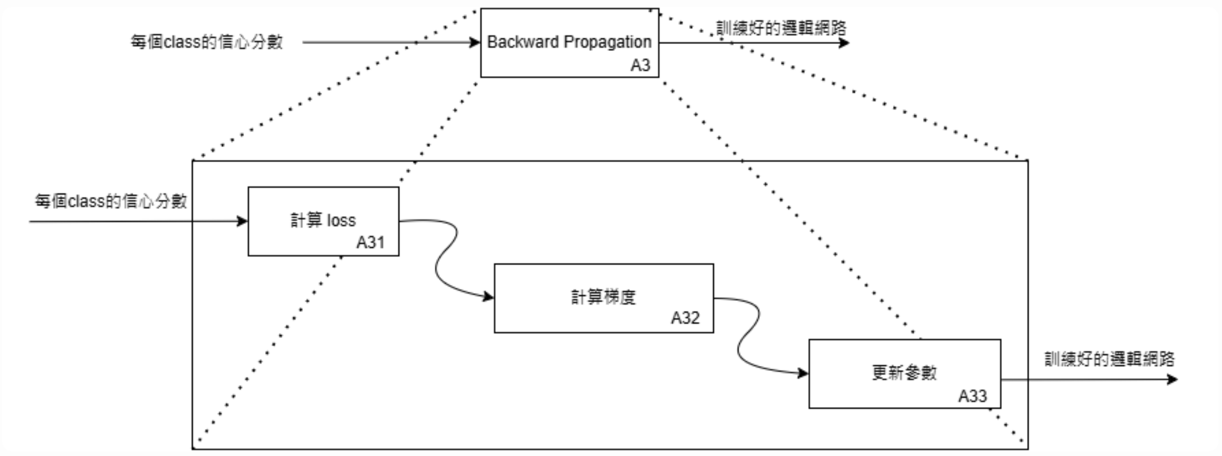
p_i 即是上面提到的每個logic gate被選擇機率，使用softmax算得
$f_i(a_1, a_2)$ 為輸入a_1, a_2，第i個logic operation 的輸出

計算aggregate output(A23):
將output layer的 n 個神經元分成 k 組(if k classes)，每組 $\frac{n}{k}$ 個神經元，計算每個class的aggregate output:

$$\hat{y}_i = \sum_{j=i\cdot n/k+1}^{(i+1)\cdot n/k} a_j \, / \, \tau + \beta$$

- $\hat{y}_i$ ：class i 的信心分數
- a_j ：output layer中第 j 個神經元的activation value
- τ 及 β : normalization value 及 offset

Backward Propagation(A3):



計算loss(A31):
Loss function: Softmax Cross-Entropy Loss，計算模型的的預測機率(q_i)相對於真實目標(t_i)的loss:

先對每個class 的aggregate output ($\hat{y}_i$) 求softmax，得 q_i:

$$q_i = \frac{e^{\hat{y}_i}}{\sum_i^k e^{\hat{y}_i}}$$

再把 q_i 代入cross entropy loss，得 L:

$$L = -\sum_i t_i * \log(q_i)$$

計算梯度(A32):
計算loss對logic gate參數 w_i 的梯度:

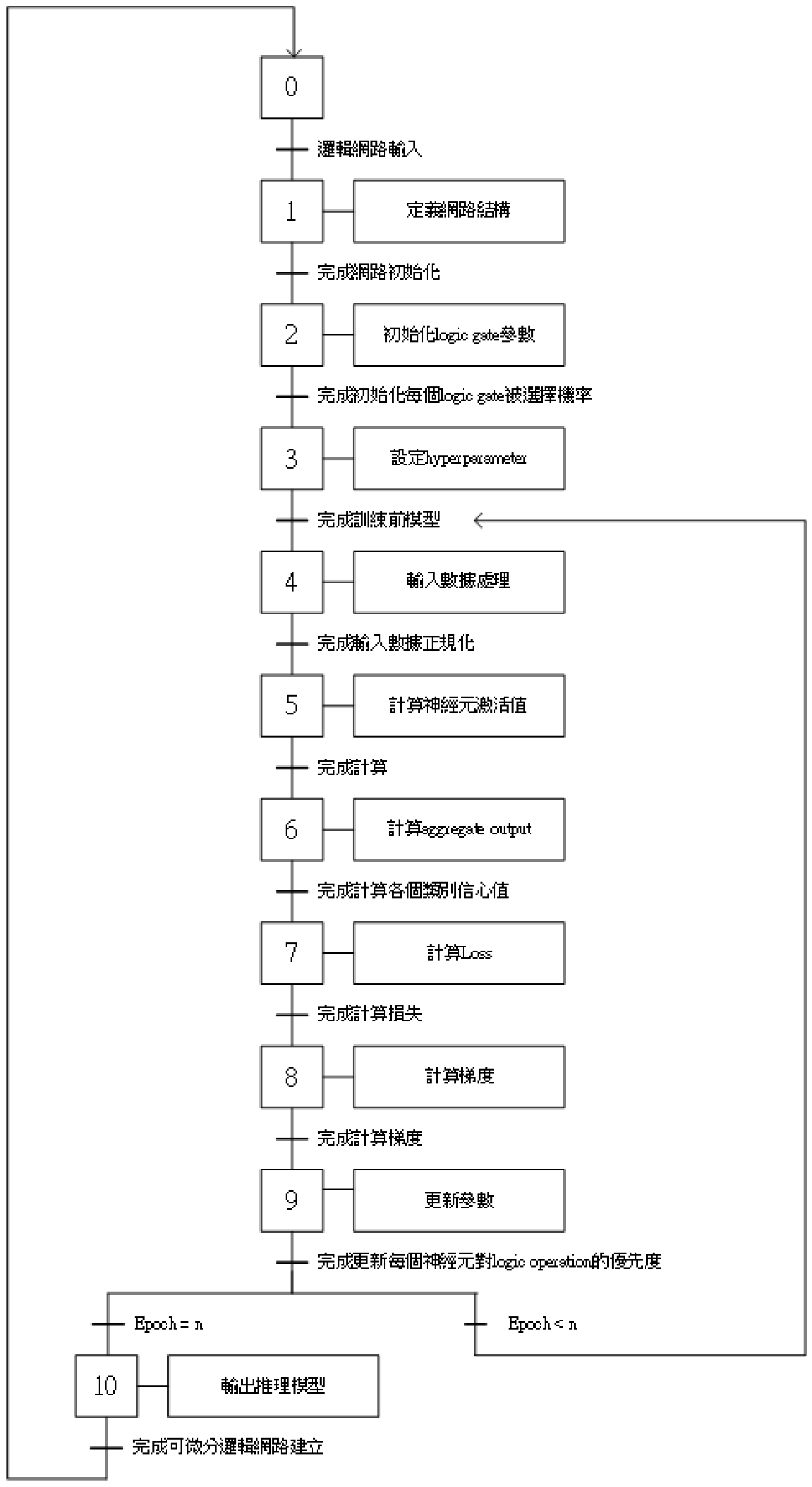
$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial x_1} * .. * \frac{\partial x_i}{\partial w_i}$$

更新參數(A33):
更新 w_i ，$\forall i$ = ith logic operation:

$$w_i = w_i - \eta * \frac{\partial L}{\partial w_i}$$

訓練階段多次迭代Foward Propagationr及Backward Prapagation 來更新 w_i ，並且更新後的 w_i 透過Softmax來更新 p_i(選取每個logic opeartion 的機率)，
使最適合的 p_i 最大化，進而在推理階段時讓每個神經元選擇最適合的logic operation

Grafcet 離散事件模型



以 MIAT 方法論合成 Python

以下程式碼以3層hidden layer，每層layer 6個神經元為例

構建Differentiable Logic Network結構

```
def initialize_network(layers, neurons_per_layer):
    weights = []
    for layer in range(layers):
        layer_weights = [np.random.normal(0, 1, 3) for _ in range(neurons_per_la
        weights.append(layer_weights)
    return weights
```

計算aggregate output

```
def aggregate_outputs(outputs, neurons_per_class):
    num_classes = len(outputs) // neurons_per_class
    aggregated_outputs = []
    for i in range(num_classes):
        aggregated_output = np.sum(outputs[i * neurons_per_class:(i + 1) * neuro
        aggregated_outputs.append(aggregated_output)
    return aggregated_outputs
```

Fordward Propagation計算activation value

```
def forward_propagation(inputs, weights, neuron_idx=None):

    if neuron_idx is not None:
        a1 = inputs[neuron_idx % len(inputs)]
        a2 = inputs[(neuron_idx + 1) % len(inputs)]
    else:
        a1, a2 = inputs[0], inputs[1]

    f = np.array([
        a1 * a2,
        a1 + a2 - a1 * a2,
        a1 + a2 - 2 * a1 * a2
    ])

    exp_weights = np.exp(weights)
    p = exp_weights / np.sum(exp_weights)

    a_prime = np.sum(p * f)
    return p, f, a_prime
```

計算loss

```
def cross_entropy_loss(a_prime, target):
    exp_a_prime = np.exp(a_prime)
    q1 = exp_a_prime / (1 + exp_a_prime)
    q0 = 1 / (1 + exp_a_prime)

    loss = -(target * np.log(q1) + (1 - target) * np.log(q0))
    return loss, q0, q1
```

backward_propagation更新參數

```
def backward_propagation(p, f, q1, target, weights, learning_rate=0.1):
    dL_dq1 = q1 - target
    dq1_da_prime = q1 * (1 - q1)
    da_prime_dp = f
    dp_dweights = p * (1 - p)

    gradients = dL_dq1 * dq1_da_prime * da_prime_dp * dp_dweights

    updated_weights = weights - learning_rate * gradients
    return updated_weights
```

訓練網路

```
def train_network(inputs, targets, weights, layers, neurons_per_layer, neurons_pe
    for epoch in range(epochs):
        print(f"Epoch {epoch + 1}/{epochs}")

        correct_predictions = 0

        for data_idx, input_data in enumerate(inputs):

            layer_inputs = input_data
            all_layer_outputs = []

            for layer in range(layers):
                layer_outputs = []
                for neuron_idx in range(neurons_per_layer):
                    p, f, a_prime = forward_propagation(layer_inputs, weights[la
                    layer_outputs.append(a_prime)

                layer_inputs = layer_outputs
                all_layer_outputs.append(layer_outputs)

            aggregated_outputs = aggregate_outputs(all_layer_outputs[-1], neuron
            predicted_class = np.argmax(aggregated_outputs)

            target = targets[data_idx]
            if predicted_class == target:
                correct_predictions += 1

            for layer in reversed(range(layers)):
                for neuron_idx in range(neurons_per_layer):
                    p, f, a_prime = forward_propagation(layer_inputs, weights[la
                    loss, q0, q1 = cross_entropy_loss(a_prime, target)
                    weights[layer][neuron_idx] = backward_propagation(p, f, q1, 

            print(f"  Data {data_idx + 1}: Loss = {loss:.4f}, Aggregated Outputs

        accuracy = correct_predictions / len(inputs)
        print(f"  Epoch {epoch + 1} Accuracy: {accuracy * 100:.2f}%")

    return weights
```

main function

```
def main():
    inputs = [
        [0.6, 0.8],
        [0.5, 0.1],
        [0.4, 0.9]
    ]
    targets = [1, 0, 1]

    layers = 3
    neurons_per_layer = 6
    neurons_per_class = 3

    weights = initialize_network(layers, neurons_per_layer)

    epochs = 100
    learning_rate = 0.99
    weights = train_network(inputs, targets, weights, layers, neurons_per_layer,
```

執行結果

預測結果(3 個aggregated output)對上實際結果(`targets = [1, 0, 1]` ，表示data 1應預測class 1，data 2應預測class 0，data 3應預測class 1)

```
Epoch 1/100
  Data 1: Loss = 0.4592, Aggregated Outputs = [1.5536200300160217, 1.51570160035
  Data 2: Loss = 0.9879, Aggregated Outputs = [1.3257768520023103, 1.22923697891
  Data 3: Loss = 0.4568, Aggregated Outputs = [1.5747053570045388, 1.56511656940
  Epoch 1 Accuracy: 33.33%
Epoch 2/100
  Data 1: Loss = 0.4591, Aggregated Outputs = [1.5542571775331975, 1.51692235565
  Data 2: Loss = 0.9881, Aggregated Outputs = [1.326332510587431, 1.230625213372
  Data 3: Loss = 0.4567, Aggregated Outputs = [1.575280676730379, 1.566282815223
  Epoch 2 Accuracy: 33.33%
Epoch 3/100
  Data 1: Loss = 0.4590, Aggregated Outputs = [1.5548935737202465, 1.51814050111
  Data 2: Loss = 0.9883, Aggregated Outputs = [1.3268848435686216, 1.23200956199
  Data 3: Loss = 0.4566, Aggregated Outputs = [1.5758556280726452, 1.56744620078
  Epoch 3 Accuracy: 33.33%

# .....

Epoch 19/100
  Data 1: Loss = 0.4573, Aggregated Outputs = [1.5649918994621268, 1.53728084586
  Data 2: Loss = 0.9911, Aggregated Outputs = [1.3352776097627055, 1.25363289576
  Data 3: Loss = 0.4550, Aggregated Outputs = [1.585025426376062, 1.585677281672
  Epoch 19 Accuracy: 66.67%
Epoch 20/100
  Data 1: Loss = 0.4572, Aggregated Outputs = [1.5656189505463238, 1.53845557625
  Data 2: Loss = 0.9913, Aggregated Outputs = [1.3357748598678665, 1.25495162823
  Data 3: Loss = 0.4549, Aggregated Outputs = [1.5855980101710108, 1.58679317160
  Epoch 20 Accuracy: 66.67%
Epoch 21/100
  Data 1: Loss = 0.4571, Aggregated Outputs = [1.5662456585788602, 1.53962781391
  Data 2: Loss = 0.9915, Aggregated Outputs = [1.33626895618595, 1.2562665325916
  Data 3: Loss = 0.4548, Aggregated Outputs = [1.5861706881183641, 1.58790633873
  Epoch 21 Accuracy: 66.67%

# .....

Epoch 78/100
  Data 1: Loss = 0.4511, Aggregated Outputs = [1.6022169601649199, 1.60257626713
  Data 2: Loss = 1.0013, Aggregated Outputs = [1.3595326613379122, 1.32504124647
  Data 3: Loss = 0.4491, Aggregated Outputs = [1.619866825816591, 1.647140211069
  Epoch 78 Accuracy: 100.00%
Epoch 79/100
  Data 1: Loss = 0.4510, Aggregated Outputs = [1.6028677875666095, 1.60361724952
  Data 2: Loss = 1.0014, Aggregated Outputs = [1.3598603452724267, 1.32614237999
  Data 3: Loss = 0.4490, Aggregated Outputs = [1.6204929530276257, 1.64811047666
  Epoch 79 Accuracy: 100.00%
Epoch 80/100
  Data 1: Loss = 0.4509, Aggregated Outputs = [1.6035198671382487, 1.60465619896
  Data 2: Loss = 1.0016, Aggregated Outputs = [1.3601854485141631, 1.32723997818
  Data 3: Loss = 0.4489, Aggregated Outputs = [1.6211208783535749, 1.64907853642
  Epoch 80 Accuracy: 100.00%

# .....

Epoch 100/100
  Data 1: Loss = 0.4488, Aggregated Outputs = [1.616873495415931, 1.625019347100
  Data 2: Loss = 1.0050, Aggregated Outputs = [1.3661618918149232, 1.34845684985
  Data 3: Loss = 0.4469, Aggregated Outputs = [1.6341052390970516, 1.66798903852
  Epoch 100 Accuracy: 100.00%
```

經過訓練後正確預測結果。