1

# 组会

## 1. 20220419 组会

### 1 Evidential FitNet-4 classifier

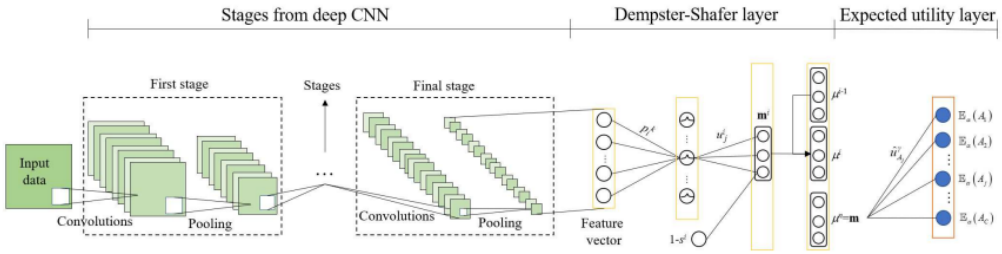



Figure 1: Architecture of an evidential deep-learning classifier.

### 1.1 数据集

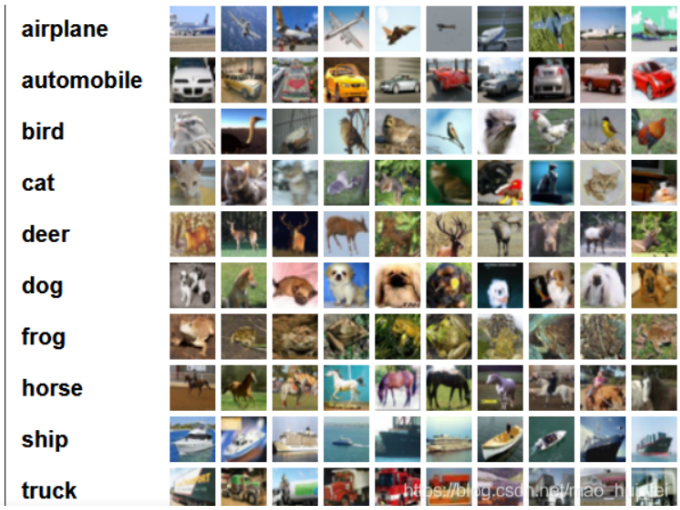
- ```
  (x_train, y_train), (x_test, y_test) = cifar10.load_data()
  x_train=x_train.astype("float32") / 255.0
  x_test=x_test.astype("float32") / 255.0
  y_train_label = y_train
  y_test_label = y_test
  y_train = np_utils.to_categorical(y_train)
  y_test = np_utils.to_categorical(y_test)
  ```

- cifar10

- **CIFAR-10 数据集简介**

  **CIFAR-10 是由 Hinton 的学生 Alex Krizhevsky 和 Ilya Sutskever 整理的一个用于识别普适物体的小型数据集。一共包含 10 个类别的 RGB 彩色图 片：飞机（ a叩lane ）、汽车（ automobile ）、鸟类（ bird ）、猫（ cat ）、鹿（ deer ）、狗（ dog ）、蛙类（ frog ）、马（ horse ）、船（ ship ）和卡车（ truck ）。图片的尺寸为 32×32 ，数据集中一共有 50000 张训练圆片和 10000 张测试图片。CIFAR-10 的图片样例如图所示。**

  下面这幅图就是列举了10各类，每一类展示了随机的10张图片：

- y_train = np_utils.to_categorical(y_train)
  ```

- `y_onehot = tf.one_hot(tf.cast(y, dtype=tf.int32), depth=100)`

- 其他的特征也都这么表示：

| | feature1 | feature2 | feature3 |
|---|---|---|---|
| sample1 | 01 | 1000 | 100 |
| sample2 | 10 | 0100 | 010 |
| sample3 | 01 | 0010 | 010 |
| sample4 | 10 | 0001 | 001 |

这样，4个样本的特征向量就可以这么表示：

sample1 -> [0,1,1,0,0,0,1,0,0]

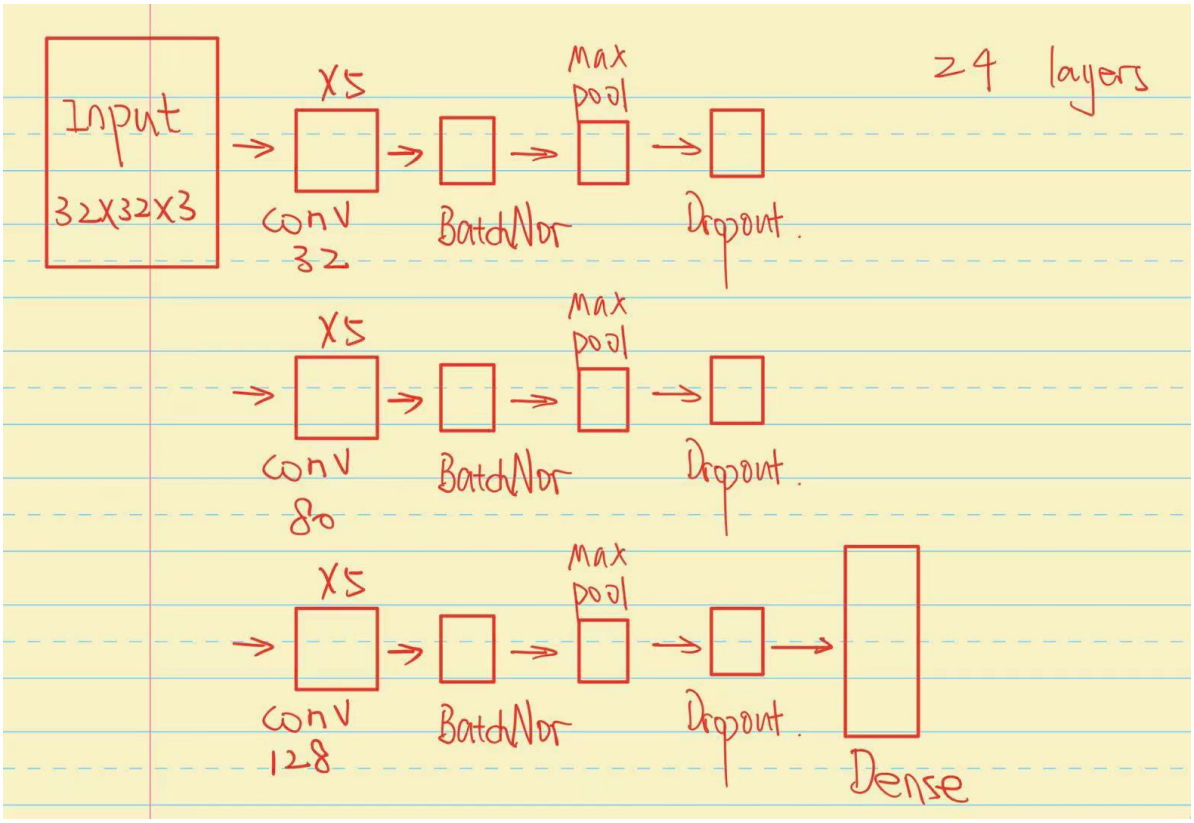
sample2 -> [1,0,0,1,0,0,0,1,0]

sample3 -> [0,1,0,0,1,0,0,1,0]

sample4 -> [1,0,0,0,0,1,0,0,1]

---

## 1.2 卷积层




```
IMG_WIDTH = 32
IMG_HEIGHT = 32
IMG_CHANNELS = 3
inputs_pixels = IMG_WIDTH * IMG_HEIGHT
prototypes = 200
num_class = 10

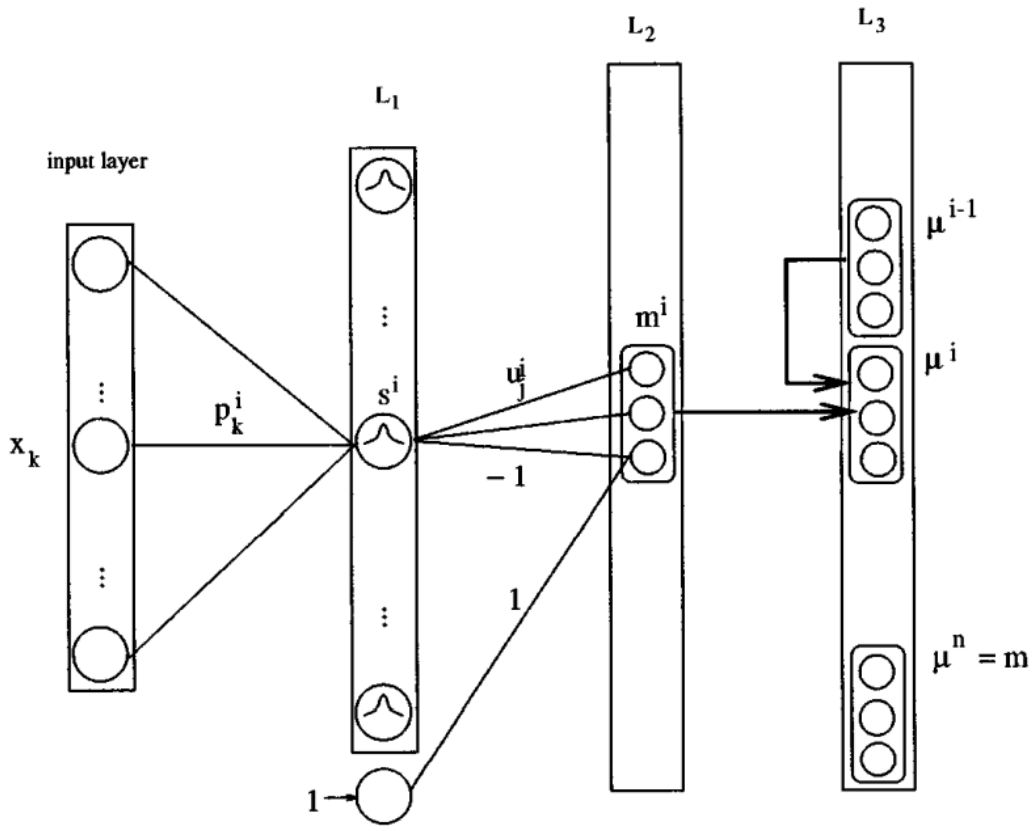
inputs = tf.keras.layers.Input(IMG_HEIGHT, IMG_WIDTH, IMG_CHANNELS)
```

```
#convolution stages
c1_1 = layers.Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(inputs)
c1_2 = layers.Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c1_1)
c1_3 = layers.Conv2D(32, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c1_2)
c1_4 = layers.Conv2D(48, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c1_3)
c1_5 = layers.Conv2D(48, (3, 3), activation='relu',
kernel_initializer='he_normal', padding='same')(c1_4)

# ICS问题会影响模型的训练，而BN通过将每一层网络的输入进行标准化以及scale和shift，保证了每层输入分布的均值与方差固定在一定范围内，保留了网络的表达能力，缓解了ICS问题，使得网络能够适应更高的学习率，并大大地加速了模型的训练速度。最后，由于BN训练过程中使用的是mini-batch的均值与方差作为总体均值与方差的估计，引入了随机噪声，在一定程度上对模型起到了正则化的效果。
bt1 = layers.BatchNormalization()(c1_5)
# 池化层
p1 = layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='same')(bt1)
# 随机删除神经元，防止过度拟合
dr1 = layers.Dropout(0.5)(p1)
```

- (58条消息) Batch Normalization(BN)简介*seven_777k*的博客-*CSDN*博客batchnormalization()

## 1.3 DS层

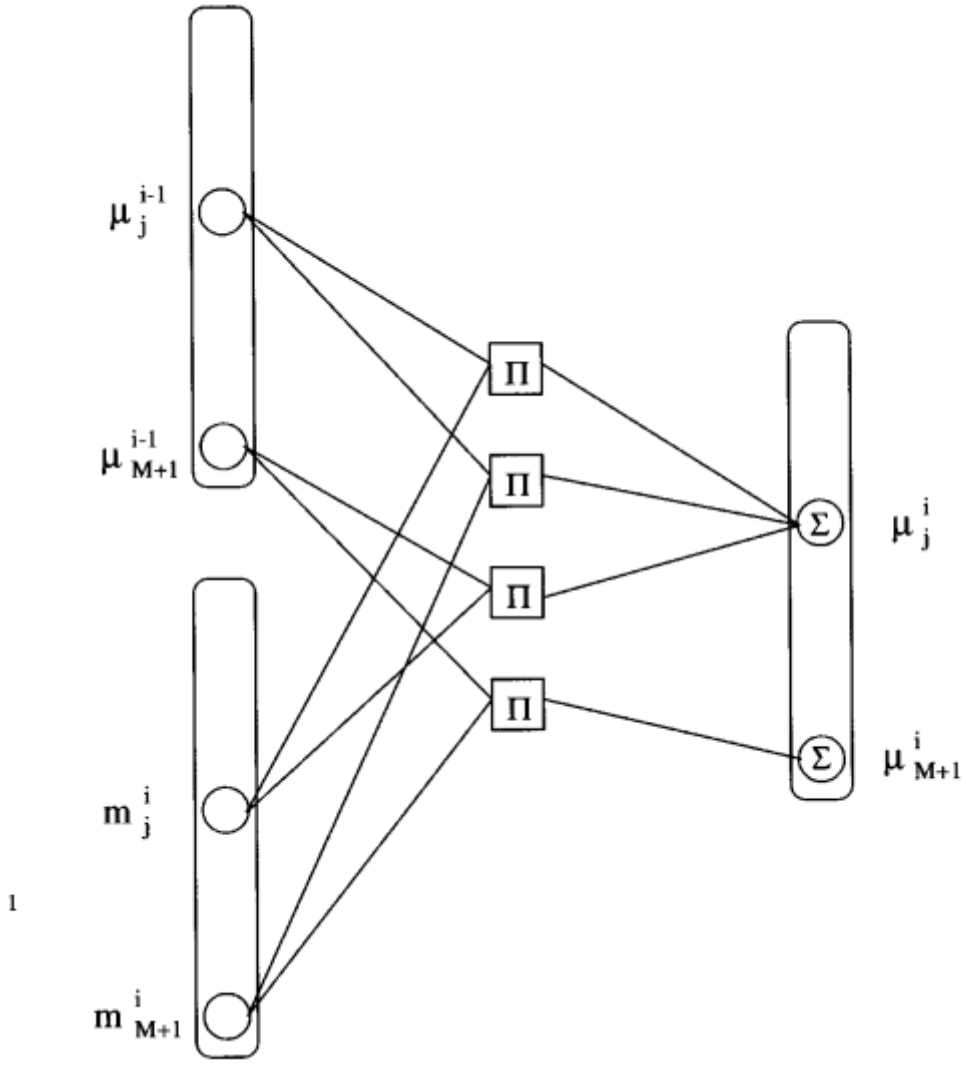



```
# DS layers
ED = ds.DS1(prototypes,128)(flatten1)
ED_ac = ds.DS1_activate(prototypes)(ED)
mass_prototype = ds.DS2(prototypes, num_class)(ED_ac)
```

```
mass_prototype_omega = ds.DS2_omega(prototypes, num_class)(mass_prototype)
mass_Dempster = ds.DS3_Dempster(prototypes, num_class)(mass_prototype_omega)
mass_Dempster_normalize = ds.DS3_normalize()(mass_Dempster)

# Utility layer for training
outputs = uti_train.DM(0.9, num_class)(mass_Dempster_normalize)

model_e = keras.Model(inputs=[inputs], outputs=[outputs])
model_e.compile(optimizer=keras.optimizers.Nadam(lr=0.001, beta_1=0.9,
beta_2=0.999, epsilon=None, schedule_decay=0.005),
                loss='categorical_crossentropy',
                metrics=['accuracy'])
model_e.summary()
```




$$\mu_j^i = \mu_j^{i-1} m_j^i + \mu_j^{i-1} m_{M+1}^i + \mu_{M+1}^{i-1} m_j^i \quad j = 1, \cdots, M \tag{30}$$

$$\mu_{M+1}^i = \mu_{M+1}^{i-1} m_{M+1}^i. \tag{31}$$

### 1.4 配置参数

```
# get the feature using the probabilistic classifier
model_e.load_weights("C:\DLDoc\CIFAR10")
feature = keras.Model(inputs=[inputs], outputs=[flatten1])
x_train_feature = feature.predict(x_train)
x_test_feature = feature.predict(x_test)

# use the feature to train DS layer
inputs = layers.Input(128)
ED = ds.DS1(prototypes, 128)(inputs)
ED_ac = ds.DS1_activate(prototypes)(ED)
mass_prototype = ds.DS2(prototypes, num_class)(ED_ac)
mass_prototype_omega = ds.DS2_omega(mass_prototype, num_class)(mass_prototype)
mass_Dempster = ds.DS3_Dempster(prototypes, num_class)(mass_prototype_omega)
mass_Dempster_normalize = ds.DS3_normalize()(mass_Dempster)

outputs = uti_train.DM(0.9, num_class)(mass_Dempster_normalize)

model_mid = keras.Model(inputs=[inputs], outputs=[outputs])
model_mid.compile(optimizer=keras.optimizers.Nadam(lr=0.001, beta_1=0.9,
beta_2=0.999, epsilon=None, schedule_decay=0.005),
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

model_mid.fit(x_train_feature, y_train, batch_size=25, epochs=10, verbose=1,
validation_data=(x_test_feature, y_test), shuffle=True)

#give the trained parameters to the evidential model
DS1_W = model_mid.layers[1].get_weights()
DS1_activate_W = model_mid.layers[2].get_weights()
DS2_W = model_mid.layers[3].get_weights()

model_e.layers[26].set_weights(DS1_W)
model_e.layers[27].set_weights(DS1_activate_W)
model_e.layers[28].set_weights(DS2_W)

filepath_e = 'C:\DLDoc\model_e'
checkpoint_callback_e = ModelCheckpoint(
    filepath_e, monitor='val_accuracy', verbose=1,
    save_best_only=True, save_weights_only=True,
    period=1
)

model_e.fit(x_train, y_train, batch_size=25, epochs=20, verbose=1, callbacks=
[checkpoint_callback_e], validation_data=(x_test, y_test), shuffle=True)
```

### 1.5 对比

- 普通卷积神经网络
- loss = 0.45
- accuracy = 0.86

```
In 16  1 model_p.load_weights("C:\DLDoc\CIFAR10")
       2 model_p.evaluate(x_train, y_train, batch_size=25, verbose=1)
       3 model_p.evaluate(x_test, y_test, batch_size=25, verbose=1)

         50000/50000 [==============================] - 6s 121us/sample - loss: 0.0990 - accuracy: 0.9667
         10000/10000 [==============================] - 1s 117us/sample - loss: 0.4564 - accuracy: 0.8649

Out 16   [0.45640941910678523, 0.8649]
```

- 证据卷积神经网络
- loss = 0.55
- accuracy = 0.86

```
In 8   1 model_e.load_weights('C:\DLDoc\model_e')
       2 model_e.evaluate(x_train, y_train, batch_size=25, verbose=1)
       3 model_e.evaluate(x_test, y_test, batch_size=25, verbose=1)

         50000/50000 [==============================] - 48s 959us/sample - loss: 0.1715 - accuracy:
          0.9641
         10000/10000 [==============================] - 6s 604us/sample - loss: 0.5550 - accuracy:
         0.8644

Out 8    [0.5549526660330594, 0.8644]
```

```
In _   1 model_p.load_weights('/content/gdrive/My Drive/cifar10_evidential/weights_zoo/cnn_checkpoint_final')# replace the path wit
         own
       2 model_p.evaluate(x_train, y_train, batch_size=25, verbose=1)
       3 model_p.evaluate(x_test, y_test, batch_size=25, verbose=1)

          2000/2000 [==============================] - 10s 3ms/step - loss: 0.0021 - accuracy: 0.9993
          400/400 [==============================] - 1s 3ms/step - loss: 0.8291 - accuracy: 0.8708

Out 4     [0.8290520310401917, 0.8708000183105469]
```

```
In _   1 model_e.load_weights('/content/gdrive/My Drive/cifar10_evidential/weights_zoo/evidential_DL_200_checkpoint')
       2 model_e.evaluate(x_train, y_train, batch_size=25, verbose=1)
       3 model_e.evaluate(x_test, y_test, batch_size=25, verbose=1)

          WARNING:tensorflow:Inconsistent references when loading the checkpoint into this object graph. Either the Trackable object
          references in the Python program have changed in an incompatible way, or the checkpoint was generated in an incompatible
          program.

          Two checkpoint references resolved to different objects (<ds_layer.DS2 object at 0x7f0aa3c47210> and <ds_layer
          .DS3_normalize object at 0x7f0aa3b8d710>).
          2000/2000 [==============================] - 51s 19ms/step - loss: 0.0631 - accuracy: 0.9970
          400/400 [==============================] - 8s 19ms/step - loss: 0.6747 - accuracy: 0.8757

Out 7     [0.6746578216552734, 0.8756999969482422]
```

# 2 下一步计划

- 1 进一步完成DS层 imprecise_results

# 2. 20220510 组会

- DS层代码与公式推导

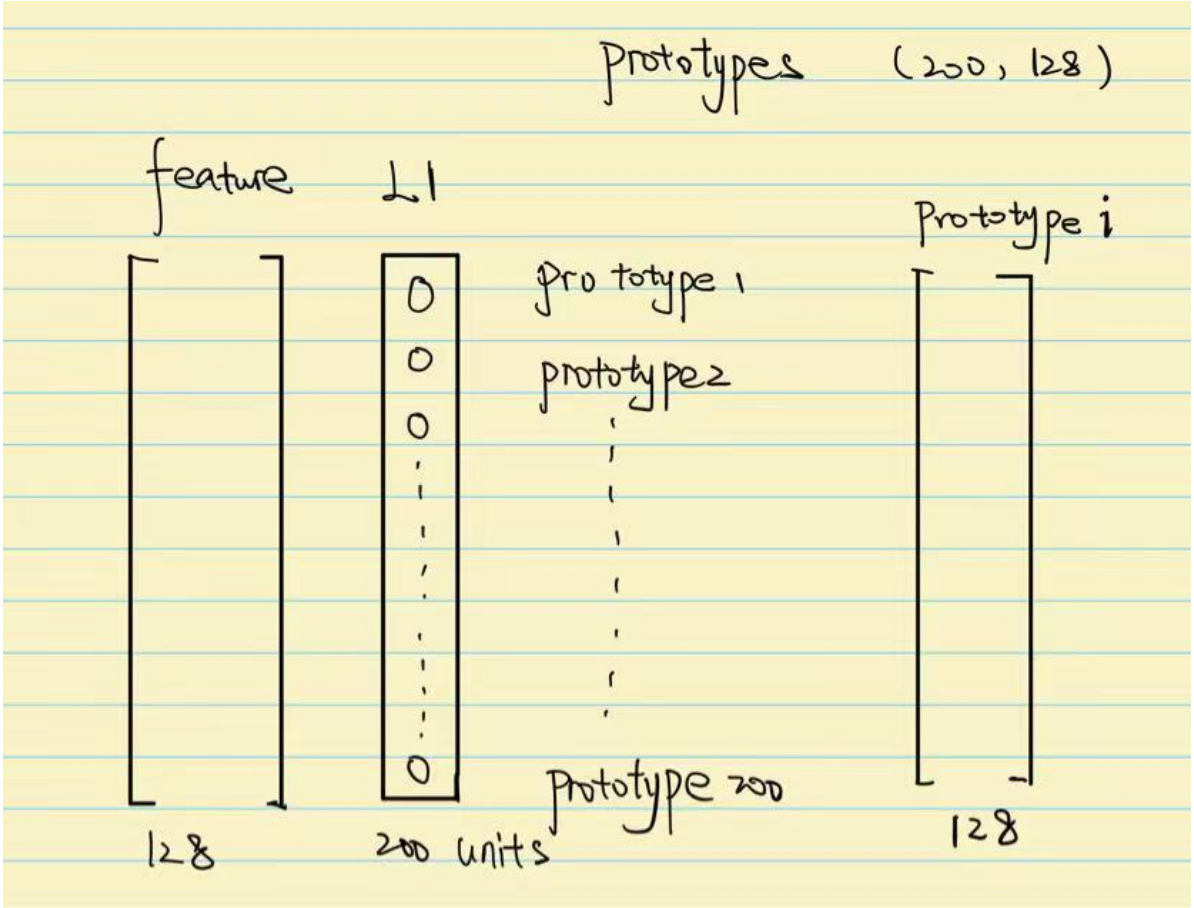
## 2.1.1 L1 求距离

*Step 1:* The distances $d^i$ between $\boldsymbol{x}$ and each prototype vector $\boldsymbol{p}^i$ are computed according to some metric, for example the Euclidean one:

$$d^i = \|\boldsymbol{x} - \boldsymbol{p}^i\| \qquad i = 1, \cdots, n. \tag{16}$$

- 这里求距离距离是二范数，范数是长度的推广与抽象，具有正定性，齐次性与满足三角不等式。

$$||x||_2 = (\sum_{i=1}^{n}(x_i)^2)^{1/2}$$




```
class DS1(tf.keras.layers.Layer):
    def __init__(self, units, input_dim):
        super(DS1, self).__init__()
        self.w = self.add_weight(
            name='Prototypes',
            shape=(units, input_dim),
            initializer='random_normal',
            trainable=True
        )
        self.units = units
```

```
    def call(self, inputs):
        for i in range(self.units):
            if i == 0:
                un_mass_i = tf.subtract(self.w[i, :], inputs, name=None)
                un_mass_i = tf.square(un_mass_i, name=None)
                un_mass_i = tf.reduce_sum(un_mass_i, -1, keepdims=True)
                un_mass = un_mass_i

            if i >= 1:
                un_mass_i = tf.subtract(self.w[i, :], inputs, name=None)
                un_mass_i = tf.square(un_mass_i, name=None)  #(N, 128)
                un_mass_i = tf.reduce_sum(un_mass_i, -1, keepdims=True)  #(N)
                un_mass = tf.concat([un_mass, un_mass_i], -1)  #(N, 200)

        return un_mass
```

## 2.1.2 L1 激活函数

$$\phi^i(d^i) = \exp(-\gamma^i(d^i)^2) \tag{20}$$

Step 1: The distance-based support between $\boldsymbol{x}$ and each reference pattern $\boldsymbol{p}^i$ is computed as

$$s^i = \alpha^i \exp(-\left(\eta^i d^i\right)^2) \quad i = 1, \ldots, n, \tag{5}$$

where $d^i = \|\boldsymbol{x} - \boldsymbol{p}^i\|$ is the Euclidean distance between $\boldsymbol{x}$ and prototype $\boldsymbol{p}^i$, and $\alpha^i \in (0, 1)$ and $\eta^i \in \mathbb{R}$ are parameters associated with prototype $\boldsymbol{p}^i$. Prototype vectors $\boldsymbol{p}^1, \ldots, \boldsymbol{p}^n$ can be considered as vectors of connection weights between the input layer and a hidden layer of $n$ Radial basis Function (RBF) units.

```
class DS1_activate(tf.keras.layers.Layer):
    def __init__(self, input_dim):
        super(DS1_activate, self).__init__()
        self.xi = self.add_weight(
            name='xi',
            shape=(1, input_dim),
            initializer='random_normal',
            trainable=True
        )
        self.eta = self.add_weight(
            name='eta',
            shape=(1, input_dim),
            initializer='random_normal',
            trainable=True
        )
        self.input_dim = input_dim

    def call(self, inputs):
        gamma = tf.square(self.eta, name=None)

        alpha = tf.negative(self.xi, name=None)
        alpha = tf.exp(alpha, name=None) + 1
```

```
        alpha = tf.divide(1, alpha, name=None)
        #\alpha = \frac{1}{e^{-x_i}+1}

        si = tf.multiply(gamma, inputs, name=None)
        si = tf.negative(si, name=None)
        si = tf.exp(si, name=None)
        si = tf.multiply(si, alpha, name=None) #（N, 200）
        #MAX si 为特征距离最近的， divide是对si进行加权
        si = tf.divide(si, (tf.reduce_max(si, axis=-1, keepdims=True) + 0.001),
name=None)

        return si
```

$$\alpha = \frac{1}{e^{-x_i}+1}$$

---

## 2.2.1 L2 求质量函数

Step 2: The mass function $m^i$ associated to reference pattern $\boldsymbol{p}^i$ is computed as

$$m^i(\{\omega_j\}) = h_j^i s^i, \quad j = 1, \ldots, M \tag{6a}$$

$$m^i(\Omega) = 1 - s^i, \tag{6b}$$

where $h_j^i$ is the degree of membership of prototype $\boldsymbol{p}^i$ to class $\omega_j$ with $\sum_{j=1}^{M} h_j^i = 1$. We denote the vector of masses induced by prototype $\boldsymbol{p}^i$ as

$$\boldsymbol{m}^i = (m^i(\{\omega_1\}), \ldots, m^i(\{\omega_M\}), m^i(\Omega))^T.$$

Eq. (6) can be regarded as computing the activation of units in a second hidden layer of the ENN classifier, composed of $n$ modules of $M + 1$ units each. The result of module $i$ corresponds to the belief masses assigned by $m^i$.

```
class DS2(tf.keras.layers.Layer):
    def __init__(self, input_dim, num_class):
        super(DS2, self).__init__()
        self.beta = self.add_weight(
            name='beta',
            shape=(input_dim, num_class),
            initializer='random_normal',
            trainable=True
        )
        self.input_dim = input_dim
        self.num_class = num_class

    def call(self, inputs):
        beta = tf.square(self.beta, name=None)  #(200,10)
        beta_sum = tf.reduce_sum(beta, -1, keepdims=True)  #(200,1)
        u = tf.divide(beta, beta_sum, name=None)  #(200, 10)

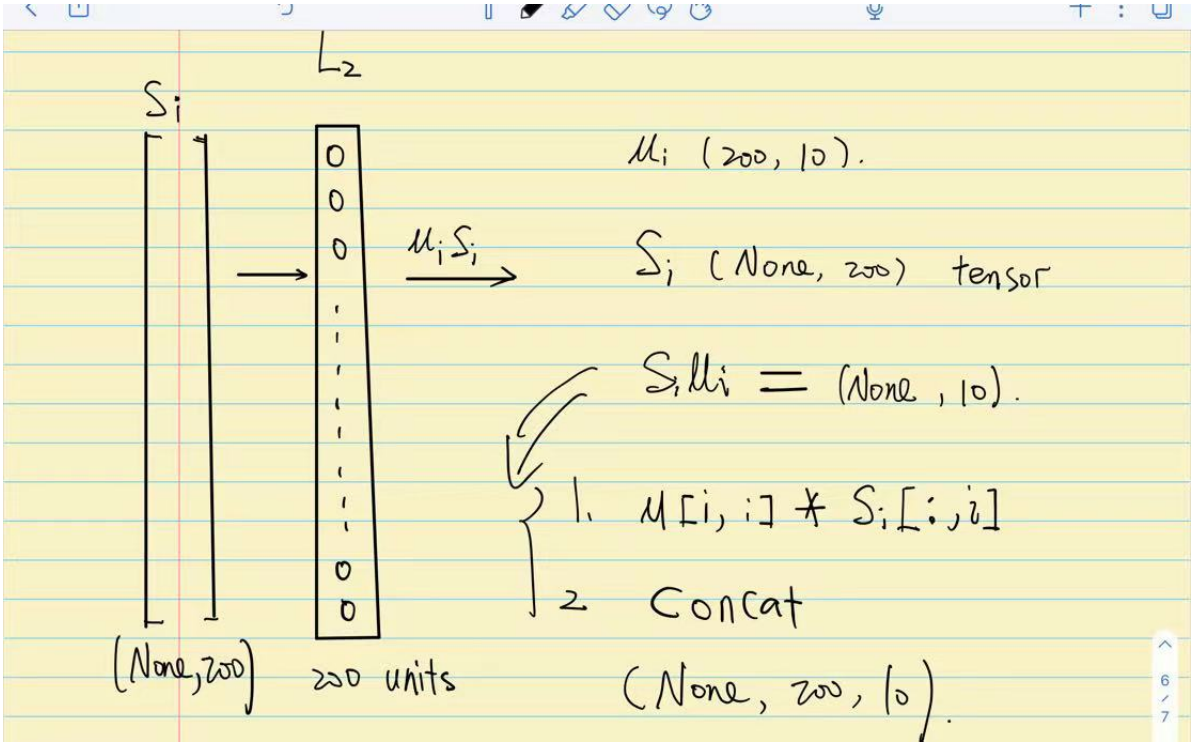
        inputs_new = tf.expand_dims(inputs, -1)  #(None,200) -> (None,200, 1)

        for i in range(self.input_dim):
            if i == 0:
```

```
                # (None,10) * (None,1) -> (None,10)
                mass_prototype_i = tf.multiply(u[i, :], inputs_new[:, i],
name=None)
                mass_prototype = tf.expand_dims(mass_prototype_i, -2)  #
(None,1,10)

            if i >= 1:
                # (None,1,10)
                mass_prototype_i = tf.expand_dims(tf.multiply(u[i, :],
inputs_new[:, i], name=None), -2)
                mass_prototype = tf.concat([mass_prototype, mass_prototype_i],
-2)  # (None, 200, 10)

        mass_prototype = tf.convert_to_tensor(mass_prototype)
        return mass_prototype
```



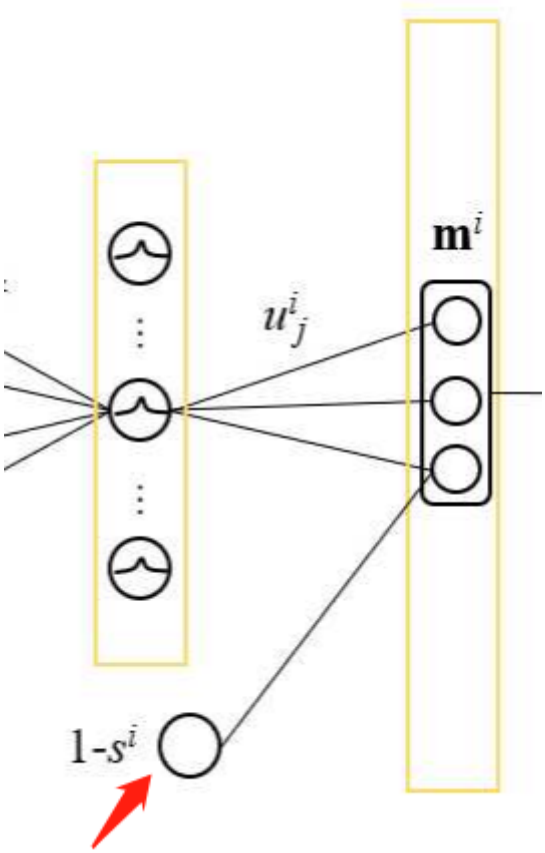

## 2.2.2 L2 求不确定类质量函数

-

$$\sum_{A \subseteq \Omega} m^i(A) = \sum_{q=1}^{M} m^i(\{\omega_q\}) + m^i(\Omega) = 1 \qquad (19)$$

```
class DS2_omega(tf.keras.layers.Layer):
    def __init__(self, input_dim, num_class):
        super(DS2_omega, self).__init__()
        self.input_dim = input_dim
        self.num_class = num_class

    def call(self, inputs):
        mass_omega_sum = tf.reduce_sum(inputs, -1, keepdims=True)  # (None, 200)
        mass_omega_sum = tf.subtract(1., mass_omega_sum[:, :, 0], name=None)  #
(None, 200)
        mass_omega_sum = tf.expand_dims(mass_omega_sum, -1)  # (None, 200, 1)
        mass_with_omega = tf.concat([inputs, mass_omega_sum], -1)  # (None, 200,
11)
        return mass_with_omega
```




### 2.3.1 D-S 融合

The quantity $\text{bel}(A)$ can be interpreted as a global measure of one's belief that hypothesis $A$ is true, while $\text{pl}(A)$ may be viewed as the amount of belief that could *potentially* be placed in $A$, if further information became available [26].

Two BBA's $m_1$ and $m_2$ on $\Omega$, induced by two independent items of evidence, can be combined by the so-called *Dempster's rule of combination* to yield a new BBA $m = m_1 \oplus m_2$, called the orthogonal sum of $m_1$ and $m_2$, and defined as:

$$m(\emptyset) = 0 \tag{5}$$

$$m(A) = \frac{\sum_{B \cap C = A} m_1(B) m_2(C)}{\sum_{B \cap C \neq \emptyset} m_1(B) m_2(C)} \qquad A \neq \emptyset. \tag{6}$$

The computation of $m$ is possible if and only if there exist at least two subsets $B$ and $C$ of $\Omega$ with $B \cap C \neq \emptyset$ such that $m_1(B) \neq 0$ and $m_2(C) \neq 0$. $m_1$ and $m_2$ are then said to be *combinable*. The orthogonal sum is commutative and associative.

Alternatively, Smets [24] proposed to use the *unnormalized* or *conjunctive* rule of combination $\cap$, defined for all $A \subseteq \Omega$ by

$$m = m_1 \cap m_2 \Leftrightarrow m(A) = \sum_{B \cap C = A} m_1(B) m_2(C), \quad \forall A \subseteq \Omega. \tag{7}$$

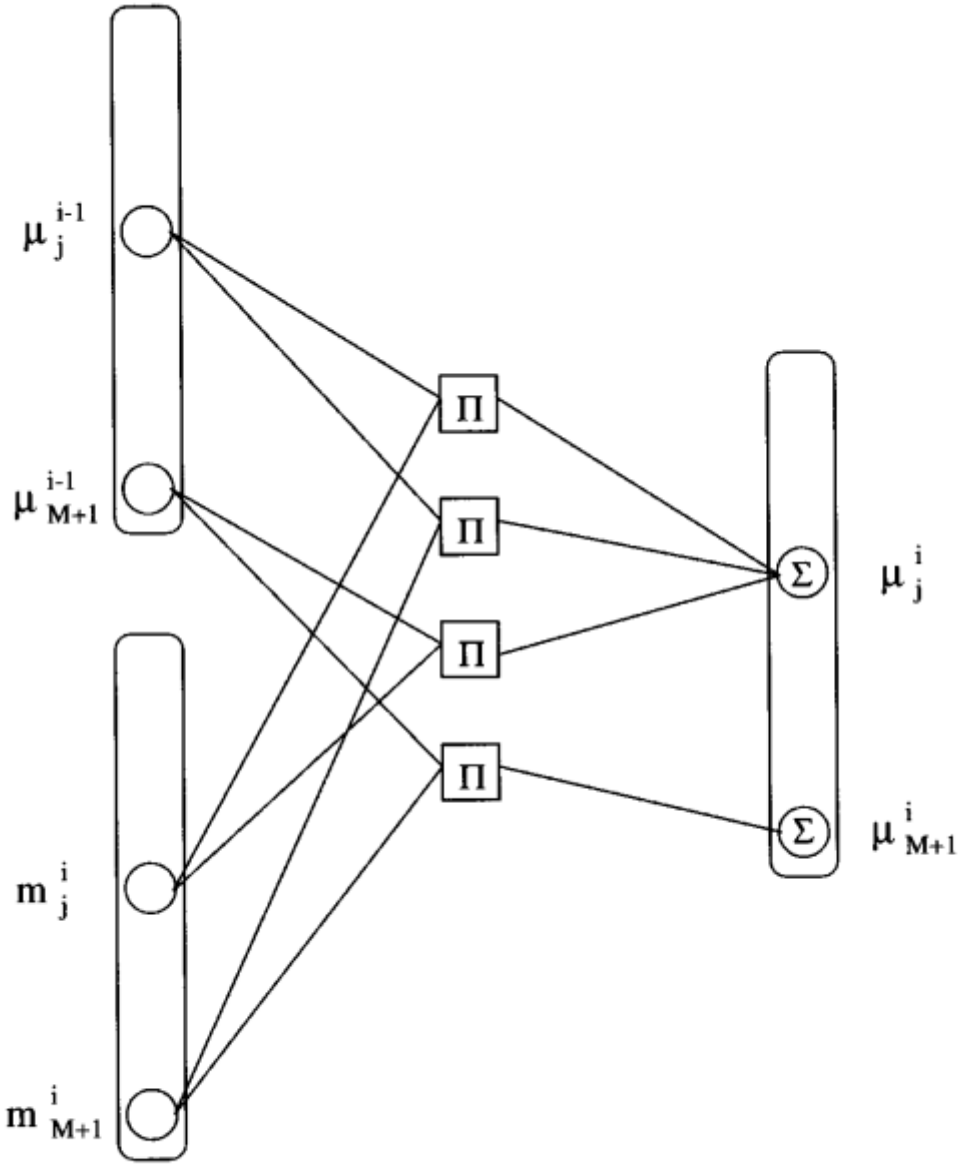



Fig. 3. Details of incoming connections to module $i$ of layer $L_3$. All connection weights are fixed and equal to unity.

where $\mu^i$ is the conjunctive combination of the BBA's $m^1, \cdots, m^i$:

$$\mu^1 = m^1 \tag{28}$$

$$\mu^i = \bigcap_{k=1}^{i} m^k = \mu^{i-1} \cap m^i \qquad i = 2, \cdots, n. \tag{29}$$

The activation vectors $\vec{\mu}^i$ for $i = 2, \cdots, n$ can be recursively computed using the following formula:

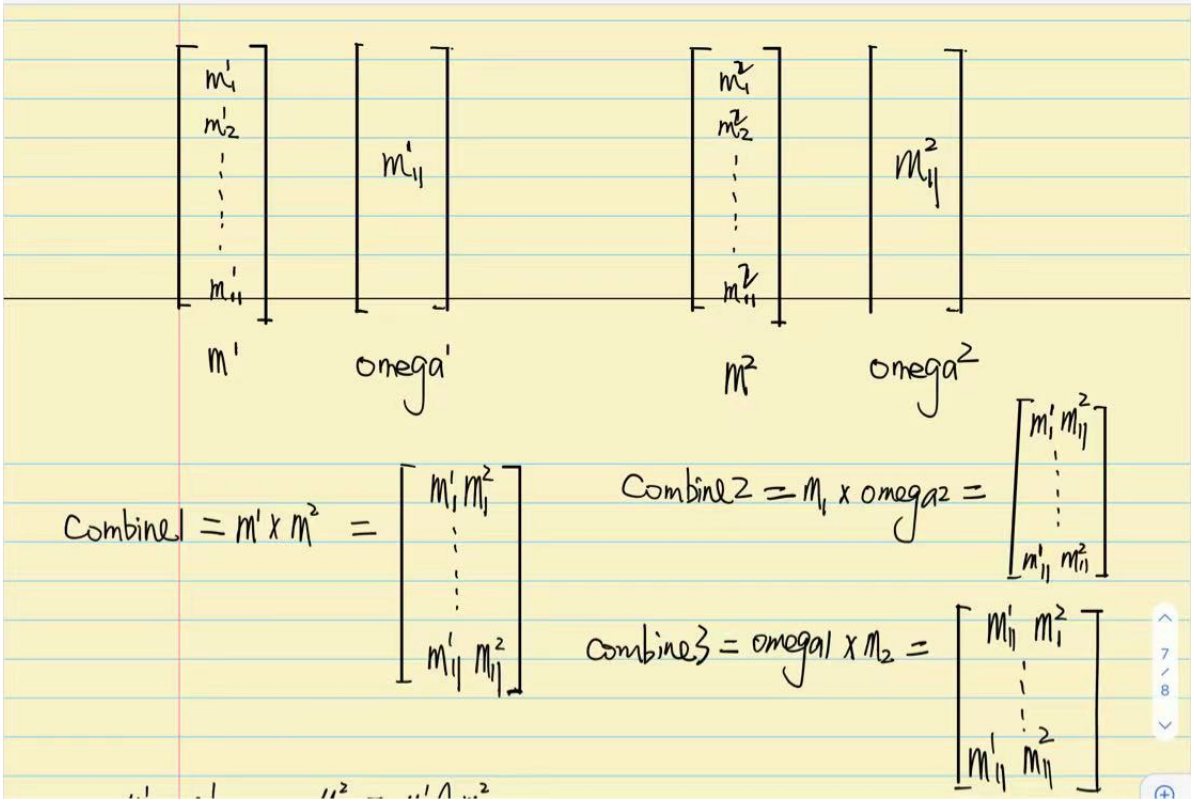
$$\mu_j^i = \mu_j^{i-1} m_j^i + \mu_j^{i-1} m_{M+1}^i + \mu_{M+1}^{i-1} m_j^i \qquad j = 1, \cdots, M \tag{30}$$

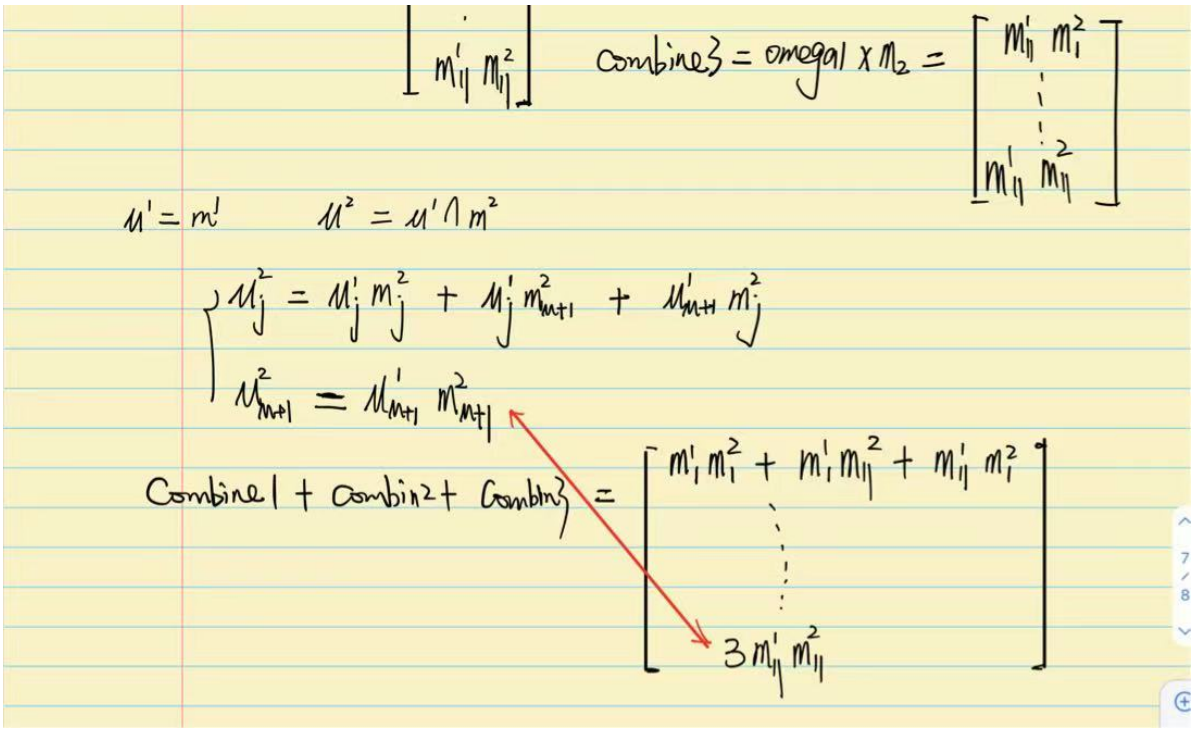
$$\mu_{M+1}^i = \mu_{M+1}^{i-1} m_{M+1}^i. \tag{31}$$

This calculation can be performed by each computing element in module $i > 1$ of layer $L_3$, provided it receives input from module $i - 1$ in the same layer, and from module $i$ of layer $L_2$ (Fig. 3). This can be achieved by fixed-weight connections between layers $L_2$ and $L_3$, and inside layer $L_3$.

```
class DS3_Dempster(tf.keras.layers.Layer):
    def __init__(self, input_dim, num_class):
        super(DS3_Dempster, self).__init__()
        self.input_dim = input_dim
        self.num_class = num_class

    def call(self, inputs):
        m1 = inputs[:, 0, :]  # (None, 11)
        omega1 = tf.expand_dims(inputs[:, 0, -1], -1)  # (None, 1 )
        for i in range(self.input_dim - 1):
            m2 = inputs[:, (i + 1), :]
            omega2 = tf.expand_dims(inputs[:, (i + 1), -1], -1)  # (None, 1)
            combine1 = tf.multiply(m1, m2, name=None)  # (None, 11)
            combine2 = tf.multiply(m1, omega2, name=None)  # (None, 11)
            combine3 = tf.multiply(omega1, m2, name=None)
            combine1_2 = tf.add(combine1, combine2, name=None)
            combine2_3 = tf.add(combine1_2, combine3, name=None)
            combine2_3 = combine2_3 / tf.reduce_sum(combine2_3, axis=-1,
keepdims=True)
            m1 = combine2_3
            omega1 = tf.expand_dims(combine2_3[:, -1], -1)
        return m1
```




### 2.4.1 DM 决策层

Ω of classes) to each class. A transformed output vector $\boldsymbol{P}_\nu = (P_{\nu,1}, \cdots, P_{\nu,M})^t$ is then defined as

$$P_{\nu,q} = m_q + \nu m_{M+1} \qquad q = 1, \cdots, M \qquad (35)$$

```python
class DM(tf.keras.layers.Layer):
    def __init__(self, nu, num_class):
        super(DM, self).__init__()
        self.nu = nu
        self.num_class = num_class

    def call(self, inputs):
        upper = tf.expand_dims((1-self.nu) * inputs[:,-1], -1)  # (None, 1)
        # 在第二个维度上复制11次
        upper = tf.tile(upper, [1, self.num_class + 1])  # (None, 11)
        # 1到10个元素，包前不包后
        outputs = tf.add(inputs, upper, name=None)[:, 0:-1]  # (None, 10)
        return outputs
```

```python
gama = tf.keras.layers.Input(11)
print(gama)
# 不包尾 gama[:, 0:-1] = gama[:, 0:11]
print(gama[:,0:-1])
# 包尾
print(gama[:,0:10])
```

```
Tensor("input_3:0", shape=(None, 11), dtype=float32)
Tensor("strided_slice_1:0", shape=(None, 10), dtype=float32)
Tensor("strided_slice_2:0", shape=(None, 10), dtype=float32)
```

## 2.5 Expected utility layer

- 1 Ordered Weighted Average (OWA)

Let $\Omega = \{\omega_1, \ldots, \omega_M\}$ be the set of classes. For classification problems with only precise prediction, an act is defined as the assignment of an example to one and only one of the $M$ classes. The set of acts is $\mathcal{F} = \{f_{\omega_1}, \ldots, f_{\omega_M}\}$, where $f_{\omega_i}$ denotes assignment to class $\omega_i$. To make decisions, we define a utility matrix $\mathbb{U}_{M\times M}$, whose general term $u_{ij} \in [0,1]$ is the utility of assigning an example to class $\omega_i$ when the true class is $\omega_j$. Here, $\mathbb{U}_{M\times M}$ is called the *original utility matrix*. For decision-making with belief functions, each act $f_{\omega_i}$ induces expected utilities, such as the lower and upper expected utilities defined by (3).

For classification problems with imprecise prediction, Ma and Denœux [38] proposed an approach to conduct set-valued classification under uncertainty by generalizing the set of acts as partially assigning a sample to a non-empty subset $A$ of $\Omega$. Thus, the set of acts becomes $\mathcal{F} = \{f_A, A \in 2^\Omega \backslash \emptyset\}$, in which $2^\Omega$ is the power set of $\Omega$ and $f_A$ denotes the assignment to a subset $A$. In this study, subset $A$ is defined as a *multi-class set* if $|A| \geq 2$. For decision-making with $\mathcal{F}$, the original utility matrix $\mathbb{U}_{M\times M}$ is extended to $\mathbb{U}_{(2^\Omega-1)\times M}$, where each element $\widehat{u}_{A,j}$ denotes the utility of assigning a sample to set $A$ of classes when the true label is $\omega_j$.

When the true class is $\omega_j$, the utility of assigning a sample to set $A$ is defined as an Ordered Weighted Average (OWA) aggregation [69] of the utilities of each precise assignment in $A$ as

$$\widehat{u}_{A,j} = \sum_{k=1}^{|A|} g_k \cdot u^A_{(k)j}, \tag{10}$$

- The elements in weight vector g represent the DM's tolerance to imprecision.

- gama
- cross entropy

In this study, following [38], we determine the weight vector $\boldsymbol{g}$ of the OWA operators by adapting O'Hagan's method [46]. We define the *imprecision tolerance degree* as

$$TDI(\boldsymbol{g}) = \sum_{k=1}^{|A|} \frac{|A| - k}{|A| - 1} g_k = \gamma, \tag{11}$$

which equals to 1 for the maximum, 0 for the minimum, and 0.5 for the average. In practice, we only need to consider values of $\gamma$ between 0.5 and 1 as a precise assignment is preferable to an imprecise one when $\gamma$<0.5 [38]. Given a value of $\gamma$, we can compute the weights of the OWA operator by maximizing the entropy

$$ENT(\boldsymbol{g}) = -\sum_{k=1}^{|A|} g_k \log g_k \tag{12}$$

subject to the constraints $TDI(\boldsymbol{g}) = \gamma$, $\sum_{k=1}^{|A|} g_k = 1$, and $g_k \geq 0$.

```
# aim func: cross entropy
def func(x):
    fun=0
    for i in range(len(x)):
        fun += x[i] * math.log10(x[i])
    return fun

#constraint 1: the sum of weights is 1
def cons1(x):
    return sum(x)

#constraint 2: define tolerance to imprecision
def cons2(x):
    tol = 0
    for i in range(len(x)):
        tol += (len(x) -(i+1)) * x[i] / (len(x) - 1)
    return tol

#compute the weights g for ordered weighted average aggregation
num_class = 10
for j in range(2,(num_class+1)):
    num_weights = j
    ini_weights = np.asarray(np.random.rand(num_weights))

    name='weight'+str(j)
    locals()['weight'+str(j)]= np.zeros([5, j])

    for i in range(5):
        tol = 0.5 + i * 0.1

        cons = ({'type': 'eq', 'fun' : lambda x: cons1(x)-1},
                {'type': 'eq', 'fun' : lambda x: cons2(x)-tol},
                {'type': 'ineq', 'fun' : lambda x: x-0.00000001}
                )

        res = minimize(func, ini_weights, method='SLSQP', options={'disp':
True}, constraints=cons)
```

```
        locals()['weight'+str(j)][i] = res.x
        print (res.x)


#function for power set
def PowerSetsBinary(items):
    #generate all combination of N items
    N = len(items)
    #enumerate the 2**N possible combinations
    set_all=[]
    for i in range(2**N):
        combo = []
        for j in range(N):
            if(i >> j ) % 2 == 1:
                combo.append(items[j])
        set_all.append(combo)
    return set_all


utility_matrix = np.zeros([len(act_set), len(class_set)])
tol_i = 3
#tol_i = 0 with tol=0.5, tol_i = 1 with tol=0.6, tol_i = 2 with tol=0.7, tol_i =
3 with tol=0.8, tol_i = 4 with tol=0.9
for i in range(len(act_set)):
    intersec = class_set and act_set[i]
    if len(intersec) == 1:
        utility_matrix[i, intersec] = 1

    else:
        for j in range(len(intersec)):
            utility_matrix[i, intersec[j]] = locals()['weight' +
str(len(intersec))][tol_i, 0]
```

- 效用层输出

Based on an extended utility matrix $\mathbb{U}_{(2^{\Omega}-1)\times M}$ and the outputs of a DS layer $\boldsymbol{m}$, we can compute the expected utility of an act assigning a sample to set $A$ using the generalized Hurwicz criterion (4) as

$$\mathbb{E}_{m,\nu}(f_A) = \nu\underline{\mathbb{E}}_m(f_A) + (1-\nu)\overline{\mathbb{E}}_m(f_A), \tag{13a}$$

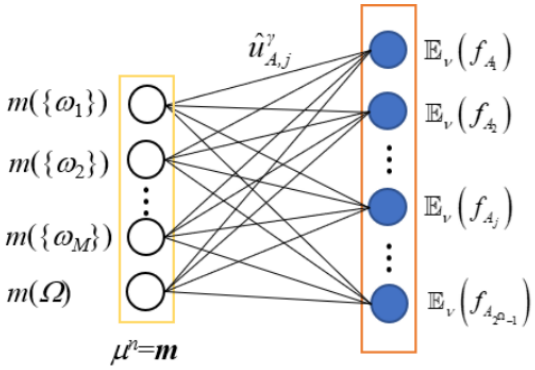



Figure 2: Architecture of the expected utility layer.

where $\underline{\mathbb{E}}_m(f_A)$ and $\overline{\mathbb{E}}_m(f_A)$ are, respectively, the lower and upper expected utilities

$$\underline{\mathbb{E}}_m(f_A) = \sum_{B\subseteq\Omega} m(B) \min_{\omega_k\in B} \widehat{u}_{A,j}, \tag{13b}$$

$$\overline{\mathbb{E}}_m(f_A) = \sum_{B\subseteq\Omega} m(B) \max_{\omega_k\in B} \widehat{u}_{A,j}, \tag{13c}$$

and $\nu$ is the pessimism index, which is considered as a hyperparameter of the proposed classifier. The sample is finally assigned to set $A$ such that

$$A = \arg \max_{\emptyset\neq B\subseteq\Omega} \mathbb{E}_{m,\nu}(f_B). \tag{14}$$

```
import tensorflow as tf


class DM_test(tf.keras.layers.Layer):
    def __init__(self, num_class, num_set, nu):
        super(DM_test, self).__init__()
        self.num_class = num_class
        self.nu = nu
        self.utility_matrix = self.add_weight(
            name='utility_matrix',
            shape=(num_set, num_class),
            initializer='random_normal',
            trainable=False
        )

    def call(self, inputs):
        for i in range(len(self.utility_matrix)):
            if i == 0:
                # （None, 10）
                precise = tf.multiply(inputs[:, 0:self.num_class],
self.utility_matrix[i], name=None)
                precise = tf.reduce_sum(precise, -1, keepdims=True) # （None, 1）

                # （None, 1）
                omega1 = tf.multiply(inputs[:, -1],
tf.reduce_max(self.utility_matrix[i]), name=None)
                omega2 = tf.multiply(inputs[:, -1],
tf.reduce_min(self.utility_matrix[i]), name=None)
```

```
                omega = tf.expand_dims(self.nu * omega1 + (1 - self.nu) *
omega2, -1)
                omega = tf.dtypes.cast(omega, tf.float32)
                # （None, 1）
                utility = tf.add(precise, omega, name=None)

                if i>1:
                    precise = tf.multiply(inputs[:, 0: self.num_class],
self.utility_matrix[i], name=None)
                    precise = tf.reduce_sum(precise, -1, keepdims=True)
                    omega1 = tf.multiply(inputs[:, -1],
tf.reduce_max(self.utility_matrix[i]), name=None)
                    omega2 = tf.multiply(inputs[:, -1],
tf.reduce_min(self.utility_matrix[i]), name=None)
                    omega = tf.expand_dims(self.nu * omega1 + (1 - self.nu) *
omega2, -1)
                    omega = tf.dtypes.cast(omega, tf.float32)
                    # （None, 1）
                    utility_i = tf.add(precise, omega, name=None)
                    # （None, 1023）
                    utility = tf.concat([utility, utility_i], -1)

                return utility
```