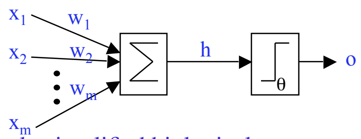
Neural Networks

McCulloch and Pitts Neurons



- Greatly simplified biological neurons
- Sum the inputs
  - If total is less than some threshold, neuron fires
  - Otherwise does not

McCulloch and Pitts Neurons

$$h = \sum_{i=1}^{m} x_i w_i \qquad o = \begin{cases} 1 & h \geq \theta \\ 0 & h < \theta \end{cases}$$

for some threshold θ

- The weight w_j can be positive or negative
  - Inhibitory or exitatory
- Use only a linear sum of inputs
- No refractory period
- Use a simple output instead of a pulse (spike train)

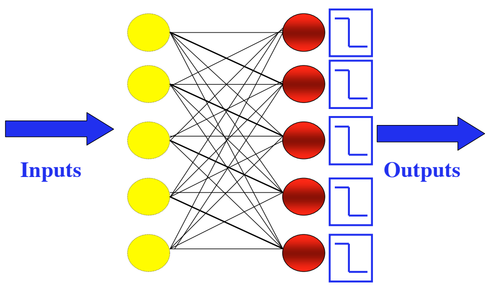
Neural **Networks**

- Can put lots of McCulloch & Pitts neurons together
- Connect them up in any way we like
- In fact, assemblies of the neurons are capable of *universal computation*
  - Can perform any computation that a normal computer can
  - Just have to solve for all the weights w_{ij}

Training Neurons

- Adapting the weights is learning
  - How does the network know it is right?
  - How do we adapt the weights to make the network right more often?
- Training set with target outputs
- Learning rule

The Perceptron Network



Updating the Weights

$$w_{ij} \leftarrow w_{ij} + \Delta w_{ij}$$

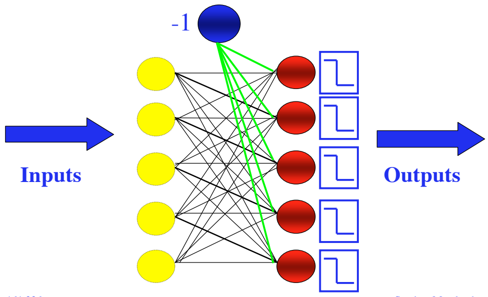
- We want to change the values of the weights
- Aim: minimise the *error* at the output
- If $E = t\text{-}y$, want E to be 0
- Use:

Learning rate Input

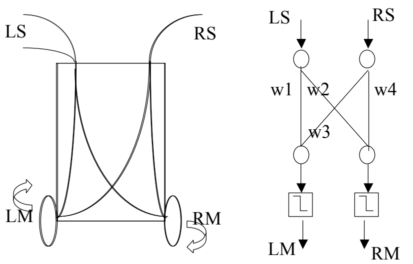
$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

Error

Biases Replace Thresholds



Obstacle Avoidance with the Perceptron

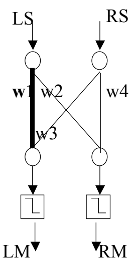


$\eta = 0.3$
$\theta = -0.01$

Obstacle Avoidance with the Perceptron

LS	RS	LM	RM
0	**0**	**1**	**1**
0	1	-1	1
1	0	1	-1
1	1	X	X

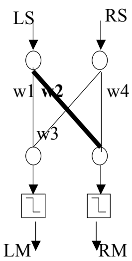
Obstacle Avoidance with the Perceptron



$$\Delta w_{ij} = \eta \cdot (t_j - y_j) \cdot x_i$$

w1=0+0.3 * (1-1) * 0 = 0

Obstacle Avoidance with the Perceptron



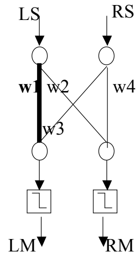
w2=0+0.3 * (1-1) * 0 = 0

And the same for w3, w4

Obstacle Avoidance with the Perceptron

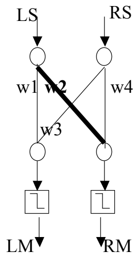
LS	RS	LM	RM
0	0	1	1
0	**1**	**-1**	**1**
1	0	1	-1
1	1	X	X

Obstacle Avoidance with the Perceptron



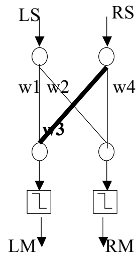
w1=0+0.3 * (-1-1) * 0 = 0

Obstacle Avoidance with the Perceptron



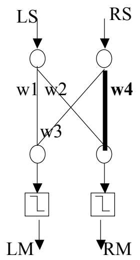
w1=0+0.3 * (-1-1) * 0 = 0
w2=0+0.3 * (1-1) * 0 = 0

Obstacle Avoidance with the Perceptron



w1=0+0.3 * (-1-1) * 0 = 0
w2=0+0.3 * (1-1) * 0 = 0
w3=0+0.3 * (-1-1) * 1 = -0.6

Obstacle Avoidance with the Perceptron

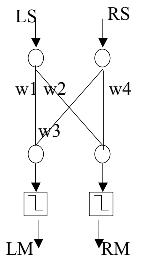


w1=0+0.3 * (-1-1) * 0 = 0
w2=0+0.3 * (1-1) * 0 = 0
w3=0+0.3 * (-1-1) * 1 = -0.6
w4=0+0.3 * (1-1) * 1 = 0

Obstacle Avoidance with the Perceptron

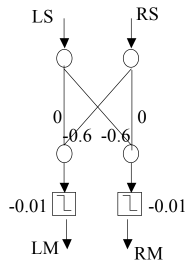
LS	RS	LM	RM
0	0	1	1
0	1	-1	1
1	**0**	**1**	**-1**
1	1	X	X

Obstacle Avoidance with the Perceptron



w1=0+0.3 * (1-1) * 1 = 0
w2=0+0.3 * (-1-1) * 1 = -0.6
w3=-0.6+0.3 * (1-1) * 0 = -0.6
w4=0+0.3 * (-1-1) * 0 = 0

Obstacle Avoidance with the Perceptron



Implementation

- The forward stage of the network is easy:

```
activations = dot(inputs,weights)
activations = where(activations>0,1,0)
```


Implementation

- Need to include the bias node bit:

```
inputs =concatenate((ones
  ((self.nData,1)),inputs),axis=1)
```


Implementation

- And then it is just the weight update:

```
weights += eta*dot(transpose(inputs),
  targets-activations)
```

That's pretty much all there is to it

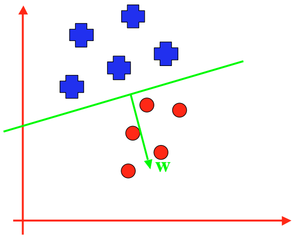
Linear Separability

- Outputs are:

$$y_j = \text{sign}\left(\sum_{i=1}^{n} w_{ij} x_i\right)$$

$$\Rightarrow \mathbf{w} \cdot \mathbf{x} > 0$$

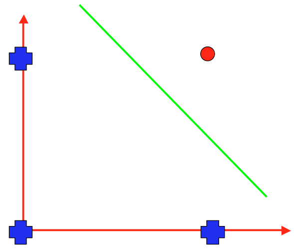
Linear Separability



161.326 1.25 Stephen Marsland

Linear Separability

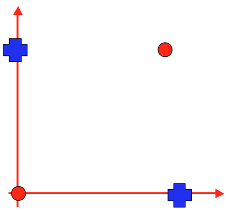
The Binary AND Function



161.326 1.26 Stephen Marsland

Limitations of the Perceptron

Linear Separability

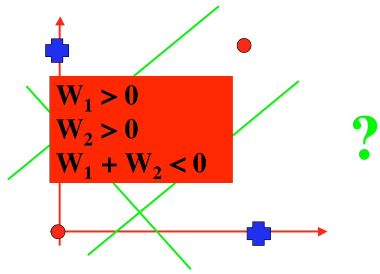


The Exclusive Or (XOR) function.

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

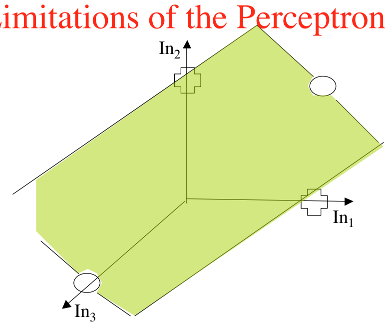
161.326 1.27 Stephen Marsland

Limitations of the Perceptron



161.326 1.28 Stephen Marsland

Limitations of the Perceptron?

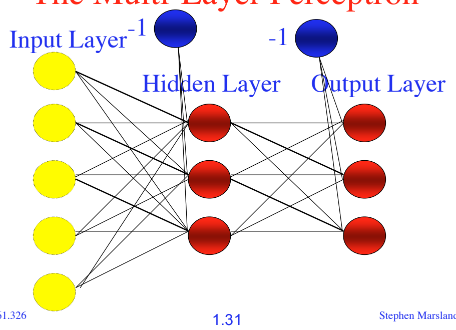


161.326 1.29 Stephen Marsland

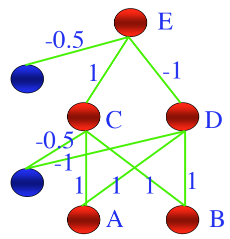
The Multi-Layer Perceptron

161.326 1.30 Stephen Marsland

The Multi-Layer Perceptron



XOR Again



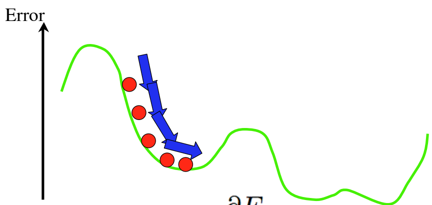
XOR Again

A	B	C_{in}	C_{out}	D_{in}	D_{out}	E
0	0	-0.5	0	-1	0	-0.5
0	1	0.5	1	0	0	0.5
1	0	0.5	1	0	0	0.5
1	1	1.5	1	1	1	-0.5

Gradient Descent

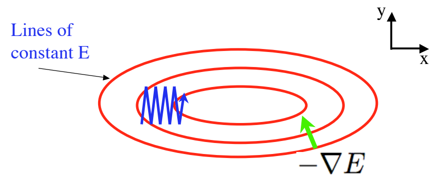
- The MLP can solve XOR
- How do we choose the weights?
- Harder than Perceptron
  - More weights
  - Which weights are wrong? Input-hidden or hidden-output?
- Use gradient descent learning
- Compute gradient -> differentiation

Gradient Descent



$$\Delta w_{ik} = -\eta \frac{\partial E}{\partial w_{ik}}$$

Gradient Descent in 2D



- Local gradient does not point at minimum
- Gradient descent oscillates across valley

An Error Function

- For Perceptron, looked at *(t-y)*
- Better: sum-of-squares error

$$E(\mathbf{w}) = \frac{1}{2}\sum_k (t_k - y_k)^2 = \frac{1}{2}\sum_k \left(t_k - \sum_i w_{ik}x_i\right)^2$$

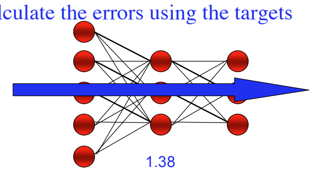
- One more thing - we will ignore the threshold function in the neurons

$$\Rightarrow \frac{\partial E}{\partial w_{ik}} = \sum_k (t_k - y_k)(-x_i)$$

Training MLPs

(1) Forward Pass

- Put the input values in the input layer
- Calculate the activations of the hidden nodes
- Calculate the activations of the output nodes
- Calculate the errors using the targets



Training MLPS

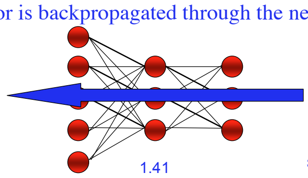
- For output nodes
  - Don't know input
- For hidden nodes
  - Don't know targets
- For extra hidden layers
  - Don't know either
- Therefore, hard to use gradient descent

Backpropagation of Error

Training MLPs

(2) Backward Pass

- From output errors, update last layer of weights
- From these errors, update next layer
- Work backwards through the network
- Error is backpropagated through the network

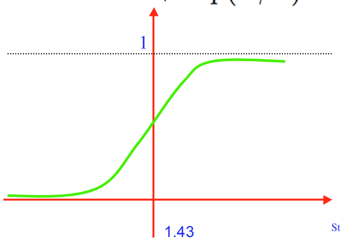


Activation Function

- In the analysis we've ignored the activation function
  - The thresholder is not differentiable
- What do we want in an activation function?
  - Differentiable
  - Should saturate (become constant at ends)
  - Change between saturation values quickly

Sigmoid Functions

$$g(a) = \frac{1}{1+\exp(-\beta a)}$$



Error Terms

- Need to differentiate the sigmoid function
- Gives us the following *error terms* (deltas)
  - For the outputs

$$\delta_k = (y_k - t_k)\, y_k (1 - y_k)$$

  - For the hidden nodes

$$\delta_j = a_j (1 - a_j) \sum_k w_{jk} \delta_k$$

Update Rules

- This gives us the necessary update rules
  - For the weights connected to the outputs:

$$w_{jk} \leftarrow w_{jk} - \eta \delta_k a_j^{\text{hidden}}$$

  - For the weights connect to the hidden nodes:

$$v_{ij} \leftarrow v_{ij} - \eta \delta_j x_i$$

Summary of Backpropagation

- Introduce inputs
- Feed values forward through network
- Compute sum-of-squares error at outputs
- Compute the delta terms at the output by differentiation
- Use this to update the weights from the outputs to the last hidden layer

Summary of Backpropagation

- Once these are correct, propagate errors back to the neurons of the hidden layers
- Compute the delta terms for these neurons
- Use them to update the next set of weights
- Repeat until reach the inputs

Implementation

- Forwards isn't much different to the Perceptron (except do it twice):

```
inputs = concatenate((inputs,-ones
   ((self.ndata,1))),axis=1)
hidden = dot(inputs, weights1);
hidden = 1.0/(1.0+exp(-beta*hidden))
hidden = concatenate((hidden,-ones
   ((ndata,1))), axis=1)
outputs = dot(hidden, weights2);
return 1.0/(1.0+exp(-beta*outputs))
```

Implementation

- The updates are more involved - here's the one for the output weights

```
deltah = zeros(nhidden+1)
for j in range(nhidden+1):
   sumk = sum(weights2[j,:]*deltao[d,:])
   deltah[j] = hidden[d,j]*
       (1.0-hidden[d,j])*sumk
```


Implementation

- One new function

```
random.shuffle(change)
inputs = inputs[change,:]
targets = targets[change,:]
```

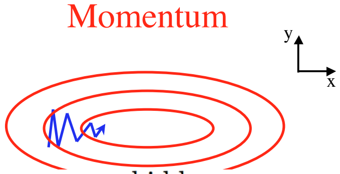

Network Topology

- How many layers?
- How many neurons per layer?
- No good answers
  - At most 3 layers, usually 2
  - Guess size of layers (usually get smaller)
  - Test several different networks

Batch Learning

- When should the weights be updated?
  - After all inputs seen (batch)
    - ✓More accurate estimate of gradient
    - ✓Converges to local minimum faster
  - After each input is seen (sequential)
    - ✓Simpler to program
    - ✓May escape from local minima (change order or presentation)
- Both ways, need many epochs - passes through the whole dataset

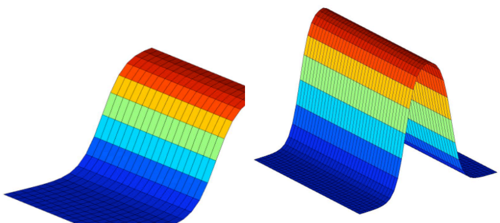
Momentum



$$w_{ij}^{\tau} \leftarrow w_{ij}^{\tau-1}\eta\delta_j a_j^{\text{hidden}} + \alpha\Delta w_{ij}^{\tau-1},$$

- Give more weight to the ball
- Can use smaller learning rate (more stable)
- May overcome local minima

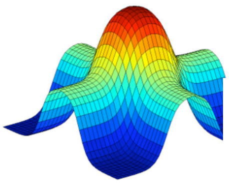
Learning Capacity



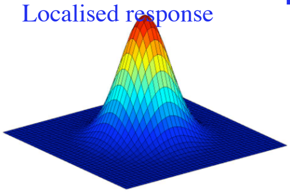
Addition of two sigmoids

Output of one sigmoid

Learning Capacity

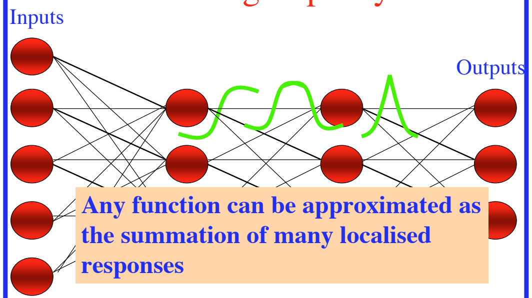


Addition of more ridges and transformation with another sigmoid
Localised response



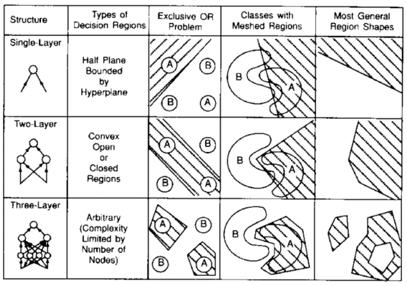
Addition of two ridge
Unique maximum

Learning Capacity



Any function can be approximated as the summation of many localised responses

Decision Boundaries

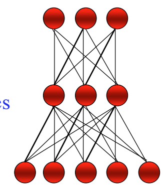


Structure	Types of Decision Regions	Exclusive OR Problem	Classes with Meshed Regions	Most General Region Shapes
Single-Layer	Half Plane Bounded by Hyperplane	A B B A	B A	
Two-Layer	Convex Open or Closed Regions	A B B A	B A	
Three-Layer	Arbitrary (Complexity Limited by Number of Nodes)	A B B A	B A	

Amount of Training

- How much training data is needed?
- How many epochs are needed?

- Data:
  - Count the weights
  - Rule of thumb: use 10 times more data than the number of weights
  - Expect 10% error

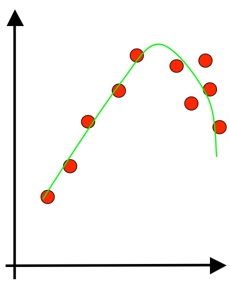


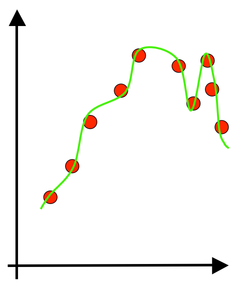
Generalisation

- Aim of neural network learning:
- Generalise from training examples to all possible inputs

- Undertraining is bad
- Overtraining is worse

Overfitting





Overfitting

- MLP has easily enough variation to fit any surface
- We want to learn the data without the noise
- Overtraining lets the network overfit
  - Then does not generalise
  - Function is too complicated

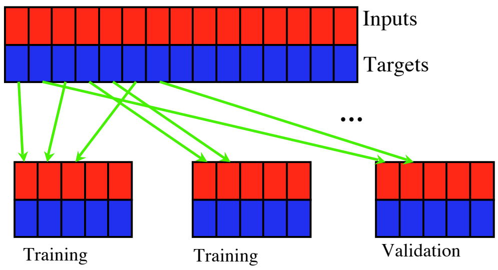
Testing

- How do we evaluate our trained network?
- Can’t just compute the error on the training data - unfair, can’t see overfitting
- Keep a separate testing set
- After training, evaluate on this test set
- How do we check for overfitting?
- Can’t use training or testing sets

Validation

- Keep a third set of data for this
- Train the network on training data
- Periodically, stop and evaluate on validation set
- After training has finished, test on test set
- This is coming expensive on data!

Hold Out Cross Validation



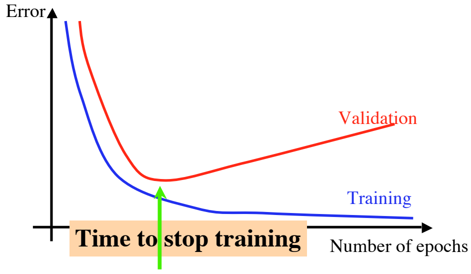
Hold Out Cross Validation

- Partition training data into K subsets
- Train on K-1 of subsets, validate on Kth
- Repeat for new network, leaving out a different subset
- Choose network that has best validation error
- Traded off data for computation
- Extreme version: leave-one-out

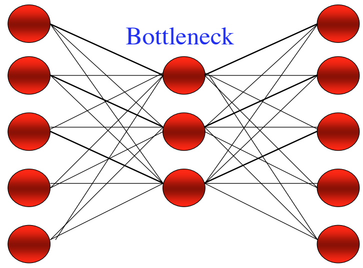
Early Stopping

- When should we stop training?
  - Could set a minimum training error
    - ✓Danger of overfitting
  - Could set a number of epochs
    - ✓Danger of underfitting or overfitting
  - Can use the validation set
    - ✓Measure the error on the validation set during training

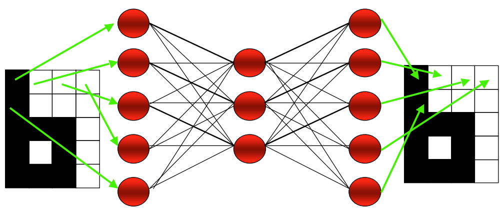
Early Stopping



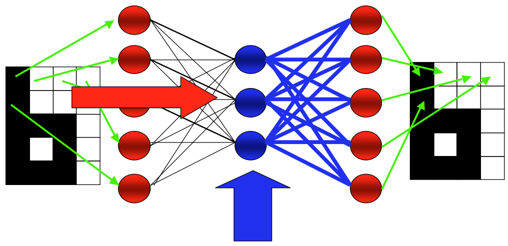
Autoassociative Network



Autoassociative Network

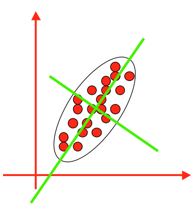


Autoassociative Network



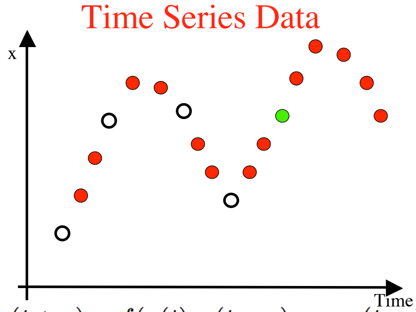
Store the second layer of weights and these activations

Principal Components



- Set of perpendicular axes (rotation and translation of red axes) in the directions of maximum variation in the data
- Eigenvectors of the covariance matrix

Time Series Data



$$y = x(t+\tau) = f(x(t), x(t-\tau), \ldots, x(t-k\tau))$$

Classification

What is Classification?

- Based on exemplars of each class, choose which of N classes an input belongs to
- Split the input set into the N classes
- What about inputs that aren't in any of the classes?
  - Novelty detection

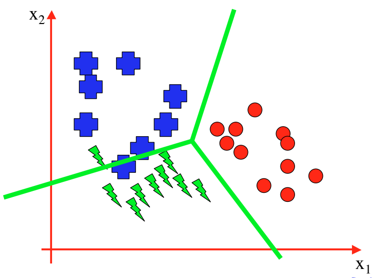
Example: Vending Machine

- Recognise coins fed into machine
- Choose features
  - Weight
  - Diameter
  - Shape
  - Colour

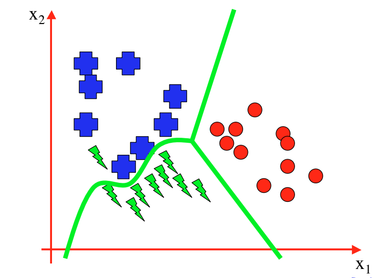
Example: Vending Machine

- Based on examplars of each class, calculate feature values
- Use these to recognise coins
- What about foreign coins?
  - Novelty detection

Decision Boundaries



Decision Boundaries



Choosing Features

- Art or science?
  - Not too many features - curse of dimensionality
  - Enough, so can separate classes
  - Not all features are equally useful

Classification with the Multi-Layer Perceptron

Classification & the MLP

- Inputs to MLP are feature values
- What about outputs?
- Use one neuron with:

$$\left\{ \begin{array}{ll} C_1 & \text{if } y \leq -0.5 \\ C_2 & \text{if } -0.5 < y \leq 0 \\ C_3 & \text{if } 0 < y \leq 0.5 \\ C_4 & \text{if } y > 0.5 \end{array} \right\}$$

1 of N Encoding

- 1 output neuron for each class
- Target data has 1 output true, rest false
- In use, pick class as the neuron with the highest activation
- Often use softmax activation for output nodes

$$\frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)}$$