NLP Programming Tutorial 7 - Neural Networks

Graham Neubig
Nara Institute of Science and Technology (NAIST)

Prediction Problems

Given x, predict y

Example we will use:

- Given an introductory sentence from Wikipedia
- Predict whether the article is about a person

Given		Predict
Gonso was a Sanron sect priest (754-827) in the late Nara and early Heian periods.	→	Yes!
Shichikuzan Chigogataki Fudomyoo is a historical site located at Magura, Maizuru City, Kyoto Prefecture.	→	No!

- This is binary classification (of course!)

Linear Classifiers

$$y = \text{sign}(\mathbf{w} \cdot \boldsymbol{\varphi}(x))$$
$$= \text{sign}\left(\sum_{i=1}^{I} w_i \cdot \varphi_i(x)\right)$$

- x: the input
- **φ(x)**: vector of feature functions $\{\varphi_1(x), \varphi_2(x), \ldots, \varphi_I(x)\}$
- **w**: the weight vector $\{w_1, w_2, \ldots, w_I\}$
- y: the prediction, +1 if "yes", -1 if "no"
  - (sign(v) is +1 if v >= 0, -1 otherwise)

Example Feature Functions: Unigram Features

- Equal to "number of times a particular word appears"

x = A site , located in Maizuru , Kyoto

$\varphi_{\text{unigram “A”}}(x) = 1$ $\varphi_{\text{unigram “site”}}(x) = 1$ $\varphi_{\text{unigram “,”}}(x) = 2$

$\varphi_{\text{unigram “located”}}(x) = 1$ $\varphi_{\text{unigram “in”}}(x) = 1$

$\varphi_{\text{unigram “Maizuru”}}(x) = 1$ $\varphi_{\text{unigram “Kyoto”}}(x) = 1$

$\left.\begin{array}{l} \varphi_{\text{unigram “the”}}(x) = 0 \quad \varphi_{\text{unigram “temple”}}(x) = 0 \\ \ldots \end{array}\right\}$ The rest are all 0

- For convenience, we use feature names ($\varphi_{\text{unigram “A”}}$) instead of feature indexes (φ_1)

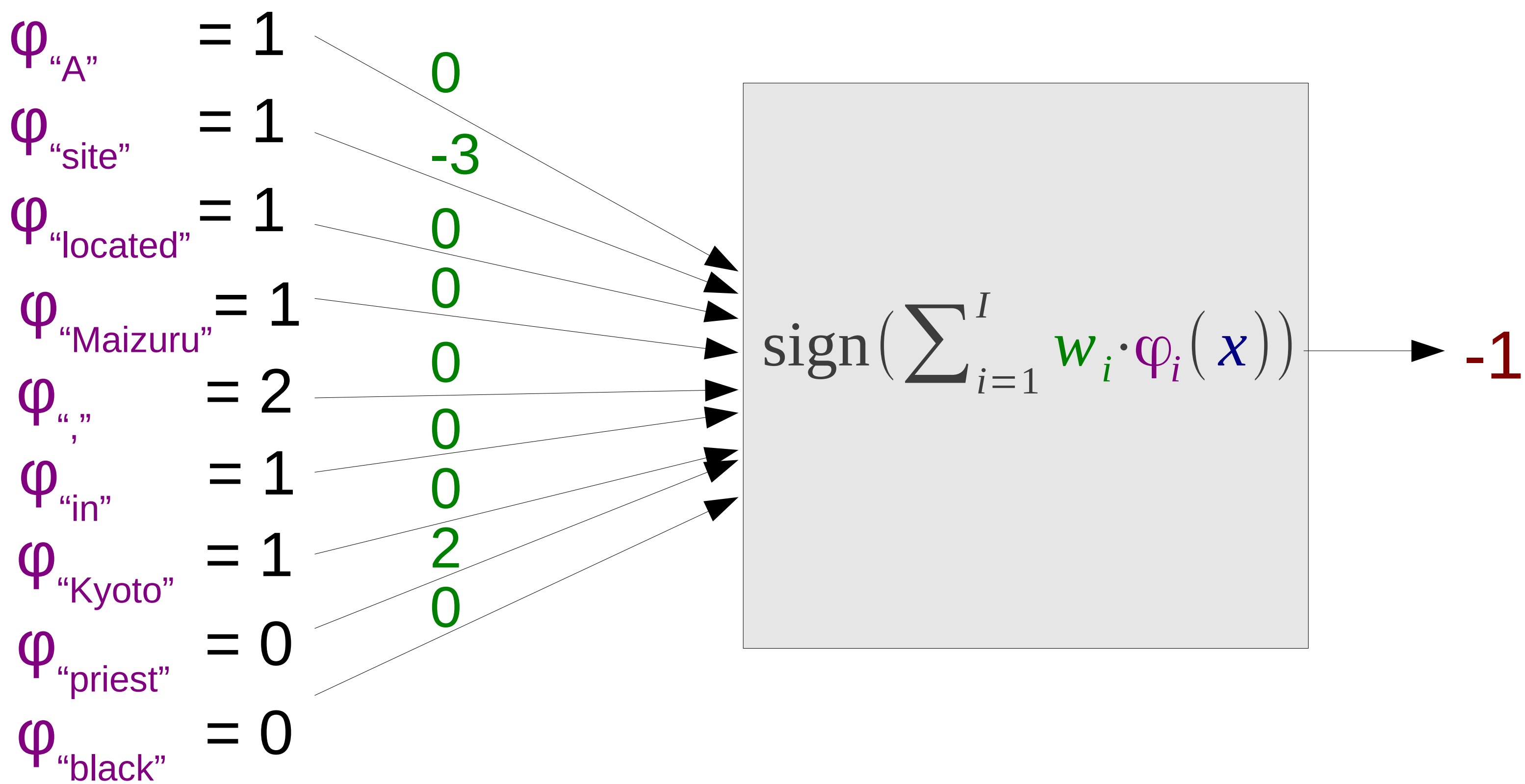
Calculating the Weighted Sum

x = A site , located in Maizuru , Kyoto

$$
\begin{array}{llcllcl}
\varphi_{\text{unigram "A"}}(x) & = 1 & & w_{\text{unigram "a"}} & = 0 & & 0 \; + \\
\varphi_{\text{unigram "site"}}(x) & = 1 & & w_{\text{unigram "site"}} & = -3 & & -3 \; + \\
\varphi_{\text{unigram "located"}}(x) & = 1 & & w_{\text{unigram "located"}} & = 0 & & 0 \; + \\
\varphi_{\text{unigram "Maizuru"}}(x) & = 1 & & w_{\text{unigram "Maizuru"}} & = 0 & & 0 \; + \\
\varphi_{\text{unigram ","}}(x) & = 2 & * & w_{\text{unigram ","}} & = 0 & = & 0 \; + \\
\varphi_{\text{unigram "in"}}(x) & = 1 & & w_{\text{unigram "in"}} & = 0 & & 0 \; + \\
\varphi_{\text{unigram "Kyoto"}}(x) & = 1 & & w_{\text{unigram "Kyoto"}} & = 0 & & 0 \; + \\
\varphi_{\text{unigram "priest"}}(x) & = 0 & & w_{\text{unigram "priest"}} & = 2 & & 0 \; + \\
\varphi_{\text{unigram "black"}}(x) & = 0 & & w_{\text{unigram "black"}} & = 0 & & 0 \; + \\
 & & \ldots & & & & \ldots \\
 & & & & & & = \\
 & & & & & & -3 \rightarrow \text{No!}
\end{array}
$$

The Perceptron

- Think of it as a “machine” to calculate a weighted sum



Perceptron in Numpy

What is Numpy?

- A powerful computation library in Python
- Vector and matrix multiplication is easy
- A part of SciPy (a more extensive scientific computing library)

Example of Numpy (Vectors)

```
import numpy as np

a = np.array( [20,30,40,50] )
b = np.array( [0,1,2,3] )
print(a - b)                # Subtract each element
print(b ** 2)               # Take the power of each element
print(10 * np.tanh(b))      # Hyperbolic tangent * 10 of each element
print(a < 35)               # Check if each element is less than 35
```

Example of Numpy (Matrices)

```
import numpy as np

A = np.array( [[1,1],
               [0,1]] )
B = np.array( [[2,0],
               [3,4]] )

print(A * B)           # elementwise product
print(np.dot(A,B))     # dot product
print(B.T)             # transpose
```

Perceptron Prediction

```
predict_one(w, phi)
    score = 0
    for each name, value in phi          # score = w*φ(x)
        if name exists in w
            score += value * w[name]
    return (1 if score >= 0 else -1)
```

↓ numpy

```
predict_one(w, phi)
    score = np.dot( w, phi )
    return (1 if score[0] >= 0 else -1)
```

Converting Words to IDs

- numpy uses vectors, so we want to convert names into indices

```
ids = defaultdict(lambda: len(ids))  # A trick to convert to IDs

CREATE_FEATURES(x):
    create list phi
    split x into words
    for word in words
        phi[ids["UNI:"+word]] += 1
    return phi
```

Initializing Vectors

- Create a vector as large as the number of features
- With zeros

```
w = np.zeros(len(ids))
```

- Or random between [-0.5,0.5]

```
w = np.random.rand(len(ids)) – 0.5
```

Perceptron Training Pseudo-code

```
# Count the features and initialize the weights
create map ids
for each labeled pair x, y in the data
    create_features(x)
w = np.zeros(len(ids))

# Perform training
for I iterations
    for each labeled pair x, y in the data
        phi = create_features(x)
        y' = predict_one(w, phi)
        if y' != y
            update_weights(w, phi, y)

print w to weight_file
print ids to id_file
```

Perceptron Prediction Code

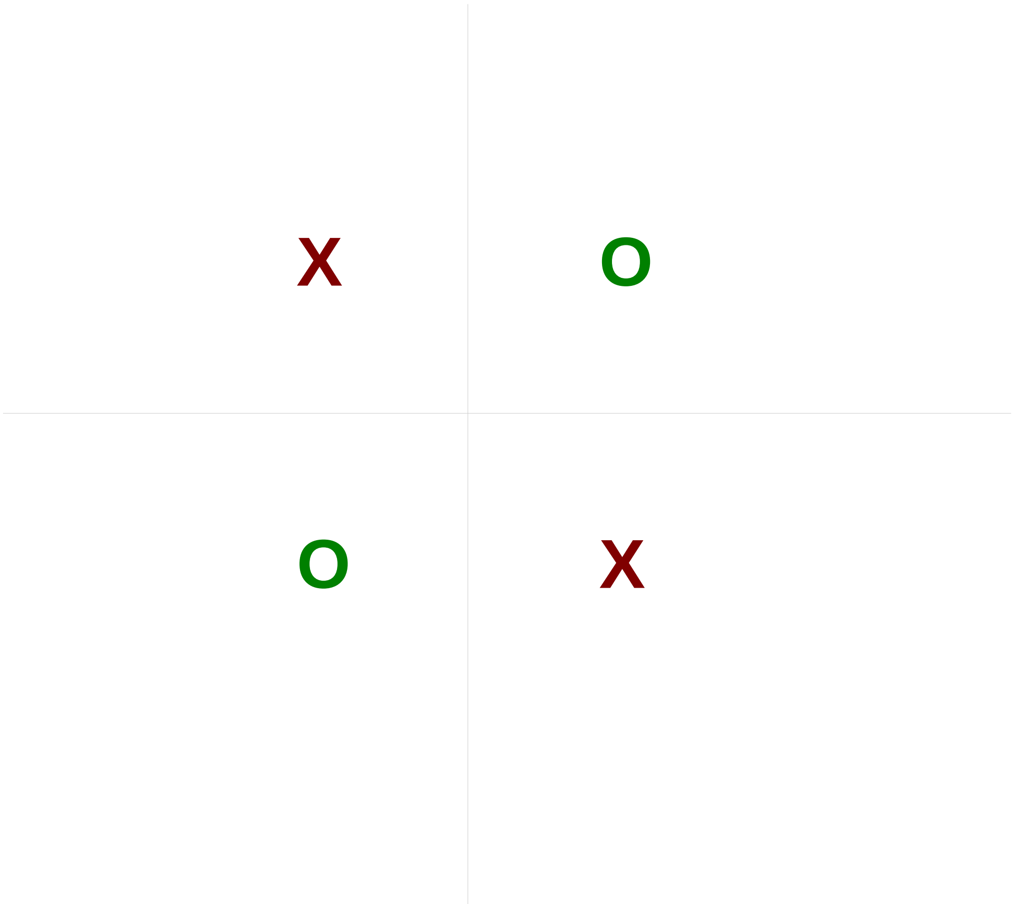
```
read ids from id_file
read w from weights_file

for each example x in the data
    phi = create_features(x)
    y' = predict_one(w, phi)
```

Neural Networks

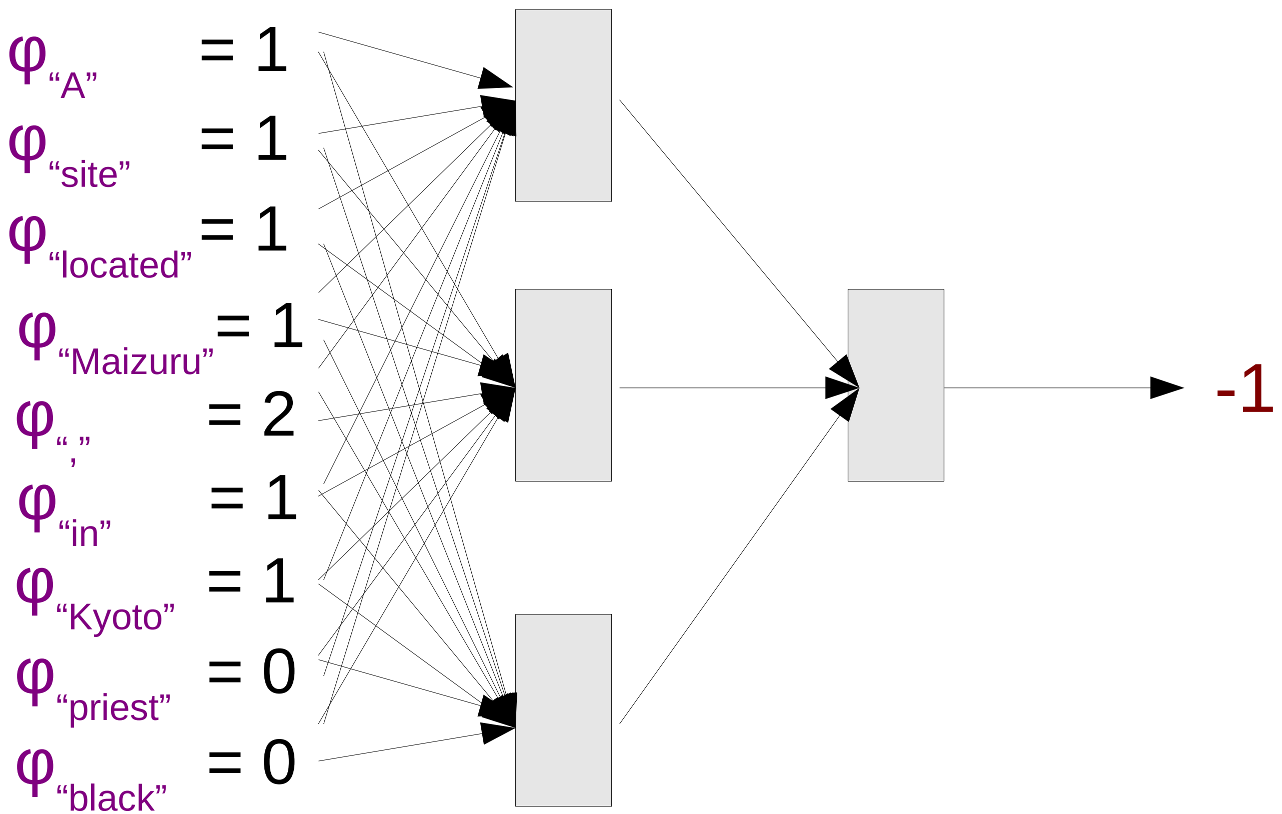
Problem: Only Linear Classification

- Cannot achieve high accuracy on non-linear functions



Neural Networks

- Connect together multiple perceptrons

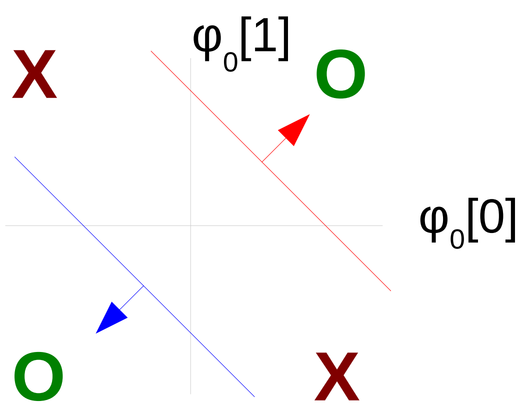


- Motivation: Can represent non-linear functions!

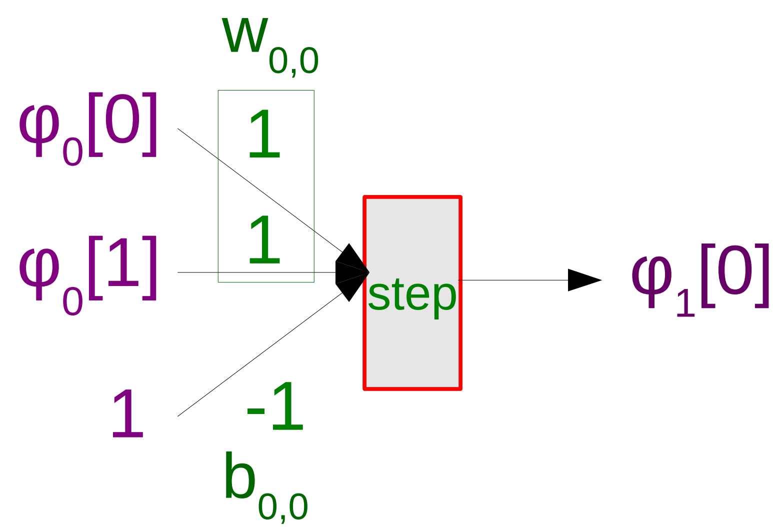
Example

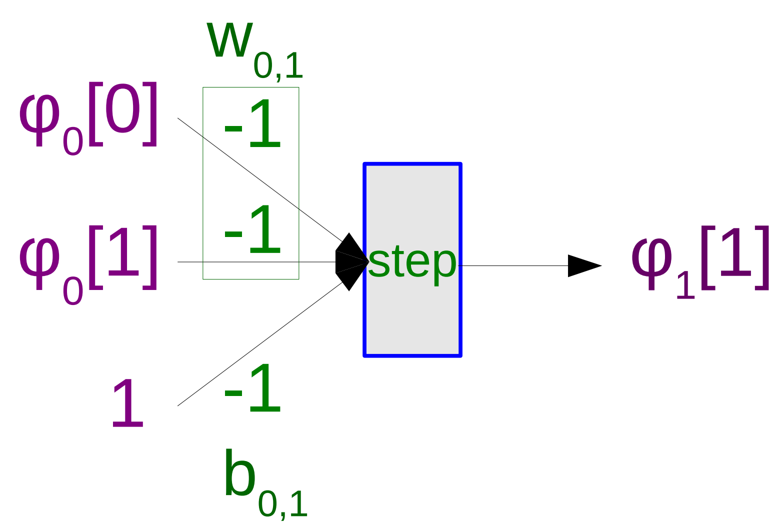
- Create two classifiers

$\varphi_0(x_1) = \{-1, 1\}$ $\varphi_0(x_2) = \{1, 1\}$



$\varphi_0(x_3) = \{-1, -1\}$ $\varphi_0(x_4) = \{1, -1\}$



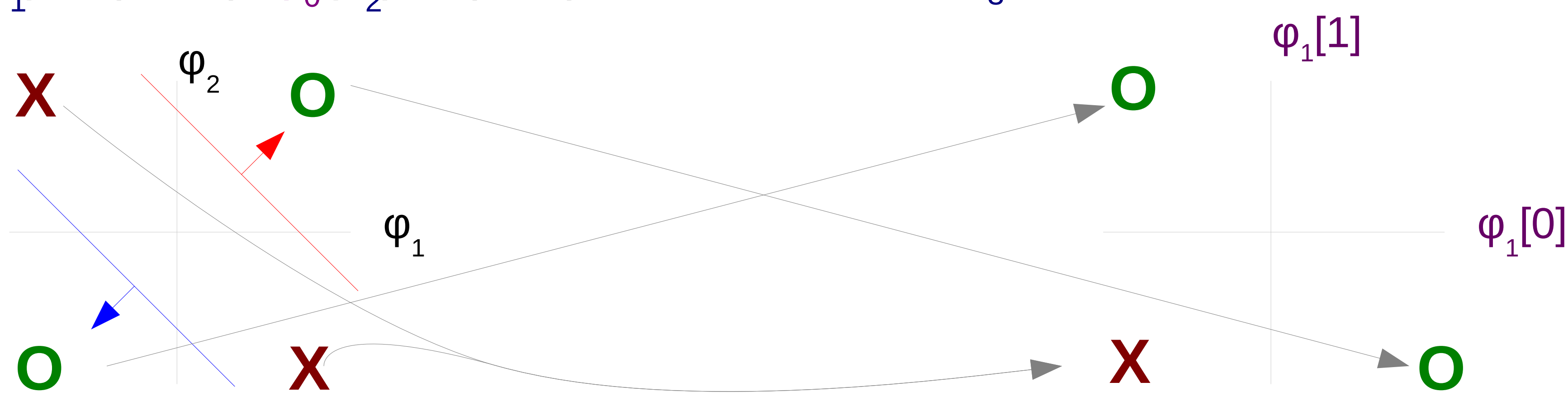


Example

- These classifiers map to a new space

$\varphi_0(x_1) = \{-1, 1\}$ $\varphi_0(x_2) = \{1, 1\}$

$\varphi_1(x_3) = \{-1, 1\}$



$\varphi_0(x_3) = \{-1, -1\}$ $\varphi_0(x_4) = \{1, -1\}$

$\varphi_1(x_1) = \{-1, -1\}$ $\varphi_1(x_2) = \{1, -1\}$

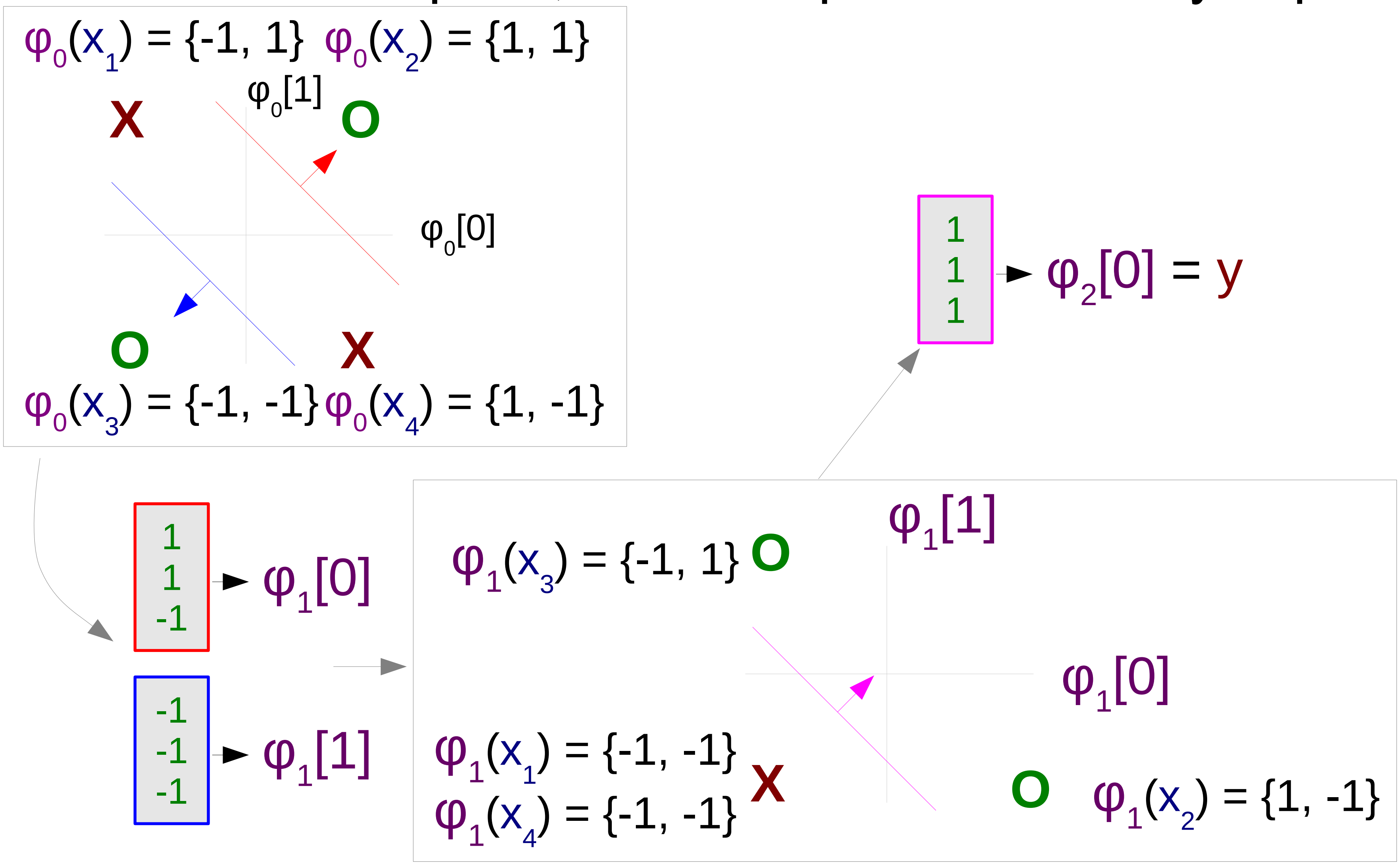
$\varphi_1(x_4) = \{-1, -1\}$

$$\begin{bmatrix}1\\1\\-1\end{bmatrix} \rightarrow \varphi_1[0]$$

$$\begin{bmatrix}-1\\-1\\-1\end{bmatrix} \rightarrow \varphi_1[1]$$

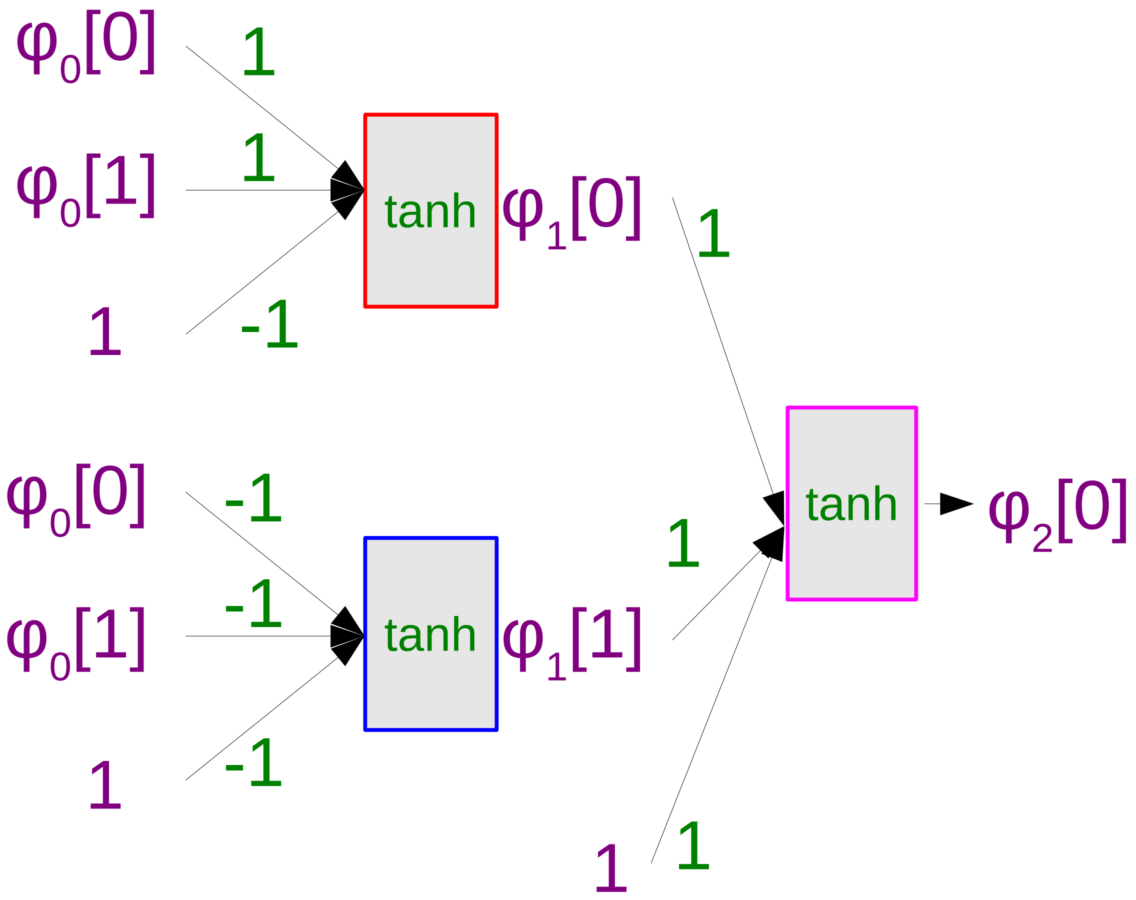
Example

- In the new space, the examples are linearly separable!



Example

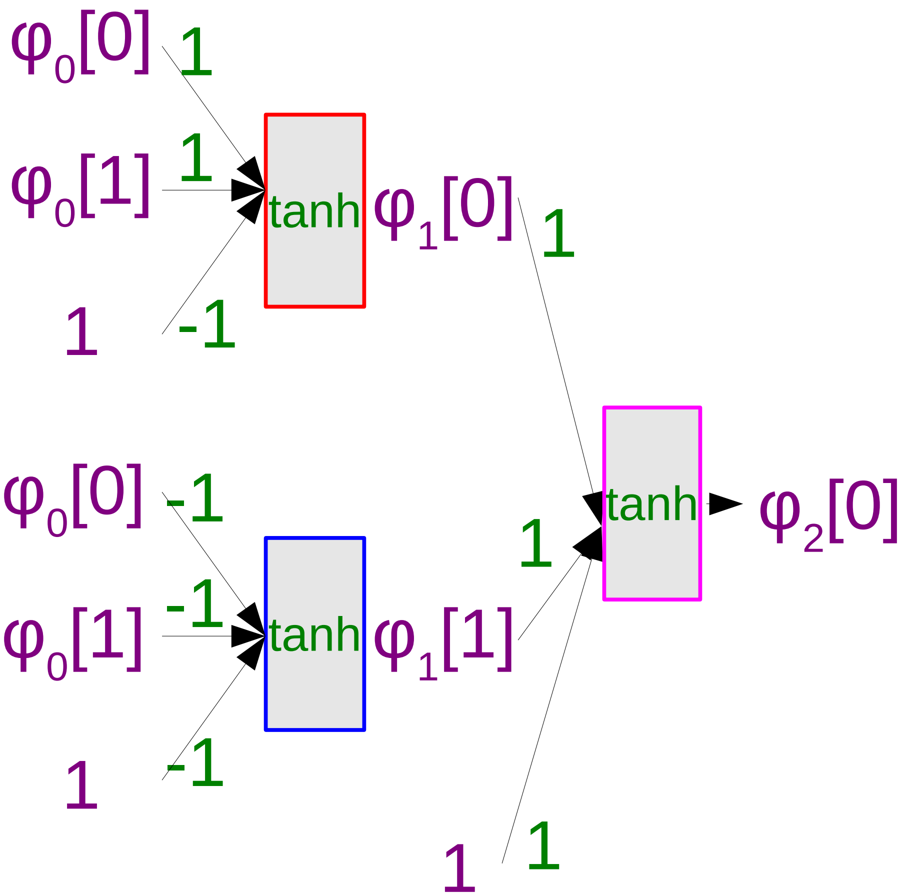
- The final net



Calculating a Net (with Vectors)

<u>Input</u>

φ_0 = np.array([1, -1])



<u>First Layer Output</u>

$w_{0,0}$ = np.array([1, 1])

$b_{0,0}$ = np.array([-1])

$w_{0,1}$ = np.array([-1, -1])

$b_{0,1}$ = np.array([-1])

φ_1 = np.zeros(2)

$\varphi_1[0]$ = np.tanh($\varphi_0 w_{0,0}$ + $b_{0,0}$)[0]

$\varphi_1[1]$ = np.tanh($\varphi_0 w_{0,1}$ + $b_{0,1}$)[0]

<u>Second Layer Output</u>

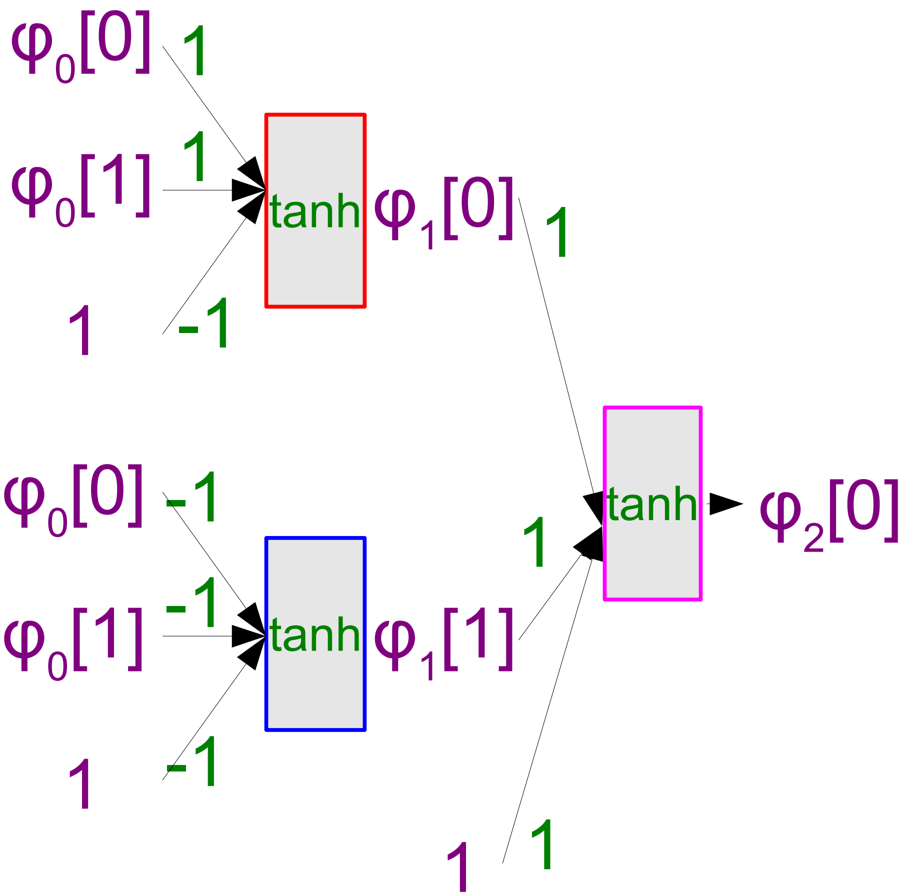
$w_{1,0}$ = np.array([1, 1])

$b_{1,0}$ = np.array([-1])

φ_2 = np.zeros(1)

$\varphi_2[0]$ = np.tanh($\varphi_1 w_{1,0}$ + $b_{1,0}$)[0]

Calculating a Net (with Matrices)



__Input__

$\boldsymbol{\varphi}_0$ = np.array([1, -1])

__First Layer Output__

$\mathbf{w}_0$ = np.array([[1, 1], [-1,-1]])

$\mathbf{b}_0$ = np.array([-1, -1])

$\boldsymbol{\varphi}_1$ = np.tanh(np.dot($\mathbf{w}_0$, $\boldsymbol{\varphi}_0$) + $\mathbf{b}_0$)

__Second Layer Output__

$\mathbf{w}_1$ = np.array([[1, 1]])

$\mathbf{b}_1$ = np.array([-1])

$\boldsymbol{\varphi}_2$ = np.tanh(np.dot($\mathbf{w}_1$, $\boldsymbol{\varphi}_1$) + $\mathbf{b}_1$)

Forward Propagation Code

```
forward_nn(network, φ0)
    φ= [ φ0 ] # Output of each layer
    for each layer i in 0 .. len(network)-1:
        w, b = network[i]
        # Calculate the value based on previous layer
        φ[i] = np.tanh( np.dot( w, φ[i-1] ) + b )
    return φ # Return the values of all layers
```

Calculating Error with tanh

- Error function: Squared error

$$\text{err} = (y' - y)^2 / 2$$

Correct Answer → y' Net Output → y

- Gradient of the error:

$$\text{err}' = \delta = y' - y$$

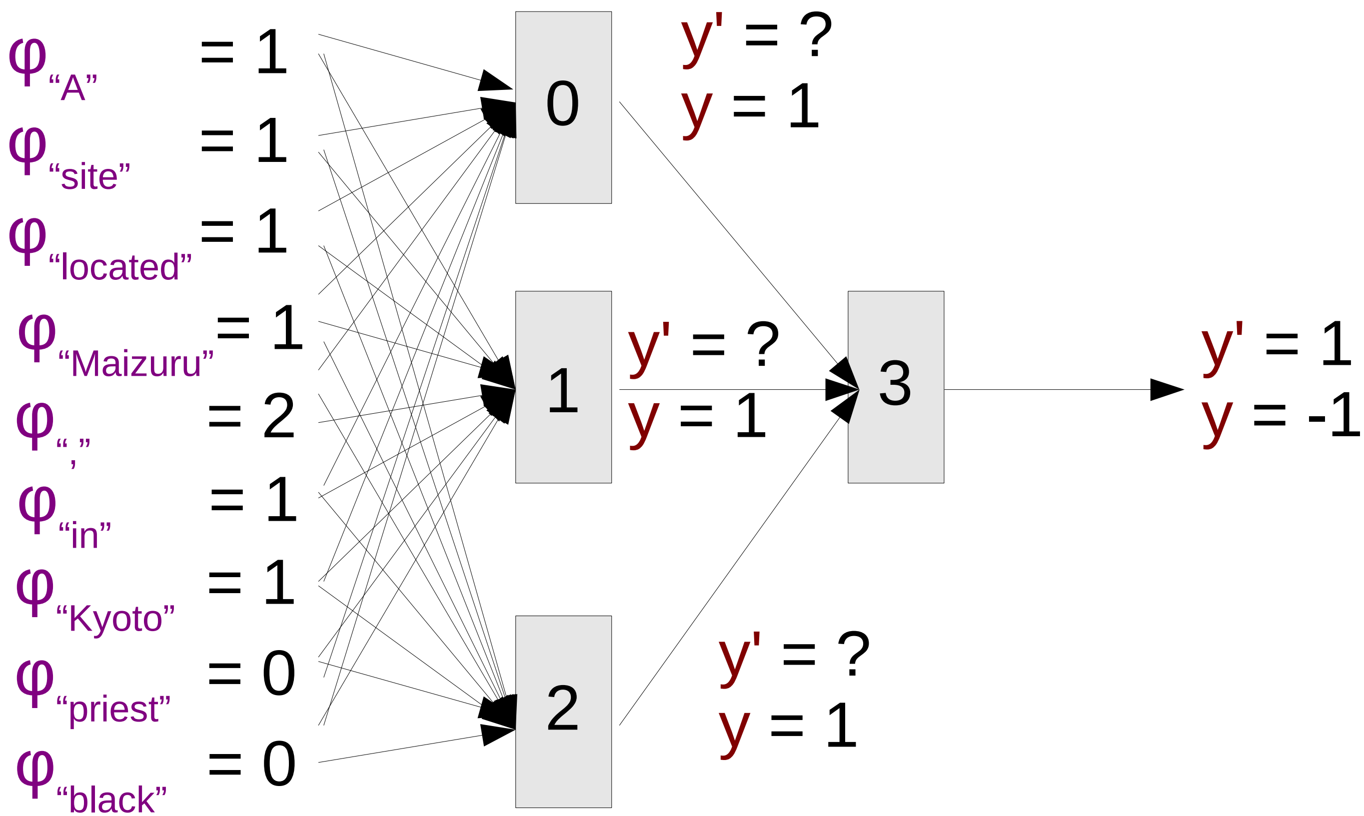
- Update of weights:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \lambda \cdot \delta \cdot \boldsymbol{\varphi}(x)$$

- λ is the learning rate

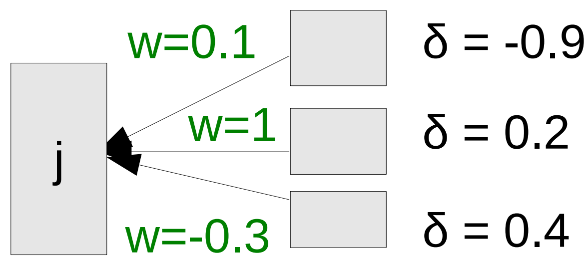
Problem:
Don't know error for hidden layers!

- The NN only gets the correct label for the final layer



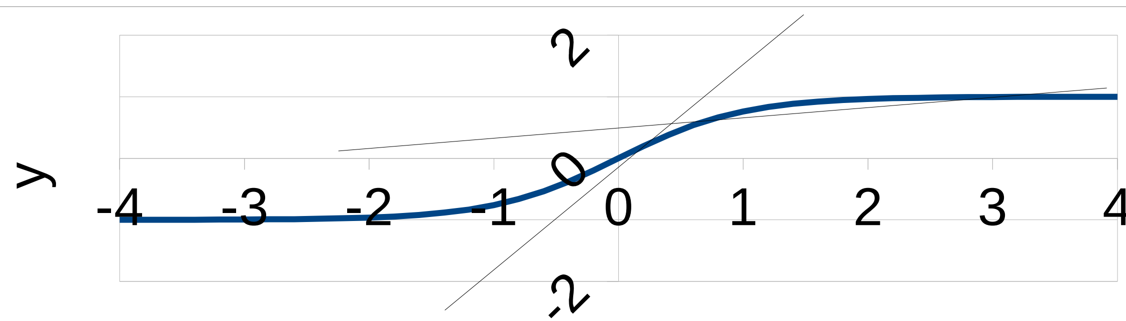
Solution: Back Propagation

- Propagate the error backwards through the layers



$$\sum_i \delta_i w_{j,i}$$

- Also consider the gradient of the non-linear function



$$d\tanh(\varphi(x)*w) = 1 - \tanh(\varphi(x)*w)^2 = 1 - y_j^2$$

- Together:

$$\delta_j = (1 - y_j^2) \sum_i \delta_i w_{j,i}$$

Back Propagation

<u>Error of the Output</u>

$\boldsymbol{\delta}_2$ = np.array([y'-y])

<u>Error of the First Layer</u>

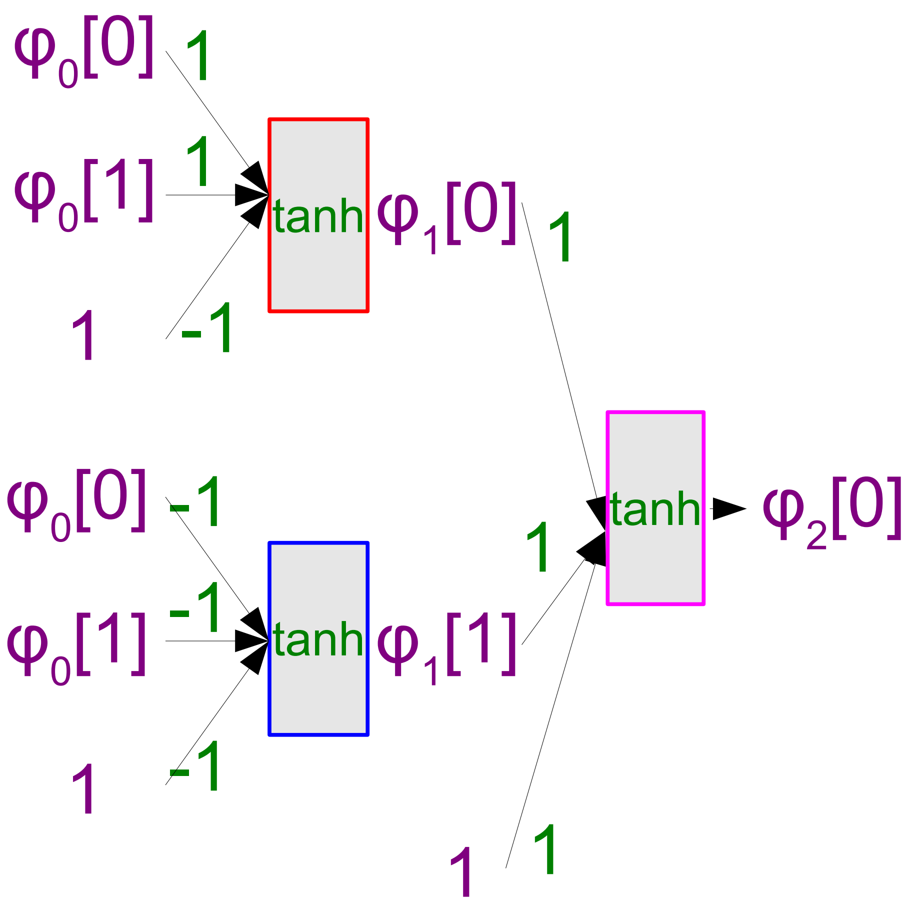
$\boldsymbol{\delta'}_2 = \boldsymbol{\delta}_2 * (1-\boldsymbol{\varphi}_2^{\,2})$

$\boldsymbol{\delta}_1$ = np.dot($\boldsymbol{\delta'}_2$, $\mathbf{w}_1$)

<u>Error of the 0th Layer</u>

$\boldsymbol{\delta'}_1 = \boldsymbol{\delta}_1 * (1-\boldsymbol{\varphi}_1^{\,2})$

$\boldsymbol{\delta}_0$ = np.dot($\boldsymbol{\delta'}_1$, $\mathbf{w}_0$)



Back Propagation Code

```
backward_nn(net, φ, y')
    J = len(net)
    create array δ = [ 0, 0, …, np.array([y' – φ[J][0]]) ] # length J+1
    create array δ' = [ 0, 0, …, 0 ]
    for i in J-1 .. 0:
        δ'[i+1] = δ[i+1] * (1 – φ[i+1]²)
        w, b = net[i]
        δ[i] = np.dot(δ'[i+1], w)
    return δ'
```

Updating Weights

- Finally, use the error to update weights
- Grad. of weight ***w*** is outer prod. of next ***δ'*** and prev ***φ***

$$\text{-derr/d}\mathbf{w}_i = \text{np.outer}(\ \boldsymbol{\delta}'_{i+1},\ \boldsymbol{\varphi}_i\ )$$

- Multiply by learning rate and update weights

$$\mathbf{w}_i \mathrel{+}= \lambda * \text{-derr/d}\mathbf{w}_i$$

- For the bias, input is 1, so simply **δ'**

$$\text{-derr/d}\mathbf{b}_i = \boldsymbol{\delta}'_{i+1}$$

$$\mathbf{b}_i \mathrel{+}= \lambda * \text{-derr/d}\mathbf{b}_i$$

Weight Update Code

```
update_weights(net, φ, δ', λ)
    for i in 0 .. len(net)-1:
        w, b = net[i]
        w += λ * np.outer( δ'[i+1], φ[i] )
        b += λ * δ'[i+1]
```

Overall View of Learning

```
# Create features, initialize weights randomly
create map ids, array feat_lab
for each labeled pair x, y in the data
    add (create_features(x), y ) to feat_lab
initialize net randomly

# Perform training
for I iterations
    for each labeled pair φ0, y in the feat_lab
        φ= forward_nn(net, φ0)
        δ'= backward_nn(net, φ, y)
        update_weights(net, φ, δ', λ)

print net to weight_file
print ids to id_file
```

Tricks to Learning Neural Nets

Stabilizing Training

- NNs have many parameters, objective is non-convex → training is less stable
- Initializing Weights:
  - Randomly, e.g. uniform distribution between -0.1-0.1
- Learning Rate:
  - Often start at 0.1
  - Compare error with previous iteration, and reduce rate a little if error has increased (*= 0.9 or *= 0.5)
- Hidden Layer Size:
  - Usually just try several sizes and pick the best one

Testing Neural Nets

- **Easy Way:** Print the error and make sure it is more or less decreasing everty iteration
- **Better Way:** Use the finite differences method

Idea:

When updating weights, calculate grad. for w_i: derr/dw_i

If we change that weight by a small amount (ω):

If	w_i = x ↓ err = y	then	w_i = x + ω ↓ err ≈ y + ω * derr/dw_i

In the finite differences method, we change w_i by ω and check to make sure that the error changes by the expected amount

Details: http://cs231n.github.io/neural-networks-3/

Exercise

Exercise (1)

- Implement
  - train-nn: A program to learn a NN
  - test-nn: A program to test the learned NN
- Test
  - Input: test/03-train-input.txt
  - One iteration, one hidden layer, to hidden nodes
  - Check the update by hand

Exercise (2)

- Train `data/titles-en-train.labeled`
- Predict `data/titles-en-test.word`
- Measure Accuracy
    - script/grade-prediction.py data-en/titles-en-test.labeled your_answer
- Compare
    - Simple perceptron, SVM, or logistic regression
    - Numbers of nodes, learning rates, initialization ranges
- Challenge
    - Implement nets with multiple hidden layers
    - Implement method to decrease learning rate when error increases

Thank You!