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

**Give**

Gonso was a Sanron sect priest (754-827) in the late Nara and early Heian periods.

**Predic**

Yes!

Shichikuzan Chigogataki Fudomyoo is a historical site located at Magura, Maizuru City, Kyoto Prefecture.

**Predic**

No!

- This is **binary classification** (of course!)
Linear Classifiers

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

- \( x \): the input
- \( \varphi(x) \): 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”  
  - (\( \text{sign}(v) \) is +1 if \( v \geq 0 \), -1 otherwise)
Example Feature Functions: Unigram Features

- Equal to “number of times a particular word appears”

\[
\begin{align*}
x &= \text{A site , located in Maizuru , Kyoto} \\
\phi_{\text{unigram } “A”}(x) &= 1 \\
\phi_{\text{unigram } “site”}(x) &= 1 \\
\phi_{\text{unigram } “,”}(x) &= 2 \\
\phi_{\text{unigram } “located”}(x) &= 1 \\
\phi_{\text{unigram } “in”}(x) &= 1 \\
\phi_{\text{unigram } “Maizuru”}(x) &= 1 \\
\phi_{\text{unigram } “Kyoto”}(x) &= 1 \\
\phi_{\text{unigram } “the”}(x) &= 0 \\
\phi_{\text{unigram } “temple”}(x) &= 0 \\
\phi_{\text{unigram } “the”} &\text{ (x) } = 0 \quad \text{The rest are all 0}
\end{align*}
\]

- For convenience, we use feature names \((\phi_{\text{unigram } “A”})\) instead of feature indexes \((\phi_1)\)
Calculating the Weighted Sum

\[ x = \text{A site, located in Maizuru, Kyoto} \]

<table>
<thead>
<tr>
<th>( \phi ) unigram</th>
<th>( \phi(x) )</th>
<th>( W ) unigram</th>
<th>( W(x) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;A&quot;</td>
<td>1</td>
<td>&quot;a&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;site&quot;</td>
<td>1</td>
<td>&quot;site&quot;</td>
<td>-3</td>
</tr>
<tr>
<td>&quot;located&quot;</td>
<td>1</td>
<td>&quot;located&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;Maizuru&quot;</td>
<td>1</td>
<td>&quot;Maizuru&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;,,&quot;</td>
<td>2</td>
<td>&quot;,,&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;in&quot;</td>
<td>1</td>
<td>&quot;in&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;Kyoto&quot;</td>
<td>1</td>
<td>&quot;Kyoto&quot;</td>
<td>0</td>
</tr>
<tr>
<td>&quot;priest&quot;</td>
<td>0</td>
<td>&quot;priest&quot;</td>
<td>2</td>
</tr>
<tr>
<td>&quot;black&quot;</td>
<td>0</td>
<td>&quot;black&quot;</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-3</td>
</tr>
</tbody>
</table>

\[ \ldots \]

\[ \ldots \]

\[ = -3 \rightarrow \text{No!} \]
The Perceptron

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

\[
sign\left(\sum_{i=1}^{I} w_i \cdot \phi_i(x)\right)
\]

\[
\begin{align*}
\phi_{\text{"A"}} &= 1 \\
\phi_{\text{"site"}} &= 1 \\
\phi_{\text{"located"}} &= 1 \\
\phi_{\text{"Maizuru"}} &= 1 \\
\phi_{\text{","}} &= 2 \\
\phi_{\text{"in"}} &= 1 \\
\phi_{\text{"Kyoto"}} &= 1 \\
\phi_{\text{"priest"}} &= 0 \\
\phi_{\text{"black"}} &= 0 
\end{align*}
\]
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)

```python
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)

```python
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

```python
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)
```

```python
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

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

CREATE_FEATURES(x):
    create listphi
    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

\[ w = \text{np.zeros}(\text{len}(\text{ids})) \]

- Or random between \([-0.5, 0.5]\)

\[ w = \text{np.random.rand}(\text{len}(\text{ids})) - 0.5 \]
Perceptron Training Pseudo-code

```python
# 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

```python
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

\[
\begin{align*}
\phi_{\text{"A"}} &= 1 \\
\phi_{\text{"site"}} &= 1 \\
\phi_{\text{"located"}} &= 1 \\
\phi_{\text{"Maizuru"}} &= 1 \\
\phi_{\text{".\"}} &= 2 \\
\phi_{\text{"in"}} &= 1 \\
\phi_{\text{"Kyoto"}} &= 1 \\
\phi_{\text{"priest"}} &= 0 \\
\phi_{\text{"black"}} &= 0 \\
\end{align*}
\]

- Motivation: Can represent non-linear functions!
Example

- Create two classifiers

\[ \varphi_0(x_1) = \{-1, 1\} \quad \varphi_0(x_2) = \{1, 1\} \]
\[ \varphi_0(x_3) = \{-1, -1\} \quad \varphi_0(x_4) = \{1, -1\} \]
Example

- These classifiers map to a new space

\[ \varphi_0(x_1) = \{ -1, 1 \} \quad \varphi_0(x_2) = \{ 1, 1 \} \]
\[ \varphi_0(x_3) = \{ -1, -1 \} \quad \varphi_0(x_4) = \{ 1, -1 \} \]
\[ \varphi_1(x_1) = \{ -1, -1 \} \quad \varphi_1(x_2) = \{ 1, 1 \} \]
\[ \varphi_1(x_3) = \{ -1, 1 \} \quad \varphi_1(x_4) = \{ -1, -1 \} \]
Example

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

\[ \phi_0(x_1) = \{-1, 1\} \quad \phi_0(x_2) = \{1, 1\} \]

\[ \phi_0(x_3) = \{-1, -1\} \quad \phi_0(x_4) = \{1, -1\} \]

\[ \phi_0(x) = \{-1, 1\} \quad \phi_0(y) = \{-1, -1\} \]

\[ \phi_1(x_1) = \{-1, 1\} \quad \phi_1(x_2) = \{1, -1\} \]

\[ \phi_1(x_3) = \{-1, 1\} \quad \phi_1(x_4) = \{-1, -1\} \]

\[ \phi_1(x) = \{-1, 1\} \quad \phi_1(y) = \{-1, -1\} \]

\[ \phi_2(x) = \{-1, 1\} \quad \phi_2(y) = \{-1, -1\} \]
Example

• The final net

\[ \varphi_0[0] \quad \varphi_0[1] \quad \varphi_1[0] \quad \varphi_1[1] \quad \varphi_2[0] \quad \varphi_2[1] \]

\[ \begin{align*}
\varphi_0[0] & 
\begin{array}{c}
\varphi_0[1] \\
1 \\
-1 \\
1
\end{array}
\end{align*} \]

\[ \begin{align*}
\varphi_0[0] & 
\begin{array}{c}
\varphi_1[0] \\
1 \\
-1 \\
1
\end{array}
\end{align*} \]

\[ \begin{align*}
\varphi_0[1] & 
\begin{array}{c}
\varphi_1[1] \\
1 \\
-1 \\
1
\end{array}
\end{align*} \]

\[ \begin{align*}
\varphi_1[0] & 
\begin{array}{c}
\varphi_2[0] \\
1 \\
1
\end{array}
\end{align*} \]

\[ \begin{align*}
\varphi_1[1] & 
\begin{array}{c}
\varphi_2[1] \\
1 \\
1
\end{array}
\end{align*} \]
Calculating a Net (with Vectors)

\[
\phi_0 = \text{np.array}( [1, -1] )
\]

Input

\[
\begin{align*}
\phi_0[0] &= 1 \\
\phi_0[1] &= -1 \\
\phi_0 &= \text{np.zeros}( 1 )
\end{align*}
\]

First Layer Output

\[
\begin{align*}
w_{0,0} &= \text{np.array}( [1, 1] ) \\
b_{0,0} &= \text{np.array}( [-1] ) \\
w_{0,1} &= \text{np.array}( [-1, -1] ) \\
b_{0,1} &= \text{np.array}( [-1] ) \\
\phi_1 &= \text{np.zeros}( 2 ) \\
\phi_1[0] &= \text{np.tanh}( w_{0,0} \phi_0 + b_{0,0} )[0] \\
\phi_1[1] &= \text{np.tanh}( w_{0,1} \phi_0 + b_{0,1} )[0]
\end{align*}
\]

Second Layer Output

\[
\begin{align*}
w_{1,0} &= \text{np.array}( [1, 1] ) \\
b_{1,0} &= \text{np.array}( [-1] ) \\
\phi_2 &= \text{np.zeros}( 1 ) \\
\phi_2[0] &= \text{np.tanh}( w_{1,0} \phi_1 + b_{1,0} )[0]
\end{align*}
\]
Calculating a Net (with Matrices)

Input

\[ \phi_0 = \text{np.array( [1, -1] )} \]

First Layer Output

\[ w_0 = \text{np.array( [[1, 1], [-1, -1]] )} \]
\[ b_0 = \text{np.array( [-1, -1] )} \]
\[ \phi_1 = \text{np.tanh( np.dot(} w_0, \phi_0 \text{)+ b_0 \text{)} } \]

Second Layer Output

\[ w_1 = \text{np.array( [[1, 1]] )} \]
\[ b_1 = \text{np.array( [-1] )} \]
\[ \phi_2 = \text{np.tanh( np.dot(} w_1, \phi_1 \text{)+ b_1 \text{)} } \]
Forward Propagation Code

```python
def 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**

\[
err = \frac{(y' - y)^2}{2}
\]

- **Gradient of the error:**

\[
err' = \delta = y' - y
\]

- **Update of weights:**

\[
w \leftarrow w + \lambda \cdot \delta \cdot \varphi(x)
\]

- \(\lambda\) is the learning rate
Problem: Don't know error for hidden layers!

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

```
φ “A” = 1
φ “site” = 1
φ “located” = 1
φ “Maizuru” = 1
φ “,” = 2
φ “in” = 1
φ “Kyoto” = 1
φ “priest” = 0
φ “black” = 0
```

```
\[ y' = ? \quad y = 1 \]
\[ y' = ? \quad y = 1 \]
\[ y' = ? \quad y = 1 \]
\[ y' = 1 \quad y = -1 \]
\[ y' = ? \quad y = 1 \]
```
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

\[ d \tanh(\varphi(x) \ast w) = 1 - \tanh(\varphi(x) \ast w)^2 = 1 - y_j^2 \]

- Together:

\[ \delta_j = (1 - y_j^2) \sum_i \delta_i w_{j,i} \]
Back Propagation

Error of the Output
\[ \delta_2 = np.array([y' - y]) \]

Error of the First Layer
\[
\delta'_2 = \delta_2 \times (1 - \varphi_2^2)
\]
\[ \delta_1 = np.dot(\delta'_2, w_1) \]

Error of the 0\textsuperscript{th} Layer
\[
\delta'_1 = \delta_1 \times (1 - \varphi_1^2)
\]
\[ \delta_0 = np.dot(\delta'_1, w_0) \]
Back Propagation Code

```python
def backward_nn(net, φ, y_prime):
    J = len(net)
    δ = [0, 0, ..., np.array([y_prime - φ[J][0]])]  # length J+1
    create array δ' = [0, 0, ..., 0]
    for i in range(J-1, -1, -1):
        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 $\delta'$ and prev $\varphi$
  
  $$-\text{derr}/d\,w_i = \text{np.outer}(\delta'_{i+1}, \varphi_i)$$

- Multiply by learning rate and update weights
  
  $$w_i += \lambda \times -\text{derr}/d\,w_i$$

- For the bias, input is 1, so simply $\delta'$
  
  $$-\text{derr}/d\,b_i = \delta'_{i+1}$$
  $$b_i += \lambda \times -\text{derr}/d\,b_i$$
Weight Update Code

```python
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]
```
# 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 / 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 every iteration

- **Better Way:** Use the finite differences method

  **Idea:**

  When updating weights, calculate grad. for $w_i$: $\frac{derr}{dw_i}$

  If we change that weight by a small amount ($\omega$):

  \[
  \begin{array}{c|c}
  w_i = x & w_i = x + \omega \\
  \downarrow & \downarrow \\
  \text{err} = y & \text{err} \approx y + \omega \times \frac{derr}{dw_i}
  \end{array}
  \]

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

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!