Neural networks

CMSC 723 / LING 723 / INST 725

MARINE CARPUAT
marine@cs.umd.edu

Slides credit: Graham Neubig
Outline

• Perceptron: recap and limitations
• Neural networks
  – Multi-layer perceptron
  – Forward propagation for prediction
  – Back propagation for learning weights
  – Multiclass prediction
Prediction Problems

Given $x$, predict $y$
Our prediction problem

Given an introductory sentence in Wikipedia predict whether the article is about a person

**Given**

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

**Predict**

Yes!

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

**Predict**

No!
Formalizing binary prediction

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

- **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”
  - (\( \operatorname{sign}(v) \) is +1 if \( v \geq 0 \), -1 otherwise)
Example feature functions: Unigram features

- Number of times a particular word appears

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

\[ \varphi_{\text{unigram "A"}}(x) = 1 \quad \varphi_{\text{unigram "site"}}(x) = 1 \quad \varphi_{\text{unigram ","}}(x) = 2 \]

\[ \varphi_{\text{unigram "located"}}(x) = 1 \quad \varphi_{\text{unigram "in"}}(x) = 1 \]

\[ \varphi_{\text{unigram "Maizuru"}}(x) = 1 \quad \varphi_{\text{unigram "Kyoto"}}(x) = 1 \]

\[ \varphi_{\text{unigram "the"}}(x) = 0 \quad \varphi_{\text{unigram "temple"}}(x) = 0 \]

\[ \ldots \]\[ \text{The rest are all 0} \]
Calculating the Weighted Sum

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

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

\[
\begin{align*}
w_{\text{unigram}} \text{“a”} &= 0 \\
w_{\text{unigram}} \text{“site”} &= -3 \\
w_{\text{unigram}} \text{“located”} &= 0 \\
w_{\text{unigram}} \text{“Maizuru”} &= 0 \\
w_{\text{unigram}} \text{“,”} &= 0 \\
w_{\text{unigram}} \text{“in”} &= 0 \\
w_{\text{unigram}} \text{“Kyoto”} &= 0 \\
w_{\text{unigram}} \text{“priest”} &= 2 \\
w_{\text{unigram}} \text{“black”} &= 0
\end{align*}
\]

\[
\begin{align*}
\cdots & \\
\times & \\
\cdots & = \\
\phantom{=} & -3
\end{align*}
\]
The Perceptron:
a “machine” to calculate a weighted sum

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

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

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

↓ numpy

```python
predict_one(w, phi)
    score = np.dot(w, phi)
    return (1 if score[0] >= 0 else -1)
```
The Perceptron: Geometric interpretation
The Perceptron: Geometric interpretation
Outline

• Perceptron: recap and limitations
• Neural networks
  – Multi-layer perceptron
  – Forward propagation for prediction
  – Back propagation for learning weights
  – Multiclass prediction
Exercise

Sentiment analysis for movie reviews
Limitation of perceptron

- can only find linear separations between positive and negative examples
Neural Networks

- Connect together multiple perceptrons

- Motivation: Can represent non-linear functions!
Neural Networks: key terms

- Input (aka features)
- Output
- Nodes
- Layers
- Activation function (non-linear)

- Multi-layer perceptron
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

\[
\begin{align*}
\varphi_0(x_1) &= \{-1, 1\} & \varphi_0(x_2) &= \{1, 1\} & \varphi_1(x_3) &= \{-1, 1\} \\
\varphi_0(x_3) &= \{-1, -1\} & \varphi_0(x_4) &= \{1, -1\} & \varphi_1(x_1) &= \{-1, -1\} & \varphi_1(x_2) &= \{1, -1\} \\
\varphi_1(x_4) &= \{-1, -1\} & \varphi_1[0] &= \{-1, -1\} & \varphi_1[0] &= \{-1, -1\} & \varphi_1[1] &= \{1, -1\} \\
\end{align*}
\]
Example

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

- The final net
Calculating a Net (with Vectors)

Input
\[ \varphi_0 = np.array([1, -1]) \]

First Layer Output
\[ w_{0,0} = np.array([1, 1]) \]
\[ b_{0,0} = np.array([-1]) \]
\[ w_{0,1} = np.array([-1, -1]) \]
\[ b_{0,1} = np.array([-1]) \]
\[ \varphi_1 = np.zeros(2) \]
\[ \varphi_1[0] = np.tanh(w_{0,0}\varphi_0 + b_{0,0})[0] \]
\[ \varphi_1[1] = np.tanh(w_{0,1}\varphi_0 + b_{0,1})[0] \]

Second Layer Output
\[ w_{1,0} = np.array([1, 1]) \]
\[ b_{1,0} = np.array([-1]) \]
\[ \varphi_2 = np.zeros(1) \]
\[ \varphi_2[0] = np.tanh(\varphi_1 w_{1,0} + b_{1,0})[0] \]
Calculating a Net (with Matrices)

**Input**
\[
\phi_0 = np.array([1, -1])
\]

**First Layer Output**
\[
w_0 = np.array([[1, 1], [-1, -1]])
\]
\[
b_0 = np.array([-1, -1])
\]
\[
\phi_1 = np.tanh(np.dot(w_0, \phi_0) + b_0)
\]

**Second Layer Output**
\[
w_1 = np.array([[1, 1]])
\]
\[
b_1 = np.array([-1])
\]
\[
\phi_2 = np.tanh(np.dot(w_1, \phi_1) + b_1)
\]
forward_propagation

```python
forward_nn(network, φ_0)

φ = [φ_0]  # Output of each layer
for each layer i in 1 .. len(network):
    w, b = network[i-1]
    # 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
```
Outline

• Perceptron: recap and limitations
• Neural networks
  – Multi-layer perceptron
  – Forward propagation for prediction
  – Back propagation for learning weights
  – Multiclass prediction
The perceptron: An online learning algorithm

```
create map w
for / 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)
```
Perceptron weight update

\[ w \leftarrow w + y \varphi(x) \]

- If \( y = 1 \), increase the weights for features in \( \varphi(x) \)
- If \( y = -1 \), decrease the weights for features in \( \varphi(x) \)
Stochastic gradient ascent (or descent)

- Online training algorithm for logistic regression
  - and other probabilistic models

```plaintext
create map w
for / iterations
    for each labeled pair x, y in the data
        w += α * dP(y|x)/dw
```

- Update weights for every training example
- Move in direction given by gradient
- Size of update step scaled by learning rate
Calculating Error with tanh

- Error (aka loss) function: Squared error
  \[ \text{err} = (y' - y)^2 / 2 \]
  
  Correct Answer: Net Output

- Gradient of the error:
  \[ \text{err}' = \delta = y' - y \]

- Update of weights:
  \[ \mathbf{w} \leftarrow \mathbf{w} + \lambda \cdot \delta \cdot \phi(x) \]
  
  \( \lambda \) is the learning rate
Problem: We don't know the error for hidden layers!

The NN only gets the correct label for the final layer

\[
\begin{align*}
\varphi_{\text{A}} &= 1 \\
\varphi_{\text{site}} &= 1 \\
\varphi_{\text{located}} &= 1 \\
\varphi_{\text{Maizuru}} &= 1 \\
\varphi_{\text{,}} &= 2 \\
\varphi_{\text{in}} &= 1 \\
\varphi_{\text{Kyoto}} &= 1 \\
\varphi_{\text{priest}} &= 0 \\
\varphi_{\text{black}} &= 0
\end{align*}
\]
Solution: Back Propagation

- Propagate the error backwards through the layers

\[ \delta_i w_{j,i} \]

- Also consider the gradient of the non-linear function

\[ dtanh(\phi(x) * w) = 1 - \tanh(\phi(x) * w)^2 = 1 - y_j^2 \]

- Together:

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

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

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

**Error of the 0th Layer**
\[ \delta'_1 = \delta_1 \times (1 - \varphi_1^2) \]
\[ \delta_0 = \text{np.dot}(\delta'_1, w_0) \]
```python
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 $\delta'$ and prev $\varphi$

\[-\frac{\text{derr}}{dw_i} = \text{np.outer}(\delta'_{i+1}, \varphi_i)\]

- Multiply by learning rate and update weights

\[w_i += \lambda \times \frac{-\text{derr}}{dw_i}\]

- For the bias, input is 1, so simply $\delta'$

\[\frac{-\text{derr}}{db_i} = \delta'_{i+1}\]

\[b_i += \lambda \times \frac{-\text{derr}}{db_i}\]
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]
```
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
Outline

• Perceptron: recap and limitations
• Neural networks
  – Multi-layer perceptron
  – Forward propagation for prediction
  – Back propagation for learning weights
  – Multiclass prediction
Review: Prediction Problems

• Given \( x \), predict \( y \)

<table>
<thead>
<tr>
<th>A book review</th>
<th>Is it positive?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oh, man I love this book!</td>
<td>yes</td>
</tr>
<tr>
<td>This book is so boring...</td>
<td>no</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A tweet</th>
<th>Its language</th>
</tr>
</thead>
<tbody>
<tr>
<td>On the way to the park!</td>
<td>English</td>
</tr>
<tr>
<td>公園に行くなう！</td>
<td>Japanese</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A sentence</th>
<th>Its syntactic parse</th>
</tr>
</thead>
<tbody>
<tr>
<td>I read a book</td>
<td></td>
</tr>
<tr>
<td></td>
<td>S</td>
</tr>
<tr>
<td></td>
<td>VP</td>
</tr>
<tr>
<td></td>
<td>NP</td>
</tr>
<tr>
<td></td>
<td>VBD</td>
</tr>
<tr>
<td></td>
<td>DET</td>
</tr>
<tr>
<td></td>
<td>N</td>
</tr>
<tr>
<td>I read a book</td>
<td></td>
</tr>
</tbody>
</table>

- Binary Prediction (2 choices)
- Multi-class Prediction (several choices)
- Structured Prediction (millions of choices)
Multiclass classification with neural networks
Sigmoid Function

- The sigmoid softens the step function

\[ P(y = 1 \mid x) = \frac{e^{w \cdot \phi(x)}}{1 + e^{w \cdot \phi(x)}} \]
Softmax Function for multiclass classification

- Sigmoid function for multiple classes

\[ P(y | x) = \frac{e^{w \cdot \phi(x, y)}}{\sum_{\tilde{y}} e^{w \cdot \phi(x, \tilde{y})}} \]

- Can be expressed using matrix/vector ops

\[ \mathbf{r} = \exp(\mathbf{W} \cdot \phi(x)) \]

\[ \mathbf{p} = \mathbf{r} / \sum_{\tilde{r} \in \mathbf{r}} \tilde{r} \]
What we’ve learned today

• Perceptron: recap and limitations
• Neural networks
  – Multi-layer perceptron
  – Forward propagation for prediction
    • Implemented as matrix operations
  – Back propagation for learning weights
    • Gradient descent + chain rule
  – Multiclass prediction
Coming next

• Recurrent neural networks
• Applications to NLP

• Readings
  • Bengio et al. 2005