Prof. Dr. Andreas Hotho, Albin Zehe, Jonas Kaiser
Data Science Chair

08.05.2024

2. Assignment in "Machine Learning for Natural Language Processing"

Summer Term 2024

1 General Questions

1. Name three common activation functions and their derivatives!

   - Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$, $f'(z) = f(z) \cdot (1 - f(z))$
   - Hyperbolic Tangent: $f(z) = tanh(z)$, $f'(z) = 1 - tanh^2(z)$
   - Rectified Linear Unit (ReLU): $f(z) = max(0, z)$,

   $$f'(z) = \begin{cases} 0, z \leq 0 \\ 1, z > 0 \end{cases}$$

2. Given two word embeddings E and E' for a corpus of words, how could you rate the quality of the embeddings relative to one another? Is there a possibility to give an absolute, global rating for word embeddings?

   Word embeddings can not be rated by themselves, but must be evaluated by their performance when used to solve some task. There are several commonly used tasks used to rate word embeddings:

   - Human Intuition Datasets. These are lists of pairs of words rated by human annotators regarding how similar their meaning is. One example of this is the WS-353 dataset (`http://alfonseca.org/eng/research/wordsim353.html`). Word embeddings are rated by their ability to predict the similarity scores assigned by the annotators. More detail on this soon in the lecture!
   - Evaluation by some classifier that uses word embeddings to represent its input. One task commonly used for this is sentiment analysis. Here, the

input sentences are often represented by concatenating the embeddings of their words. If embedding E provides a "better" representation than E', then the same classifier will perform better using E rather than E'.

Note that this evaluation is very specific to the task! There may be a task where E clearly outperforms E' and another where E' clearly outperforms E. For example, in sentiment analysis, the words *good* and *bad* are clearly on the opposite ends of the "meaning-scale", while for POS-tagging they are basically equivalent.

? Something to think about

3. Suggest a combination of GloVe and FastText!

You could do a modification of GloVe similar to what FastText changes from Word2Vec, that is, instead of optimising

$$w_i^T \tilde{w}_k = \log(X_{ik}) - \log(X_i),$$

optimise

$$\sum_{g \in \mathcal{G}} g_i^T \tilde{w}_k = \log(X_{ik}) - \log(X_i)$$

for the character n-grams $\mathcal{G}$ contained in w_i

2 Neural Networks

Bias Trick

Formally, a feed-forward layer in a neural network is defined by a number of parameters consisting of a weight matrix W and a bias vector b. The output y that is passed to the activation function is then calculated as

$$y = Wx + b.$$

In practice, we want to keep the number of vectorised operations as small as possible, to enable maximum training/prediction speed (remember that GPUs can parallelise the matrix operations). Therefore, these two parameter sets are often combined into one

matrix

$$W' = W \| b = \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b_1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b_2 \\ \vdots & & \ddots & & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nm} & b_n \end{pmatrix}$$

1. How does the input x need to be modified to enable computing the output y in a single operation?

The input needs to be modified by appending a 1 to it, i.e. $x' = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \\ 1 \end{pmatrix}$

2. How is the output y calculated using the modified W' and input x'?

$$y = Wx + b = W' \cdot x'$$

3. Proof that this always yields the same result as $Wx + b$!

$$
\begin{aligned}
Wx + b &= \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1m} \\ w_{21} & w_{22} & \cdots & w_{2m} \\ \vdots & & \ddots & \\ w_{n1} & w_{n2} & \cdots & w_{nm} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\
&= \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1m}x_m \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2m}x_m \\ \vdots \\ w_{n1}x_1 + w_{n2}x_2 + \cdots + w_{nm}x_m \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix} \\
&= \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1m}x_m + b_1 \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2m}x_m + b_2 \\ \vdots \\ w_{n1}x_1 + w_{n2}x_2 + \cdots + w_{nm}x_m + b_n \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
W'x' &= \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1m} & b_1 \\ w_{21} & w_{22} & \cdots & w_{2m} & b_2 \\ \vdots & & \ddots & & \\ w_{n1} & w_{n2} & \cdots & w_{nm} & b_n \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} w_{11}x_1 + w_{12}x_2 + \cdots + w_{1m}x_m + 1b_1 \\ w_{21}x_1 + w_{22}x_2 + \cdots + w_{2m}x_m + 1b_2 \\ \vdots \\ w_{n1}x_1 + w_{n2}x_2 + \cdots + w_{nm}x_m + 1b_n \end{pmatrix}
\end{aligned}
$$

$$
Wx + b = W'x'
$$

3 Python

In the lecture, some commonly used optimisers for neural networks were introduced. Implement functions for

1. `sgd_update`
2. `nesterov_momentum_update`
3. `adam_update`

in Python. Each function should

- take as input
  - the current value of *one* input parameter,
  - a function returning the gradient regarding this parameter, and
  - the necessary hyper-parameters such as the learning rate, and
- return the new value for this parameter after the update.

You do *not* have to implement backpropagation in this assignment!

> The jupyter notebook "Optimisers" contains implementations for the functions as well as the animations shown in the lecture.

```python
import numpy as np

def sgd_update(x, dx, learning_rate):
    x -= learning_rate * dx(x)
    return x

v = 0
def nesterov_momentum_update(x, dx, learning_rate, mu):
    global v

    x_ahead = x + mu * v

    v = mu * v - learning_rate * dx(x_ahead)
    x += v
    return x

m = 0
v = 0
```

```
def adam_update(x, dx, learning_rate, beta1, beta2):
    global m, v

    m = beta1 * m + (1 - beta1) * dx(x)
    v = beta2 * v + (1 - beta2) * (dx(x) ** 2)

    x += -learning_rate * m / (np.sqrt(v) + 1e-8)

    return x
```