Assignment 2: Memory task

LSTM and Recurrent Nets UE, 04.11.2019
Frederik Kratzert

JOHANNES KEPLER
UNIVERSITY LINZ

Overview

- Assignment 1 Discussion
- Assignment 2

Assignment 1 Discussion

Gradient of bias terms

$$\boldsymbol{s}[t] = \boldsymbol{W}^T \boldsymbol{x}[t] + \boldsymbol{R}^T \boldsymbol{a}[t-1] + \boldsymbol{b}_s$$

$$\boldsymbol{a}[t] = \tanh(\boldsymbol{s}[t])$$

$$\widehat{\boldsymbol{y}} = \boldsymbol{V}^T \boldsymbol{a}[t = T] + \boldsymbol{b}_y$$

$$\frac{\partial L}{\partial \boldsymbol{b}_s[t]} = \dots$$

$$\frac{\partial L}{\partial \boldsymbol{b}_y[t]} = \dots$$

Weight initialization

```python
def __init__(self, input_size: int, hidden_size: int, output_size: int):
    """Initialization

    Parameters
    ----------
    input_size : int
        Number of input features per time step
    hidden_size : int
        Number of hidden units in the RNN
    output_size : int
        Number of output units.
    """
    super(RNN, self).__init__()
    self.input_size = input_size
    self.hidden_size = hidden_size
    self.output_size = output_size

    # create and initialize weights of the network
    self.W = np.random.uniform(-0.2, 0.2, (input_size, hidden_size))
    self.R = np.random.uniform(-0.2, 0.2, (hidden_size, hidden_size))
    self.bs = np.zeros((hidden_size, 1))
    self.V = np.random.uniform(-0.2, 0.2, (hidden_size, output_size))
    self.by = np.zeros((output_size, 1))
```

Forward pass

```python
def forward(self, x: np.ndarray) -> np.ndarray:
    """Forward pass through the RNN.

    Parameters
    ----------
    x : np.ndarray
        Input sequence(s) of shape [sequence length, number of features]

    Returns
    -------
    NumPy array containing the network prediction for the input sample.
    """
    seq_length, _ = x.shape

    self.a = np.zeros((seq_length+1, self.hidden_size))

    a_0 = np.zeros((self.hidden_size))
    self.a[-1] = a_0.copy()

    for t in range(seq_length):
        s_t = np.dot(x[t], self.W) + np.dot(self.a[t-1], self.R) + self.bs.T
        a_t = np.tanh(s_t)
        self.a[t] = a_t

    y_hat = np.dot(a_t, self.V) + self.by.T
    self.y_hat = y_hat.copy()
    self.x = x.copy()

    return y_hat
```

Store activations for backward pass

$$\boldsymbol{s}[t] = \boldsymbol{W}^T \boldsymbol{x}[t] + \boldsymbol{R}^T \boldsymbol{a}[t-1] + \boldsymbol{b}_s$$

$$\boldsymbol{a}[t] = \tanh(\boldsymbol{s}[t])$$

$$\widehat{\boldsymbol{y}} = \boldsymbol{V}^T \boldsymbol{a}[t = T] + \boldsymbol{b}_y$$

Backward pass

```
def backward(self, d_loss: np.ndarray) -> Dict:
    """Calculate the backward pass through the RNN.

    Parameters
    ----------
    d_loss : np.ndarray
        The gradient of the loss w.r.t the network output in the shape [output_size,]

    Returns
    -------
    Dictionary containing the gradients for each network weight as key-value pair.
    """
    seq_length, _ = self.x.shape

    # track gradients of weight matrices
    d_V = np.zeros_like(self.V)
    d_by = np.zeros_like(self.by)
    d_R = np.zeros((seq_length, *self.R.shape))
    d_W = np.zeros((seq_length, *self.W.shape))
    d_bs = np.zeros((seq_length, *self.bs.shape))
```

Backward pass

```python
# track gradients of weight matrices
d_V = np.zeros_like(self.V)
d_by = np.zeros_like(self.by)
d_R = np.zeros((seq_length, *self.R.shape))
d_W = np.zeros((seq_length, *self.W.shape))
d_bs = np.zeros((seq_length, *self.bs.shape))

for t in reversed(range(seq_length)):
    if t == seq_length - 1:

        d_V = np.dot(self.a[t].reshape(-1,1), d_loss.reshape(1, -1))
        d_a = np.dot(d_loss.reshape(1,-1), self.V.T)
        d_by = d_loss.reshape(self.by.shape)
    else:
        d_a = d_a_next

    d_s = d_a * (1 - self.a[t]*self.a[t])
    d_W[t] = np.dot(self.x[t].reshape(-1,1), d_s)
    d_R[t] = np.dot(self.a[t-1].reshape(-1,1), d_s)
    d_bs[t] = d_s.reshape(d_bs[t].shape)

    d_a_next = np.dot(d_s, self.R.T)

self.grads = {'d_V': d_V, 'd_W': d_W, 'd_R': d_R, 'd_bs': d_bs, 'd_by': d_by}

return self.grads
```

$$\boldsymbol{\delta}(t)^\top = \frac{\partial L}{\partial \boldsymbol{s}(t)}$$

Parameter update

```python
def update(self, lr: float):
    """Update the network parameter.

    Parameters
    ----------
    lr : float
        Learning rate used for the weight update
    """
    if not self.grads:
        raise RuntimeError("You have to call the .backward() function first")

    for key, grad in self.grads.items():
        if len(grad.shape) == 3:
            self.grads[key] = grad.sum(axis=0)

    self.W -= lr*self.grads['d_W']
    self.R -= lr*self.grads['d_R']
    self.V -= lr*self.grads['d_V']
    self.bs -= lr*self.grads['d_bs']
    self.by -= lr*self.grads['d_by']

    # reset internal class attributes
    self.grads = {}
    self.y_hat, self.a = None, None
```

Numerical Gradient

$$\frac{\partial f}{\partial w_i} \approx \frac{f(x, w_1, ...w_i + \epsilon, ...w_n) - f(x, w_1, ...w_i - \epsilon, ...w_n)}{2 * \epsilon}$$

Numerical Gradient

```
weights = model.get_weights()
numerical_gradients = {key: np.zeros_like(val) for key, val in weights.items()}

for name, weight in weights.items():
    new_weights = {key: val for key, val in weights.items() if key != name}
    for i in range(weight.size):
        new_weight = weight.copy()
        m, n = np.unravel_index(i, weight.shape)

        # upper approximation
        new_weight = weight.copy()
        new_weight[m,n] += eps
        new_weights[name] = new_weight
        model.set_weights(new_weights)
        y_hat_upper = model.forward(x)

        # lower approximation
        new_weight = weight.copy()
        new_weight[m,n] -= eps
        new_weights[name] = new_weight
        model.set_weights(new_weights)
        y_hat_lower = model.forward(x)

        # calculate gradient approximation
        numerical_gradients[name][m,n] = np.sum((y_hat_upper - y_hat_lower) / (2*eps))

numerical_gradients = {f"d_{key}": val for key, val in numerical_gradients.items()}

return numerical_gradients
```

Analytical Gradient

```python
def get_analytical_gradient(model: RNN, x: np.ndarray) -> Dict:
    """Helper function to get the analytical gradient.

    Note: In contrast to the RNN update function, use the sum of the recurrent gradients
    over time (and average over batch samples)

    Parameters
    ----------
    model : RNN
        The RNN model object
    x : np.ndarray
        Input sequence(s) of shape [sequence length, number of features]

    Returns
    -------
    A dictionary containing the analytical gradients for each weight of the RNN.
    """
    _ = model.forward(x)
    analytical_grads = model.backward(np.ones((model.output_size)))
    for key, grad in analytical_grads.items():
        if len(grad.shape) == 3:
            analytical_grads[key] = grad.sum(axis=0)

    return analytical_grads
```

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \widehat{u}} \frac{\partial \widehat{y}}{\partial w}$$

$$\longrightarrow \frac{\partial L}{\partial \widehat{y}} = 1$$

Assignment 2

Memory task

0.2	-0.3	0.1	0.7	-0.4	-0.6	0.5
x_1	x_2	x_3	x_4	x_5	x_6	x_7

$\longrightarrow$ $y = 0.2$

Tasks

- Implement data generator
- Implement MSE loss (forward + backward)
- Implement learner class
- Train RNN for different sequence lengths
- Visualize vanishing gradients

→ RNN implementation is provided

Data Generator

```
def generate_samples(batch_size: int, seq_length: int) -> Tuple[np.ndarray, np.ndarray]:
    """Data generator for memory task

    Note: Implement this function as a Python generator.

    Parameters
    ----------
    batch_size : int
        Number of samples in one batch
    seq_length : int
        Length of sequence of random numbers

    Returns
    -------
    x : np.ndarray
        Array of shape [sequence length, batch size, 1], where each sample is a sequence
        of random generated numbers between -1 and 1.
    y : np.ndarray
        Array of shape [batch size, 1], where each element i contains the label corresponding
        to sample i of the input array. The label is the first element of the sequence.

    """
    ########################
    # Your code comes here #
    ########################
```

MSE Loss

- Forward pass: $$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \widehat{y}_i)^2$$
- Backward pass: $\longrightarrow$ $$\frac{\partial L}{\partial \widehat{y}}$$

$\longrightarrow$ Check (backward pass) implementation using numerical gradient check, again. Hint: This is a parameter free function, so you apply the variations directly to the model output.

Learner Class

- Used to facilitate training
  - Tracks loss
  - Trains model
  - Makes prediction
  - Plots results

→ Use provided code skeleton and add missing code

Model Training

- What is the maximum sequence length the RNN can solve.
- For each length, train multiple repetitions.
- Plot model accuracy (with error bars) for a number of (new) random batches.

Vanishing Gradient

- Be creative and find a way to visualize the problem of vanishing gradients.
  - E.g. look at gradients for different sequence lengths