Deep Learning in NLP

Homework P6: Solution to Ex. 1 explained

Exercise 1

The goal of the first exercise is to write a batching-enabled version of a feed-forward network with a single hidden layer (we call it FFN in the rest of the document). We will implement it in the `fnn.py` module.

Regular FFN (reminder)

A linear layer L with input size n and output size m has two parameters:

- $\mathbf{M} \in \mathbb{R}^{m \times n}$: matrix corresponding to the linear transformation
- $\mathbf{b} \in \mathbb{R}^m$: the corresponding bias vector

Given the input vector $\mathbf{x} \in \mathbb{R}^n$, the forward calculation formula is:

$$L(\mathbf{x}) = \mathbf{M}\mathbf{x} + \mathbf{b} \tag{1}$$

FFN F can be defined as a combination of two linear transformation layers L_1, L_2, with a non-linear element-wise activation function σ applied in between. Formally:

$$\begin{aligned} F(\mathbf{x}) &= L_2(\mathbf{h}), \text{ where} \\ \mathbf{h} &= \sigma(L_1(\mathbf{x})) \end{aligned} \tag{2}$$

Alternatively, using function composition:

$$F(\mathbf{x}) = (L_2 \circ \sigma \circ L_1)(\mathbf{x}) \tag{3}$$

Batching-enabled FFN

The goal of the first exercise is to implement a batching-enabled FFN. Our point of departure is Eq. 3 (or the equivalent Eq. 2). To make it work in batches, we basically need to adapt a linear layer to work in batches, too.

Let b be the batch size and $\mathbf{X}$ the matrix with input vectors, one vector per row. Hence, $\mathbf{X}_{[1]}$ is the first input vector, $\mathbf{X}_{[2]}$ the second input vector, etc.[1]

[1] Don't forget that, in comp. sci., indexing starts with 0...

Let L be a linear layer. Its batching-enabled version L^* should satisfy the following equation:

$$\forall_{i=1}^{b} L(\mathbf{X}_{[i]}) = L^*(\mathbf{X})_{[i]} \tag{4}$$

Given a FFN F, the formula for a batching-enabled FFN F^* is then:

$$F^*(\mathbf{X}) = (L_2^* \circ \sigma \circ L_1^*)(\mathbf{X}) \tag{5}$$

You can check that, provided that Eq. 4 holds, the following equation is also satisfied. In the code, this is checked using doctests.

$$\forall_{i=1}^{b} F^*(\mathbf{X})_{[i]} = F(\mathbf{X}_{[i]}) \tag{6}$$

Your job is to implement a batching-enabled linear layer in the `ffn.py` module. A batching-enabled FFN is already implemented. The missing parts are marked with `TODO`s.

Hints:

- Try replacing the matrix-vector product (`torch.mv`) with the matrix-matrix product (`torch.mm`). Be careful with the order of the arguments!
- Try implementing a linear layer **without** the bias vector first. Once it works, you can include the bias vector by adding it iteratively to each row in the resulting batch matrix. Is this solution optimal?

1b (optional)

Once you implement the batching-enabled linear layer in the `ffn.py` module, shortly explain how your code works and try to briefly show that it satisfies Eq. 4.

Solution

The initialization method:

```
def __init__(self, idim: int, odim: int):
    # Pay attention to the order of dimensions (idim, odim).  This is
    # related to how we multiply the matrix M by the input vector
    # (see the forward method).
    self.register("M", torch.randn(idim, odim))
    # We use ‘torch.randn‘ for simplicity, the Linear class in PyTorch
    # uses a different initialization method (see the docs).
    self.register("b", torch.randn(odim))
```

The forward method:

```
def forward(self, X: TT):
    # Check if the sizes of the row vectors in X match
    # the transformation layer input size.
    assert X.shape[1] == self.isize()
    # Note that ‘self.M‘ is the second argument of ‘torch.mm‘.
```

```
    # This is because the input vectors are stored in the rows
    # of the X matrix.
    return torch.mm(X, self.M) + self.b
```

A somewhat easier to understand alternative would be to stack `self.b` on top of each other to create a bias matrix `B` and add it to the result. In the code above, the last line works somewhat magically by adding the bias vector to each row in the result.

We now show that this code (without the bias vector) satisfies Eq. 4.

Proposition 1. *Let $\mathbf{M} \in \mathbb{R}^{m \times n}$ and $\mathbf{X} \in \mathbb{R}^{n \times b}$, where b is the batch size. In particular, $\mathbf{X}$ is a matrix of input vectors,* ***one vector per column****. Then, for each i: $0 < i \leq b$:*

$$(\mathbf{MX})_{[,i]} = \mathbf{M}(\mathbf{X}_{[,i]}) \tag{7}$$

where $\mathbf{X}_{[,i]}$ is the i-the column of matrix $\mathbf{X}$.

Proposition 1 motivates the first solution we have designed, where the input matrix $\mathbf{X}$ is first transposed in order to make the input vectors organized in columns:

```
def forward(self, X: TT):
    # Transpose the input matrix
    X = X.t()
    # Perform the linear transformation
    Y = torch.mm(self.M, X)
    # Return the output matrix, after transposition
    return Y.t()
```

At the end, the resulting output matrix must be also transposed, because (before transposition) it contains output vectors in columns.

Proposition 2. *Let $\mathbf{M} \in \mathbb{R}^{m \times n}$ and $\mathbf{X} \in \mathbb{R}^{n \times b}$, where b is the batch size. Then:*

$$(\mathbf{MX})^T = \mathbf{X}^T \mathbf{M}^T \tag{8}$$

where $\mathbf{X}^T$ is the transpotision of $\mathbf{X}$ (the first column is swapped with the first row, the second column with the second row, etc.).

Proposition 2 motivates the final solution, in which

- $\mathbf{X}^T$ is the input matrix, one input vector per row.
- $\mathbf{M}^T$ is the (transposed) linear transformation matrix.

In the code, we don't transpose $\mathbf{X}$, since the input matrix already contains input vectors in rows. We don't transpose $\mathbf{M}$ either, and keep the transposed matrix $\mathbf{M}^T$ as the model's parameter instead.