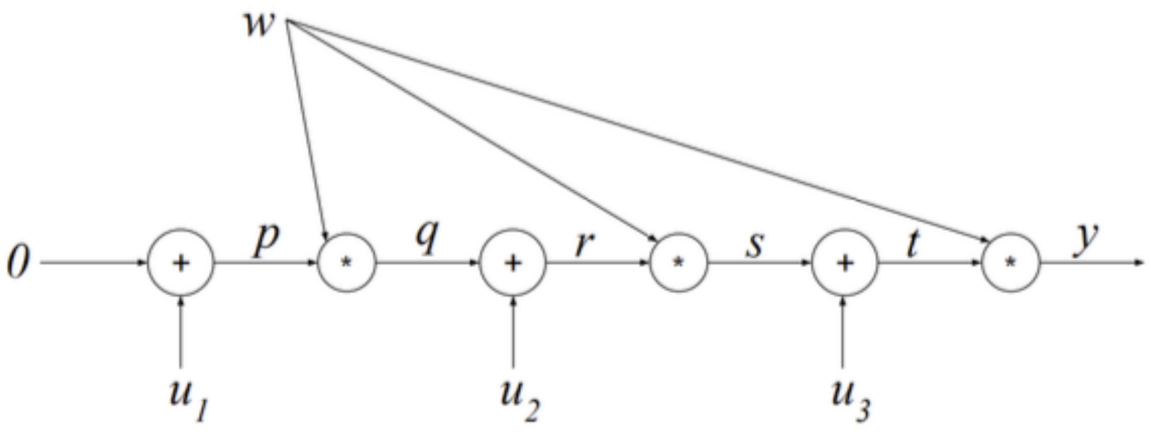
Hw12

周亦涵 2020012853 未央-水木01

Backprop Through a Simple RNN

Here we have $h_0 = 0$. According to the computational graph, we can express p, q, r, s and t one by one and finally express y with our inputs $u_1,\ u_2,\ u_3$ and the parameter w.



The detainled computational process is as following:

$$
\begin{align}
p &= h_0 + u_1 = 0 + u_1 = u_1 \tag{1}\\
q &= wp = wu_1 \tag{2}\\
r &= q + u_2 = wu_1 + u_2 \tag{3}\\
s &= wr = w^2u_1 + wu_2 \tag{4}\\
t &= s + u_3 = w^2u_1 + wu_2 + u_3 \tag{5}\\
y &= wt = w^3u_1 + w^2u_2 + wu_3 \tag{6}
\end{align}
$$

So expressions for t and y:

$$
\begin{align}
t &= w^2u_1 + wu_2 + u_3 \tag{7}\\
y &= w^3u_1 + w^2u_2 + wu_3 \tag{8}
\end{align}
$$

Compute $\frac{dy}{dw}$ and $\frac{\partial y}{\partial p}$:

$$
\begin{align}
\frac{dy}{dw} &= 3u_1w^2 + 2u_2w + u_3 \tag{9}\\
\frac{\partial y}{\partial p} &= w^3 \tag{10}
\end{align}
$$

Also we can use the back propagation method to solve the problem:

$$\frac{dy}{dw} = \frac{dy}{dw}|_y + \frac{dy}{dw}|_s + \frac{dy}{dw}|_q = t + wr + w^2p = 3u_1w^2 + 2u_2w + u_3 \tag{11}$$
$$\frac{\partial y}{\partial p} = \frac{\partial y}{\partial t} \cdot \frac{\partial t}{\partial s} \cdot \frac{\partial s}{\partial r} \cdot \frac{\partial r}{\partial q} \cdot \frac{\partial q}{\partial p} = w \cdot 1 \cdot w \cdot 1 \cdot w = w^3 \tag{12}$$

LSTM Gradients

Note that if $y = a \circ x$, we have $\frac{dy}{dx} = diag(a); d\tanh x/dx = 1 - \tanh^2 x$.

$$\because h_{T-1} = tanh(C_{T-1}) \circ o_T \tag{13}$$
$$\therefore \frac{\partial h_{T-1}}{\partial C_{T-1}} = diag(o_T)(1 - \tanh^2 C_{T-1}) \tag{14}$$
$$\because C_T = f_T \circ C_{T-1} + i_T \circ \tilde{C}_t \tag{15}$$
$$\therefore \frac{\partial C_T}{\partial C_{T-1}} = diag(f_T) \tag{16}$$

Now let's consider computing $\frac{\partial L}{\partial C_{T-1}}$:

$$\frac{\partial L}{\partial C_{T-1}} = \frac{\partial L}{\partial h_{T-1}} \frac{\partial h_{T-1}}{\partial C_{T-1}} + \frac{\partial L}{\partial C_T} \frac{\partial C_T}{\partial C_{T-1}} \tag{17}$$
$$= \frac{\partial L}{\partial h_{T-1}} diag(o_T)(1 - \tanh^2 C_{T-1}) + \frac{\partial L}{\partial C_T} diag(f_T) \tag{18}$$

Here the output gate $o^{(t)} = \sigma(W_o h^{(t-1)} + U_o x^{(t)} + b_o)$ is an elementwise multiplication operation and it's less likely to vanish compared to the gradients in traditional RNNs.

We can see that the formula uses $+$ to compute the gradients from a cell to its forward cell, instead of $\times$. While in RNN, $\frac{\partial L}{\partial h_{T-1}} = \frac{\partial L}{\partial h_T} \frac{\partial h_T}{\partial h_{T-1}}$, we use $\times$ to propagate the gradients. So in LSTM, the gradient won't be changed to a gread extent, it should vanish less quickly than RNN, and we can have a longer short-term memory.

Play with Transformer

The code filled is as following:

```
def __init__(self, nvoc, dim=256, nhead=8, num_layers = 4):
    super(LMModel_transformer, self).__init__()
    self.drop = nn.Dropout(0.5)
    self.encoder = nn.Embedding(nvoc, dim)
    # WRITE CODE HERE witnin two '#' bar
    ########################################
    # Construct you Transformer model here. You can add additional
parameters to the function.
```

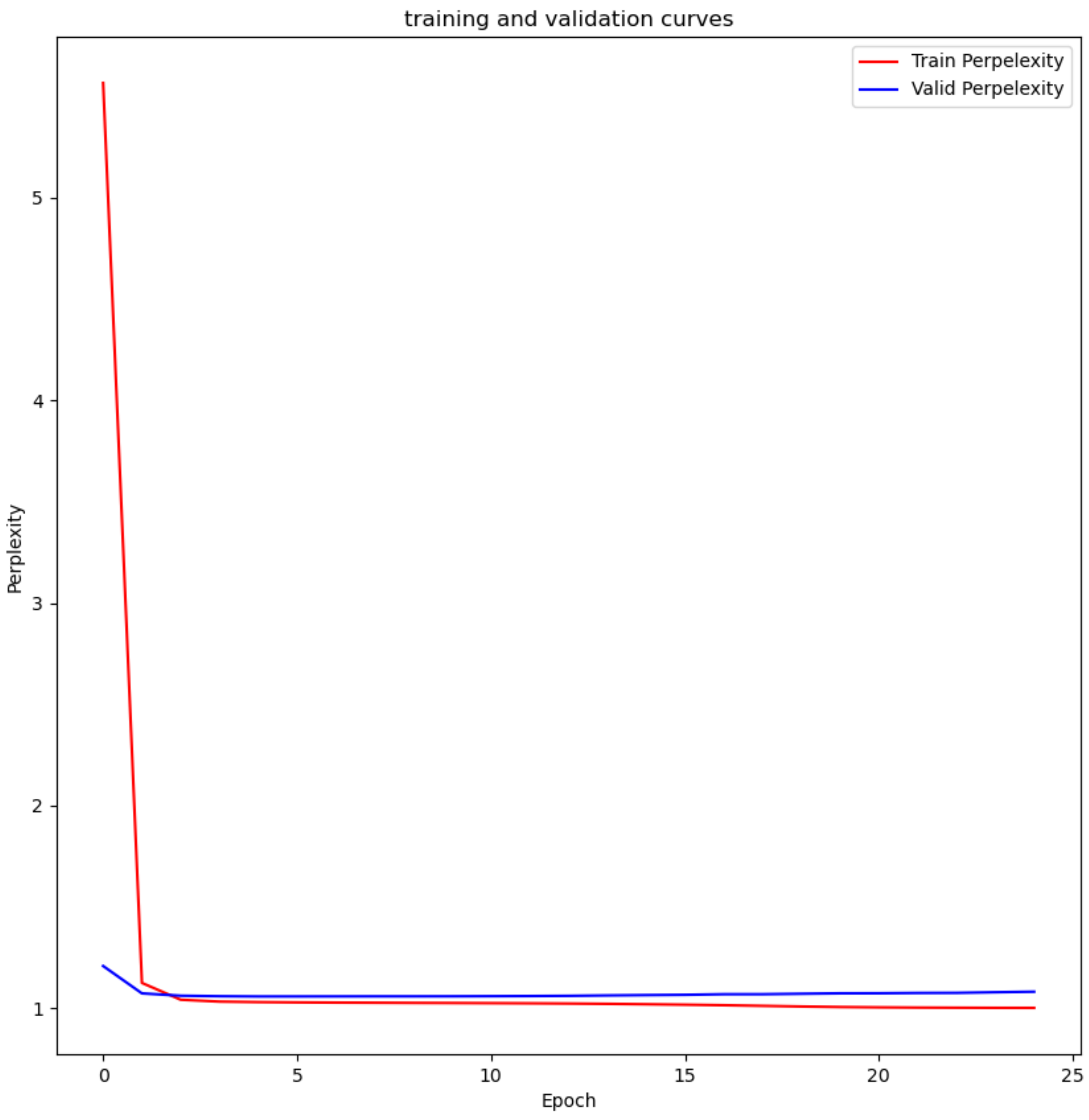
```
        self.dim = dim
        self.transformer = nn.Transformer(d_model=dim,
    nhead=nhead,num_encoder_layers=num_layers,num_decoder_layers=num_layers)
        ########################################
        self.decoder = nn.Linear(dim, nvoc)
        self.init_weights()

    def forward(self, input):
        #print(input.device)
        embeddings = self.drop(self.encoder(input))

        # WRITE CODE HERE within two '#' bar
        ########################################
        # With embeddings, you can get your output here.
        # Output has the dimension of sequence_length * batch_size * number of
    classes
        L = embeddings.size(0)
        src_mask = torch.triu(torch.ones(L, L) * float('-inf'),
    diagonal=1).to(input.device.type)
        src = embeddings * math.sqrt(self.dim)
        #TODO: use your defined transformer, and use the mask.
        target = input[1:,:]
        target_with_zeros =
    torch.cat([target,torch.zeros(1,target.size(1),dtype=target.dtype,device=inp
    ut.device)],dim=0)
        tgt = self.encoder(target_with_zeros) * math.sqrt(self.dim)
        output = self.transformer(src,tgt,src_mask=src_mask)
        ########################################
        output = self.drop(output)
        decoded = self.decoder(output.view(output.size(0)*output.size(1),
    output.size(2)))
        return decoded.view(output.size(0), output.size(1), decoded.size(1))
```

The training and validation curves:

Let's visualize the results into curves:



We need a source mask becaouse some meaningless part can generate attention too, which we don't want; while calculating the attention, it can sometimes depend on future information since we use all of the outcomes to perform embedding, so we should mask them in case the future information is used ahead of time. By setting the masked place to be `-inf`, after softmax function, it approximately equals to 0 and won't affect q.

Questions

> *Summarize the differences between preprocessing text data for language modeling with preprocessing image data for image recognition.*

Specifically, while preprocessing text data for language modeling in the code, we first extract the vocabulary line by line, iterate through all the words in the lines and assign a unique ID to each word using a dictionary ("self.word_id") Then we tokenize the text data by converting each word into its corresponding ID from the vocabulary. And we generate the data into batches, reshape them and truncate them. Finally, we use `get_batch` to retrieve mini-batches.

The difference can be summarized into the following three catogories:

- **Data representation:**

  Text is typically represented as a sequence of words or characters, we should tokenize the text by breaking it into individual units and create numerical representations such as word embeddings.

  Images are represented as a grid of pixels, we should resize the images to a standardized resolution, convert them to a suitable color space and normalize the pixel values to a specific range.
- **Preprocessing techniques:** While preprocessing texts, we often truncate them, or we remove punctuation or handle special characters. While preprocessing image data, we often apply transformations like rotation, scaling or flipping to increase the size and diversity of the training set.
- **Feature Extraction:** We use Word2Vec, RNN/LSTM or Transformer models to extract features of text data; while we use CNN to extract features from image data. The basic methods behind them are different.