The goal of this short note is to quickly put all the pieces of the course so far together into a single full description of building an NLP system to perform the task of machine translation.

Goal: Learn function f : source text $\rightarrow$ target text.

Data

Our dataset is a set of M pairs (aka examples) of strings,

$$\mathcal{D} = \left\{ \left(x^{(i)}, y^{(i)}\right) \right\}_{i=1}^{M}$$

where $x^{(i)}$ refers to the x string (the first element in the pair) in the i-th example. In our case, string x is a French sentence (or sequence of text), and string y is an English one. We call these the *source* and *target* text respectively. To give a concrete example for $M = 2$:

$$\mathcal{D} = \big\{(\text{“Bonjour”}, \text{“Hello”}), (\text{“J'aime les films”}, \text{“I like movies”})\big\}.$$

Tokenization

We learn our vocabulary by performing Byte-Pair Encoding on our dataset:

$$\mathcal{V} = \text{BPE}(\mathcal{D}).$$

The simple way to do this is to concatenate both strings in every example together into a huge chunk of text, then feed to BPE (this is what was done in A2). Another way is to learn a separate vocabulary for the source and target texts.

Now that we have our vocabulary (and hence have learned a tokenizer), we can tokenize each example in $\mathcal{D}$ such that for all $i \in [1, M]$:

$$x^{(i)} \leftarrow \text{tokenize}(x^{(i)})$$
$$y^{(i)} \leftarrow \text{tokenize}(y^{(i)})$$

meaning that

$$x^{(i)} = x_1^{(i)}, \ldots, x_t^{(i)} \qquad x_t^{(i)} \in \mathcal{V}$$
$$y^{(i)} = y_1^{(i)}, \ldots, y_{t'}^{(i)} \qquad y_{t'}^{(i)} \in \mathcal{V}.$$

Note that we can't directly use this for language modeling yet - recall that our goal is to learn $P(y^{(i)} \mid x^{(i)})$, but the language models we've learned so far can only take in a single sequence as input[1]. The trick is to concatenate our source and target strings demarked by special tokens `<START>` and `<END>`[2]:

$$s^{(i)} = x_1^{(i)}, \ldots, x_t^{(i)}, \texttt{<START>}, y_1^{(i)}, \ldots, y_{t'}^{(i)}, \texttt{<END>}.$$

The reason we have the `<START>` token is so that the model knows exactly when to start predicting English - without it, the model would need to also learn to predict when the French text ends, which not only wastes model capacity but is also pretty tough to do. We also append an `<END>` token to each s so that the model explicitly learns when to stop generating.

[1] Unlike traditional machine learning, we can't only have x as input and y as the label because we need to learn our target *autoregressively*, i.e. $P(y_{2:t'} \mid x, y_1)$, then $P(y_{3:t'} \mid x, y_1, y_2)$, and so on.

[2] We add these tokens as unique integers to $\mathcal{V}$.

Model

Feed-forward layers

Stacking layers

Let's adapt our simple language model setup from previous lectures to machine translation. To recap, this model is defined as:

$$P(w_t \mid w_{<t}) = \text{softmax}(E^\top h^{(L)}_{w<t})$$
$$h^{(0)}_{w<t} = \frac{1}{t-1}\sum_{j=1}^{T-1} \sigma(Ew_j + p_j)$$

We now STACK LAYERS as follows:

$$\tilde{h}^{(i)}_{<t} = \text{FF}(h^{(i)}_{<t} \tag{1}$$
$$h^{(i+1)}_{<t} = \text{Attention}(h_{<1}, \ldots, h_{<t}) \tag{2}$$
$$\tag{3}$$

where $E \in \mathbb{R}^{d \times |\mathcal{V}|}, h \in \mathbb{R}^d$, and $p_j \in \mathbb{R}^d$ is our positional embedding. This works straight out of the box if we define each sequence as $s = w_1, \ldots, w_t, \ldots, w_T$ where $w_t \in \mathbb{R}^{|\mathcal{V}|}$ is a one-hot vector.

Recall the attention operation:

$$\text{Attention}(h_{<1}, \ldots, h_{<t}) = \sum_{j=1}^{t-1} \alpha_{t-1,j} h_j \tag{4}$$

$$\alpha_{t-1,j} = \frac{\exp(h_{t-1}^\top h_j)}{\sum_{j=1}^{t-1} \exp(h_{t-1}^\top h_j)} \tag{5}$$

Training

Intuitively, if our example sequence looks like:

$$s = \text{J'aime les films } \texttt{<START>} \text{ I like movies } \texttt{<END>}$$

then we need to learn:

$$P(\text{I} \mid \text{J'aime les films } \texttt{<START>})$$
$$P(\text{like} \mid \text{J'aime les films } \texttt{<START>} \text{ I})$$
$$P(\text{movies} \mid \text{J'aime les films } \texttt{<START>} \text{ I like})$$
$$P(\texttt{<END>} \mid \text{J'aime les films } \texttt{<START>} \text{ I like movies})$$

Our only parameter to be trained is E, and we initialize it randomly, i.e. $E \sim U[-0.0001, 0.0001]^{|\mathcal{V}| \times d}$. We train it via gradient descent,

$$E \leftarrow E - \nabla_E \mathcal{L}(E)$$

where $\mathcal{L}(E)$ is the negative log likelihood (NLL) over **only the target tokens** (let's call the index of the first target token k):

$$\mathcal{L}(E) = \overbrace{\frac{1}{M}\sum_{i=1}^{M}}^{\text{Average over dataset}} \underbrace{\sum_{j=k}^{T} -\log P(s_j \mid s_{<j})}_{\text{NLL over target tokens only}}$$

Generation

Now that we've trained our model to learn $P(y \mid x)$, how can we use this to actually generate a translation given x?

One strategy is **greedy decoding**, which just iteratively picks the most likely next word. To illustrate this, let's use our earlier example. Suppose

$$P(\cdot \mid \text{J'aime les films } \texttt{<START>}) = \begin{bmatrix} \vdots \\ 0.9 \\ 0.1 \\ 0.05 \\ \vdots \end{bmatrix}$$

and that the values shown are at indexes $7, 8, 9$ and correspond to strings "I", "we", and "us" respectively. Clearly, "I", is the most likely next word, and the way we represent this mathematically is via $\arg\max_{w \in \mathcal{V}}$, which just says: which token w in $\mathcal{V}$ has the highest probability? So here, we have

$$\hat{y}_1 = \arg\max_{w \in \mathcal{V}} P(w \mid \text{J'aime les films } \texttt{<START>}) = 7.$$

Note that if we were to take the max instead, the value would be 0.9.

Putting this all together, here's the formal algorithm:

Algorithm 1 Greedy Decoding Algorithm

input: $x_{1:t}$, P
$\hat{y}_0 \leftarrow$ `<START>`
$j \leftarrow 1$
while $j < t'$ **or** $y_{j-1} \neq$ `<END>` **do**
  $\hat{y}_j \leftarrow \arg\max_{w \in \mathcal{V}} P(w \mid x_{1:t}, \hat{y}_{0:j-1})$
  $j \leftarrow j + 1$
end while
return $\hat{y}_{1:j-1}$

A few things of note here:

1. We predict the next token based on the previous *predicted* token(s) - this means errors can compound if the previous token was predicted incorrectly.
2. In practice, instead of $j < t'$ the stop condition is a user-defined `max_new_tokens`, which can be much larger than t', and the `<END>` token is used to stop generation early.

Lastly, we need to **detokenize** each token in $\hat{y}_{1:t'}$, i.e. converting 7 in the earlier example to "I".

Evaluation

We evaluate with another dataset

$$\mathcal{D}' = \left\{ \left(x^{(i)}, y^{(i)} \right) \right\}_{i=1}^{N}$$

that was not seen during training (not used to compute loss or gradients or update weights). This dataset is called the *validation* set if we use it to help us improve our model[3], otherwise it's called the *test* set[4] if it's completely held out until the final evaluation.

First, we generate predictions (sequences) for all examples via the Greedy Decoding Algorithm (GDA),

$$\hat{y}_{1:t'}^{(i)} = \texttt{GDA}(x_{1:t}, P) \qquad \forall i \in [1, N]$$

Then, we compute the BLEU score between each predicted $\hat{y}_{1:t'}^{(i)}$ and the true $y_{1:t'}^{(i)}$, and take the average.

Model Building Process

To build a better model, we

1. Change $h_{w<t}$ to be a better neural network
2. Check if BLEU score went up
3. Rinse and repeat
4. Perform final evaluation on test set
5. Publish results if good, else perish.

[3]The validation set is very useful for indicating if your model is overfitted to the training data, especially if you run it periodically during training.

[4]Note that we can and should have both when possible. Without a test set, it is possible for a researcher to overfit during the model development cycle.