Natural Language Processing

Notes taken by Runqiu Ye
Carnegie Mellon University

Spring 2025

Contents

1 Introduction

Below is a list of topics that will be covered in this note. The topics include fundamental theoretical knowledge in language processing and language modeling in the first few sections, as well as frontier research results in later sections.

1. Language modeling fundamentals
   (a) Representing words
   (b) Language modeling
   (c) Sequence modeling architectures
2. Training and inference models
   (a) Decoding and Generation Algorithms
   (b) In-context learning
   (c) Pre-training
   (d) Fine-tuning
   (e) Reinforcement Learning
3. Evaluation and Experimental Design
   (a) Evaluating Language Generators
   (b) Experimental Design
   (c) Human Annotation
   (d) Debugging/Interpretation Techniques
4. Advanced Algorithms and Architectures
   (a) Advanced Pretraining, Post-Training, and Inference
   (b) Retrieval and Retrieval-augmented Generation
   (c) Long Sequence Models
   (d) Distillation and Quantization
   (e) Ensembling and Mixture of Experts
5. NLP Applications and Society
   (a) Complex Reasoning Tasks
   (b) Language Agents
   (c) Multimodal NLP
   (d) Multilingual NLP
   (e) Bias and Fairness

2 Language modeling fundamentals

2.1 Neural networks and auto differentiation

This process calculates the gradient of the loss with respect to different parameters. Use computational graph and chain rule to compute back propogation efficiently, and implmented in PyTorch. With the calculated gradient, we can use Stochastic Gradient Descent (SGD) to update the model parameters. Other than PyTorch, TensorFlow and JAX are also popular neural network frameworks with different advantages.

2.2 language and sequence modeling

In language modeling, other than binary or multi-class classification, sometimes there are exponential/infinitely many labels. This is called **structured prediction**. For example, assgining a part-of-speech tag for each word in the sentence. It is easy to note that in this case the number of labels grows exponentially with respect to the length of the sequence. There are also distinctions between **uncondictioned prediction** and **conditioned prediction**, which predict the distribution of $P(X)$ and $P(Y \mid X)$, respectively.

A language model is a probability distribution over all sequences. It can be used to **score** sequences and **generate** sequences. For example, in conditional generation, we condition on an input text and tries to continue it

$$\widehat{x}_{t+1:T} \sim p(X_{t+1:T} \mid x_{1:t}).$$

For another example, it can be used to translate between languages (machine translation). In all of these scenarios, the key is to model the following distribution:

$$p(x_{t+1} \mid x_1, x_2, \ldots, x_n).$$

2.2.1 Bigram models and N-gram models

The assumption of bigram models is

$$p(X) \approx \prod_{t=1}^{T} p_\theta(x_t \mid x_{t-1}).$$

This is to say all tokens only depend on the token right before it. To train the bigram models, we simply count

$$p(x_t \mid x_{t-1}) = \frac{\text{count}(x_{t-1}, x_t)}{\sum_{x'} \text{count}(x_{t-1}, x')}.$$

We can view $\theta_{i,j} = p(x_j \mid x_i)$ the parameters of the model. The reason behind the counting procedure is to produce a **maximum likelihood estimation**:

$$\max_\theta \sum_{x \in D_{\text{train}}} \log p_\theta(x).$$

This essentially tries to match p_θ to p_{data}, which can be more formally derived with KL divergence.

To evaluate the model, we can use the **log-likelihood** and the **perplexity** of the model. The log-likelihood is simply defined as

$$\text{LL}(\mathcal{X}_{\text{test}}) = \sum_{X \in \mathcal{X}_{\text{test}}} \log p(X).$$

We can also use the per-word log-likelihood $\text{WLL}(\mathcal{X}_{\text{test}})$, which is defined as the log-likelihood divided by the length of the test data. The perplexity of the model is $\exp\left(-\text{WLL}(\mathcal{X}_{\text{test}})\right)$, which can be used to simulate the number of tries it need to take before getting the correct prediction. Therefore, lower perplexity is better!

The bigram models or N-gram models in general have a lot of problems. When N gets large, the number of parameters will grow exponential with respect to N, but the occurrence of these parameters will be extremely rare. Additionally, it cannot share strength among similar words, it cannot condition on context with intervening words, and it cannot handle long-distance dependencies. To try it out, use the `kenlm`[1] toolkit.

2.2.2 Feedforward neural language model

Use a neural netowrk to model

$$p(X) = \prod_{t=1}^{T} p(x_t \mid x_{t-1}, \dots, x_{t-n+1})$$

by miniming the loss

$$L = -\log p_\theta(x_t \mid x_{1:t-1}).$$

This actually correspond to cross entropy loss. An example structure of this is shown in Figure 1.

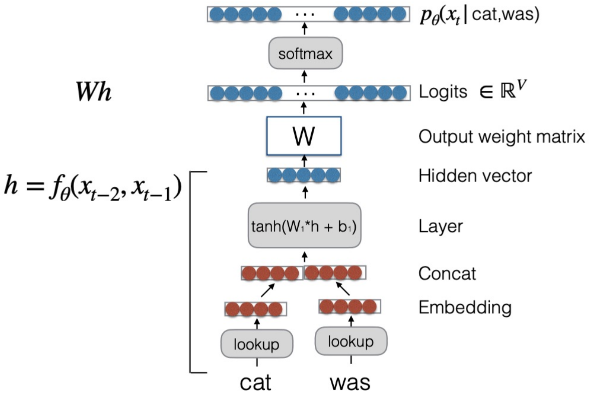

Figure 1: An illustration of feedforward neural language model.

Note that since we can make the embeddings of word closed to each other if the words have similar meaning, it solves the problem of N-gram models that they cannot share strength between similar words. Neural models can also condition on context with intervening words, which is another advantage compare to N-gram models.

2.2.3 Practice tips for neural language models

It is good to initialize weight matrices using the **Xavier initialization** technique, which is implmented in function `nn.init.xavier_uniform_` in PyTorch. Weight initialization is important in deep learning models, since it has a huge impact on the training results. For optimizer, instead of SGD, the most standard optimization method is **Adam**, which is also implmented in PyTorch. For the choice of learning rate, we also have consine learning rate schedule and warmup techniques.

Check out paper to learn more about these methods.

[1] `https://github.com/kpu/kenlm`

3 Recurrent neural networks

3.1 Setup of recurrent neural networks

A **sequence model** takes in a sequence $x_1, \dots, x_T$ and output some hidden state

$$f_\theta(x_1, \dots, x_T) \to h_1, \dots, h_T,$$

where $h_t \in \mathbb{R}^d$. For language modeling, we can have

$$p_\theta(\cdot \mid x_{<t}) = \mathrm{softmax}(Wh_t^T),$$

where W is some matrix we learned. In contrast, a **recurrent neural network** (RNN) has the following recurrence

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b),$$

where σ is some activation function (tanh, ReLU, ...). The parameters of the RNN model are $W_h \in \mathbb{R}^{d\times d}$, $W_x \in \mathbb{R}^{d\times d}$, and $b \in \mathbb{R}^d$.

RNN can be used for several applications. We can just run the RNN on a input sequence and use the final hidden state to do sentiment analysis. We can also train an output matrix W and use the hidden states to produce a sequence.

To train RNNs, we need to construct loss for each output (or just the final output if we are doing, for example, sentiment analysis) and run the back propogation algorithm to compute the gradients.

Note that in RNNs, computing h_t requires h_{t-1}, h_{t-2}, ..., so it is hard to parallelize the training process. Also, for inference, we generate one token, use the new hidden state (newly generated token) for the next step, and repeat. We only need to store previous hidden state and the computation for a length T sequence is simply T times the cost of a local computation.

3.2 Vanishing gradient

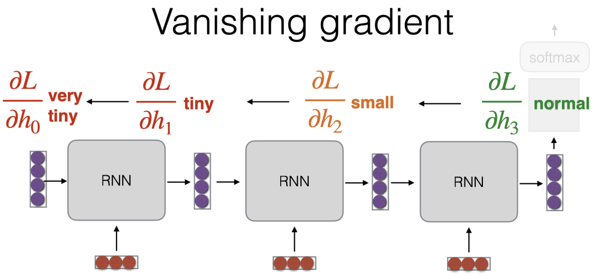
Vanishing gradient
softmax
$\frac{\partial L}{\partial h_0}$
very tiny
$\frac{\partial L}{\partial h_1}$
tiny
$\frac{\partial L}{\partial h_2}$
small
$\frac{\partial L}{\partial h_3}$
normal
RNN
RNN
RNN

Figure 2: An illustration of vanishing gradient in RNNs.

One problem with RNNs is vanishing gradient, which makes the model struggle to learn long-term dependencies. A illustration of vanishing gradient is in Figure 2. The mathematical detail can be seen through the following equation and derivation.

*** TO-DO: add math for vanish grad ***

A solution to this problem is gating and additive connections. The basic idea is to pass information across timesteps with a learned "gate" z_t, and the recurrence would just be

$$h_t = (1 - z_t)h_{t-1} + z_t\widetilde{h}_t.$$

More concretely, we replace each RNN black with a Gated Recurrent Unit (GRU). The GRU has two gates: a "reset gate" and an "update gate". The update gate z_t controls how much of the previous hidden state should be passed to the current hidden state. The reset gate r_t determines how much of the past information to forget. The equations for the GRU are as follows:

$$\begin{aligned} z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z), \\ r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r), \\ \widetilde{h}_t &= \tanh(W_h x_t + r_t \odot (U_h h_{t-1}) + b_h), \\ h_t &= (1 - z_t) \odot h_{t-1} + z_t \odot \widetilde{h}_t, \end{aligned}$$

where in the third equation $\widetilde{h}_t$ is a "candidate state" and the fourth equation gives recurrent update rule.

A similar but improved architecture compared to GRU is Long Short Term Memory (LSTM).

*** TO-DO: add architecture for LSTM ***

3.3 Encoder-decoder

The motivation is conditional generation – generate a sequence given a sequence with $p_\theta(y_1, \ldots, y_T \mid x)$. These models are historically developed for machine translation, dialogues, etc. The architecture of encoder-decoder is illustrated in Figure 3. The first RNN (encoder) read in the sequence x to generate the final hidden state (context vector) to represent the whole sequence. Then, the second RNN use the context vector to initialize the inital hidden state and hence generate a new sequence.

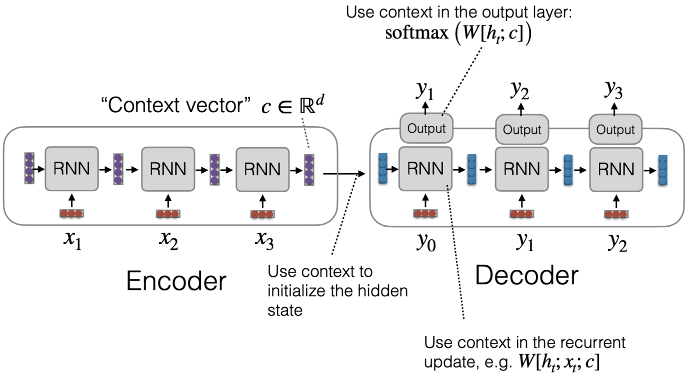
Use context in the output layer:
softmax $(W[h_t; c])$
"Context vector" $c \in \mathbb{R}^d$
y_1
y_2
y_3
Output
Output
Output
RNN
RNN
RNN
RNN
RNN
RNN
x_1
x_2
x_3
y_0
y_1
y_2
Encoder
Use context to initialize the hidden state
Decoder
Use context in the recurrent update, e.g. $W[h_t; x_t; c]$

Figure 3: An illustration of encoder-decoder architecture.

Typically, the parameters of the two RNNs are separator. Additionally, having a differentiable connection between the two RNNs is important, as the decoder need to learn its parameters from the sequence generated and the corresponding loss function.

Encoder-decoder only has one context vector for the whole sequence. Can we do better?

3.4 Attention