EECS 182/282A Lecture 17: Transformer Models

Instructor: Anant Sahai
Scribe: Chengyuan Li, Jackson Gao

November 13, 2022

1 Introduction

We have spent a lot of time understanding the idea of self-supervision where we only have access to unlabeled data (just like in unsupervised learning) and use data themselves to generate labels. One example that proved such learning scheme useful is doing **PCA-style dimensionality reduction "purification"** in an autoencoder style with self-supervision. Apart from PCA, **k-means style clustering** where we assign unlabeled data to groups is another kind of traditional unsupervised learning. In the first part of the lecture, we attempt to make k-means clustering compatible with gradient descent. Then, we will connect to the idea of attention and introduce transformer models.

2 Classic Lloyd's Algorithm for k-means

Given inputs $\{\vec{x_i}\}$, the following algorithm outputs cluster centers $\{\vec{r_j}\}$ such that $\vec{x_i}$ belongs to cluster j described by its center $\vec{r_j}$:

1. Initialize k cluster centers $\vec{r_1}, \vec{r_2}, \ldots, \vec{r_k}$ (e.g. randomly pick k input points)
2. Assign each data point $\vec{x_i}$ to a cluster j based on the closest $\vec{r_j}$
3. Update $\vec{r_j}$ to be the mean of cluster j
4. Repeat steps 2 & 3 until converged

When the algorithm sees a new point, it assigns the point to cluster j based on the closest $\vec{r_j}$. This is important because we want to use what we have learned on new data. Now we attempt to make the algorithm "trainable" with gradient descent.

2.1 SGD Approach (First Attempt)

Suppose we have initialized the cluster centers, we want to use our data points one by one to move the centers to be in a good place. This is different from the Lloyd's Algorithm in which we need to go through all data points in each update step.

1. Pick a point $\vec{x_i}$ (viewed as a mini-batch)
2. Find $\vec{r} = \text{argmin}_{\vec{r_j}} ||\vec{x_i} - \vec{r_j}||$
3. Minimize $||\vec{x_i} - \vec{r}||^2$ over $\vec{r}$ by taking a gradient step: $\vec{r} = \vec{r} + \eta \cdot 2(\vec{x_i} - \vec{r})$
4. Repeat steps 1 & 2 & 3 until converged

We can consider step 3 as an analog of step 3 in Lloyd's algorithm since setting $\vec{r_j}$ to be the mean of cluster j essentially minimizes the squared error between $\vec{r_j}$ and points in cluster j. However, step 2 is problematic because argmin is not differentiable. We want to replace argmin with an operation that is differentiable.

2.2 Softmax Idea

We first investigate the behavior of the following quantity:

$$e^{-\gamma||\vec{x_i}-\vec{r_j}||^2}. \tag{1}$$

Observe that when $\gamma > 0$, (1) is close to 0 if $\vec{r_j}$ is far from $\vec{x_i}$ and is close to 1 if $\vec{r_j}$ is close to $\vec{x_i}$. By scaling this quantity to be between 0 and 1, we can then apply normalization to turn them into probabilities and take expectations of interest.

Let $\alpha_j = \frac{e^{-\gamma||\vec{x_i}-\vec{r_j}||^2}}{\sum_l e^{-\gamma||\vec{x_i}-\vec{r_l}||^2}}$, we can view α_j as the amount we want $\vec{x_i}$ to pull $\vec{r_j}$. If $\vec{x_i}$ is far from $\vec{r_j}$, α_j is close to 0, and we do not want $\vec{x_i}$ to pull $\vec{r_j}$. If $\vec{x_i}$ is close to $\vec{r_j}$, α_j is close to 1, and we want $\vec{x_i}$ to pull $\vec{r_j}$. Therefore, minimizing the following quantity by taking gradient descent steps achieves our goal of getting $\vec{r_j}$ updated:

$$\sum_j \alpha_j ||\vec{x_i} - \vec{r_j}||^2. \tag{2}$$

Now we have everything ready for using gradient descent on k-means. Let's understand it by drawing the computational graph:

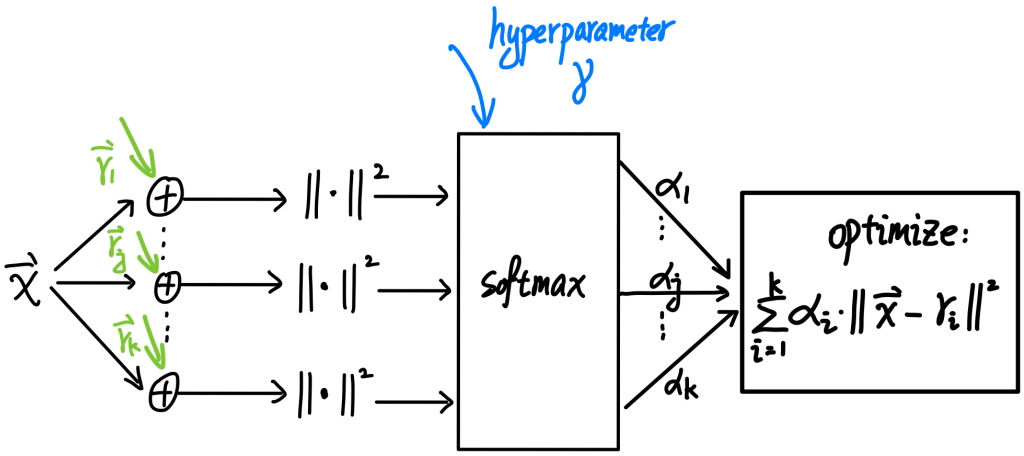

Figure 1: Computational graph (k-means with softmax idea)

Starting from the left, we first take the difference between input $\vec{x}$ and network parameters $\vec{r_1}, \vec{r_2}, \ldots, \vec{r_k}$, then take the squared norm of the differences and feed them into the softmax operator (with hyperparameter γ), which outputs α for the computation of our objective function (2). From there, we use gradient descent to backprop. (Note: In deep learning, the hyperparameter γ is traditionally referred to as "temperature" due to its analogy to statistical mechanics.)

Let's compare the softmax idea with our first attempt of using SGD. In our first attempt, we pick a point $\vec{x_i}$ and use it to update the closest $\vec{r_j}$ in each iteration, and the amount we move $\vec{r_j}$ is based on the learning rate η and the distance between $\vec{x_i}$ and $\vec{r_j}$. In the softmax idea, we do similar things in a batch, updating all $\vec{r_j}$ at once, but we control how much each $\vec{r_j}$ is updated by α_j (i.e. the softmax version of the distance between $\vec{x_i}$ and $\vec{r_j}$). Using the softmax idea, we successfully eliminated the problem we had in SGD while accomplishing our goal.

2.3 Goofy Alternative

Suppose we have the ideal case where clusters are widely separated as shown in Figure 2. $\vec{r_1}$, $\vec{r_2}$, $\vec{r_3}$ represent the center of each cluster. Let's consider a point $\vec{x_i}$ (circled, in the upper right cluster). With the softmax idea (1) where γ is relatively large, we have α_1, α_3 close to 0, and α_2 close to 1. This motivates us to think about an alternative approach for gradient descent.

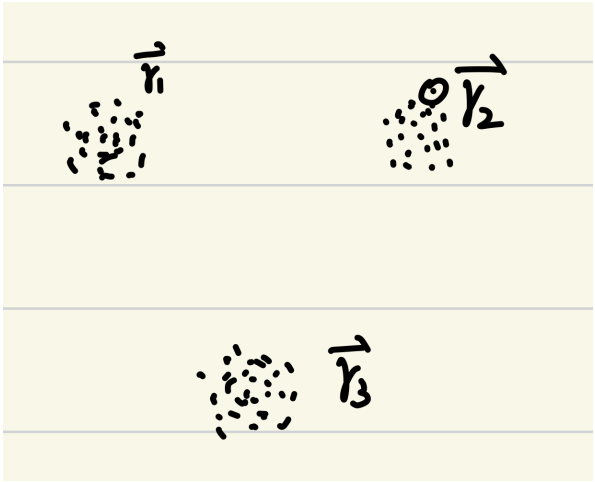

Figure 2: Motivation of goofy alternative

Instead of optimizing (2), we treat α_j as the probability that $\vec{x_i}$ is the closest to $\vec{r_j}$ and compute the expectation $\vec{\hat{r}} = \sum \alpha_j \vec{r_j}$ of where the closest cluster center is (to $\vec{x_i}$). We then minimize the distance $||\vec{\hat{r}} - \vec{x_i}||^2$ by gradient descent, which looks more like the standard gradient descent objective.

The goofy approach is interesting due to its connection with the attention mechanism. In attention, we have $\vec{k}$ (key), $\vec{q}$ (query), and $\vec{v}$ (value). We take the average of the values based on similarities to the keys that have gone through appropriate softmax. Here in $\vec{\hat{r}} = \sum \alpha_j \vec{r_j}$, we can treat $\vec{r_j}$ as the value $\vec{v}$ and α_j as the key $\vec{k}$ that have gone through softmax.

Again, let's understand it by drawing out the computational graph:

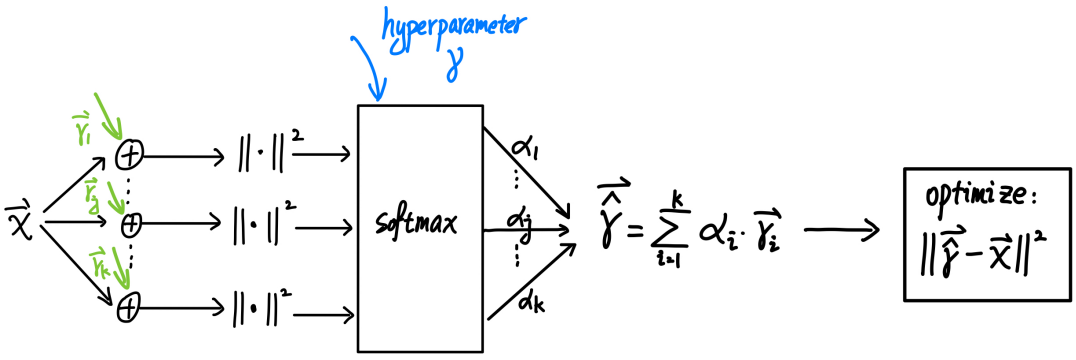

Figure 3: Computational graph (k-means with goofy alternative)

Comparing with the previous computational graph, we see that the only difference is the objective function to optimize at the end. Instead of directly optimize (2), we first compute the expectation $\vec{\hat{r}}$, then optimize the distance $||\vec{\hat{r}} - \vec{x}||^2$.

3 Transformer Models

Transformer models are first introduced in the paper *Attention is all you need*[1]. We will introduce two key ideas of transformers.

3.1 Key Idea 1

In the previous lecture, we talked about RNN which uses states to memorize knowledge. Attention mechanism can also be viewed as a kind of memory, which is a soft approximate hash table. Here we try to only use attention to "write things down" instead of using states as internal memory. Now, what we can do is to write memory and query memory. Then it comes to the first key idea of transformers: "I don't need internal memory if I can write notes to myself". What happens is we drop recurrence from the RNN approach, but keep weight sharing (across time).

Based on that, we can intuitively consider transformer models as GNN, which share weights between different nodes. Nodes in transformer models that pass across time can also be viewed as a fully connected graph. These basic ideas are similar.

This idea relies on the attention mechanism. We have introduced two ways of understanding attention:

1. Soft approximate hash table
2. Queryable softmax pooling

In the field of computer science, we can consider attention as a hash table; but in deep learning style, it is a pooling layer combined with softmax.

3.1.1 Recall attention

From the previous lecture, we have three vectors in attention: $\vec{k}, \vec{q}, \vec{v}$, and we have a hash table containing pairs of $\vec{k_i}$ and $\vec{v_i}$.

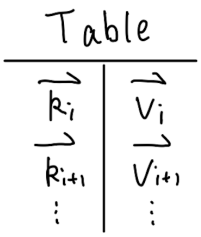

Figure 4: Attention hash table

When we apply a query $\vec{q_t}$, we can calculate the output as follows:

$$e_{i,t} = \frac{< \vec{q_t}, \vec{k_i} >}{\sqrt{d}} \tag{3}$$

$$\alpha_{i,t} = \frac{e^{e_{i,t}}}{\sum_j e^{e_{j,t}}} \tag{4}$$

$$output = \sum_i \alpha_{i,t} \vec{v_i} \tag{5}$$

In equation (3), d represents the dimension of the key. This process is what we called queryable softmax pooling, and equation (4) is doing softmax. Besides, if the process generating q is also populating the table with k, v, we call it self-attention. Basically, if you are writing to the hash table at the same time, it is self-attention; otherwise, if you only read the value from others, it is not. Just as Figure 5 shows, if x_{i+1}'s key also query its own key, then it is self-attention.

Aside: The product of $\vec{q_t}$ and $\vec{k_i}$ can be negative or positive, which represents the direction of the resulting vector. In transformer models, the negative value means they are quite different, but in other models it may have different meanings.

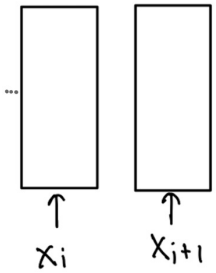

Figure 5: Self-attention

3.2 Key Idea 2

It is useful and important to use multiple channels in conv-nets, and we also want to use multiple channels in transformer models. We call it multi-headed attention in transformer models, and it has many parallel queryable softmax pooling.

If we think about max pooling in conv-nets, different channels may select different features. Now, when we generate different queries, it may comes up with different values.

Note: This lecture ended early as the scheduled fire alarm drill occurred.

References

[1] Ashish Vaswani et al. "Attention Is All You Need". In: (2017). DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.