Deep Set Prediction Network

Motivation

Current approaches for predicting sets from feature vectors ignore the unordered nature of sets and suffer from discontinuity issues as a result. The main difficulty in predicting sets comes from the ability to permute the elements in a set freely, which means that there are $n!$ equally good solutions for a set of size n. Models that do not take this set structure into account properly, such as MLPs *(Multilayer perceptrons)* or RNNs (*recurrent neural networks*), result in discontinuities, which is the reason why they struggle to solve simple toy set prediction tasks.

Background

Representation Sets of feature vectors with the feature vector describing properties of the element. A set of size n wherein each feature vector has dimensionality d is represented as a matrix $Y \in \mathbb{R}^{d \times n}$ with the elements as columns in an arbitrary order, $\mathbf{Y} = [\mathbf{y}_1, \ldots \mathbf{y}_n]$. To properly treat this as a set, it is important to only apply operations with certain properties to it: **permutation-invariance or permutation-equivariance**. In other words, operations on sets should not rely on the arbitrary ordering of the elements.

Definition 1 (Permutation-invariant)**.** A function $f : \mathbb{R}^{n \times c} \to \mathbb{R}^d$ is permutation-invariant iff it satisfies

$$f(\mathbf{X}) = f(\mathbf{PX}) \tag{1}$$

for all permutation matrices $\mathbf{P}$.

Definition 2 (Permutation-equivariant)**.** A function $g : \mathbb{R}^{n \times c} \to \mathbb{R}^{n \times d}$ is permutation-equivariant iff it satisfies

$$\mathbf{P}g(\mathbf{X}) = g(\mathbf{PX}) \tag{2}$$

for all permutation matrices $\mathbf{P}$.

Set encoders (which turn such sets into feature vectors) are usually built by composing permutation-equivariant operations with a permutation-invariant operation at the end. A simple example is the model in

$$f(\mathbf{Y}) = \sum_{i=1}^{n} g(\mathbf{y}_i) \tag{3}$$

where g is a neural network. Because g is applied to every element individually, it does not rely on the arbitrary order of the elements. We can think of this as turning the set $\{\mathbf{y}_i\}_{i=1}^n$ into $\{g(\mathbf{y}_i)\}_{i=1}^n$. This is permutation-equivariant because changing the order of elements in the input set affects the output set in a predictable way. Next, the set is summed to produce a single feature vector. Since summing is commutative, the output is the same regardless of what order the elements are in. In other words, summing is permutation-invariant. This gives us an encoder that produces the same feature vector regardless of the arbitrary order the set elements were stored in.

Loss We need to compute a loss between predicted set $\hat{\mathbf{Y}} = [\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_n]$ and the target set $\mathbf{Y} = [\mathbf{y}_1, \ldots \mathbf{y}_n]$. The elements of each set are in an arbitrary order, so we cannot simply compute a pointwise distance. We use a loss function that is permutation-invariant in both its arguments.

1. Chamfer loss
$$L_{\text{cha}}\left(\mathbf{Y}, \hat{\mathbf{Y}}\right) = \sum_i \min_j \left\|\hat{\mathbf{y}}_i - \mathbf{y}_j\right\|^2 + \sum_j \min_i \left\|\hat{\mathbf{y}}_i - \mathbf{y}_j\right\|^2 \tag{4}$$
Note that this does not work well for multi-sets, for example $[a, b, b]$ and $[a, a, b]$ - the loss is 0.

2. Hungarian loss
$$L_{\text{hun}}\left(\mathbf{Y}, \hat{\mathbf{Y}}\right) = \min_{\pi \in \Pi} \left\|\hat{\mathbf{y}}_i - \mathbf{y}_{\pi(i)}\right\|^2 \tag{5}$$
where Π is the space of permutations. This has the benefit that every element in one set is associated to exactly one element in the other set, which is not the case for the Chamfer loss.

Deep Set Prediction Networks

A model for decoding a feature vector into a set of feature vectors. The main idea is based on the observation that the gradient of a set encoder with respect to the input set is permutation-equivariant.

Theorem 1. *The gradient of a permutation-invariant function $f : \mathbb{R}^{n \times c} \to \mathbb{R}^d$ with respect to its input is permutation-equivariant*

$$\boldsymbol{P}\frac{\partial f(\boldsymbol{X})}{\partial \boldsymbol{X}} = \boldsymbol{P}\frac{\partial f(\boldsymbol{PX})}{\partial \boldsymbol{PX}}. \tag{6}$$

That means: *to decode a feature vector into a set, we can use gradient descent to find a set that encodes to that feature vector.* This gives rise to a **nested optimisation** : an inner loop that changes a set to encode more similarly to the input feature vector, and an outer loop that changes the weights of the encoder to minimise a loss over a dataset.

Auto-encoding fixed size sets

In a set auto-encoder, the goal is to turn the input set $\mathbf{Y}$ into a small latent space $\mathbf{z} = g_{enc}(\mathbf{Y})$ with the encoder genc and turn it back into the predicted set $\hat{\mathbf{Y}} = g_{dec}(\mathbf{z})$ with the decoder g_{dec}. We define a representation loss and the corresponding decoder as

$$L_{\text{repr}}\left(\mathbf{z}, \hat{\mathbf{Y}}\right) = \left\|g_{\text{enc}}\left(\hat{\mathbf{Y}}\right) - \mathbf{z}\right\|^2 \tag{7}$$

$$g_{\text{dec}}(\mathbf{z}) = \arg\min_{\hat{\mathbf{Y}}} L_{\text{repr}}\left(\mathbf{z}, \hat{\mathbf{Y}}\right) \tag{8}$$

In essence, L_{repr} compares $\hat{\mathbf{Y}}$ to $\mathbf{Y}$ in the latent space.
Since the problem is non-convex when g_{enc} is a neural network, it is infeasible to solve (8)

exactly. Instead, we perform gradient descent to approximate a solution. Starting from some initial set $\hat{\mathbf{Y}}^{(0)}$, gradient descent is performed for a fixed number of steps T with the update rule

$$\hat{\mathbf{Y}}^{(t+1)} = \hat{\mathbf{Y}}^{(t)} - \eta \cdot \frac{\partial L_{\text{repr}}\left(\mathbf{z}, \hat{\mathbf{Y}}^{(t)}\right)}{\partial \hat{\mathbf{Y}}^{(t)}} \tag{9}$$

with η as the learning rate and the prediction being the final state, $g_{\text{dec}}(\mathbf{z}) = \hat{\mathbf{Y}}^{(T)}$. This is the aforementioned inner optimisation loop.
To obtain a good representation $\mathbf{z}$, we still have to train the weights of g_{enc}. For this, we compute the auto-encoder objective $L_{\text{set}}\left(\mathbf{Y}, \hat{\mathbf{Y}}^{(T)}\right)$ with $L_{\text{set}} = L_{\text{cha}}$ (4) or L_{hun} (5) and differentiate with respect to the weights as usual, backpropagating through the steps of the inner optimisation. This is the aforementioned outer optimisation loop.

Predicting sets from a feature vector

Since the target representation $\mathbf{z}$ can come from a separate model (for example an image encoder F encoding an image $\mathbf{x}$), producing both the latent representation as well as decoding the set, is no longer possible in the general set prediction setup. When naïvely using $\mathbf{z} = F(\mathbf{x})$ as input to our decoder, our decoding process is unable to predict sets correctly from it. Because the set encoder is no longer shared in our set decoder, there is no guarantee that optimising $g_{enc}(\hat{\mathbf{Y}})$ to match $\mathbf{z}$ converges towards $\mathbf{Y}$ (or a permutation thereof). To fix this, we simply add a term to the loss of the outer optimisation that encourages $g_{enc}(\mathbf{Y}) \approx \mathbf{z}$ again. In other words, the target set should have a very low representation loss itself. This gives us an additional L_{repr} term in the loss function of the outer optimisation for supervised learning

$$\mathcal{L} = L_{\text{set}}\left(\mathbf{Y}, \hat{\mathbf{Y}}\right) + \lambda L_{\text{repr}}(\mathbf{Y}, \mathbf{z}). \tag{10}$$

With this, minimising $L_{\text{repr}}(\mathbf{Y}, \mathbf{z})$ in the inner optimisation will converge towards $\mathbf{Y}$.

Algoritmus 1: One forward pass of the set prediction algorithm within the training loop.

```
1 $\mathbf{z} = F(\mathbf{x})$
2 $\hat{\mathbf{Y}}^{(0)} \leftarrow init$
3 for $t \leftarrow 1, T$ do
4     $l \leftarrow L_{\text{repr}}\left(Y^{(t-1)}, z\right)$
5     $\hat{Y}^{(t)} \leftarrow \hat{Y}^{(t-1)} - \eta \cdot \frac{\partial l}{\partial \hat{Y}^{(t-1)}}$
6 end
7 predict $\hat{Y}^{(T)}$
8 $\mathcal{L} = \frac{1}{T}\sum_{t=0}^{T} L_{\text{set}}\left(Y, \hat{Y}^{(t)}\right) + \lambda L_{\text{repr}}(Y, z)$
```