CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2022 Marvin Zhang

Discussion 9

This discussion focuses on machine translation and distribution shift.

1 Machine Translation

In this section we will review several techniques in applying deep learning models to machine translation. As we have discussed in last section, machine translation is a typical text to text NLP problem, where the input sequence has the around the same amount of information as the output sequence. Typically, modern machine translation model uses transformer architecture with both encoder and decoder, and the model is trained with supervised learning. At test time, the translation output is generated using beam search.

1.1 Compensating for Bad Models

One problem of imperfect machine translation models is that they often bias towards very short output sequences. This is because these models often assign low probabilities to all the tokens in the vocabulary, therefore a long output sequence would have lower overall probability because of the multiplication of these low probability tokens. Let's take a look at a concrete example.

Problem: Model Bias Towards Short Sequences

Imagine our vocabulary contains N tokens, including the `<END>` token that terminates the decoding process. Now suppose we have a really bad model that assigns uniform probability to each of the N tokens at every step regardless of the inputs. Now given the beginning token `<BEGIN>` of the sentence, which output sequence has the highest probability under this model?

In order to compensate for the bias towards short sequences, several correction techniques have been proposed, including normalizing the log probability by the sequence length, or adding a bonus for each additional word in the sequence. These corrections are applied during the beam search to rank the sequences in the beam.

1.2 Byte Pair Encoding

So far, we treat our sentences as sequences of words, and convert each word into a token. However, this is often not the best practice in NLP. In order to train our NLP model to be able to handle all tokens accurately as input and output, each token should be common enough that the embeddings can be estimated robustly and all the tokens should have been observed during training. Such condition is not true if we use words as tokens, since there are many rare words and made up words that don't appear frequently. A naive solution is to replace them by the unknown word token `<UNK>`. This approach is problematic because once we replace all the rare words by a single token `<UNK>`, the `<UNK>` token starts to become a common word because it combines all the probabilities of different rare words. Hence our model will output a lot of `<UNK>` tokens. A better approach is to use subword tokenization, where we break down a word into multiple subwords and use these subwords as tokens. For example, the word "learning" might be broken down into "learn" and "ing".

There are many different ways to break down a word into subwords. We introduce a popular approach called byte pair encoding (BPE). BPE was originally proposed as a form of data compression and recently re-purposed for tokenization in NLP. The BPE process starts with a vocabulary set with all the characters in the corpus. In each step, we find the most frequent pair of tokens, combine them to create a new token,

and add the new token to the vocabulary set. This step is repeated until the vocabulary set reaches the desired size. We describe BPE in the followling algorithm.

Algorithm 1 Byte Pair Encoding

Hyperparameters: Desired vocabulary size L.
Initialize vocabulary V set from all characters.
while $|V| < L$ **do**
  Find most frequent pair of tokens $x, y \in V$
  Combine x, y into new token w, and add to vocabulary $V \leftarrow V \cup \{w\}$

2 Distribution Shift

In this section, we will review the problem of distribution shift. Recall that in order to solve a machine learning problem, we train our model on the training set, tune our hyperparameters on the validation set and report the final performance on the test set. These three partitions are randomly partitioned from the same dataset, and therefore follow the same distribution. This practice follows the paradigm of **empirical risk minimization**, where we use the empirical (dataset) distribution to approximate the true data distribution and minimize our risk on top of it. Suppose that we obtain a model θ from ERM, it can be easily shown that the expected loss of θ under the true data distribution is tightly bounded by the empirical loss on the test set. For example, suppose our loss function $l(\theta, x, y)$ is bounded between 0 and 1, then by Hoeffding's inequality:

$$\mathbb{P}\left(\left|\frac{1}{N}\sum_{i=1}^{N} l(\theta, x_i, y_i) - \mathbb{E}[l(\theta, \mathbf{x}, \mathbf{y})]\right| \geq t\right) \leq \exp\left(-\frac{2t^2}{N}\right)$$

Hence, we can deploy our model safely if our test data follow the same distribution as our dataset. However, real world problems are full of **distribution shift**, where the test data follow a different distribution, and directly deploying our model will result in worse performance.

2.1 Robustifying Against Distribution Shift

There are several common strategies of making our model more robust against distribution shift. The most straightforward one is simply to train a larger model with a larger and more diverse dataset. This will improve the general performance of our model as well as the robustness. However, in many cases we do not have access to a larger dataset or the compute resources to train a larger model. Alternatively, we can apply data augmentation during training to improve the robustness. Here are a few typical methods for data augmentation:

- **Mixup** applies element-wise convex combination of two examples to produce a new training example. For instance, given input x_1, x_2 and label y_1, y_2, mixup produces a new training example with input $\alpha x_1 + (1-\alpha)x_2$ and label $\alpha y_1 + (1-\alpha)y_2$. Mixup improves robustness against corruption.
- **AutoAugment** finds complex augmentation strategy by searching the space of combinations of elementary data augmentation operations.
- **AugMix** mixes together random augmentations, using many of the same operations from AutoAugment.
- **PixMix** mixes in a completely different image dataset and results in consistently good performance across several metrics.

2.2 Anomaly and Out-of-Distribution Detection

While there are many strategies to improve the robustness of our models against distribution shift, sometimes the distribution shift might be too large for any robust model to perform well. For example, imagine that our training set is MNIST hand written digits and our test set is ImageNet. In this case there's no way for any model to perform well under this kind of distribution shift. Instead, one thing we can do is to detect this distribution shift, and this approach is known as anomaly detection. A 2D example of anomaly detection can be seen in Figure 1.

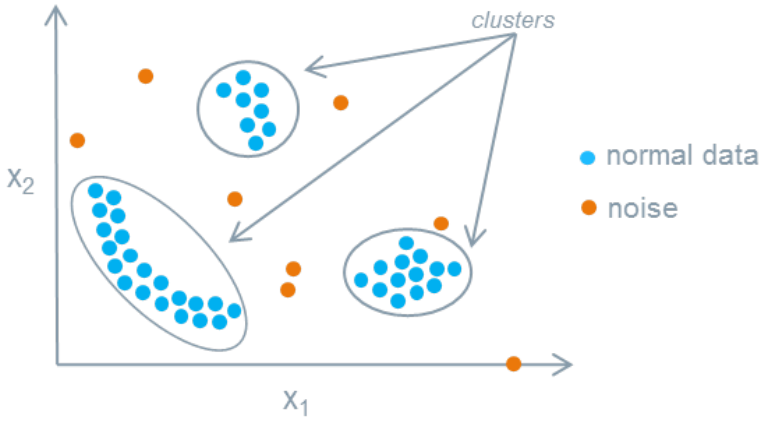

Figure 1: 2D example of anomaly detection.

A very good baseline for anomaly detection is to use the model's **confidence** $\max_k p_\theta(y = k|x)$. Specifically, the confidence measures how sure the model is about its own prediction. The lower the confidence is, the more likely the model's prediction is wrong. This simple baseline works reliably across computer vision, NLP, and speech recognition classification tasks, though it can't detect adversarial examples.