Batch Normalization Notes

Piyush Patil

September 19, 2017

1 Background

In the standard statistical learning theoretic framework for general function learning, problems are modeled as follows: there's some vector space X of possible inputs and a vector space Y of possible outputs, and some (unknown) probability distribution p over $X \times Y$, i.e. $p(x, y)$, for $x \in X$ and $y \in Y$, is the probability density of sampling the input x and the output y. Thus, p is the joint distribution of the corresponding probability distributions over X and Y. The problem, then, is to search some parameterized function space $\mathcal{H}$ so as to converge to a function $f_\theta : X \to Y \in \mathcal{H}$ which minimizes the risk

$$\forall g \in \mathcal{H} : I[g] = \int_{X \times Y} L(f(x), y) dp(x, y)$$

where L is a loss function.

In practice, when we train on a finite training set, we assume that the training set is a finite random sample of the full input space X, as is the test set. Thus, though we have to deal with the inevitable noise in our dataset, we can approximate the optimal function in $\mathcal{H}$, and converge to it in expectation as the size of the training set grows. This is referred to as minimizing the empirical risk, rather than the true risk, where the empirical risk is defined, for dataset $\{(x_i, y_i), 1 \leq i \leq N\}$, as

$$\forall g \in \mathcal{H} : R[g] = \frac{1}{N} \sum_{i=1}^{N} L(x_i, y_i)$$

For this reason, it's critical that the distribution of inputs from which the training set is drawn matches the distribution of inputs from which the test set is drawn. As a trivial example to demonstrate this, if we took our test set and doubled every input vector, clearly a model trained on a corresponding training set, whose elements are now from a distribution half as large as the test set distribution, will not generalize to the test set very well at all.

Furthermore, there are various transformations we can do to the input space X to make it more amenable to learning (again, as long as we remember to put both all our data, including training, test, validation, etc. sets through the same transformation). For instance, under the conventional representation learning paradigm, we assume that each point in the embeddings of our input correspond to some feature. It makes sense to normalize our features, i.e. by mapping raw, numerical feature values to their z-scores, computing over the entire dataset:

$$x_i \mapsto \frac{x_i - \mathop{\mathbb{E}}_j[x_j]}{\mathop{\mathrm{Var}}_j[x_j]}$$

This unbiases the model, in a sense, because numerically it's seeing feature values that can be interpreted numerically the same way over all features. Without the standardizing step, certain features might have much larger or translated ranges relative to others, forcing the model to learn these feature-specific differences an develop, for each feature, its own "sense" of what the average "feature" looks like and how varied they are (and perhaps higher-order moments of the feature distribution as well). By forcing the mean and variance of each feature distribution to be zero and one, respectively, we remove this unnecessary bias.

2 Internal covariate shift

Consider a deep neural model with many layers. Though the first layer sees the raw input distribution, a problem is the "input distribution" subsequent layers see is really the true input distribution put through several composed transformations, corresponding to the previous layers. Put more formally, let's use $F_l(u, \theta_l)$ to denote the transformation computed by layer l of our model, on input u and with layer parameters θ_l. Then if the input to the entire model is x and we use o_l to denote the output of layer l, we have

$$o_l = F_l(o_{l-1}, \theta_l) = F_l(F_{l-1}(o_{l-2}, \theta_{l-1}), \theta_l) = F_l(F_{l-1}(\cdots(F_1(x, \theta_1), \theta_2)\cdots, \theta_{l-1}), \theta_l)$$

Clearly, the input distribution seen by layer l is the true distribution put through $l-1$ transformations, depending on the parameters $\theta_1, \cdots, \theta_{l-1}$. This poses a problem when we apply iterative optimization methods such as gradient descent. Since the transformations through which we put the input distribution depend on the parameters from previous layers, the input distribution seen by layer l changes every time we update the parameters, since after all, we've now essentially applied a different transformation to the true input distribution before handing the input to layer l. This phenomenon, wherein the input distribution seen by layers changes as parameters change, is known as internal covariate shift, and hinders learning since the model is now forced to not only learn a mapping so as to minimize the (empirical) risk, but also to unlearn biases specific to certain input distributions depending on outdated parameters that it's accumulated. More intuitively, it's simply more difficult to learn in a non-stationary environment, i.e. an environment which changes every time we make a move in the environment.

3 Proposed solution

Ideally, the input distribution would remain the same for a layer across parameter updates. To fully "undo" the difference in transformations thorugh which put the input distribution caused by a parameter update, we would need to learn an auxiliary, complex inversion function, which would also have the antagonistic effect of fighting gradient descent updates in its effort to pull input distributions back to the very first distribution seen in a random parameter initialization. Further, this would prove computationally very taxing. Instead, batch normalization is an idea which proposes an approximate, quick and computationally feasible solution. Though an imperfect and incomplete solution, heuristically we'd expect that if we applied the naive, initial whitening step of standardizing the input distribution not just to the training set (which is the "true" inputs) but to every layer's inputs, it'd be a step in the right direction towards forcing the input distributions seen by layers across parameter updates to being stationary. Of course, because we don't have a full picture of the new input distribution that a layer sees, even standardization is impossible (we simply don't have the mean and variance). Instead, we can approximate these distribution moments by standardizing our input batch relative to the mean and variance of the batch.

One issue this approach might raise is that because we've now put our inputs through a standardizing transformation, we've inherently reduced the expressivity of our model. After all, the activation functions through which we put our input distribution now essentially has a restricted range, since we've modified its domain to conform to a standard distribution. In many cases, this actually accelerates neuron saturation. To account for this, we simply introduce new parameters into each layer which let that layer choose what the ideal activation range it wants to use is. Specifically, because standardization is a linear transformation, we introduce, for each layer l, two new parameters: α_l and β_l. If u_l is the input to layer l, and $\hat{u_l}$ is the standardized input, then we compute the output of layer l as

$$o_l = F_l(\alpha_l \hat{u_l} + \beta_l, \theta_l)$$

By tuning α_l and β_l, the layer is free to manipulate the standardized input distribution to have whatever range it desires it to have. Indeed, in the "worse case" that it turns out that the addition of batch normalization is, overall, detrimental, the layer is free to learn values which simply invert the standardization transform, reducing the network to the vanilla model we started with.

4 Conclusion

Ideally, by taking a step towards making the input distributions layers see stationary over parameter updates, by whitening inputs using standardization, we allow the model to perform learning in a more unbiased loss landscape. This accelerates learning and allows us to use larger learning rates.