Finetuning

Through pretraining on a broad range of text, we've come to a probability distribution p_{θ_p}. We might not want to sample from this distribution, though! However, we also have the learned parameters θ_p themselves, of a neural network. These parameters can serve as the initialization of an optimization problem for more specific task. This is **finetuning**. It is a fascinating empirical fact of neural networks that the pretrained parameters θ_p are an excellent starting point.

Finetuning on a task

Consider a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$ of examples, where each $x_i \in \mathcal{V}^*$, and $y_i \in \mathcal{V}^*$. We want a probability distribution that generates outputs like the y when given inputs like the x. We have θ_p to start with. We're going to take minibatches from $\mathcal{D}$, start with $\theta_0 = \theta_p$, and then run stochastic gradient descent:

$$\theta_0 = \theta_p \tag{1}$$

$$\theta_i = \theta_{i-1} - \alpha \nabla_\theta \mathbb{E}_{\text{minibatch}(\mathcal{D})} \left[-\log p_\theta(y_i \mid x_i) \right] \tag{2}$$

So, we're just optimizing over the parameters of our network the same loss function had we started from a random initialization instead of our pretrained θ_p. But we're hoping that starting from where we do will lead to good things. And in practice it does! But one should ask, why? When we write out an optimization problem:

$$\min_\theta \mathbb{E} \left[-\log p_\theta(y \mid x) \right] \tag{3}$$

if we do a great job of computing that minimum, it shouldn't matter so much where we start. In general, non-convex optimization is hard, though, so maybe starting from θ_p helps us achieve a lower minimum loss. Unfortunately, this isn't really true either. Even for relatively large datasets $\mathcal{D}$, for the large neural networks that we use, we can tend to achieve roughly minimum (e.g., 0) loss via gradient descent even without pretrained initialization. So, if pretraining does not necessarily make optimization better, why would we want to start from θ_p?

Intuitions of finetuning

At a high level, the success of finetuning is due to the parameters of the network not straying "too far" from the pretrained parameters. The pretrained parameters θ_p lead to building all sorts of interesting representations, similarities between words and phrases with similar meanings, subskills relating to understanding language. When finetuning, we leverage these learned representations, and just "tweak" them a bit such that the output distribution is what we'd like. Let's use an example to demonstrate this.

Consider two models: θ_0, which has randomly initialized parameters, and θ_p, our pretrained model. We train both of them on the dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^m$, where x_i is a question about a university's location and y_i is the corresponding answer. For example, one training pair could be (Q: Where is the University of Pennsylvania?, A: Philadelphia). Now, if we select an example from the held-out set — say, Q: Where is Pennsylvania State University? — and ask both models to predict its location, what answers could we get? The randomly initialized model θ_0 might output a biased answer such as "Philadelphia", since "Pennsylvania" and "Philadelphia" could have close embedding space. In contrast, the pretrained model θ_p, having likely seen information about Pennsylvania State University during pretraining on large-scale web data, would be more likely to output the correct answer, State College. Therefore, by starting from

θ_p, we encourage the model not to stray "too far" from its pretrained parameters. In other words, finetuning mainly adjusts our desired model behavior, such as its format of response, instead of changing the actual information it has already learned.

Adapting parameters

The intuition of not straying "too far" from the pretrained parameters is nice because we can try to define what "too far" means, and develop our intuitions. One natural idea is L_2 distance — simply, in a natural vector space metric, how much have we moved due to gradient descent?

$$\|\theta_p - \theta\|_2 \tag{4}$$

Is this the right notion of distance in our finetuning? One way of exploring this is to run our optimization procedure with an explicit penalty on L_2 distance, as follows:

$$\min_\theta \mathbb{E}\left[-\log p_\theta(y \mid x)\right] + \beta\|\theta_p - \theta\|_2 \tag{5}$$

where β is a hyperparameter deciding the relative importance of not moving too far in weight space. This is called an L_2 penalty, and it's a reasonable strategy, but in practice isn't used too much. Intuitively, this penalizes every single parameter equally for how much it strays from its initial position, and maybe some parameters should move a lot, while others shouldn't move much at all.

Another interesting space to measure "too far" in is distributional space. Instead of penalizing the deviation of the parameters themselves, maybe we want to make sure our learned distribution p_θ isn't too different from the pretrained distribution p_{θ_p}. One way to do this is to decide a distribution over text, say $\mathcal{T}$ and then penalize the KL-divergence of the token-level distributions of each model on the prefixes defined by that text:

$$\mathbb{E}_{x\sim\mathcal{T}}\left[\mathrm{KL}(p_{\theta_p}(\cdot \mid x_{<t})\|p_\theta(\cdot \mid x_{<t}))\right] \tag{6}$$

Recall that $p_{\theta_p}(\cdot \mid x_{<t})$ is a $|\mathcal{V}|$-sized categorical distribution over tokens x_t given prefix $x_{<t}$, and the KL-divergence of one such conditional distribution is

$$\mathrm{KL}(p_{\theta_p}(\cdot \mid x_{<t})\|p_\theta(\cdot \mid x_{<t})) = \sum_{w\in\mathcal{V}} p_{\theta_p}(w \mid x_{<t}) \log \frac{p_{\theta_p}(w \mid x_{<t})}{p_\theta(w \mid x_{<t})} \tag{7}$$

where $p_{\theta_p}(w \mid x_{<t})$ is the weight of token w according to θ_p, and $\frac{p_{\theta_p}(w|x_{<t})}{p_\theta(w|x_{<t})}$ shows how different p_{θ_p} and p_θ are for that token prediction. So, if the pretrained distribution places a lot of probability on a word in a given prefix and our finetuned distribution does not, the parameters are penalized and pushed towards placing more probability on that word. Is this the right intuition for not straying too far from the pretrained parameters? Sort of, sometimes. Intuitively, sometimes we want to change the distribution quite a bit — say we pretrained mostly on English text, but we want to finetune on translation to Tamil. It turns out that the English pretraining is still quite useful for this task, despite the pretrained distribution being quite different from the one we'd like. Therefore, we can add this **KL penalty** as an extra objective, which is a common tool. So now, the objective becomes:

$$\min_\theta(\mathbb{E}\left[-\log p_\theta(y \mid x)\right] + \mathbb{E}_{w<t}\left[\mathrm{KL}(p_{\theta_p}(\cdot \mid x_{<t})\|p_\theta(\cdot \mid x_{<t}))\right]). \tag{8}$$

Together, this objective is saying 1) increase the likelihood of the finetuning data, while 2) don't forget what model has learned in pretraining.

Perhaps unintuitively, the strongest intuition for "not straying too far from the pretrained parameters" may come from the simple intuition of a **small learning rate** during gradient descent. When finetuning pretrained models, we almost always use a small learning rate, such that each gradient descent step doesn't take us very far (in L_2 space) from the pretrained parameters. It is sometimes said that this *implicitly* penalizes the optimization process from exploring too far away from the pretrained parameters. For practitioners, one can take away that figuring out the right learning rate is often one of the more important discrete searches to perform when trying to improve finetuning.

Parameter-efficient finetuning

In pretraining and in finetuning so far, we've considered every learnable parameter of our network to be changeable independently when we perform stochastic gradient descent. However, we can also change just a subset of the parameters, a technique known as **parameter-efficient finetuning**.

We'll go over two popular methods for performing parameter-efficient finetuning: **prefix-tuning** and **low-rank adaption (LoRA)**.

In prefix-tuning, a set of learnable prefix vectors $\{f_1, f_2, \ldots, f_k\}$, where each $f_i \in \mathbb{R}^d$ and $f_i \sim \mathcal{U}(-\epsilon, \epsilon)$, is prepended to the input sequence as additional context for the Transformer blocks. Strictly speaking, if these learnable vectors are placed only at the input layer, this was originally called prompt-tuning (Lester et al., 2021), while if they are placed at every layer of the Transformer, this was originally called prefix-tuning (Li and Liang, 2021). We'll just go over the version with a single layer of learnable prefix parameters.

All parameters of the pretrained model are frozen, and only the prefix parameters are optimized to minimize the loss. Formally,

$$\min_{\{f_1, f_2, \ldots, f_k\}} \mathbb{E}_{(x,y)\sim\mathcal{D}} \left[\mathcal{L}(x, y)\right] \tag{9}$$

where $\mathcal{L}(x, y)$ is the loss as applied to our network while using the prefix vectors.

Intuitively, although we are not updating the **preexisting** model parameters, introducing the prefixes adapts hidden representations at each layer. As a result, the model's outputs can change even though its original parameters are fixed.

Low-Rank Adaptation, or LoRA (Hu et al., 2022), is one of the most popular methods for parameter-efficient finetuning. It works by introducing a learnable low-rank matrix as an update to the pretrained parameters. Suppose we have a weight matrix $W_Q \in \mathbb{R}^{d\times d}$, and we do not want to update all of its parameters. Instead, we introduce a rank-1 matrix $uv^\top$, where $u \in \mathbb{R}^d$ and $v \in \mathbb{R}^d$. Now, we have $\tilde{W_Q} = W_Q + uv^\top$. By introducing this rank-1 matrix, LoRA allows the model to only adjust its behavior along a small number of directions. In practice, we can introduce a matrix with rank up to k, where $k < d$. That is, $\tilde{W_Q} = W_Q + \sum_{j=1}^{k} u^{(j)} {v^{(j)}}^\top$, which allows the model to adapt along k independent directions in the parameter space:

$$\min_{u_1, v_1, u_2, v_2, \ldots, u_k, v_k} \mathbb{E}_{(x,y)\sim D} \left[\mathcal{L}(x, y)\right] \tag{10}$$

where $\mathcal{L}(x, y)$ is the loss as applied to our network while using the low-rank modified parameters.

References

Hu, E. J., yelong shen, Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. (2022). LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Lester, B., Al-Rfou, R., and Constant, N. (2021). The power of scale for parameter-efficient prompt tuning. In Moens, M.-F., Huang, X., Specia, L., and Yih, S. W.-t., editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3045–3059, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Li, X. L. and Liang, P. (2021). Prefix-tuning: Optimizing continuous prompts for generation. In Zong, C., Xia, F., Li, W., and Navigli, R., editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online. Association for Computational Linguistics.