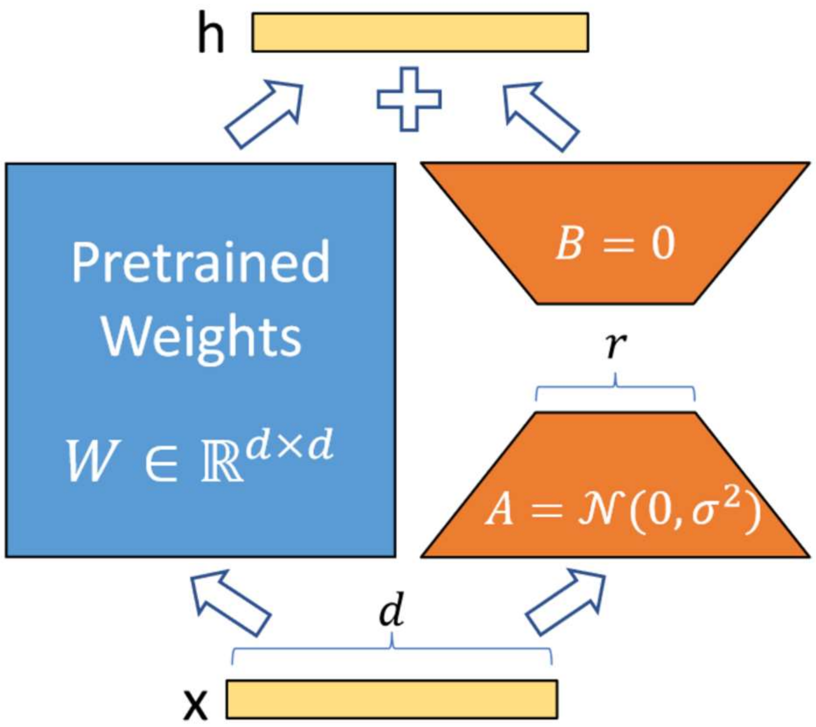
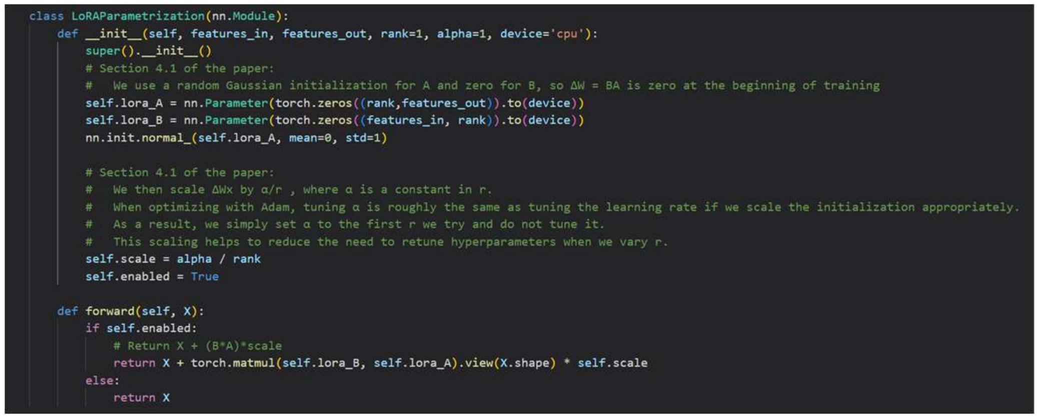
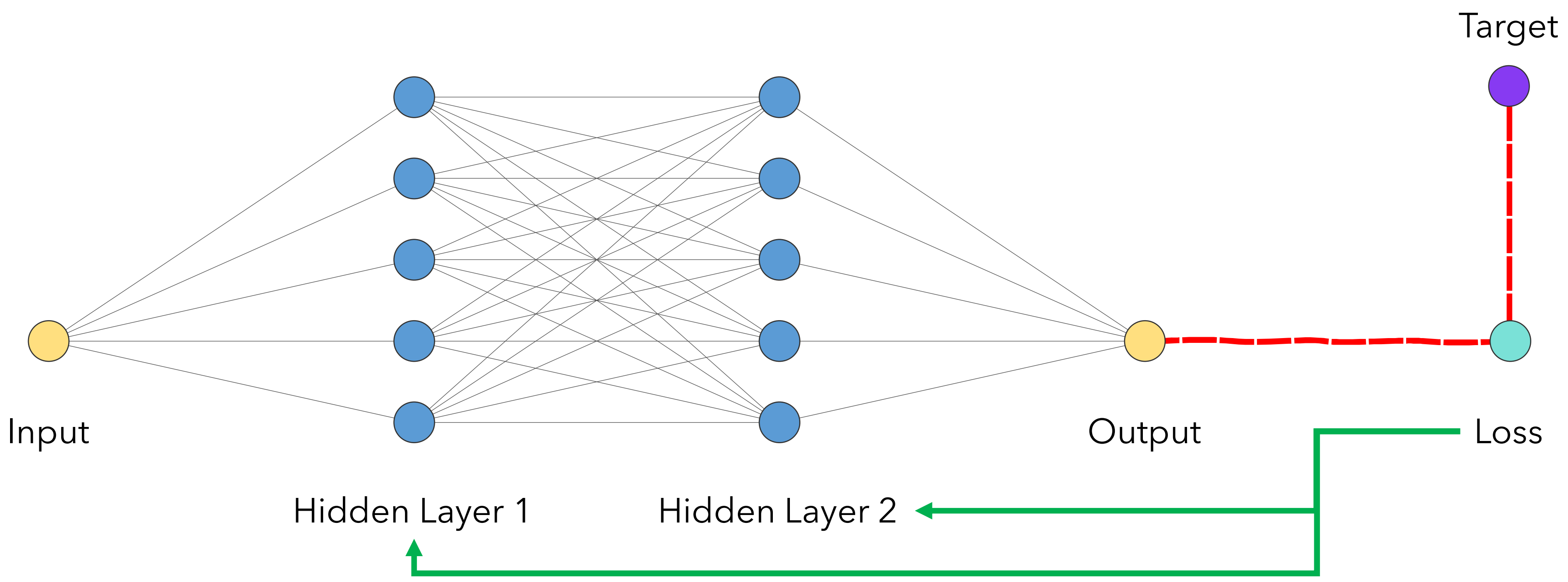

h
Pretrained Weights
$W \in \mathbb{R}^{d \times d}$
$B = 0$
r
$A = \mathcal{N}(0, \sigma^2)$
d
x


```python
class LoRAParametrization(nn.Module):
    def __init__(self, features_in, features_out, rank=1, alpha=1, device='cpu'):
        super().__init__()
        # Section 4.1 of the paper:
        #   We use a random Gaussian initialization for A and zero for B, so ΔW = BA is zero at the beginning of training
        self.lora_A = nn.Parameter(torch.zeros((rank,features_out)).to(device))
        self.lora_B = nn.Parameter(torch.zeros((features_in, rank)).to(device))
        nn.init.normal_(self.lora_A, mean=0, std=1)

        # Section 4.1 of the paper:
        #   We then scale ΔWx by α/r , where α is a constant in r.
        #   When optimizing with Adam, tuning α is roughly the same as tuning the learning rate if we scale the initialization appropriately.
        #   As a result, we simply set α to the first r we try and do not tune it.
        #   This scaling helps to reduce the need to retune hyperparameters when we vary r.
        self.scale = alpha / rank
        self.enabled = True

    def forward(self, X):
        if self.enabled:
            # Return X + (B*A)*scale
            return X + torch.matmul(self.lora_B, self.lora_A).view(X.shape) * self.scale
        else:
            return X
```

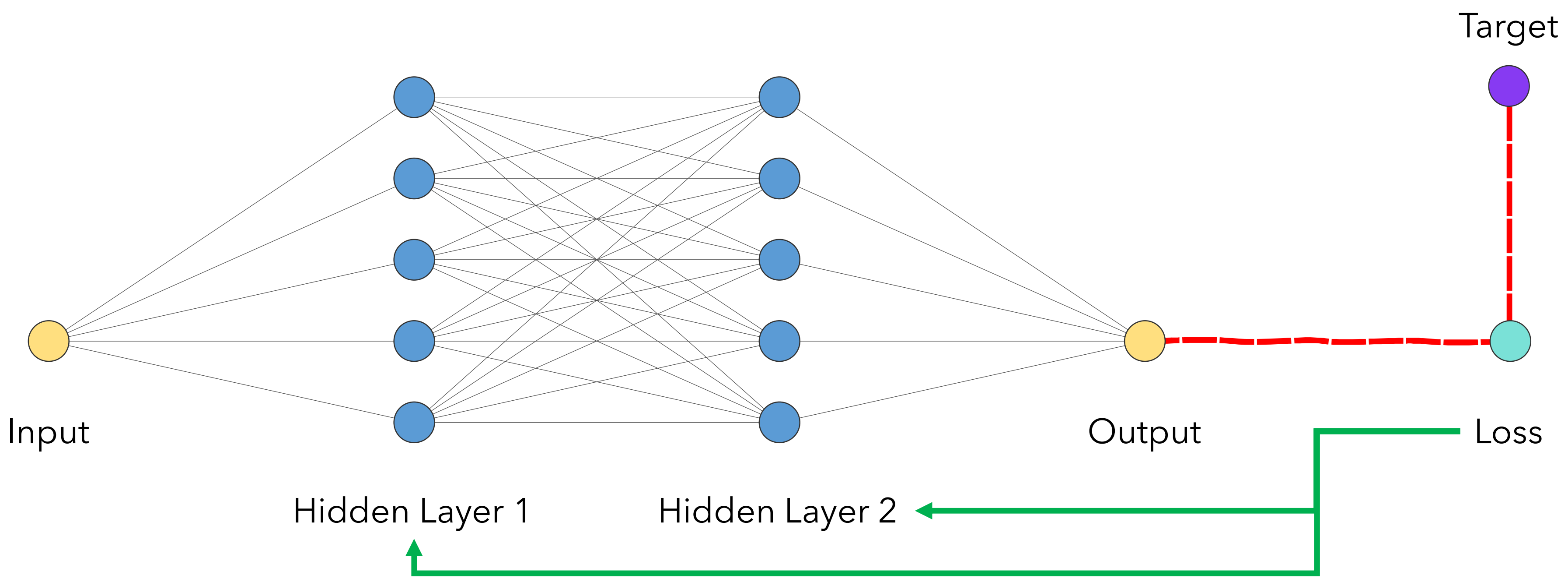
LoRA: Low-Rank Adaptation of Large Language Models

Umar Jamil

How do neural networks work?



Fine Tuning

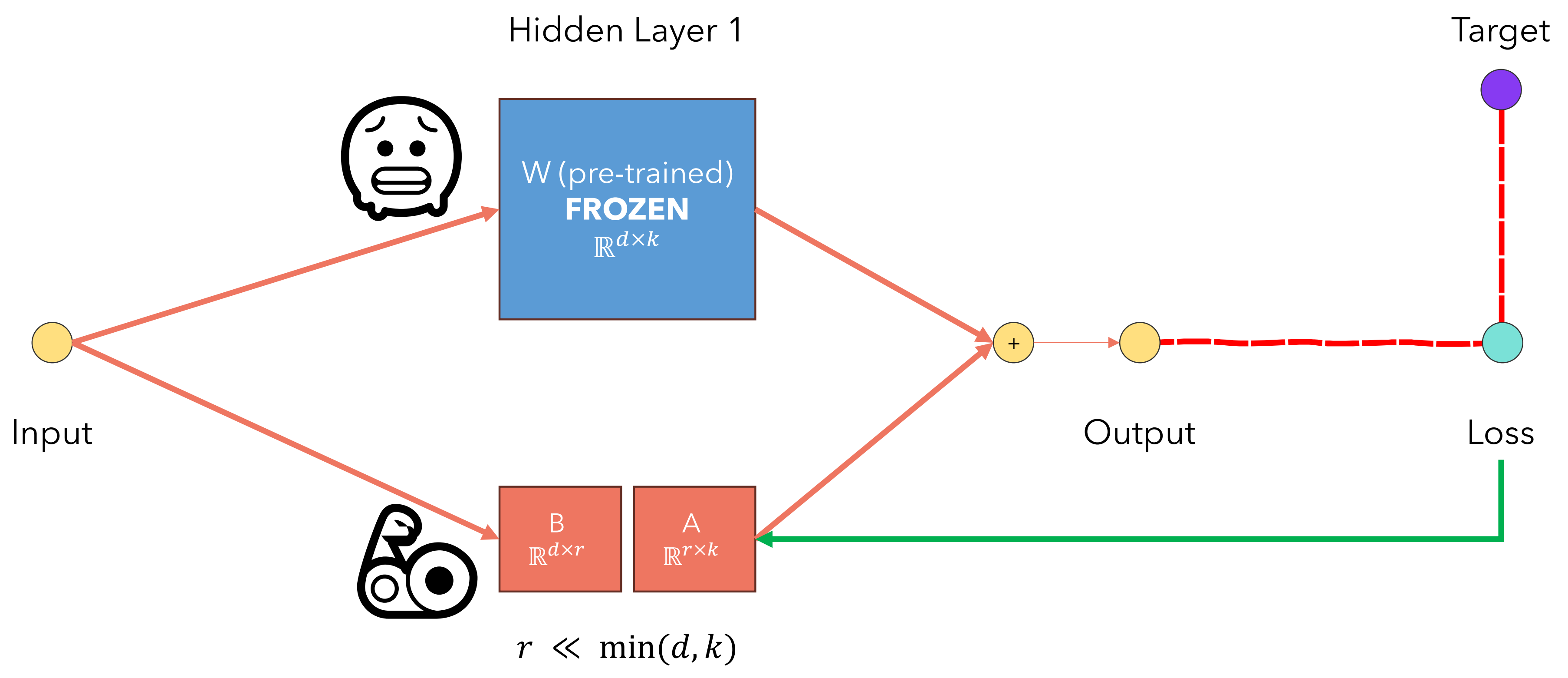


Fine tuning means training a pre-trained network on new data to improve its performance on a specific task. For example, we may fine-tune a LLM that was trained on many programming languages and fine-tune it for a new dialect of SQL.

Problems with fine-tuning

1. We must train the full network, which it computationally expensive for the average user when dealing with Large Language Models like GPT
2. Storage requirements for the checkpoints are expensive, as we need to save the entire model on the disk for each checkpoint. If we also save the optimizer state (which we usually do, then the situation gets worse!)
3. If we have multiple fine-tuned models, we need to reload all the weights of the model every time we want to switch between them, which can be expensive and slow. For example, we may have a model fine-tuned for helping users write SQL queries and one model for helping users write Javascript code.

Introducing LoRA

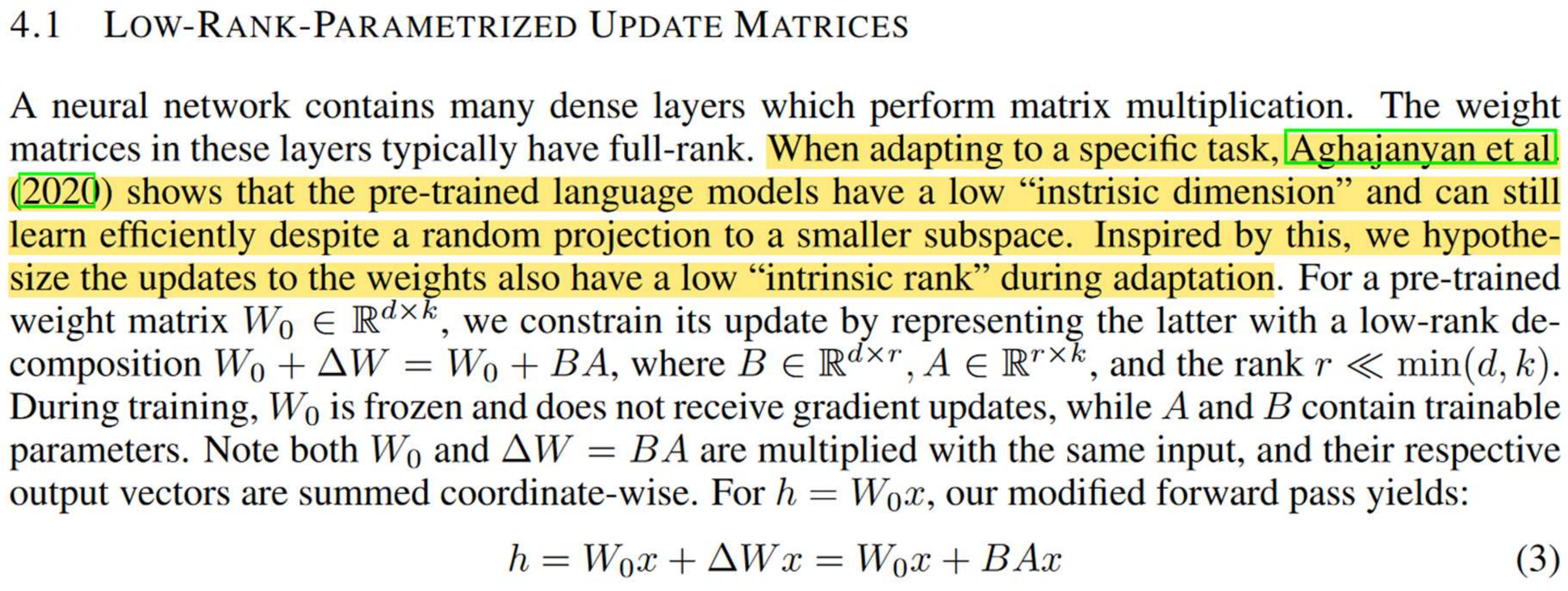


Umar Jamil - https://github.com/hkproj/pytorch-lora

What are the benefits?

1. Less parameters to train and store: if $d = 1000, k = 5000, (d \times k) = 5{,}000{,}000$; using $r = 5$, we get $(d \times r) + (r \times k) = 5{,}000 + 25{,}000 = 30{,}000$. Less than 1% of the original.
2. Less parameters = less storage requirements
3. Faster backpropagation, as we do not need to evaluate the gradient for most of the parameters
4. We can easily switch between two different fine-tuned model (one for SQL generation and one for Javascript code generation) just by changing the parameters of the A and B matrices instead of reloading the W matrix again.

Why does it work?



4.1 LOW-RANK-PARAMETRIZED UPDATE MATRICES

A neural network contains many dense layers which perform matrix multiplication. The weight matrices in these layers typically have full-rank. When adapting to a specific task, Aghajanyan et al. (2020) shows that the pre-trained language models have a low "instrisic dimension" and can still learn efficiently despite a random projection to a smaller subspace. Inspired by this, we hypothesize the updates to the weights also have a low "intrinsic rank" during adaptation. For a pre-trained weight matrix $W_0 \in \mathbb{R}^{d \times k}$, we constrain its update by representing the latter with a low-rank decomposition $W_0 + \Delta W = W_0 + BA$, where $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, and the rank $r \ll \min(d, k)$. During training, W_0 is frozen and does not receive gradient updates, while A and B contain trainable parameters. Note both W_0 and $\Delta W = BA$ are multiplied with the same input, and their respective output vectors are summed coordinate-wise. For $h = W_0 x$, our modified forward pass yields:

$$h = W_0 x + \Delta W x = W_0 x + BAx \tag{3}$$

It basically means that the W matrix of a pretrained model contains many parameters that convey the same information as others (so they can be obtained by a combination of the other weights); This means we can get rid of them without decreasing the performance of the model. This kind of matrices are called rank-deficient (they do not have full-rank).

A brief tutorial on the rank of a matrix…

Thanks for watching!
Don’t forget to subscribe for more amazing content on AI and Machine Learning!