Training the Deep Linear Network

Dylan Hu

March 23, 2023

Contents

1 Introduction

With the expansion of computing capabilities in the past decade, deep approaches to machine learning formulated decades prior have finally become tractable. Moreover, with the abundance and accessibility of data as well as more complex and powerful model architectures, the impact and potential of deep learning and its applications have gained significant attention (pun intended).

In just the past few months, accessible tools powered by deep learning such as ChatGPT have catapulted deep learning into the mainstream. While exciting new developments in deep learning seem to arise each week, we take a step back and analyze some of the fundamental mathematical principles behind deep learning.

Arora, Cohen, and Hazan [1] postulate that increased model *depth* not only leads to greater expressiveness, but also that the *overparameterization* leads to implicit acceleration of the optimization.

In this report, we investigate and demonstrate the acceleration effect of overparameterization on the optimization of the deep linear network. We also explore further the mathematical basis behind overparameterization through empirical analysis.

2 Method

We build the deep linear network (DLN) as formulated by Arora, Cohen, and Hazan [1]. The DLN is a deep (multi-layer) neural network of N linear layers of size $d \times d$ without activation functions. We therefore may represent the network as a chain of N square $d \times d$ matrices:

$$\boldsymbol{W} = (W_N, W_{N-1}, \ldots, W_1)$$

The observable of the DLN is the product of the matrices in the chain:

$$W := \pi(\boldsymbol{W}) = W_N W_{N-1} \ldots W_1$$

We investigate the acceleration effect of overparameterization by comparing the optimization dynamics of the DLN to that of a single-layer linear network of size $d \times d$, where the chain $\boldsymbol{W}$ reduces to the observable W and thus can be considered an end-to-end matrix W.

2.1 Model

We implement the DLN model in PyTorch:

```
class DLN(nn.Module):
    def __init__(self, d: int, N: int):
        super().__init__()
        self.d = d
        self.N = N
        self.params = nn.ParameterList(
            [nn.Parameter(torch.randn((d, d))) for _ in range(N)])

    def forward(self) -> Tensor:
        x = torch.eye(self.d)
        for param in self.params:
            x = param @ x
        return x
```

2.2 Optimization

In investigating the effect of overparameterization on the optimization of the DLN, we consider the optimization problem of matrix completion. Given a random target $d \times d$ matrix Φ, the optimization objective is to find $\boldsymbol{W}$ such that the observable W has diagonal entries that match those of Φ.

We minimize the following energy functions:

$$E(\boldsymbol{W}) = \frac{1}{2} \left\| \operatorname{diag}(\Phi - \pi(\boldsymbol{W})) \right\|_2^2, \quad F(W) = \frac{1}{2} \left\| \operatorname{diag}(\Phi - W) \right\|_2^2$$

Note that we distinguish $E(\boldsymbol{W})$ from $F(W)$, as they have different domains and induce different gradient dynamics for optimization.

We denote optimization on the chain of matrices $\boldsymbol{W}$ as operating on the "upstairs", whereas optimization on the end-to-end matrix W is operating on the "downstairs".

Training on the upstairs evolves each of the matrices of $\boldsymbol{W}$ according to the gradient flow

$$\dot{W}_i = -\nabla_{W_i} E(\boldsymbol{W})$$

whereas training on the downstairs evolves the end-to-end matrix W by the differential equation

$$\dot{W} = -\frac{1}{N}\sum_{j=1}^{N} \left(WW^T\right)^{\frac{N-j}{N}} \partial_W F(W) \left(W^T W\right)^{\frac{j-1}{N}} \tag{1}$$

where $\partial_W F(W) = -\text{diag}(\Phi - W)$.

2.2.1 Loss function

We train the DLN upstairs using supervision imposed by a mean-squared-error loss function for which minimization is equivalent to that for the energy function $E(\boldsymbol{W})$:

$$\mathcal{L}_{\text{upstairs}} = \text{MSE}\left(\text{diag}(\Phi), \text{diag}(\pi(\boldsymbol{W}))\right) = \frac{1}{d}\sum_{i=1}^{d} \text{diag}(\Phi - \pi(\boldsymbol{W}))_i^2$$

Although this is not the loss function for the downstairs training, we return this loss for both upstairs and downstairs training for the sake of comparison:

$$\mathcal{L}_{\text{downstairs}} = \text{MSE}\left(\text{diag}(\Phi), \text{diag}(W)\right) = \frac{1}{d}\sum_{i=1}^{d} \text{diag}(\Phi - W)_i^2$$

2.2.2 Training upstairs

The implementation of the upstairs training uses PyTorch's automatic differentiation to compute the gradient of the loss function with respect to the parameters of the DLN. We then update the parameters of the DLN using a built-in optimizer. We experiment with a variety of optimizers, the results of which are discussed in Section **??**.

```
def upstairs_func(model: DLN,
                  target: Tensor,
                  optimizer: torch.optim.Optimizer,
                  scheduler: Optional[torch.optim.lr_scheduler.LRScheduler]
                    = None):
    def upstairs(train: bool) -> Tensor:
        model.train(train)
        with torch.set_grad_enabled(train):
            optimizer.zero_grad()
            W = model.forward()
            if train:
                loss = F.mse_loss(torch.diag(W), torch.diag(target))
                loss.backward()
                optimizer.step()
                if scheduler:
                    scheduler.step()
                return loss
        return W
    return upstairs

upstairs_model = DLN(d, N) # by default, d = 2, N = 3
optimizer = torch.optim.SGD(upstairs_model.parameters(), lr)
target = torch.randn((d, d))

iteration = 0
upstairs_losses = []

upstairs = upstairs_func(upstairs_model, target, optimizer)
while (loss := upstairs(train=True)) > 1e-7:
    upstairs_losses.append(loss.item())
    iteration += 1
```

2.2.3 Training downstairs

The downstairs training does not use PyTorch's automatic differentiation but instead uses manual gradient computation and composition according to Eq. 1.

```
def downstairs_func(model: DLN, target: Tensor, lr: float, N: int):
    def downstairs() -> Tensor:
        model.train(False)
        with torch.no_grad():
            W = model.forward()
            d_F = torch.diag_embed(torch.diag(W - target))
            WWT = W @ W.T
            WTW = W.T @ W
            d_W = torch.zeros_like(W)
            for i in range(1, N + 1):
```

```
                d_W = d_W + \
                    scipy.linalg.fractional_matrix_power(
                        WWT, (N - i) / N) @ \
                    np.array(d_F) @ \
                    scipy.linalg.fractional_matrix_power(
                        WTW, (i - 1) / N)
            d_W = d_W * torch.ones_like(W) / N
            W = W - lr * d_W
            model.params[0] = W
            return F.mse_loss(torch.diag(W), torch.diag(target))
    return downstairs
```

2.3 Further exploration

2.4 Network width

We experiment with $d = 4, d = 6$ network widths with a fixed $N = 3$. Results are shown in Figure 4.

2.4.1 Network depth

We experiment with $N > 3$ network depths with a fixed $d = 2$. Results are shown in Figure 5.

2.4.2 Optimizers

For the results in Section 3.1, we use stochastic gradient descent (SGD) for upstairs. However, we also experiment with other optimizers, including Adam and RMSprop. For all tests, we use $d = 2, N = 3$. The results are shown in Figure 6.

2.4.3 Learning rate

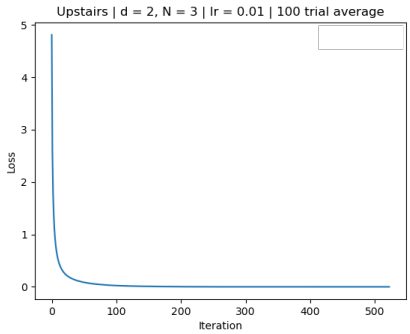
We also experiment with different learning rates with SGD for both upstairs and downstairs training. As expected, we find that the learning rate has a significant effect on the performance of the DLN, with higher learning rates amenable to shallow networks but not deep networks, where smaller learning rates are required.

This reflects the intuition that as the network becomes deeper, the manifold of possible solutions becomes more complex, and the DLN must be more careful in its exploration of the manifold.

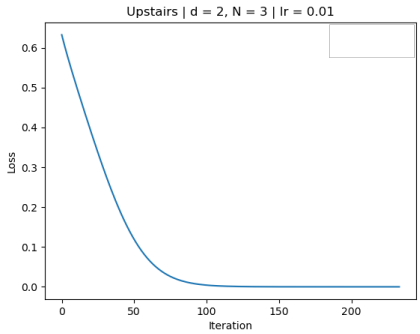
The results are shown in Figure 7.

3 Results

3.1 Implicit acceleration

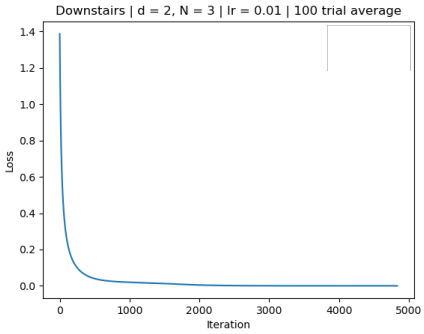


(a) 100 trial average

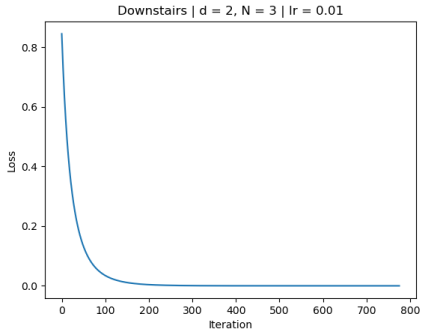


(b) a particular trajectory

Figure 1: Training upstairs with $d = 2, N = 3$, and learning rate = 0.01



(a) 100 trial average



(b) a particular trajectory

Figure 2: Training downstairs with $d = 2, N = 3$, and learning rate = 0.01

3.1.1 Training upstairs

With $d = 2, N = 3$, and learning rate = 0.01, we are able to successfully train and consistently converge the DLN upstairs using the SGD optimizer. Measured across 100 trials, it takes an average of 204.58 iterations with

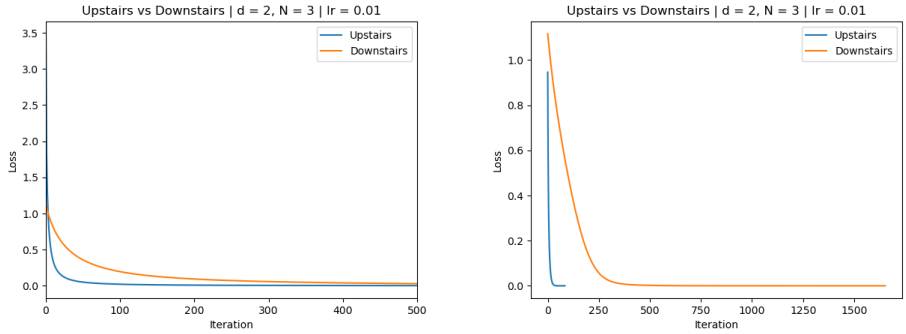
standard deviation 101.03 iterations to converge. The average trajectory and a particular trajectory are shown in Figure 1.

3.1.2 Training downstairs

With $d = 2, N = 3$, and learning rate $= 0.01$, we are also able to successfully train and consistently converge downstairs. However, we find that the convergence is much slower and the trajectory is much more varied than upstairs. Measured across 100 trials, it takes an average of 1410.79 iterations with standard deviation of 1019.89 iterations to converge. The average trajectory and a particular trajectory are shown in Figure 2.

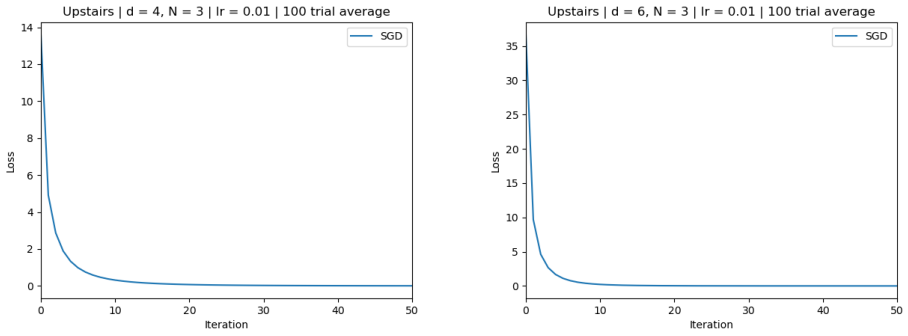
3.1.3 Comparison

We observe that the convergence of the upstairs occurs more quickly and in a more predictable fashion than that of the downstairs, supporting the claim that overparameterization can lead to implicit acceleration. We plot average and particular trajectories for both upstairs and downstairs in Figure 3.



(a) 100 trial average (b) a particular trajectory

Figure 3: Training upstairs compared to downstairs with $d = 2, N = 3$, and learning rate $= 0.01$



(a) 100 trial average (b) a particular trajectory

Figure 4: $d = 2$ and $d = 6$ for $N = 3$ and learning rate $= 0.01$

3.2 Further exploration

3.2.1 Network width

Only $N = 5$ resulted in reliable enough convergence across 100 trials; the results are shown in Figure 5. With $N > 5$, the DLN is able to converge, but the convergence becomes less and less reliable.

We hypothesize that this is the vanishing or exploding gradient problem, which is a common problem in deep learning. The DLN is able to converge with $N = 5$ because the gradient cannot compound to the point where it becomes too large or too small. We therefore experiment by adding various activations between each layer of the DLN.

With ReLU activations, the model fails to converge at $N = 7$ and becomes stuck at a local minimum. With sigmoid activations, the model also fails to converge reliably but does consistently reach better local minima than ReLU. Lastly, with the tanh activation, the model does indeed reliably converge at $N = 7$, and it seems to reliably converge up until at least $N = 80$! Greater N could not be tested for eventual convergence due to the increased runtime.

We leave a principled explanation for the success of the tanh activation in this setting as future work.

3.2.2 Network depth

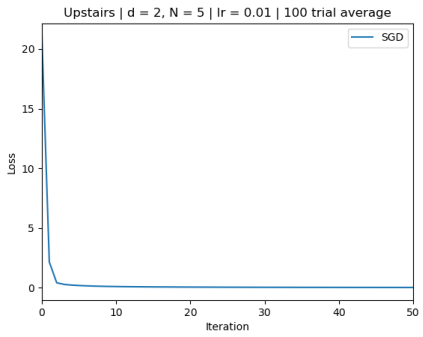


Figure 5: N = 5

In the case of $d = 4$, the mean number of iterations measured across 100 trials is 142.34 iterations with standard deviation 159.37. In the case of $d = 6$, the mean number of iterations measured across 100 trials is 79.03 iterations with standard deviation 44.76. In general, we find that increasing network width results in faster convergence, as there are more parameters to simultaneously tune in order to reduce the mean error. The results are shown in Figure 5.

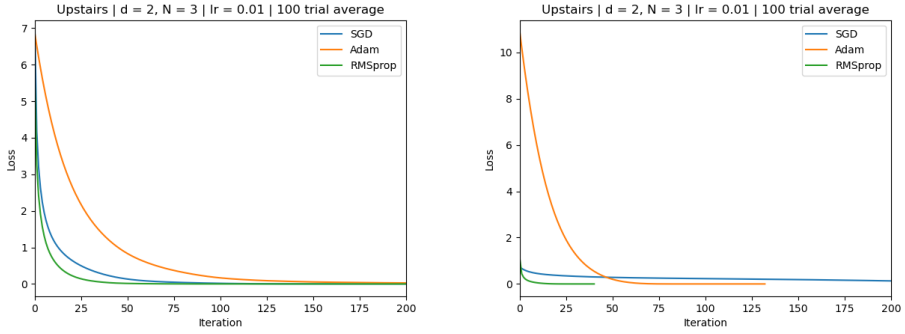
3.2.3 Optimizers

We find that RMSprop performs best and slightly better than SGD, while Adam performs worst. The results are shown in Figure 6.

A potential intuition for these results is that RMSprop and SGD are both first-order methods, while Adam is a first-order and second-order method. As the gradient flow is a first-order differential equation, it is possible that first-order optimizers are better suited for this task.

Another potential intuition is that Adam may simply be too agressive and tend to overshoot. In fact, we notice that trajectories with adam are significantly less smooth.

A more principled investigation of the effects of different optimizers is left for future work.

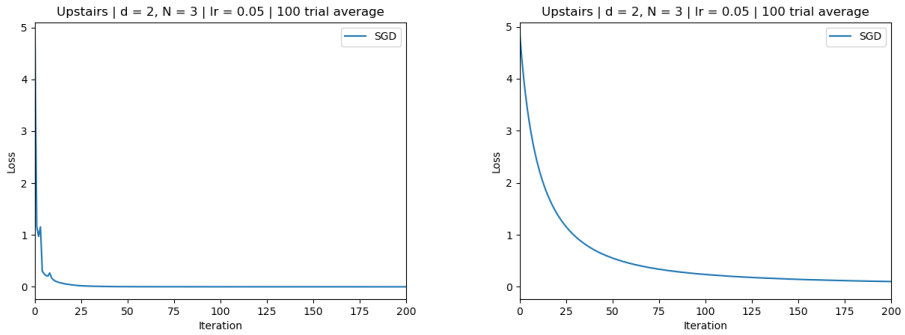


(a) 100 trial average (b) a particular trajectory

Figure 6: SGD, Adam, and RMSprop compared with $d = 2, N = 3$, and learning rate = 0.01

3.2.4 Learning rate

With a learning rate of 0.05, the mean number of iterations for convergence is 51.67 with a standard deviation of 37.75. With a learning rate of 0.001, the mean number of iterations for convergence is 4811.72 with a standard deviation of 4742.84. We find that a learning rate of 0.05 is too large for



(a) Learning rate = 0.05 (b) Learning rate = 0.001

Figure 7: SGD with learning rates 0.05 and 0.001 with $d = 2, N = 3$

even $N = 5$ to converge, whereas a learning rate of 0.01 failed to converge five out of 100 trials, and a learning rate of 0.001 consistently enabled convergence.

This matches intuition, as deeper networks induce a more complex loss landscape, and a larger learning rate can lead to a more unstable trajectory. We hypothesize that the fragility in these experiments is caused by the lack of activation functions which are essential for avoiding vanishing or exploding gradients. As such, the DLN efficiently demonstrates both the effect of learning rate on deep networks and the importance of activation functions.

References

[1] Sanjeev Arora, Nadav Cohen, and Elad Hazan. On the optimization of deep networks: Implicit acceleration by overparameterization, 2018.