Gradient Descent Provably Optimizes Over-Parameterized Neural Networks

Yuanxin Zhang yz6201
2023.5.10

Schedule

- Background
- Comparison with Previous Methods
- Continuous Time Analysis
- Discrete Time Analysis
- Numerical Experiments
- Conclusion and Discussion

Background

- **Phenomenon:** randomly initialized first order methods(gradient descent) can achieve zero training loss even though the objective function is non-convex and non-smooth.

- **Previous Assumption:** The reason why a neural network can fit all training labels is that the neural network is over-parameterized so that there must exists one such neural network of this architecture to fit all training data.

- **Current Question:**
  - The existence does not imply why the network found by a **randomly initialized** first order method can fit all the data.
  - The traditional analysis from convex optimization can't solve the problem with non-smooth, non-convex optimization objective function.

Background

- In this paper, we demystify on an **two-layer neural networks with rectified linear unit (ReLU) activation.**

$$f(\mathbf{W}, \mathbf{a}, \mathbf{x}) = \frac{1}{\sqrt{m}} \sum_{r=1}^{m} a_r \sigma\left(\mathbf{w}_r^\top \mathbf{x}\right) \tag{1}$$

- We focus on the empirical risk minimization problem with a quadratic loss. (MSE) Given the dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$

$$L(\mathbf{W}, \mathbf{a}) = \sum_{i=1}^{n} \frac{1}{2} \left(f(\mathbf{W}, \mathbf{a}, \mathbf{x}_i) - y_i\right)^2. \tag{2}$$

- We fix the second layer and apply gradient descent (GD) to optimize the first layer

$$\mathbf{W}(k+1) = \mathbf{W}(k) - \eta \frac{\partial L(\mathbf{W}(k), \mathbf{a})}{\partial \mathbf{W}(k)}. \tag{3}$$

observations

- **Previous work vs current research:**
  1. dynamics of the parameter (**W**)
  2. dynamics of prediction space (**u(t**)), since it's non-convex and non-smooth; So we analyze the property of the Gram matrix.

- **Observations:**
  - **Initialization phase:** Gram matrix has a lower bounded least eigenvalue.
  - **Iteration phase:** the Gram matrix is close to the initialization phase
  - **Gram matrix convergence:** if most of the activation patterns $(\mathbb{I}\left\{\mathbf{w}_r^\top \mathbf{x}_i \geq 0\right\})$ don't change, then the Gram matrix is close to its initialization.
  - **Weight convergence:** overparameterization, random initialization, and the linear convergence restrict every weight vector to be close to its initialization.

Continuous-Time Analysis VS Discrete-Time Analysis

- **Continuous-Time Analysis**
  - Model optimization as a continuous trajectory in time.
  - Described by a system of differential equations that govern the evolution of the optimization variables over time
  - Suitable for studying the long-term behavior of optimization algorithms.
  - Can be computationally expensive.

- **Discrete-Time Analysis**
  - Model optimization as a sequence of discrete updates.
  - Described by a sequence of equations that govern the update rule for the optimization variables.
  - Suitable for studying the short-term behavior of optimization algorithms.
  - Can be computationally efficiently

Continuous Time Analysis

- Here we consider about a significant ODE defined by:

$$\frac{d\mathbf{w}_r(t)}{dt} = -\frac{\partial L(\mathbf{W}(t), \mathbf{a})}{\partial \mathbf{w}_r(t)}$$

for r ∈ [m]. We denote $u_i(t) = f(W(t),a,x_i)$ the prediction on input x_i at time t and we let $u(t)=(u_1(t),...,u_n(t)) \in R^n$ be the prediction vector at time t. We state our main assumption.

Continuous Time Analysis

Assumption 3.1. *Define matrix* $\mathbf{H}^{\infty} \in \mathbb{R}^{n \times n}$ *with* $\mathbf{H}_{ij}^{\infty} = \mathbb{E}_{\mathbf{w} \sim N(\mathbf{0}, \mathbf{I})} \left[\mathbf{x}_i^{\top} \mathbf{x}_j \mathbb{I} \left\{ \mathbf{w}^{\top} \mathbf{x}_i \geq 0, \mathbf{w}^{\top} \mathbf{x}_j \geq 0 \right\} \right]$. *We assume* $\lambda_0 \triangleq \lambda_{\min} \left(\mathbf{H}^{\infty} \right) > 0$.

To justify this assumption, the following theorem shows if no two inputs are parallel the least eigenvalue is strictly positive.

Theorem 3.1. *If for any* $i \neq j$, $\mathbf{x}_i \nparallel \mathbf{x}_j$, *then* $\lambda_0 > 0$.

Theorem 3.2 (Convergence Rate of Gradient Flow)**.** *Suppose Assumption 3.1 holds and for all* $i \in [n]$, $\|\mathbf{x}_i\|_2 = 1$ *and* $|y_i| \leq C$ *for some constant* C. *Then if we set the number of hidden nodes* $m = \Omega \left(\frac{n^6}{\lambda_0^4 \delta^3} \right)$ *and we i.i.d. initialize* $\mathbf{w}_r \sim N(\mathbf{0}, \mathbf{I})$, $a_r \sim \text{unif} \left[\{-1, 1\} \right]$ *for* $r \in [m]$, *then with probability at least* $1 - \delta$ *over the initialization, we have*

$$\|\mathbf{u}(t) - \mathbf{y}\|_2^2 \leq \exp(-\lambda_0 t) \|\mathbf{u}(0) - \mathbf{y}\|_2^2 .$$

Continuous Time Analysis

The proof of Theorem 2 is significant. So we have a rigorous proof here.

- **Step 1:** prove the dynamics of prediction space (u(t)) is determined by the property of Gram Matrix.

$$\frac{d}{dt}u_i(t) = \sum_{r=1}^{m} \langle \frac{\partial f(\mathbf{W}(t), \mathbf{a}, \mathbf{x}_i)}{\partial \mathbf{w}_r(t)}, \frac{d\mathbf{w}_r(t)}{dt} \rangle$$
$$= \sum_{j=1}^{n} (y_j - u_j) \sum_{r=1}^{m} \langle \frac{\partial f(\mathbf{W}(t), \mathbf{a}, \mathbf{x}_i)}{\partial \mathbf{w}_r(t)}, \frac{\partial f(\mathbf{W}(t), \mathbf{a}, \mathbf{x}_j)}{\partial \mathbf{w}_r(t)} \rangle \triangleq \sum_{j=1}^{n} (y_j - u_j) \mathbf{H}_{ij}(t)$$

where $\mathbf{H}(t)$ is an $n \times n$ matrix with (i, j)-th entry

$$\mathbf{H}_{ij}(t) = \frac{1}{m} \mathbf{x}_i^\top \mathbf{x}_j \sum_{r=1}^{m} \mathbb{I}\left\{ \mathbf{x}_i^\top \mathbf{w}_r(t) \geq 0, \mathbf{x}_j^\top \mathbf{w}_r(t) \geq 0 \right\}.$$

With this $\mathbf{H}(t)$ matrix, we can write the dynamics of predictions in a compact way:

$$\frac{d}{dt}\mathbf{u}(t) = \mathbf{H}(t)(\mathbf{y} - \mathbf{u}(t)).$$

Continuous Time Analysis

The proof of Theorem 2 is significant. So we have a rigorous proof here.

- **Step 2:** Show that when m goes to infinity, u(t) is determined by the Gram Matrix.

Lemma 3.1. *If* $m = \Omega\left(\frac{n^2}{\lambda_0^2}\log\left(\frac{n}{\delta}\right)\right)$, *we have with probability at least* $1-\delta$, $\|\mathbf{H}(0) - \mathbf{H}^{\infty}\|_2 \leq \frac{\lambda_0}{4}$ *and* $\lambda_{\min}(\mathbf{H}(0)) \geq \frac{3}{4}\lambda_0$.

Lemma 3.2. *If* $\mathbf{w}_1, \ldots, \mathbf{w}_m$ *are i.i.d. generated from* $N(\mathbf{0}, \mathbf{I})$, *then with probability at least* $1-\delta$, *the following holds. For any set of weight vectors* $\mathbf{w}_1, \ldots, \mathbf{w}_m \in \mathbb{R}^d$ *that satisfy for any* $r \in [m]$, $\|\mathbf{w}_r(0) - \mathbf{w}_r\|_2 \leq \frac{c\delta\lambda_0}{n^2} \triangleq R$ *for some small positive constant c, then the matrix* $\mathbf{H} \in \mathbb{R}^{n\times n}$ *defined by*

$$\mathbf{H}_{ij} = \frac{1}{m}\mathbf{x}_i^{\top}\mathbf{x}_j \sum_{r=1}^{m} \mathbb{I}\left\{\mathbf{w}_r^{\top}\mathbf{x}_i \geq 0, \mathbf{w}_r^{\top}\mathbf{x}_j \geq 0\right\}$$

satisfies $\|\mathbf{H} - \mathbf{H}(0)\|_2 < \frac{\lambda_0}{4}$ *and* $\lambda_{\min}(\mathbf{H}) > \frac{\lambda_0}{2}$.

Lemma 3.3. *Suppose for* $0 \leq s \leq t$, $\lambda_{\min}(\mathbf{H}(s)) \geq \frac{\lambda_0}{2}$. *Then we have* $\|\mathbf{y} - \mathbf{u}(t)\|_2^2 \leq \exp(-\lambda_0 t)\|\mathbf{y} - \mathbf{u}(0)\|_2^2$ *and for any* $r \in [m]$, $\|\mathbf{w}_r(t) - \mathbf{w}_r(0)\|_2 \leq \frac{\sqrt{n}\|\mathbf{y}-\mathbf{u}(0)\|_2}{\sqrt{m}\lambda_0} \triangleq R'$.

Lemma 3.4. *If* $R' < R$, *we have for all* $t \geq 0$, $\lambda_{\min}(\mathbf{H}(t)) \geq \frac{1}{2}\lambda_0$, *for all* $r \in [m]$, $\|\mathbf{w}_r(t) - \mathbf{w}_r(0)\|_2 \leq R'$ *and* $\|\mathbf{y} - \mathbf{u}(t)\|_2^2 \leq \exp(-\lambda_0 t)\|\mathbf{y} - \mathbf{u}(0)\|_2^2$.

Continuous Time Analysis

Joint Training Both Layers

- The only difference is the ODE defined by:

$$\frac{d\mathbf{w}_r(t)}{dt} = -\frac{\partial L(\mathbf{W}(t), \mathbf{a}(t))}{\partial \mathbf{w}_r(t)} \text{ and } \frac{d\mathbf{w}_r(t)}{dt} = -\frac{\partial L(\mathbf{W}(t), \mathbf{a}(t))}{\partial \mathbf{a}_r(t)}$$

Theorem 3.3 (Convergence Rate of Gradient Flow for Training Both Layers). *Under the same assumptions as in Theorem 3.2, if we set the number of hidden nodes* $m = \Omega\left(\frac{n^6 \log(m/\delta)}{\lambda_0^4 \delta^3}\right)$ *and we i.i.d. initialize* $\mathbf{w}_r \sim N(\mathbf{0}, \mathbf{I})$, $a_r \sim \text{unif}\left[\{-1, 1\}\right]$ *for* $r \in [m]$, *with probability at least* $1 - \delta$ *over the initialization we have*

$$\|\mathbf{u}(t) - \mathbf{y}\|_2^2 \leq \exp(-\lambda_0 t) \|\mathbf{u}(0) - \mathbf{y}\|_2^2 .$$

Discrete Time Analysis

- **Purpose:** To analyze randomly initialized gradient descent with a constant positive step size converges to the global minimum at a linear rate.

Theorem 4.1 (Convergence Rate of Gradient Descent). *Under the same assumptions as in Theorem 3.2, if we set the number of hidden nodes* $m = \Omega\left(\frac{n^6}{\lambda_0^4 \delta^3}\right)$, *we i.i.d. initialize* $\mathbf{w}_r \sim N(\mathbf{0}, \mathbf{I})$, $a_r \sim \text{unif}\left[\{-1, 1\}\right]$ *for* $r \in [m]$, *and we set the step size* $\eta = O\left(\frac{\lambda_0}{n^2}\right)$ *then with probability at least* $1 - \delta$ *over the random initialization we have for* $k = 0, 1, 2, \ldots$

$$\|\mathbf{u}(k) - \mathbf{y}\|_2^2 \leq \left(1 - \frac{\eta \lambda_0}{2}\right)^k \|\mathbf{u}(0) - \mathbf{y}\|_2^2 .$$

Numerical Experiments

- **Purpose:** Utilize the synthetic data to corroborate the theoretical findings.

- **Initialization:** epoch = 100, learning rate = 0.1, n = 1000, d = 1000,
  y generated from a one-dimensional standard Gaussian distribution,
  m = 500, 1000, 2000, 4000, 8000

Numerical Experiments

```python
import numpy as np
import pandas as pd

# Set the random seed for reproducibility
np.random.seed(1234)

# Generate n=1000 data points from a d=1000 dimensional unit sphere
X = np.random.normal(size=(1000, 1000))
X /= np.linalg.norm(X, axis=1, keepdims=True)

# Generate labels from a one-dimensional standard Gaussian distribution
y = np.random.normal(size=1000)

# Convert the data and labels to a Pandas DataFrame
df = pd.DataFrame(data=X)
df['y'] = y

# Save the data and labels to a CSV file
df.to_csv('data.csv', index=False)

print("Data saved to 'data.csv'")
```

```
Data saved to 'data.csv'
```

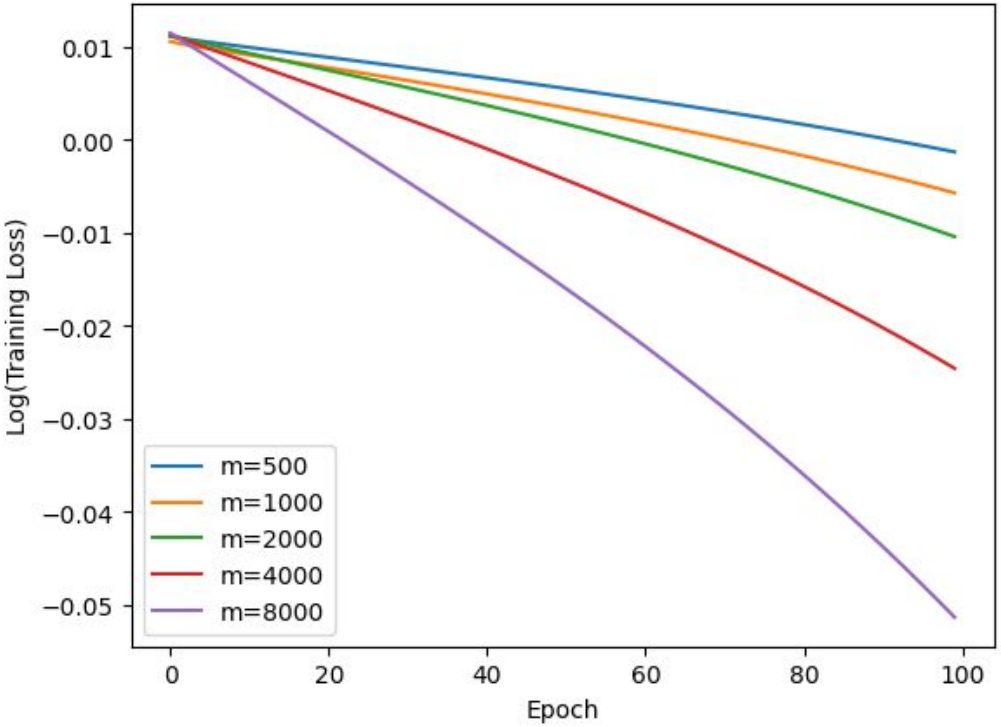
I try to simplify the synthetic data, so here we normalize all x, as well as randomizing data in terms of Gaussian Distributions.

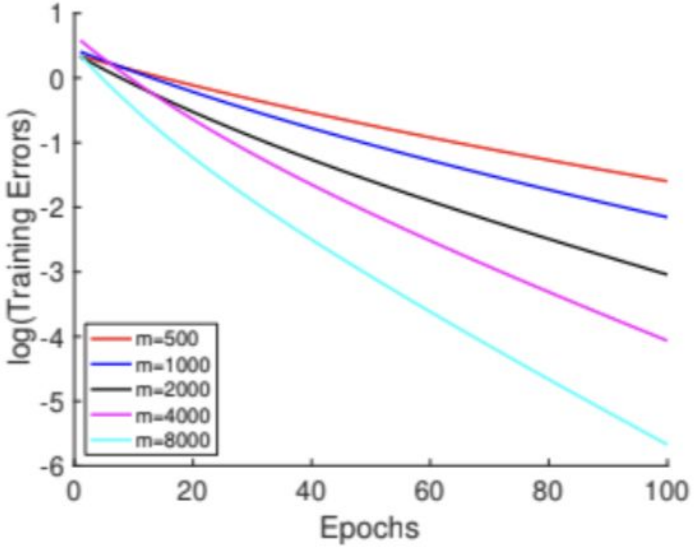
Numerical Experiments

```python
import torch
import torch.nn as nn
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# Load data
df = pd.read_csv("/content/drive/My Drive/ds&ml/data.csv")
X = df.iloc[:, :-1].values
y = df.iloc[:, -1].values

# Convert data and labels to tensors
X = torch.from_numpy(X).float()
y = torch.from_numpy(y).float()

# Define neural network architecture
class Net(nn.Module):
    def __init__(self, m):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(X.shape[1], m)
        self.fc2 = nn.Linear(m, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```python
criterion = nn.MSELoss()
# Train neural network for each value of m
m_list = [500, 1000, 2000, 4000, 8000]
epochs = 100
learning_rate = 0.1
train_loss_list = []
sign_diff_list = []
max_distance_list = []

for m in m_list:
    net = Net(m)
    optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
    train_loss = []
    sign_diff = []
    max_distance = 0.0
    for epoch in range(epochs):
        # Forward pass
        y_pred = net(X).squeeze()
        loss = criterion(y_pred,y)
        train_loss.append(loss.item())
                          (variable) y_pred: Any
        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    train_loss_list.append(train_loss)
new_net = Net(m)
new_net.load_state_dict(torch.load('saved_weights.pth'))
plt.figure(figsize=(10, 5))
for i, m in enumerate(m_list):
    plt.subplot(1, 2, 1)
    plt.plot(range(epochs), np.log(train_loss_list[i]), label=f"m={m}")
    plt.xlabel("Epoch")
    plt.ylabel("Log(Training Loss)")
    plt.legend()
torch.save(net.state_dict(), 'saved_weights.pth')
```

Comparison





(a) Convergence rates.

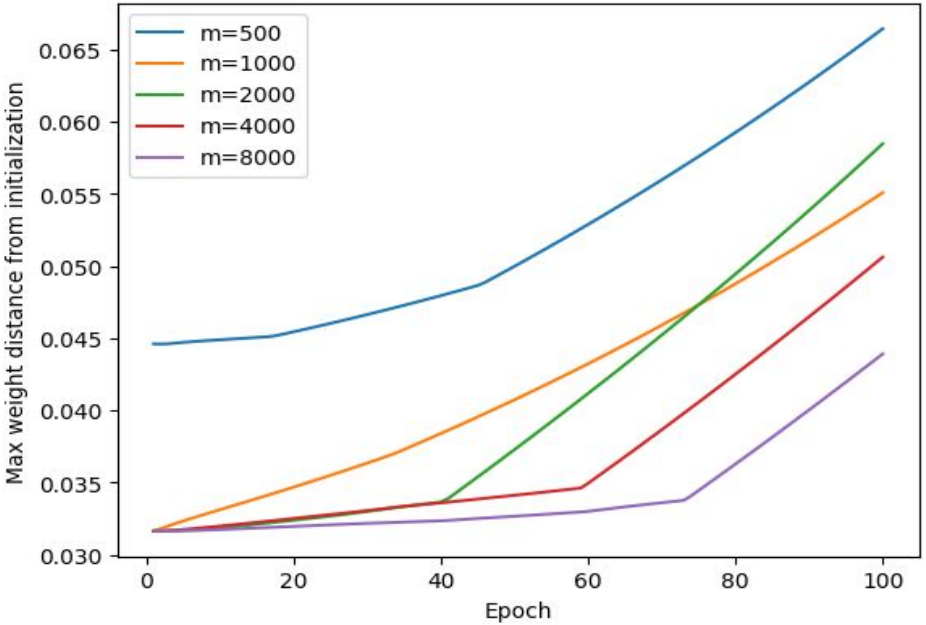
Numerical Experiment

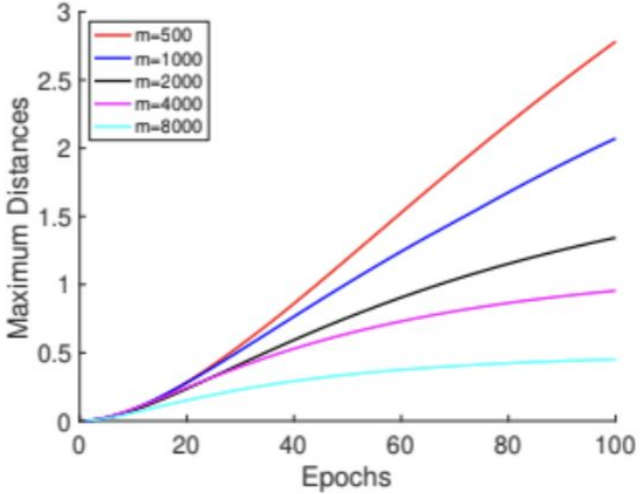
Last, we test the relation between the amount of over-parameterization and the maximum of the distances between weight vectors and their initializations. Formally, at a given iteration k, we check $\max_{r\in[m]} \|\mathbf{w}_r(k) - \mathbf{w}_r(0)\|_2$. This aims to verify Lemma 3.3 and Corollary 4.1.

```
    def max_weight_distance_from_initial(self):
        max_distance = 0
        for param in self.parameters():
            init_param = torch.zeros_like(param)
            distance = torch.max(torch.abs(param - init_param)).item()
            if distance > max_distance:
                max_distance = distance
        return max_distance
```

```
    # Calculate max weight distance at iteration k=50
    max_distance = net.max_weight_distance_from_initial()
    max_weight_distance[epoch] = max_distance  # append to array

max_weight_distance_list.append(max_weight_distance)
```

Comparison





(c) Maximum distances from initialization.

Numerical Experiments

parameterization affects the convergence rates. Second, we test the relation between the amount of over-parameterization and the number of pattern changes. Formally, at a given iteration k, we check $\frac{\sum_{i=1}^{m}\sum_{r=1}^{m}\mathbb{I}\left\{\text{sign}\left(\mathbf{w}_r(0)^\top \mathbf{x}_i\right)\neq\text{sign}\left(\mathbf{w}_r(k)^\top \mathbf{x}_i\right)\right\}}{mn}$ (there are mn patterns). This aims to verify Lemma 3.2.

```
# Define function to calculate sign difference percentage
def calc_sign_diff(net, X, m):
    w0 = net.fc1.weight.detach().numpy()
    w = net.fc1.weight.detach().clone()
    sign_diff_count = 0
    for i in range(X.shape[0]):
        for r in range(m):
            if np.sign(w0[r,:].T.dot(X[i])) != np.sign(w[r,:].T.dot(X[i])):
                sign_diff_count += 1
    return sign_diff_count / (m * X.shape[0])
```

```
    # Calculate sign difference percentage
    if epoch % 10 == 0:
        sign_diff_pct = calc_sign_diff(net, X, m)
        sign_diff.append(sign_diff_pct)


sign_diff_list.append(sign_diff)
```

Conclusion and Discussion

- **Conclusion:** With over-parameterization, gradient descent provable converges to the global minimum of the empirical loss at a linear convergence rate.
- **Key proof:** to show the over-parameterization makes Gram matrix remain positive definite for all iterations, which in turn guarantees the linear convergence.
- **Future Directions:**
  - our approach can be generalized to deep neural networks.

$$f(\mathbf{x}, \mathbf{W}, \mathbf{a}) = \mathbf{a}^\top \sigma\left(\mathbf{W}^{(H)} \cdots \sigma\left(\mathbf{W}^{(1)}\mathbf{x}\right)\right)$$

The conclusion is still the same:
Using the same arguments, as long as the Gram matrix has a lower bounded least eigenvalue, gradient flow converges to zero training loss at a linear convergence rate.

  - The number of hidden nodes m required can be reduced. Using advanced tools from probability and matrix perturbation theory to analyze H(t), we may be able to tighten the bound.
  - If we use other loss functions instead of the empirical loss, we may be able to prove the convergence rates of accelerated methods.

Reference

Du, S. S., Zhai, X., Póczos, B., & Singh, A. (2018). Gradient Descent Provably Optimizes Over-parameterized Neural Networks. *ArXiv (Cornell University)*. https://www.arxiv.org/pdf/1810.02054