Stochastic Gradient Descent in Over-Parameterized Learning

Jeffrey Mei, Cody Melcher, Kamaljeet Singh
Department of Mathematics,
The University of Arizona,
jmei@math.arizona.edu, cmelcher@math.arizona.edu, kamaljeetsingh@math.arizona.edu

Abstract

Many modern machine learning models are over-parameterized. While the classical theory for under-parameterized models has been thoroughly analyzed, the theory for over-parameterized regimes is largely undeveoped. Machine learning practitioners have observed empirical evidence that over-parameterized models defy the conventional theory: local minima have the tendency to also be global, stochastic gradient descent converges particularly fast. In this report, we analyze some of the benefits stochastic gradient descent receives in over-parameterized models and provide numerical results to support the claims.

1 Introduction

Conventional wisdom recommends the bias-variance trade-off as the guiding principle for fitting models. An *underfit* model fails to capture the data's underlying patterns (high bias, low variance), while an *overfit* model will "memorize" the training data and will fail to generalize (low bias, high variance). This trade-off is the justification for a myriad of methods that balance model fit with model complexity: regularization (e.g. lasso, ridge regression), ensemble methods (e.g. random forest), various information criteria (e.g. AIC, BIC), cross-validation, etc.

However, the bias-variance trade-off has failed to explain the success of neural networks. In the 1990's, when compute was becoming more accessible, neural network models exploded in size, reaching capacities to perfectly fit the training data. By ignorance or by wisdom, computer scientists ignored the bias-variance trade-off and trained their models to interpolate the training data. Shockingly, despite overfitting the training data, the models seemed to generalize well, contradicting the bias-variance trade-off. This contradiction went on unexplained until the discovery of the double descent phenomenon [2].

Recent evidence suggests SGD possesses many advantages in these over-parameterized contexts [4, 1]: a tendency for local minima to also be global minima, faster SGD convergence, and improved efficiency of smaller batch sizes. These are all incredible claims that warrant further investigation. In this report, we focus on the improved efficiency of batch sizes in over-parameterized models.

1.1 Applications

SGD is a powerful and flexible method used in various optimization problems. In this report, we will focus on its application to over-parameterized neural networks. As the success of large neural networks proliferates, it is increasingly important to understand how SGD behaves in over-parameterized settings. Neural networks have widespread applications in image classification [6], deep reinforcement learning [5], and generative adversarial networks [3]. In training these large models, SGD provides significant computational savings relative to the deterministic full gradient descent method. This is especially crucial as models increase in size.

The theory of over-parameterized models still lags behind its empirical understanding. By understanding the mechanisms that drive SGD efficiency in over-parameterized regimes, new methods can be developed that leverage the advantages of over-parameterization.

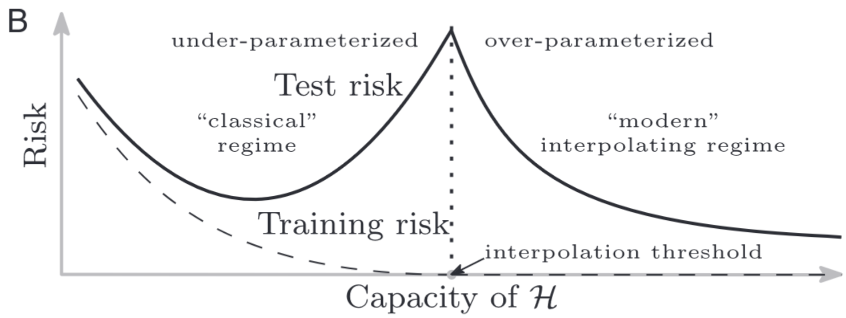


Figure 1: Double descent phenomenon [2]. $\mathcal{H}$ is the model complexity.

1.2 Literature review

Since the discovery of the double descent phenomenon [2], significant effort has been invested into studying the benefits of over-parameterized models. The double descent phenomenon attributes particular significance to the *interpolation threshold* – the point where the model achieves perfect training error. It separates the classical U-shaped generalization error in the under-parameterized regime from the monotonically decreasing generalization error in the over-parameterized regime (see Figure 1).

While under-parameterized models have been thoroughly studied, recent work demonstrates the many benefits of over-parameterized models [1]. For example, while optimization procedures are likely to get stuck in local minima in under-parameterized regimes, it is more likely the case that local minima are also global minima in over-parameterized regimes. Likewise, SGD is known to have sub-optimal convergence rates in under-parameterized settings. This motivates the development of variance reduction techniques (e.g. SVRG, SAGA, FISTA, etc.) to achieve fast convergence. However, in over-parameterized settings, SGD converges quickly because it receives variance reduction "for free" [4].

In this report, we will focus on the *batch size saturation* effect in over-parameterized models. Whereas one iteration of batch size m is approximately as expensive as m iterations of batch size one in the under-parameterized setting, there exists a cutoff for this *linear* effect in the over-parameterized setting [4]. After a *critical batch size* m^*, there are diminishing returns for the batch size. In other words, SGD for a moderate batch size can be approximately as effective as a full gradient descent. This is the focus of our paper.

2 Methodology/Algorithm description

The general SGD algorithm with a fixed number of iterations is as follows:

Algorithm 1 Stochastic Gradient Descent

Step	**Description**
Input	Initial parameter $x_0 \in \mathbb{R}^m$, minibatch size m, number of iterations T
For each iteration $k = 0, \ldots, T-1$	- Choose a random subset $I_k \subset \{1, \ldots, n\}$ with $\lvert I_k \rvert = m < n$ - Set step size $\alpha_k = \frac{1}{\sqrt{k}}$ - Compute minibatch gradient: $g_k = \frac{1}{m} \sum_{i \in I_k} \nabla f_i(x_k)$ - Update parameter: $x_{k+1} = x_k - \alpha_k g_k$
Output	Final parameter x_T

3 Numerical Experiments

3.1 Implementation

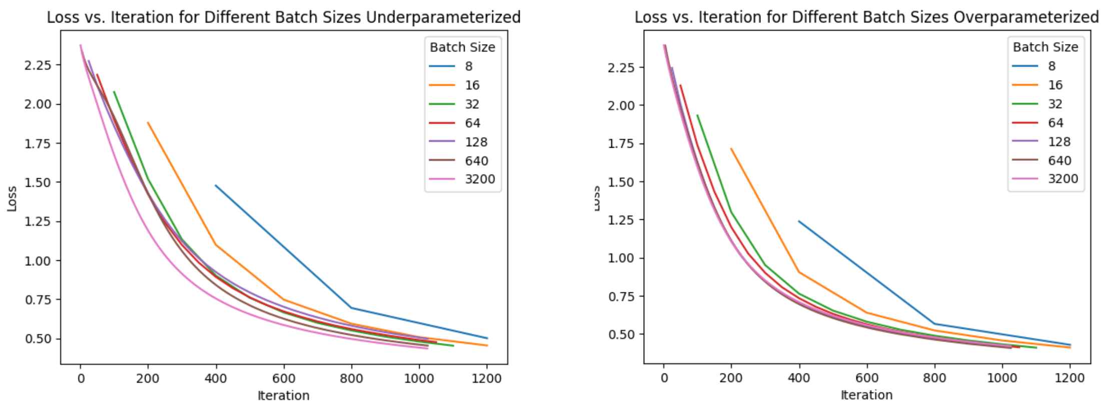
To evaluate the qualitative differences between under-parameterized and over-parameterized models, we trained two fully connected, single layer, neural networks: one under-parameterized (32 hidden nodes), and one over-parameterized (128 hidden nodes). The neural networks were trained on the MNIST data set with $n = 3200$ training samples. To ensure the selected models were truly under-parameterized and over-parameterized, we reproduced the double-descent phenomenon, although the double-descent was not nearly as pronounced as the one illustrated in the original paper [2].

3.2 Batch Size Saturation

To study the extent of batch size saturation in over-parameterized models, we examined how the training error decreased after each iteration of SGD on the under-parameterized and over-parameterized models for various batch sizes (see Figure 2). Specifically, we evaluate batch sizes of 8, 16, 32, 64, 128, 640, and 3200 (full gradient descent). While we plot iterations versus loss in Figure 2, we note that the loss is only recorded after each epoch ($n = 3200$). That is why for a batch size of 8, the number of iterations begins at 400 ($3200/8 = 400$).

If there is batch saturation, we can expect some clustering of the loss curves near the full gradient descent loss curve. This would indicate that moderate batch sizes are approximately as good as the full gradient descent.

We observe that in Figure 2b, batch sizes of 128, 640, and 3200 are overlapping. This suggests there is some batch saturation, as a batch size of 128 is approximately equivalent to a batch size of 3200. Consequently, we can expect the critical batch size to be between 64 and 128. In contrast, we see that the curves do not merge into each other at or after any batch size in the under-parameterized model in Figure 2a. There is a clear spectrum of curves for varying batch sizes, indicating the absence of linear scaling and saturation regions in the under-parameterized regime.



(a) Under-parameterized (b) Over-parameterized

Figure 2: Numerical results demonstrating batch size saturation in the over-parameterized model.

3.3 Estimating m^*

In practice, being able to explicitly compute the critical batch size for a given problem is invaluable, as it would enable researchers to train their models with optimal batch sizes. The critical batch size is proved to be a function of the $\beta-$smoothing parameter and eigenvalues associated with the loss function [4]. It is shown to be $m^* = \frac{\beta}{\lambda_1 - \lambda_k} + 1$, where the λ_i are the ordered eigenvalues of the Hessian of the loss function, with λ_1 being the largest positive eigenvalue, and λ_k being the smallest positive eigenvalue.

It should be noted that additional assumptions can be placed on the problem that allow m^* to be a function of β, λ_1 and n. This is due to concerns that exist about estimating the smallest positive eigenvalue of a matrix, which has been found to be difficult and unreliable estimate due to it usually be near 0. However, our estimate of λ_k was not near 0 and so we felt that using the original m^* relationship was fine.

Estimating β was done analytically and λ_1, λ_k was done via the NumPy package in Python. For β, we know from the neural network that β is the supremum of the product of the spectral norms of the three weight matrices, the norm of the reLU, and the norm of the softmax. Estimating these norms was done via Numpy. Given the intricate and non-linear nature of neural network models, these results should be treated with caution.

Our results are given in the below table:

Parameter	**Under-parameterized**	**Over-parameterized**
β	8.5223	8.9454
λ_1	1.7334	1.6857
λ_k	0.7199	0.8965
m^*	**9.4094**	**12.3491**

Thus, our analytical result indicates the batch size of 12 is optimal. However, comparing this to the empirical results in Figure 2b, m^* should be between $m = 128$ and $m = 640$. It is unclear why the empirical and analytical results disagree, but we feel the empirical results are more convincing than the analytical results.

4 Conclusion and Future Direction

The discovery of the double-descent phenomenon has brought much attention to over-parameterized models. Our empirical results indicate that there appears to be a critical batch size in the over-parameterized setting, such that the critical batch size has nearly identical loss curves as the full gradient descent.

Our analytical estimate of the critical batch size resulted in a value signiciantly lower than our empirical results suggested. Empirically, we observe the critical batch size m^* is between $m = 128$ and $m = 640$, but our analytical estimate is $m = 12$.

4.1 Future Directions

Given more time on this project, we would like to evaluate the other claims made about over-parameterized models. Specifically, we would like to evaluate the claim that local optima reached in over-parameterized settings tend to also be global optima. While we found theory that supports this claim, convincing empirical evidence is scant. Alternatively, we would also like to evaluate the claim that SGD receives variance reduction "for free." Specifically, it would be interesting to compare SGD to variance reduction techniques in the under-parameterized and over-parameterized settings. If SGD is dominated in the under-parameterized setting and is competitive in the over-parameterized setting, this would indicate that SGD indeed receives additional benefits in over-parameterized models.

References

[1] M. Belkin, *Fit without fear: remarkable mathematical phenomena of deep learning through the prism of interpolation*, Acta Numerica, 30 (2021), pp. 203–248.

[2] M. Belkin, S. Ma, and S. Mandal, *To understand deep learning we need to understand kernel learning*, June 2018. arXiv:1802.01396 [cs, stat].

[3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, *Generative Adversarial Networks*, June 2014. arXiv:1406.2661 [cs, stat].

[4] S. Ma, R. Bassily, and M. Belkin, *The Power of Interpolation: Understanding the Effectiveness of SGD in Modern Over-parametrized Learning.*

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing Atari with Deep Reinforcement Learning*, Dec. 2013. arXiv:1312.5602 [cs].

[6] S. Zagoruyko and N. Komodakis, *Learning to compare image patches via convolutional neural networks*, in 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Boston, MA, USA, June 2015, IEEE, pp. 4353–4361.

5 Appendix

```python
import tensorflow as tf
import tensorflow.keras.datasets.mnist as mnist
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.utils import to_categorical
import pandas as pd
import matplotlib.pyplot as plt
from keras.optimizers import SGD
import random
from keras.callbacks import EarlyStopping
import math

# Load MNIST dataset
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Shape of the training and testing data
print("Training images shape:", train_images.shape)
print("Training labels shape:", train_labels.shape)
print("Testing images shape:", test_images.shape)
print("Testing labels shape:", test_labels.shape)

# Normalize the pixel values to be between 0 and 1
train_images = train_images.astype('float32') / 255
test_images = test_images.astype('float32') / 255


# One-hot encode the labels
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

def create_model(num_parameters, train_images, train_labels):
    if not isinstance(num_parameters, int) or num_parameters <= 0:
        raise ValueError("num_parameters must be a positive integer.")

    model = Sequential([
        Flatten(input_shape=(28, 28)),  # Flatten the input images
        Dense(num_parameters, activation='relu'),   # Fully connected layer with num_parameters neurons\renewcommand{\thesection}{Appendix \Alph{section}}
```

```
        Dense(10, activation='softmax')  # Output layer with 10 neurons (one for each class)
    ])

    # Compile the model
#    model.compile(optimizer='adam',
#                  loss='categorical_crossentropy',
#                  metrics=['accuracy'])

    # Compile the model
    #model.compile(optimizer='adam',
    #                loss='categorical_crossentropy',
    #                metrics=['accuracy'])
    sgd = SGD(momentum=0.95)
    model.compile(optimizer=sgd,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    # Fit the model and store the training history
    history = model.fit(train_images, train_labels, epochs=50, batch_size=64, validation_split=0.2)

    # Extract the training and validation error from the history
    train_error = history.history['loss'][-1]  # Training error is the final loss value
    val_error = history.history['val_loss'][-1]  # Validation error is the final validation loss value

    return train_error, val_error, model.count_params()

# repeat of above cell, but with 4000 images (aligned with double-descent paper)
train_images = train_images[:4000]
train_labels = train_labels[:4000]


def get_num_hidden(n):
  # gives an approximation of the number of hidden layers for a desired number of parameters (n)
    return (n - 10) / (785 + 10 + 1)

#params = [2**i for i in range(2,8)] # these are number of units in the hidden layer
#params = [4, 8, 16, 32, 33, 34, 35, 40, 45, 49, 50, 51, 52, 55, 60, 64, 128]
```

```python
#params = [4, 8, 16, 32] + list(range(40, 60)) + [64, 128]

interpolation_threshold = len(train_images) * 10
interp_thresh_H = round(get_num_hidden(interpolation_threshold))
params = [4, 8, 16, 32] + list(range(interp_thresh_H-8, interp_thresh_H+8, 2)) + [64,76, 88, 100]

random.seed(321)
error = []
for num_params in params:
    train_error, val_error, num_params = create_model(num_params, train_images, train_labels)
    error.append({'number of parameters': num_params, 'Training Error': train_error, 'Validation Error': val_error})

#error = pd.DataFrame(columns=['Training Error', 'Validation Error'])
error= pd.DataFrame(error)#, columns=['Training Error', 'Validation Error'])
print(error)

# ploting double descent curve
number_of_examples = len(train_images)
plt.plot(error['number of parameters'], error['Validation Error'], label='Validation Error', marker='o')
plt.plot(error['number of parameters'], error['Training Error'], label='Training Error', marker='o')
#plt.axvline(number_of_examples, color='black', linestyle='-', linewidth=1, marker='o') # last edited by Jeff, p
plt.axvline(number_of_examples * 10, color='black', linestyle='-', linewidth=1, marker='o') # jeff changed this (interpolation threshold)
plt.xlabel('Number of Parameters')
plt.ylabel('Error')
plt.title(f'Training and Validation Error vs. Number of Parameters\n(Number of Training Examples: {number_of_examples})')
plt.legend()
plt.ticklabel_format(style='plain', axis='x')
plt.ylim(0, 0.6)
plt.xticks(rotation=45)
plt.show()

#kamal
# find two models in the two regimes having same score
# vary batch_size and claim that after a certain size, it saturates
```

```
# compare variances for different weight initializations in two regimes.

# Define early stopping criteria
early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=1)

# Train the model with early stopping
num_parameters = 55 # <------------------------------------CHANGE THIS PARAMETER
# I propose num_parameters:
  # Under-Parameterized: 45
  # Over-Parameterized: 55, 64

model = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten the input images
    Dense(num_parameters, activation='relu'),   # Fully connected layer
    Dense(10, activation='softmax')  # Output layer with 10 neurons
])
sgd = SGD(momentum=0.95)
model.compile(optimizer=sgd,
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(train_images, train_labels,
                    epochs=50,
                    batch_size=64,
                    validation_split=0.2,
                    callbacks=[early_stopping])

# Determine the number of epochs it took for convergence
num_epochs_to_convergence = len(history.history['loss'])
print(f'Number of epochs to convergence: {num_epochs_to_convergence}')

# Calculating the convergence rate for different batch sizes
early_stopping = EarlyStopping(monitor='val_loss', patience=5, verbose=0)
def model_convergence(batch_size):

    model1 = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten the input images
    Dense(32, activation='relu'),   # --------------------------  I used 45 for underparameterized model
```

```
    Dense(10, activation='softmax')  # Output layer with 10 neurons
    ])
    sgd = SGD(momentum=0.95)
    model1.compile(optimizer=sgd,
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

    history1 = model1.fit(train_images, train_labels,
                          epochs=50,
                          batch_size=batch_size,
                          validation_split=0.2,
                          callbacks=[early_stopping],verbose =0)

    # Determine the number of epochs it took for convergence
    num_epochs_to_convergence1 = len(history1.history['loss'])

    model2 = Sequential([
    Flatten(input_shape=(28, 28)),  # Flatten the input images
    Dense(64, activation='relu'),   # ------------------------ I used 55 for overparameterized model
    Dense(10, activation='softmax')  # Output layer with 10 neurons
    ])
    sgd = SGD(momentum=0.95)
    model2.compile(optimizer=sgd,
                   loss='categorical_crossentropy',
                   metrics=['accuracy'])

    history2 = model2.fit(train_images, train_labels,
                          epochs=50,
                          batch_size=batch_size,
                          validation_split=0.2,
                          callbacks=[early_stopping],verbose=0)

    # Determine the number of epochs it took for convergence
    num_epochs_to_convergence2 = len(history2.history['loss'])


    return num_epochs_to_convergence1 , num_epochs_to_convergence2

batch_sizes = [2**i for i in range(1,8)]
#batch_sizes = [4,48,64,100,128,160,200,256]
batch_sizes

val_pct = 0.2 # percent used for validation
num_samples = len(train_images) * (1 - val_pct)
```

```
epochs = []
for i in range(5):
    for batch_size in batch_sizes:

        x,y = model_convergence(batch_size)
        epochs.append({'model1': x*(math.ceil(num_samples/batch_size)), 'model2': y*(math.ceil(num_samples/batch_size)), "batch size":batch_size, "iteration":i})

epochs= pd.DataFrame(epochs)
print(epochs)

epochs = epochs.groupby('batch size').mean().reset_index()

# Plot for Multiple Runs
plt.scatter(epochs["batch size"], epochs['model1'], label='Underparameterized model', marker='o')
plt.scatter(epochs["batch size"], epochs['model2'], label='Overparameterized model', marker='o')
plt.plot(epochs['batch size'], epochs['model1'])
plt.plot(epochs['batch size'], epochs['model2'])
plt.xlabel('Batch Size')
plt.ylabel('Number of Iterations until Convergence')
plt.title("Number of Iterations vs Batch Size for Underparameterized and Overparameterized regime")
plt.legend()


plt.show()

batch_sizes = [4, 12, 16, 48, 64, 84, 100]
num_iter = 2 # number of times to run the same conditions

# Average Multiple Runs
epochs = []
for batch_size in batch_sizes:
  for i in range(num_iter):
    x,y = model_convergence(batch_size)
    epochs.append({'model1': x, 'model2': y, "batch size":batch_size})
epochs= pd.DataFrame(epochs)
print(epochs)

# Translate Epochs to Iterations

```

```python
val_pct = 0.2 # percent used for validation
num_samples = len(train_images) * (1 - val_pct)

num_iter = (num_samples / batch_size) * num_epochs
model1_num_iter = (num_samples / epochs['batch size']) * epochs['model1']
model2_num_iter = (num_samples / epochs['batch size']) * epochs['model2']

iter_df = {
    'batch size': epochs['batch size'],
    'model1': model1_num_iter,
    'model2': model2_num_iter
}
iter_df = pd.DataFrame(iter_df)
print(iter_df)

# Take Average of Runs
iter_summary = iter_df.groupby('batch size').mean().reset_index()

# Plot for Multiple Runs
plt.scatter(iter_df["batch size"], iter_df['model1'], label='Underparameterized model', marker='o')
plt.scatter(iter_df["batch size"], iter_df['model2'], label='Overparameterized model', marker='o')
plt.plot(iter_summary['batch size'], iter_summary['model1'])
plt.plot(iter_summary['batch size'], iter_summary['model2'])
plt.xlabel('Batch Size')
plt.ylabel('Number of Iterations until Convergence')
plt.title("Number of Iterations vs Batch Size for Underparameterized and Overparameterized regime")
plt.legend()


plt.show()

from keras.callbacks import Callback

class StopAfterIterations(Callback):
  # Saves the loss after each batch
  # Stops training after t iterations

    def __init__(self, max_iterations):
        super(StopAfterIterations, self).__init__()
        self.max_iterations = max_iterations
```

```
        self.iterations = 0
        self.accuracy = []
        self.loss = []

    def on_batch_end(self, batch, logs=None):
        self.accuracy.append(logs.get('accuracy'))
        self.loss.append(logs.get('loss'))
        self.iterations += 1
        if self.iterations >= self.max_iterations:
            self.model.stop_training = True
            print(f"Stopped training after {self.max_iterations} iterations.")

    def on_epoch_begin(self, epoch, logs=None):
            self.total_iterations = 0

# Instantiate the custom callback
batch_sizes = [32,64,128,512,1024,2048]
max_iterations=500
output_under = np.zeros((max_iterations,len(batch_sizes)+1))
column_names = ['Iteration'] + [f'Batch_{batch_size}' for batch_size in batch_sizes]

output_over = np.zeros((max_iterations,len(batch_sizes)+1))
i=1
for batch_size in batch_sizes:
    # Under-Parameterized Model
    num_parameters = 45
    under_model = Sequential([
        Flatten(input_shape=(28, 28)),  # Flatten the input images
        Dense(num_parameters, activation='relu'),   # Fully connected layer with num_parameters neurons\renewcommand{\thesection}{Appendix \Alph{section}}
        Dense(10, activation='softmax')  # Output layer with 10 neurons (one for each class)
    ])

    # Over-Parameterized Model
    num_parameters = 55
    over_model = Sequential([
        Flatten(input_shape=(28, 28)),  # Flatten the input images
        Dense(num_parameters, activation='relu'),   # Fully connected layer with num_parameters neurons\renewcommand{\thesection}{Appendix \Alph{section}}
```

```
        Dense(10, activation='softmax')  # Output layer with 10
            neurons (one for each class)
    ])

    under_sgd = SGD() # SGD must be defined for both regimes
    under_model.compile(optimizer=under_sgd,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    over_sgd = SGD() # SGD must be defined for both regimes
    over_model.compile(optimizer=over_sgd,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    # Fit the model and store the training history
    under_stop_after_iterations = StopAfterIterations(max_iterations
        )
    under_history = under_model.fit(train_images, train_labels,
        epochs=max_iterations, batch_size=batch_size,
        validation_split=0.2, callbacks=[under_stop_after_iterations
        ],verbose=0)
    output_under[:,i] = under_stop_after_iterations.loss
    over_stop_after_iterations = StopAfterIterations(max_iterations)
    over_history = over_model.fit(train_images, train_labels, epochs
        =max_iterations, batch_size=batch_size, validation_split=0.2,
         callbacks=[over_stop_after_iterations],verbose=0)
    output_over[:,i] = over_stop_after_iterations.loss
    i=i+1
output_under[:,0] = range(under_stop_after_iterations.iterations)
output_over[:,0] = range(over_stop_after_iterations.iterations)

output_under = pd.DataFrame(output_under, columns = column_names)

output_over = pd.DataFrame(output_over, columns = column_names)
#yy = under_stop_after_iterations.loss

for i, column in enumerate(output_under.columns[1:], start=1):
    plt.scatter(output_under.iloc[:, 0], output_under.iloc[:, i],
        alpha=1,label=column, marker='.', linestyle='-')

plt.xlabel('Iteration')
plt.ylabel('Loss')  # Adjust ylabel according to your data
plt.title('Loss vs. Iteration for Different Batch Sizes 
   Underparameterized')
plt.legend(title='Batch Size')
```

```
plt.show()

"""
xx = range(over_stop_after_iterations.iterations)
yy = over_stop_after_iterations.loss
plt.scatter(xx, yy, label='Overparameterized model', marker='o')
"""


for i, column in enumerate(output_over.columns[1:], start=1):
    plt.scatter(output_over.iloc[:, 0], output_over.iloc[:, i], alpha=1,label=column, marker='.', linestyle='-')

plt.xlabel('Iteration')
plt.ylabel('Loss')  # Adjust ylabel according to your data
plt.title('Loss vs. Iteration for Different Batch Sizes Overparameterized')
plt.legend(title='Batch Size')
plt.show()

plt.scatter(output_over[:,0], output_over[:,1], label='8', alpha=0.5, marker='.', linestyle='-')
plt.scatter(output_over[:,0], output_over[:,2], label='128', alpha=0.5, marker='.', linestyle='-')
plt.scatter(output_over[:,0], output_over[:,3], label='3200', alpha=0.5, marker='.', linestyle='-')

"""
xx = range(over_stop_after_iterations.iterations)
yy = over_stop_after_iterations.loss
plt.scatter(xx, yy, label='Overparameterized model', marker='o')
"""

#plt.plot(iter_summary['batch size'], iter_summary['model1'])
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.title("Number of Iterations vs Batch Size for Underparameterized and Overparameterized regime")
plt.legend()


plt.show()

output_under[350:400,0]

```

```
# Instantiate the custom callback
num_train = 3200
batch_sizes = [8, 16, 32, 64, 128, 640, num_train]
max_iter = 1024
output_under = np.zeros((max_iter,len(batch_sizes)+1))
column_names = ['Iteration'] + [f'Batch_{batch_size}' for batch_size
    in batch_sizes]


# ========
# Fit Data
# ========
under_lst = [0] * len(batch_sizes)
over_lst = [0] * len(batch_sizes)

output_over = np.zeros((max_iter,len(batch_sizes)+1))
i=0
for batch_size in batch_sizes:

    # Under-Parameterized Model
    num_parameters = 32
    under_model = Sequential([
        Flatten(input_shape=(28, 28)),  # Flatten the input images
        Dense(num_parameters, activation='relu'),
        Dense(10, activation='softmax')  #
    ])

    # Over-Parameterized Model
    num_parameters = 64
    over_model = Sequential([
        Flatten(input_shape=(28, 28)),  # Flatten the input images
        Dense(num_parameters, activation='relu'),
        Dense(10, activation='softmax')
    ])

    under_sgd = SGD() # SGD must be defined for both regimes
    under_model.compile(optimizer=under_sgd,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

    over_sgd = SGD() # SGD must be defined for both regimes
    over_model.compile(optimizer=over_sgd,
                  loss='categorical_crossentropy',
                  metrics=['accuracy'])

```

```
    # ============================================
    # Fit the model and store the training history
    # ============================================
    batches_per_epoch = int(num_train / batch_size)
    num_epochs = math.ceil(max_iter / batches_per_epoch)
    start_iter = batches_per_epoch
    end_iter = batches_per_epoch * num_epochs + batches_per_epoch

    # Underparameterized
    under_history = under_model.fit(train_images, train_labels,
                                    epochs=num_epochs, batch_size=batch_size,
                                    validation_split=0.2, verbose=0)
    under_df = {
        't': list(range(start_iter, end_iter, batches_per_epoch)),
        'loss': under_history.history['loss']
    }
    under_lst[i] = pd.DataFrame(under_df)

    # Overparameterized
    over_history = over_model.fit(train_images, train_labels,
                                  epochs=num_epochs, batch_size=batch_size,
                                  validation_split=0.2, verbose=0)
    over_df = {
        't': list(range(start_iter, end_iter, batches_per_epoch)),
        'loss': over_history.history['loss']
    }

    over_lst[i] = pd.DataFrame(over_df)
    i=i+1

# Plot Results (Underparameterized)
for i in range(len(batch_sizes)):
    plt.plot(under_lst[i]['t'], np.log(under_lst[i]['loss']), alpha=1, label=batch_sizes[i])

plt.xlabel('Iteration')
plt.ylabel('Log Loss')  # Adjust ylabel according to your data
plt.title('Loss vs. Iteration for Different Batch Sizes Underparameterized')
plt.legend(title='Batch Size')
plt.show()

# Plot Results (Overparameterized)
for i in range(len(batch_sizes)):
```

```python
    plt.plot(over_lst[i]['t'], np.log(over_lst[i]['loss']), alpha=1, label=batch_sizes[i])

plt.xlabel('Iteration')
plt.ylabel('Log Loss')  # Adjust ylabel according to your data
plt.title('Loss vs. Iteration for Different Batch Sizes Overparameterized')
plt.legend(title='Batch Size')
plt.show()

# Get Ratio of Underparameterized / Ovewrparameterized
for i in range(5):
  print((under_lst[i] / over_lst[i]))

#cody stuff

#define function to grab the norm for the weight matrices
def get_model_spectral_norms(model):
    norms = []
    for layer in model.layers:
        if isinstance(layer, tf.keras.layers.Dense):
            weights, biases = layer.get_weights()
            singular_values = np.linalg.svd(weights, compute_uv=False)
            spectral_norm = np.max(singular_values)
            norms.append(spectral_norm)
    return norms

# Calculate norms of weight matrices for the underparameterized model
under_spectral_norms = get_model_spectral_norms(under_model)
print("Spectral norms of the underparameterized model weight matrices:")
for i, norm in enumerate(under_spectral_norms, 1):
    print(f"Layer {i}: {norm}")

# Calculate norms of weight matrices for the overparameterized model
over_spectral_norms = get_model_spectral_norms(over_model)
print("Spectral norms of the overparameterized model weight matrices:")
for i, norm in enumerate(over_spectral_norms, 1):
    print(f"Layer {i}: {norm}")


```

```python
# Define function to grab the largest and smallest positive eigenvalues for the weight matrices
def get_model_eigenvalues(model):
    eigenvalues = []
    for layer in model.layers:
        if isinstance(layer, tf.keras.layers.Dense):
            weights, _ = layer.get_weights()
            singular_values = np.linalg.svd(weights, compute_uv=False)
            max_eigenvalue = np.max(singular_values)  # Largest eigenvalue
            min_eigenvalue = np.min(singular_values[singular_values > 0]) if np.any(singular_values > 0) else 0  # Smallest positive eigenvalue
            eigenvalues.append((max_eigenvalue, min_eigenvalue))
    return eigenvalues

# Under-parameterized Model (e.g., num_parameters = 45)
under_model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(45, activation='relu'),
    Dense(10, activation='softmax')
])
under_sgd = SGD(momentum=0.95)
under_model.compile(optimizer=under_sgd, loss='categorical_crossentropy', metrics=['accuracy'])
# Assume the model is already trained before this call
under_eigenvalues = get_model_eigenvalues(under_model)
print("Eigenvalues of the underparameterized model weight matrices:")
for i, (max_eig, min_eig) in enumerate(under_eigenvalues, 1):
    print(f"Layer {i} - Largest: {max_eig}, Smallest Positive: {min_eig}")

# Over-parameterized Model (e.g., num_parameters = 64)
over_model = Sequential([
    Flatten(input_shape=(28, 28)),
    Dense(64, activation='relu'),
    Dense(10, activation='softmax')
])
over_sgd = SGD(momentum=0.95)
over_model.compile(optimizer=over_sgd, loss='categorical_crossentropy', metrics=['accuracy'])
# Assume the model is already trained before this call
over_eigenvalues = get_model_eigenvalues(over_model)
```

```python
print("Eigenvalues of the overparameterized model weight matrices:")
for i, (max_eig, min_eig) in enumerate(over_eigenvalues, 1):
    print(f"Layer {i} - Largest: {max_eig}, Smallest Positive: {min_eig}")
```