

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 undeveloped. 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.

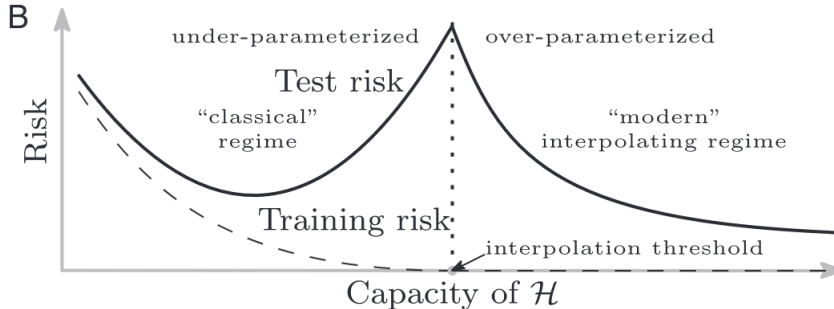


Figure 1: Double descent phenomenon [2]. \mathcal{H} is the model complexity.

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 $k = 0, \dots, T - 1$	<ul style="list-style-type: none"> - Choose a random subset $I_k \subset \{1, \dots, n\}$ with $I_k = m < n$ - 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.

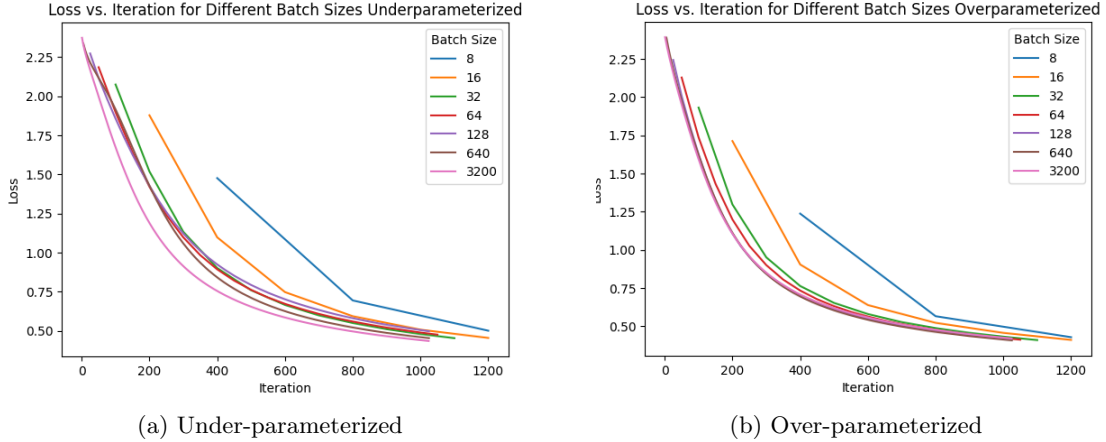


Figure 2: Numerical results demonstrating batch size saturation in the over-parameterized model.

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 β -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 significantly 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

```
1 import tensorflow as tf
2 import tensorflow.keras.datasets.mnist as mnist
3 import numpy as np
4 from tensorflow.keras.models import Sequential
5 from tensorflow.keras.layers import Dense, Flatten
6 from tensorflow.keras.utils import to_categorical
7 import pandas as pd
8 import matplotlib.pyplot as plt
9 from keras.optimizers import SGD
10 import random
11 from keras.callbacks import EarlyStopping
12 import math
13
14 # Load MNIST dataset
15 (train_images, train_labels), (test_images, test_labels) = mnist.
    load_data()
16
17 # Shape of the training and testing data
18 print("Training_images_shape:", train_images.shape)
19 print("Training_labels_shape:", train_labels.shape)
20 print("Testing_images_shape:", test_images.shape)
21 print("Testing_labels_shape:", test_labels.shape)
22
23 # Normalize the pixel values to be between 0 and 1
24 train_images = train_images.astype('float32') / 255
25 test_images = test_images.astype('float32') / 255
26
27
28 # One-hot encode the labels
29 train_labels = to_categorical(train_labels)
30 test_labels = to_categorical(test_labels)
31
32 def create_model(num_parameters, train_images, train_labels):
33     if not isinstance(num_parameters, int) or num_parameters <= 0:
34         raise ValueError("num_parameters must be a positive integer.
        ")
35
36     model = Sequential([
37         Flatten(input_shape=(28, 28)), # Flatten the input images
38         Dense(num_parameters, activation='relu'), # Fully
            connected layer with num_parameters neurons\renewcommand
            {\thesection}{Appendix \Alph{section}}
```

```

39         Dense(10, activation='softmax') # Output layer with 10
        neurons (one for each class)
40     ])
41
42     # Compile the model
43     # model.compile(optimizer='adam',
44     #               loss='categorical_crossentropy',
45     #               metrics=['accuracy'])
46
47     # Compile the model
48     #model.compile(optimizer='adam',
49     #              loss='categorical_crossentropy',
50     #              metrics=['accuracy'])
51     sgd = SGD(momentum=0.95)
52     model.compile(optimizer=sgd,
53                  loss='categorical_crossentropy',
54                  metrics=['accuracy'])
55
56     # Fit the model and store the training history
57     history = model.fit(train_images, train_labels, epochs=50,
58                        batch_size=64, validation_split=0.2)
59
60     # Extract the training and validation error from the history
61     train_error = history.history['loss'][-1] # Training error is
        the final loss value
62     val_error = history.history['val_loss'][-1] # Validation error
        is the final validation loss value
63
64     return train_error, val_error, model.count_params()
65
66 # repeat of above cell, but with 4000 images (aligned with double-
67 # descent paper)
68 train_images = train_images[:4000]
69 train_labels = train_labels[:4000]
70
71 def get_num_hidden(n):
72     # gives an approximation of the number of hidden layers for a
73     # desired number of parameters (n)
74     return (n - 10) / (785 + 10 + 1)
75
76 #params = [2**i for i in range(2,8)] # these are number of units in
77 #the hidden layer
78 #params = [4, 8, 16, 32, 33, 34, 35, 40, 45, 49, 50, 51, 52, 55, 60,
79 #          64, 128]

```

```

76 #params = [4, 8, 16, 32] + list(range(40, 60)) + [64, 128]
77
78 interpolation_threshold = len(train_images) * 10
79 interp_thresh_H = round(get_num_hidden(interpolation_threshold))
80 params = [4, 8, 16, 32] + list(range(interp_thresh_H-8,
    interp_thresh_H+8, 2)) + [64,76, 88, 100]
81
82 random.seed(321)
83 error = []
84 for num_params in params:
85     train_error, val_error, num_params = create_model(num_params,
        train_images, train_labels)
86     error.append({'number_of_parameters': num_params, 'Training_
        Error': train_error, 'Validation_Error': val_error})
87
88 #error = pd.DataFrame(columns=['Training Error', 'Validation Error
    '])
89 error= pd.DataFrame(error)#, columns=['Training Error', 'Validation
    Error'])
90 print(error)
91
92 # plotting double descent curve
93 number_of_examples = len(train_images)
94 plt.plot(error['number_of_parameters'], error['Validation_Error'],
    label='Validation_Error', marker='o')
95 plt.plot(error['number_of_parameters'], error['Training_Error'],
    label='Training_Error', marker='o')
96 #plt.axvline(number_of_examples, color='black', linestyle='--',
    linewidth=1, marker='o') # last edited by Jeff, p
97 plt.axvline(number_of_examples * 10, color='black', linestyle='--',
    linewidth=1, marker='o') # jeff changed this (interpolation
    threshold)
98 plt.xlabel('Number_of_Parameters')
99 plt.ylabel('Error')
100 plt.title(f'Training_and_Validation_Error_vs._Number_of_Parameters\n
    (Number_of_Training_Examples:{number_of_examples})')
101 plt.legend()
102 plt.ticklabel_format(style='plain', axis='x')
103 plt.ylim(0, 0.6)
104 plt.xticks(rotation=45)
105 plt.show()
106
107 #kamal
108 # find two models in the two regimes having same score
109 # vary batch_size and claim that after a certain size, it saturates

```



```

110 # compare variances for different weight initializations in two
    regimes.
111
112 # Define early stopping criteria
113 early_stopping = EarlyStopping(monitor='val_loss', patience=5,
    verbose=1)
114
115 # Train the model with early stopping
116 num_parameters = 55 # <-----CHANGE
    THIS PARAMETER
117 # I propose num_parameters:
118 # Under-Parameterized: 45
119 # Over-Parameterized: 55, 64
120
121 model = Sequential([
122     Flatten(input_shape=(28, 28)), # Flatten the input images
123     Dense(num_parameters, activation='relu'), # Fully connected
        layer
124     Dense(10, activation='softmax') # Output layer with 10 neurons
125 ])
126 sgd = SGD(momentum=0.95)
127 model.compile(optimizer=sgd,
128               loss='categorical_crossentropy',
129               metrics=['accuracy'])
130
131 history = model.fit(train_images, train_labels,
132                    epochs=50,
133                    batch_size=64,
134                    validation_split=0.2,
135                    callbacks=[early_stopping])
136
137 # Determine the number of epochs it took for convergence
138 num_epochs_to_convergence = len(history.history['loss'])
139 print(f'Number of epochs to convergence: {num_epochs_to_convergence}
    ')
140
141 # Calculating the convergence rate for different batch sizes
142 early_stopping = EarlyStopping(monitor='val_loss', patience=5,
    verbose=0)
143 def model_convergence(batch_size):
144
145     model1 = Sequential([
146         Flatten(input_shape=(28, 28)), # Flatten the input images
147         Dense(32, activation='relu'), # ----- I
            used 45 for underparameterized model

```

```

148     Dense(10, activation='softmax') # Output layer with 10 neurons
149 ]
150 sgd = SGD(momentum=0.95)
151 model1.compile(optimizer=sgd,
152               loss='categorical_crossentropy',
153               metrics=['accuracy'])
154
155 history1 = model1.fit(train_images, train_labels,
156                      epochs=50,
157                      batch_size=batch_size,
158                      validation_split=0.2,
159                      callbacks=[early_stopping], verbose =0)
160
161 # Determine the number of epochs it took for convergence
162 num_epochs_to_convergence1 = len(history1.history['loss'])
163
164 model2 = Sequential([
165     Flatten(input_shape=(28, 28)), # Flatten the input images
166     Dense(64, activation='relu'), # ----- I
167     used 55 for overparameterized model
168     Dense(10, activation='softmax') # Output layer with 10 neurons
169 ])
170 sgd = SGD(momentum=0.95)
171 model2.compile(optimizer=sgd,
172               loss='categorical_crossentropy',
173               metrics=['accuracy'])
174
175 history2 = model2.fit(train_images, train_labels,
176                      epochs=50,
177                      batch_size=batch_size,
178                      validation_split=0.2,
179                      callbacks=[early_stopping], verbose=0)
180
181 # Determine the number of epochs it took for convergence
182 num_epochs_to_convergence2 = len(history2.history['loss'])
183
184 return num_epochs_to_convergence1 , num_epochs_to_convergence2
185
186 batch_sizes = [2**i for i in range(1,8)]
187 #batch_sizes = [4,48,64,100,128,160,200,256]
188 batch_sizes
189
190 val_pct = 0.2 # percent used for validation
191 num_samples = len(train_images) * (1 - val_pct)

```

```

192 epochs = []
193 for i in range(5):
194     for batch_size in batch_sizes:
195
196         x,y = model_convergence(batch_size)
197         epochs.append({'model1': x*(math.ceil(num_samples/batch_size
198             )), 'model2': y*(math.ceil(num_samples/batch_size)), "
199             batch_size":batch_size, "iteration":i})
200
201 epochs= pd.DataFrame(epochs)
202 print(epochs)
203
204 epochs = epochs.groupby('batch_size').mean().reset_index()
205
206 # Plot for Multiple Runs
207 plt.scatter(epochs["batch_size"], epochs['model1'], label='
208     Underparameterized_model', marker='o')
209 plt.scatter(epochs["batch_size"], epochs['model2'], label='
210     Overparameterized_model', marker='o')
211 plt.plot(epochs['batch_size'], epochs['model1'])
212 plt.plot(epochs['batch_size'], epochs['model2'])
213 plt.xlabel('Batch_Size')
214 plt.ylabel('Number_of_Iterations_until_Convergence')
215 plt.title("Number_of_Iterations_vs_Batch_Size_for_Underparameterized
216     and_Overparameterized_regime")
217 plt.legend()
218
219 plt.show()
220
221 batch_sizes = [4, 12, 16, 48, 64, 84, 100]
222 num_iter = 2 # number of times to run the same conditions
223
224 # Average Multiple Runs
225 epochs = []
226 for batch_size in batch_sizes:
227     for i in range(num_iter):
228         x,y = model_convergence(batch_size)
229         epochs.append({'model1': x, 'model2': y, "batch_size":batch_size
230             })
231
232 epochs= pd.DataFrame(epochs)
233 print(epochs)
234
235 # Translate Epochs to Iterations
236

```

```

231 val_pct = 0.2 # percent used for validation
232 num_samples = len(train_images) * (1 - val_pct)
233
234 num_iter = (num_samples / batch_size) * num_epochs
235 model1_num_iter = (num_samples / epochs['batch_size']) * epochs['
    model1']
236 model2_num_iter = (num_samples / epochs['batch_size']) * epochs['
    model2']
237
238 iter_df = {
239     'batch_size': epochs['batch_size'],
240     'model1': model1_num_iter,
241     'model2': model2_num_iter
242 }
243 iter_df = pd.DataFrame(iter_df)
244 print(iter_df)
245
246 # Take Average of Runs
247 iter_summary = iter_df.groupby('batch_size').mean().reset_index()
248
249 # Plot for Multiple Runs
250 plt.scatter(iter_df["batch_size"], iter_df['model1'], label='
    Underparameterized_model', marker='o')
251 plt.scatter(iter_df["batch_size"], iter_df['model2'], label='
    Overparameterized_model', marker='o')
252 plt.plot(iter_summary['batch_size'], iter_summary['model1'])
253 plt.plot(iter_summary['batch_size'], iter_summary['model2'])
254 plt.xlabel('Batch_Size')
255 plt.ylabel('Number_of_Iterations_until_Convergence')
256 plt.title("Number_of_Iterations_vs_Batch_Size_for_Underparameterized
    and_Overparameterized_regime")
257 plt.legend()
258
259
260 plt.show()
261
262 from keras.callbacks import Callback
263
264 class StopAfterIterations(Callback):
265     # Saves the loss after each batch
266     # Stops training after t iterations
267
268     def __init__(self, max_iterations):
269         super(StopAfterIterations, self).__init__()
270         self.max_iterations = max_iterations

```

```

271         self.iterations = 0
272         self.accuracy = []
273         self.loss = []
274
275     def on_batch_end(self, batch, logs=None):
276         self.accuracy.append(logs.get('accuracy'))
277         self.loss.append(logs.get('loss'))
278         self.iterations += 1
279         if self.iterations >= self.max_iterations:
280             self.model.stop_training = True
281             print(f"Stopped training after {self.max_iterations} iterations.")
282
283     def on_epoch_begin(self, epoch, logs=None):
284         self.total_iterations = 0
285
286 # Instantiate the custom callback
287 batch_sizes = [32,64,128,512,1024,2048]
288 max_iterations=500
289 output_under = np.zeros((max_iterations,len(batch_sizes)+1))
290 column_names = ['Iteration'] + [f'Batch_{batch_size}' for batch_size
    in batch_sizes]
291
292 output_over = np.zeros((max_iterations,len(batch_sizes)+1))
293 i=1
294 for batch_size in batch_sizes:
295     # Under-Parameterized Model
296     num_parameters = 45
297     under_model = Sequential([
298         Flatten(input_shape=(28, 28)), # Flatten the input images
299         Dense(num_parameters, activation='relu'), # Fully
    connected layer with num_parameters neurons\renewcommand
    {\thesection}{Appendix \Alph{section}}
300         Dense(10, activation='softmax') # Output layer with 10
    neurons (one for each class)
301     ])
302
303     # Over-Parameterized Model
304     num_parameters = 55
305     over_model = Sequential([
306         Flatten(input_shape=(28, 28)), # Flatten the input images
307         Dense(num_parameters, activation='relu'), # Fully
    connected layer with num_parameters neurons\renewcommand
    {\thesection}{Appendix \Alph{section}}

```

```

308         Dense(10, activation='softmax') # Output layer with 10
           neurons (one for each class)
309     ])
310
311     under_sgd = SGD() # SGD must be defined for both regimes
312     under_model.compile(optimizer=under_sgd,
313                         loss='categorical_crossentropy',
314                         metrics=['accuracy'])
315
316     over_sgd = SGD() # SGD must be defined for both regimes
317     over_model.compile(optimizer=over_sgd,
318                       loss='categorical_crossentropy',
319                       metrics=['accuracy'])
320
321     # Fit the model and store the training history
322     under_stop_after_iterations = StopAfterIterations(max_iterations
323                                                       )
324     under_history = under_model.fit(train_images, train_labels,
325                                   epochs=max_iterations, batch_size=batch_size,
326                                   validation_split=0.2, callbacks=[under_stop_after_iterations
327                                                                    ], verbose=0)
328     output_under[:, i] = under_stop_after_iterations.loss
329     over_stop_after_iterations = StopAfterIterations(max_iterations)
330     over_history = over_model.fit(train_images, train_labels, epochs
331                                  =max_iterations, batch_size=batch_size, validation_split=0.2,
332                                  callbacks=[over_stop_after_iterations], verbose=0)
333     output_over[:, i] = over_stop_after_iterations.loss
334     i=i+1
335     output_under[:, 0] = range(under_stop_after_iterations.iterations)
336     output_over[:, 0] = range(over_stop_after_iterations.iterations)
337
338     output_under = pd.DataFrame(output_under, columns = column_names)
339
340     output_over = pd.DataFrame(output_over, columns = column_names)
341     #yy = under_stop_after_iterations.loss
342
343     for i, column in enumerate(output_under.columns[1:], start=1):
344         plt.scatter(output_under.iloc[:, 0], output_under.iloc[:, i],
345                     alpha=1, label=column, marker='.', linestyle='-')
346
347     plt.xlabel('Iteration')
348     plt.ylabel('Loss') # Adjust ylabel according to your data
349     plt.title('Loss vs. Iteration for Different Batch Sizes Underparameterized')
350     plt.legend(title='Batch Size')

```

```

344 plt.show()
345
346 """
347 xx = range(over_stop_after_iterations.iterations)
348 yy = over_stop_after_iterations.loss
349 plt.scatter(xx, yy, label='Overparameterized model', marker='o')
350 """
351
352
353 for i, column in enumerate(output_over.columns[1:], start=1):
354     plt.scatter(output_over.iloc[:, 0], output_over.iloc[:, i],
355                 alpha=1, label=column, marker='.', linestyle='-')
356
357 plt.xlabel('Iteration')
358 plt.ylabel('Loss') # Adjust ylabel according to your data
359 plt.title('Loss vs. Iteration for Different Batch Sizes Overparameterized')
360 plt.legend(title='Batch Size')
361 plt.show()
362
363 plt.scatter(output_over[:,0], output_over[:,1], label='8', alpha
364             =0.5, marker='.', linestyle='-')
365 plt.scatter(output_over[:,0], output_over[:,2], label='128', alpha
366             =0.5, marker='.', linestyle='-')
367 plt.scatter(output_over[:,0], output_over[:,3], label='3200', alpha
368             =0.5, marker='.', linestyle='-')
369
370 """
371
372 xx = range(over_stop_after_iterations.iterations)
373 yy = over_stop_after_iterations.loss
374 plt.scatter(xx, yy, label='Overparameterized model', marker='o')
375 """
376
377 #plt.plot(iter_summary['batch size'], iter_summary['model1'])
378 plt.xlabel('Iterations')
379 plt.ylabel('Loss')
380 plt.title("Number of Iterations vs Batch Size for Underparameterized
381           and Overparameterized regime")
382 plt.legend()
383
384 plt.show()
385
386 output_under[350:400,0]

```

```

383 # Instantiate the custom callback
384 num_train = 3200
385 batch_sizes = [8, 16, 32, 64, 128, 640, num_train]
386 max_iter = 1024
387 output_under = np.zeros((max_iter, len(batch_sizes)+1))
388 column_names = ['Iteration'] + [f'Batch_{batch_size}' for batch_size
    in batch_sizes]
389
390
391 # =====
392 # Fit Data
393 # =====
394 under_lst = [0] * len(batch_sizes)
395 over_lst = [0] * len(batch_sizes)
396
397 output_over = np.zeros((max_iter, len(batch_sizes)+1))
398 i=0
399 for batch_size in batch_sizes:
400
401     # Under-Parameterized Model
402     num_parameters = 32
403     under_model = Sequential([
404         Flatten(input_shape=(28, 28)), # Flatten the input images
405         Dense(num_parameters, activation='relu'),
406         Dense(10, activation='softmax') #
407     ])
408
409     # Over-Parameterized Model
410     num_parameters = 64
411     over_model = Sequential([
412         Flatten(input_shape=(28, 28)), # Flatten the input images
413         Dense(num_parameters, activation='relu'),
414         Dense(10, activation='softmax')
415     ])
416
417     under_sgd = SGD() # SGD must be defined for both regimes
418     under_model.compile(optimizer=under_sgd,
419         loss='categorical_crossentropy',
420         metrics=['accuracy'])
421
422     over_sgd = SGD() # SGD must be defined for both regimes
423     over_model.compile(optimizer=over_sgd,
424         loss='categorical_crossentropy',
425         metrics=['accuracy'])
426

```



```

427 # =====
428 # Fit the model and store the training history
429 # =====
430 batches_per_epoch = int(num_train / batch_size)
431 num_epochs = math.ceil(max_iter / batches_per_epoch)
432 start_iter = batches_per_epoch
433 end_iter = batches_per_epoch * num_epochs + batches_per_epoch
434
435 # Underparameterized
436 under_history = under_model.fit(train_images, train_labels,
437                                epochs=num_epochs, batch_size=batch_size,
438                                validation_split=0.2, verbose=0)
439 under_df = {
440     't': list(range(start_iter, end_iter, batches_per_epoch)),
441     'loss': under_history.history['loss']
442 }
443 under_lst[i] = pd.DataFrame(under_df)
444
445 # Overparameterized
446 over_history = over_model.fit(train_images, train_labels,
447                               epochs=num_epochs, batch_size=batch_size,
448                               validation_split=0.2, verbose=0)
449 over_df = {
450     't': list(range(start_iter, end_iter, batches_per_epoch)),
451     'loss': over_history.history['loss']
452 }
453
454 over_lst[i] = pd.DataFrame(over_df)
455 i=i+1
456
457 # Plot Results (Underparameterized)
458 for i in range(len(batch_sizes)):
459     plt.plot(under_lst[i]['t'], np.log(under_lst[i]['loss']), alpha
460             =1, label=batch_sizes[i])
461
462 plt.xlabel('Iteration')
463 plt.ylabel('Log Loss') # Adjust ylabel according to your data
464 plt.title('Loss vs. Iteration for Different Batch Sizes -
465           Underparameterized')
466 plt.legend(title='Batch Size')
467 plt.show()
468
469 # Plot Results (Overparameterized)
470 for i in range(len(batch_sizes)):

```

```

469     plt.plot(over_lst[i]['t'], np.log(over_lst[i]['loss']), alpha=1,
              label=batch_sizes[i])
470
471 plt.xlabel('Iteration')
472 plt.ylabel('Log Loss') # Adjust ylabel according to your data
473 plt.title('Loss vs. Iteration for Different Batch Sizes
              Overparameterized')
474 plt.legend(title='Batch Size')
475 plt.show()
476
477 # Get Ratio of Underparameterized / Overparameterized
478 for i in range(5):
479     print((under_lst[i] / over_lst[i]))
480
481 # cody stuff
482
483 # define function to grab the norm for the weight matrices
484 def get_model_spectral_norms(model):
485     norms = []
486     for layer in model.layers:
487         if isinstance(layer, tf.keras.layers.Dense):
488             weights, biases = layer.get_weights()
489             singular_values = np.linalg.svd(weights, compute_uv=
                False)
490             spectral_norm = np.max(singular_values)
491             norms.append(spectral_norm)
492     return norms
493
494 # Calculate norms of weight matrices for the underparameterized
    model
495 under_spectral_norms = get_model_spectral_norms(under_model)
496 print("Spectral norms of the underparameterized model weight
    matrices:")
497 for i, norm in enumerate(under_spectral_norms, 1):
498     print(f"Layer {i}: {norm}")
499
500 # Calculate norms of weight matrices for the overparameterized model
501 over_spectral_norms = get_model_spectral_norms(over_model)
502 print("Spectral norms of the overparameterized model weight matrices
    :")
503 for i, norm in enumerate(over_spectral_norms, 1):
504     print(f"Layer {i}: {norm}")
505
506

```

```

507 # Define function to grab the largest and smallest positive
    eigenvalues for the weight matrices
508 def get_model_eigenvalues(model):
509     eigenvalues = []
510     for layer in model.layers:
511         if isinstance(layer, tf.keras.layers.Dense):
512             weights, _ = layer.get_weights()
513             singular_values = np.linalg.svd(weights, compute_uv=
                False)
514             max_eigenvalue = np.max(singular_values) # Largest
                eigenvalue
515             min_eigenvalue = np.min(singular_values[singular_values
                > 0]) if np.any(singular_values > 0) else 0 #
                Smallest positive eigenvalue
516             eigenvalues.append((max_eigenvalue, min_eigenvalue))
517     return eigenvalues
518
519 # Under-parameterized Model (e.g., num_parameters = 45)
520 under_model = Sequential([
521     Flatten(input_shape=(28, 28)),
522     Dense(45, activation='relu'),
523     Dense(10, activation='softmax')
524 ])
525 under_sgd = SGD(momentum=0.95)
526 under_model.compile(optimizer=under_sgd, loss='
    categorical_crossentropy', metrics=['accuracy'])
527 # Assume the model is already trained before this call
528 under_eigenvalues = get_model_eigenvalues(under_model)
529 print("Eigenvalues of the underparameterized model weight matrices:"
    )
530 for i, (max_eig, min_eig) in enumerate(under_eigenvalues, 1):
531     print(f"Layer {i} - Largest: {max_eig}, Smallest Positive: {
        min_eig}")
532
533 # Over-parameterized Model (e.g., num_parameters = 64)
534 over_model = Sequential([
535     Flatten(input_shape=(28, 28)),
536     Dense(64, activation='relu'),
537     Dense(10, activation='softmax')
538 ])
539 over_sgd = SGD(momentum=0.95)
540 over_model.compile(optimizer=over_sgd, loss='
    categorical_crossentropy', metrics=['accuracy'])
541 # Assume the model is already trained before this call
542 over_eigenvalues = get_model_eigenvalues(over_model)

```

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