**Hand-Written Letter Recognition**

Jorge Velasquez

Zhiquan Cao

Professor: [Linda M Sellie](mailto:lms578@nyu.edu)

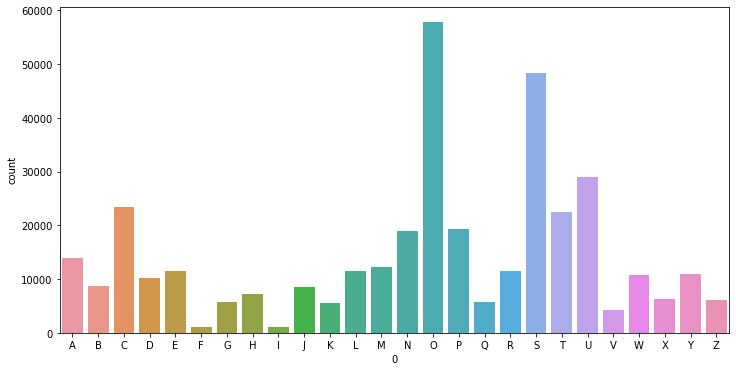
Date: Dec 7th

**Introduction**

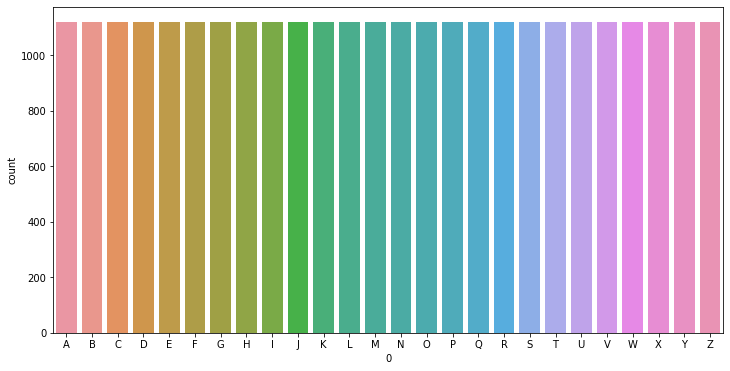
Our project was about creating machine learning models that can recognize and classify handwritten letters. Our data set is a collection of over 370k gray-scaled handwritten letters in 28x28 pixel images.

**Dataset:**

Bellow is a distribution of the dataset:

  
As can be seen, the data has an uneven distribution of letters. There’s an overabundance of O’s and S’s and very few F’s and I’s. Thus, to ensure that no feature set is overrepresented and to improve runtime, we resampled the data to balance out the category set and cut down on examples.

Bellow is a distribution of the resampled dataset:



**Logistic Regression:**

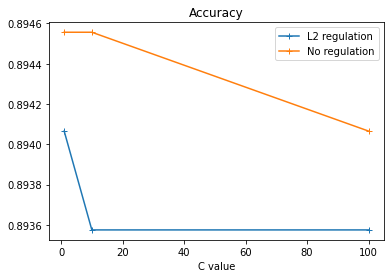
**Choice of models**

For the logistic regression models we used the sklearn.linear\_model LogisticRegression class. We used this class since it’s included in sklearn, the most commonly used python machine learning library. The LogisticRegression class handles all of the model training. The class supports multiple hyperparameters such as different penalty functions and C values. In our project we created models with the permutations of the following hyperparameters.

* Penalties: L2, none
* C values: 1, 10, 100

We were planning on generating more models, but while developing our project we had multiple issues with runtimes. Using L1 penalties causes extraordinarily long run times, upwards to 3 hours per model, which triggered time outs on Google CoLab, our juypter notebook editor. In the end we decided to focus on models that could be generated much quicker to fit within the project time frame. The LogisticRegression class also did not provide training error, making it difficult to judge overfitting and underfitting.

**Results**



The difference in accuracy between L2 regulation models and non-regulation models were barely noticeable. The difference between the most and least accurate models is 0.00098087297.

**Conclusion**

The best models had no regulation and a C value of 1 and 10. Interestingly, the non-regulation models performed better than the L2 regulation models, although given how close the values are that isn’t saying much. For L2 regulation, when the C value increases, the accuracy doesn’t decrease, which means that L2 regulation is underfit. Meanwhile, no regulation is overfit given the opposite circumstances. Due to how close all the accuracy values are, it’s difficult to determine if any differences are due to overfitting/underfitting or if due to data variance.

**Model Table**

|  | L2 | None |
| --- | --- | --- |
| 1 | 0.8940657184894556 | 0.8945561549779304 |
| 10 | 0.8935752820009809 | 0.8945561549779304 |
| 100 | 0.8935752820009809 | 0.8940657184894556 |

**SVM:**

**Choice of models**

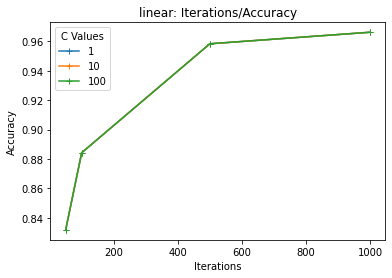
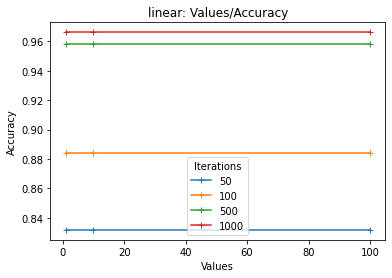
For the SVM models, we trained them using sklearn’s svm.SVC (support vector classifier) class. We chose to use this class simply because sklearn is the stand python machine learning library and there’s a lot of documentation online. Each instance of an SVC handles the entire training process, making it very convenient to generate models. One issue though is that there isn’t a way to record an SVC’s training error. All of our svm data only has validation error. SVCs support multiple hyperparameters, and in our models we tried different combinations of kernels, max iterations, C values, and in the case of the polynomial kernel, degrees. We generated models with permutations of the following hyperparameters:

* Kernels: Linear, Radial-Basis, Sigmoid, 1st degree polynomial, 2nd degree polynomial, 3rd degree polynomial
* C values: 1, 10, 100
* Max iterations: 50, 100, 500, 1000

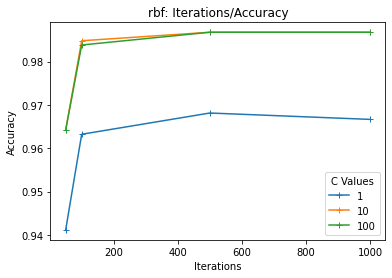
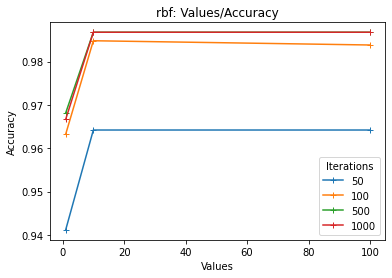
By having 6 kernels, 3 C values, and 4 max iterations, we created 72 svm models.

**Results**

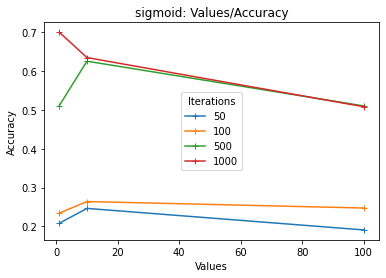
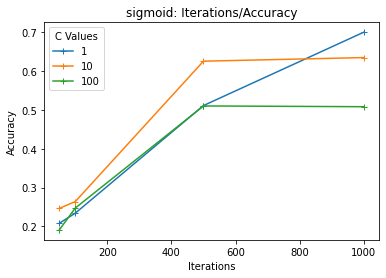
Each model was separated into their respective kernels for performance comparison.



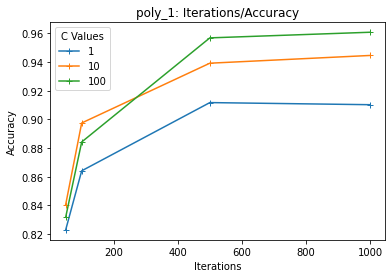
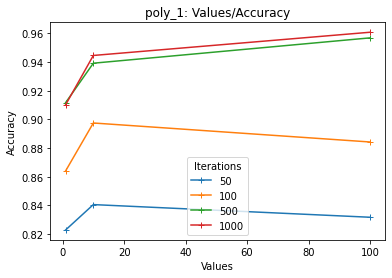
For the linear model, the most important hyperparameter was the number of iterations for the model. As you can see in the Iterations/Accuracy graph, accuracy rose at the same exact rate for each C value as iterations increased, thus the C values did not have much of an effect on the accuracy of the models.



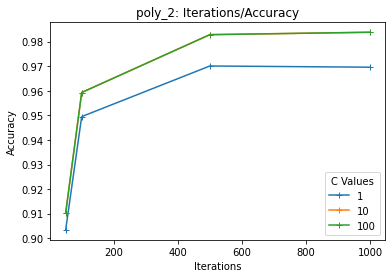
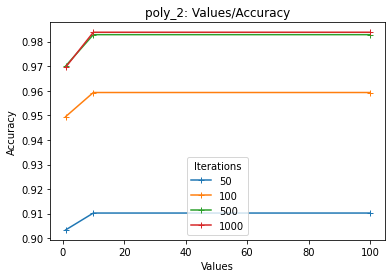
For the radial basis function kernel, there was a sharp cut off for the impact hyperparameters had on accuracy. There was very little difference between a model that ran for 100 or 1000 iterations, while the difference between models that ran for 50 and models that ran for 100 was almost a whole 2 points. The same holds true for models that have a C value of 1 and a C value of 10.



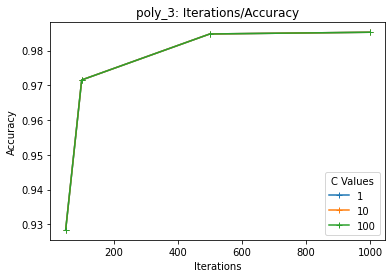
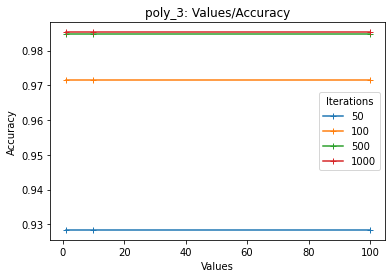
The sigmoid model is unique because lower C values performed almost as well as higher C values. In fact the most accurate model was the 1000 iteration model with a C value of 1. Overall though, a majority of the models produced accuracy scores below .7, making sigmoid the weakest kernel.



For the 1st degree polynomial kernel, models with lower max iterations benefited from lower C values, with both 100 and 50 iteration models reaching their highest accuracy score with a C value of 10.



For the 2nd degree polynomial kernel, all the model accuracy rates followed more standard paths. Greater max iterations resulted in better accuracy scores. C values only improved accuracy once they reached 10, from then onwards they only had a minimal effect.



The 3rd degree polynomial kernels were actually only impacted by the number of max iterations. Changing the C values did not improve the accuracy. Interestingly this model most closely resembles the behavior of the linear models, despite being the highest degree polynomial kernel.

**Conclusion**

Across all kernels, the higher the iteration the better the result. For the C values, however, different kernels perform differently. Except for the Poly\_1 kernel, all the other kernels perform better with larger C values, which means that most of the kernels are relatively underfit and need more precision with harsher penalties imposed by higher C\_values. This is especially true for sigmoid kernel. The slow rate of increase in accuracy in the sigmoid kernel graph indicates that it is even more underfitted than other kernels.

Best kernel: Radial Basis Function (rbf) with a C value of 100 and 1000 max iterations with an accuracy of 0.986758214811182.

**Model Table**

Key: [kernel]\_[C value]\_[max iterations]

| Model Name | Validation Accuracy |
| --- | --- |
| linear\_1\_50 | 0.8317802844531633 |
| linear\_1\_100 | 0.8842569887199607 |
| linear\_1\_500 | 0.9583128984796468 |
| linear\_1\_1000 | 0.9661598822952427 |
| linear\_10\_50 | 0.8317802844531633 |
| linear\_10\_100 | 0.8842569887199607 |
| linear\_10\_500 | 0.9583128984796468 |
| linear\_10\_1000 | 0.9661598822952427 |
| linear\_100\_50 | 0.8317802844531633 |
| linear\_100\_100 | 0.8842569887199607 |
| linear\_100\_500 | 0.9583128984796468 |
| linear\_100\_1000 | 0.9661598822952427 |
| rbf\_1\_50 | 0.9411476213830309 |
| rbf\_1\_100 | 0.9632172633643943 |
| rbf\_1\_500 | 0.9681216282491417 |
| rbf\_1\_1000 | 0.9666503187837175 |
| rbf\_10\_50 | 0.9641981363413438 |
| rbf\_10\_100 | 0.984796468857283 |
| rbf\_10\_500 | 0.986758214811182 |
| rbf\_10\_1000 | 0.986758214811182 |
| rbf\_100\_50 | 0.9641981363413438 |
| rbf\_100\_100 | 0.9838155958803335 |
| rbf\_100\_500 | 0.986758214811182 |
| rbf\_100\_1000 | 0.986758214811182 |
| sigmoid\_1\_50 | 0.20794507111329083 |
| sigmoid\_1\_100 | 0.2339382050024522 |
| sigmoid\_1\_500 | 0.5110348209906818 |
| sigmoid\_1\_1000 | 0.6998528690534576 |
| sigmoid\_10\_50 | 0.24619911721432075 |
| sigmoid\_10\_100 | 0.26385483079941147 |
| sigmoid\_10\_500 | 0.6253065228052967 |
| sigmoid\_10\_1000 | 0.6346248160863168 |
| sigmoid\_100\_50 | 0.19077979401667483 |
| sigmoid\_100\_100 | 0.24717999019127024 |
| sigmoid\_100\_500 | 0.5100539480137323 |
| sigmoid\_100\_1000 | 0.5080922020598333 |
| poly\_1\_50\_1 | 0.8229524276606179 |
| poly\_1\_100\_1 | 0.8641490926924963 |
| poly\_1\_500\_1 | 0.9117214320745464 |
| poly\_1\_1000\_1 | 0.9102501226091221 |
| poly\_10\_50\_1 | 0.8406081412457087 |
| poly\_10\_100\_1 | 0.8974987739087789 |
| poly\_10\_500\_1 | 0.9391858754291319 |
| poly\_10\_1000\_1 | 0.9445806768023541 |
| poly\_100\_50\_1 | 0.8317802844531633 |
| poly\_100\_100\_1 | 0.8842569887199607 |
| poly\_100\_500\_1 | 0.9568415890142227 |
| poly\_100\_1000\_1 | 0.9607650809220206 |
| poly\_1\_50\_2 | 0.9033840117704757 |
| poly\_1\_100\_2 | 0.9494850416871016 |
| poly\_1\_500\_2 | 0.9700833742030407 |
| poly\_1\_1000\_2 | 0.969592937714566 |
| poly\_10\_50\_2 | 0.9102501226091221 |
| poly\_10\_100\_2 | 0.9592937714565963 |
| poly\_10\_500\_2 | 0.982834722903384 |
| poly\_10\_1000\_2 | 0.9838155958803335 |
| poly\_100\_50\_2 | 0.9102501226091221 |
| poly\_100\_100\_2 | 0.9592937714565963 |
| poly\_100\_500\_2 | 0.982834722903384 |
| poly\_100\_1000\_2 | 0.9838155958803335 |
| poly\_1\_50\_3 | 0.9283962726826875 |
| poly\_1\_100\_3 | 0.971554683668465 |
| poly\_1\_500\_3 | 0.984796468857283 |
| poly\_1\_1000\_3 | 0.9852869053457577 |
| poly\_10\_50\_3 | 0.9283962726826875 |
| poly\_10\_100\_3 | 0.971554683668465 |
| poly\_10\_500\_3 | 0.984796468857283 |
| poly\_10\_1000\_3 | 0.9852869053457577 |
| poly\_100\_50\_3 | 0.9283962726826875 |
| poly\_100\_100\_3 | 0.971554683668465 |
| poly\_100\_500\_3 | 0.984796468857283 |
| poly\_100\_1000\_3 | 0.9852869053457577 |

**Neural Network:**

**Choice of models**

For a neural network solution, we plan to use the existing packages in Keras of the Tensorflow to generate the sequential models. We didn’t choose use the neural network template structured step by step from the HW5 for several reasons:

1, the input of our dataset is too large to fit into the elementary model

2, the runtime will be too slow to work with

3, it will be hard to find a working inner structure of the neural network if applied to our dataset

4, convoluted 2D neural network far more a better solution to train image data

The first two reasons are self-explanatory. We have over 784 inputs in the first layer compared with 64 inputs in the homework. Besides, the increased amount of input parameters adds to the complexity of the model. On one hand, the runtime might have increased quadratically compared with HW5. On the other hand, it will be hard to determine how the model should be structured, such as the number of layers and how many nodes it should contain for each layer. As a result, if using this elementary model where we have the entire customization freedom, there will be too many variables for us to manually test even before doing any analysis about the model.

The model provided by Keras, on the other hand, has automated much of the inner process of constructing the model. One line of code adds a layer to the model. The training time is also controlled within a reasonable range, which is about 20 minutes for each outcome. Besides, Keras provides the convoluted2D layer further benefiting the training of the image inputs.

An example of three different kinds of layers using Keras package:

Conv2D(32, kernel\_size=3, strides=2, padding='same', kernel\_initializer='he\_normal', input\_shape=(28, 28, 1)),

MaxPool2D(),

BatchNormalization(),

**Experiment with Batch\_Size and Number of Layers**

Even after simplified by the Keras package, there are still many variables that can change for different models. For example, the number and arrangement of different layers of either the same or different kinds, the parameters for within each layer, or the parameters in the model.compile() or model.fit() functions. Honestly, there are thousands of ways to configure a model using neural networks. To keep it simple, we decided to study only two fields for the model: the number of layers in model construction and batch\_size in model training. In the meantime, we will try to keep other variables unchanged.

Before introducing the design of the experiment, we should explain the two vocabularies: batch size and epoch. First of all, each update on the weights is one iteration. The number of training data contributed to the calculation in this iteration is the batch size. For example, stochastic mode is when batch size is set to one, and so-called batch mode is when the batch size is set to the size of the entire training dataset. Now, we have reached an agreement that, with a batch size set, it takes a certain number of iterations to go through the entire dataset. However, usually one traversal of the entire dataset doesn’t provide enough iterations to give a decent result. Therefore, several traversals of the entire training dataset is recommended. The fancy word “epoch” simply refers to the number of traversals of the entire dataset during the training process.

Back to the design of the experiment, to test the impact of the number of layers to the model performance, which is revealed by the test accuracy, we design three models respectively with **one** convoluted layer, **three** convoluted layers, and **five** convoluted layers.

In the model fitting part, we add another variation of batch size. Selected from the common batch sizes, we choose as fitting batch sizes to test with each model mentioned above. Therefore, there will be in total coming out of the experiment. Furthermore, to evaluate the performance of each outcome, a pair of graphs will be drawn to show the development of accuracy and error for training and validation dataset as the number of epochs (or, simply understood as the total number of iterations) increases.

**Result**

Accuracy for different neural network models and batch sizes

|  | Model 1 | Model 2 | Model3 |
| --- | --- | --- | --- |
| 32 | 0.6194 | 0.9823 | 0.9745 |
| 128 | 0.5012 | 0.9814 | 0.9804 |
| 256 | 0.4909 | 0.9823 | 0.9804 |

Let’s first look at the comparison between different models. Since this comparison is made across the models, the process of how accuracy fluctuates versus epochs is not important to take into consideration here. The best accuracies from the three models are selected:

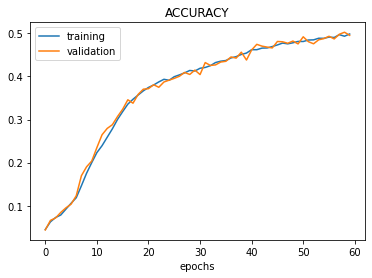
Model1 accuracy (with one convolution layer): 0.6194

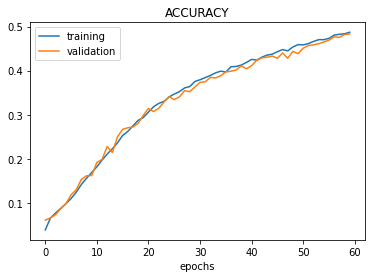
Model2 accuracy (with three convolution layer): 0.9823

Model3 accuracy (with five convolution layer): 0.9804

From this comparison, we see that model 1 with only one convoluted layer performs the worst likely due to the underfitting. Once the number of convoluted layers is added to three, the accuracy of the validation has increased significantly up to 98.23%. When another two layers are added, the accuracy, however, not only doesn’t increase much, but decreases a little bit. From the data, we can conclude that three convoluted layers performs the same, if not better than five layers. As an inference, at least we can say the performance of the neural network doesn’t increase linearly with the increase of convoluted layers, considering that we didn’t test the performance of more than five layers.

The next comparison is about batch sizes. Let’s see how batch size affects the performance of each model. We will use accuracy to demonstrate the model 1 for better illustration. The top left is model 1 accuracy versus epoch with a batch size of 32. Top right is the model with batch size of 128. The graph on the bottom is the model with batch size of 256.





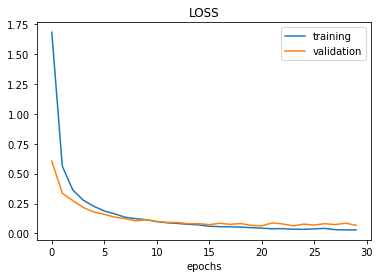
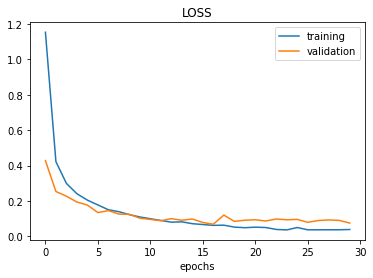
Batch size 32 accuracy: 0.6194

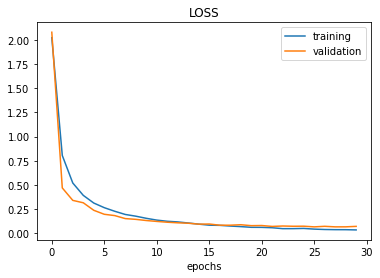
Batch size 128 accuracy: 0.5012

Batch size 256 accuracy: 0.4909

It can be seen that the batch-32 version has more fluctuation during the process of training the data. Batch-128 version eases the fluctuations but the increasing rate of accuracy has slowed down. Batch-256 version further proved this trend showing a smoother curve with more flattened slope. From these three graphs, we have an assumption that the larger the batch size, the less fluctuation in the training process. Also, since the large batch size reduces the total number of iterations if the epoch value remains constant, it slows down the rate to improve the accuracy. This also explains why the less the batch size the better the accuracy while training the model 1 with epoch value set to 60.

For model 2, the loss graph and accuracy graph are similar in showing the results. We arbitrarily choose the loss graph to show the influence of batch size for model 2.





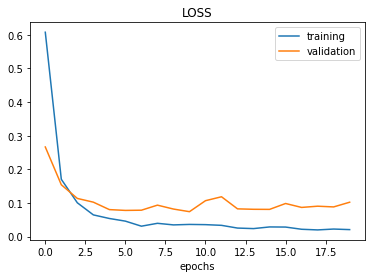
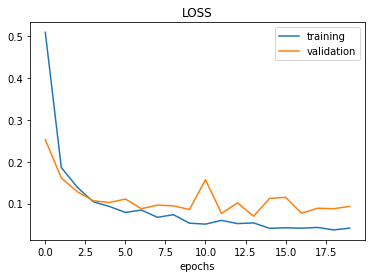
Batch size 32 accuracy: 0.9823

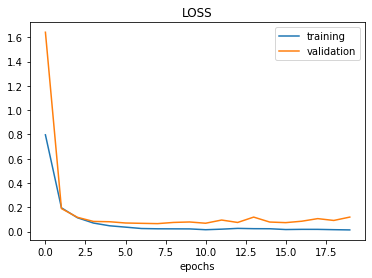
Batch size 128 accuracy: 0.9814

Batch size 256 accuracy: 0.9823

As for how batch size eases the fluctuation, model 2 graphs are showing the similar result as that from model 1. Since the difference between the accuracies of the three batch size choices is so small that becomes negligible, we know that the amount of total iteration cut down by the elevated batch size doesn’t affect the accuracy. Then, it can be inferred that the necessary number of iterations for model 2 is far less than that of model 1. (60 epochs are not enough for model 1 while model 2 may only need about 15 epochs according to the graphs) This gives away another insight apart from what’s been concluded from model1’s graphs: model with more layers require less iterations to train since the complex model captures more information when calculating each training data, or to say, each batch.

For model 3, loss graphs are more dramatic in fluctuations and thus better for illustration.





Batch size 32 accuracy: 0.9745

Batch size 128 accuracy: 0.9804

Batch size 256 accuracy: 0.9804

Model 3 shows a different scenario than the previous models. We can assume that due to some intrinsic features of how the layers are configured in this model, it has more prominent fluctuations than the other two models. Unlike model 1, model 3 has better accuracy when the batch size is large. We are thinking that this could be because the fluctuation for model 3 is so large that it affects the accuracy, and when the fluctuation is balanced by the increased batch size, the accuracy is improved. However, according to [other people's explanation,](https://towardsdatascience.com/exploit-your-hyperparameters-batch-size-and-learning-rate-as-regularization-9094c1c99b55) the small batch size also helps to find a broader local minimum while large batch size provides a smooth convergence to a deep local minimum. According to this author, the small batch size provides an “implicit” regularization. His words proved our observation and inferences from model 2 and model 1’s graphs, but it also makes us aware that small batch size isn't always a bad setting.

As for why model 3 has more fluctuation than model 2, we suspect that it may be because a more complex model captures more information within a batch. However, as stated above, we can’t conclude that this fluctuation always negatively contributes to accuracy.

**Conclusion**

One convoluted layer is so few that the model is underfit, requiring more epochs to train the model. With more layers, the model becomes relatively more complex, and therefore captures more details from each batch. Accordingly, there are likely more variance appearing on the error or accuracy from each batch or epoch during the training process.

The batch size influences the performance of the models differently, but we can still find some general patterns about how batch size affects the accuracy. First of all, smaller batch size contains more variance and results in more fluctuation in error or accuracy across each epoch during the training process (shown from the graphs). Secondly, assuming that the model requires the same number of iterations to train, with a larger batch size, more epochs are needed during the training process.

If the goal is to smooth the variance of the error or accuracy during the training process, increase the batch size, and vice-versa. Since we don’t fully understand the effect of batch size regarding the broad and local minimization, we can’t give suggestions to either only increase the variance or decrease the variance. We have gained a better understanding about how the complexity of the model affects the accuracies. However, it remains elusive in theory about what’s the optimum level of complexity of the model. Only experiment renders the practical answer.

**Acknowledgements:**

Thanks to our professor Linda Sellie for this incredible and fulfilling course of the semester.

Thanks to the TAs who greatly facilitated the course

Acknowledgements to the online forums, especially medium.com, providing clear explanations about neural networks.