THE PENNSYLVANIA STATE UNIVERSITY
SCHREYER HONORS COLLEGE

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

UNDERSTANDING DOUBLE DESCENT BEHAVIOR IN DEEP LEARNING NEURAL NETWORKS.

SHUBHANGAM DUTTA
SPRING 2020

A thesis
submitted in partial fulfillment
of the requirements
for a baccalaureate degree
in Computer Engineering
with honors in Computer Engineering

Reviewed and approved* by the following:

Mehrdad Mahdavi
Assistant Professor of Computer Science and Engineering
Thesis Supervisor

John Morgan Sampson
Assistant Professor of Computer Science and Engineering
Honors Adviser

* Electronic approvals are on file.

ABSTRACT

The statistical understanding of the phenomenon of the U- shaped curve in performance of modern machine learning regimes owes to the existence of the bias–variance tradeoff between the models. However, most modern deep learning networks exhibit a double descent behavior where an increase in certain parameters such as model size, epochs leads to an increase in performance superseding the U-shaped curve past an interpolation threshold. The notion of Effective Model Complexity incorporates all these factors and conjectures a generalized double descent with respect to these factors [1]. This research work builds upon this notion of Effective Model Complexity and tests these various factors across the ResNet v2 model. This research paper also elaborates on the reason behind picking this particular model to test this hypothesis and demonstrates the effect of external parameters such as label noise on this double descent behavior.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to thank my thesis supervisor Dr. Mehrdad Mahdavi for introducing this topic as well as providing me resources to aid my research. I also would like to thank my thesis advisor Dr. John Morgan Sampson in supporting me during my tenure at Penn State as an advisor as well as in reading my thesis.

I would also like to thank my parents Mr. Bibhutosh Dutta and Mrs. Barsha Dutta for encouraging me to join Schreyer Honors College and in keeping me motivated towards my final goal as well as my friends who supported me throughout my time at my University.

Chapter 1

Introduction

Recently there has been an abundant interest in the area of Machine Learning, and how it is being widely used in science and technology because of its cognitive abilities. The cognitive features of these machine learning models amount to the ability of these algorithms to make out of sample predictions based on the training data these models are fed. According to the statistical analyses, to make out of sample predictions by reducing the generalization error, these models calibrate according to the bias (approximation error) vs variance (estimation error) tradeoff by controlling the effective model complexity [2]. The section below describes bias and variance in machine learning models in detail.

1.1 Bias and Variance

Bias (approximation error) is the expressive power of a model and exists because of our assumption about the model space. The model makes these simplifying assumptions to make the target function easier to approximate. Whereas as the variance (estimation error) equals to how well parameters can be estimated. It is the amount that the estimate of the target function will change if different training data were used [3].

Statistically speaking, assume that there is a function with noise $y = f(x) + \in$, $f(x)$ is the true function and $\in$ is the noise on the sample $S = \{(x_1, y_1) \ldots (x_n, y_n)\}$.

Let $g(x;S)$ be the function that approximates the true function and we optimize mean squared error to approximate the function. Then bias and variance can be expressed as,

$$Bias_S[g(x;S)] = E_s[g(x;S)] - f(x)$$

$$Var_S[g(x;S)] = E_s[g(x;S)^2] - E_s[g(x;S)]^2$$

Where $E_s[g(x;S)]$ is the expected error on an unknown sample [4][5]. (Note:

$E_s\left[\left(y - g(x;S)\right)^2\right]$ = $(Bias_S[g(x;S)])^2 + Var_S[g(x;S)] + \sigma^2$ (Irreducible Error))

1.2 Bias-Variance tradeoff and Double Descent

One of the most fundamental principles in classical statistical learning is the bias-variance trade-off. The application of this principle is omnipresent in machine learning models in the under-parameterized regime where the models are adjusted according to this principle. Following this principle, increasing the (effective) model complexity will lead to an increase in the accuracy until it reaches a "sweet spot", once the (effective) model complexity passes this threshold, models overfit with the variance term dominating the test error, and this leads to a decrease in performance as shown in Figure 1.

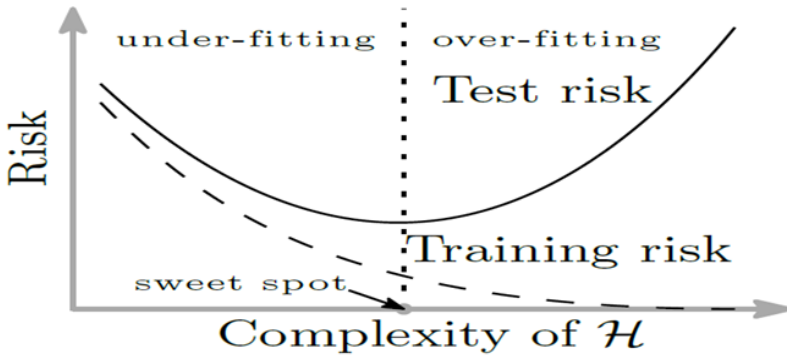

Figure 1: Classical U-shaped Curve (Risk vs Model Complexity) [2]

However, most deep learning neural networks tend to use overparametrized models to achieve near-zero training accuracy and are still able to decrease the test risk and perform much better on many tasks than even smaller models [2]. In fact, Figure 2 below represents the behavior of machine learning models in overparametrized regime and articulates the contradictory behavior of these models to the above principle. It is evident from the plot that past a certain interpolation threshold, increasing the (effective) model complexity leads to a decrease in test risk.

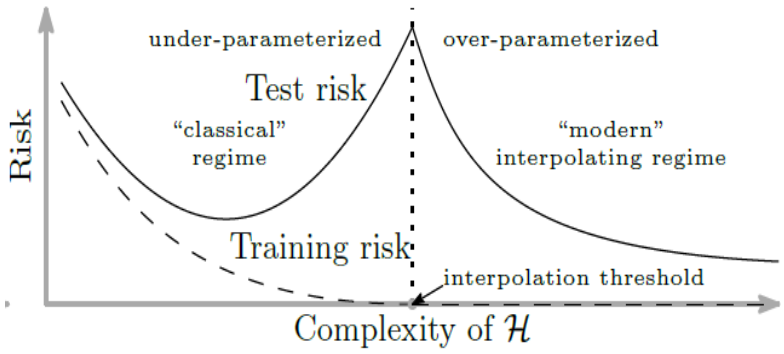

Figure 2: Double Descent Curve (Risk vs Model Complexity) [2]

This double descent behavior occurs across a variety of tasks, architectures, and optimization methods. In this research work, we conjecture the various factors such as model size or epochs that lead to this behavior in the notion of Effective model complexity (first described by Nakkiran et al. [1]) and verify the performance of these functions in the ResNet v2 model.

Chapter 2

Effective Model Complexity

The notion of Effective model complexity incorporates the various functions which affect this double descent behavior in the modern machine learning regimes. It was first defined in the research published by Nakkiran et al. [1]. According to Nakkiran et al., the effective model complexity can be defined as follows:

Let T be any training procedure that takes as input a set $S = \{(x_1, y_{1)} \dots (x_n, y_n)\}$ of labeled training samples and outputs a classifier $T(S)$ mapping data to labels. The Effective Model complexity of T with respect to the distribution D to be the maximum number of samples n on which T achieves on average close to zero training error.

$$EMC_{D,\epsilon}(T) = \max\{n \mid E_{S\sim D^n}\left[Error\left(T(S)\right)\right] \leq \epsilon$$

Here D is the distribution, ϵ is a parameter whose value is greater than 0 and $Error_S(M)$ is the mean error of the model M on the train sample S.

According to Nakkiran et al., the deep double descent behavior in deep learning neural networks can be categorized into 3 different regimes:

If $EMC_{D,\epsilon}(T) << n$, the model will be in under-parameterized regime and any increases in effective complexity will **decrease** the test error.

If $EMC_{D,\epsilon}(T)) >> n$, the model will be in over-parameterized regime any increase in effective complexity will **decrease** the test error.

If $EMC_{D,\epsilon}(T) \approx n$, the model will be in critically-parameterized regime and any perturbation of T that increases its effective complexity might **decrease or increase** the test error.

Where T is the neural-network-based training procedure and D is any natural data distribution [1]. **Note:** While n here accounts for the sample size used for training, it is sometimes confused with only the number of data points. While this case can be true for single output, for multiclass classifications, the n accounts for the number of data points * number of outputs.

The advantage of utilizing this notion over model complexity is that model complexity generally accounts for only the model parameters (or size) and architecture while evaluating the test risk of the classifier across a varying number of parameters. However, the notion of effective model complexity not only incorporates the model parameters and architecture of the classifier but also the training procedure and data distribution used during the evaluation of the test risk.

In this research work, we build upon this hypothesis of effective model complexity and conduct various experiments to verify these factors that affect the double descent behavior on the ResNet v2 model. But before conducting these experiments, it is mandatory to justify the reason for picking this particular neural network. In Chapter 3 we briefly elaborate on properties of ResNets and provide justification for using this model to conduct experiments.

Chapter 3

Why ResNets?

In recent years, acknowledging the double descent behavior in neural networks, many practitioners used deeper networks to train their models to increase the performance by training them in overparameterized regimes. However, while training these deeper networks, instead of observing an enhancement in performance, the problem of accuracy degradation was observed i.e. adding more layers to the network either made the accuracy value to saturate or it abruptly started to diminish. The reason behind this accuracy degradation is the vanishing gradient effect which is observed prominently in deeper networks.

During backpropagation, the error is calculated, and gradient values are determined. The gradients are then sent back to the hidden layers and the weights are updated accordingly. This process of gradient determination is continued between the subsequent hidden layers until the input layer is reached. Throughout this process, the gradients gradually decrease and eventually diminish as they reach the end of the network. This leads to the weights of the initial layers being updated very slowly or they remain almost unchanged i.e. the initial layers of the network don't learn effectively. Hence, the accuracy during training either starts to degrade or diminish to a certain value. Although this hindrance was minimized using the normalized initialization of weights, the performance of these deep learning models still didn't improve [7].

To diminish this effect of vanishing gradients, ResNet uses the notion of "identity shortcut connection" which involves skipping one or more layers, as shown in the Figure 3.

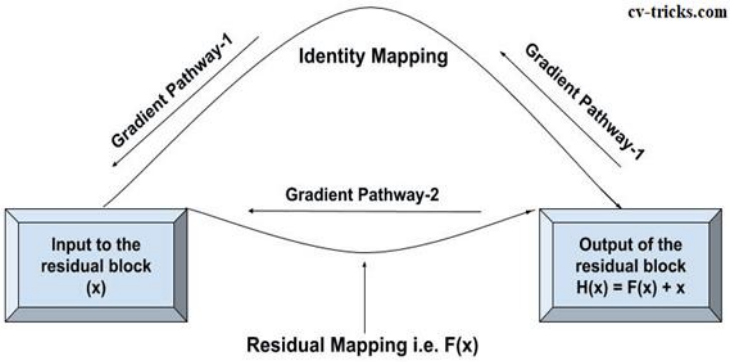

Figure 3: Intuitive representation of gradient pathways in ResNet

Note: For the sake of simplicity, the weight layers and activation function are not included in this diagram.

As you can see in the above figure, during backpropagation, the gradient can pass from the residual mapping way (and eventually suffer vanishing gradient effect) or skip the residual mapping and transit from the identity mapping i.e. through gradient pathway-1. Transiting from this path, the gradients don't have to encounter any weight layer, hence, there won't be any change in the value of computed gradients after they reach the initial layers, thereby helping them to learn the weights accurately [7].

In general, in a multilayer neural network, the learning algorithm will tune all of the weights to fit the training data, typically using versions of optimizers like Adam, with backpropagation to compute partial derivatives [8]. Since the calculation of partial derivatives is crucial for the loss function, the effect of identity mapping eradicates the effect of vanishing gradient present in other neural networks. This makes ResNet fit for evaluating this behavior. Additionally, since we are using the Cross-entropy function to evaluate the train error, this loss function makes the calculation of partial derivatives easier as it reduces the optimization problem to be a convex function to be computed by optimizers like Adam or SGD [8].

The ResNet Model (imported from Keras) comes in 2 versions, the principal difference is that in version 2, batch normalization and ReLU activation comes before 2D convolution and the uses stack of 1 × 1 - 3 × 3 - 1 × 1 BN-ReLU-Conv2D [9]. Because of the slight improvement, will use ResNet v2 for our experiments. In this Chapter 4 we will elaborate on the experiments performed on ResNet v2.

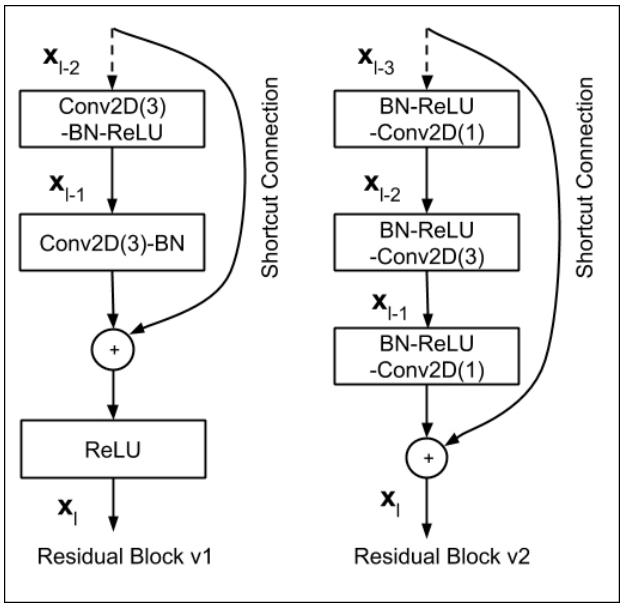

Figure 4: ResNet v1 and ResNet v2 Model [10]

Chapter 4

Experiments & Analysis

4.1 Experiments to verify Model Wise Double Descent Behavior

Experimental Setup

To verify the model wise double descent behavior, ResNet v2-20 model was used, and the optimizer used for this model was Adam [8]. The learning rate was set to be constant at 0.001 and the parameter width was varied. The loss function used was categorial-crossentropy and data augmentation was also done to achieve better accuracy. The dataset used for this experiment was CIFAR-10 which was subsampled into half for validation. The details of the experimental setup are summarized below:

Table 1: Experimental Setup used to test Model Wise Behavior

Model	ResNet v2 -20
Parameter Width	1-64
Optimizer	Adam
Learning Rate	0.001
Loss function	Categorial Crossentropy
Dataset	CIFAR-10
Label Noise ratio	0
Kernel size	3
Strides	1
Padding	Same
Regularization	Disabled
Data augmentation	Yes

The figures below give a detailed model summary of the Resnetv2-20 for parameter width = 8. The ResNet v2 model uses Batch normalization and ReLu activation and comprises of a 20 layer deep Convolutional layers.

Layer (type)	Output Shape	Param #	Connected to
input_48 (InputLayer)	(None, 32, 32, 3)	0	
conv2d_493 (Conv2D)	(None, 32, 32, 8)	224	input_48[0][0]
batch_normalization_425 (BatchN	(None, 32, 32, 8)	32	conv2d_493[0][0]
activation_424 (Activation)	(None, 32, 32, 8)	0	batch_normalization_425[0][0]
conv2d_494 (Conv2D)	(None, 32, 32, 8)	72	activation_424[0][0]
batch_normalization_426 (BatchN	(None, 32, 32, 8)	32	conv2d_494[0][0]
activation_425 (Activation)	(None, 32, 32, 8)	0	batch_normalization_426[0][0]
conv2d_495 (Conv2D)	(None, 32, 32, 8)	584	activation_425[0][0]

Figure 5: Initial Layers of ResNet v2-20 Width Parameter = 8

batch_normalization_442 (BatchN	(None, 8, 8, 64)	256	conv2d_513[0][0]
activation_441 (Activation)	(None, 8, 8, 64)	0	batch_normalization_442[0][0]
conv2d_514 (Conv2D)	(None, 8, 8, 128)	8320	activation_441[0][0]
add_139 (Add)	(None, 8, 8, 128)	0	add_138[0][0] conv2d_514[0][0]
batch_normalization_443 (BatchN	(None, 8, 8, 128)	512	add_139[0][0]
activation_442 (Activation)	(None, 8, 8, 128)	0	batch_normalization_443[0][0]
average_pooling2d_23 (AveragePo	(None, 1, 1, 128)	0	activation_442[0][0]
flatten_23 (Flatten)	(None, 128)	0	average_pooling2d_23[0][0]
dense_23 (Dense)	(None, 10)	1290	flatten_23[0][0]

Figure 6: Final Layers of ResNet v2-20 Width Parameter = 8

Results & Analysis

The test and train errors evaluated after varying the parameter width of ResNet v2-20 is summarized below. The training across varying parameter widths were simulated multiple times and the errors were averaged (and smoothened) across a fixed number of epochs. The resulting plots are given below. The resulting code is given in Appendix A.

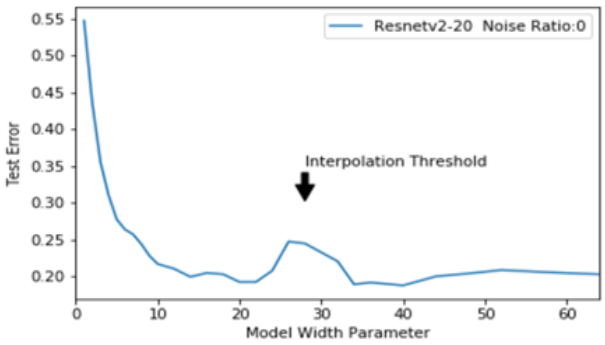

Figure 7: Test Error after multiple simulations (averaged & smoothened) vs Model Size

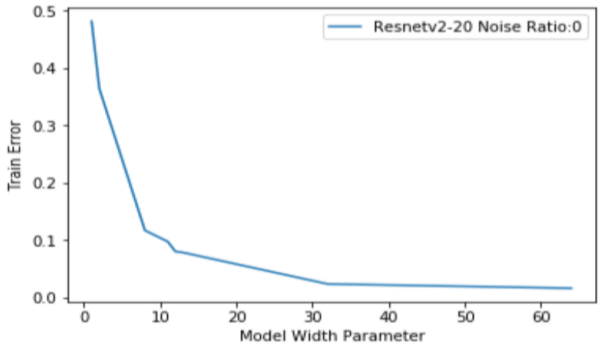

Figure 8: Train Error after multiple simulations (averaged & smoothened) vs Model Size

As observed the Resnet v2-20 model exhibits a parameter wise deep double descent behavior. The model is able to reduce the test error until the test error reaches a “sweet spot” after which the test error starts to increase and reaches the interpolation threshold. The resulting plots indicate an

interpolation threshold at approximately parameter width ≈ 27. The training plots also indicate something similar as the training error reaches close to zero at parameter width ≈ 32. Since this plot use averaging and smoothening of multiple simulations, we believe that the slight discrepancy between the exact parameter width where the interpolation occurs because of this.

The model is in under-parameterized regime for the model parameter width approximately ranging from 1 to 25, followed by a critically parameterized regime for parameter widths ranging from 26 to 35 and eventually follows into an over-parametrized regime after that. After reaching the interpolation threshold, the test error starts to decrease while the training error remains almost stagnant.

When the model size (parameter width) is much smaller compared to the sample size, classical statistical arguments imply that the training risk is close to the test risk. Thus, for small models increasing the parameter width increases the number of trainable parameters which yields improvements in both the training and test risks. However, as the number of parameters approach the sample size (number of data points * outputs) after the increment in the parameter width, parameters not present or only weakly present in the data are forced to fit the training data nearly perfectly. This results in classical over-fitting as predicted by the bias-variance trade-off. To the right of the interpolation threshold, multiple models are able to minimize the training loss to almost zero error. While this might not always guarantee an enhancement in test accuracy, the intuition is that with the increase in the number of interpolating models will help us constructively make better approximations in finding the function with the small categorial cross-entropy. This intuition is confirmed from Figures 7 and 8.

4.2 Experiments to verify Epoch Wise Double Descent Behavior

Experimental Setup

For the purpose of testing the epoch wise behavior, the ResNet v2-20 model was used (imported from Keras [9].) The details of the experimental setup are summarized below:

Table 2: Experimental Setup used to test Epoch Wise Behavior

Model	ResNet v2 -20
Parameter Width	8,16,64
Optimizer	Adam
Learning Rate	0.001
Loss function	Categorial Crossentropy
Dataset	CIFAR-10
Label Noise ratio	0/0.2
Kernel size	3
Strides	1
Padding	Same
Regularization	Disabled
Data augmentation	Yes

Results & Analysis

The test error of the ResNet v2-20 models was tested across various parameter widths and the results of a few models are given below:

Parameter width - 8

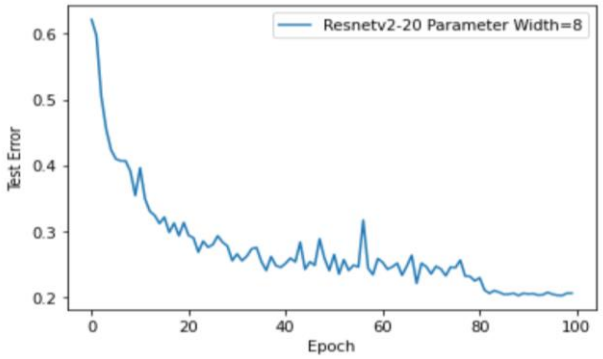

Figure 9: Test Error after single simulation vs Epochs

Parameter width – 64

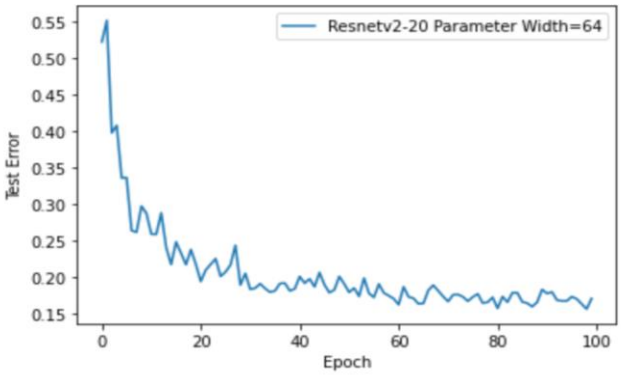

Figure 10: Test Error after single simulation vs Epochs

As observed from the results, the epoch wise double descent behavior is barely visible in these models with no label noise. The test errors almost show a monotonic behavior as the test error decreases

with an increase in epochs, for a fixed parameter width. The results of the average and smoothening of multiple simulations of these test results for ResNet v2-20 model (parameter width = 8, noise ratio = 0 %) validates this observation even further.

Parameter Width- 16

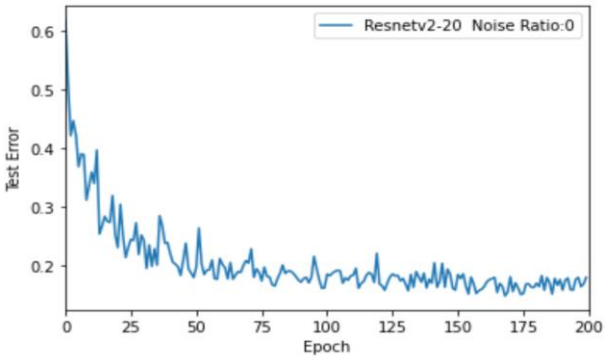

Figure 11: Test Error after single simulation vs Epochs

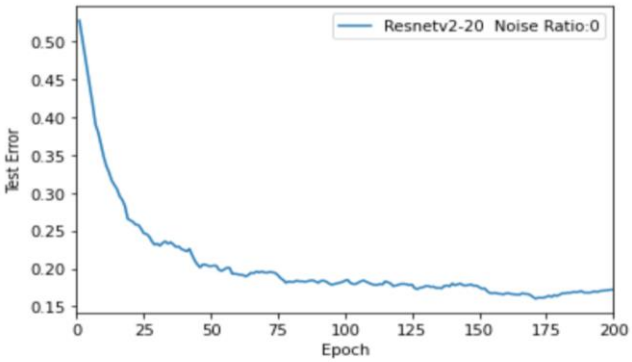

Figure 12: Test Error after multiple simulations (averaged & smoothened) vs Epochs

To evaluate the epoch wise behavior by varying the label noise on ResNet v2-20 Parameter width = 16, the trained with noisy labels, the noisy dataset was split into two halves and we perform cross-validation: training on a subset and testing on the other. The noise pattern was symmetric, and the

label noise used for the testing was 0.2. The figure below represents how the noise ratio affects test accuracy, label precision, and label recall on manually corrupted CIFAR-10.

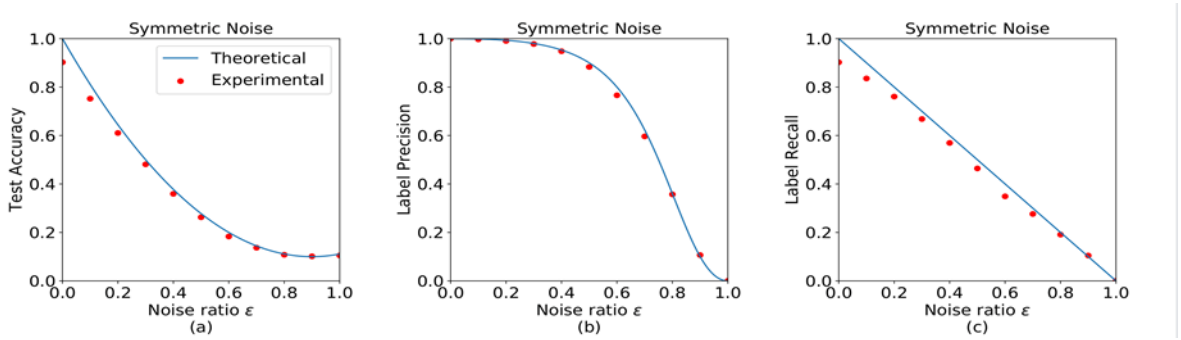

Figure 13: How Noise ratio affected Test accuracy, Label precison and Label Recall on CIFAR-10

The results of testing different models with a Noise Ratio of 0.2 are given below:

Noise Ratio 0.2

Parameter Width – 16

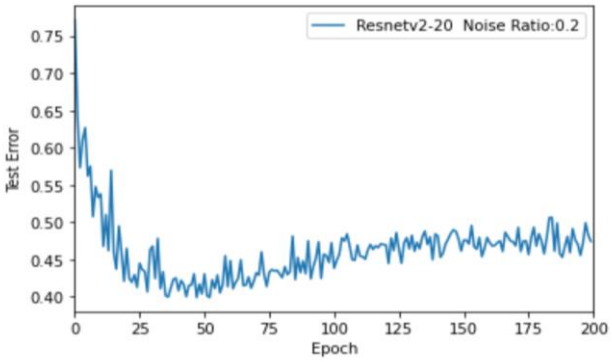

Figure 14: Test Error after single simulation vs Epochs

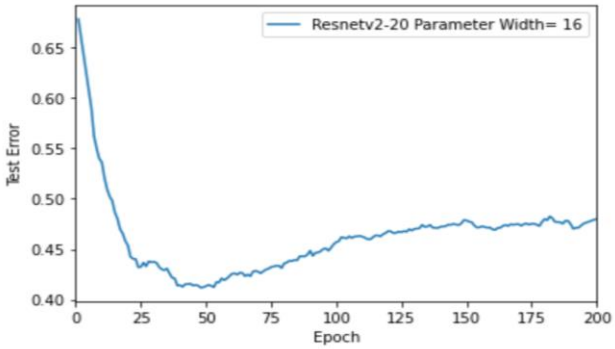

Figure 15: Test Error after multiple simulations (averaged & smoothened) vs Epochs

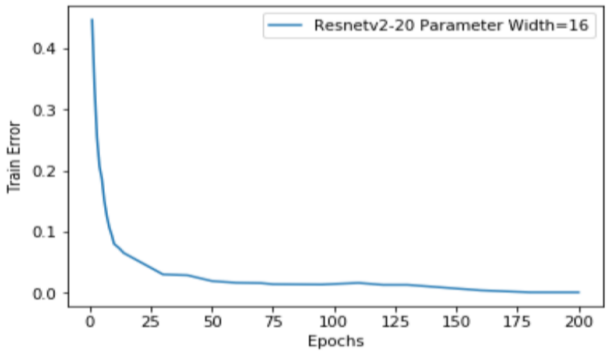

Figure 16: Train Error after multiple simulations (averaged & smoothened) vs Epochs

As observed from the above plots, the test error for the ResNetv2-20 Parameter width=16 decreases with epochs (Noise Ratio = 0.2) when under the under-parameterized, reaches a sweet spot and then starts to overfit until around 150 epochs, with almost a constant test error after that.

Noise Ratio 0.2

Parameter Width - 64

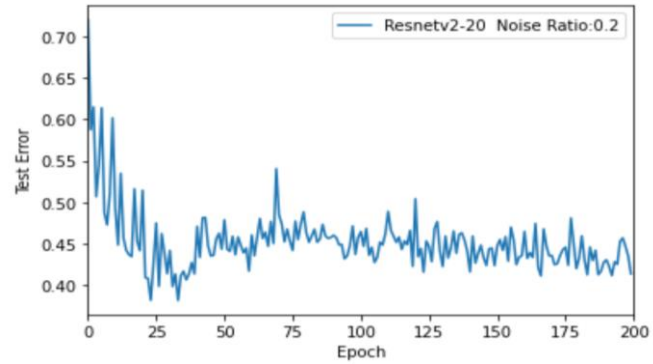

Figure 17: Test Error after single simulation vs Epochs

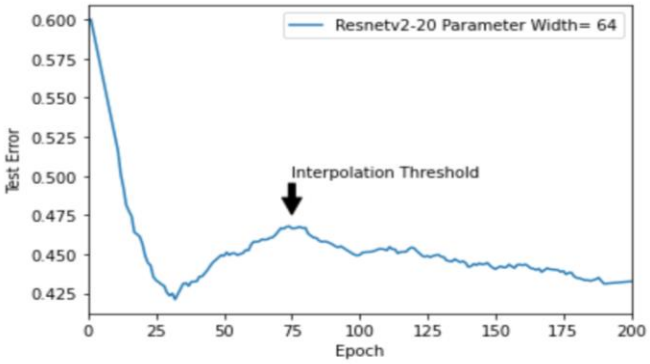

Figure 18: Test Error after multiple simulations (averaged & smoothened) vs Epochs

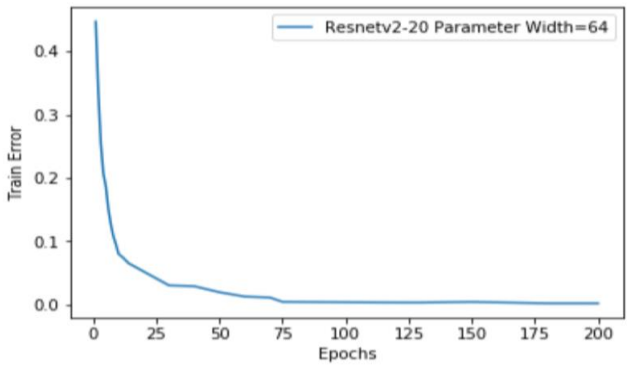

Figure 19: Train Error after multiple simulations (averaged & smoothened) vs Epochs

We can notice in the above plots, the model possesses a deep double descent behavior, where the model reaches the interpolation threshold at around 75 epochs as observed from the above plot. The test error is observed to decrease past this threshold.

The amalgamation of test errors of these two models with different parameter width is plotted below.

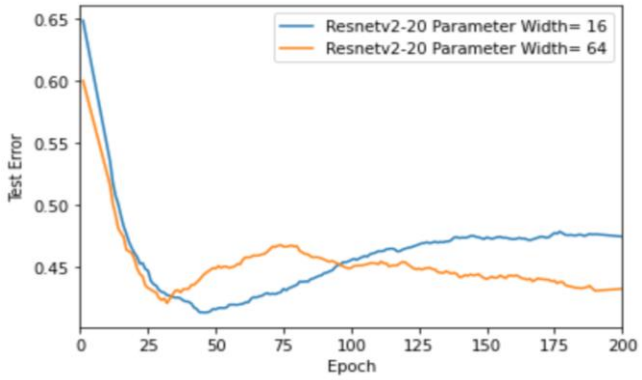

Figure 20: Test Error after multiple simulations (averaged & smoothened) vs Epochs

Based on our observations from these experiments, the medium-sized model's test error tends to decrease because the model tries to decrease the generalization error until it reaches its sweet spot, after the sweet spot the model starts to overfit for a large number of epochs until it reaches an almost constant rate. For larger models, the epoch wise test behavior exhibits a double descent behavior for some label noise ratio.

However, models that were trained with 0 noise ratio tend to exhibit monotonic behavior even after varying parameter widths for a large number of epochs. The intuition behind this (also mentioned in Narikkam 2020) is that around interpolation thresholds, there is usually one model that fits the train data this interpolating model is very sensitive to noise since noisy labels can affect the model's calculated

parameters and thereby reducing its accuracy which is evident on medium-sized models [1]. Having no label noise will, therefore, ensure the model doesn't go through much perturbation making double descent behavior difficult to observe.

While epochs increase, larger models will eventually reach overparametrized regions, which leads to these models having a greater number of models that fit the train set. With an increase in epochs, the larger model is able to find even more interpolating models. Eventually one of these models will start to perform well (by absorbing the noise) using the optimizers like Adam which eventually leads to greater accuracy [1].

Note: Small models weren't plotted since they tend to be in under parameterized regime for a large number of epochs masking any deep double descent behavior.

4.3 Experiment to verify Sample wise Non-Monotonicity

Experimental Setup

Using the same Resnetv2-20 model sample wise test error was produced across 25000 samples and 5000 samples. L2-regularization was not used for this evaluation. According to the research work , double descent phenomenon is largely observed for unregularized or under-regularized models in practice. For non-isotropic Gaussian covariates, we can achieve sample-wise monotonicity with a regularizer (that depends on the population covariance matrix of data.) In order to avoid conflict with the effect of regularization, this experiment uses regularizer-less setting. The results produced are plotted below. The details of the experimental setup are summarized below:

Table 3: Experimental Setup used to test Sample wise Non-Monotonicity

Model	ResNet v2 - 20
Parameter Width	1 - 64
Optimizer	Adam
Learning Rate	0.001
Loss function	Categorial Crossentropy
Dataset	CIFAR-10
Label Noise ratio	0
Kernel size	3
Strides	1
Padding	Same
Regularization	Disabled
Data augmentation	Yes

Results & Analysis

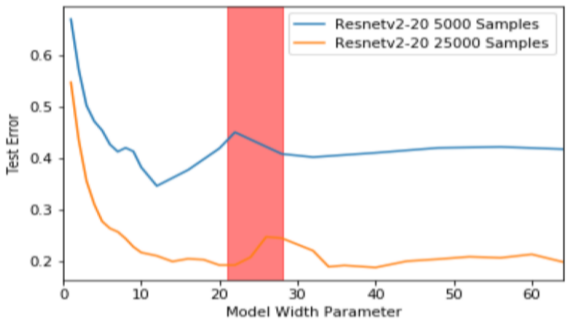

Figure 21: Test Error after multiple simulations (averaged & smoothened) vs Model Width Parameter for 5000 and 25000 Samples

As evident from the plots, with a greater number of samples, the test error has decreased. However, the interpolation threshold for the model trained on a larger sample size is shifted to the right as compared to the model trained on smaller sample size.

While for the experiment on Resnetv2-20, the test error is still less for a model trained on a larger sample size, it is quite illustrative that for certain sample sizes slightly close to the sample size of 25000, more data can have an opposite effect by decreasing accuracy. In fact, there are certain experiments conducted in the past (Appendix B) that have demonstrated this intuition.

Chapter 5

Discussion & Evaluation

There has been a multitude of experiments conducted in the field of Deep Learning Neural Networks that demonstrate the presence of Double Descent behavior in the performance of these models with respect to model complexity [1][2][11][12][13]. The notion of effective model complexity sheds light on the interaction between optimization algorithms, model size, and test performance with this behavior and helps reconcile some of the competing intuitions about them [1]. The experiments conducted in this research-work demonstrate certain interesting results that build upon the hypothesis stated by Nakkiran et al. and affect our understanding of the significance of various intrinsic and extrinsic parameters such as model size, sample size, noise, and epochs in the training of the model.

The experimental results demonstrate that the test performance across these models isn't just a function of model complexity (or size), but it can also be affected by the data distribution, the number of epochs, and the label noise. In section 4.2, we demonstrate that with minor label noise the Resnet v2 model exhibits a double descent behavior when tested against the number of epochs. This result demonstrates that for a certain number of epochs (around the interpolation threshold), an increase in epochs results in the degradation of test performance for a certain model.

In section 4.3, we demonstrate a similar yet even more drastic analogy that affects our understanding of the amount of sample size required for training. According to this analogy, for a certain amount of parameter widths (or model complexity) the model shows little or no improvements in increasing the sample size. In fact, in Appendix B, we demonstrate an experiment conducted by Nakkiran et al. that shows that for certain embedding dimensions, an increase in samples leads to an overall decrease in test loss [1]. This result stands contradictory to the popular notion that more data will always improve performance.

Chapter 6

Conclusion

This paper introduces the concept of bias and variance and how the bias-variance tradeoff determines the classical U-shaped curve observed in the performance of various machine learning models across a range of model complexity in the under-parameterized and before the critically-parameterized regime. This research paper then elaborates upon the double descent behavior initially demonstrated in Belkin et al. where modern deep learning neural networks tend to perform well in overparameterized regimes [2]. This double descent behavior is conjectured under the notion of effective model complexity as defined in Nakkiran et al. and in our experiments, we test the factors that affect this double descent behavior [1]. To evaluate these factors such as model & epoch wise double descent behavior and sample non-monotonicity we chose ResNet v2 (imported from Keras) in our experiments [8].

The research builds upon and emphasizes the findings of Nakkiran et al. in their paper by providing experimental results and theoretical analysis of these results. However, this research work differed from Nakkiran et al.'s work in various aspects such as the experimental setup, model used, training samples, noise encoding method, and the justification of establishment of a particular setup [1]. Additionally, the model chosen for this experiment was also discussed in detail and the justification for choosing it for these experiments was provided. The noise ratio encoded to the labels were also discussed by demonstrating how the label noise ratio encoded, affected the test accuracy, label precision, and label recall in different models trained on corrupted CIFAR -10 database.

Appendix A

Code for our Experiments

These experiments were conducted on Google Colab Pro with Hardware accelerator set as “GPU” and the runtime shape set as “High-RAM”. The base of these experiments was based on the setup created on https://github.com/chenpf1025/noisy_label_understanding_utilizing and the Resnet v2 model was imported from it’s implementation on Keras [9].

To vary the number of parameter width in the ResNet v2 model, the number of input filters in the ResNet v2 model was varied according to the input supplied. This in turn generated models of various sizes.

A.1 The Code Setup

```
from keras.callbacks import Callback, LearningRateScheduler
from keras import optimizers
from keras.preprocessing.image import ImageDataGenerator
from keras.utils import multi_gpu_model
from sklearn.metrics import accuracy_score
import data
import numpy as np
import os
#os.environ['CUDA_VISIBLE_DEVICES']='1'

import argparse
""" parameters """
noise_ratio = 0 #args.noise_ratio
noise_pattern = 'sym' #args.noise_pattern #'sym' or 'asym'
batch_size = 128
epochs = 50
save_dir = 'Theory'
network = 'ResNet20'
if not os.path.isdir(save_dir):
```

```
    os.makedirs(save_dir)
filepath = os.path.join(save_dir,network+'.h5')
print('\n#########################################\n noise_ratio: %.2f noise
_pattern: %s\n#########################################\n'
      %(noise_ratio,noise_pattern))
#########################################################################
#########################################################
""" Data preparation """
x_train, y_train, _, _, x_test, y_test = data.prepare_cifar10_data(data_di
r='/content/drive/My Drive/noisy_label_understanding_utilizing-
master/data/cifar-10-batches-py')#data/cifar-10-batches-py')
y_train_noisy = data.flip_label(y_train, pattern=noise_pattern, ratio=nois
e_ratio, one_hot=True)
input_shape = list(x_train.shape[1:])
n_classes = y_train.shape[1]
n_train = x_train.shape[0]
np.save('y_train_total.npy',y_train)
np.save('y_train_noisy_total.npy',y_train_noisy)
clean_index = np.array([(y_train_noisy[i,:]==y_train[i,:]).all() for i in
range(n_train)])# For tracking only, unused during training
noisy_index = np.array([not i for i in clean_index])
# Generator for data augmantation
datagen = ImageDataGenerator(width_shift_range=4./32,  # randomly shift im
ages horizontally (fraction of total width)
                             height_shift_range=4./32,  # randomly shift i
mages vertically (fraction of total height)
                             horizontal_flip=True
                             )  # randomly flip images
#########################################################################
#########################################################
""" Build model """

val_idx = np.array([True for i in range(n_train)])
val_idx_int = np.array([i for i in range(n_train) if val_idx[i]]) # intege
r index
np.random.shuffle(val_idx_int)
n_val_tenth = int(np.sum(val_idx)/10)
val1_idx = val_idx_int[:n_val_tenth] # integer index
val2_idx = val_idx_int[n_val_tenth:] # integer index

#checkpoint = ModelCheckpoint(filepath=filepath, monitor='val_acc', verbos
e=1, save_best_only=False)

class Noisy_acc(Callback):

    def on_epoch_end(self, epoch, logs={}):

        idx = val2_idx[np.random.choice(len(val2_idx),1000)] # train on th
e first half while test on the second half

        predict = self.model.predict(x_train[idx,:])
        predict = np.argmax(predict,axis=1)
```

```
        _acc_mix = accuracy_score(np.argmax(y_train_noisy[idx,:],axis=1),
predict)
        _acc_clean = accuracy_score(np.argmax(y_train_noisy[idx,:][clean_i
ndex[idx],:],axis=1), predict[clean_index[idx]])
        _acc_noisy = accuracy_score(np.argmax(y_train_noisy[idx,:][noisy_i
ndex[idx],:],axis=1), predict[noisy_index[idx]])

        print("- acc_mix: %.4f - acc_clean: %.4f - acc_noisy: %.4f\n" % (_
acc_mix, _acc_clean, _acc_noisy))
        return
noisy_acc = Noisy_acc()

def lr_schedule(epoch):
    # Learning Rate Schedule
    lr = 1e-3
    print('Learning rate: ', lr)
    return lr
lr_callback = LearningRateScheduler(lr_schedule)


# Define optimizer and compile model
optimizer = optimizers.Adam(lr_schedule(0))#, beta_1=0.9, beta_2=0.999, ep
silon=1e-08, decay=0.0)

for s in range(1,64):
  model = create_model(input_shape=input_shape, classes=n_classes, name=ne
twork,k=s, architecture=network)
  model.summary()

#parallel_model = multi_gpu_model(model, gpus=2)
  model.compile(optimizer=optimizer, loss='categorical_crossentropy', metr
ics = ['accuracy'])


##########################################################################
#####################################################

  results = model.fit_generator(datagen.flow(x_train[val1_idx,:], y_train_
noisy[val1_idx,:], batch_size = batch_size),
                                epochs = epochs,
                                validation_data=(x_train[val2_idx,:], y_tra
in_noisy[val2_idx,:]),
                                callbacks=[noisy_acc, lr_callback])



  rr=[]
  for t in results.history['val_accuracy']:
    rr.append(1-t)
  test_err7[s]=rr

  rs=[]
  for t in results.history['accuracy']:
```

```
    rs.append(1-t)
  train_err[s]=rs

    rl=[]
  for t in results.history['loss']:
    rl.append(1-t)
  t_loss[s]=rl



  scores = model.evaluate(x_test, y_test, verbose=1)
  print('Test loss:', scores[0])
  print('Test accuracy:', scores[1])
  test_e[s] = 1 - scores[1]


  y_pred = np.argmax(model.predict(x_train[val2_idx,:]), axis=1)
  y_true_noisy = np.argmax(y_train_noisy[val2_idx,:],axis=1)
  select_idx =  val2_idx[y_pred==y_true_noisy]# integer index
  print('Noisy Validation Accuracy: %.4f'%(len(select_idx)/len(y_pred)))
  print('Label Precision: %.4f'%(np.sum(clean_index[select_idx])/len(selec
t_idx)))
  print('Label Recall: %.4f'%(np.sum(clean_index[select_idx])/np.sum(clean
_index[val2_idx])))

  y_test_pred = np.argmax(model.predict(x_test), axis=1)
  print('Noise ratio: %.2f'%noise_ratio)
```

A.2 The ResNet v2 model

The create_model function was used to create an instance of Resnet v2-20 as seen below.

```
def create_model(input_shape, classes, name,k, architecture='ResNet20'):
    if architecture == 'ResNet20':
      return resnet_v2(input_shape,k, depth=20, num_classes=classes)
```

The ResNet v2 model used in our experiments is given below:

```
def resnet_layer(inputs,
                 num_filters=16,
                 kernel_size=3,
                 strides=1,
                 activation='relu',
                 batch_normalization=True,
                 conv_first=True):
```

```
    """2D Convolution-Batch Normalization-Activation stack builder
    # Arguments
        inputs (tensor): input tensor from input image or previous layer
        num_filters (int): Conv2D number of filters
        kernel_size (int): Conv2D square kernel dimensions
        strides (int): Conv2D square stride dimensions
        activation (string): activation name
        batch_normalization (bool): whether to include batch normalization
        conv_first (bool): conv-bn-activation (True) or
            bn-activation-conv (False)
    # Returns
        x (tensor): tensor as input to the next layer
    """
    conv = Conv2D(num_filters,
                  kernel_size=kernel_size,
                  strides=strides,
                  padding='same',
                  kernel_initializer='he_normal',
                  #kernel_regularizer=l2(1e-4)
                  )


    x = inputs
    if conv_first:
        x = conv(x)
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
    else:
        if batch_normalization:
            x = BatchNormalization()(x)
        if activation is not None:
            x = Activation(activation)(x)
        x = conv(x)
    return x


def resnet_v2(input_shape,k, depth, num_classes=10):
    """ResNet Version 2 Model builder [b]
    Stacks of (1 x 1)-(3 x 3)-(1 x 1) BN-ReLU-Conv2D or also known as
    bottleneck layer
    First shortcut connection per layer is 1 x 1 Conv2D.
    Second and onwards shortcut connection is identity.
```

```
    At the beginning of each stage, the feature map size is halved (downsa
mpled)
    by a convolutional layer with strides=2, while the number of filter ma
ps is
    doubled. Within each stage, the layers have the same number filters an
d the
    same filter map sizes.
    Features maps sizes:
    conv1  : 32x32,  16
    stage 0: 32x32,  64
    stage 1: 16x16, 128
    stage 2:  8x8,  256
    # Arguments
        input_shape (tensor): shape of input image tensor
        depth (int): number of core convolutional layers
        num_classes (int): number of classes (CIFAR10 has 10)
    # Returns
        model (Model): Keras model instance
    """
    if (depth - 2) % 9 != 0:
        raise ValueError('depth should be 9n+2 (eg 56 or 110 in [b])')
    # Start model definition.
    num_filters_in = k
    num_res_blocks = int((depth - 2) / 9)

    inputs = Input(shape=input_shape)
    # v2 performs Conv2D with BN-
ReLU on input before splitting into 2 paths
    x = resnet_layer(inputs=inputs,
                     num_filters=num_filters_in,
                     conv_first=True)

    # Instantiate the stack of residual units
    for stage in range(3):
        for res_block in range(num_res_blocks):
            activation = 'relu'
            batch_normalization = True
            strides = 1
            if stage == 0:
                num_filters_out = num_filters_in * 4
                if res_block == 0:  # first layer and first stage
                    activation = None
                    batch_normalization = False
            else:
```

```
            num_filters_out = num_filters_in * 2
            if res_block == 0:  # first layer but not first stage
                strides = 2    # downsample

        # bottleneck residual unit
        y = resnet_layer(inputs=x,
                         num_filters=num_filters_in,
                         kernel_size=1,
                         strides=strides,
                         activation=activation,
                         batch_normalization=batch_normalization,
                         conv_first=False)
        y = resnet_layer(inputs=y,
                         num_filters=num_filters_in,
                         conv_first=False)
        y = resnet_layer(inputs=y,
                         num_filters=num_filters_out,
                         kernel_size=1,
                         conv_first=False)
        if res_block == 0:
            # linear projection residual shortcut connection to match
            # changed dims
            x = resnet_layer(inputs=x,
                             num_filters=num_filters_out,
                             kernel_size=1,
                             strides=strides,
                             activation=None,
                             batch_normalization=False)
        x = keras.layers.add([x, y])

    num_filters_in = num_filters_out

# Add classifier on top.
# v2 has BN-ReLU before Pooling
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = AveragePooling2D(pool_size=8)(x)
y = Flatten()(x)
outputs = Dense(num_classes,
                activation='softmax',
                kernel_initializer='he_normal')(y)
# Instantiate model.
model = Model(inputs=inputs, outputs=outputs)
return model
```

Appendix B

Sample wise Non-Monotonicity in a Transformer

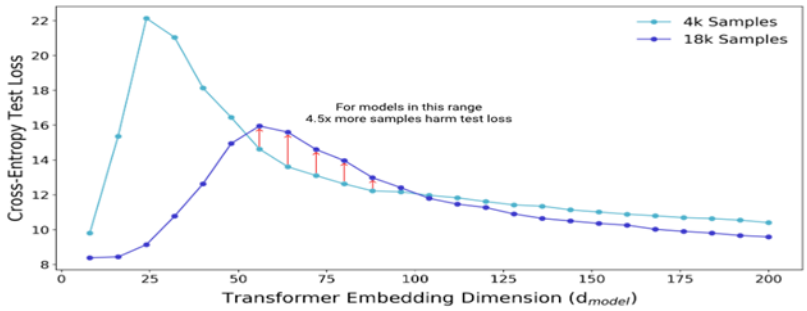

Figure 22: Transformer Embedding Dimension vs Cross-Entropy Test Loss

Nakkiran et al. conducted an experiment where test loss (per-token perplexity) as a function of Transformer model size (embedding dimension dmodel) on language translation (IWSLT‘14 German-to-English). The curve for 18k samples is generally lower than the one for 4k samples, but also shifted to the right, since fitting 18k samples requires a larger model. Thus, for some models, the performance for 18k samples is worse than for 4k samples for width parameter ranging from 50 to 100 [1].

BIBLIOGRAPHY

[1] P. Nakkiran, G. Kaplun, Y. Bansal, T. Yang, B. Barak and I. Sutskever, "Deep Double Descent: Where Bigger Models and More Data Hurt", arXiv.org, 2020. [Online]. Available: https://arxiv.org/abs/1912.02292.

[2] M. Belkin, D. Hsu, S. Ma and S. Mandal, "Reconciling modern machine learning practice and the bias-variance trade-off", arXiv.org, 2020. [Online]. Available: https://arxiv.org/abs/1812.11118.

[3] J. Brownlee, "Gentle Introduction to the Bias-Variance Trade-Off in Machine Learning", Machine Learning Mastery, 2020. [Online]. Available: https://machinelearningmastery.com/gentle-introduction-to-the-bias-variance-trade-off-in-machine-learning.

[4] G. James, D. Witten, T. Hastie, R.Tibshirani *An Introduction to Statistical Learning*. Springer. 2013.

[5] T. Hastie; R. Tibshirani, J. H. Friedman. *The Elements of Statistical Learning*. 2009.

[6] U. Gupta. "Detailed Guide to Understand and Implement ResNets - CV-Tricks.com", CV-Tricks.com, 2020. [Online]. Available: https://cv-tricks.com/keras/understand-implement-resnets.

[7] "chenpf1025/noisy_label_understanding_utilizing", GitHub, 2020. [Online]. Available: https://github.com/chenpf1025/noisy_label_understanding_utilizing.

[8] D. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization", arXiv.org, 2020. [Online]. Available: https://arxiv.org/abs/1412.6980.

[9] "CIFAR-10 ResNet - Keras Documentation", Keras.io, 2020. [Online]. Available: https://keras.io/examples/cifar10_resnet/.

[10] A comparison of residual blocks between ResNet v1 and ResNet v2. March 20, 2020. Retrieved From https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788629416/2/ch02lvl1sec13/resnet-v2

[11] S. Spigler, M. Geiger, S. d'Ascoli, L. Sagun, G. Biroli and M. Wyart, "A jamming transition from under- to over-parametrization affects generalization in deep learning", 2020. [Online]. Available: https://arxiv.org/abs/1810.09665.

[12] M. Geiger et al., "Scaling description of generalization with number of parameters in deep learning", 2020. [Online]. Available: https://arxiv.org/abs/1901.01608.

[13] S. Mei and A. Montanari, "The generalization error of random features regression: Precise asymptotics and double descent curve", arXiv.org, 2020. [Online]. Available: https://arxiv.org/abs/1908.05355.

ACADEMIC VITA

SHUBHANGAM DUTTA

Shubhangam.dutta@gmail.com

Objective

To obtain a full time opportunity in the field of Information Technology/ Computer Science or Cybersecurity.

Education

Pennsylvania State University – University Park, PA

Bachelor of Science in Computer Engineering (Intended Minor in Cybersecurity). Expected graduation – May 2020

- **Academic Honors/Awards** – Dean's list, Certificate of merit from honor's society of Penn State
- **Awards/Honors –** Scholar blazer holder (3 consecutive years with scholar badge), Presentation project winner, proficiency certificate holder.

Experience/ Projects

- **Software Engineering Intern at Hughes Network Systems, Germantown, Maryland.** **May 2019 – August 2019**
  - Participated in Design and development of Identity and Access Management systems.
  - Project on deploying the IAM software on Amazon Web Services Cloud using a VPC.
  - Developed solutions in Java, JavaScript and web-based technologies.
  - Involved in Development of Responsive User Interface for user management.
- **Software Development Intern at Four Colors Technology, LLC in North Carolina.** **May – August 2018**
  - Project on Machine Learning in Python using the data from Boeing Aircraft.
  - Created an application in Python for Data preprocessing and predicting the change in the upcoming years.
  - Worked with software developers under a professional and an agile environment.
- **Undergraduate Researcher (Computer and Data Science) at Penn State.** **January 2019 - Present**
  - Completed honors research work on the topic of Double Descent behavior in Machine Learning.
  - Researched on various factors that affect the performance of deep learning neural networks.
- **Data Structure and Algorithms Grader and Mathematics Tutor at Penn State learning.**
  - Teaching and Grading higher level mathematics and Computer Science respectively to undergraduates.

Clubs / Projects

- Designing an interactive robotics platform and a tablet friendly GUI of the robotics platform that can be used to illustrate cybersecurity concepts for cyber-physical systems.
- Developed an application in Python that converts chunks of data from Excel to machine interpretable models for better data analysis and management.
- Represented Penn State University at the Cyberthreat Case Competition held at Deloitte University.
- Completed numerous projects using concurrent programming, LINUX and system's programming.
- Researched on developing techniques to improve cache performance in the CPU.
- Designed a single-cycle CPU (written in Verilog) using the Xilinx design package for FPGAs.
- Club: Institute of Electrical and Electronics Engineers (IEEE).
- Penn State Cricket Club

Skills / Specialization

- C/C++
- Python
- Java
- Verilog
- Linux
- Machine Learning
- AWS Cloud
- JavaScript
- HTML
- CSS/ Bootstrap
- Computer architecture
- Agile SWD
- Cybersecurity
- IAM
- FPGA
- Angular
- REST API
- DS & Algorithms
- MIPS
- MATLAB