Marked assignment 1 [20 points]

Hand-in format Please use the .ipynb skeleton file uploaded to blackboard. You will hand in this .ipynb file via the Blackboard (see Assessments and Mark Schemes folder and then upload it into 'Coursework 1 Drop Box Spring 24'). For the written parts, write your answers using the markdown cells (which also allow you to write formula's). Make sure to save your notebook such that you show your results (i.e. I'll need to be able to check the values you report in your results).

Part 1: Quickfire questions [3 points]

Answer each question *in short.* Each question is worth 0.25 points.

1. Give the definitions of true risk and empirical risk and discuss their core difference.
2. Explain the benefits and downsides of a 'large' or ''rich' hypothesis class (note: with large / rich we mean a class which can fit many kinds of functions).
3. Suppose I split my dataset into 80% train data and leave 20% as validation data. I optimise many models (with varying model architecture) over the train data and use the validation data to select the optimal model architecture. I then choose the model that has the best validation accuracy and deploy this model. Is it fair to expect the performance of the model on unseen data to match what I obtained on the validation data?
4. Explain Occam's razor and discuss its relation to naturally occurring data such as images.
5. Discuss the generalisation error for a good model (i.e. is it large or small and why).
6. Explain the meaning of a high empirical Rademacher complexity for the hypothesis set $\mathcal{F}$.
7. Discuss the motivation for using Rademacher complexity in generalisation bounds (i.e. what are the downsides of the function class dependency in the bound in (73) in the lecture notes of Week 1)?
8. Explain the usefulness of including a regularisation term in the loss function when optimising the model parameters.
9. Explain the motivation behind momentum gradient descent (i.e. which downsides of regular gradient descent does it help resolve)?

10. Explain the optimisation algorithm Adam highlighting the gradient descent and momentum components.

11. Explain AdaGrad.

12. Why would one use a decaying learning rate? Explain by discussing how this will affect your trajectory through the loss landscape.

Part 2: Short-ish proofs [6 points]

Part I: Bounds on the risk [1 point]

In the lecture and in the lecture notes we discussed Hoeffding's inequality and its application to risk bounds.

1. Present in your own words and formula's how we can go from Hoeffding's inequality (Theorem 4.2 in the lecture notes of Week 1) to the result in Corollary 4.6.

2. Discuss the dependence on the dataset size.

3. Discuss the meaning and relevance of the term $|\mathcal{F}|$ in Theorem 4.8.

Part II: On semi-definiteness [1 point]

Consider $x \in \mathbb{R}^d$. Remember the definition of convexity:

$$f(y) \geq f(x) + \nabla f(x)^T (y - x). \tag{1}$$

We would like to prove that this implies that for all x, $\nabla^2 f(x) \succeq 0$, i.e. the Hessian is positive semi-definite.

In the lecture we presented the proof for the one-dimensional case $d = 1$. Here we will extend the analysis to the multi-dimensional case. Define a function $g(t) := f(x + tv)$ for an arbitrary direction $v \in \mathbb{R}^d$.

Show that g is convex and twice differentiable. Derive $g''(t)$ and use this to show that $\nabla^2 f(x) \succeq 0$ (remember that this means $v^T \nabla^2 f(x) v \geq 0$ for any $v \in \mathbb{R}^d$).

Part III: A quick recap of momentum [1 point]

In the lecture and lecture notes we discussed momentum. Here we will recap the main benefits of momentum for an objective function of the form $f(x) = \frac{1}{2}x^T Sx + bx$ with $x \in \mathbb{R}^d$ and denote with x^* the optimum.

1. Consider the eigenvalue decomposition $S = Qdiag(\lambda_1, ...\lambda_d)Q^T$ and discuss *in your own words* what the change of basis $w_k = Q^T(x_k - x^*)$ allows you to obtain (i.e. why do we do this change of basis in the analysis). [0.25pt]

2. Discuss the downsides of the optimal learning rate for gradient descent (as derived in (75) in the lecture notes of Week 2). [0.25pt]

3. In the momentum updates we derived the evolution through the matrix R as in (85) in the lecture notes of Week 2. Explain in your own words how we can use properties of R to analyse the convergence. [0.25pt]

4. Discuss in your own words why we consider eigenvalues of R for which $(\beta + 1 - \alpha\lambda_i)^2 - 4\beta < 0$. [0.25pt]

Part IV: Convergence proof [3 points]

We will explore Newton's method. Specifically our goal will be to show the following statement:

Suppose $f \in C^3$ (i.e., three times continuously differentiable) and $x^* \in \mathbb{R}^d$ is such that $\nabla f(x^*) = 0$ and $\nabla^2 f(x^*)$ is invertible. Then, there exists $\epsilon > 0$ such that iterations of Newton's method starting from any point $x_0 \in B(x^*, \epsilon)$ (the ball of radius ϵ around x^*) are well-defined and converge to x^*. Moreover, the convergence rate is quadratic.

1. Present the evolution of Newton's gradient descent method. [0.2pt]

2. Consider $f(x) = \frac{1}{2}x^TQx + b^Tx + c$ with $x \in \mathbb{R}^d$ and $Q \succ 0$. Show that Newton's method will converge in a single step. [0.4pt]

3. Explain the meaning of the above statement in your own words. [0.4pt]

4. For the remainder of the question you may use the following lemmas without proof (stating clearly where you have used them):

   Lemma 0.1. *Let A be an $n \times n$ matrix and let $\|\cdot\|$ denote a vector norm, as well as its associated induced matrix norm $\|A\| = \max_{y=1} \|Ay\|$. Then, if $x \in \mathbb{R}^n$ is a vector, $\|Ax\| \leq \|A\|\|x\|$.*

   Lemma 0.2. *Let $A : \mathbb{R}^n \to \mathbb{R}^{n\times n}$ be a matrix valued function that is continuous at a point $u \in \mathbb{R}^n$. If $A(u)^{-1}$ exists, then there exists a scalar $\epsilon > 0$ such that $A(v)^{-1}$ also exists for all $v \in B(u, \epsilon)$.*

Using Newton's iterative step show that

$$\|x_1 - x^*\| \leq \|(\nabla^2 f(x_0))^{-1}\| \|\nabla^2 f(x_0)(x_0 - x^*) - \nabla f(x_0)\|.$$

[0.4pt]

5. Show that there is an $\epsilon > 0$ such that for $x_0 \in B(x^*, \epsilon)$ there are constants $c_1, c_2 > 0$ satisfying
$$\|x_1 - x^*\| \leq c_1 c_2 \|x_0 - x^*\|^2.$$
[0.6pt]

6. Suppose $x_0 \in B(x^*, \epsilon)$ satisfies
$$\|x_0 - x^*\| \leq \frac{\alpha}{c_1 c_2}$$
for some $0 < \alpha < 1$. Show that this implies that $x_1 \in B(x^*, \epsilon)$ and that $\|x_1 - x^*\| \leq \frac{\alpha}{c_1 c_2}$. [0.4pt]

7. Derive an upper bound for $\|x_{k+1} - x^*\|$ in terms of $\|x_k - x^*\|$, for $x_k \in B(x^*, \epsilon)$ satisfying $\|x_k - x^*\| \leq \frac{\alpha}{c_1 c_2}$. [0.2pt]

8. Conclude that $x_k \to x^*$ as $k \to \infty$ and that the convergence is quadratic. [0.4pt]

Part 3: A deeper dive into neural network implementations [3 points]

Making a neural network 'work' typically requires lots of trial and error with different architectures and training hyperparameters. However, there do exist certain (theoretical or numerical) results on how particular hyperparameter configurations affect performance. In this assignment we will explore some of these dependencies numerically. Specifically, your task will be to implement a fully-connected neural network and analyse how the performance of the model depends on the choice of hyperparameters (network width and depth, batch size, learning rate).

Getting ready Load the following libraries:

```
import numpy as np
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import torch.optim as optim
```

We will be working with the MNIST and CIFAR10 datasets. An example of how to download the CIFAR10 data is as follows:

```
train_set = torchvision.datasets.CIFAR10(
  root="./",
  download=True,
  train=True,
  transform=transforms.Compose([transforms.ToTensor()]),
)

test_set = torchvision.datasets.CIFAR10(
  root="./",
  download=True,
  train=False,
  transform=transforms.Compose([transforms.ToTensor()]),
)
```

You can pre-process the data in whichever way you wish: you can normalise it, or scale the pixel values (by dividing by 255). Just clarify what you did (if anything). For MNIST just replace CIFAR10 by MNIST.
Remember for MNIST the classes are the digits from 0-9. For CIFAR10 these are the classes of the above dataset:

```
    classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

Use your student ID (CID number) to set the seed:

```
    # set seed for reproducibility
    SEED = 'CID number'
    np.random.seed(SEED)
    torch.manual_seed(SEED)
```

Part I: Implementations [1 point]

In the first part we will be implementing the neural network and train and testing functions.

Task 1: Implement a fully-connected neural network [0.2pt] Implement a class (`class Net(nn.Module)`) with two functions:

- The function that initialises the layers that we will use. It has four parameters: 'dim': the dimension of the input, 'nclass': the number of output classes for our prediction problem, 'width': the width in each layer (we will consider all layers have the same width), 'depth': the depth of the neural network:

  ```
  def __init__(self, dim, nclass, width, depth):
          super().__init__()
          ... FILL IN THE REST ...
  ```

  You should define five types of layers: i) a layer that flattens the input, ii) a linear layer that takes the input and connects it to a hidden layer, iii) a linear layer that takes a hidden layer and connects it to the next hidden layer, iv) a ReLU activation layer, v) a linear layer that connects the hidden layer to the output. Use the `self.` to define the layers.

- The forward function that passes the input through the hidden layers and returns the output. The function should look as follows:

  ```
  def forward(self, input):
  ```

  This function should work for different numbers of hidden layers. You can do this by writing a for loop.

Task 2: Implement the data loading function [0.2pt] Implement a function that loads the data given a certain batchsize of the following form:

```
def loading_data(batch_size, train_set, test_set):
    ... FILL IN...
return trainloader, testloader
```

Use the `torch.utils.data.DataLoader` function.

Task 3: Implement a function that does one training epoch [0.2 point] This function should implement a single training epoch. Remember one epoch is a pass over all the training data. It should thus pass over all the items in the 'trainloader', compute the model output, compute the loss associated to the criterion, and take an update step for the optimization algorithm (the 'optimizer'). The function should have the following form:

```
    def train_epoch(trainloader, net, optimizer, criterion):
```

and should return the loss (associated to the criterion we will pass it) over the train data (`loss = criterion(outputs, labels)`).

Task 4: Implement a function that does one test epoch [0.2 point] This function should take as input the test data and compute the loss (associated to the criterion) and the error (how many samples the model predicted incorrectly) over all the test datasets using a specific model configuration (the 'net'). The function should have the following form:

```
    def test_epoch(testloader, net, criterion):
```

and should return two things: i) the *average* test loss over all the datapoints, ii) the error over all the datapoints where the error is computed as how many samples are predicted incorrectly (for this you will thus need to take the predicted class from the model and compare it to the true label; if they are the same, the model is correct; if different, the model is incorrect).

Task 5: Write a piece of code that sets the hyperparameters and that allows to run the train and test epochs [0.2 point] You will need to define the following hyperparameters:

- batch size
- dim (the dimension of the flattened input; it is 3072 for CIFAR10 and $28 * 28$ for MNIST)
- nclass (the number of classes; it is 10 for MNIST and CIFAR10)
- width
- depth
- the learning rate
- the number of training epochs

Then you will need to define i) the criterion (we will use the CrossEntropyLoss), ii) the optimizer (we can use Adam `optim.Adam` or SGD `optim.SGD`, but if the question doesn't specify which one to use you can try out others if you want). Then you will need to run all the functions: i) load the datasets using the function you created, ii) create a for loop that runs over the number of epochs and computes for each epoch the train and test outputs and prints *in each epoch* the results as follows:

```
print(f'Epoch: {epoch:03} | Train Loss: {train_loss:.04} |
    Test Loss: {test_loss:.04} | Test Error: {test_err:.04}')
```

Comment: I use 'error' to refer to 1-'accuracy'. Accuracy is thus the proportion of samples predicted correctly and error is the incorrect proportion.

Part II: Numerical exploration [2]

In this second part we will be working with the neural network on MNIST. Our goal will be to understand how the performance (in terms of train and test error) depends on the choice of hyperparameters.

We will fix certain hyperparameters, but other hyperparameters are up to you to decide! *Report all the used hyperparameters in the text.* Include the tables as plain text in the notebook. Make sure that in the notebook you show that the train and test losses you present in the tables are indeed obtained from running your model (i.e. print out all your train and test losses). If you do not print out the train and test losses or if we cannot see whether the train and test losses you reported in the table indeed were obtained from your code, you won't get any points. You will get partial points as long as your implementations are correct and correspond to the numbers you present in the tables. Full points can be obtained if you also managed to achieve some desired result (which is some pattern/result that should arise based on research results in recent years). In general: for all the numbers in the tables, keep all the parameters except the parameter in the table fixed for all your results.

Comment: How many epochs to use? In principle it's up to you to decide what you consider enough epochs (i.e. when the performance is good enough). Just report the used number.

Task 6: Analyse the performance for wide vs. deep neural networks [1 point] Fix the number of nodes per layer to 256. Choose the other hyperparameters yourself (and keep them fixed for all the results you present). Fill in the following table (include it in the notebook):

Depth	Train loss	Test loss
1		
5		
10		

Include in the notebook a discussion of your result. Some potential things you may observe: deep neural networks could lead to more overfitting on the data (smaller train error but worse test performance); also they may be harder to train and for example require a smaller learning rate (if you train with SGD).

Task 9: Analyse the train and test errors as a function of width [1 point] Consider a neural network with *one* hidden layers. Choose the other hyperparameters and optimisation algorithm yourself. Fill in the below table and present a plot of the train and test losses as a function of width in your notebook.

Width	Train loss	Test loss
4		
8		
16		
32		
64		
128		
256		
512		
1024		

Discuss the results. What kind of width do you need to fit the data? Recent work [Neyshabur et al., 2018] has shown that the test error continues to decrease (or at least not increase) even if we make the number of parameters very large. If we were to use the standard generalisation bounds, we would conclude that at some point we overfit on the data and the test error would start to increase. However, with stochastic gradient descent there seems to be some implicit bias which does not allow this to happen.

Comments: The results from Figure 1 is something we'd expect to see.

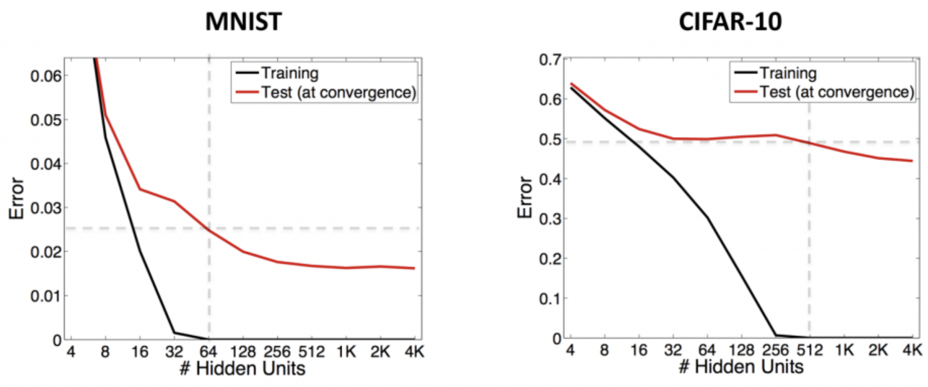

Figure: Training (empirical) and test (true) error for one-hidden-layer networks of increasing width, trained with SGD [20].

Figure 1: The results we'd expect to see; from Volkan Cehver's lectures `https://www.epfl.ch/labs/lions/wp-content/uploads/2022/01/lecture_08_2021.pdf`

Part 4: The link between Neural Networks and Gaussian Processes [8 points]

Part I: Proving the relationship between a Gaussian process and a neural network [4pt]

Setup We will follow here a more recent line of work that has made the connection between deep neural networks and Gaussian processes. Initially, the connection was made in [Neal, 1996] for neural networks with a single hidden layer. More recently this has been expanded upon by for example [Matthews et al., 2018, Lee et al., 2017]. Even more recently, the connection has been made that also the *training* process of a neural network under certain settings can be characterised through a kernel [Jacot et al., 2018].

One hidden layer Consider a fully-connected neural network. Suppose the input x is of dimensionality N_0 and we have N_1 hidden nodes in layer 1 and N_2 output nodes. The activation function is given by ϕ. The output for node i in the first layer is given by,

$$f_i^{(1)}(x) = \sum_{j=1}^{N_0} w_{ij}^{(1)} x_j + b_i^{(1)}, \tag{2}$$

where $w_{ij}^{(1)}$ is the weight in the first layer connecting node i in layer 1 with input j and $b_i^{(1)}$ is the bias we add in layer 1 to output node i. This is then passed through the non-linearity to obtain:

$$g_i^{(1)}(x) = \phi(f_i^{(1)}(x)). \tag{3}$$

The output layer is consequently given by,

$$f_i^{(2)}(x) = \sum_{j=1}^{N_1} w_{ij}^{(2)} g_j^{(1)}(x) + b_i^{(2)}. \tag{4}$$

Note that in the above notation we make explicit the dependence on x.

We will assume a distribution on the parameters of the network. Conditional on the inputs, this will induce a distribution on the other nodes. In particular, we will assume independent normal distributions on the weights and biases,

$$w_{ij}^{(l)} \sim \mathcal{N}(0, C_w^{(l)}), \text{ i.i.d.} \tag{5}$$

$$b_i^{(l)} \sim \mathcal{N}(0, \sigma_b^{(l)}), \text{ i.i.d.}, \tag{6}$$

where $l = \{1, 2\}$ and denotes the number of the layer, $C_w^{(l)}$ is the covariance for the weights in layer l and $\sigma_b^{(l)}$ is the covariance for the biases in layer l. Note that because the weight and bias parameters are taken to be i.i.d., the post-activations $g_j^{(1)}$, $g_{j'}^{(1)}$ are independent for $j \neq j'$.

Task 1: Proper weight scaling [0.5pt] We will be interested in the behavior of this network as the width N_1 becomes large (goes to infinity). In order to not have divergence in the variance of the output, we will scale the weight variances for $l \geq 2$ accordingly (in the one hidden layer case only $l = 2$ gets scaled). The appropriate scaling will be,

$$C_w^{(l)} = \frac{\sigma_w^{(l)}}{N_{l-1}},\ l \geq 2. \tag{7}$$

Can you explain in your own words why such a scaling would make sense?

Task 2: Derive the GP relation for a single hidden layer [1pt] We are now interested in deriving the distribution of the outputs $f_i^{(2)}(x)$ (for different inputs). Specifically, we want to show it converges to a Gaussian process. Use the multivariate central limit theorem to show that $f_i^{(2)}$ follows a Gaussian process distribution and derive the mean μ^1 and covariance matrix K^1.

Hint: arrange the terms into vectors, show that the items we are summing are i.i.d. and apply the CLT. The covariance matrix will be a function of the activation functions and you can leave it as an expected value of the previous layer's output.

Multiple hidden layers Consider an L-hidden layer fully-connected neural network where the hidden layers are of width N_l (for layer l) and the activation function for each layer is given by ϕ. We would now like to extend the arguments in the previous section to the setting of multiple hidden layers, i.e. deep neural networks. We will take the limits $N_1 \to \infty$, $N_2 \to \infty$ and so on in *succession.*

Task 0: Why in succession? [0.25pt] Explain why we apply the limit in succession.

Task 1: The derivation [2.25pt] Suppose thus that $f_j^{(l-1)}$ is a GP (over the inputs) identical and independent for every j (and hence $g_j^{(l-1)}(x)$ are independent and identically distributed). The output in the l-th layer is,

$$f_i^{(l)}(x) = \sum_{j=1}^{N_l} w_{ij}^{(l)} g_j^{(l-1)}(x) + b_i^{(l)}, \;\; g_j^{(l-1)}(x) = \phi(f_j^{(l-1)}(x)). \tag{8}$$

Show that $f_i^{(l)}$ is a Gaussian process with mean μ^l and covariance matrix K^l. Make it clear why it is valid to apply the CLT in this case. Derive a recursive expression for the covariance K^l; the final expression will again contain an expected vaue over the previous layer's output.

Part II: Analysing the performance of the Gaussian process and a neural network [4pt]

Consider the ReLU activation function. In this case it can be derived that the recurrence relation for the kernel is given by,

$$K^l(\mathbf{x}, \mathbf{x}') = \sigma_b^2 + \frac{\sigma_w^2}{2\pi}\sqrt{K^{l-1}(\mathbf{x},\mathbf{x})K^{l-1}(\mathbf{x}',\mathbf{x}')}\left(\sin\theta_{\mathbf{x},\mathbf{x}'}^{l-1} + (\pi - \theta_{\mathbf{x},\mathbf{x}'}^{l-1})\cos\theta_{\mathbf{x},\mathbf{x}'}^{l-1}\right), \tag{9}$$

$$\theta_{\mathbf{x},\mathbf{x}'}^{l} = \cos^{-1}\left(\frac{K^l(\mathbf{x},\mathbf{x}')}{\sqrt{K^l(\mathbf{x},\mathbf{x})K^l(\mathbf{x}',\mathbf{x}')}}\right), \tag{10}$$

where $\mathbf{x}$ and $\mathbf{x}'$ are two datapoints of dimension N_0. For K^0 we would have,

$$K^0(\mathbf{x},\mathbf{x}') = \sigma_b^2 + \sigma_w^2\left(\frac{\mathbf{x}\cdot\mathbf{x}'}{N_0}\right). \tag{11}$$

We will use this recurrence in the implementations.

Dataset We will use the CIFAR10 dataset. However, we will select just two classes of the dataset so that we are working with binary classification. You are free to select the classes you want. Also subsample 1000 train samples (i.e. we will not work with the full dataset).

Task 0: Formulate classification as a regression problem [.25pt] We will perform classification using a regression-type output. Change the class labels to -0.5 and +0.5. How can you use the Gaussian process to perform classification and how can you formulate your decision rule to perform the classification?

Task 1: Implement the kernel of the Gaussian process [1pt] Use relation (10) to implement a function that defines the GP kernel. This function should take as input the number of layers L, σ_w^2, σ_b^2 and two datasets X^1, X^2 of size $M_1 \times N_0$ and $M_2 \times N_0$ where N_0 is, as before, the dimension of the input and M_1 and M_2 are the number of datapoints in the first and second dataset, respectively. The output from this function should be a matrix K^L where each element K_{ij}^L is the kernel function applied to element i from X^1 and element j from X^2.

Task 2: Implement a method to compute the mean and covariance of the Gaussian process [0.75pt] Implement a method to compute the mean and covariance of the Gaussian process posterior using:

$$f^*|X, y, X^* \sim \mathcal{N}(K^L(X^*, X)[K^L(X, X) + \sigma^2 I)]^{-1}\mathbf{y}, \tag{12}$$

$$K^L(X^*, X^*) - K^L(X^*, X)[K^L(X, X) + \sigma^2 I]^{-1} K^L(X, X^*)), \tag{13}$$

respectively, where $(X, \mathbf{y})$ is the train dataset and X^* is the dataset for which we want to obtain the prediction, σ is the noise level, and K^L is the kernel matrix for layer L. This function should take as input the train dataset $(X, \mathbf{y})$, the test points X^*, the noise σ, the weight and bias variance parameters σ_b^2 and σ_w^2 and the number of layers L. The output from the function should be the posterior mean and covariance. Hint: if you get NaN values, you may need to clip the outputs in the kernel function before passing it into the acos; you can do this using *torch.clamp*.

Task 3: Analyse the performance of the Neural Network Gaussian process [1.25pt] Let us measure the performance in terms of the accuracy (the proportion of samples predicted correctly) on a test set. Compute the accuracy over 1000 test samples using the posterior mean. You will also need to set the σ_b^2 and σ_w^2 in an appropriate manner. One option is to define a grid of possible values and choose the values which achieve highest test performance. Present results on the performance of the NNGP changes as you increase the number of layers L from 1 to 10 (in a table of in a figure). Discuss your results: what is the impact of depth on the performance? What σ_w^2 and σ_b^2 are optimal? Any other observations on the performance? Note: the σ_b^2 and σ_w^2 can thus differ for each L.

Task 4: Analyse the uncertainty [0.5pt] For each test point you can also compute the posterior covariance. Present the 2 test samples for which the uncertainty (i.e. the covariance) is highest and the 2 test samples for which it is lowest.

Task 5: Computational cost analysis [0.25pt] Can you identify where the computational cost of the NNGP model lies? What would happen if you increased your dataset size? Discuss.

References

[Jacot et al., 2018] Jacot, A., Gabriel, F., and Hongler, C. (2018). Neural tangent kernel: Convergence and generalization in neural networks. *Advances in neural information processing systems*, 31.

[Lee et al., 2017] Lee, J., Bahri, Y., Novak, R., Schoenholz, S. S., Pennington, J., and Sohl-Dickstein, J. (2017). Deep neural networks as gaussian processes. *arXiv preprint arXiv:1711.00165*.

[Matthews et al., 2018] Matthews, A. G. d. G., Rowland, M., Hron, J., Turner, R. E., and Ghahramani, Z. (2018). Gaussian process behaviour in wide deep neural networks. *arXiv preprint arXiv:1804.11271*.

[Neal, 1996] Neal, R. M. (1996). Priors for infinite networks. In *Bayesian Learning for Neural Networks*, pages 29–53. Springer.

[Neyshabur et al., 2018] Neyshabur, B., Li, Z., Bhojanapalli, S., LeCun, Y., and Srebro, N. (2018). Towards understanding the role of over-parametrization in generalization of neural networks. *arXiv preprint arXiv:1805.12076*.