A Study of Adaptive Random Features Models in Machine Learning based on Metropolis Sampling

BING BAI
Acknowledgements

Prof. Anders Szepessy and Ass. Prof. Mattias Sandberg have supervised this thesis. Special thanks to the supervisor for their patient supervision and instruction. Dr Aku Kammonen has provided many instructions and suggestions, which help a lot to accomplish this thesis. Juan Sebastian Diaz Boada has provided advice and helps about the thesis writing. Many thanks to their kindly help.
Abstract

Artificial neural network (ANN) is a machine learning approach where parameters, i.e., frequency parameters and amplitude parameters, are learnt during the training process. Random features model is a special case of ANN that the structure of random features model is as same as ANN’s but the parameters’ learning processes are different. For random features model, the amplitude parameters are learnt during the training process but the frequency parameters are sampled from some distributions. If the frequency distribution of the random features model is well-chosen, both models can approximate data well. Adaptive random Fourier features with Metropolis sampling is an enhanced random Fourier features model which can select appropriate frequency distribution adaptively. This thesis studies Rectified Linear Unit and sigmoid features and combines them with the adaptive idea to generate another two adaptive random features models. The results show that using the particular set of hyper-parameters, adaptive random Rectified Linear Unit features model can also approximate the data relatively well, though the adaptive random Fourier features model performs slightly better.
Sammanfattning

# Contents

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>iii</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Statement of the problem</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Thesis objective</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Outline of Thesis</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Background</strong></td>
<td>8</td>
</tr>
<tr>
<td>2.1 Machine Learning</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Artificial Neural Network</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Training Artificial Neural Networks</td>
<td>11</td>
</tr>
<tr>
<td>2.4 From Stochastic Gradient Descent to Adaptive Moment Estimation</td>
<td>13</td>
</tr>
<tr>
<td>2.5 Metropolis-Hasting Algorithm</td>
<td>16</td>
</tr>
<tr>
<td><strong>3 Design of Experiments</strong></td>
<td>19</td>
</tr>
<tr>
<td><strong>4 Implementations and Results</strong></td>
<td>24</td>
</tr>
<tr>
<td>4.1 Compare one hidden-layer neural network and Adaptive random Fourier features on classification task</td>
<td>24</td>
</tr>
<tr>
<td>4.2 Compare one hidden-layer neural network and Adaptive random Fourier features on regression task</td>
<td>26</td>
</tr>
<tr>
<td>4.3 Adaptive random features model with different random features</td>
<td>31</td>
</tr>
<tr>
<td><strong>5 Conclusions and Discussion</strong></td>
<td>32</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>34</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Statement of the problem

A central problem in machine learning is to find a parameterized function that approximates the data well. Tuning the function parameters, minimizing the empirical risk, producing a function that can generalize well for population risk and, furthermore, predicting unseen data by using the parameterized function are the main targets of machine learning. Many models have been proposed and investigated in the past decades to accomplish these targets. Among those models, artificial neural network (ANN) is an excellent one that gets practical success and over-performs than many other models.

However, for an one-hidden layer neural network model in which the parameters in both the input and output layers are updated, the computation amount will be relatively large. Moreover, in practice, it is sophisticated to tune the learning rate of gradient descent dynamics to converge to a global minimum or a small enough local minimum. When the width $K$ is large, usually, it will trap into some relatively large local minimum and miss the global minimum. These problems make ANN is a delicately tuned and fragile model.

To mitigate these two problems, one can use random features models, which inherits the structure of ANN and employs another training process to make the model becomes more easy-to-use and robust.
1.2 Thesis objective

The random features model is first proposed in [2] as a kernel method with the kernel defined by the initialization. From another perspective, the random features model can be seen as an one-hidden layer neural network with the frequency parameters that are fixed or sampled from some probability distribution $p(\omega)$.

![One-hidden layer neural network structure](image)

Figure 1.1: one-hidden layer neural network structure

The structure of an one-hidden layer neural network is demonstrated in figure 1.1. The parameter $d$ is the dimension of the input data and the parameter $m$ represents the number of nodes in the hidden-layer. In this figure, the dimension of the output data is 1, which is a special instance. Actually, the output data can be multi-dimensional.

The training data set is given by $\{(x_n, y_n)\}_{n=1}^{N}$ with the data independent identically distributed (i.i.d) samples from an unknown distribution $\rho$, which is assumed fixed but only known through the samples. We assume that there exists a function $f : \mathbb{R}^d \rightarrow \mathbb{R}$, such that $y_n = f(x_n) + \epsilon_n$, where the noise component is represented by i.i.d. random variables $\epsilon_n$ with $\mathbb{E}[\epsilon_n] = 0$ and $\mathbb{E}[\epsilon_n^2] = \sigma^2$.

We assume that the target function $f(x)$ can be approximated by an one-hidden layer neural network which defines an approximation $\hat{f} : \mathbb{R}^d \times \mathbb{R}^{m \times d} \times \mathbb{C}^m \rightarrow \mathbb{R}$.
\[ \hat{f}(x; \omega, \hat{\beta}) = \hat{\beta}^T \sigma(z) = \beta^T \sigma(\omega x) = \sum_{i=1}^{m} \hat{\beta}_i \sigma(\omega_i x) \quad (1.1) \]

where we use the notation for the parameters of the network \( \hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_m \in \mathbb{C}, \hat{\beta} = (\hat{\beta}_1, \hat{\beta}_2, \ldots, \hat{\beta}_m) \in \mathbb{C}^m, \omega_1, \omega_2, \ldots, \omega_m \in \mathbb{R}^d, \omega = (\omega_1, \omega_2, \ldots, \omega_m)^T \in \mathbb{R}^{m \times d}. \)

\( \hat{\beta} \) and \( \omega \) are amplitude parameters and frequency parameters, respectively.

The ultimate goal of the neural network is to minimize, over the set of parameters \((\hat{\beta}, \omega) \in \mathbb{C}^m \times \mathbb{R}^{m \times d}, \) the population risk defined by

\[ \mathcal{L}(\hat{\beta}, \omega) = \frac{1}{2} \mathbb{E}_{x,y}[(\hat{f}(x; \omega, \hat{\beta}) - y)^2]. \quad (1.2) \]

However, in practice, since the distribution \( \rho \) is unknown, we can only work with the following empirical risk defined by:

\[ \hat{\mathcal{L}}(\hat{\beta}, \omega) = \frac{1}{N} \sum_{n=1}^{N} \ell(\hat{f}(x_n; \omega, \hat{\beta}), y_n). \quad (1.3) \]

We focus on the reconstruction with the regularized least squares type risk function

\[ \ell(\hat{f}(x_n; \omega, \hat{\beta}), y_n) := |\hat{f}(x_n; \omega, \hat{\beta}) - y_n|^2 + \lambda \sum_{i=1}^{m} |\hat{\beta}_i|^2. \quad (1.4) \]

The least-square functional is augmented by the regularization term with a Tikhonov regularization parameter \( \lambda > 0. \) For the sake of brevity we often omit the arguments \( \hat{\beta}, \omega \) and use the notation \( \hat{f}(x) \) for \( \hat{f}(x; \omega, \hat{\beta}). \) We also use \(|\hat{\beta}|^2 := \sum_{i=1}^{m} |\hat{\beta}_i|^2 \) for the Euclidean norm on \( \mathbb{C}^m. \)

Approximately reconstructing \( f \) from the data based on the least squares method is a common task in statistics and machine learning, cf. [3], which in a basic setting takes the form of the minimization problem...
\[ \min_{f \in \mathcal{N}_m} \left\{ \mathbb{E}_\rho \left[ |y_n - \hat{f}(x_n)|^2 \right] + \lambda \sum_{i=1}^m |\hat{\beta}_i|^2 \right\} \tag{1.5} \]

where

\[ \mathcal{N}_m := \left\{ \hat{f}(x) = \sum_{i=1}^m \hat{\beta}_i \sigma(\omega_i, x) \right\} \tag{1.6} \]

represents an artificial neural network with one hidden layer.

We assume that the frequencies \( \omega \) are random and we denote the conditional expectation with respect to the distribution of \( \omega \) conditioned on the data \( (x, y) \) as \( \mathbb{E}_{\omega}[g(\omega, x, y)] := \mathbb{E}[g(\omega, x, y)|x, y] \). Since a minimum is always smaller than or equal to its mean, there holds

\[ \min_{(\hat{\beta}, \omega) \in \mathbb{C}^m \times \mathbb{R}^{m \times d}} \left\{ \mathbb{E}_\rho \left[ |y_n - \hat{f}(x_n)|^2 \right] + \lambda |\hat{\beta}|^2 \right\} \leq \mathbb{E}_{\omega} \left[ \min_{\hat{\beta} \in \mathbb{C}^m} \left\{ \mathbb{E}_\rho \left[ |y_n - f(x_n)|^2 \right] + \lambda |\hat{\beta}|^2 \right\} \right]. \tag{1.7} \]

The minimization in the right hand side of the inequality (1.7) is also known as the random features problem, cf. [4] [5] [1].

Especially, when the we consider a particular activation function, which is also known as Fourier features, \( \sigma(\omega, x) = e^{i\omega \cdot x} \), for \( \omega \in \mathbb{R}^d, x \in \mathbb{R}^d \), the minimization in the right hand side of the inequality (1.7) is also known as the random Fourier features problem. And a discrete version of problem (1.5) can be formulated, for training data \( (x_n, y_n)_{n=1}^N \), as the standard least squares problem

\[ \min_{\hat{\beta} \in \mathbb{C}^m} \left\{ N^{-1} |S\hat{\beta} - y|^2 + \lambda |\hat{\beta}|^2 \right\} \tag{1.8} \]

where \( S \in \mathbb{C}^{N \times m} \) is the matrix with elements \( S_{n,j} = e^{i\omega_j \cdot x_n}, n = 1, \ldots, N, j = 1, \ldots, m \) and \( y = (y_1, \ldots, y_N) \in \mathbb{R}^N \). Problem (1.8) has the corresponding linear normal equations:

\[ (S^T S + \lambda N I)\hat{\beta} = S^T y. \tag{1.9} \]
The differences and relations between the one-hidden layer neural network and random features model are apparent from the above statement. The structures of the one-hidden layer neural network and random features model are the same. However, the training processes of these two approaches are different. In the one-hidden layer neural network, amplitude parameters and frequency parameters are learnt during the training step. In the random features model, only amplitude parameters are learnt directly from the train data set, whereas the frequency parameters are usually sampled from some distributions. Thus, in some sense, the random features model is a particular case of one-hidden layer neural network model and the random Fourier features model is a particular case of one-hidden layer neural network model with cosine function as the activation function.

The motivation of splitting one minimization in the left hand side of the inequality (1.7) into two minimizations in the right hand side of the inequality (1.7) is that the inner minimization of the right hand side is a convex problem. So that there exists several robust solution methods to solve this inner minimization.

E et al. [6] have proved that the functions obtained from an over-parametrized one-hidden layer neural network are uniformly close to those found in an associated random features model. An over-parameterized model is a model that has more parameters than can be estimated from the data. For a one-hidden layer neural network, it is over-parametrized if the width $K$ is greater than the number $N$ of training points, i.e., $K > N$ [1]. In other words, in the over-parameterized regime, the relations and differences between one-hidden layer neural network and random features model are clear. Thus, in this thesis, we study and explore the differences between these two methods when $K < N$.

This thesis implements two types of optimization methods to train the model with one-hidden layer neural network structure. When we use the first type optimization methods, such as stochastic gradient descent and Adam, which are classical optimizers and introduced in chapter 2, the model is called one-hidden neural network as customary. When we use the second type optimization methods, which are random features methods, the model is called random features model in this thesis. We compare the performance of these two types of optimization methods on MNIST dataset. And the result shows that if the probability distribution $p(\omega)$ of the random Fourier features model is well-chosen, both models can get a similar and good prediction accuracy.

However, how to choose an appropriate probability distribution $p(\omega)$ is the
crucial problem. From Rahimi’s work [2], Barron’s work [4] and Jones’ work [5], to minimize the right-hand side of the error estimation above, \( p(\omega) = p^*_\omega := |\hat{f}(\omega)|/\|\hat{f}\|_{L^1(\mathbb{R}^d)}, \) where \( \hat{f} \) is the Fourier transform of \( f \). Whereas, the target function \( f \) is unknown, and the optimal distribution \( p^*_\omega \) cannot be calculated explicitly. Thus, the aim is to formulate an adaptive method that approximately generates independent samples from the density \( p^*_\omega \).

To accomplish this aim, Kammonen et al. [1] have proposed an improved random features model, i.e., adaptive random Fourier features with Metropolis sampling, a systematic method to approximately sample from the optimal distribution \( p^*_\omega \).

Metropolis algorithm is a Markov chain Monte Carlo (MCMC) method to generate a collection of random samples from the desired distribution from which direct sampling is difficult. Metropolis algorithm is introduced more detailed in Chapter 2.

In adaptive random features model, the application of Metropolis sampling can be summarized in two parts. Firstly, the distribution \( \hat{\beta}(\omega) \), which is asymptotically equidistributed by the optimal distribution \( p^*_\omega \) is proved. This property is the motivation for using Metropolis sampling. The second part is using the Metropolis algorithm to obtain a sequence of random samples from the distribution \( \hat{\beta}(\omega) \). For more detailed information, it can be seen in [7].

It is necessary to state that though random features model is a more general model (with many other activation functions can be used) compared with random Fourier features model, we still study this "particular case" because the error estimation of this "particular case" is explicit.

Following Barron’s work [4] [5], Kammonen et al. [1] has derived the known error estimation for the random Fourier features model:

\[
E_\omega \left[ \min_{\beta \in \mathcal{C}^K} \left\{ E_p[|y_n - \beta(x_n)|^2] + \lambda |\hat{\beta}|^2 \right\} \right] \leq K^{-1} \left( \frac{1}{(2\pi)^d} \int_{\mathbb{R}^d} \frac{|\hat{f}(\omega)|^2}{p(\omega)} d\omega - f^2(x) \right).
\]

(1.10)

Though Adaptive random Fourier features with Metropolis sampling provides a systematic method to approximately sample from the optimal distribution \( p^*_\omega \), it only uses Fourier features which might be a limitation of this systematic method. Thus, in this thesis, another two random features(activation functions), i.e., Sigmoid function and ReLU function, are combined with the
adaptive idea above and generates another two adaptive random features models.

From the results of the numerical experiments, for other random features (activation functions), such as Sigmoid function and ReLU function, we can see that though we cannot prove them rigorously at now, it also works well when we use Metropolis sampling idea to approximately sample from the optimized distribution $p_*(\omega)$.

### 1.3 Outline of Thesis

Chapter 2 includes a contextualization of machine learning and an explanation of the mathematical theory behind it. The chapter covers a brief explanation of the methods which are implemented in the thesis, such as stochastic gradient descent (SGD), adaptive moment estimation (Adam) and Metropolis sampling methods. Chapter 3 introduces the design of experiments. The simulation methods and datasets that have been used to in these experiments are introduced. And the motivations and aims of these designed experiments are also covered in this chapter. Chapter 4 gives details about the implementations of the designed experiments, including the datasets and the methods used. At the same time, this chapter includes all the results of these experiments. The comparison and explanation of the results are also given in this chapter. Chapter 5 analyses the results of the experiments through all the experiments of the projects and presents the thesis’s conclusions, including the contributions, the findings, the limitations and the future work.
Chapter 2

Background

2.1 Machine Learning

Machine Learning categories

Machine learning is a subject that using algorithms to build a model based on sample data, in order to make predictions or decisions without being explicitly programmed to do so\footnote{This definition of the term “machine learning” was popularized in 1959 by Arthur Samuel, an American pioneer in the field of artificial intelligence.}. More specifically, it is the family of computational methods with a data-driven approach, which extracts patterns from data without knowing a precise mathematical model in advance. Machine Learning methods can mainly be classified into two big groups, depending on the learning goal and the type of data: supervised and unsupervised learning. Supervised learning uses labelled data to perform tasks like regression or classification, while unsupervised learning uses unlabelled data to perform clustering or dimensionality reduction. The machine learning methods studied and researched in this thesis all exclusively belong to supervised learning methods.

Supervised Learning

In practice, supervised learning methods are characterized by exploiting a predictive ability. This means that generally, its primary function is to predict the label of an unseen instance. The labels of the data can be of different types, such as numerical or categorical. Generally speaking, the type of label will
define the nature of the supervised problem: if the output is numerical, the problem will be defined as a regression problem, otherwise it will be a classification problem. In this thesis, in order to obtain a more comprehensive and general result, we design both regression problems and classification problems to test and compare the performance of different machine learning methods.

2.2 Artificial Neural Network

The perceptron: An artificial neural unit

Artificial neural networks (ANN) were inspired by biological neural networks. Biological neural networks are composed of fundamental learning units called neurons. A neuron receives stimuli by other neurons through connections called synapses. Multiple synapses can stimulate a neuron simultaneously, having an overall effect equal to the sum of stimuli Figure 2.1(a). If a neuron receives enough inputs to reach a certain threshold, it will trigger a strong response in the neuron called an action potential. The action potential is a binary response, for which it is often described as an "all or none" response [9].

![Diagram of biological neuron](image1)

(a) Stimulation of a biological neuron.

![Diagram of perceptron](image2)

(b) Perceptron.

Figure 2.1: The resemblance between biological neurons and perceptrons. a) Red neurons give stimuli to the blue neuron. When the sum of the stimuli reaches the threshold, the blue neuron generates an output. b) The perceptron’s inputs are combined linearly with some weights and introduced in a non-linear activation function.

A perceptron is the artificial analogue of a biological neuron. It receives a
set of $n$ inputs $x_i$ modulated by a corresponding set of weights $\omega_i$ that mimic the stimuli of a biological neuron (see Figure 2.1(b)). Each input represents a feature of the data, meaning that a set of simultaneous inputs would represent a single data point being evaluated in the neuron. The inputs then are transformed by an inner vector product, often called logit $z$ equation (2.1). An additional term $b$ called bias is usually added to the vector product. Then, the "all or none" response $h$ is modelled evaluating the logit by an activation function $\sigma$. One of the most simplified activation functions is the step function, which outputs 1 if the logit reaches the desired threshold $t$ or 0 otherwise:

$$z = \omega_1 x_1 + \omega_2 x_2 + \cdots + \omega_p x_p + b = \sum_{i}^{p} \omega_i x_i + b = \omega \cdot x + b$$  (2.1)

$$h = \sigma(z) = \begin{cases} 0, & \text{for } z \leq t \\ 1, & \text{for } z > t. \end{cases}$$  (2.2)

**Artificial neural network: A perceptron network**

In artificial neural networks, a single neuron has no significant influence if it is not connected to others. Therefore, multiple perceptrons can be connected to tackle more complicated tasks. A set of connected artificial neurons working together to perform a specific task is known as an artificial neural network.

One way of integrating multiple neurons is by connecting them in parallel. With this configuration, the different neurons receive a unique linear combination of the inputs since all the weight combinations are different. After the respective activation functions evaluate their logits, each neuron returns its binary response. A subsequent neuron, called output neuron, can then linearly combine these results to generate the output of the network figure 2.2.

The network as mentioned above would be composed of three parts: the input neurons, each of them representing a feature of the input data; the set of neurons evaluating the inputs known as the hidden layer; and the output neuron, which consolidates the responses of the neurons in the hidden layer. Such a network is called an one hidden-layer neural network.
2.3 Training Artificial Neural Networks

Training dataset: Learning from examples

Training process is an essential step of every supervised learning method in which it can learn from labelled examples. Though each machine learning method may have its particular properties, all of them follow the same general procedure in the training process. The process starts by selecting a representative subset of the available data known as the training dataset while leaving the remaining examples as the test dataset. The training dataset is used to tune the parameters, while the test dataset is left to evaluate the method’s generalisation ability after training. Sometimes, an additional subset of the dataset known as the validation dataset is built to monitor the performance during training and change the value of non-training parameters, known as hyper-parameter. The basic hyper-parameter of an one hidden-layer neural network is the number of neurons per layer.

Cost function: Quantification of the error

The next step of the training process is to quantify the error between the predictions of the model and the true value. The true value of an instance $x_i$ is its label $y_i$. In contrast, the prediction $\hat{y}_i$ of instance $x_i$ is the output given by
the neural network based on evaluating \( x_i \). The required error function should measure the discrepancy between \( y_i \) and \( \hat{y}_i \) for all training data. This function has to take relatively large values when \( y_i \) and \( \hat{y}_i \) are pretty different, and relatively small values when they are similar. Such a function \( J \) is known as the cost function.

Different problems need to choose specific cost functions that fit the problem best. For regression problems and classification problems, the most appropriate cost functions are usually quite different. The mean squared error (MSE) is an common choice for a regression problem:

\[
\hat{L} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2, \tag{2.3}
\]

where \( N \) is the number of the training data.

For binary classification problems, cross-entropy is often used:

\[
\hat{L} = -\sum_{i=1}^{N} \left( y_i \log_2(\hat{y}_i) + (1 - y_i) \log_2(1 - \hat{y}_i) \right). \tag{2.4}
\]

For multi-class classification, the cross-entropy takes the form:

\[
\hat{L} = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{i,k} \log_2(\hat{y}_{i,k}), \tag{2.5}
\]

where \( K \) represents the number of different classes of output.

The cost function depends on the values of \( \hat{y}_i \), which, in another word, depends on each trainable weight \( \omega \). As the aim is to find the optimal values for each weight \( \hat{w} \), the cost function has to be minimized, and then take the values \( \hat{w} \) corresponding to the minimum. However, due to the high dimension of the parameter space and the fact that the neural network implements non-linear activation functions in the hidden layers, the cost function becomes non-convex [10], leading to the minimum estimation to be made by iterative algorithms.
2.4 From Stochastic Gradient Descent to Adaptive Moment Estimation

Stochastic Gradient Descent

An appropriate iterative algorithm to minimize the cost function is stochastic gradient descent (SGD). SGD is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or sub-differentiable). It can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). The basic idea behind stochastic approximation can be traced back to the Robbins–Monro algorithm of the 1950s [11].

In this method, on one hand, the cost function is estimated with fewer data points in each iteration, which reduces the computational burden, achieving faster iterations, especially in high-dimensional optimization problems. On the other hand, the cost function is estimated over more iterations. This means that the gradient calculation may not be exact in each iteration but it has several iterations to correct itself, reducing the risk of getting stuck in a flat region [12] [13].

Generally, the stochastic gradient descent method can be divided into three steps. The first step is calculating the stochastic gradient of the cost function regarding the current parameter \( \omega^i_t \):

\[
g_t = \nabla \hat{L}(\omega^i_t). \tag{2.6}\]

Here \( \hat{L} \) is the cost function to be minimized. The subscript \( t \) of \( \omega^i_t \) represents the iteration times. The superscript \( i \) of \( \omega^i_t \) represents only one or a small batch of train data is used to calculate the gradient (If \( i \) is a stochastic number, then one train data is used. If \( i \) represents several stochastic numbers, then a small batch of train data is used). And \( \omega^i_t \) in this expression represents all the parameters to be trained in the model.

The second step is calculating the current gradient descent:

\[
\eta_t = \alpha \cdot g_t \tag{2.7}\]
where $\alpha$ is the learning rate of SGD method.

The last step is updating the parameter $\omega$ according to the current gradient descent:

$$\omega_{t+1} = \omega_t - \eta_t.$$  \hspace{1cm} (2.8)

To summarize these three steps in one equation:

$$\omega_{t+1} = \omega_t - \alpha \cdot \nabla \hat{L}(\omega_t).$$ \hspace{1cm} (2.9)

Due to the updating of parameters in each iteration, the current solution moves to a new point in the cost function, so a new gradient must be calculated. The number of iterations depends on the number of batches, which are subsets of training data points used to estimate the cost function and update the network’s weights. The number of batches equals the total number of the data divided by the number of data of each batch which is \textit{batch size}.

Usually the iterations are stopped when the cost function arrive at the minimal value or the cost function doesn’t decrease for $k$ iterations. This hyper-parameter $k$ is called as \textit{patience}. However, in neural network training, another frequently-used stopping schedule is setting the number of times the whole dataset will be used in gradient descent as a stopping criterion. Each of these complete passes is called an \textit{epoch}.

Another important hyper-parameter of SGD method is \textit{learning rate} $\alpha$ which determines the method’s convergence speed. A too small learning rate will make tiny steps towards the solution, which will cause the algorithm taking too many iterations to reach the minimum. However, a too big learning rate may cause the divergence of the algorithm and miss the desired minimum. In practice, we usually need a relatively large learning rate at the beginning of the iterations to approach a minimal region fast. After that a smaller learning rate is preferred to explore this minimal region meticulously. Thus, several learning rate schedules have been proposed to tune the learning rate delicately. Among of these learning rate schedules, \textit{exponential decayed schedule} is widely used and performs quite well. There are three parameters in exponential decayed schedule, which are \textit{initial learning rate} $\alpha_0$, \textit{decay step} $k$ and \textit{decay rate} $\theta$, respectively. When the current number of iteration is $K$, the current learning rate is:
\[ \alpha = \alpha_0 \cdot \theta^K. \]  

(2.10)

SGD is a popular algorithm for training a wide range of models in machine learning, including (linear) support vector machines, logistic regression (see, e.g., Vowpal Wabbit) and graphical models.\[14\] When combined with the backpropagation algorithm, it is the de facto standard algorithm for training artificial neural networks.\[15\] Though SGD is a quite old optimizer, it is still a classical optimizer with strong vitality. It is necessary to state that the adjustment of SGD’s learning rate is tricky and needs abundant experience. But if an appropriate learning rate schedule is chosen, this classical optimizer may obtain brilliant results.

**Stochastic Gradient Descent with Momentum**

Based on the SGD method, many improvements have been proposed and used which generate several extensions and variants of SGD. One of the most significant improvements is the introduction of *Momentum*, which is also known as first-order momentum. Moreover, stochastic gradient descent with momentum (SGDM) is proposed. To restrain the fluctuation during the iterations of SGD algorithm, we could use SGDM algorithm to decrease the influence of current gradient. The first and last steps of SGDM algorithm are as same as SGD algorithm. However, the step of calculating the current gradient descent is improved:

\[ m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1)g_t \]  

(2.11)

where \( m \) is the first-order momentum, which represent the accumulation of former gradients and \( m_0 = g_0 \). \( \beta_1 \), the hyper-parameter in SGDM, is usually set as 0.9 which is an empirical value. And the current gradient descent is:

\[ \eta_t = \alpha \cdot m_t. \]  

(2.12)

**Adaptive Gradient Algorithm**

Another significant improvement of SGD is the introduction of second-order momentum. Based on the introduction of second-order momentum, adaptive gradient algorithm (AdaGrad) is proposed. The only different step between SGD and AdaGrad is still the step of calculating the current gradient descent:
where $v_t$ is the second-order momentum, which is the summation of former gradients’ square.

Thus, in AdaGrad algorithm, the learning rate $\alpha$ will be tuned adaptively, which is the reason of the name. Compared with SGD, convenience is a significant advantage of AdaGrad.

**Adaptive Moment Estimation**

Adaptive moment estimation (Adam) is an extension of SGD. In this optimization algorithm, both first-order momentum and second-order momentum are used.

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) g_t$$  \hspace{1cm} (2.15)

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) g_t^2$$  \hspace{1cm} (2.16)

where $\beta_2$, the hyper-parameter in Adam, is usually set as 0.999 which is also an empirical value. And the current gradient descent is:

$$\eta_t = \frac{\alpha}{\sqrt{v_t}} \cdot m_t.$$  \hspace{1cm} (2.17)

One can see from above, not only the fluctuation during the iterations of the algorithm is restrained, but also the learning rate is tuned adaptively. Combining these two advantages, Adam is becoming more and more popular nowadays.

**2.5 Metropolis-Hasting Algorithm**

The Metropolis–Hastings algorithm is a Markov chain Monte Carlo (MCMC) method for obtaining a sequence of random samples from a probability dis-
distribution $P(x)$ from which direct sampling is difficult. A precondition of this algorithm is a function $f(x)$ is known, which is proportional to $P(x)$.

To accomplish this, the algorithm uses a Markov process, which asymptotically reaches a unique stationary distribution $\pi(x)$ such that $\pi(x) = P(x)$.

A Markov process is uniquely defined by its transition probabilities $P(x'|x)$, the probability of transitioning from any given state $x$ to any other given state $x'$. It has a unique stationary distribution $\pi(x)$ when the following two conditions are met [16]:

**Existence of stationary distribution**: there must exist a stationary distribution $\pi(x)$. A sufficient but not necessary condition is detailed balance, which requires that each transition $x \to x'$ is reversible: for every pair of states $x, x'$, the probability of being in state $x$ and transitioning to state $x'$ must be equal to the probability of being in state $x'$ and transitioning to state $x$, i.e., $\pi(x)P(x'|x) = \pi(x')P(x|x')$.

**Uniqueness of stationary distribution**: the stationary distribution $\pi(x)$ must be unique. This is guaranteed by ergodicity of the Markov process, which requires that every state must be aperiodic and positive recurrent.

The Metropolis–Hastings algorithm involves designing a Markov process (by constructing transition probabilities) that fulfills the two above conditions, such that its stationary distribution $\pi(x)$ is chosen to be $P(x)$. The detailed derivation of the algorithm can be seen in [16].

**Steps of Metropolis-Hasting Algorithm**

1. Initialize
   1) Pick an initial state $x_0$.
   2) Set $t = 0$.

2. Iterate
   1) Generate a random candidate state $x'$ according to $g(x'|x_t)$.
   2) Calculate the acceptance probability $A(x', x_t) = \min\left(1, \frac{P(x')g(x_t|x')}{P(x_t)g(x'|x_t)}\right)$.
   3) Accept or reject:

---

2 aperiodic means the system does not return to the same state at fixed intervals
3 positive recurrent means the expected number of steps for returning to the same state is finite
i) generate a uniform random number \( u \in [0, 1] \);

ii) if \( u \leq A(x', x_t) \), then accept the new state and set \( x_{t+1} = x' \);

iii) if \( u > A(x', x_t) \), then reject the new state, and copy the old state forward \( x_{t+1} = x_t \);

4) Increment: set \( t = t + 1 \).

Note that \( P(x) \) can be replaced by \( f(x) \), if \( f(x) \) is proportional to \( P(x) \). The requirement that \( f(x) \) must only be proportional to the density, rather than exactly equal to it, makes the Metropolis–Hastings algorithm particularly useful, because calculating the necessary normalization factor is often extremely difficult in practice.

It is important to notice that it is not clear, in a general problem, which distribution \( g(x'|x) \) one should use or the number of iterations necessary for proper estimation; both are free parameters of the method, which must be adjusted to the particular problem in hand.

**Metropolis Sampling**

Metropolis sampling is a special case of the Metropolis–Hastings algorithm where the proposal function \( g(x'|x) \) is symmetric. One of frequently-used proposal functions is normal distribution. In these situation, the acceptance probability will be simplified to:

\[
A(x', x) = min\left(1, \frac{P(x')}{P(x)}\right) = min\left(1, \frac{f(x')}{f(x)}\right)
\]  

(2.18)

while the other steps of Metropolis sampling are as same as Metropolis–Hastings algorithm.
Chapter 3
Design of Experiments

The project first implements the random Fourier features method and one-hidden layer neural network method on MNIST dataset. This experiment aims to reveal that if the distribution of frequency parameters is well-chosen, the random Fourier features model and one-hidden layer neural network model will achieve similar prediction accuracy on this classification task.

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits commonly used for training various image processing systems. The dimension of each input data is $28 \times 28 = 784$ and the labels of each data are $0, 1, 2, \ldots, 9$, which are 10 classes. The prediction accuracy is the indicator to measure the performance of different machine learning methods. The algorithms will predict the labels of test data and the prediction accuracy is:

\[
\text{prediction accuracy} = \frac{\text{the number of correctly predicted labels of test data}}{\text{the total data number of the test data}} \times 100
\]

(3.1)

MNIST database is widely used for training and testing in the field of machine learning [17]. Thus, the MNIST dataset will be appropriate and straightforward enough to evaluate the performance of different machine learning models.

In this first experiment three neural network’s activation functions are used, which are sigmoid function, ReLU (Rectified Linear Unit) function and cosine function, respectively. And neural network’s optimizer are applied, which are
SGD and Adam, respectively.

A sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve. The sigmoid function in this thesis is specified as the logistic function and defined by the formula below:

$$s(x) = \frac{1}{1 + e^{-x}}. \quad (3.2)$$

Logistic functions are often used in neural networks to introduce non-linearity in the model or to clamp signals to within a specified interval. A popular neural net element computes a linear combination of its input signals, and applies a bounded logistic function as the activation function to the result; this model can be seen as a "smoothed" variant of the classical threshold neuron.

In the context of artificial neural networks, the ReLU activation function defined as the positive part of its argument:

$$f(x) = x^+ = \max(0, x). \quad (3.3)$$

ReLU activation function was first introduced to a dynamical network by Hahnloser et al. in 2000 with strong biological motivations and mathematical justifications [19] [20]. It was demonstrated for the first time in 2011 to enable better training of deeper networks [21] compared to the widely used activation functions prior to 2011, e.g., the logistic sigmoid (which is inspired by probability theory) and its more practical [4] counterpart, the hyperbolic tangent [22]. The ReLU is, as of 2017, the most popular activation function for deep neural networks [23].

To get an intuitive understanding of these two activation functions, the figures of them are shown in figure 3.1.

Though cosine function is not a frequently-used activation function, it is also involved in this experiment. Since cosine function, which is the Fourier features, i.e., trigonometric features, is used as the activation function in random features model.

The project then implements the adaptive random Fourier features with Metropolis sampling model and one-hidden layer neural network model on a generated dataset (sampled from the designed functions). This experiment intends to indicate that adaptive random Fourier features with Metropolis sampling model
can choose an appropriate distribution of frequency parameters automatically, which reproduces the conclusion of Kammonen’s work [1].

In this experiment, one-hidden layer neural network model still applies three activation functions, which are sigmoid function, ReLU function and cosine function. For the adaptive random Fourier features with Metropolis sampling model, of course, only Fourier features, i.e., cosine function, is used. The last experiment compares the performance of two methods under a classification problem. In contrast, this experiment compares the performance of two methods under a regression problem.

To enhance the interpretability of the comparison, rather than choosing a real-world function, we decide to choose a generated target function, which is defined as:

\[ f(x) = Si\left(\frac{x_1}{a}\right)e^{-\frac{|x|^2}{2}} \quad (3.4) \]

\[ Si(v) := \int_0^v \frac{\sin(t)}{t} dt. \quad (3.5) \]

There are two parameters of the target function, i.e., parameter \( a \) and the dimension of the input data \( x \). To show the target function intuitively, the figures 3.2 to 3.4 plot the target function when the dimension of input data \( x \) is 1 and the parameter \( a = 0.1, 0.03 \) and 0.01 respectively:
Figure 3.2: Target function with $d=1$, $a=0.1$

Figure 3.3: Target function with $d=1$, $a=0.03$

Figure 3.4: Target function with $d=1$, $a=0.01$
The figures reveal that the parameter $a$ determines the steepness of the target function. The smaller $a$ is, the steeper the target function will be, which means higher frequency component in the target function. The function’s high-frequency component is usually the hardest or the latest part for machine learning models to learn.

It is necessary to state the motivation of choosing this target function. On the one hand, this target function is simple enough. It is straightforward to describe the target function and convenient to sample from the the target function. On the other hand, this target function is complicated enough. The smaller parameter $a$ is used, the higher frequency components the function will contain. The high frequency components are usually the intractable problem for the machine learning models to approximate precisely. Thus, the target function can generate hard enough challenge to test the machine learning models and compare the performances of different methods.

In the last experiment, the project following inherits the Metropolis sampling idea and generalizes from Fourier features to ReLU and sigmoid activation functions. To be specific, the project constructs another two similar adaptive random features model with ReLU and sigmoid features and apply these two models on the same dataset, which is sampled from the generated target function. This experiment aims to demonstrate that the adaptive random features model with Metropolis sampling can not only be used on Fourier features but also on some other feature types.
Chapter 4

Implementations and Results

4.1 Compare one hidden-layer neural network and Adaptive random Fourier features on classification task

The first experiment is designed to compare the random Fourier features model with one-hidden layer neural network model on MNIST dataset. The number of train data in MNIST is 60,000, and the number of test data is 10,000.

The project implements the neural network model using the TensorFlow framework with version 2.1.0 and implements the random Fourier features model using MATLAB with version R2020a.

For this experiment, the neural network’s hidden layer activation function is the sigmoid function. To normalize the output of the network, the output layer activation function is the softmax function, which is defined as:

\[ \sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, \ldots, K \text{ and } z = (z_1, \ldots, z_K) \in \mathbb{R}^K. \]  

The loss function is the cross-entropy function, and the iteration method is stochastic gradient descent (SGD). For the parameters of SGD, the learning rate is 1.0, the batch size is 100, and the experiment runs the whole dataset with 10 epochs. The only variable of the neural network in this experiment is
the hidden layer’s width, i.e., the number of nodes, ranging from 256 to 4096. For the random Fourier features model, the number of random features and the distributions of frequency parameters are the variables. Moreover, selecting the distributions of frequency parameters is the crucial part, which is selected personally. And it is pretty experience dependent. After many trials, three appropriate distributions are selected. To be specific, \( N(0, 0.1) \), \( N(0, 0.15) \), which are two normal distribution with both means of the distributions equal 0 and the standard deviations of the distributions equal 0.1 and 0.15 respectively and a uniform distribution \( U(-0.25, 0.25) \), which is an open uniform distribution with the bounds are -0.25 and 0.25 respectively.

The result is shown in figure 4.1

![Figure 4.1: Accuracy comparison of one-hidden layer NN and RF on MNIST dataset](image)

From the result, one can observe that when the number of nodes becomes more extensive, the prediction accuracy of the neural network model does not improve, or even decrease a little bit, since the number of the epoch is limited. As a contrast, when the number of nodes becomes more massive, the prediction accuracy of the random Fourier features model improves significantly. Moreover, one can predict that when the number of nodes becomes large enough, these two models’ prediction accuracy will be similar, though random Fourier features model’s prediction accuracy is a little bit smaller at the current level.

It is worth emphasizing that the random Fourier features model’s training time
is much shorter, which indicates the calculation amount. Since the frequency parameters are sampled from the selected distribution rather than trained from the dataset, the random Fourier feature model’s calculation amount decreased significantly.

The training time for random Fourier features models with different frequency parameters’ distribution is similar. Because sampling the random frequency parameters is not the time determining step which needs \(O(K)\) times calculation. The figure 4.2 compares the training time of these two models.

![Time Cost of Random Features Model and Neural Network Model](image)

**Figure 4.2**: Time comparison of one-hidden layer NN and RF on MNIST dataset

4.2 **Compare one hidden-layer neural network and Adaptive random Fourier features on regression task**

As stated above, in the first experiment, the frequency parameters’ distribution of RF models are selected personally, which may be a tricky step. In the second experiment, the adaptive random Fourier features with Metropolis sampling, which will sample the frequency parameters adaptively, is implemented and
compared with one-hidden layer neural network model on the generated target functions, which is defined in equations (3.4) and (3.5).

The pseudo-code of adaptive random Fourier features with Metropolis sampling is shown in below [1]:

Algorithm 1 Adaptive random Fourier features with Metropolis sampling

**Input:** \( \{(x_n, y_n)\}_{n=1}^N \) \{data\}

**Output:** \( x \rightarrow \sum_{k=1}^K \hat{\beta}_k e^{i\omega_k \cdot x} \) random features

Choose a sampling time \( T \), a proposal step length \( \delta \), an exponent \( \gamma \), a Tikhonov parameter \( \lambda \) and a frequency \( m \) for \( \hat{\beta} \) updates

\( M \leftarrow \text{integer part } (T/\delta^2) \)

\( \omega \leftarrow \text{the zero vector in } \mathbb{R}^{Kd} \)

\( \hat{\beta} \leftarrow \text{minimizer of the problem (1.8) given } \omega \)

for \( i = 1 \) to \( M \) do

\( r_N \leftarrow \text{standard normal random vector in } \mathbb{R}^{Kd} \)

\( \omega' \leftarrow \omega + \delta \gamma N \text{ random walk Metropolis proposal} \)

\( \hat{\beta}' \leftarrow \text{minimizer of the problem (1.8) given } \omega' \)

for \( k = 1 \) to \( K \) do

\( r_u \leftarrow \text{sample from uniform dist. on } [0, 1] \)

if \(|\hat{\beta}'_k|^\gamma / |\hat{\beta}_k|^\gamma > r_u \) Metropolis test then

\( \omega_k \leftarrow \omega'_k \)

\( \hat{\beta}_k \leftarrow \hat{\beta}'_k \)

end if

end for

if \( i \mod m = 0 \) then

\( \hat{\beta} \leftarrow \text{minimizer of the problem (1.8) with adaptive } \omega \)

end if

end for

\( \hat{\beta} \leftarrow \text{minimizer of the problem (1.8) with adaptive } \omega \)

\( x \rightarrow \sum_{k=1}^K \hat{\beta}_k e^{i\omega_k \cdot x} \)

To control the variables, we fix the parameters of the target function as \( a = 0.01 \) and \( d = 3 \). All the target functions in below are same. And both the train dataset and test dataset contain 10000 random samples from the target function. To be specific, 10000 d-dimension samples are sampled from the multi-dimension normal distribution \( N(\mu, \Sigma) \), where \( \mu \) is d-dimension zero vector and \( \Sigma \) is \( d \times d \) dimension identical matrix. And then these samples are transferred to the target function to calculate the corresponding labels. Since the high frequency component of the target function occurs around the zero point, having more data points around zero point will be easier for the machine
learning algorithm to learn the high frequency component. Thus, we choose a multi-dimension normal distribution rather than multi-dimension uniform distribution.

Firstly, we investigate the performance of one-hidden layer neural network with different nodes on the target function. During this part, different combinations of activation functions and optimizers are tested.

When using sigmoid function as the activation function, no matter using SGD or Adam as the optimizer, the generalization error will always be pretty large. For example, setting the number of the nodes in hidden-layer as 512 and using stochastic gradient descent optimizer, if the activation function is ReLU function, the mean square error between the prediction and the label is around 0.0128. However, the mean square error is around 0.6186, which is 40 times larger, if sigmoid activation function is used. This situation reveals that the optimizer always traps into some local minimum, which is relatively large. This phenomenon may be related to gradient vanishing of the sigmoid function. Thus, only the results using ReLU activation function are showed in figure 4.3.

![Figure 4.3: Adam vs SGD: generalization error on target function](image)

In this experiment, if using ReLU function as the activation function, Adam as the optimizer will get a slightly better generalization error than SGD, i.e., the
minimal generalization error using Adam is smaller than using SGD. However, it can not say that Adam is a better optimizer than SGD for this problem. Since adjusting the learning rate of SGD is highly delicate, it may exist a learning rate schedule that works better than Adam. For this experiment, several learning-rate schedules have been tested. Finally, an exponentially decayed learning rate schedule is selected (the initial learning rate is $0.025$, the decay step is $4000$ iterations, and the decay rate is $0.985$). And with this learning-rate schedule, the amplitude of generalization error’s fluctuation of SGD is smaller than Adam, which means SGD is a more steady optimizer than Adam.

However, an interesting phenomenon occurs on both Adam and SGD optimizer. When the width (the number of nodes) is over $500$, the generalization error will not decrease anymore as the width increases. This phenomenon reveals that when the width of the one-hidden layer neural network is large enough, the optimizers can not find a better local minimum even if the width becomes more extensive, which is the bottleneck of one-hidden layer neural network model.

 Supplements:
 1. The maximal epochs of the iteration is $10000$ for both Adam and SGD optimizers;
 2. Both Adam and SGD optimizers use early-stopping strategy with patient=100;
 3. All these results are computed three times and averaged;

Secondly, we investigate the performance differences between one-hidden layer neural network with different activation functions and adaptive random Fourier features with Metropolis sampling. From above experiment, one can know for this regression problem the one-hidden layer neural network performs best when the width is around $500$. Thus, in this part, we set the number of nodes for both models as $512$. Moreover, the activation functions we test are ReLU, sigmoid and cosine (Fourier feature). The optimizers of the neural network are still Adam and SGD.

The result is shown in figure 4.4.

When using the sigmoid activation function, the neural network model performs poorly. Whether using Adam or SGD, the neural network model converges pretty slow, and the converged result is unsatisfactory. For the cosine activation function, the result is similar to the sigmoid activation function for the neural network model, but the result is much better for the random feature model.
Figure 4.4: Adaptive random Fourier features with Metropolis sampling model vs one-hidden layer neural network model: the generalization errors on target function

Though the random feature model with cosine feature is not as good as the neural network model with the ReLU activation function, it is much more stable. For the neural network model with the ReLU activation function, the generalization error fluctuates severely, especially when using Adam optimizer. In contrast, the adaptive random Fourier features model’s generalization error is steady and similar to the training error.

Notably, unlike the neural network model, random feature models’ generalization error will improve if the number of random features increases continually. Thus, we next investigate the performance of adaptive random Fourier features model with a larger number of nodes.

We implement the adaptive random Fourier features model with K equals 512, 1024 and 2048, respectively. From the results in figure 4.5, it is clear that when the number of random features becomes more extensive, the adaptive random Fourier features model’s generalization error decreases steadily.

Figure 4.5: Adaptive random Fourier features with Metropolis sampling model: generalization error on target function with more random features
4.3 Adaptive random features model with different random features

The last experiment investigates adaptive random features model’s performance with Metropolis sampling when using other random features. The adaptive idea with Metropolis sampling provides a systematic method to approximately sample from the optimal distribution $p_*(\omega)$. This adaptive idea may work if it combines with some other random features, which is the original intent of designing this experiment. We still focus on the regression problem to approximate the generated target function.

We select sigmoid function and ReLU function as the new random features. Since these two functions are widely used as activation functions in neural network model. Especially, ReLU function is the activation function that performs best in the above experiments.

There are some hyper-parameters in the adaptive random features model with Metropolis sampling. In this experiment, we do not investigate the influences of these hyper-parameters and inherit these hyper-parameters’ value from the original paper [1] ($\lambda = 1.1, \delta = (2.4^2)/(2d), \gamma = 3d - 2$). Furthermore, we implement the adaptive random features model with cosine (Fourier), ReLU and sigmoid features, respectively.

<table>
<thead>
<tr>
<th>Adv RF M=4000</th>
<th>K=512</th>
<th>K=1024</th>
<th>K=2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cosine</td>
<td>0.0274</td>
<td>0.0201</td>
<td>0.0169</td>
</tr>
<tr>
<td>ReLU</td>
<td>0.0411</td>
<td>0.0287</td>
<td>0.0223</td>
</tr>
<tr>
<td>Sigmoid</td>
<td>0.0697</td>
<td>0.0556</td>
<td>0.0466</td>
</tr>
</tbody>
</table>

Figure 4.6: adaptive random features models with different features and different number of nodes

The results in figure 4.6 shows that the cosine feature works best among these three random features if the hyper-parameters are as same as the original paper. Though the adaptive random features models with ReLU and sigmoid features do not perform as good as Fourier features, ReLU and sigmoid features may also be ideal choices if the hyper-parameters are different. In the following work, we will investigate the influences of the hyper-parameters.
Chapter 5

Conclusions and Discussion

The thesis implements several experiments and investigates the performance differences between different activation functions and optimizers of one-hidden layer neural network and random feature models focus on two specific problems (one classification problem and one regression problem). Furthermore, the thesis accomplishes the adaptive random Fourier feature with Metropolis sampling and combine this adaptive idea with other random features to generate two new similar models. We also make the performance comparison between these three models and one-hidden neural network model.

1. For the two specific problems, choosing ReLU function as the neural network’s activation function will get better results than sigmoid function no matter using SGD or Adam as the optimizer. This phenomenon may be related to the gradient vanish of the sigmoid function.

2. In the experiments, the Adam optimizer usually converges faster than SGD, and the convergence results are usually better. However, it can not directly claim that Adam is a better optimizer than SGD. Because tuning the learning rate of SGD is sophisticated and delicate. A well-tuned SGD may obtain a better result and usually with a more steady process.

3. If the frequency parameters’ distribution is well-chosen, the random feature model will attain a similar prediction accuracy or generalization error as one-hidden neural network model.

4. Since the frequency parameters are sampled rather than trained, the random features model usually require less computation, i.e., faster, than one-hidden neural network model.
5. The thesis reproduces the model adaptive random Fourier features with Metropolis sampling, which can approximately sample frequency parameters from the optimal distribution $p_\star(\omega)$, i.e., select the frequency parameters’ distribution automatically rather than personally.

6. The experiments reveal that the new adaptive random features models which combine the adaptive idea using Metropolis sampling with other random features can also work well. However, if the hyper-parameters are as same as the original paper, the Fourier (cosine) feature works best.

7. The exploratory experiment indicates that the hyper-parameters will affect adaptive random feature models’ performance and determine which random feature fits best. However, the hyper-parameters’ influence is not clear enough now, which needs future works to investigate.
Bibliography


