Numerical Study of a Partial Differential Equation Approach to Deep Neural Networks

by

Helsa Chan

A research paper
presented to the University of Waterloo
in partial fulfillment of the
requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Prof. Justin Wan

Waterloo, Ontario, Canada, 2017

Abstract

Deep neural networks, a technique in machine learning, have had remarkable achievements in various domains, including visual recognition, natural language processing, anomaly detection and text generation. Deep neural networks are typically trained by the stochastic gradient descent method, which seeks the parameters that minimize a non-convex objective function.

Chaudhari et al. [3] have developed four optimization algorithms using a partial differential equation approach. The algorithms, namely ESGD, H-ESGD, HEAT and HJ, have modified the stochastic gradient method by minimizing the objective function smoothened via local entropy, heat equation and Hamilton-Jacobi equations.

This paper provides a numerical study of the performance of the four optimization algorithms under different parameter settings, including the number of iterations per update, amount of smoothing, momentum and mini-batch size. Each of the algorithms are applied to four datasets and deep neural network architectures, and compared to the stochastic gradient descent method based on iteration count and wall clock time. According to the experimental results, stochastic gradient descent still remains the most efficient algorithm, but the new optimization algorithms could bring improvement to the solution under certain conditions.

Acknowledgements

I would like to thank my supervisor Prof. Justin Wan for his guidance towards this project, and Prof. Yaoliang Yu for taking his time to read my paper.

Dedication

This is dedicated to my family.

Table of Contents

List of Figures

Chapter 1

Introduction

Humans perform many activities in daily life without much difficulty. We could easily recognize faces and understand what a person says. Computers, on the other hand, find it much difficult to interpret sentences or identify objects. While we want computers to be able to perform similar tasks, the rules cannot be described to computers explicitly.

Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed [21]. The "machine", often an agent for machine learning algorithms rather than a physical machine, is given a large set of data, in which it discovers and learns the underlying patterns by itself. For instance, given the image of a digit, a machine can "learn" to identify the digit shown in the image after observing a sufficient amount of "examples" containing thousands of images and their corresponding digits shown; similarly, a machine could learn to estimate housing prices based on a dataset containing information of many houses and their corresponding prices. Such information available as inputs are collectively known as "features". For instance, if you evaluate the price of a house based on its size and geographical location, then "size" and "geographical location" are the features of a house; similarly, we can think of each pixel of an image as a feature.

During training, a dataset is pre-divided into the training set, validation set and testing set [9]. First, the machine builds upon its knowledge on the training set based on the input features and the corresponding labels which act as "supervisors" for the machine to learn the parameters. Then, the pre-trained machine is applied on the validation set, which is used to assess the performance of the machine on an unseen dataset and tuning. Finally, the trained machine is applied on the testing set, which is unused during the training process, for a fair evaluation of performance of the machine. Supervised learning [9] is

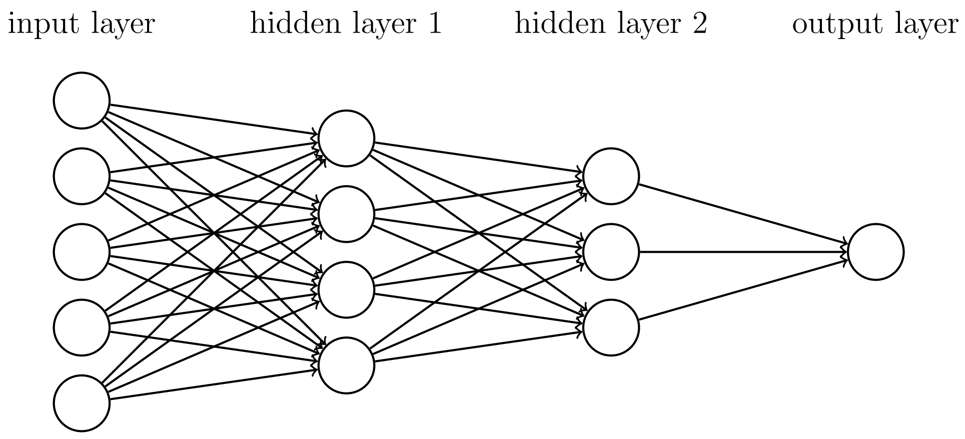


Figure 1.1: A deep neural network

further divided into classification and regression problems. The purpose of a classification problem is to classify an instance into a category. For instance, in the digit recognition problem, the goal is to classify each image into one of the ten possible digits. The purpose of a regression problem is to predict continuous values. For instance, in the house price estimation problem, the goal is to assign each house with a value representing the price of the house.

Among all the machine learning techniques, deep neural networks [7] have become one of the most successful and fastest-growing tools. Deep learning, the study of deep neural networks, have had remarkable achievements in various domains, including visual recognition [12], natural language processing [4], anomaly detection [26] and even text generation [22].

A deep neural network [7], more often conceived as layers with interconnections, is a composition of many mathematical functions. Each input vector of a dataset can be seen as entering the neural network through the first (input) layer, then proceeds to move along the subsequent (hidden) layers, until it exits the last (output) layer producing an output (Figure 1.1). In between layers, there are parameters which decide the weighting of the values sent from a layer to the next layer. Such an architecture provides neural networks with high flexibility to approximate and express the underlying functions which cannot be written explicitly.

In the case of a neural network, the machine undergoes the training process for the best parameters producing the correct outputs on the training set, and is then evaluated according to its performance on the validation set. The question is, how do we quantify the performance of a neural network? One approach is to define a loss function to measure the discrepancy between the desired output and the output values predicted by the trained neural network. A smaller loss implies a better parameter combination and neural network, as the neural network produces predictions that are closer to the correct outputs. Hence, the training goal of a neural network is to seek parameters that minimize the loss function. The challenge lies in solving the optimization problem, in which the objective loss function is very often non-convex and has dimensions as high as millions. Second-order methods such as Newton's method are generally infeasible, as computing the Hessian is complex and of huge cost. While first-order methods such as gradient descent provide an option, ideally we also want a method that converges to a satisfactory solution in a reasonable amount of time. As the loss function is often noisy and contains many local minima, the traditional gradient descent also suffers from the problem of being trapped in local minima, and solutions fail to update any further. Variants of gradient descent have arose to alleviate the problem, with the stochastic gradient descent method (Section 2.4.2) becoming the default choice for training neural networks parameters [7].

Another common issue in deep neural networks is over-fitting, i.e. the parameters give low loss on the training set but high loss on the validation or testing set, implying the parameters fail to generalize to unseen data. This usually happens when the neural network contains too many layers, causing the network to fit "too well" to capture the noise of the training set as well. Fortunately, there are many regularization remedies to reduce over-fitting (Section 2.3.3); techniques such as batch normalization [10] (Section 2.2.1) also leads to faster learning and better solution in general.

Chaudhari et al. [3] have recently proposed a new model which establishes connections between the optimization of deep neural networks and partial differential equations. In particular, the solutions of the Hamilton-Jacobi equations and a new function called the local entropy [1] are found to smoothen the loss function of deep neural networks. It is anticipated that the four algorithms ESGD, H-ESGD, HEAT and HJ, which all apply a different form of smoothing to the loss function, could bring improvements to the generalized error of deep neural networks. Based on the theoretical results discussed in the paper, we are interested in a more comprehensive numerical study of the algorithms. We will compare their effectiveness with stochastic gradient descent method, the traditional way to train deep neural networks.

This research paper is organized as follows. Chapter 2 provides readers with the necessary background in machine learning and deep neural networks. Chapter 3 gives a summary of the previous results by Chaudhari et al. [3]. The numerical study is discussed in Chapter 4. Finally, we give a conclusion in Chapter 5.

Chapter 2

Background

Machine learning allows a machine to learn underlying patterns of a given data and perform tasks such as digit recognition and house prices estimation. Different machine learning algorithms allow machines to learn under different mechanisms. For deep neural networks, our ultimate goal is to seek the parameters to give accurate and efficient predictions.

We will first introduce fully connected neural networks and convolutional neural networks. Then, we will discuss how a deep neural network is formulated into an optimization problem, trained, and evaluated.

2.1 Deep Neural Networks

A neural network is a composition of many mathematical functions with a high number of parameters. It is called a "neural network" because it is conceived as a sequence of layers with interconnections, resembling a neural network. The number of layers determine the "deepness" of a neural network.

We will first introduce fully connected neural networks, the simplest kind of deep neural networks. Then, we move on to convolutional neural networks [7], a variation of fully connected neural networks involving convolutions, which are more complex but often give better predictions for image datasets.

2.1.1 Fully Connected Neural Networks

Mathematically, a fully connected neural network is written as a nested composition of M functions $\sigma_1, \ldots, \sigma_M$, where σ_i is a composition of a linear function g_i and a non-linear function h_i. We use d_i to denote the number of outputs of σ_i. We define d_0 to be the dimension of an input instance (or equivalently, the number of features).

Denote z_i the input at layer i. The linear function g_i can be alternatively written as

$$g_i(z_i) = W_i z_i + \vec{b}_i$$

or

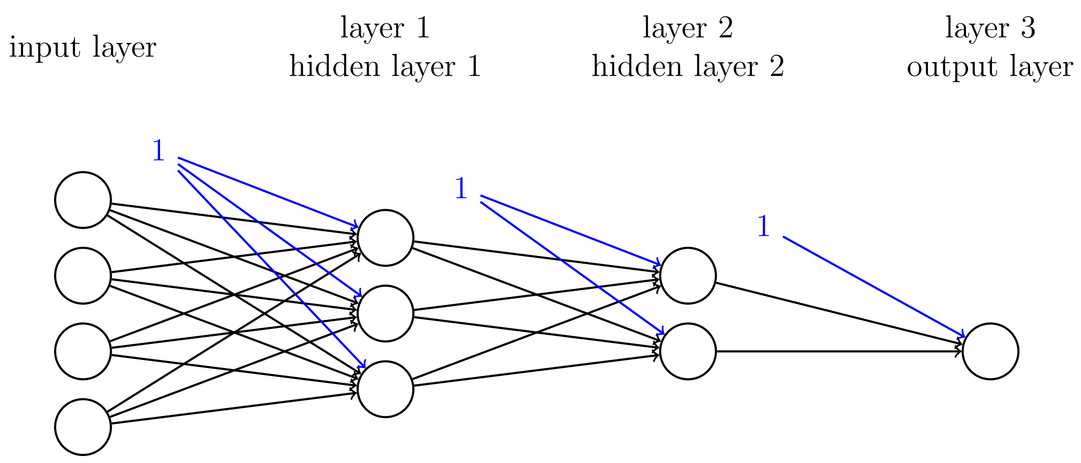
$$g_i(z_i) = \begin{pmatrix} \vec{b}_i & W_i \end{pmatrix} \begin{pmatrix} 1 \\ z_i \end{pmatrix}, \tag{2.1}$$

where $W_i \in \mathbb{R}^{d_i \times d_{i-1}}$ is the weight parameter and $\vec{b}_i \in \mathbb{R}^{d_i}$ is the bias parameter. The non-linear function h_i, which provides more flexibility to the network, applies an activation function $h : \mathbb{R} \to \mathbb{R}$ to each element of $g_i(z_i)$, giving an output of dimension d_i. More information on how to choose h is described in Section 2.2.3.

For each input ξ, the overall function is given by

$$\pi(\xi) = \sigma_M(\sigma_{M-1}(\ldots \sigma_1(\xi)) \ldots) = h_M(g_M(h_{M-1}(g_{M-1}(\ldots h_1(g_1(\xi))) \ldots). \tag{2.2}$$

We can portray (2.2) as a sequence of layers. Figure 2.1 shows the architecture of a simple fully connected neural network of two hidden layers. The term "fully connected" means the nodes on two consecutive layers form a fully connected bipartite graph with weights along the edges. Black and blue edges represent weight parameter terms and bias terms respectively. Each layer is referred to as a "fully connected layer", and the d_i dimensions of each function output are often referred to as "units" or "nodes" of a layer. For each layer i, the extra input "1" is introduced to include the bias term. It is then easier to use the matrix multiplication as in (2.1) to think of the linear transition from layer i to layer $i+1$. Although the multiplication vector and matrix now have one extra dimension, the extra node is excluded when we describe the number of nodes of the layer. Each of the d_i linear outputs is then passed through the activation function to give the final output of the layer $z_{i+1} = h_i(g_i(z_i))$, which is also the input of the next layer. This process repeats until the output layer is reached.



$$\xi = z_1 \xrightarrow[\sigma_1]{h_1(W_1 z_1 + b_1)} z_2 \xrightarrow[\sigma_2]{h_2(W_2 z_2 + b_2)} z_3 \xrightarrow[\sigma_3]{h_3(W_3 z_3 + b_3)} z_4 = \pi(\xi)$$

$$d_0 = 4 \qquad d_1 = 3 \qquad d_2 = 2 \qquad d_3 = 1$$

Figure 2.1: A fully connected neural network with 2 hidden layers.

2.1.2 Convolutional Neural Networks

A convolutional neural network is a special type of fully connected neural networks which involves the use of a convolutional layer. Convolutional neural networks are especially useful for processing input data that contains a spatial structure, for instance images, which can be seen as 2D grids of pixels.

Given an image, a convolution operation can be viewed as the weighted sum of the neighboring pixels. Figure 2.2 shows a simple example of the operation of a 2D convolution. Suppose we have a 3×4 image and a 2×2 matrix, which is called a "kernel" and contains some values determining the weights. Then, there are 2×3 ways to place the kernel matrix right above of the image, so that all the elements of the kernel coincide with the elements of the image. For each possible placement of the kernel, we calculate the sum of the element-wise product of the kernel and the 2×2 block of the image. This results in 2×3 sums corresponding to different positions of the image. The resulting 2×3 matrix is the convolution of the image and the kernel, often called the "feature map".

The kernel can be thought of as a "sliding window" which allows the machine to focus on a small region of the whole picture. Different weights of the kernel correspond to different

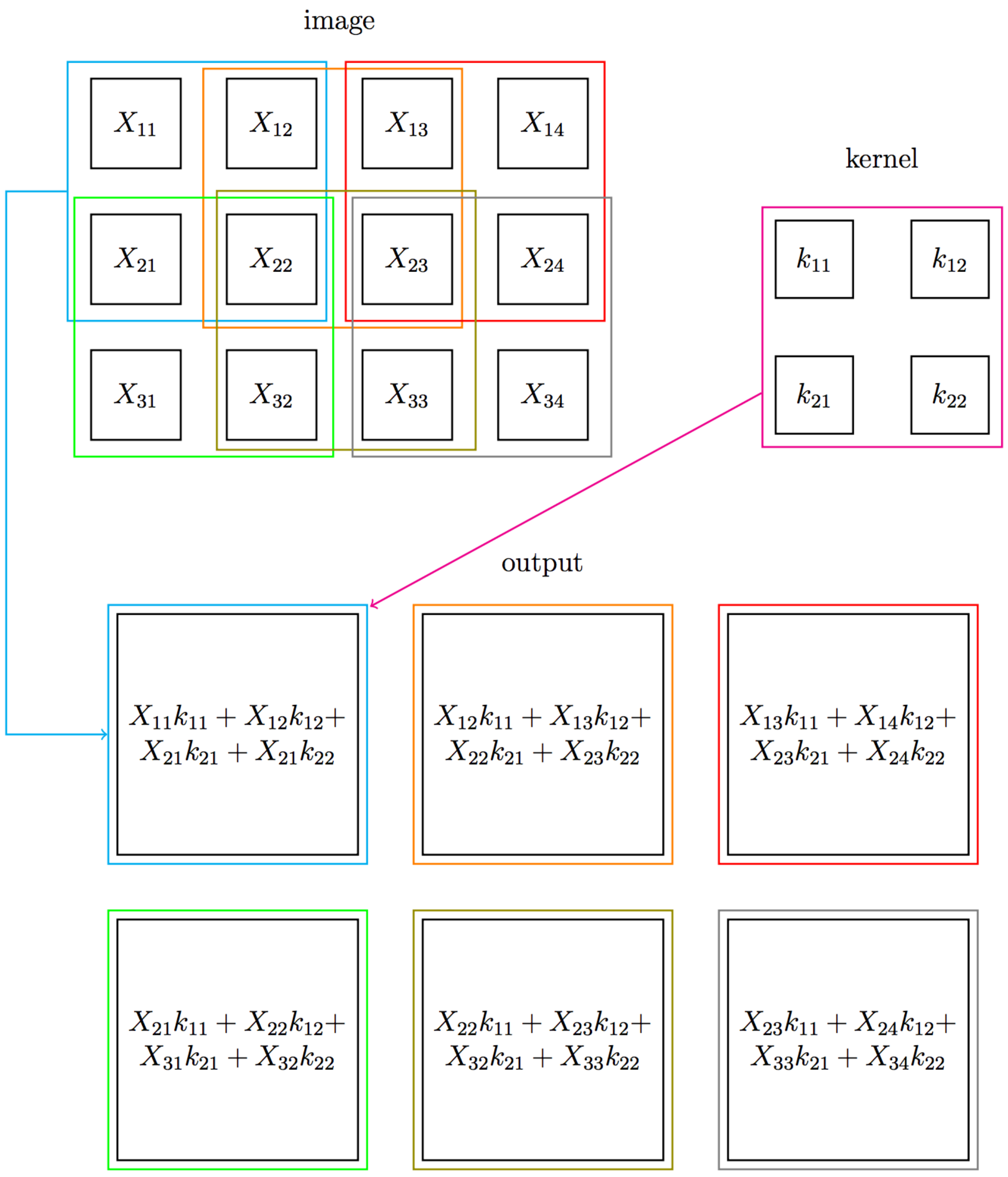


image
X_{11}
X_{12}
X_{13}
X_{14}
X_{21}
X_{22}
X_{23}
X_{24}
X_{31}
X_{32}
X_{33}
X_{34}
kernel
k_{11}
k_{12}
k_{21}
k_{22}
output
$X_{11}k_{11} + X_{12}k_{12} + X_{21}k_{21} + X_{21}k_{22}$
$X_{12}k_{11} + X_{13}k_{12} + X_{22}k_{21} + X_{23}k_{22}$
$X_{13}k_{11} + X_{14}k_{12} + X_{23}k_{21} + X_{24}k_{22}$
$X_{21}k_{11} + X_{22}k_{12} + X_{31}k_{21} + X_{32}k_{22}$
$X_{22}k_{11} + X_{23}k_{12} + X_{32}k_{21} + X_{33}k_{22}$
$X_{23}k_{11} + X_{24}k_{12} + X_{33}k_{21} + X_{34}k_{22}$

Figure 2.2: An example of a 2D convolution

image features that we want to extract. For instance, in digit recognition, if we want to extract the feature "1", a suitable kernel would be one that captures the vertical stroke characteristics of the digit "1". The size and number of kernels correspond to the "width of the window" and "number of windows" respectively. The specific weights of the kernel are the parameters that we want the machine to learn. We can similarly define convolution for higher dimensions.

Unlike fully connected layers, the parameters or weights in a convolutional layer are shared among all the pixels in the image in the form of a kernel matrix. In the case of Figure 2.2, the 3×4 image is first reshaped into a 12-dimensional vector, then left-multiplied by the 6×12 weight matrix, which is a sparse matrix containing non-zero elements $k_{11}, k_{12}, k_{21}, k_{22}$ corresponding to the kernel matrix, giving the 6-dimensional output which is reshaped back to 2×3. The bias term is zero. Since the weights are shared, a convolutional neural network often has fewer parameters to train compared to the fully connected layers.

A convolutional layer is often followed by a pooling layer to allow feature extraction. The pooling layer also has a sliding window for feature selection. The pooling layer provides summary statistics of the neighboring outputs. For example, the max pooling layer gives the maximum value within a neighborhood of the image, as demonstrated in Figure 2.3.

Up till now, convolutions and pooling are performed using the "sliding window" kernel which shifts 1 pixel at a time. When the size and number of kernels get large, the amount of computation is huge. To increase efficiency, it is common to introduce a "stride" which determines the number of pixels that the window is shifted before one operation is performed. The resulting neural network is smaller, and the image is downsampled by a factor of the stride. Figure 2.4 gives an example of a 2×2 maximum pooling with stride 2.

2.2 Techniques in Deep Neural Networks

2.2.1 Batch Normalization

Batch normalization [10] follows the same procedures as usual normalization, i.e. normalizing outputs of a linear layer by subtracting the mean and dividing by the standard deviation, except that the mean and standard deviation are approximated using a small subset of randomly selected instances called the mini-batch. The overall effect is still a linear function, but performing batch normalization generally makes training more robust by avoiding very large or small outputs that usually accumulate along the network. In practice, batch normalization is optionally performed after the activation function of each hidden layer.

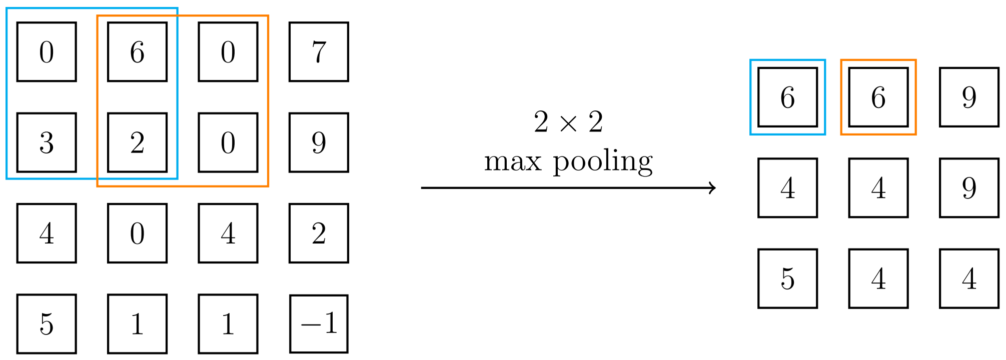


Figure 2.3: An example of 2×2 max pooling

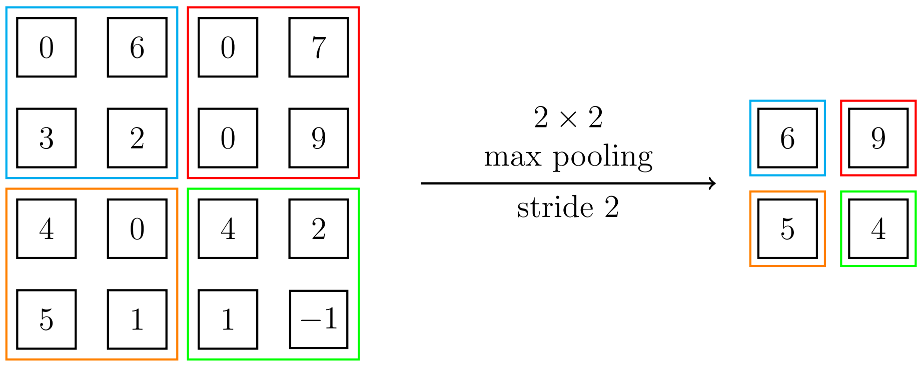


Figure 2.4: An example of 2×2 max pooling with stride 2

2.2.2 Data Pre-processing

Data pre-processing refers to the process of modifying, selecting, refining and tidying up data to prepare them for the training stage.

Data pre-processing is necessary when the raw collected data is problematic for training. For instance, missing values during the data collection stage must be handled, since training cannot proceed with incomplete data. One solution is to simply discard all the instances containing missing values; another solution is to perform imputation, i.e. we fill in the missing values with the mean, median or mode of the particular feature. Another example of data pre-processing is the normalization of input data, which is especially important when different features of the input have very different ranges of values. For image datasets, it is typical to rescale the values linearly to numbers between 0 and 1.

In other cases, data pre-processing is performed as an optional procedure to improve neural network performance. Feature selection and data cleansing are often performed to select relevant features from a dataset before passing to training. When it comes to categorical variables, it is also common to use one-hot encoding, i.e. transform the variables into boolean columns to indicate whether an instance fall into each category.

2.2.3 Choosing Activation Functions

The activation function $h : \mathbb{R} \mapsto \mathbb{R}$ not only alters the outputs of the linear function for a more flexible network, but also allows extraction of useful information.

The activation function h is chosen depending on the machine learning task. For instance, if we are performing binary classification, we want the single output value to be a binary value indicating whether an instance belongs to a class. A suitable choice would be the threshold function

$$h(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases},$$

where "1" indicates the input belongs a certain class and "0" otherwise. A"softer" version of the threshold function is the sigmoid function defined as

$$h(z) = \frac{1}{1 + \exp(-z)}, \tag{sigmoid}$$

which squishes a real value to $(0,1)$ and outputs a probability value. The softmax function defined as

$$h(z_k) = \frac{\exp z_k}{\sum_j \exp z_j}$$

generalizes this effect to multi-class classification, in which the output values and represent a categorical distribution. Thus, the number of nodes on the output layer is the same as the number of classes, and the output values add up to 1. For the case of regression, we set the activation function to be the identity function because there is no need to alter the linear outputs.

For hidden layers, the preferred activation functions are those with easily computable derivatives within an appropriate range, as they facilitate the training process of the neural network. The derivatives of the activation functions in the hidden layers are related to the speed of parameters update.

Gradients that are close to zero should be avoided, as they cause the network to learn very slowly. The sigmoid function has the derivative

$$\frac{d}{dz}\frac{1}{1+\exp(-z)} = \frac{\exp(-z)}{(1+\exp(-z))^2},$$

which is very close to zero for most $z \in \mathbb{R}$. During the training process which involves the gradient descent method (Section 2.4.1), the parameters would then be updated extremely slowly. Gradients that are very large should also be avoided, as they cause the parameters to change too quickly and "overshoot" during the gradient descent. The neural network's training progress would all be lost as a result.

One appropriate choice of the activation function for hidden layers is the rectified linear units (ReLUs)[17]

$$h(x) = \max\{0, x\}.$$

Rectified linear units[1] are found to give satisfactory results when used as activation functions for the hidden layers. We will also use ReLUs as the default activation function choice for our experiments.

[1]Note that ReLUs are not differentiable at 0 [17]. We define the derivative of ReLUs at 0 to be 0.

2.2.4 Constructing Deep Neural Networks

In a given machine learning problem, the input and output dimensions are often pre-determined. Before proceeding to training, the number of hidden layers and nodes on each hidden layer have to be determined.

Deep neural networks are known for its flexibility and capability to capture the complex underlying function. A higher number of hidden layers and nodes allows more "twists" in the function represented by the neural network, and are capable in expressing more complex functions. However, if the number of hidden layers and nodes are set too high, the neural network not only takes a longer time to train, but could also increase the generalized error and lead to the problem of overfitting. Section 2.3.2 further elaborates this point.

In general, the more amount of input data, the more hidden layers and nodes are required. However, the specific numbers chosen are often heuristic. The number of hidden nodes on each layer can be as large as hundreds or thousands, and hence the number of parameters can easily reach the millions.

Once we have decided the architecture of the deep neural network, the next step is to train the network.

2.3 Training Deep Neural Networks

2.3.1 Loss Function

To quantify the performance of a neural network, we define a loss function to measure the discrepancy between the desired output and the output values predicted by the trained neural network. A smaller loss implies a better parameter combination and neural network, as the network produces predictions that are closer to the correct outputs. The training goal of a neural network is to seek parameters W_i and $\vec{b}_i$ that minimize the loss function.

For convenience, we will use x to denote the parameter vector that is formed by re-shaping and concatenating all the W_i's and $\vec{b}_i$'s:

$$x := \begin{pmatrix} \vec{w}_1 \\ \vec{b}_1 \\ \vdots \\ \vec{w}_M \\ \vec{b}_M \end{pmatrix}$$

where $\vec{w_i}$ is obtained by reshaping all the elements of W_i into a long vector.

For an input ξ_i, we use $f_i(x)$ to denote the discrepancy between the prediction $\pi(x;\xi_i)$ and the actual desired output π_i. Let N be the number of training examples. The overall objective is the empirical loss function that takes the average of all the discrepancies

$$f(x) := \frac{1}{N}\sum_{i=1}^{N} f_i(x). \tag{2.3}$$

There are many ways to define f_i. In a classification problem with C classes, the final output $\pi(x;\xi_i)$ is a length-C vector representing the categorical distribution of ξ_i, and the actual desired output π_i is represented as another length-C vector, with value 1 for the desired class and 0 otherwise. To compare the two outputs, we usually choose the cross entropy

$$f_i(x) := -\sum_{c=1}^{C} \pi_{ic} \ln \phi_{ic},$$

where

$$\pi_{ic} = \begin{cases} 1 & \text{if } \xi_i \text{ belongs to class } c \\ 0 & \text{otherwise} \end{cases}$$

is the c-th element of π_i, and ϕ_{ic}, which predicts the probability of ξ_i falling into class c, is the c-th element of $\pi(x;\xi_i)$.

Ideally, we want the final outputs to indicate a high probability for the correct class and low probabilities for the wrong classes. The cross-entropy produces a small value when π_i is close to $\pi(x;\xi_i)$ and a large value when π_i is far away from $\pi(x;\xi_i)$. By minimizing the cross-entropy, the deep neural network is encouraged to be more confident in its predictions.

To perform the classification task, an input instance is classified to the class with the highest probability output. We can then compare the predicted class with the correct class and evaluate the performance of the neural network by the accuracy, i.e. the percentage of instances that are classified into the correct class. A higher accuracy indicates that the neural network has better predictions. In most cases, the accuracy of the neural network is reported as the performance of the neural network.

In regression problems, the usual choice is the square loss given by

$$f_i(x) := \left\| \pi(x;\xi_i) - \pi_i \right\|^2,$$

which is non-negative. It is common to report the loss value directly as the performance of the neural network.

Our goal is to seek the parameters x^* to minimize $f(x)$, i.e. to solve the optimization problem

$$x^* = \arg\min_x \frac{1}{N} \sum_{i=1}^{N} f_i(x). \tag{2.4}$$

In most situations, this is a non-convex minimization problem with a high number of parameters.

2.3.2 Overfitting and Underfitting

One common issue in deep neural networks is overfitting, i.e. the parameters give low loss on the training set but high loss on the validation or testing set, implying the parameters fail to generalize to unseen data (Figure 2.5). This usually happens when the neural network contains too many layers, causing the network to fit "too well" to capture the noise of the training set as well. As opposed to overfitting, underfitting is the problem that occurs when the network does not express the data well enough.

The bias-variance tradeoff [9] studies the balance between underfitting and overfitting. Bias refers to the generalized error caused by the limited flexibility of the neural network, and a high bias indicates underfitting; on the other hand, the variance refers to the neural network's sensitivity to the training data, and a high variance indicates overfitting.

The expected generalization error of a neural network can be decomposed into the bias, variance, and irreducible error caused by the noise in collecting data. For instance, suppose we have a regression problem with one output. The expected square loss on an unseen input ξ' and actual output π' is given by

$$\begin{aligned} E[(\pi(x;\xi') - \pi')^2] &= \underbrace{E[\pi(x;\xi') - \pi']^2}_{\text{bias}} + \underbrace{E[\pi(x;\xi')^2] - E[\pi(x;\xi')]^2}_{\text{variance}} \\ &= \text{bias}^2 + \text{variance} + \text{irreducible noise}. \end{aligned}$$

In general, when a neural network has a higher complexity (i.e. contains more hidden layers and nodes), the bias decreases and the variance increases. Ideally, we want a model that has appropriate complexity and strikes a balance between the bias and variance, so that the expected error is minimized. To reduce underfitting, we can simply expand the neural network by adding more nodes and layers to reduce underfitting. Other remedies in regularization (Section 2.3.3) can also be applied to mitigate overfitting.

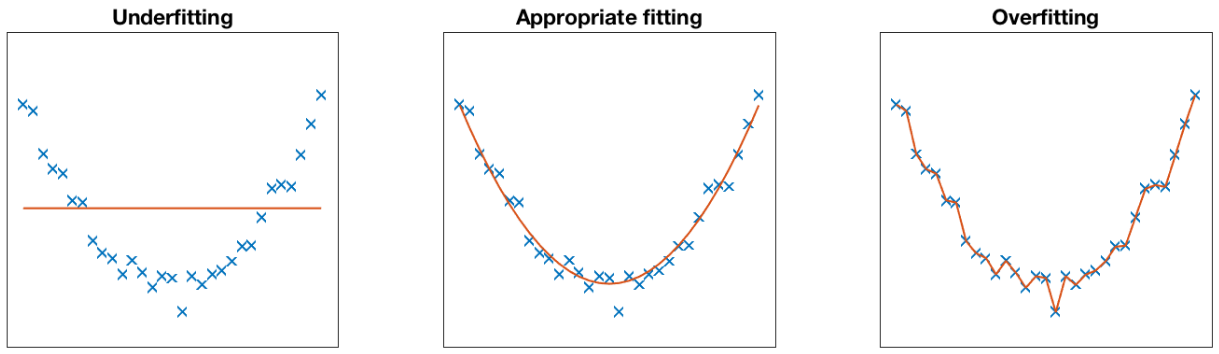


Figure 2.5: Underfitting, appropriate fitting and overfitting

2.3.3 Regularization

Regularization is a common technique in deep learning which aims to modify the loss function, so that the generalized error is reduced instead of the training error.

One example to avoid overfitting is to introduce a dropout layer, which explicitly sets a random fraction of units to zero and implicitly "dropping" them out from the neural network. Another way to apply regularization is to add a penalty term $\frac{\lambda}{2}\|x\|_2^2$ to the loss function $f(x)$, where $\lambda > 0$ controls the weight of the regularization term, so that the parameters are encouraged to be smaller in magnitude and does not overly favor a good loss in the training examples. A similar method is to smoothen the loss function, so that the parameters do not over-fit to the small fluctuations existing in the underlying function $\pi(x;\xi)$. Section 3 provides a summary of Chaudhari et al. [3] for their applications of local entropy to smoothen the loss function.

2.4 Optimization in Deep Neural Networks

In optimization, the traditional way of minimizing a non-convex function is gradient descent. However, in deep learning where accuracy, efficiency and computational costs are all crucial, gradient descent may not be the most appropriate optimization algorithm.

In this section, we will first discuss the application and drawbacks of gradient descent. Then, we introduce the stochastic gradient descent algorithm, a variation of gradient descent which becomes the default optimization algorithm choice in training deep neural networks. We will also briefly discuss other optimization algorithms, including AdaGrad, RMSProp and Newton's method.

2.4.1 Gradient Descent

Gradient descent (or steepest descent) [19] is a traditional method in non-convex optimization. Given the objective function $f(x)$, we perform the update

$$x^{k+1} = x^k - \eta_k \nabla f(x^k) \tag{2.5}$$

with any initial guess x^0. The parameter $\eta_k > 0$ is called the step-size or learning rate at iteration k.

In the case of training deep neural networks, the gradients are computed based on the training examples. Recall the objective $f(x)$ which is the average of discrepancies of all the training examples. Using equation (2.3), the update is given as

$$x^{k+1} = x^k - \frac{\eta_k}{N} \sum_{i=1}^{N} \nabla f_i(x^k). \tag{2.6}$$

There are two major drawbacks of gradient descent. Firstly, gradient descent can get stuck at a local minimum and fail to give updates, especially when the underlying function is noisy. Secondly, from (2.6) we see that the gradient descent is calculated based on the whole training set. A large dataset also requries high computational time. These issues make gradient descent method impractical for training of large data sets.

2.4.2 Stochastic Gradient Descent

Stochastic gradient descent (SGD) [7] is developed to address the drawbacks of gradient descent. It is similar to the usual gradient descent (2.5), except that the gradient is approximated using a randomly selected sample from the training dataset. The stochastic gradient descent procedure is written as

$$x^{k+1} = x^k - \eta_k \nabla f_{i_k}(x^k), \tag{2.7}$$

where i_k is sampled uniformly from the set $\{1, \ldots, N\}$ and represents one of the N training examples. Since the time to compute the gradient based on one randomly selected sample is much shorter, stochastic gradient descent could take many more steps and updates within a fixed amount of time. Unlike gradient descent which steadily moves the solution towards a direction that decreases the loss value, stochastic gradient descent has a much noisier trajectory and does not easily get stuck at a local minimum. The overall empirical performance of stochastic gradient descent is that the loss value decreases as the number of iterations increases, but there could be small fluctuations in the loss value.

However, the gradient calculated based on only one randomly selected sample to represent the overall gradient is noisy and highly inaccurate. It is more common to avoid either extremes and take the balance by calculating the average of gradients evaluated using a "mini-batch", which is a subset of the training set. This variation of stochastic gradient descent is sometimes also called the "mini-batch gradient descent". Let $B \subset \{1, \ldots, N\}$, we denote the mini-batch gradient as $\nabla f_B(x)$, and the update rule becomes

$$\begin{aligned} x^{k+1} &= x^k - \eta_k \nabla f_B(x^k) \\ &= x^k - \frac{\eta_k}{|B|} \sum_{i \in B} \nabla f_i(x^k). \end{aligned} \tag{2.8}$$

In practice, we will first shuffle the order of the entire training set, and then divide the queue of training sets into mini-batches of a fixed mini-batch size N_{mb}. Then, we evaluate the gradient based on each mini-batch. We use the first mini-batch to perform the first iteration, and then the next mini-batch for the next iteration, and so on, until we go through all the mini-batches. The training set is re-shuffled and divided into mini-batches and the iteration continues. An "epoch" is defined as the number of iterations required to go through all the mini-batches, given by $\lceil N/N_{mb} \rceil$, and it is conventional to report the performance of a deep neural network in terms of epochs.

It is necessary to reduce η_k as k increases in practice [7]. The stochastic gradient descent algorithm guarantees solution convergence if

$$\sum_{k=1}^{\infty} \eta_k = \infty \quad \text{and} \quad \sum_{k=1}^{\infty} \eta_k^2 < \infty.$$

Choosing the learning rate is a trial-and-error process and there is no absolute right answer. The usual practice is $\eta_k = O(k^{-1})$, or decrease the initial learning rate η_0 linearly until iteration τ, then keep η constant after iteration τ. Let η_0 be the initial learning rate, a possible setting is

$$\eta_k = \max \left\{ \frac{\eta_0}{1 + 0.5k}, 0.01\eta_0 \right\}. \tag{2.9}$$

The values of η_0 are specified in the corresponding experiments.

A modification of the stochastic gradient descent involves the Nesterov momentum [23], which often results in faster convergence. The idea of the Nesterov momentum is to take the previous movement into account when deciding the movement for the current iteration. The gradient step often then escapes from a local minimum and is less prone to the large changes of directions.

To obtain stochastic gradient descent with Nesterov momentum, we replace the update in (2.7) by:

$$\begin{aligned} v^{k+1} &= \mu v^k - \eta_k \nabla f_B(x^k + \mu v^k) \\ x^{k+1} &= x^k + v^{k+1} \end{aligned} \tag{2.10}$$

where μ, the momentum, is typically chosen to be $0.5, 0.9$ or 0.99. It is worth noting that if we denote $x_* = x + \mu v$ and rename x_* back to x, then the above update becomes

$$\begin{aligned} v^{k+1} &= \mu v^k - \eta_k \nabla f_B(x^k) \\ x^{k+1} &= x^k - \mu v^k + (1+\mu) v^{k+1}. \end{aligned} \tag{2.11}$$

This form is more similar to the original stochastic gradient descent update, and hence is easier to apply to variations of stochastic gradient descent. For instance, if a given method has the form

$$x^{k+1} = x^k - \eta_k \phi(x_k),$$

we can easily modify (2.11) to obtain the corresponding Nesterov form

$$\begin{aligned} v^{k+1} &= \mu v^k - \eta_k \phi(x_k) \\ x^{k+1} &= x^k - \mu v^k + (1+\mu) v^{k+1}. \end{aligned} \tag{2.12}$$

Recall that the parameter x is a reshaped and concatenated vector formed by the weight parameters W_i and bias parameters b_i occurring in layer i. Each element of x as parameters of the neural network do not appear as direct inputs of a function, but are arranged in a hierarchical manner in the network. Using the chain rule in advanced calculus, the gradient of f with respect to a parameter in the earlier part of the neural network is written in term of the gradients with respect to parameters in the later part of the neural network. This process of calculating gradients is called "backpropagation" [7][20], which means changes in the output layer are propagated backwards along the network.

The backpropagation process naturally updates parameters on the same layer of similar values with a similar amount of change. If all the parameters are set to the same value, the neural network could fail to converge to the appropriate underlying function as all the parameters on the same layer would change in the same manner. A small amount of variance is necessary to break the symmetry in the neural network architecture. In addition, setting x^0 close to zero has the advantage of avoiding z_i from getting too large and causing overflow errors. Hence, the initial guess x^0 is typically set to be random values taken from the uniform distribution with mean zero and small variance.

2.4.3 Other Optimization Algorithms

Besides stochastic gradient descent, AdaGrad [5] and RMSProp [7] are also popular first-order methods in deep neural networks. The AdaGrad algorithm is a variation of gradient descent with an adaptive learning rate for each parameter: parameters with a larger partial derivative have a larger decrease in learning rate, while parameters with a smaller partial derivative have a relatively smaller decrease in their learning rate. However, AdaGrad works well on only some of the deep neural networks [7]. The reason is that AdaGrad adapts the learning rate based on the entire history of the training process, which could possibly make the learning rate too small in the later part of training. The RMSProp algorithm modifies AdaGrad's gradient accumulation setting to an exponentially decaying average, hence discarding the history from the early iterations. The resulting algorithm is empirically effective and has satisfactory performance on deep neural networks.

In optimization, second order methods such as Newton's method [7] are usually seen as having faster convergence in non-convex optimization problems. However, in the case of deep neural networks, the number of parameters is often in the millions. Computing the Hessian is complex and of huge cost, especially when the Hessian has to be computed for each iteration. In addition, Newton's method is suitable only when the Hessian is positive definite. These requirements have limited the application of Newton's method to the case when the neural network has very few parameters. Hence, Newton's method is generally seen as impractical for training deep neural networks.

Chapter 3

Methodology

This chapter provides a summary of Chaudhari et al. [3] on the connections between non-convex optimization methods for training deep neural networks and the theory of partial differential equations (PDE). The proposed main idea is to apply stochastic gradient descent on the smoothened version of the original loss function $f(x)$ for a better chance to converge to an improved solution.

In the previous results, the viscous and non-viscous Hamilton-Jacobi equations (HJ equations) studied turn out to have solutions that smoothen the original loss function as the partial differential equations evolve with time. In addition, the heat equation and homogenization of stochastic differential equations are studied to provide more variations in terms of smoothing. As a result, four new optimization algorithms are proposed. Each of the new algorithms, namely ESGD, H-ESGD, HEAT and HJ, corresponds to a different method to apply smoothing to $f(x)$ and are beneficial to the optimization procedure in different ways.

The methods of smoothing can be further categorized into three main "smoothing tools", namely local entropy, heat equation, and inf-convolution. We will begin this chapter by introducing the smoothing of loss function performed by the local entropy. We will study the viscous Hamilton-Jacobi equation, which has the local entropy of $f(x)$ as its solution. We will explore the PDE properties of the local entropy and subsequently show that the evolution of the time-scale can be identified with the smoothing parameter of the local entropy. The local entropy gives rise to two additional optimization algorithms: the local entropy-equivalent Elastic-SGD algorithm, known as ESGD, and its homogenized version, known as H-ESGD. Next, we introduce the HEAT algorithm, which is similar to ESGD but instead derived from the heat equation. Finally, we introduce the smoothing method by the inf-convolution. We will study the non-viscous Hamilton-Jacobi equation, which has a solution $u(x,t)$ that preserves the local minima of $f(x)$ and widens the convex region of $f(x)$. We will derive the HJ algorithm, which performs gradient descent on $u(x,t)$ instead of $f(x)$, with an elegant expression for $\nabla_x u(x,t)$ which does not depend on the dimension of x.

All the algorithms listed in this section are variations of stochastic gradient descent. Hence it should be well understood that during the actual implementation, all gradients are calculated based on a mini-batch of data. The algorithms are slightly modified to facilitate the implementation of the numerical study.

3.1 Smoothing of the Loss Function by Local Entropy and Homogenization

Under certain assumptions, the local entropy smoothens the original loss function $f(x)$ and leads to improved generalized performance in deep neural networks. This yields the ESGD algorithm, which performs stochastic gradient descent on the local entropy of $f(x)$.

We will also introduce a tool called "homogenization" in stochastic differential equations, which achieves a smoothing effect on the loss function by considering the averaged solution for a system of stochastic differential equations with rapidly changing coefficients. This method yields the H-ESGD algorithm, i.e. the homogenized version of the ESGD algorithm.

3.1.1 Local Entropy and the Viscous Hamilton-Jacobi Equation

The local entropy of f is defined as

$$f_\gamma(x) = -\log\left[G_\gamma(x) * e^{-f(x)}\right],$$

where $G_\gamma(x) = (\sqrt{2\pi\gamma})^{-d} e^{-\frac{\|x\|^2}{2\gamma}}$ is the heat kernel, d is the dimension of x, and $*$ is the convolution operation defined as

$$(f * g)(x) = \int f(x-y)g(y)dy.$$

The parameter γ controls the variance of the Gaussian smoothing kernel; a larger γ can be seen as more amount of smoothing applied.

The first result is given by the following theorem, which shows that the local entropy $f_\gamma(x)$ with $\gamma = t$ is the solution of the viscous Hamilton-Jacobi equation given as

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{1}{2}\left|\frac{\partial u}{\partial x}\right|^2 &= \frac{1}{2}\Delta u \\ u(x,0) &= f(x), \end{aligned} \tag{3.1}$$

where Δ is the Laplacian.

Theorem 3.1.1. *The local entropy $f_\gamma(x)$ with $\gamma = t$ is the solution of the initial value problem for the viscous Hamilton-Jacobi equation.*

Proof. Define $v(x,t) = e^{-f_t(x)} = G_t(x) * e^{-f(x)}$. Then, $v(x,t)$ is the solution of the heat equation $\frac{\partial v}{\partial t} = \frac{1}{2}\Delta v$ with initial condition $v(x,0) = e^{-f(x)}$. Let $u(x,t) = -\log v(x,t)$, then

$$\frac{\partial v}{\partial t} = -v\frac{\partial}{\partial t}f_t(x)$$

and it follows that

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{1}{2}\left|\frac{\partial u}{\partial x}\right|^2 &= -\frac{1}{v}\frac{\partial v}{\partial t} + \frac{1}{2}\left|-\frac{1}{v}\frac{\partial v}{\partial x}\right|^2 \\ &= -\frac{1}{v}\frac{\partial v}{\partial t} + \frac{1}{2v^2}\left|\frac{\partial v}{\partial x}\right|^2. \end{aligned}$$

On the other hand,

$$\begin{aligned} \frac{1}{2}\Delta u &= \frac{1}{2}\sum_{i=1}^{n}\frac{\partial^2 u}{\partial x_i^2} \\ &= \frac{1}{2}\sum_{i=1}^{n}\frac{\partial}{\partial x_i}\left(-\frac{1}{v}\frac{\partial v}{\partial x_i}\right) \\ &= \frac{1}{2}\sum_{i=1}^{n}\left(-\frac{1}{v}\frac{\partial^2 v}{\partial x_i^2} + \frac{1}{v^2}\left(\frac{\partial v}{\partial x_i}\right)^2\right) \\ &= -\frac{1}{2v}\Delta v + \frac{1}{2v^2}\left|\frac{\partial v}{\partial x}\right|^2 \\ &= -\frac{1}{v}\frac{\partial v}{\partial t} + \frac{1}{2v^2}\left|\frac{\partial v}{\partial x}\right|^2, \end{aligned}$$

which is the same as $u_t + \frac{1}{2}|\nabla u|^2$. □

As t increases and (3.1) evolves, and the solution is given by $f_t(x) = -\log\left[G_t(x) * e^{-f(x)}\right]$. A larger t has more smoothing effect on $f(x)$, and hence t can now also be viewed as the parameter controlling the amount of smoothing applied on $f(x)$. Theorem 3.1.1 explicitly identified γ and t and connects the local entropy to an evolving PDE. Throughout the paper, γ and t will be used interchangably, with t referring to the time-scale of the evolution of PDE and γ otherwise.

3.1.2 Gradient of Local Entropy

If we wish to apply gradient descent on $f_\gamma(x)$, the update is given as

$$x^{k+1} = x^k - \eta_k \nabla f_\gamma(x),$$

and it would be useful to know $-\nabla f_\gamma(x)$, the negative gradient of $f_\gamma(x)$. There are two ways to express $-\nabla f_\gamma(x)$, both obtained by first differentiating $e^{-f_\gamma(x)} = G_\gamma * e^{-f(x)}$. One approach is as follows:

$$\begin{aligned}
-\nabla f_\gamma(x) e^{-f_\gamma(x)} &= \nabla\left(e^{-f(x)} * G_\gamma\right) \\
&= \left(\nabla e^{-f(x)}\right) * G_\gamma \\
&= -\int \nabla f(x-y) e^{-f(x-y)} G_\gamma(y) dy \\
\implies -\nabla f_\gamma(x) &= \int \left[-\nabla f(x-y)\right] \rho_1^\infty(y;x) dy, \qquad (3.2)
\end{aligned}$$

where $\rho_1^\infty(y;x) \propto \exp\left(-f(x-y) - \frac{\|y\|^2}{2\gamma}\right)$. Alternatively,

$$\begin{aligned}
-\nabla f_\gamma(x) e^{-f_\gamma(x)} &= \nabla\left(G_\gamma * e^{-f(x)}\right) \\
&= (\nabla G_\gamma) * e^{-f(x)} \\
&= -\int \gamma^{-1}(x-y) G_\gamma(x-y) e^{-f(y)} dy \\
\implies -\nabla f_\gamma(x) &= \int \left[-\gamma^{-1}(x-y)\right] \rho_2^\infty(y;x) dy, \qquad (3.3)
\end{aligned}$$

where $\rho_2^\infty(y;x) \propto \exp\left(-f(y) - \frac{\|x-y\|^2}{2\gamma}\right)$. Both expressions (3.2) and (3.3) explicitly take the average over a probability distribution function.

3.1.3 Local Entropy via Stochastic Differential Equations

Equations (3.2) and (3.3) provide the gradient of $\nabla f_\gamma(x)$ in terms of a probability distribution. Now, the difficulty to apply gradient descent on $f_\gamma(x)$ lies in computing the probability distributions $\rho_1^\infty(x)$ and $\rho_2^\infty(x)$.

The Langevin dynamics [24] is proposed to compute the distributions $\rho_1^\infty(x)$ and $\rho_2^\infty(x)$. The Langevin dynamics provide the updates for y, which is then coupled with the updates of x to form a system of stochastic differential equations. The convergence to the probability distributions is exponentially slow for non-convex energies $f(y)+\frac{1}{2\gamma}\|x-y\|^2$. However, when $I+\gamma\nabla^2 f(x)\geq 0$, we have exponential convergence.

With reference to Pavliotis and Stuart [18], we first consider the system of stochastic differential equations (SDEs) given by

$$\begin{aligned} dx(s) &= h(x,y)ds \\ dy(s) &= -\frac{1}{\epsilon}g(x,y)ds + \frac{1}{\sqrt{\epsilon}}dW(s), \end{aligned} \tag{3.4}$$

where h, g are smooth functions and $W(s)\in\mathbb{R}^n$ is the standard Wiener process, i.e. $dW(s)=\sqrt{ds}N(0,1)$. For simplicity, we may assume the heterogenieties in (3.4) to be periodic in space. The parameter $\epsilon \ll 1$ is a microscopic length scale of the problem which corresponds to the period of heterogenieties. Meanwhile, we also define the microscopic length scale L, which determines the size of the domain of s. During the implementation of algorithms, L specifies the number of iterations per mini-batch and ϵ is set to be the reciprocal of L. However, to maintain an overall learning rate of η_k per mini-batch update, we have modified the algorithms by scaling down the learning rate by a factor of L.

To determine the updates for $x(s)$, we use a common tool in stochastic differential equation called "homogenization". Homogenization is used to analyze dynamical systems with multiple time-scales, which have slowly-evolving variables that can be coupled with one another. The theory of homogenization is used to obtain averages of solutions of partial differential equations when the coefficients of the equations are rapidly changing. In the case of (3.4), we are interested in its limit when $\epsilon\to 0$.

Notice that as we take $\epsilon\to 0$, the coefficients of $dy(s)$ in (3.4) varies rapidly. The aim of homogenization is to characterize the highly varying system of SDEs in (3.4) with constant coefficients. The effect of homogenization is similar to the application of smoothing of $x(s)$.

The homogenized vector field for x is defined as

$$\bar{h}(x) = \int h(x,y)\rho^\infty(y;x)dy$$

for some probability distribution function ρ^∞ of the variable y. Under certain conditions, it is shown that the given dynamics in (3.4) can be approximated by

$$d\bar{x}(s) = \bar{h}(x)ds \tag{3.5}$$

with an upper bound on the approximation error given by

$$\mathbb{E}\left[\sup_{0\le s\le T} \|x(s) - \bar{x}(s)\|^p\right] \le C\epsilon^{\frac{p}{2}}$$

for some constant C and all $p > 1$, where T is the time of the SDE.

Recall that we want to apply gradient descent on $f_\gamma(x)$. Using (3.2) and (3.3), we have $\bar{h}(x) = -\nabla f_\gamma(x)$ by choosing $h(x, y) = -\nabla f(x - y)$, $\rho^\infty = \rho_1^\infty$ and $h(x, y) = -\gamma^{-1}(x - y)$, $\rho^\infty = \rho_2^\infty$ respectively.

First, we use the setting in (3.2) and consider (3.5). This yields the approximation

$$d\bar{x}(s) = \bar{h}(x)ds = -\nabla f_\gamma(x)ds, \tag{3.6}$$

which suggests that we should put $h(x, y) = -\nabla f(x - y)$ in (3.4). In addition, by the Langevin dynamics, we can compute the distribution $\rho_1^\infty(y; x)$ and yield the update

$$y(s) = -\frac{1}{\epsilon}\left[\frac{y}{\gamma} - \nabla f(x - y)\right]ds + \frac{1}{\sqrt{\epsilon}}dW(s). \tag{3.7}$$

We can hence combine (3.6) and (3.7) and consider the following system of SDEs as a model for the optimization problem (2.4):

$$\begin{aligned} dx(s) &= -\nabla f(x - y)ds \\ dy(s) &= -\frac{1}{\epsilon}\left[\frac{y}{\gamma} - \nabla f(x - y)\right]ds + \frac{1}{\sqrt{\epsilon}}dW(s). \end{aligned} \tag{3.8}$$

By an Euler discretization, we obtain the ESGD algorithm for the k-th mini-batch:

$$\begin{aligned} x_k^{j+1} &= x^j - \frac{\eta_k}{L}\nabla f_B(x_k^j - y^j) \\ y^{j+1} &= y^j - \frac{\eta_k}{L}\left[\frac{1}{\epsilon}\left(\frac{y^j}{\gamma} - \nabla f_B(x - y)\right)\right] + \sqrt{\frac{\eta_k}{L\epsilon}}N(0, 1) \\ &= \left(1 - \frac{\eta_k}{L\epsilon\gamma}\right)y^j + \frac{\eta_k}{L\epsilon}\nabla f_B(x - y) - \sqrt{\frac{\eta_k}{L\epsilon}}N(0, 1), \end{aligned} \tag{ESGD}$$

with $x_k^0 = x^k, y^0 = 0, j = 0, 1, \ldots, L-1$, and we set $x^{k+1} = x_k^L$. $L = 20$ is chosen by default.

Similarly, we can use the setting in (3.3) and consider (3.5). In this case, the system of SDEs becomes

$$
\begin{aligned}
dx(s) &= -\gamma^{-1}(x-y)ds \\
dy(s) &= -\frac{1}{\epsilon}\left[\nabla f(y) + \frac{1}{\gamma}(y-x)\right] ds + \frac{1}{\sqrt{\epsilon}} dW(s).
\end{aligned}
\tag{3.9}
$$

The system (3.9) converges to the homogenized dynamics given by $d\bar{x}(s) = -\nabla f_\gamma(x)ds$. In addition, $\nabla f_\gamma = \gamma^{-1}(\bar{x} - \langle y \rangle)$ where

$$
\langle y \rangle = \int y \rho^\infty(y; \bar{x}) dy = \lim_{T\to\infty} \int_0^T y(s) ds \tag{3.10}
$$

and $y(s)$ is the solution of (3.9) when x is fixed. Putting $L = \epsilon^{-1}$ in (3.10) corresponds to $T = 1$; to investigate the effect of $T \to \infty$ in (3.10), we will also explore the possibilities of setting $T = L\epsilon$, the time of the PDE evolution, to a value other than 1.

By performing an Euler discretization on (3.9), we obtain the update rule

$$
\begin{aligned}
x^{k+1} &= \begin{cases} x^k - \eta_k \gamma^{-1}(x^k - y^k) & \text{if } (\epsilon k) \text{ is an integer} \\ x^k & \text{otherwise} \end{cases} \\
y^{k+1} &= y^k - \eta_k \left[\nabla f_B(y^k) + \gamma^{-1}\left(y^k - x^k\right)\right].
\end{aligned}
\tag{3.11}
$$

The parameter L should be set to 20 for H-ESGD. We can view (3.11) as performing L updates on y per update on x, and rewrite it as

$$
\begin{aligned}
y^{j+1} &= y^j - \frac{\eta_k}{L}\left[\nabla f_B(y^j) + \gamma^{-1}\left(y^j - x^k\right)\right], \quad j = 0, 1, \ldots, L-1 \\
x^{k+1} &= x^k - \eta_k \gamma^{-1}(x^k - y^L)
\end{aligned}
\tag{H-ESGD}
$$

with $y^0 = x^k$.

3.2 Smoothing of the Loss Function by the Heat Equation

Compared to the smoothing performed by the local entropy, smoothing performed by heat equation is more commonly seen in the deep learning literature [8][15].

The dynamics for the heat equation is given by:

$$\begin{aligned} dx(s) &= -\nabla f(x-y)ds \\ dy(s) &= -\frac{1}{\epsilon\gamma} y ds + \frac{1}{\sqrt{\epsilon}} dW(s). \end{aligned} \tag{3.12}$$

The HEAT algorithm is obtained by performing a similar derivation that yields ESGD, but using the heat equation instead of the viscous Hamilton-Jacobi equation. The solution to the heat equation yields the update rule for the k-th mini-batch

$$\begin{aligned} x_k^{j+1} &= x^j - \frac{\eta_k}{L} \nabla f_B(x_k^j - y^j) \\ y^{j+1} &= y^j - \frac{\eta_k}{L\epsilon\gamma} y^j - \sqrt{\frac{\eta_k}{L\epsilon}} N(0,1) \\ &= \left(1 - \frac{\eta_k}{L\epsilon\gamma}\right) y^j - \sqrt{\frac{\eta_k}{L\epsilon}} N(0,1), \end{aligned} \tag{HEAT}$$

with $x_k^0 = x^k, y^0 = 0, j = 0, 1, \dots, L-1$, and we set $x^{k+1} = x_k^L$. The parameter $L = 20$ is chosen by default.

Notice that the y update rule in (HEAT) is different from (ESGD) by one term, which is the difference between the smoothing performed by local entropy and the smoothing performed by heat equation. In fact, the heat equation itself also differs from the viscous Hamilton-Jacobi equation by one term.

3.3 Smoothing of the Loss Function by Inf-convolution

The use of $f_\gamma(x)$ in the optimization has benefits that are independent of the coefficient of the viscosity in (3.1). In contrast to the viscous Hamilton-Jacobi equation, the non-viscous Hamilton-Jacobi equation not only has a simpler expression, but also preserves the local minimum of $f(x)$ for small t, and widens the convex regions of $f(x)$.

3.3.1 The Non-viscous Hamilton-Jacobi Equation

The non-viscous Hamilton-Jacobi equation is given as

$$\begin{aligned} \frac{\partial u}{\partial t} + \frac{1}{2}\left|\frac{\partial u}{\partial x}\right|^2 &= 0 \\ u(x,0) &= f(x), \end{aligned} \tag{3.13}$$

which is the same as (3.1) without the viscosity term. By the Hopf-Lax formula [6], the solution of (3.13) at time t is given as the inf-convolution or Moreau envelope function [16][25] of $f(x)$ and $\frac{1}{2t}\|x\|^2$, i.e.

$$u(x,t) = \inf_y \left\{ f(y) + \frac{1}{2t}\|x-y\|^2 \right\}. \tag{HL}$$

Recall that t is identified with γ in Theorem 3.1.1. The same applies here as the PDE given in (3.13) has more smoothing effect as t increases. However, the evolution of PDE is more interesting here.

The first observation is that the convex regions of $u(x,t)$ are quickly widened and the concave regions are shrunk as t increases. This can be shown using the fact that solution $u(x,t)$ of (3.13) is semi-concave. A function f is said to be semi-concave with a constant C if $f(x) - \frac{C}{2}\|x\|^2$ is concave.

To show that $u(x,t)$ is semi-concave, we first define $g(x;y) = f(y) + \frac{1}{2t}\|x-y\|^2$ to be a function of x parametrized by y. Observe that each $g(x;y)$ is semi-concave with $C = \frac{1}{t}$, since

$$\begin{aligned} g(x;y) - \frac{C}{2}\|x\|^2 &= f(y) + \frac{1}{2t}\|x-y\|^2 - \frac{C}{2}\|x\|^2 \\ \frac{d}{dx}g(x;y) &= \frac{1}{2t}(x-y) - \frac{C}{2}x \\ \frac{d^2}{dx^2}g(x;y) &= \frac{1}{2t} - \frac{C}{2} \end{aligned}$$

is concave when $C \geq \frac{1}{t}$. Subsequently, $u(x,t) = \inf_y g(x;y)$ is also semi-concave with $C = \frac{1}{t}$ as well. The proof of the widening of convex regions of $u(x,t)$ can be found in Section 5 of [3].

The second observation is the neatly computed gradient of $u(x,t)$ with respect to x. The gradient can be computed as

$$\nabla_x u(x,t) = \frac{x-y^*}{t} = \nabla f(y^*),$$

where y^* is the optimizer of (HL). Moreover, suppose $I + t\nabla^2 f(x) \geq \lambda I$ for some $\lambda > 0$. Then the gradient $p^* := \nabla_x u(x,t)$ is the unique steady solution to

$$\frac{dp(s)}{ds} = -t\left(p(s) - \nabla f(x - tp(s))\right). \tag{3.14}$$

Using Euler's method, we can discretize (3.14) as

$$p^{j+1} = (1 - t\delta_s)p^j + t\delta_s \nabla f(x - tp^j). \tag{3.15}$$

The stability condition for (3.15) is $|1 - t\delta_s| \leq 1$, i.e. $0 \leq \delta_s \leq t^{-1}$, setting $\delta_s = t^{-1}$ yields the iterative scheme

$$p^{j+1} = \nabla f(x - tp^j), \quad \text{with } p^0 = 0 \tag{3.16}$$

where j is the index of the iteration. Similar to the convergence to the gradient of $f_\gamma(x)$ mentioned in Section 3.1.3, the iteration is exponentially convergent if $I + t\nabla^2 f(x) > 0$.

By (3.15), after we identify t back to γ, the HJ algorithm can be written as

$$\begin{aligned} p^{j+1} &= \nabla f_B(x^k - \gamma p^j), \quad j = 0, 1, \ldots, L-1, \text{with } p^0 = 0 \\ x^{k+1} &= x^k - \eta_k p^L \end{aligned} \tag{HJ}$$

Typically, we perform $L = 5$ time steps of p to estimate p^*. Notice that setting $L = 1$ recovers the usual stochastic gradient descent method.

Finally, unlike the HEAT and ESGD method, smoothing performed by the inf-convolution only requires one update of x per mini-batch. The computational cost of HJ is therefore lower than HEAT and ESGD. This allows the HJ algorithm to work effectively in the setting of deep neural networks, where the parameter x often has high dimensions.

Chapter 4

Empirical Evaluation

In this section, we discuss the experimental results on deep neural networks using the aforementioned PDE methods (HJ, HEAT, ESGD and H-ESGD) and compare them to the stochastic gradient descent method. Chaudhari et al. [3] showed that the new proposed methods led to improved classification on image datasets and convolutional neural networks. The specific datasets used in their paper are the MNIST dataset [13] and CIFAR-10 dataset [11].

To provide a more comprehensive view, we demonstrate the methods on four datasets of different sizes and neural network architectures, and compare the performance of the PDE methods on different datasets and hyperparameters[1].

The datasets and networks are described in Section 4.1 and the experimental results can be found in Section 4.2.

4.1 Setup

4.1.1 Notations

We use input_d to indicate an input layer with d dimensions. We use $\text{fc}_{d,h}$ to denote a fully connected layer with d output dimensions, followed by the activation function h, if

[1]Hyperparameters refer to the parameters that are manually tuned in a neural network before training, such as the learning rate and mini-batch size. The usage of the word "hyperparameter" here is simply to distinguish them from the parameter x that we would like the machine to learn by itself.

applicable. In our networks, we consider rectified linear units and the softmax function as the activation functions, and we will denote them as "ReLU" and "softmax" respectively. If batch normalization is applied after h, we will use "BN" to indicate the process.

For instance, the neural network

$$\text{input}_{112} \to \text{fc}_{100,\,\text{ReLU},\,\text{BN}} \to \text{fc}_{2,\,\text{softmax}}$$

has an input layer with 112 nodes. The hidden layer has 100 nodes, which has the rectified linear unit as the activation function, and is followed by batch normalization. The output layer has 2 nodes, with the softmax function as the activation function.

4.1.2 Datasets

It was previously mentioned that we typically divide the dataset into training set, validation set and testing set. However, Chaudhari et al. [3] suggested that it is customary in the deep learning literature to not use a separate testing set for the datasets considered. Hence, instead of a separate testing set, we follow their practice and report test error on the validation set itself, and the validation error is reported directly.

For each dataset, the default initial learning rate η_0 and smoothing parameter γ are found manually. We decay the learning rate using (2.9) as the number of iterations k increases, and keep γ constant throughout the learning process.

The four datasets and their corresponding neural networks, learning rates and γ are specified as follows:

1. **Mushroom** (UCI Machine Learning Repository)

   The Mushroom dataset is provided by the UCI Machine Learning Repository [14] with 8,124 instances of mushrooms' information including color, odor, habitat and other characteristics, summing up to 23 categorical features. The dataset is split into 7,311 for training and 813 for validation. The aim is to determine whether each mushroom is edible or poisonous. This is a binary classification problem.

   The "stalk root" feature is excluded because it has missing attributes. After performing one-hot encoding, each input is expressed as a binary vector of dimension 112.

   Since the dataset is relatively small and easy to handle, we use a simple neural network structure with one hidden layer of 100 units. The fully-connected network on the Mushroom dataset is defined as

$$\text{mushroom-fc}: \text{input}_{112} \rightarrow \text{fc}_{100,\,\text{ReLU},\,\text{BN}} \rightarrow \text{fc}_{2,\,\text{softmax}}$$

The default setting is $\eta_0 = 0.1$ and $\gamma = 0.05$.

2. **Covertype** (UCI Machine Learning Repository)

   The Covertype dataset contains 581,012 samples for predicting forest cover types from four wilderness areas found in the Roosevelt National Forest of northern Colorado. There are 54 features in total, including 44 binary variables specifying wilderness areas and soil type, and 10 quantitative variables specifying elevation, aspect, slope and other measurements. The quantitative variables are normalized to zero mean and unit variance. The aim is to classify the inputs into one of the 7 forest cover types: Spruce/Fir, Lodgepole Pine, Ponderosa Pine, Cottonwood/Willow, Aspen, Douglas-fir, and Krummholz. This is a multi-class classification problem.

   This dataset is reported to have 70% accuracy using one 120-unit hidden layer of neural networks [2]. We use the same setting here:

$$\text{coverytype-fc}: \text{input}_{54} \rightarrow \text{fc}_{120,\,\text{ReLU},\,\text{BN}} \rightarrow \text{fc}_{7}$$

   The train/validation/test split equal to 11,340/3,780/565,892. Since we do not intend to use a separate test set, the given test set is combined into the validation set.

   The default setting is $\eta_0 = 0.05$, $\gamma = 0.02$.

3. **House Sales** (Kaggle datasets)

   The House Sales dataset[2] contains 21,613 instances of house sale prices for King County, USA between May 2014 and May 2015. The dataset is split into 6,483 for training and 15,130 for validation. Each house contains 19 features, and the aim is to predict the price of the house. This is a regression problem with one output value.

   In our experiments, the "zip code" feature is excluded and the quantitative features are normalized to zero mean and unit variance. Integral features, such as the number of bathrooms, bedrooms and floors, are treated as categorical and one-hot encoding is applied. The following neural network architecture is applied:

$$\text{house-fc}: \text{input}_{54} \rightarrow \text{fc}_{15,\,\text{ReLU},\,\text{BN}} \rightarrow \text{fc}_{8,\,\text{ReLU},\,\text{BN}} \rightarrow \text{fc}_{4,\,\text{ReLU},\,\text{BN}} \rightarrow \text{fc}_{1}$$

   The default setting is $\eta_0 = 0.0001$, $\gamma = 0.0005$.

[2]The House Sales dataset can be found on this website: https://www.kaggle.com/harlfoxem/housesalesprediction

4. **Dow Jones Index** (UCI Machine Learning Repository)

   The Dow Jones Index dataset contains weekly data for the Dow Jones Industrial Index of 30 stocks in the first six months of 2011. The 360 instances in the first quarter (January to March) are used for training and the 390 instances in the second quarter (April to June) are used for validation. The dataset contains attributes including the name of the stock, the last business day of the work, opening price, closing price, highest price and lowest price of the stock during the week, and the percentage change of volume and price over the last week. The goal is to predict information about the opening price, closing price and the percentage of change and return in the following week, and the number of days till the next dividend.

   The dataset is pre-processed by converting the date of each instance to the ordinal day of the year[3], and then normalizing to zero mean and unit variance. After applying one-hot encoding to the names of the stock, the total number of features is 39 and the output dimension is 5. Hence, this is a regression problem with 5 output values.

   The neural network architecture is given by:

$$\text{dowjones-fc}: \ \text{input}_{39} \to \text{fc}_{15,\,\text{ReLU},\,\text{BN}} \to \text{fc}_{10,\,\text{ReLU},\,\text{BN}} \to \text{fc}_{5}$$

   The default setting is $\eta_0 = 0.001$, $\gamma = 0.002$.

For classification problems (Mushroom and Covertype), we train the network using the cross-entropy loss and report the accuracy; for regression problems (House Sales and Dow Jones Index), we train the network using the mean-square loss and report the loss value.

4.2 Experiments

We run the experiments with the Nesterov Momentum variation in (2.11) for the update of x in SGD, HJ, HEAT, ESGD and H-ESGD. By default, we set $L = 1$ for SGD, $L = 5$ for HJ, and $L = 20$ for HEAT, ESGD, H-ESGD. The default mini-batch size is set to be 128, and the default momentum is set to be 0.9. All experiments are performed using the Tensorflow library in Python using a 128GB RAM on a Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60GHz.

In the upcoming experiments, we explore the effect of various hyperparameters including L, T (i.e. L times ϵ), γ, momentum and mini-batch size. We are mainly interested

[3]For example, February 5, 2011 translates to Day 36 of Year 2011.

in the performance (accuracies and losses) of different algorithms versus the number of epochs (the size of the training set divided by the size of the mini-batch), which provide a direct comparison of the numerical convergence rate to a local minimum. However, since some algorithms take a longer time to finish an iteration than the others, the wall clock time measurement to converge to a certain level of accuracy or loss is often more relevant to practical usage. Hence, the accuracies and losses versus CPU time are also studied whenever necessary.

For each experiment, we report and plot the mean across 10 random seeds on the validation set. Each random seed corresponds to a set of initialized parameters randomly picked from a uniform distribution of zero mean and 0.1 variance. We plot the curves of each of the algorithms for the mean values against number of epochs; if applicable, the results against CPU time are also displayed. Then, we shade the regions around the corresponding colored curves to indicate the region within 1 standard deviation from the mean. The higher the standard deviation, the more likely that the performance of an algorithm depends on the initialization of parameters.

Starting from the first epoch, all of the curves are plotted at an interval of 1 epoch. The case at epoch 0 is omitted and corresponds to the performance of the initialized parameters without any tuning. We run the Covertype dataset for 10 epochs and the other datasets for 20 epochs. The range of the y-axis is chosen to best highlight the loss or accuracy difference. As a result, some of the reported losses or accuracies for the first few epochs could be truncated. For a curve plotted against CPU time, the additional time period before the curve starts indicate the amount of time required to build the neural network framework and perform the first epoch of parameter tuning.

4.2.1 Loss and Accuracy versus the Number of Epochs

In the first experiment, we provide the direct comparison of each algorithm versus number of epochs when applied to each deep neural network. The performance of each algorithm against the number of epochs provides information on the rate of convergence that is independent of hardware installations. The results are shown in Figure 4.1.

In all plots, SGD and HJ have indistinguishable behavior and their curves almost completely overlap with each other. This is possibly because the values of γ make little difference between the original loss function f and the inf-convolution of f, which SGD and HJ aim to minimize respectively. However, the same γ values applied to ESGD and H-ESGD have a noticeable difference in their learning curves when compared to SGD, implying that the same value of γ has more smoothing effect in local entropy than in inf-convolution.

In most cases, HEAT and ESGD also have very similar behaviors, because their update rules differ by only one term. In the Mushroom dataset, the curves of HEAT and ESGD coincide each other; in the Covertype and Dow Jones Index datasets, ESGD outperforms HEAT slightly. The exception is the House Sales dataset, in which HEAT outperforms all of the other algorithms. A possible explanation is that smoothing performed by HEAT manages to reduce the noise of the original loss function and cause the update of x to escape from a local minimum, whereas the same choice of γ leads to little difference with smoothing by local entropy. From this observation, we can deduce that the same value of γ has more smoothing effect in heat equation than in local entropy. Meanwhile, we also observe that H-ESGD has achieved a better solution than ESGD on average.

In general, the algorithms have very different relative performance on each dataset. For instance, ESGD converges slower than SGD in Mushroom and Covertype datasets but much faster in Dow Jones Index. H-ESGD, on the other hand, behaves relatively well in House Sales but poor on the other datasets. SGD and HJ tend to have smaller variance than the other three algorithms, suggesting that they may be more stable and have little dependence on the initialization of parameters.

Up till now, all of the newly introduced algorithms except HJ have been shown to outperform SGD on at least one dataset. An appropriate amount and method of smoothing can indeed lead to more efficient optimization in training deep neural networks.

4.2.2 Loss and Accuracy versus CPU Time

Sometimes, the size of the dataset could be huge and it is time-consuming to train a deep neural network. However, in situations such as stock prices and weather prediction, time is an important factor. In these scenarios, we are often more interested in algorithms that can converge to a solution with decent loss or accuracy in a timely manner, rather than algorithms that eventually converge to a much better solution after a prolonged period of time. This experiment studies the performance of each algorithm in terms of CPU time. The results allows us to compare and contrast the algorithms for their efficiency during actual implementation.

Figure 4.2 shows the performance of each of the algorithms in terms of CPU seconds. Recall that L is the number of updates per mini-batch. The amount of overall CPU time for each algorithm is hence directly proportional to both L and the number of epochs. In this case, as compared to HEAT, ESGD and H-ESGD which all have default values $L = 20$,

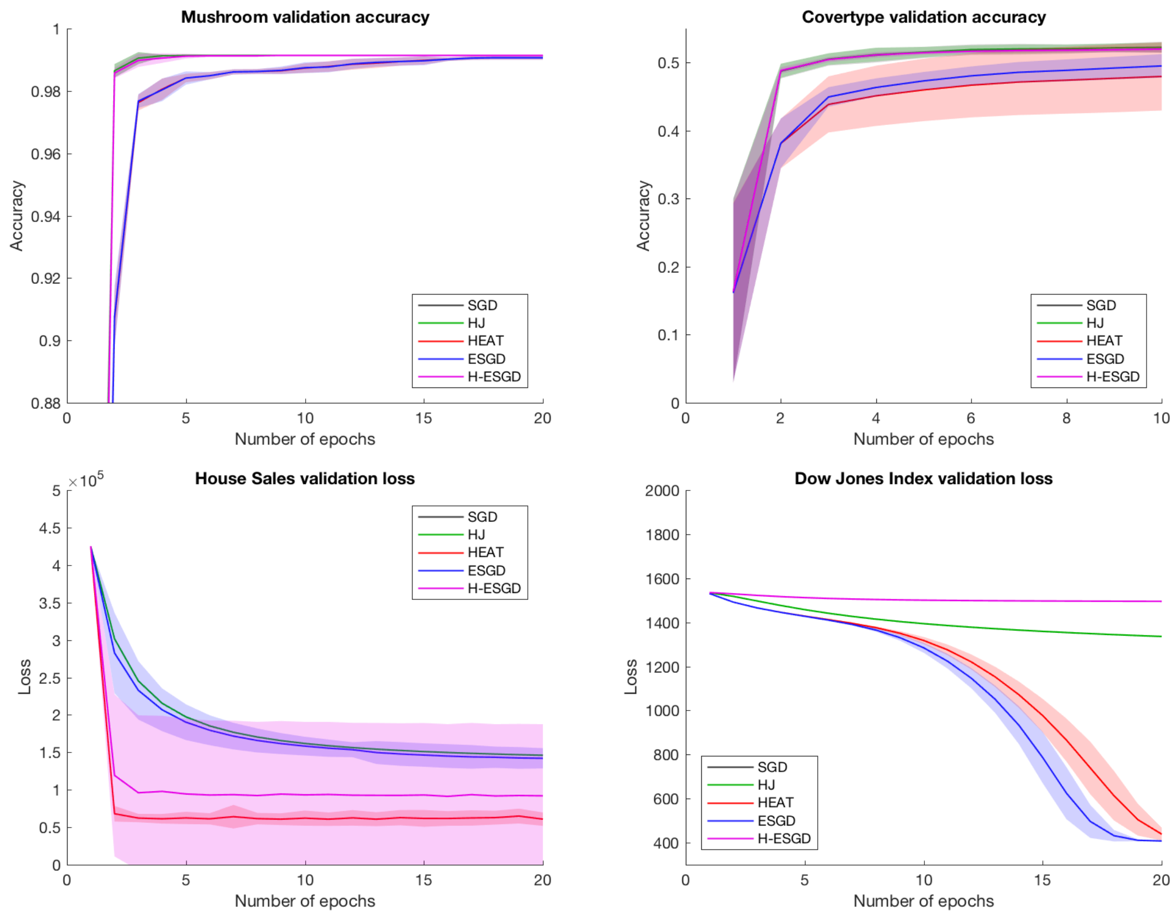


Figure 4.1: Comparison of accuracy or loss against number of epochs for the optimization methods SGD (black), HJ (green), HEAT (red), ESGD (blue) and H-ESGD (purple) on four datasets.

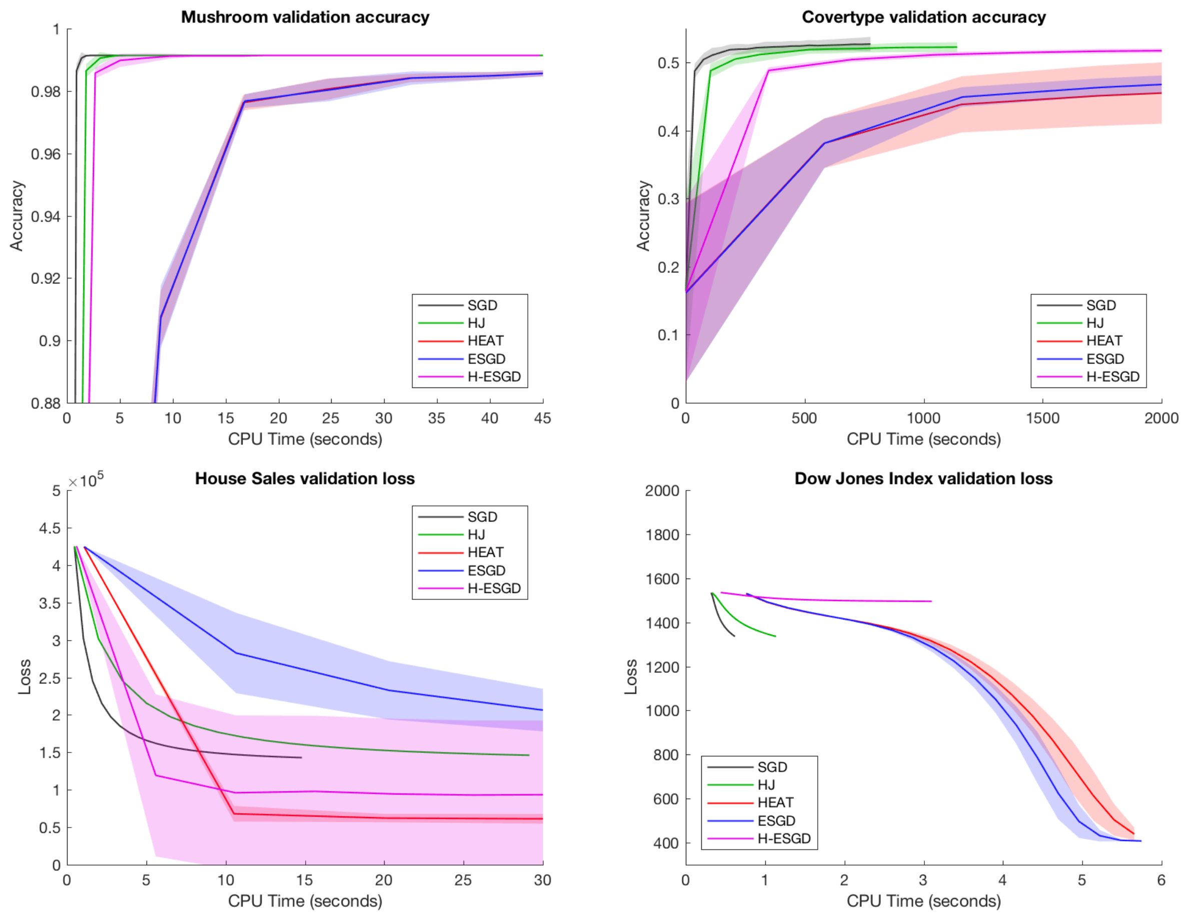


Figure 4.2: Comparison of accuracy or loss against CPU time for the optimization methods SGD (black), HJ (green), HEAT (red), ESGD (blue) and H-ESGD (purple) on four datasets.

SGD which has a default value of $L = 1$ requires a significantly less amount of time to finish one iteration, and could be a more practical choice of algorithm. In the case of the Mushroom and Covertype datasets, SGD indeed reaches a better accuracy than the other algorithms in a shorter amount of time. As for House Sales and Dow Jones Index, SGD also tends to reach a better loss in the first few seconds. This is quickly followed by HJ with $L = 5$.

Since L is the number of iterations per mini-batch, L is directly proportional to the wall clock time required to finish one mini-batch update. Hence, a smaller L likely gives a more efficient convergence. This motivates us to perform the next experiment, in which we fix the value of ϵ and investigate the results with different L.

4.2.3 Varying L and T

The previous experiment suggests that a smaller L or $T = L\epsilon$ could be beneficial if we want faster convergence to a satisfactory accuracy or loss value in terms of wall clock time. We perform the experiment with $\epsilon = 0.05$ and $L = 20, 40, 100$; this corresponds to $T = 1, 2, 5$.

Figure 4.3 shows the results on the Mushroom dataset. Different values of L give similar results to HJ and H-ESGD, where all of the three curves coincide and are indistinguishable from the plots. A larger L directly increases the number of iterations in HEAT and ESGD and gives an improved performance. A similar trend is observed in the Covertype (Figure 4.4) and Dow Jones Index (Figure 4.6).

However, for the ESGD algorithm in the loss for House Sales (Figure 4.5) suggest that $L = 100$ gives a worse performance than $L = 20$ or 40. For HJ, the results are similar, with the curves $L = 20$ and $L = 40$ coincide and both achieve a better loss than $L = 100$. One possible explanation is that the the neural network of House Sales has large gradients. When L is large, there could be round-off errors in the calculation of gradients that accumulate and make the update directions inaccurate during implementation.

Another reason to explain the behavior of HJ is that the choice of γ is too large. Recall that L determines the number of iterations to estimate the fixed-point solution p^*, which approximates the gradient of $u(x, t)$, the smoothened loss function with inf-convolution. According to (3.14), a larger L should converge to a more accurate solution to the ODE; however, the loss in the first few epochs of HJ is higher when L is larger. Recall that the exponential convergence is guaranteed only if $I + \gamma\nabla^2 f(x) > 0$, which suggests that the value of γ may be too large and the convergence condition is not guaranteed. Finally, a similar trend is observed for both HEAT and L-ESGD for all choices of L.

To conclude, a larger L gives a better performance in most cases, but can have a worse performance when γ or the gradients of the network is large. In addition, since a larger L also directly increases the run time of each algorithm, a smaller value of L could converge to a better solution in a shorter period of time. Hence, a small value of L, for instance $L = 20$, is still recommended.

4.2.4 Varying L with T fixed

In this experiment, we are interested in the effect of L on HJ, HEAT, ESGD and H-ESGD while keeping $T = L\epsilon$ fixed at 1. We perform the experiments with $L = 1, 5, 20, 100$ and produce the plots for each dataset. Note that HJ solely depends on L and does not depend on ϵ, and the case $L = 1$ in HJ is identical to the stochastic gradient descent. In addition, similar to the previous experiment, a larger L require a proportionally higher amount of run time.

An overview of the results shows that the effect of L highly depends on the choice of algorithms, datasets and neural networks. Unlike the previous experiment which suggests that a larger L gives a better performance in general, the results in this experiment do not have a distinct trend or pattern. For instance, in the Mushroom dataset (Figure 4.7), a smaller L gives a faster convergence in terms of the number of epochs; however, a larger L is preferred for Dow Jones Index (Figure 4.10). For the Covertype and House Sales dataset (Figures 4.8 and 4.9), ESGD performs better when L is small, while H-ESGD performs better when L is large. It is therefore easier to analyze the results by looking at each algorithm individually.

For the HJ algorithm, we observe that HJ has the same performance for different values of L. The curves for each HJ plot overlapped each other, except for the House Sales dataset which has a slightly poorer loss for $L = 100$. This result matches the previous experiment. Since HJ is independent from ϵ, this experiment is essentially the same as the one described in the previous section, where we set L to values 20, 40 and 100 and fix ϵ. As we see that the convergence in terms of number of epochs is similar when we set $L = 1, 5, 20, 100$ for HJ, it is recommended that L is set to 1 for a shorter run time. In other words, SGD is preferred over HJ.

For the H-ESGD algorithm, the results in Mushroom and Dow Jones Index show little difference when different values of L are used. The curves formed by different L values overlap each other. Both the results of Covertype and House Sales show that $L = 5, 20, 100$ have similar performance, and the performance with $L = 1$ is poorer. To explain this behavior, recall that ϵ is the period of heterogenieties in (3.4). When L is large, $\epsilon = L^{-1}$

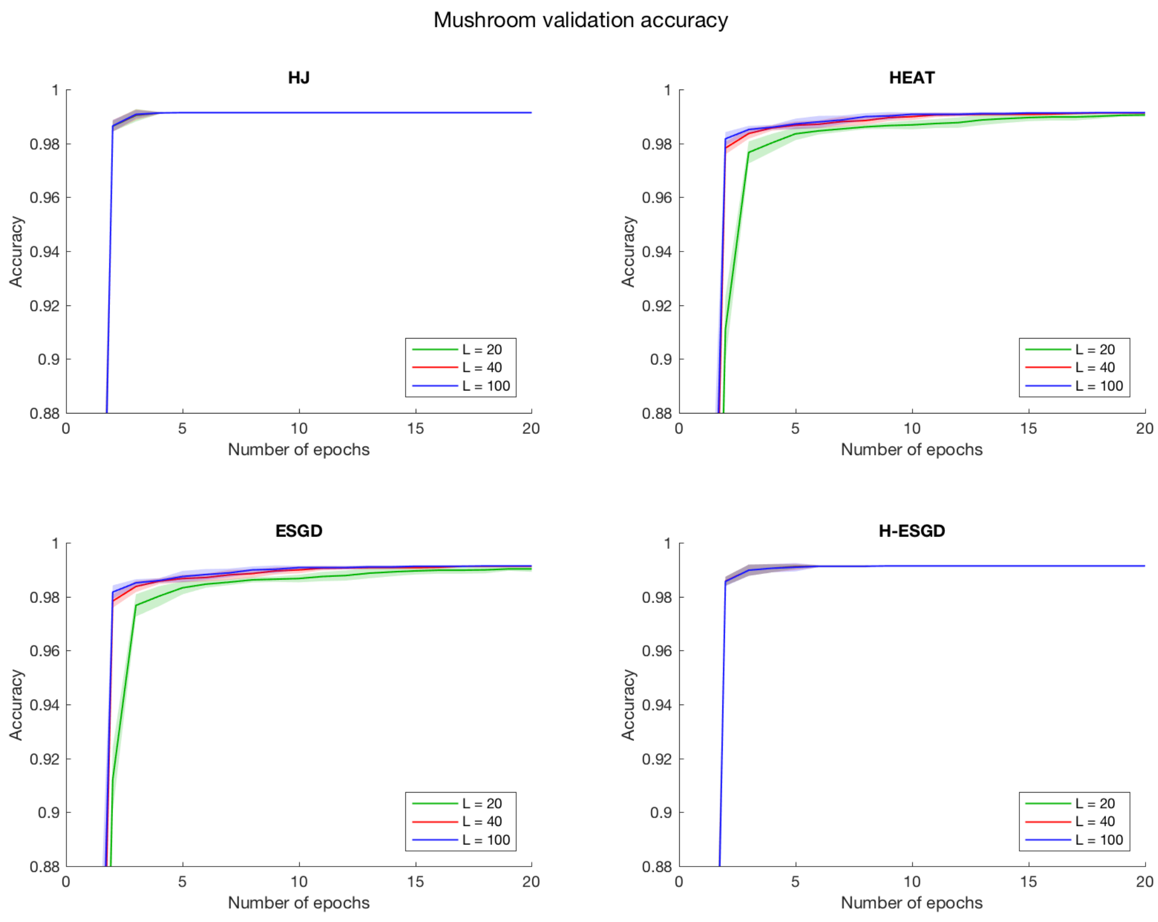


Figure 4.3: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Mushroom dataset with L set to be 20 (green), 40 (red) and 100 (blue).

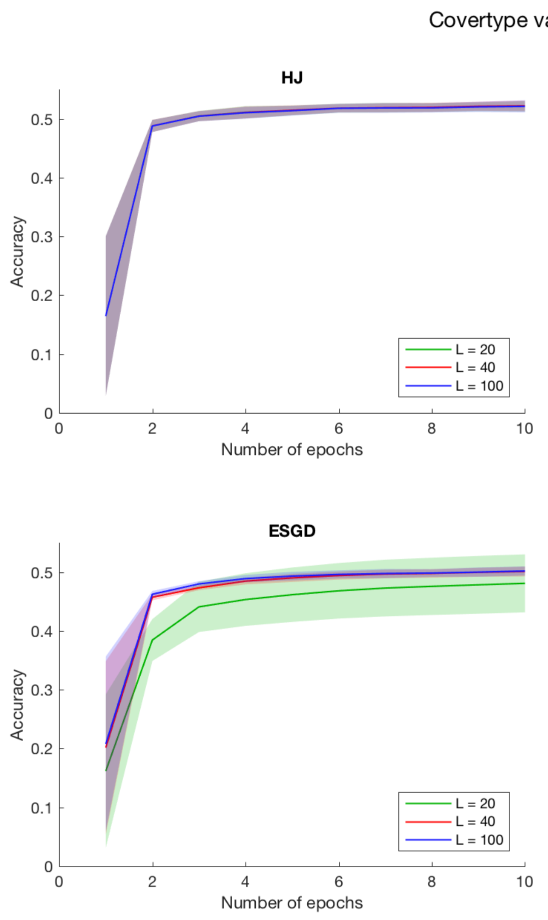


Figure 4.4: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Covertype dataset with L set to be 20 (green), 40 (red) and 100 (blue).

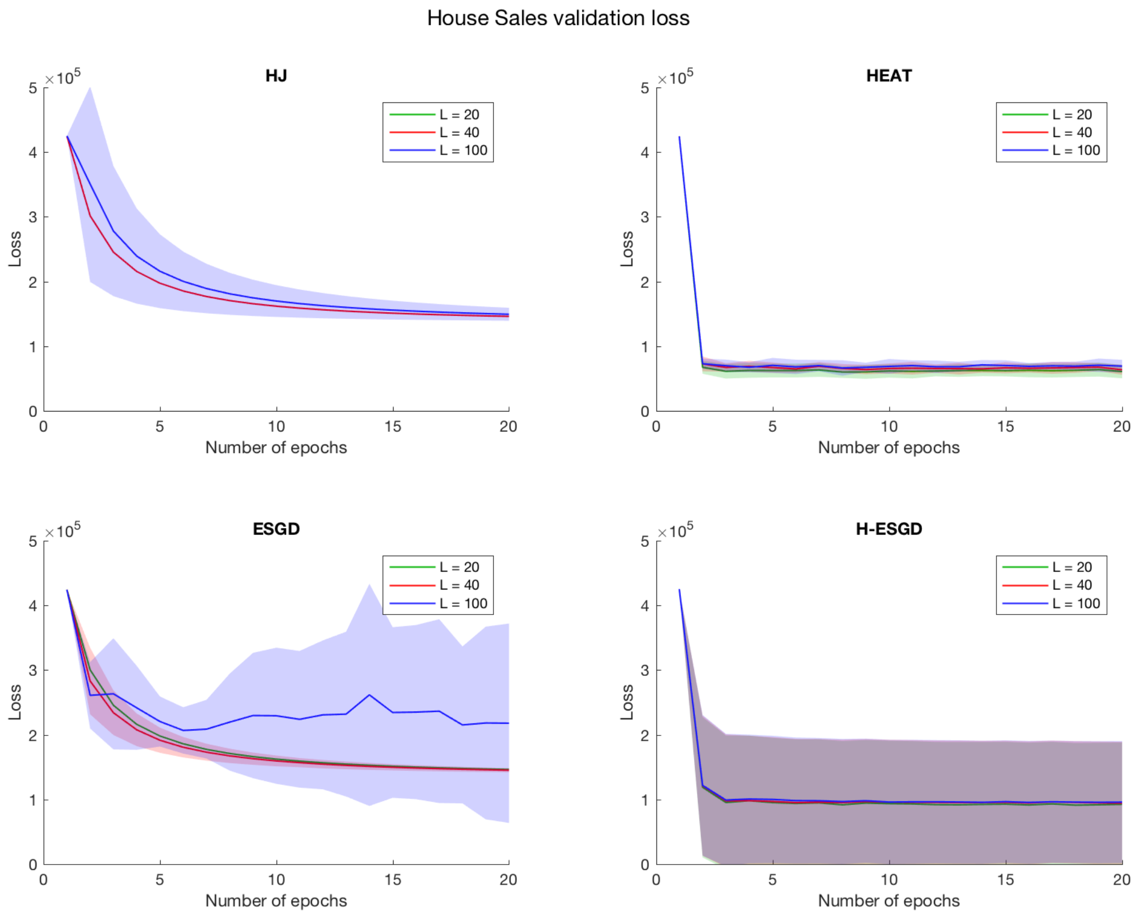


Figure 4.5: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the House Sales dataset with L set to be 20 (green), 40 (red) and 100 (blue).

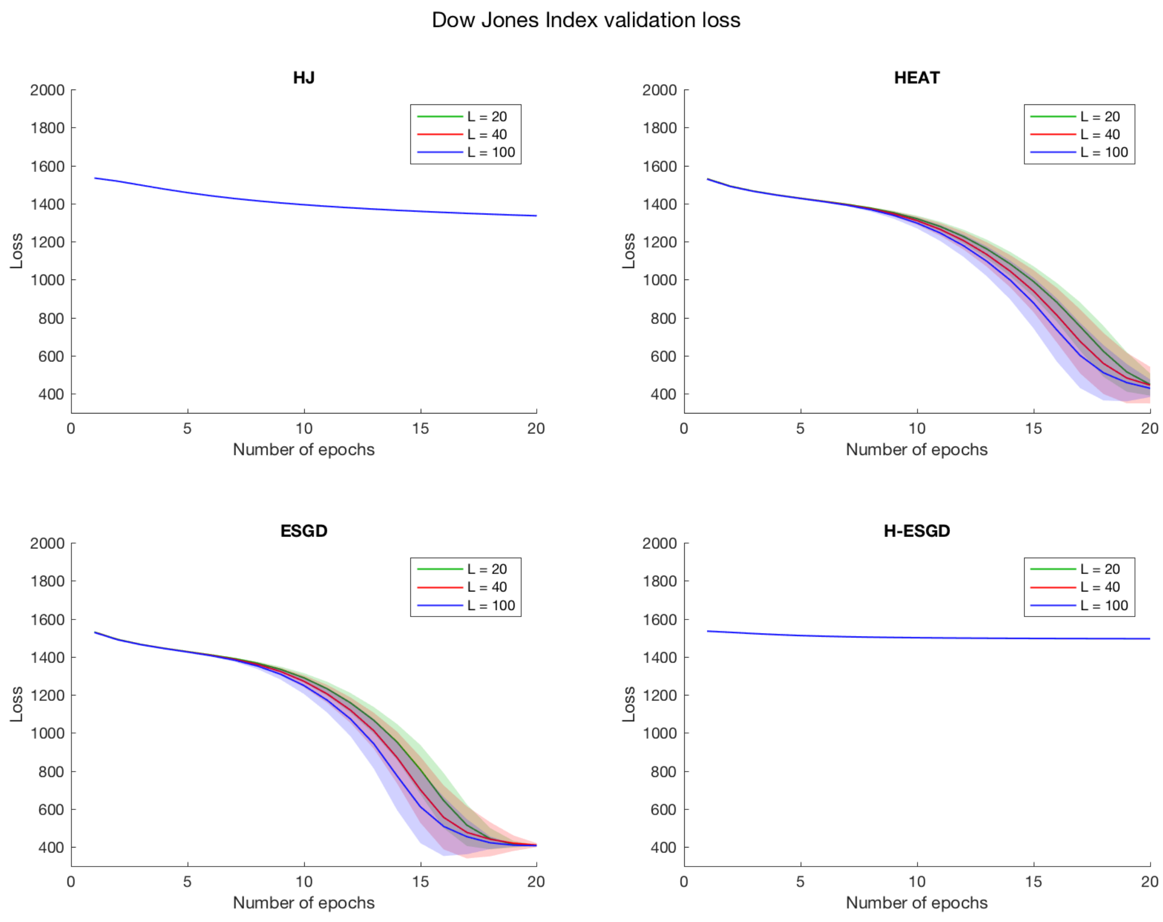


Figure 4.6: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Dow Jones Index dataset with L set to be 20 (green), 40 (red) and 100 (blue).

decreases, and the equation (3.9) converges to the homogenized dynamics. Taking the run time into account, it is recommended that we choose $L = 5$ for H-ESGD.

The cases for HEAT and ESGD are more complicated. For HEAT, a smaller L gives a better accuracy in Mushroom and Covertype, while a larger L gives a better loss in House Sales and Dow Jones Index. For ESGD, a larger L gives a better performance in Dow Jones Index, while a smaller L is preferred for the other three datasets. To explain this, recall that both HEAT and ESGD have a stochastic term present in their update rules given by (HEAT) and (ESGD). A larger L has the advantage of allowing higher number of iterations per mini-batch allows more gradient steps to improve the loss with a smaller learning rate. The small learning rate makes learning proceed slower, but also does not "overshoot" the local minimum. A larger L also implies having to perform more updates using the stochastic term and gives a higher uncertainty. The parameters update could take a step in a wrong direction, causing the loss to grow instead, and the accuracy to drop. Hence, setting $L = 1$ or 100 on HEAT and ESGD could be beneficial or harmful to the learning process. For a higher efficiency, it is recommended that L is set to 5.

Overall, L has little effect on HJ, and the influence of L on the other algorithms highly depend on the datasets and neural networks. A value of $L = 1$ or 100 is extreme and can potentially give improved results, while the result of $L = 5$ or 20 have more stable results. For HEAT, ESGD and H-ESGD, it is recommended that L is set to 5. For HJ, it is recommended that L is set to 1 to recover the SGD method.

4.2.5 Varying Amount of Smoothing

As observed in Section 4.2.1, the same smoothing parameter γ can have different degree or results of smoothing when referred to different smoothing methods. The results in Section 4.2.1 suggested that the same value of γ on the same dataset has the most smoothing effect in heat equation, and the least smoothing effect in inf-convolution.

In this section, we further explore the performance of the algorithms when using different choices of γ. For each dataset, we manually set a default value for γ and compare with the results obtained when γ is multiplied by 2 or 5.

We have seen that SGD performs the best on the Mushroom dataset in Section 4.2.1. As shown in Figure 4.11, HJ has a similar learning trend when different γ values are applied, and the learning curves coincide. For HEAT, ESGD and H-ESGD, a smaller γ improves the results for the Mushroom dataset, which suggests that the Mushroom dataset has an underlying function which requires little to no smoothing. When too much smoothing is applied, for instance in H-ESGD with $\gamma = 0.25$, the decision boundary is blurred and the

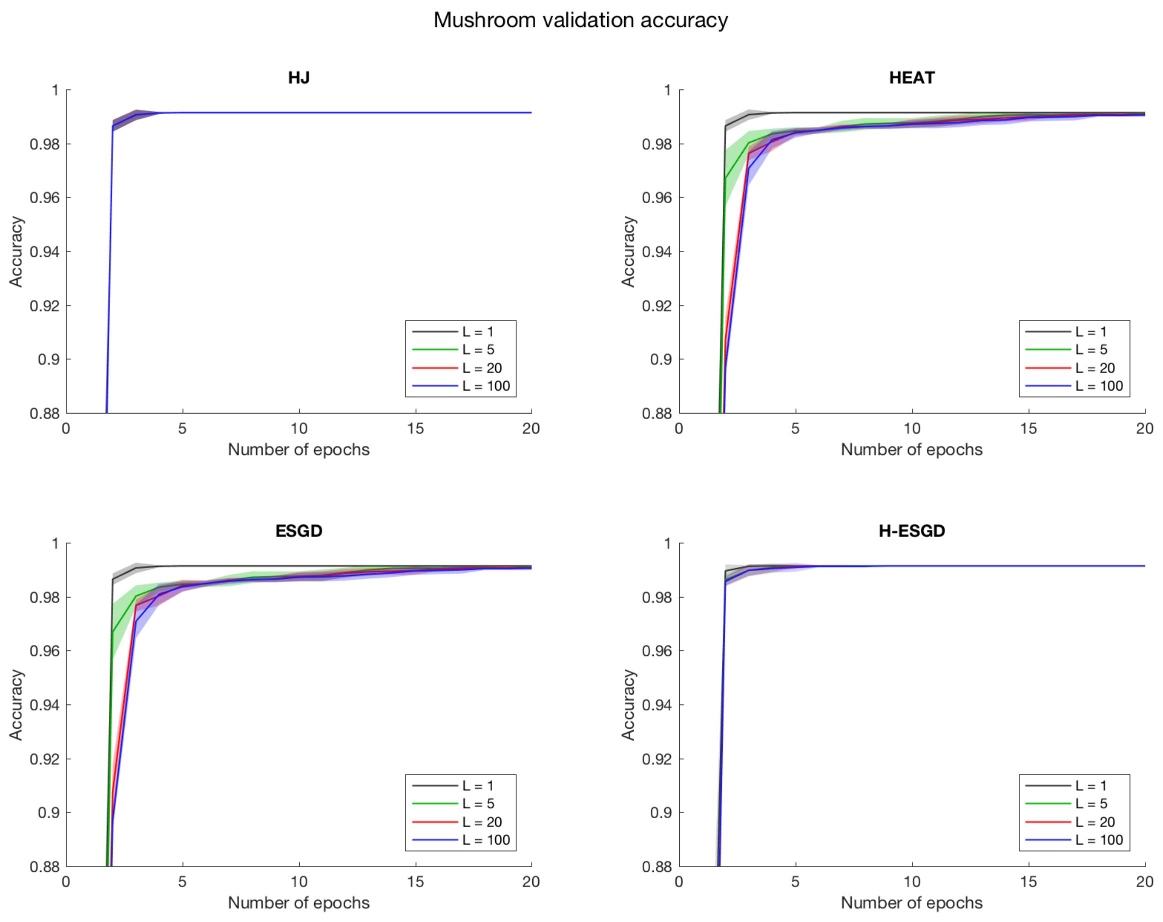


Figure 4.7: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Mushroom dataset with L set to be 1 (black), 5 (green), 20 (red) and 100 (blue).

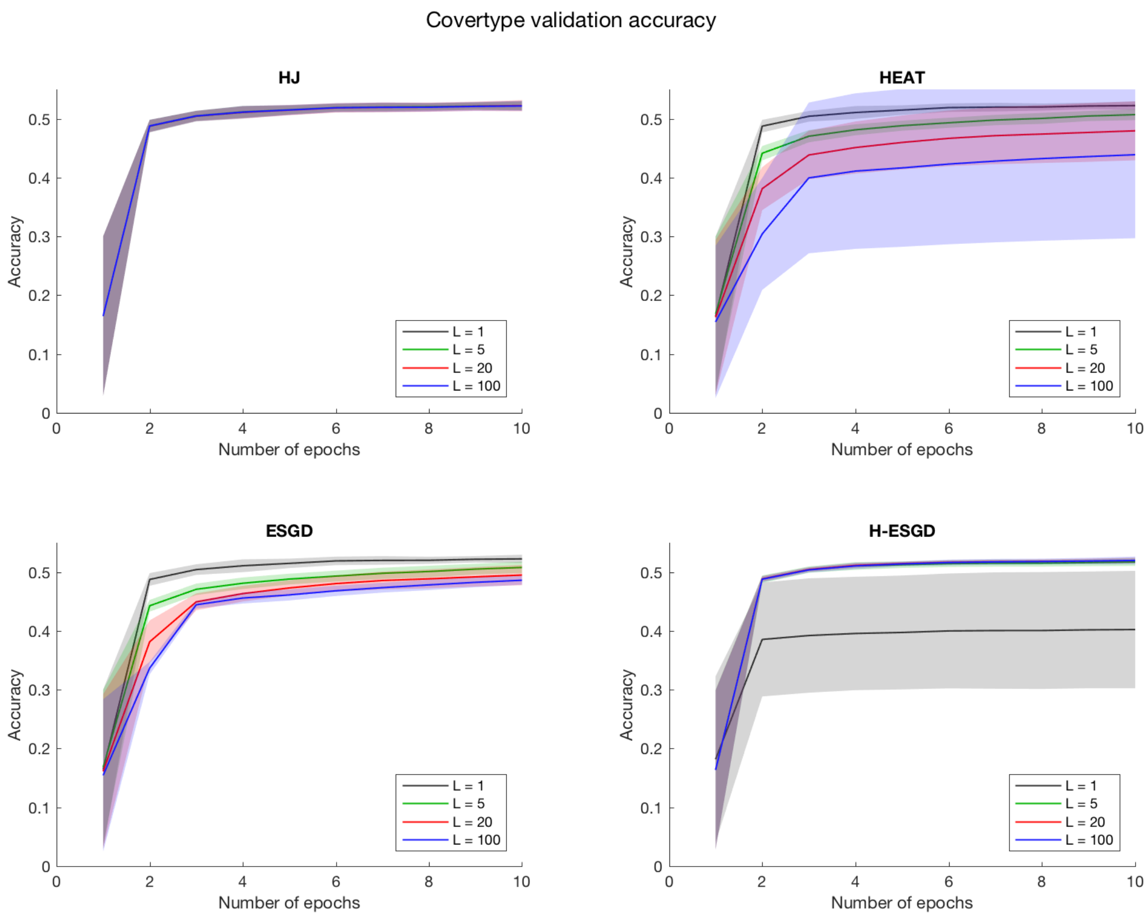


Figure 4.8: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Covertype dataset with L set to be 1 (black), 5 (green), 20 (red) and 100 (blue).

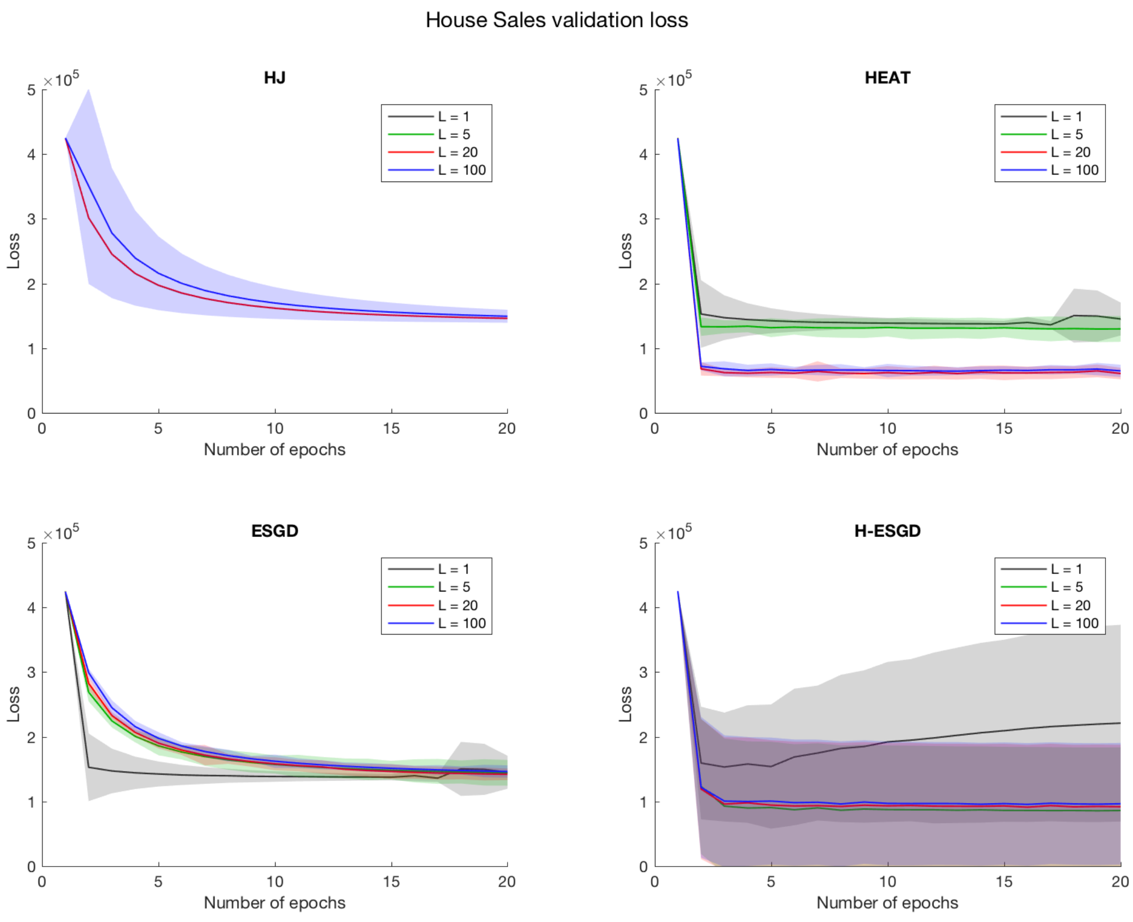


Figure 4.9: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the House Sales dataset with L set to be 1 (black), 5 (green), 20 (red) and 100 (blue).

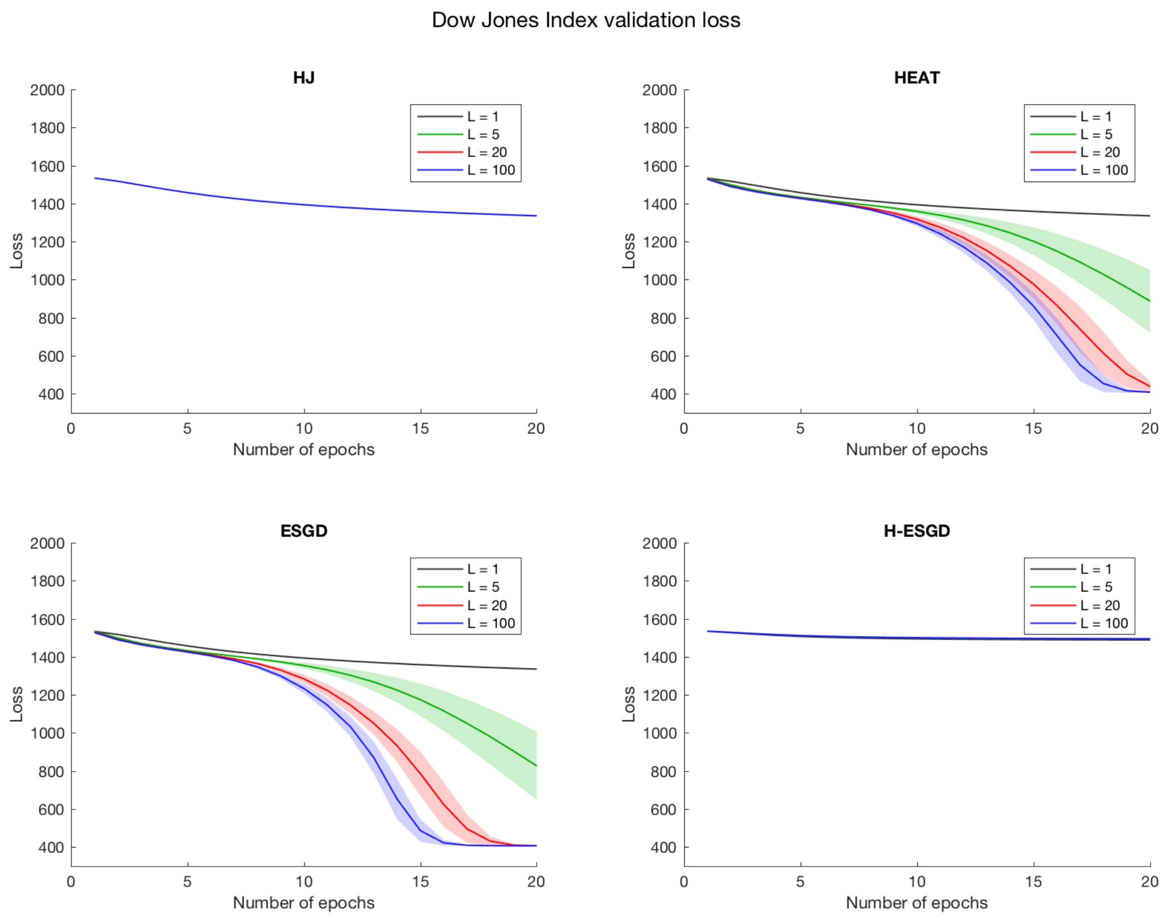


Figure 4.10: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Dow Jones Index dataset with L set to be 1 (black), 5 (green), 20 (red) and 100 (blue).

predictions become less accurate, indicating an underfitting issue. For all algorithms, a value of $\gamma \leq 0.05$ is preferred for the Mushroom dataset.

Figure 4.12 shows the validation loss for the Covertype dataset. Similar to the Mushroom dataset, different values of γ have very little impact on HJ. For HEAT and ESGD, a smaller γ gives a larger standard deviation of the predictions. This can be explained by consider the objective function, which is noisy and has more local minima when less smoothing is applied. When the objective function is noisier, the behavior of gradient descent becomes more dependent on the random initialization; solutions are more likely trapped in different local minima. We can also observe that a larger γ gives a better initial accuracy for HEAT and ESGD, mostly because the smoothing effect allows gradient descent update in a more effective direction. For H-ESGD, a slightly better accuracy is obtained when γ is smaller. A value of $\gamma = 0.02$ works the best.

Figure 4.13 shows the validation loss for House Sales. Once again, γ has little impact on HJ and HEAT. Notice that $\gamma = 0.001$ gives the lowest loss in ESGD while $\gamma = 0.0005$ and $\gamma = 0.0025$ leads to similar losses, which suggests that the appropriate value of γ for ESGD is likely between $\gamma = 0.0005$ and $\gamma = 0.0025$. As for H-ESGD, choosing $\gamma = 0.0025$ again produces too much smoothing and leads to a poor loss. A value of $\gamma = 0.0005$ works the best for HEAT and H-ESGD; a value of $\gamma = 0.001$ works the best for ESGD.

Finally, the results of Dow Jones Index are shown in Figure 4.14. HJ and H-ESGD have little change with γ and the curves in each plot almost coincide completely. For HEAT and ESGD, increasing γ improves the loss and achieves a similar effect as increasing L. A value of $\gamma = 0.01$ works the best for HEAT and ESGD; a value of $\gamma = 0.002$ works the best for ESGD.

To conclude, γ has little effect on HJ. While smaller γ is preferred for H-ESGD, a larger γ potentially improves the results for HEAT and ESGD. Recall that γ is applied to smoothen the loss function only as an aid to improve the training process of the neural network. If γ is set to a large value, the smoothened loss function could depart too much from the original loss function. For practitioners, it is recommended that γ is picked with a small initial value (e.g. 0.005), and then decreased to zero as learning proceeds. However, the optimal choice of γ is still a trial-and-error process, and the effect of γ on the performance of HEAT and ESGD depends on the neural network architecture and datasets.

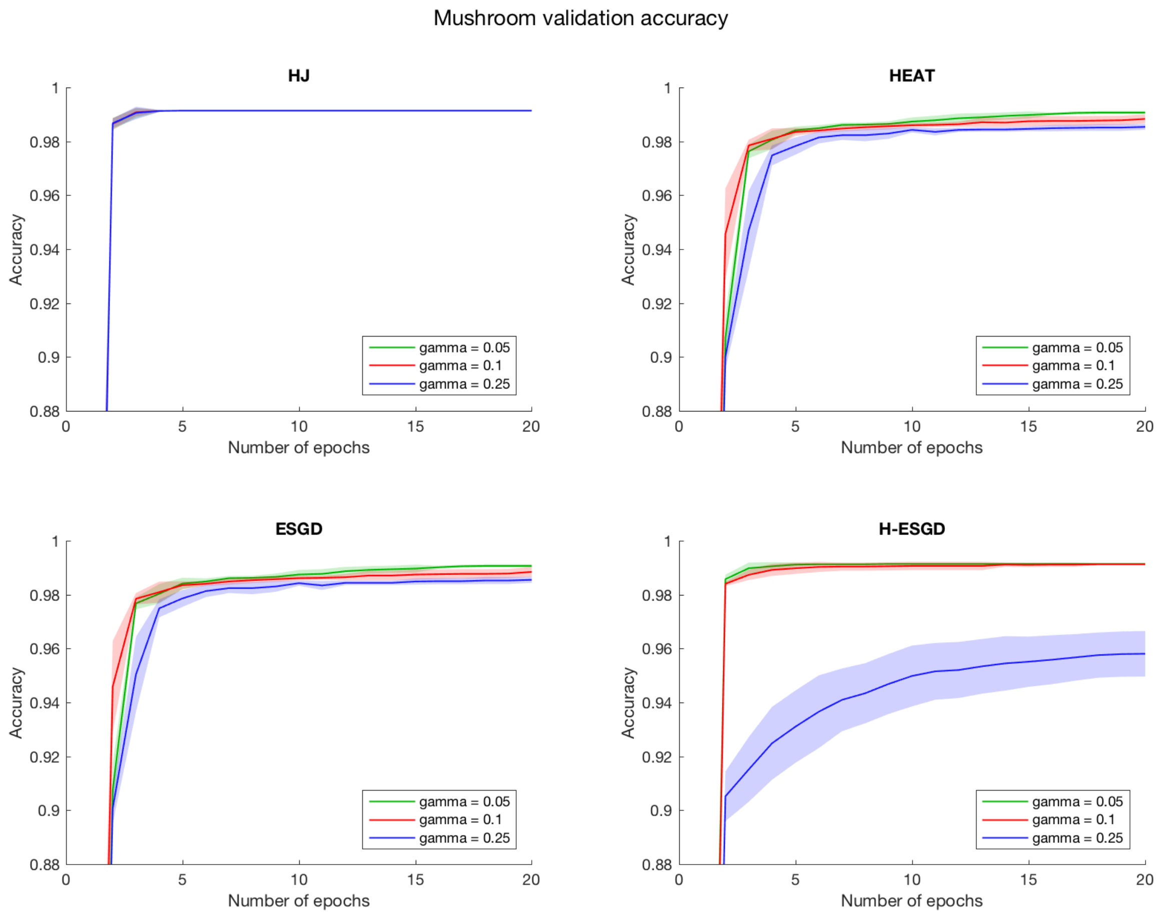


Figure 4.11: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Mushroom dataset with $\gamma = 0.05$ (green), 0.1 (red) and 0.25 (blue).

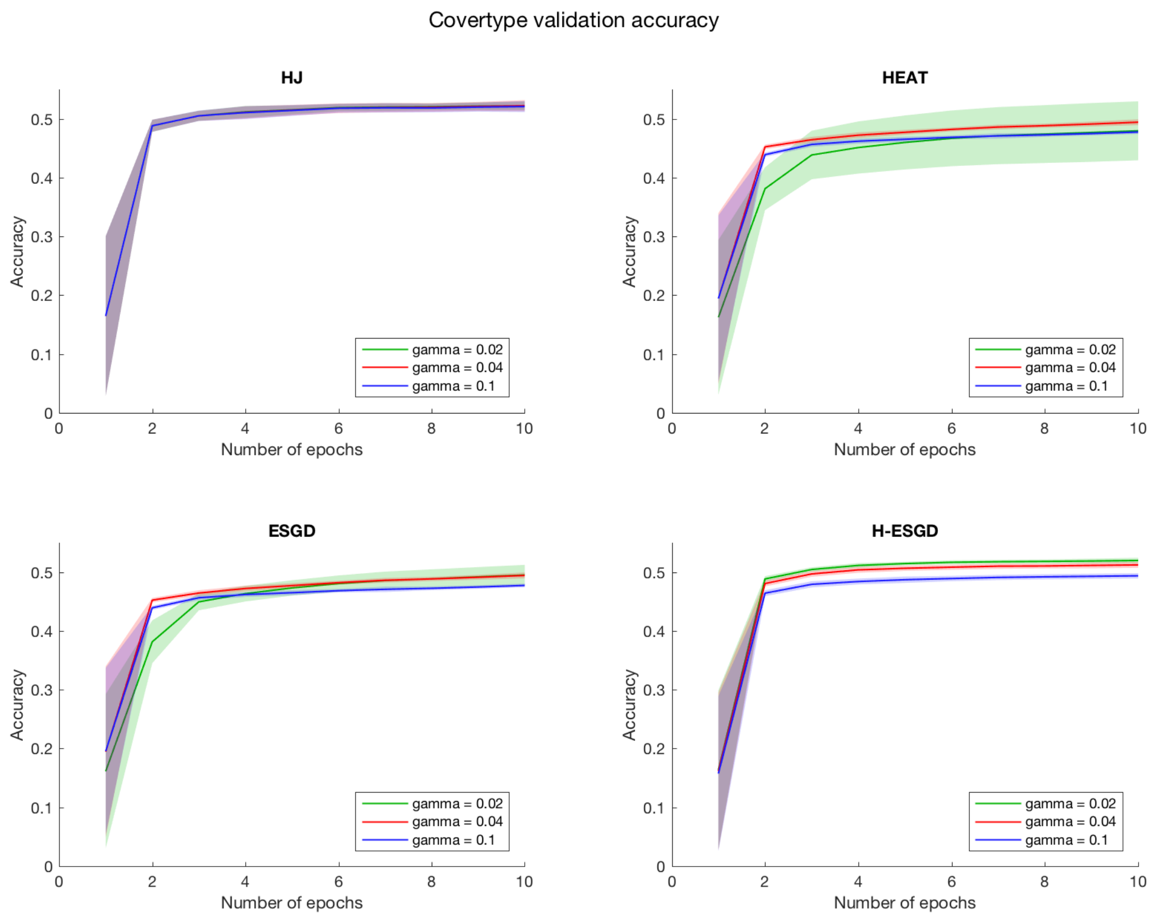


Figure 4.12: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Covertype dataset with $\gamma = 0.02$ (green), 0.04 (red) and 0.1 (blue).

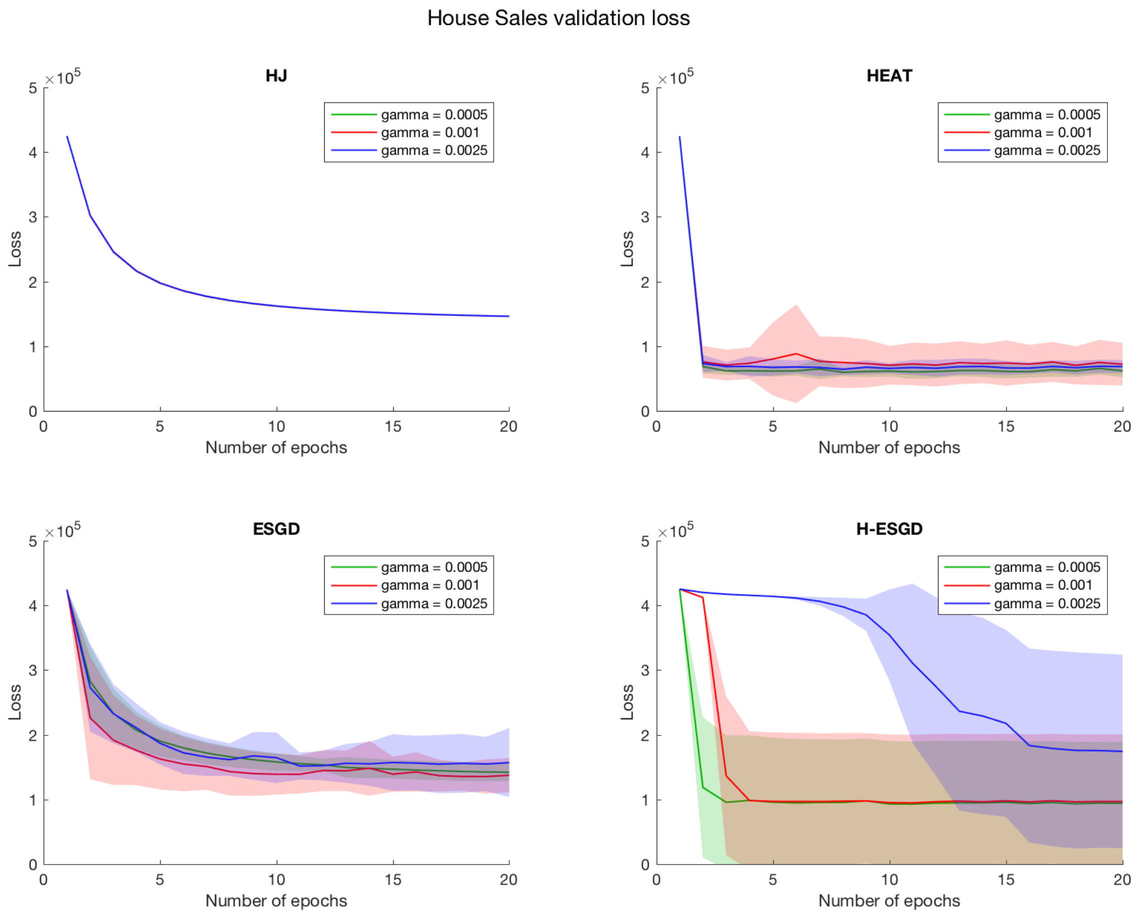


Figure 4.13: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the House Sales dataset with $\gamma = 0.0005$ (green), 0.001 (red) and 0.0025 (blue).

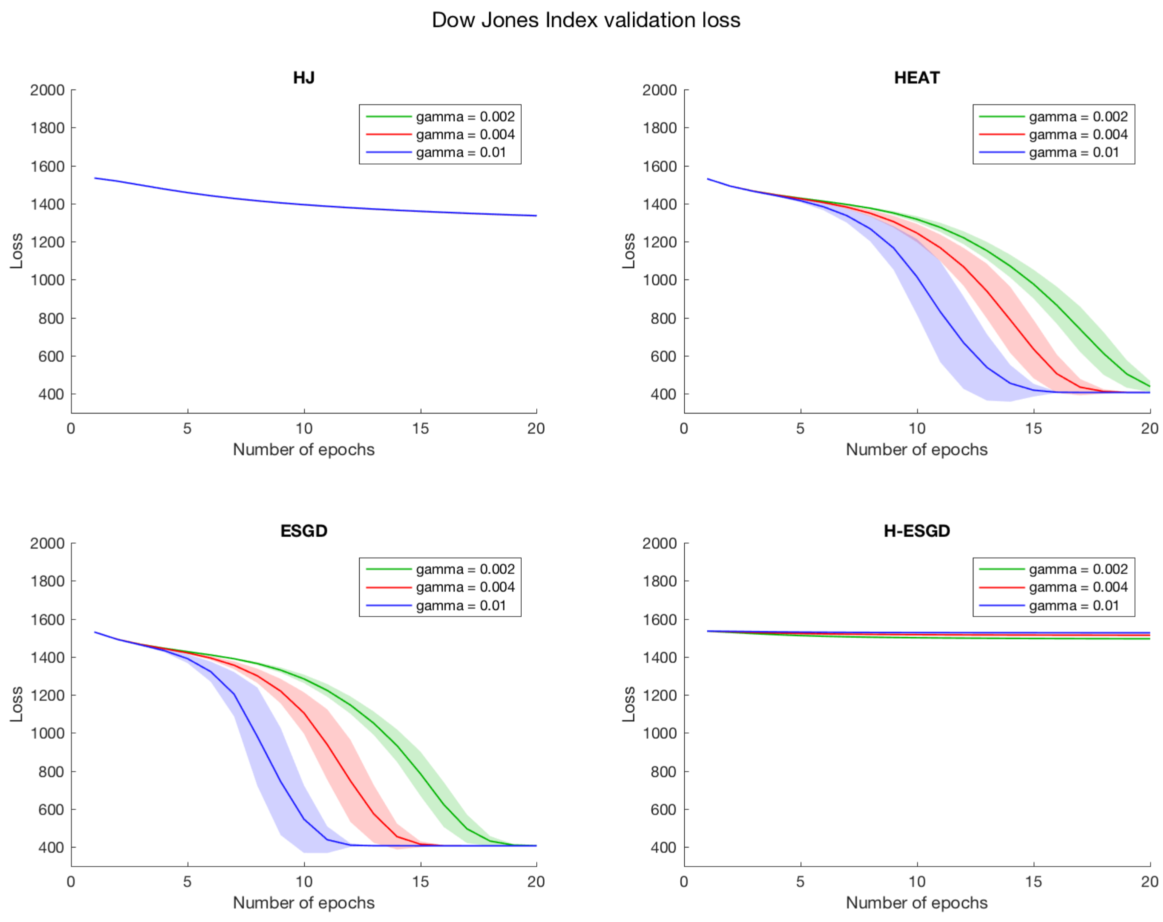


Figure 4.14: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Dow Jones Index dataset with $\gamma = 0.002$ (green), 0.004 (red) and 0.01 (blue).

4.2.6 Varying Momentum

Recall from Section 2.4.2 that the Nesterov momentum is introduced to consider the accumulated movements to determine the movement for the current iteration. The typical values to set the momentum μ in (2.12) is 0.5, 0.9, 0.99. We explore the performance of each algorithm when different momentum values are used.

Figure 4.15 shows the result for different values of momentum on SGD. For Mushroom and Dow Jones Index, setting the momentum to 0.9 and 0.99 give indistinguishable results that outperform a momentum of 0.5. For House Sales, a larger momentum has a distinctively faster convergence to a local minimum. However, the loss also bounces back after the first epoch: the large momentum "pushes" the gradient step too hard and misses the local minimum. For Covertype, the optimal momentum is 0.9. A momentum of 0.5 has slower convergence, while a momentum of 0.99 induced a large update step and jumps to a local minimum with a worse accuracy. In addition, as a larger momentum accumulates previous movements further, when the initialized parameters have highly-varied descending directions, a large standard deviation can be observed.

Figure 4.16 shows the performance of different algorithms on the Mushroom dataset. For all of the algorithms, a larger momentum induces faster convergence. For the case of HJ and H-ESGD, the curves obtained by setting 0.9 and 0.99 overlap. While setting the momentum to 0.9 and 0.99 has little difference, both outperform the momentum of 0.5. A similar observation applies to the results for Dow Jones Index dataset, as shown in Figure 4.20.

Figure 4.17 gives the results on the Covertype dataset. Similar to the case of SGD, a momentum of 0.9 outperforms 0.5, and a momentum of 0.99 gives poor results in all of the four algorithms. However, on average, H-ESGD showed the least difference in accuracy from the other momentum values, while the difference for HEAT and ESGD between momentum of 0.99 and other momentum values is substantial. A larger standard deviation is also observed for HJ and H-ESGD for a momentum of 0.99. This observation is different from the intuition that HEAT and ESGD are more stochastic than HJ and H-ESGD due to the stochastic term $N(0, 1)$ in their update rules.

The results on the House Sales dataset are given in Figure 4.18. The first key observation is that a momentum of 0.99 gives the best loss for HJ, and ESGD, and a momentum of 0.5 gives the worst results. A similar pattern is observed in H-ESGD, except that a momentum of 0.9 slightly outperforms 0.99. In addition, a larger momentum has a smaller variation for ESGD and H-ESGD. For HEAT, a momentum of 0.99 has a high mean and standard deviation caused by one of the 10 random seeds with a huge loss value. The

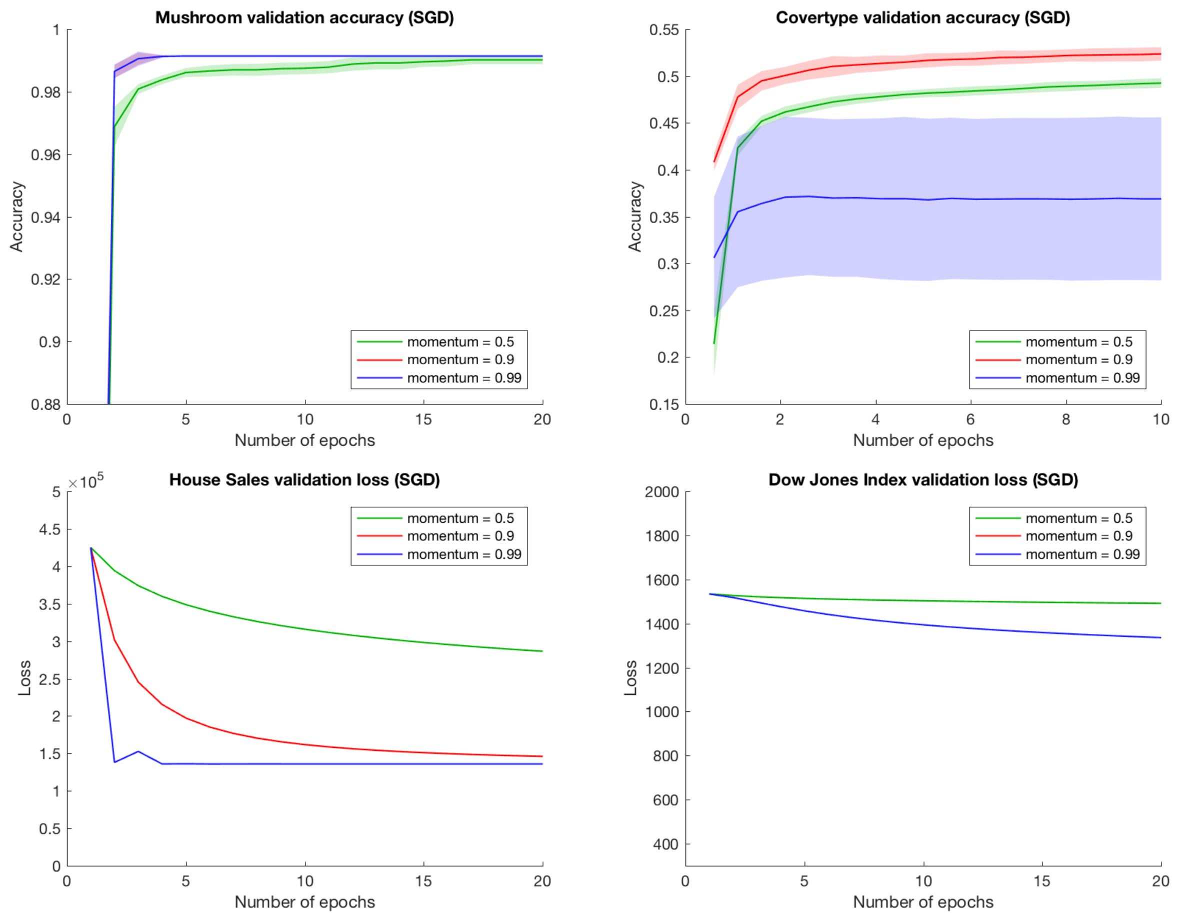


Figure 4.15: Comparison of accuracy or loss against number of epochs for SGD on the four datasets with the momentum set to 0.5 (green), 0.9 (red) and 0.99 (blue).

stochastic term in the y update of (HEAT) can give a wrong update direction and a induce a large increase in loss. Excluding this instance gives Figure 4.19. A momentum of 0.5 and 0.9 gives similar results, while a momentum of 0.99 gives a poorer loss.

Overall, the results of different momentum on different algorithms are similar to SGD, and a momentum of 0.9 gives the best results.

4.2.7 Varying Mini-batch Size

In general, a larger mini-batch size provides a more accurate estimate of the gradient and requires a smaller number of iterations per epoch, while a smaller mini-batch size is noisier and achieves a better generalization error. In this section, we explore the effect of different mini-batch sizes on different algorithms.

Figure 4.21 and 4.22 provides the overview of different mini-batch sizes applied to SGD in terms of number of epochs and CPU time in seconds respectively. As we can see from the results, a smaller mini-batch size gives a higher accuracy and lower loss in most cases. The only exception is the Covertype dataset, which achieves a higher accuracy when a mini-batch size of 128 is used instead of 32 and 64, possibly because the mini-batch size of 128 is already relatively small for the Covertype dataset, and a further decrease in mini-batch size makes the predictions highly stochastic.

In general, a larger mini-batch size also has a shorter runtime for the same number of epochs, since the number of mini-batch updates per epoch is smaller. For Dow Jones Index and Mushroom which have a relatively smaller data size, a mini-batch size of 64 appears require a very slightly smaller amount of time than a mini-batch size of 128. This is possibly due to small round-off errors when the overall CPU time is small. The difference is more apparent when the data size is larger and the neural network is trained for a longer period of time, for instance in Covertype and House Sales.

Figure 4.23 gives the results of the Mushroom dataset in terms of number of epochs and Figure 4.24 gives the results in terms of CPU time. In general, a mini-batch size of 32 achieved the best accuracy in the same amount of epochs, while a mini-batch size of 64 achieved the best accuracy in the shortest amount of time. The overall training time to finish 20 epochs is the shortest for mini-batch size of 128 and longest for 32. This difference in runtime is the most apparent in H-ESGD.

The results of Covertype dataset are shown in 4.25 and 4.26. A mini-batch size of 32 achieved a better accuracy for HEAT and ESGD, while a mini-batch size of 128 achieved a better accuracy for HJ and H-ESGD. Recall that the stochastic term is present in the

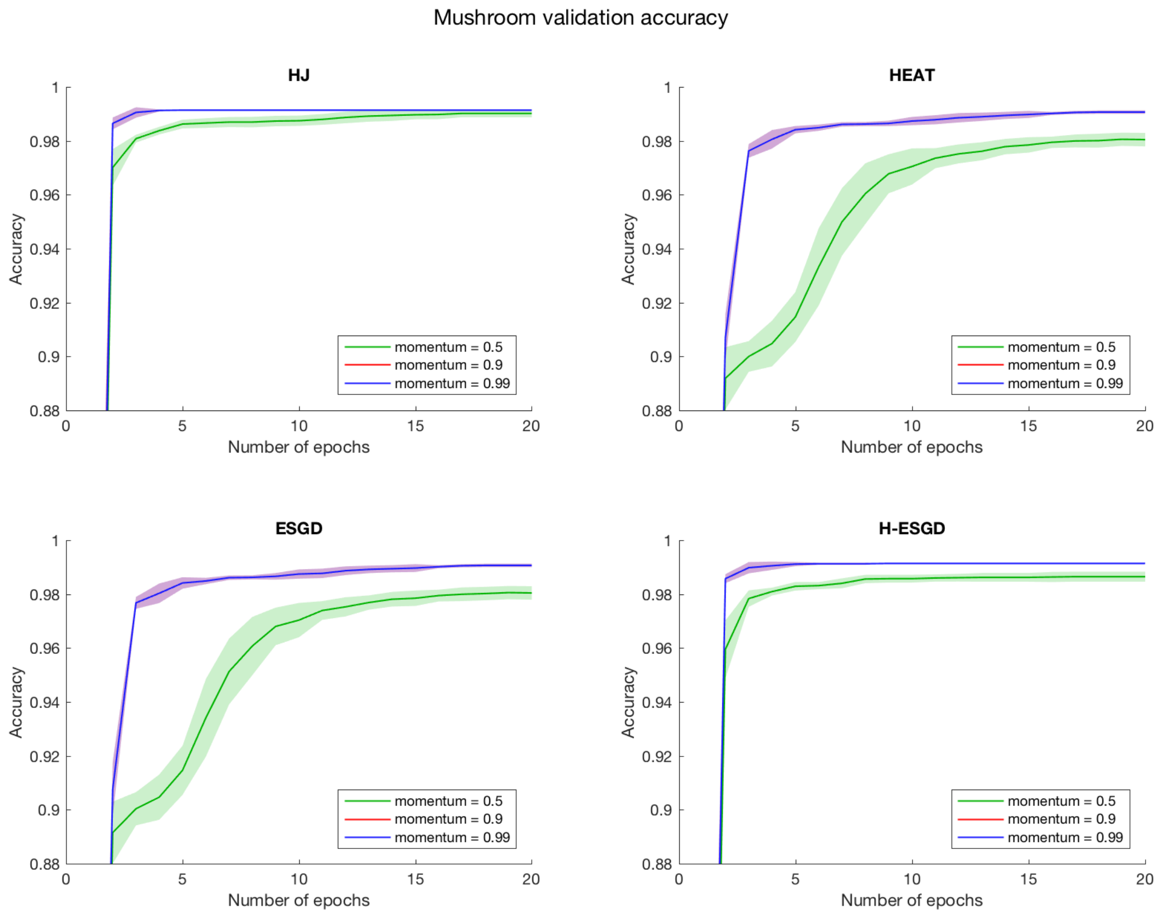


Figure 4.16: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Mushroom dataset with the momentum set to 0.5 (green), 0.9 (red) and 0.99 (blue).

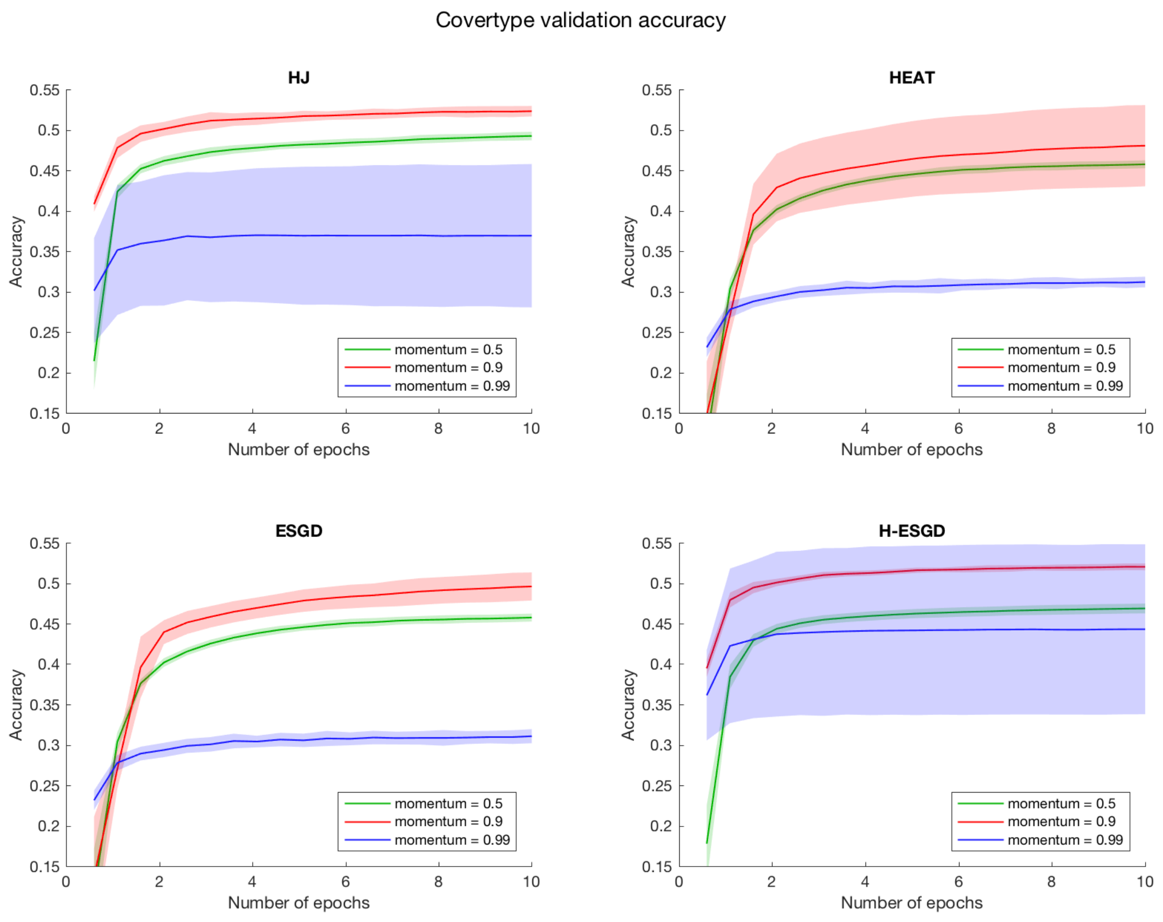


Figure 4.17: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Covertype dataset with the momentum set to 0.5 (green), 0.9 (red) and 0.99 (blue).

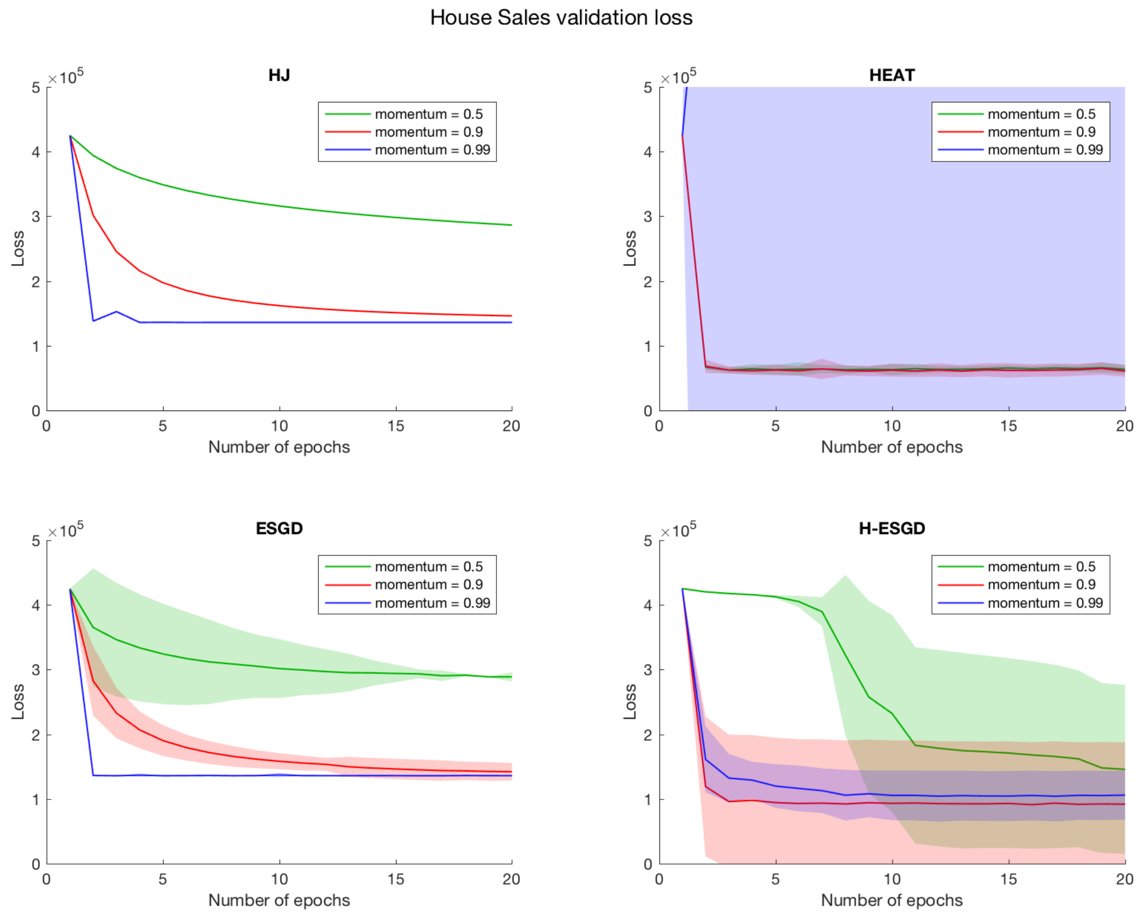


Figure 4.18: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the House Sales dataset with the momentum set to 0.5 (green), 0.9 (red) and 0.99 (blue).

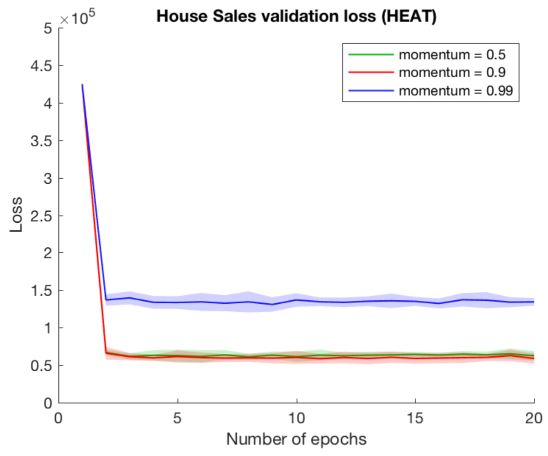


Figure 4.19: Comparison of loss against number of epochs for HEAT on the House Sales dataset with the momentum set to 0.5 (green), 0.9 (red) and 0.99 (blue), after the deletion of a deviant instance.

updates rules (HEAT) and (ESGD), but not in HJ and H-ESGD, possibly because the stochastic term has an alleviation and regularization effect of the noisy updates caused by the small mini-batch size. In all cases, a larger mini-batch size requires a significantly shorter amount of time to finish running 10 epochs.

For the results of House Sales dataset against number of epochs (Figure 4.27), HJ achieves a better learning curve with a smaller mini-batch size. HEAT has an opposite result: a smaller mini-batch size gives a higher and noisier loss. Similar to the case of HEAT in Section 4.2.5, a certain instance in ESGD has caused the mean and standard deviation of the loss to rise quickly for a mini-batch size of 32. Removing this instance and re-plotting gives Figure 4.28. We can see that a smaller mini-batch size gives the fastest convergence to a local minimum. For this case, changing the mini-batch size has opposite effect on HEAT and ESGD. For the results versus CPU time (Figure 4.29), the removal of the abnormal instance in HEAT yields Figure 4.30. The difference in mini-batch size on H-ESGD is mainly reflected in the CPU time rather than the number of epochs, where the three curves all eventually reach a similar loss. A mini-batch size of 128 requires the shortest amount of time to finish the whole training.

As for the Dow Jones Index dataset, a smaller mini-batch size has an advantage of a faster convergence to a local minimum in terms of both the number of epochs and CPU time. The results can be found in Figures 4.31 and 4.32 respectively.

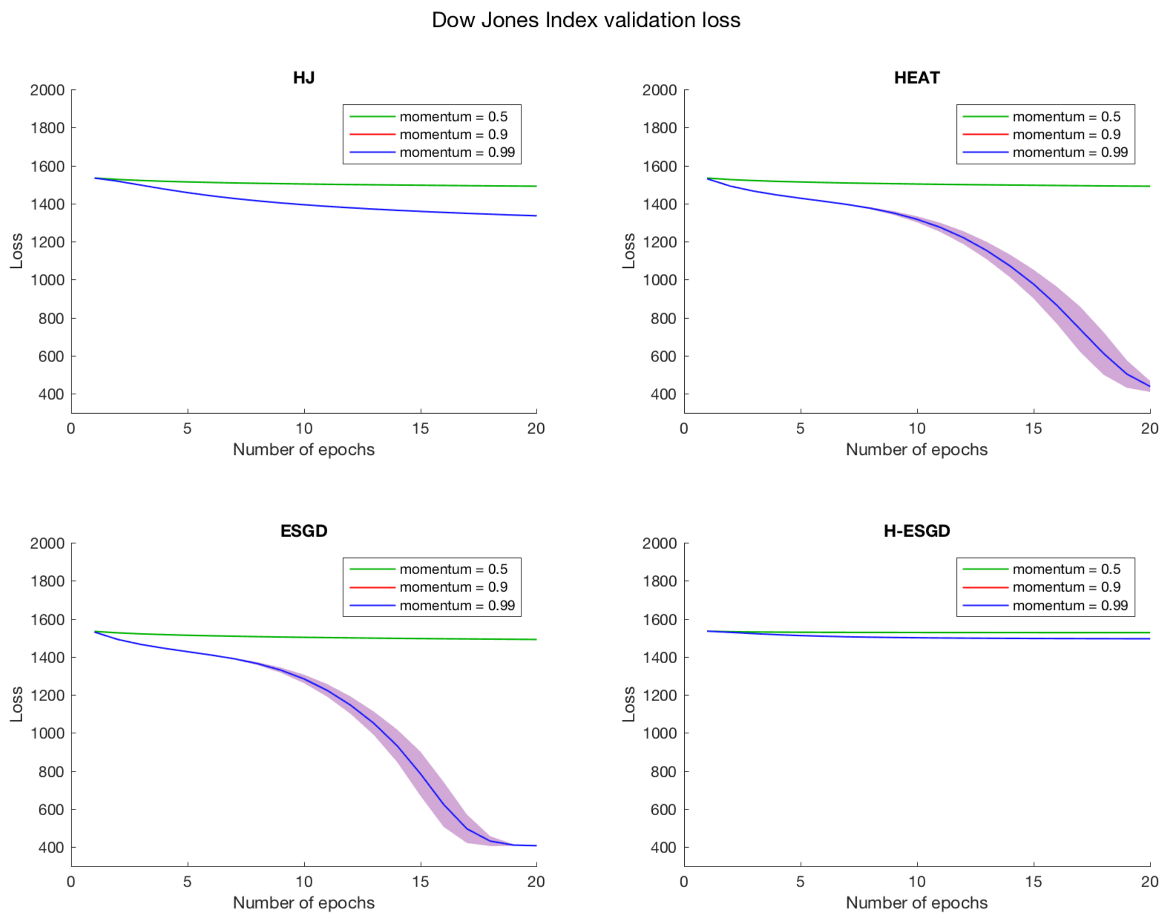


Figure 4.20: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Dow Jones dataset with the momentum set to 0.5 (green), 0.9 (red) and 0.99 (blue).

To sum up, a similar difference in mini-batch size is observed for all of the algorithms as in SGD. A smaller mini-batch size tend to give a better loss or accuracy while requiring a longer time of training.

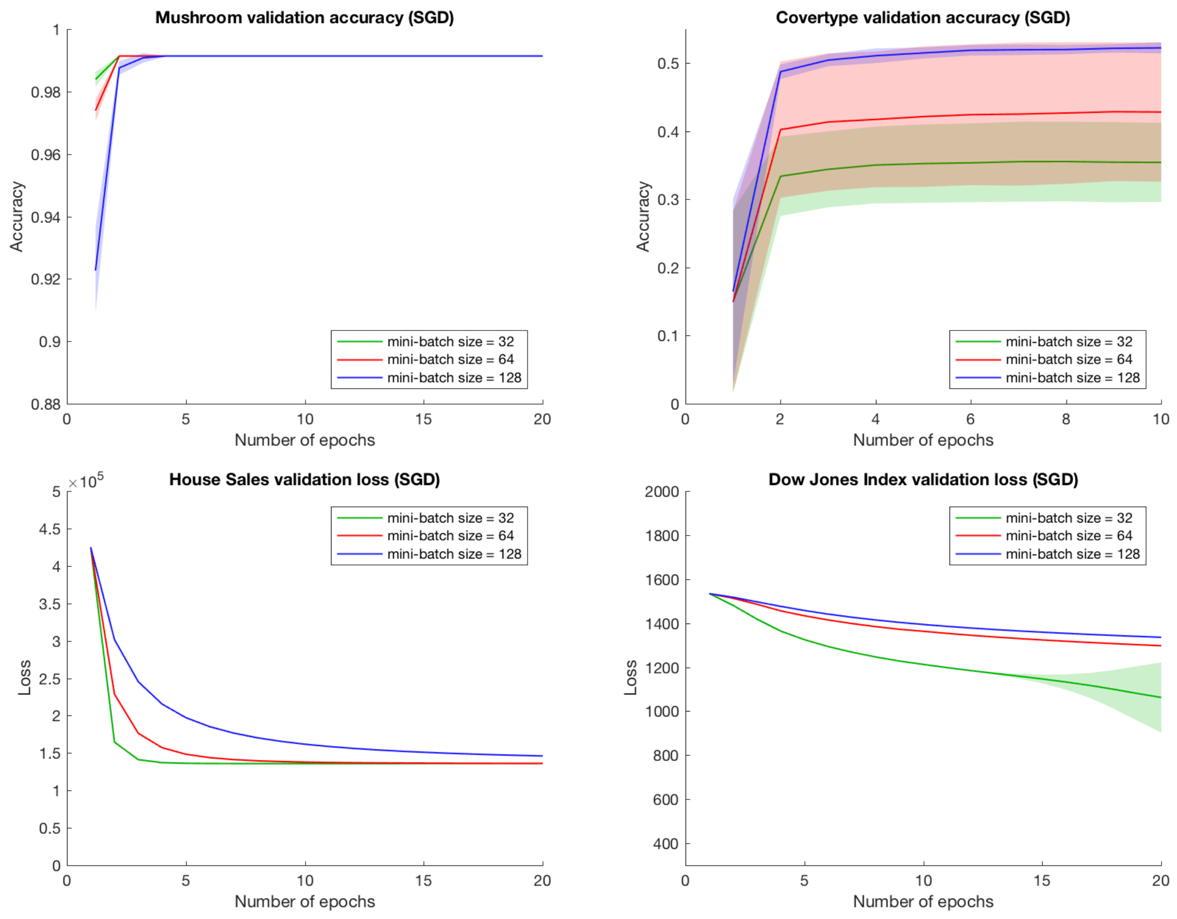


Figure 4.21: Comparison of accuracy or loss against number of epochs for SGD on the four sets with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

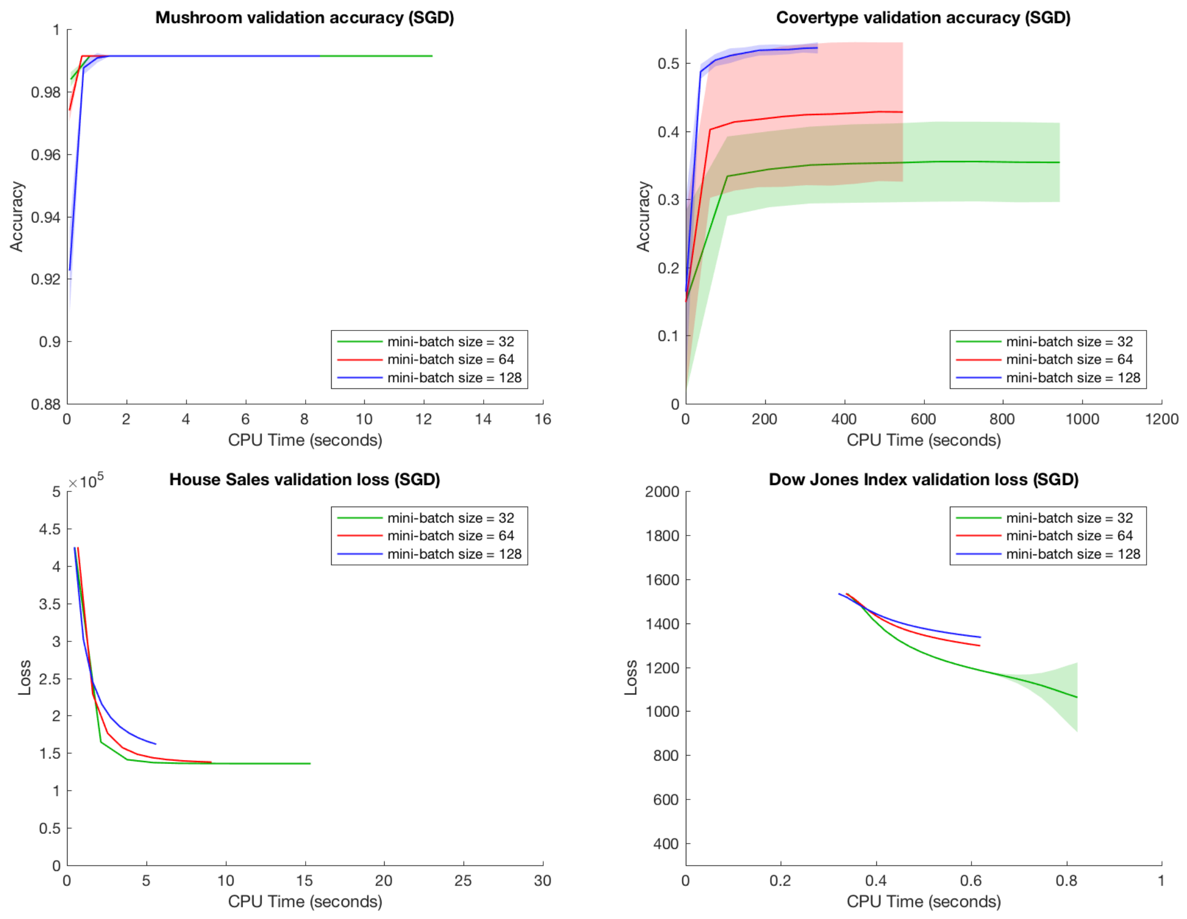


Figure 4.22: Comparison of accuracy or loss against CPU time on the four sets with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

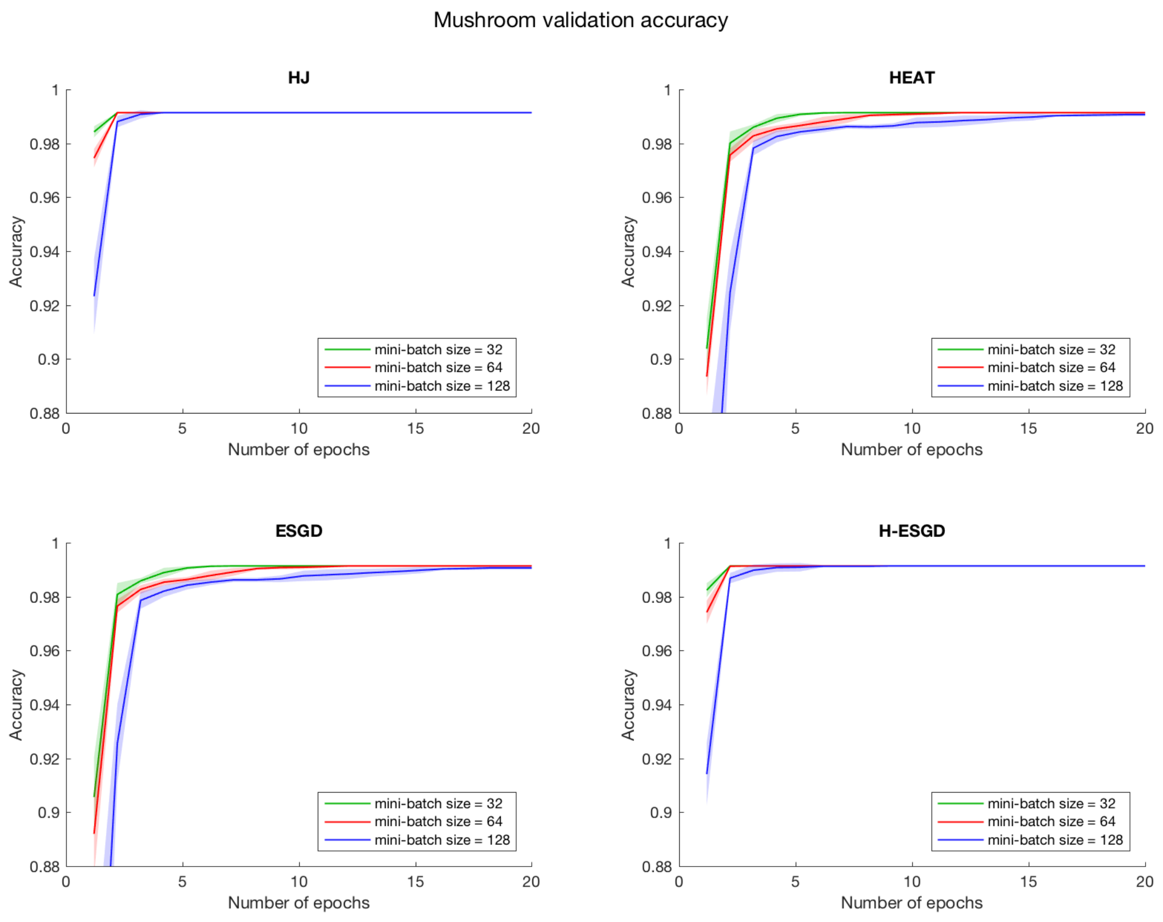


Figure 4.23: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Mushroom dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

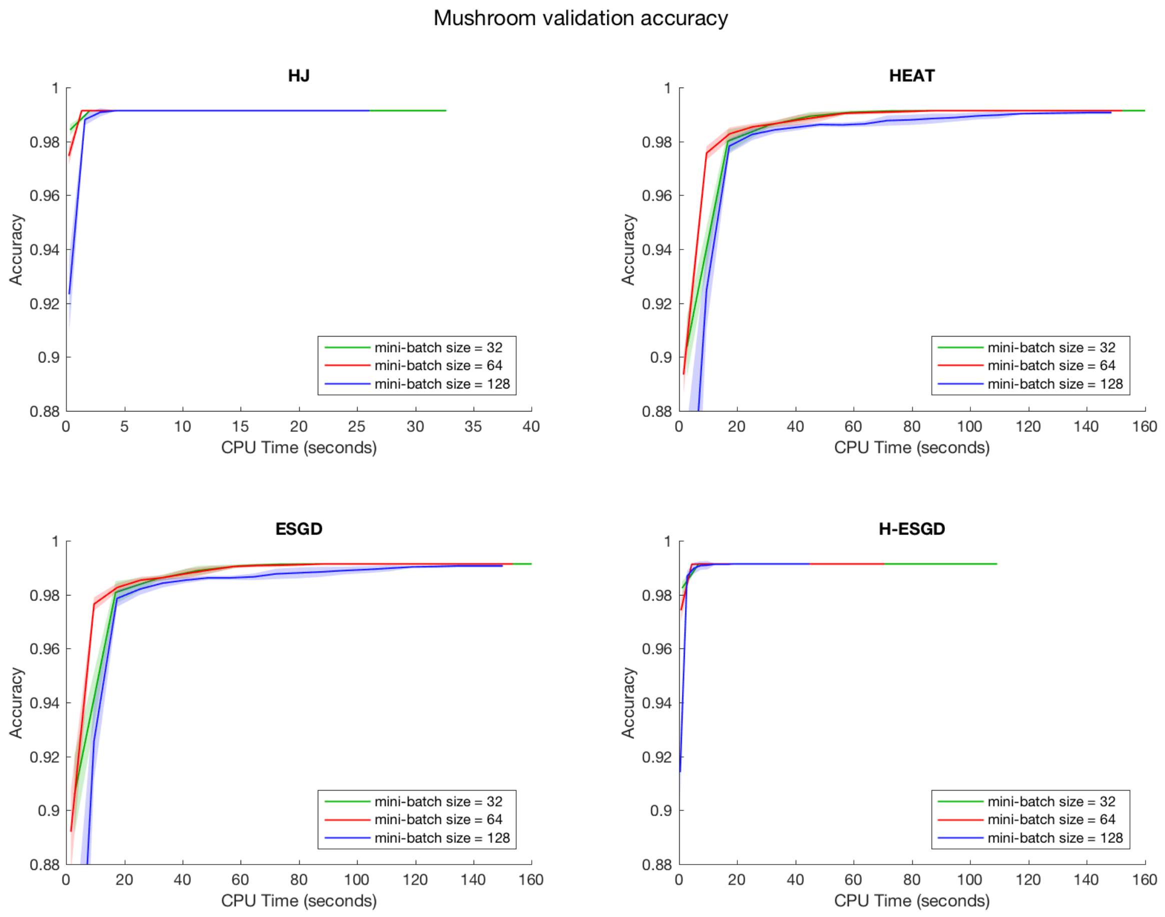


Figure 4.24: Comparison of accuracy against CPU time for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Mushroom dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

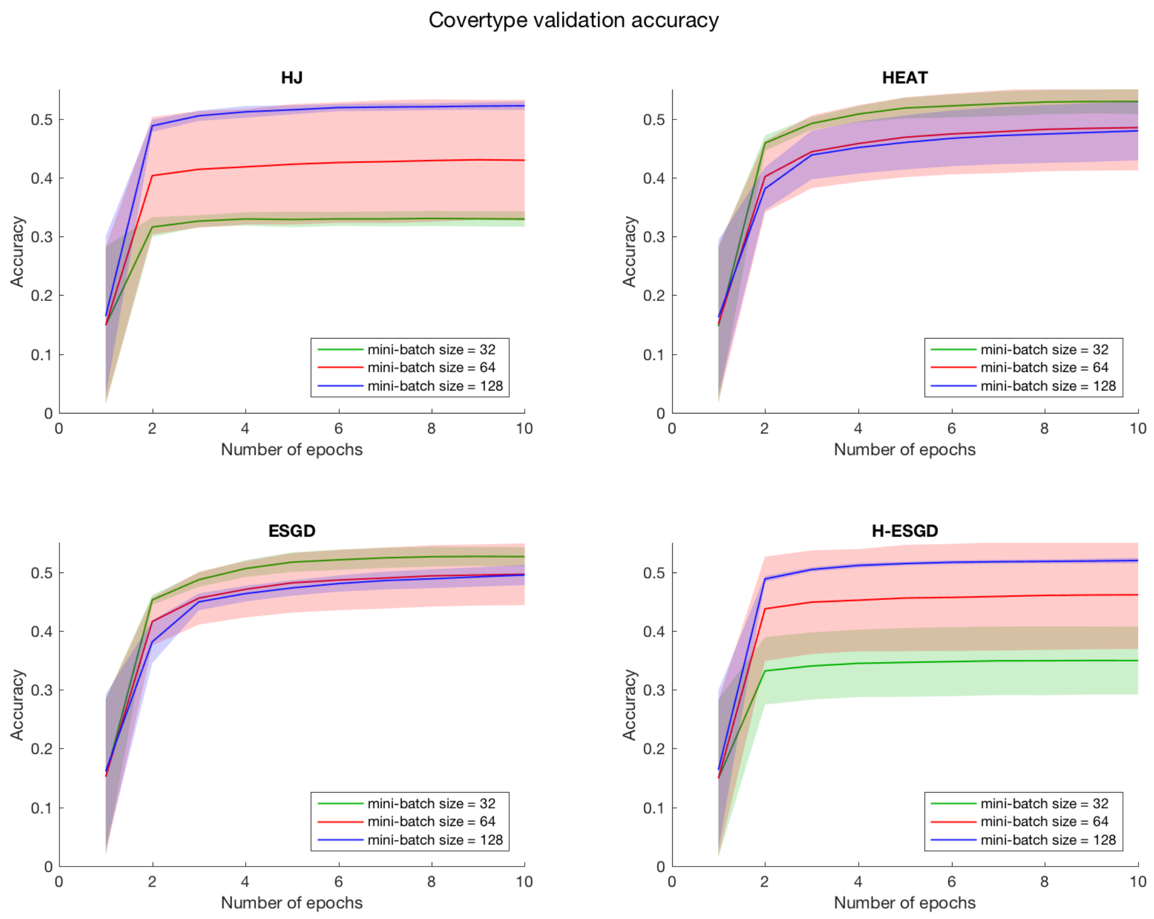


Figure 4.25: Comparison of accuracy against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Covertype dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

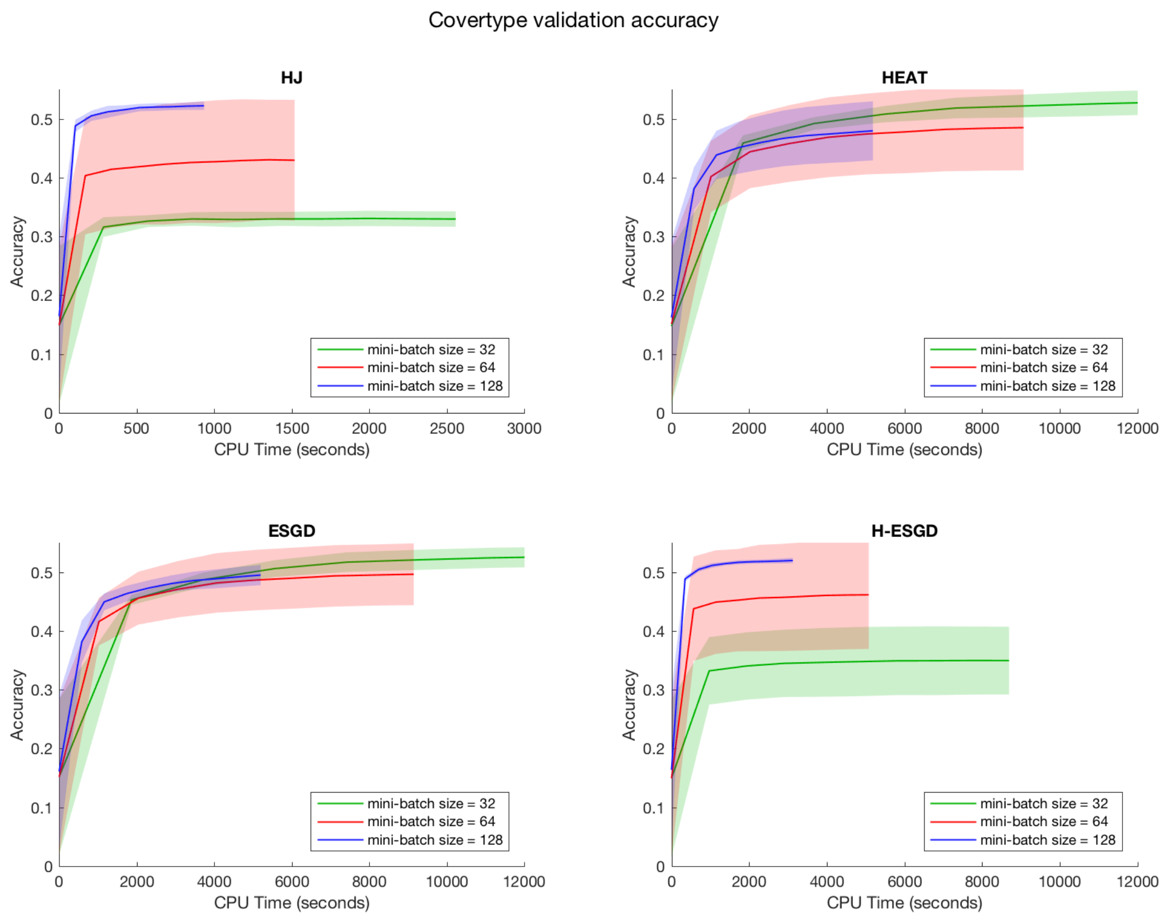


Figure 4.26: Comparison of accuracy against CPU time for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Covertype dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

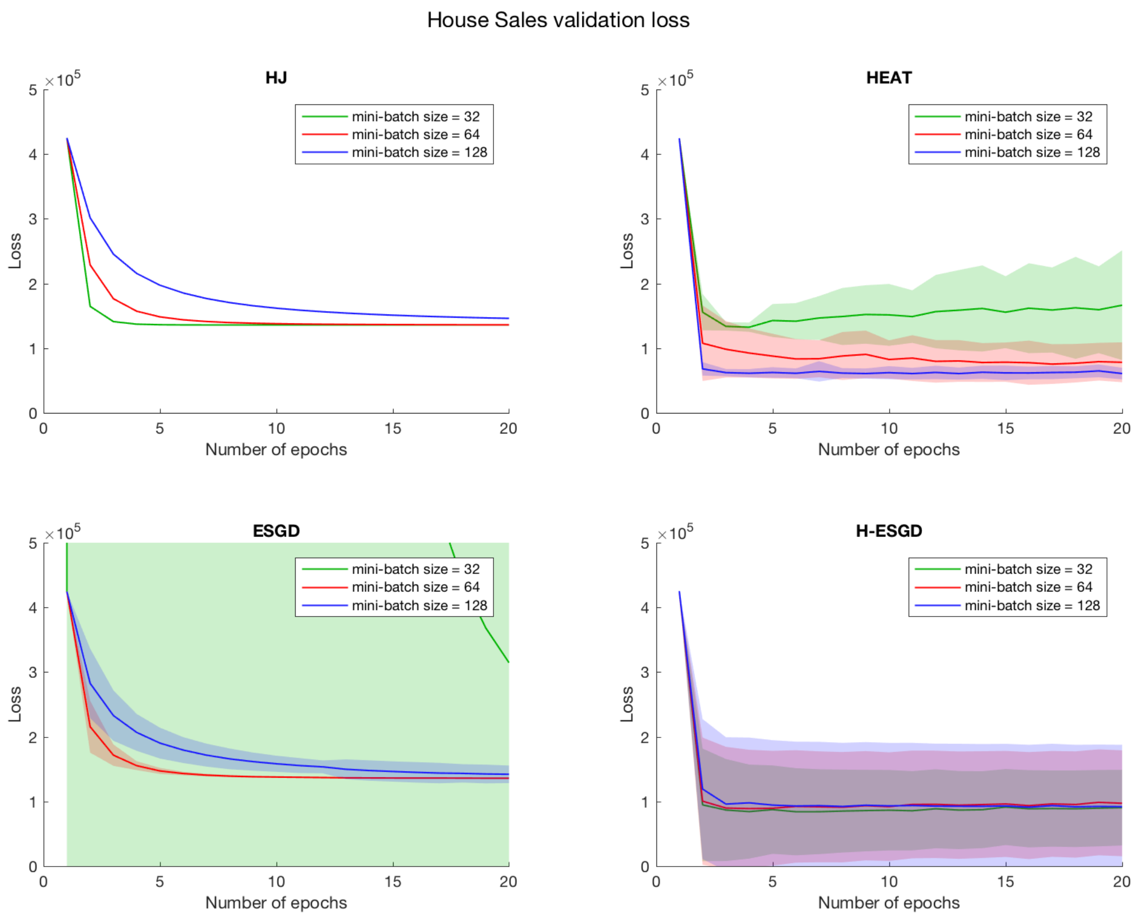


Figure 4.27: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the House Sales dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

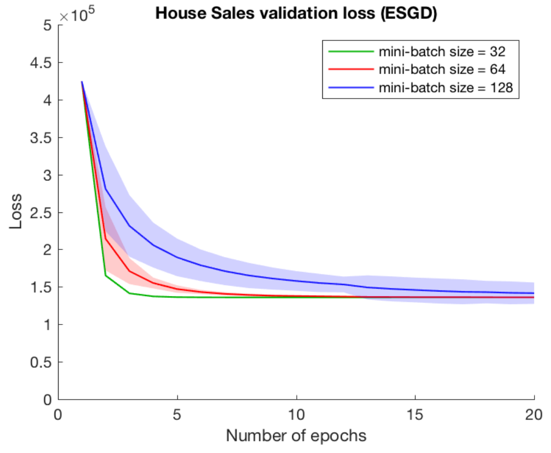


Figure 4.28: Comparison of loss against number of epochs for ESGD on the House Sales dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue), after the deletion of a deviant instance.

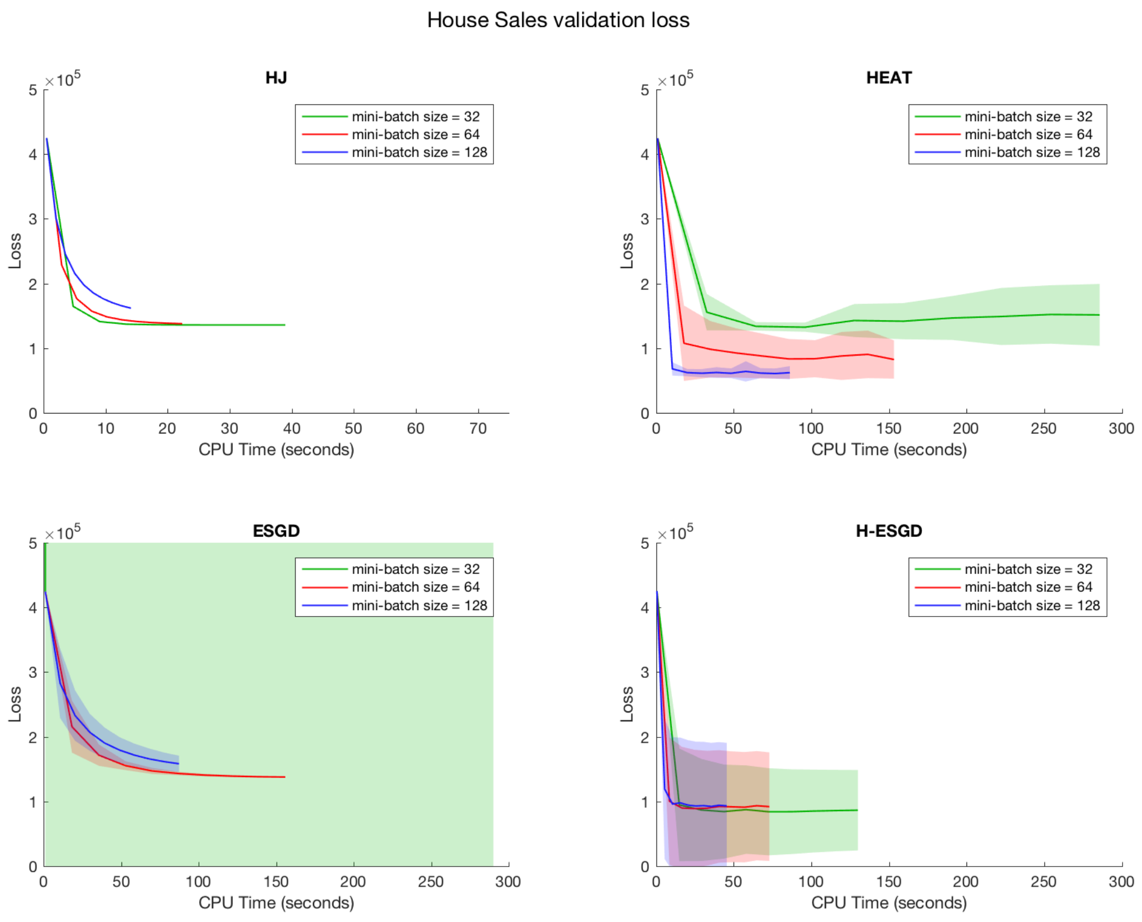


Figure 4.29: Comparison of loss against CPU time for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Dow Jones dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

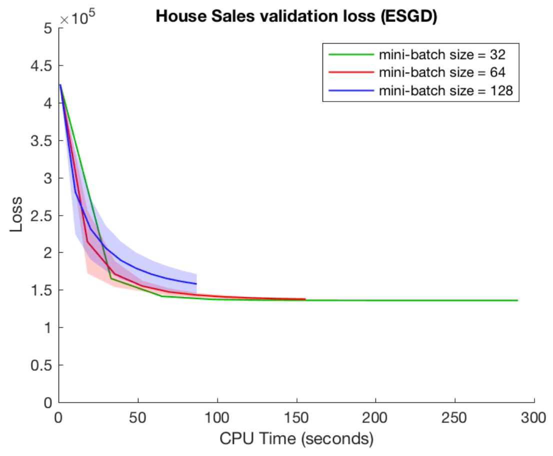


Figure 4.30: Comparison of loss against CPU time for ESGD on the House Sales dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue), after the deletion of a deviant instance.

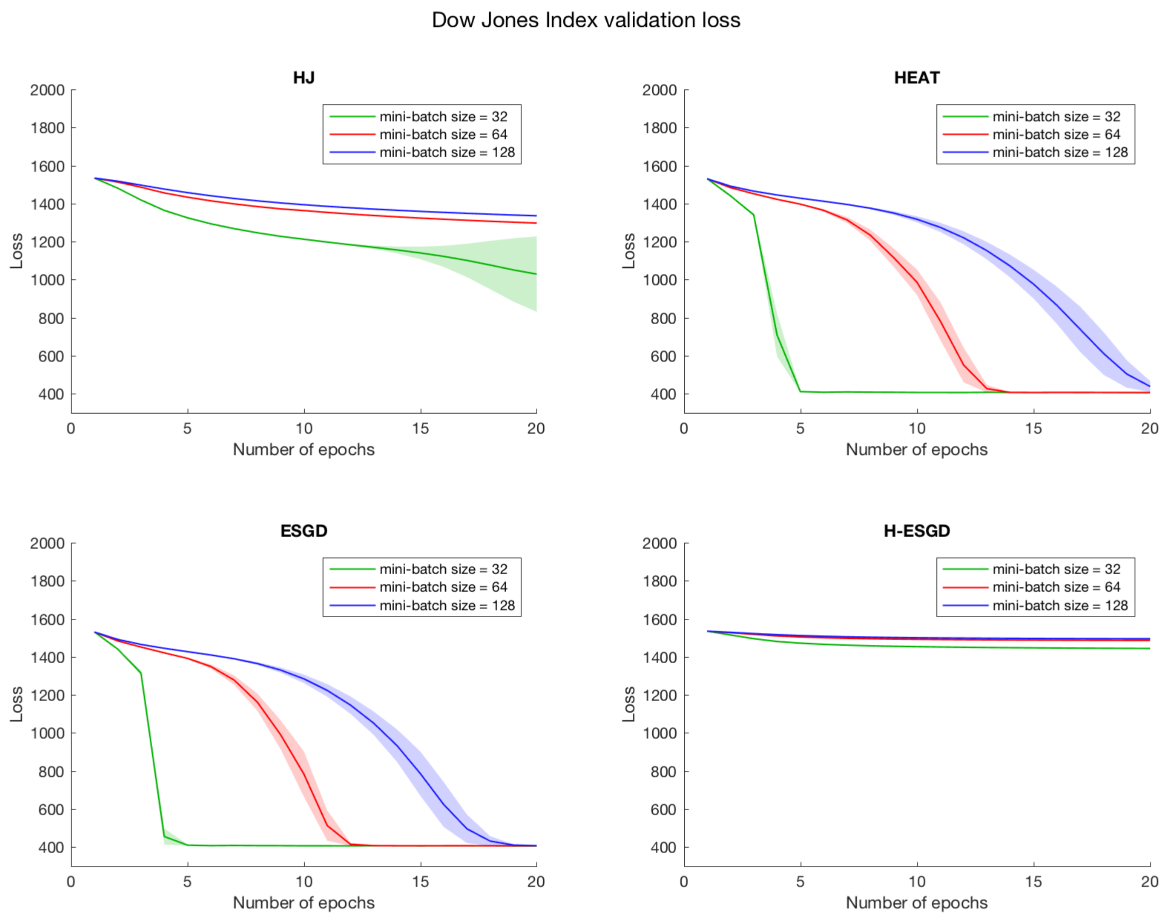


Figure 4.31: Comparison of loss against number of epochs for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Dow Jones Index dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

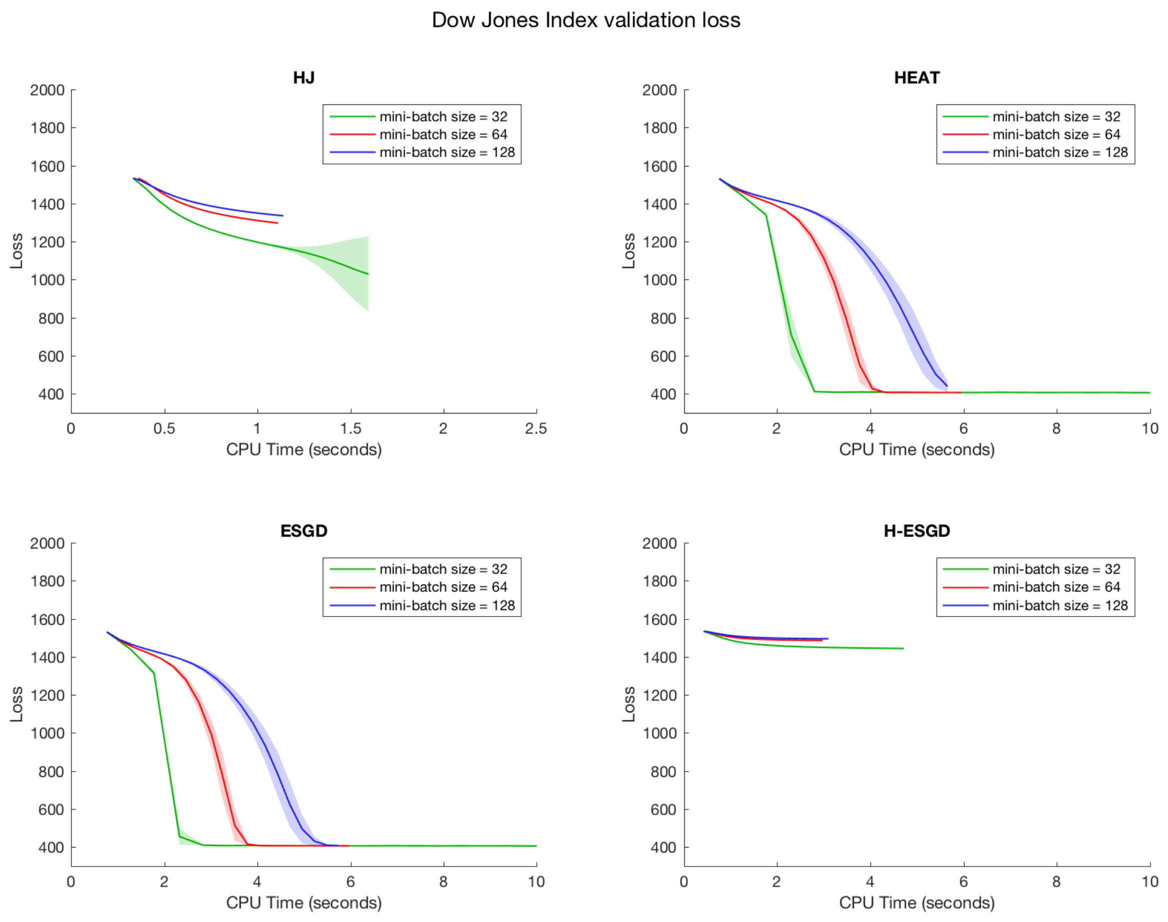


Figure 4.32: Comparison of loss against CPU time for the optimization methods HJ, HEAT, ESGD and H-ESGD on the Dow Jones Index dataset with the mini-batch size set to 32 (green), 64 (red) and 128 (blue).

Chapter 5

Conclusion

In this paper, we have explored the four optimization algorithms HJ, HEAT, ESGD and H-ESGD for their performance on neural networks. We studied the effect of hyperparameters on each algorithm when applied to different datasets.

Among all the algorithms, SGD has the shortest CPU run time and performs the best when the underlying function requires little smoothing. HJ is theoretically useful, but the effect is not highlighted in the tested datasets. HEAT, ESGD and H-ESGD tend to achieve a better solution on more complicated machine learning tasks. It is advised that L is set to 5 and ϵ is set to 0.2. However, the optimal learning rate and smoothing parameter γ highly depend on the datasets and neural network architecture. HEAT and ESGD tend to be more dependent on the initialization of parameters; on the other hand, H-ESGD tends to have a more stable performance. This means that HEAT and ESGD can potentially produce the best performance among all the algorithms; however, the algorithms could also result in a poor performance if the neural network contains large gradients.

For all of the algorithms, a momentum of 0.9 is recommended. The choice of mini-batch size depends on the purpose of training the neural network. If efficiency is more important than the overall quality of the solution, a mini-batch size of 128 is recommended; otherwise, a mini-batch size of 32 likely gives an improved solution.

To summarize, if the time allowed to train the neural network is limited, SGD is likely the best choice of algorithm. In general, SGD is recommended over HJ for a higher efficiency, whereas HEAT, ESGD and H-ESGD could be better algorithms than SGD when accuracy is more crucial and the underlying function is noisy. We conclude that HEAT, ESGD and H-ESGD do improve the neural network performance by applying smoothing to the loss function of the tested datasets, but the effect of HJ is not as noticeable.

References

[1] C. Baldassi, A., C. Lucibello, L. Saglietti, and R. Zecchina. Local entropy as a measure for sampling solutions in constraint satisfaction problems. *Journal of Statistical Mechanics: Theory and Experiment*, 2(023301), 2016.

[2] A. Blackard, Jock, and Denis Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. 24:131–151, 12 1999.

[3] P. Chaudhari, A. Oberman, S. Osher, S. Soatto, and G. Carlier. Deep relaxation: Partial differential equations for optimizing deep neural networks. April 2017.

[4] R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. International Conference on Machine Learning, 2013.

[5] J. Duchi, E. Hazan, and Y. Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(2011):2121–2159.

[6] L. C. Evans. *Partial Differential Equations: Second Edition (Graduate Studies in Mathematics)*. American Mathematics Society, 1998.

[7] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] C. Gulcehre, M. Moczulski, M. Denil, and Y. Bengio. Noisy activation functions. April 2016.

[9] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction, Second Edition*. Springer, 2009.

[10] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. March 2015.

[11] A. Krizhevsky, V. Nair, and G. Hinton. Cifar-10 (canadian institute for advanced research). 2009. http://www.cs.toronto.edu/~kriz/cifar.html.

[12] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012.

[13] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010. http://yann.lecun.com/exdb/mnist/.

[14] M. Lichman. UCI machine learning repository, 2013. http://archive.ics.uci.edu/ml.

[15] H. Mobahi. Training recurrent neural networks by diffusion. February 2016.

[16] J. Moreau. Proximité et dualtité dans un espace Hilbertien. *Bulletin de la Société Mathématique de France*, 93:273–299, 1965.

[17] V. Nair and G. Hinton. Rectified linear units improve restricted boltzmann machines. International Conference on Machine Learning, 2010.

[18] G. A. Pavliotis and A. M. Stuart. *Multiscale Methods – Averaging and Homogenization*. Springer, 2007.

[19] S. Ruder. An overview of gradient descent optimization algorithms. June 2017.

[20] D. Rumelhart, G. Hinton, and R. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536.

[21] A. Samuel. Some studies in machine learning using the game of checkers. 1959.

[22] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. Generating text with recurrent neural networks. International Conference on Machine Learning, 2011.

[23] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. International Conference on Machine Learning, 2013.

[24] M. Welling and Y. W. Teh. Bayesian learning via stochastic gradient Langevin dynamics. 2011.

[25] Y. Yu. Better approximation and faster algorithm using the proximal average. In *Advances in Neural Information Processing Systems*, 2013b.

[26] S. Zhai, Y. Cheng, W. Lu, and Z. Zhang. Deep structured energy based models for anomaly detection. June 2016.