Deep Neural Networks and Accelerating them using Singular Value Decomposition

Samyak S Sarnayak
samyak201@gmail.com

Abhijit Mohanty
mohantyabhijit074@gmail.com

Srinidhi Temkar
srinidhitemkar6699@gmail.com

May 17, 2020

Contents

1 Introduction

1.1 A Brief Introduction to Neural Networks

Traditional machine learning systems require carefully curated, structured and processed data to be useful. Important features have to be extracted manually from the data, usually with the help of domain knowledge. This limits the amount and variety that can be used to train successful models. Methods which allow a model to learn representations from raw data and successfully classify or predict the output are collectively called *representation learning* [1].

Deep learning is a form of representation learning where a number of connected layers of non-linear units, called *neurons*, help the so-called neural network learn any arbitrary function. In theory, deep neural networks can behave as universal function approximators [2] by using an infinite number of neurons in a single layer. In practice, a large number of layers are used with finite number of neurons in each of them. Each layer of a neural network transforms the outputs from the previous layer into an abstract representation, the data becomes more and more abstract as one traverses down the layers.

Deep Neural Networks (DNNs) have seen many applications in many domains where it has surpassed state-of-the-art models in classification and regression tasks. DNNs are very well suited for high-dimensional data where it can learn even very complex underlying structures.

Most of the current techniques in deep learning use *supervised learning* [1]. In this process, a large amount of data is collected, processed and models are trained using it. The principle is that the size of the dataset will help the model learn as well as generalize the under-lying structures in the data. For example, to build a classifier to predict whether a cat is present in the picture or not, hundreds of images with cats and without cats are collected and labelled manually. These images and labels are then directly fed into the model and trained for several iterations. Finally, the models learns to effectively classify the images and is even generalized to work on any given image.

During training, the model adjusts its internal parameters, called *weights*, such that the difference between the model's output and the actual output, called *loss*, is minimized. The weights define the model's output given the inputs, adjusting them is analogous to adjusting knobs of a black box machine where only the inputs and outputs can be observed. Neural networks, and some other machine learning models, seek to optimise an *objective function* [1, 3]. In most cases, the objective function computes the difference between the model's output and the required or correct output for a given input, this is commonly called a loss function and the neural network seeks to minimise this function. This minimisation is analogous to descending down the slope of the objective function, to reach the local minima. The updating of parameters is usually done by computing gradients of the loss function with respect to those parameters, thereby pushing the model down the slope of the objective function. This process is known as *gradient descent* [3].

Artificial neural networks (ANNs) can be represented using a graph, which is divided into multiple layers beginning with the input layer, some hidden layers in between and finally the output layer. In a typical ANN, each node of a layer is connected with all the nodes of the next layer. This arrangement is called a *Fully connected layer* or a *dense layer*. Each node in the graph, called a neuron, consists of a linear function between the inputs, weights and biases which is followed by a non-linear function called *activation function*. This graph is formally known as a **computation graph** since each node computes an input for the next node using the previous node's output. The process of computing the output given an input, by traversing all nodes in the computation graph in order is known as *forward propagation*. In an ANN, after obtaining the outputs, the loss value is calculated which is to be used for getting the gradients. The chain rule of derivatives is used to produce gradients with respect to each of the parameters. For example, let $\frac{dJ}{dW2}$ be the derivative of the loss function with respect to the parameters of the last layer $W2$. To obtain the derivative of loss function with respect to parameters of the previous layer $W1$, we can make use of the chain rule as follows: $\frac{dJ}{dW1} = \frac{d}{dW1} \cdot \frac{dJ}{dW2}$

This can extended to all the layers by computing gradients starting from the output layer and traversing backwards, this process is known as backward propagation or *backprop*. The parameters are then updated by adding the computed gradients to them, this is called gradient descent. The gradients usually scaled down by a factor known as *learning rate*. Extensive research has been conducted

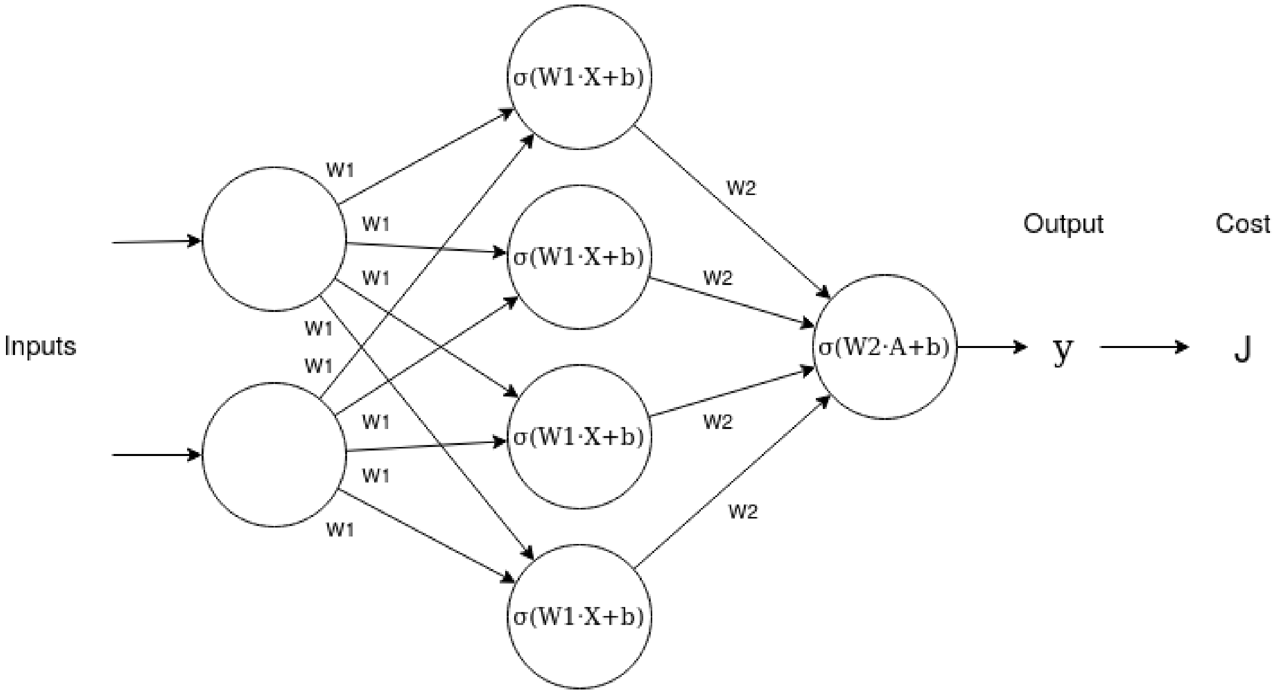

Figure 1: This figure shows a high-level overview of the computation graph of a basic neural network with 2 inputs, a hidden layer with 4 neurons and one output neuron. Each input is sent to all hidden neurons, with weights multiplied and biases added, where the neurons compute the non-linear activations A which are sent to the output layer. σ is the activation function, it can be a different function for each layer. Note that the weights W_1 and W_2 are not shared between the connections, each connection has its own weights which are represented collectively in a matrix for each layer. After obtaining the outputs, the cost function J is computed by comparing the actual output y and predicted output $\hat{y}$. The gradients of the cost function are then computed by traversing the same computation graph in the opposite direction. Gradients are computed in the following order: $\frac{dJ}{d\hat{y}}$, $\frac{dJ}{dW_2}$, $\frac{dJ}{db_2}$, $\frac{dJ}{dW_1}$, $\frac{dJ}{db_1}$.

in this area to find the optimal method of gradient descent. In summary the steps involved in training a neural network are as follows.

1. Compute the outputs or predictions $\hat{y}$ by forward propagation.
2. Calculate the loss value e using the cost function $J(y, \hat{y})$.
3. Compute gradients of the loss with respect to each and every parameter - dW, db, etc.
4. Apply gradient descent to update the parameters.
5. Repeat the above steps until the loss converges to a low value.

1.2 The Need for Faster Training Methods

DNNs are being used in many products recently, but most of them are developed by highly-funded companies and organisations. One of the reasons for this is the amount and scale of resources required to train them. DNNs can have hundreds of millions of parameters and they are trained with millions of data points, such scale required a huge amount of computation power. Even with a large amount of computation power, some models can take days or even weeks to train. This is a major problem with DNNs that needs to be solved. In this report, we look at a fast training method which uses singular value decomposition to reduce the time complexity of training.

2 Literature Review

DNNs with the back-propagation algorithm for training them, as described earlier, were introduced in the 1980s [4, 5]. DNNs found many applications including document recognition [6, 7] but it was evident that neural networks require large amount of computation. Hence, work began on making neural networks simpler and faster. One of the methods was Dynamic Node Creation (DNC) [8] that solves the issue by adding more neurons sequentially until the model reaches convergence. LeCun et al. in 1990 introduced *optimal brain damage* that removes unimportant weights using an iterative second-derivative based algorithm. Several other improvements were proposed for DNNs around the same time period [9, 10, 11]. All of these techniques focused on making DNNs simpler either by removing parameters or making the network smaller, they were not focused on improving the training time since some of the techniques involve expensive computation.

In recent times, Graphics Processing Units (GPUs) are used to accelerate training of DNNs by making use of their ability to process large matrices quickly [12]. Another method introduced in recent times is using a cluster of computers to distribute and divide the task of training DNNs, these types of DNNs are named Distributed DNNs [13, 14]. Although both of these techniques improve the training time significantly, they require specialised hardware or large number of computers.

Even with these techniques available for decreasing the training time of DNNs, they have their drawbacks and it does not diminish the fact that DNNs require a number of large matrix multiplications which accounts for most of the time taken in training of DNNs. Thus, there is a need for an approach which deals with this fundamental fact of training DNNs, this approach can then be used along with the existing techniques.

Singular Value Decomposition (SVD) is a popular technique in linear algebra used to decompose a matrix into singular value, it is described in detail in subsection 3.2. One of the widely used methods to compute SVD of a matrix was given by Golub et al. in 1970 [15].

Using Singular Value Decomposition (SVD) to accelerate the training of DNNs was introduced by Cai et al. [16]. In this report, we review the implementation of a DNN from scratch and the SVD training algorithm given by Cai et al, along with evaluation of the fast training algorithm.

3 Report

3.1 Building a Neural Network

3.1.1 The Basics

As described earlier, the basic building block of an ANN is a neuron consisting of weight matrix W, bias vector b and an activation function σ. Given an input vector X, the output of the neuron is calculated as

$$A = \sigma(W \cdot X + b)$$

The input X can be a matrix of inputs with each column being one input vector.

$$X = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x_1 & x_2 & \dots & x_n \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

For a fully connected or dense layer of input vector length m and output vector length n, the weight matrix has dimensions $n \times m$. (This is done to facilitate direct matrix multiplication in the linear step, instead of dot product)

$$W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix}$$

The bias vector has length n

$$b = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

The following demonstrates the linear step in a layer of input vector length m and output vector length n, with p input vectors.

$$Z_{n\times p} = W_{n\times m} \times X_{m\times p} + b_{n\times 1}$$

$$Z = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix} \times \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1p} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{mp} \end{bmatrix} + \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

The bias vector will be added to all columns of the resultant of the matrix multiplication, through broadcasting. The activation function is then applied element-wise to Z, to obtain the final output of this layer A. This output A is then passed on to the next layer where it is considered as the input. Let the output vector length of the next layer be q. The forward pass in this layer is calculated as follows. Note that the superscript $[l]$ denotes that the parameters or output belongs to layer l.

$$Z^{[2]}_{q\times p} = W^{[2]}_{q\times n} \times A^{[1]}_{n\times p} + b^{[2]}_{q\times 1}$$

3.1.2 Activation Functions

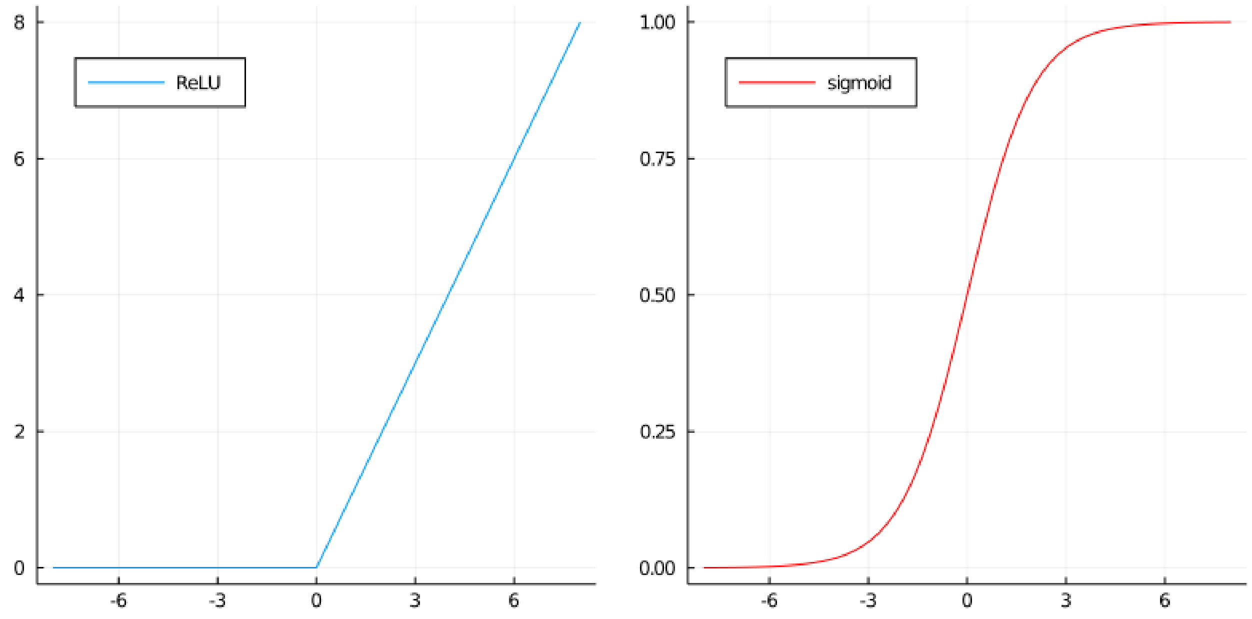

Figure 2: The ReLU and sigmoid activation function

Activation functions introduce non-linearity into the layers by transforming the linear output Z into non-linear A. In theory, the activation function can be any *continuously differentiable* function (though in practice, non-differentiability at some points can be tolerated). This means that the activation function can even be the identity function, which will produce a linear output. Linear activations are not ideal for a neural network, since multiple linear layers stacked together can be converted to an equivalent single-layer network.

$$W_3 * (W_2 * (W_1 * X)) = (W_3 * W_2 * W_1) * X$$

where the product $(W_3 * W_2 * W1)$ can be pre-computed, thereby reducing the 3-layer network into a single-layer.

The most commonly used activation function is Rectified Linear Unit (ReLU) [17, 3]. It consists of two linear parts, defined as follows.

$$relu(x) = \begin{cases} 0 & x \leq 0 \\ x & x > 0 \end{cases}$$

It can be noticed that the function is non-differentiable at $x = 0$. This does not pose a problem since an exact value of $0.00...$ in floating point is unlikely to occur. Hence, the derivative of ReLU at $x = 0$ can be arbitrarily defined to be either 0 or 1. The ReLU is most widely used in hidden layers of neural networks. There are many variations of ReLU including Exponential Linear Unit (ELU) [18], Gaussian Error Linear Unit (gelu) [19], Continuously Differentiable Exponential Linear Unit (CELU) [20] and Leaky ReLU. ReLU is not used in the final output layer of neural networks due to two reasons. In classification problems, the output needs to be in one of n classes, but the ReLU function is unbounded and cannot be partitioned. In regression problems, it is only valid when negative outputs are not needed. Thus, for the output layer, a different activation function is needed.

Binary classification problems involve classifying the input into two classes, which are represented by 0 and 1. Therefore, the final layer's output must be either 0 or 1. Having such discrete values will lead to non-differentiability. Thus, we use a function which outputs a real number in the range $(0, 1)$ which represents the probability of the input belonging to that particular class. For this purpose, the logistic *sigmoid* function is used [3]. This function is continuously differentiable and the output is always between 0 and 1. Mathematically it is defined as follows.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

For multi-class classification problems, *softmax* is used [3]. Softmax is nothing but a generalized sigmoid. We have used all three of these activation functions for our experiments.

3.1.3 Cost Function

The cost function is nothing but the loss function (described in section 1) averaged over all the training samples. The cost functions quantifies how far the predicted output is from the actual output. The simplest cost function is **Binary cross-entropy** or *log loss* which is used in binary classification problems. Mathematically, it is defined as follows.

$$bincrossentropy(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} y_i \cdot log(\hat{y_i}) + (1 - y_i) \cdot log(1 - \hat{y_i})$$

Since the predictions $\hat{y}$ is always between 0 and 1, the log terms will be negative which is why the negative sign is required. This function heavily penalizes large deviations from the actual output.

A generalisation of the binary cross-entropy, named *cross-entropy* [3] is used for multi-class classification. The outputs are generally one-hot encoded which means that each output will be vector of length m for classification into m classes. To represent class n, the value at position n in the vector will be 1 and the rest 0. Let y be the output matrix of size $m \times n$ i.e., m classes and n examples. Also let $\hat{y}$ be the predicted outputs of the same dimensions. For y and $\hat{y}$, cross-entropy is mathematically defined as follows.

$$crossentropy(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} y_{(j,i)} \cdot log(\hat{y}_{(j,i)})$$

Essentially, this function strongly penalizes a low probability predicted for the actual class.

For the purpose of this project, we implemented binary cross entropy and used an existing implementation of softmax and generalized cross entropy.

3.1.4 Backward propagation

As described earlier, backward propagation or *backprop* refers to the process of computing gradients of each and every parameter with respect to the cost function, using the chain rule of differentiation. In this section we describe the implementation details of backprop using binary classification as an example.

The first step is calculating the value of loss or cost J, which is binary cross-entropy in this case. The first differentiation involved is with respect to the output $\hat{y}$ which is a vector of 1s and 0s.

$$\frac{\partial J}{\partial \hat{y}} = -\frac{\partial}{\partial \hat{y}}(y \cdot log(\hat{y}) + (1 - y) \cdot log(1 - \hat{y})) = -\frac{y}{\hat{y}} + \frac{1 - y}{1 - \hat{y}}$$

This operation is applied element-wise to the vector to obtain the gradient vector $d\hat{y}$. This is also the gradient of the output dA for the last layer.

Consider a layer with weights W, biases b, linear output Z, final output A, activation function σ and derivative of activation function σ'. Let A_{prev} denote the outputs from the previous layer and m

be the number of training examples. The gradients are described mathematically as follows, note that these gradients are with respect to the binary cross-entropy loss.

$$\begin{aligned}
\frac{\partial J}{\partial Z} &= \frac{\partial J}{\partial A} * \sigma'(Z) \\
\frac{\partial J}{\partial W} &= \frac{1}{m}\left(\frac{\partial J}{\partial\, Z} \cdot A^T_{prev}\right) \\
\frac{\partial J}{\partial b} &= \frac{1}{m}\sum \frac{\partial J}{\partial\, Z} \\
\frac{\partial J}{\partial A_{prev}} &= W^T \cdot \frac{\partial J}{\partial Z}
\end{aligned}$$

Note that this computation requires the use of intermediate outputs Z and A obtained during forward propagation, this implies that they need to be stored in a cache preferably in a hashing-based data structure such as a dictionary. The above computation is repeated for all the layers, beginning at the output layer and ending at the input.

3.1.5 Gradient Descent

After obtaining the gradients through backprop, the parameters need to be updated using those gradients. Adding the gradients directly will introduce many inconsistencies since the data samples might push it towards different directions which leads to a rough slope down the loss curve. To solve this, a learning rate is used which scales down the gradients leading to a smoother descent. The learning rate is denoted by η. Thus, the parameter update is defined as follows.

$$\begin{aligned}
W &= W - \eta \cdot \frac{\partial J}{\partial W} \\
b &= b - \eta \cdot \frac{\partial J}{\partial b}
\end{aligned} \tag{1}$$

This type of gradient descent works fairly well, but it has a few issues. The descent can be very irregular (sideways descent) and it can be slow during the initial steps. To solve this issues, the ADAM (Adaptive moment estimation) optimiser was introduced by Kingma et al. [21]. The ADAM optimiser combines the principles of momentum and RMSprop [3] to obtain the following.

$$\begin{aligned}
V_{dW} &= \frac{\beta_1 V_{dW} + (1-\beta_1)\cdot \frac{\partial J}{\partial W}}{1-\beta_1^t} \\
V_{db} &= \frac{\beta_1 V_{db} + (1-\beta_1)\cdot \frac{\partial J}{\partial b}}{1-\beta_1^t}
\end{aligned}$$

$$\begin{aligned}
S_{dW} &= \frac{\beta_2 S_{dW} + (1-\beta_2)\cdot \frac{\partial J}{\partial W}^2}{1-\beta_2^t} \\
S_{db} &= \frac{\beta_2 S_{db} + (1-\beta_2)\cdot \frac{\partial J}{\partial b}^2}{1-\beta_2^t}
\end{aligned}$$

$$W = W - \eta \cdot \frac{V_{dW}}{\sqrt{S_{dW} + \Sigma}}$$
$$b = b - \eta \cdot \frac{V_{db}}{\sqrt{S_{db} + \Sigma}}$$

Here, β_1 and β_2 are hyper-parameters that need to be manually specified. Σ is a small nonnegative value introduced to prevent divide-by-zero errors. In this project, we have implemented gradient descent as in (1) and used an existing implementation of ADAM optimiser.

3.1.6 Parameter Initialization

One of the issues faced in DNNs is the problem of vanishing and exploding gradients. If the weights are too small (close to zero), the activations A will be even smaller and as it proceeds through the network, the activations will keep becoming smaller. Finally, the gradient computed will also be very small values which essentially stops the gradient descent. If the weights are too large (more than 1), the activations will be large and some activation functions will become saturated leading to a zero gradient. To prevent this, appropriate parameter initialisation is extremely important. Generally, weights are initialised from a normal distribution of random numbers and biases are initialised to zero [3]. To reduce the vanishing or exploding gradients further, several initialisation methods are used. Xavier et al. [22] proposed what is known as the Xavier or Glorot initialisation. One of the forms of Xavier initialisation is given below ($n^{[l]}$ denotes the number of neurons in layer l, randn is function to randomly generate a $m \times n$ matrix with values from a normal distribution).

$$W^{[i]} = randn(n^{[l]}, n^{[l-1]}) \cdot \sqrt{\frac{1}{n^{[l-1]}}}$$

We implemented the above described initialisation for this project.

3.1.7 L1 and L2 Regularisation

DNNs perform very well at fitting the given data, but this can lead to over-fitting where the model learns the input data too well which leads to it performing badly on unseen data (or test data). Regularisation is a collection of techniques designed to improve accuracy on the test set by sacrificing some accuracy on the training set. Regularisation is applied in the cost function as a function Ω of the parameters, collectively denoted by θ [3].

$$\tilde{J}(y, \hat{y}, \theta) = J(y, \hat{y}, \theta) + \lambda \cdot \Omega(\theta)$$

Two of the widely used regularisation techniques are L1 and L2 norms. Given a vector x with i components, the p-norm is defined as follows.

$$||x||_p = \left(\sum_i |x_i|^p \right)^{1/p}$$

The 2-norm or L2-norm, which is simply the Euclidean distance - distance between two points in space, is defined as follows.

$$||x||_2 = \sqrt{\left(\sum_i x_i^2\right)} = \sqrt{x_1^2 + x_2^2 + \ldots + x_i^2}$$

The 1-norm or L1-norm is commonly called the taxicab distance since it is the distance between two points such that only a single dimension is travelled at once.

$$||x||_1 = \sum_i |x_i| = |x_1| + |x_2| + \ldots + |x_i|$$

L2-norm is the most commonly used regularisation technique. The loss function is penalised with the L2-norm of the weights which leads to smaller weights, hence it is also called *weight decay* [3].

$$\tilde{J}(y, \hat{y}, W_1, W_2, ..., W_l) = J(y, \hat{y}) + \frac{\lambda}{2 \cdot m} \sum_{i=1}^{l} ||W_i||_2$$

In this project, we implemented L2-norm but it lead to a very high increase in the training time, which was due to the auto-differentiator computing gradients on more parameters. This was against the final purpose of our project which is to reduce training time. Thus, L2 regularisation was not used in the final models.

3.1.8 Hyper-parameters

DNNs have thousands or sometimes millions of parameters that are tuned during training. These parameters are the weights and biases of the model. There exist more parameters, which are not learnt during training, that need to be tuned to achieve good results. These parameters are called hyper-parameters since they are external to the model's parameters. Some hyper-parameters include number of layers, number of neurons in each layer, choice of activation functions, learning rate η, optimiser parameters (β_1 and β_2 for ADAM optimiser) and regularisation parameters λ. Hyper-parameters need to be tuned and models have to be trained again to determine their affect. Algorithms have been developed to automatically search the best set of hyper-parameter values but this can increase the training time exponentially.

3.2 Singular Value Decomposition

The Singular Value Decomposition (SVD) [15, 23] of a matrix X of dimensions $m \times n$ is given by:

$$X_{m \times n} = U_{m \times m} \Sigma_{m \times n} V_{n \times n}^T$$

where U and V are square orthogonal matrices and Σ is a diagonal matrix of dimensions $m \times n$

$$U^T U = I_d$$

$$V^T V = I_n$$

Some additional notes:

1. D is not necessarily square.
2. The SVD can be applied to any matrix.
3. SVD is different from the eigen value decomposition of a matrix.

Eigenvalue decomposition of a matrix is defined as follows. First, it exists for a matrix X if and only if X is square and the eigenvectors form a base in the matrix dimension space. If that's the case, then one can write:

$$X = Q \Lambda Q^T$$

Where Q is the matrix of the eigenvectors and Λ is a diagonal matrix of the eigenvalues of X - here, Λ is square. **SVD is a generalization of eigenvalue decomposition** since it can be applied to any matrix.

3.2.1 The Basics

The idea behind SVD is to take a high dimensional, highly variable set of data points and reducing it to a lower dimensional space that exposes the substructure of the original data more clearly and orders it from most variation to the least.

In other terms the singular value decomposition of a matrix A is the factorisation of A into the product of three matrices $A = U \Sigma V^T$ where the columns of U and V are orthonormal and the matrix Σ is diagonal, but rectangular, with positive real entries. Those positive real entries are the square roots of eigenvalues from both $A^T A$ and AA^T, which are called **singular values** of A denoted by $\sigma_1, \sigma_2, ..., \sigma_r$ where r is the rank of Σ. The columns of U are eigenvectors of AA^T and columns of V are eigenvectors of $A^T A$.

3.2.2 Applications

1. **Data Compression and Reduction**: SVD can applied to problems where very large matrices are involved such as image processing. For data compression purposes, SVD is first applied to a large matrix $A_{m \times n}$ and the resultant $U_{m \times m}$, $\Sigma_{m \times n}$ and $V^T_{n \times n}$ matrices are truncated to keep only the first l singular values so that three smaller matrices $U_{m \times l}$, $\Sigma_{l \times l}$ and $V^T_{l \times n}$ are obtained. The

truncated matrices can then be used for transferring across a bandwidth-limited place (such as a satellite image being transferred from the satellite to Earth) or the operations can be re-written in terms of the decomposed matrices (this is demonstrated in subsection 3.3).

2. **Noise Reduction**: Similar to the usage described in the previous application, truncated SVD can be used to reduce noise in images. SVD can be applied to a noisy image and then reconstructed using some l singular values to a de-noised image output. As l in increased, the image becomes more noisy. This is due to the fact that truncated SVD captures the most important parts of the image which does not usually include the noisy portions.

3.3 Accelerating Training using SVD

3.3.1 The Basics

The layers of a DNN have weights which are very high-dimensional, sometimes even hundreds or thousands of dimensions. Even with efficient algorithms, it nonetheless has to compute thousands or millions of multiplications. To reduce the number of multiplications, we use SVD to decompose a $m \times n$ matrix into three matrices and then select the first l singular values, this is named **Truncated SVD**, to effectively reduce the number of dimensions to operate upon. The following equations describe the decomposition of a $m \times n$ weight matrix W using truncated SVD by keeping the first l singular values.

$$\begin{aligned} W_{m\times n} &= U_{m\times m} \cdot \Sigma_{m\times n} \cdot V^T_{n\times n} \\ &\approx U_{m\times l} \cdot \Sigma_{l\times l} \cdot V^T_{l\times n} \\ &= W_{1_{m\times l}} \cdot W_{2_{l\times n}} \end{aligned}$$

Thus a $m \times n$ weight matrix is decomposed into two smaller weight matrices.

3.3.2 Applying Truncated SVD to a DNN

According to Cai et al. [16], applying truncated SVD to the initial weight matrices does not yield good results since the singular values do not hold any significant information (they are random). Pre-training the DNN for a few steps before applying truncated SVD achieves better results since the singular values then represent the most influential weights instead of random values. Thus, the basic principle is to pre-train a DNN with the conventional methods for a small number of steps and then apply truncated SVD to the pre-trained weights. A fraction of the total steps to be trained for is used for pre-training, we denote this fraction or percentage by φ.

Training of the SVD based model (SVD-DNN) is described in the next section. After training the model to convergence, it can be converted back to a conventional DNN by simply multiplying the weight matrices $W_{1_{m\times l}}$ and $W_{2_{l\times n}}$ to get a weight matrix compatible with the original matrix $W_{m\times n}$. Effectively, this reduces the training complexity of the model without reducing the number of parameters in the final model. In summary, the fast training method consists of the following steps.

1. Pre-train the DNN using conventional methods for a small number of steps.
2. Apply truncated SVD to pre-trained model.
3. Train the SVD-DNN via the method described in the next section.
4. Convert the SVD-DNN to a conventional DNN.

3.3.3 Training a SVD-DNN

Training is similar to a conventional DNN with one major difference - there are two weight matrices. Consider a layer of a SVD-DNN with input vector length m, output vector length n, activation function σ, loss function $J(y, \hat{y})$, weights $W_{1_{n\times l}}$ and $W_{2_{l\times m}}$. Let the number of training examples be p and actual outputs be y. A training step is outlined below.

1. The forward propagation computes:

$$\begin{aligned} Z_{n\times p} &= (W_{1_{n\times l}} \cdot (W_{2_{l\times m}} \cdot X_{m\times p} + b_{n\times 1})) \\ \hat{y}_{n\times p} &= \sigma(Z_{n\times p}) \end{aligned}$$

2. Cost calculation:

$$e = J(y_{n\times p}, \hat{y}_{n\times p})$$

3. Backprop:

$$\begin{aligned} \frac{\partial J}{\partial Z} &= \frac{\partial J}{\partial \hat{y}} \cdot \sigma'(Z_{n\times p}) \\ \frac{\partial J}{\partial W_1} &= \frac{1}{p}\left(\frac{\partial J}{\partial Z} \cdot (W_{_{l\times m}} \cdot X_{m\times p})^T\right) \\ \frac{\partial J}{\partial W_2} &= \frac{1}{p}\left(W_{1_{n\times l}}^T \cdot \frac{\partial J}{\partial Z} \cdot X_{m\times p}^T\right) \\ \frac{\partial J}{\partial b} &= \frac{1}{p}\sum \frac{\partial J}{\partial Z} \\ \frac{\partial J}{\partial \hat{y}_{prev}} &= W_{2_{l\times m}}^T \cdot \left(W_{1_{n\times l}} \cdot \frac{\partial J}{\partial Z}\right) \end{aligned}$$

4. Gradient descent, with learning rate η:

$$\begin{aligned} W_1 &= W_1 - \eta \cdot \frac{\partial J}{\partial W_1} \\ W_2 &= W_2 - \eta \cdot \frac{\partial J}{\partial W_2} \\ b &= b - \eta \cdot \frac{\partial J}{\partial b} \end{aligned}$$

3.3.4 Performance Improvements

In both the forward and backward propagation steps, the number of multiplications involved in the linear step is reduced from $m \times n$ to $l \times (m + n)$.

Proof. Consider a layer of a DNN which has weights of dimensions $m \times n$. When converted to a SVD-DNN there will be two weight matrices $W1_{m \times l}$ and $W2_{l \times m}$. Computing $W_{m \times n} \cdot X$ requires $m \times n$ multiplications while computing $W1_{m \times l} \cdot (W2_{l \times m} \cdot X)$ requires $m \times l + l \times n = l \times (m + n)$ real number multiplications.

There will be a performance improvement only when $l \times (m+n) < m \times n$. This implies l should be chosen such that:

$$l < \frac{m \times n}{m + n}$$

3.3.5 Hyper-parameter Tuning

The SVD-DNN introduces new hyper-parameters to tune - the value of l (singular values to keep) for each layer and φ the percentage of pre-training steps. Collectively, the values of l for all layers are named *l-values*. It was observed experimentally that applying SVD to the final layer resulted in a high increase in loss, thus SVD is only applied to the intermediate layers. A general rule to obtain l-values is to divide the layer dimensions by powers of 2 i.e., $1/2$ or $1/4$ of the weight dimensions. We denote this scale with α. Thus, SVD-DNN adds two new hyper-parameters to tune, α and φ.

4 Results

We applied SVD-DNN to three tasks - XOR, Cat classification and the MNIST digit recognition dataset. SVD-DNN was implemented using the Flux.jl Julia library [24, 25] with Zygote.jl auto-differentiation [26].

4.1 Experiments with XOR

Here, we construct a DNN to compute bitwise XOR of two input bits. The inputs and outputs are as follows.

$$X = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \qquad Y = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Each column in X represents one input vector and the corresponding row in Y is the output. The DNN consisted of two hidden layers with dimensions $(2, 20)$, $(20, 20)$ and the output layer with dimensions

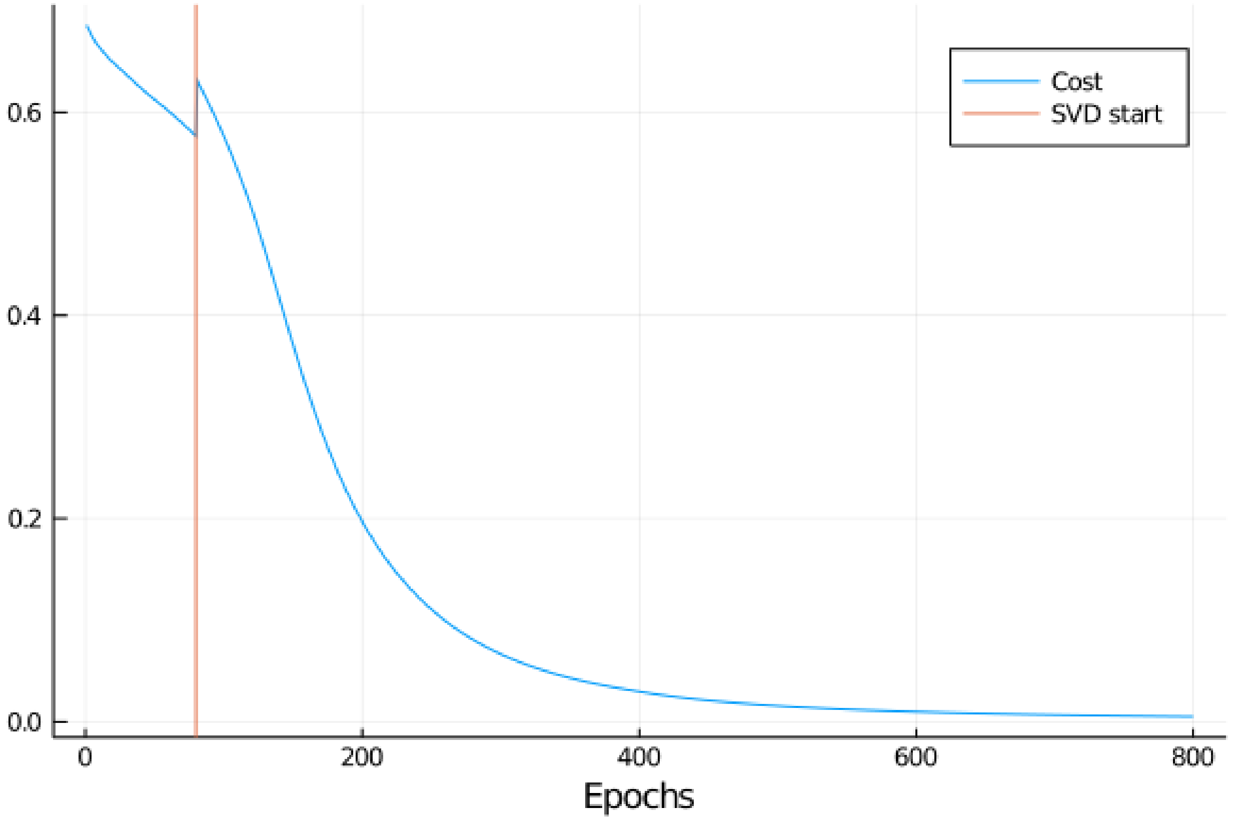

Figure 3: Variation of cost after every epoch with XOR.

$(20, 1)$. This was treated as a binary classification problem since the output consists of only two labels. Binary cross-entropy was used as the cost function. The following values of hyper-parameters were used in this experiment.

$$\alpha = 1/2$$
$$\eta = 0.01$$
$$epochs = 800$$
$$\varphi = 10\%$$

Additionally, Glorot initialiser was used for initialising weights. The following results were obtained.

Type of NN	Relative Training speed per epoch	Cost after 800 epochs
DNN	$1\times$	0.03845
SVD-DNN	$2.7\times$	0.00897

Table 1: Results on XOR

As is evident in Table 1, SVD-DNN not only improved the training speed per epoch but also led to a much lower cost after 800 epochs.

It can be seen in Figure 3 that there is a slight increase in the cost after converting the model to a SVD-DNN, this suggests that there is a some loss of information associated with truncated SVD.

4.2 Experiments with Cat vs No cat classification

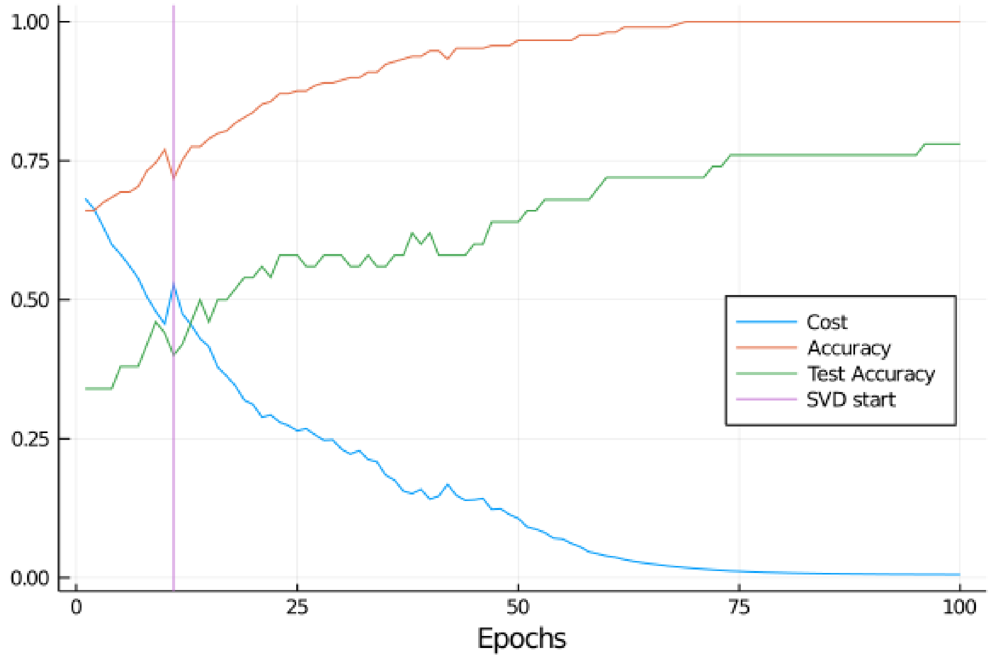

Figure 4: Variation of cost after every epoch with the Cats dataset.

A second set of experiments was conducted with the cat vs no cat binary classification dataset [1]. The inputs are RGB images of size 64×64 with the RGB values in the range $[0, 255]$ and the outputs are 0 if no cat is present in the image and 1 if there is a cat present. It consists of 209 training samples and 50 test samples. The data was pre-processed to convert the $64 \times 64 \times 3$ image matrices to 12288 length vectors and all the values were divided by 255 to standardise the inputs to the range $[0, 1]$. The DNN used consisted of a single hidden layer with 100 neurons. The following values of hyper-parameters were used in this experiment.

$$\alpha = 1/2$$
$$\eta = 0.001$$
$$epochs = 100$$
$$\varphi = 10\%$$

Binary cross-entropy was the cost function used and Glorot method was used for initialising model parameters.

[1] The dataset can be found at `https://www.floydhub.com/deeplearningai/datasets/cat-vs-noncat`

Type of NN	Relative Training speed	Final cost	Train accuracy	Test accuracy
DNN	$1\times$	0.01854	100%	72%
SVD-DNN	$1.46\times$	0.00578	100%	78%

Table 2: Summary of results obtained on Cats dataset

Table 2 shows that SVD-DNN not only improves the training speed, but also improves generalisation ability of the DNN which is evident by the higher test set accuracy. As seen with the XOR data, there is a small increase in the cost (and incidentally decrease in accuracy) when it is converted to a SVD-DNN model indicating that there is some loss of information.

4.3 Experiments with MNIST Digit Classification

The MNIST data set, introduced by LeCun et al. [7], consists of images of handwritten digits. There are a total of $60,000$ training images and $10,000$ testing images. The images are gray-scale and are of size 28×28 pixels. Pre-processing involves flattening the image matrices into 784 length vectors and one-hot encoding the outputs to make the outputs a vector of length 10. Thus, the problem is a multi-class classification problem with 10 classes. For these experiments, we used cross-entropy as the cost function and ADAM optimiser for gradient descent. A 3-layer DNN was used with 512 and 256 neurons in the hidden layers respectively, the output layer had 10 neurons. Thus, the model dimensions were $(784, 512)$, $(512, 256)$ and $(256, 10)$. The following values of hyper-parameters were used in these experiments (multiple values indicate different sets of experiments). Note that a φ value of 100% indicates that it is a simple DNN.

$$\begin{aligned}
\eta &= 3 \times 10^{-4} \\
\beta_1 &= 0.9 \\
\beta_2 &= 0.999 \\
batch\ size &= 2048 \\
epochs &= 20 \\
\alpha &= 1/2,\ 1/4,\ 1/8 \\
\varphi &= 10\%,\ 0\%,\ 100\%
\end{aligned}$$

With this dataset, as seen in Table 3, there was not a major improvement in the training time but given the increase in both training and testing accuracy along with the speed-up it is worthwhile to use a SVD-DNN. Table 3 also shows that training a SVD-DNN without pre-training leads to noticeably lower accuracy.

Type of NN	φ	Relative Training speed	Final cost	Train accuracy	Test accuracy
DNN	100%	$1.000\times$	0.04781	98.749%	97.636%
SVD-DNN	10%	$1.195\times$	0.03989	98.878%	97.734%
SVD-DNN	0%	$1.253\times$	0.05691	98.363%	97.254%

Table 3: Summary of results obtained on MNIST dataset with $\alpha = 1/2$.

A second set of experiments with the MNIST dataset was performed to quantify the effect of changing the scaling factor α. Figure 5 shows that a α factor of $1/4$ or $1/8$ negatively impacts the loss and accuracy which leads to slower convergence and lower final accuracy. An α of $1/2$ lets the training proceed as normal (with the speed up) leading to convergence after 20 epochs.

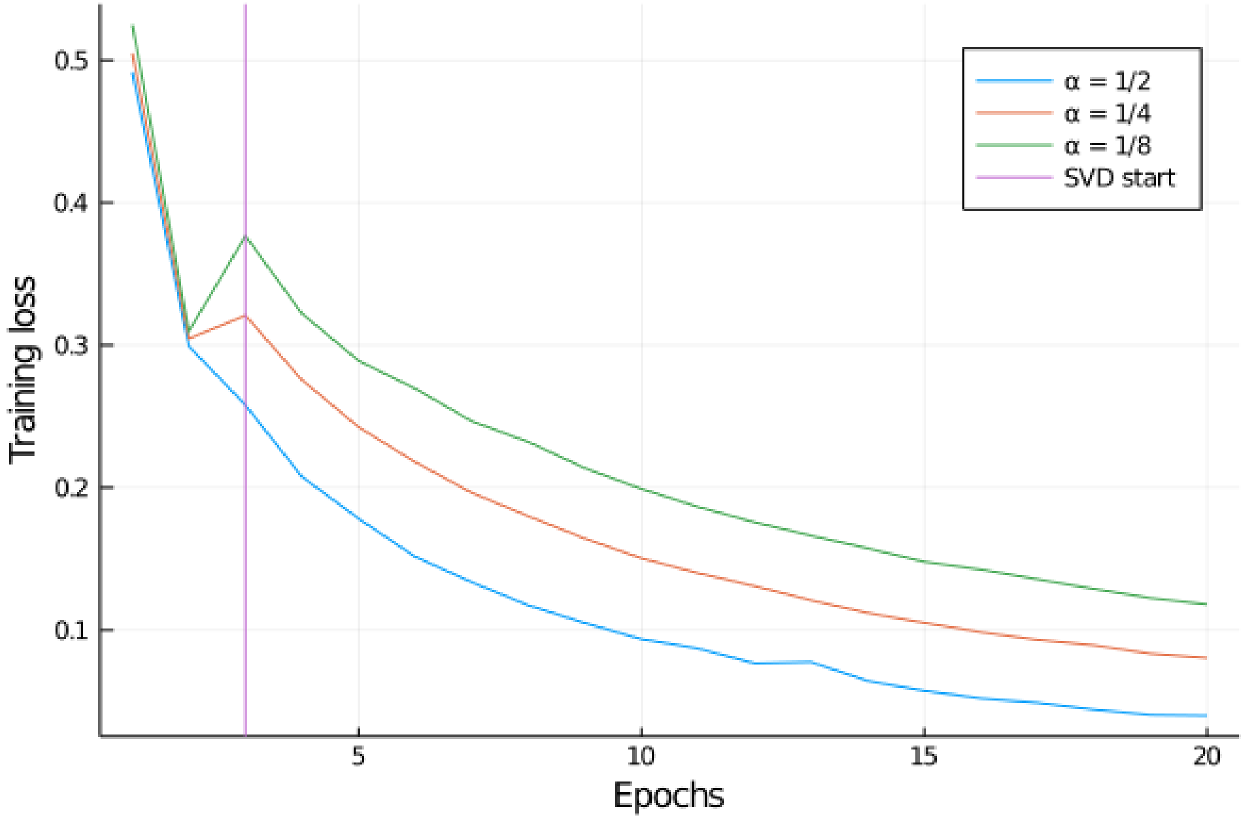

Figure 5: Effect of changing the α factor on training loss with the MNIST dataset.

Experimentally it was also found that SVD-DNN can negatively impact the performance if the weight dimensions and/or number of layers are low (for the given task). With a 1-hidden layer network (with 50 neurons) on the MNIST task, it was found that SVD-DNN reduced the accuracy significantly although the conventional DNN was also not able to achieve high accuracy indicating that it was under-powered for the task at hand.

5 Conclusions

In this report, we have reviewed the working of a Deep Neural Network (DNN) along with all the details and some enhancements. Following that, we experimented with a method to accelerate the training of DNNs using truncated SVD. Experimental results proved that the SVD-DNN not only improved training times but also improved accuracies on relatively large DNNs (relative to the task at hand). The two hyper-parameters can be safely left to their usual values, $\alpha = 1/2$ and $\varphi = 10\%$, thereby not increasing the number of hyper-parameters to be tuned. Thus, the truncated SVD method can be used for any (fully connected) DNN which has to train on a large amount of data for some significant time.

6 Future Work

The following describes some future prospects for SVD-DNNs.

1. Experimenting on larger DNNs with large amount of data.
2. Expanding SVD-DNN to be applicable to other types of NNs including Recurrent Neural Networks (RNNs), Convolutional Neural Networks (CNNs) and others.
3. Document the effects of hyper-parameters α and φ through more robust experiments.
4. Evaluate the speed-up achieved when training SVD-DNNs on Graphics Processing Units (GPUs) and other accelerators.

References

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

[2] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989.

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. Chapters 6, 7 and 8.

[4] Yann Lecun. A theoretical framework for back-propagation. In D. Touretzky, G. Hinton, and T. Sejnowski, editors, *Proceedings of the 1988 Connectionist Models Summer School, CMU, Pittsburg, PA*, pages 21–28. Morgan Kaufmann, 1988.

[5] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, Oct 1986.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[7] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4):541–551, Dec 1989.

[8] T. Ash. Dynamic node creation in backpropagation networks. In *International 1989 Joint Conference on Neural Networks*, pages 623 vol.2–, 1989.

[9] Geoffrey E. Hinton and Drew van Camp. Keeping neural networks simple. In Stan Gielen and Bert Kappen, editors, *ICANN '93*, pages 11–18, London, 1993. Springer London.

[10] H. White. Learning in artificial neural networks: A statistical perspective. *Neural Computation*, 1(4):425–464, Dec 1989.

[11] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In S. J. Hanson, J. D. Cowan, and C. L. Giles, editors, *Advances in Neural Information Processing Systems 5*, pages 164–171. Morgan-Kaufmann, 1993.

[12] Zhongwen Luo, Hongzhi Liu, and Xincai Wu. Artificial neural network computation on graphic process unit. In *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, volume 1, pages 622–626 vol. 1, July 2005.

[13] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, Quoc V. Le, and Andrew Y. Ng. Large scale distributed deep networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1223–1231. Curran Associates, Inc., 2012.

[14] S. Zhang, C. Zhang, Z. You, R. Zheng, and B. Xu. Asynchronous stochastic gradient descent for dnn training. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 6660–6663, 2013.

[15] Yu Zhang, Min-Qiang Li, Yu Zhong, Wei Li, Bai Sun, Dao-Yang Yu, and Jin-Huai Liu. Application of singular value decomposition for identification of liquid precursor chemicals using energy-dispersive x-ray scattering. *Instrumentation Science & Technology*, 39(1):20–33, 2011.

[16] Chenghao Cai, Dengfeng Ke, Yanyan Xu, and Kaile Su. Fast learning of deep neural networks via singular value decomposition. In Duc-Nghia Pham and Seong-Bae Park, editors, *PRICAI 2014: Trends in Artificial Intelligence*, pages 820–826, Cham, 2014. Springer International Publishing.

[17] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey Gordon, David Dunson, and Miroslav Dudík, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, volume 15 of *Proceedings of Machine Learning Research*, pages 315–323, Fort Lauderdale, FL, USA, 11–13 Apr 2011. PMLR.

[18] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs). *arXiv e-prints*, page arXiv:1511.07289, November 2015.

[19] Dan Hendrycks and Kevin Gimpel. Gaussian Error Linear Units (GELUs). *arXiv e-prints*, page arXiv:1606.08415, June 2016.

[20] Jonathan T. Barron. Continuously Differentiable Exponential Linear Units. *arXiv e-prints*, page arXiv:1704.07483, April 2017.

[21] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2015.

[22] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*, 2010.

[23] Gilbert Strang. *Linear Algebra and Its Applications*. Thomson, Brooks/Cole, fourth edition, 2006. Chapter 6, pg. 367 - 370.

[24] Michael Innes, Elliot Saba, Keno Fischer, Dhairya Gandhi, Marco Concetto Rudilosso, Neethu Mariya Joy, Tejan Karmali, Avik Pal, and Viral Shah. Fashionable modelling with flux. *CoRR*, abs/1811.01457, 2018.

[25] Mike Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018.

[26] Michael Innes. Don't unroll adjoint: Differentiating ssa-form programs. *CoRR*, abs/1810.07951, 2018.