Deep Learning and Precision

Course Report

Licence d'informatique L3

Université Pierre et Marie Curie

Marion Caumartin

August 11, 2017

Contents

1 Introduction

For many years now Mankind has been trying to create an intelligent machine. For it to happen, one must find the theory to tasks Man considers simple such as image recognition. This task, that we do unconsciously every day is arduous to theorize; thus the creation of "machine learning", or more precisely "deep learning". Deep learning enables machines to identify objects on a picture without explicit programming for example.

Section 2 introduces the notion of "machine learning". Section 3 defines what "deep learning" is, presents example of neural networks and presents one of the most used training algorithm – the backpropagation algorithm based on the Stochastic Gradient Descent. Section 4 presents the limits of "deep learning" and the possible improvements to the neural networks' efficiency regarding time and memory. Section 5 defines the notion of stability of a learning algorithm. Section 6 presents a proof of the numerical stability of the Stochastic Gradient Descent when the network has no hidden layer. Section 7 presents a analysis of the forward error of a neural network.

2 Machine Learning

Many people may already have seen machine learning algorithms unconsciously. For example, ads on Google are generated through a machine learning algorithm. The principle is to identify ads that might interest people based on their search history. These algorithms are also used to produce news feeds on social networks. The aim of this section is to presents what these algorithms are.

Machine Learning is a division of Artificial Intelligence that Arthur Samuel defines in 1959 as

> "the field of study that gives computer the ability to learn without being explicitly programmed."

It allows software to predict outcomes more and more accurately without being explicitly programmed. Machine Learning consists of building an algorithm that takes inputs data and uses statistical analyses to predict the result with an acceptable accuracy. Nowadays, in image recognition, some algorithms provide less than 1% of error.

In 1997, Thomas Mitchell gives another definition of Machine Learning.

> "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

Let explain this definition with an example. Let assume that someone wants a program to predict the traffic pattern of an intersection (task T) using a machine learning algorithm. The program will need some data about past traffic patterns (experience E). If the program learned the data right, then it will predict traffic pattern better (performance measure P).

Many problems such as image recognition or natural language processing cannot always be solved perfectly. The theory behind such problems is not always known and is often very complex. Machine Learning helps to solve problems that cannot be solved by numerical means alone as such algorithms do not need to be explicitly programmed.

There are several types of machine learning algorithm. Supervised learning and unsupervised learning are two common types of machine learning algorithm.

- **Supervised learning** : in a supervised algorithm, the output of training examples is known. The program class data depends on outputs.
- **Unsupervised Learning** : in an unsupervised algorithm, the output of training examples is not known. The program should find patterns to classify data.

Next, we will only concentrate on supervised learning.
[1, 2, 3]

3 Deep Learning

3.1 What is Deep Learning ?

Deep Learning is a part of Machine Learning that has been introduced to move Machine Learning closer to one of its goals: Artificial Intelligence. The principle of Deep Learning is to build artificial neural networks. This notion was first thought of in the 50s and has been inspired by the way animals used their visual cortex.

Neural networks are multilayer networks made of one input, one output, and at least one hidden layers. These layers may be assimilated to vectors or matrices whose elements represent neurons (or units) of the network. The neurons of different layers are linked together through weighted lines. The weights of these links are called networks weights. Some might be nulls and as such are not represented.

The training of a network consists in finding weights for which the network send back the waited value most of the time. It is made on a preset training set. For supervised learning, every element of the training unit is a couple (x, t) where x is the network's input and t the desired output. The aim of such training is to minimize the difference between the network's output and the targeted value t. One of the first supervised learning algorithm, and also one of the most used, is the backpropagation algorithm presented in section 3.3.

A network is said to be efficient if it great generalization ability. The generalization ability is the ability of a network to classify well data that it never learn.

The way neurons are linked together is called the network design. Two prevalent network design in supervised learning for image recognition is fully-connected neural networks (section 3.2.1) and convolutional neural networks (section 3.2.2). [4, 5]

3.2 Neural Networks

In this section are presented two common architectures for supervised learning of image recognition.

3.2.1 Multilayer Perceptrons

Multilayer perceptrons are organized in successive layers. The links can only be made between two consecutive layers, i.e. between the layers l and $l + 1$.

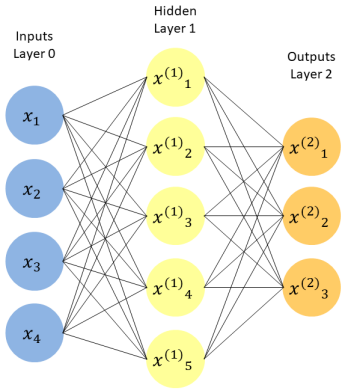

Figure 1: Multilayer Perceptrons with one hidden layer.

Multilayer perceptrons are made of an input layer, an output layer, and at least one hidden layer. Figure 1 displays a network with an input layer of four neurons, a hidden layer with five neurons and an output layer of three neurons. One may notice n_l the number of neurons of layer l.

The hidden and output layers all have inputs and outputs. In the example of Figure 1, inputs of the hidden layer are neurons of the input layer and the inputs of the output layer are neurons of the hidden layer. The outputs of the hidden layer match the values of these neurons. The outputs of the output layer match the output of the network. We notice x the vector including the values of the input layer and $x^{(l)}$ the vector including the values of layer l. As such, in our example we have $x = (x_1\ x_2\ x_3\ x_4)^T$, $x^{(1)} = (x_1^{(1)}\ x_2^{(1)}\ x_3^{(1)}\ x_4^{(1)}\ x_5^{(1)})^T$, and $x^{(2)} = (x_1^{(2)}\ x_2^{(2)}\ x_3^{(2)})^T$.

Every line between 2 neurons have a certain weight, we notice $w_{j,i}^{(l)}$ the weight between the neurons $x_i^{(l-1)}$ and $x_j^{(l)}$ and we notice $\theta^{(l)} = (w_{j,i}^{(l)})_{\substack{1 \leq i \leq n_{l-1} \\ 1 \leq j \leq n_l}}$. Each neuron of the layer l is linked to every neuron of the layer $l-1$.

Every layer l has an activation function $\varphi^{(l)} : \mathbb{R}^{n_l} \to \mathbb{R}^{n_l}$ which is nonlinear, continue and differential. The values of neurons of the layer $l+1$ are calculated this way: $x^{(l+1)} = \varphi^{(l+1)}(\theta^{(l+1)}x^{(l)})$. For the example of figure 1, we, therefore, find $x^{(2)} = \varphi^{(2)}(\theta^{(2)}\varphi(1)(\theta^{(1)}x))$.

The measurement of optimal weights (i.e. the training) of the network is made through the backpropagation algorithm. The way it works is detailed in section 3.3. [5, 6]

3.2.2 Convolutional Neural Networks

Convolutional Networks are very suitable for image recognition. In this report, they will only be considered for those tasks.

Before Convolutional Networks, define the notion of filters. A filter is applied to every pixel of a picture. For example, on a 5×5 filter: the pixel whose coordinates on the output picture are 3,3 will match the weighted sum of a 5×5 square of the input picture – whose central pixel has the coordinates 3,3 – and weights of the filter that are represented in a 5×5 square. A problem appears nonetheless for the picture's borders. There are some techniques to go around it: the first would simply need to ignore the borders; another would be to duplicate the picture's borders or to add pixel lines and columns of 0 value.

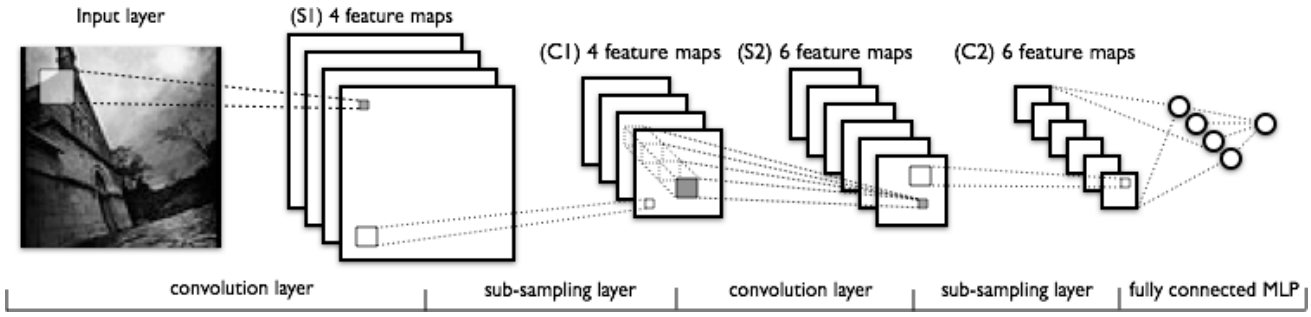

Figure 2: Typical Convolutional Neural Networks: LeNet-5

Now, about Convolutional Networks. Figure 2 presents an example of a convolutional network. It is the network LeNet-5. Filters used in this network are 5×5 filters.

Convolutional networks are made of different sort of layers :

- **Convolutional layers** : these layers simply are the application of filters on the input picture. The product of a filter is called *feature map*.
  For the first convolutional layer of the LeNet-5 network, four 5×5 filters are applied on the input picture of the network.
  For the second convolutional layer, six 5×5 filters are applied on each four feature maps. The computation of the value of pixels of the six feature maps is made as shown in Figure 3.

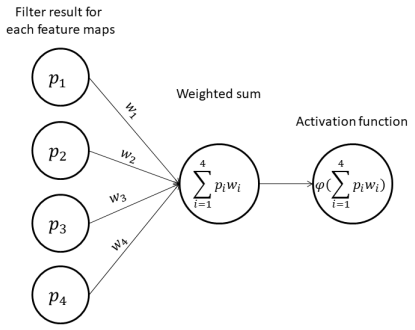

Figure 3: Compute of pixels' value of one feature map.

Each feature map has a weight - this weight is the same for each pixel of the feature map. First, we compute the weighted sum of the result of the filter on each feature maps and weights of features maps. Next, an activation function is applied to this sum. The result is the value of one pixel of the feature map associated with the applied filter.

- **Subsampling layers** : these layers reduce the size of input pictures. For instance, if the input picture is a 32×32 picture and the layer has a 2×2 subsampling region, the output image will be a 16×16 image. To apply a 2×2 subsampling region we first cut the input picture in 2×2 regions. Then we apply an $\mathbb{R}$-value function on each sample. The result is a picture twice as small as the input picture.

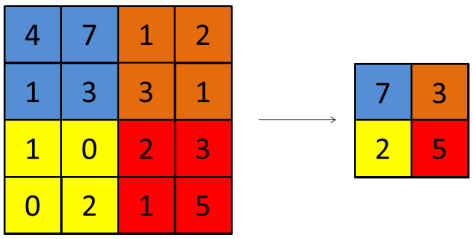

Figure 4: An example of sub-sampling: the max-pooling.

Figure 4 shows a subsampling layer that has a 2×2 sub-sampling region and applies the function $f(A) = \max_{i,j} a_{i,j}$.

- **Fully-connected layers** : the last sub-sampling or convolutional layer is linked to one or several fully-connected layers.

The training of Convolutional Neural Networks uses a modified version of the backpropagation algorithm that takes into account the sub-sampling.

Convolutional Neural Networks don't need as many weights as multilayer perceptrons and are more effective. Actually, the error made by a Convolutional Neural Network can be twice as small as the one made by multilayer perceptrons. [6, 7]

3.3 Training

As said in section 3.1, training network matches with finding weights that provide the network the ability to classify data of a training set accurately.

Here is a question one can ask: Do such weights always exist? the universal approximation theorem[1] proves that the answer is fortunately yes. It proves that a network with a finite number of neurons, at least one hidden layer and that has nonlinear activation function can approach any function.

One of the flagship algorithms in supervised learning is the backpropagation algorithm.

In this section, we consider a network that has $n \in \mathbb{N}$ hidden layers. We notice θ the set of its weights. Notations that have been introduced in section 3.2.1 are reused here. Let S be a training set. We notice m the number of elements of S. Each element of S is a pair (x, t) where x is the input of the network and t the targeted value, $S = \{z_1, \dots, z_m\} = \{(x_1, t_1), \dots, (x_m, t_m)\}$. We notice $y_\theta(x)$ the output of the network with weights θ for the input x.

Let $z = (x, t) \in S$, the loss function is defined as follows $f(\theta, z) = \frac{1}{2}\|y_\theta(x) - t\|^2$.

Training the network consists of finding the weights θ^* that minimize $f_{\text{MSE}}(\theta) = \frac{1}{m}\sum_{i=1}^{m} f(\theta, z_i)$. To this end, one can use the stochastic gradient descent.

3.3.1 Stochastic Gradient Descent

The first method to minimize f_{MSE} was to use the gradient descent. The gradient descent consists in computing the direction in weights space in which the decrease of f_{MSE} is maximum. The gradient descent can be described as follows

$$\begin{cases} \theta_0 = \theta \\ \theta_{i+1} = \theta_i - \alpha_i \nabla f_{\text{MSE}}(\theta_i) \end{cases} \tag{1}$$

The step size α_n allows adjusting the convergence speed of the algorithm, it is also called the learning rate.

In practice, the computation of ∇f_{MSE} is really time-consuming. So, we use the **Stochastic Gradient Descent**. This algorithm consists in approximating ∇f_{MSE} by $\nabla_\theta f(\theta, z)$ where z is randomly chosen in S at each iteration. The algorithm can be described as follows

$$\begin{cases} \theta_0 = \theta \\ \theta_{i+1} = \theta_i - \alpha_i \nabla_\theta f(\theta_i, z_i) \text{ with } z_i \text{ randomly chosen in } S \end{cases} \tag{2}$$

The convergence of this algorithm has been proved in cases in which: $\begin{cases} \sum \alpha_i = +\infty \\ \sum \alpha_i^2 < +\infty \end{cases}$ [8, 9].

To compute $\nabla_\theta f(\theta, z)$ one only has to compute partial derivative of f with respect to weights of the network. To this end, the backpropagation algorithm is used. [5]

3.3.2 Backpropagation

The backpropagation algorithm allows computing the partial derivatives of f with respect to the weights of layer l as a function of the partial derivatives of f with respect to the weights of layer $l + 1$.

Let θ be the weights of the network, $z = (x, t) \in S$, and y the output of the network associated with the input x.

We notice $J^{(l)}$ the Jacobian matrix of $\varphi^{(l)}$,

$E = \frac{1}{2}\|y - t\|^2$,

[1] https://en.wikipedia.org/wiki/Universal_approximation_theorem

$net_j^{(l)} = \sum_{i=1}^{n_{l-1}} w_{i,j}^{(l)} x_i^{(l-1)}$,

$e_j^{(l)} = \dfrac{\partial E}{\partial net_j^{(l)}} = \sum_{i=1}^{n_l} \dfrac{\partial E}{\partial x_i^{(l)}} \dfrac{\partial x_i^{(l)}}{\partial net_j^{(l)}}$.

$\dfrac{\partial E}{\partial w_{b,a}^{(l)}} = \sum_{i=1}^{n_l} \dfrac{\partial E}{\partial x_i^{(l)}} \sum_{j=1}^{n_l} \dfrac{\partial x_i^{(l)}}{\partial net_j^{(l)}} \dfrac{\partial net_j^{(l)}}{\partial w_{b,a}^{(l)}}$.

$\dfrac{\partial net_j^{(l)}}{\partial w_{b,a}^{(l)}} = \dfrac{\partial \sum_{i=1}^{n_{l-1}} w_{i,j}^{(l)} x_i^{(l-1)}}{\partial w_{b,a}^{(l)}} = \delta_{j,b} x_a^{(l-1)}$ with $\delta_{j,b} = \begin{cases} 1 \text{ if } j = b \\ 0 \text{ else} \end{cases}$.

So $\dfrac{\partial E}{\partial w_{b,a}^{(l)}} = x_a^{(l-1)} \sum_{i=1}^{n_l} \dfrac{\partial E}{\partial x_i^{(l)}} \dfrac{\partial x_i^{(l)}}{\partial net_b^{(l)}} = x_a^{(l-1)} \dfrac{\partial E}{\partial net_b^{(l)}} = x_a^{(l-1)} e_b^{(l)}$.

- **Output layer :** $l = n+1$

  $e_j^{(n+1)} = \sum_{i=1}^{n_{n+1}} \dfrac{\partial E}{\partial x_i^{(n+1)}} \dfrac{\partial x_i^{(n+1)}}{\partial net_j^{(n+1)}} = \sum_{i=1}^{n_{n+1}} \dfrac{\partial E}{\partial y_i} \dfrac{\partial y_i}{\partial net_j^{(n+1)}}$.

  $E = \frac{1}{2}\|y - t\|^2 = \frac{1}{2} \sum_{i=1}^{n_{n+1}} (y_i - t_i)^2$, so $\dfrac{\partial E}{\partial y_i} = y_i - t_i$.

  Moreover $\dfrac{\partial y_i}{\partial net_j^{(n+1)}} = J_{i,j}^{(n+1)}(net_j^{(n+1)})$.

  So $e_j^{(n+1)} = \sum_{i=1}^{n_{n+1}} (y_i - t_i) J_{i,j}^{(n+1)}(net_j^{(n+1)})$.

- **Hidden layers :** $1 \leq l \leq n$

  $e_j^{(l)} = \sum_{i=1}^{n_l} \dfrac{\partial E}{\partial x_i^{(l)}} \dfrac{\partial x_i^{(l)}}{\partial net_j^{(l)}}$.

  $\dfrac{\partial x_i^{(l)}}{\partial net_j^{(l)}} = J_{i,j}^{(l)}(net_j^{(l)})$ and $\dfrac{\partial E}{\partial x_i^{(l)}} = \sum_{k=1}^{n_{l+1}} w_{k,i}^{(l+1)} \dfrac{\partial E}{\partial net_k^{(l+1)}} = \sum_{k=1}^{n_{l+1}} w_{k,i}^{(l+1)} e_k^{(l+1)}$.

  So $e_j^{(l)} = \sum_{i=1}^{n_l} J_{i,j}^{(l)}(net_j^{(l)}) \sum_{k=1}^{n_{l+1}} w_{k,i}^{(l+1)} e_k^{(l+1)}$.

Hence we have $\dfrac{\partial E}{\partial w_{b,a}^{(l)}} = x_a^{(l-1)} e_b^{(l)}$ with $e_j^{(l)}$ defined by $\begin{cases} e_j^{(n+1)} = \sum_{i=1}^{n_{n+1}} (y_i - t_i) J_{i,j}^{(n+1)} (\sum_{k=1}^{n_n} w_{k,j}^{(n+1)} x_k^{(n)}) \\ e_j^{(l)} = \sum_{i=1}^{n_l} J_{i,j}^{(l)} (\sum_{p=1}^{n_{l-1}} w_{p,j}^{(l)} x_p^{(l-1)}) \sum_{k=1}^{n_{l+1}} w_{k,i}^{(l+1)} e_k^{(l+1)} \text{ for all } 1 \leq l \leq n \end{cases}$.

We notice $w_{i,j}^{(l)}(t)$ the value of $w_{i,j}^{(l)}$ at iteration t. It follows that $w_{i,j}^{(l)}(t+1) = w_{i,j}^{(l)}(t) - \alpha_t x_j^{(l-1)} e_i^{(l)}$.

We notice $\Delta w_{i,j}^{(l)}(t) = w_{i,j}^{(l)}(t) - w_{i,j}^{(l)}(t-1)$. There exists a variant of backpropagation that uses an inertia term (momentum) $\mu \in [0,1]$. The iteration becomes $w_{i,j}^{(l)}(t+1) = w_{i,j}^{(l)}(t) - (1-\mu)\alpha_t x_j^{(l-1)} e_i^{(l)} - \mu \Delta w_{i,j}^{(l)}(t)$.

Another variant of backpropagation uses weight decay λ. The iteration becomes $w_{i,j}^{(l)}(t+1) = w_{i,j}^{(l)}(t) - \alpha_t x_j^{(l-1)} e_i^{(l)} - \lambda\alpha w_{i,j}^{(l)}(t)$.

The Stochastic Gradient Descent is often run on several examples at the same time. The set of example is called a batch. If the batch size is not small compared to the number of training examples, the algorithm is called Batch Gradient Descent. [10]

4 Reduced Precision in Deep Learning

4.1 Datasets for image recognition

Most datasets are divided into two sets: a training set, and a test set. The training set is used to train a machine learning algorithm such as neural networks. The test set is used to experiment the generalisation ability of a machine learning algorithm.

4.1.1 MNIST

The MNIST dataset (Modified National Institute of Standards and Technology dataset) is a large dataset of handwritten digits. It contains $60,000$ training images and $10,000$ test images. Each image is a 28×28 normalised image. Most neural networks are tested using this dataset. If a network does not class well the MNIST dataset, it means this network is not efficient. Figure 5 shows examples of images from MNIST dataset. [11]

Figure 5: Images from MNIST dataset

4.1.2 CIFAR-10

The CIFAR-10 dataset consists of $60,000$ 32×32 colour images classified into ten classes. It contains $50,000$ training images and $10,000$ test images. Figure 6 shows the classes in the dataset, as well as ten random images from each. [12]

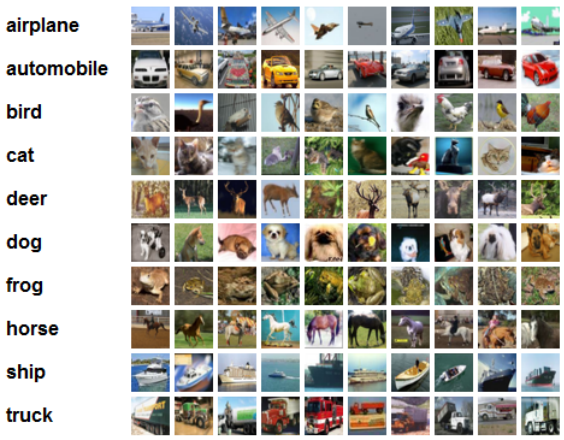

Figure 6: Images from CIFAR-10 dataset

4.1.3 SVHN

The SVHN dataset (Street View House Numbers dataset) is a real-world image dataset. It can be seen as similar in flavor to MNIST. It contains over $600,000$ of 32×32 colour images, each image shows a digit from a house number. The SVHN dataset is obtained from house numbers in Google Street View images. It consists of $73,257$ training images, $26,032$ test images, and $531,131$ additional images. Figure 7 shows examples of images from SVHN dataset. [13]

Figure 7: Images from SVHN dataset

4.1.4 WordNet

The WordNet dataset is not used in image recognition. It is a large lexical dataset of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. The WordNet dataset contains $117,000$ synsets. [14]

4.1.5 ImageNet

The ImageNet dataset is an image dataset organized according to the WordNet hierarchy. The ImageNet dataset contains on average $1,000$ images to illustrate each synset of the WordNet dataset. Figure 8 shows examples of images from ImageNet dataset. [15]

Figure 8: Images from ImageNet dataset

4.2 Test of some networks using single and double precision

Networks described bellow are used to present some accuracy results.

- **Network 1** : This network is a Multilayer Perceptrons that has 784 inputs and four hidden layers.
  The first layer has 5 units and its activation function is the *hyperbolic tangent* function.
  The second layer has 20 units and its activation function is the *rectified linear* function which is defined as follow :

$$\begin{aligned} \mathbb{R} &\to \mathbb{R}_+ \\ x &\mapsto \max(x, 0). \end{aligned}$$

  The third hidden layer has 30 units and its activation function is the *hyperbolic tangent* function.
  The fourth hidden layer has 40 units and its activation function is the *rectified linear* function.
  The output layer has 10 units and its activation function is the *softmax* function which is defined as follow :

$$\begin{aligned} \mathbb{R}^n &\to \mathbb{R}^n \\ (x_i)_{1\leq i\leq n} &\mapsto \left(\frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \right)_{1\leq i\leq n}. \end{aligned}$$

  The network is trained using the Stochastic Gradient Descent with a batch size of 100, a constant learning rate $\alpha = 0.01$, a weight decay $\lambda = 0.0005$ for two firsts layers and $\lambda = 0.00005$ for other layers., and a momentum sequence defined as follow : $\begin{cases} \mu_0 = 0.5 \\ \mu_t = \min(\mu_0 + 0.049t, 0.99) \end{cases}$.

- **Network 2** : This network is a Multilayer Perceptrons that has 784 inputs and four hidden layers.
  The first layer has 10 units and its activation function is the *hyperbolic tangent* function.
  The second layer has 40 units and its activation function is the *rectified linear* function.
  The third hidden layer has 60 units and its activation function is the *hyperbolic tangent* function.
  The fourth hidden layer has 80 units and its activation function is the *rectified linear* function.
  The output layer has 10 units and its activation function is the *softmax* function.
  The network is trained using the Stochastic Gradient Descent with a batch size of 100, a constant learning rate $\alpha = 0.01$, a weight decay $\lambda = 0.0005$ for two firsts layers and $\lambda = 0.00005$ for other layers, and a momentum sequence defined as follow : $\begin{cases} \mu_0 = 0.5 \\ \mu_t = \min(\mu_0 + 0.049t, 0.99) \end{cases}$.

- **Network 3** : This network is a Multilayer Perceptrons that has 784 inputs and four hidden layers.
  The first layer has 50 units and its activation function is the *hyperbolic tangent* function.
  The second layer has 200 units and its activation function is the *rectified linear* function.
  The third hidden layer has 300 units and its activation function is the *hyperbolic tangent* function.
  The fourth hidden layer has 400 units and its activation function is the *rectified linear* function.
  The output layer has 10 units and its activation function is the *softmax* function.
  The network is trained using the Stochastic Gradient Descent with a batch size of 100, a constant learning rate $\alpha = 0.01$, a weight decay $\lambda = 0.0005$ for two firsts layers and $\lambda = 0.00005$ for other layers, and a momentum sequence which is defined as follow : $\begin{cases} \mu_0 = 0.5 \\ \mu_t = \min(\mu_0 + 0.049t, 0.99) \end{cases}$.

- **Network 4** : This network is a Multilayer Perceptrons that has 784 inputs and four hidden layers.
  The first layer has 500 units and its activation function is the *hyperbolic tangent* function.
  The second layer has 2000 units and its activation function is the *rectified linear* function.
  The third hidden layer has 3000 units and its activation function is the *hyperbolic tangent* function.
  The fourth hidden layer has 4000 units and its activation function is the *rectified linear* function.
  The output layer has 10 units and its activation function is the *softmax* function.
  The network is trained using the Stochastic Gradient Descent with a batch size of 100, a constant learning rate $\alpha = 0.01$, a weight decay $\lambda = 0.0005$ for two firsts layers and $\lambda = 0.00005$ for other layers, and a momentum sequence defined as follow : $\begin{cases} \mu_0 = 0.5 \\ \mu_t = \min(\mu_0 + 0.049t, 0.99) \end{cases}$.

- **Network 5** : This network is a Multilayer Perceptrons that has 784 inputs and one hidden layers.
  The hidden layer has 500 units and its activation the sigmoid function which is defined as follow :

$$\begin{aligned} \mathbb{R} &\to]0,1[\\ x &\mapsto \frac{1}{1+e^{-x}}. \end{aligned}$$

  . The output layer has 10 units and its activation function is the *softmax* function.
  The network is trained using the Batch Gradient Descent with a batch size of 10000.

- **Network 6** : This network is a Multilayer Perceptrons that has 784 inputs and two hidden layers.
  The first hidden layer has 500 units and its activation function is the *rectified linear* function.
  The first hidden layer has 1000 units and its activation function is the *rectified linear* function.
  The output layer has 10 units and its activation function is the *softmax* function.
  The network is trained using the Stochastic Gradient Descent with a batch size of 100, a constant learning rate $\alpha = 0.01$, and a momentum sequence which defined as follow : $\begin{cases} \mu_0 = 0.5 \\ \mu_t = \min(\mu_0 + 0.049t, 0.99) \end{cases}$.

- **Network 7** : This network is a Multilayer Perceptrons that has 784 inputs and two hidden layers.
  The first hidden layer has 500 units and its activation function is the *rectified linear* function.
  The first hidden layer has 1000 units and its activation function is the *rectified linear* function.
  The output layer has 10 units and its activation function is the *softmax* function.
  The network is trained using the Stochastic Gradient Descent with a batch size of 100, a constant learning rate $\alpha = 0.01$, a weight decay $\lambda = 0.00005$, and a momentum sequence defined as follow : $\begin{cases} \mu_0 = 0.5 \\ \mu_t = \min(\mu_0 + 0.049t, 0.99) \end{cases}$.

- **Network 8** : This network is a Multilayer Perceptrons that has 784 inputs and two hidden layers.
  First, the *maxout* units. Let l be a layer that uses maxout units. We notice y the output of this layer, x the input, and w the weights. Let n and m be the length of y and x respectively. y is defined as follow : $y_i = \max_{j=1}^{m}(w_{i,j}x_j + b_i)$.
  The first hidden layer has 240 *maxout* units.
  The second hidden layer has 240 *maxout* units.
  The output layer has 10 units and its activation function is the *softmax* function.
  The network is trained using the Stochastic Gradient Descent with a batch size of 100, a learning rate sequence defined as follow : $\begin{cases} \alpha_0 = 0.01 \\ \alpha_t = \max(\alpha_0 e^{-1.000004 \times t}, 0.000001) \end{cases}$, and a momentum sequence defined as follow : $\begin{cases} \mu_0 = 0.5 \\ \mu_t = \min(\mu_0 + 0.0008t, 0.7) \end{cases}$.

- **Network 9** : This network is a Convolutional Network that has 784 inputs and two convolutional layers, two subsampling layers and one fully-connected layer.
  The first convolutional layer has 64 feature maps, a 5×5 filter and its activation function is the rectified linear function.
  The first subsampling layer uses a 2×2 subsampling region and the max function.
  The first convolutional layer has 64 feature maps, a 5×5 filter and its activation function is the rectified linear function.
  The first subsampling layer uses a 2×2 subsampling region and the max function.
  The fully-connected layer correspond to the output layer. It has 10 units and its activation function is the *softmax* function.
  The network is trained using the Stochastic Gradient Descent with a batch size of 100, a constant learning rate $\alpha = 0.01$, a weight decay $\lambda = 0.00005$, and a momentum sequence defined as follow : $\begin{cases} \mu_0 = 0.5 \\ \mu_t = \min(\mu_0 + 0.049t, 0.99) \end{cases}$.

The code is available in Appendix A. Tests were made on the MNIST dataset and using a Tesla K40c GPU. Table 1 presents the accuracy of networks described above when training in single and in double precision. Accuracy is measured using the testset of the MNIST dataset.

Network	Single precision	Double precision
1	11.1%	11.03%
2	8.16%	7.9%
3	5.3%	5.17%
4	5.13%	5.27%
5	1.94%	1.85%
6	1.89%	1.8%
7	1.65%	1.55%
8	1.18%	1.24%
9	1.19%	1.07%

Table 1: Accuracy when training networks in single and double precision

The difference between single and double precision floating point is not significant. However, training networks in single precision floating point is faster than in double precision floating point.

4.3 Research on reduced precision in deep learning

Deep Learning allows computers to perform tasks that are simple for us but hard to theorize such as image recognition or speech recognition. Although effective, most neural networks involve thousands or millions of weights and are time-consuming. Training may take several days, even several weeks. Many pieces of research are made to improve deep learning algorithms regarding time and memory.

The main idea to improve algorithms regarding time is to use GPU instead of CPU to train and test the network. Another idea is to use fixed point format instead of floating point format. Fixed point operations are the same as integer operations and then are less time-consuming than in floating point operations.

To reduce the memory required for weights storage, one could reduce the precision of weights.

4.3.1 Tools to improve neural networks performance on CPUs

The aim of the article [16] is to introduce to neural network researchers some tools which can significantly improve the performance of neural networks on CPU. Different approaches are introduced in this article in order to improve the CPU's performance :

1. Using the right memory locality and parallel accumulator.
   Let A and B be two matrices, and $C = A \times B$. $c_{i,j} = \sum_k a_{i,k} b_{k,j}$. k walks the columns of A and the rows of B. Hence A is best stored in row-major order and B in column-major order. By storing them in that way, the computation of C will be faster.
   Let a and b be two column vectors and $c = a^T b = \sum_i a_i b_i$. The most common way to compute c is to use the instruction `c+=a[i]*b[i]`. A technique to compute c more efficiently is to use multiple accumulators in parallel :
   ```
   c0+=a[i]*b[i]
   c1+=a[i+1]*b[i+1]
   c2+=a[i+2]*b[i+2]
   c3+=a[i+3]*b[i+3]
   ```
   `c=c0+c1+c2+c3` where `c0`, `c1`, `c2`, and `c3` are computed in parallel.

2. Using 1, SSE2, and SIMD instructions.
   SIMD instructions perform multiple operations in parallel. They operate on 16 bytes worth data at a time. They operate faster on 16 byte blocks that are 16-byte aligned (memory address of the first byte is a multiple of 16). If data is not 16-byte aligned, one only has to force it. If the block size is not 16 byte, one only has to use zero-padding to add zero to the block until the size is a multiple of 16.
   SSE2 instructions provide the basic instructions to perform the multiply-and-add step using floating-point SIMD arithmetic.

3. Using 2, and 8-bit fixed point. The different neurons of the network can be seen as probabilities, they live in the [0,1] interval. Hence, they can be represented as unsigned integers. In this approach, the inputs of the network are represented in single-precision floating-point, neurons as 8-bit unsigned integer, biases as 32-bit signed integer and weights as 8-bit signed integer.

4. Using 3, and SSSE3 instructions. The storage of data, weights and biases is the same as in 3. Instead of using SSE2 instructions, one has only to use SSSE3 instructions.

5. Using 3, and SSE4 instructions. The storage of data, weights and biases is the same as in 4. Instead of using SSSE3 instructions, one has only to use SSE4 instructions.

6. Using 5, and batching. It just means that the network take several inputs into account at the same time.

7. Using 5, and lazy evaluation. The lazy evaluation consists in computing only neurons that are needed by the decoder. The decoder is used to interpret the outputs of the network.

8. Using 5, and batched lazy evaluation. The batched lazy evaluation uses the lazy evaluation when several inputs are taken into account.

The neural network used in the article [16] has 5 layers. The input layer has 440 inputs. Each hidden layer uses the sigmoid function and has 2000 units. The output has 7969 units and uses the softmax function. This network is used to perform speech recognition. Table 2 show the time of the different approaches to process 1s of speech.

	Time to process 1s of speech
Floating point on GPU	0.49s
Floating point on CPU	3.89s
Floating point with multiple accumulators in parallel (1)	3.09s
Floating point with SSE2 instructions (2)	1.36s
8-bit fixed point (3)	1.52s
8-bit fixed point with SSSE3 instructions (4)	0.51s
8-bit fixed point with SSE4 instructions (5)	0.47s
Batching (6)	0.36s
Lazy evaluation (7)	0.26s
Batched lazy evaluation (8)	0.21s

Table 2: Summary of the results

This article shows that using the right approach, neural network implemented on CPU can outperform those implemented on GPU.

4.3.2 A rounding mode that improves neural networks performance

Current large scale deep neural networks are often difficult to train. Actually, their training is often constrained by the available computational resources. The article [17] studies the effect of limited precision data representation and computation on neural network training. Using low-precision fixed-point computations, the impact of the rounding scheme in determining the network's behavior during training is studied. The stochastic rounding mode and an inner product computation approach are introduced :

1. Rounding modes.
   Let $\texttt{IL}$ be the number of integer bits, $\texttt{FL}$ the number of fractional bits and $\texttt{WL} = \texttt{IL} + \texttt{FL}$ the word length. We use the notation $\langle \texttt{IL}, \texttt{FL} \rangle$ to denote a fixed point representation and ϵ to denote the smallest positive number that may be represented in the given fixed point format.
   Given x and the target fixed point representation $\langle \texttt{IL}, \texttt{FL} \rangle$, we define $\lfloor x \rfloor$ as the largest multiple of ϵ less than or equal to x.
   The rounding mode that is usually used is the Round-to-nearest mode:
   $$Round(x, \langle \texttt{IL}, \texttt{FL} \rangle) = \begin{cases} \lfloor x \rfloor & \text{if } \lfloor x \rfloor \leq x \leq \lfloor x \rfloor + \frac{\epsilon}{2} \\ \lfloor x \rfloor + \epsilon & \text{if } \lfloor x \rfloor + \frac{\epsilon}{2} < x \leq \lfloor x \rfloor + \epsilon \end{cases}.$$
   The stochastic rounding is defined as follow:

$$Round(x, \langle \texttt{IL}, \texttt{FL} \rangle) = \begin{cases} \lfloor x \rfloor & \text{with probability } 1 - \frac{x - \lfloor x \rfloor}{\epsilon} \\ \lfloor x \rfloor + \epsilon & \text{with probability } \frac{x - \lfloor x \rfloor}{\epsilon} \end{cases}.$$

Using the stochastic rounding mode instead of round-to-nearest mode provides results that are closer to results obtained in floating-point representation.

Irrespective of the rounding mode, we define the convert operation as follow :

$$Convert(x, \langle \texttt{IL}, \texttt{FL} \rangle) = \begin{cases} -2^{\texttt{IL}-1} & \text{if } x \leq -2^{\texttt{IL}-1} \\ 2^{\texttt{IL}-1} - 2^{-\texttt{FL}} & \text{if } x \geq 2^{\texttt{IL}-1} - 2^{-\texttt{FL}} \\ Round(x, \langle \texttt{IL}, \texttt{FL} \rangle) & \text{otherwise} \end{cases} \tag{3}$$

.

2. Multiply and accumulate operation.
   Let a and b be two d-dimensional vectors such as each component is represented in the fixed point format $\langle \texttt{IL}, \texttt{FL} \rangle$, and define $c_0 = a^T b$. c_0 is represented in some fixed point format $\langle \tilde{\texttt{IL}}, \tilde{\texttt{FL}} \rangle$. First, compute $z = \sum_{i=1}^{d} a_i b_i$.
   The product of a_i and b_i produces a fixed point number in the $\langle 2 * \texttt{IL}, 2 * \texttt{FL} \rangle$ format. z is represented with enough width to prevent saturation/overflow and avoid any loss of precision while accumulating the sum over all products $a_i b_i$. Then convert z in $\langle \tilde{\texttt{IL}}, \tilde{\texttt{FL}} \rangle$ format : $c_0 = Convert(z, \langle \tilde{\texttt{IL}}, \tilde{\texttt{FL}} \rangle)$.

Three networks are trained. For the three networks, the word length `WL` for the fixed-point format is set to 16 bits.

- The first network is a fully-connected neural network and is trained using the MNIST dataset. This network has 784 inputs and two hidden layers. Each hidden layer contains 1000 units and its activation function is the rectified linear function. The output layer has 10 units and its activation function is the softmax function.
  The network is trained using the Stochastic Gradient Descent with a batch size of 100.
  The `float` baseline achieves a test error of 1.4%. Irrespective of the rounding mode, allocating 14 bits to the fractional part produces no noticeable degradation in either the convergence rate or classification accuracy. A reduction in the precision bellow 14 bits begins to negatively impact the network's ability to learn when the round-to-nearest scheme is adopted. Actually, most of the parameter updates are rounded down to zero. On the contrary, the stochastic rounding preserves the gradient information and the network is able to learn with as few as 8 bits of precision without significant loss of accuracy. However, at a precision lower than 8 bits, even the stochastic rounding scheme is unable to fully prevent the loss of gradient information.

- The second network is a convolutional network ans is trained using the MNIST dataset. This network has 784 inputs, 2 convolutional layers each followed by a subsampling layer, the last subsampling layer is connected to a fully-connected layer which is connected to the output layer. The first convolutional layer has 8 feature maps, a 5×5 filter and its activation function is the rectified linear function. The second convolutional layer has 15 feature maps, a 5×5 filter and its activation function is the rectified linear function. Each subsampling layer uses a 2×2 subsampling region and the max function. The fully-connected layer has 128 units and its activation function is the rectified linear function. The output layer has 10 units and its activation function is the softmax function.
  The network is trained using the Stochastic Gradient Descent with a weight decay $\lambda = 0.0005$, a momentum $\mu = 0.9$ and a learning rate sequence defined as follow : $\begin{cases} \alpha_0 = 0.1 \\ \alpha_t = \alpha_0 e^{-0.95t} \end{cases}$.
  When trained using `float`, the network achieves a test error of 0.77%. When the round-to-nearest scheme is adopted during fixed-point computations, the training procedure fails to converge. When stochastic rounding is used, the network achieves a test error of 0.83% and 0.90% for 14-bit and 12-bit precision-respectively. This results correspond to only a slight degradation from the `float` baseline.

- The last network is a convolutional network and is trained using the CIFAR10 dataset. This network has 3072 inputs, 3 convolutional layers each followed by a subsampling layer, the last subsampling layer is connected to the output layer. The convolutional layers consists of 64 5×5 filters and subsampling layers implement the max pooling function over a window of size 3×3. The output layer has 10 units and its activation function is the softmax function.
  The network is trained using the Stochastic Gradient Descent with a learning rate of 0.01 which is reduced by a factor 2 after 50, 75, and 100 epochs.
  When trained using `float`, the network achieves a test error of 24.6%. When the round-to-nearest scheme

is adopted during fixed-point computations, the training procedure fails to converge. In contrast, when stochastic rounding is used, the network achieves a test error of 25.4% and 28.8% for 14-bit and 12-bit precision, respectively.

The article [17] shows that training a network using the stochastic rounding scheme and fixed-point format provide results that are close to those obtained when using 32-bit floating-point format.

4.3.3 A new format that improves neural networks performance

The training of deep neural network is often limited by hardware. Hardware is mainly made out of memories and arithmetic operators. Multipliers are the most space and power-hungry arithmetic operators of deep learning. The objective of article [18] is to access the possibility to reduce the precision of multipliers for deep learning. Three approaches are introduced in this article :

1. Multiplier-Accumulator. Applying a deep neural network mainly consists of convolutions and matrix multiplications. The key arithmetic operation of deep neural networks is thus the multiply-accumulate operation. Artificial neurons are basically multiplier-accumulators computing weighted sums of their inputs. The cost of a fixed point multiplier varies as the square of the precision for small widths while the cost of adders and accumulators varies as a linear function of the precision. The cost of a fixed point multiplier-accumulator mainly depends on the precision of the multiplier. Hence, the idea is to use low precision multiplier and high precision accumulator.

2. Dynamic Fixed Point. When training deep neural networks, units, gradients and parameters have very different ranges, and gradients ranges slowly diminish during training. The fixed point format is thus ill-suited to deep learning. The dynamic fixed point is a variant of the fixed point format in which there are several radix point position instead of a single global one. Those radix point position are not fixed. Dynamic fixed point format is a compromise between floating point format and fixed point format. Each layer's weights, bias, weighted sum, outputs, and the respective gradients vectors and matrices is associated with a different radix point position. Those radix point positions are initialized with a global value.

3. Update vs propagations. The precision used for parameters during the updates is higher that the one used during the forward and backward propagation. The idea is to be able to accumulate small changes in the parameters and on the other hand sparing a few bits of memory bandwidth during forward propagation.

Four networks are trained. Each network was trained with low precision multipliers and high precision accumulator. The precision used for parameters during the updates is higher that the one used during the forward and backward propagation. Networks were trained using the stochastic gradient with momentum.

- **Network 1** : This network is a fully-connected network and was trained using the MNIST dataset. It has 784 inputs and two hidden layers. Each hidden layer is a maxout layer. The output layer has 10 units and its activation function is the softmax function.

- **Network 2** : This network is a convolutional network and was trained using the MNIST dataset. It has 784 inputs, three convolutional maxout layers each followed by a subsampling layer, the last subsampling layer is connected to the output layer. Subsampling layers implement the max pooling function. The output layer has 10 units and its activation function is the softmax function.

- **Network 3** : This network is a convolutional network and was trained using the CIFAR10 dataset. It has 3072 inputs, three convolutional maxout layers each followed by a subsampling layer, the last subsampling layer is connected to the output layer. Subsampling layers implement the max pooling function. The output layer has 10 units and its activation function is the softmax function.

- **Network 4** : This network is a convolutional network and was trained using the MNIST dataset. It has 3072 inputs, three convolutional maxout layers each followed by a subsampling layer, the last subsampling layer is connected to a fully-connected maxout layer which is connected to the output layer. Subsampling layers implement the max pooling function. The output layer has 10 units and its activation function is the softmax function.

Each network was trained using floating point, fixed point and dynamic fixed point. Baseline results are those obtained with single precision floating point.

- **Floating point** : Half precision floating format has little to no impact on the test error.
- **Fixed point** : The optimal radix point position is after the fifth most important bit. The corresponding range is approximately [-32,32]. The minimum bit-width for propagations in fixed point is 19 (20 with the sign). Below this bit-width, the test error rises very sharply. The minimum bit-width for parameter updates in fixed point is 19 (20 with the sign). Below this bit-width, the test error rises very sharply. Using 19 (20 with the sign) bits for both the propagations and the parameter updates has little impact on the test error.
- **Dynamic fixed point** : The minimum bit-width for propagations in fixed point is 9 (10 with the sign). Below this bit-width, the test error rises very sharply. The minimum bit-width for parameter updates in fixed point is 11 (12 with the sign). Below this bit-width, the test error rises very sharply. Using 9 (10 with the sign) bits for the propagations and 11 (12 with the sign) bits for the parameter updates has little impact on the test error. This is significantly better than fixed point format.

Table 3 presents the test error achieved when training the networks using different precision.

Format	Prop.	Up.	Network 1	Network 2	Network 3	Network 4
Single precision floating point	32	32	1.05%	0.51%	14.05%	2.71%
Half precision floating point	16	16	1.10%	0.51%	14.14%	3.02%
Fixed point	20	20	1.39%	0.57%	15.98%	2.97%
Dynamic fixed point	10	12	1.28%	0.59%	14.82%	4.95%

Table 3: Test error rates. Prop. is the bit-width of the propagations and Up. is the bit-width of the parameters updates.

Their code is available on `https://github.com/MatthieuCourbariaux/deep-learning-multipliers`.
The article [18] shows that using dynamic fixed point format for training provides more accuracy than fixed point format and faster computation than floating point format.

4.3.4 Using 8-bits representation for Deep Learning

Deep learning currently provides the best solutions to many problems in image recognition, speech recognition and natural language processing. As the complexity of the neural networks increases over years, the performance of them are highly limited by the hardware resources. Actually, memory capacity and memory bandwidth are bottlenecks for highly parallel and high-performance deep neural network implementations. To continue improving accuracy of large scale image and video recognition, the size and complexity of neural networks has increased over time. Many deep learning algorithms spend a lot time (up to a few weeks) on training over a huge amount of training data in order to achieve good accuracy.

Some convolutional neural networks require large memory capacity for inputs images, weights, and intermediate outputs. The memory capacity requirement has an increasing trend along time as the convolutional neural networks with more weights have potentials to achieve better accuracy. Furthermore, high-capacity memory cannot meet the bandwidth requirement as the performance of deep learning algorithms continue to scale up.

No existing memory technologies can meet both the memory capacity requirement and the memory bandwidth requirement at the same time. The article [19] proposes to use an approximator in the memory controller to reduce the memory bandwidth requirement while still enjoying the high capacity of the system.

The 8-bit representation is investigated. Two options of 8-bit representation are considered :

- 1 bit for sign, 5 bits for exponent and 2 bits for mantissa
- 1 bit for sign, 4 bits for exponent and 3 bits for mantissa

The exponential bias will also be modified correspondingly. Not all the weight are prone to the 8-bit floating point representation, an approximator in memory controller is thus introduced to decide whether to approximate or not for each memory access.

Both floating point representation are considered for the weights. The baseline is the single precision floating

point representation.

First, the error distribution is studied. The error introduced by reduced precision comes from two sources : the exponent is too small to be covered by fewer bits exponent, and the value is rounded due to the reduced number of bits for mantissa. The relative error is studied on both convolutional layers and fully-connected layers. The relative error is calculated with

$$e = \frac{f' - f}{f} \tag{4}$$

f' is the reduced precision representation of the initial single precision floating point f. f'_{43} denotes 8-bit floating point representation with 4 bits for exponent and 3 bits for mantissa. f'_{52} denotes 8-bit floating point representation with 5 bits for exponent and 2 bits for mantissa. For both convolutional layers and fully-connected layers, using f'_{43} causes more weights having 100% relative error. Using f'_{52} can cover larger exponent range so that most of weights have less than 15% relative error. Fully-connected layers have a larger fraction of weights that need more than 4% bits for exponents. They also have smaller errors because the magnitude of weights in these layers are smaller.

Then, the impact of reduced precision on accuracy is studied using two networks : VGG-D and AlexNet. Both networks are trained using the ImageNet dataset. The accuracy of VGG-D and AlexNet in image recognition task when approximating weights with different percentages and in different layers is analyzed.

- **VGG-D**. This network has 13 convolutional layers and 3 fully-connected layers.
  First, the software tolerance with uniformly distributed approximation in all layers is studied. The VGG-D accuracy loss when approximating 100% of the weights with f'_{43} is significant (92.22%). If the approximation is less than 20%, the accuracy loss is within 1%, which is acceptable for many deep learning applications.
  Then, the software tolerance with nonuniform approximation approximation in different weight layers is studied. The error tolerance to low precision varies in different layers. In convolutional layers, the reduced precision weights can cause significant accuracy loss in classification output especially when approximating more than 50% of weights. However, the classification output is less sensitive to the fully-connected layer weights approximation. Even if 100% of the fully-connected weights are reduced precision, the accuracy loss is still within 0.5%. Fully-connected layers have much more weights than convolutional layers. Fully-connected layers are thus more error resilient and can achieve significant bandwidth savings by approximating all fully-connected layer weights.

- **AlexNet**. This network has 5 convolutional layers and 3 fully-connected layers.
  First, the software tolerance with uniformly distributed approximation in all layers is studied. The accuracy loss is also within 1% if less than 20% of all the weights are approximated.
  Then, the software tolerance with nonuniform approximation approximation in different weight layers is studied. The accuracy loss is less than 0.5% even if all weights in fully-connected layers use reduced precision representation. AlexNet is more error resilient than VGG-D.

The article [19] shows that using 8-bits floating point to represent 20% of all weights less than 1% accuracy degradation in classification and can achieve significant memory bandwidth savings.

4.3.5 Customized floating point format for Deep Learning

With ever-increasing computational demand for deep learning, it is critical to investigate the implications of the numeric representation and precision of deep neural networks model weights and neurons on computational efficiency. Actually, precision requirements do not generalize across all neural networks. Moreover, many large-scale deep neural networks require considerably more precision for fixed point arithmetic than previously found from small-scale evaluations. For instance, GoogLeNet requires on the order of 40 bits when implemented with fixed-point arithmetic, as opposed to less than 16 bits for LeNet-5. Finally, floating-point representations are more efficient than fixed-point representations when selecting the optimal precision settings. For instance, 17-bit floating-point representation is acceptable for GoogLeNet, while over 40 bits are required for the fixed-point representation. Current platform designers should reconsider the use of the floating-point representations for deep neural networks computation instead of the commonly used fixed-point representation.

The article [20] introduced the customized floating-point precision. Its goal is to use precision that delivers sufficient accuracy while attaining large improvements in power, and speed over standard floating-point designs.

In a fixed-point representation, one select the number of bits as well as the position of the radix point. Fixed-point representations with a particular number of bits have a fixed level of precision. By varying the position of the radix point, one change the representable range. There are three parameters to select when designing a floating-point representation : the bit-width of the mantissa, the bit-width of the exponent, and an exponent bias. Both fixed-point and floating-point representations have limitations in terms of the precision and the dynamic ranges available given particular representations, manifesting themselves computationally as rounding and saturation errors. These errors propagate through the deep neural network in a way that is globally difficult to evaluate, prompting experimentation in the deep neural network itself.

The multiply-accumulate operation implements the sum-of-product operation that is fundamental to the activation of each neuron. Reducing the floating-point precision bit width improves hardware performance in two ways. First, reduced bit width makes the computation unit faster. Second, reduced bit width makes the computation unit smaller and require less energy.

Customized representations are evaluate through five different networks : GoogLeNet, VGG, AlexNet, CIFARNET, and LeNet-5. GoogLeNet, VGG, and AlexNet are evaluated on the ImageNet dataset, CIFARNET on the CIFAR10 dataset, and LeNet-5 on the MNIST dataset.

For GoogLeNet, VGG, and AlexNet the floating-point format is superior to the fixed-point format. The standard single precision floating-point format is faster than all fixed-point configurations that achieve above 40% accuracy. Customized precision floating-point representations are more efficient because less bits are needed for similar accuracy. By comparing the results across the five networks, it appears that the size and the structure of the network impacts the customized precision flexibility of the network.

The configuration with the highest performance that provides less than 1% loss of accuracy is a floating-point representation with 6 exponent bits and 7 mantissa bits, which yields a 7.2× speedup and a 3.4× savings in energy over the single precision IEEE 754 floating-point format. If more stringent accuracy requirement is necessary, 0.3% accuracy degradation, the representation with one additional bit in the mantissa can be used, which achieves a 5.7× speedup and 3.0× energy savings.

The article [20] shows that inference using customized representations on production-grade deep neural networks, including GoogLeNet and VGG, achieves an average speedup of 7.6× with less than 1% degradation in inference accuracy relative to single-precision floating point.

4.3.6 Two algorithms that outperforms Stochastic Gradient Descent

Stochastic Gradient Descent methods (SGDs) have been extensively employed in machine learning. They are simple to implement and also fast for problems that have many training examples. Despite its easy implementation, SGDs are difficult to tune and parallelize. They require much manual tuning of optimization parameters such as learning rate and convergence criteria. Moreover, it is very difficult to parallelize them using GPUs. These problems make it challenging to develop, debug and scale up deep learning algorithms with SGDs.

Limited memory BFGS (L-BFGS) and Conjugate gradient (CG) are usually much more stable to train and easier to check for convergence. The article [21] explores the use of L-BFGS and CG to train neural networks.

Two types of networks are trained : convolutional neural networks and autoencoders.
An autoencoder is an unsupervised learning architecture. Given an unlabelled dataset $\{x_i\}_{1\leq i\leq m}$, an autoencoder is a two-layer network that learns nonlinear codes to represent the data. Specifically, we want to learn representations $h(x_i;\theta,b) = \sigma(\theta x_i + b)$ such that $\sigma(\theta^T h(x_i;\theta,b) + c)$ is approximately x_i,

$$\min_{\theta,b,c} \sum_{i=1}^{m} \|\sigma(\theta^T \sigma(\theta x_i + b) + c) - x_i\|_2^2 \tag{5}$$

Typically, σ is the sigmoid or the hyperbolic tangent function.
The gradient of the autoencoder objective can be computed exactly and this gives rise to an opportunity to use more advanced methods, such as L-BFGS and CG, to train the networks.
Autoencoders have densely-connected network architecture which do not scale well to large scale images. For large

images, the most common approach is to use convolutional neural networks. With convolutional neural networks, the cost of communicating the gradient over the network is often cheaper than the cost of computing it.

The experiments were made using the MNIST dataset. For SGDs, a momentum and a learning rate schedule of $\frac{\alpha}{\beta+t}$ where t is the iteration number are used, and the number of examples used to compute the gradient is varied. L-BFGS and CG are ran with a fixed minibatch for 3 iterations and 20 iterations respectively and then resample a new minibatch from the larger training set.

1. **Autoencoder training**. The autoencoder has $10,000$ hiden units and the sigmoid activation function. For L-BFGS, the minibatch size varied in $\{1000, 10000\}$. For CG, the minibatch size varied in $\{100, 10000\}$. For SGDs, 20 combinations of optimization parameters are tried, including varying the minibatch size in $\{1, 10, 100, 1000\}$. The test errors of the different optimization methods are compared. The results show that minibatch L-BFGS and CG converge faster than SGDs. CG perform better compared to L-BFGS because computing the conjugate information can be less expensive than estimating the Hessian. CG perform also better than SGDs thanks to conjugate information.

2. **Training autoencoders with GPUs**. Using the same experimental protocols as in 1, optimization methods and their gains switching from CPUs to GPUs are compared. The speed up gains are much higher for L-BFGS and CG than SGDs. L-BFGS and CG prefer larger minibatch sizes which can be parallelized more efficiently on the GPUs.

3. **Parallel training of dense networks**. Optimization methods for training autoencoders in distributed fashion using the Map-Reduce framework are explored. The Map-Reduce framework can be described as follow : one central machine (the master) runs the optimization procedure while the slaves compute the objective values and gradients. At every step during optimization, the master sends the parameter across all slaves, the slaves then compute the objective function and gradient and send back to the master. The settings are the same as in 1. The results show that parallelizing densely connected networks in this way can result in slower convergence than running the method on a standalone machine.

4. **Parallel training of supervised convolutional neural networks**. Different optimization methods are compared for supervised training of two-layer convolutional networks. The first hidden layer has 16 maps of 5×5 filters, followed by pooling units that pool over 3×3 region. The second hidden layer has 16 maps of 4×4 filters, without any pooling units. The output layer is a softmax layer that has 10 units. In this experiment, the gradient computations are distributed across many machine with GPUs. The results show that L-BFGS is better than CG and SGDs on this problem because of the low dimension.

5. **Classification on standard MNIST**. Experiments are carried out to determine if L-BFGS affects classification accuracy. A convolutional neural network is used. Its first layer has 32 maps of 5×5 filters and 3×3 pooling with subsampling. The second layer has 64 maps of 5×5 filters and 2×2 pooling with subsampling. The output layer has 10 units and the softmax activation function. The network is trained using four machine. The results show that the convolutional neural network, trained with L-BFGS, achieves a test error of 0.69%.

The article [21] shows that more sophisticated off-the-shelf optimization methods such as L-BFGS and CG can significantly simplify and speed up the process of training deep algorithms.

5 Stability (learning theory)

The article [22] introduces the notion of stability of a learning algorithm. The article [23] proves that the Stochastic Gradient Method is stable in sense of *Bousquet and Elisseeff*.

5.1 Notations and definitions

Let $\mathcal{X}$ be the set of inputs and $\mathcal{Y}$ the set of outputs of a network. Let $\mathcal{Z} = \mathcal{X} \times \mathcal{Y}$. Therefore, we have $S \in \mathcal{Z}^m$. Let A be an algorithm. We notice $A(S)$ the weights obtained when applied the algorithm A to the training set S. We notice $S^{\setminus i} = \{z_1, \ldots, z_{i-1}, z_{i+1}, \ldots, z_m\}$. Let Θ be the set of possible parameters θ of a network.

Definition 1. An algorithm A has **hypothesis stability** β with respect to the loss function f if the following holds

$$\forall i \in \{1, \ldots, m\},\ \mathbb{E}_{S,z}[|f(A(S), z) - f(A(S^{\setminus i}), z)|] \leq \beta. \tag{6}$$

Definition 2. An algorithm A has **pointwise hypothesis stability** β with respect to the loss function f if the following holds

$$\forall i \in \{1, \ldots, m\},\ \mathbb{E}_{S}[|f(A(S), z_i) - f(A(S^{\setminus i}), z_i)|] \leq \beta. \tag{7}$$

Definition 3. An algorithm A has **error stability** β with respect to the loss function f if the following holds

$$\forall S \in \mathcal{Z}^m,\ \forall i \in \{1, \ldots, m\},\ |\mathbb{E}_z[f(A(S), z)] - \mathbb{E}_z[f(A(S^{\setminus i}), z)]| \leq \beta. \tag{8}$$

Definition 4. An algorithm A has **uniform stability** β with respect to the loss function f if the following holds

$$\forall S \in \mathcal{Z}^m,\ \forall i \in \{1, \ldots, m\},\ \|f(A(S), .) - f(A(S^{\setminus i}), .)\|_{\infty}. \leq \beta \tag{9}$$

Definition 5. An algorithm A is said to be stable if β decrease as $\frac{1}{m}$.

5.2 Properties

Before talking about the stability of Stochastic Gradient Method let introduce some definitions.

Definition 6. A function $g : \Theta \to \mathbb{R}$ is *L-Lipschitz* if for all $u, v \in \Theta$ we have

$$\|g(u) - g(v)\| \leq L\|u - v\| \tag{10}$$

Definition 7. A function $g : \Theta \to \mathbb{R}$ is *β-smooth* if for all $u, v \in \Theta$ we have

$$\|\nabla g(u) - \nabla g(v)\| \leq \beta\|u - v\| \tag{11}$$

Definition 8. A function $g : \Theta \to \mathbb{R}$ is *convex* if for all $u, v \in \Theta$ we have

$$g(u) \geq g(v) + \langle \nabla g(v), u - v \rangle \tag{12}$$

Definition 9. A function $g : \Theta \to \mathbb{R}$ is *γ-strongly convex* if for all $u, v \in \Theta$ we have

$$g(u) \geq g(v) + \langle \nabla g(v), u - v \rangle + \frac{\gamma}{2}\|u - v\|^2 \tag{13}$$

5.3 Stability of Stochastic Gradient Method

We notice ϵ_{stab} the infimum over all β for which (6) holds.

Theorem 1. Assume that the loss function $f(\cdot, z)$ is β-smooth, convex, and L-Lipschitz for all z. Suppose that the Stochastic Gradient Descent is ran with step sizes $\alpha_n \leq \frac{2}{\beta}$ for N steps. Then the Stochastic Gradient Descent satisfies uniform stability with

$$\epsilon_{\text{stab}} \leq \frac{2L^2}{m} \sum_{n=1}^{N} \alpha_n$$

Theorem 2. Assume that the loss function $f(\cdot, z)$ is β-smooth, γ-strongly convex for all z, and $L = \sup_{\theta \in \Theta} \sup_z \|\nabla f(\theta, z)\|_2$. Suppose that the Stochastic Gradient Descent is ran with a constant step size $\alpha \leq \frac{1}{\beta}$ for N steps. Then the Stochastic Gradient Descent satisfies uniform stability with

$$\epsilon_{\text{stab}} \leq \frac{2L^2}{\gamma m}$$

Theorem 3. Assume that the loss function $f(\cdot, z) \in [0; 1]$ is γ-strongly convex, is β-smooth for all z, and $L = \sup_{\theta \in \Theta} \sup_z \|\nabla f(\theta, z)\|_2$. Suppose that the Stochastic Gradient Descent is ran with step sizes $\alpha_n = \frac{1}{\gamma n}$ for N steps. Then the Stochastic Gradient Descent satisfies uniform stability with

$$\epsilon_{\text{stab}} \leq \frac{2L^2 + \beta\rho}{\gamma m}$$

with $\rho = \sup_{\theta \in \Theta} \sup_z \|f(\theta, z)\|_2$

Theorem 4. Assume that the loss function $f(\cdot, z) \in [0; 1]$ is L-Lipschitz, and β-smooth for all z. Suppose that the Stochastic Gradient Descent is ran with monotonically non-increasing step sizes $\alpha_n \leq \frac{c}{n}$ for N steps. Then the Stochastic Gradient Descent satisfies uniform stability with

$$\epsilon_{\text{stab}} \leq \frac{1 + \frac{1}{\beta c}}{m - 1} (2cL^2)^{\frac{1}{\beta c + 1}} N^{\frac{\beta c}{\beta c + 1}} \lessapprox \frac{N^{1 - \frac{1}{\beta c + 1}}}{m}$$

All the bounds decrease as $\frac{1}{m}$. Therefore, the Stochastic Gradient Descent is stable.

6 Numerical Stability of Stochastic Gradient Descent

In this section, we present a proof of the numerical stability of stochastic gradient descent. Notations used in this section are the same as in section 3.2.1 and 3.3. [24, 25]

6.1 Case of a fully-connected network without hidden layers

Let m be the number of units of the input layer, p the number of units of the output layer, and y the output of the network.

Let us consider the Euclidean matrix norm and dot product and the Euclidean vector norm and dot product. We notice $\langle \cdot, \cdot \rangle$ the dot product and $\|\|$ the norm.

We also notice u the unit round-off.

In floating point arithmetic, we have

$$\hat{\theta}_{n+1} = \hat{\theta}_n - \alpha_n(\nabla_\theta f(\hat{\theta}_n, z_n) + E_n) + \epsilon_n \tag{14}$$

where

- E_n is the error made when computing the residual $\nabla_\theta f(\hat{\theta}_n, z_n)$,
- ϵ_n is the error made when adding $\alpha_n(\nabla_\theta f(\hat{\theta}_n, z_n) + E_n)$ to $\hat{\theta}_n$.

We assume that $\nabla_\theta f(\hat{\theta}_n, z_n)$ is computed in a possibly extended precision $\bar{u} \leq u$ before rounding back to working precision u, and that $\hat{\theta}_n$, $\alpha_n(\nabla_\theta f(\hat{\theta}_n, z_n) + E_n)$ are computed at precision u.
Hence we assume that there exists a function ψ depending on f, $\hat{\theta}_n$, u and $\bar{u}$ such that

$$\|E_n\| \leq u\|\nabla_\theta f(\hat{\theta}_n, z_n)\| + \psi(f, \hat{\theta}_n, u, \bar{u}). \tag{15}$$

For the error ϵ_n we have

$$\|\epsilon_n\| \leq u(\|\hat{\theta}_n\| + \|\hat{d}_n\|). \tag{16}$$

where $\hat{d}_n = \nabla_\theta f(\hat{\theta}_n, z_n) + E_n$.

$f(\theta, z) = \frac{1}{2}\|y_\theta(x) - t\|^2 = \frac{1}{2}\|\varphi(\theta x) - t\|^2$

Let compute $\nabla_\theta f(\theta, z)$

$\|\varphi(\theta x) - t\|^2 = \sum_{i=1}^{p} (\varphi_i(\theta x) - t_i)^2$

Let $(a, b) \in \{1, \dots m\} \times \{1, \dots, p\}$,

$$\frac{\partial f(\theta, z)}{\partial w_{b,a}} = \sum_{i=1}^{p} \frac{\partial f}{\partial y_i} \sum_{j=1}^{p} \frac{\partial y_i}{\partial net_j} \frac{\partial net_j}{\partial w_{b,a}}.$$

$$\frac{\partial net_j}{\partial w_{b,a}} = \frac{\partial \sum_{k=1}^{m} w_{j,k} x_k}{\partial w_{b,a}} = \delta_{j,b} x_a.$$

$\frac{\partial y_i}{\partial net_j} = J_{i,j}(net)$ where $net = \theta x$

$$\frac{\partial f}{\partial y_i} = \frac{\partial \frac{1}{2} \sum_{k=1}^{p} (y_k - t_k)^2}{\partial y_i} = y_i - t_i.$$

$$\frac{\partial f(\theta, z)}{\partial w_{b,a}} = x_a \sum_{i=1}^{p} (y_i - t_i) J_{i,b}(net).$$

So $\nabla_\theta f(\theta, z) = J(x\theta)(y - t)x^T$.

Hence

$$\theta_{n+1} = \theta_n - \alpha_n J(\theta_n x_n)(\varphi(\theta_n x_n) - t_n)x_n^T \tag{17}$$

In floating point arithmetic, we have

$$\hat{\theta}_{n+1} = \hat{\theta}_n - \alpha_n (J(\hat{\theta}_n x_n)(\varphi(\hat{\theta}_n x_n) - t_n)x_n^T + E_n) + \epsilon_n \tag{18}$$

with

$$\|E_n\| \leq u\|J(\hat{\theta}_n x_n)(\varphi(\hat{\theta}_n x_n) - t_n)x_n^T\| + \psi(f, \hat{\theta}_n, u, \bar{u}) \tag{19}$$

$$\|\epsilon_n\| \leq u(\|\hat{\theta}_n\| + \|\hat{d}_n\|) \tag{20}$$

where $\hat{d}_n = \alpha_n(J(\hat{\theta}_n x_n)(\varphi(\hat{\theta}_n x_n) - t_n)x_n^T + E_n)$.

Assume that φ is L-Lipschitz and β-smooth.

We consider the change in error for a single step iteration of the form (18). For notational convenience, we write $\hat{\theta}_{n+1} = \bar{\theta}$, $\hat{\theta} = \theta$, $\alpha_n = \alpha$, $x_n = x$, $t_n = t$, $E_n = E$, $\epsilon_n = \epsilon$, $\hat{d}_n = d$, $r = \varphi(\hat{\theta}_n x)$ and $J = J(\hat{\theta}_n x)$.

Hence we have

$$\bar{\theta} = \theta - \alpha(J(\theta x)(\varphi(\theta x) - t)x^T + E) + \epsilon \tag{21}$$

and

$$\|E\| \leq uJ(\theta x)(\varphi(\theta x) - t)x^T + \psi(f, \theta, u, \bar{u}), \tag{22}$$

$$\|\epsilon\| \leq u(\|\theta\| + \|d\|) \tag{23}$$

with

$$d = \alpha(J(\theta x)(\varphi(\theta x) - t)x^T + E) \tag{24}$$

We will use the following lemma to prove the stability of Stochastic Gradient Descent.

Lemma 1. For any $u, v \in \mathbb{R}^p$,

$$\|\varphi(v) - \varphi(u) - J(u)(v - u)\| \leq \frac{\beta}{2}\|v - u\|^2 \tag{25}$$

Assume that there is a θ_* such that $J(\theta_* x)(\varphi(\theta_* x) - t)x^T = 0 \Leftrightarrow J(\theta_* x)\varphi(\theta_* x)x^T = J(\theta_* x)tx^T$.

We notice $J_* = J(\theta_* x)$ and $r_* = \varphi(\theta_* x)$.

Let assume that J_* and J are non-singular $\Rightarrow \varphi(\theta_* x)x^T = tx^T$, that

$$\|J_*(J^{-1} - J_*^{-1})\| \leq \mu < 1, \tag{26}$$

that

$$\|J_*^{-1}(\theta - \theta_*)\| \leq \eta < 1, \tag{27}$$

and that

$$\|J_*(\theta - \theta_*)xx^T)\| \leq \nu < 1. \tag{28}$$

From the identity

$$J = (I_p + J_*(J^{-1} - J_*^{-1}))^{-1}J_* \tag{29}$$

it then follow that

$$\|J\| \leq \frac{\|J_*\|}{1 - \|J_*(J^{-1} - J_*^{-1})\|} \leq \frac{\|J_*\|}{1 - \mu}. \tag{30}$$

We have

$$\begin{aligned}\bar{\theta}-\theta_* &= \theta-\theta_*-\alpha(J(r-t)x^T+E)+\epsilon\\ &= \theta-\theta_*-\alpha J(r-r_*)x^T-\alpha E+\epsilon\\ &= (I_p-(J_*^{-1}+(J^{-1}-J_*^{-1}))^{-1}J_*^{-1})(\theta-\theta_*)+(J_*^{-1}+(J^{-1}-J_*^{-1}))^{-1}J_*^{-1}(\theta-\theta_*)-\alpha J(r-r_*)x^T-\alpha E+\epsilon,\end{aligned}$$

which gives

$$\|\bar{\theta}-\theta_*\| \le \|I_p-(J_*^{-1}+(J^{-1}-J_*^{-1}))^{-1}J_*^{-1}\|\|\theta-\theta_*\|+\|(J_*^{-1}+(J^{-1}-J_*^{-1}))^{-1}J_*^{-1}(\theta-\theta_*)-\alpha J(r-r_*)x^T\|+\|\alpha E\|+\|\epsilon\|.$$

From

$$\begin{aligned}I_p-(J_*^{-1}+(J^{-1}-J_*^{-1}))^{-1}J_*^{-1} &= (J_*^{-1}+(J^{-1}-J_*^{-1}))^{-1}(J^{-1}-J_*^{-1})\\ &= (I_p+J_*(J^{-1}-J_*^{-1}))^{-1}J_*(J^{-1}-J_*^{-1})\end{aligned}$$

it follows that

$$\|I_p-(J_*^{-1}+(J^{-1}-J_*^{-1}))^{-1}J_*^{-1}\| \le \frac{\|J_*(J^{-1}-J_*^{-1})\|}{1-\mu}.$$

We have

$$\begin{aligned}\|\alpha E\| &\le \alpha(u\|J(r-r_*)x^T\|+\psi(f,\theta,u,\bar{u}))\\ &\le \alpha(u\|J\|\|r-r_*\|\|x^T\|+\psi(f,\theta,u,\bar{u}))\\ &\le \frac{u}{1-\mu}\alpha Lm\|J_*\|\|\theta-\theta_*\|+\alpha\psi(f,\theta,u,\bar{u})),\end{aligned}$$

and

$$\begin{aligned}\|\epsilon\| &\le u(\|\theta\|+\|\alpha(J(r-r_*)x^T+E)\|)\\ &\le u(\|\theta-\theta_*\|+\|\theta_*\|+(1+u)\alpha\|J(r-r_*)x^T\|+\alpha\psi(f,\theta,u,\bar{u}))\\ &\le u(\|\theta-\theta_*\|+\|\theta_*\|+\frac{1+u}{1-\mu}\alpha Lm\|J_*\|\|\theta-\theta_*\|+\alpha\psi(f,\theta,u,\bar{u})).\end{aligned}$$

From

$$\begin{aligned}(J_*^{-1}+(J^{-1}-J_*^{-1}))^{-1}J_*^{-1}(\theta-\theta_*)-\alpha J(r-r_*)x^T &= JJ_*^{-1}(\theta-\theta_*)-\alpha J(r-r_*)x^T\\ &= J\left[J_*^{-1}(\theta-\theta_*)-\alpha(r-r_*-J_*(\theta x-\theta_* x))x^T-\alpha J_*(\theta x-\theta_* x)x^T\right]\\ &= J\left[J_*^{-1}(\theta-\theta_*)-\alpha(r-r_*-J_*(\theta x-\theta_* x))x^T-\alpha J_*(\theta-\theta_*)xx^T\right]\\ &= J\left[J_*^{-1}(\theta-\theta_*)-\alpha J_*(\theta x-\theta_* x)x^T\right]-\alpha J(r-r_*-J_*(\theta x-\theta_* x))x^T,\end{aligned}$$

and Lemma 1 it follows that

$$\begin{aligned}\|\alpha J(r-r_*-J_*(\theta x-\theta_* x))x^T\| &\le \alpha\|J\|\|r-r_*-J_*(\theta x-\theta_* x)\|\|x^T\|\\ &\le \alpha\frac{\|J_*\|}{1-\mu}\frac{\beta}{2}\|\theta x-\theta_* x\|^2\|x^T\|\\ &\le \alpha\frac{\beta m\sqrt{m}}{2(1-\mu)}\|J_*\|\|\theta-\theta_*\|^2.\end{aligned}$$

From equation (27) and (28) we have

$$\begin{aligned}\|J\left[J_*^{-1}(\theta-\theta_*)-\alpha J_*(\theta x-\theta_* x)x^T\right]\| &\le \|J\|(\|J_*^{-1}(\theta-\theta_*)\|+\alpha\|J_*(\theta x-\theta_* x)x^T\|)\\ &\le \frac{\|J_*\|}{1-\mu}(\eta+\alpha\nu)\end{aligned}$$

Hence

$$\|\bar{\theta}-\theta_*\| \le G\|\theta-\theta_*\|+g \tag{31}$$

with

$$G = \frac{\|J_*(J^{-1} - J_*^{-1})\|}{1-\mu} + \frac{\alpha\beta m\sqrt{m}}{2(1-\mu)}\|J_*\|\|\theta - \theta_*\| + \frac{u(2+u)}{1-\mu}\alpha L m\|J_*\| + u, \tag{32}$$

and

$$g = (1+u)\alpha\psi(f,\theta,u,\bar{u}) + u\|\theta_*\| + \frac{\|J_*\|(\eta+\alpha\nu)}{1-\mu}. \tag{33}$$

For MNIST dataset, we have $m = 784$ and $p = 10$. Let assume that $\forall\ 0 \leq i < p,\ \varphi_i(x) = \dfrac{1}{1+e^{-x_i}}$. φ is $\frac{1}{4}$-Lipschitz, J is $\frac{1}{8}$-Lipschitz and $\|J_*\| \leq \frac{1}{4}$. Hence for $\mu \leq \frac{1}{8}$, $\nu \leq \frac{1}{8}$, $\eta \leq \frac{1}{8}$ and $\alpha \leq 0.1$ we have $G \leq \frac{1}{5}$.

7 Forward Error Analysis

We study the forward error on one layer. We notice x the inputs, m the number of inputs, θ the weights, φ the activation function, y the outputs, and p the number of outputs of the layer. Hence we have $y = \varphi(\theta x)$. We notice $\hat{y}$, $\hat{\varphi}$ and $fl(\theta x)$ the computed approximation to y, φ and θx respectively.

Assume that $\hat{y}$ is computed at precision u. For $n \in \mathbb{N}$ such that $nu < 1$, we notice $\gamma_n = \dfrac{nu}{1-nu}$.

Assume that φ is L-Lipschitz and that $\hat{y} = \hat{\varphi}(fl(\theta x))$

We have

$$\begin{aligned}
\|y - \hat{y}\|_2 &= \|\varphi(\theta x) - \hat{\varphi}(fl(\theta x))\|_2 \\
&= \|\varphi(\theta x) - \varphi(fl(\theta x)) + \varphi(fl()\theta x) - \hat{\varphi}(fl(\theta x))\|_2 \\
&\leq \|\varphi(\theta x) - \varphi(fl(\theta x))\|_2 + \|\varphi(fl(\theta x)) - \hat{\varphi}(fl(\theta x))\|_2
\end{aligned}$$

with

$$\begin{aligned}
\|\varphi(\theta x) - \varphi(fl(\theta x))\|_2 &\leq L\|\theta x - fl(\theta x)\|_2 \\
&\leq L\sqrt{\min(m,p)}\gamma_m\|\theta\|_2\|x\|_2,
\end{aligned}$$

and

$$\begin{aligned}
\|\varphi(fl(\theta x)) - \hat{\varphi}(fl(\theta x))\|_2 &\leq u\|\varphi(fl(\theta x))\|_2 \\
&\leq u\|\varphi(fl(\theta x)) - \varphi(\theta x) + \varphi(\theta x)\|_2 \\
&\leq u\|\varphi(fl(\theta x)) - \varphi(\theta x)\|_2 + u\|\varphi(\theta x)\|_2 \\
&\leq uL\|fl(\theta x) - \theta x\|_2 + u\|y\|_2 \\
&\leq uL\sqrt{\min(m,p)}\gamma_m\|\theta\|_2\|x\|_2 + u\|y\|_2.
\end{aligned}$$

Hence

$$\begin{aligned}
\|y - \hat{y}\|_2 &\leq L\sqrt{\min(m,p)}\gamma_m\|\theta\|_2\|x\|_2 + uL\sqrt{\min(m,p)}\gamma_m\|\theta\|_2\|x\|_2 + u\|y\|_2 \\
&\leq (1+u)L\sqrt{\min(m,p)}\gamma_m\|\theta\|_2\|x\|_2 + u\|y\|_2.
\end{aligned}$$

References

[1] https://www.toptal.com/machine-learning/machine-learning-theory-an-introductory-primer.

[2] https://www.analyticsvidhya.com/blog/2017/04/comparison-between-deep-learning-machine-learning/.

[3] http://whatis.techtarget.com/definition/machine-learning.

[4] Li Deng and Dong Yu. Deep learning: Methods and applications. Technical report, May 2014.

[5] https://www.technologies-ebusiness.com/enjeux-et-tendances/le-deep-learning-pas-a-pas.

[6] https://www.toptal.com/machine-learning/an-introduction-to-deep-learning-from-perceptrons-to-deep-netw

[7] http://deeplearning.net/tutorial/lenet.html#lenet.

[8] Léon Bottou. Online algorithms and stochastic approximations. In David Saad, editor, *Online Learning and Neural Networks*. Cambridge University Press, Cambridge, UK, 1998. revised, oct 2012.

[9] H. Robbins and D. Siegmund. *A Convergence Theorem for Non Negative Almost Supermartingales and Some Applications*, pages 111–135. Springer New York, New York, NY, 1985.

[10] http://www.grappa.univ-lille3.fr/~gilleron/PolyApp/node28.html.

[11] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[12] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009.

[13] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5, 2011.

[14] C. Fellbaum. *WordNet: An Electronic Lexical Database*. Language, speech, and communication. MIT Press, 1998.

[15] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[16] Vincent Vanhoucke, Andrew Senior, and Mark Z Mao. Improving the speed of neural networks on cpus. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1, page 4, 2011.

[17] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on Machine Learning (ICML-15)*, pages 1737–1746, 2015.

[18] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*, 2014.

[19] Zhaoxia Deng, Cong Xu, Qiong Cai, and Paolo Faraboschi. Reduced-precision memory value approximation for deep learning, 2015.

[20] Parker Hill, Babak Zamirai, Shengshuo Lu, Yu-Wei Chao, Michael Laurenzano, Mehrzad Samadi, Marios Papaefthymiou, Scott Mahlke, Thomas Wenisch, Jia Deng, et al. Rethinking numerical representations for deep neural networks. 2016.

[21] Jiquan Ngiam, Adam Coates, Ahbik Lahiri, Bobby Prochnow, Quoc V Le, and Andrew Y Ng. On optimization methods for deep learning. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 265–272, 2011.

[22] Olivier Bousquet and André Elisseeff. Stability and generalization. *Journal of Machine Learning Research*, 2(Mar):499–526, 2002.

[23] Moritz Hardt, Benjamin Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. *arXiv preprint arXiv:1509.01240*, 2015.

[24] https://perso.esiee.fr/~bercherj/New/polys/FILT_ADA_jfb.pdf.

[25] Françoise Tisseur. Newton's method in floating point arithmetic and iterative refinement of generalized eigenvalue problems. *SIAM Journal on Matrix Analysis and Applications*, 22(4):1038–1057, 2001.

A Code

The code is available on https://github.com/marioncaum/Stage-Repport-2017

Requirements

- Theano 0.9
- Pylearn2 0.1
- a CUDA capable GPU

How to run networks_1_2_3_4.py ?

This program run well on CPU but is much faster when executed on GPU. To run this program on GPU, create a file named .theanorc and write the two following lines in this file :
[global]
device = cuda

Command Line

python networks_1_2_3_4.py	[format][number of neurons of the first layer] [number of neurons of the second layer][number of neurons of the third layer] [number of neurons of the fourth layer]

Format

There are 2 different format :

- single precision floating point (float32)
- double precision floating point (float64)

Number of neurons

The number of neurons should be a strictly positive integers.

Examples

python networks_1_2_3_4.py float32 5 20 30 40
python networks_1_2_3_4.py float64 5 20 30 40
python networks_1_2_3_4.py float32 10 40 60 80
python networks_1_2_3_4.py float64 10 40 60 80
python networks_1_2_3_4.py float32 50 200 300 400
python networks_1_2_3_4.py float64 50 200 300 400
python networks_1_2_3_4.py float32 500 2000 3000 4000
python networks_1_2_3_4.py float64 500 2000 3000 4000

Results of training

The results of the different networks are saved in the file named mlp_3_best.pkl. To see the accuracy you can use the program named print_monitor.py that is located in the pylearn2/scripts/ repertory of pylearn2 repertory.

How to run network_5.py ?

This program run well on CPU but is much faster when executed on GPU. To run this program on GPU, create a file named .theanorc and write the two following lines in this file :
[global]
device = cuda

Command Line

python network_5.py [format][number of neurons of the first layer]

Format

There are 2 different format :

- single precision floating point (float32)
- double precision floating point (float64)

Number of neurons

The number of neurons should be a strictly positive integers.

Examples

python network_5.py float32 500
python network_5.py float64 500

Results of training

The results of the different networks are saved in the file named mlp_best.pkl. To see the accuracy you can use the program named print_monitor.py that is located in the pylearn2/scripts/ repertory of pylearn2 repertory.

How to run network_6.py ?

This program run well on CPU but is much faster when executed on GPU. To run this program on GPU, create a file named .theanorc and write the two following lines in this file :
[global]
device = cuda

Command Line

python network_6.py [format][number of neurons of the first layer][number of neurons of the second layer]

Format

There are 2 different format :

- single precision floating point (float32)
- double precision floating point (float64)

Number of neurons

The number of neurons should be a strictly positive integers.

Examples

python network_6.py float32 500 1000
python network_6.py float64 500 1000

Results of training

The results of the different networks are saved in the file named mlp_2_best.pkl. To see the accuracy you can use the program named print_monitor.py that is located in the pylearn2/scripts/ repertory of pylearn2 repertory.

How to run network_7.py ?

This program run well on CPU but is much faster when executed on GPU. To run this program on GPU, create a file named .theanorc and write the two following lines in this file :
[global]
device = cuda

Command Line

python network_7.py [format][number of neurons of the first layer][number of neurons of the second layer]

Format

There are 2 different format :

- single precision floating point (float32)
- double precision floating point (float64)

Number of neurons

The number of neurons should be a strictly positive integers.

Examples

python network_7.py float32 500 1000
python network_7.py float64 500 1000

Results of training

The results of the different networks are saved in the file named mlp_3_best.pkl. To see the accuracy you can use the program named print_monitor.py that is located in the pylearn2/scripts/ repertory of pylearn2 repertory.

How to run network_8.py ?

This program run well on CPU but is much faster when executed on GPU. To run this program on GPU, create a file named .theanorc and write the two following lines in this file :
[global]
device = cuda

Command Line

python network_8.py [format][number of neurons of the first layer][number of neurons of the second layer]

Format

There are 2 different format :

- single precision floating point (float32)
- double precision floating point (float64)

Number of neurons

The number of neurons should be a strictly positive integers.

Examples

python network_8.py float32 500 1000
python network_8.py float64 500 1000

Results of training

The results of the different networks are saved in the file named maxout_best.pkl. To see the accuracy you can use the program named print_monitor.py that is located in the pylearn2/scripts/ repertory of pylearn2 repertory.

How to run network_9.py ?

This program have to be executed on GPU. To run this program on GPU, create a file named .theanorc and write the two following lines in this file :
[global]
device = gpu

Command Line

python network_9.py [format]

Format

There are 2 different format :

- single precision floating point (float32)
- double precision floating point (float64)

If you run this program with double precision floating point it will run on CPU, else it will run on GPU.

Examples

python network_9.py float32
python network_9.py float64

Results of training

The results of the different networks are saved in the file named convolutional_network_best.pkl. To see the accuracy you can use the program named print_monitor.py that is located in the pylearn2/scripts/ repertory of pylearn2 repertory.

Code

```
import sys
import theano
from pylearn2.train import Train
from pylearn2.datasets.mnist import MNIST
from pylearn2.costs.cost import SumOfCosts
from pylearn2.training_algorithms.sgd import SGD
from pylearn2.costs.mlp import Default, L1WeightDecay, WeightDecay
from pylearn2.models.mlp import MLP, Tanh, RectifiedLinear, Softmax
from pylearn2.train_extensions.best_params import MonitorBasedSaveBest
from pylearn2.termination_criteria import And, MonitorBased, EpochCounter
from pylearn2.training_algorithms.learning_rule import MomentumAdjustor, Momentum

# Importing the format in which the model will be trained
theano.config.floatX = sys.argv[1]

# Loading the MNIST dataset
train_set = MNIST(which_set = 'train',
                  start = 0,
                  stop = 50000)

valid_set = MNIST(which_set = 'train',
                  start = 50000,
                  stop = 60000)

test_set = MNIST(which_set = 'test')

ds = {"train" : train_set,
      "valid" : valid_set,
      "test" : test_set}

# Building the first hidden layer
mlp3_h0 = Tanh(layer_name = 'h0',
               dim = int(sys.argv[2]),
               sparse_init = 15)

# Building the second hidden layer
mlp3_h1 = RectifiedLinear(layer_name = 'h1',
                          dim = int(sys.argv[3]),
                          sparse_init = 2)

# Building the second Multilayer Perceptrons
mlp3 = MLP(layer_name = 'mlp3',
           layers = [mlp3_h0, mlp3_h1])

# Building the third hidden layer
mlp2_h0 = Tanh(layer_name = 'h0',
               dim = int(sys.argv[4]),
               sparse_init = 15)

# Building the fourth hidden layer
mlp2_h1 = RectifiedLinear(layer_name = 'h1',
                          dim = int(sys.argv[5]),
                          sparse_init = 2)

# Building the output layer
mlp2_y = Softmax(layer_name = 'y',
                 n_classes = 10,
                 irange = 0.)

# Building the third Multilayer Perceptrons
mlp2 = MLP(layer_name = 'mlp2',
           layers = [mlp2_h0, mlp2_h1, mlp2_y])

# Building the first Multilayer Perceptrons
mlp1 = MLP(layer_name = 'mlp1',
           layers = [mlp3, mlp2])

# Building the model
model = MLP(layers = [mlp1],
```

```
            nvis = 784)

# Defining algorithm properties
algo = SGD(batch_size = 100,
           learning_rate = .01,
           monitoring_dataset = ds,
           cost = SumOfCosts(costs = [Default(),
                                      L1WeightDecay(coeffs = [[[.0005, .0005], [.00005, .00005,
    .00005]]]),
                                      WeightDecay(coeffs = [[[.0005, .0005], [.00005, .00005,
    .00005]]])]),
           learning_rule = Momentum(init_momentum = .5),
           termination_criterion = And(criteria = [MonitorBased(channel_name = "
    valid_mlp1_mlp2_y_misclass",
                                                                prop_decrease = 0.,
                                                                N = 10),
                                                   EpochCounter(max_epochs = 10000)]))

# Pull it all together
train = Train(dataset = train_set,
              model = model,
              algorithm = algo,
              extensions = [MonitorBasedSaveBest(channel_name = 'valid_mlp1_mlp2_y_misclass',
                                                 save_path = "mlp_3_best.pkl"),
                            MomentumAdjustor(start = 1,
                                             saturate = 10,
                                             final_momentum = .99)])

#Training the model
train.main_loop()
```

networks_1_2_3_4.py

```
import sys
import theano
from pylearn2.train import Train
from pylearn2.datasets.mnist import MNIST
from pylearn2.training_algorithms.bgd import BGD
from pylearn2.models.mlp import MLP, Sigmoid, Softmax
from pylearn2.train_extensions.best_params import MonitorBasedSaveBest
from pylearn2.termination_criteria import And, MonitorBased, EpochCounter

# Importing the format in which the model will be trained
theano.config.floatX = sys.argv[1]

# Loading the MNIST dataset
train_set = MNIST(which_set = 'train',
                  start = 0,
                  stop = 50000)

valid_set = MNIST(which_set = 'train',
                  start = 50000,
                  stop = 60000)

test_set = MNIST(which_set = 'test')

ds = {"train" : train_set,
      "valid" : valid_set,
      "test" : test_set}

# Building the first hidden layer
h0 = Sigmoid(layer_name = 'h0',
             dim = int(sys.argv[2]),
             sparse_init = 15)

# Building the output layer
y = Softmax(layer_name = 'y',
            n_classes = 10,
            irange = 0.)

# Building the model
```

```
model = MLP(layers = [h0, y],
            nvis = 784)

# Defining algorithm properties
algo = BGD(batch_size = 10000,
           line_search_mode = 'exhaustive',
           conjugate = 1,
           updates_per_batch = 10,
           monitoring_dataset = ds,
           termination_criterion = And(criteria = [MonitorBased(channel_name = "valid_y_misclass")
    ,
                                                   EpochCounter(max_epochs = 10000)]))

# Pull it all together
train = Train(dataset = train_set,
              model = model,
              algorithm = algo,
              extensions = [MonitorBasedSaveBest(channel_name = 'valid_y_misclass',
                                                 save_path = "mlp_best.pkl")])

# Training the model
train.main_loop()
```

network_5.py

```
import sys
import theano
from pylearn2.train import Train
from pylearn2.datasets.mnist import MNIST
from pylearn2.training_algorithms.sgd import SGD
from pylearn2.models.mlp import MLP, RectifiedLinear, Softmax
from pylearn2.train_extensions.best_params import MonitorBasedSaveBest
from pylearn2.termination_criteria import And, MonitorBased, EpochCounter
from pylearn2.training_algorithms.learning_rule import MomentumAdjustor, Momentum

# Importing the format in which the model will be trained
theano.config.floatX = sys.argv[1]

# Loading the MNIST dataset
train_set = MNIST(which_set = 'train',
                  start = 0,
                  stop = 50000)

valid_set = MNIST(which_set = 'train',
                  start = 50000,
                  stop = 60000)

test_set = MNIST(which_set = 'test')

ds = {"train" : train_set,
      "valid" : valid_set,
      "test" : test_set}

# Building the first hidden layer
h0 = RectifiedLinear(layer_name = 'h0',
                     dim = int(sys.argv[2]),
                     sparse_init = 15)

# Building the second hidden layer
h1 = RectifiedLinear(layer_name = 'h1',
                     dim = int(sys.argv[3]),
                     sparse_init = 15)

# Building the output layer
y = Softmax(layer_name = 'y',
            n_classes = 10,
            irange = 0.)

# Building the model
model = MLP(layers = [h0, h1, y],
            nvis = 784)
```

```
# Defining algorithm properties
algo = SGD(batch_size = 100,
           learning_rate = .01,
           monitoring_dataset = ds,
           learning_rule = Momentum(init_momentum = .5),
           termination_criterion = And(criteria = [MonitorBased(channel_name = "valid_y_misclass",
                                                                prop_decrease = 0.,
                                                                N = 10),
                                                   EpochCounter(max_epochs = 10000)]))

# Pull it all together
train = Train(dataset = train_set,
              model = model,
              algorithm = algo,
              extensions = [MonitorBasedSaveBest(channel_name = 'valid_y_misclass',
                                                 save_path = "mlp_2_best.pkl"),
                            MomentumAdjustor(start = 1,
                                             saturate = 10,
                                             final_momentum = .99)])

# Training the model
train.main_loop()
```

network_6.py

```
import sys
import theano
from pylearn2.train import Train
from pylearn2.datasets.mnist import MNIST
from pylearn2.costs.cost import SumOfCosts
from pylearn2.training_algorithms.sgd import SGD
from pylearn2.costs.mlp import Default, WeightDecay
from pylearn2.models.mlp import MLP, RectifiedLinear, Softmax
from pylearn2.train_extensions.best_params import MonitorBasedSaveBest
from pylearn2.termination_criteria import And, MonitorBased, EpochCounter
from pylearn2.training_algorithms.learning_rule import MomentumAdjustor, Momentum

# Importing the format in which the model will be trained
theano.config.floatX = sys.argv[1]

# Loading the MNIST dataset
train_set = MNIST(which_set = 'train',
                  start = 0,
                  stop = 50000)

valid_set = MNIST(which_set = 'train',
                  start = 50000,
                  stop = 60000)

test_set = MNIST(which_set = 'test')

ds = {"train" : train_set,
      "valid" : valid_set,
      "test" : test_set}

# Building the first hidden layer
h0 = RectifiedLinear(layer_name = 'h0',
                     dim = int(sys.argv[2]),
                     sparse_init = 15)
# Building the second hidden layer
h1 = RectifiedLinear(layer_name = 'h1',
                     dim = int(sys.argv[3]),
                     sparse_init = 15)

# Building the output layer
y = Softmax(layer_name = 'y',
            n_classes = 10,
            irange = 0.)

# Building the model
```

```
model = MLP(layers = [h0, h1, y],
            nvis = 784)

# Defining algorithm properties
algo = SGD(batch_size = 100,
           learning_rate = .01,
           monitoring_dataset = ds,
           cost = SumOfCosts(costs = [Default(),
                                      WeightDecay(coeffs = [.00005, .00005, .00005])]),
           learning_rule = Momentum(init_momentum = .5),
           termination_criterion = And(criteria = [MonitorBased(channel_name = "valid_y_misclass",
                                                                prop_decrease = 0.,
                                                                N = 10),
                                                   EpochCounter(max_epochs = 10000)]))

# Pull it all together
train = Train(dataset = train_set,
              model = model,
              algorithm = algo,
              extensions = [MonitorBasedSaveBest(channel_name = 'valid_y_misclass',
                                                 save_path = "mlp_3_best.pkl"),
                            MomentumAdjustor(start = 1,
                                             saturate = 10,
                                             final_momentum = .99)])

# Training the model
train.main_loop()
```

network_7.py

```
import sys
import theano
from pylearn2.train import Train
from pylearn2.models.maxout import Maxout
from pylearn2.datasets.mnist import MNIST
from pylearn2.models.mlp import MLP, Softmax
from pylearn2.costs.mlp.dropout import Dropout
from pylearn2.termination_criteria import MonitorBased
from pylearn2.training_algorithms.sgd import SGD, ExponentialDecay
from pylearn2.train_extensions.best_params import MonitorBasedSaveBest
from pylearn2.training_algorithms.learning_rule import Momentum, MomentumAdjustor

# Importing the format in which the model will be trained
theano.config.floatX = sys.argv[1]

# Loading the MNIST dataset
train_set = MNIST(which_set = 'train',
                  start = 0,
                  stop = 50000)

valid_set = MNIST(which_set = 'train',
                  start = 50000,
                  stop = 60000)

test_set = MNIST(which_set = 'test')

ds = {'train' : train_set,
      'valid' : valid_set,
      'test' : test_set}

# Building the first hidden layer
h0 = Maxout(layer_name = 'h0',
            num_units = 240,
            num_pieces = 5,
            irange = .005,
            max_col_norm = 1.9365)

# Building the second hidden layer
h1 = Maxout(layer_name = 'h1',
            num_units = 240,
            num_pieces = 5,
```

```
            irange = .005,
            max_col_norm = 1.9365)

# Building the output layer
y = Softmax(max_col_norm = 1.9365,
            layer_name = 'y',
            n_classes = 10,
            irange = .005)

# Building the model
model = MLP(layers = [h0, h1, y],
            nvis = 784)

# Defining algorithm properties
algo = SGD(batch_size = 100,
           learning_rate = .1,
           learning_rule = Momentum(init_momentum = .5),
           monitoring_dataset = ds,
           cost = Dropout(input_include_probs = { 'h0' : .8 },
                          input_scales = { 'h0': 1. }),
           termination_criterion = MonitorBased(channel_name = "valid_y_misclass",
                                                prop_decrease = 0.,
                                                N = 100),
           update_callbacks = ExponentialDecay(decay_factor = 1.000004,
                                               min_lr = .000001))

# Pull it all together
train = Train(dataset = train_set,
              model = model,
              algorithm = algo,
              extensions = [MonitorBasedSaveBest(channel_name = 'valid_y_misclass',
                                                 save_path = "maxout_best.pkl"),
                            MomentumAdjustor(start = 1,
                                             saturate = 250,
                                             final_momentum = .7)],
              save_path = "maxout"+theano.config.floatX+".pkl",
              save_freq = 1)

# Training the model
train.main_loop()
```

network_8.py

```
import sys
import theano
from pylearn2.train import Train
from pylearn2.space import Conv2DSpace
from pylearn2.datasets.mnist import MNIST
from pylearn2.costs.mlp import WeightDecay
from pylearn2.training_algorithms.sgd import SGD
from pylearn2.costs.cost import SumOfCosts, MethodCost
from pylearn2.models.mlp import MLP, ConvRectifiedLinear, Softmax
from pylearn2.train_extensions.best_params import MonitorBasedSaveBest
from pylearn2.termination_criteria import And, MonitorBased, EpochCounter
from pylearn2.training_algorithms.learning_rule import MomentumAdjustor, Momentum

# Importing the format in which the model will be trained
theano.config.floatX = sys.argv[1]

# Loading the MNIST dataset
train_set = MNIST(which_set = 'train',
                  start = 0,
                  stop = 50000)

valid_set = MNIST(which_set = 'train',
                  start = 50000,
                  stop = 60000)

test_set = MNIST(which_set = 'test')

ds = {"train" : train_set,
```

```
        "valid" : valid_set,
        "test" : test_set}

# Building the first hidden layer
h2 = ConvRectifiedLinear(layer_name = 'h2',
                         output_channels = 64,
                         irange = .05,
                         kernel_shape = [5, 5],
                         pool_shape = [4, 4],
                         pool_stride = [2, 2],
                         max_kernel_norm = 1.9365)

# Building the second hidden layer
h3 = ConvRectifiedLinear(layer_name = 'h3',
                         output_channels = 64,
                         irange = .05,
                         kernel_shape = [5, 5],
                         pool_shape = [4, 4],
                         pool_stride = [2, 2],
                         max_kernel_norm = 1.9365)

# Building the output layer
y = Softmax (max_col_norm = 1.9365,
             layer_name = 'y',
             n_classes = 10,
             istdev = .05)

# Building the model
model = MLP(batch_size = 100,
            input_space = Conv2DSpace(shape = [28, 28],
                                      num_channels = 1),
            layers = [h2, h3, y])

# Defining algorithm properties
algo = SGD(batch_size = 100,
           learning_rate = .01,
           learning_rule = Momentum(init_momentum = .5),
           monitoring_dataset = ds,
           cost = SumOfCosts(costs = [MethodCost(method = 'cost_from_X'),
                                      WeightDecay(coeffs = [.00005, .00005, .00005])]),
           termination_criterion = And(criteria = [MonitorBased(channel_name = "valid_y_misclass",
                                                                prop_decrease = 0.50,
                                                                N = 10),
                                                   EpochCounter(max_epochs = 10000)]))

# Pull it all together
train = Train(dataset = train_set,
              model = model,
              algorithm = algo,
              extensions = [MonitorBasedSaveBest(channel_name = 'valid_y_misclass',
                                                 save_path = "convolutional_network_best.pkl"),
                            MomentumAdjustor(start = 1,
                                             saturate = 10,
                                             final_momentum = .99)])

# Training the model
train.main_loop()
```

network_9.py