Tushar Agarwal (.270)
ECE 5759 Project

A Survey of Gradient-Descent Optimizer Variants

Introduction

Machine learning (ML) is a very powerful tool in the tool-box of modern-day engineers. It can be applied to wide range of problems and this project is one of its many applications. ML is broadly divided into supervised and unsupervised learning. Supervised learning is training a model using inputs having corresponding targets or labels. Unsupervised learning tries to find hidden structure in unlabeled data. Optimization is an essential part of all ML algorithms and the most extensively used ones, for this purpose, are the gradient descent method and its variations. This project, Digit Recognizer, is a fundamental example of how combination of image processing with supervised learning gives the machines the astounding capability of "Computer Vision".

Deep learning, a subfield of ML, works based on artificial neural networks (ANNs) [1]. These ANNs are universal function approximators which try to find the system model by mapping input labelled data to its labels. The process involves minimizing a loss function and to achieve this minimization, numerical optimization must be used. The unknown parameters in ANNs are the tunable weights and biases. The loss functions (functions to be minimized) are chosen to be convex (usually mean squared error). Given the high complexity of ANNs, the most extensively used algorithm for optimization of parameters in ANNs is the Gradient Descent (GD) method. The GD method in its original form is not suitable for calculations with large datasets (>1000000). So, the algorithm is modified in several ways which makes its application easier and suitable for specific applications. Reader is encouraged to refer [1], if unfamiliar with the ANNs.

The primary objective of this project is to make a program that recognizes handwritten digits using basic deep learning. For this purpose, an ANN is trained using the MNIST (Modified National Institute of Standards and Technology) dataset released in 1999 and provided by Kaggle.com for this project. The effectiveness of the model is then tested using accuracy score (percentage of correct predictions) on validation and test datasets. Another objective here is to test different optimizer algorithms for the same training task and determine preferable optimizers for such image processing tasks. The resulting program is submitted to the Kaggle.com competition.

Optimizers for Neural Networks

Mathematically, the GD algorithm can be simply expressed as

$$W_{k+1} = W_k - \alpha \nabla_W J(W_k) \tag{1}$$

Where , $W_k \in \mathbb{R}^p$ is the vector of parameters (weights and biases), $J(W_k): \mathbb{R}^p \rightarrow \mathbb{R}$ is the loss function, α is the step-size which is also referred to as the learning rate & subscript k denotes the k_{th} time-step.

There are three variants of gradient descent, which differ in how much data is used to compute the gradient of the loss function $\nabla_W J(W_k)$. Depending on the amount of data, a trade-off is made between the accuracy of the parameter update and the time it takes to perform an update.

Batch gradient descent

This is the basic GD algorithm. It computes the gradient of the loss function with respect to the weights for the entire training dataset and then takes their mean as the final value of the gradient. Mathematically,

$$\nabla_W J(W_k) = \frac{1}{n}\sum_{i=1}^{n} \nabla_W J(W_k, x_i, y_i) \tag{2}$$

Where x_i & y_i denote training data features and labels respectively.
Batch gradient descent is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces. But it can be very slow because the gradients for the whole dataset need to be calculated for a single update and is impractical for large datasets. It also doesn't allow real-time update of the model.

Stochastic gradient descent

Stochastic gradient descent (SGD) performs a parameter update for each training example

$$\nabla_W J(W_k) = \nabla_W J(W_k, x_i, y_i)$$

It is believed [2] that not all training samples are unique, i.e. there exists redundancies when batch gradient descent calculates mean gradient for large datasets, as it computes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. SGD doesn't have such redundancies and therefore, is usually much faster while also providing flexibility to update the model in real-time. SGD performs frequent updates with a high variance that cause the objective function to fluctuate heavily as in Figure 1.

In cases where multiple stationary points exist, batch gradient descent converges to the local minimum of the basin the parameters are placed in. SGD's fluctuation, on the one hand, enables it to jump to new and potentially better local minima but on the other hand, it might keep overshooting making it difficult to converge to a minimum.

Mini-batch gradient descent

Mini-batch GD is a middle-ground between SGD and Batch GD. It finds gradients on a mini-batch of some m training samples, where m<n.

$$\nabla_W J(W_k) = \frac{1}{m}\sum_{i=1}^{m} \nabla_W J(W_k, x_i, y_i)$$

Thus, it reduces the variance of the parameter updates which can lead to more stable convergence and can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch, very efficient. Common mini-batch sizes range between 50 and 256 but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network, is widely used and usually referred to as Stochastic Gradient Descent instead of SGD itself. For sake of simplicity, the parameters x_i and y_i are left out here onwards and should be taken for granted when calculating loss function J or its gradient.

Variations of Gradient descent optimization

The Gradient descent in its original form has many challenges to overcome, especially when it comes to large data-sets. Therefore, many variations of this have been developed specifically for the deep learning applications. The following list is not exhaustive as some variations, which are infeasible to compute in practice for high-dimensional data sets, have been skipped. For e.g. second-order methods like Newton's method.

Momentum Term

SGD has trouble navigating through areas where the surface curves much more steeply in one dimension than the other [3], a common phenomenon around local optima. In these scenarios, SGD oscillates across the steep slopes of the surface while only making slow progress towards the local optimum. Momentum [4] is a method that helps accelerate SGD in the relevant direction and dampens oscillations by adding a fraction γ (usually set ≈ 0.9) of the update vector v of the current time step to the next update vector.

$$v_{k+1} = \gamma v_k + \alpha \nabla_W J(W_k) \tag{4}$$

$$W_{k+1} = W_k - v_{k+1} \tag{5}$$

Analogous to the momentum of a ball rolling down-hill, the momentum term increases updates for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. This results in faster convergence and reduced oscillations.

Nesterov accelerated gradient

Nesterov accelerated gradient (NAG) [5] is an improvement on the momentum method. As the momentum term γv_k is used to move the parameters of W_k, W_k is replaced by $(W_k - \gamma v_k)$ in Eq. (4) as this gives an approximation of the next position of the parameters. Thus, the gradient is calculated w.r.t. the approximate future position of parameters and therefore this algorithm has a predictive behavior:

$$v_{k+1} = \gamma v_k + \alpha \nabla_W J(W_k - \gamma v_k) \tag{6}$$

The anticipatory behavior of NAG has significantly increased the performance of Recurrent Neural Networks [6].

Adagrad

Adagrad [7] is an algorithm for gradient-based optimization that adapts the learning rate to individual parameters, performing larger updates for infrequent and smaller updates for frequent parameters. It is therefore, well-suited for dealing with sparse data.
Let i^{th} parameter of the vector W_k at k^{th} time-step be $W_{k,i}$. In its update rule, Adagrad modifies the general learning rate α at each time step k for every $W_{k,i}$ based on its past gradients:

$$W_{k+1,i} = W_{k,i} - \frac{\alpha}{\sqrt{G_{k,ii} + \epsilon}} . \nabla_{W_{k,i}} J(W_k) \tag{7}$$

Where $G_k \in \mathbb{R}^{p \times p}$ is a diagonal matrix whose each diagonal element $G_{k,ii} = \sum_{j=1}^{k} \left(\nabla_{W_{j,i}} J(W_j) \right)^2$, ϵ is a smoothening term that avoids division by zero (usually of the order $1e^{-8}$).
Let $g_{k,i} = \nabla_{W_{k,i}} J(W_k) \Rightarrow g_k = \nabla_{W_k} J(W_k)$

This elementwise notation can be vectorized using the elementwise matrix-vector product $\odot$ as:

$$W_{k+1} = W_k - \frac{\alpha}{\sqrt{G_k + \epsilon}} \odot g_k = W_k - \Delta W_k \tag{8}$$

Where ΔW_k is the update vector. The following algorithms are expressed in terms of modifications to this update vector.
Demerit of using Adagrad is its accumulation of the squared gradients in the denominator causing the learning rate to aggressively decrease to zero and so, the algorithm is unable to acquire additional knowledge.

RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Professor Geoffrey Hinton in one of his Lectures [8]. It is an extension of Adagrad that seeks to reduce its rapidly diminishing learning rate by restricting the window of accumulated past gradients to some fixed size d. The running average G_k at time step k then depends (as a fraction $\gamma = 0.9$, similar to the Momentum term) only on the previous average and the current gradient:

$$G_k = \gamma G_{k-1} + (1-\gamma)(g_k)^2 \Rightarrow G_k = E[g^2]_k = \gamma E[g^2]_{k-1} + (1-\gamma) g_k^2 \tag{9}$$

Where $E[]$ refers to the expectation operator and $g_k^2 \in \mathbb{R}^{p\times p}$ is a diagonal matrix who's each diagonal element is the square of the gradient with respect to each individual parameter W_i.

Adadelta

Adadelta [9] can be considered an extension to RMSprop algorithm. It also tries to match the units of the update ΔW_k and the parameter W_k. For this, α is replaced by decaying average of squared parameter updates $\sqrt{\mathrm{E}[(\Delta W^2)_{k-1}] + \epsilon}$ where $\mathrm{E}[(\Delta W^2)_{k-1}] = \gamma \mathrm{E}[(\Delta W^2)_{k-2}] + (1-\gamma)(\Delta W_{k-1})^2$.
Thus, the final expression for Adadelta update rule is:

$$\Delta W_k = \frac{\sqrt{\mathrm{E}[(\Delta W^2)_{k-1}] + \epsilon}}{\sqrt{E[g^2]_k + \epsilon}} . g_k \tag{10}$$

Adadelta eliminates the need to specify a learning rate.

Adam

Adaptive Moment Estimation (Adam) [10] is an algorithm which uses both, decaying averages of past gradients (m_t) and squared gradients (v_t)

$$m_k = \beta_1 m_{k-1} + (1 - \beta_1) g_k, \qquad n_k = \beta_2 n_{k-1} + (1 - \beta_2) g_k^2 \tag{11}$$

m_k and n_k are estimates of the first and second moment of the gradients respectively, hence the name of the method. As m_k and n_k are initialized as vectors of zeros, they are biased towards zero, especially during the initial time steps and when β_1 and β_2 are close to 1. So, bias-corrected first and second moment estimates $\hat{m}_k = \frac{m_k}{1-\beta_1^k}$ and $\hat{n}_k = \frac{n_k}{1-\beta_2^k}$ are used in the final update rule as follows:

$$\Delta W_k = \frac{\alpha}{\sqrt{\hat{n}_k} + \epsilon} \hat{m}_k$$

(12)

Usually $\beta_1 \approx 0.9$ and $\beta_2 \approx 0.999$. Adam has been shown to work well in practice.

Adamax

The $\sqrt{n_k}$ term in Adam update basically scales the gradient inversely proportional to the l_2 norm of the past and current gradients. This can be extended to any general l_p norm but for $p > 2$ they are generally unstable [2]. But l_∞ norm has been found to give stable results [10] and the Adamax optimizer is based on this concept. So the update rule becomes:

$$\Delta W_k = \frac{\alpha}{u_k} \widehat{m}_k \tag{13}$$

Where $u_k = \beta_2^\infty u_{k-1} + (1 - \beta_2^\infty)|g_k|^\infty \Rightarrow u_k = \max(\beta_2 . u_{k-1}, |g_k|)$

Experiment

The programming for this project has been done in Python because it has inbuilt libraries for machine learning models and convenient IDEs like Spyder to use. The model fitting functions are used from python's keras library in conjunction with using the Tensorflow backend.

The dataset contains a total of 76000 scanned images of handwritten digits, together with their correct classifications. The data files are named train.csv (with 48000 images) and test.csv (with 28000 images) named appropriately and contain gray-scale images of hand-drawn digits, from zero through nine. Each image is 28 pixels in height and 28 pixels in width. Each pixel-value is an integer between 0 and 255, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. The test data has been taken from a distinct set of 250 people compared to the training data. If the program works good for the test data, it will indicate that system can recognize digits from people whose writing it didn't see during training. The training data is further divided randomly into a training set (80%) and a validation set (20%). The accuracy scores displayed in the results section are for the validation data-set.

The image data is provided in the form of a (48000 X 785) table or data-frame where first column represents the output label (Y) of the image and remaining 784 columns represent each pixel values. These 784 columns will be the input or features (X) of the model. The first column containing output labels cannot be used as is because training an ANN requires a categorically distributed output with binary values. Therefore, the technique referred to as *one-hot encoding* is used which distributes a single column of data with n types of values (n=10 here) into n columns of data with only binary values. Each row in this new data-frame has all 0's except one 1 in the column of the category that row originally belonged to.

For each of the 28000 images in the test set, the output of the program will be a single line containing the ImageId and the predicted digit. For example, if the prediction is that the first image is of a 3 and second image is of a 7, then the output would look like:

(ImageId,Label)
(1,3)
(2,7)
…(27998 more lines)…

Preprocessing is a crucial step before fitting any supervised learning models. For this project, the images are converted to binary images with a 0 for all values below a threshold value, th, and 1 for all values above th. This is a common technique used in image-processing and is referred to as *thresholding*. th value for this project has been taken as 0. This is done because the different gray-scale intensities are unimportant for this application. The identification of different shapes of the digits can be done with just a pair of colors, black (0) and white (1). This increases the effectiveness of training the models as well as prediction accuracies by removing non-essential features (color, in this case) which the model would have tried to learn.

The ANN model is built with 2 fully connected hidden layers having activation function 'ReLU' $f(x) = \max(0, x)$. The output layer uses the sigmoid activation function $\sigma(x) = \frac{1}{1+e^{-x}}$. These activation functions are essential for the ANN as they help the ANN to capture non-linearities of the function being approximated. The 1st and 2nd hidden layers have 256 and 64 neurons respectively. The model is trained for 15 epochs where 1 *epoch* refers to set of iterations which uses all of the data in different non-overlapping batches, to update the gradient. The final loss function used in categorical cross-entropy.

All of the 8 optimizers mentioned above are used to train the ANN to see their respective performances in terms of accuracy over validation dataset and total time consumed.

Results & Discussion

The results from executing the program are as follows.

Table 1

Optimization Algorithm	Time consumed for training (s)	Final Validation Accuracy	Final Loss Function value
SGD	15.0303	0.9168	0.2744
SGD with momentum	14.7540	0.9676	0.1252
SGD with momentum and Nesterov	15.3754	0.9711	0.1265
Adagrad	16.8868	0.9636	0.2480
RMSprop	16.1281	0.9742	0.1784
Adadelta	19.0121	0.9746	0.1838
Adam	17.9393	0.9694	0.2210
Adamax	17.5371	0.9740	0.1938

Table 1 shows the processing time, final accuracy and loss function values of these algorithms for training the ANN. In this particular application, Adadelta has maximum accuracy, SGD with momentum has the least training time but NAG seems to give the best overall performance. SGD has a significantly poor accuracy because the training dataset, like most ANN datasets, is sparse and therefore remaining optimizers are especially designed to cater this need.

The graphs in figure 1 show that all the optimizers except SGD give accuracy of greater than 95% but some optimizers like Adam, Adagrad, RMSprop and Adamax give better accuracy and faster convergence. However, the total processing time for these algorithms is also high with Adadelta being the worst of the lot. Another important thing to note here is the stability of the algorithms. Adam, RMSprop and Adadelta seem to have a lot of oscillations after convergence while other algorithms are

quite stable. Especially, Adagrad and Adamax are quite stable given that they are very similar to RMSprop and Adam respectively.

Evolution of validation dataset accuracy (y-axis) with each epoch (x-axis) for different optimizers

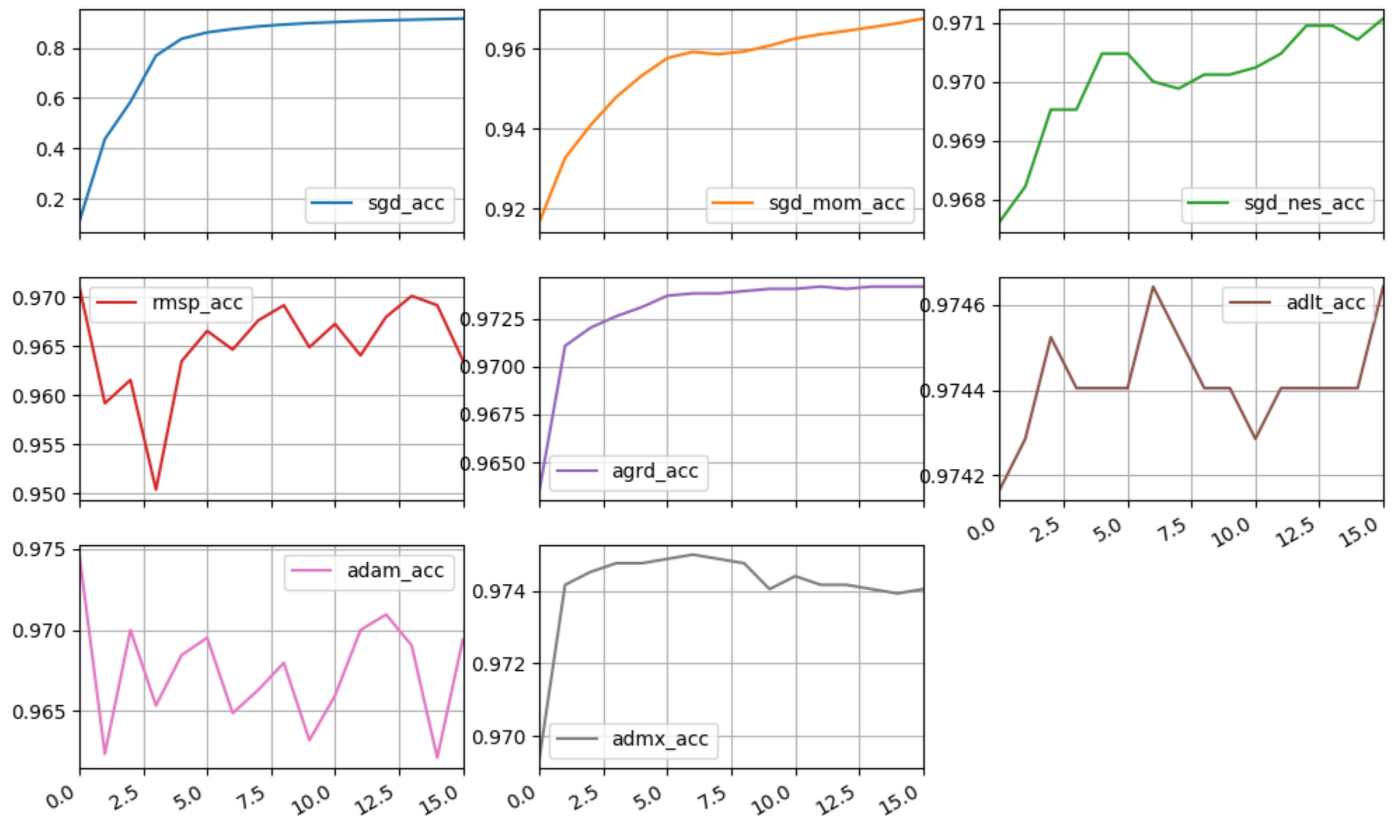

Figure 1

The results of the test dataset were submitted to Kaggle.com as the part of the competition and received >97% accuracy of prediction and a public leaderboard rank of 980. To depict the prediction ability of the model for the purposes of this project, a random image is chosen from the test dataset and given to the model. The images given and the predictions by the model are shown in figure 2.

It is important to note here that more recent and advanced version of ANNs called Convolutional Neural Networks (CNNs) are especially designed for image base learning models like this one and must be the preferred method of model creation in such tasks. This dataset was simple enough to be implemented using a basic ANN and still give great prediction accuracy. The purpose here was to demonstrate performance of various optimizers in conjunction to capabilities of ANNs in general.

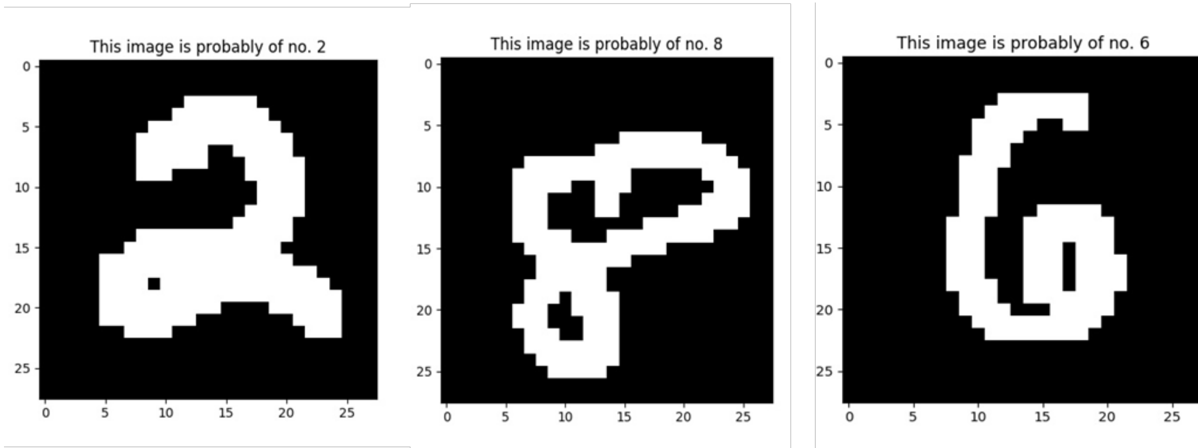

Figure 2

Conclusion and Future Work

In terms of choosing the optimizer for neural network application which usually have sparse training data, it can be concluded that any optimizer among RMSprop, Adam, Adadelta, Adamax and even NAG can be used. For simpler applications like this one, NAG can give overall best performance accounting for the low processing time for training also.

The results of this project are convincing enough to say that machine learning using ANNs provide the ability to make a machine learn and identify almost any image. This project serves as a proof of concept and can be developed into numerous useful applications. It can be used to build a hand-writing recognition software for touch-screen devices or to build an optical character recognition system which can make digital copies of books in a blink of an eye.

References

1. Wikipedia.org, Artificial Neural Networks. en.wikipedia.org/wiki/Artificial_neural_network
2. Ruder S., An overview of gradient descent optimization algorithms, 2016, http://sebastianruder.com/optimizing-gradient-descent/index.html
3. Sutton R. S., Two problems with backpropagation and other steepest-descent learning procedures for networks, 1986.
4. Qian N., On the momentum term in gradient descent learning algorithms. Neural networks: the official journal of the International Neural Network Society, 12(1):145–151, 1999.
5. Nesterov Y., A method for unconstrained convex minimization problem with the rate of convergence o(1/k2). Doklady ANSSSR (translated as Soviet.Math.Docl.), 269:543–547.
6. Bengio Y., Boulanger-Lewandowski N., Pascanu R., Advances in Optimizing Recurrent Networks. 2012.
7. Duchi J., Hazan E., Singer Y., Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. Journal of Machine Learning Research, 12:2121–2159, 2011.
8. Zeiler M. D., ADADELTA: An Adaptive Learning Rate Method. arXiv preprint arXiv:1212.5701, 2012.
9. Hinton G. et al., Neural Networks for Machine Learning, Lecture 6e, https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.
10. Kingma D. P. and Ba J. L., Adam: a Method for Stochastic Optimization. International Conference on Learning Representations, pages 1–13, 2015.