Optimization Algorithms for Deep Learning

Piji Li
Department of Systems Engineering and Engineering Management,
The Chinese University of Hong Kong
pjli@se.cuhk.edu.hk

Abstract

Gradient descent algorithms are the most important and popular techniques for optimizing deep learning related models. Considering the large scale dataset and the limited computation memory especially on GPUs, the traditional typical batch gradient decent method and stochastic gradient decent method cannot conduct the training effectively and efficiently. Moreover, tricks of tuning of step size (learning rate) also play an critical role during training procedure. To address this problem, several variations of gradient methods have been proposed. In this paper, we investigate the basic methodology and property of the methods. Moreover, we also conduct several experiments on various of tasks to compare the performance.

1 Introduction

Nowadays, Deep Learning [3] shows almost the state-of-the-art performance on various tasks in different fields, such as speech recognition, computer vision and natural language processing. There are huge number of different variants of deep architectures, such as Deep Neural Network(DNN), Deep Belief Network(DBN), Deep Convolutional Neural Network(CNN) and Recurrent Neural Network(RNN) [3]. The basic techniques of AlphaGo is also neural network [14].

Back-Propagation (BP) [13] is a common method of training neural networks. The algorithm repeats a two phase cycle, propagation and weight update. Errors obtained from the output layer will propagate to the other nodes in a backward direction. These errors are used to calculate the gradient of the loss function with respect to the weights in the network. Then the gradient is fed to the optimization method, which in turn uses it to update the weights, in an attempt to minimize the loss function.

According to the data used to compute the gradient of the objective function, gradient decent methods can be divided into three categories: batch gradient decent method, mini-batch gradient decent method, and stochastic gradient decent method [12]. Considering the large scale dataset and the limited computation memory especially on GPUs, the traditional typical batch gradient decent method and stochastic gradient decent method cannot conduct the training effectively and efficiently. Moreover, step size adjusting also plays an critical role during training procedure. But it is very difficult the tune the step size precisely in a task needs large dataset and long training period. To address these problems, several variations of typical gradient methods have been proposed. In the following sections, we will introduce the basic framework of the related methods. We also conduct experiments on image classification and headline generation to show the different performance of those methods.

2 Optimization Algorithms

Assume that the objective function need to be minimized is $f(x)$, and $x \in \mathbb{R}^n$. The corresponding gradient is $\Delta f(x)$. The step size for iteration k is t_k.

2.1 Batch Gradient Decent

The batch gradient decent algorithm updates the parameters x after scanning the whole training set:

$$x_{k+1} = x_k - t_k \Delta f(x_k)^{(1:n)} \tag{1}$$

Batch gradient descent is guaranteed to converge to the global minimum for convex problem and to a local minimum for non-convex problem. However, in the related tasks of deep learning, the training set contains about millions of or even billions of samples, which will cost a long time to scan the entire training set to calculate the gradient. So it is to slow to perform once parameter update. Moreover, the computation memory is also limited and is difficult to feed all the data into the model at one time. Therefore, rare of deep learning models use batch gradient decent method to handle the optimization problem.

2.2 Stochastic Gradient Decent

In contrast, stochastic gradient decent calculates the gradient and update the parameters for each training sample.

$$x_{k+1} = x_k - t_k \Delta f(x_k)^{(i)} \tag{2}$$

However, due to the high variance of different training samples, updating parameters frequently will cause the objective function to fluctuate heavily. Although a small step size can let SGD convergent to a good point, it make the training slow. Moreover, when we use GPUs to conduct the computation, the frequent data commutation between gpu memory and local memory also decrease the efficiency.

2.3 Mini-Batch Gradient Decent

Mini-Batch gradient decent integrates the advantages of batch gradient decent and stochastic gradient decent, and update the parameters after obtaining the gradient of a mini-batch of samples:

$$x_{k+1} = x_k - t_k \Delta f(x_k)^{(i:i+m)} \tag{3}$$

where the mini-batch size is m. Mini-batch gradient decent can not guarantee good convergence, and the tuning of step size also need some experience. Therefore some researchers extend it with some more useful tricks and techniques to improve the convergence. For convenience, people also call mini-batch gradient decent as SGD.

2.4 Gradient Decent with Momentum

If there is a long shallow ravine with steep walls on the direction to the optimal point, then the standard SGD will tend to oscillate across the narrow ravine. Momentum is one of the mechanism which is used to fix the direction:

$$\begin{aligned} v_k &= m v_{k-1} + t_k \Delta f(x_k) \\ x_{k+1} &= x_k - v_k \end{aligned} \tag{4}$$

where $m \in (0, 1]$ determines for how many iterations the previous gradients are incorporated into the current update. Generally m is set to 0.5 until the initial learning stabilizes and then is increased to 0.9 or higher.

2.5 Nesterov Accelerated Gradient

We can use Nesterov accelerated gradient (NAG) [10] to conduct the improvement by looking ahead to approximate the direction:

$$\begin{aligned} v_k &= m v_{k-1} + t_k \Delta f(x_k - m v_{k-1}) \\ x_{k+1} &= x_k - v_k \end{aligned} \tag{5}$$

2.6 Adagrad

Adagrad [2] scales the step size for each parameter according to the history of gradients for that parameter which is basically done by dividing current gradient in update rule by the sum of previous gradients:

$$\begin{aligned} G_k &= G_{k-1} + \Delta f(x_k)^2 \\ x_{k+1} &= x_k - \frac{t}{\sqrt{G_k + \varepsilon}} \Delta f(x_k) \end{aligned} \tag{6}$$

where G is the accumulation of the history gradients, and ε is a smoothing term that avoids division by zero (can be $1e - 6$). The step size is different for each of the parameters. It is greater for parameters where the historical gradients were small (since G is small) and the rate is small whenever historical gradients were relatively big. Therefore, we need not to manually tune the step size t. We can use a default value of 0.01. But when accumulation G becomes larger, the step size will reach zero at infinity. So the following methods are proposed.

2.7 Adadelta

Adadelta [15] is derived from Adagrad in order to improve upon the two main drawbacks of the method: 1) the continual decay of learning rates throughout training, and 2) the need for a manually selected global learning rate. Adadelta integrates the advantages of momentum and Adagrad. Specifically, it scales the step size based on the historical gradient. But it only uses the latest time window instead of the whole history as Adagrad. It also uses a component that serves an acceleration term, that accumulates historical updates (similar to momentum). Follows are the operation details of Adadelta:

$$\begin{aligned} \mathbb{E}[\Delta f(x)^2]_k &= \rho \mathbb{E}[\Delta f(x)^2]_{k-1} + (1-\rho)\Delta f(x_k)^2 \\ \hat{x}_k &= -\frac{\sqrt{\mathbb{E}[\hat{x}^2]_{k-1} + \varepsilon}}{\sqrt{\mathbb{E}[\Delta f(x)^2]_k + \varepsilon}} \Delta f(x_k) \\ \mathbb{E}[\hat{x}^2]_k &= \rho \mathbb{E}[\hat{x}^2]_{k-1} + (1-\rho)\hat{x}_k^2 \\ x_{k+1} &= x_k + \hat{x}_k \end{aligned} \tag{7}$$

where ρ is a decay constant (e.g., 0.95) and ε is a small value (e.g., 1e-6) for numerical stability.

2.8 RMSprop

RMSprop [5] is also proposed to tackle the problem of step size vanishing of Adagrad. It also employs the decaying average of the history gradients:

$$\begin{aligned} \mathbb{E}[\Delta f(x)^2]_k &= \rho \mathbb{E}[\Delta f(x)^2]_{k-1} + (1-\rho)\Delta f(x_k)^2 \\ x_{k+1} &= x_k - \frac{t}{\sqrt{\mathbb{E}[\Delta f(x)^2]_k + \varepsilon}} \Delta f(x_k) \end{aligned} \tag{8}$$

where ρ is a decay constant (e.g., 0.9).

2.9 Adam

Adam [6] is another method that computes adaptive step size for each parameter. It uses both the decaying average of history gradients and their squared values. Adam update rule consists of the following steps:

$$\begin{aligned} m_k &= \beta_1 m_{k-1} + (1-\beta_1)\Delta f(x_k) \\ v_k &= \beta_2 v_{k-1} + (1-\beta_2)\Delta f(x_k)^2 \\ \hat{m}_k &= \frac{m_k}{1-\beta_1^k}, \hat{v}_k = \frac{v_k}{1-\beta_2^k} \\ x_{k+1} &= x_k - \frac{t}{\sqrt{\hat{v}_k} + \varepsilon} \hat{m}_k \end{aligned} \tag{9}$$

where β_1 can be 0.9, β_2 can be 0.999, and ε can be $1e - 8$.

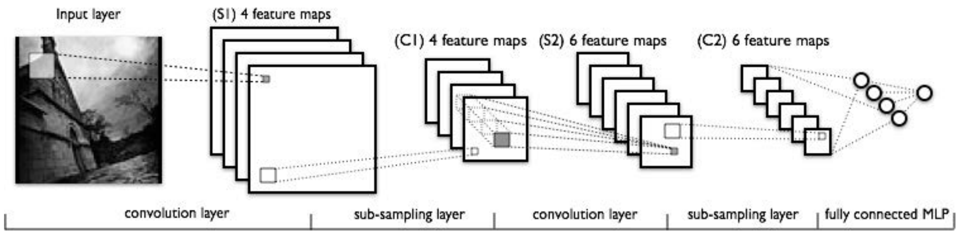


Figure 1: Structure of LeNet [9].

2.10 Adapg

Combine adadelta and adam, we can get a new method:

$$\begin{aligned}
&E[\Delta f(x)]_k = \rho E[\Delta f(x)]_{k-1} + (1-\rho)\Delta f(x_k)\\
&E[\Delta f(x)^2]_k = \rho E[\Delta f(x)^2]_{k-1} + (1-\rho)\Delta f(x_k)^2\\
&\hat{x}_k = -\frac{\sqrt{E[\hat{x}^2]_{k-1} + \varepsilon}}{\sqrt{E[\Delta f(x)^2]_k + \varepsilon}} E[\Delta f(x)]_k\\
&E[\hat{x}^2]_k = \rho E[\hat{x}^2]_{k-1} + (1-\rho)\hat{x}_k^2\\
&x_{k+1} = x_k + \hat{x}_k
\end{aligned} \tag{10}$$

3 Image Classification

3.1 Frameworks

To investigate the performance of those mentioned gradient methods in different model structures, we employ two neural network models to handle the handwritten digits recognition problem: Multi-Layer Perceptron (MLP) and Convolutional Neural Networks (CNN).

3.1.1 Multi-Layer Perceptron (MLP)

MLP is a one-hidden-layer artificial neural network. For the handwritten digits recognition problem, we use vector $\mathbf{x} \in \mathbb{R}^d$ to denote the input image, the operations in the feed-forward direction are as follows:

$$\begin{aligned}
\mathbf{h} &= \sigma(\mathbf{W}_h x + \mathbf{b}_h)\\
\hat{\mathbf{y}} &= softmax(\mathbf{W}_{\mathbf{y}} h + \mathbf{b}_{\mathbf{y}})
\end{aligned} \tag{11}$$

where $\mathbf{W}_h \in \mathbb{R}^{k\times d}$ and $\mathbf{W}_y \in \mathbb{R}^{c\times k}$ are the weight matrix, $\mathbf{b}_h \in \mathbb{R}^d$ and $\mathbf{b}_y \in \mathbb{R}^c$ are hte biases. $\sigma(\cdot)$ is the sigmoid function: $\sigma(\mathbf{z}) = \frac{1}{1+e^{-\mathbf{z}}}$, and $softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{c} e^{z_j}}$. The model's prediction $y^\star$ is the class whose probability is maximal, specifically: $y^* = \arg\max_i \hat{\mathbf{y}}_i$. Actually, we can regard $\hat{\mathbf{y}}$ as the probability distribution for each class, so we have $P(y^\star = i|\mathbf{x}, \Theta) = \hat{\mathbf{y}}_i$.

3.1.2 Convolutional Neural Networks (CNN)

CNN is a biologically-inspired variant of MLP. It consists of convolutional operation layers and the pooling layers. A feature map is obtained by repeated application of convolution operation of the input image with a linear filter, adding a bias term and then applying a non-linear function.

$$h_{ij} = \tanh((\mathbf{W} * \mathbf{x})_{ij} + b) \tag{12}$$

where $*$ is the convolutional operator. After each convolutional operation, we add a poling layer, which is a form of non-linear down-sampling. Max-pooling partitions the input image into a set of non-overlapping rectangles and, for each such sub-region, outputs the maximum value. In our experiments, we use LeNet-5 [9] as the basic framework, as shown in Figure 1.

3.2 Learning

Learning optimal model parameters involves minimizing a loss function. In the case of multi-class logistic regression, it is very common to use the negative log-likelihood as the loss. This is equivalent to maximizing the likelihood of the data set D under the model parameterized by Θ:

$$\min_{\Theta} \mathcal{J} = -\sum_{i=1}^{n} \log P(y^{*(i)} = y^{(i)} | \mathbf{x}^{(i)}, \Theta) + \lambda \|\Theta\|_2^2 \tag{13}$$

where $\lambda\|\Theta\|_2^2$ is an L2-regularization term (aka penalty) that penalizes complex models; and $\lambda > 0$ is a non-negative hyperparameter that controls the magnitude of the penalty.

4 Neural Headline Generation

4.1 Framework

In this task we will create a neural network model which can learn to generate headlines for news automatically like human.

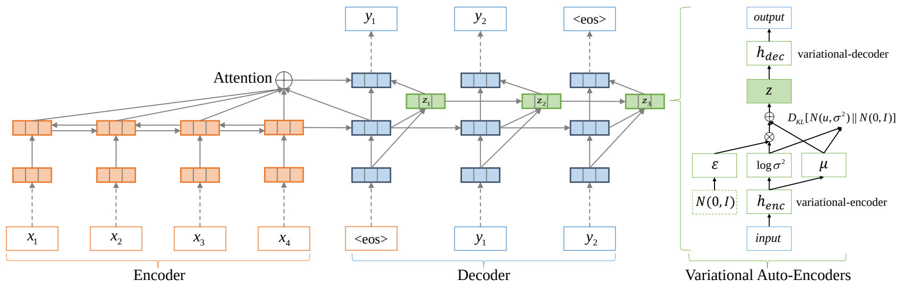


Figure 2: Our deep recurrent generative decoder (DRGN) for latent structure modeling.

As shown in the left block of Figure 2, the encoder is designed based on bidirectional recurrent neural networks. Let $\mathbf{x}_t$ be the word embedding vector of the t-th word in the source sequence. GRU maps $\mathbf{x}_t$ and the previous hidden state $\mathbf{h}_{t-1}$ to the current hidden state $\mathbf{h}_t$ in feed-forward direction and back-forward direction respectively:

$$\overrightarrow{\mathbf{h}}_t = GRU(x_t, \overrightarrow{\mathbf{h}}_{t-1}), \overleftarrow{\mathbf{h}}_t = GRU(x_t, \overleftarrow{\mathbf{h}}_{t-1}) \tag{14}$$

Then the final hidden state $\mathbf{h}_t^e \in \mathbb{R}^{2k_h}$ is concatenated using the hidden states from the two directions: $\mathbf{h}_t^e = \overrightarrow{\mathbf{h}}_t || \overleftarrow{\mathbf{h}}$. As shown in the middle block of Figure 2, the decoder consists of two components: discriminative deterministic decoding and generative latent structure modeling.

The discriminative deterministic decoding is an improved attention modeling based recurrent sequence decoder. The first hidden state $\mathbf{h}_1^d$ is initialized using the average of all the source input states: $\mathbf{h}_1^d = \frac{1}{T^e} \sum_{t=1}^{T^e} \mathbf{h}_t^e$, where $\mathbf{h}_t^e$ is the source input hidden state. T^e is the input sequence length. The deterministic decoder hidden state $\mathbf{h}_t^d$ is calculated using two layers of GRUs. On the first layer, the hidden state is calculated only using the current input word embedding $\mathbf{y}_{t-1}$ and the previous hidden state $\mathbf{h}_{t-1}^{d_1}$: $\mathbf{h}_t^{d_1} = GRU_1(\mathbf{y}_{t-1}, \mathbf{h}_{t-1}^{d_1})$. where the superscript d_1 denotes the first decider GRU layer. Then the attention weights at the time step t are calculated based on the relationship of $\mathbf{h}_t^{d_1}$ and all the source hidden states $\{\mathbf{h}_t^e\}$. Let $a_{i,j}$ be the attention weight between $\mathbf{h}_i^{d_1}$ and $\mathbf{h}_j^e$, which can be calculated using the following formulation:

$$a_{i,j} = \frac{\exp(e_{i,j})}{\sum_{j'=1}^{T^e} \exp(e_{i,j'})}, \quad e_{i,j} = \mathbf{v}^T \tanh(\mathbf{W}_{hh}^d \mathbf{h}_i^{d_1} + \mathbf{W}_{hh}^e \mathbf{h}_j^e + \mathbf{b}_a)$$

where $\mathbf{W}_{hh}^d \in \mathbb{R}^{k_h \times k_h}$, $\mathbf{W}_{hh}^e \in \mathbb{R}^{k_h \times 2k_h}$, $\mathbf{b}_a \in \mathbb{R}^{k_h}$, and $\mathbf{v} \in \mathbb{R}^{k_h}$. The attention context is obtained by the weighted linear combination of all the source hidden states: $\mathbf{c}_t = \sum_{j'=1}^{T^e} a_{t,j'} \mathbf{h}_{j'}^e$.

The final deterministic hidden state $\mathbf{h}_t^{d_2}$ is the output of the second decoder GRU layer, jointly considering the word $\mathbf{y}_{t-1}$, the previous hidden state $\mathbf{h}_{t-1}^{d_2}$, and the attention context $\mathbf{c}_t$:

$$\mathbf{h}_t^{d_2} = GRU_2(\mathbf{y}_{t-1}, \mathbf{h}_{t-1}^{d_2}, \mathbf{c}_t) \tag{15}$$

For the component of recurrent generative model, inspired by some ideas in previous works [7, 11, 4], we assume that both the prior and posterior of the latent variables are Gaussian, i.e., $p_\theta(\mathbf{z}_t) = \mathcal{N}(0, \mathbf{I})$ and $q_\phi(\mathbf{z}_t|\mathbf{y}_{<t}, \mathbf{z}_{<t}) = \mathcal{N}(\mathbf{z}_t; \boldsymbol{\mu}, \boldsymbol{\sigma}^2\mathbf{I})$, where $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ denote the variational mean and standard deviation respectively, which can be calculated via a multilayer perceptron. Precisely, given the word embedding $\mathbf{y}_{t-1}$, the previous latent structure variable $\mathbf{z}_{t-1}$, and the previous deterministic hidden state $\mathbf{h}_{t-1}^{d}$, we first project it to a new hidden space:

$$\mathbf{h}_t^{e_z} = g(\mathbf{W}_{yh}^{e_z}\mathbf{y}_{t-1} + \mathbf{W}_{zh}^{e_z}\mathbf{z}_{t-1} + \mathbf{W}_{hh}^{e_z}\mathbf{h}_{t-1}^{d} + \mathbf{b}_h^{e_z})$$

where $\mathbf{W}_{yh}^{e_z} \in \mathbb{R}^{k_h \times k_w}$, $\mathbf{W}_{zh}^{e_z} \in \mathbb{R}^{k_h \times k_z}$, $\mathbf{W}_{hh}^{e_z} \in \mathbb{R}^{k_h \times k_h}$, and $\mathbf{b}_h^{e_z} \in \mathbb{R}^{k_h}$. g is the sigmoid activation function: $\sigma(\mathbf{x}) = 1/(1 + e^{-\mathbf{x}})$. Then the Gaussian parameters $\boldsymbol{\mu}_t \in \mathbb{R}^{k_z}$ and $\boldsymbol{\sigma}_t \in \mathbb{R}^{k_z}$ can be obtained via a linear transformation based on $\mathbf{h}_t^{e_z}$:

$$\boldsymbol{\mu}_t = \mathbf{W}_{h\mu}^{e_z}\mathbf{h}_t^{e_z} + \mathbf{b}_\mu^{e_z}, \ \ \log(\boldsymbol{\sigma}_t^2) \ \ = \mathbf{W}_{h\sigma}\mathbf{h}_t^{e_z} + \mathbf{b}_\sigma^{e_z} \tag{16}$$

The latent structure variable $\mathbf{z}_t \in \mathbb{R}^{k_z}$ can be calculated using the reparameterization trick:

$$\boldsymbol{\varepsilon} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \ \ \mathbf{z}_t = \boldsymbol{\mu}_t + \boldsymbol{\sigma}_t \otimes \varepsilon \tag{17}$$

where $\boldsymbol{\varepsilon} \in \mathbb{R}^{k_z}$ is an auxiliary noise variable. The process of inference for finding $\mathbf{z}_t$ based on neural networks can be teated as a variational encoding process.

To generate summaries precisely, we first integrate the recurrent generative decoding component with the discriminative deterministic decoding component, and map the latent structure variable $\mathbf{z}_t$ and the deterministic decoding hidden state $\mathbf{h}_t^{d_2}$ to a new hidden variable:

$$\mathbf{h}_t^{d_y} = tanh(\mathbf{W}_{zh}^{d_y}\mathbf{z}_t + \mathbf{W}_{hh}^{d_z}\mathbf{h}_t^{d_2} + \mathbf{b}_h^{d_y}) \tag{18}$$

Given the combined decoding state $\mathbf{h}_t^{d_y}$ at the time t, the probability of generating any target word y_t is given as follows:

$$\mathbf{y}_t = \varsigma(\mathbf{W}_{hy}^{d}\mathbf{h}_t^{d_y} + \mathbf{b}_{hy}^{d}) \tag{19}$$

where $\mathbf{W}_{hy}^{d} \in \mathbb{R}^{k_y \times k_h}$ and $\mathbf{b}_{hy}^{d} \in \mathbb{R}^{k_y}$. $\varsigma(\cdot)$ is the softmax function. Finally, we use a beam search algorithm [8] for decoding and generating the best summaries

4.2 Learning

We use $\{X\}_N$ and $\{Y\}_N$ to denote the training source and target sequence. The final objective function - negative log-likelihood (NLL), which needs to be minimized is organized as follows under the model parameterized by Θ:

$$\min_{\Theta} \mathcal{J} = \frac{1}{N}\sum_{n=1}^{N}\sum_{t=1}^{T}\left\{ -\log\left[p(y_t^{(n)}|y_{<t}^{(n)}, X^{(n)})\right]\right\} \tag{20}$$

5 Experiments and Discussions

5.1 Datesets

We train and evaluate our framework on two popular datasets. For the image classification, we use MNIST[1] dataset. For headline generation, we use Gigawords, which is an English sentence summarization dataset prepared based on Annotated Gigawords[2] by extracting the first sentence from articles to create a source-summary pair. It roughly contains 3.8M training pairs, 190K validation pairs, and 2,000 test pairs.

[1]http://yann.lecun.com/exdb/mnist/

[2]https://catalog.ldc.upenn.edu/ldc2012t21

Table 1: Examples of the generated headlines. S: input source document. G: ground truth. P: generated abstractive headlines.

S(1): factory orders for manufactured goods rose #.# percent in september , the commerce department said here thursday. **G**: us september factory orders up #.# percent. **P**: **us factory orders up #.# percent in september.**
S(2): nick zito , who had three horses finish second during rival trainer d. wayne lukas ' streak of six straight victories in triple crown races , won the preakness on saturday with louis quatorze , who had finished ##th in the kentucky derby. **G**: UNK louis UNK wins preakness. **P**: **zito wins preakness for louis UNK.**
S(3): UNK thorpe remembers her difficult marriage to jim thorpe , called the greatest athlete of the modern era , and their harsh life outside the spotlight. **G**: late olympian 's wife recalls hard times <unk> photo available. **P**: **thorpe remembers her difficult marriage to thorpe.**

5.2 Experimental Settings

For the MLP and CNN framework, the hidden size is 500. For the experiments on the English dataset Gigawords, we set the dimension of word embeddings to 300, and the dimension of hidden states and latent variables to 500. The maximum length of documents and summaries is 100 and 50 respectively. The batch size of mini-batch training is 256. The beam size of the decoder was set to be 10. Our neural network based framework is implemented using Theano [1] on a single GTX 1080 GPU.

5.3 Performance of Different Gradient Methods

As shown in Figure 3 and Figure 4, we can see that the mentioned gradients methods perform different on various tasks in different neural network structures. For MLP, methods of momentum, adagrad, adam converge faster than the other methods. For CNN, adadelta also performs well. However, in the RNN model, the methods without tuning the step size, adadelta and adapg, perform the best. Considering that it will cost several weeks to finish the training state on some large scale tasks, less tuning of step size make the work more convenient.

5.4 Case Analysis for Headline Generation

As shown in Figure 3, we report the headlines generated by our own framework DRGD and the baseline methods under ROUGE (overlapping between prediction and ground truth) metric. We can see that our new neural network based framework with adadelta training methods outperform all the baselines. As shown in Table 1, we select several samples cases to demonstrate the performance of our framework. From the cases we can see that our framework can generate headlines with a good linguistic quality.

6 Conclusions

In the investigations we compare the performance of different gradient methods in deep learning and find that the structure of the meodels will affect the choice of the methods.

References

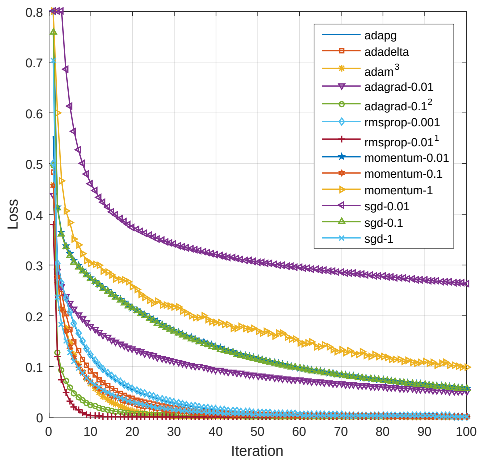
[1] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[2] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

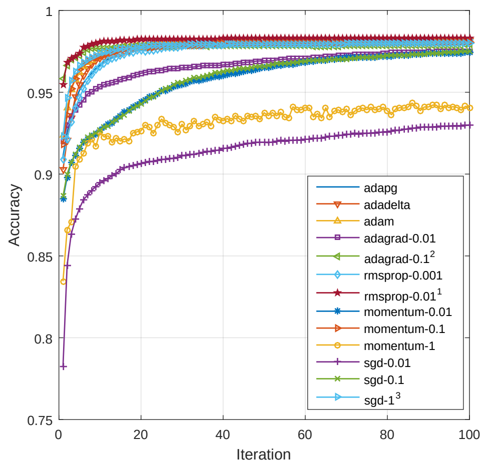
[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT Press, 2016.

[4] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. In *ICML*, pages 1462–1471, 2015.

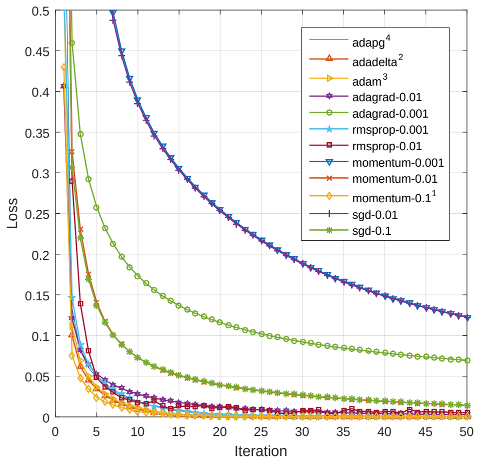
[5] Geoffrey Hinton, N Srivastava, and Kevin Swersky. Lecture 6a overview of mini–batch gradient descent. *Coursera Lecture slides https://class. coursera. org/neuralnets-2012-001/lecture,[Online*, 2012.



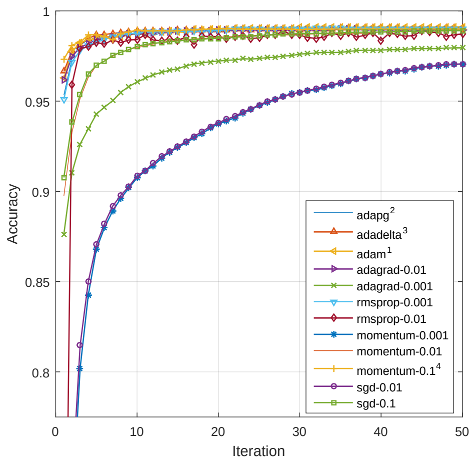
(a) MLP: objective value.



(b) MLP: accuracy.



(c) CNN: objective value.



(d) CNN: accuracy.

Figure 3: Performance of different gradient methods in MLP and CNN framework.

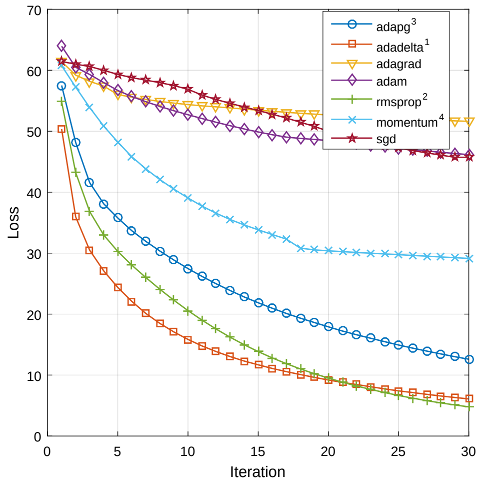


Figure 4: RNN: objective value.

System	R-1	R-2	R-L
ABS	29.55	11.32	26.42
ABS+	29.78	11.89	26.97
RAS-LSTM	32.55	14.70	30.03
RAS-Elman	33.78	15.97	31.15
ASC + FSC$_1$	34.17	15.94	31.92
lvt2k-1sent	32.67	15.59	30.64
lvt5k-1sent	35.30	16.64	32.62
DRGD	**36.27**	**17.57**	**33.62**

Figure 5: ROUGE-F1 on Gigawords

[6] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[7] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. In *ICLR*, 2014.

[8] Philipp Koehn. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*, pages 115–124. Springer, 2004.

[9] 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.

[10] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/k2). In *Doklady an SSSR*, volume 269, pages 543–547, 1983.

[11] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *ICML*, pages 1278–1286, 2014.

[12] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[13] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

[14] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[15] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.