CS544 – Project

Yuxuan Zhou
yuxuanz6@illinois.edu

05/10/2019

1 Introduction

As a deep learning engineer, we'd like to promote the efficiency of our deep networks.The goal of this project is to study the performance about different optimizers in object classification task. The dataset here is based on CIFAR100, which contains 100 different image class and each class contains 600 images(60000=100*600). Therefore, I choose the image train set to be 45000(100* 450),validation set to be 5000(100*50) and test set to be 10000(100*100). I use 10 different types of optimizers in pytorch.optim, which contains most optimizer types. I use two popular deep networks in this project: one is adapted from vggnet-16 and I call it vggnet-16', another one is resnet-18. I record the train and test accuracy, loss and running time for those optimizers, and compared them from their default parameters settings to the suitable parameters setting in real lab. I'll analysis these lab results based on their theoretical math analysis and compare their performance.

2 Implementation

2.0 types of optimizers

I use the pytorch package in this lab. However, different software packages have different parameters settings. Take the adam optimizer for example, following shows the difference:
TensorFlow: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08.
Keras: lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0.
Caffe: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-08
Torch: learning_rate=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8

Table 1. In the following shows the settings I use in this lab, include their explain.

Table 1. laboratory optimizers

Optimizer Type	**Explain**
optim.SGD(net.parameters(),lr=1e-3)	SGD suitable learning rate lr=1e-3 for classification
optim.SGD(net.parameters(), lr=0.001, momentum=0.9)	SGD suitable learning rate lr=0.001, add momentum setting to make it more efficient
optim.ASGD(net.parameters(), lr=1e-3)	ASGD suitable learning rate lr=1e-3 for classification
optim.Adam(net.parameters(), lr=1e-3)	Adam suitable learning rate lr=1e-3 for classification
optim.Adadelta(net.parameters(), lr=1.0, rho=0.9, eps=1e-06, weight_decay=0) # default	Use the default Adadelta parameters setting
optim.Adagrad(net.parameters(), lr=0.01, lr_decay=0, weight_decay=0, initial_accumulator_value=0) # default	Use the default Adagrad parameters setting
optim.Adamax(net.parameters(), lr=0.002, betas=(0.9, 0.999), eps=1e-08, weight_decay=0) # default	Use the default Adamax parameters setting
optim.RMSprop(net.parameters(), lr=0.01, alpha=0.99, eps=1e-08, weight_decay=0, momentum=0, centered=False) # default	Use the default RMSprop parameters setting
optim.ASGD(net.parameters(), lr=0.01, lambd=0.0001, alpha=0.75, t0=1000000.0, weight_decay=0) # default	Use the default ASGD parameters setting
optim.Rprop(net.parameters(), lr=0.01, etas=(0.5, 1.2), step_sizes=(1e-06, 50)) # default	Use the default Rprop parameters setting

2.1 theoretical math analysis

Although I have 10 optimizers, I classify them into 4 categories:

1. SGD related(SGD,SGD+momentum,ASGD_lab,ASGD_default)
2. Adam related (Adam, Adamax)
3. Adagrad related (Adagrad, Adadelta)
4. Rprp related(Rprop, RMSprop)

In each category, there is one modified from the basic one and own more advanced property. Please note that most my optimiers are based on first order gradient descent method and Newton's Method and BFGS, which is required to calculate second order partial derivative matrix or Hessian matrix and a lot memory. Beside make them into 4 categories, we could also make them into 2 - SGD mini batch methods and the rest are adaptive learning methods.

2.1.1 SGD related

First, let me introduce SGD, which standards for Stochastic Gradient Descent. When training input is very large, gradient descent is quite slow to converge. Stochastic Gradient Descent is the preferred variation of gradient descent which estimates the gradient from a small sample of randomly chosen training input in each iteration called minibatches.

Gradient descent:

$$\theta = \theta - \alpha \nabla_\theta E[J(\theta)]$$ ………… (1)

where the expectation in the above equation (1) is approximated by evaluating the cost and gradient over the full training set. Stochastic Gradient Descent (SGD) simply does away with the expectation in the update and computes the gradient of the parameters using only a single or a few training examples. The new update is given by (2) with a pair (x(i),y(i)) from the training set.

Stochastic gradient descent:

$$\theta = \theta - \alpha \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$ ………… (2)

Every complete exposure of the training dataset is called epoch. The SGD algorithm iterates for a given number of epochs. Batch gradient descent performs redundant computations for large datasets, as it recomputes gradients for similar examples before each parameter update. SGD does away with this redundancy by performing one update at a time. It is therefore usually much faster and efficient.

Momentum:

$$\begin{aligned} v &= \gamma v + \alpha \nabla_\theta J(\theta; x^{(i)}, y^{(i)}) \\ \theta &= \theta - v \end{aligned}$$ ………..(3)

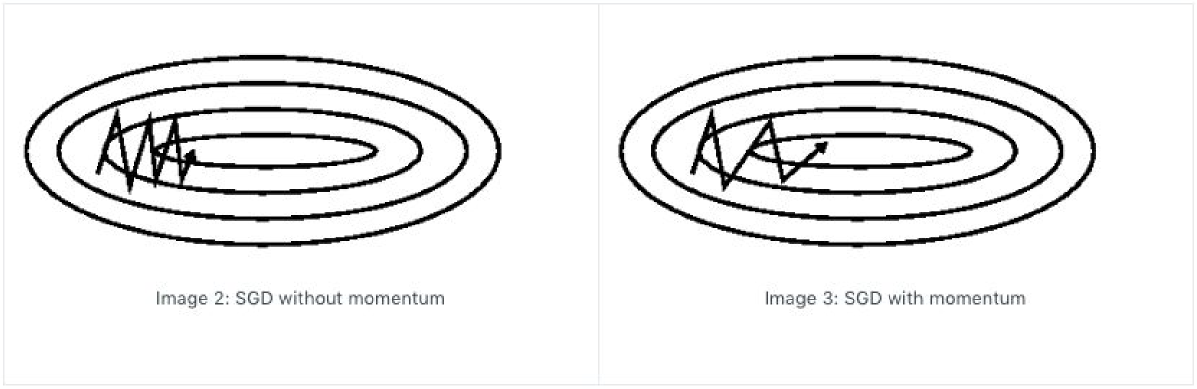


Figure 17. [10] SGD momentum

Momentum technique is an approach which provides an update rule that is motivated from the physical perspective of optimization. As Figure 17. shown above, Imagine we have a ball in a hilly terrain is trying to reach the deepest valley. When the slope of the hill is very high, the ball gains a lot of momentum and is able to pass through slight hills in its way. As the slope decreases the momentum and speed of the ball decreases, eventually coming to rest in the deepest position of valley. [10]

Some implementations exchange the signs in the equations. The momentum term γ is usually set to 0.9 or a similar value. Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. γ<1). The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation. [10]

ASGD:

$$\bar{\theta}_n = \frac{1}{n}\sum_{i=1}^{n}\theta_i \quad ....(4)$$

$$\theta_{n+1} = \theta_n - [\nabla^2 Q(\theta_n)]^{-1}\nabla Q_{i_n}(\theta_n) \quad ....(5)$$

Define the averaged stochastic gradient descent (ASGD) as applying SGD with Polyak-Ruppert averaging. The simplest variant which improves SGD use Polyak-Ruppert averaging. As function(4)(5) shown above, while the convergence rate of a second-order SGD is still sublinear, the constants are much improved. This is reflected in practice, and it comes at very little cost due to the simplicity of the small change in implementation. In my lab, sometime ASGD shows the promote efficiency and the run time. I'll analysis it based on my results.

2.1.2 Adam related

Then, we move forward into Adam, which is called Adaptive Moment Estimation. From now on, all the optimers are adaptive learning rate. We could adaptively tune the learning throughout the training phases and know which direction to accelerate and which to decelerate.

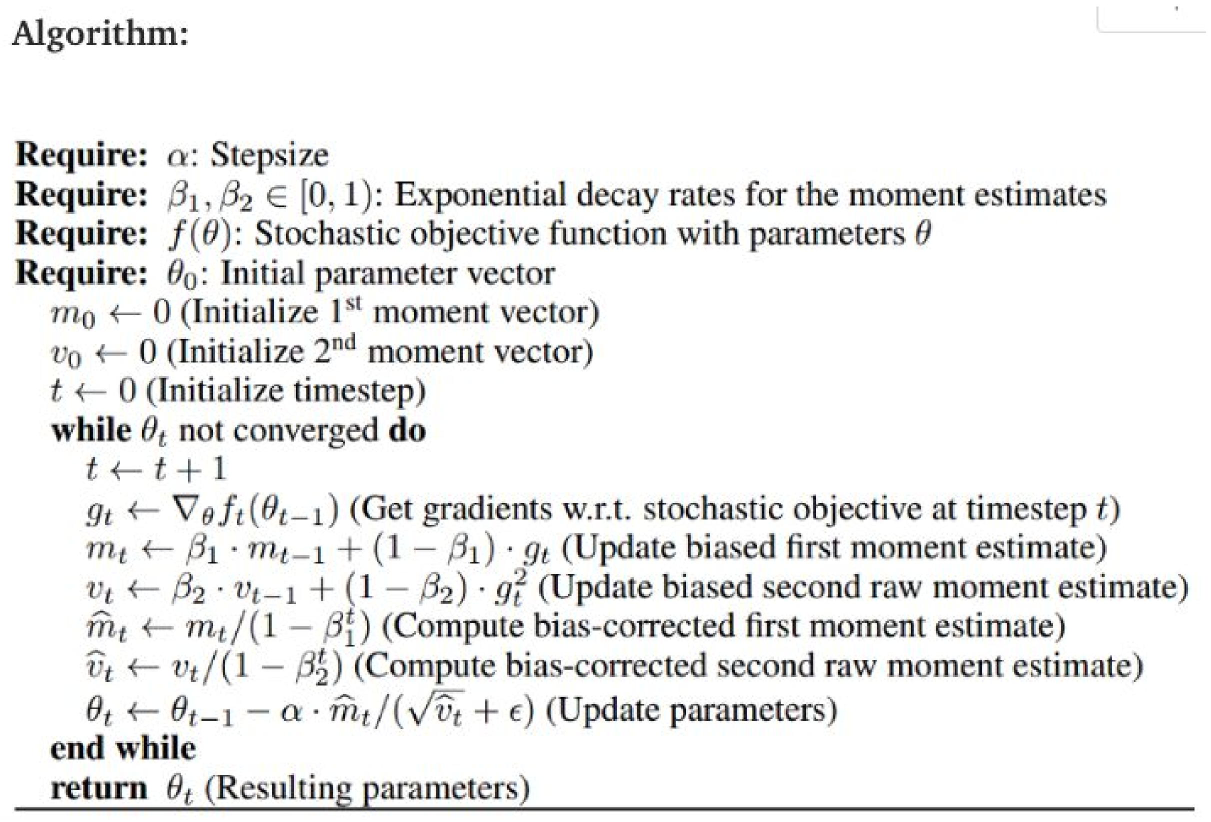
Adam:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \quad \text{....(6)}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \quad \text{....(7)}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t. \quad \text{...(8)}$$

- β1 — This is used for decaying the running average of the gradient (0.9)
- β2 — This is used for decaying the running average of the square of gradient (0.999)
- α — Step size parameter (0.001)
- ε- It is to prevent Division from zero error. (10^-8)



```
Algorithm:

Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t/(1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t/(1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t/(\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

Figure 18. [11] Adam Implement Algorithm

The above formulas(6)(7)(8) and Figure 18. shows the details about Adam, which proves to be very efficient and typical optimizer in adaptive learning rate optimizers.

Adamax:

supposed to be used when you’re using some setup that has sparse parameter updates (ie word embeddings). This is because the |g_t| term is essentially ignored when it’s small. This means that parameter updates u_1…u_n are influenced by less gradients, and therefore less susceptible to gradient noise.

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)|g_t|^2 \qquad \ldots(9)$$

$$v_t = \beta_2^p v_{t-1} + (1-\beta_2^p)|g_t|^p \qquad \ldots(10)$$

$$\begin{aligned} u_t &= \beta_2^\infty v_{t-1} + (1-\beta_2^\infty)|g_t|^\infty \\ &= \max(\beta_2 \cdot v_{t-1}, |g_t|) \end{aligned} \qquad \ldots(11)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{u_t}\hat{m}_t \qquad \ldots(12)$$

Compared with Adam, we supposed the |g_t| term is ignored when it’s small(9)(10)(11), and (12) shows instead of (8), we use u_t here when has sparse parameter updates. Therefore, when the training is about sparse parameter updates such as word embeddings, we prefer to use Adamax which would be more efficient than Adam.

2.1.3 Adagrad related

Thirdly, I’ll talk about Adagrad(Adaptive Gradient Algorithm), which is a theoretically sound method for learning rate adaptation which has has the advantage of being particularly simple to implement. The learning rate is adapted component-wise, and is given by the square root of sum of squares of the historical, component-wise gradient as the following(13).

Adagrad:

$$\begin{aligned} s_{t+1} &= s_t + (\nabla\mathcal{L})^2 \\ W_{t+1} &= W_t - \frac{\alpha \cdot \nabla\mathcal{L}}{\sqrt{s_{t+1} + \epsilon}} \end{aligned} \qquad \ldots\ldots\ldots\ldots\ldots(13)$$

Adadelta:

a slight improvement over AdaGrad that fixes a few things, a method that uses the magnitude of recent gradients and steps to obtain an adaptive step rate(14). An exponential moving average over the gradients and steps is kept(15)(16)(17)(18); a scale of the learning rate is then obtained by their ration(19).

$$\lambda_{t+1} = \lambda_t \cdot \mu + (1-\mu) \cdot (\nabla \mathcal{L})^2$$
$$\Delta W_{t+1} = \nabla \mathcal{L} \cdot \sqrt{\frac{\delta_t + \epsilon}{\lambda_{t+1} + \epsilon}}$$
$$\delta_{t+1} = \delta_t \cdot \mu + (1-\mu) \cdot (\Delta W_{t+1})^2$$
$$W_{t+1} = W_t - \Delta W_{t+1}$$
….(14)

$$E[g^2]_t = \gamma E[g^2]_{t-1} + (1-\gamma) g_t^2.$$
......(15)

$$\Delta\theta_t = -\eta \cdot g_{t,i} \qquad \theta_{t+1} = \theta_t + \Delta\theta_t$$ => $$\Delta\theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.$$ => $$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$ => $$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t$$ ...(16)

$$E[\Delta\theta^2]_t = \gamma E[\Delta\theta^2]_{t-1} + (1-\gamma)\Delta\theta_t^2$$
……(17)

$$RMS[\Delta\theta]_t = \sqrt{E[\Delta\theta^2]_t + \epsilon}.$$
.......(18)

$$\Delta\theta_t = -\frac{RMS[\Delta\theta]_{t-1}}{RMS[g]_t} g_t$$
$$\theta_{t+1} = \theta_t + \Delta\theta_t$$
.....(19)

2.1.4 Rprop related

Last, let’s focus on the Rprop. First we skip Rprop and looks at RMSprop - Root Mean Square Propagation. It has advanced property than Rprop and more reliable in most case. RMSprop optimizer is similar to the gradient descent algorithm with momentum. The RMSprop optimizer restricts the oscillations in the vertical direction. Therefore, we can increase our learning rate and our algorithm could take larger steps in the horizontal direction converging faster. The difference between RMSprop and gradient descent is on how the gradients are calculated(20). The following equations show how the gradients are calculated for the RMSprop and gradient descent with momentum(21)(22). The value of momentum is denoted by beta and is usually set to 0.9(21). If you are not interested in the math behind the optimizer, you can just skip the following equations.

RMSprop:

$$s_{t+1} = \mu \cdot s_t + (1 - \mu) \cdot (\nabla \mathcal{L})^2$$
$$W_{t+1} = W_t - \frac{\alpha \cdot \nabla \mathcal{L}}{\sqrt{s_{t+1} + \epsilon} + \epsilon}$$

.....(20)

$$E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2$$ ......(21)

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t$$ ......(22)

Rprop:

A learning algorithm for multilayer feedforward networks, RPROP (resilient propagation). To overcome the inherent disadvantages of pure gradient-descent, RPROP performs a local adaptation of the weight-updates according to the behavior of the error function. Contrary to other adaptive techniques, the effect of the RPROP adaptation process is not blurred by the unforeseeable influence of the size of the derivative, but only dependent on the temporal behavior of its sign. One of the key differences between RPROP and standard back-prop is that in RPROP each weight and bias has a different, variable, implied learning rate — as opposed to standard back-prop which has one fixed learning rate for all weights and biases. Put differently, with RPROP you don't specify a learning rate. Instead, each weight has a delta value that increases when the gradient doesn't change sign (meaning you're headed in the correct direction) or decreases when the gradient does change sign.In high-level pseudo-code, for one pass through the training data, RPROP is something like:

```
for each weight and bias
  if prev grad and curr grad have same sign
    increase the previously used delta
    update weight using new delta
  else if prev and curr have different signs
    decrease the previously used delta
    revert weight to prev value
  end if
  prev delta = new delta
  prev gradient = curr gradient
end-for
```

I suspect that even though RPROP is often more effective than regular back-prop, RPROP isn't used very often because it's very tricky to implement. In a lot cases, it will be useless.

2.1.5 Optimiers Visualizon

Figure 18.[10] shows different optimizers in surface contours and saddle points.

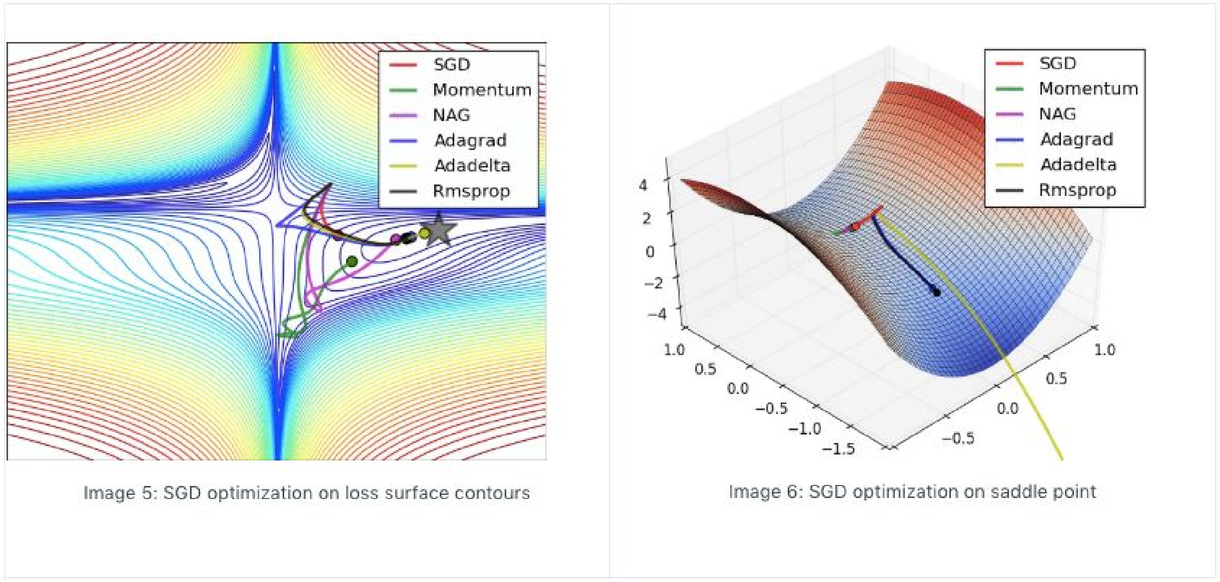
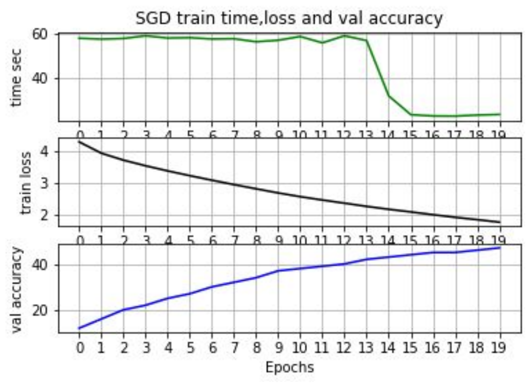
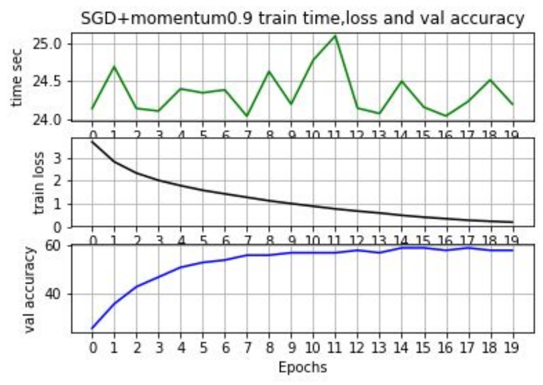
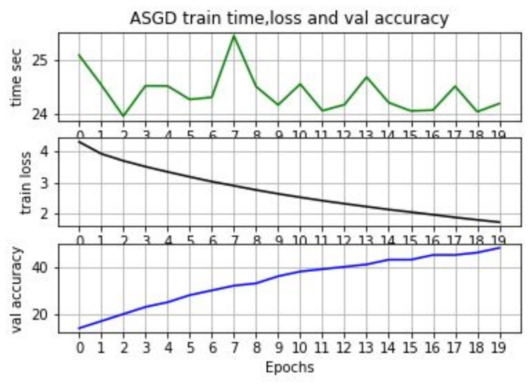
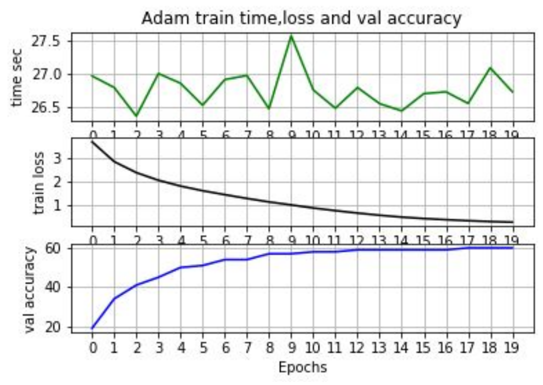
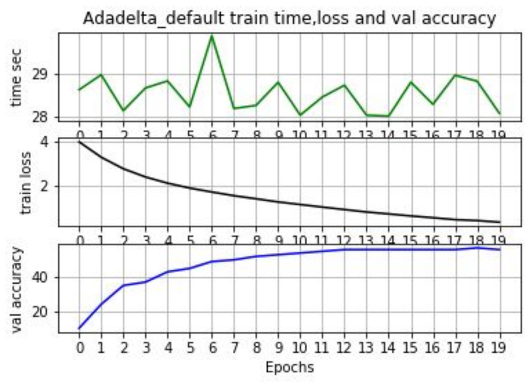
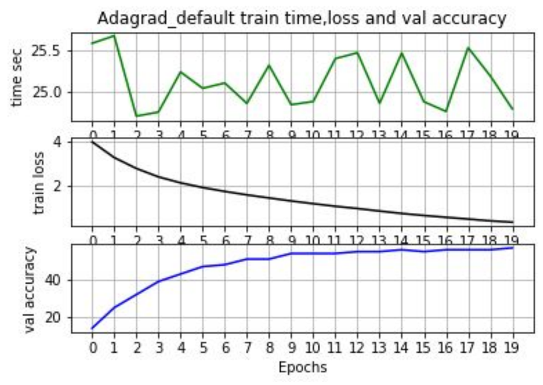


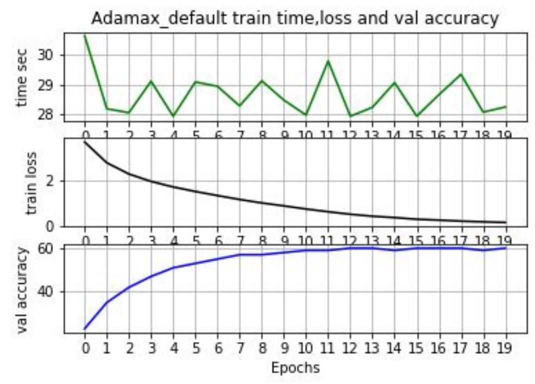
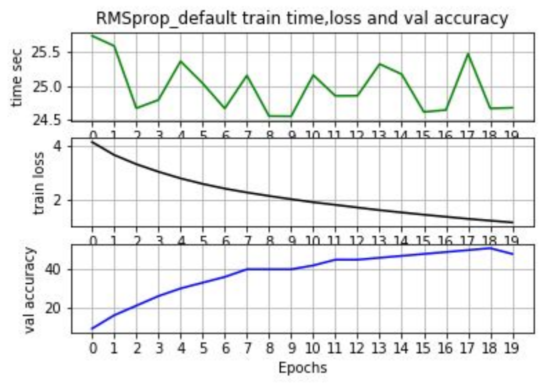
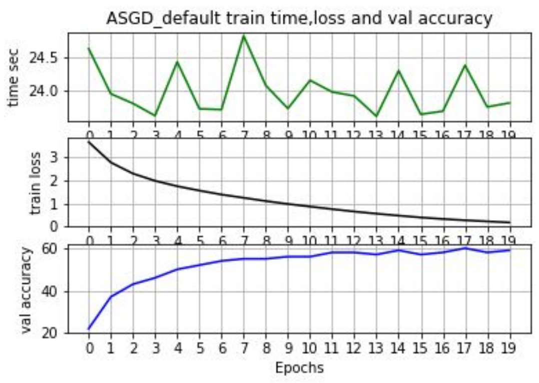
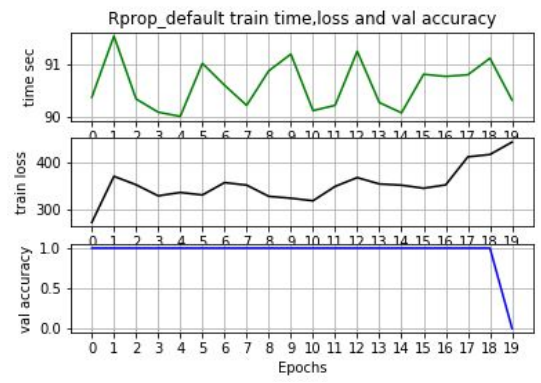
Figure 18. [10] Optimiers Visualizon

3 Results and Analysis

3.1.0 Vggnet-16’ single optimizer train and validation

Table 2. vggnet-16’ optimizer train loss, validation accuracy, time cost

The above Table 2. shows all my optimers results towards vggnet-16' and except the Rprop is useless, the other 9 work fine, in the next following pages, I would also exclude Rprop and compare the other 9 optimiers.

The ASGD_default and SGD+momentum in SGD optimizer group obtain over 60% accuracy.

The Adam and Adamax_default in Adam optimizer group obtain over 60% accuracy. The other 5 obtain from 40% to 60%.

The Adamax_default obtains less than 1% loss. The ASGD_default, SGD+momentum and the Adam obtains less than 0.5% loss. No others are less than 0.5%.

The Adamax_default's running time is 28-29 sec/epoch; ASGD_default's running time is 23.5-24.5 sec/epoch; SGD+momentum's running time is 24-24.5 sec/epoch; Adam's running time is 26.5-27 sec/epoch; No others are less than 24.5 sec/epoch.

The interesting thing here is when the optimer can keep a high accuracy, which will also keep less loss and less running time. Combine them together, the best one for this task is ASGD_default : accuracy >60%, loss < 0.5%, run time 23.5 - 24.5 sec/ epoch

3.1.1 Vggnet-16’ combine optimizers loss

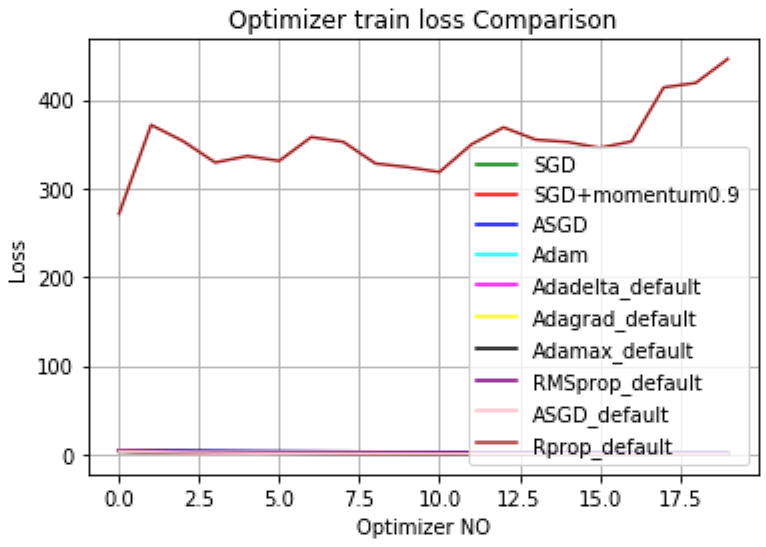


Figure 1. vggnet-16’ optimizers train loss/epoch with Rprop

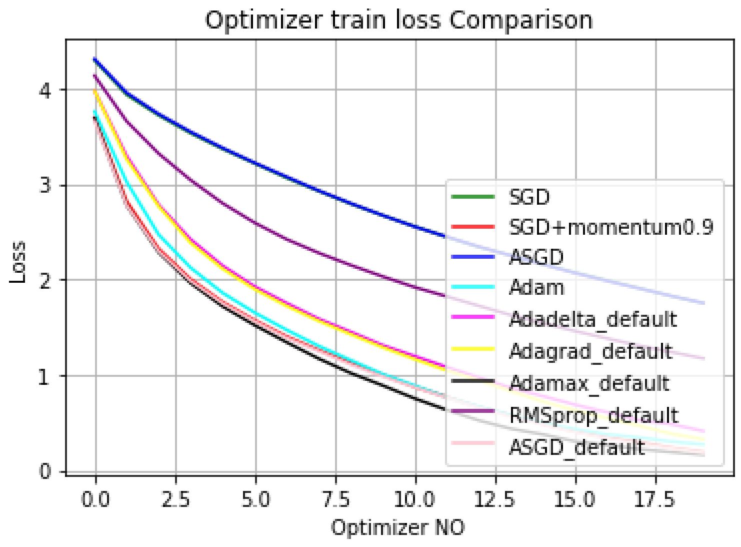


Figure 2. vggnet-16’ optimizers train loss/epoch without Rprop

Figure 1. contains the Rprop optimizer, which doesn’t work and thus others all not distinguish. Figure 2. excludes the Rprop one and compare the rest 9 towards the vggnet-16’ in train loss. We could find out Adamax_default(black one) owns the least loss.ASGD(blue) lab one is most error one. ASGD_default(pink) would be much better. Except the ASGD lab one and the RMSprop, others could achieve loss less than 1% eventually which are similar.

3.1.2 Vggnet-16’ combine optimizers accuracy

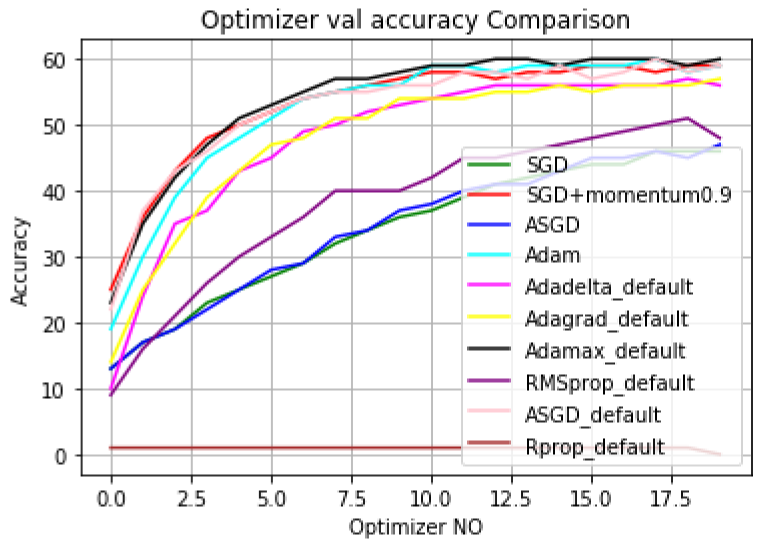


Figure 3. vggnet-16’ optimizers validation accuracy/epoch with Rprop

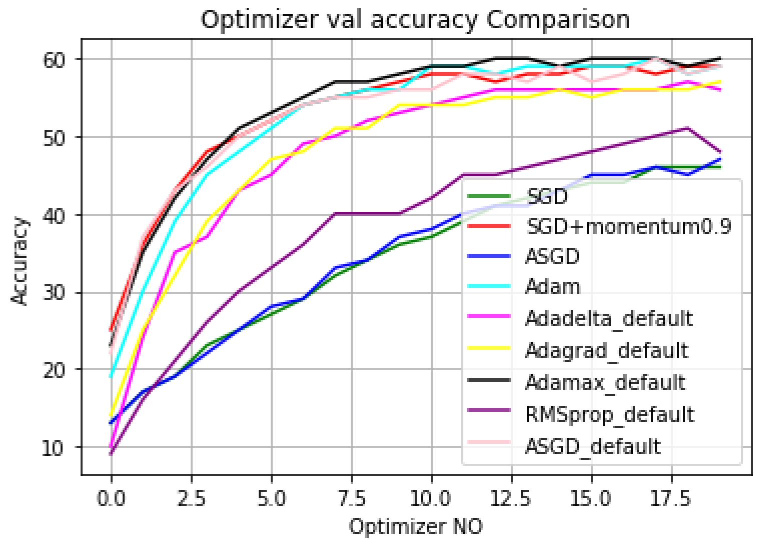


Figure 4. vggnet-16’ optimizers validation accuracy/epoch without Rprop

Figure 3. contains the Rprop optimizer, which doesn’t work and the accuracy equals near zero. Figure 4. excludes the Rprop one and compare the rest 9 towards the vggnet-16’ the validation accuracy. We could find out Adamax_default(black one) owns the most accuracy. Here we could know the final accuracy from the Figure 4. more accurate than the initial analysis from

Table 2. The Adamax_default(black), SGD+momentum0.9(red) and ASGD_default(pink) are very similar from the beginning and last Adamax_default(black) wins the highest accuracy. The SGD(green), ASGD lab(blue) and RMSprop(purple) are less accuracy than others, which are easy to tell less than 52%. Except these 3 ones, others could achieve about 60% eventually.

3.1.3 Vggnet-16' combine optimizers time cost

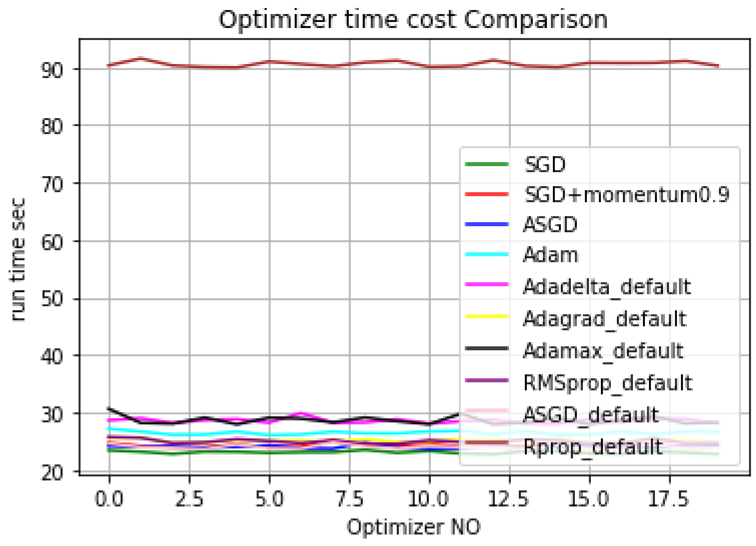


Figure 5. vggnet-16' optimizers time cost/epoch with Rprop

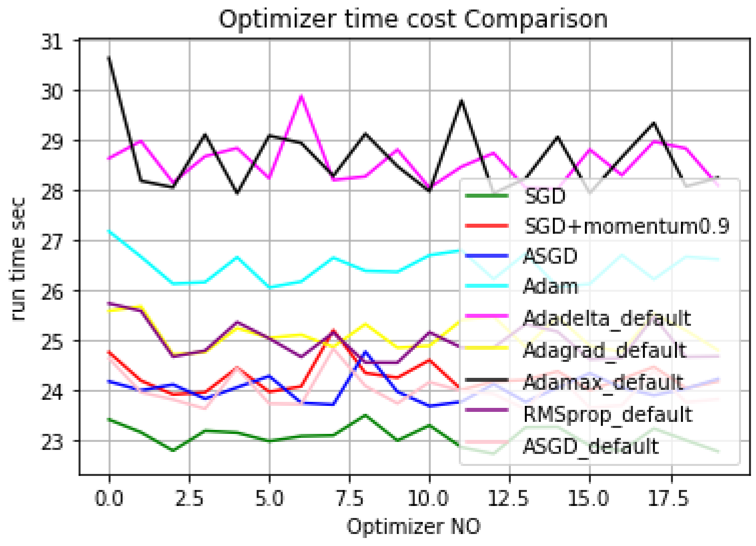


Figure 6. vggnet-16' optimizers time cost/epoch without Rprop

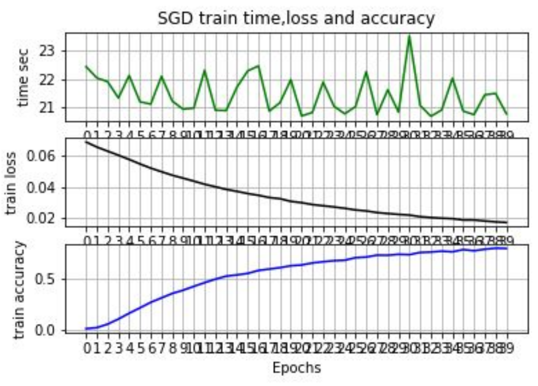
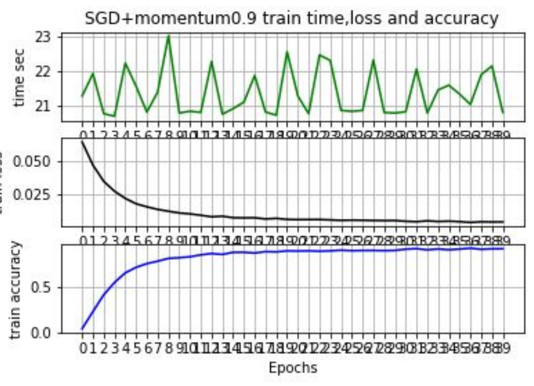
Figure 5. contains the Rprop optimizer, which is wrong and the highest time cost than the others Figure 6. excludes the Rprop one and compare the rest 9 towards the vggnet-16' the run time. We could find out the SGD(green) runs the least time than others, around 23 sec/ epoch. The Adamax_default(black) and RMSprop(purple) are most time cost, around 28-29 sec/epoch.
From the Figure 4. Analysis, though the Adamax_default(black) win the highest accuracy, it's time cost also highest. SGD+momentum0.9(red) and ASGD_default(pink) are very similar win the second and third highest accuracy and time cost is 24 sec/epoch which is near the least one 23 sec/epoch. For the loss, they are similar.

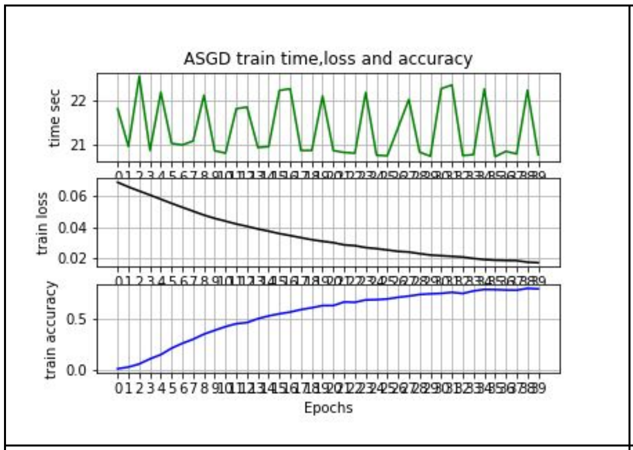
3.1.4 Vggnet-16' results sum up

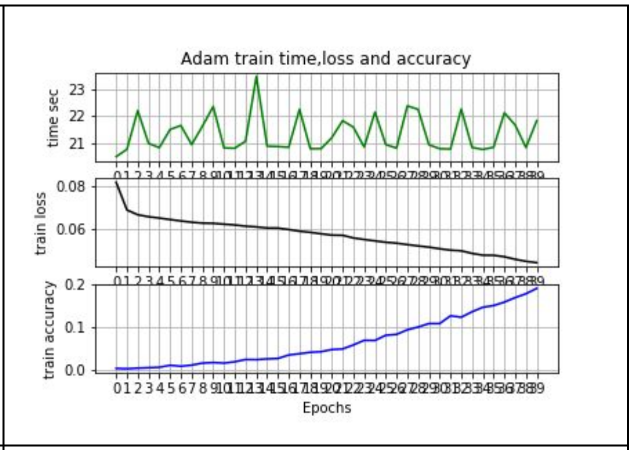
To sum up Figure 1,2,3,4,5,6. , SGD+momentum0.9(red) and ASGD_default(pink) are the best choice if we care about time cost. If not, then Adamax_default(black) is the best choice. The RMSprop(purple) and the Rprop(brown) are not suitable to finish such task or we need to do more lab to adjust their parameter setting to fit such task. The parameter setting of ASGD lab one is not well adjusted at all because the ASGD_default is supposed to be the best choice.
Use the math analysis in chapter 2.1, we find SGD group optimiers are better choice than adaptive learning rate group optimiers. I think one reason is when the input data is large, well labeled and organized, SGD group optimizers would keep stable learning rate and less time cost for it needn't change learning rate and less tricky math calculations, memory cost etc.

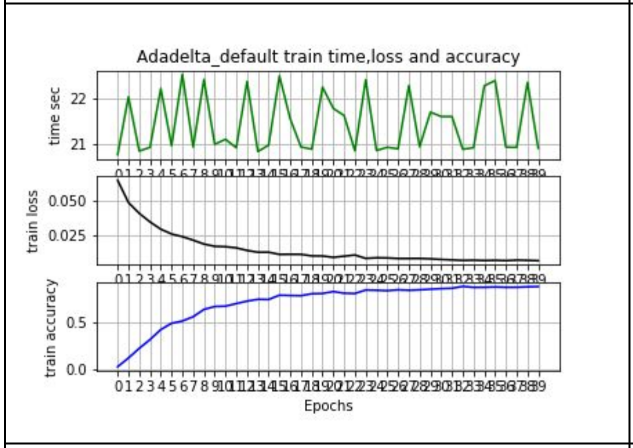
3.2.0 Resnet-18 single optimizer train and validation

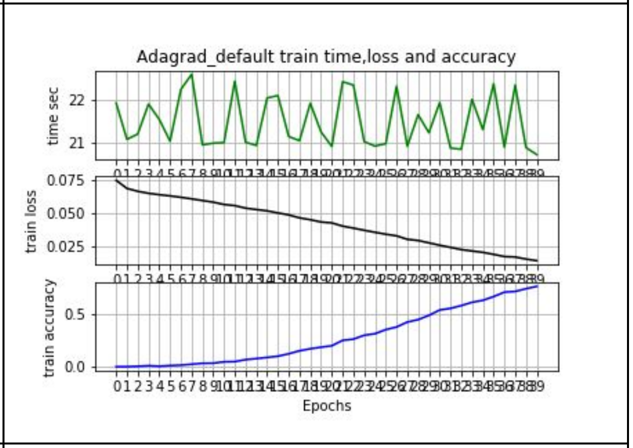
Table 3. Resnet-18 optimizer train loss, accuracy, time cost

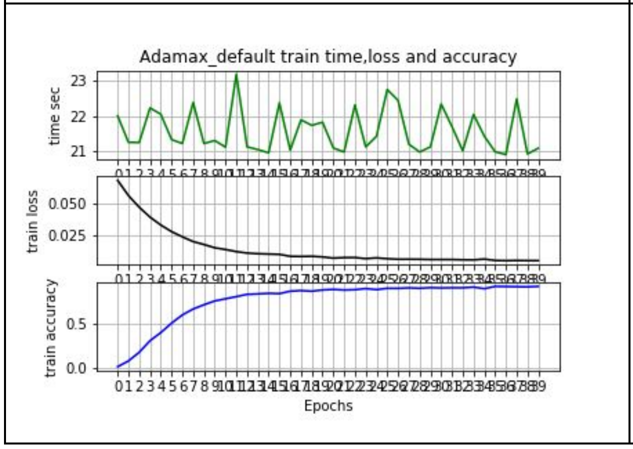
	

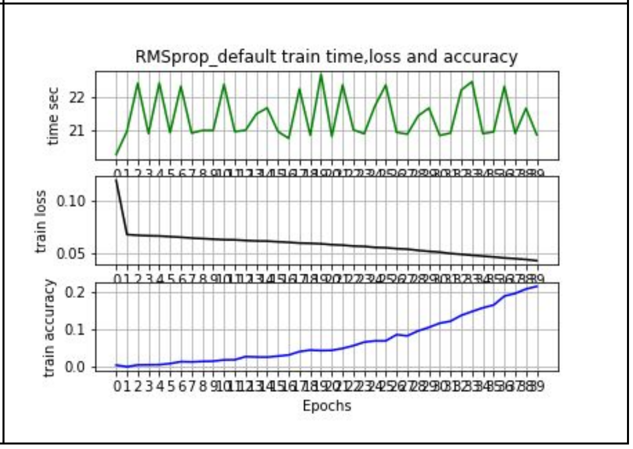

ASGD train time,loss and accuracy
time sec
train loss
train accuracy
Epochs

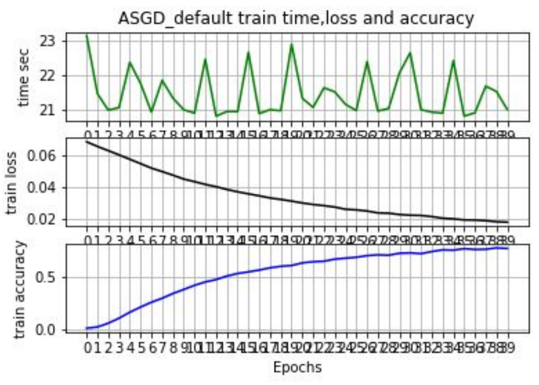
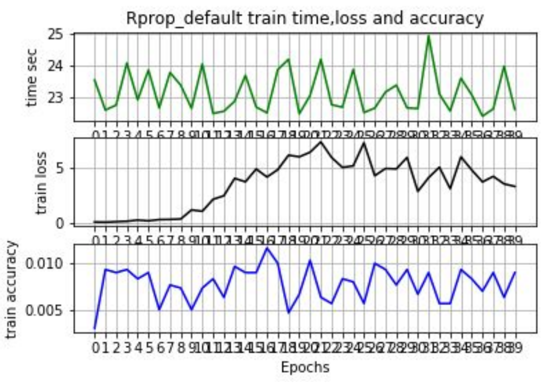

Adam train time,loss and accuracy
time sec
train loss
train accuracy
Epochs


Adadelta_default train time,loss and accuracy
time sec
train loss
train accuracy
Epochs


Adagrad_default train time,loss and accuracy
time sec
train loss
train accuracy
Epochs


Adamax_default train time,loss and accuracy
time sec
train loss
train accuracy
Epochs


RMSprop_default train time,loss and accuracy
time sec
train loss
train accuracy
Epochs

The above Table 3. shows all my optimers results towards resnet-18 and except the Rprop is useless, the other 9 work fine, in the next following pages, I would also exclude Rprop and compare the other 9 optimiers.

The SGD+momentum, ASGD, Adamax_default, Adadelta_default obtain over 90% accuracy. The SGD, ASGD_default and Adagrad_default obtain 80% - 90% accuracy. The Adam and the RMSprop obtain around 20%. However, they are not converge and also plus Adagrad_default.

The SGD+momentum , Adamax_default and Adadelta_default obtains less than 0.5% loss. The ASGD obtains less than 1% loss. No others are less than 0.5%.

The Adamax_default's running time is 21-22 sec/epoch,max 23 sec/epoch; Adadelta_default 's running time is 21-22 sec/epoch, max 22.8 sec/epoch, many over 22 sec; SGD+momentum's running time is 21-22 sec/epoch,max 23 sec/epoch; No others are less than 21 sec/epoch.

The interesting thing here is when the optimer can keep a high accuracy, which will also keep less loss and less running time. 3 optimizers doesn't converge completely.

Combine the rest 6 optimizers together, the best one for this task is Adadelta_default: accuracy > 90%, loss < 0.5%, run time 21 - 22 sec/ epoch, max 22.5 sec/epoch. The second and third ones are Adamax_default and SGD+momentum0.9: accuracy > 90%, loss < 0.5%, run time 21 - 22 sec/ epoch,max 23 sec/epoch.

3.2.1 Resnet-18 combine optimizers accuracy

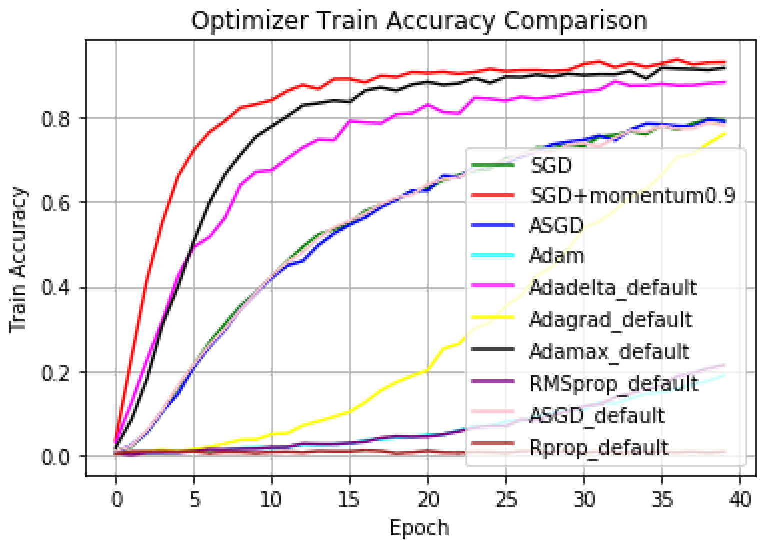


Figure 7. resnet-18 optimizer train accuracy with Rprop

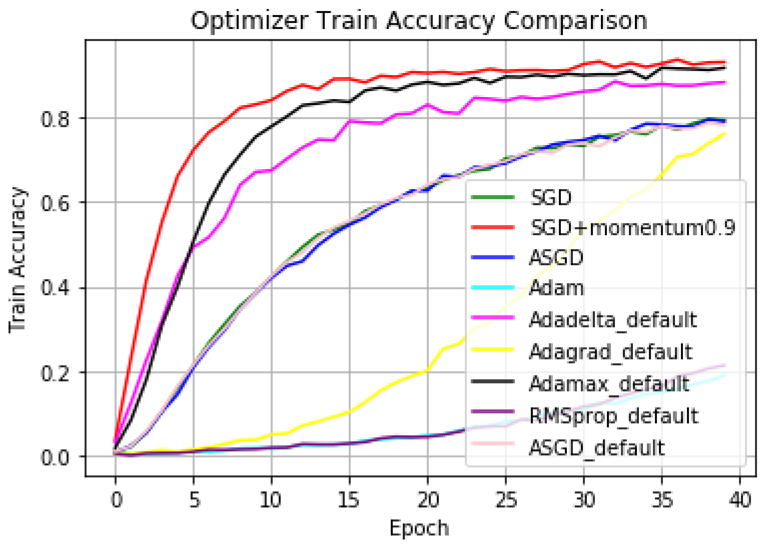


Figure 8. resnet-18 optimizer train accuracy without Rprop

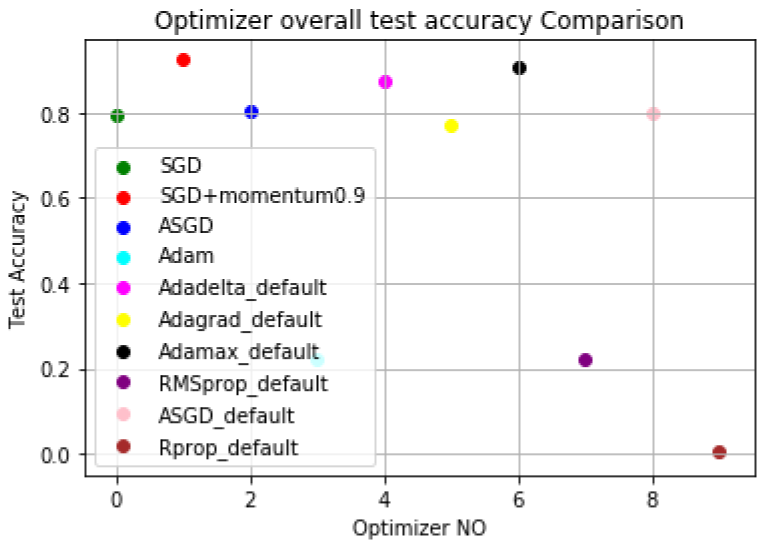


Figure 9. resnet-18 optimizer test accuracy

Figure 7. contains the Rprop optimizer, which is wrong and has accuracy around 1%.Figure 8. excludes the Rprop one and compare the rest 9 towards the resnet-18 the train accuracy. And Figure 9. contains 10 optimizers towards the resnet-18 the test accuracy.

We could find out during the train time the SGD+momentum0.9(red) owns the highest accuracy near 90%. The Adamax_default(black) and Adadelta_default(purple) are the second and third one, over 85%. During the test time, the result are the same as the train time, also we could see clearly that the Rprop one is 0.0 from Figure 9. Beside the Rprop, the adam and RMSprop_default are less accurate which are 20%. Compared with Table 3., we know they are not converged at this time and thus we couldn't judge their final performance here.

3.2.2 Resnet-18 combine optimizers loss

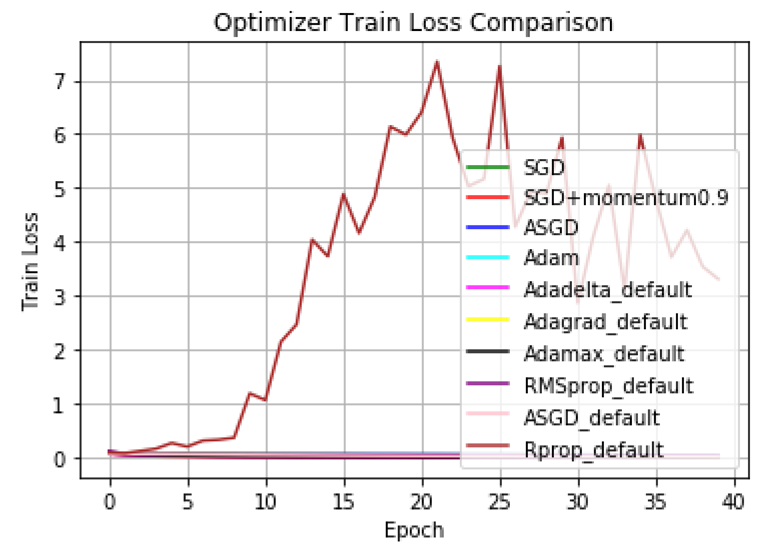


Figure 10. resnet-18 optimizers train loss with Rprop

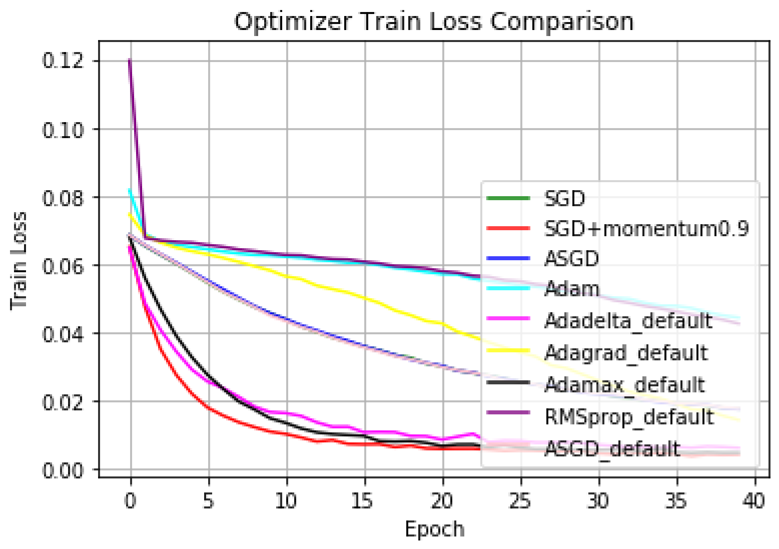


Figure 11. resnet-18 optimizers train loss without Rprop

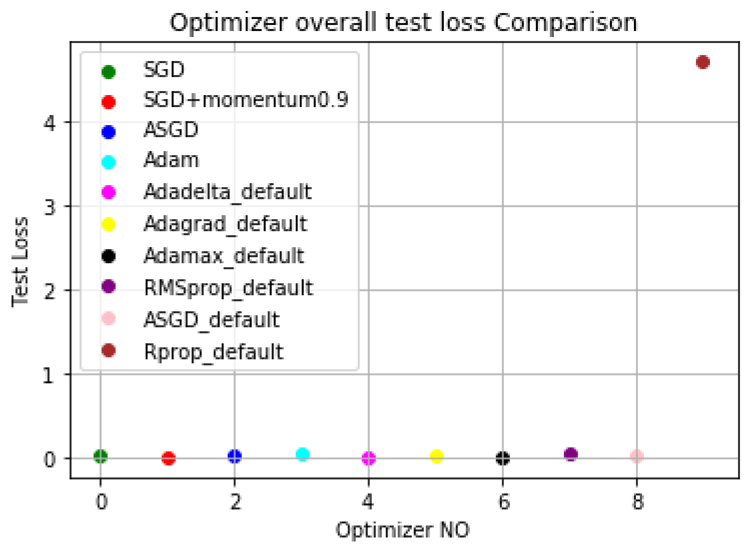


Figure 12. resnet-18 optimizer test loss

Figure 10. contains the Rprop optimizer, which is wrong and has loss over 100%.Figure 11. excludes the Rprop one and compare the rest 9 towards the resnet-18 the train loss. And Figure 12. contains 10 optimizers towards the resnet-18 the test loss.

We could find out during the train time the SGD+momentum0.9(red) owns the least loss less than 0.5%. The Adamax_default(black) and Adadelta_default(purple) are the second and third one, less than 1%. During the test time, the result are the same as the train time, also we could see clearly that the Rprop one is over 400% from Figure 12. Beside the Rprop, the adam and RMSprop_default are less accurate which are around 4%. Compared with Table 3., we know they are not converged at this time and thus we couldn't judge their final performance here.

3.2.3 Resnet-18 combine optimizers time cost

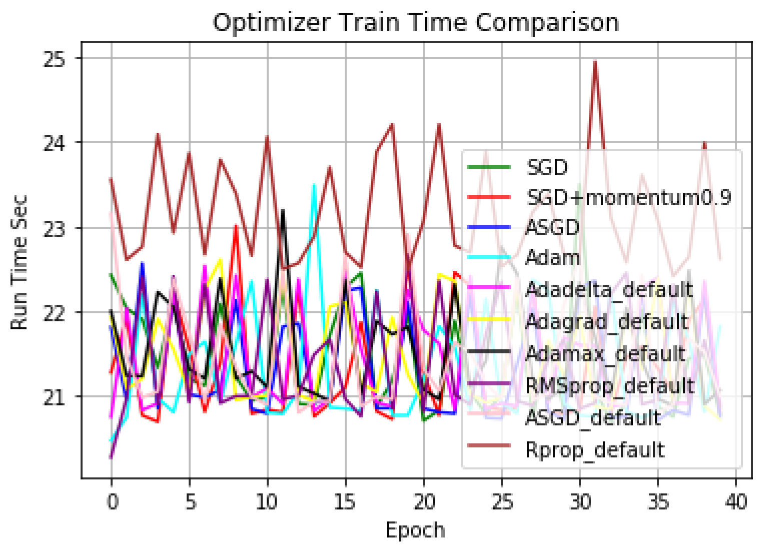


Figure 13. resnet-18 optimizer train time per epoch with Rprop

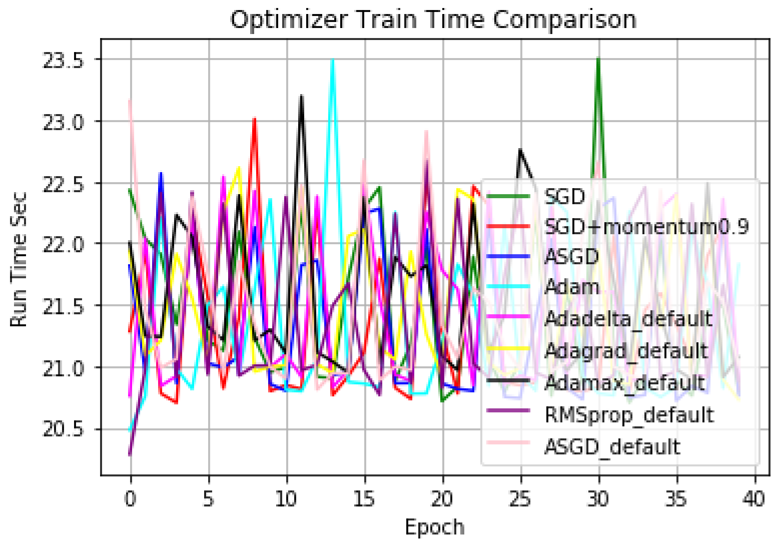


Figure 14. resnet-18 optimizer train time per epoch without Rprop

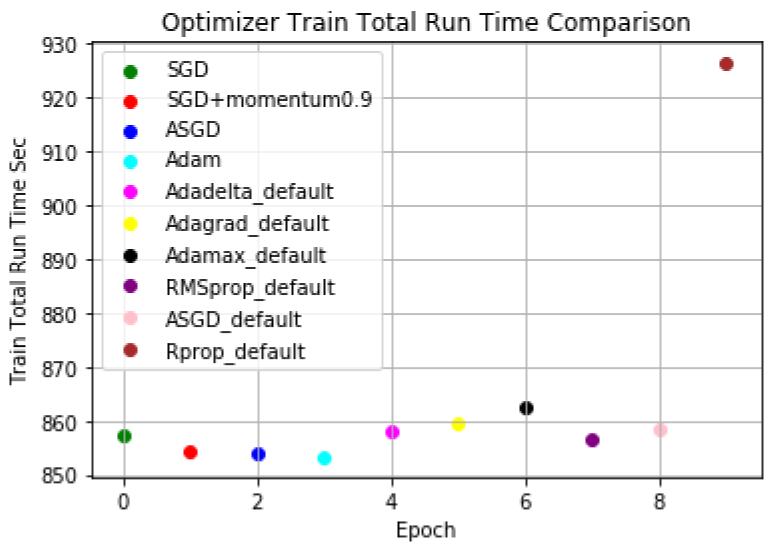


Figure 15. resnet-18 optimizer total train time

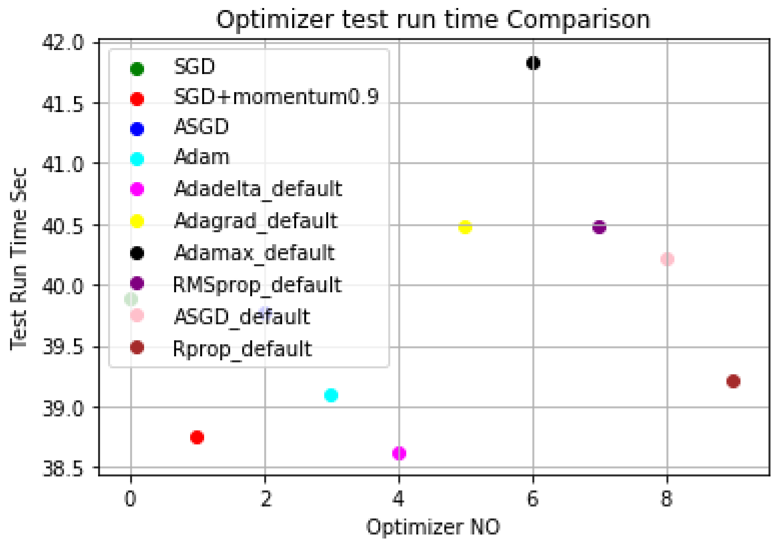


Figure 16. resnet-18 optimizer total test time

Figure 13. contains the Rprop optimizer, which is wrong from the previous analysis and has time cost 23 -24 sec/epoch. Figure 14. excludes the Rprop one and compare the rest 9 towards the resnet-18 the run time cost, and they are messed up from 22-23 sec/epoch. Figure 15. contains 10 optimizers total run time cost during the train time towards the resnet-18. Figure 16. contains 10 optimizers total run time cost during the test time towards the resnet-18.

From Figure 15., we could find out during the train time the Adam(cyan) owns the least total time cost, about 853s; ASGD(blue) is the second, about 854s; SGD+momentum0.9(red) is the third, about 855s. However, they are very similar, they are from 850s - 865s. Only the Rprop one is 925s. From Figure 16., we could find out during the test time SGD+momentum0.9(red), Adadelta_default(purple) and Adam(cyan) are least ones , which are around 39s.
From the run time data, we could not say which one is better than others. On the one hand, during the train time, their total run time is very similar. On the other hand, during the test time, their total run epoch is not a lot and has random cases. But generally speaking, the one owns most accuracy and least loss would be best.
One notable behavior is about Adam(cyan), because it doesn't converge at this time and its time cost are nearly the least one. If we could give it chance to compare with the highest accuracy ones with more epoches, which might end up with a even better result.

3.2.4 resnet-18 results sum up

To sum up, the SGD+momentum0.9(red) are the best choice; the Adamax_default(black) and Adadelta_default(purple) are the Top 3 choices. The Rprop(brown) are not suitable to finish such task. The adam(cyan) and RMSprop_default(purple) are not converged, we need more labs to adjust their parameter setting to fit such task.
Use the math analysis in chapter 2.1, we find the advanced optimizer from each group optimizers are better choice than basic one. I guess this is because the resnet-18 are more common than vgg-16'(I adapted), and all the default parameters are well defined to be fitable. But again the SGD group optimizer wins the adaptive learning rate group optimiers. I think again the reason is that when the input data is large, well labeled and organized, SGD group optimizers would keep stable learning rate and less time cost for it needn't change learning rate and less tricky math calculations, memory cost etc.

Reference

1. https://github.com/shaoanlu/dogs-vs-cats-redux/blob/master/opt_experiment.ipynb<Draw Diagrams in one figure with different optimizers with keras/tensorflow>
2. https://www.programcreek.com/python/example/101174/torch.optim<Good example code for implenmenting different optimizers with pytorch>
3. https://wiseodd.github.io/techblog/2016/06/22/nn-optimization/<optimiers comparison>
4. https://deepnotes.io/sgd-momentum-adaptive<sgd with other optimiers comparison>
5. https://pytorch.org/docs/stable/optim.html <pytorch optimiers>
6. https://www.cs.toronto.edu/~kriz/cifar.html<CIFAR dataset >
7. http://luthuli.cs.uiuc.edu/~daf/courses/Opt-2019/Notes/stochasticgradientdescent.pdf
8. http://luthuli.cs.uiuc.edu/~daf/courses/Opt-2019/Notes/StrikingGD.pdf
9. http://luthuli.cs.uiuc.edu/~daf/courses/Opt-2019/Notes/gradboost.pdf
10. http://ruder.io/optimizing-gradient-descent/index.html#batchgradientdescent
11. https://medium.com/@nishantnikhil/adam-optimizer-notes-ddac4fd7218