MAY 22, 2020

Birkbeck
UNIVERSITY OF LONDON

Machine Learning- COIY065H7

COURSEWORK REPORT

STUDENT NAME: ADAM RUSSELL
MSC DATA SCIENCE
Emails:arusse13@dcs.bbk.ac.uk

Academic Declaration

I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software.

1. Introduction

Weight-wise Adaptive Learning rates with Moving Estimator (WAME) (Magoulas & Mosca, 2017) is a deep learning optimiser that utilises updates rules for improved training speeds and classification performance. This report outlines the implementation and testing of WAME optimisation on a convolutional neural network (CNN) classifier trained on the CIFAR10 dataset. The CIFAR10 dataset contains 60000 32x32 colour images of 10 classes each class with 6000 images. The image classes are ‘airplane’, ‘automobile’, ‘bird’, ‘cat’, ‘deer’, ‘dog’, ‘frog’, ‘horse’, ‘ship’, ‘truck’ (Krizhevsky, et al., n.d.). Section 2 outlines the implementation of WAME to a predeveloped Pytorch model deployed on Google Colab. Section 3 outlines the tests performed on the trained model include tuning of the learning rate and a comparison of accuracy with other common optimizers.

2. Methodology and Design

To streamline implementation the ‘Tutorials > Deep Learning with PyTorch: A 60 Minute Blitz > Training a Classifier’ (Chintala, 2017) was chosen as the test bed for the implementation. It’s a predesigned CNN classifier for CIFAR10 dataset training built on the Pytorch framework and deployed on Google Colab for cloud computation. The structure of the network is shown in Figure 1 structure of the CNN classifier for CIFAR10 (Kernix Lab, 2017).

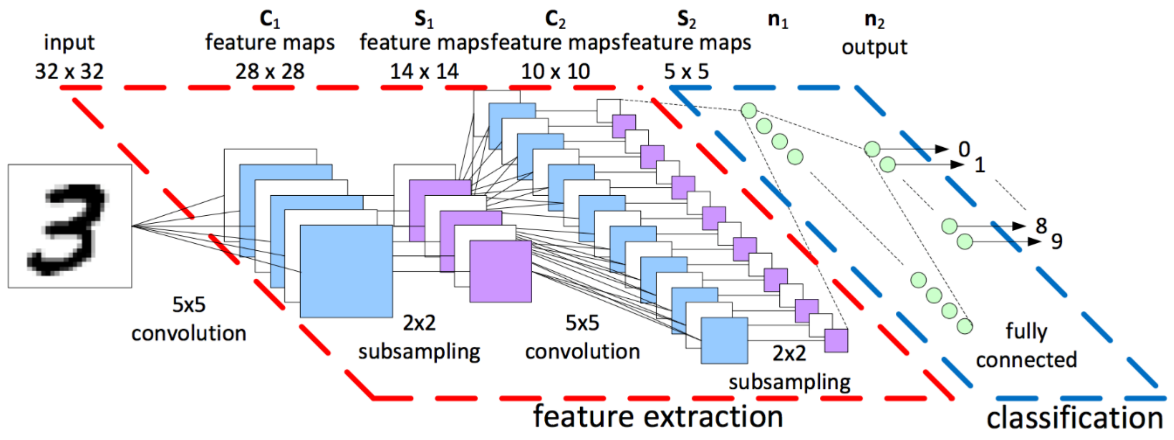

Figure 1 structure of the CNN classifier for CIFAR10

The model utilises torchvision to pull the CIFAR10 data set directly into the notebook where the programming splits the data for training and testing. The model specifies cross entropy as the loss function and Stochastic Gradient Decent (SGD) as the default optimiser. Training is performed in mini batches over 2 epochs with loss updates every 2000 mini-batches. The model features programming to automatically assess accuracy on 10000 test images and accuracy per classes.

Pytorch does not directly feature a WAME optimizer (Magoulas & Mosca, 2017). To implement this the source code for Pytorch RProp (Pytorch, n.d.) and RMSProp (Pytorch, n.d.) was extracted. The RMSProp code which calculates moving average of the squared gradient for each weight (Hinton, et al., n.d.) was extracted. This constitutes line 9 of the WAME algorithm. The RProp code which calculates the step sizes for weight updates was extracted. This constitutes lines 3 to 8 of the WAME algorithm. The step size exponentially–weighted moving average (EWMA) tensor was initialised and coded the same way as the moving average of the squared gradient tensor, only using step size tensor in place of gradient squared. Then the step size tensor is divided by the EWMA tensor to calculate smoothing for each steps. This covers all the variables required for weight updates on line 11 of the WAME tensor. Momentum, decay and centring were not used for this implementation thought the code has been replicated so that RProp and RMSProp can swapped in and out for comparison. The full equation is shown as Equation 1 Weight Update Formula.

$$\Delta w_{ij}(t) = -\lambda \frac{\zeta_{ij}(t)}{Z_{ij}(t)} \frac{\partial E(t)}{\partial w_{ij}} \frac{1}{\sqrt{\theta_{ij}(t)} + \varepsilon}$$

Equation 1 Weight Update Formula

3. Experiments, findings and discussion

Initial runs of the model with default values indicated the model was not learning. This was evident as the True positive test accuracy was 10%, the same as a random choice selection of the 10 classes. It was discovered that the learning rate λ had the biggest impact on this. To assess the optimal learning rate the model was run with 6 different training rates from 1e-2 to 1e-7 for each decimal places in-between. The results are shown in Table 1 Learning Rate λ Tuning and Figure 2 Test Accuracy vs. Model Learning Rate.

At λ = 0.01 model appears to find a local minimum quickly and stagnates. At λ = 1e-7 the model appears to stagnate at the initiation site as the weight updates are insignificantly small. The optimal value appears to be λ = 0.0001 with true positive accuracy on the test data of 46%. Figure 3 Loss Function tracking with mini batches shows that λ = 0.0001 model makes the biggest gains in loss function across mini batch training.

Learning Rate	1.00E-02	1.00E-03	1.00E-04	1.00E-05	1.00E-06	1.00E-07
accuracy	10.00%	42.00%	46.00%	32.00%	17.00%	9.00%
Max. accuracy	100.00%	76.00%	75.00%	58.00%	63.00%	95.00%
Min. accuracy	0.00%	13.00%	14.00%	8.00%	0.00%	0.00%
2000	78.330	1.950	2.073	2.300	2.304	2.305
4000	2.321	1.766	1.892	2.267	2.303	2.304
6000	2.320	1.740	1.785	2.168	2.301	2.303
8000	2.322	1.720	1.724	2.095	2.300	2.303
10000	2.320	1.744	1.677	2.045	2.299	2.303
12000	2.323	1.751	1.647	2.013	2.294	2.304
14000	2.324	1.751	1.605	1.998	2.294	2.304
16000	2.324	1.740	1.575	1.960	2.291	2.303
18000	2.324	1.788	1.531	1.957	2.289	2.303
20000	2.324	1.829	1.561	1.935	2.285	2.302
22000	2.324	1.805	1.541	1.933	2.281	2.302
24000	2.322	1.811	1.505	1.911	2.275	2.303

Table 1 Learning Rate λ Tuning

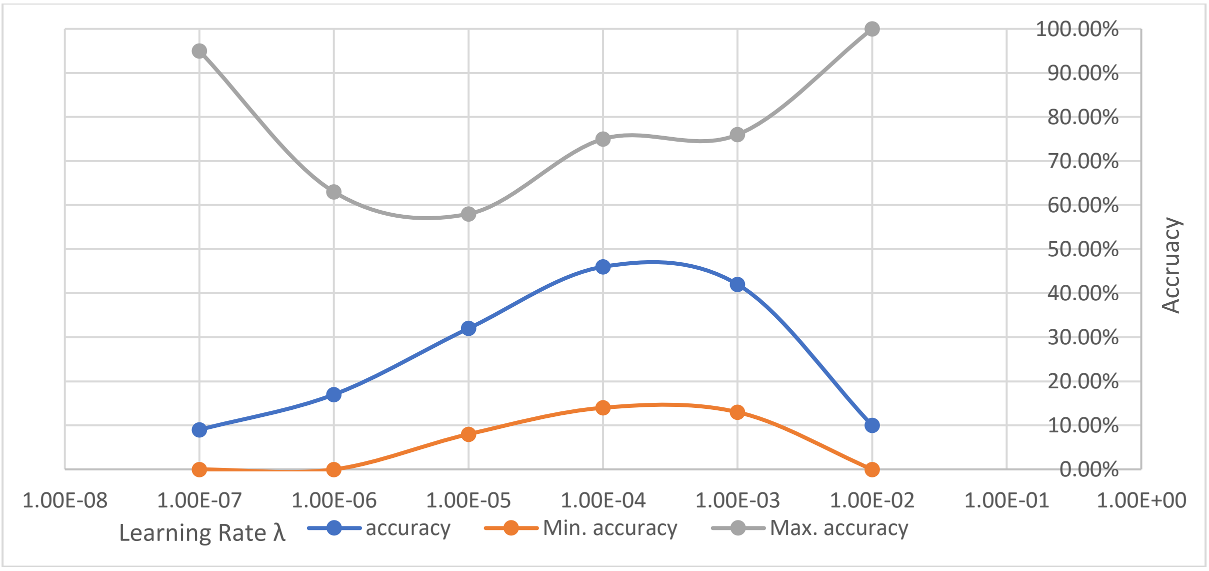

Figure 2 Test Accuracy vs. Model Learning Rate

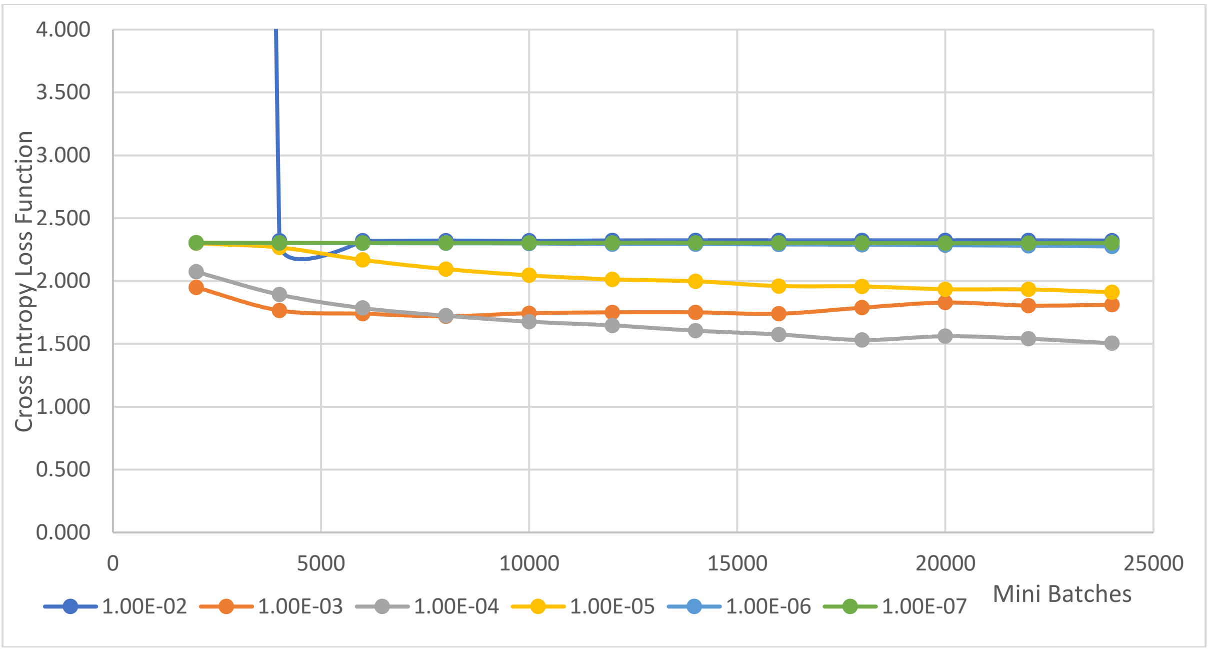

Figure 3 Loss Function tracking with mini batches

RProp, RMSProp and SGD optimizer models were also trained for comparison. Any common hyperparameters with WAME were set to the same value accordingly. The results are shown below Table 2 Accuracy of other optimizers with same hyper parameters. We see that RProp and SGD fails any learning due to the model being setup for mini batches despite having low loss functions. We see that WAME and RMSProp have comparable true positive test accuracy suggesting the moving average of the squared gradient for each weight in the algorithm is the successful common factor.

Optimizer	WAME	RMSProp	RProp	SGD
accuracy	46.00%	46.00%	10.00%	10.00%
Max. accuracy	75.00%	71.00%	100.00%	79.00%
Min. accuracy	14.00%	25.00%	0.00%	0.00%
2000	2.073	2.105	4.732	2.303
4000	1.892	1.893	8.709	2.303
6000	1.785	1.798	4.211	2.302
8000	1.724	1.713	3.030	2.302
10000	1.677	1.671	3.419	2.303
12000	1.647	1.633	3.042	2.301
14000	1.605	1.563	3.051	2.302
16000	1.575	1.545	2.715	2.301
18000	1.531	1.542	2.659	2.301
20000	1.561	1.513	2.773	2.301
22000	1.541	1.477	3.107	2.301
24000	1.505	1.463	2.865	2.301

Table 2 Accuracy of other optimizers with same hyper parameters

4. Conclusions

In conclusion the WAME implementation performed equally well on the CNN CIFAR10 classifier as RMSprop. The optimal learning rate was found to be λ = 0.0001. Further cross validation work should be conducted for further turning of hyper parameters on this model.

5. References

Chintala, S., 2017. [Online]
Available at: https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html#sphx-glr-beginner-blitz-cifar10-tutorial-py
[Accessed May 2020].
Hinton, G., Srivastava, N. & Swersky, K., n.d. *Lecture 6a Overview of mini-batch gradient descent.* [Online]
Available at: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf
[Accessed May 2020].
Kernix Lab, 2017. *A toy convolutional neural network for image classification with Keras.* [Online]
Available at: https://www.kernix.com/article/a-toy-convolutional-neural-network-for-image-classification-with-keras/
[Accessed May 2020].
Krizhevsky, A., Nair, V. & Hinton, G., n.d. *The CIFAR-10 dataset.* [Online]
Available at: https://www.cs.toronto.edu/~kriz/cifar.html
[Accessed May 2020].
Magoulas, G. D. & Mosca, A., 2017. *Training Convolutional Networks with Weight–wise Adaptive Learning Rates.* Bruges (Belgium), ESANN 2017 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, pp. 53-58.
Pytorch, n.d. *rmsprop.py.* [Online]
Available at: https://github.com/pytorch/pytorch/blob/master/torch/optim/rmsprop.py
[Accessed May 2020].
Pytorch, n.d. *rprop.py.* [Online]
Available at: https://github.com/pytorch/pytorch/blob/master/torch/optim/rprop.py
[Accessed May 2020].

Appendix 1- Code

```
import torch
from torch.optim.optimizer import Optimizer


class WAME(Optimizer):
  """
      ETA -> n+, n-, RProp algo, pair of (etaminus, etaplis), that are multiplicative increase and
decrease factors (default: (0.5, 1.2))
      Alpha -> RMSProp smoothing constant (default: 0.99)
```

```
        ZETA -> MIN, MAX, RProp a pair of minimal and maximal allowed step sizes (default:
(1e-6, 50))
        LAMDA -> learning rate lr, RProp learning rate (default: 1e-2)

    """

    def __init__(self, params, lr=1e-2, etas=(0.1, 1.2), step_sizes=(0.01, 100), alpha=0.9,
eps=1e-5, weight_decay=0, momentum=0, centered=False):
        if not 0.0 <= lr:
            raise ValueError("Invalid learning rate: {}".format(lr))
        if not 0.0 < etas[0] < 1.0 < etas[1]:
            raise ValueError("Invalid eta values: {}, {}".format(etas[0], etas[1]))
        if not 0.0 <= eps:
            raise ValueError("Invalid epsilon value: {}".format(eps))
        if not 0.0 <= momentum:
            raise ValueError("Invalid momentum value: {}".format(momentum))
        if not 0.0 <= weight_decay:
            raise ValueError("Invalid weight_decay value: {}".format(weight_decay))
        if not 0.0 <= alpha:
            raise ValueError("Invalid alpha value: {}".format(alpha))

        defaults = dict(lr=lr, etas=etas, step_sizes=step_sizes, momentum=momentum,
alpha=alpha, eps=eps, centered=centered, weight_decay=weight_decay)
        super(WAME, self).__init__(params, defaults)

        def __setstate__(self, state):
            super(WAME, self).__setstate__(state)
            for group in self.param_groups:
              group.setdefault('momentum', 0)
              group.setdefault('centered', False)

    @torch.no_grad()
    def step(self, closure=None):
        """Performs a single optimization step.
        Arguments:
            closure (callable, optional): A closure that reevaluates the model
                and returns the loss.
        """
        loss = None
        if closure is not None:
            with torch.enable_grad():
                loss = closure()

        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:
                    continue
            grad = p.grad
            if grad.is_sparse:
               raise RuntimeError('Rprop does not support sparse gradients')
            state = self.state[p]

            # State initialization
            if len(state) == 0:
               state['step'] = 0
               state['prev'] = torch.zeros_like(p, memory_format=torch.preserve_format)
               state['step_size'] = grad.new().resize_as_(grad).fill_(group['lr'])
               state['square_avg'] = torch.zeros_like(p, memory_format=torch.preserve_format)
               state['EWMA'] = torch.zeros_like(p, memory_format=torch.preserve_format)
               if group['momentum'] > 0:
                  state['momentum_buffer'] = torch.zeros_like(p,
memory_format=torch.preserve_format)
               if group['centered']:
                  state['grad_avg'] = torch.zeros_like(p,
memory_format=torch.preserve_format)

            etaminus, etaplus = group['etas']
            step_size_min, step_size_max = group['step_sizes']
            step_size = state['step_size']

            square_avg = state['square_avg']
            EWMA = state['EWMA']
            alpha = group['alpha']

            state['step'] += 1

            sign = grad.mul(state['prev']).sign()
            sign[sign.gt(0)] = etaplus
            sign[sign.lt(0)] = etaminus
            sign[sign.eq(0)] = 1

            # update stepsizes with step size updates
            step_size.mul_(sign).clamp_(step_size_min, step_size_max)


            # for dir<0, dfdx=0
            # for dir>=0 dfdx=dfdx
            grad = grad.clone(memory_format=torch.preserve_format)
            grad[sign.eq(etaminus)] = 0


            EWMA.mul_(alpha).add_(step_size,alpha=1 - alpha)

            if group['weight_decay'] != 0:
                grad = grad.add(p, alpha=group['weight_decay'])

            square_avg.mul_(alpha).addcmul_(grad, grad, value=1 - alpha)

            if group['centered']:
                grad_avg = state['grad_avg']
                grad_avg.mul_(alpha).add_(grad, alpha=1 - alpha)
                avg = square_avg.addcmul(grad_avg, grad_avg, value=-
1).sqrt_().add_(group['eps'])
            else:
                avg = square_avg.sqrt().add_(group['eps'])

            if group['momentum'] > 0:
                buf = state['momentum_buffer']
                buf.mul_(group['momentum']).addcdiv_(grad, avg)
                p.add_(buf, alpha=-group['lr'])
            else:
                EWMA_avg = torch.div(step_size, EWMA)
                num = torch.mul(EWMA_avg, grad)
                p.addcdiv_(num, avg, value=-group['lr'])

            # update parameters

            state['prev'].copy_(grad)

        return loss
```