Hand Digit Recognition and effect of loss function and optimizer

Naman Ajmera
Machine Learning Intern
AI Technology and systems
Delhi, India
naman00398@gmail.com
https://ai-techsystems.com/
(Author)

***Abstract*—This project report discusses hand digit recognition using a Convolutional Neural Network. The data is provided by MNIST. This report provides an insight on how different loss function and optimizers have affect on the model accuracy to classify digits.**

***Keywords*—Machine Learning, Convolutional Neural Network, loss function, optimizer, classification.**

I. Introduction

MNIST dataset of handwritten digits contain of 60,000 28x28 grayscale images of the 10 digits. I have divided the dataset in 48,000 for training and 12,000 for validation. I had used a simple Convolutional Neural Network to classify images into digits and also I am training on a GPU.

II. Loss Function

It's a method of evaluating how well your algorithm models your dataset. If your predictions are totally off, your loss function will output a higher number. If they're pretty good, it'll output a lower number. As you change pieces of your algorithm to try and improve your model, your loss function will tell you if you're getting anywhere. In fact, we can design our own (very) basic loss function to further explain how it works. For each prediction that we make, our loss function will simply measure the absolute difference between our prediction and the actual value.

A. *Sparse Categorical Cross Entropy*

When doing multi-class classification, categorical cross entropy loss is used a lot. It compares the predicted label and true label and calculates the loss. In Keras with TensorFlow backend support Categorical Cross-entropy, and a variant of it: Sparse Categorical Cross. Sparse Categorical Cross-entropy use the same equation and should have the same output. The only difference between sparse categorical cross entropy and categorical cross entropy is the format of true labels. When we have a single-label, multi-class classification problem, the labels are mutually exclusive for each data, meaning each data entry can only belong to one class.

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

III. Optimizer

They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction..

The Learning Rate

Changing our weights too fast by adding or subtracting too much (i.e. taking steps that are too large) can hinder our ability to minimize the loss function. We don't want to make a jump so large that we skip over the optimal value for a given weight.

To make sure that this doesn't happen, we use a variable called "the learning rate." This thing is just a very small number, usually something like 0.001, that we multiply the gradients by to scale them. This ensures that any changes we make to our weights are pretty small. In math talk, taking steps that are too large can mean that the algorithm will never converge to an optimum.

At the same time, we don't want to take steps that are too small, because then we might never end up with the right values for our weights. In math talk, steps that are too small might lead to our optimizer converging on a local minimum for the loss function, but not the absolute minimum.

A. *Stochastic Gradient Descent*

The standard gradient descent algorithm updates the parameters θ of the objective J(θ)J(θ) as,

$$\theta=\theta-\alpha\nabla\theta E[J(\theta)]$$

where the expectation in the above equation 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,

$$\theta=\theta-\alpha\nabla\theta J(\theta;x(i),y(i))$$

with a pair (x(i),y(i)) from the training set.
Generally each parameter update in SGD is computed w.r.t a few training examples or a minibatch as opposed to a single example. The reason for this is twofold: first this reduces the variance in the parameter update and can lead to more stable convergence, second this allows the computation to take advantage of highly optimized matrix operations that should be used in a well vectorized computation of the cost and gradient. A typical minibatch size is 256, although the optimal size of the minibatch can vary for different applications and architectures.

In SGD the learning rate αα is typically much smaller than a corresponding learning rate in batch gradient descent because there is much more variance in the update.

Choosing the proper learning rate and schedule (i.e. changing the value of the learning rate as learning progresses) can be fairly difficult. One standard method that works well in practice is to use a small enough constant learning rate that gives stable convergence in the initial epoch (full pass through the training set) or two of training and then halve the value of the learning rate as convergence slows down.

B. *Adam*

Adam stands for adaptive moment estimation, and is another way of using past gradients to calculate current gradients. Adam also utilizes the concept of momentum by adding fractions of previous gradients to the current one. This optimizer has become pretty widespread, and is practically accepted for use in training neural nets.
It's easy to get lost in the complexity of some of these new optimizers. Just remember that they all have the same goal: minimizing our loss function. Even the most complex ways of doing that are simple at their core.
Adam realizes the benefits of both AdaGrad and RMSProp.

Instead of adapting the parameter learning rates based on the average first moment (the mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance). Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

C. *RMSprop*

The 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. The following equations show how the gradients are calculated for the RMSprop and gradient descent with momentum. The value of momentum is denoted by beta and is usually set to 0.9.

$$v_{dw} = \beta \cdot v_{dw} + (1-\beta) \cdot dw^2$$

$$v_{db} = \beta \cdot v_{dw} + (1-\beta) \cdot db^2$$

$$W = W - \alpha \cdot \frac{dw}{\sqrt{v_{dw}} + \epsilon}$$

$$b = b - \alpha \cdot \frac{db}{\sqrt{v_{db}} + \epsilon}$$

D. *Adagrad*

Adagrad adapts the learning rate specifically to individual features: that means that some of the weights in your dataset will have different learning rates than others. This works really well for sparse datasets where a lot of input examples are missing. Adagrad has a major issue though: the adaptive learning rate tends to get really small over time. Some other optimizers below seek to eliminate this problem.

IV. Approach

A. *Data Acquisition*

The Dataset used in this project was taken from MNIST us using keras mnist.load_data() consisting of images of 10digits.

B. *Reading and Displaying the data*

Keras has been used to read dataset. Matplotlib has been used to display loss and accuracy.

C. *Convolutional Neural Network*

CNN has been used to classify image, using different optimizer and Sparse Categorical Cross Entropy as loss function.
Keras library has been used to create the model. The model consist of 4 convolutional layers (the first one is the input layer), there are 4 max pooling layers, 2 fully connected layers and a output layer.

S.No	**Compile**		
	Optimizer	***Time***	***Accuracy***
1.	Adam	38.09s	0.983
2.	SGD	33.56s	0.983
3.	RMSprop	39.48s	0.978
4.	Adagrad	36.73	0.942

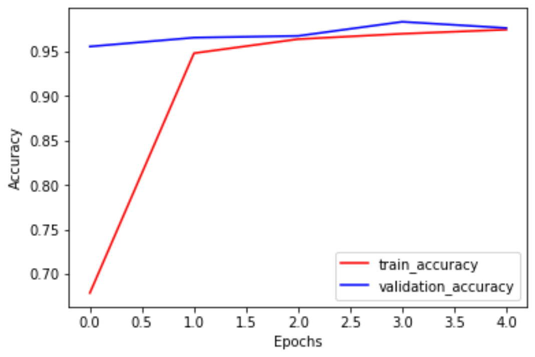

Fig. 1. SGD Optimizer. (*Accuracy vs Epochs)*

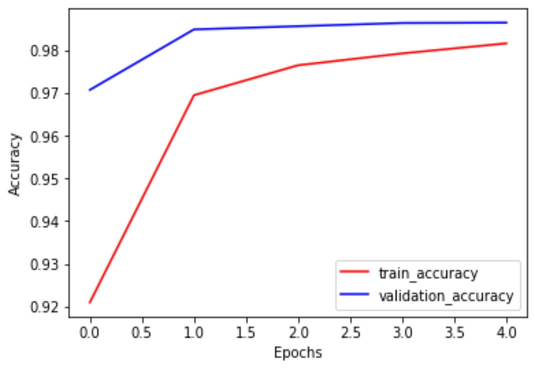

Fig. 2. Adam Optimizer. (*Accuracy vs Epochs*)

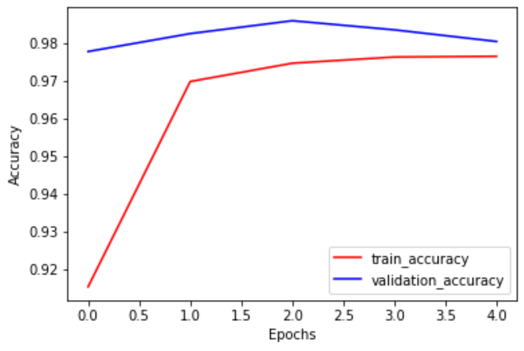

Fig. 3. RMPSprop Optimizer. (Accuracy vs Epochs)

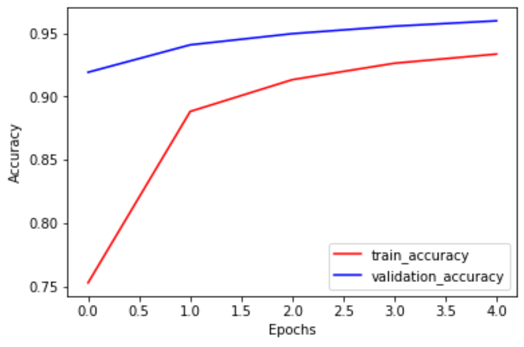

Fig. 4. Adagrad Optimizer. (Accuracy vs Epochs)