Hand Digit Recognition (FNN)

*with dataset at https://keras.io/examples/mnist_dataset_api/

Visheshwar Pratap Singh
AITS Internship 2019 Batch 5
AI Tech & Systems
visheshwarpratap.chat@gmail.com

www.ai-techsystems.com

***Abstract*—Handwritten character recognition is one of the important parts of pattern recognition. The applications of digit recognition include postal mail sorting, bank check processing, form data entry. This report presents a simple three-layer feed forward neural network model trained on MNIST dataset. Different optimizers and loss functions give different accuracy values.**

***Keywords*—machine learning, Keras, sequential model, neural network, classification algorithm.**

I. Introduction

There are many ways and models which can be used for handwritten digit recognition. In project I used sequential model with three neural network layers. For training the model, MNIST dataset is used which is public dataset in Keras. As digit recognition is basically a classification problem therefore both fully connected network and CNN can be used. The accuracy of the model depends upon the images of the digits, optimizers and loss functions used while compiling the model.

II. Libraries

A. Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.

B. TensorFlow

TensorFlow is a free and open-source software library for dataflow and differentiable programming across a range of tasks. Its flexible architecture allows for the easy development computation across a variety of platforms and from desktops to clusters of servers to mobile and edge devices.

III. Prepare Dataset

The dataset used in this project is the public MNIST dataset available with keras. This dataset contains images of handwritten digits. Each image is of size 28*28 pixels.

A. Loading and Splitting Dataset

MNIST dataset is loaded using load_data function and split into training and testing data. For sparse categorical crossentropy the classes i.e the Y labels are used directly. But for the categorical crossentropy and the kullback leibler divergence loss function, we need to one-hot encode the classes.

B. Normalizing

- Normalize function from keras utilities is used to scale the pixel values of each pixel of the image.
- Normalizing the pixel value means to scale the value of a pixel between 0 and 1.
- Normalizing function highly effects the loss and accuracy of the model while training and evaluation.
- Without normalizing the images, the accuracy was reduced to almost half and the loss was increased.

IV. Multi-Class Classification Loss Function

Multi-class classification are those predictive modeling problems where examples are assigned one of the more than two classes. The problem is often framed as predicting an integer value, where each class is assigned a unique integer value from 0 to one less than total number of classes. The problem is often implemented as predicting the probability of the example belonging to each known class.

A. Categorical cross-entropy

Cross-entropy is the default loss function to use for multi-class classification problems.

In this case, it is intended for use with multi-class classification where the target values are in the set {0, 1, 3, …, n}, where each class is assigned a unique integer value.

Mathematically, it is the preferred loss function under the inference framework of maximum likelihood. It is the loss function to be evaluated first and only changed if you have a good reason.

Cross-entropy will calculate a score that summarizes the average difference between the actual and predicted probability distributions for all classes in the problem. The score is minimized and a perfect cross-entropy value is 0.

Cross-entropy can be specified as the loss function in Keras by specifying '*categorical_crossentropy*' when compiling the model.

B. *Sparse Multiclass Cross-Entropy Loss*

A possible cause of frustration when using cross-entropy with classification problems with a large number of labels is the one hot encoding process.

For example, predicting words in a vocabulary may have tens or hundreds of thousands of categories, one for each label. This can mean that the target element of each training example may require a one hot encoded vector with tens or hundreds of thousands of zero values, requiring significant memory.

Sparse cross-entropy addresses this by performing the same cross-entropy calculation of error, without requiring that the target variable be one hot encoded prior to training.

Sparse cross-entropy can be used in keras for multi-class classification by using '*sparse_categorical_crossentropy*' when calling the *compile()* function.

C. *Kullback Leibler Divergence Loss*

Kullback Leibler Divergence, or KL Divergence for short, is a measure of how one probability distribution differs from a baseline distribution.

A KL divergence loss of 0 suggests the distributions are identical. In practice, the behavior of KL Divergence is very similar to cross-entropy. It calculates how much information is lost (in terms of bits) if the predicted probability distribution is used to approximate the desired target probability distribution.

As such, the KL divergence loss function is more commonly used when using models that learn to approximate a more complex function than simply multi-class classification, such as in the case of an autoencoder used for learning a dense feature representation under a model that must reconstruct the original input. In this case, KL divergence loss would be preferred. Nevertheless, it can be used for multi-class classification, in which case it is functionally equivalent to multi-class cross-entropy.

KL divergence loss can be used in Keras by specifying 'kullback_leibler_divergence' in the compile () function.

V. OPTIMIZERS

Optimization algorithms helps us to ***minimize (or maximize)*** an **Objective** function (*another name for* ***Error*** *function*) **E(x)** which is simply a mathematical function dependent on the Model's internal learnable parameters which are used in computing the target values(**Y**) from the set of *predictors*(**X**) used in the model.

A. *Stochastic gradient descent*

Stochastic Gradient Descent (SGD) on the other hand performs a parameter update for each training example. It is usually much faster technique. It performs one update at a time.

$\theta=\theta-\eta\cdot\nabla J(\theta;x(i);y(i))$, where {x(i) ,y(i)} are the training examples.

Now due to these frequent updates ,parameters updates have high variance and causes the Loss function to fluctuate to different intensities. This is actually a good thing because it helps us discover new and possibly better local minima.

B. *Adagrad*

It simply allows the learning Rate -η to adapt based on the parameters. So it makes big updates for infrequent parameters and small updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data.

It uses a different learning Rate for every parameter θ at a time step based on the past gradients which were computed for that parameter.

Previously, we performed an update for all parameters **θ** at once as every parameter **θ(i)** used the same learning rate **η**. As Adagrad uses a different learning rate for every parameter **θ(i)** at every time step **t**, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set **g(t,i)** to be the gradient of the loss function w.r.t. to the parameter**θ(i)** at time step **t .**

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.$$

Fig. 1. The formula for Parameter updates

Adagrad modifies the general learning rate η at each time step t for every parameter θ(i) based on the past gradients that have been computed for θ(i).

C. *AdaDelta*

It is an extension of AdaGrad which tends to remove the *decaying learning Rate* problem of it. Instead of accumulating all previous squared gradients, *Adadelta* limits the window of accumulated past gradients to some fixed size w.

Instead of inefficiently storing **w** previous squared gradients, the sum of gradients is recursively defined as a *decaying* **mean** of all past squared gradients. The running average **E[g²](t)** at time step **t** then depends (as a fraction **γ** similarly to the Momentum term) *only on the previous average and the current gradient.*

$E[g^2](t)=\gamma.E[g^2](t-1)+(1-\gamma).g^2(t)$, We set **γ** to a similar value as the momentum term, around 0.9.

$\Delta\theta(t)=-\eta\cdot g(t,i)$.

$\theta(t+1)=\theta(t)+\Delta\theta(t)$.

$$\Delta\theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t.$$

As the denominator is just the root mean squared (RMS) error criterion of the gradient, we can replace it with the criterion short-hand:

$$\Delta\theta_t = -\frac{\eta}{RMS[g]_t} g_t.$$

Fig. 2. The final formula for Parameter Updates

Another thing with AdaDelta is that we don’t even need to set a default learning Rate.

D. *Adam*

Stochastic Adam stands for Adaptive Moment Estimation. Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta, *Adam also keeps an exponentially decaying average of past gradients M(t), similar to momentum*:

M(t) and V(t) are values of the first moment which is the ***Mean*** and the second moment which is the ***uncentered variance*** of the *gradients* respectively.

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}.$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}.$$

Fig. 3. The formulas for the first moment(mean) and the second moment (the variance) of the Gradients

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

Fig. 4. The final formula for Parameter Updates

The values for β1 is 0.9, 0.999 for β2, and (10 x exp(-8)) for ϵ.

Adam works well in practice and compares favorably to other adaptive learning-method algorithms as it converges very fast and the learning speed of the Model is quiet fast and efficient and also it rectifies every problem that is faced in other optimization techniques such as vanishing Learning rate, slow convergence or High variance in the parameter updates which leads to fluctuating Loss function.

VI. Evaluation

TABLE I. Accuracy

Optimizer	**Loss Function**		
	Categorial_crossentropy	***Kullback_leibler_divergence***	***Sparse_categorical_crossentropy***
AdaDelta	96.27	96.26	96.51
Adagrad	95.18	94.75	94.91
Adam	96.9	96.9	96.92
Adamax	95.55	95.77	95.86
Nadam	97.54	96.79	97.36
RMSprop	98.36	98.26	98.86
SGD	90.64	90.58	90.38

Values are given in percentage (%)

TABLE II. Loss

Optimizer	**Loss Function**		
	Categorial_crossentropy	***Kullback_leibler_divergence***	***Sparse_categorical_crossentropy***
AdaDelta	0.1297	0.1252	0.1156
Adagrad	0.1719	0.1826	0.1788
Adam	0.0985	0.1031	0.1025
Adamax	0.1477	0.1410	0.1407
Nadam	0.0864	0.1032	0.0884
RMSprop	0.0961	0.0421	0.0961
SGD	0.3519	0.3485	0.3552

.

References

[1] https://keras.io/optimizers/

[2] https://keras.io/losses/

[3] https://medium.com/tebs-lab/how-to-classify-mnist-digits-with-different-neural-network-architectures-39c75a0f03e3

[4] https://machinelearningmastery.com/tutorial-first-neural-network-python-keras/

[3] https://towardsdatascience.com/counting-no-of-parameters-in-deep-learning-models-by-hand-8f1716241889