A CLASSIFICATION NERUAL NETWORK WITHOUT USING DEEP LEARNING PACKAGES

Weilun Wang and Sun'ao Liu

Jan. 7, 2019

University of Science and Technology of China

Department of Electronic Engineering and Information Science

Technical Report No. EEIS-2019-1

USTC

A CLASSIFICATION NERUAL NETWORK WITHOUT USING DEEP LEARNING PACKAGES

Weilun Wang and Sun'ao Liu

University of Science and Technology of China
Department of Electronic Engineering and Information Science
No.96, JinZhai Road Baohe District
Hefei,Anhui, 230026,P.R.China
`www.ustc.edu.cn`

Jan. 7, 2019

Technical Report No. EEIS-2019-1

Summary

In this project, we implement a neural network to predict the class labels of a given image without using any deep learning packages. We use technology including adam optimizer, batch normalization, Xavier initialization, etc, to build the network. We also designed three structures for comparison and finally selected the optimal structure. In the end, we got an accuracy rate of 0.750 **top-3** on the validation set and 0.662 **top-3** on the testing set.

Contents

1 Introduction

In this project, we implement a neural network to predict the class labels of a given image without using any deep learning packages.

The dataset we use in this task contains 10,000 images in total, which are divided into 20 classes (each class has 500 images).

The classifier's input is a image while its output is the confidence scores of all classes. If the input image is belong to one of the 20 classes, the confidence score corresponding to its true class is expected to be the largest one. Otherwise, the confidence score of a label "unknown" is supposed to be the largest one. Thus, we indeed have 21 class labels.

For each testing image, the classifier will output the top three class labels with the highest confidence scores. If the true class label is one of the top three labels, we will say the classifier has correctly predicted the class label of the input image; otherwise, the classifier made a mistake.

2 Dataset

This dataset contains 10,000 images in total, which are divided into 20 classes (each class has 500 images). The class labels are as follows.

Table 1: The class labels

label index	1	2	3	4	5
text description	goldfish	frog	koala	jellyfish	penguin
label index	6	7	8	9	10
text description	dog	yak	house	bucket	instrument
label index	11	12	13	14	15
text description	nail	fence	cauliflower	bell peper	mushroom
label index	16	17	18	19	20
text description	orange	lemon	banana	coffee	beach

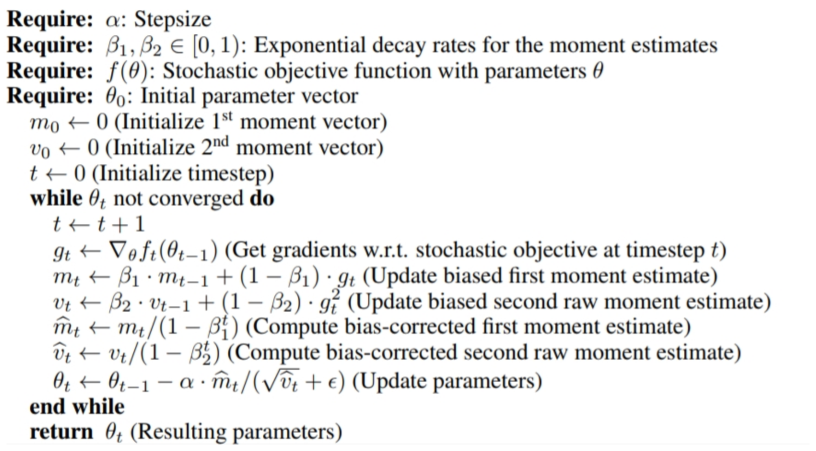
The dataset is cultured from the tiny ImageNet dataset. All training images are color images, and each of them has 64 * 64 pixels. The size of the testing images is the same as the training images.

In the actual training, we searched the Internet for 1000 64*64 RGB images as training materials for the unknown class.

3 Implement details

3.1 Optimizer:Adam

We use Adam algorithm [3] for our optimization.Adam is an adaptive learning rate optimization algorithm and is presented as follow.The name "Adam" derives from the phrase "adaptive moments".It can be seen as a variant on the combination of RMSProp and momentum with a few distinctions.Because of its efficiency, Adam has become one of the most popular optimization algorithm since proposed in 2015.



```
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 1: Adam algorithm

In our program, we use the default settings ($\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 10^{-8}$) with learning rate decay 0.95.

To improve the efficiency, we changed the order of computation by replaceing the last three lines in the loop with following lines:$\alpha_t = \alpha \cdot \sqrt{1 - \beta_2^t}/(1 - \beta_1^t)$ and $\theta_t \leftarrow \theta_{t-1} - \alpha_t \cdot m_t/(\sqrt{v_t} + \hat{\epsilon})$

3.2 Initialization:Xavier

It is important to initialize the deep neural network properly. In our program, we use the Xavier Initialization:

$$W \sim U[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}],$$

where n_j and n_{j+1} denote the input and output dimension for the j-th layer, respectively. Xavier initialization [1], or the normalized initialization, aims to approximately maintain activation variances and back-propagated gradients variances as one moves up or down the network. It has been proved to perform well in practice.

3.3 Accelerate convolution:im2col

The implement of naive convolution is very inefficient. Now most networks choose to use GPUs to accelerate this process.Unfortunately we don't have enough time to write our own CUDA codes, but we can use the method to change convolution into matrix product.A simple example is shown as follow.

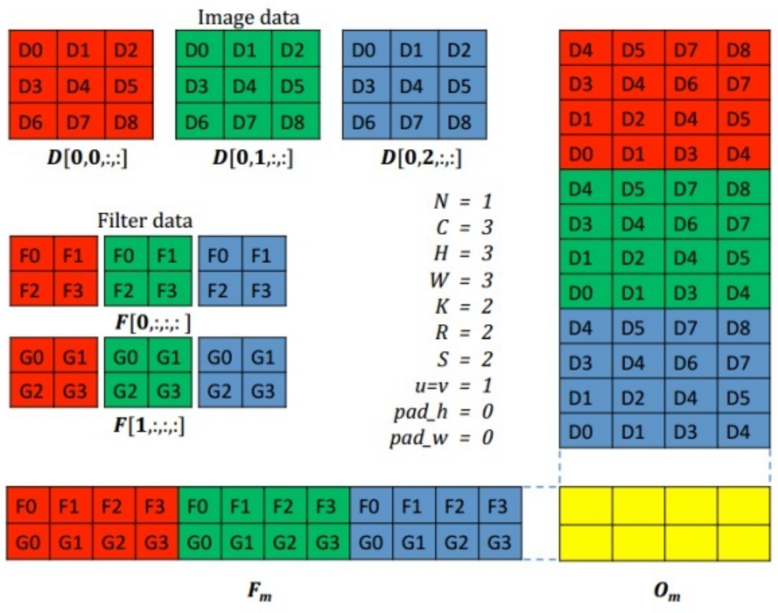


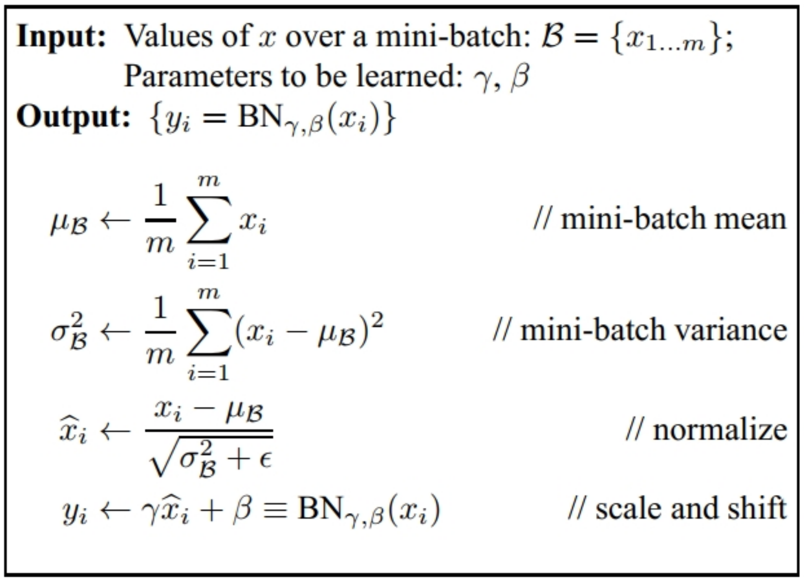
Figure 2: Convolution using matrix

We use some fancy index operation to implement this method.Though still pretty slow compared to GPU-based methods, it performs much better than the naive convolution.

3.4 Batch Normalization

Expect the basic convolution layer ,pooling layer and fully-connected layer, we also used Batch Normalization [2] to improve the perfomance of our network. Batch Normalization allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout.

The BN transform is shown as follows:



Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 3: Batch Normalizing transform

The gradients with respect to the parameters of the BN transform are:

$$\frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i} \cdot \gamma$$

$$\frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} = \sum_{i=1}^{m} \frac{\partial l}{\partial \hat{x}_i} \cdot (x_i - \mu_{\mathcal{B}}) \cdot \frac{-1}{2}(\sigma_{\mathcal{B}}^2 + \epsilon)^{-3/2}$$

$$\frac{\partial l}{\partial \mu_{\mathcal{B}}} = (\sum_{i=1}^{m} \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}) + \frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{\sum_{i=1}^{m} -2(x_i - \mu_{\mathcal{B}})}{m}$$

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \frac{\partial l}{\partial \sigma_{\mathcal{B}}^2} \cdot \frac{2(x_i - \mu_{\mathcal{B}})}{m} + \frac{\partial l}{\partial \mu_{\mathcal{B}}} \cdot \frac{1}{m}$$

$$\frac{\partial l}{\partial \gamma} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i} \cdot \hat{x}_i$$

$$\frac{\partial l}{\partial \beta} = \sum_{i=1}^{m} \frac{\partial l}{\partial y_i}$$

During the training process, we calculated the running average of mean and variance with $momentum = 0.9$, which are used for the testing process.

4 Model structure

We choose three different structures and test their performance on this problem separately. These three structures are as follows.

Table 2: The class labels

model	4-layers model	6-layers model	VGG-like model

structure	conv_relu(32,3,3)
	max_pooling(2,2)
	fc(100)
	fc(21)
		conv_relu_bn(32,5,5)
		max_pooling(2,2)
		conv_relu(64,3,3)
		conv_relu_bn(64,3,3)
		max_pooling(2,2)
		fc(128)
		fc(21)
			conv_relu(32,3,3)
			max_pooling(2,2)
			conv_relu_bn(64,3,3)
			max_pooling(2,2)
			conv_relu(128,3,3)
			conv_relu_bn(128,3,3)
			max_pooling(2,2)
			conv_relu(256,3,3)
			conv_relu_bn(256,3,3)
			max_pooling(2,2)
			fc(256)
			fc(256)
			fc(21)

We have chosen some small models because small models are more likely to converge than large models in the case of limited computing resources, which may result in better performance.

Based on the performance of the validation(which shows below), we chose VGG-like model as our final model. Its forward propagation process is as follows.

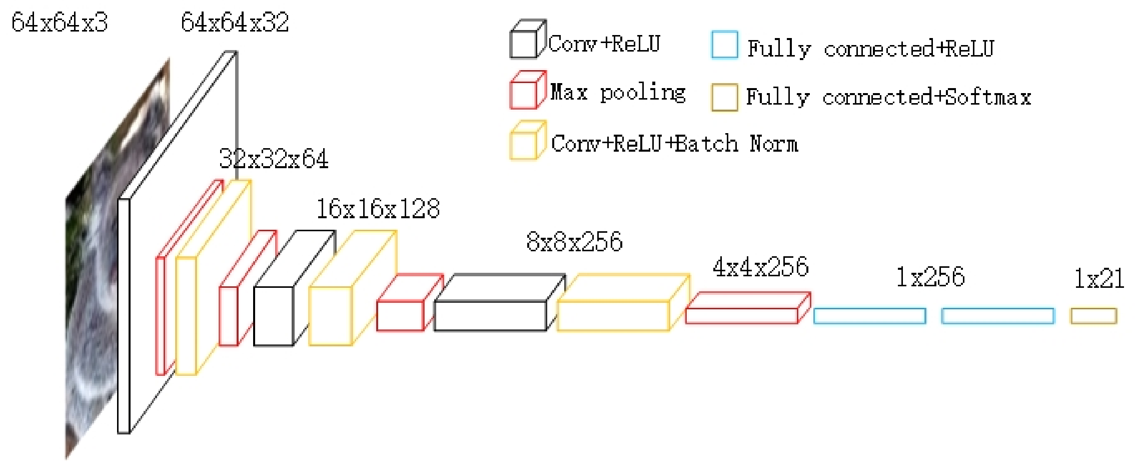


Figure 4: forward propagation process

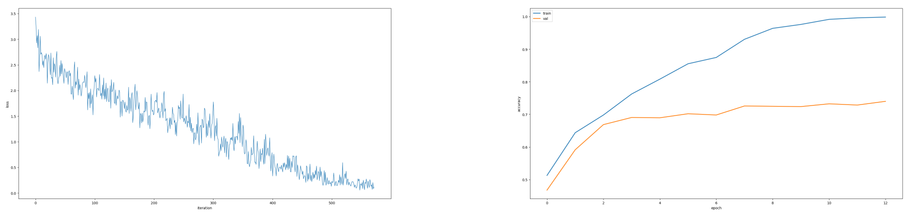
5 Training

We trained 15 epochs on the training set with three different models and tested the accuracy of the model on the validation set after each epoch. We selected the best performance of 15 epochs as the best performance under one structure.

Table 3: The performace of models

model	4-layers model	6-layers model	VGG-like model
performace	0.674	0.702	0.750

Based on the performance on the validation set, we choose VGG-like model as our final model. The Loss function and accuracy of the model during training are as follows.



(a) Loss function (b) Accuracy

Figure 5: Loss function and Accuracy

6 Testing

Finally, we tested our model on a given test set and got an accuracy of 0.662.

7 Roles and responsibilities of the team

In this task, Sun'ao Liu is mainly responsible for coding the adam optimizer and the batch normalization while Weilun Wang is mainly responsible for coding the convolution layer, the pooling layer and processing the training set and validation set. After the code is written, we all spend time on the code's debuging and training. Therefore, We believe that our division of labor should be 50%, 50%.

References

[1] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 249–256, 2010.

[2] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[3] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.