Imagenette Classification : A Brief Report

Haowei Lin*
Yuanpei College
Peking University
linhaowei@pku.edu.cn

Abstract

This report includes details about my first programming homework for *Introduction to Artificial Intelligence Practice Course: Trustworthy Machine Learning*. Based on *Wide ResNet-50-2*, I designed a model to achieve more than **92%** accuracy on **Imagenette classification** task. My code is public on `https://github.com/linhaowei1/Introduction-to-AI/tree/main/Imagenette`.

1 Experimental setup

1.1 Platform

The framework of my code was implemented basically on Python and Pytorch. For more details about packages' versions, see table 1.

Table 1: Package Version

Package	Version
torch	1.2.0
torchvision	0.4.0
Python	3.6.8

1.2 Dataset

Imagenette is a subset of 10 easily classified classes from Imagenet (tench, English springer, cassette player, chain saw, church, French horn, garbage truck, gas pump, golf ball, parachute). I downloaded the '320px' version [2] as my dataset, which contains 9469 images for training, 3925 images for testing.

In order to adjust the parameters and hyperparameters, I split the whole training set into train set and validation set(8:2). With 1893 images for validation, I tuned my hyperparameters and then used the whole training set to train my model, and finally tested on the test data. Details about hyperparameters will be talked about in section 1.4.

1.3 Model

Wide ResNet-50-2 is applied to this task without any revision(except the dimension for the final logits, from original 1000 to 10). The model is the same as ResNet except for the bottleneck number of channels which is twice larger in every block. The number of channels in outer 1x1 convolutions is the same, e.g. last block in ResNet-50 has 2048-512-2048 channels, and in Wide ResNet-50-2 has

*`https://github.com/linhaowei1`

[2] `https://github.com/fastai/imagenette`

Preprint. Under review.

2048-1024-2048. The model has 68.88M parameters. More details about the architecture can be found in [1] and [2].

The code was referred to the official version from torchvision[3].

1.4 Hyperparameters

Owing to the powerful learning ability of Wide ResNet-50-2, I didn't need to overdo the parameter tuning. With *tensorboardX*, I tried several parameter setting and tune the hyperparameters for faster convergence and better generalization on validation set. The final hyperparameters I used are as follows. See table 2.[4]

Table 2: Hyperparameters

optimizer	Adam
learning rate	1e-4
batch size	32
epoch	500
random seed	1111(random)

1.5 Training and Evaluation Protocol

I tuned the hyperparameters on training set and validation set, which are devided from the whole training set and the size proportion is $8:2$. Then I trained my model on the whole training set for 500 epochs and finally tested it using test set (never seen during training time).

For the sake of simplicity, I use the whole training set and test set for further analysis in section 4. Due to the long time training and tuning a Resnet, I directly plot the training process using test set. I know it's sort of cheating in machine learning, but I just adopt it for ablation study. The model I submit was strictly trained without seeing the test set until the final test.

2 Preprocess

2.1 Normalization

For preparing training data, I first calculate the mean and stdev of the raw images. Normalization is usually essential in image classification tasks, which helps backpropagation work well in training process. With `stdMeanCalc.py`, I get the stdev and mean vectors for Imagenette training samples. See table 3.

Table 3: mean and stdev

	channel1	channel2	channel3
mean	0.463	0.458	0.430
stdev	0.241	0.235	0.243

2.2 Resize

Apart from normalization, I need to crop and resize the images to 224×224 since ResNet was designed for inputs of shape like this. I follow the official tutorial[5] from Pytorch Team to build the preprocess pipeline: First resize one edge to 256, and then crop to 224×224.

[3] `https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py`

[4] Adam is of default setting from `https://pytorch.org/docs/stable/optim.html?highlight=adam#torch.optim.Adam`

[5] `https://pytorch.org/hub/pytorch_vision_resnet/`

2.3 Augmentation

To boost the performance in such a small dataset, proper augmentation is of great significance for preventing overfiting. According to my experience, I apply *random crop* and *random horizontal flip* to training data, since they are effective and safe augmentations I used to exploit in former application. The augmentation also worked well on validation set.

In addition, the above operations would spend a lot of time preprocessing data. I used 8 cpu to preprocess data in parallel (set numbers of worker in dataloader) , which accelerated the training process remarkably.

3 Results

3.1 Final Accuracy

Final accuracy is 92.94%.

3.2 Plot of Loss and Accuracy

Fig. 1 and Fig. 2 present the training loss and test accuracy. The test accuracy was recorded during the training time. That is to say, the model was tested on test set at the end of every epoch. And the training loss denotes the average loss in one epoch.

As we can see, 2 figures both indicate that the training process is fine. The descending training loss and the increasing test accuracy tell the good generalization the model performs. We do not see overfitting appears, however, there exists some potential for the model to gain higher accuracy, as the acc curve still has slight upward trend.

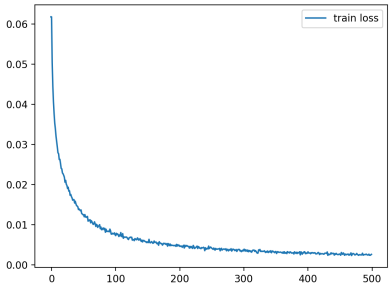

Figure 1: Training Loss

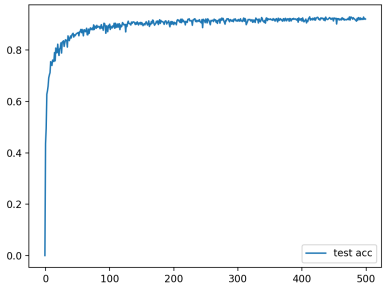

Figure 2: Test Accuracy

3.3 Computation Cost

table 4 shows my hardware configuration. It took 1min14s to train a single epoch, thus the whole training process spent about 10h. During training time, the average power is 250W and the memory usage is 6799M. Therefore, it is estimated that the whole training process consumed energy about $2.5kw \cdot h$.

Table 4: hardware configuration

NVIDIA Graphic Card	Cuda Version	Graphic Card Driver Version
GeForce RTX 2080	10.0	410.93

4 Optional Analysis: Ablation Study

In this section, I will talk about three factors that affect model performance: learning rate, model structure and augmentation.

4.1 Learning Rate

Fig. 3 and Fig. 4 show that learning rate is an important factor in training. It is evident that if learning rate is too small, like $lr = 1e-5$ in the picture, the loss will descend rather slow, and easy to stuck in local optimal. While if learning rate is too large, like $lr = 1e-1$ in the picture, the gradient descend may be weird and hard to converge, for its too large pace.

In the figures, $lr = 1e-4$ has the most rapid convergence on lower loss and the highest accuracy, followed by $lr = 1e-3$ and $lr = 1e-2$. Usually we consider that the gradient will descend faster with higher learning rate, but here comes a contradiction. I think it maybe because of the complex shape of the projection function, and $lr = 1e-4$ can lead the gradient towards the more accurate direction.

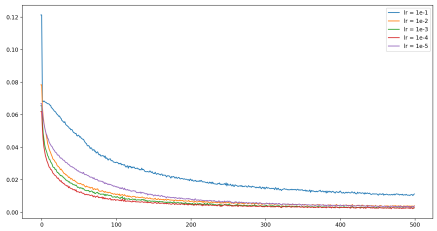

Figure 3: training loss

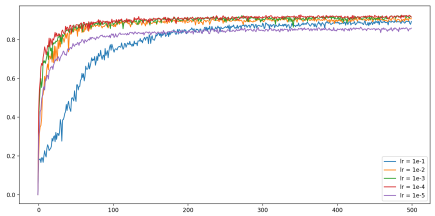

Figure 4: Test Accuracy

4.2 Model Structure

Table 5 shows performance of different models based on Resnet. I finetune them using pretrained parameters provided by torchvision. Although all finetuned models reach extremely high accuracy, we could still draw some conclusion about the model capacity. For example, there exists positive correlation between model depth and accuracy. Resnet50 is much deeper than resnet34, but considering FLOPs (floating point operations), resnet50 is not such more complex than resnet34 thanks to the design of bottleneck block. Apart from depth, the width and structure also impact a lot.

Table 5: different pretrain models

model	accuracy
resnet18	97.962
resnet34	98.803
resnet50	99.261
resnet101	98.777
resnet152	99.439
wide resnet50-2	99.389
wide resnet101-2	99.567
resnext50-32x8d	99.669
resnext101-32x4d	99.414

4.3 Augmentation

Augmentation is very practical in image recognition tasks. Fig. 5 and Fig. 6 show that augmentation plays an important role in better generalization. Without augmentation, the training loss decreased rapidly, which means the model fit the training data pretty well. But test accuracy is much lower than augmentation setting.

The reason why augmentation can boost performance in this task, I think, has several aspects. Remember the augmentation I use is random crop and random horizontal flip, so firstly it prevents the bias on directions and positions of the objects. For such a small dataset, augmentation helps to enhance the diversity of the training samples. Moreover, I noticed that in many of the classes, e.g. tench, English springer, French horn, there are humans in the images. Noise like this could mislead

the model to make classification according to noise features. Fortunately, random crop could weaken the noise impact to some extent.

I think the augmentation hasn't been studied exhustively in my experiments. To achieve required 92% acc, simple augmentation is just enough. Maybe adding human figures to every class could help resolve the problem mentioned above, and this could be set for further exploration.

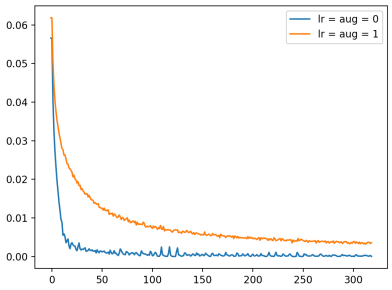

Figure 5: Training Loss

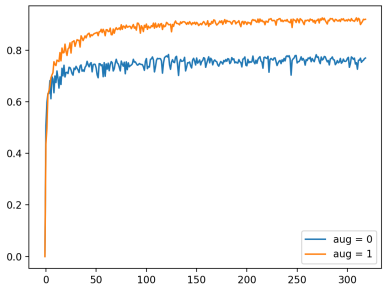

Figure 6: Test Accuracy

References

[1] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *IEEE*, 2016.

[2] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *CoRR*, abs/1605.07146, 2016.