HPC and Big Data
Assignment - Machine Learning

Florian Wolf - 12393339(UvA)
flocwolf@gmail.com

February 9, 2021

1 Introduction

The last assignment of the course High Performance Computing and big Data (HPC) focuses on Machine Learning (ML). ML is a part of artificial intelligence (AI) and builds applications which learn to represent and predict data [1]. While there are again various subsections of ML, the challenge of this assignment is a supervised learning classification problem. The applied methods include deep learning, parallel and cluster computing techniques. More specific, the task of this assignment is to profile the performance of training a neural network. Therefore we build a model which learns to successfully classify images from the CIFAR10 dataset [2]. For image classification, convolutional neural nets (CNN) yield great results [3]. Such models have a large number of parameters to train, empirically starting at 10^7, thus require high computational resources. We train the image classifier on the LISA partition of the cluster computer facility SURF [4]. Further we compare the computation time when training a model in parallel fashion on one, two or three nodes.

2 Implementation

To profile the performance of training a neural network on multiple processors/workers, we first need to implement a classifier and a working deep learning pipeline. Deep learning can be accelerated by parallelizing the training by dividing the workload on multiple workers. This does not only speed up deep learning but also makes it attractive for the enhancement of other Big Data applications such as large scale numerical simulations [5]. The implementation is purely in Python language and is executed on LISA through bash scripts. For the implementation of image classifier model, we rely on the ML library Tensorflow [6]. The management and distribution of parallel training on different threats is managed by the Horovod library [7].

Bonus part of the assignment is to use a pre-trained state of the art image classifier and use transfer learning to adopt it to our specific task. Thus we start by introducing our self-build model and then the implementation of transfer learning for the two models EfficientNet b0 and ResNet50 as well a final fine-tune. We follow the implementation example for transfer learning given on the Keras site [8].

The home-made model consists of two convolution layers, followed by max-pooling, dropout and flattening of the image data. The hidden state is then passed through two fully-connected and one dropout layer. The resulting vector is activated by the softmax function and represents the categorical probability of the input image for each of the 10 classes.

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
```

```
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Keras is a high-level interface for Tensorflow, which between others provides the user with pre-trained models [9]. We use this feature to implement EfficientNet B0 [10]. The model comes with pre-trained parameters, yet on a different dataset. Therefore we implement a different input layer, which adopts to the image size of the CIFAR10 dataset. Further we add a custom output layer to match the prediction vector to the number of classes `num_classes`= 10. To avoid changing the pre-trained parameters during training, we freeze all EfficientNet B0 contained parameters,

```
effNet = tf.keras.applications.EfficientNetB0(
    include_top=False,
    weights="imagenet",
    input_tensor=None,
    input_shape=(32, 32, 3),
    classes=1000,
    classifier_activation=None,
    )
effNet.trainable = False
inputs = Input(shape=(32, 32, 3))
x = effNet(inputs, training=False)
x = GlobalAveragePooling2D()(x)
outputs = Dense(num_classes, activation='softmax')(x)
model = Model(inputs, outputs)
```

A similar implementation for ResNet50 as base model [11]:

```
resNet = tf.keras.applications.ResNet50(
    include_top=False,
    weights="imagenet",
    input_tensor=None,
    input_shape=(32, 32, 3),
    classes=1000,
    classifier_activation=None,
    )
```

In a final step, the pre-trained model is fine-tuned by unfreezing all parameters and training for one more epoch with a low learning rate of $lr_{ft} = 10^{-5}$. This allows the model to make small adjustments after it successfully transfer learned the new data.

```
model.trainable = True
opt = tf.keras.optimizers.Adadelta(1e-5)
opt = hvd.DistributedOptimizer(opt)
model.compile(loss=tf.keras.losses.sparse_categorical_crossentropy, optimizer=opt,
                                        metrics=['accuracy'])
model.fit(x_train, y_train,
          batch_size=batch_size,
          callbacks=callbacks,
          epochs=1,
          verbose=1 if hvd.rank()==0 else 0,
          validation_data=(x_test, y_test))
```

In table 1 the parameter count for the different models is displayed. We see that the pre-trained models have a higher total parameter count, yet in terms of trainable parameters the custom model has order $\mathcal{O}(10^3)$ more parameters.

	Custom model	EfficientNet B0	ResNet50
total	1,626,442	4,062,381	23,628,902
trainable	1,626,442	12,810	41,190

Table 1: Total and trainable parameter count for the image classifiers.

3 Parallel Machine Learning

In this section the results of our experiments are presented and compared. We train the three different models under similar conditions. The implementation allows parallel computing on up to three nodes. GPU nodes on LISA have four graphics processing units (GPU) which allow a parallel machine learning task on each. Thus, reserving more nodes for a job results in a multiple of four parallel work distribution. The Python library Horovod recognizes the number of available GPUs and automatically creates the fitting number of workers. Further it also adjusts the learning rate and epochs to the number of workers. The expected speedup factor is $s = 4$ per additional node.

Table 2 shows the results of profiling the performance of training the different models.

	Custom model	EfficientNet B0	ResNet50
test accuracy	0.737 ± 0.016	0.100 ± 0.011	0.108 ± 0.009
time on 1 node	7m31.477s	7m54.129s	16m55.145s
time on 2 nodes	12m33.988s	7m51.971s	16m49.621s
time on 3 nodes	7m42.059s	7m53.392s	8m23.146s

Table 2: Test accuracy and elapsed training time for each model and number of nodes.

The results of our experiments are presented in table 2. The experiments are evaluated on the metrics accuracy and run-time. The accuracy indicates how accurate the model learns to predict the data. The presented accuracy is averaged over three runs. In each run the model first completes the training on the data set and following is evaluated on its prediction accuracy on the unseen test set. In this experiment the baseline accuracy is a random guess. Considering that we predict on 10 classes, the baseline accuracy is 0.1. We observe three abnormalities which we discuss further in the nest section:

- The custom model outperforms both pre-trained models,
- The two pre-trained do not outperform the baseline.
- Fine-tuning worsened the models prediction accuracy.

The time was logged using the Linux command `time`, which prints the required time for the process for the three different timeframes `real`, `user` ,`system` [12]. The presented time is the wall clock time for training he model. The same script is run requesting $1, 2$ and 3 nodes. In theory, the use of more nodes allows the work distribution to more workers, implying higher parallelism and thus speedup of the training. Against our expectations we observe the following:

- The custom model needs more time when trained on 2 nodes than on one.
- EfficientNet takes the same time to train, disregarding the number of nodes available.
- ResNet50 experiences a speedup of 2, halving its training time. when run on 3 nodes.
- `real` and `sys` time are similar, whereas `user` time is 150% higher.

Besides the speedup of 2 for ResNet50 trained on 3 nodes, the measured times do not align with our theoretical expectations. The pre-trained models did both not outperform the baseline, which indicates that transfer learning failed. Further we note that most of the time was spent inside the kernel, which is the `sys` time. During this time the model is training and Horovod [7] should parallelize the workload on the available nodes, which should speed up the training. These observations are also discussed in the next section.

4 Conclusion

Concluding this report, we discuss the unexpected results. Rather than providing the correct answers, we aim at asking the right questions, and proposing new hypotheses based on the acquired insights.

Why do the pre-trained models perform so poorly?

Looking at table 1 we assume, that the models do not have enough parameters to transfer their knowledge form the original to the new dataset. The keyword dataset brings us to the next assumption. Taking a glance at the Imagenet dataset [13] which both models were pre-trained on, we notice a much higher number of data points and classes, further resulting in a higher variance of the data. Combining the facts that our most simple custom model performed adequately on the CIFAR10 dataset and that EfficientNet and ResNet50 have 4 and 20 times more parameters, raises the question, if these parameter-heavy models overfit on such simple data?

The additional step of fine-tuning showed, that the performance does not improve when updating the pre-trained parameters. This enforced the previous assumption, that both models are unproportionality complex.

The second elephant in the room is:

Why does the processing time not behave as predicted when using multiple nodes?

A simple answer, free of responsibility is, that we suspect LISA to in fact not assign the right resources requested for the experiments., Reasons for this could be an erroneous design of the job script, missing user rights or simply a faulty implementation of the Horovod multithreading part.

Enforcing this assumption is that the used account did not have the rights to access GPU nodes, and instead ran all experiments on solely CPU nodes. Furthermore the experiments could not be repeated due to a job waiting time of more than one day and finally an exceeded budget.

None the less we want to wrap this report up with good results. The decline in processing time for the training of ResNet50 indicates that multiple nodes do in fact speedup, yet, most probably not linearly. The observed speedup factor from one to three nodes is $s_3 = 2$, thus we get a speedup factor of $s = \frac{2}{3}$ per additional node.

An unsolved mystery to the curiosity of the author stays the vast difference between `real` and `user` time.

List of Tables

1 Total and trainable parameter count for the image classifiers. 2
2 Test accuracy and elapsed training time for each model and number of nodes. . . . 3

References

[1] IBM. Ibm cloud education. `https://www.ibm.com/cloud/learn/education`, 2 2021.

[2] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[3] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[4] Song Feng, ETH Torsten Hoefler, Switzerland Jessy Li, Damian Podareanu, Qifan Pu, Judy Qiu, Vikram Saletore, Mikhail E Smorkalov, and Jordi Torres. Valeriu codreanu, surfsara, netherlands ian foster, uchicago & anl, usa zhao zhang, tacc, usa.

[5] Caspar van Leeuwen, Damian Podareanu, Valeriu Codreanu, Maxwell X Cai, Axel Berg, Simon Portegies Zwart, Robin Stoffer, Menno Veerman, Chiel van Heerwaarden, Sydney Otten, et al. Deep-learning enhancement of large scale numerical simulations. *arXiv preprint arXiv:2004.03454*, 2020.

[6] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[7] Alexander Sergeev and Mike Del Balso. Horovod: fast and easy distributed deep learning in TensorFlow. *arXiv preprint arXiv:1802.05799*, 2018.

[8] Keras. Transfer learning & fine-tuning. https://keras.io/guides/transfer_learning/, February 2021.

[9] Nikhil Ketkar. Introduction to keras. In *Deep learning with Python*, pages 97–111. Springer, 2017.

[10] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International Conference on Machine Learning*, pages 6105–6114. PMLR, 2019.

[11] Zifeng Wu, Chunhua Shen, and Anton Van Den Hengel. Wider or deeper: Revisiting the resnet model for visual recognition. *Pattern Recognition*, 90:119–133, 2019.

[12] Stack Overflow. What do 'real', 'user' and 'sys' mean in the output of time. https://stackoverflow.com/questions/556405/what-do-real-user-and-sys-mean-in-the-output-of-time1, February 2021.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.