Technische Universiteit Eindhoven
University of Technology

Department of Mathematics and Computer Science
Data Mining Research Group

Benchmark Studies of Various Deep Learning Architecture

Data Mining Seminar

Irfan Nur Afif

Supervisors:
Joaquin Vanschoren

version 1.0

Eindhoven, December 2017

Contents

Chapter 1

Introduction

1.1 Context & Motivation

Machine learning has become an integral part of today's technology. It has a lot of applications in our daily lives, for example a recommender system or a prediction models. One of the machine learning technique that we often see nowadays is deep learning. It is the latest topics in the machine learning research which is proven to do well in solving complex classification task such as image and speech recognition [5]. The ability to discover intricate structure makes it strong tools for high dimensional data processing such as image data. In this research, we are primarily interested to explore deep learning on image datasets.

The challenging part for doing a deep learning is deciding the architecture to use for a given image datasets. There is no exact guidelines on designing a deep learning architecture. Several architecture has been proposed for a specific datasets, however we rarely see the performance comparison of the proposed design for the same datasets. Such comparison will tells us in what cases does an architecture performs better compared to the other. A comparison also tells us whether there exists an architecture that generally works well for a general image dataset or what kind of datasets criteria that works well in an architecture.

To solve this problem, we propose a benchmarking studies of multiple deep learning architecture on many image datasets. The goal of the research is to have a benchmark analysis of various deep learning architecture performance on multiple image datasets.

1.2 Research Question

The works tries to answer the following research question: "How does the performance comparison of deep learning architecture model looks like for a given image classification dataset?" In answering this research question, the approach that we try to use is to implement the state-of-the art and widely-used deep learning architecture in various datasets and analyze its performance. Some results from previous related research paper will also be used for benchmarking. There are also some sub-questions to solve the main research questions, which are:

1. Which architecture that works bests for a given datasets?
2. What kind of datasets characteristics that makes a deep learning architecture works well?
3. Is there any architecture that generally works well for image classification?

Chapter 2

Literature Study

In this chapter, we describe the related state-of-the-art and widely used deep learning architecture for image processing. The datasets that will be used for experiment will also explained in this section.

2.1 Deep Learning

Deep learning is a special form of neural networks that uses complex model to solve a problem. Deep learning technique that heavily used for dealing with image classification problem is convolutional neural network (CNN). In CNN, usually the model goes into three kinds of layers (other than the input and output layers).

The first layer type is convolutional layer. In this layer an element wise multiplication operation is implemented using a moving kernel/filter. A convolutional layer produces some feature maps for the next process. The output of the convolutional layer is usually connected to an activation function such as ReLU, tanh or sigmoid. The second type of layer is subsampling/pooling layer. This layer's function is to reduce the number of tuned parameters. The common operators for subsampling layer are: max-pooling and average pooling. The last type is fully-connected layers. In this layer, the networks try to determine the probability of the input falls into each class by learning the high level abstraction from convolutional and maxpooling layer output. There is also an optional layer that serves as regularization layer such as dropout layer. There are a lot of deep learning architecture that was proposed. Below, we highlight some of the most interesting architecture to be tested in the experiment.

2.1.1 LeNet-5

LeNet-5 was proposed by Yann LeCun, et al. [6] consists of seven non-input layers The first layer is a 5x5 convolutional layer with six feature maps. The second layer is a 2x2 non-overlapping subsampling layer. The third layer is a 5x5 convolutional layer with sixteen feeature maps. The fourth layer consists of 2x2 non-overlapping subsampling layer. The fifth layer is a 5x5 convolutional layer with 120 feeature maps. The sixth layer is an 84-units fully-connected layer. The output layer is composed of Euclidean RBF units, one for each class, with 84 inputs each. The architecture of LeNet-5 can be seen in figure 2.1. LeNet-5 is one of the first initial architectures of CNN that was tested to classify hand-written number using MNIST dataset.

2.1.2 Alex-Net

Alex Net was proposed by Alex Krizhevsky, et al. [5] based on their winning on ILSVRC (ImageNet Large-Scale Visual Recognition Challenge) 2012. The architecture consists of eight layers: five convolutional and three fully-connected layers. The first convolutional layer is using a 11x11 filter size with a stride of 4. The other convolutional layer use 3x3 filter size. The subsampling layer

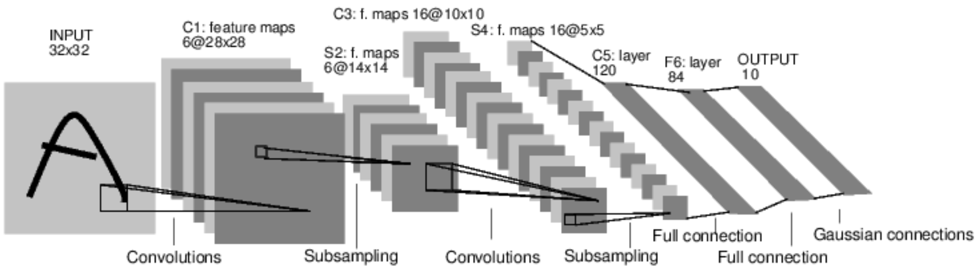

Figure 2.1: LeNet-5 Architecture [6]

that was used is max-pooling. An architecture of Alex-Net can be seen in figure 2.2. It was tested on ILSVRC 2012 dataset and achieves top 5 test error rate of 15.4%.

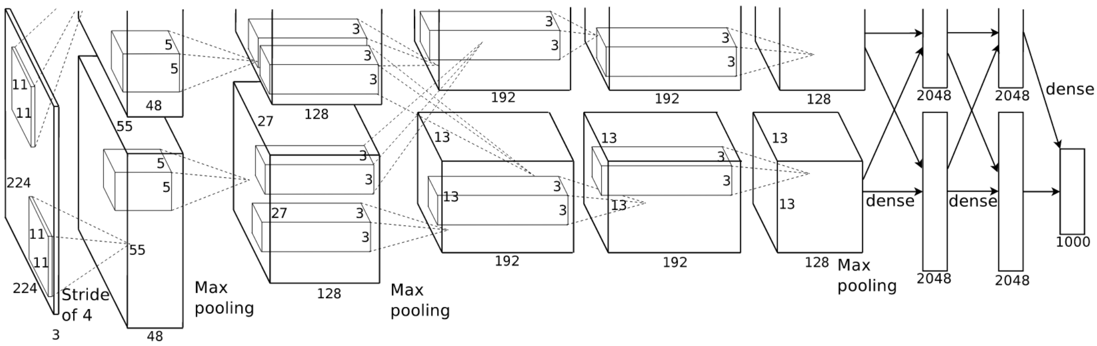

Figure 2.2: Alex Net Architecture [5]

2.1.3 VGG Net

VGG Net was proposed by Karen Simonyan and Andrew Zisserman [8] based on their winning on ILSVRC 2014. The architecture has 6 variations as shown in figure 2.3. The key idea of this architecture is using a 3x3 filter size with a stride of 1 for every convolutional layer. In subsampling layer, VGG-Net uses 2x2 max pooling layer with stride size of 2. It was tested on ILSVRC-2012 dataset and can achieve a top-5 test error of 6.8%.

2.1.4 ResNet

In 2015, Microsoft proposed a very deep architecture called ResNet [1]. It can go up to 1202 layers, but the best results for CIFAR-10 dataset is 6.43% error rate by using 110 layers. The idea behind ResNet is stacking convolutional layer with 3x3 filter size until it reaches a very deep network. This initial version of ResNet also called ResNetV1. In 2016, ResNetV2 is introduced [2] as improvement from ResNetV1.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 2.3: VGG Net Configuration [8]

2.2 Dataset

Benchmarking the proposed deep learning architecture can be done by collecting some datasets to evaluate the architecture that was presented before. Here we presents some interesting image datasets.

2.2.1 MNIST

The MNIST Datasets (http://yann.lecun.com/exdb/mnist/) is a dataset of handwritten digits with 784 features (28x28 grayscale images). There are 10 classes, 60,000 training examples and 10,000 testing examples. An example of MNIST dataset can be seen in figure 2.4.

We choose this dataset because it is heavily used as validation datasets for image recognition learning algorithm. Also it doesn't need preprocessing and formatting steps since the data is quite clean, thus we can focus on the learning implementation.

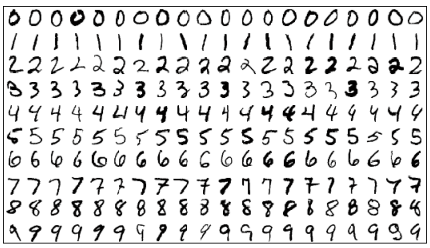

Figure 2.4: An example of MNIST dataset

2.2.2 Fashion-MNIST

Fashion-MNIST [9] is a dataset of Zalando's article images consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Zalando intends Fashion-MNIST to serve as a direct drop-in replacement for the original MNIST dataset for benchmarking machine learning algorithms. It shares the same image size and structure of training and testing splits.

Compared to MNIST, this datasets are quite new. Thus, unlike MNIST, these datasets are not heavily studied. We choose this datasets because of the similarities with MNIST datasets in terms of image size and representation.

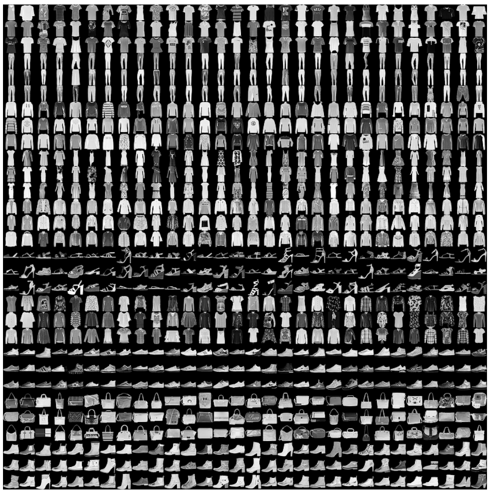

Figure 2.5: Fashion-MNIST sprite. Each three rows in the sprite corresponds to a single class example. [9]

2.2.3 CIFAR-10

CIFAR-10 is a labeled subset of the 80 million tiny images dataset that were collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton [4]. It consists of 32x32 color images representing 10 classes of objects: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. An example of CIFAR-10 dataset can be seen in figure 2.6.

CIFAR-10 contains 6000 images per class. The original train-test split randomly divided these into 5000 train and 1000 test images per class. The classes are completely mutually exclusive. For example, there is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.

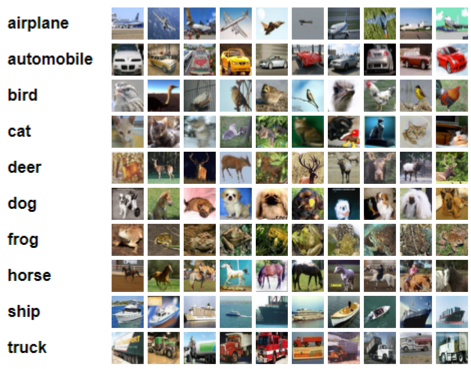

Figure 2.6: Example of CIFAR-10 dataset [4]

2.2.4 SVHN

The Street View House Numbers (SVHN) Dataset [7] is a dataset obtained from house numbers in Google Street View images. It is similar to MNIST (e.g., the images are of small cropped digits), but incorporates an order of magnitude more labeled data (over 600,000 digit images) and comes from a significantly harder, unsolved, real world problem (recognizing digits and numbers in natural scene images).

There are 10 classes, 73257 digits for training, 26032 digits for testing, and 531131 additional, somewhat less difficult samples, to use as extra training data. There are two formats of this datasets. The one that we are consider is the second format which is an MNIST-like dataset with 32-by-32 images centered around a single character (many of the images do contain some distractors at the sides).

Figure 2.7: Examples of SVHN datasets [7]

Chapter 3

Experiment

In this chapter, experiment setup will be presented. Any adjustment of from the original architecture implementation also stated in here. Lastly, dataset preprocessing step is explained here.

3.1 Experiment Setup

All of the experiment is executed on portable computer with specification as follows:

Processor	Intel Core i7 6700HQ Processor
RAM	8.0 GB
GPU	NVIDIA GeForce GTX 960M
VRAM	2.0 GB

For training process, we use Keras with TensorFlow as backend library. Python version that was used is 3.6. For every experiment, we limit it to 10 epochs because the amount to train 1 epoch is quite time consuming for a certain architecture.

3.1.1 Architecture Implementation

Since we are testing on a less powerful machine, we have to adjust the implementation of the architecture as follows.

LeNet

There are no adjustment on implementing LeNet from its original version [6] since the network itself is quite simple and straightforward. The order of the layer as follows: the first layer consists of 6 convolution with filter size of 3x3 and tanh activation function. The second layer consists of MaxPooling layer with a size of 2x2. The third layer is 16 convolution with 2x2 filter size and tanh activation function. The fourth layer is MaxPooling with pool size of 2x2. The fifth layer is flatten layer. The sixth layer is 256 fully-connected layer with tanh activation function. The last layer is 10 fully connected layer with softmax activation function. We use adadelta as the optimizer and categorical cross entropy as the loss. The data was run with 10 epoch and batch size of 128. The code to implement LeNet can be seen on Appendix A.1 and the overview of the architecture design can be seen on 3.1. All of the codes are available at https://github.com/lightzard/dm-seminar .

VGG-Net

We propose simpler version of VGG-Net because the data is quite small. If we use the original version of VGG, we are running out of input dimension when doing training. Also, implementing original VGG-Net is quite slow and consumes a lot of memory on current machine. The simpler version architecture can bee seen on 3.2 and the code can be seen on Appendix A.2. As we can

Layer (type)	Output Shape	Param #
conv2d_9 (Conv2D)	(None, 26, 26, 6)	60
max_pooling2d_7 (MaxPooling2	(None, 13, 13, 6)	0
conv2d_10 (Conv2D)	(None, 12, 12, 16)	400
max_pooling2d_8 (MaxPooling2	(None, 6, 6, 16)	0
flatten_4 (Flatten)	(None, 576)	0
dense_8 (Dense)	(None, 256)	147712
dense_9 (Dense)	(None, 10)	2570

Figure 3.1: LeNet architecture overview

see in the simplified version of VGG-Net, there are only 4 convolutional layer compared to the simplest version of VGG which has 8 convolutional layer. But the design is similar: we have 3x3 filter size in convolutional layer and MaxPool layer in-between convolutional layer.

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 28, 28, 32)	320
conv2d_2 (Conv2D)	(None, 28, 28, 64)	18496
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 64)	0
dropout_1 (Dropout)	(None, 14, 14, 64)	0
conv2d_3 (Conv2D)	(None, 14, 14, 128)	73856
conv2d_4 (Conv2D)	(None, 12, 12, 256)	295168
max_pooling2d_2 (MaxPooling2	(None, 4, 4, 256)	0
dropout_2 (Dropout)	(None, 4, 4, 256)	0
flatten_1 (Flatten)	(None, 4096)	0
dense_1 (Dense)	(None, 256)	1048832
leaky_re_lu_1 (LeakyReLU)	(None, 256)	0
dropout_3 (Dropout)	(None, 256)	0
dense_2 (Dense)	(None, 256)	65792
leaky_re_lu_2 (LeakyReLU)	(None, 256)	0
dense_3 (Dense)	(None, 10)	2570

Figure 3.2: Simpler VGG-Net architecture overview

ResNet V1

We implement the simplest form of ResNetV1 which is ResNet20V1 due to hardware constraint. In this variation, there are 20 stacks of "resnet block". Each block consists of 2 x (3 x 3) Convolution-Batch Normalization-ReLU layer. In applying ResNet, we use ResNet builder library that are available at https://github.com/keras-team/keras/blob/master/examples/cifar10_resnet.py. Full implementation code can be see at Appendix A.3.

ResNet V2

Implementing ResNet V2 is the same as ResNetV1. We just need to change the parameter of n=2 (so that the depth is equal to 20) and version=2 in the ResNet builder. The difference of ResNet V2 and V1 is that, in V2 each block consists of (1 x 1)-(3 x 3)-(1 x 1) Batch Normalization-ReLU-Convolution layer. At the beginning of each 'stage', the feature map size is halved (downsampled) by a convolutional layer with strides=2, while the number of filter maps is doubled. Within each 'stage', the layers have the same number filters and the same filter map sizes.

AlexNet/SqueezeNet

Initially we plan to implement AlexNet. But, when we try to implement it on the current machine we failed to implement it because the number of trainable parameters is too big. Then, we discover SqueezeNet [3], a simpler version of AlexNet that could achieve similar accuracy with less memory resource and training time. The architecture idea can be seen on 3.3. The architecture consists of convoultional layer and fire module. A fire module consists of 1x1 convolutional layer followed by a mix of 1x1 and 3x3 convolution layer. Then, it gradually increase the number of filters per fire module from the beginning to the end of the network. SqueezeNet performs max-pooling with a stride of 2 after layers conv1, fire4, fire8, and conv10.

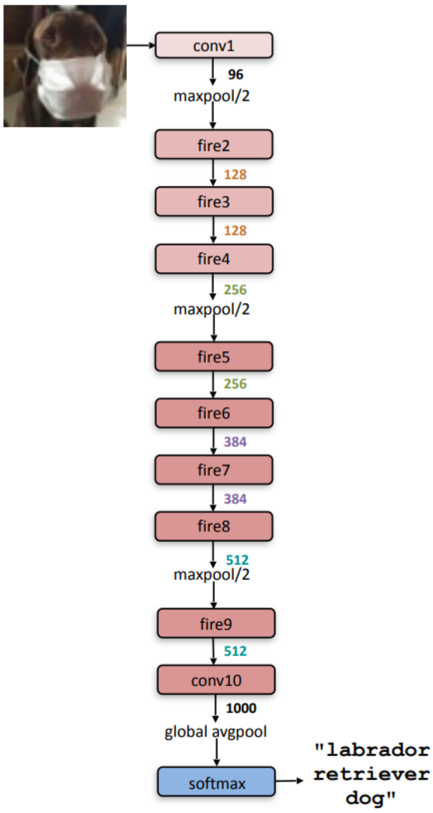

Figure 3.3: SqueezeNet architecture

This implementation is slight modification from SqueezeNet implementation on MNIST dataset as available on https://github.com/titu1994/Kaggle/blob/master/MNIST/SqueezeNet.py

3.1.2 Dataset Preprocessing

MNIST, Fashion MNIST and CIFAR-10 dataset are available directly from `keras.datasets` package. The code to load these 3 datasets can be seen on Appendix A.5, A.6 and A.7 respectively. As we can see from these codes, the only pre-processing steps applied for these 3 datasets are reshaping input to (28,28,1) (for MNIST and Fashion-MNIST) and converting classes from numerical type to categorical type. For SVHN dataset, we have to load the data manually and apply preprocessing. First step is to transpose the data. The next step is to rename label '10' to '0' because every '0' images in the dataset is labelled as '10' by default. The next step is to take balanced subsample for training data because the size of the training data is too big. We took 6000 images per class randomly as training data thus we have 60000 images as training data and 26032 testing data since we don't do any sampling on testing data. Full code can be seen on A.8

Chapter 4

Results and Discussion

In this chapter we report the results of the training and validation of the neural networks described in the experimental approach. Then, we discuss the results in terms of the research questions defined. Lastly, we assess the limitations of our experimental procedure and results

4.1 Experiment Result

Table 4.1 shows the result of the experiments in terms of accuracy. The execution times and number of parametes for each experiment is shown on Table 4.2 and Figure 4.1 respectively.

		MNIST	Fashion MNIST	CIFAR-10	SVHN
LeNet-5	total params	150,742	150,742	204,098	204,098
	trainable params	150,742	150,742	204,098	204,098
	non-trainable params	0	0	0	0
VGG-like	total params	1,505,034	1,505,034	1,505,610	1,505,610
	trainable params	1,505,034	1,505,034	1,505,610	1,505,610
	non-trainable params	0	0	0	0
resnet20v1	total params	274,090	274,090	274,442	274,442
	trainable params	272,746	272,746	273,066	273,066
	non-trainable params	1,344	1,344	1,376	1,376
resnet20v2	total params	573,738	573,738	574,090	574,090
	trainable params	570,282	570,282	570,602	570,602
	non-trainable params	3,456	3,456	3,488	3,488
squeezenet	total params	711,956	711,956	720,084	720,084
	trainable params	711,956	711,956	720,084	720,084
	non-trainable params	0	0	0	0

Figure 4.1: Number of parameters for each experiment.

	MNIST	Fashion MNIST	CIFAR-10	SVHN
LeNet-5	0.9834	0.8816	0.6561	0.809
VGG-like	0.9946	0.91	0.4075	0.067
Resnet20v1	0.9246	0.592	0.7567	0.866
Resnet20v2	0.9222	0.8425	0.679	0.893
SqueezeNet	0.9858	0.8813	0.555	0.775

Table 4.1: Accuracy table

	MNIST	Fashion MNIST	CIFAR-10	SVHN
LeNet-5	220s	220s	330s	220s
VGG-like	4986s	4469s	6816s	6526s
Resnet20v1	7970s	7909s	7204s	9310s
Resnet20v2	13900s	13910s	12060s	15693s
SqueezeNet	13800s	13907s	13630s	17550s

Table 4.2: Execution time table

4.2 Discussion

In this section, we discuss results of the experiments to answer the following research questions that was previously stated: "How does the performance comparison of deep learning architecture model looks like for a given image classification dataset?" From the execution time, it is shown that LeNet is the fastest network to train (around 22 seconds/epoch) followed by Simplified VGG-Net, Resnet20V1. ResNet20V2 and SqueezeNet are the most time consuming network to train. LeNet is also the least memory consuming network, followed by ResNet20V1, ResNet20V2, SqueezeNet and Simplified VGG network respectively. In terms of accuracy, there are no single architecture that produces the best result for each datasets. Then, we still need to answer these following additional research questioned that was stated earlier.

1. Which architecture that works bests for a given datasets?

   From table 4.1, we can see that Simplified VGG architecture performs best on MNIST and Fashion MNIST dataset. ResNet20V1 and ResNet20V2 performs best on CIFAR-10 and SVHN dataset respectively. In terms of memory usage and time consumption, it is shown that LeNet produces the simplest network for all datasets.

2. What kind of datasets characteristics that makes a deep learning architecture works well?

   From table 4.1, we can see that LeNet generally works well across all dataset except CIFAR-10. With a low amount of training time and low memory consumption, this network can be an initial tester for classifying an image dataset.

   Simplified VGGNet performs well on grayscale dataset: MNIST and Fashion-MNIST but gives low accuracy for colored dataset: CIFAR-10 and SVHN dataset. This is an indication that maybe VGG-like architecture performs better on grayscale dataset compared to colored dataset. Also since CIFAR-10 and SVHN image size are bigger than MNIST and Fashion-MNIST, it could be an indication that Simplified VGG-Net works better on smaller image dataset.

   ResNet20v1 gives high accuracy in digit recognition dataset: MNIST and SVHN but performs worse on object recognition dataset: Fashion-MNIST and CIFAR-10. It could be that the architecture is suitable for digit recognition dataset compared to object-recognition dataset.

   ResNet20v2 which is a refinement from ResNet20v1 does not always increase the performance of ResNet20v1. However, on Fashion MNIST dataset, the improvemance is quite huge, around 25% accuracy. This architecture gives the most stable performance compared to other architecture, with the lowest accuracy is 67.9% on CIFAR-10 dataset.

   SqueezeNet is the most time consuming network to train, however it does not yields the best solution for any dataset. We need to test it on bigger epoch and see whether it can produces better result or not. If it is indeed yields a better result, then we could say that this architecture has slow improvement rate for every epoch. If is not, then we could say that this architecture is not suitable for these datasets.

3. Is there any architecture that generally works well for image classification?

There are no single architecture that gives best performance across all datasets. However, based on table 4.1 we can see that in general, ResNet20v2 gives the most stable and good performance on average across all datasets. This does not means that this architecture always gives stable and good performance on most datasets. Further experiment needs to be done to confirm this findings.

4.3 Limitation

Due to the time and hardware contraint, this studies was conducted only on 4 datasets and 5 architectures. A more powerful machine/hardware is needed to do further benchmarking studies. As an example, applying ResNet20V1 on CIFAR-10 dataset, using current machine took 720 seconds/epoch while applying it on GTX1080Ti took 35 seconds/epoch (according to https://github.com/keras-team/keras/blob/master/examples/cifar10_resnet.py). Faster experiment means better classification performance and more architecture and dataset that could be implemented.

Chapter 5

Conclusion

Based on the findings on the previous section, we can summarize it as follows:

1. There are no single architecture that gives the best performance across all tested dataset.
2. LeNet is the most simple yet powerful network. This networks can become the tester network for in an image dataset because of its low time to train and memory to use. We can use LeNet classification result as initial performance benchmarks compared to other sophisticated architecture.
3. Simplified VGGNet performs well on grayscale dataset compared to colored-dataset. Also it performs better on smaller image. It can achieve an accuracy of 99.46% and 91% on MNIST and Fashion-MNIST dataset respectively.
4. ResNetV1 works better in digit recognition compared to object recognition. It can achieve an accuracy of 92.46% and 86.6% on MNIST and SVHN dataset.
5. ResNetV2 does not always increase the performance of ResNetv1. However, on Fashion MNIST dataset, the improvemance is quite huge, around 25% accuracy.
6. ResNetV2 gives quite good and stable performance compared to other network.
7. SqueezeNet consumes the most memory and time to train but does not yields the best classification performance on any dataset.

5.1 Future Works

Further works can be done by doing larger benchmark studies by applying more architecture and more dataset in the experiment. By doing this, we can also verifying the initial findings that was presented in the conclusion. Increasing the number of epoch can be a good way to verify the initial finding that was presented here. Furthermore, we only see memory usage, time consumption and accuracy for the experiment. Other matrices should be explored for future works.

Bibliography

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 3

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016. 3

[3] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size. *arXiv:1602.07360*, 2016. 9

[4] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. 2009. 5

[5] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 1, 2, 3

[6] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. 2, 3, 7

[7] Yuval Netzer, Tao Wang, Adam Coates, Alessandro Bissacco, Bo Wu, and Andrew Y Ng. Reading digits in natural images with unsupervised feature learning. In *NIPS workshop on deep learning and unsupervised feature learning*, volume 2011, page 5, 2011. 6

[8] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 3, 4

[9] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017. 4, 5

Appendix A

Python Code

```
model = Sequential()
model.add(Conv2D(6, (3, 3), activation="tanh", input_shape=(28, 28, 1)))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(16, (2, 2), activation="tanh"))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(256, activation="tanh"))
model.add(Dense(10, activation="softmax"))
model.compile("adadelta", "categorical_crossentropy", metrics=['accuracy'])
model.summary()
model.fit(x_train, y_train, batch_size=128, epochs=10, validation_data=(x_test,
    y_test))

score = model.evaluate(x_test, y_test, verbose=1)
```

Listing A.1: LeNet code

```
model=Sequential()
model.add(Conv2D(filters=32, kernel_size=(3, 3), padding="same",
input_shape=x_train.shape[1:], activation='relu'))
model.add(Conv2D(filters=64, kernel_size=(3, 3), padding="same", activation='relu')
    )
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))
model.add(Conv2D(filters=128, kernel_size=(3, 3), padding="same", activation='relu'
    ))
model.add(Conv2D(filters=256, kernel_size=(3, 3), padding="valid", activation='relu
    '))
model.add(MaxPooling2D(pool_size=(3, 3)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(256))
model.add(LeakyReLU())
model.add(Dropout(0.5))
model.add(Dense(256))
model.add(LeakyReLU())
model.add(Dense(10, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adadelta', metrics=['
    accuracy'])
model.summary()

model.fit(x_train, y_train, batch_size=128, epochs=10, validation_data=(x_test,
    y_test))

score = model.evaluate(x_test, y_test, verbose=1)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

Listing A.2: VGG-like Net Code

```
#resnet https://github.com/keras-team/keras/blob/master/examples/cifar10_resnet.py

# Training parameters
batch_size = 32  # orig paper trained all networks with batch_size=128
epochs = 10
data_augmentation = True
num_classes = 10

#Adjusting both parameter for using ResNet V1 or V2 and its depth
n = 3
version = 1

# Computed depth from supplied model parameter n
if version == 1:
depth = n * 6 + 2
elif version == 2:
depth = n * 9 + 2

# Model name, depth and version
model_type = 'ResNet%dv%d' % (depth, version)

print('x_train2 shape:', x_train2.shape)
print(x_train2.shape[0], 'train samples')
print(x_test2.shape[0], 'test samples')
print('y_train shape:', y_train2.shape)


def lr_schedule(epoch):
"""Learning Rate Schedule
Learning rate is scheduled to be reduced after 80, 120, 160, 180 epochs.
Called automatically every epoch as part of callbacks during training.
# Arguments
epoch (int): The number of epochs
# Returns
lr (float32): learning rate
"""
lr = 1e-3
if epoch > 180:
lr *= 0.5e-3
elif epoch > 160:
lr *= 1e-3
elif epoch > 120:
lr *= 1e-2
elif epoch > 80:
lr *= 1e-1
print('Learning rate: ', lr)
return lr

def resnet_first_block(inputs,
num_filters=16,
kernel_size=3,
strides=1,
activation='relu',
batch_normalization=True,
conv_first=True):

x = ZeroPadding2D(padding=(2, 2), data_format=None)(inputs)

y = Conv2D(num_filters,
kernel_size=kernel_size,
strides=strides,
padding='same',
kernel_initializer='he_normal',
kernel_regularizer=l2(1e-4))(x)

return y
```

```
def resnet_block(inputs,
num_filters=16,
kernel_size=3,
strides=1,
activation='relu',
batch_normalization=True,
conv_first=True):
"""2D Convolution-Batch Normalization-Activation stack builder
# Arguments
inputs (tensor): input tensor from input image or previous layer
num_filters (int): Conv2D number of filters
kernel_size (int): Conv2D square kernel dimensions
strides (int): Conv2D square stride dimensions
activation (string): activation name
batch_normalization (bool): whether to include batch normalization
conv_first (bool): conv-bn-activation (True) or
activation-bn-conv (False)
# Returns
x (tensor): tensor as input to the next layer
"""
x = inputs
if conv_first:
x = Conv2D(num_filters,
kernel_size=kernel_size,
strides=strides,
padding='same',
kernel_initializer='he_normal',
kernel_regularizer=l2(1e-4))(x)
if batch_normalization:
x = BatchNormalization()(x)
if activation:
x = Activation(activation)(x)
return x
if batch_normalization:
x = BatchNormalization()(x)
if activation:
x = Activation('relu')(x)
x = Conv2D(num_filters,
kernel_size=kernel_size,
strides=strides,
padding='same',
kernel_initializer='he_normal',
kernel_regularizer=l2(1e-4))(x)
return x


def resnet_v1(input_shape, depth, num_classes=10):
if (depth - 2) % 6 != 0:
raise ValueError('depth should be 6n+2 (eg 20, 32, 44 in [a])')
# Start model definition.
inputs = Input(shape=input_shape)
num_filters = 16
num_sub_blocks = int((depth - 2) / 6)

x = resnet_first_block(inputs=inputs)
# Instantiate convolutional base (stack of blocks).
for i in range(3):
for j in range(num_sub_blocks):
strides = 1
is_first_layer_but_not_first_block = j == 0 and i > 0
if is_first_layer_but_not_first_block:
strides = 2
y = resnet_block(inputs=x,
num_filters=num_filters,
strides=strides)
y = resnet_block(inputs=y,
num_filters=num_filters,
```

```
activation=None)
if is_first_layer_but_not_first_block:
x = resnet_block(inputs=x,
num_filters=num_filters,
kernel_size=1,
strides=strides,
activation=None,
batch_normalization=False)
x = keras.layers.add([x, y])
x = Activation('relu')(x)
num_filters = 2 * num_filters

# Add classifier on top.
# v1 does not use BN after last shortcut connection-ReLU
x = AveragePooling2D(pool_size=8)(x)
y = Flatten()(x)
outputs = Dense(num_classes,
activation='softmax',
kernel_initializer='he_normal')(y)

# Instantiate model.
model = Model(inputs=inputs, outputs=outputs)
return model


def resnet_v2(input_shape, depth, num_classes=10):

if (depth - 2) % 9 != 0:
raise ValueError('depth should be 9n+2 (eg 56 or 110 in [b])')
# Start model definition.
inputs = Input(shape=input_shape)
num_filters_in = 16
num_filters_out = 64
filter_multiplier = 4
num_sub_blocks = int((depth - 2) / 9)

# v2 performs Conv2D with BN-ReLU on input before splitting into 2 paths
x = resnet_block(inputs=inputs,
num_filters=num_filters_in,
conv_first=True)

# Instantiate convolutional base (stack of blocks).
activation = None
batch_normalization = False
for i in range(3):
if i > 0:
filter_multiplier = 2
num_filters_out = num_filters_in * filter_multiplier

for j in range(num_sub_blocks):
strides = 1
is_first_layer_but_not_first_block = j == 0 and i > 0
if is_first_layer_but_not_first_block:
strides = 2
y = resnet_block(inputs=x,
num_filters=num_filters_in,
kernel_size=1,
strides=strides,
activation=activation,
batch_normalization=batch_normalization,
conv_first=False)
activation = 'relu'
batch_normalization = True
y = resnet_block(inputs=y,
num_filters=num_filters_in,
conv_first=False)
y = resnet_block(inputs=y,
```

```
num_filters=num_filters_out,
kernel_size=1,
conv_first=False)
if j == 0:
x = resnet_block(inputs=x,
num_filters=num_filters_out,
kernel_size=1,
strides=strides,
activation=None,
batch_normalization=False)
x = keras.layers.add([x, y])

num_filters_in = num_filters_out

# Add classifier on top.
# v2 has BN-ReLU before Pooling
x = BatchNormalization()(x)
x = Activation('relu')(x)
x = AveragePooling2D(pool_size=8)(x)
y = Flatten()(x)
outputs = Dense(num_classes,
activation='softmax',
kernel_initializer='he_normal')(y)

# Instantiate model.
model = Model(inputs=inputs, outputs=outputs)
return model


if version == 2:
model = resnet_v2(input_shape=input_shape, depth=depth)
else:
model = resnet_v1(input_shape=input_shape, depth=depth)

model.compile(loss='categorical_crossentropy',
optimizer=Adam(lr=lr_schedule(0)),
metrics=['accuracy'])
model.summary()
print(model_type)
save_dir = os.path.join(os.getcwd(), 'saved_models')
model_name = 'cifar10_%s_model.{epoch:03d}.h5' % model_type
if not os.path.isdir(save_dir):
os.makedirs(save_dir)
# Prepare model model saving directory.
filepath = os.path.join(save_dir, model_name)
# Prepare callbacks for model saving and for learning rate adjustment.
checkpoint = ModelCheckpoint(filepath=filepath,
monitor='val_acc',
verbose=1,
save_best_only=True)

lr_scheduler = LearningRateScheduler(lr_schedule)

lr_reducer = ReduceLROnPlateau(factor=np.sqrt(0.1),
cooldown=0,
patience=5,
min_lr=0.5e-6)

callbacks = [checkpoint, lr_reducer, lr_scheduler]

# Run training, with or without data augmentation.
if not data_augmentation:
print('Not using data augmentation.')
model.fit(x_train2, y_train2,
batch_size=batch_size,
epochs=epochs,
validation_data=(x_test2, y_test2),
```

```
shuffle=True,
callbacks=callbacks)
else:
print('Using real-time data augmentation.')
# This will do preprocessing and realtime data augmentation:
datagen = ImageDataGenerator(
# set input mean to 0 over the dataset
featurewise_center=False,
# set each sample mean to 0
samplewise_center=False,
# divide inputs by std of dataset
featurewise_std_normalization=False,
# divide each input by its std
samplewise_std_normalization=False,
# apply ZCA whitening
zca_whitening=False,
# randomly rotate images in the range (deg 0 to 180)
rotation_range=0,
# randomly shift images horizontally
width_shift_range=0.1,
# randomly shift images vertically
height_shift_range=0.1,
# randomly flip images
horizontal_flip=True,
# randomly flip images
vertical_flip=False)

# Compute quantities required for featurewise normalization
# (std, mean, and principal components if ZCA whitening is applied).
datagen.fit(x_train2)

# Fit the model on the batches generated by datagen.flow().
model.fit_generator(datagen.flow(x_train2, y_train2, batch_size=128),
validation_data=(x_test2, y_test2),
epochs=10, verbose=1, workers=-1,
callbacks=callbacks)

# Score trained model.
scores = model.evaluate(x_test2, y_test2, verbose=1)
print('Test loss:', scores[0])
print('Test accuracy:', scores[1])
```

Listing A.3: ResNet Code

```
#conv 1
conv1 = Convolution2D(96, 3, 3, activation='relu', init='glorot_uniform',subsample
    =(2,2),border_mode='valid')(input_layer)

#maxpool 1
maxpool1 = MaxPooling2D(pool_size=(2,2))(conv1)

#fire 1
fire2_squeeze = Convolution2D(16, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(maxpool1)
fire2_expand1 = Convolution2D(64, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire2_squeeze)
fire2_expand2 = Convolution2D(64, 3, 3, activation='relu', init='glorot_uniform',
    border_mode='same')(fire2_squeeze)
merge1 = merge(inputs=[fire2_expand1, fire2_expand2], mode="concat", concat_axis=1)
fire2 = Activation("linear")(merge1)

#fire 2
fire3_squeeze = Convolution2D(16, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire2)
fire3_expand1 = Convolution2D(64, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire3_squeeze)
fire3_expand2 = Convolution2D(64, 3, 3, activation='relu', init='glorot_uniform',
```

```
    border_mode='same')(fire3_squeeze)
merge2 = merge(inputs=[fire3_expand1, fire3_expand2], mode="concat", concat_axis=1)
fire3 = Activation("linear")(merge2)

#fire 3
fire4_squeeze = Convolution2D(32, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire3)
fire4_expand1 = Convolution2D(128, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire4_squeeze)
fire4_expand2 = Convolution2D(128, 3, 3, activation='relu', init='glorot_uniform',
    border_mode='same')(fire4_squeeze)
merge3 = merge(inputs=[fire4_expand1, fire4_expand2], mode="concat", concat_axis=1)
fire4 = Activation("linear")(merge3)

#maxpool 4
maxpool4 = MaxPooling2D((2,2))(fire4)

#fire 5
fire5_squeeze = Convolution2D(32, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(maxpool4)
fire5_expand1 = Convolution2D(128, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire5_squeeze)
fire5_expand2 = Convolution2D(128, 3, 3, activation='relu', init='glorot_uniform',
    border_mode='same')(fire5_squeeze)
merge5 = merge(inputs=[fire5_expand1, fire5_expand2], mode="concat", concat_axis=1)
fire5 = Activation("linear")(merge5)

#fire 6
fire6_squeeze = Convolution2D(48, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire5)
fire6_expand1 = Convolution2D(192, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire6_squeeze)
fire6_expand2 = Convolution2D(192, 3, 3, activation='relu', init='glorot_uniform',
    border_mode='same')(fire6_squeeze)
merge6 = merge(inputs=[fire6_expand1, fire6_expand2], mode="concat", concat_axis=1)
fire6 = Activation("linear")(merge6)

#fire 7
fire7_squeeze = Convolution2D(48, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire6)
fire7_expand1 = Convolution2D(192, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire7_squeeze)
fire7_expand2 = Convolution2D(192, 3, 3, activation='relu', init='glorot_uniform',
    border_mode='same')(fire7_squeeze)
merge7 = merge(inputs=[fire7_expand1, fire7_expand2], mode="concat", concat_axis=1)
fire7 =Activation("linear")(merge7)

#fire 8
fire8_squeeze = Convolution2D(64, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire7)
fire8_expand1 = Convolution2D(256, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire8_squeeze)
fire8_expand2 = Convolution2D(256, 3, 3, activation='relu', init='glorot_uniform',
    border_mode='same')(fire8_squeeze)
merge8 = merge(inputs=[fire8_expand1, fire8_expand2], mode="concat", concat_axis=1)
fire8 = Activation("linear")(merge8)

#maxpool 8
maxpool8 = MaxPooling2D((2,2))(fire8)

#fire 9
fire9_squeeze = Convolution2D(64, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(maxpool8)
fire9_expand1 = Convolution2D(256, 1, 1, activation='relu', init='glorot_uniform',
    border_mode='same')(fire9_squeeze)
fire9_expand2 = Convolution2D(256, 3, 3, activation='relu', init='glorot_uniform',
    border_mode='same')(fire9_squeeze)
```

```
merge8 = merge(inputs=[fire9_expand1, fire9_expand2], mode="concat", concat_axis=1)
fire9 = Activation("linear")(merge8)
fire9_dropout = Dropout(0.5)(fire9)

#conv 10
conv10 = Convolution2D(10, 1, 1, init='glorot_uniform',border_mode='valid')(
    fire9_dropout)

#avgpool 1
avgpool10 = AveragePooling2D((13,13), strides=(1,1), border_mode='same')(conv10)

flatten = Flatten()(avgpool10)

softmax = Dense(10, activation="softmax")(flatten)

model = Model(input=input_layer, output=softmax)

model.summary()

model.compile(optimizer='adadelta', loss="categorical_crossentropy", metrics=["
    accuracy"])
model.fit(x_train, y_train, batch_size=128, epochs=10, validation_data=(x_test,
    y_test))

score = model.evaluate(x_test, y_test, verbose=1)
print('Test score:', score[0])
print('Test accuracy:', score[1])
```

Listing A.4: SqueezeNet Code

```
from keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
print("x_train shape: {}".format(x_train.shape))
print("y_train shape: {}".format(y_train.shape))
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)

y_train = np_utils.to_categorical(y_train, 10)
y_test = np_utils.to_categorical(y_test, 10)
```

Listing A.5: Code for loading MNIST dataset

```
from keras.datasets import fashion_mnist

(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()
print("x_train shape: {}".format(x_train.shape))
print("y_train shape: {}".format(y_train.shape))
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)

y_train = np_utils.to_categorical(y_train, 10)
y_test = np_utils.to_categorical(y_test, 10)
```

Listing A.6: Code for loading Fashion MNIST dataset

```
from keras.datasets import cifar10

(x_train, y_train), (x_test, y_test) = cifar10.load_data()
y_train = keras.utils.to_categorical(y_train, 10)
y_test = keras.utils.to_categorical(y_test, 10)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

Listing A.7: Code for loading CIFAR-10 dataset

```
def load_data(path):
""" Helper function for loading a MAT-File"""
data = loadmat(path)
return data['X'], data['y']

X_train, y_train = load_data('train_32x32.mat')
X_test, y_test = load_data('test_32x32.mat')

X_train, y_train = X_train.transpose((3,0,1,2)), y_train[:,0]
X_test, y_test = X_test.transpose((3,0,1,2)), y_test[:,0]

print("Training", X_train.shape)
print("Test", X_test.shape)
print('')

# Calculate the total number of images
num_images = X_train.shape[0] + X_test.shape[0] + X_extra.shape[0]

y_train[y_train == 10] = 0
y_test[y_test == 10] = 0

def balanced_subsample_categorical(y, s):
"""Return a balanced subsample of the population"""
sample = []
# For every label in the dataset
for ii in range(10):
# Get the index of all images with a specific label
images = np.ndarray.flatten(np.where((y[:, ii] == 1))[0])
print(images)
# Draw a random sample from the images
random_sample = np.random.choice(images, size=s, replace=False)
# Add the random sample to our subsample list
sample += random_sample.tolist()
return sample

# Pick 1000 samples per class from the training samples
train_samples = balanced_subsample_categorical(y_train, 6000)
X_train = np.take(X_train,train_samples,axis=0)
y_train = np.take(y_train,train_samples,axis=0)
```

Listing A.8: Code for loading SVHN dataset