Visual Recognition Assignment 3

Swasti Shreya Mishra(IMT2017043), Ananya Appan(IMT2017004),
Seelam Lalitha(IMT2017027)

April 26, 2020

1 Part A

1.1 Introduction

In this part we try to illustrate the difference between feature extraction and feature learning. **Classical Machine Learning** approach requires us to come up with the features that we give as input to the models explicitly. Where as in **Deep Learning** approach, the model learns the features on its own during the training process.

1.2 Classical Machine Learning (Feature Extraction)

Here we are using classical machine learning models such as Linear Models to classify the images after the features are extracted. The features extracted are just the raw pixel value array of the image (flattened array). Some of the classical linear models used as a multi-class classifier and the accuracy gained by them on the **MNIST dataset** are as follows:

NOTE: LR - Linear Regression. The “sag” solver uses Stochastic Average Gradient descent. It is faster than other solvers for large datasets, when both the number of samples and the number of features are large. The “saga” solver is a variant of “sag” that also supports the non-smooth penalty=“l1”. This is therefore the solver of choice for sparse multinomial logistic regression. It is also the only solver that supports penalty=“elasticnet”.

Linear Models	**Accuracy**
Ridge Regression	0.625
Lasso Regression	0.626
Bayesian Ridge Regression	0.626
LR (penalty=‘l1’)	0.914
LR (penalty=‘elasticnet’)	0.915
LR (penalty=‘elasticnet’, l1_ratio=0.5, multi_class=‘multinomial’)	0.923

From the above observations we can notice that, using raw features along with classical ML models, we get a maximum accuracy of around 92%.

1.3 Deep Learning (Feature Learning)

Deep learning has shown a significant performance and accuracy gain in the field of computer vision. A neural network containing over 60 million parameters significantly beat previous state-of-the-art approaches to image recognition in a popular ImageNet computer vision competition: ISVRC-2012. The boom started with the convolutional neural networks and the modified architectures of ConvNets. By now it is said that some convNet architectures are so close to 100% accuracy of image classification challenges, sometimes beating the human eye!

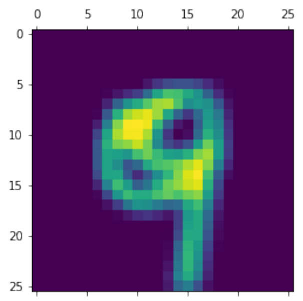
Let us now look at different architectures of Neural Networks used for multi-class classification task on the **MNIST** dataset:

NOTE: Conv - Convolution layer; Pool - Pooling layer; FC - Fully connected layer; Drop - Dropout layer; TT - Training Time in seconds

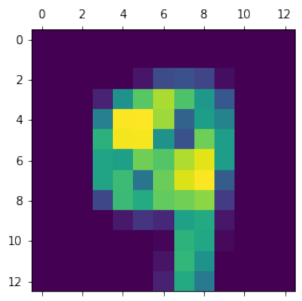
Neural Networks	**Accuracy**	**TT**
Conv + Pool + FC-1 + Drop(0.2) + FC-2	0.984	240
Conv + FC-1 + Drop(0.2) + FC-2	0.978	580
Conv-1 + Pool-1 + Conv-2 + Pool-2 + FC-1 + Drop(0.2) + FC-2	0.988	150
Conv + Pool + FC-1 + FC-2	0.983	160
Conv + Pool + FC-1 + Drop(0.4) + FC-2	0.977	160
Conv + Pool + FC-1 + Drop(0.1) + FC-2 + Drop(0.2) + FC-3	0.983	170
Conv-1+Pool-1+Conv-2+Pool-2+FC-1+Drop(0.1)+FC-2+Drop(0.2)+FC-3	0.990	160

1.4 Activation Maps

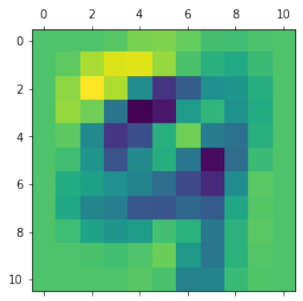
Activation map after Conv 1



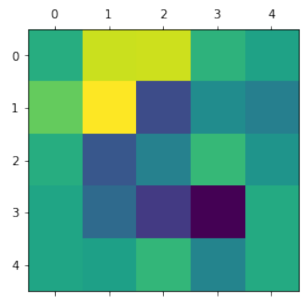
Activation map after Conv1+Pool1



Activation map after Conv1+Pool1+Conv2

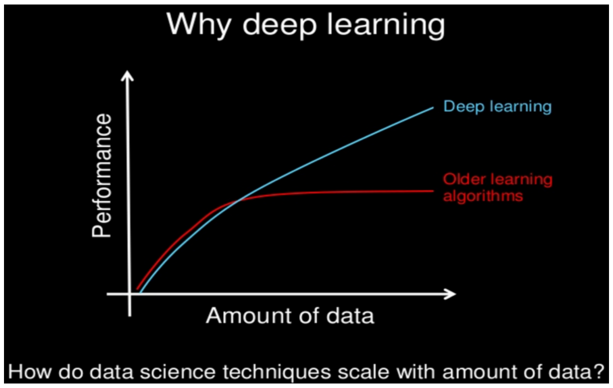


Activation map after Conv1+Pool1+Conv2+Pool2



1.5 Conclusion

As the graph below suggests, feature learning i.e. using Deep Learning techniques is beneficial when we have enough training data with us (like in our case the MNIST dataset). Convolutional Neural Networks can outperform classical Machine Learning models in such scenarios.



2 Part B

2.1 Introduction

As we have seen in the previous assignments, trying to use traditional ML methods for image classification on large datasets such as cifar-10 can give an accuracy of around 45% to 55% at best. We aim to show here that using Convolutional Neural Networks, we can get far better accuracies. We experiment with different factors of the Network- including the architecture of the network (Dropout, Batch Normalization and zero padding), the optimizer used and the number of training epochs.

2.2 Architectures Used

We trained three different networks. These are described below.

1. **ShallowNet** : This network is quite "shallow" with one convolutional layer block with two convolutional layers, and one fully connected layer block with three fully connected layers. This was used to experiment around with factors like optimization strategies used and Dropout.

```
    class ShallowNet(nn.Module):
        def __init__(self):
            super(ShallowNet, self).__init__()
            self.conv1 = nn.Conv2d(3, 6, 5)
            self.pool = nn.MaxPool2d(2, 2)
            self.conv2 = nn.Conv2d(6, 16, 5)
            self.fc1 = nn.Linear(16 * 5 * 5, 120)
            self.fc2 = nn.Linear(120, 84)
            self.fc3 = nn.Linear(84, 10)
            self.dropout = nn.Dropout(0.1)

        def forward(self, x):
```

```python

            x = self.pool(F.relu(self.conv1(x)))
            x = self.pool(F.relu(self.conv2(x)))
            x = x.view(-1, 16 * 5 * 5)
            x = self.dropout(x)
            x = F.relu(self.fc1(x))
            x = self.dropout(x)
            x = F.relu(self.fc2(x))
            x = self.dropout(x)
            x = self.fc3(x)
            return x

```

Listing 1: ShallowNet

2. **MediocreNet** : This network is middle deep with two convolutional layer blocks with two convolutional layers each. It was used to experiment with how changes in architecture affect training time.

```python
    class MediocreNet(nn.Module):

        def __init__(self, activation_fn, zero_padding,dropout_2d,
    dropout):
            super(MediocreNet, self).__init__()

            self.conv_layer = nn.Sequential(

                # Conv Layer block 1
                nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3,
    padding=zero_padding),
                nn.BatchNorm2d(32),
                activation_fn(),
                nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
     padding=zero_padding),
                activation_fn(),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.Dropout2d(p=dropout_2d),

                # Conv Layer block 2
                nn.Conv2d(in_channels=64, out_channels=128, kernel_size
    =3, padding=zero_padding),
                nn.BatchNorm2d(128),
                activation_fn(),
                nn.Conv2d(in_channels=128, out_channels=256, kernel_size
    =3, padding=zero_padding),
                activation_fn(),
                nn.MaxPool2d(kernel_size=2, stride=2),
                nn.Conv2d(in_channels=256, out_channels=256, kernel_size
    =3, padding=1),
                activation_fn(),
                nn.MaxPool2d(kernel_size=2, stride=2),
            )


            self.fc_layer = nn.Sequential(
                nn.Dropout(p=dropout),
                nn.Linear(4096, 1024),
                activation_fn(),
                nn.Linear(1024, 512),
                activation_fn(),
                nn.Dropout(p=dropout),
                nn.Linear(512, 10)
            )


        def forward(self, x):
            x = self.conv_layer(x)
            x = x.view(x.size(0), -1)
            x = self.fc_layer(x)

            return x

```

Listing 2: MediocreNet

3. **AwesomeNet** : This is a deep network with three convolutional layer blocks with two convolutional layers each, and one fully connected layer block with three fully connected layers. This Network was used to experiment around with factors like Batch Normalization, Zero padding, 2d Dropout and activation functions.

```
class AwesomeNet(nn.Module):

    def __init__(self, activation_fn, zero_padding,dropout_2d,
dropout):
        super(AwesomeNet, self).__init__()

        self.conv_layer = nn.Sequential(

            # Conv Layer block 1
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3,
padding=zero_padding),
            nn.BatchNorm2d(32),
            activation_fn(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
 padding=zero_padding),
            activation_fn(),
            nn.MaxPool2d(kernel_size=2, stride=2),

            # Conv Layer block 2
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size
=3, padding=zero_padding),
            nn.BatchNorm2d(128),
            activation_fn(),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size
=3, padding=zero_padding),
            activation_fn(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Dropout2d(p=dropout_2d),

            # Conv Layer block 3
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size
=3, padding=zero_padding),
            nn.BatchNorm2d(256),
            activation_fn(),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size
=3, padding=zero_padding),
            activation_fn(),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )

```

```python

        self.fc_layer = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(4096, 1024),
            activation_fn(),
            nn.Linear(1024, 512),
            activation_fn(),
            nn.Dropout(p=dropout),
            nn.Linear(512, 10)
        )


    def forward(self, x):
        x = self.conv_layer(x)
        x = x.view(x.size(0), -1)
        x = self.fc_layer(x)
        return x

```

Listing 3: AwesomeNet

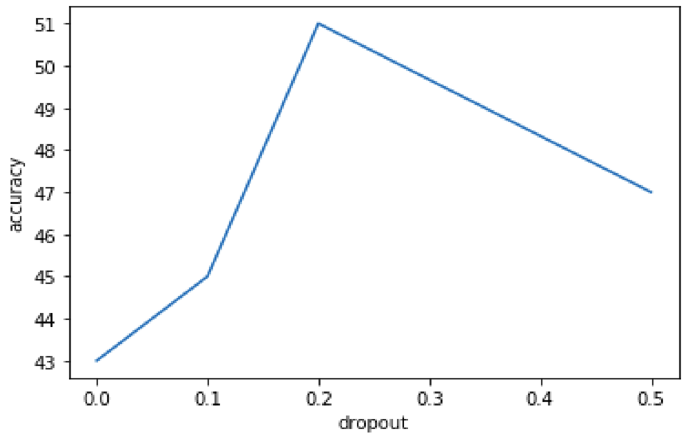
2.3 Experimenting with ShallowNet

2.3.1 Dropout

Dropout refers to the probability with which units (nodes) in a neural network are ignored during while training. This is done primarily as a regularization measure, in order to ensure that overfitting does not happen. It helps re-weight neurons, so that the individual features are captured, as well as co dependencies.

In ShallowNet, we apply dropout in the fully connected block, after the activation function.

The following graph shows the variation of accuracy with dropout.



As we can see, we get a maximum accuracy of 51% with a dropout of 0.2 . We can infer from this that a dropout of less than 0.2 tends to make the model overfit the training data, while a dropout of greater than 0.2 leads to too much bias.

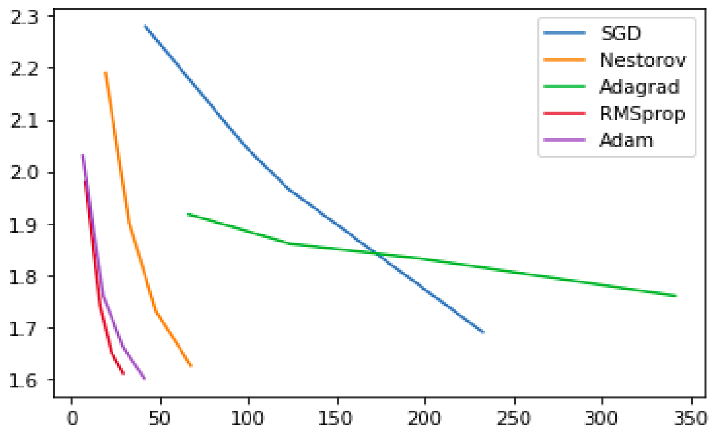
2.4 Optimization Algorithms

The accuracies on running ShallowNet on the following optimization algorithms is given below. A momentum of 0.9 was used.

Optimizer	Accuracy
SGD	43%
Nesterov	52%
Adagrad	32%
RMSprop	52%
Adam	54%

As we can see, we get a maximum accuracy of 54% using Adam as our optimizer. Using Adagrad has given us an unexpectedly low accuracy of 32%. Adagrad tends to increase velocity along the parameter with less change in gradient, and vice versa. Hence, it slows you down and prevents overshoot. However, if you get stuck in a saddle point, it will be very difficult to get out. While all other optimization strategies try to build up velocity, Adagrad brings it down. This may be the reason for our observed accuracies.

Given below is a graph of batch loss with a mini batch size of 2000, versus time.



Here, we see that using RMS Prop helps reach the minimum quicker, but reduction in batch loss stagnates after some time. Adam in comparison, reaches the minimum almost as fast. The delay is because we overshoot the minimum a little. However, as we build up velocity, we are able to reduce our batch loss further. Adagrad on the other hand, starts off with a very low batch loss. But the slope of it's graph is almost horizontal, indicating that it may have reached a local minima.

2.5 Experimenting with MediocreNet

The following table shows the time taken to train MediocreNet and the resultant accuracy for activation functions Sigmoid, Tanh and ReLU. The network was trained using a learning rate of 0.001 and a momentum of 0.9.

Activaion Function	Accuracy	Time taken
Sigmoid	10%	1m 46s
Tanh	72%	1m 50s
ReLU	76%	1m 45s

From the above results, it is clear that ReLU is computationally less expensive. This will be clear if we see the expressions for each activation function.

Activaion Function	Expression
Sigmoid	$\frac{1}{1+e^{-x}}$
Tanh	$\frac{2}{1+e^{-2x}}+1$
ReLU	$\begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$

Tanh and Sigmoid have an exponential term which makes them more computationally expensive than ReLU.

2.6 Experimenting with AwesomeNet

2.6.1 Activation Functions

AwesomeNet was used to experiment with Batch Normalization and momentum for ReLU, Sigmoid and Tanh activation functions. SGD was used as the optimizer with a momentum of 0.9.

Experiments	Sigmoid	Tanh	ReLU
Without Batch Norm and momentum	10%	63%	52%
With Batch Norm and no momentum	10%	64%	69%
With Batch Norm and momentum	10%	71%	75%

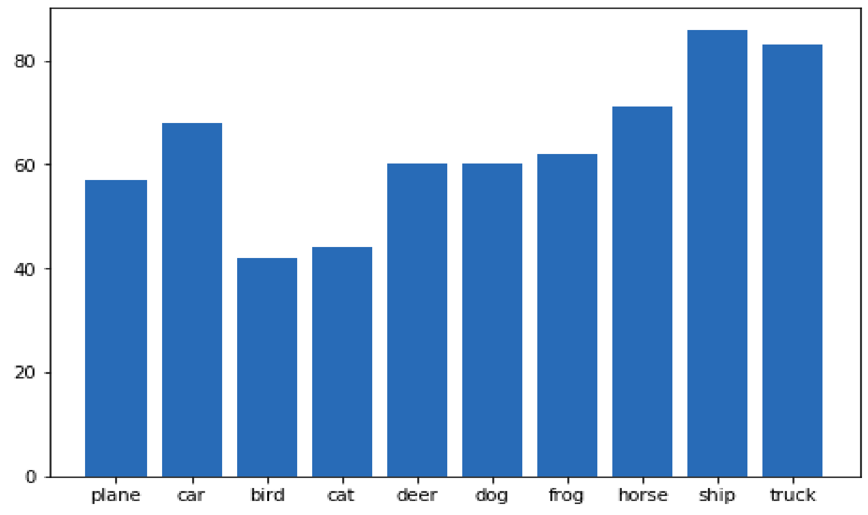
The above results show that ReLU is definitely the best activation function and outperforms Sigmoid and Tanh except when batch normalization is not used. In this case, Tanh outperforms ReLU.

The reason behind this could be that ReLLU is **not a zero centered** activation function. Hence, ReLU gives an output which is either positive or negative, causing the gradient to proceed in a zig zag fashion during gradient descent. This may have even landed us in a local minima.

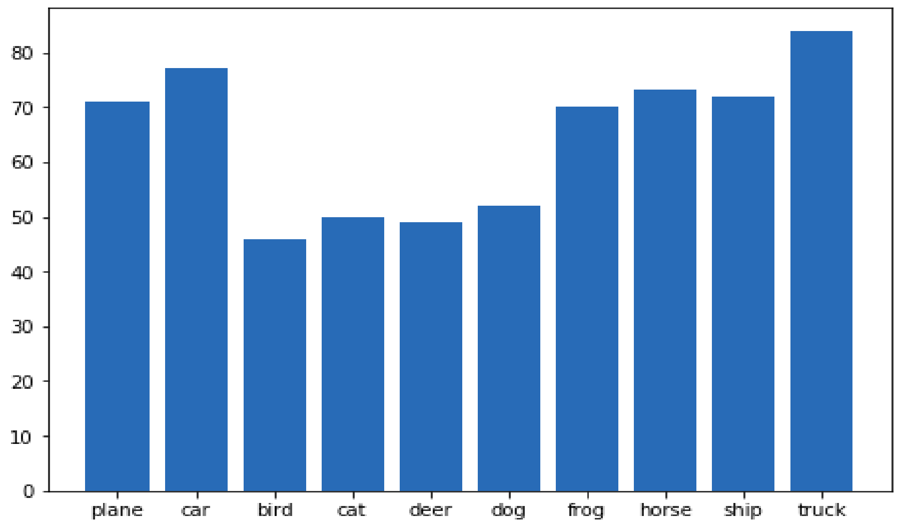
We also see that Sigmoid has not performed well, with an accuracy of only 10%. This was when the images of the dataset were normalized with a standard deviation and mean of 0.5 for all three channels and a learning rate of 0.9 was used with SGD as the optimizer. In order to get better accuracies, we tried a different used a different transform to normalize the data, and used a learning rate of $1e^{-3}$ with Adam as out optimizer. Here are the outputs.

Experiment	Accuracy
With Batch Norm and without momentum	28%
Without Batch Norm and with momentum	32%
With Batch Norm and momentum	53%

2.6.2 Analysis of Batch Normalization



Category wise accuracy without batch normalization



Category wise accuracy with batch normalization

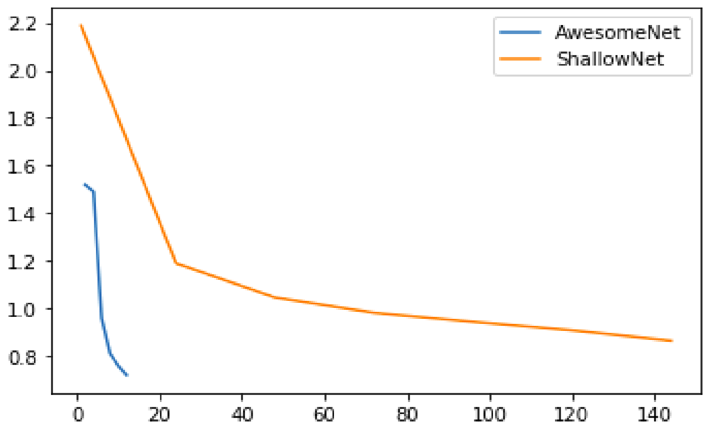
The above bar charts show the category wise accuracy with and without batch normalization while using Tanh as the activation function. We can see from the above table that the overall accuracy only increased by 1% from 63% to 64%. However, from the above charts, it is quite clear the individual accuracies for each category fare much better.

While accuracies for the categories ship and horse have gone down, accuracies for plane, car and cat have gone up.

The basic idea behind batch normalization is to limit covariate shift by normalizing the activations of each layer (transforming the inputs to be mean 0 and unit variance). This allows each layer to learn on a more stable distribution of inputs, and would thus accelerate the training of the network.

2.7 Comparison of ShallowNet and AwesomeNet

The following graph compares the decrease in batch loss with number of batches for AwesmeNet and ShallowNet.



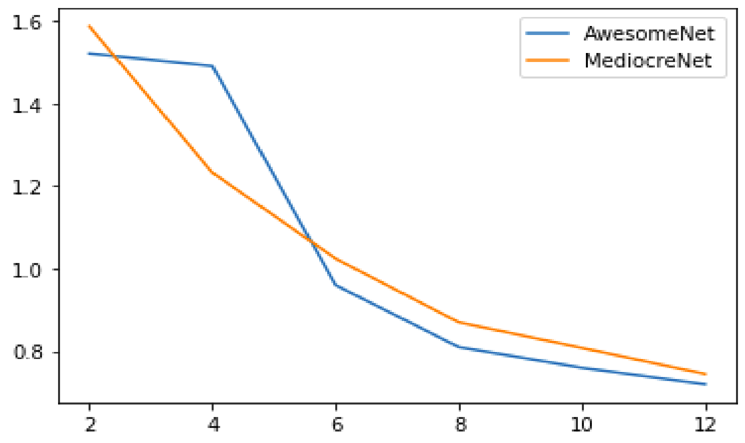
We trained ShallowNet for 25 epochs to get a final accuracy of 69%, while AwesomeNet gave an accuracy of 75% for just 2 epochs. The training time for both these networks was however the same. (around 5 min).

We would like to explore why deeper networks can give such high accuracies on the test data even when they train for much lesser epochs.

It would seem that wide, shallow networks are very good at memorization, but not so good at generalization. So, if you train the network with every possible input value, a super wide network could eventually memorize the corresponding output value that you want. Multiple layers are much better at generalizing because they learn all the intermediate features between the raw data and the high-level classification. For instance, the first layer may train itself to recognize edges, the next layer to recognize shapes, the next to recognize features like faces and so on.

2.8 Comparison of MediocreNet and AwesomeNet

The following shows the decrease in batch loss with number of batches for MediocreNet and AwesomeNet.



Both the networks were trained for two epochs. Here, we see that AwesomeNet seems to be training better with a lesser batch loss of 0.72 versus a batch loss of 0.74 for MediocreNet. However, the following table shows the accuracies of both networks on the test data for Tanh and ReLU activation functions.

Activaion Function	MediocreNet	AwesomeNet
Tanh	72%	71%
ReLU	76%	75%

It seems that MediocreNet is not so mediocre after all! We see that in both cases, it outperforms AwesomeNet. However, AwesomeNet had a lesser overall batch loss.

One explanation could be that AwesomeNet being a deeper network perhaps overfit the train data. But with Dropout and Batch normalization, this should have been taken care of. The other reason would be the same reason for the introduction of residual networks. Accuracy may have diminished over many layers due to vanishing gradients. As layers get deeper, gradients get smaller leading to worse performance.

2.9 Recommended Architecture

From the above experiments we can conclude that the following architecture would be recommendable, keeping in mind both training time and accuracy.

1. **Depth :** A medium deep network like MediocreNet. This performs better with respect to both training time as well as accuracy.
2. **Activation Function :** ReLU gives the best results and has the least computation complexity.
3. **Dropout :** A Dropout of 0.1 - 0.2 along with a 2d Dropout of 0.05.
4. **Batch Normalization :** It is always better to use batch normalization before activation, especially for classification problems.
5. **Momentum :** Is definitely preferred, especially considering convergence time. A momentum of 0.9 is recommended.

6. **Optimization Algorithm :** Adam gives the best accuracy and convergence time.

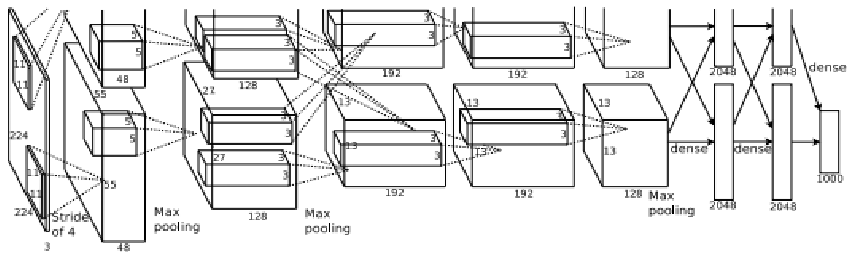
3 Part C

3.1 Introduction

In earlier assignments, we have used SIFT and SURF to get keypoints and descriptors. We then used BOVW and VLAD to get features for the images. But here we try to use Alex Net as a feature extractor for images and use any model on the top and to obtain scores for the classification.

3.2 AlexNet : Architecture

AlexNet famously won the 2012 ImageNet LSVRC-2012 competition by a large margin (15.3% VS 26.2% (second place) error rates). As per the paper ImageNet Classification with Deep Convolutional Neural Networks, AlexNet contains five convolution layers and three fully connected layers. AlexNet used rectified linear activation function or ReLU, as the non-linearity after each convolution layer instead of tanh or logistic functions. We also use max pooling methods where it was found, overlapping pooling to outperform non-overlapping pooling. To address the issue of overfitting, the newly proposed dropout method was used between the fully connected layers of the classifier part of the model to improve generalization error. The following figure summarises the AlexNet architecture



3.3 AlexNet : Processing Image

- The first convolution layer filters 224 X 224 X 3 input image with 96 kernels of size 11 X 11 X 3 with a stride of 4 pixels.
- The response-normalized and pooled output of first convolution layer is taken and passed through the second convolution layer.
- The second convolution layer filters the output of the first convolution layer using 256 kernels of size 5 X 5 X 48.
- The third, fourth and fifth convolution layers are connected to each other without any intervening pooling or normalization layers where
  - The third convolution layer filters the second convolution layers normalized and pooled output using 384 kernels of size 3 X 3 X 256

  - The fourth convolution layer filters the third convolution layer output using 384 kernels of size 3 X 3 X 192
  - The fifth convolution layer filters the fourth convolution layer output using 256 kernels of size 3 X 3 X 192

- The fifth convolution layer output is flattened and then passed through the fully connected layers.
- In the last connected layer we do a softmax to obtain the classified category

3.4 AlexNet : Feature Extractor

In the description above, the last layer is used for image classification for the 1000 categories of ImageNet. But here we try to use AlexNet as a feature extractor. So, we decided to take the output of the seventh layer (The second fully connected layer) as the feature extractor. Extracting features using AlexNet is pretty useful because the image has been processed through various layers and functions.

We've also tried taking the output of other layers as well and used them as features for our image. But the output dimension of our feature vector is 4096 (Since the dimension input and output of first and second connected layers is 4096 and the last layer output is 1000(Since ImageNet has 1000 categories), so we have also tried to reduce the dimension using PCA and obtained features for the images and compare the results.

3.5 AlexNet : Fine tuning

We can use the **Transfer Learning** technique, where we use the pre-trained AlexNet weights as the weight initialization for the model. Here while training the model, we modify the last layer of AlexNet to get the desired number of features for the classification task and then perform a softmax over those features. But, to fine tune the model, we do not freeze the gradients as we do in the feature extraction technique.
The observations of this method is in the upcoming section.

3.6 Experiments and Observations

We have used two datasets, one with over 15000 images for traning (**Caltech 101 Categories dataset**) which is a fairly large dataset and another one, a smaller dataset (**Ants and Bees**) with over 300 images. We will compare our results for the same.

We will also use last layer (1000) features of AlexNet for getting our feature vector, as well as the last but one fully connected layer (one without the activation layer) which gives us a 4096 dimension feature vector.
Here we are using the **Transfer Learning** technique by using pre-trained AlexNet (which is trained on the ImageNet dataset) to get the features for the images in our dataset.

Caltech 101 Categories dataset
Using AlexNet for feature extraction and **K-Nearest Neighbour** as the classifier:

Model	Accuracy
With last layer and PCA = 0.9	0.657
With last layer and PCA = 0.8	0.613
With last layer and without PCA	0.652
Without last two layers and without PCA	0.690
Without last two layers and and PCA = 0.8	0.637
Without last two layers and and PCA = 0.9	0.688

Using AlexNet for feature extraction and **Support Vector Classifier** as the classifier:

Model	Accuracy
With last layer and PCA = 0.9	0.7115
Without last two layers and without PCA	0.7836

Using AlextNet as a classifier (**Multi Layer Perceptron**) at the end as a classifier:

Model	Accuracy	Training Time
Modified last layer to get 101 features + Softmax	0.479401	7m 29s
Fine tuning + modified last layer to get 101 features + Softmax	0.411985	38m 57s

From the above results we can infer that, using a pre-trained model as a feature extractor can outperform the model if the model is used alone. AlexNet as a feature extractor outperformed even when we fine tune our pre trained AlexNet model on the dataset.

We also notice that, fine-tuning the model performs worse than modifying the last layer to make it a classifier for the given dataset. This tells us that the generalization is destroyed when we fine tune the model since AlexNet is trained on a much larger dataset with around 1000 classes.

Ants and Bees dataset
Using AlexNet for feature extraction and **Support Vector Classifier** as the classifier:

Model	Accuracy
Without PCA	0.8758
PCA n_components = 210	0.8823
PCA n_components = 200	0.9019
PCA n_components = 180	0.8954
PCA n_components = 100	0.8954

Using AlextNet as a classifier (**Multi Layer Perceptron**) at the end as a classifier:

Model	Accuracy	Training Time
Modified last layer to get 2 features + Softmax	0.921569	0m 45s
Fine tuning + modified last layer to get 2 features + Softmax	0.881132	4m 38s

From the above results we can infer that, on a smaller dataset, using AlexNet itself as a classifier can out perform the pre-trained AlexNet as a feature extractor. But, we also notice here that pre-trained AlexNet as a feature extractor is still better that fine-tuning the model.

3.7 Conclusion

As observed from the above experiments, the results can vary depending on the dataset. What we should take note of is that, transfer learning as a technique is very powerful and more efficient as it can give better results as well as take lesser time to train. We also note that fine tuning might make the model overfit on the dataset and hence perform bad on the validation set.

4 References

- Deep Learning Vs. Traditional Computer Vision: `https://naadispeaks.wordpress.com/2018/08/12/deep-learning-vs-traditional-computer-vision/`
- Transfer Learning for Computer Vision Tutorial: `https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html`
- Activation Functions: `https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6`
- CIFAR-10 pytorch tutorial: `https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html`
- Deep CNN tutorial for CIFAR-10: `https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch-for-cifar-10\-photo-classification/`
- AlexNet as feature extractor: `https://www.learnopencv.com/pytorch-for-beginners-image-classification-using\-pre-trained-models/`
- AlexNet Paper: `https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional\-neural-networks.pdf`