Machine Learning and Deep Learning - HW2 Report

Hadi Nejabat s246601

1. Introduction

Our task will be to train a convolutional neural network (CNN) that can identify objects in images using Pytorch. We'll be using the Caltech101 dataset which has images in 101 categories.

Figure 1 Caltech101 Dataset

Pictures of objects belonging to 101 categories. About 40 to 800 images per category. Most categories have about 50 images. Collected in September 2003 by Fei-Fei Li, Marco Andreetto, and Marc 'Aurelio Ranzato. The size of each image is roughly 300 x 200 pixels. The Image Distribution in the Caltech101 Dataset is imbalanced. If we remove the BACKGROUND Google label and its images, then we have 8677 images in total which are not enough images to get very high accuracy.

2. Material and Methods

2.1. Getting more data

This is one of the classic approaches to dealing with an imbalanced image or computer vision datasets. In this approach, we need to collect more data per category which has a smaller number of images. This method works really well as the dataset size can increase substantially. And we know that, when a neural network trains on a larger dataset, then its performance can increase substantially.

2.2. Using Data Augmentation (our approach)

It is not always possible to get more data. The dataset may be very different from other datasets, or it may be a very new dataset on which many experiments have not been done. In such cases, data augmentation works very well. Data augmentation makes the neural network to see different types of images by changing the images randomly. The changes can be in size, cropping, rotation, color, and more.

2.3. Data Preparation

One of the most important aspects of PyTorch is it having a GPU-friendly framework. Where an efficient data generation scheme is crucial to leverage the full potential of your GPU during the training process. So, first we write the initialization function of the class. We make the latter inherit the properties of torch, utils, data, Dataset so that we can later leverage nice functionalities such as multiprocessing. To implement this best is by writing a custom module to be passed into PyTorch's Dataloader.

2.4. important notes

2.4.1. About the custom module

The Custom Module "Caltech" is used for Custom data loading for all the Train, Validation and Test Datasets. it is calibrated to work with the Caltech101 Dataset and its input parameters are as follows:

Root: pass the directory your notebook and 101_ObjectCategories folder containing the dataset.

Split: pass the dataset split that you are using for creating the Data loader Obj. 'train' or 'validation' or 'test'

Split Size: pass the size you find adequate for splitting the train-Validation split

transform: pass the transformation you want to apply to the images.

2.4.2. split method

the selection of data for train and test was determined by a given .txt files containing paths to curtain images. Moreover, the demand for splitting the original train data into `Train_dataset` and `Validation_dataset` was so that each one must include images from all classes with a determined size.

To be able to meet these properly I used Stratified Splitting; where I first used one-hot encoding for labels and passed that into stratified argument of split function.

```
lb = LabelBinarizer()
labels = lb.fit_transform(labels)
(x_train, x_val , y_train, y_val) = train_test_split(data, labels, test_size=self.split_size, stratify=labels, random_state=42)
```

2.4.3. Define the Image Transforms

```
# define transforms
train_transform = transforms.Compose(
    [transforms.ToPILImage(),
        transforms.Resize((224, 224)),
     transforms.ToTensor(),
     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])])
val_transform = transforms.Compose(
    [transforms.ToPILImage(),
        transforms.Resize((224, 224)),
     transforms.ToTensor(),
     transforms.Normalize(mean=[0.485, 0.456, 0.406],
                          std=[0.229, 0.224, 0.225])])
```

we define two transforms. One is `train_transform` and the other one is `val_transform`. We will apply the `val_transform` to our training dataset and `val_transform` to the validation and test dataset. We are resizing the images to 224×224 pixels, converting them to tensors, and applying normalization as per the pre-trained ImageNet weights.

In training from scratch section both of the transforms are similar. This is for those cases when we want to apply some image augmentation to the training set but not to the validation and test set. It is common practice in deep learning and computer vision experiments to apply image augmentations to the training set only. For those situations, if we want to change the `train_transform`, then we need not touch the `val_transform`. This will enable us to carry out deep learning experiments very smoothly.

Now, we have to modify our PyTorch script accordingly so that it accepts the generator that we just created. In order to do so, we use PyTorch's DataLoader class, which in addition to our Dataset class, also takes in the following important arguments:

- **`Batch_size`:** which denotes the number of samples contained in each generated batch.
- **`Shuffle`:** If set to True, we will get a new order of exploration at each pass (or just keep a linear exploration scheme otherwise). Shuffling the order in which examples are fed to the classifier is helpful so that batches between epochs do not look alike. Doing so will eventually make our model more robust.
- **`Num_workers`:** which denotes the number of processes that generate batches in parallel. A high enough number of workers assures that CPU computations are efficiently managed, *i.e.* that the bottleneck is indeed the neural network's forward and backward operations on the GPU (and not data generation).

2.5. Models:

2.5.1. AlexNet:

AlexNet contained eight layers; the first five were convolutional layers, some of them followed by max-pooling layers, and the last three were fully connected layers.

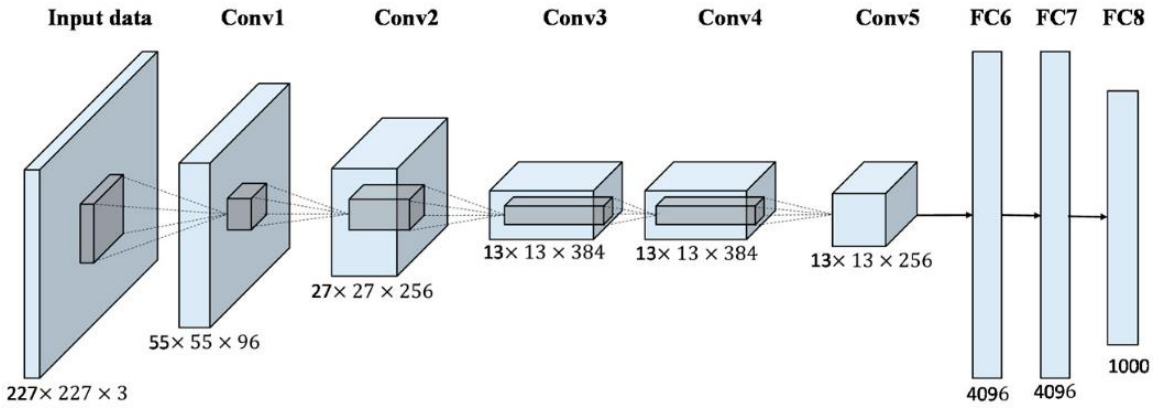


Figure 2- AlexNet Network Architecture

Features:

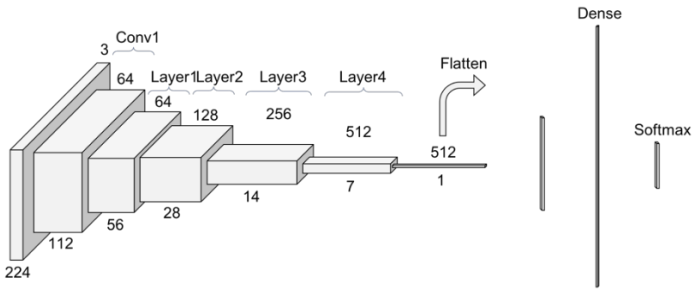
ReLU Nonlinearity: AlexNet uses Rectified Linear Units (ReLU) instead of the tanh function, which was standard at the time. ReLU's advantage is in training time; a CNN using ReLU was able to reach a 25% error on the CIFAR-10 dataset six times faster than a CNN using tanh.

Multiple GPU Support: AlexNet allows for multi-GPU training by putting half of the model's neurons on one GPU and the other half on another GPU. Not only does this mean that a bigger model can be trained, but it also cuts down on the training time.

Overlapping Pooling: CNNs traditionally "pool" outputs of neighboring groups of neurons with no overlapping. However, when the authors introduced overlap, they saw a reduction in error by about 0.5% and found that models with overlapping pooling generally find it harder to overfit.

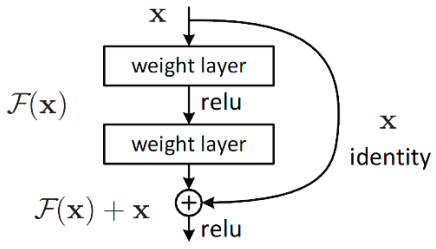
2.5.2. ResNet34:

ResNet is a short name for a residual network**.** Deep residual learning framework for image classification task. Which supports several architectural configurations, allowing to achieve a suitable ratio between the speed of work and quality.



ResNet34 Network Architecture

One of the problems ResNet solves is the famous known vanishing gradient. This is because when the network is too deep, the gradients from where the loss function is calculated easily shrink to zero after several applications of the chain rule. This result on the weights never updating its values and therefore, no learning is being performed. With ResNet, the gradients can flow directly through the skip connections backwards from later layers to initial filters. The skip connection schema is illustrated figure.



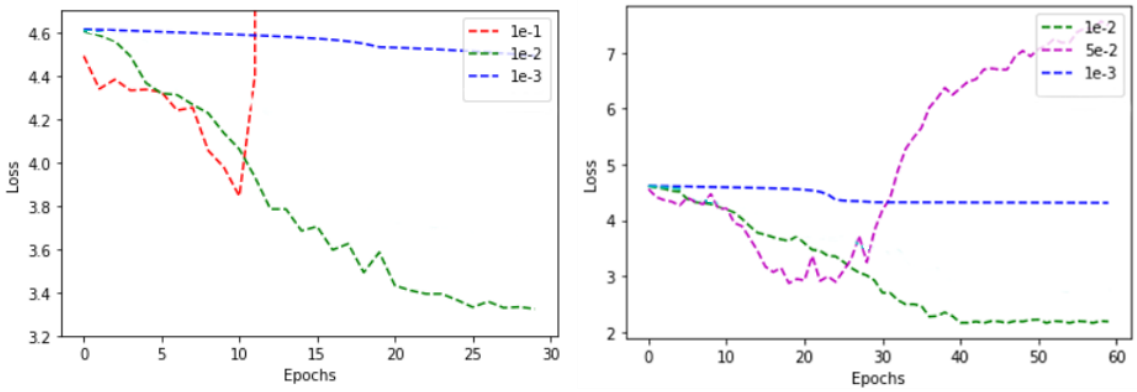
2.6. Final approach:

Our task is to train a simple Convolutional Neural Network for image classification. We will create our network with 2 different structures (AlexNet & ResNet34) and the classic optimizer and loss function. (SGD and Cross Entropy respectively)

2.6.1. Training from scratch:

At first, we will start the training with no weight initialization and just input the required hyperparameters for the network to start. Throughout the possible variations of input parameters and after many runs with wide range of parameters, I kept some of them fixed in all of the experiments (e.g., **`Batch_size =256`**) and I have only reported the ones with the most significant change both in loss and accuracy in (Table 1).

After observing the results and output diagrams of multiple runs I have encountered different obstacles resulted in me making corrections to my inputs to get is the correct path for training the model from scratch. For **Tuning LR, I** have analyzed the model's behavior by changing LR with multiple of 10 by performing a **`grid search`** and keeping other parameters unchanged. As can be seen in the chart below with a higher learning rate the model is able to reduce the cost function. Yet for learning rate 0.1 we can observe an overfitting in epoch 10. This means that the model has learned the training dataset too well and is less good to generalize to previously unseen data, resulting in an increase in the error. Moreover, we perform another hyper-parameter tuning for #Epochs. For this reason, the number of epochs has been doubled (and consequently the step size has been increased). as can be clearly seen the best learning rate found with 30 epochs confirmed to be an acceptable learning curve



Also, it is evident that my model wasn't able to train efficiently where this could be seen from little changes in losses and of course very low accuracies. Moreover, normally it is not recommended to train a network with no weight initialization and it is better to use "**Glorot Normal"** initialization as a starting point.

Since our ultimate goal is to find the best possible results by contributing the lowest run-time; it is best to allocate our time to finetune a pretrained model on large datasets such as ImageNet and try to process and improve that model. Following this Idea, I used Transfer Learning on AlexNet which was pretrained on ImageNet dataset.

2.6.2. Transfer Learning (Freezing weights):

Neural networks typically need huge amounts of data in order to be effective, in the case of classification. In order to address the issue of low accuracy when training from scratch, it's possible to use Transfer Learning.

Before feeding the net, the images are preprocessed in the same way described above. But this time we normalize the input images with the mean and standard deviation from ImageNet, which are the vectors (0.485, 0.456, 0.406) and (0.229, 0.224, 0.224). By using the same parameters of the first implementation trained the following results are obtained. As it can be seen from the results table the significant increase in performance achieved with transfer learning in comparison with the training done with the default hyperparameters: in particular, the validation accuracy is boosted from 15.4% to 97.10%.

Now, we can implement a widely used method to accelerating the training phase; by *'freezing'* some part of the network, i.e. performing the training only on a subset of the layers. By training the net with the best set of hyperparameters found so far, the results obtained are reported in the Results table. From the table we can see that freezing the convolutional layers does not impact the overall precision of the model. This is likely since that convolutional part typically deals with generic features of the images, that can be learned from ImageNet in pretraining. Instead, freezing the fully-connected layers brings to poor performances, since those are in charge of the final classification according to the specific categories in the Caltech dataset.

2.6.3. Transfer Learning (+Data Augmentation):

As it can be seen from the reported graphs, we are experiencing overfitting in the training process and best way to tackle this is by Data Augmentation. Data augmentation can include flipping, rotating, cropping, changing the color palette and many more operations. By applying these transformations to the baseline model (with frozen convolutional layer), the new improved results are reported in the table 1.

2.6.4. Beyond AlexNet (training on ResNet):

We will use ResNet34 for training on the Caltech101 dataset. And we will not be designing the architecture from scratch. In fact, we will be using a pre-trained ResNet34 model with ImageNet weights. So, we will use transfer learning which we know is a really powerful method to gain high performance improvements in deep learning. For the hidden layer weights, we will not update them. But we will fine-tune the head of the ResNet34 neural network model to support our use case. While fine-tuning, we will also add a Dropout layer.

One of the main reasons for the high accuracy in a very small number of training epochs is the additional dropout layer. You will observe that removing the dropout layer before the fully connected layer will give you worse results.

3. Results:

1- Training from scratch

	Varried inputs (Batch_size=256)			Outputs				
	Num. of Epochs	Learning Rate	Split_size	Best loss		Best Accuracy(%)		Test Accuracy(%)
				Train	Validation	Train	Validation	
Parameter tuning Phase	30	0.1	0.5	0.016	0.016	9.37	9.44	9.7
	30	0.01	0.5	0.0126	0.0128	26.28	26.76	27.5
	30	0.001	0.2	0.016	0.0035	14.57	13.14	15.4
	60	0.1	0.2	-	0.0034	-	15.44	16.57
	60	0.01	0.2	-	0.0034	-	22.12	23.25
	60	0.001	0.2	-	0.0033	-	24.76	25.89

2- Transfer Learning (input Network weights trained on ImageNet)

	Varried inputs (Batch_size=256)			Outputs				
	Num. of Epochs	Learning Rate	Split_size	Best loss		Best Accuracy(%)		Test Accuracy(%)
				Train	Validation	Train	Validation	
Parameter tuning Phase	30	0.001	0.5	0.0001	0.0033	97.06	80.64	91
	20	0.01	0.5	0	0.0028	97.23	82.43	91.6
	30	0.001	0.2	0.0003	0.0005	97.77	74.89	96.2
	10	0.01	0.2	0.0002	0.0005	98.21	75.81	96.9

3- Transfer Learning (Freezing weights)

	Varried inputs (Batch_size=256)			Outputs				
Selected Layers	Num. of Epochs	Learning Rate	Split_size	Best loss		Best Accuracy(%)		Test Accuracy(%)
				Train	Validation	Train	Validation	
Freeze all	20	0.01	0.2	0.0003	0.0004	98.47	75.45	96.4
Freeze all Conv	20	0.01	0.2	0.0012	0.0005	94.66	75.28	94.5
Freeze all FC	20	0.01	0.2	0.0002	0.0004	98.88	75.8	96.8

4- TL + Data Augmentation

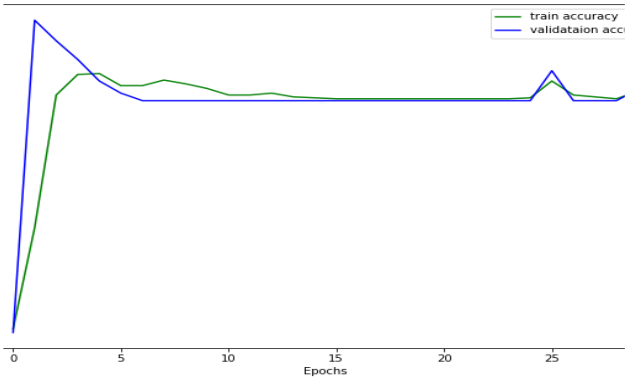
	Varried inputs (Batch_size=256)			Outputs				
DA Transform Method	Num. of Epochs	Learning Rate	Split_size	Best loss		Best Accuracy(%)		Test Accuracy(%)
				Train	Validation	Train	Validation	
ColorJitter	30	0.001	0.5	0.0001	0.0031	96.82	80.33	90.8
ColorJitter	20	0.01	0.5	0	0.0029	97.03	81.43	91
ColorJitter	15	0.01	0.2	0.0001	0.0005	99.05	76.75	97.1
ColorJitter	15	0.01	0.3	0	0.0001	99.2	75.75	94.8
RandomHorizontalFlip	15	0.01	0.2	0.0001	0.0006	99.2	75.89	97
RandomRotation	15	0.01	0.2	0.0001	0.0006	99.2	74.94	96.6
Combination of all	20	0.01	0.2	0.0001	0.0005	99.23	76.1	96.6
ColorJitter+Rand.H.Flip	20	0.01	0.2	0	0.0006	99.27	76.4	97.1
ColorJitter+Rand.H.Flip	15	0.001	0.2	0.0002	0.0005	97.69	75.8	96.9
ColorJitter+Rand.H.Flip	75	0.0001	0.2	0.0004	0.0005	96.28	74.68	96.5

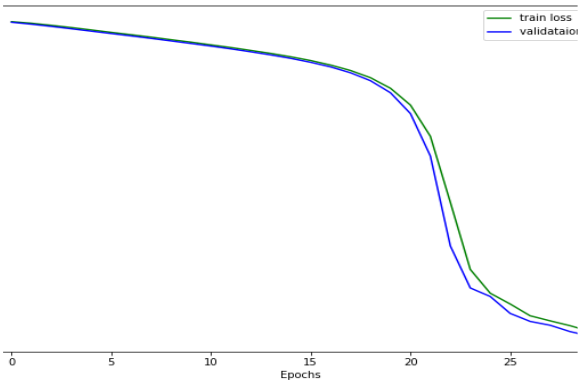
5- Beyond AlexNet (training with ResNet34)

	Varried inputs (Batch_size=16)			Outputs				
	Num. of Epochs	Learning Rate	Split_size	Best loss		Best Accuracy(%)		Test Accuracy(%)
				Train	Validation	Train	Validation	
Parameter tuning Phase	6	0.001	0.2	0.001	0.0048	99.89	90.23	97.9
	8	0.001	0.2	0.0006	0.0047	99.85	90.49	98.1

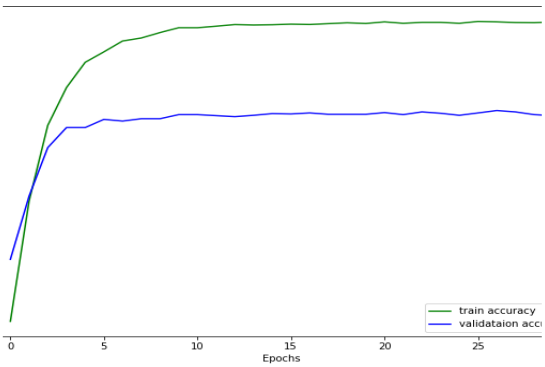
Table - Results Table

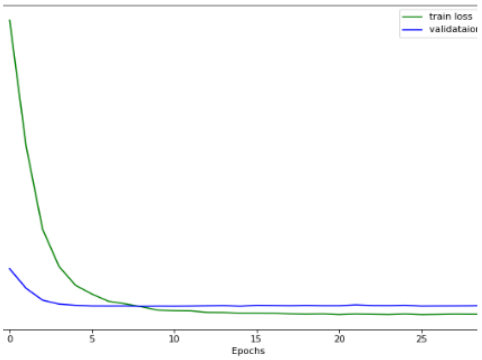
Trainingn from scratch (lr=1e-3, split_size=0.5,Num_epoch=30)



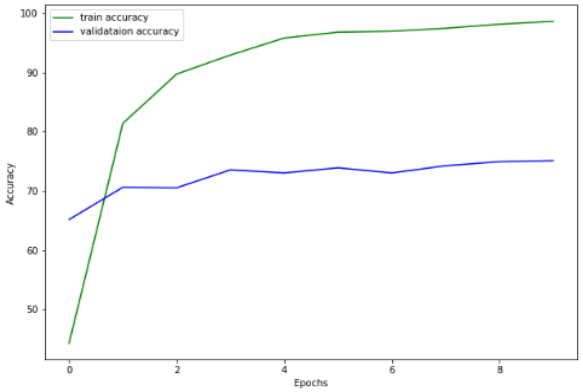


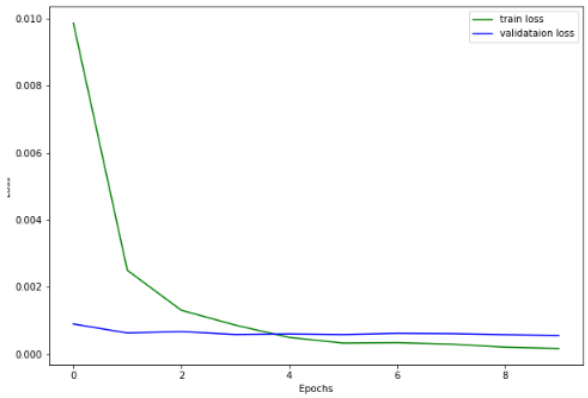
Transfer Learning(only pretrained)(lr=1e-3,split_size=0.2,Num_epochs=30)



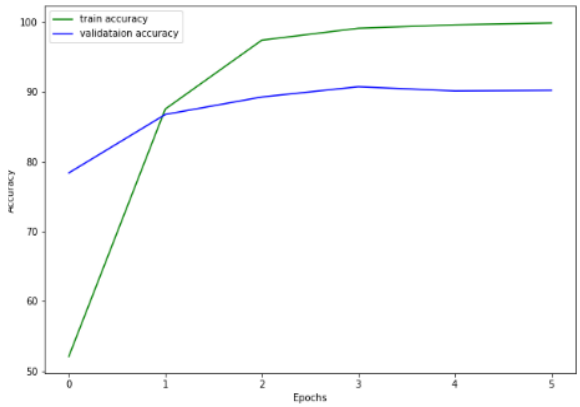


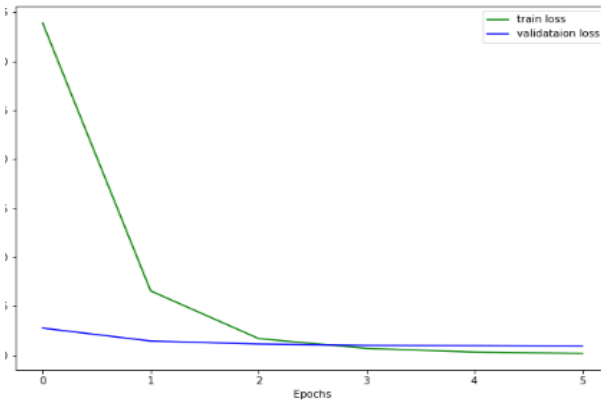
Transfer Learning(Freeze+Data Aug.)(lr=1e-2,split_size=0.2,Num_epochs=20)





Resnet34(lr=1e-2,split_size=0.2,Num_epochs=6)





6-References:

1- http://www.vision.caltech.edu/Image_Datasets/Caltech101/

2- https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce

3- https://neurohive.io/en/popular-networks/alexnet-imagenet-classification-with-deep-convolutional-neural-networks/

4- https://debuggercafe.com/getting-95-accuracy-on-the-caltech101-dataset-using-deep-learning/

5- https://discuss.pytorch.org/t/how-the-pytorch-freeze-network-in-some-layers-only-the-rest-of-the-training/7088/9

6- https://kratzert.github.io/2017/02/24/finetuning-alexnet-with-tensorflow.html

7- https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel

8- He, Kaiming, et al. 2016."Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*.

9- A. Krizhevsky, I. Sutskever, GE. Hinton. 2012. “ImageNet Classification with Deep Convolutional Neural Networks”. Advances in neural information processing systems. 25. 10.1145/3065386.

10- Tan, Chuanqi, et al. 2018. "A survey on deep transfer learning." *International conference on artificial neural networks*. Springer, Cham.