WARSAW UNIVERSITY OF TECHNOLOGY
FACULTY OF MATHEMATICS AND INFORMATION SCIENCE

PROJECT 1: REPORT

Image Classification with Convolutional Neural Networks

Dora Doljanin, Filip Pavičić

Warsaw, April 2022.

Table of contents

Introduction 1

1. Research problem 2

1.1. Description of the problem 2

1.2. Requirements 2

2. Theoretical introduction 3

2.1. Deep learning 3

2.2. Convolutional neural networks 4

3. Instruction of the application 5

3.1. Resources 5

3.1.1. Google Collab 5

3.1.2. Kaggle 5

3.2. Technologies 5

3.2.1. PyTorch 5

3.2.2. TensorFlow 6

3.3. Code 6

3.3.1. Preparing the dataset 6

3.3.2. Defining the models 7

3.3.3. Data augmentation 9

3.3.4. Training 11

3.3.5. Defining the hyper-parameters 13

4. Description of the conducted experiments 14

4.1. Choosing the hyper-parameters 14

4.2. Augmentation techniques 15

5. Result analysis 17

5.1. Results 17

5.1.1. Model created from the outset 17
5.1.2. Pre-trained MobileNetV3 Large model 18
5.2. Comparison 19
Conclusion 22
Literature 23

Introduction

Neural networks allow computer programs to recognize patterns and solve common problems in artificial intelligence, machine learning, and deep learning [1]. In the last few decades, there has been significant growth in the research of neural networks, including the development of fundamental principles and new algorithms [21]. In addition to that, there has also been an increasing amount of effort put into their real-world applications. An important area of application of neural networks is in robotics, where the major task involves making movements dependent on sensor data [7]. Other applications include the steering and path-planning of autonomous robot vehicles. Neural networks are also used for visual information processing, such as recognition (the classification of the input data in one of several possible classes), perceiving geometric information about the environment (which is important for autonomous systems), and the compression of the image for storage and transmission [7]. There are many types of neural networks, including convolutional neural networks (CNN), which are primarily used for classification of images, clustering of images and object recognition [22].

Apart from neural networks, recently there has been immense hype around topics of artificial intelligence, machine learning and deep learning. Deep learning algorithms are being used for a variety of purposes, such as image classification, speech recognition, handwriting transcription, machine translation, text-to-speech conversion, autonomous driving, improving ad targeting and search results on the web, and digital assistants such as Google Now and Amazon Alexa [5].

The aim of this project is to solve the problem of image classification using deep learning and convolutional neural networks. The goal is to test and compare the performances of different network architectures by conducting various experiments. In the following chapter we will firstly describe the research problem. Then, in the second chapter, we will provide a theoretical introduction to deep learning, convolutional neural networks, and image classification. In the third chapter, we will present the application developed to solve the proposed problem and provide an overview of the technologies and tools used in product implementation. In the last two chapters, we will explain the conducted experiments and analyse the results.

1. Research problem

1.1. Description of the problem

We aim to solve the problem of image classification. That includes identifying the subject of each image in the dataset by determining which class it belongs to. We use the CIFAR-10 dataset, which consists of 60,000 labelled images, to classify images in one of 10 object classes [25]. The 10 different classes include airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. We want to solve the proposed problem using deep learning and convolutional neural networks that learn by example. The CIFAR-10 dataset can be used to teach a neural network how to recognize objects. We further want to test and compare the performances of different neural network architectures by conducting several experiments.

1.2. Requirements

We want to use data augmentation as a regularization technique to reduce the risk of overfitting when training a model. We use data augmentation techniques to increase the amount of data by adding slightly modified copies of already existing data or newly created synthetic data from existing data [6].

The requirements are the following:

- At least 2 hyper-parameters should be related to the training process and 2 related to the regularization.
- We want to investigate the influence of at least:
    - 3 data augmentation techniques (for standard operations, such as rotations, zooming, cropping etc.)
    - 1 data augmentation technique (for more advanced data techniques, such as mix-up, cut-mix, cut-out etc.)

2. Theoretical introduction

2.1. Deep learning

The field of artificial intelligence puts in the effort to automate intellectual tasks usually performed by humans [5]. It includes machine learning and some approaches that don't involve learning at all. For instance, early chess programs involved large handcraft sets of explicit rules. This approach is known as symbolic AI and it dominated from the 1950s to the late 1980s [5]. Although symbolic AI could solve well-defined, logical problems, it was unable to solve more complex problems, such as image classification, speech recognition, and language translation [5]. Problems like these required a different approach – machine learning. In symbolic AI, humans input rules and data to be processed according to these rules, and outcome answers. With machine learning, humans input data as well as the answers expected from the data, and the outcome of the rules. This way, the rules are trained rather than explicitly programmed, and can then be applied to new data to produce original answers.

Deep learning is a subfield of machine learning. It experienced a rapid rising in popularity in early 2010s [5]. It learns representations from data by learning successive layers of increasingly meaningful data representations [5]. Information from the data goes through successive filters which perform simple data transformations. The data transformation implemented by a layer is parameterized by its weights (parameters) and is learned by exposure to examples. Learning means finding a set of values for the weights of all layers in a network, such that the network will correctly map example inputs to their associated targets. To find the right parameters, we need to be able to measure how far this output is from what you expected. This is the job of the loss function of the network, also called the objective function. The loss function takes the predictions of the network and the true target (what you wanted the network to output) and computes a distance score, capturing how well the network has done on this specific example. The loss function provides feedback on how to adjust the value of the weights a little, in a direction that will lower the loss score for the current example. This adjustment is the job of the optimizer, which implements what's called the Backpropagation algorithm: the central algorithm in deep

learning. It provides a way to train chains of parametric operations using gradient-descent optimization. With every example the network processes, the weights are adjusted a little in the correct direction, and the loss score decreases. This way, with each layer information is increasingly purified and more useful concerning the given task [5]. A depth of the model is equal to the number of layers in the model. In deep learning, the layered representations are most often learned using neural networks.

2.2. Convolutional neural networks

The first wave of interest in neural networks arose after the introduction of simplified neurons by McCulloch and Pitts in 1943 [23]. They were presented as models of biological neurons with as ability to perform computational tasks [7]. In 1989, the first successful practical application of neural networks appeared from Bell Labs as a solution to the problem of classifying handwritten digits [5]. The resulting network was used by the United States Postal Service in the 1990s to automate the reading of ZIP codes on mail envelopes [24].

With the rise of the Artificial Neural Network (ANN), the field of machine learning has experienced significant development. One of the most popular forms of ANN architecture is the Convolutional Neural Network (CNN). CNNs are primarily used to solve difficult tasks such as computer vision, image recognition, and pattern recognition [2]. Convolutional neural networks use principles from linear algebra, particularly matrix multiplication, to identify patterns within an image [1].

3. Instruction of the application

We implemented the application in the Python programming language in the form of a Jupyter notebook. We used the PyTorch framework and the TensorFlow platform. In addition to that, we utilized the resources provided by Google Collab and Kaggle. The following is an overview of the technologies and resources used in product implementation.

3.1. Resources

We used the GPUs provided by Google Collab and Kaggle hosted Jupyter Notebooks because they are free, yet very powerful open-source services.

3.1.1. Google Collab

Collab is a hosted Jupyter notebook service that requires no setup to use, while providing access free of charge to computing resources including GPUs [8].

3.1.2. Kaggle

Kaggle is an online community of data scientists and machine learning practitioners. It offers a no-setup, customizable, Jupyter Notebooks environment. It enables access to free GPUs and a huge repository of community published data and code [9].

3.2. Technologies

3.2.1. PyTorch

PyTorch is a free open-source machine learning framework based on the Torch library, used for applications such as computer vision and natural language processing [6]. It is widely used due to its useful features and large online community [10].

3.2.2. TensorFlow

TensorFlow is an open-source platform for machine learning. It provides an extensive set of tools, libraries and community resources that enables its users to easily build and deploy machine learning applications [26].

3.3. Code

3.3.1. Preparing the dataset

We downloaded the dataset from Kaggle into the working environment's file system and unzipped the test and train dataset. We split the original training dataset on training and validation datasets.

```
[ ]  # google colab
     ! mkdir ~/.kaggle
     ! cp kaggle.json ~/.kaggle/
     ! chmod 600 ~/.kaggle/kaggle.json
     ! kaggle competitions download cifar-10
     !unzip cifar-10.zip

[ ]  #kaggle notebook
     !pip install torchsummary
     !pip install --upgrade torch torchvision

[ ]  !pip install py7zr

     !python -m py7zr x ./train.7z # colab
     #!python -m py7zr x ../input/cifar-10/train.7z #kaggle notebook

[ ]  !python -m py7zr x ./test.7z
```

1. Downloading and unzipping the dataset

For loading the data, we created our class *CustomImageDataset*. We modelled our code on the example from the original PyTorch documentation [11]. Besides the code in the template, we added two methods. The first one returns a string label vector for an index of an instance, and the other method acts as a label encoder which turns the index of the class into a string label.

```
class CustomImageDataset(Dataset):
    def __init__(self,  img_dir,annotations_file = None,transform = None):
      self.img_labels = None
      self.le = preprocessing.LabelEncoder()
      if  annotations_file is not None:
        self.img_labels = pd.read_csv(annotations_file)
        self.img_labels['index_label'] = self.le.fit_transform(self.img_labels.iloc[:,1])
      self.img_dir = img_dir
      self.transform = transform

    def __len__(self):
      if self.img_labels is None:
        return len(os.listdir(self.img_dir))
      return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, str(idx + 1) + '.png')
        image = read_image(img_path)
        label = F.one_hot(torch.tensor(1),10).float()
        if  self.img_labels is not None:
          label = self.img_labels.iloc[idx, 2]
          label = torch.tensor(int(label))
          label = F.one_hot(label,10).float()

        # apply transformations
        if self.transform:
          for t in self.transform:
            image = t(image)
        return image, label

    def getLabel(self,idx):
      if  self.img_labels is not None:
        return self.img_labels.iloc[idx,1]
      return None

    def encodeLabel(self, idx):
      if self.img_labels is None:
        return None
      return self.le.inverse_transform(idx)
```

2. *CustomImageDataset* class for loading the data

After that, we split the data into training and validation sets, where the validation set contains 20% of the original labelled data. Furthermore, for each dataset we created an instance of the *DataLoader* class [12] which will provide batches for further training.

```
[ ]  train_datasets, val_datasets = train_test_split(cifar10_dataset, test_size=0.2)
```

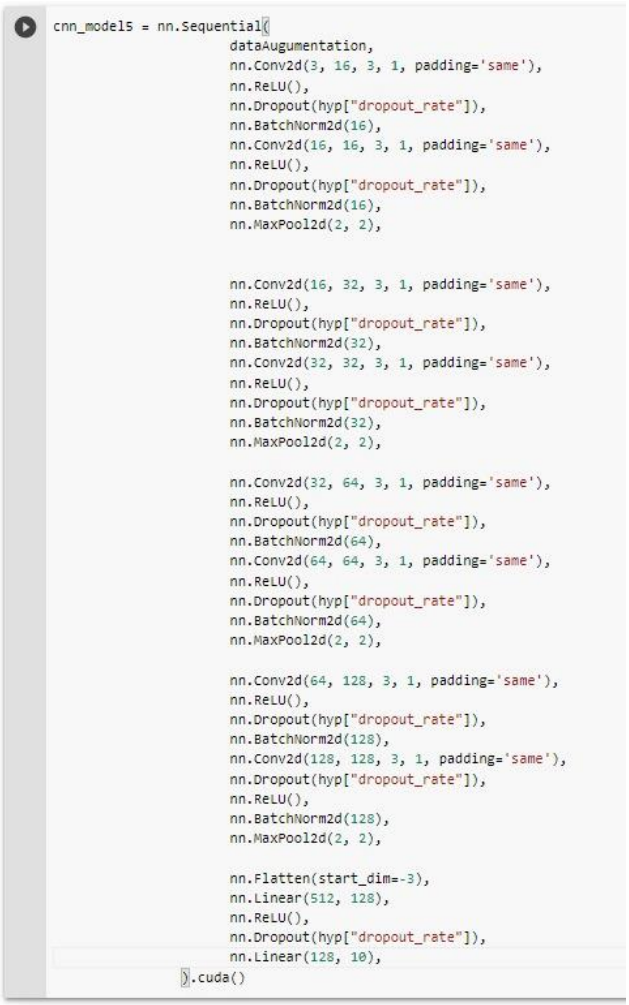
3. Splitting the dataset

3.3.2. Defining the models

We used two models: the first model we created from the outset, while the second model is pre-trained, and we adjusted it to our problem.

Model created from the outset

For this model, we used only basic layers, such as augmentation layer, convolutional layer, dropout layer, activation layer, max-pooling layer, and linear layer. The purpose of this model is to discover the possibilities of a simple model whose elements were described in the lectures.



```
cnn_model5 = nn.Sequential(
                        dataAugumentation,
                        nn.Conv2d(3, 16, 3, 1, padding='same'),
                        nn.ReLU(),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.BatchNorm2d(16),
                        nn.Conv2d(16, 16, 3, 1, padding='same'),
                        nn.ReLU(),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.BatchNorm2d(16),
                        nn.MaxPool2d(2, 2),


                        nn.Conv2d(16, 32, 3, 1, padding='same'),
                        nn.ReLU(),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.BatchNorm2d(32),
                        nn.Conv2d(32, 32, 3, 1, padding='same'),
                        nn.ReLU(),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.BatchNorm2d(32),
                        nn.MaxPool2d(2, 2),

                        nn.Conv2d(32, 64, 3, 1, padding='same'),
                        nn.ReLU(),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.BatchNorm2d(64),
                        nn.Conv2d(64, 64, 3, 1, padding='same'),
                        nn.ReLU(),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.BatchNorm2d(64),
                        nn.MaxPool2d(2, 2),

                        nn.Conv2d(64, 128, 3, 1, padding='same'),
                        nn.ReLU(),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.BatchNorm2d(128),
                        nn.Conv2d(128, 128, 3, 1, padding='same'),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.ReLU(),
                        nn.BatchNorm2d(128),
                        nn.MaxPool2d(2, 2),

                        nn.Flatten(start_dim=-3),
                        nn.Linear(512, 128),
                        nn.ReLU(),
                        nn.Dropout(hyp["dropout_rate"]),
                        nn.Linear(128, 10),
                    ).cuda()
```

4. The structure of the first model

We managed to get high quality results for such a simple model, which will be described in more detail in the fifth chapter.

Pre-trained MobileNetV3 Large model

Pre-trained models are often used for obtaining better results, since they have already been highly optimized and trained on similar datasets. We chose the MobileNetV3 Large model because it has a relatively small number of parameters (5.48 million) compared to other similar models. Furthermore, small models with high accuracy are often highly desirable because they are widely used in mobile applications for smartphones [13]. It is the third version of *MobileNet*, a model developed by Google that can run in embedded systems [27].

We changed the original MobileNetV3 Large model by adding two additional layers. The first layer we added is a data augmentation layer, which will be described in more detail in the fourth chapter. The second layer we added is a resizing layer which resizes the images

from 32x32 pixels (original size of images in the CIFAR-10 dataset) into 224x224 pixels (dimensions of the images which the MobileNetV3 Large model had previously been trained on). In addition to that, we changed the last linear classifier layer into a combination of linear layers which will predict the labels of the instances into one of the 10 classes of the CIFAR-10 dataset.

```
mobilenet_v3_large = models.mobilenet_v3_large(pretrained=True)

# reset final fully connected layer
num_ftrs = list(mobilenet_v3_large.classifier.children())[0].in_features

mobilenet_v3_large.classifier = nn.Sequential(
                nn.Linear(num_ftrs, 256),
                nn.ReLU(),
                nn.Dropout(0.3),
                nn.Linear(256, 10))

resize = TransformationLayer(image_transformation = [T.Resize((224,244))])

mobilenet_v3_large = nn.Sequential(
    dataAugumentation,
    resize,
    mobilenet_v3_large
)

mobilenet_v3_large = mobilenet_v3_large.cuda()
```

5. Defining the MobileNetV3 model

```
class TransformationLayer(nn.Module):
    def __init__(self, image_transformation = []):
        super().__init__()
        self.image_transformation = image_transformation

    def forward(self, x):
        for t in self.image_transformation:
            x = t(x)
        return x
```

6. Transformation layer

3.3.3. Data augmentation

We used two types of data augmentation techniques: the standard techniques that change only the original image in a certain way, and the more advanced techniques, that change not only the image but also its label. The following are the implementations of data augmentation methods in the code, while the reader can find more details about the used techniques in the fourth chapter.

Standard augmentation techniques

For image manipulations we used the methods from the torchvision.transforms module [14]. We wrapped the methods into the *RandomProbabilityWrapper* class which executes a certain transformation with a given probability.

```
class RandomProbabilityWrapper:

    def __init__(self,probability,transformation):
        self.transformation = transformation
        self.probability = probability


    def __call__(self, img):
        if random.random() < self.probability:
            return self.transformation(img)
        return img
```

7. *RandomTransformationWrapper* class

Finally, we created a data augmentation layer which executes all the given wrapped transformations only while training the model.

```
class DataAugmentationLayer(nn.Module):
    def __init__(self, image_transformation = []):
        super().__init__()
        self.image_transformation = image_transformation

    def forward(self, x):
        if self.training == True:
            for t in self.image_transformation:
                x = t(x)
        return x
```

8. The data augmentation layer

```
dataAugumentation = DataAugmentationLayer(image_transformation = [rotation,resize_cropper,affine])
```

9. Instantiating the data augmentation layer

Advanced augmentation techniques

We created our own implementation of the cut-mix data augmentation technique inspired by [15], in a form of a class. The method *__call__* executes the cut-mix method on given images and their corresponding labels. The method *transform* receives the batch as the argument and executes the cut-mix method on a group of instances from the batch according to a given probability.

```
class Cutmix:
  def __init__(self,probability, dataset):
    self.probability = probability
    self.dataset = dataset

  def transform(self, batch):
    indexes = np.where(np.random.rand(len(batch[0])) < self.probability)[0]
    print(indexes)
    for i in indexes:
        rand_index = random.randint(0,len(self.dataset) - 1 )
        second_image, second_label = self.dataset[rand_index]
        batch[0][i], batch[1][i] = self(batch[0][i], batch[1][i],second_image, second_label)
    return batch



  def __call__(self, image1, label1, image2, label2, beta_coff = 0.3,debug = False):
    ratio = np.random.beta(beta_coff, beta_coff, size=None)
    a = int(math.sqrt(ratio) * IMG_SIZE)
    x = random.randint(0, IMG_SIZE -1)
    y = random.randint(0, IMG_SIZE -1)
    x2 = x + a if x + a < IMG_SIZE else IMG_SIZE
    y2 = y + a if y + a < IMG_SIZE else IMG_SIZE
    width = x2 - x
    height = y2 - y
    ratio = width * height * 1.0 / (IMG_SIZE ** 2)

    crop2 = T.functional.crop(image2, y, x, height, width) #take only selected part
    image2 = T.functional.pad(crop2, [x, y, IMG_SIZE - x2, IMG_SIZE - y2]) #set it on image dimenzions black image

    crop1 = T.functional.crop(image1, y, x, height, width)
    img1 = T.functional.pad(crop1, [x, y, IMG_SIZE - x2, IMG_SIZE - y2])

    image1 = image1 - img1
    image = image1 + image2

    label = ratio * label2 + (1 - ratio) * label1
    if debug == True:
        print("a = " + str(a))
        print("x, y = " +str(x) + ", "+str(y))
        print("x2, y2 = " +str(x2) + ", "+str(y2))
        print("width, height = " +str(width) + ", "+str(height))

    return image, label
```

10. The *Cutmix* class

3.3.4. Training

We created a training method inspired by [16]. Inside the method, we defined the Adam optimizer, which executes the stochastic gradient descent algorithm. In addition to that, we defined a way of calculating the loss using the cross-entropy value.

```
def train(model, train_dataloader, test_dataloader = None, num_epochs=1, lr=1e-3, keepBest = False, mix_trainsformation = None):
  optimizer = torch.optim.Adam(model.parameters(), lr=lr)
  array_loss_train = np.empty([0])
  array_loss_val = np.empty([0])
  #with wandb.init(project="UZOP_LAB"):
  #model.cuda(0)
  best_model_wts = None
  best_lost = None

  for epoch in range(num_epochs):
    correct_train = 0
    model.train()
    with tqdm(enumerate(train_dataloader), total=len(train_dataloader), desc=f'Training (epoch={epoch}/{num_epochs})') as epoch_progress:
      for batch_idx, train_batch in epoch_progress:
        if mix_trainsformation is not None:
            for t in mix_trainsformation:
                train_batch = t.transform(train_batch)

        x, y = train_batch
        x = x.float().cuda()
        y = y.cuda()
        logits = model(x)
        value, predicted = torch.max(logits, 1)
        loss = F.cross_entropy(logits, y)
        epoch_correct = (predicted==torch.argmax(y, dim=1)).sum().item()
        correct_train += epoch_correct
        epoch_progress.set_postfix({'loss': 1 - epoch_correct / hyp["batch_size"]})

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    train_loss = 1 - (correct_train / len(train_dataloader.dataset))
    array_loss_train = np.append(array_loss_train,train_loss)
    print("Train loss " + str(train_loss))

    if test_dataloader is not None:
      with torch.no_grad():
        model.eval()
        correct_val = 0
        for test_batch in tqdm(test_dataloader, desc="Testing"):
          x, y = test_batch
          x = x.float().cuda()
          y = y.cuda()

          logits = model(x)
          value, predicted = torch.max(logits, 1)
          loss = F.cross_entropy(logits, y)
          epoch_correct = (predicted==torch.argmax(y, dim=1)).sum().item()
          correct_val += epoch_correct
```

11. The training method - part 1

```
        val_loss = 1 - (correct_val / len(test_dataloader.dataset))
        array_loss_val = np.append(array_loss_val,val_loss)
        print("Val loss " + str(val_loss))

      if best_lost is None or best_lost > val_loss:
            best_lost = val_loss
            best_model_wts = copy.deepcopy(model.state_dict())

  if test_dataloader is not None:
    if keepBest:
        model.load_state_dict(best_model_wts)

    model.eval()

    with torch.no_grad():
      test_loss = []
      correct = 0
      count = 0


      for test_batch in tqdm(test_dataloader, desc="Testing"):
        x, y = test_batch
        x = x.float().cuda()
        y = y.cuda()
        logits = model(x)
        loss = F.cross_entropy(logits, y)
        test_loss.append(loss)
        correct += (logits.argmax(dim=-1) == torch.argmax(y, dim=1)).float().sum()
        count += len(y)

      test_loss = torch.mean(torch.tensor(test_loss))
      test_acc = correct / count

      print()
      print(f"--- TEST ---")
      print("loss: ", test_loss.item())
      print("accuracy: ", test_acc.item())

  model.train()
  return array_loss_train, array_loss_val
```

12. The training method - part 2

3.3.5. Defining the hyper-parameters

As defined in the requirements, at minimum of two hyper-parameters should be related to the training process, and a minimum of two related to the regularization. The hyper-parameters we used are the following:

Regularization hyper-parameters

- ***Dropout rate*** – the percentage of nodes that will be turned off in certain layers, which helps the networks to widen the variety of nodes that will be activated in the training of each example
- ***Data augmentation rate*** – the probability of each standard data augmentation method being applied on an example
- ***Data augmentation rate mix*** **-** the probability of each advanced data augmentation method being applied on an example

Training hyper-parameters

- ***Batch size*** – the number of examples in one training group
- ***Learning rate*** – the number that multiplies the gradient, which determines how much the weights will be adjusted in the direction opposite of the gradient
- ***Shuffle*** – a Boolean value which determines whether we will mix the examples between the batches in each epoch
- ***Number of epochs*** – this parameter will be determined according to the results on the validation set (when the validation loss reaches its minimum value)

For each model we created a separate dictionary containing the corresponding hyper-parameters.

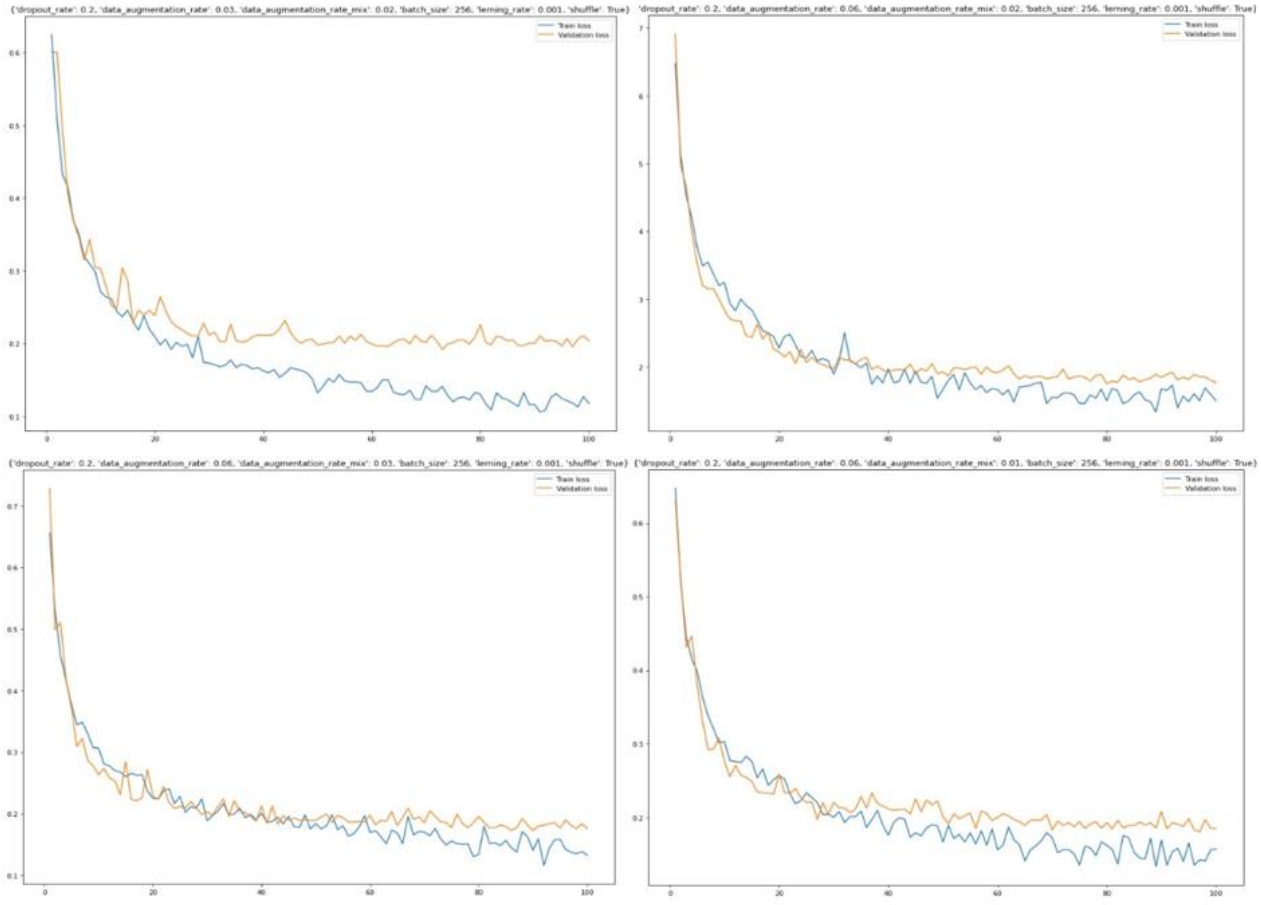
```
[ ]  #variables and hyperparameters

     #variable
     IMG_SIZE = 32

     hyp = {
         #regularization hyperparameters
         "dropout_rate" : 0.2,
         "data_augmentation_rate" : 0.06, # probabilty of any transformation = 1 - (1 - data_augmentation_rate)**(Number of transformation)
         "data_augmentation_rate_mix" : 0.06,



         #leraning hyperparameters
         "batch_size" : 64,
         "lerning_rate" : 1e-3,
         "shuffle" : True
     }
```

13. Hyper-parameters dictionary

4. Description of the conducted experiments

4.1. Choosing the hyper-parameters

To investigate the influence of the hyper-parameters on obtained results, we trained the model on the training dataset for certain combinations of hyper-parameters and validated them on the validation dataset. The optimal hyper-parameters are those that minimise the value of the loss function on the validation set. For that reason, we conducted several experiments by changing the hyper-parameters, training the model, and observing the effect on the validation loss. In addition to that, we tried to narrow the gap between the training loss and validation loss, because a high value of the gap can indicate a possibility of over-fitting.



14. Train loss and validation loss variations depending on the values of hyper-parameters

The hyper-parameters:

- Upper left picture:

{'dropout_rate': 0.2, 'data_augmentation_rate': 0.03, 'data_augmentation_rate_mix': 0.02, 'batch_size': 256, 'learning_rate': 0.001, 'shuffle': True}

- Upper right picture:

{'dropout_rate': 0.2, 'data_augmentation_rate': 0.06, 'data_augmentation_rate_mix': 0.02, 'batch_size': 256, 'learning_rate': 0.001, 'shuffle': True}

- Lower left picture:

{'dropout_rate': 0.2, 'data_augmentation_rate': 0.06, 'data_augmentation_rate_mix': 0.03, 'batch_size': 256, 'learning_rate': 0.001, 'shuffle': True}

- Lower right picture:

{'dropout_rate': 0.2, 'data_augmentation_rate': 0.06, 'data_augmentation_rate_mix': 0.01, 'batch_size': 256, 'learning_rate': 0.001, 'shuffle': True}

From the given graphs, we can see that the gap between the train loss and the validation loss is the smallest on the lower left picture. We decided to choose the hyper-parameters from that model and acquire the optimal number of epochs depending on the value of validation loss during the training.

After choosing the hyper-parameters, we reset the model on the initial weights and trained it on the whole dataset (train + validation sets).

4.2. Augmentation techniques

Specialized image and video classification tasks often have insufficient data. This is especially true in the medical industry, where patients' data is extremely sensitive and access to it is limited due to privacy concerns. The problem with small datasets is that models trained on them do not generalize well on the validation and test sets [17]. Therefore, these models are at high risk of overfitting. To reduce overfitting, we can use several methods, including regularization terms on the norm of the weights, transfer learning, batch normalization, or dropout. Data augmentation is another way we can reduce the overfitting of models. The idea is to increase the amount of training data by performing various transformations of the existing data in the training set. Rather than starting with a large set of unlabelled data, we can instead take a small set of labelled data and augment it in a way that increases the performance of models trained on it [17].

```
[ ]  rotation = RandomProbabilityWrapper(hyp["data_augmentation_rate"], lambda x : T.functional.rotate(x,int(np.random.choice([90,180,270]))))
     resize_cropper = RandomProbabilityWrapper(hyp["data_augmentation_rate"], lambda x :T.RandomResizedCrop(size=(32, 32))(x))
     affine = RandomProbabilityWrapper(hyp["data_augmentation_rate"], lambda x :T.RandomAffine(degrees=0, translate=(0.1, 0.4), scale=(1,1))(x))

     cutmix = Cutmix(hyp["data_augmentation_rate_mix"],cifar10_dataset)
```

15. Data augmentation techniques in the code

We used three standard data augmentation techniques and one advanced data augmentation technique. The techniques are the following:

Standard augmentation techniques

- ***Rotation*** [18]– rotates the original image by given angle
- ***Random Resized Crop*** [19] – crops a random portion of image and resizes it to a given size
- ***Translate*** [20] – translates the original image using horizontal and vertical translations

Advanced augmentation technique

- ***CutMix*** – Instead of removing pixels and filling them with black or grey pixels or Gaussian noise, the method replaces the removed regions with a patch from another image, while the ground truth labels are mixed proportionally to the number of pixels of combined images. [15]

16. An example of the *CutMix* technique on two images

5. Result analysis

5.1. Results

5.1.1. Model created from the outset

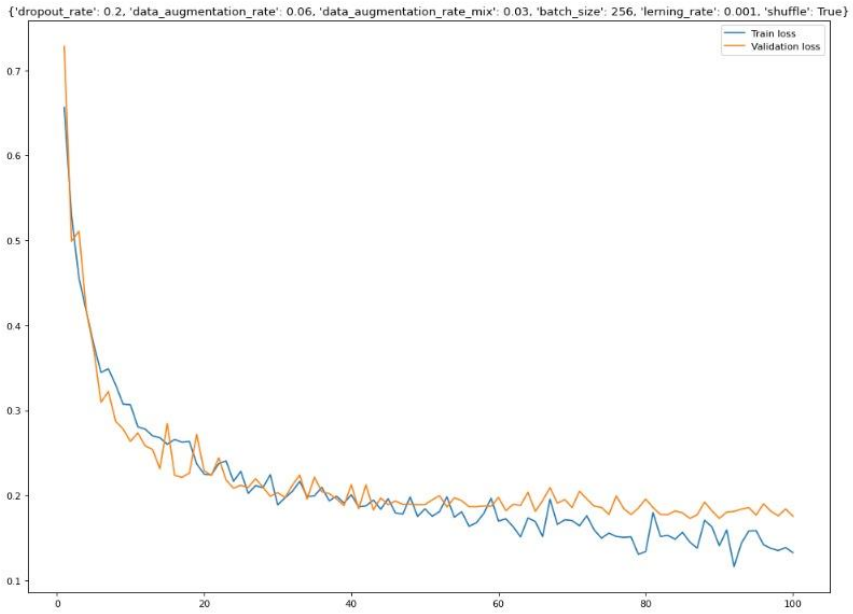
In respect to the conducted experiments, we have chosen the following hyper-parameters:

{'dropout_rate': 0.2, 'data_augmentation_rate': 0.06, 'data_augmentation_rate_mix': 0.03, 'batch_size': 256, 'learning_rate': 0.001, 'shuffle': True}.

After 100 epochs, we managed to achieve an accuracy of 82.26%. Additionally, we tried training with 200 and 300 epochs. However, the accuracy did not change significantly so we decided to stop at 100 epochs.

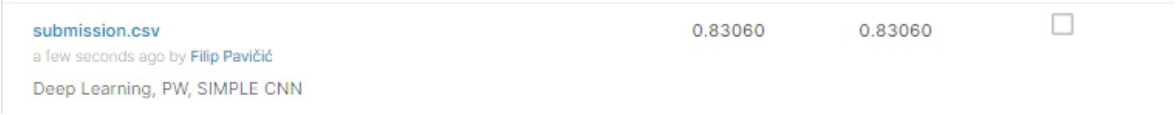
Training (epoch=0/100): 100%|██████████| 157/157 [00:02<00:00, 63.57it/s, loss=0.898]
Train loss 0.683225
Testing: 100%|██████████| 40/40 [00:00<00:00, 178.92it/s]
Val loss 0.6874
Training (epoch=1/100): 100%|██████████| 157/157 [00:02<00:00, 68.02it/s, loss=0.898]
Train loss 0.5509999999999999
Testing: 100%|██████████| 40/40 [00:00<00:00, 182.53it/s]
Val loss 0.5217
…
Training (epoch=99/100): 100%|██████████| 157/157 [00:02<00:00, 67.33it/s, loss=0.77]
Train loss 0.15144999999999997
Testing: 100%|██████████| 40/40 [00:00<00:00, 179.57it/s]
Val loss 0.1774
Testing: 100%|██████████| 40/40 [00:00<00:00, 244.25it/s]

--- TEST ---
loss: 0.6602978706359863
accuracy: 0.8226000070571899



17. Train and validation loss of the first model

After having trained the model on the whole dataset (train + validation), we managed to achieve an accuracy of 83.06% on the test dataset.



18. Accuracy of the first model on the test dataset

5.1.2. Pre-trained MobileNetV3 Large model

For the second model, we have chosen the following hyper-parameters:

{'dropout_rate': 0.2, 'data_augmentation_rate': 0.09, 'data_augmentation_rate_mix': 0.05, 'batch_size': 64, 'learning_rate': 0.001, 'shuffle': True}.

After 15 epochs, we managed to achieve an accuracy of 92.79%. However, we noticed that after 14 epochs the validation loss started rising, so we decided to choose 14 as the number of epochs for training on the whole dataset later.

Training (epoch=0/15): 100%|| 625/625 [01:33<00:00, 6.66it/s, loss=0.141]
Train loss 0.219375
Testing: 100%|| 157/157 [00:05<00:00, 29.51it/s]
Val loss 0.15190000000000003
…
Training (epoch=13/15): 100%|| 625/625 [01:33<00:00, 6.69it/s, loss=0.0625]
Train loss 0.05852500000000005
Testing: 100%|| 157/157 [00:05<00:00, 29.69it/s]
Val loss 0.06569999999999998
Training (epoch=14/15): 100%|| 625/625 [01:33<00:00, 6.72it/s, loss=0.0156]

Train loss 0.05612499999999998

Testing: 100%|██████████| 157/157 [00:05<00:00, 29.66it/s]

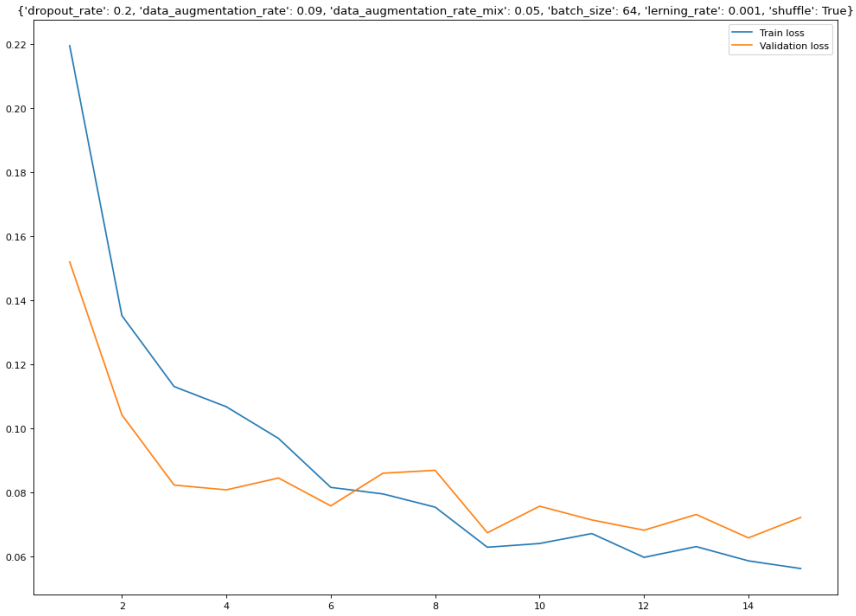
Val loss 0.07210000000000005

Testing: 100%|██████████| 157/157 [00:05<00:00, 31.03it/s]

--- TEST ---

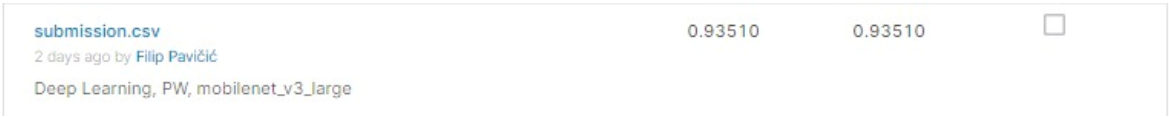
loss: 0.26329612731933594

accuracy: 0.927899956703186



19. Train and validation loss of the second model

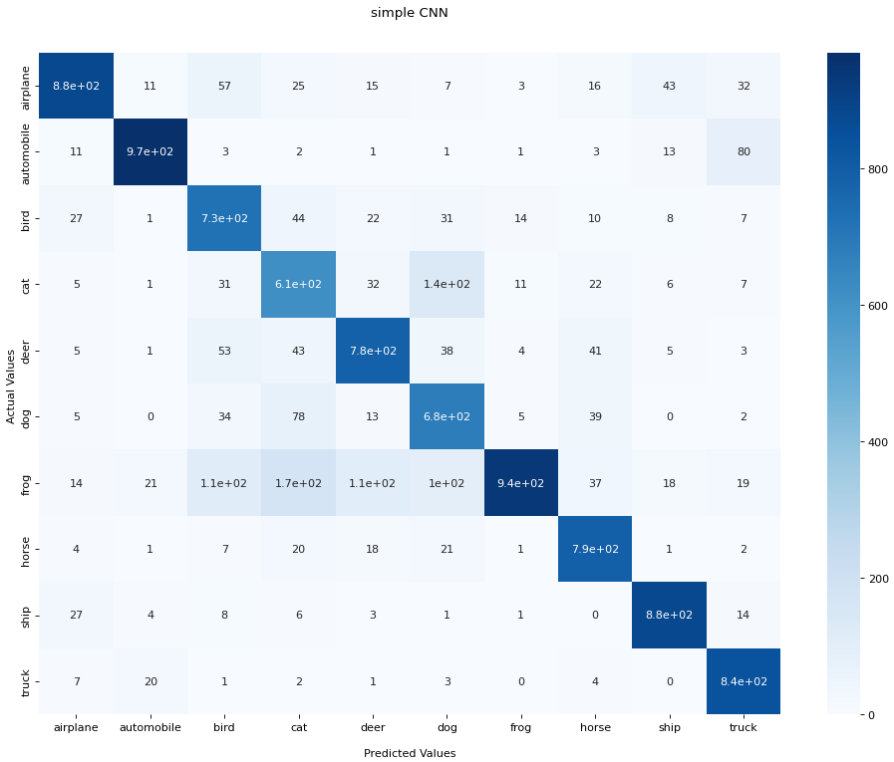
We trained this model on the whole dataset (train + validation) and obtained an accuracy of 93.51% on the test dataset.



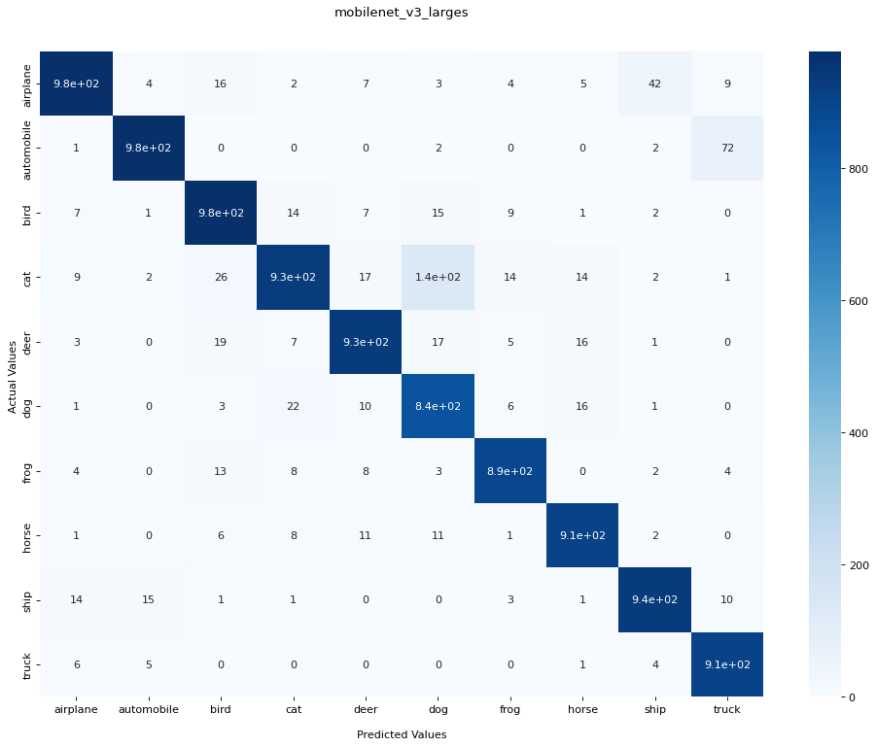
20. Accuracy of the second model on the test dataset

5.2. Comparison

From the results obtained, we may conclude that the pre-trained mode achieved better accuracy (93.51%) than the model made from the output (83.06%), as expected. In order to further observe the performances and behaviour of both models, we created corresponding confusion matrices of the models on the validation sets.



21. Confusion matrix of the first model on validation dataset



22. Confusion matrix of the second model on validation dataset

From the given matrices, we may draw useful conclusions about the models. We can see that the simple CNN has some problems with recognising frogs in images. Several frogs in images have been classified as some other class, mostly birds, cats, deer, or dogs. We may

notice that, even though the model classified some frogs incorrectly, it most frequently still classified it as some animal, rather than a vehicle. We assume that this behaviour is due to the model picking up on natural background of the images with animals. Furthermore, we may notice that both models tend to classify cats as dogs a certain number of times. This is probably the case because of their physical similarities (four legs, ears, tail etc).

Conclusion

According to the results of the conducted experiments, we may conclude that the existing pre-trained models are highly accurate in solving the problem of image classification. A vast majority of them are able to obtain an accuracy higher than 90%, including the smaller models aimed at mobile applications. In this project, we tested one of the small mobile models, the MobileNetV3 Large, and obtained an accuracy of 93.51%. For this reason, we may consider the problem of image classification solved. In addition to that, we have demonstrated that it is possible to solve the same problem using a very simple architecture with fundamental layers. Even rudimentary models can obtain relatively high accuracy, taking their simplicity into account.

We may conclude that we have successfully solved the problem of image classification, which helps us to classify the subject in an image. As a follow up to this project, the next step could be to solve the problem of image localization, which specifies the location of a single object in an image. Another extension could be to unravel the problem of object detection, which specifies the location of multiple objects in the image. Finally, another challenge to solve could be image segmentation, which creates a pixel-wise mask of each object in the images, enabling the identification of shapes of different objects in the image.

Literature

[1] *Neural Networks*, IBM Cloud Education, (17 August 2020). Link: https://www.ibm.com/cloud/learn/neural-networks

[2] O'Shea, Keiron, and Ryan Nash. *An introduction to convolutional neural networks.* arXiv preprint arXiv:1511.08458 (2015).

[3] D. Lu & Q. Weng (2007). *A survey of image classification methods and techniques for improving classification performance*, International Journal of Remote Sensing, 28:5, 823-870

[4] Lorica, Ben (3 August 2017). "*Why AI and machine learning researchers are beginning to embrace PyTorch*". O'Reilly Media. (Retrieved 11 December 2017).

[5] Chollet, Francois. *Deep learning with Python*. Simon and Schuster, (2021).

[6] Shorten, Connor; Khoshgoftaar, Taghi M. (2019). "*A survey on Image Data Augmentation for Deep Learning"*. Mathematics and Computers in Simulation. springer. 6: 60. doi:10.1186/s40537-019-0197-0

[7] Kröse, Ben, et al. "*An introduction to neural networks*." (1993).

[8] *Google Collab*, Link: https://colab.research.google.com/?utm_source=scs-index

[9] *Kaggle*, Link: https://www.kaggle.com/

[10] *PyTorch*, Link: https://pytorch.org/

[11] Sasank Chilamkurthy, *Writing custom datasets, dataloaders and transforms*, https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

[12] *torch.utils.data.DataLoader*, https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader

[13] Xu, Mengwei, et al. "*A first look at deep learning apps on smartphones*." The World Wide Web Conference. (2019).

[14] *Transforming and augmenting images*, Link: https://pytorch.org/vision/master/transforms.html

[15] Sayan Nath, *CutMix data augmentation for image classification*, Link: https://keras.io/examples/vision/cutmix/

[16] Mile Šikić, class materials from the course *Introduction to Data Science* at the Faculty of Electrical Engineering and Computing, University of Zagreb

[17] Perez, Luis, and Jason Wang. "*The effectiveness of data augmentation in image classification using deep learning."* arXiv preprint arXiv:1712.04621 (2017).

[18] *torchvision.transforms.functional.rotate*, Link: https://pytorch.org/vision/master/generated/torchvision.transforms.functional.rotate.html#torchvision.transforms.functional.rotate

[19] *torchvision.transforms.RandomResizedCrop*, Link: https://pytorch.org/vision/master/generated/torchvision.transforms.RandomResizedCrop.html#torchvision.transforms.RandomResizedCrop

[20] *torchvision.transforms.RandomAffine*, Link: https://pytorch.org/vision/master/generated/torchvision.transforms.RandomAffine.html#torchvision.transforms.RandomAffine

[21] Bishop, Chris M. "*Neural networks and their applications.*" Review of scientific instruments 65.6 (1994): 1803-1832.

[22] Pratik Shukla, Roberto Iriondo, *Main Types of Neural Networks and its Applications—Tutorial,* (July 13, 2020), Link: https://towardsai.net/p/machine-learning/main-types-of-neural-networks-and-its-applications-tutorial-734480d7ec8e

[23] McCulloch, Warren S., and Walter Pitts. "*A logical calculus of the ideas immanent in nervous activity.*" The bulletin of mathematical biophysics 5.4 (1943): 115-133.

[24] LeCun, Yann, et al. "*Backpropagation applied to handwritten zip code recognition.*" Neural computation 1.4 (1989): 541-551.

[25] The CIFAR-10 dataset, Link: http://www.cs.toronto.edu/~kriz/cifar.html

[26] *TensorFlow*, Link: https://www.tensorflow.org/

[27] Vandit Jain, *Everything you need to know about MobileNetV3*, (Nov 22, 2019), Published in *Towards Data Science*, Link: https://towardsdatascience.com/everything-you-need-to-know-about-mobilenetv3-and-its-comparison-with-previous-versions-a5d5e5a6eeaa