Deep Learning - Home Exam

Lorenzo Mandelli
lma130@uit.no
Candidate no:27
UiT The artic university of Norway

April 28, 2022

1 Problem 1

1.1 (1a)

The cross-entropy between $z \in \mathbb{R}^2$ and $y \in \mathbb{R}^2$ is defined as:

$$CE(z, y) = -(\mathbb{1}_{y=c_0} \log(z_0) + \mathbb{1}_{y=c_1} log(z_1))$$

where $z = (z_0, z_1)$ is the output of a previous layer, $y = (y_0, y_1)$ is the label, c_0 and $c_1 \in \mathbb{R}^2$ are the possible classes and $\mathbb{1}_{y=c_0}$ is a function defined as follows:

$$\mathbb{1}_{y=c_0} = \begin{cases} 1 & \text{if } y = c_0 \\ 0 & \text{otherwise} \end{cases}$$

We denote the classes $c_0 = (1, 0)$ and $c_1 = (0, 1)$. If we assume $\hat{y} = (1, 0)$ we can write the previous equation as:

$$CE(z, \hat{y}) = -(1 \cdot \log(z_0) + 0 \cdot log(z_1)) = -\log(z_0)$$

1.2 (1b)

If we consider the case in which z is not stochastic, it no longer represents the probability of an event and we encounter the following problems:

- The variable $z = (z_0, z_1)$ is no longer limited between 0 and 1. If $z_i < 0$ the logarithmic function is not defined, if $z_i > 0$ we will get a negative loss which generally doesn't make sense.
- The sum of the elements of z is no longer be $\sum_{z_i} = 1$. So if we want to minimize the loss the easiest solution will be to have all the components equal to $+\infty$, without even take into consideration the value of the label y.

1.3 (1c)

Let's now consider the following implementations [1], [2] and [3] in the Python programming language of multiclass cross entropy loss between $z, y \in \mathbb{R}^{N \times C}$, where N is the number of samples of the dataset and C the number of classes. In [1] we use the $Numpy$ library to calculate the function following its definition: first we do the logarithm of every element of z and then we multiply element-by-element with y, which defines the class label of each samples according to the one-hot encoding. With this element-by-element multiplication we obtain the same results of using the function $\mathbb{1}_{y=c_0}$. Finally we sum all the elements of the matrix $result \in \mathbb{R}^{N \times C}$ and we divide it by N to obtain the mean loss value. In [2] we do the same operations of [1] in a one-line version of the code.

```
def cross_entropy(z, y):
    N = y.shape[0]
    result = y * np.log(z)
    result = - np.sum(result) / N
    return result
```

Code 1: Classical implementation of cross entropy function for N samples.

```
def cross_entropy(z, y):
    return - np.sum(y * np.log(z)) / y.shape[0]
```

Code 2: One-line code implementation of cross entropy for N samples.

If we want to reduce the number of the operation we can implement [3] which requires just $2N$ operations, N logs and N sums. This version doesn't require the multiplication with the matrix y because the same result is obtained by the operation of indexing. This is possible considering $y \in \mathbb{R}^N$ without the one-hot encoding, we instead encode the class c_i in its number $i \in [0, N-1]$ for each sample. In this way we can use directly the label y for the operation of indexing.

```
def cross_entropy(z, y):
    N = y.shape[0]
    result = np.log(z[range(N), y.astype(int)])
    result = - np.sum(result)/N
    return result
```

Code 3: Implementation of cross entropy that requires 2N operations.

1.4 (1d)

1. $\mathbf{Argmin_z CE(z, y)}$:

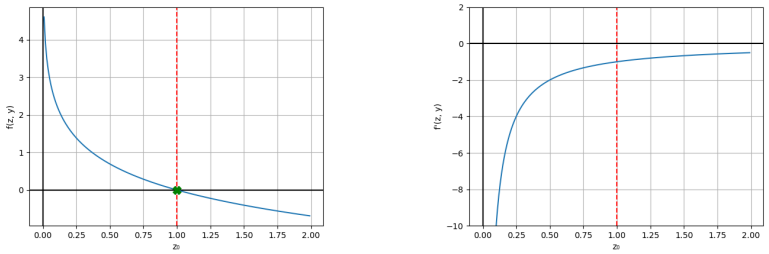
$$argmin_{z=(z_0,z_1)} f(z, y) =$$

$$argmin_{z=(z_0,z_1)} CE(z, y) =$$

$$argmin_{z=(z_0,z_1)} - (\mathbb{1}_{y=c_0} \log(z_0) + \mathbb{1}_{y=c_1} log(z_1)) =$$

Let's consider $y = (1, 0) = c_0$:

$$argmin_{z=(z_0,z_1)} - \log(z_0)$$



(a) **(b)**

Figure 1: Cross-entropy plot without regularization (a) and its derivative (b). The green dot indicates the minimum point of the function.

Let's consider now:

$$\nabla f(z,y) = [\frac{\partial f(z,y)}{\partial z_0}, \frac{\partial f(z,y)}{\partial z_1}] = [-\frac{1}{z_0}, 0]$$

with $z_0, z_1 \in [0,1]$.
The derivative $\frac{\partial f(z,y)}{\partial z_0} \leq 0$ always, so the function $f(z,y)$ is decreasing in the considered interval. So the minimum value is obtained for the extreme of the considered interval $z_0^* = 1$ and consequently for $z_1^* = 0$. The plot of the function and its derivative are shown in figure 1.

2. $\mathbf{Argmin_z CE(z,y) + ||z||_2^2}$:

$$argmin_{z=(z_0,z_1)} f(z,y) =$$

$$argmin_{z=(z_0,z_1)} CE(z,y) + ||z||_2^2 =$$

Let's consider $y = (1,0) = c_0$:

$$argmin_{z=(z_0,z_1)} - \log(z_0) + z_0^2 + z_1^2$$

Considering z stochastic we have $z_0 + z_1 = 1$ and we can write:

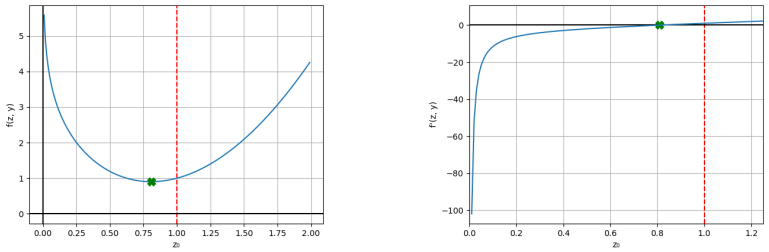
$$argmin_{z=(z_0,z_1)} - \log(z_0) + z_0^2 + (1-z_0)^2 =$$

$$argmin_{z=(z_0,z_1)} - \log(z_0) + 2z_0^2 - 2z_0 + 1$$

We derive and we obtain:

$$\frac{\partial f(z,y)}{\partial z_0} = \frac{4z_0^2 - 2z_0 - 1}{z_0}$$

Which has minimum in $z_0^* = \frac{1+\sqrt{5}}{4}$ and consequently for $z_1^* = 1 - z_0^*$. The plot of the function and its derivative are shown in figure 2.



(a) **(b)**

Figure 2: Cross-entropy plot with L2 regularization (a) and its derivative (b). The green dot indicates the minimum point of the function.

3. $\mathbf{Argmin_z CE(z, y) + ||z||_1}$:

$$argmin_{z=(z_0,z_1)} f(z, y) =$$
$$argmin_{z=(z_0,z_1)} CE(z, y) + ||z||_1 =$$
$$argmin_{z=(z_0,z_1)} CE(z, y) + |z_0| + |z_1| =$$

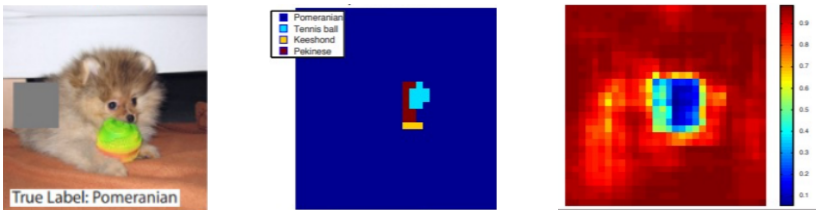
$z = (z_0, z_1)$ represents a probability so $\sum_i z_i = 1$:

$$argmin_{z=(z_0,z_1)} CE(z, y) + 1 =$$
$$argmin_{z=(z_0,z_1)} CE(z, y)$$

So the calculation of the minimum value is equal to point 1 and we obtain $z^* = [1, 0]$ for $y = (1, 0)$.

1.5 (1e)

From the previous results it can be seen that for the minimization of the binary cross entropy function without regularization the minimum value occurs when the probability of the correct class corresponding to y is maximum equal to 1 and the probability of the wrong class is equal to 0.
We obtain the same result in the case of regularization L1: since z is a stochastic variable, it is already regularized.
The regularization L2, on the other hand, prevents the components of z from assuming too large values by placing the weight of them squared and in fact in the considered case it brings the minimum value from $z_0^* = 1$ to $z_0^* = \frac{1+\sqrt{5}}{4} \approx 0.8$.



(a) **(b)** **(c)**

Figure 3: Heatmap visualization: (a) input image, (b) most probable class, (c) probability of the correct class.

2 Problem 2

2.1 (2a)

To qualitatively understand the behavior of a neural network it is possible to investigate the response of the model (or the response of a layer/neuron) for certain specific inputs or otherwise try to evaluate a given model globally.
In the first case, one of the most used tools are the heatmaps. In the case of convolutional networks given an input image, it is possible to generate a heatmap showing which parts of the image the network focuses to obtain the prediction as shown in figure 3.
To generate the heatmap we start by covering part of the image with an empty patch. At each iteration the patch will be shifted and the patched image is classified. In this way we expect that the classification will give correct result when the patch does not cover the main subject to be classified and will give wrong result otherwise. At each iteration we map the probability of predicting the correct class in the position of the patch on the image and in this way we obtain the heatmap. This method can also be applied to layers inside the network to evaluate the response.

One method to evaluate the model from a global point of view is the class model visualization. It consists of creating artificial images for each possible class that show for which inputs the model produces the maximum value as shown in figure 4. Considering class c, we want to find the image I that maximizes:

$$\arg\max_I S_c(I) - \lambda||I||_2^2$$

with I initial random noise image. At each iteration we consider the image produced at the previous step and through the forward pass we obtain the score for the class c. Then we do the backpropagation up to the input with the frozen network weights and we modify the input itself, obtaining a new image for the next iteration. In these passages, however, we must consider unnormalized class scores as we do not want the values of one class to depend on the others and therefore to be correlated.

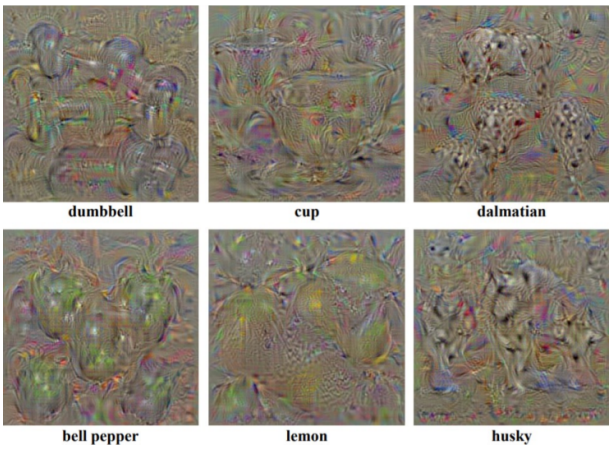


Figure 4: Class model visualization for 6 different class. Image taken from [1].

2.2 2b

2.2.1 VGG16

VGG16 is a Convolutional Neural Network (CNN) Architecture used for ImageNet, a large visual database of images used in computer vision and deep learning research.
To pre-process the dataset before the training the input images are cropped to the fixed-size of 224 x 224 and the mean RGB value computed on the training set is subtracted from each pixel.
The architecture is composed by 13 convolutional layers, 5 max pooling, and 3 fully connected: the first two have 4096 channels each, while the third contains 1000 channels, one for each class of the ImageNet dataset. At the end of the network following the FC layers there is a soft-max layer.
For the convolution operations are considered small filter of size 3x3 with stride and the spatial padding equal to 1 for keep the same size after the operations. Max-pooling is performed over a 2×2 pixel window, with stride 2.
The VGG16 architecture is shown in figure 5.

2.2.2 Implementation

In order to carry out tests on the VGG16 network, an implementation was made using the Pytorch library which provides a VGG16 class which gives the possibility to directly load the weights of the trained network on the ImageNet dataset.
The validation set provided for the assignment has been elaborated through the *process_data.py* file to be input to the ImageFolder class of the Pytorch library: each image is inserted in a folder representing its class and then all images have been reduced, cropped and finally normalized to have a uniform dataset

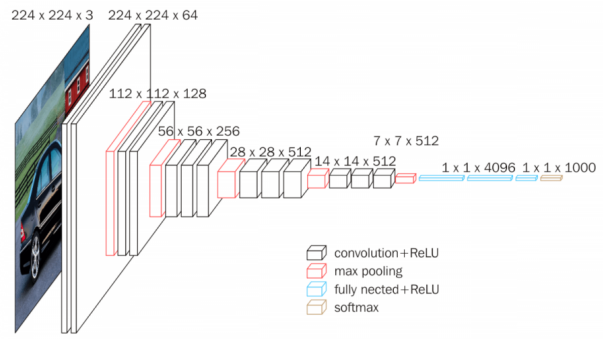


Figure 5: VGG16 architecture.

as shown in Code 4.

```
from torchvision.models import vgg16

# load ImageNet VGG16 Weights
VGG16 = vgg16(pretrained=True)

# pre-process the dataset
normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
transf = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    normalize,
])
# create DataLoader for the validation set
val_loader = torch.utils.data.DataLoader(
    datasets.ImageFolder(val_images_path, transform=transf),
    batch_size=batch_size
)
```

Code 4: PyTorch implementation of VGG16 network with pre-trained ImageNet weights.

With these settings an average **accuracy** value equal to 0.73 and a **loss** value equal to 0.98 were obtained.
The network distinguishes images on 1000 classes having been trained on the ImageNet dataset, but in the validation set considered the number of classes is only 91. One way to increase the network performance would be to restrict the number of possible classes, by restricting the number of neurons of the output layer and retraining the last layers of the network on the considered subset of classes.

2.3 2c

Through the Pytorch library a function has been implemented that creates a model class visualization for a specific input class. The code starts by generating an image through random uniform sampling with the instruction:

```
pil_image = np.uint8 (np.random.uniform (0, 255, (224, 224, 3)))
```

and then for each i iteration the image is processed to obtain a normalized tensor:

```
for i in range(1, iterations):
    # Process image and return tensor variable
    if gauss and i% gauss_freq == 0:
        tensor_img = transform_gauss (pil_image)
    else:
        tensor_img = transform (pil_image)
    ...
```

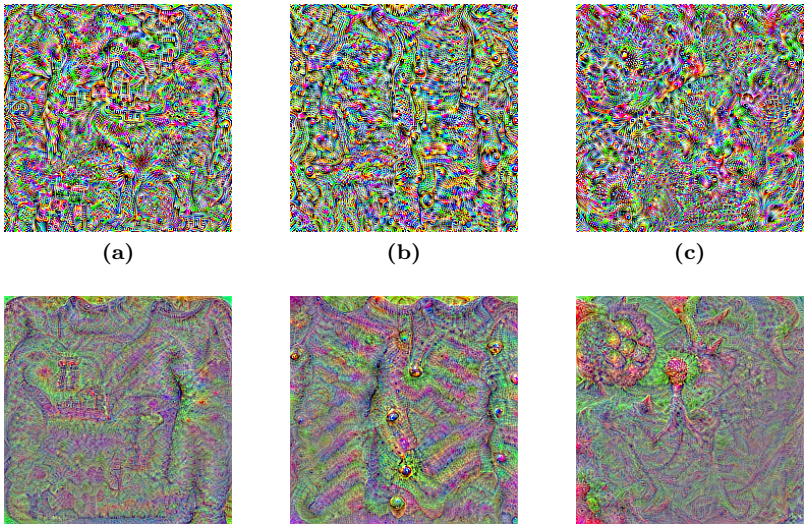
The image processing is realised through the composition of the Transforms [2] functions of the Pytorch library.
As is shown the figure blow they allow to transform Pil Images and tensors ($ToTensor$ and $ToPILImage$), normalize tensors ($normalize$ and $inv\_normalize$) and through custom fuctions modify the dimensions ($squeeze$ and $unsqueeze$) and apply a Gaussian filter to the image ($gauss\_filter$).

```
# Process: from a Pil image to a normalized tensor that can be given as input to the model
normalize = transforms.Normalize(
    mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225]
)
transform = transforms.Compose([
    transforms.ToTensor(),
    normalize,
    transforms.Lambda(unsqueeze),
    transforms.Lambda(auto_grad)
])
# Process: variant with a gauss filter
transform_gauss = transforms.Compose([
    transforms.Lambda(gauss_filter),
    transforms.ToTensor(),
    normalize,
    transforms.Lambda(unsqueeze),
    transforms.Lambda(auto_grad)
])
# Inverse process: from tensor to a Pil Image
inv_normalize = transforms.Normalize(
    mean=[-0.485 / 0.229, -0.456 / 0.224, -0.406 / 0.225],
    std=[1 / 0.229, 1 / 0.224, 1 / 0.225]
)
inv_transform = transforms.Compose([
    transforms.Lambda(squeeze),
    inv_normalize,
    transforms.Lambda(rescale),
    transforms.ToPILImage()
])
```

Pytorch allows also to automatically track and calculate gradients for a tensor if its $requires\_grad$ attribute is set to true. In the code this setting is done using the above $auto\_grad$ function.
In this way, through the backward pass function, the image tensor will be automatically updated for the next iteration:

```
for i in range(1, num_iterations):

    ... Process image ...

    optimizer = SGD([tensor_img], lr=initial_learning_rate)
    # Forward pass to get the image classification
    output = model(tensor_img)
    class_loss = - output[0, target_class]

    model.zero_grad()
    # Backward pass
    class_loss.backward()
    # Update the tensor image
    optimizer.step()

    # Recreate the Pil Image
    pil_image = inv_transform(tensor_img)
```

The results of the model class visualization after 150 iterations with and without the convolution with a Guassian kernel every 5 iterations are shown in the figure 6.



(a) **(b)** **(c)**

(d) **(e)** **(f)**

Figure 6: Model class visualization for the classes sweatshirt (841), cardigan (474) and strawberry (949). The first row shown the normal results, the second row the results obtained with the convolution of a Guassian kernel every 5 iterations.

Qualitatively, the patterns in the images produced, especially those with a Gaussian kernel, recall the respective classes and show which characteristics the network tends to look for each classes as shown in figure 7.
In fact in the image of the raspberry class it is possible to distinguish the characteristic shape of the fruit in the top-left border, in the cardigan image class the various little circles recall by shape and vertical arrangement the typical buttons and for the sweatshirt image class the shape resembles that of the garment and in the upper left there is a pattern that could indicate the presence of the pocket.

2.4 2d

The top five predictions of the model obtained from the 5 test images are shown in the tables below.

1.JPEG	1 result	2 result	3 result	4 result	5 result
class	ski	alp	ski mask	shovel	dogsled
probability	0.72571	0.27219	0.00082	0.00042	0.00029

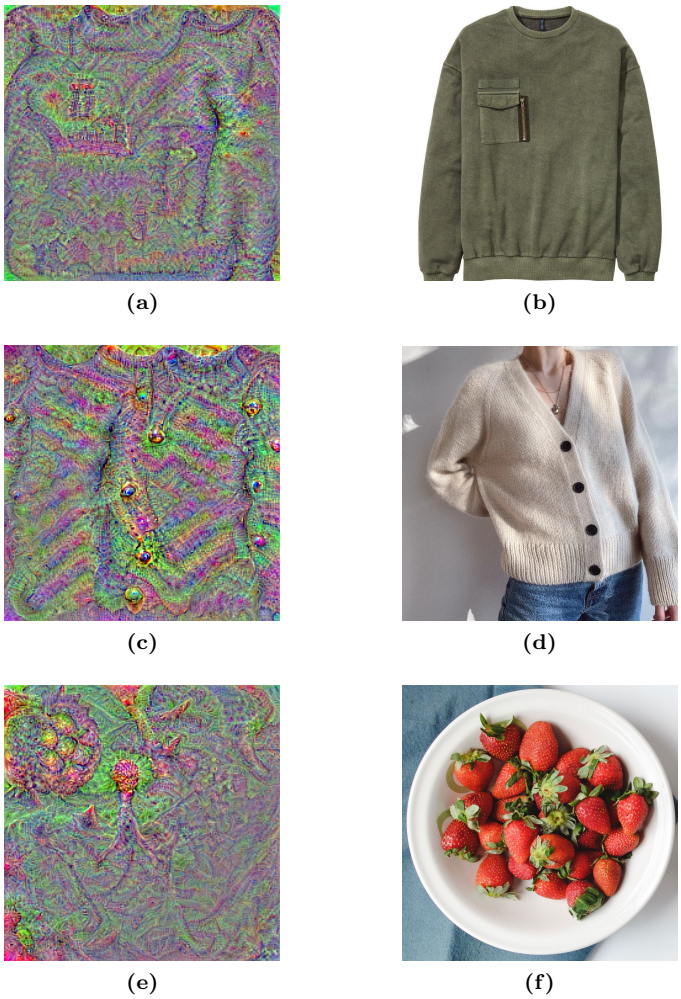
2.JPEG	1 result	2 result	3 result	4 result	5 result
class	kite	vulture	bald eagle	black stork	coucal
probability	0.75790	0.20217	0.03551	0.00066	0.00055

3.JPEG	1 result	2 result	3 result	4 result	5 result
class	white wolf	timber wolf	dingo	Arctic fox	Eskimo dog
probability	0.96277	0.03656	0.00041	0.00012	5.0e-05

4.JPEG	1 result	2 result	3 result	4 result	5 result
class	tile roof	birdhouse	dome	boathouse	greenhouse
probability	0.18325	0.15211	0.04932	0.03522	0.02592

5.JPEG	1 result	2 result	3 result	4 result	5 result
class	rain barrel	rocking chair	folding chair	lawn mower	park bench
probability	0.75931	0.04397	0.04337	0.02274	0.01680

Qualitatively, looking at the images and the predictions produced, the model is able to correctly distinguish the classes of all the images. It is also possible to notice how the best prediction of image 4, even if correct, has a low probability compared to the others.



(a) **(b)** **(c)** **(d)** **(e)** **(f)**

Figure 7: Comparison of model class visualization images and real images of the respective classes.

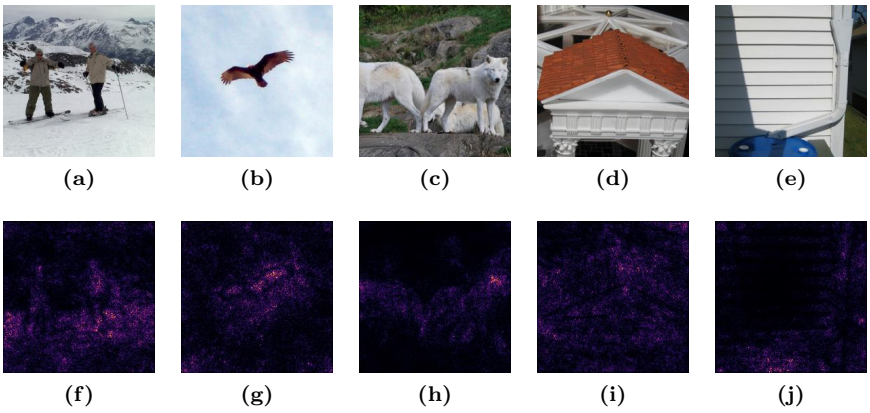


Figure 8: Class saliency map (from **f** to **j**) obtained for the 5 test images (from **a** to **e**).

2.5 2e

A function has been created that implements the generation of Class Saliency Maps according to [1]. First, the derivative of the score with respect to the image is calculated by setting $requires\_grad = True$ through the Pytorch transformations for the image tensor and performing a backward step:

```
...
pil_image = Image.open(d)
tensor_image = transf(pil_image)

output = model(tensor_image)
class_loss, indices = torch.max(output, 1)
# Get the derivative of the score with respect to the image
class_loss.backward()
grad = tensor_image.grad[0]
...
```

Then for each pixel the maximum value between the absolute values of the color channels is taken and the result obtained is saved as an image.

```
...
# M is equal to max of the abs values in the channel axis for every pixel
M, _ = torch.max(torch.abs(grad), dim=0)
# Normalize to [0,1]
M = (M - M.min()) / (M.max() - M.min())

plt.imshow(M, cmap="inferno")
```

The results of the Class Saliency Maps of the test images are shown in figure 8. As it's possible to see from the examples of the test set, with this method we got an estimation of which pixels were the most important for the classification of the image. For example, in images (c) and (h) the wolf's face is highlighted and in images (b), (g) the areas of the image where the bird is present are highlighted too. However, the output is visually very noisy.

2.6 2f

To reduce the amount of noise in the visualization of Class Saliency Maps, a variant that uses guided backpropagation has been implemented. To achieve this aim, the Hooks of the Pytorch library have been used which allow to invoke code during the forward and backward pass of some specific layers of the network: in our case the ReLu layers.
In the forward pass the output is saved in memory to be reused in the backward pass.
In the backward pass the value zero is set and all elements < 0 and also in the same positions that in saved forward pass were < 0.

```
list_relu_forward = []

def guided_relu_forward(module, ten_in, ten_out):
    # save the outputs
    list_relu_forward.append(ten_out)

def guided_relu_backward(module, grad_in, grad_out):
    # Get last forward output
    forward_output = list_relu_forward.pop()
    # Set to zero the elements of the backwards pass < 0
    guided_grad_out = torch.clamp(grad_in[0], min=0.0)
    # Remember the forward stored pass and set to zero the elements that were negative
    forward_output[forward_output > 0] = 1
    guided_grad_out = forward_output * guided_grad_out
    return (guided_grad_out,)

# Modify all Relu to apply guided backpropagation
for i, module in enumerate(model.modules()):
    if isinstance(module, torch.nn.ReLU):
        module.register_forward_hook(guided_relu_forward)
        module.register_backward_hook(guided_relu_backward)
```

The results of the Class Saliency Maps with guided backpropagation of the test images are shown in figure 9.
Comparing the images now obtained with images of Class Saliency Maps without backpropagation it is possible to see how the generated images are much less noisy and they allows to estimate more precisely which pixels the network has considered most for the purposes of classification as is shown in figure 10 where the results of the image 2.JPEG allows to distinguish perfectly the shape pf the bird.

2.7 2g

Dropout is conventionally used during the training phase as regularization method and consists in setting a probability value according to which a particular neuron will be inactive, but it's possible to use it also in the testing phase in order to estimate the model uncertainties.
Leaving the dropout layers active during the test phase creates a different configuration every time that corresponds to a different sample from the weight distribution. In this way multiple forward passes with different dropout configurations yield a different outputs and so a predictive distribution of the data. Taking the mean of the results of the various configurations provides a more stable estimate of the output of the network and the variance of the results indicates how much the network is stable.

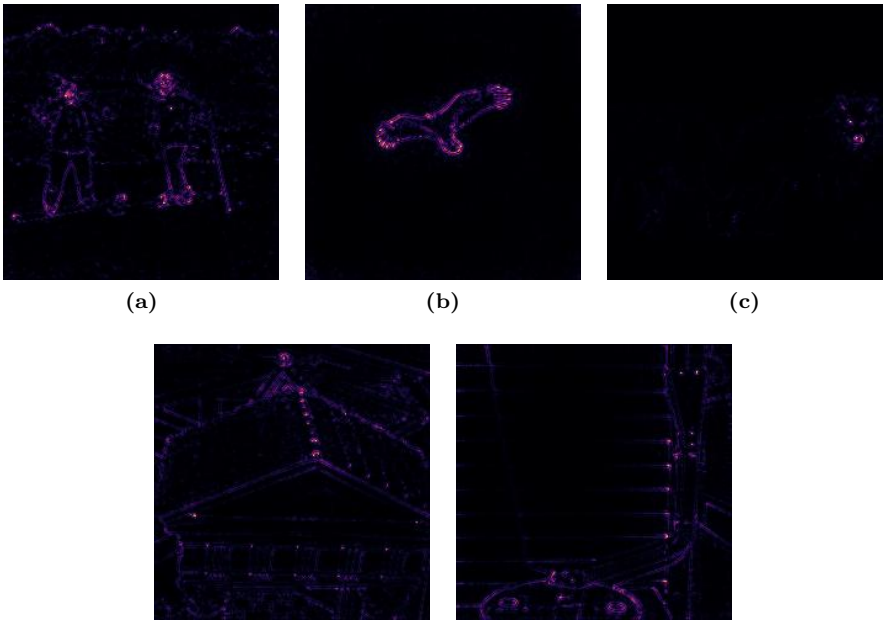


Figure 9: Class saliency map with guided backpropagation obtained for the 5 test images.

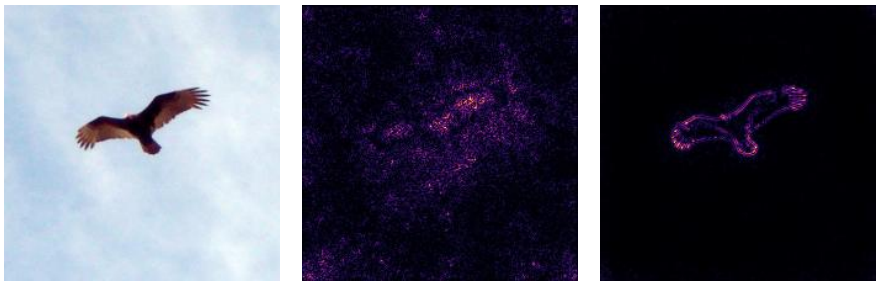


Figure 10: Comparison of the original test image (a), the class saliency map (b) and the class saliency map with guided backpropagation (c).

2.8 2h

To measure the uncertainty of the model via Monte Carlo Dropout on the test images, 100 experiments were performed with the dropout layers of the VGG16 network (dropout ratio = 0.5) also active during the test phase.
For each experiment the network output is saved in a data stack and finally the mean and variance are calculated on it. Through the mean it is possible to understand for each image which class has been predicted most of the times and for this class it is possible to obtain the variance of the probability values obtained from the network. The results are shown obtained in the table below.

Image	Most probable class	variance of the most probable class	Mean per class variance
1.JPEG	795	0.047	9.6e-05
2.JPEG	21	0.025	4.4e-05
3.JPEG	270	0.002	2.6e-06
4.JPEG	858	0.011	6.4e-05
5.JPEG	756	0.050	8.3e-05

The value of the mean variance per class and the variance of the most probable class of image 3 are particularly low, this is in accordance with the fact that the network for this image has as an output a probability value very close to 1 (see table at point 2d) and therefore being very sure of the prediction, even by activating the dropout layers the result remains stable.
The code to reproduce all the experiments can be found at this GitHub link.

3 Problem 3

3.1 3a

To be able to correctly rearrange the image the problem was considered as a classification problem where the network must be able to classify in which of the possible 4! = 24 permutations the image has been mixed.
With this point of view a VGG11 network pretrained on the ImageNet dataset equipped with batch normalization after each convolutional layer was implemented and the last fully connected layer was modified so that it had 24 output neurons corresponding to the 24 possible permutations of the four sections of the image.
The use of batch normalization allows to rescale the output of the convolutional layers for each mini batch to have a mean of zero and a standard deviation of one. In this way we are making the optimization problem more smooth, the network more stabile and it will speed-up the training process.
To increase performance even further, network initialization is made by pretrained weights on the ImageNet dataset.

3.2 3b

To be able to obtain the required level of accuracy, the network described above is not sufficient. This happens due to the lack of data of the trainset which contains only 2500 examples with the respective labels.
The solution was to implement data augmentation through a function that shuffles the reordered image according to any of the 24 permutations. Applying the function to each trainset images we obtain a new trainset with $24 \times 2500 = 60000$ new examples. The data augmentation function is shown in the following code.

```
def super_augmentation(train_images, labels):
    '''
    :param train_images: trainset of images
    :param labels: labels of the trainset
    :return: (new_train_images, new_train_labels) augmented dataset generating for each image all his 24
      shuffle permutation images.
    '''
    new_train_images = []
    new_train_labels = []
    num_images = train_images.shape[0]
    for i in range(num_images):
        image = train_images[i]
        label = labels[i]

        reorder_image = deshuffler(image, label)
        perms = list(permutations(range(0, 4)))

        for perm in perms:
            disordered_image = shuffler(reorder_image, perm)
            new_train_images.append(disordered_image)
            new_train_labels.append(perm)

    return np.array(new_train_images), np.array(new_train_labels)
```

The dataset obtained was then divided into train set and validation set according to a 80% / 20% split and the model is evaluated every epoch. If the loss obtained on the validation set is the lowest ever found, the state of the weights is saved in the memory.
With this settings the network is able to obtain a **average loss** of 0.65, a **top1 accuracy** of 84.5% and a **quarter-level accuracy** of 87.5% on the testset. The loss over the epochs during the training is shown in figure 11.
The use of the pretrained weights on the ImageNet dataset for the initialization of the network is fundamental, without them the maximum achievable accuracy is only around 60%.
The code to reproduce the experiments can be found at this GitHub link.

3.3 3c

The pre-trained network has been removed from its last fully connected layer so that given an input image it can produce a vector of 4096 features.
The images of the *DataNormal* dataset are passed in the network and the obtained features were used to create a nearest neighbor classifier. Its performance to classify unseen data has been compared with a classifier trained directly on the input space (the features are the flatten images) and a classifier whose features are extracted from a network with random weights. The results obtained are shown in the below.

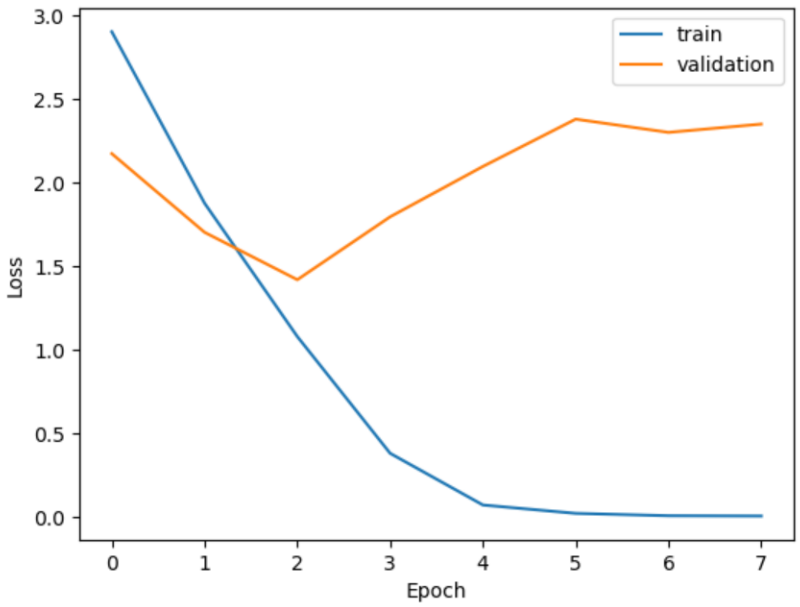


Figure 11: Loss of the network on trainset and validation over the epochs.

Features origin	Accuracy
Pre-trained model	0.40
Input space	0.26
random model	0.15

As it's possible to see from the table, although the model has not been trained on such a dataset, it manages to correctly extract some useful features in order to correctly classify the new data through a nearest neighbor classifier and overcome the other two methods.

3.4 3d

The problem of image deshuffling can be found in artistic contexts such as the reassembly of damaged works of art or archaeological finds. It can also be connected with the field of automatic assembly of mechanical parts.
In conclusion with this work it was possible to implement a network that is able to solve the deshuffling problem and to reach excellent levels of accuracy through the initialization of the weights according to the ImageNet dataset and a phase of data augmentation on the domain of the data.
Finally we have shown how the network is able to generate feature vectors that can also be used for other new unseen contexts, being able to store the information contained in the subjects of the images in a small and meaningful vector.

References

[1] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Deep inside convolutional networks: Visualising image classification models and saliency maps.

[2] PyTorch Transforms. https://pytorch.org/vision/stable/transforms.html