EXTENSION A

SOURCE CODE

MNIST-with-CNN.ipynb

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
from torchvision.utils import make_grid

import numpy as np
import pandas as pd
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
%matplotlib inline

transform = transforms.ToTensor()

train_data = datasets.MNIST(root='../Data', train=True, download=True,
transform=transform)
test_data = datasets.MNIST(root='../Data', train=False, download=True,
transform=transform)

train_data
test_data

train_loader = DataLoader(train_data, batch_size=10, shuffle=True)
test_loader = DataLoader(test_data, batch_size=10, shuffle=False)

# Define layers
conv1 = nn.Conv2d(1, 6, 3, 1)
conv2 = nn.Conv2d(6, 16, 3, 1)

# Grab the first MNIST record
for i, (X_train, y_train) in enumerate(train_data):
    break

# Create a rank-4 tensor to be passed into the model
# (train_loader will have done this already)
x = X_train.view(1,1,28,28)
print(x.shape)
```

```
# Perform the first convolution/activation
x = F.relu(conv1(x))
print(x.shape)

# Run the first pooling layer
x = F.max_pool2d(x, 2, 2)
print(x.shape)

# Perform the second convolution/activation
x = F.relu(conv2(x))
print(x.shape)

# Run the second pooling layer
x = F.max_pool2d(x, 2, 2)
print(x.shape)

# Flatten the data
x = x.view(-1, 5*5*16)
print(x.shape)

class ConvolutionalNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 3, 1)
        self.conv2 = nn.Conv2d(6, 16, 3, 1)
        self.fc1 = nn.Linear(5*5*16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84,10)

    def forward(self, X):
        X = F.relu(self.conv1(X))
        X = F.max_pool2d(X, 2, 2)
        X = F.relu(self.conv2(X))
        X = F.max_pool2d(X, 2, 2)
        X = X.view(-1, 5*5*16)
        X = F.relu(self.fc1(X))
        X = F.relu(self.fc2(X))
        X = self.fc3(X)
        return F.log_softmax(X, dim=1)

torch.manual_seed(42)
model = ConvolutionalNetwork()
model

def count_parameters(model):
    params = [p.numel() for p in model.parameters() if
p.requires_grad]
```

```
    for item in params:
        print(f'{item:>6}')
    print(f'______\n{sum(params):>6}')

count_parameters(model)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

import time
start_time = time.time()

epochs = 5
train_losses = []
test_losses = []
train_correct = []
test_correct = []

for i in range(epochs):
    trn_corr = 0
    tst_corr = 0

    # Run the training batches
    for b, (X_train, y_train) in enumerate(train_loader):
        b+=1

        # Apply the model
        y_pred = model(X_train)  # we don't flatten X-train here
        loss = criterion(y_pred, y_train)

        # Tally the number of correct predictions
        predicted = torch.max(y_pred.data, 1)[1]
        batch_corr = (predicted == y_train).sum()
        trn_corr += batch_corr

        # Update parameters
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Print interim results
        if b%600 == 0:
            print(f'epoch: {i:2}  batch: {b:4} [{10*b:6}/60000]  loss:
{loss.item():10.8f}  \
accuracy: {trn_corr.item()*100/(10*b):7.3f}%')

    train_losses.append(loss.item())
    train_correct.append(trn_corr.item())
```

```
    # Run the testing batches
    with torch.no_grad():
        for b, (X_test, y_test) in enumerate(test_loader):

            # Apply the model
            y_val = model(X_test)

            # Tally the number of correct predictions
            predicted = torch.max(y_val.data, 1)[1]
            tst_corr += (predicted == y_test).sum()

    loss = criterion(y_val, y_test)
    test_losses.append(loss)
    test_correct.append(tst_corr)

print(f'\nDuration: {time.time() - start_time:.0f} seconds') # print
the time elapsed
plt.plot(train_losses, label='training loss')
plt.plot(test_losses, label='validation loss')
plt.title('Loss at the end of each epoch')
plt.legend();
test_losses
plt.plot([t/600 for t in train_correct], label='training accuracy')
plt.plot([t/100 for t in test_correct], label='validation accuracy')
plt.title('Accuracy at the end of each epoch')
plt.legend();
# Extract the data all at once, not in batches
test_load_all = DataLoader(test_data, batch_size=10000, shuffle=False)
with torch.no_grad():
    correct = 0
    for X_test, y_test in test_load_all:
        y_val = model(X_test)  # we don't flatten the data this time
        predicted = torch.max(y_val,1)[1]
        correct += (predicted == y_test).sum()
print(f'Test accuracy: {correct.item()}/{len(test_data)} =
{correct.item()*100/(len(test_data)):7.3f}%')
# print a row of values for reference
np.set_printoptions(formatter=dict(int=lambda x: f'{x:4}'))
print(np.arange(10).reshape(1,10))
print()
# print the confusion matrix
print(confusion_matrix(predicted.view(-1), y_test.view(-1)))

misses = np.array([])
for i in range(len(predicted.view(-1))):
    if predicted[i] != y_test[i]:
```

```
        misses = np.append(misses,i).astype('int64')

# Display the number of misses
len(misses)
# Display the first 10 index positions
misses[:10]
# Set up an iterator to feed batched rows
r = 12   # row size
row = iter(np.array_split(misses,len(misses)//r+1))
nextrow = next(row)
print("Index:", nextrow)
print("Label:", y_test.index_select(0,torch.tensor(nextrow)).numpy())
print("Guess:",
predicted.index_select(0,torch.tensor(nextrow)).numpy())
images = X_test.index_select(0,torch.tensor(nextrow))
im = make_grid(images, nrow=r)
plt.figure(figsize=(10,4))
plt.imshow(np.transpose(im.numpy(), (1, 2, 0)));

x = 2019
plt.figure(figsize=(1,1))
plt.imshow(test_data[x][0].reshape((28,28)), cmap="gist_yarg");

model.eval()
with torch.no_grad():
    new_pred = model(test_data[x][0].view(1,1,28,28)).argmax()
print("Predicted value:",new_pred.item())
```

Pneumonia-Preprocess.ipynb

```
from pathlib import Path
import pydicom
import numpy as np
import cv2
import pandas as pd
import matplotlib.pyplot as plt
from tqdm.notebook import tqdm

labels = pd.read_csv("/path/to/rsna-pneumonia-detection-
challenge/stage_2_train_labels.csv")
labels.head(6)
```

```
# Remove duplicate entries
labels = labels.drop_duplicates("patientId")
labels.head()
ROOT_PATH = Path("/path/to/rsna-pneumonia-detection-
challenge/stage_2_train_images/")
SAVE_PATH = Path("Processed/")
fig, axis = plt.subplots(3, 3, figsize=(9, 9))
c = 0
for i in range(3):
    for j in range(3):
        patient_id = labels.patientId.iloc[c]
        dcm_path = ROOT_PATH/patient_id
        dcm_path = dcm_path.with_suffix(".dcm")
        dcm = pydicom.read_file(dcm_path).pixel_array

        label = labels["Target"].iloc[c]

        axis[i][j].imshow(dcm, cmap="bone")
        axis[i][j].set_title(label)
        c+=1
sums = 0
sums_squared = 0
for c, patient_id in enumerate(tqdm(labels.patientId)):
    dcm_path = ROOT_PATH/patient_id  # Create the path to the dcm file
    dcm_path = dcm_path.with_suffix(".dcm")  # And add the .dcm suffix
    # Read the dicom file with pydicom and standardize the array
    dcm = pydicom.read_file(dcm_path).pixel_array / 255
    # Resize the image as 1024x1024 is way to large to be handeled by
Deep Learning models at the moment
    # Let's use a shape of 224x224
    # In order to use less space when storing the image we convert it
to float16
    dcm_array = cv2.resize(dcm, (224, 224)).astype(np.float16)
    # Retrieve the corresponding label
```

```
    label = labels.Target.iloc[c]
    # 4/5 train split, 1/5 val split
    train_or_val = "train" if c < 24000 else "val"
    current_save_path = SAVE_PATH/train_or_val/str(label) # Define
save path and create if necessary
    current_save_path.mkdir(parents=True, exist_ok=True)
    np.save(current_save_path/patient_id, dcm_array)  # Save the array
in the corresponding directory
    normalizer = dcm_array.shape[0] * dcm_array.shape[1]  # Normalize
sum of image
    if train_or_val == "train":  # Only use train data to compute
dataset statistics
        sums += np.sum(dcm_array) / normalizer
        sums_squared += (np.power(dcm_array, 2).sum()) / normalizer
mean = sums / 24000
std = np.sqrt(sums_squared / 24000 - (mean**2))
```

Pneumonia-Train.ipynb

```
import torch
import torchvision
from torchvision import transforms
import torchmetrics
import pytorch_lightning as pl
from pytorch_lightning.callbacks import ModelCheckpoint
from pytorch_lightning.loggers import TensorBoardLogger
from tqdm.notebook import tqdm
import numpy as np
import matplotlib.pyplot as plt

def load_file(path):
    return np.load(path).astype(np.float32)
```

```
train_transforms = transforms.Compose([
                                    transforms.ToTensor(),  # Convert
numpy array to tensor
                                    transforms.Normalize(0.49, 0.248),
# Use mean and std from preprocessing notebook
                                    transforms.RandomAffine( # Data
Augmentation
                                        degrees=(-5, 5), translate=(0,
0.05), scale=(0.9, 1.1)),
                                transforms.RandomResizedCrop((224,
224), scale=(0.35, 1))
])

val_transforms = transforms.Compose([
                                    transforms.ToTensor(),  # Convert
numpy array to tensor
                                    transforms.Normalize([0.49],
[0.248]),  # Use mean and std from preprocessing notebook
])
train_dataset = torchvision.datasets.DatasetFolder(
    "Processed/train/",
    loader=load_file, extensions="npy", transform=train_transforms)
val_dataset = torchvision.datasets.DatasetFolder(
    "Processed/val/",
    loader=load_file, extensions="npy", transform=val_transforms)
fig, axis = plt.subplots(2, 2, figsize=(9, 9))
for i in range(2):
    for j in range(2):
        random_index = np.random.randint(0, 20000)
        x_ray, label = train_dataset[random_index]
        axis[i][j].imshow(x_ray[0], cmap="bone")
        axis[i][j].set_title(f"Label:{label}")
batch_size = 64#TODO
num_workers = 4# TODO
```

```
train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=batch_size, num_workers=num_workers, shuffle=True)
val_loader = torch.utils.data.DataLoader(val_dataset,
batch_size=batch_size, num_workers=num_workers, shuffle=False)
print(f"There are {len(train_dataset)} train images and
{len(val_dataset)} val images")
np.unique(train_dataset.targets, return_counts=True),
np.unique(val_dataset.targets, return_counts=True)
class PneumoniaModel(pl.LightningModule):
    def __init__(self, weight=1):
        super().__init__()
        self.model = torchvision.models.resnet18()
        # change conv1 from 3 to 1 input channels
        self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7),
stride=(2, 2), padding=(3, 3), bias=False)
        # change out_feature of the last fully connected layer (called
fc in resnet18) from 1000 to 1
        self.model.fc = torch.nn.Linear(in_features=512,
out_features=1)

        self.optimizer = torch.optim.Adam(self.model.parameters(),
lr=1e-4)
        self.loss_fn =
torch.nn.BCEWithLogitsLoss(pos_weight=torch.tensor([weight]))
        # simple accuracy computation
        self.train_acc = torchmetrics.Accuracy()
        self.val_acc = torchmetrics.Accuracy()
    def forward(self, data):
        pred = self.model(data)
        return pred
    def training_step(self, batch, batch_idx):
        x_ray, label = batch
```

```
        label = label.float()  # Convert label to float (just needed
for loss computation)
        pred = self(x_ray)[:,0]  # Prediction: Make sure prediction
and label have same shape
        loss = self.loss_fn(pred, label)  # Compute the loss

        # Log loss and batch accuracy
        self.log("Train Loss", loss)
        self.log("Step Train Acc", self.train_acc(torch.sigmoid(pred),
label.int()))
        return loss
    def training_epoch_end(self, outs):
        # After one epoch compute the whole train_data accuracy
        self.log("Train Acc", self.train_acc.compute())
    def validation_step(self, batch, batch_idx):
        # Same steps as in the training_step
        x_ray, label = batch
        label = label.float()
        pred = self(x_ray)[:,0]  # make sure prediction and label have
same shape

        loss = self.loss_fn(pred, label)
        # Log validation metrics
        self.log("Val Loss", loss)
        self.log("Step Val Acc", self.val_acc(torch.sigmoid(pred),
label.int()))
        return loss

    def validation_epoch_end(self, outs):
        self.log("Val Acc", self.val_acc.compute())

    def configure_optimizers(self):
        #Caution! You always need to return a list here (just pack
your optimizer into one :))
```

```
        return [self.optimizer]

model = PneumoniaModel()  # Instanciate the model

# Create the checkpoint callback
checkpoint_callback = ModelCheckpoint(
    monitor='Val Acc',
    save_top_k=10,
    mode='max')

# Create the trainer
# Change the gpus parameter to the number of available gpus on your
system. Use 0 for CPU training

gpus = 1 #TODO
trainer = pl.Trainer(gpus=gpus,
logger=TensorBoardLogger(save_dir="./logs"), log_every_n_steps=1,
                     callbacks=checkpoint_callback,
                     max_epochs=35)

trainer.fit(model, train_loader, val_loader)

device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")

# Use strict=False, otherwise we would want to match the pos_weight
which is not necessary
model = PneumoniaModel.load_from_checkpoint("weights/weights_1.ckpt")
model.eval()
model.to(device);

preds = []
labels = []
```

```
with torch.no_grad():
    for data, label in tqdm(val_dataset):
        data = data.to(device).float().unsqueeze(0)
        pred = torch.sigmoid(model(data)[0].cpu())
        preds.append(pred)
        labels.append(label)
preds = torch.tensor(preds)
labels = torch.tensor(labels).int()
acc = torchmetrics.Accuracy()(preds, labels)
precision = torchmetrics.Precision()(preds, labels)
recall = torchmetrics.Recall()(preds, labels)
cm = torchmetrics.ConfusionMatrix(num_classes=2)(preds, labels)
cm_threshed = torchmetrics.ConfusionMatrix(num_classes=2,
threshold=0.25)(preds, labels)
print(f"Val Accuracy: {acc}")
print(f"Val Precision: {precision}")
print(f"Val Recall: {recall}")
print(f"Confusion Matrix:\n {cm}")
print(f"Confusion Matrix 2:\n {cm_threshed}")
fig, axis = plt.subplots(3, 3, figsize=(9, 9))
for i in range(3):
    for j in range(3):
        rnd_idx = np.random.randint(0, len(preds))
        axis[i][j].imshow(val_dataset[rnd_idx][0][0], cmap="bone")
        axis[i][j].set_title(f"Pred:{int(preds[rnd_idx] > 0.5)},
Label:{labels[rnd_idx]}")
        axis[i][j].axis("off")
```

Penumonia-Interpretability.ipynb

```
%matplotlib notebook
import torch
import torchvision
from torchvision import transforms
```

```
import pytorch_lightning as pl
import numpy as np
import matplotlib.pyplot as plt

def load_file(path):
    return np.load(path).astype(np.float32)

val_transforms = transforms.Compose([
                                transforms.ToTensor(),
                                transforms.Normalize(0.49, 0.248),
])
val_dataset = torchvision.datasets.DatasetFolder("Processed/val/",
loader=load_file, extensions="npy", transform=val_transforms)
temp_model = torchvision.models.resnet18()
temp_model
list(temp_model.children())[:-2]  # get all layers up to avgpool
torch.nn.Sequential(*list(temp_model.children())[:-2])
class PneumoniaModel(pl.LightningModule):
    def __init__(self):
        super().__init__()

        self.model = torchvision.models.resnet18()
        # Change conv1 from 3 to 1 input channels
        self.model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=(7, 7),
stride=(2, 2), padding=(3, 3), bias=False)
        # Change out_feature of the last fully connected layer (called
fc in resnet18) from 1000 to 1
        self.model.fc = torch.nn.Linear(in_features=512,
out_features=1)
        # Extract the feature map
        self.feature_map =
torch.nn.Sequential(*list(self.model.children())[:-2])
    def forward(self, data):
        # Compute feature map
```

```
        feature_map = self.feature_map(data)
        # Use Adaptive Average Pooling as in the original model
        avg_pool_output =
torch.nn.functional.adaptive_avg_pool2d(input=feature_map,
output_size=(1, 1))
        print(avg_pool_output.shape)
        # Flatten the output into a 512 element vector
        avg_pool_output_flattened = torch.flatten(avg_pool_output)
        print(avg_pool_output_flattened.shape)
        # Compute prediction
        pred = self.model.fc(avg_pool_output_flattened)
        return pred, feature_map

def cam(model, img):
    with torch.no_grad():
        pred, features = model(img.unsqueeze(0))
    features = features.reshape((512, 49))
    weight_params = list(model.model.fc.parameters())[0]
    weight = weight_params[0].detach()



    cam = torch.matmul(weight, features)
    cam_img = cam.reshape(7, 7).cpu()
    return cam_img, torch.sigmoid(pred)

# Use strict to prevent pytorch from loading weights for
self.feature_map
model = PneumoniaModel.load_from_checkpoint("weights/weights_3.ckpt",
strict=False)
model.eval();

def cam(model, img):
    """
    Compute class activation map according to cam algorithm
```

```
    """
    with torch.no_grad():
        pred, features = model(img.unsqueeze(0))
    b, c, h, w = features.shape

    # We reshape the 512x7x7 feature tensor into a 512x49 tensor in
order to simplify the multiplication
    features = features.reshape((c, h*w))

    # Get only the weights, not the bias
    weight_params = list(model.model.fc.parameters())[0]

    # Remove gradient information from weight parameters to enable
numpy conversion
    weight = weight_params[0].detach()
    print(weight.shape)
    # Compute multiplication between weight and features with the
formula from above.
    # We use matmul because it directly multiplies each filter with
the weights
    # and then computes the sum. This yields a vector of 49 (7x7
elements)
    cam = torch.matmul(weight, features)
    print(features.shape)

    ### The following loop performs the same operations in a less
optimized way
    #cam = torch.zeros((7 * 7))
    #for i in range(len(cam)):
    #    cam[i] = torch.sum(weight*features[:,i])
    ######################################################################
    # Normalize and standardize the class activation map (Not always
necessary, thus not shown in the lecture)
    cam = cam - torch.min(cam)
```

```
    cam_img = cam / torch.max(cam)
    # Reshape the class activation map to 512x7x7 and move the tensor
back to CPU
    cam_img = cam_img.reshape(h, w).cpu()

    return cam_img, torch.sigmoid(pred)

def visualize(img, heatmap, pred):
    """
    Visualization function for class activation maps
    """
    img = img[0]
    # Resize the activation map of size 7x7 to the original image size
(224x224)
    heatmap = transforms.functional.resize(heatmap.unsqueeze(0),
(img.shape[0], img.shape[1]))[0]

    # Create a figure
    fig, axis = plt.subplots(1, 2)

    axis[0].imshow(img, cmap="bone")
    # Overlay the original image with the upscaled class activation
map
    axis[1].imshow(img, cmap="bone")
    axis[1].imshow(heatmap, alpha=0.5, cmap="jet")
    plt.title(f"Pneumonia: {(pred > 0.5).item()}")

def visualize(img, cam, pred):
    img = img[0]
    cam = transforms.functional.resize(cam.unsqueeze(0), (224,
224))[0]

    fig, axis = plt.subplots(1, 2)
    axis[0].imshow(img, cmap="bone")
```

```
    axis[1].imshow(img, cmap="bone")
    axis[1].imshow(cam, alpha=0.5, cmap="jet")
    plt.title(pred)

img = val_dataset[-6][0]  # Select a subject
activation_map, pred = cam(model, img)  # Compute the Class activation
map given the subject

visualize(img, activation_map, pred)  # Visualize CAM
```

Liver-Tumour-Data.ipynb

```
%matplotlib notebook
from pathlib import Path
import nibabel as nib
import matplotlib.pyplot as plt
import numpy as np
from celluloid import Camera
from IPython.display import HTML
root = Path("Task03_Liver_rs/imagesTr/")
label = Path("Task03_Liver_rs/labelsTr/")
def change_img_to_label_path(path):
    """
    Replaces imagesTr with labelsTr
    """
    parts = list(path.parts)  # get all directories within the path
    parts[parts.index("imagesTr")] = "labelsTr"  # Replace imagesTr
with labelsTr
    return Path(*parts)  # Combine list back into a Path object
sample_path = list(root.glob("liver*"))[0]  # Choose a subject
sample_path_label = change_img_to_label_path(sample_path)

data = nib.load(sample_path)
label = nib.load(sample_path_label)
```

```
ct = data.get_fdata()
mask = label.get_fdata().astype(int)  # Class labels should not be
handled as float64
nib.aff2axcodes(data.affine)
fig = plt.figure()
camera = Camera(fig)  # Create the camera object from celluloid
for i in range(ct.shape[2]):  # Axial view
    plt.imshow(ct[:,:,i], cmap="bone")
    mask_ = np.ma.masked_where(mask[:,:,i]==0, mask[:,:,i])
    plt.imshow(mask_, alpha=0.5)
    # plt.axis("off")
    camera.snap()  # Store the current slice
plt.tight_layout()
animation = camera.animate()  # Create the animation
HTML(animation.to_html5_video())
```

Liver-Tumour-Model.ipynb

```
import torch

class DoubleConv(torch.nn.Module):
    """
    Helper Class which implements the intermediate Convolutions
    """
    def __init__(self, in_channels, out_channels):

        super().__init__()
        self.step = torch.nn.Sequential(torch.nn.Conv3d(in_channels,
out_channels, 3, padding=1),
                                        torch.nn.ReLU(),
                                        torch.nn.Conv3d(out_channels,
out_channels, 3, padding=1),
                                        torch.nn.ReLU())
```

```
    def forward(self, X):
        return self.step(X)


class UNet(torch.nn.Module):
    """
    This class implements a UNet for the Segmentation
    We use 3 down- and 3 UpConvolutions and two Convolutions in each
step
    """

    def __init__(self):
        """Sets up the U-Net Structure
        """
        super().__init__()


        ############## DOWN #####################
        self.layer1 = DoubleConv(1, 32)
        self.layer2 = DoubleConv(32, 64)
        self.layer3 = DoubleConv(64, 128)
        self.layer4 = DoubleConv(128, 256)
        ############### UP ########################
        self.layer5 = DoubleConv(256 + 128, 128)
        self.layer6 = DoubleConv(128+64, 64)
        self.layer7 = DoubleConv(64+32, 32)
        self.layer8 = torch.nn.Conv3d(32, 3, 1)  # Output: 3 values ->
background, liver, tumor

        self.maxpool = torch.nn.MaxPool3d(2)

    def forward(self, x):
```

```
        ####### DownConv 1#########
        x1 = self.layer1(x)
        x1m = self.maxpool(x1)
        ####### DownConv 2#########
        x2 = self.layer2(x1m)
        x2m = self.maxpool(x2)
        ####### DownConv 3#########
        x3 = self.layer3(x2m)
        x3m = self.maxpool(x3)
        ##### Intermediate Layer ##
        x4 = self.layer4(x3m)
        ####### UpCONV 1#########
        x5 = torch.nn.Upsample(scale_factor=2, mode="trilinear")(x4)
# Upsample with a factor of 2
        x5 = torch.cat([x5, x3], dim=1)  # Skip-Connection
        x5 = self.layer5(x5)
        ####### UpCONV 2#########
        x6 = torch.nn.Upsample(scale_factor=2, mode="trilinear")(x5)
        x6 = torch.cat([x6, x2], dim=1)  # Skip-Connection
        x6 = self.layer6(x6)
        ####### UpCONV 3#########
        x7 = torch.nn.Upsample(scale_factor=2, mode="trilinear")(x6)
        x7 = torch.cat([x7, x1], dim=1)
        x7 = self.layer7(x7)
        ####### Predicted segmentation#########
        ret = self.layer8(x7)
        return ret

model = UNet()

random_input = torch.randn(1, 1, 128, 128, 128)

with torch.no_grad():
    output = model(random_input)
```

```
assert output.shape == torch.Size([1, 3, 128, 128, 128])
```

Liver-Tumour-Train.ipynb

```
from pathlib import Path

import torchio as tio
import torch
import pytorch_lightning as pl
from pytorch_lightning.callbacks import ModelCheckpoint
from pytorch_lightning.loggers import TensorBoardLogger
import matplotlib.pyplot as plt
import numpy as np

from model import UNet

def change_img_to_label_path(path):
    """
    Replace data with mask to get the masks
    """
    parts = list(path.parts)
    parts[parts.index("imagesTr")] = "labelsTr"
    return Path(*parts)

path = Path("Task03_Liver_rs/imagesTr/")
subjects_paths = list(path.glob("liver_*"))
subjects = []

for subject_path in subjects_paths:
    label_path = change_img_to_label_path(subject_path)
    subject = tio.Subject({"CT":tio.ScalarImage(subject_path),
"Label":tio.LabelMap(label_path)})
    subjects.append(subject)

for subject in subjects:
```

```
    assert subject["CT"].orientation == ("R", "A", "S")

process = tio.Compose([
            tio.CropOrPad((256, 256, 200)),
            tio.RescaleIntensity((-1, 1))
            ])


augmentation = tio.RandomAffine(scales=(0.9, 1.1), degrees=(-10, 10))


val_transform = process
train_transform = tio.Compose([process, augmentation])

train_dataset = tio.SubjectsDataset(subjects[:105],
transform=train_transform)
val_dataset = tio.SubjectsDataset(subjects[105:],
transform=val_transform)

sampler = tio.data.LabelSampler(patch_size=96, label_name="Label",
label_probabilities={0:0.2, 1:0.3, 2:0.5})
#sampler = tio.data.UniformSampler(patch_size=96)

# Todo: Adapt max_length and num_workers to your hardware

train_patches_queue = tio.Queue(
     train_dataset,
     max_length=40,
     samples_per_volume=5,
     sampler=sampler,
     num_workers=4,
    )

val_patches_queue = tio.Queue(
```

```
     val_dataset,
     max_length=40,
     samples_per_volume=5,
     sampler=sampler,
     num_workers=4,
    )

# TODO, adapt batch size according to your hardware
batch_size = 2

train_loader = torch.utils.data.DataLoader(train_patches_queue,
batch_size=batch_size, num_workers=0)
val_loader = torch.utils.data.DataLoader(val_patches_queue,
batch_size=batch_size, num_workers=0)

class Segmenter(pl.LightningModule):
    def __init__(self):
        super().__init__()

        self.model = UNet()

        self.optimizer = torch.optim.Adam(self.model.parameters(),
lr=1e-4)
        self.loss_fn = torch.nn.CrossEntropyLoss()

    def forward(self, data):
        pred = self.model(data)
        return pred

    def training_step(self, batch, batch_idx):
        # You can obtain the raw volume arrays by accessing the data
attribute of the subject
        img = batch["CT"]["data"]
```

```
        mask = batch["Label"]["data"][:,0]  # Remove single channel as
CrossEntropyLoss expects NxHxW
        mask = mask.long()

        pred = self(img)
        loss = self.loss_fn(pred, mask)

        # Logs
        self.log("Train Loss", loss)
        if batch_idx % 50 == 0:
            self.log_images(img.cpu(), pred.cpu(), mask.cpu(),
"Train")
        return loss


    def validation_step(self, batch, batch_idx):
        # You can obtain the raw volume arrays by accessing the data
attribute of the subject
        img = batch["CT"]["data"]
        mask = batch["Label"]["data"][:,0]  # Remove single channel as
CrossEntropyLoss expects NxHxW
        mask = mask.long()

        pred = self(img)
        loss = self.loss_fn(pred, mask)

        # Logs
        self.log("Val Loss", loss)
        self.log_images(img.cpu(), pred.cpu(), mask.cpu(), "Val")

        return loss

    def log_images(self, img, pred, mask, name):
```

```
        results = []
        pred = torch.argmax(pred, 1) # Take the output with the
highest value
        axial_slice = 50  # Always plot slice 50 of the 96 slices

        fig, axis = plt.subplots(1, 2)
        axis[0].imshow(img[0][0][:,:,axial_slice], cmap="bone")
        mask_ = np.ma.masked_where(mask[0][:,:,axial_slice]==0,
mask[0][:,:,axial_slice])
        axis[0].imshow(mask_, alpha=0.6)
        axis[0].set_title("Ground Truth")

        axis[1].imshow(img[0][0][:,:,axial_slice], cmap="bone")
        mask_ = np.ma.masked_where(pred[0][:,:,axial_slice]==0,
pred[0][:,:,axial_slice])
        axis[1].imshow(mask_, alpha=0.6, cmap="autumn")
        axis[1].set_title("Pred")

        self.logger.experiment.add_figure(f"{name} Prediction vs
Label", fig, self.global_step)



    def configure_optimizers(self):
        #Caution! You always need to return a list here (just pack
your optimizer into one :))
        return [self.optimizer]


# Instanciate the model
model = Segmenter()

# Create the checkpoint callback
checkpoint_callback = ModelCheckpoint(
```

```
    monitor='Val Loss',
    save_top_k=10,
    mode='min')

# Create the trainer
# Change the gpus parameter to the number of available gpus in your
computer. Use 0 for CPU training

gpus = 1 #TODO
trainer = pl.Trainer(gpus=gpus,
logger=TensorBoardLogger(save_dir="./logs"), log_every_n_steps=1,
                     callbacks=checkpoint_callback,
                     max_epochs=100)

# Train the model.
# This might take some hours depending on your GPU
trainer.fit(model, train_loader, val_loader)

from IPython.display import HTML
from celluloid import Camera

model = Segmenter.load_from_checkpoint("weights/epoch=97-
step=25773.ckpt")
model = model.eval()
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
model.to(device);

# Select a validation subject and extract the images and segmentation
for evaluation
IDX = 4
mask = val_dataset[IDX]["Label"]["data"]
imgs = val_dataset[IDX]["CT"]["data"]
```

```
# GridSampler
grid_sampler = tio.inference.GridSampler(val_dataset[IDX], 96, (8, 8,
8))
# GridAggregator
aggregator = tio.inference.GridAggregator(grid_sampler)
# DataLoader for speed up
patch_loader = torch.utils.data.DataLoader(grid_sampler, batch_size=4)
# Prediction
with torch.no_grad():
    for patches_batch in patch_loader:
        input_tensor = patches_batch['CT']["data"].to(device)  # Get
batch of patches
        locations = patches_batch[tio.LOCATION]  # Get locations of
patches
        pred = model(input_tensor)  # Compute prediction
        aggregator.add_batch(pred, locations)  # Combine predictions
to volume
# Extract the volume prediction
output_tensor = aggregator.get_output_tensor()
fig = plt.figure()
camera = Camera(fig)  # create the camera object from celluloid
pred = output_tensor.argmax(0)
for i in range(0, output_tensor.shape[3], 2):  # axial view
    plt.imshow(imgs[0,:,:,i], cmap="bone")
    mask_ = np.ma.masked_where(pred[:,:,i]==0, pred[:,:,i])
    label_mask = np.ma.masked_where(mask[0,:,:,i]==0, mask[0,:,:,i])
    plt.imshow(mask_, alpha=0.1, cmap="autumn")
    #plt.imshow(label_mask, alpha=0.5, cmap="jet")  # Uncomment if you
want to see the label

    # plt.axis("off")
    camera.snap()  # Store the current slice
animation = camera.animate()  # create the animation
HTML(animation.to_html5_video())  # convert the animation to a video
```

ДОДАТОК Б

Відомість кваліфікаційної роботи

Позначення	Найменування	Дод. відомості
	Текстові документи	
1.	Пояснювальна записка	103 с.
	Інші документи	
2.	Презентаційні матеріали	__ плакатів

.	.	Прізвище та ініціали.	Підп.	Дата			
Розробив		Арутюнов Е.Р.			Сегментація зображень пневмонії та пухлини за допомогою згорткових нейронних мереж з набором даних MNIST	Шифр групи	Код напр./спец.
Перевірив		Філатов В.О.				СШІм-20-3	122
Н.контр.		Малєєва І.А.				ХНУРЕ кафедра ШІ	
Затв.		Філатов В.О.					