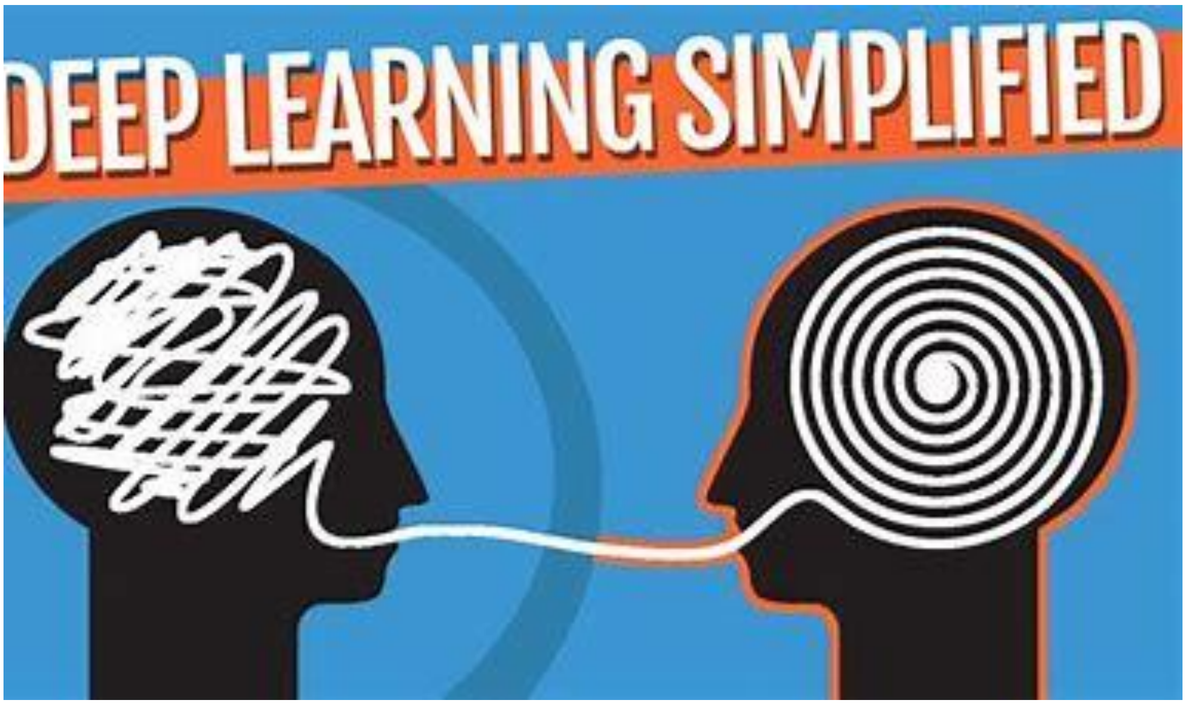
Projet Deep learning

M2 IA School 2020/2021



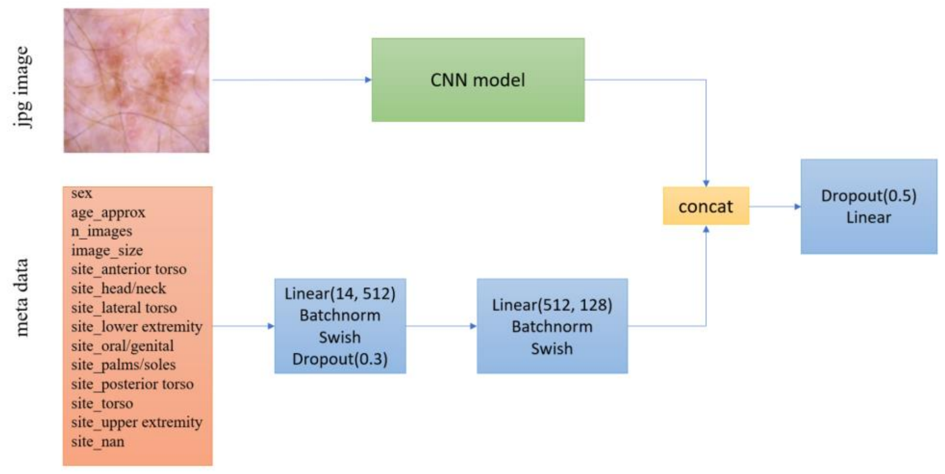
Sujet : SIIM- ISIC Melanoma Classification

Url Kaggle : https://www.kaggle.com/c/siim-isic-melanoma-classification/discussion/177726

Equipe : MBAREK Marouene , Munirah ALAFALEQ , Teddy NIEKE

Sommaire :

1- Introduction

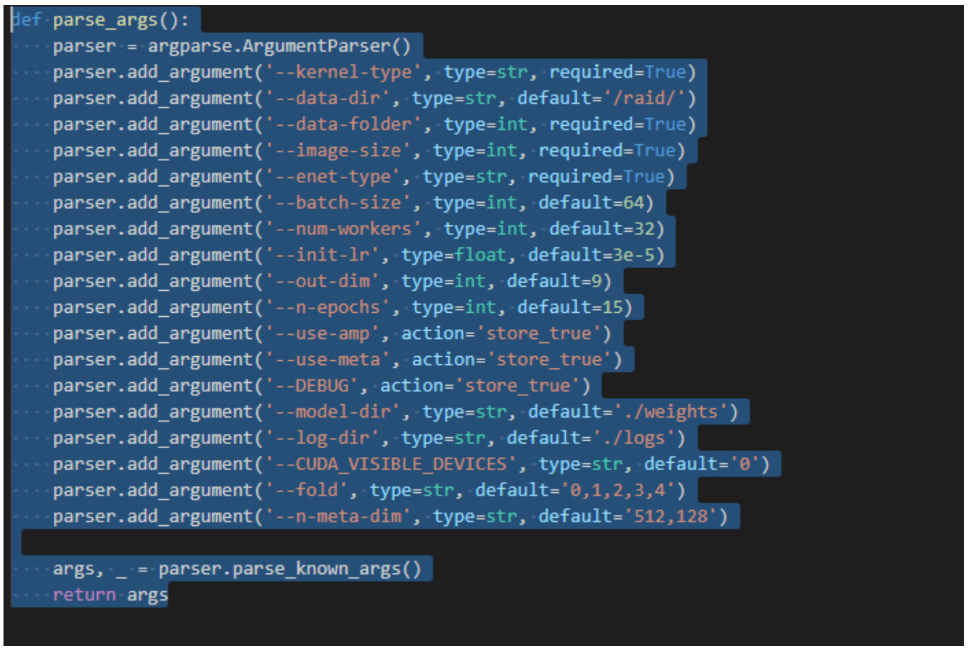


Cette compétition consiste à développer un réseau de neurones permettant de prédire l'existence d'un mélanome de peau. Ce réseau est constitué de deux couches : la première est un réseau de neurones convolutifs (Resnest , Seresnext ou Effnet) sur lequel on va entrainer une liste d'images quant à la deuxième il s'agit d'un réseau neuronal ordinaire (FNN) composé de (Linear (14,512) + Batchnorm + Swich + Dropout(0,3) + Linear (512, 128) + Batchnorm + Swich) sur lequel on va entrainer des données au format CSV . Le résultat final est le concaténation des deux résultats envoyé par les deux sous couches

2- Exécution

2-1 Passage des arguments

Pour pouvoir exécuter ce réseau de neurones l'utilisateur doit passer une liste de neurones

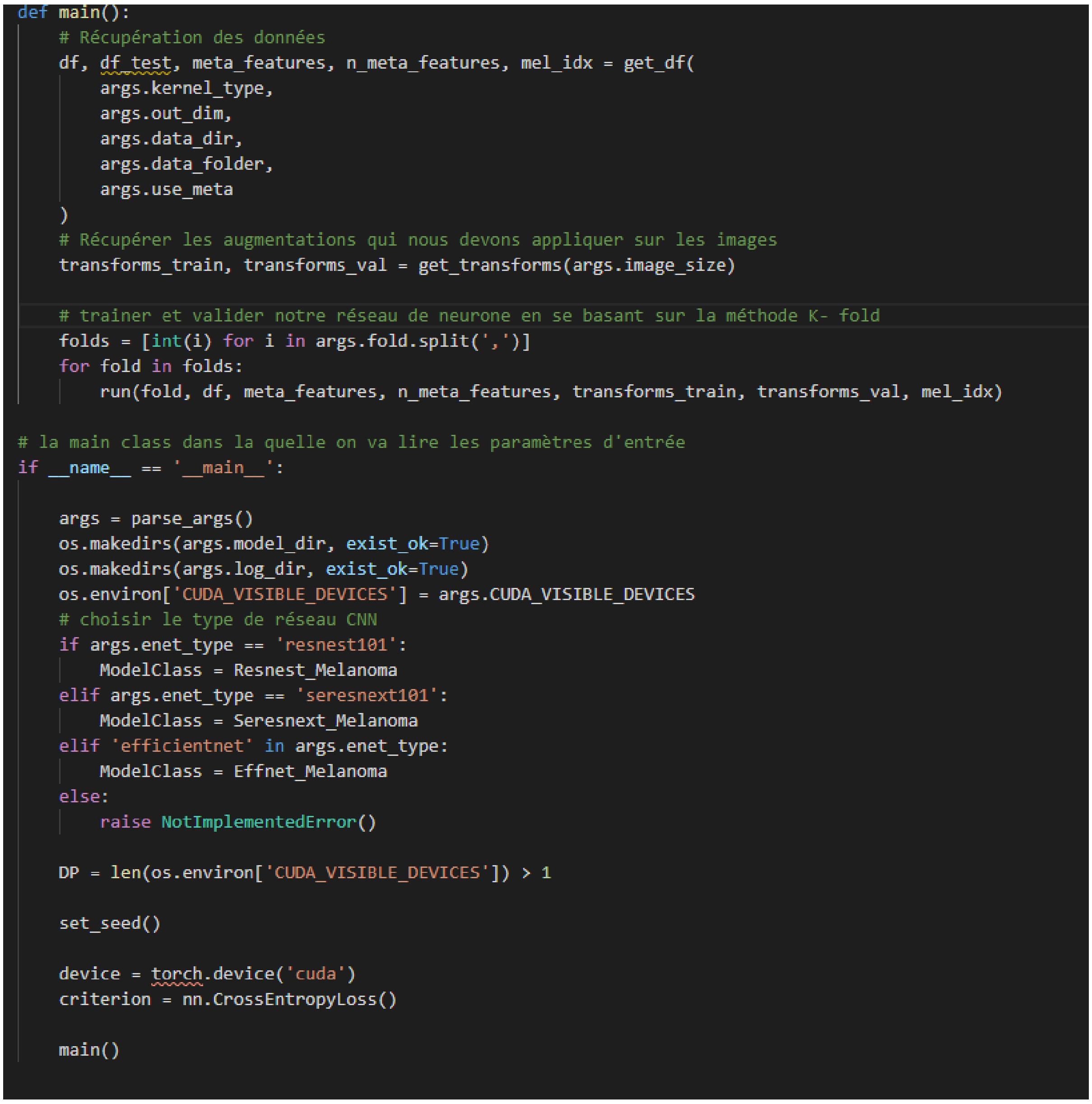


```
def parse_args():
    parser = argparse.ArgumentParser()
    parser.add_argument('--kernel-type', type=str, required=True)
    parser.add_argument('--data-dir', type=str, default='/raid/')
    parser.add_argument('--data-folder', type=int, required=True)
    parser.add_argument('--image-size', type=int, required=True)
    parser.add_argument('--enet-type', type=str, required=True)
    parser.add_argument('--batch-size', type=int, default=64)
    parser.add_argument('--num-workers', type=int, default=32)
    parser.add_argument('--init-lr', type=float, default=3e-5)
    parser.add_argument('--out-dim', type=int, default=9)
    parser.add_argument('--n-epochs', type=int, default=15)
    parser.add_argument('--use-amp', action='store_true')
    parser.add_argument('--use-meta', action='store_true')
    parser.add_argument('--DEBUG', action='store_true')
    parser.add_argument('--model-dir', type=str, default='./weights')
    parser.add_argument('--log-dir', type=str, default='./logs')
    parser.add_argument('--CUDA_VISIBLE_DEVICES', type=str, default='0')
    parser.add_argument('--fold', type=str, default='0,1,2,3,4')
    parser.add_argument('--n-meta-dim', type=str, default='512,128')

    args, _ = parser.parse_known_args()
    return args
```

Certaines de ces arguments sont obligatoires :

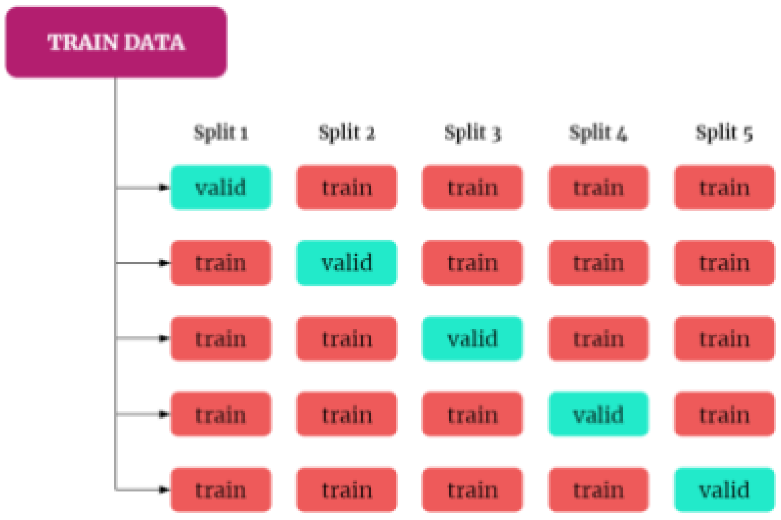
- data folder : les dossier contenant les images et le fichier csv
- enet – type : le type de réseau CNN choisie (Resnet , Effnet ou Seresnext)
- image-size : taille des images

2-1 Exécution de la classe Main



```python
def main():
    # Récupération des données
    df, df_test, meta_features, n_meta_features, mel_idx = get_df(
        args.kernel_type,
        args.out_dim,
        args.data_dir,
        args.data_folder,
        args.use_meta
    )
    # Récupérer les augmentations qui nous devons appliquer sur les images
    transforms_train, transforms_val = get_transforms(args.image_size)

    # trainer et valider notre réseau de neurone en se basant sur la méthode K- fold
    folds = [int(i) for i in args.fold.split(',')]
    for fold in folds:
        run(fold, df, meta_features, n_meta_features, transforms_train, transforms_val, mel_idx)

# la main class dans la quelle on va lire les paramètres d'entrée
if __name__ == '__main__':

    args = parse_args()
    os.makedirs(args.model_dir, exist_ok=True)
    os.makedirs(args.log_dir, exist_ok=True)
    os.environ['CUDA_VISIBLE_DEVICES'] = args.CUDA_VISIBLE_DEVICES
    # choisir le type de réseau CNN
    if args.enet_type == 'resnest101':
        ModelClass = Resnest_Melanoma
    elif args.enet_type == 'seresnext101':
        ModelClass = Seresnext_Melanoma
    elif 'efficientnet' in args.enet_type:
        ModelClass = Effnet_Melanoma
    else:
        raise NotImplementedError()

    DP = len(os.environ['CUDA_VISIBLE_DEVICES']) > 1

    set_seed()

    device = torch.device('cuda')
    criterion = nn.CrossEntropyLoss()

    main()
```

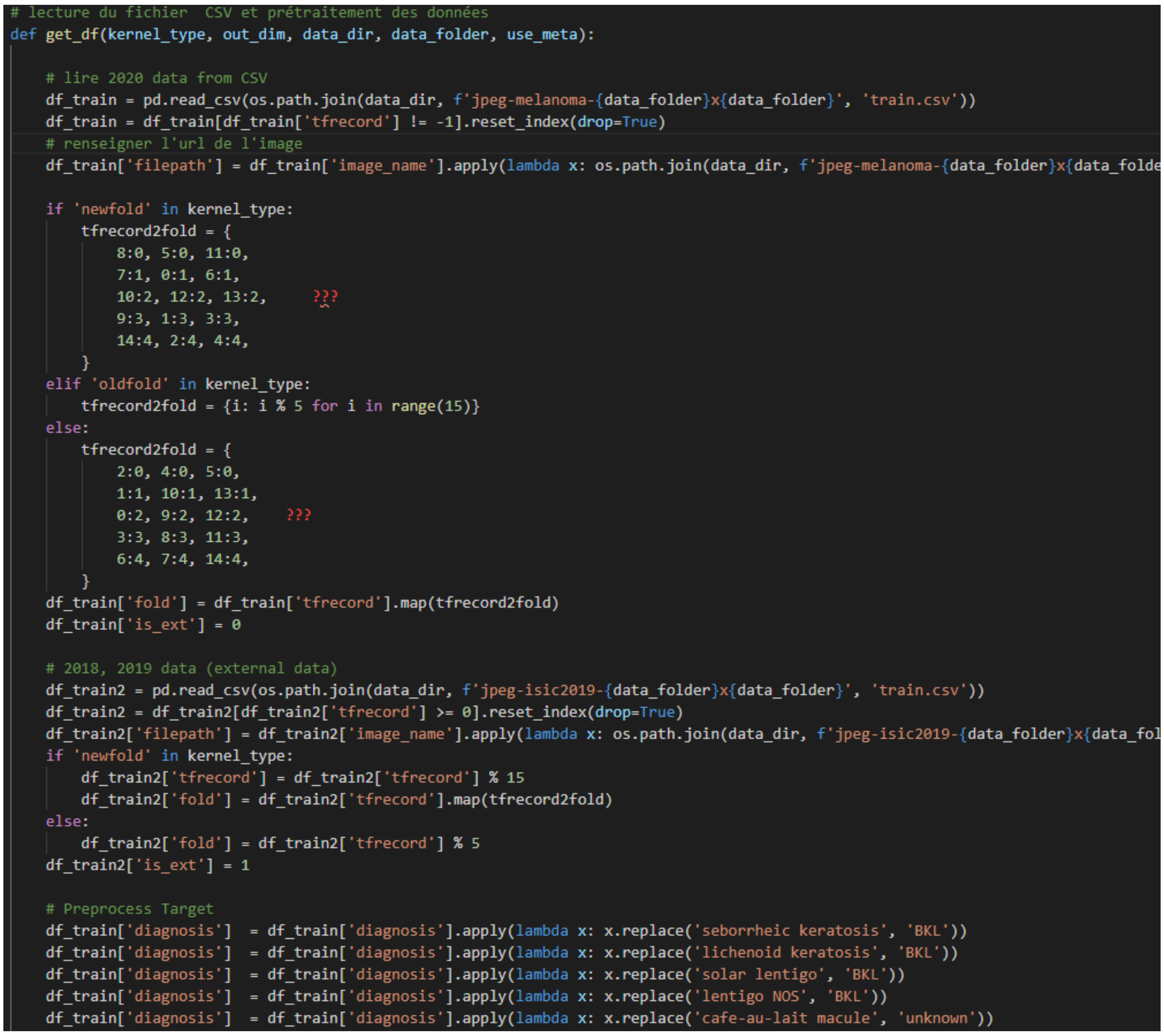
Méthode k – fold cross validation :



Cette méthode consiste à découper nos données (image + Csv) en des paquets de données, 5 paquets dans de le cas de notre réseau de neurones. Par la suite on va utiliser ces paquets de données pour entrainer et valider notre modèle sur 5 itérations différentes. A chaque itération n, le paquet de données utilisé pour valider le modèle doit être différent de celui utilisé à l'itération n-1. Le but de cette méthode est de donner plus d'échantillon de test à notre réseaux de neuronnes et par conséquent améliorer la précision.

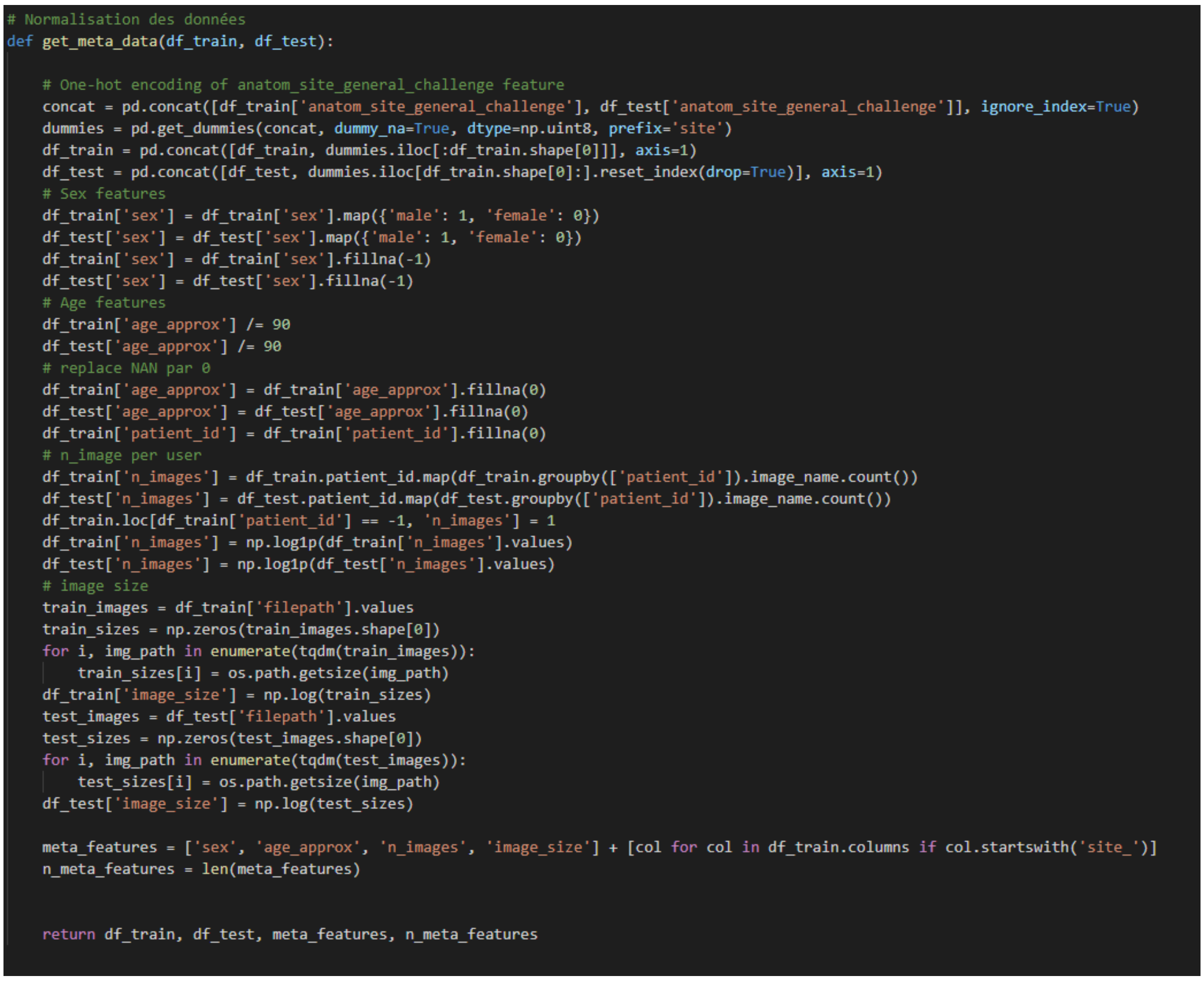
3- Préparation et récupération des donnée

3-1 Lecture de données



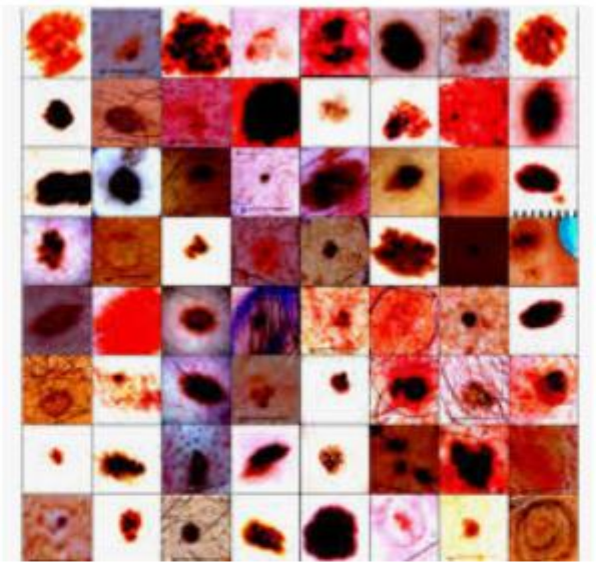
```python
# lecture du fichier  CSV et prétraitement des données
def get_df(kernel_type, out_dim, data_dir, data_folder, use_meta):

    # lire 2020 data from CSV
    df_train = pd.read_csv(os.path.join(data_dir, f'jpeg-melanoma-{data_folder}x{data_folder}', 'train.csv'))
    df_train = df_train[df_train['tfrecord'] != -1].reset_index(drop=True)
    # renseigner l'url de l'image
    df_train['filepath'] = df_train['image_name'].apply(lambda x: os.path.join(data_dir, f'jpeg-melanoma-{data_folder}x{data_folde

    if 'newfold' in kernel_type:
        tfrecord2fold = {
            8:0, 5:0, 11:0,
            7:1, 0:1, 6:1,
            10:2, 12:2, 13:2,      ???
            9:3, 1:3, 3:3,
            14:4, 2:4, 4:4,
        }
    elif 'oldfold' in kernel_type:
        tfrecord2fold = {i: i % 5 for i in range(15)}
    else:
        tfrecord2fold = {
            2:0, 4:0, 5:0,
            1:1, 10:1, 13:1,
            0:2, 9:2, 12:2,     ???
            3:3, 8:3, 11:3,
            6:4, 7:4, 14:4,
        }
    df_train['fold'] = df_train['tfrecord'].map(tfrecord2fold)
    df_train['is_ext'] = 0

    # 2018, 2019 data (external data)
    df_train2 = pd.read_csv(os.path.join(data_dir, f'jpeg-isic2019-{data_folder}x{data_folder}', 'train.csv'))
    df_train2 = df_train2[df_train2['tfrecord'] >= 0].reset_index(drop=True)
    df_train2['filepath'] = df_train2['image_name'].apply(lambda x: os.path.join(data_dir, f'jpeg-isic2019-{data_folder}x{data_fol
    if 'newfold' in kernel_type:
        df_train2['tfrecord'] = df_train2['tfrecord'] % 15
        df_train2['fold'] = df_train2['tfrecord'].map(tfrecord2fold)
    else:
        df_train2['fold'] = df_train2['tfrecord'] % 5
    df_train2['is_ext'] = 1

    # Preprocess Target
    df_train['diagnosis']  = df_train['diagnosis'].apply(lambda x: x.replace('seborrheic keratosis', 'BKL'))
    df_train['diagnosis']  = df_train['diagnosis'].apply(lambda x: x.replace('lichenoid keratosis', 'BKL'))
    df_train['diagnosis']  = df_train['diagnosis'].apply(lambda x: x.replace('solar lentigo', 'BKL'))
    df_train['diagnosis']  = df_train['diagnosis'].apply(lambda x: x.replace('lentigo NOS', 'BKL'))
    df_train['diagnosis']  = df_train['diagnosis'].apply(lambda x: x.replace('cafe-au-lait macule', 'unknown'))
```

3-1 Normalisation des données

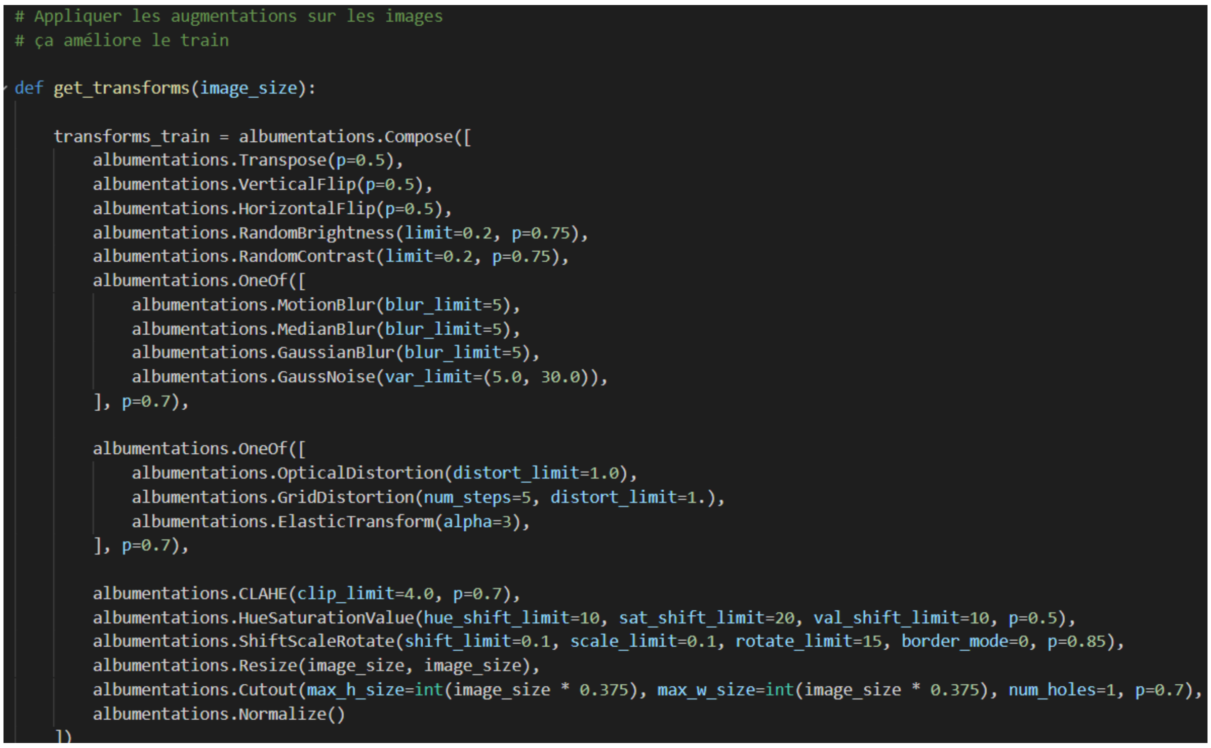


```python
# Normalisation des données
def get_meta_data(df_train, df_test):

    # One-hot encoding of anatom_site_general_challenge feature
    concat = pd.concat([df_train['anatom_site_general_challenge'], df_test['anatom_site_general_challenge']], ignore_index=True)
    dummies = pd.get_dummies(concat, dummy_na=True, dtype=np.uint8, prefix='site')
    df_train = pd.concat([df_train, dummies.iloc[:df_train.shape[0]]], axis=1)
    df_test = pd.concat([df_test, dummies.iloc[df_train.shape[0]:].reset_index(drop=True)], axis=1)
    # Sex features
    df_train['sex'] = df_train['sex'].map({'male': 1, 'female': 0})
    df_test['sex'] = df_test['sex'].map({'male': 1, 'female': 0})
    df_train['sex'] = df_train['sex'].fillna(-1)
    df_test['sex'] = df_test['sex'].fillna(-1)
    # Age features
    df_train['age_approx'] /= 90
    df_test['age_approx'] /= 90
    # replace NAN par 0
    df_train['age_approx'] = df_train['age_approx'].fillna(0)
    df_test['age_approx'] = df_test['age_approx'].fillna(0)
    df_train['patient_id'] = df_train['patient_id'].fillna(0)
    # n_image per user
    df_train['n_images'] = df_train.patient_id.map(df_train.groupby(['patient_id']).image_name.count())
    df_test['n_images'] = df_test.patient_id.map(df_test.groupby(['patient_id']).image_name.count())
    df_train.loc[df_train['patient_id'] == -1, 'n_images'] = 1
    df_train['n_images'] = np.log1p(df_train['n_images'].values)
    df_test['n_images'] = np.log1p(df_test['n_images'].values)
    # image size
    train_images = df_train['filepath'].values
    train_sizes = np.zeros(train_images.shape[0])
    for i, img_path in enumerate(tqdm(train_images)):
        train_sizes[i] = os.path.getsize(img_path)
    df_train['image_size'] = np.log(train_sizes)
    test_images = df_test['filepath'].values
    test_sizes = np.zeros(test_images.shape[0])
    for i, img_path in enumerate(tqdm(test_images)):
        test_sizes[i] = os.path.getsize(img_path)
    df_test['image_size'] = np.log(test_sizes)

    meta_features = ['sex', 'age_approx', 'n_images', 'image_size'] + [col for col in df_train.columns if col.startswith('site_')]
    n_meta_features = len(meta_features)


    return df_train, df_test, meta_features, n_meta_features
```

3-4 Les augmentations sur les images

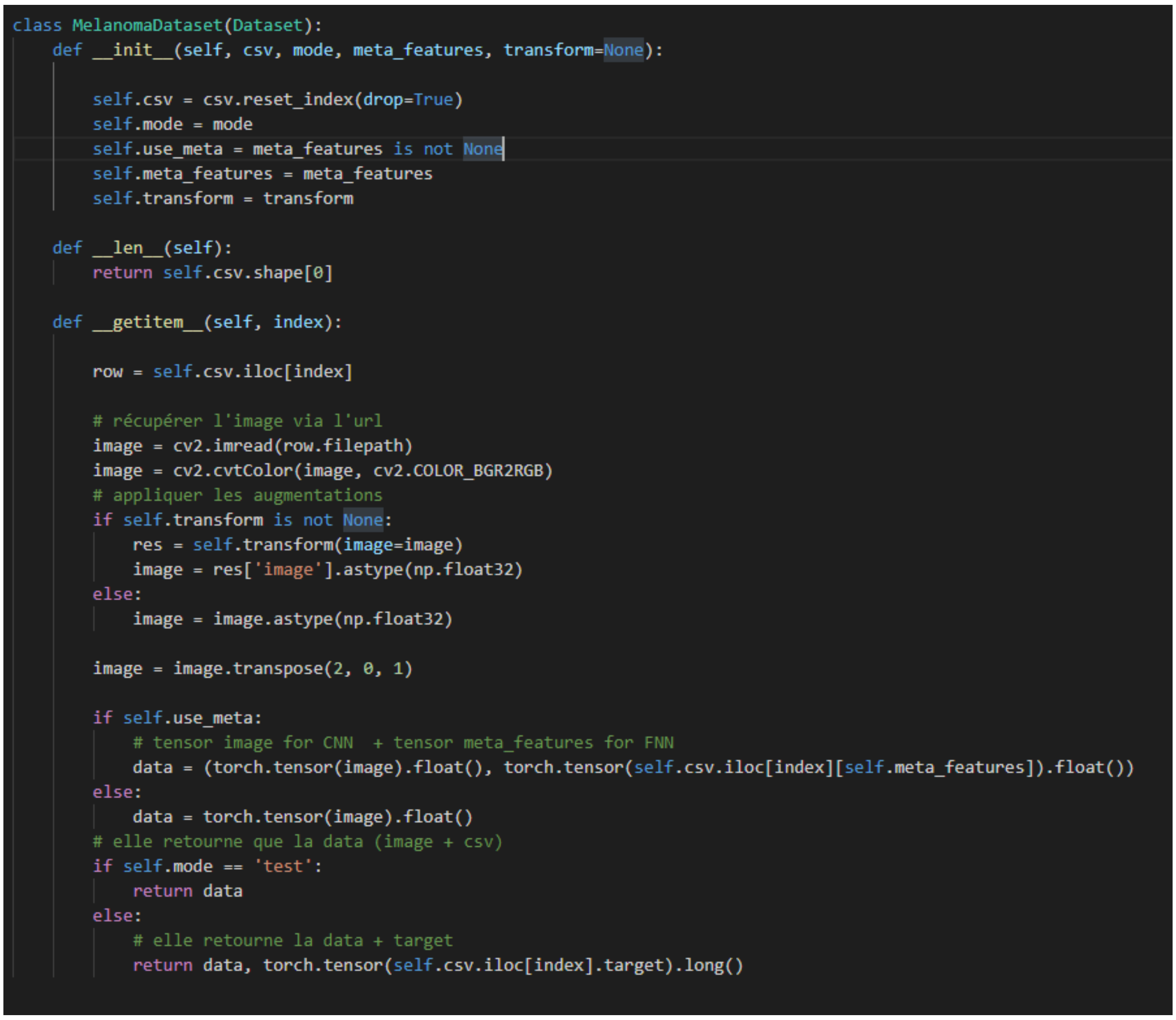


Les augmentations permettent de générer des nouvelles images à partir de celles disponibles, elles se réalisent en effectuant des opérations modifiant l'aspect de l'image sans pour autant modifier la sémantique par exemple en augmentant la luminosité ou en effectuant une rotation.



```python
# Appliquer les augmentations sur les images
# ça améliore le train

def get_transforms(image_size):

    transforms_train = albumentations.Compose([
        albumentations.Transpose(p=0.5),
        albumentations.VerticalFlip(p=0.5),
        albumentations.HorizontalFlip(p=0.5),
        albumentations.RandomBrightness(limit=0.2, p=0.75),
        albumentations.RandomContrast(limit=0.2, p=0.75),
        albumentations.OneOf([
            albumentations.MotionBlur(blur_limit=5),
            albumentations.MedianBlur(blur_limit=5),
            albumentations.GaussianBlur(blur_limit=5),
            albumentations.GaussNoise(var_limit=(5.0, 30.0)),
        ], p=0.7),

        albumentations.OneOf([
            albumentations.OpticalDistortion(distort_limit=1.0),
            albumentations.GridDistortion(num_steps=5, distort_limit=1.),
            albumentations.ElasticTransform(alpha=3),
        ], p=0.7),

        albumentations.CLAHE(clip_limit=4.0, p=0.7),
        albumentations.HueSaturationValue(hue_shift_limit=10, sat_shift_limit=20, val_shift_limit=10, p=0.5),
        albumentations.ShiftScaleRotate(shift_limit=0.1, scale_limit=0.1, rotate_limit=15, border_mode=0, p=0.85),
        albumentations.Resize(image_size, image_size),
        albumentations.Cutout(max_h_size=int(image_size * 0.375), max_w_size=int(image_size * 0.375), num_holes=1, p=0.7),
        albumentations.Normalize()
    ])
```

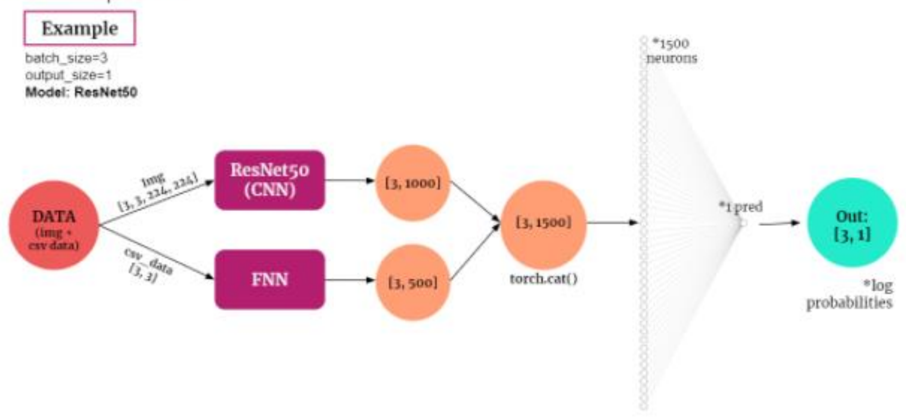
3-4 Récupération des données (get_Item)

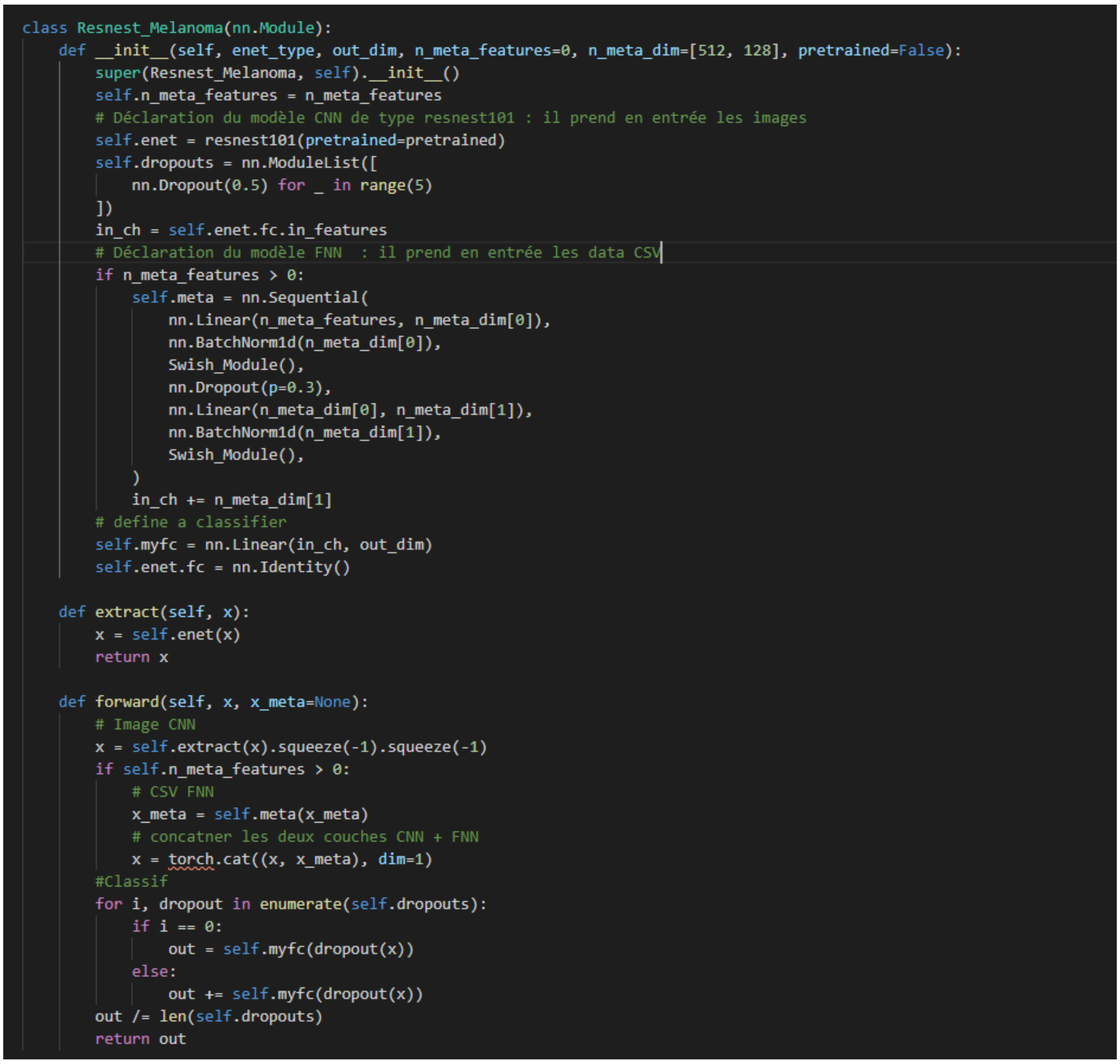


```python
class MelanomaDataset(Dataset):
    def __init__(self, csv, mode, meta_features, transform=None):

        self.csv = csv.reset_index(drop=True)
        self.mode = mode
        self.use_meta = meta_features is not None
        self.meta_features = meta_features
        self.transform = transform

    def __len__(self):
        return self.csv.shape[0]

    def __getitem__(self, index):

        row = self.csv.iloc[index]

        # récupérer l'image via l'url
        image = cv2.imread(row.filepath)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
        # appliquer les augmentations
        if self.transform is not None:
            res = self.transform(image=image)
            image = res['image'].astype(np.float32)
        else:
            image = image.astype(np.float32)

        image = image.transpose(2, 0, 1)

        if self.use_meta:
            # tensor image for CNN  + tensor meta_features for FNN
            data = (torch.tensor(image).float(), torch.tensor(self.csv.iloc[index][self.meta_features]).float())
        else:
            data = torch.tensor(image).float()
        # elle retourne que la data (image + csv)
        if self.mode == 'test':
            return data
        else:
            # elle retourne la data + target
            return data, torch.tensor(self.csv.iloc[index].target).long()
```

4- Réseau de neurones (définition du modèle)

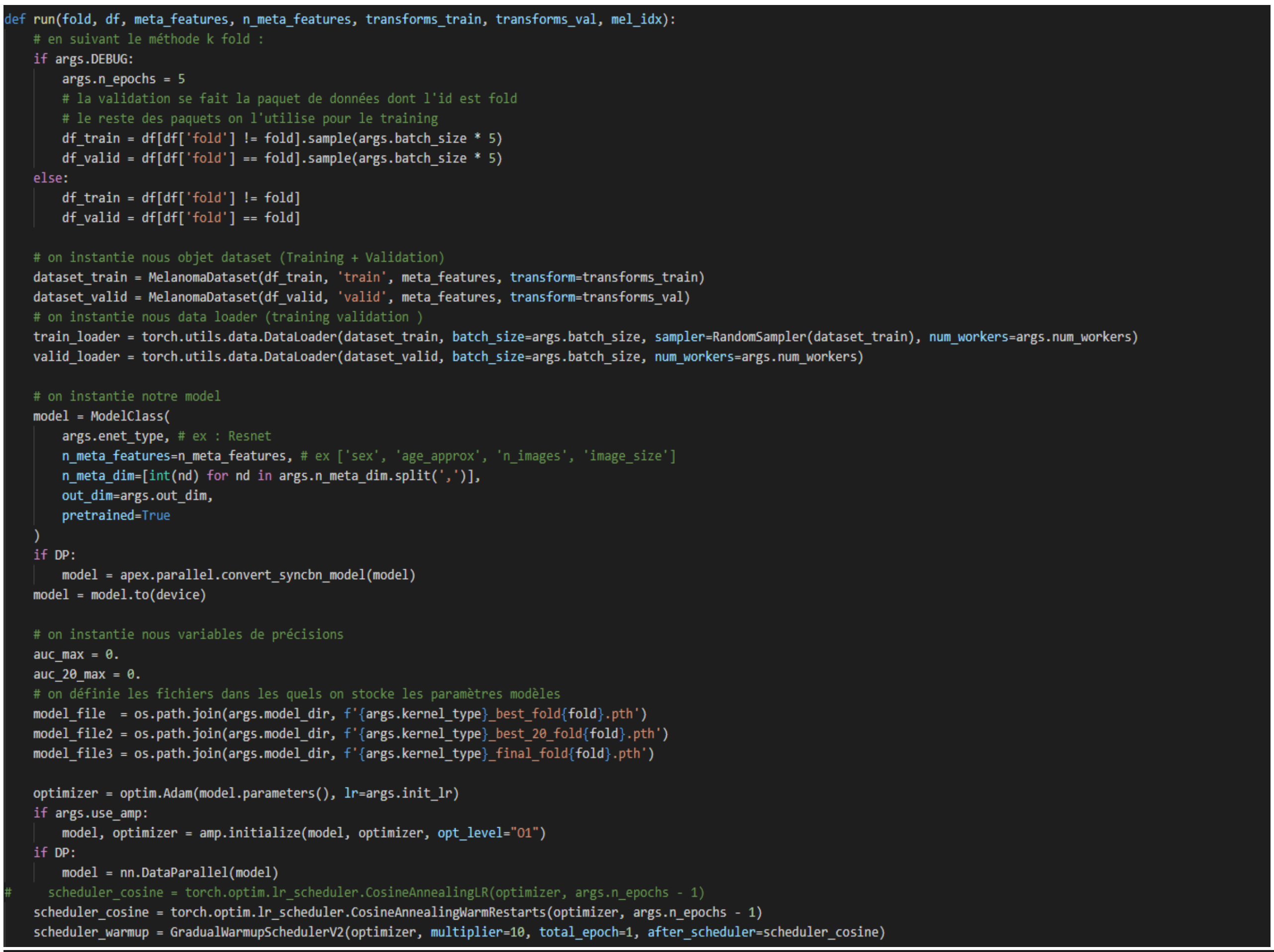
A schema of the example below:



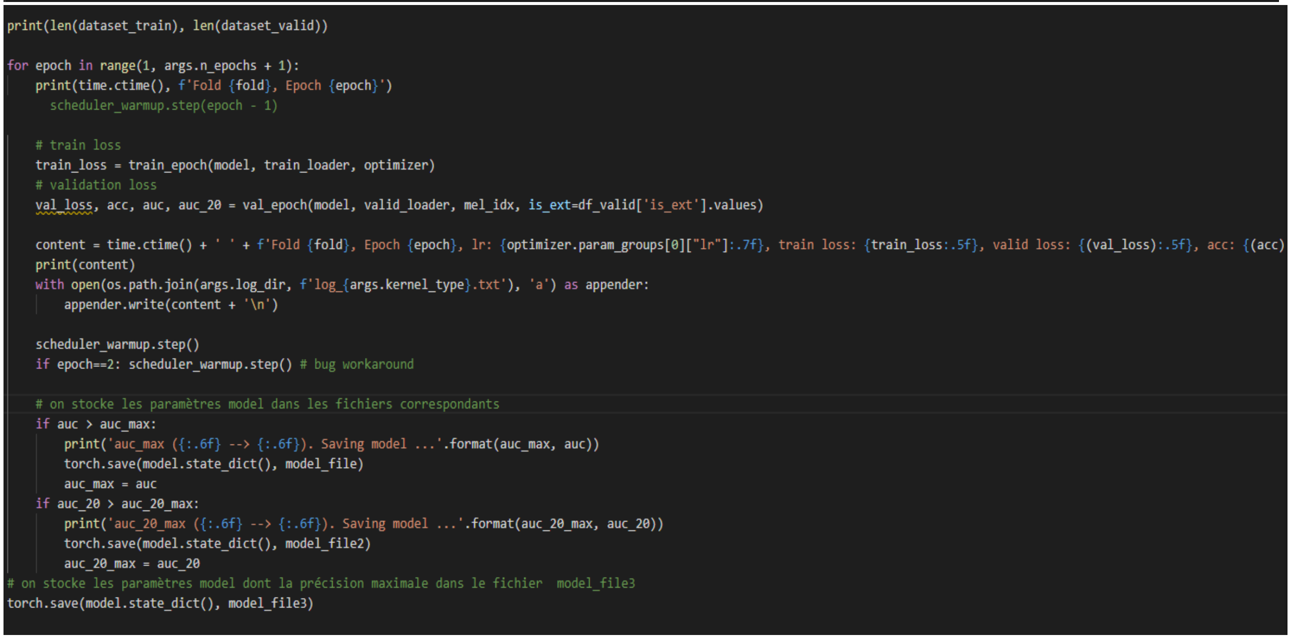


```python
class Resnest_Melanoma(nn.Module):
    def __init__(self, enet_type, out_dim, n_meta_features=0, n_meta_dim=[512, 128], pretrained=False):
        super(Resnest_Melanoma, self).__init__()
        self.n_meta_features = n_meta_features
        # Déclaration du modèle CNN de type resnest101 : il prend en entrée les images
        self.enet = resnest101(pretrained=pretrained)
        self.dropouts = nn.ModuleList([
            nn.Dropout(0.5) for _ in range(5)
        ])
        in_ch = self.enet.fc.in_features
        # Déclaration du modèle FNN  : il prend en entrée les data CSV
        if n_meta_features > 0:
            self.meta = nn.Sequential(
                nn.Linear(n_meta_features, n_meta_dim[0]),
                nn.BatchNorm1d(n_meta_dim[0]),
                Swish_Module(),
                nn.Dropout(p=0.3),
                nn.Linear(n_meta_dim[0], n_meta_dim[1]),
                nn.BatchNorm1d(n_meta_dim[1]),
                Swish_Module(),
            )
            in_ch += n_meta_dim[1]
        # define a classifier
        self.myfc = nn.Linear(in_ch, out_dim)
        self.enet.fc = nn.Identity()

    def extract(self, x):
        x = self.enet(x)
        return x

    def forward(self, x, x_meta=None):
        # Image CNN
        x = self.extract(x).squeeze(-1).squeeze(-1)
        if self.n_meta_features > 0:
            # CSV FNN
            x_meta = self.meta(x_meta)
            # concatner les deux couches CNN + FNN
            x = torch.cat((x, x_meta), dim=1)
        #Classif
        for i, dropout in enumerate(self.dropouts):
            if i == 0:
                out = self.myfc(dropout(x))
            else:
                out += self.myfc(dropout(x))
        out /= len(self.dropouts)
        return out
```

5- Training et validation pour un fold

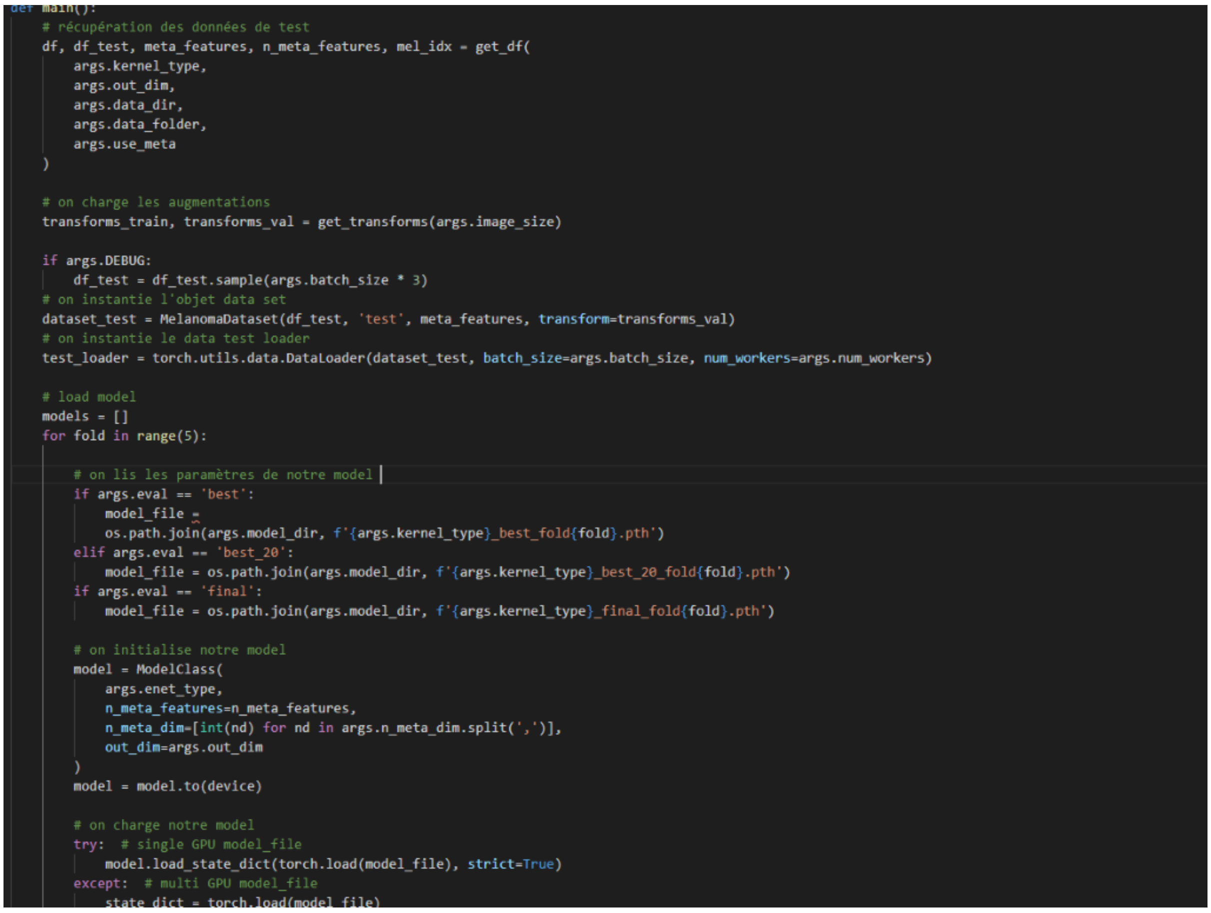


```python
def run(fold, df, meta_features, n_meta_features, transforms_train, transforms_val, mel_idx):
    # en suivant le méthode k fold :
    if args.DEBUG:
        args.n_epochs = 5
        # la validation se fait la paquet de données dont l'id est fold
        # le reste des paquets on l'utilise pour le training
        df_train = df[df['fold'] != fold].sample(args.batch_size * 5)
        df_valid = df[df['fold'] == fold].sample(args.batch_size * 5)
    else:
        df_train = df[df['fold'] != fold]
        df_valid = df[df['fold'] == fold]

    # on instantie nous objet dataset (Training + Validation)
    dataset_train = MelanomaDataset(df_train, 'train', meta_features, transform=transforms_train)
    dataset_valid = MelanomaDataset(df_valid, 'valid', meta_features, transform=transforms_val)
    # on instantie nous data loader (training validation )
    train_loader = torch.utils.data.DataLoader(dataset_train, batch_size=args.batch_size, sampler=RandomSampler(dataset_train), num_workers=args.num_workers)
    valid_loader = torch.utils.data.DataLoader(dataset_valid, batch_size=args.batch_size, num_workers=args.num_workers)

    # on instantie notre model
    model = ModelClass(
        args.enet_type, # ex : Resnet
        n_meta_features=n_meta_features, # ex ['sex', 'age_approx', 'n_images', 'image_size']
        n_meta_dim=[int(nd) for nd in args.n_meta_dim.split(',')],
        out_dim=args.out_dim,
        pretrained=True
    )
    if DP:
        model = apex.parallel.convert_syncbn_model(model)
    model = model.to(device)

    # on instantie nous variables de précisions
    auc_max = 0.
    auc_20_max = 0.
    # on définie les fichiers dans les quels on stocke les paramètres modèles
    model_file  = os.path.join(args.model_dir, f'{args.kernel_type}_best_fold{fold}.pth')
    model_file2 = os.path.join(args.model_dir, f'{args.kernel_type}_best_20_fold{fold}.pth')
    model_file3 = os.path.join(args.model_dir, f'{args.kernel_type}_final_fold{fold}.pth')

    optimizer = optim.Adam(model.parameters(), lr=args.init_lr)
    if args.use_amp:
        model, optimizer = amp.initialize(model, optimizer, opt_level="O1")
    if DP:
        model = nn.DataParallel(model)
#     scheduler_cosine = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, args.n_epochs - 1)
    scheduler_cosine = torch.optim.lr_scheduler.CosineAnnealingWarmRestarts(optimizer, args.n_epochs - 1)
    scheduler_warmup = GradualWarmupSchedulerV2(optimizer, multiplier=10, total_epoch=1, after_scheduler=scheduler_cosine)
```

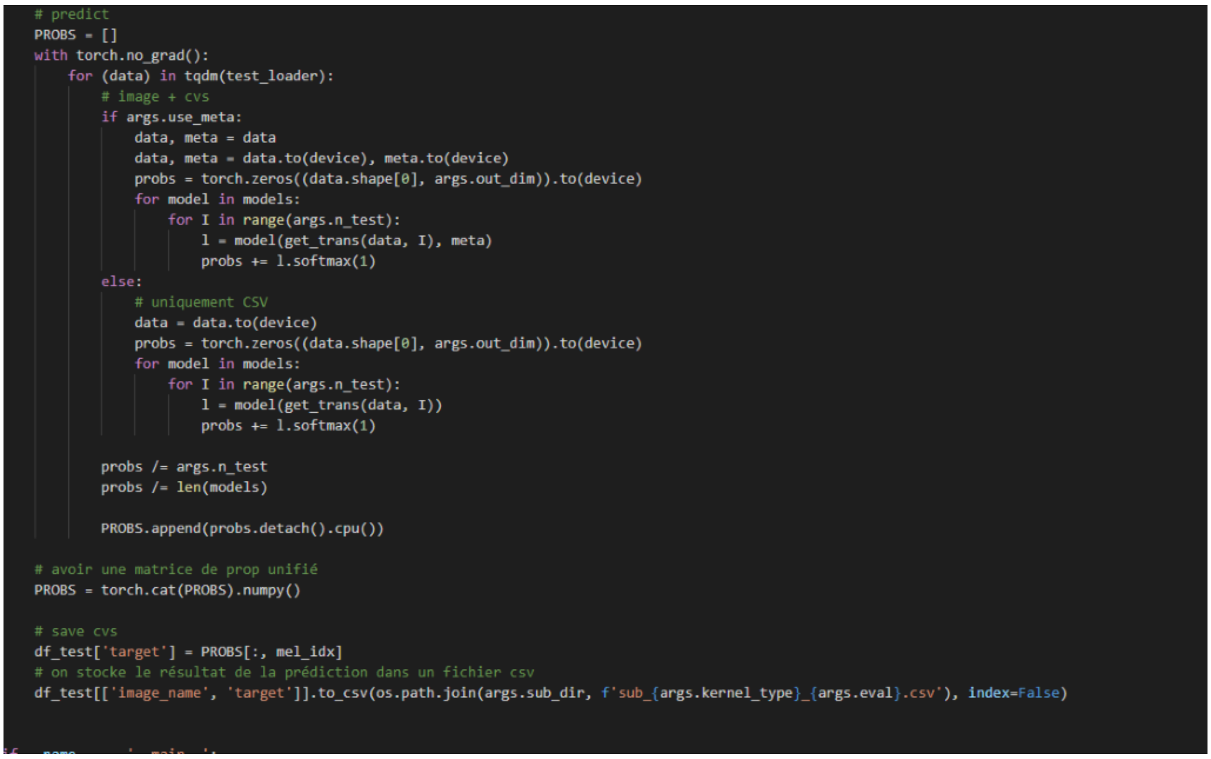


```python
print(len(dataset_train), len(dataset_valid))

for epoch in range(1, args.n_epochs + 1):
    print(time.ctime(), f'Fold {fold}, Epoch {epoch}')
      scheduler_warmup.step(epoch - 1)

    # train loss
    train_loss = train_epoch(model, train_loader, optimizer)
    # validation loss
    val_loss, acc, auc, auc_20 = val_epoch(model, valid_loader, mel_idx, is_ext=df_valid['is_ext'].values)

    content = time.ctime() + ' ' + f'Fold {fold}, Epoch {epoch}, lr: {optimizer.param_groups[0]["lr"]:.7f}, train loss: {train_loss:.5f}, valid loss: {(val_loss):.5f}, acc: {(acc)
    print(content)
    with open(os.path.join(args.log_dir, f'log_{args.kernel_type}.txt'), 'a') as appender:
        appender.write(content + '\n')

    scheduler_warmup.step()
    if epoch==2: scheduler_warmup.step() # bug workaround

    # on stocke les paramètres model dans les fichiers correspondants
    if auc > auc_max:
        print('auc_max ({:.6f} --> {:.6f}). Saving model ...'.format(auc_max, auc))
        torch.save(model.state_dict(), model_file)
        auc_max = auc
    if auc_20 > auc_20_max:
        print('auc_20_max ({:.6f} --> {:.6f}). Saving model ...'.format(auc_20_max, auc_20))
        torch.save(model.state_dict(), model_file2)
        auc_20_max = auc_20
# on stocke les paramètres model dont la précision maximale dans le fichier  model_file3
torch.save(model.state_dict(), model_file3)
```

6- Prédiction



```
def main():
    # récupération des données de test
    df, df_test, meta_features, n_meta_features, mel_idx = get_df(
        args.kernel_type,
        args.out_dim,
        args.data_dir,
        args.data_folder,
        args.use_meta
    )

    # on charge les augmentations
    transforms_train, transforms_val = get_transforms(args.image_size)

    if args.DEBUG:
        df_test = df_test.sample(args.batch_size * 3)
    # on instantie l'objet data set
    dataset_test = MelanomaDataset(df_test, 'test', meta_features, transform=transforms_val)
    # on instantie le data test loader
    test_loader = torch.utils.data.DataLoader(dataset_test, batch_size=args.batch_size, num_workers=args.num_workers)

    # load model
    models = []
    for fold in range(5):

        # on lis les paramètres de notre model
        if args.eval == 'best':
            model_file =
            os.path.join(args.model_dir, f'{args.kernel_type}_best_fold{fold}.pth')
        elif args.eval == 'best_20':
            model_file = os.path.join(args.model_dir, f'{args.kernel_type}_best_20_fold{fold}.pth')
        if args.eval == 'final':
            model_file = os.path.join(args.model_dir, f'{args.kernel_type}_final_fold{fold}.pth')

        # on initialise notre model
        model = ModelClass(
            args.enet_type,
            n_meta_features=n_meta_features,
            n_meta_dim=[int(nd) for nd in args.n_meta_dim.split(',')],
            out_dim=args.out_dim
        )
        model = model.to(device)

        # on charge notre model
        try:  # single GPU model_file
            model.load_state_dict(torch.load(model_file), strict=True)
        except:  # multi GPU model_file
            state_dict = torch.load(model_file)
```



```python
    # predict
    PROBS = []
    with torch.no_grad():
        for (data) in tqdm(test_loader):
            # image + cvs
            if args.use_meta:
                data, meta = data
                data, meta = data.to(device), meta.to(device)
                probs = torch.zeros((data.shape[0], args.out_dim)).to(device)
                for model in models:
                    for I in range(args.n_test):
                        l = model(get_trans(data, I), meta)
                        probs += l.softmax(1)
            else:
                # uniquement CSV
                data = data.to(device)
                probs = torch.zeros((data.shape[0], args.out_dim)).to(device)
                for model in models:
                    for I in range(args.n_test):
                        l = model(get_trans(data, I))
                        probs += l.softmax(1)

            probs /= args.n_test
            probs /= len(models)

            PROBS.append(probs.detach().cpu())

    # avoir une matrice de prop unifié
    PROBS = torch.cat(PROBS).numpy()

    # save cvs
    df_test['target'] = PROBS[:, mel_idx]
    # on stocke le résultat de la prédiction dans un fichier csv
    df_test[['image_name', 'target']].to_csv(os.path.join(args.sub_dir, f'sub_{args.kernel_type}_{args.eval}.csv'), index=False)
```