

Classification d'images (Deep Learning)

Noaga Ferdinand DAHANI, Ahmed Salem HABIBI, Daniel ASHRAFUL

Université Paris Cité, UFR Mathématiques Informatique

December 17, 2025

Tables des matières

1 Introduction & Configuration de pipeline

- Exploration des données (Statistiques descriptives)

2 Modèles Pré-entraînés

- ResNet
- EfficientNet

3 Data Augmentation

4 Sélection des hyperparamètres

5 Grad-CAM

- Notre modèle
- Resnet18
- EfficientB0
- Resnet50

6 Conclusion

Statistiques descriptives

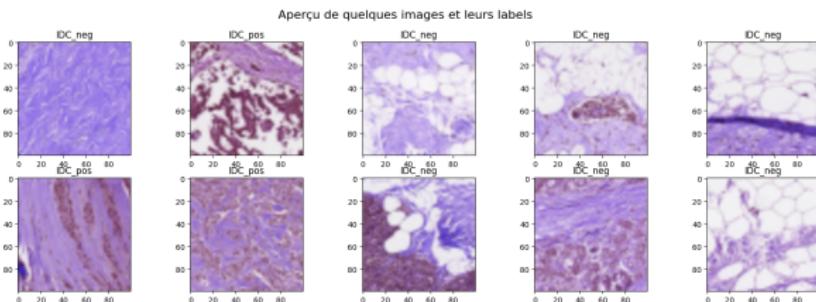
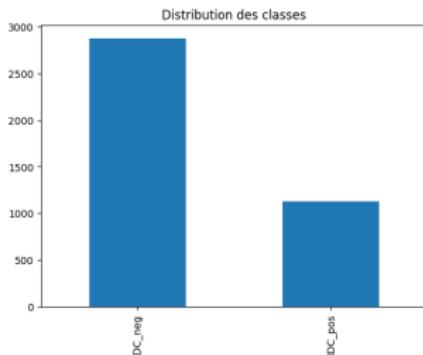
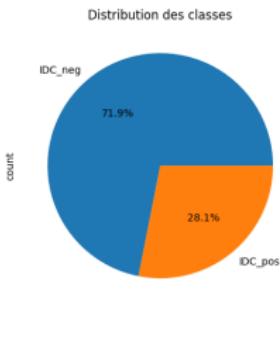


Figure: Aperçu de quelques images et leurs labels



Modèle sans Batchnormalisation

```
class CNN1(nn.Module):
def __init__(self, input_channel):
super().__init__()
self.conv1=nn.Conv2d(input_channel, 32, kernel_size=3, stride=1, padding=1)
self.pool1=nn.MaxPool2d(2, 2)

self.conv2=nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
self.pool2=nn.MaxPool2d(2, 2)
self.dropout1=nn.Dropout(0.25)

self.conv3=nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1)
self.pool3=nn.MaxPool2d(2, 2)

self.conv4=nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1)
self.pool4=nn.MaxPool2d(2, 2)
self.dropout2=nn.Dropout(0.25)
```



```
self.fc2=nn.Linear(256,2)#car on a 2 classes
def forward(self,x):
x=self.pool1(nn.ReLU()(self.conv1(x)))
x=self.pool2(nn.ReLU()(self.conv2(x)))
x=self.dropout1(x)
x=self.pool3(nn.ReLU()(self.conv3(x)))
x=self.pool4(nn.ReLU()(self.conv4(x)))
x=self.dropout2(x)
x=nn.Flatten()(x)
x=nn.ReLU()(self.fc1(x))
x=self.dropout3(x)

x=self.fc2(x)
return x
```

Modèle avec normalisation

```
class CNN2(nn.Module):
    def __init__(self, input_channel):
        super().__init__()

        self.conv1=nn.Conv2d(input_channel,32,kernel_size=3,stride=1,padding=1)
        self.bn1=nn.BatchNorm2d(32)
        self.pool1=nn.MaxPool2d(2,2)

        self.conv2=nn.Conv2d(32,64,kernel_size=3,stride=1,padding=1)
        self.bn2=nn.BatchNorm2d(64)
        self.pool2=nn.MaxPool2d(2,2)
        self.dropout1=nn.Dropout(0.25)

        self.conv3=nn.Conv2d(64,128,kernel_size=3,stride=1,padding=1)
        self.bn3=nn.BatchNorm2d(128)
        self.pool3=nn.MaxPool2d(2,2)
```



```
class CNN2(nn.Module):
def __init__(self, input_channel):
super().__init__()
self.conv1=nn.Conv2d(input_channel,32,kernel_size=3,stride=1,padding=1)
self.bn1=nn.BatchNorm2d(32)
self.pool1=nn.MaxPool2d(2,2)

self.conv2=nn.Conv2d(32,64,kernel_size=3,stride=1,padding=1)
self.bn2=nn.BatchNorm2d(64)
self.pool2=nn.MaxPool2d(2,2)
self.dropout1=nn.Dropout(0.25)

self.conv3=nn.Conv2d(64,128,kernel_size=3,stride=1,padding=1)
self.bn3=nn.BatchNorm2d(128)
self.pool3=nn.MaxPool2d(2,2)

self.conv4=nn.Conv2d(128,256,kernel_size=3,stride=1,padding=1)
self.bn4=nn.BatchNorm2d(256)
self.pool4=nn.MaxPool2d(2,2)
```



```
self.fc1=nn.Linear(256*6*6,256)
self.dropout3=nn.Dropout(0.5)

self.fc2=nn.Linear(256,2)#car on a 2 classes
def forward(self,x):
x=self.pool1(self.bn1(nn.ReLU()(self.conv1(x))))
x=self.pool2(self.bn2(nn.ReLU()(self.conv2(x))))
x=self.dropout1(x)
x=self.pool3(self.bn3(nn.ReLU()(self.conv3(x))))
x=self.pool4(self.bn4(nn.ReLU()(self.conv4(x))))
x=self.dropout2(x)
x=nn.Flatten()(x)
x=nn.ReLU()(self.fc1(x))
x=self.dropout3(x)

x=self.fc2(x)
return x
```



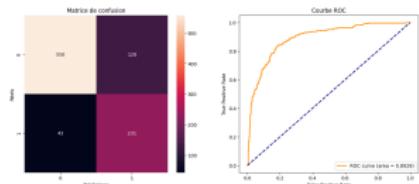


Figure: MC & courbe ROC sans Batchnormalisation

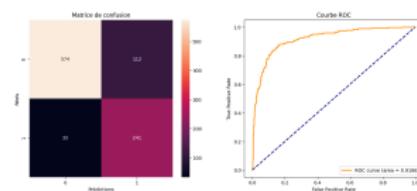


Figure: MC & courbe BatchNormalisé

Alternatives

Problème observé et objectif

Constat

- L'analyse de la matrice de confusion montre une mauvaise prédiction de la classe minoritaire.
- Le modèle est biaisé vers la classe majoritaire, phénomène typique en présence d'un déséquilibre de classes.

Limitation des données

- Il n'a pas été possible d'ajouter de nouvelles données ciblées au jeu d'entraînement.
- Les données sont tirées aléatoirement, limitant la collecte d'exemples spécifiques de la classe minoritaire.

Objectif

- Améliorer les performances sur la classe minoritaire sans modifier le dataset.

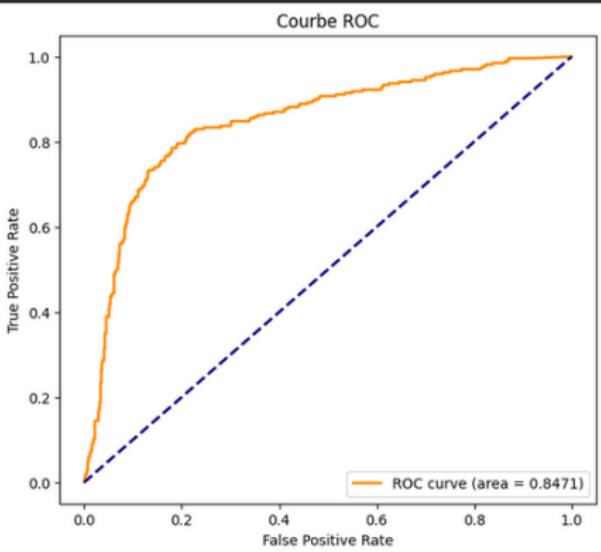
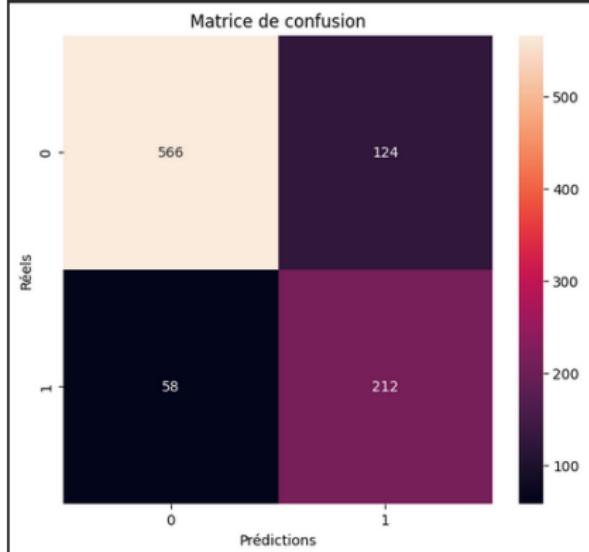
Alternative 1 : Pondération de la classe minoritaire

Principe

- Augmentation du poids de la classe minoritaire dans la fonction de perte.
- Les erreurs sur cette classe sont davantage pénalisées pendant l'apprentissage.

Alternative 1

	precision	recall	f1-score	support
0	0.9071	0.8203	0.8615	690
1	0.6310	0.7852	0.6997	270
accuracy			0.8104	960
macro avg	0.7690	0.8027	0.7806	960
weighted avg	0.8294	0.8104	0.8160	960



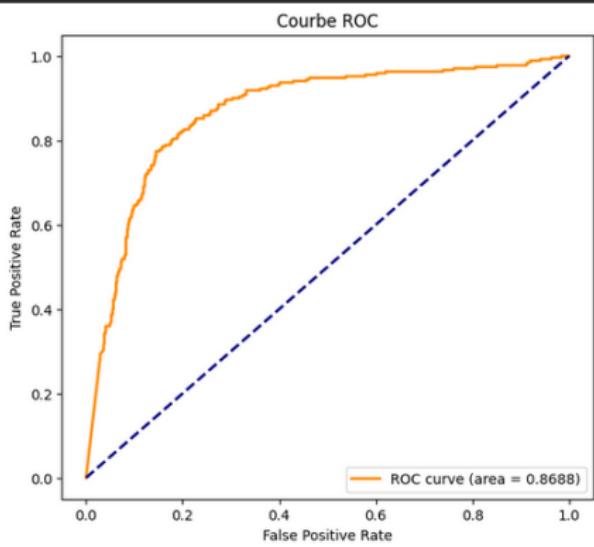
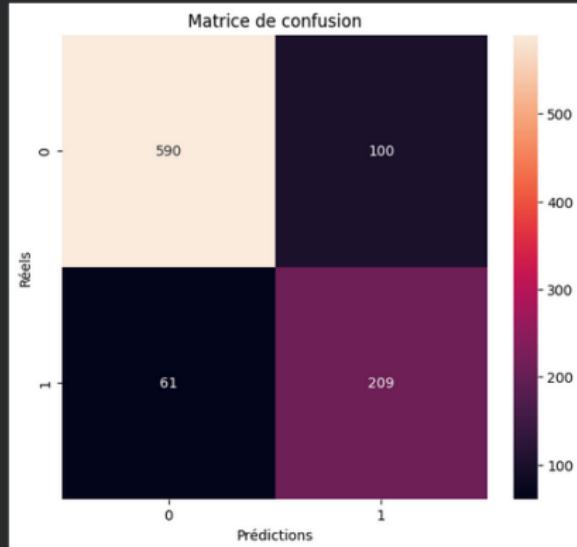
Alternative 2 : Sur-échantillonnage de la classe minoritaire

Principe

- La classe minoritaire apparaît plus fréquemment pendant l'entraînement.
- Le modèle est davantage exposé à cette classe.

Alternative 2

	precision	recall	f1-score	support
0	0.9063	0.8551	0.8799	690
1	0.6764	0.7741	0.7219	270
accuracy			0.8323	960
macro avg	0.7913	0.8146	0.8009	960
weighted avg	0.8416	0.8323	0.8355	960



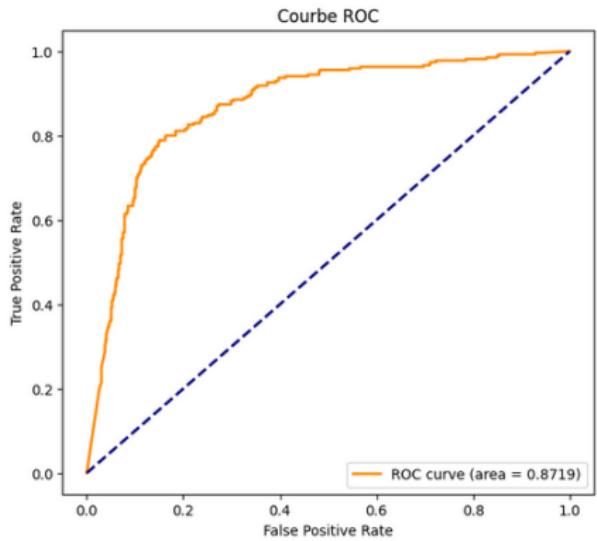
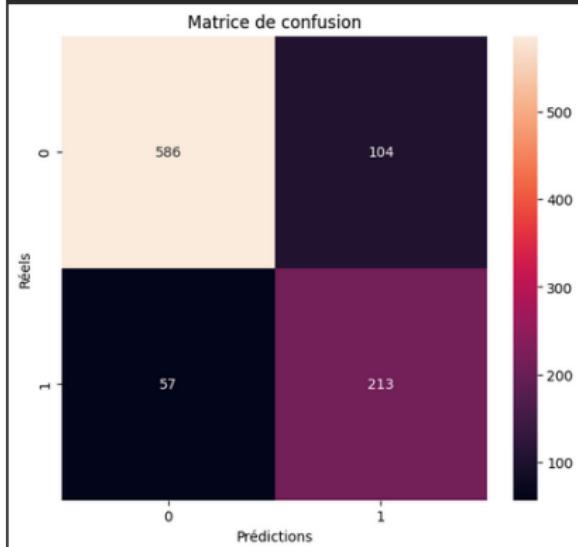
Alternative 3 : Combinaison des deux approches

Principe

- Pondération de la fonction de perte + sur-échantillonnage contrôlé.
- Objectif : réduire les faux négatifs sur la classe minoritaire.

Alternative 3

	precision	recall	f1-score	support
0	0.9114	0.8493	0.8792	690
1	0.6719	0.7889	0.7257	270
accuracy			0.8323	960
macro avg	0.7916	0.8191	0.8025	960
weighted avg	0.8440	0.8323	0.8360	960



Modèles pré entraînés

Comme son nom l'indique un modèle pré-entraîné est un modèle qui a été préalablement entraîné sur un grand nombre jeu de données pour une tâche spécifique mais à usage général. Pour ce travail nous avons utilisés des modèles issus de deux grandes familles de modèles d'apprentissage profond que sont **ResNet** et **EfficientNet**.

ResNet

Residual Network en abrégé ResNet est une architecture d'apprentissage profond développé en 2015 par des chercheurs de microsoft qui la particularité d'utiliser des connexions résiduelles ou connexions raccourcies qui permettent au modèle de sauter ou contourner certaines étapes du processus d'apprentissage c'est à dire, au lieu de forcer le modèle à faire l'information par chaque couche ces raccourcis lui permettent de transmettre plus directement les détails importants et cela évite le phénomène de disparition du gradient.

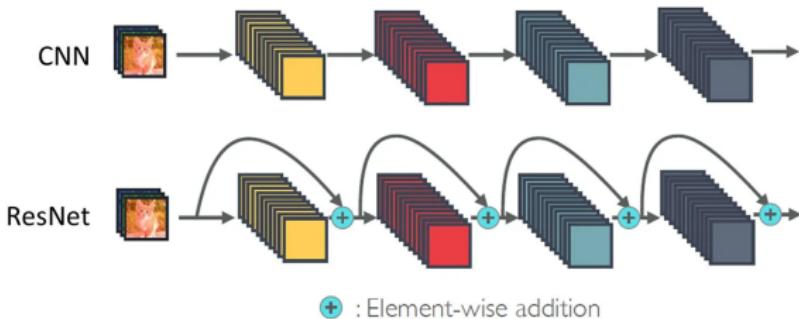


Figure: Comparaison d'architecture ResNet et CNN traditionnel

Nous avons implémenter pour notre projet **ResNet18** qui a 18 couches de convolutions et **ResNet 50** 50 couches de convolutions.

```
#importation du modèle resnet18
from torchvision import models
model_resnet18=models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)
model_resnet18.fc=nn.Linear(model_resnet18.fc.in_features ,2)
model_resnet8=model_resnet18.to(device)
#redimensions de nos images
train_transform = transforms.Compose([
transforms.Resize((224, 224)),
transforms.ToTensor(),
transforms.Normalize(
mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])])
#entraînement du modèle resnet 18
model_resnet18=train_validate(model_resnet18,train_loader_pre,val_loader)
#évaluation resnet 18
evaluate_model(model_resnet18,test_loader_pre)
#importation de resnet50
model_resnet50=models.resnet50(weights="IMAGENET1K_V1")
model_resnet50.fc=nn.Linear(model_resnet50.fc.in_features ,2)
```



```
#entraînement
model_resnet50=train_validate(model_resnet50,train_loader_pre,val_loader)
#évaluation resnet 50
evaluate_model(model_resnet50,test_loader_pre)
```

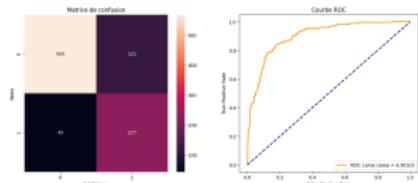


Figure: MC & courbe ROC Resnet18

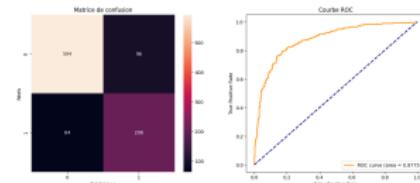


Figure: MC & courbe ROC Resnet50

EfficientNet

EfficientNet est une architecture d'apprentissage profond développée en 2019 par des chercheurs de GoogleAI. Notons que avant EfficientNet pour améliorer la performance d'un modèle on ajustait les couches ce qui implique des modèles lourds et exigeants. EfficientNet utilise l'approche **mise à l'échelle composite** qui consiste en gros à étendre la profondeur, largeur résolution tout en maintenant un certain équilibre. EfficientNet est un empilement de couches MBconv (MB pour MobileNet; convolution à goulot d'étranglements) et à l'intérieur de chaque MBconv il y a un module squeeze and excitation qui ajuste la force des différents canaux. Elle a la particularité d'utiliser une fonction d'activation *swish* qui apprend les motifs, et une régularisation stochastique.

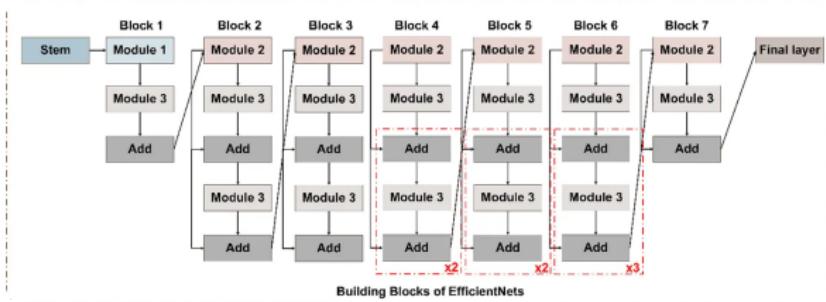
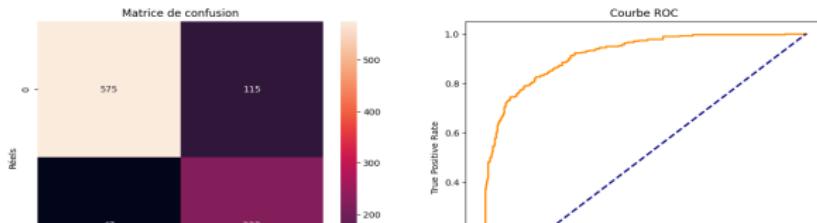


Figure: Architecture EfficientNet

Pour ce travail nous avons implémenter Efficient-B0 la base de toutes les autres versions de EfficientNet.

```
#importation de efficient-b0
model_efficientB0=models.efficientnet_b0(weights="IMAGENET1K_V1")
model_efficientB0.classifier[1]=nn.Linear(model_efficientB0.classifi
model_efficientB0=model_efficientB0.to(device)
#entraînement
modele_efficientB0=train_validate(model_efficient_B0 , train_loader_p
#évaluation
evaluate_model(modele_efficientB0 , test_loader_pre)
```

Figure: Matrice confusion & Courbe ROC Efficient-B0



Data augmentation

Data augmentation

La précision et l'efficacité du deep learning dépendent de la qualité et la quantité des données d'apprentissage, or on est confronté très souvent à une insuffisance de données afin de palier à ce problème on accroît les données artificiellement la quantité des données, en faisant des modifications(orientations,luminosité...):on diversifie l'ensemble d'apprentissage.

```
train_transform2=transforms.Compose([
transforms.Resize((224,224)),transforms.RandomHorizontalFlip(),
transforms.ColorJitter(brightness=0.2, contrast=0.2),
transforms.ToTensor(),
transforms.Normalize(
mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])])
train_dataset_pre2=CustumDataset(X_train,y_train,transform=train_tr
train_loader_aug=DataLoader(train_dataset_pre2,batch_size=32,shuffl
```

```
# Re entraînement
#resnet18
model_resnet18_aug=train_validate(model_resnet18,train_loader_aug,valid_loader)
#resnet50
model_resnet50_aug=train_validate(model_resnet50,train_loader_aug,valid_loader)
#efficient-B0
model_efficientB0_aug=train_validate(model_efficientB0,train_loader_aug,valid_loader)
#Evaluation
#resnet18
evaluate_model(model_resnet18_aug,test_loader_pre)
#resnet50
evaluate_model(model_resnet50_aug,test_loader_pre)
#efficientB0
evaluate_model(model_efficientB0_aug,test_loader_pre)
```

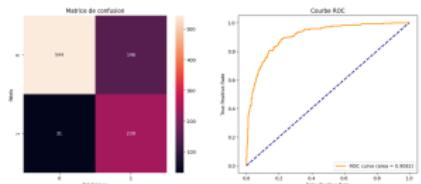


Figure: MC & courbe ROC Resnet18 après data augmentation

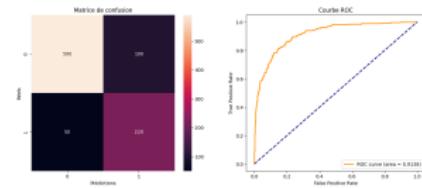
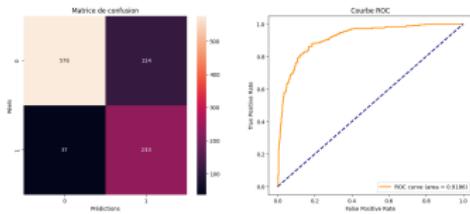


Figure: MC & courbe ROC Resnet50 après data augmentation



Sélection hyperparamètres

```
#fonction train et validate
from sklearn.metrics import recall_score, f1_score
def
    train_validate2(model,train_loader,val_loader,criterion,device,
# Choix de l'optimiseur
if opti == "adam":
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
elif opti== "sgd":
optimizer = torch.optim.SGD(model.parameters(), lr=lr,
    momentum=0.9)
else:
raise ValueError("optimizer_name_must_be_ 'adam' _or_ 'sgd' ")
best_val_loss =np.inf
patience_counter = 0
best_model =copy.deepcopy(model.state_dict())
model.to(device)
for epoch in range(n_epochs):
```



```
model.eval()
val_loss=[]
y_true,y_pred=[], []
with torch.no_grad():
for x,y in val_loader:
    images,labels=x.to(device),y.to(device)
    outputs=model(images)
    loss=criterion(outputs,labels)
    val_loss.append(loss.item())

    _,preds=torch.max(outputs,1)
    y_true.extend(labels.cpu().numpy())
    y_pred.extend(preds.cpu().numpy())
val_loss=np.mean(val_loss)
recall=recall_score(y_true,y_pred)
f1=f1_score(y_true,y_pred)
print(f"Epoch_{epoch+1}/{n_epochs}--Train_Loss:_
{train_loss:.4f}--Val_Loss:{val_loss:.4f}--recall:_
{recall:.4f}--f1:{f1:.4f}")
```



```
if val_loss < best_val_loss:  
    best_val_loss = val_loss  
    patience_counter=0  
    best_model = copy.deepcopy(model.state_dict())  
else:  
    patience_counter += 1  
if patience_counter >= patience:  
    print(f"Early_stopping_at_epoch_{epoch+1}")  
    break  
# Sauvegarder le meilleur modèle  
model.load_state_dict(best_model)  
return model,recall,f1,best_val_loss  
#selection des hyper paramètres  
optis=["adam","sgd"]  
patiences=[5,7,10]  
lrs=[0.001,0.01,0.1]  
batch_sizes=[32,64]
```

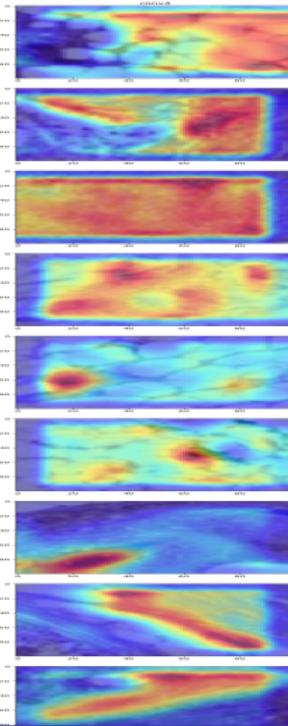
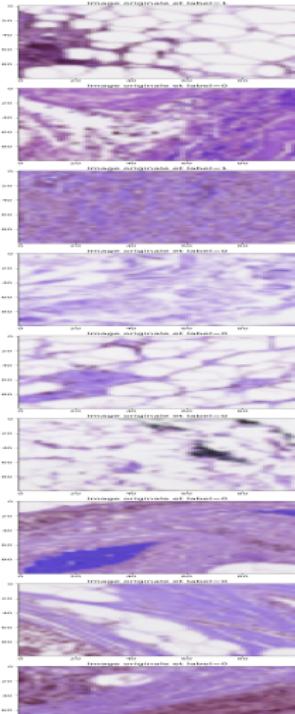
```
def selection_hyperparametres(model, train_dataset, val_dataset, optis):
    best_recall=0
    best_config=None
    resultats=[]
    for lr,opti,batch_size,patience in
        itertools.product(lrs,optis,batch_sizes,patiences):
        print(f"\nOn teste avec"
              f"\n lr={lr},opti={opti},batch_size={batch_size},patience={patience}")
        train_loader=DataLoader(train_dataset,batch_size=batch_size,shuffle=True)
        val_loader=DataLoader(val_dataset,batch_size=batch_size,shuffle=False)
        model=model_class.to(device)
        criterion=nn.CrossEntropyLoss()
        model,recall,f1,val_loss=train_validate2(model,train_loader,val_loader)
        result={"lr":lr,"opti":opti,"batch_size":batch_size,"patience":patience,
                "recall":recall,"f1":f1,"val_loss":val_loss}
        resultats.append(result)
        print(f"recall:{recall:.4f}--f1:{f1:.4f}--val_loss:{val_loss:.4f}")
    if recall>best_recall:
```



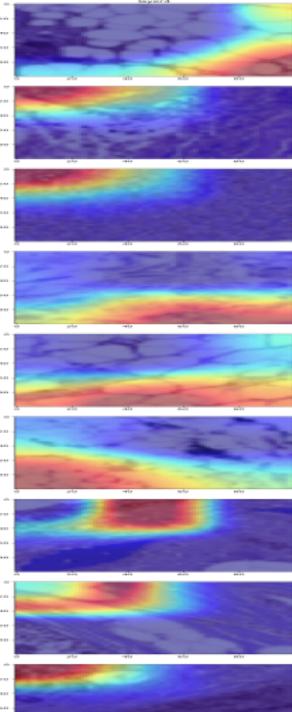
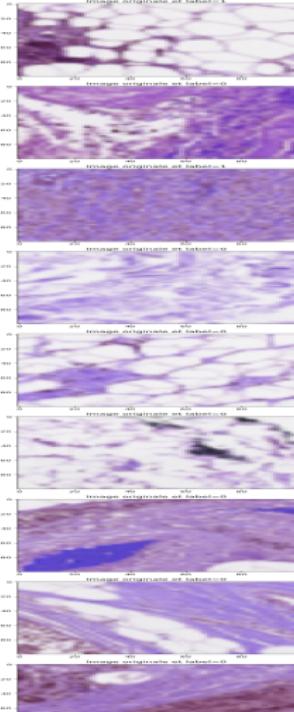
```
best_config=result
return best_config,resultats
#Applications
best_config,resultats=selection_hyperparametres(model2,train_data)
print(f"Meilleure_configuration:{best_config}")
#efficientB0
best_config,resultats=selection_hyperparametres(model_efficientB0,train_data)
print(f"Meilleure_configuration:{best_config}")
```

GRAD-CAM

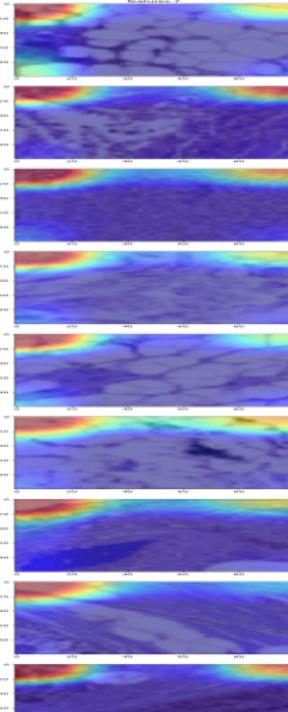
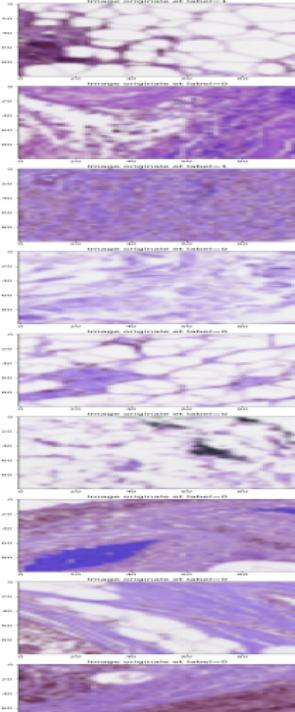
Notre modèle



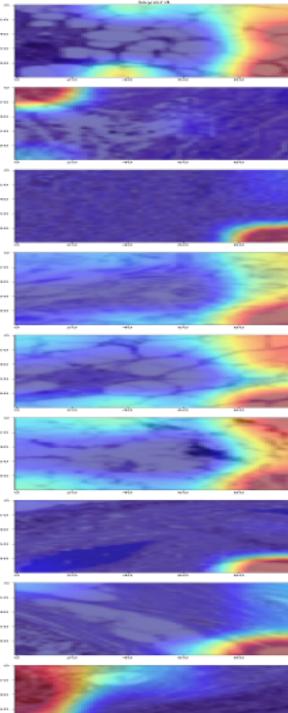
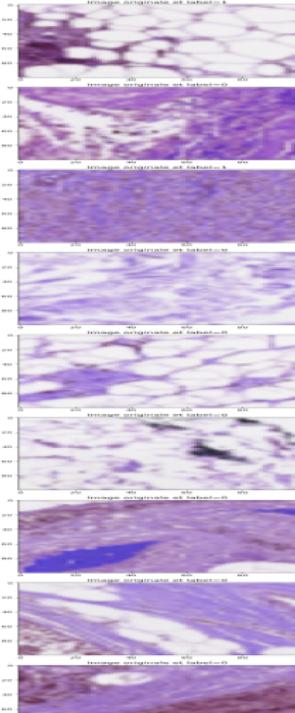
Resnet18



EfficientB0



Resnet50



Conclusion