

# Formation - Logiciels pour l'apprentissage automatique

## Cours 2 : Les bases de PyTorch Lightning

ADDAD Youva



# Sommaire

- 1 Introduction : L'écosystème PyTorch
- 2 La gestion des données (LightningDataModule)
- 3 Les modèles (LightningModule)
- 4 L'apprentissage (Trainer)

# Les librairies de Deep learning

## Un grand nombre de librairies

- PyTorch,
- Tensorflow,
- Apache MXNet,
- Microsoft Cognitive Toolkit (anciennement CNTK),
- Chainer,
- DL4J,
- PlaidML...

## Les questions à se poser

- Le langage de programmation du projet,
- Les types de réseau de neurones.
- Les services de calculs notamment en cas d'utilisation de solution de cloud.
- Les besoins en déploiement.
- Les besoins en support et en documentation.

# L'écosystème PyTorch

## PyTorch : Librairie bas niveau

- Librairie pour la création de réseau de neurones.
- Très proche de Numpy dans son fonctionnement.
- Portage sur GPU.
- Adapté à des experts pour le développement de fonctionnalité non existante.

## PyTorch Lightning : Librairies de niveau intermédiaire

- L'objectif de ces librairies est de diminuer le travail d'ingénierie.
- Prise en compte rapide du hardware, de l'apprentissage distribué, du logging, de la visualisation et des mécanismes standards.
- Adapté aux chercheurs et aux ingénieurs en machine learning.
- Lightning est construit au-dessus de PyTorch et peut être facilement enrichie.

## Lightning Flash : Librairie haut niveau

- Pour le prototypage rapide, la reproduction de baseline ou la résolution de tâches d'apprentissage standard.
- Simple et facile à prendre en main.
- Adapté aux débutants.
- Flash est construit au-dessus de Lightning et peut être facilement enrichie.

# L'écosystème PyTorch

## Torchvision

Boîte à outils pour les applications de vision. La librairie contient

- Les transformations classiques sur les images (noir et blanc, redimensionnement, miroir, translation, rotations...)
- Les modèles classiques de la littérature,
- Les bases de données classiques de la littérature.



## TorchMetrics

- À l'origine inclus dans Pytorch Lightning.
- Contient l'implémentation des principales métriques (collection de +80 métriques standards).

# Pourquoi utiliser PyTorch Lightning ?

## Atouts de PyTorch Lightning

**Simplicité** Utilise un formalisme standard permettant d'avoir un code clair et limitant les possibilités de bugs.

**Flexibilité** Compatible avec tous les modèles PyTorch et n'importe quel type de données.

**Reproductibilité** Pipelines d'entraînement facilitant la reproductibilité.

**Monitoring** Intégration avec TensorBoard pour le suivi en temps réel des métriques.

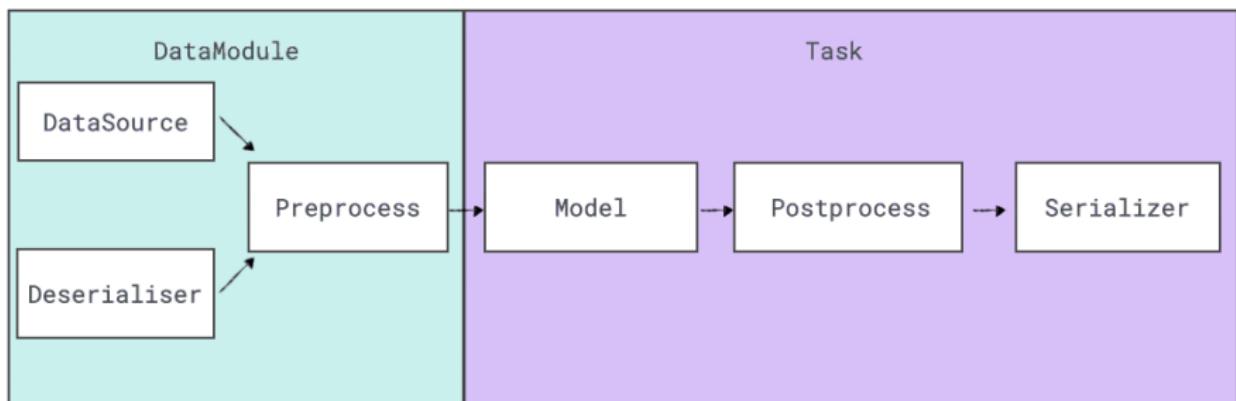
**GPU facilité** Gestion automatique de la mémoire GPU.

**Scalabilité** Possibilité de déployer sur TPU/multi-GPU et cluster de calculs.

**Personnalisation** Flexibilité pour personnaliser les pipelines d'entraînement.

# Déroulement d'un programme de Deep Learning

## Data Flow



1. Data Loading

2. Data Transforms

3. Model Forwarding

4. Predictions  
Transforms

5. Predictions  
Serialisation

# Sommaire

1 Introduction : L'écosystème PyTorch

2 La gestion des données (LightningDataModule)

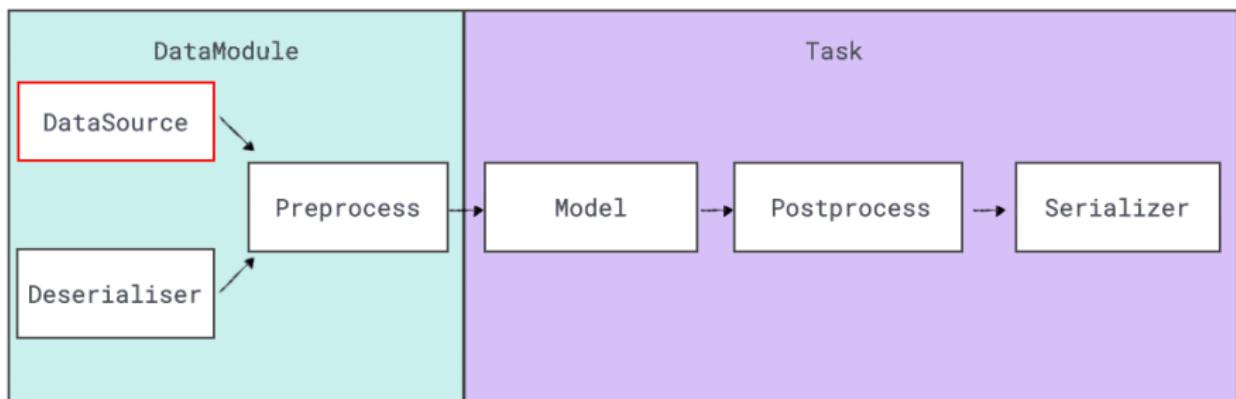
- Les bases de données en PyTorch
- Datasets avec torchvision

3 Les modèles (LightningModule)

4 L'apprentissage (Trainer)

# Déroulement d'un programme de Deep Learning

## Data Flow



1. Data Loading

2. Data Transforms

3. Model Forwarding

4. Predictions  
Transforms

5. Predictions  
Serialisation

# La gestion des bases de données en PyTorch

## Définition de la base de donnée

Les bases de données sont gérées en Pytorch au travers de classes héritant de `torch.utils.data.Dataset`. Ces classes se comportent comme des tableaux. Elles disposent des méthodes suivantes :

- `__getitem__(i)` : donnant l'élément  $i$  de la base de données
- `__len__()` : donnant la taille de la base de données

## La gestion du découpage en batch

Le découpage en mini-batch se fait en utilisant un objet de type `torch.utils.data.DataLoader` sur une base de données. L'objet généré possède une méthode `__iter__` permettant de générer un itérateur sur la base de données.

# Chargement de données avec un torch.TensorDataset

## Définition du dataset : torch.utils.data.TensorDataset

```
x_train, y_train = torch.rand(1000, 784), torch.LongTensor((np.  
    random.rand(1000)>0.5).astype('int'))  
  
train_ds = torch.utils.data.TensorDataset(x_train, y_train)
```

## Découpage en batch : torch.utils.data.DataLoader

```
train_dl = torch.utils.data.DataLoader(  
    train_ds,  
    batch_size=32,  
    shuffle=True  
)
```

# Redéfinition d'une classe torch.utils.data.Dataset

```
class RandomDataset(Dataset):
    def __init__(self, transform=None):
        self.data = np.random.random((1000, 784))
        self.y = (np.random.rand(1000)>0.5).astype('int')
        self.transform = transform

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        image = self.data[idx,:]
        label = self.y[idx]

        if self.transform:
            image = self.transform(image)

        return image.astype('float32'), label

train_ds = RandomDataset()
train_dl = DataLoader(train_ds, batch_size=32, shuffle=True)
```

# Torchvision et datasets standards

## Datasets disponibles

Les datasets disponibles sont présents dans le sous-package `torchvision.datasets` : MNIST, Fashion-MNIST, KMNIST, EMNIST, QMNIST, FakeData, COCO, LSUN, ImageFolder, DatasetFolder, ImageNet, CIFAR, STL10, SVHN, PhotoTour, SBU, Flickr, VOC, Cityscapes, SBD, USPS, Kinetics-400, HMDB51, UCF101 .

## Exemple d'utilisation : MNIST

```
>>> mnist_train = torchvision.datasets.MNIST('.', train=True,
    transform=None, target_transform=None, download=True)
>>> print(mnist_train[10])
(<PIL.Image.Image image mode=L size=28x28 at 0x13EFE5C10>, tensor
(3))
>>> mnist_train = torchvision.datasets.MNIST('.', train=True,
    transform=torchvision.transforms.ToTensor(), target_transform
    =None, download=True)
>>> train_dl = torch.utils.data.DataLoader(mnist_train, batch_size
    =32, shuffle=True)
```

Dans le cas où l'option `download` est activé, les fichiers sont téléchargés et stockés à la position indiquée (ici le dossier local).

# Datasets particuliers (1/3)

## Générateur de données aléatoires

La classe `torchvision.datasets.FakeData` permet de générer un dataset de données aléatoires.

```
>>> fake = torchvision.datasets.FakeData(  
    size=1000,  
    image_size=(3, 224, 224),  
    num_classes=10,  
    transform=None,  
    target_transform=None,  
    random_offset=0  
)  
>>> fake[10]  
(<PIL.Image.Image image mode=RGB size=224x224 at 0x13EDE7810>,  
 tensor(7.))
```

## Datasets particuliers (2/3)

### Lire des images à partir de dossiers d'images

La classe `torchvision.datasets.ImageFolder` permet de charger une base d'image se trouvant dans un dossier donné. Chaque sous-dossier est considéré comme une classe différente. Les images doivent être dans un format géré par la librairie PIL (PPM, PNG, JPEG, GIF, TIFF et BMP). Les fichiers dans un format non reconnu sont ignorés sans message d'erreur (sauf si une fonction adaptée est renseigné dans le paramètre `is_valid_file` ).

```
>>> ls images
0/ 1/
>>> data = torchvision.datasets.ImageFolder('images')
>>> data[0]
(<PIL.Image.Image image mode=RGB size=746x605 at 0x13F1A2A50>, 0)
```

## Datasets particuliers (3/3)

### Base de données à partir de dossiers de données

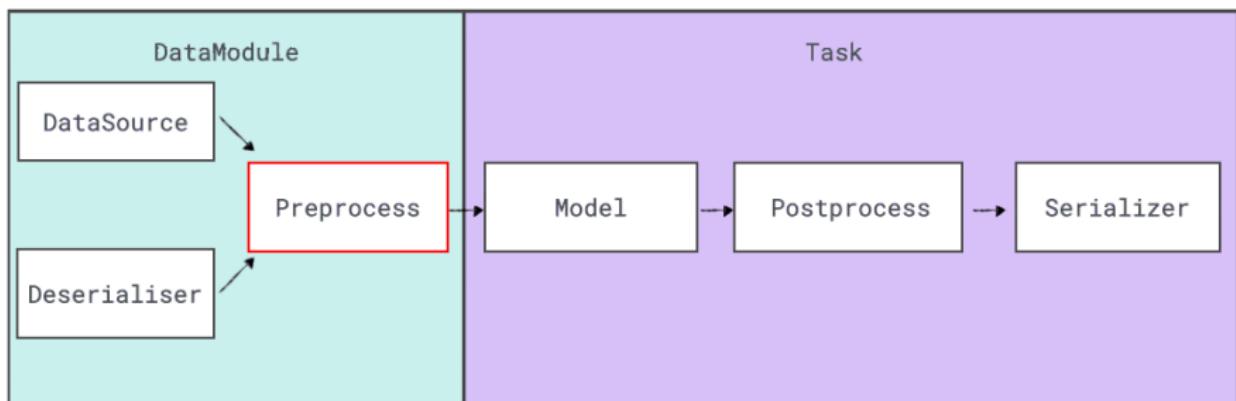
Il est possible d'avoir un comportement similaire à `torchvision.datasets.ImageFolder` pour des données quelconques en utilisant la classe `torchvision.datasets.DatasetFolder`. Il est alors nécessaire de préciser une fonction permettant la lecture des données.

```
def loader(name_file):
    return plt.imread(name_file)

d = torchvision.datasets.DatasetFolder(
    'images',
    loader,
    extensions=['png'] # liste des extensions de fichiers acceptable,
    ne pas oublier les []
)
```

# Déroulement d'un programme de Deep Learning

## Data Flow



1. Data Loading

2. Data Transforms

3. Model Forwarding

4. Predictions  
Transforms

5. Predictions  
Serialisation

# Prétraitement des données avec Torchvision

## Prétraitement des données (preprocess)

La librairie `torchvision` dans son sous-module `torchvision.transforms` contient un ensemble de fonction de pré-traitement que l'on peut appliquer sur les données avant leurs entrées dans le réseau de neurone. Ces fonctions sont utilisées pour faire par exemple de l'augmentation de données. Les traitements, en fonction de leur nature, s'appliquent soit à des images PIL soit à des `torch.Tensor`.

### `torchvision.transforms.Compose`

Le fonctionnement des objets de `torchvision.transforms` est très similaires aux objets de `torch.nn`. Vous avez des blocs définis dans `torchvision.transforms` qui appellent des fonctions `torchvision.transforms.functions` pour réaliser leurs opérations.

```
>>> new_im_pil = transforms.functional.vflip(im_pil)
>>> random_flip = transforms.RandomVerticalFlip(p=0.5) # p est la
       probabilité de faire le flip
>>> new_im_pil2 = random_flip(im_pil)
```

Vous pouvez construire un graphe enchainant les opérateurs de prétraitement en utilisant `torchvision.transforms.Compose`

```
>>> transforms.Compose([
    >>>     transforms.CenterCrop(10),
    >>>     transforms.ToTensor(),
    >>> ])
```

# Prétraitement des données

## Conversions

Les opérateurs de prétraitement s'applique soit à des images PIL soit à des tenseurs pytorch. Il est possible de passer de l'un à l'autre avec les blocs suivants :

- `torchvision.transforms.ToPILImage(mode)` , l'argument *mode* définissant l'espace couleurs à utiliser (nombre de canaux) et le type utiliser pour les pixels (identique au type donnée à mode).
- `torchvision.transforms.ToTensor()`

## Définition d'un bloc de pré-traitement à l'aide d'une fonction

```
>>> rotation_transform = torchvision.transforms.Lambda (  
    lambda x:torchvision.transforms.functional.rotate(  
        x,  
        np.random.choice([-30, -15, 0, 15, 30])  
    )  
)  
>>> rotation_transform(im_pil)
```

# Prétraitement des données

Définition d'un bloc de pré-traitement à l'aide de la définition d'une classe

Il suffit de définir une classe ayant une méthode `__call__(self,x)` :

```
class MyRotationTransform:  
    """Rotate by one of the given angles."""  
    def __init__(self, angles):  
        self.angles = angles  
  
    def __call__(self, x):  
        angle = np.random.choice(self.angles)  
        return torchvision.transforms.functional.rotate(x, angle)  
  
rotation_transform2 = MyRotationTransform(angles=[-30, -15, 0,  
                                              15, 30])  
rotation_transform2(im_pil)
```

# Prétraitement des données disponibles

## Pré-traitement sur les images PIL et les tenseurs

CenterCrop, ColorJitter, FiveCrop, Grayscale, Pad, RandomAffine, RandomApply, RandomCrop, RandomGrayscale, RandomHorizontalFlip, RandomPerspective, RandomResizedCrop, RandomRotation, RandomVerticalFlip, Resize, TenCrop, GaussianBlur, RandomInvert, RandomPosterize, RandomSolarize, RandomAdjustSharpness, RandomAutocontrast

## Pré-traitement uniquement sur des images PIL

RandomChoice, RandomOrder.

## Pré-traitement uniquement sur des Tensor

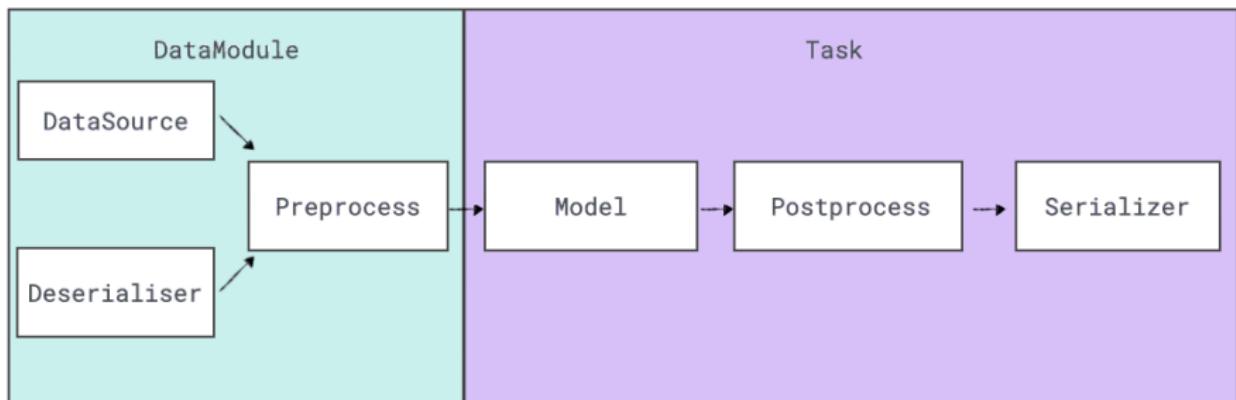
LinearTransformation, Normalize, RandomErasing, ConvertImageDtype.

# Exemple d'utilisation de pretraitement sur un Dataset

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.Resize(224),
    transforms.ToTensor()
])
cifar_full = datasets.CIFAR10(
    self.data_dir,
    train=True,
    download=False,
    transform=transform_train
)
```

# Déroulement d'un programme de Deep Learning : LightningDataModule

## Data Flow



1. Data Loading

2. Data Transforms

3. Model Forwarding

4. Predictions  
Transforms

5. Predictions  
Serialisation

# La gestion des données dans lightning

## Principe

Il est possible d'utiliser des *DataLoader* de PyTorch ou d'implémenter des objets héritant de *LightningDataModule*.

## Classe héritant de *LightningDataModule*

Les méthodes à implémenter sont les suivantes :

- `__init__` : Constructeur, permet de récupérer les paramètres.
- `prepare_data` : Permet de télécharger les données et de faire les splits. Appelé que sur le processus principal en cas de parallélisation.
- `setup` : Permet de faire l'initialisation de la base de données. Est appelé pour chaque processus en cas de parallélisation.
- `train_dataloader` : Retourne la base d'apprentissage.
- `val_dataloader` : Retourne la base de validation.
- `test_dataloader` : Retourne la base de test.

# Exemple de LightningDataModule (1/2)

```
class Cifar10DataModule(pl.LightningDataModule):
    def __init__(self,
                 data_dir:str='./data',
                 batch_size:int=32,
                 seed:int=42,
                 size_val:float=0.1
                 ):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size
        self.seed = seed
        self.size_val = size_val
        self.name_classes = ('plane', 'car', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

    def prepare_data(self):
        datasets.CIFAR10(self.data_dir, train=True, download=True)
        datasets.CIFAR10(self.data_dir, train=False, download=True)

    def train_dataloader(self) -> DataLoader:
        return DataLoader(self.cifar_train, batch_size=self.batch_size,
                          shuffle=True)
```

# Exemple de LightningDataModule (2/2)

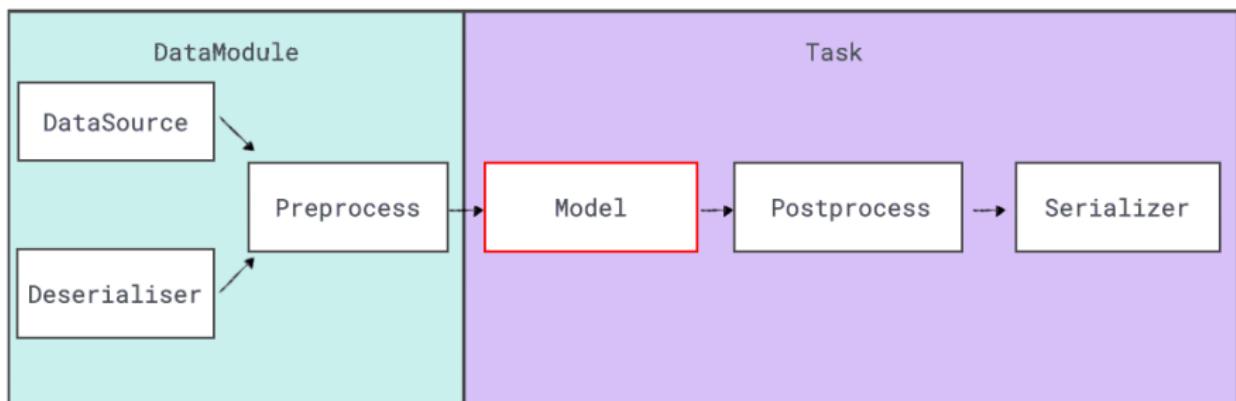
```
def setup(self, stage: Optional[str] = None) -> None:
    if stage == "test" or stage is None:
        transform_test = transforms.Compose([transforms.Resize(224),
                                              transforms.ToTensor()])
        self.cifar_test = datasets.CIFAR10(
            self.data_dir, train=False, download=False, transform=
                transform_test
        )
    if stage == "fit" or stage is None:
        transform_train = transforms.Compose(
            [transforms.RandomCrop(32, padding=4),
             transforms.Resize(224),
             transforms.ToTensor()])
        cifar_full = datasets.CIFAR10(
            self.data_dir, train=True, download=False, transform=
                transform_train
        )
        self.cifar_train, self.cifar_val = torch.utils.data.random_split(
            cifar_full,
            [len(cifar_full)-int(len(cifar_full)*self.size_val),
             int(len(cifar_full)*self.size_val)],
            generator=torch.Generator().manual_seed(self.seed)
        )
```

# Sommaire

- 1 Introduction : L'écosystème PyTorch
- 2 La gestion des données (LightningDataModule)
- 3 Les modèles (LightningModule)
  - L'optimizer
  - Torchmetrics
- 4 L'apprentissage (Trainer)

# Déroulement d'un programme de Deep Learning

## Data Flow



1. Data Loading

2. Data Transforms

3. Model Forwarding

4. Predictions  
Transforms

5. Predictions  
Serialisation

# Les modèles héritant de LightningModule

## Principe

Il est nécessaire de créer un objet héritant de la classe `LightningModule`. Cette classe permet d'organiser le code PyTorch selon les principales étapes :

`__init__` : définition des éléments de l'architecture

`forward` : Inférence du modèle

`configure_optimizers` : les optimiseurs et la gestion du taux d'apprentissage

`training_step` : la boucle d'entraînement

`validation_step` : la boucle de validation

`test_step` : la boucle de test

`predict_step` : la boucle de prédiction

# Les modèles héritant de LightningModule

## Principe

Il est nécessaire de créer un objet héritant de la classe `LightningModule`. Cette classe permet d'organiser le code PyTorch selon les principales étapes :

`__init__` : définition des éléments de l'architecture

`forward` : Inférence du modèle

`configure_optimizers` : les optimiseurs et la gestion du taux d'apprentissage

`training_step` : la boucle d'entraînement

`validation_step` : la boucle de validation

`test_step` : la boucle de test

`predict_step` : la boucle de prédiction

# Les paramètres du constructeur `__init__`

`self.save_hyperparameters` et `self.hparams`

Il est possible de sauvegarder tous les hyperparamètres du modèle dans la variable `hparams` avec la commande `self.save_hyperparameters()`. Cela permet notamment de les sauvegarder automatiquement dans les fichiers de sauvegarde et de logs.

```
def __init__(self,
             lr:float=0.1,
             momentum:float=0.9,
             weight_decay:float=1e-4,
             num_class:int=42
             *args,
             **kwargs
             ):
    super().__init__()
    self.save_hyperparameters()

    self.model = timm.create_model(
        'vgg16',
        num_classes=self.hparams.num_class
    )
```

# Torchvision : Architecture de réseau existant

## Principe

Le sous-package `torchvision.models` contient l'implémentation d'architecture standard pour des problèmes de :

- Classification d'images,
- Segmentation sémantique,
- Détection d'objets, de point d'intérêt sur des personnes ...
- Classification de vidéos.

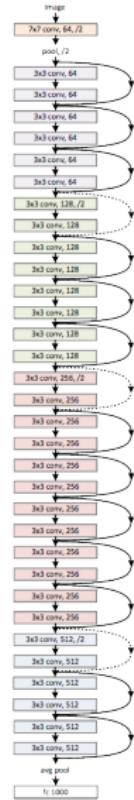
## Les réseaux pour la classification d'images

Les réseaux pour la classification sont accompagnés de poids pré-entraîné sur ImageNet.

- |                  |                |                   |                     |
|------------------|----------------|-------------------|---------------------|
| ● AlexNet        | ● Inception V3 | ● ResNet          | ● VisionTransformer |
| ● ConvNeXt       | ● MaxVit       | ● ResNeXt         | ● Wide ResNet       |
| ● DenseNet       | ● MNASNet      | ● ShuffleNet V2   |                     |
| ● EfficientNet   | ● MobileNet V2 | ● SqueezeNet      |                     |
| ● EfficientNetV2 | ● MobileNet V3 | ● SwinTransformer |                     |
| ● GoogLeNet      | ● RegNet       | ● VGG             |                     |

# Exemple d'utilisation d'une architecture ResNet

34-layer residual



## Variantes de ResNet implémentées

resnet18, resnet34, resnet50, resnet101, resnet152

## Exemple d'utilisation

```
from torchvision.models import resnet34, ResNet34_Weights
model = resnet34(weights=ResNet34_Weights.DEFAULT)
# Redefinissons de la dernière couche pour changer le
# nombre de classe cible.
model.fc = nn.Linear(512,10)
```

Si `weights=None` les poids sont initialisés aléatoirement dans le cas contraire, ils sont initialisés avec le résultat d'un apprentissage précédent. Il est possible de choisir le pre-training utilisé.

Les images attendues en entrées du réseau sont des images couleurs de taille 224x224.

# TIMM : une alternative à Torchvision

## La librairie Timm

La bibliothèque **timm**, développée par Ross Wightman, regroupe des modèles "état de l'art" en vision par ordinateur ainsi que des couches, utilitaires, optimiseurs, planificateurs, chargeurs de données, augmentations et scripts d'entraînement/validation standard. Cette bibliothèque est conçue pour reproduire les résultats d'entraînement ImageNet.

## Exemple d'utilisation

```
import timm
import torch.nn as nn

model = timm.create_model('resnet34',
    pretrained=True,
    num_classes=10
)
```

La liste des modèles disponible est accessible avec la commande  
timm.list\_models().

# Nombre de canaux d'entrées pour les modèles pré-entraînés

## Cas général

Généralement le modèle ont été pré-entraîné pour des images couleurs (à 3 canaux).

## Torchvision

```
m = torchvision.models.resnet34(weights=ResNet34_Weights.DEFAULT)
x = torch.randn(1, 1, 224, 224)
# `torchvision` raises error
try: m(x).shape
except Exception as e: print(e)
> Given groups=1, weight of size [64, 3, 7, 7], expected input [1,
  1, 224, 224] to have 3 channels, but got 1 channels instead
```

# Nombre de canaux d'entrées pour les modèles pré-entraînés

Timm

```
m = timm.create_model('resnet34', pretrained=True, in_chans=1)

# single channel image
x = torch.randn(1, 1, 224, 224)

try: m(x).shape
except Exception as e: print(e)
```

Timm : fonctionnement dans le cas 1 canal

Si le modèle à un canal n'est pas existant dans la base, la librairie prend le modèle à trois canaux et fait la somme des poids pour n'avoir qu'un seul canal.

Timm : fonctionnement dans le cas n-canaux

Les 3 canaux sont recopier pour avoir  $3(n//3)+3$  canaux puis on ne garde que les  $n$  premiers canaux.

# Les modèles héritant de LightningModule

## Principe

Il est nécessaire de créer un objet héritant de la classe `LightningModule`. Cette classe permet d'organiser le code PyTorch selon les principales étapes :

`__init__` : définition des éléments de l'architecture

`forward` : Inférence du modèle

`configure_optimizers` : les optimiseurs et la gestion du taux d'apprentissage

`training_step` : la boucle d'entraînement

`validation_step` : la boucle de validation

`test_step` : la boucle de test

`predict_step` : la boucle de prédiction

# La méthode forward

## Principe

Cette méthode retourne l'inférence du modèle sur un batch passé en argument.

## Exemple d'utilisation

```
class MyModel(L.LightningModule):
    def __init__(self, num_class=19):
        super().__init__()
        self.save_hyperparameters()
        self.model = timm.create_model(
            'resnet50',
            pretrained=True,
            in_chans=1,
            num_classes=self.hparams.num_class
        )

    def forward(self, x):
        return self.model(x)
```

# Les modèles héritant de LightningModule

## Principe

Il est nécessaire de créer un objet héritant de la classe `LightningModule`. Cette classe permet d'organiser le code PyTorch selon les principales étapes :

`__init__` : définition des éléments de l'architecture

`forward` : Inférence du modèle

`configure_optimizers` : les optimiseurs et la gestion du taux d'apprentissage

`training_step` : la boucle d'entraînement

`validation_step` : la boucle de validation

`test_step` : la boucle de test

`predict_step` : la boucle de prédiction

# L'optimiseur

## Principe

- On configure l'optimiseur dans la méthode *configure\_optimizers*.
- Les valeurs rentrées sont soit l'optimiseur, soit une liste d'optimiseur et une liste de *scheduler* pour le *learning rate*, soit un dictionnaire contenant toutes ces informations.

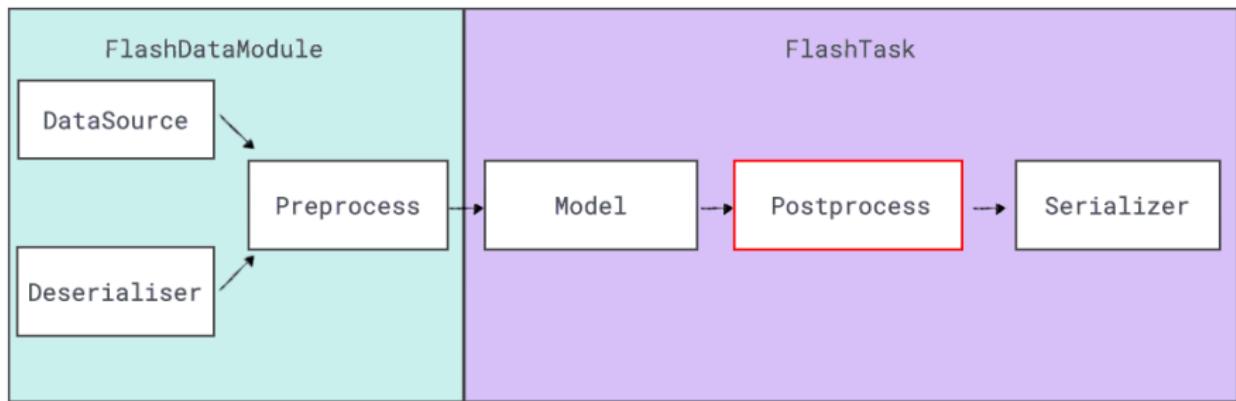
## Exemple

```
def configure_optimizers(self):  
    self.optim_algo = torch.optim.SGD(self.parameters(),  
        lr=self.hparams.lr,  
        momentum=self.hparams.momentum,  
        weight_decay=self.hparams.weight_decay  
    )  
    lr_scheduler = torch.optim.lr_scheduler.StepLR(self.optim_algo,  
        step_size=100, gamma=0.1)  
    return {'optimizer':self.optim_algo,  
        'lr_scheduler':{  
            'scheduler':lr_scheduler  
        }  
    }
```

# Déroulement d'un programme de Deep Learning



## Flash Data Flow



1. Data Loading

2. Data Transforms

3. Model Forwarding

4. Predictions  
Transforms

5. Predictions  
Serialisation

# Les modèles héritant de LightningModule

## Principe

Il est nécessaire de créer un objet héritant de la classe `LightningModule`. Cette classe permet d'organiser le code PyTorch selon les principales étapes :

`__init__` : définition des éléments de l'architecture

`forward` : Inférence du modèle

`configure_optimizers` : les optimiseurs et la gestion du taux d'apprentissage

`training_step` : la boucle d'entraînement

`validation_step` : la boucle de validation

`test_step` : la boucle de test

`predict_step` : la boucle de prédiction

# La méthode training\_step

## Principe

La méthode `training_step` contient tous les éléments effectués lors de l'étape d'apprentissage (hors mise à jour des poids du réseau). Généralement l'évaluation du modèle sur les données puis le calcul de la fonction de coût.

## Exemples

```
def __init__(self):
    ...
    self.loss_train = nn.CrossEntropyLoss()

def training_step(self, batch, batch_idx):
    x, y = batch

    y_hat = self(x)
    vloss = self.loss_train(y_hat, y)

    return vloss
```

# La gestion des Logs

## Principe

Il est possible d'enregistrer facilement les évolutions de valeurs au cours de l'apprentissage en utilisant les méthodes *log* ou *log\_dict*.

## Exemples

```
# Log d'une metric
def training_step(self, batch, batch_idx):
    ...
    self.log("my_metric", x)

# Log de plusieurs metrics sur la même figure
def training_step(self, batch, batch_idx):
    ...
    self.log("performance", {"acc": acc, "recall": recall})

# Log de plusieurs metrics sur des figures différentes
def training_step(self, batch, batch_idx):
    ...
    self.log_dict({"acc": acc, "recall": recall})
```

# La gestion des Logs

Comportement par défaut de l'enregistrement des logs dans les LightningModule

Les logs peuvent être agrégé par étape ou par époque. Par défaut, on a les paramètres suivant en fonctions des méthodes ou log est appelé :

Hook	on_step	on_epoch
on_train_start, on_train_epoch_start, on_train_epoch_end, training_epoch_end	False	True
on_before_backward, on_after_backward, on_before_optimizer_step, on_before_zero_grad	True	False
on_train_batch_start, on_train_batch_end, training_step, training_step_end	True	False
on_validation_start, on_validation_epoch_start, on_validation_epoch_end, validation_epoch_end	False	True
on_validation_batch_start, on_validation_batch_end, validation_step, validation_step_end	False	True

# Torchmetrics : Le calcul des métriques d'évaluation

## Principe

Le calcul de métriques d'évaluation permet de savoir si le réseau est performant sur la tâche cible.

## Torchmetrics

La librairie *Torchmetrics* implémente les métriques standard et s'intègre parfaitement à Pytorch Lightning.

## Exemple

```
class MyModel(LightningModule):
    def __init__(self, num_classes):
        ...
        self.valid_acc = torchmetrics.Accuracy(task="multiclass",
                                                num_classes=num_classes)
    def validation_step(self, batch, batch_idx):
        x, y = batch
        preds = self(x)
        self.valid_acc(preds, y)
        self.log('val_acc', self.valid_acc)
```

# Synthèse : Exemple de modèle héritant de LightningModule

```
class LitAutoEncoder(pl.LightningModule):
    def __init__(self): # definition du modele
        super().__init__()
        self.net = timm.create_model('vgg16', in_chans=1, pretrained=True,
                                     num_classes=10)

        self.loss_train = nn.CrossEntropyLoss()

    def forward(self,x):
        return self.net(x)

    def training_step(self, batch, batch_idx): # etape de l'
                                                # apprentissage
        x, y = batch
        x_hat = self(x)
        vloss = self.loss_train(x_hat, y)
        self.log("train_loss", vloss)
        return vloss

    def configure_optimizers(self): # configuration de l'optimiseur
        optimizer = optim.Adam(self.parameters(), lr=1e-3)
        return optimizer
```

# Sommaire

- 1 Introduction : L'écosystème PyTorch
- 2 La gestion des données (LightningDataModule)
- 3 Les modèles (LightningModule)
- 4 L'apprentissage (Trainer)

# L'objet Trainer

## Principe

L'objet *Trainer* de PyTorch Lightning est une abstraction qui automatise toute la partie ingénierie du code PyTorch. Il permet de contrôler tous les aspects de l'entraînement, comme le nombre d'époques, les accélérateurs, les callbacks, etc. Il s'agit du cœur de fonctionnement du framework PyTorch Lightning.

## Exemple

```
# Definition du modèle
model = MyLightningModule()
# Definition de la base de données
trainset = ...
valset = ...
train_dataloader = torch.utils.data.DataLoader(trainset, batch_size=32,
    shuffle=True)
val_dataloader = torch.utils.data.DataLoader(valset, batch_size=32,
    shuffle=False)
# Apprentissage du modèle
trainer = Trainer()
trainer.fit(model, train_dataloader, val_dataloader)
```

# Utilisation de LightningDataModule

## Principe

On peut passer un objet de type *LightningDataModule* à l'attribut *datamodule* de la méthode *fit* du *Trainer* :

```
cifar = Cifar10DataModule()  
model = MyModel()  
  
# Apprentissage sur 10 époques  
trainer = Trainer(max_epochs = 10)  
trainer.fit(model, datamodule = cifar)
```