RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

METEO FRANCE
À VOS CÔTÉS, DANS UN CLIMAT QUI CHANGE

Machine Learning – Recap' n°3

Pierre Lepetit
ENM, le 08/11/2024

Compter le nombre de paramètres dans un réseau :

```
class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 10)
        self.conv2 = nn.Conv2d(10, 10)
        self.fc1 = nn.Linear(490, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        ...
        return F.log_softmax(x, dim=1)
```

A la main :

Compter le nombre de paramètres dans un réseau :

```
class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 10)
        self.conv2 = nn.Conv2d(10, 10)
        self.fc1 = nn.Linear(490, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        ...
        return F.log_softmax(x, dim=1)
```

A la main :

- 10*1*5*5 + 10 + 10*10*5*5 + 10 = 2770
- 490*50 + 50 + 50*10 + 10 = 25060
- 27830 poids

Compter le nombre de paramètres dans un réseau :

```
class CNN(nn.Module):

    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 10)
        self.conv2 = nn.Conv2d(10, 10)
        self.fc1 = nn.Linear(490, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        ...
        return F.log_softmax(x, dim=1)
```

A la main :

- 10*1*5*5 + 10 + 10*10*5*5 + 10 = 2770
- 490*50 + 50 + 50*10 + 10 = 25060
- 27830 poids

```
# Avec du code:
nb_weights = 0
for module in model.modules():
  if isinstance(module, nn.Conv2d) or isinstance(module, nn.Linear) or ... :
    for parameter in module.parameters():
      nb_weights += torch.numel(parameter)
```

Fonctions de coût pour la classification multiclasse :

Les deux fonctions de coût la(les) plus utilisée(s) est(sont) l'entropie croisée et la « dice loss »

- L'entropie croisée peut être implémentée de deux plusieurs façons sous pytorch:

  - La dernière fonction d'activation du réseau est nn.Softmax(dim=k)
    La fonction de coût contient :
    - Un torch.log
    - torch.nn.NLLLoss(ignore_index=-100, reduction='mean')

  - La dernière fonction d'activation du réseau est nn.LogSoftmax(dim=k)
    La fonction de coût contient :
    - torch.nn.NLLLoss(ignore_index=-100, reduction='mean')

  - Le réseau ne se termine pas par une fonction d'activation
    La fonction de coût contient :
    - torch.nn.CrossEntropyLoss(ignore_index=-100, reduction='mean')

La tripartition standard « entraînement, validation et test »

- Un apprentissage standard se fait à partir de trois jeux de données indépendants, **représentatifs** et **non redondants** : entraînement, validation et test.

- Une boucle d'apprentissage standard contient deux phases.
  - **Phase d'entraînement** :
    Les poids sont mis à jour de manière à réduire la « loss »
    sur les éléments d'un batch d'entraînement, puisés dans le jeu d'entraînement
  - **Phase de validation** :
    Les performances en généralisations sont contrôlées sur les éléments
    d'un batch de validation, puisés dans le jeu de validation.

Boucle d’apprentissage standard :

```
for epoch in range(num_epochs):
    # init running scores
    running_corrects_val = 0.

    # Training
    model.train()
    for x, label in train_loader:
        ...
        optimizer.step() # mise à jour des poids

    # validation
    model.eval()
    for x, label in val_loader:
        with torch.no_grad():
            ...  # prédictions associées au batch de validation
        running_corrects_val += torch.sum(preds == label.data)

    # Calculate scores and store:
    epoch_acc_val = running_corrects_val.float() / val_size
    val_accs.append(epoch_acc_val)
```

Boucle d’apprentissage standard :

```
phases = ['train', 'val']
accs = {p: [] for p in phases}

for epoch in range(num_epochs):
    # init running scores
    running_corrects_val = {p: 0. for p in phases}

    for phase in phases:
        if phase == 'train':
            model.train()
        else:
            model.eval()

        for inputs, labels in dataloaders[phase]:
            with torch.set_grad_enabled(phase == 'train'):
            ...
        running_corrects[phase] += torch.sum(preds == label.data)

        # Calculate scores and store:
        epoch_acc = running_corrects[phase].float() / len(dataset[phase])
        accs[phase].append(epoch_acc[phase])
```

Boucle d’apprentissage (moins) standard :

```
phases = ['train', 'val1', 'val2']
accs = {p: [] for p in phases}

for epoch in range(num_epochs):
    # init running scores
    running_corrects_val = {p: 0. for p in phases}

    for phase in phases:
        if phase == 'train':
            model.train()
        else:
            model.eval()

        for inputs, labels in dataloaders[phase]:
            with torch.set_grad_enabled(phase == 'train'):
            ...
        running_corrects[phase] += torch.sum(preds == label.data)

        # Calculate scores and store:
        epoch_acc = running_corrects[phase].float() / len(dataset[phase])
        accs[phase].append(epoch_acc[phase])
```

Boucle d’apprentissage standard :

Question :
Au cours d’un apprentissage, combien de fois parcourt-on le jeu de test ?

Accélérer la boucle d’apprentissage:

Les principales commandes qui permettent d’accélérer l’apprentissage impliquent :

- La parallélisation du chargement **et du prétraitement** des données :

```
bs = 8
num_workers = 2 # try : print(os.cpu_count())

train_loader = DataLoader(trainval_dataset, batch_size=bs,
                          sampler=train_sampler, num_workers=num_workers)
```

Accélérer la boucle d'apprentissage:

Les principales commandes qui permettent d'accélérer l'apprentissage impliquent :

- La parallélisation du chargement **et du prétraitement** des données :
- *.forward* et *.backward* sur carte graphique :

```
device = torch.device('cuda:0')
# ou device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# ou device = torch.device("cuda:" + str(k))
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
                inputs = inputs.to(device)
                labels = labels.to(device)
```

Accélérer la boucle d'apprentissage:

Les principales commandes qui permettent d'accélérer l'apprentissage impliquent :

- La parallélisation du chargement **et du prétraitement** des données :
- *.forward* et *.backward* sur carte graphique :

```
device = torch.device('cuda:0')
# ou device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# ou device = torch.device("cuda:" + str(k))
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = model.to(device)
                inputs = inputs.to(device)
                labels = labels.to(device)
```

- La réduction partielle de la précision des calculs → notion de « mixed precision »
- L'utilisation de plusieurs gpus