PyTorch

INF8460 - Traitement automatique de la langue naturelle

Gaya Mehenni

Polytechnique Montréal

13 septembre 2024

- Librairie d'apprentissage profond à code source ouvert développée par Facebook (Meta)
- Graphes de calculs dynamique + Support GPU
- Très utilisé en recherche et en industrie

- Tableaux multi-dimensionnels comme les tableaux de Numpy
- Blocs de construction fondamentaux pour stocker et manipuler des données

- Peuvent être créés à partir de listes ou à partir de fonctions prédéfinies
- Plusieurs opérations supportées pour modifier les tenseurs
- Peuvent être exécutés sur un GPU ou CPU (ou MPS)

```python
# Créer un tenseur
x = torch.arange(1, 6)
print(x)  # tensor([1, 2, 3, 4, 5])

# Créer un tenseur 2D
y = torch.tensor([[1, 2], [3, 4], [5, 6]])
print(y)  # tensor([[1, 2], [3, 4], [5, 6]])

# Opération d'addition
z = x + 10
print(z)  # tensor([11, 12, 13, 14, 15])

# Multiplication de matrice
a = torch.matmul(y, torch.tensor([1, 2]))
print(a)  # tensor([ 5, 11, 17])
```

- **Création de tenseurs**
  - torch.zeros(shape) : Crée un tenseur rempli de 0
  - torch.arange(start, end) : Crée un tenseur contenant une liste de nombre consécutifs
- **Propriétés**
  - x.shape : Retourne les longueurs de chaque dimensions
  - x.dim() : Retourne le nombre de dimensions
- **Manipulation de tenseurs**
  - x[:4, 3 :5, :] : Prend les 4 premiers éléments de la dimension 0, les éléments 3 et 4 de la dimension 1 et tous les éléments de la dernière dimension
  - x.unsqueeze(dim) : Ajoute une dimension à l'index de la dimension indiquée par dim
- **Changement de machine** :
  - x.to(device) : Déplace le tenseur vers *device*

```python
# Créer un tenseur
x = torch.tensor([[1, 2, 3, 4, 5], [5, 4, 3, 2, 1]])
print(x.shape) # torch.Size([2, 5])
print(x.dim()) # 2

y = x.unsqueeze(1) # Ajoute une dimension à l'index 1
print(y.shape) # tensor.Size([2, 1, 5])

z = y[1, :, 3:5]
print(z) # tensor([[2, 1]])
```

Autograd

- Engin de différentiation automatique
- Calcule les gradients automatiquement pour tous les tenseurs (gradient est stocké dans l'attribute grad)
- Crée un graphe de calcul dynamique permettant de calculer les gradients
- Efficace pour la rétropropagation (backward())

Back-propagation
uses the dynamically created graph

```
W_h = torch.randn(20, 20, requires_grad=True)
W_x = torch.randn(20, 10, requires_grad=True)
x = torch.randn(1, 10)
prev_h = torch.randn(1, 20)

h2h = torch.mm(W_h, prev_h.t())
i2h = torch.mm(W_x, x.t())
next_h = h2h + i2h
next_h = next_h.tanh()

loss = next_h.sum()
loss.backward()  # compute gradients!
```

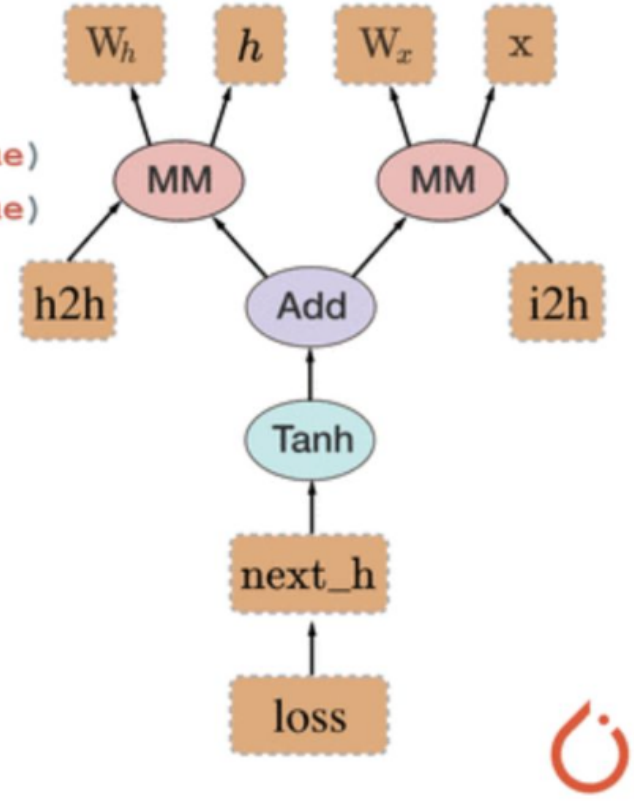


Figure – Autograd graph

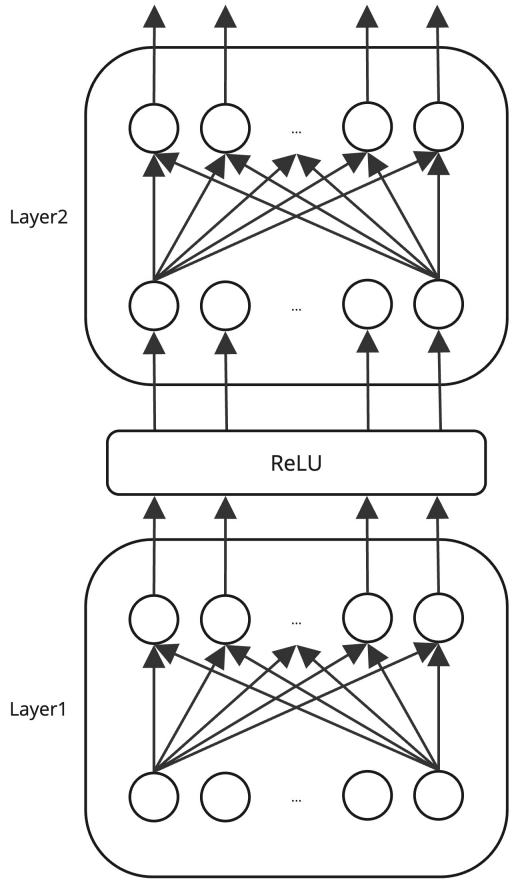
```python
# Créer des tenseurs en activant les gradients
x = torch.tensor([1.0, 2.0, 3.0], requires_grad=True)
y = torch.tensor([4.0, 5.0, 6.0], requires_grad=True)

# Opérations sur ces tenseurs
z = x * y
loss = z.sum()

# Calculer les gradients
loss.backward()

# Afficher les gradients
print(x.grad) # tensor([4., 5., 6.])
print(y.grad) # tensor([1., 2., 3.])
```

Modules

- Tous les modèles de PyTorch héritent de torch.nn.Module
- Permet de diviser facilement les couches et opérations
- Facilite l'entraînement des modèles (accès aux paramètres entraînables avec l'attribut .parameters())
- Plusieurs classes de bases offertes (nn.Linear, nn.Conv2d, nn.Sigmoid, nn.ReLU, nn.LSTM, nn.Dropout, nn.Embedding)
- Simplement besoin d'implémenter la fonction forward()

Créer un modèle


```
class SimpleNetwork(nn.Module):
    def __init__(self,
        in_dim, h_dim, out_dim
    ):
        super().__init__()
        self.layer1 = nn.Linear(
            in_dim, h_dim
        )
        self.relu = nn.ReLU()
        self.layer2 = nn.Linear(
            h_dim, out_dim
        )
    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.layer2(x)
        return x
model = SimpleNetwork(10, 20, 5)
```



Entraîner un modèle

- Initialiser l'optimiseur avec les paramètres du modèle
- Mettre modèle en mode entraînement (active le calcul des gradients)
- Pour chaque lot (batch) :
  - Réinitialiser les gradients
  - Passer les entrées à travers le modèle
  - Calculer la valeur de la fonction de perte
  - Calcule les gradients par rapport à l'entrée
  - Mettre à jour les paramètres

Entraîner un modèle

```
model = SimpleNetwork(...)
# Initialiser l'optimiseur avec les paramètres du modèle
optimizer = optim.Adam(model.parameters())
loss_function = nn.CrossEntropyLoss()
for epoch in range(5):
    model.train() # Activer les gradients
    for batch in train_iterator:
        optimizer.zero_grad() # Réinitialiser les gradients
        inputs, labels = batch
        predictions = model(inputs) # Obtenir les prédictions
        loss = loss_function(predictions, labels) # Calculer le coût

        # Calculer gradient du coût par rapport aux entrées
        loss.backward()

        optimizer.step() # Mettre à jour les paramètres
```

Ressources utiles

- PyTorch Tutorials
- PyTorch Tensor Documentation
- PyTorch Tutorial Github Repository
- UvA Deep Learning Course - Introduction to PyTorch