Deep Learning with Pytorch

Redes neuronales

- Una ANN tiene cuatro componentes principales:
  1. Capas, donde ocurre el aprendizaje (capa de entrada, ocultas y de salida).
  2. Datos de entrada y salida:
     - Datos para el entrenamiento/aprendizaje (entrada);
     - Predicción o etiqueta (salida)
  3. Función de pérdida: métrica utilizada para estimar el rendimiento de la fase de aprendizaje
  4. Optimizador: método para mejorar el aprendizaje, mediante la actualización del conocimiento en la red (pesos para la ponderación)

La ANN toma los datos de entrada para alimentar al conjunto de capas.
El rendimiento se evalúa con la función de pérdida.
El conocimiento se mejora a través del optimizador.

Ciclo de un modelo utilizando ANN

1. Definir el modelo
2. Compilar el modelo / Configurar el entrenamiento
3. Ajustar/entrenar el modelo
4. Evaluar el modelo
5. Realizar predicciones/clasificaciones/generaciones

Definir el modelo

- Implica seleccionar el tipo de modelo (MLP, RNN, CNN, etc.)
- Luego se debe determinar la arquitectura de la ANN a utilizar.
- Por último, deben definirse las capas del modelo y configurarlas, especificando el número de neuronas, la función de activación y las conexiones entre las capas del modelo.

Compilar el modelo / Configurar el aprendizaje

- Requiere definir una función de pérdida (loss function) para el entrenamiento (optimización de los pesos de la ANN):
  - Mean squared error, cross-entropy, etc.
- Requiere seleccionar un algoritmo para la optimización de los pesos:
  - Stochastic gradient descent (SGD).
  - Variantes de SGD: Adam, etc.
  - Otros
- Requiere determinar métricas relevantes para monitorear el desempeño de la etapa de entrenamiento.

Ajustar/entrenar el modelo

- El entrenamiento aplica el algoritmo de optimización especificado en la etapa previa para minimizar la función de pérdida seleccionada y actualiza los pesos del modelo utilizando un método de propagación hacia atrás (*backpropagation*) del error. Todo ello se lleva a cabo durante un proceso iterativo que se repite durante un número de épocas de entrenamiento (*epochs*).
- Para cada época de entrenamiento (de forma iterativa también) se van seleccionando subconjuntos de muestras (*batches*) de un tamaño determinado (*batch size*) para estimar el error.
  - El tamaño del batch y el del conjunto de datos de entrenamiento determina cuántas veces se estima el error (y se actualizan los pesos) durante una época de entrenamiento.

Ajustar/entrenar el modelo

- Para cada época de entrenamiento (de forma iterativa también) se van seleccionando subconjuntos de muestras (*batches*) de un tamaño determinado (*batch size*) para estimar el error.
    - El tamaño del batch y el del conjunto de datos de entrenamiento determina cuántas veces se estima el error (y se actualizan los pesos) durante una época de entrenamiento.

Ajustar/entrenar el modelo

- Así, se requiere

1. seleccionar la configuración de entrenamiento:
   - Número de épocas (epochs)
   - Tamaño del batch (batch size)

2. programar los dos bucles anidados para ejecutar el entrenamiento:
   - El bucle externo → Controla las épocas de entrenamiento.
   - El bucle interno → Consume los datos en batches y actualiza los pesos en función del error y el algoritmo de optimización.

Ajustar/entrenar el modelo

- Es la etapa que consume mayor tiempo de cómputo, dependiendo de la complejidad del problema, la complejidad del modelo, el hardware utilizado y el tamaño del conjunto de entrenamiento.

Evaluar el modelo

- Requiere determinar un conjunto de datos de validación.
- Datos no utilizados en la etapa de entrenamiento, para evitar sesgos.
- El tiempo requerido para la evaluación es proporcional al tamaño del conjunto de datos de validación (mucho más rápido que el entrenamiento)

Predecir/clasificar/generar

- Validar el modelo sobre datos desconocidos.
  - Predicción: datos para los cuales no se conoce el valor a predecir.
  - Clasificación: datos para los cuales no se conoce la clase a la que pertenecen.
  - Generación: Crear nuevas imágenes, datos tabulares, series temporales, etc.

- Los modelos pueden salvarse para ser utilizados posteriormente.
- Un modelo previamente creado (y entrenado) puede ser re-entrenado/recalibrado sobre otros conjuntos de datos.

PyTorch

- Biblioteca de código abierto desarrollada por Facebook
- Definida para trabajar de forma eficiente sobre tensores

  (clase Tensor → torch.Tensor)
  - Los tensores permiten trabajar de forma eficiente sobre arrays de números homogéneos y multidimensionales.
  - Los tensores son similares a los vectores de NumPy pero se pueden operar en GPUs Nvidia compatibles con CUDA.
- Ejecuta en una o múltiples CPUs/GPUs.

Instalación de PyTorch

- Python 3.x (también existen versiones para C++/Java)
- Se puede instalar empleando pip y conda, pero la forma más sencilla es empleando conda:
  - Instalación para **GPU/CPU**:

```
conda install pytorch torchvision torchaudio pytorch-cuda=11.7 -c pytorch -c nvidia
```

  - Instalación **solo CPU**:

```
conda install pytorch torchvision torchaudio cpuonly -c pytorch
```

https://pytorch.org/

Ejemplos de código Pytorch

- Todo el código que viene a continuación se encuentra en el notebook siguiente: https://colab.research.google.com/drive/1M0Zima4irwtbCtYuOm5d4u10MZG-X6Q3?usp=sharing

Instalación de PyTorch

- Para confirmar que Pytorch está instalado:
  - Crear un archivo check-pytorch.py.

```
import torch
# Checking pytorch version
print('+ PyTorch version: ' + torch.__version__)
# Checking GPU availability
print('+ Is GPU available? ' + str(torch.cuda.is_available()))
# Creating a random tensor
print('+ Creating a random tensor...')
print(torch.rand(5, 3))
```

```
python check-pytorch.py
```

- Salida:

```
+ PyTorch version: 1.9.0+cu111
+ Is GPU available? True
+ Creating a random tensor...
tensor([[0.8905,...5]])
```

Datos en Pytorch - Tensores

- Los datos que maneja Pytorch son siempre tensores
  - `torch.tensor` de Pytorch son similares a los arrays de Numpy `numpy.array`
- Los tipos del tensor se definen empleando `dtype=torch.xxx` (xxx es el tipo)
  - `X = torch.tensor(([2, 9], [1, 5], [3, 6]), dtype=torch.float) # 3 X 2`
    `tensor tensor([[2., 9.],`
    `[1., 5.],`
    `[3., 6.]])`
  - `y = torch.tensor(([92], [100], [89]), dtype=torch.int) # 3 X 1`
    `tensor tensor([[ 92],`
    `[100],`
    `[ 89]], dtype=torch.int32)`
- Suele ser muy útil el empleo de la función `size()` para conocer las dimensiones de los tensores
  - `print(X.size())`
    `torch.Size([3, 2])`
  - `print(y.size())`
    `torch.Size([3, 1])`

Ciclo de un modelo utilizando ANN

1. Definir el modelo
2. Compilar el modelo → Configurar el aprendizaje
3. Ajustar el modelo → Aplicar el algoritmo de aprendizaje
4. Evaluar el modelo
5. Realizar predicciones/clasificaciones/generaciones

Ciclo de un modelo en Pytorch

Pytorch: definir el modelo

- Muy similar a la solución aplicada por Tenosrflow/Keras

- Implica seleccionar el tipo de modelo (MLP, RNN, CNN, etc.)
- Luego se debe determinar la arquitectura de la ANN a utilizar.
- Por último, deben definirse las capas del modelo y configurarlas, especificando el número de neuronas, la función de activación y las conexiones entre las capas del modelo.

Pytorch: definir el modelo

- Todo se crea de en una clase que hereda de `torch.nn.Module`
- La clase debe disponer de un constructor y la función forward de propagación de la entrada a la salida

```
class MyNeuralNetwork(nn.Module):
    def__init__(self, input_parameter1, input_parameter2, ...):
        super(self).__init__()
        # Neural network building

    def forward(self, x, ...):
       # Forward operations
       return y
```

- Al igual que con Keras existen dos opciones para la construcción de la red:
  - Empleando API secuencial
  - No empleando API secuencial

Pytorch: definir el modelo con la API secuencial

- Implica instanciar la clase secuencial (Sequential) y agregar secuencialmente capas al modelo, desde la entrada a la salida (no se emplea add cono en Keras)
- Ejemplo: modelo MLP secuencial que acepta ocho entradas, tiene una capa oculta con 10 neuronas y una capa de salida

```
import torch.nn as nn
class MyNeuralNetwork(nn.Module):
    def__init__(self):
        super(MyNeuralNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(8, 10),
            nn.Linear(10, 1)
        )

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

Pytorch: definir el modelo con la API secuencial

```
# Testing our neural network
myNN = MyNeuralNetwork()
print(myNN)
     MyNeuralNetwork(
       (net): Sequential(
         (0): Linear(in_features=8, out_features=10, bias=True)
         (1): Linear(in_features=10, out_features=1, bias=True)
       )
     )

# Checking the output
X = torch.rand(1, 8)
output = myNN(X)
print('Input: ' + str(X))
print('Output: ' + str(output))
     Input: tensor([[0.2211, 0.5856, 0.2668, 0.1246, 0.4728, 0.4778, 0.0343,
     0.7971]])
     Output: tensor([[0.1968]], grad_fn=<AddmmBackward>)
```

Pytorch: definir el modelo sin la API secuencial

- No se instancia la clase Sequential(), por lo que el comportamiento de la red se define en la función forward
- Ejemplo: modelo MLP secuencial que acepta ocho entradas, tiene una capa oculta con 10 neuronas y una capa de salida

```
import torch.nn as nn
class MyNeuralNetwork(nn.Module):
    def__init__(self):
        super(MyNeuralNetwork, self).__init__()
        self.input_hidden_layer = nn.Linear(8, 10)
        self.hidden_output_layer = nn.Linear(10, 1)

    def forward(self, x):
        x = self.input_hidden_layer(x)
        x = self.hidden_output_layer(x)
        return x
```

- Al igual que con Keras nos permite definir los modelos con mayor flexibilidad y complejidad.

Pytorch: definir el modelo con la API secuencial incluyendo función de activación

- Se incluyen en secuencia en el constructor secuencial
- Ejemplo: Agregamos las funciones de activación ReLu en la capa intermedia y tangente hiperbólica a la salida

```
def __init__(self):
        super(MyNeuralNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(8, 10),
            nn.ReLU(),
            nn.Linear(10, 1),
            nn.Tanh()
        )

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

Pytorch: definir el modelo con la API secuencial

```
# Testing our neural network
myNN = MyNeuralNetwork()
print(myNN)
     MyNeuralNetwork(
       (net): Sequential(
         (0): Linear(in_features=8, out_features=10, bias=True)
         (1): ReLU()
         (2): Linear(in_features=10, out_features=1, bias=True)
         (3): Tanh()
       )
     )
```

Pytorch: definir el modelo sin la API secuencial

- Se agregan las funciones de activación en la función forward.
- Ejemplo: Agregamos las funciones de activación ReLu en la capa intermedia y tangente hiperbólica a la salida

```
def___init__(self):
        super(MyNeuralNetwork, self).__init__()
        self.input_hidden_layer = nn.Linear(8, 10)
        self.hidden_output_layer = nn.Linear(10, 1)

    def forward(self, x):
        x = self.input_hidden_layer(x)
        x = torch.relu(x)
        x =
        self.hidden_output_layer(x)
        x = torch.tanh(x)
        return x
```

Pytorch: compilar el modelo

- Requiere definir una función de pérdida (loss function) para el entrenamiento (optimización de los pesos de la ANN).
- Requiere seleccionar un algoritmo para la optimización de los pesos.
- Requiere determinar métricas relevantes para monitorear el desempeño de la etapa de entrenamiento.

- En Pytorch: invocar una función para compilar el modelo con la configuración seleccionada.
- La compilación preparara las estructuras de datos requeridas para el uso eficiente del modelo definido.

Pytorch: compilar el modelo

- Se selecciona la función de pérdida o error (loss), en Pytorch se encuentran definidas en `torch.nn`.

  - `torch.nn.BCELoss` (binary cross entropy) para clasificación binaria
  - `torch.nn.CrossEntropyLoss` (cross entropy) para clasificación multiclase
  - `torch.nn.MSELoss` (mean squared error) para problemas de regresión

Para más información visitar:
https://www.machinecurve.com/index.php/2021/07/19/how-to-use-pytorch-loss-functions/

```
myNN_loss = nn.BCELoss() # For a binary classification problem
```

Pytorch: compilar el modelo

- Se seleccionar y parametriza el optimizador.
- En Pytorch los optimizadores se encuentran definidos en `torch.optim`.
  - `torch.optim.SGD()` implementa el Descenso de Gradiente Estocástico
  - `torch.optim.Adam()` implementa Adam

Para más información visitar:
https://velascoluis.medium.com/optimizadores-en-redes-neuronales-profundas-un-enfoque-pr%C3%A1ctico-819b39a3eb5

- Ambos tienen como parámetros los parámetros del modelo y el ratio de aprendizaje (*learning rate, lr*) que marcará el tamaño de los saltos en la optimización
- SGD incorpora el *momentum* que acelera el descenso en direcciones similares a las anteriores

```
myNN_optimizer = torch.optim.SGD(myNN.parameters(), lr=0.001, momentum=0.9)
```

Pytorch: acceso a los datos

- Pytorch ofrece una serie de clases para la gestión y el acceso a los datos.

- Primero se define el conjunto de datos a utilizar. Desde el paquete `torchvision` se tiene implementado el acceso directo a conjuntos de datos "conocidos". Los conjuntos de datos son de la clase `Dataset`
  https://pytorch.org/vision/stable/datasets.html

```
from torch.utils.data import Dataset

train = datasets.QMNIST('./data/', train=True, download=True,
transform=ToTensor())

testing_dataset = datasets.KMNIST('./data/', train=False,
download=True,   transform=ToTensor())
```

Pytorch: acceso a los datos

- Segundo se define un iterador para los datos o `DataLoader` que permite la lectura de los conjuntos de datos por lotes (batches). Para ello se crea un objeto de la clase DataLoader

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_dataset, batch_size=64,
shuffle=True)

test_dataloader = DataLoader(testing_dataset, batch_size=64,
shuffle=True)
```

- Con las variables del tipo DataLoader se puede iterar en un bucle

```
for features, target in train_dataloader:
```

Pytorch: acceso a los datos

- Si no se va a emplear un conjunto de datos ofrecido por torchvision, el usuario puede crear un conjunto de datos propio.
- Se crea a partir de una clase que hereda de Dataset.
- La clase creada debe tener tres funciones (como mínimo):

  `__init__(self)`: Constructor de la clase
  `__len__(self)`: Devuelve el tamaño (número de muestras) del conjunto de datos
  `__getitem__(self, idx)`: Devuelve una muestra en función del índice `idx`

- Esta clase podrá ser llamada para construir un DataLoader para iterar sobre las muestras

Pytorch: ajustar el modelo

- El modelo se va ajustar aplicando de forma interativa el algoritmo de aprendizaje en redes neuronales que realizará por un número de épocas de entrenamiento lo siguiente:

```
for input, target in dataset:
```
- Se realiza para cada batch o porción de datos del conjunto de datos del entrenamiento

```
    optimizer.zero_grad()
```
- Se inicializan los parámetros del optimizador

```
    output = myNN(input)
```
- Se calcula la salida de nuestra red neuronal

```
    loss = myNN_loss(output, target)
```
- Se calcula el error (la pérdida)

```
    loss.backward()
```
- Se propaga el error

```
    optimizer.step()
```
- Se aplica un paso del algoritmo de optimización

Antes de todo eso hay que poner el modelo en el estado de entrenamiento mediante `myNN.train()`

Pytorch: evaluar el modelo

- Requiere determinar un conjunto de datos de validación.
- Datos no utilizados en la etapa de entrenamiento, para evitar sesgos.
- El tiempo requerido para la evaluación es proporcional al tamaño del conjunto de datos de validación (mucho más rápido que el entrenamiento)

- En Pytorch, implica poner el modelo en modo de evaluación (`myNN.eval()`) y aplicar forward de los datos de entrada para obtener las salidas. Con estas salidas se pueden obtener valores como el de la función de pérdida o precisión del modelo.
  - Se pone el modelo en modo de evaluación para mejorar el tiempo y la memoria requerida para la inferencia (uso) del modelo, puesto que se deja de cargar artefactos solo útiles para el entrenamiento como el Dropout.

Pytorch: evaluar el modelo

- Requiere determinar un conjunto de datos de validación.
- Datos no utilizados en la etapa de entrenamiento, para evitar sesgos.
- El tiempo requerido para la evaluación es proporcional al tamaño del conjunto de datos de validación (mucho más rápido que el entrenamiento)

- En Pytorch, implica poner el modelo en modo de evaluación (`myNN.eval()`) y aplicar forward de los datos de entrada para obtener las salidas. Con estas salidas se pueden obtener valores como el de la función de pérdida o precisión del modelo.
  - Se pone el modelo en modo de evaluación para mejorar el tiempo y la memoria requerida para la inferencia (uso) del modelo, puesto que se deja de cargar artefactos solo útiles para el entrenamiento como el Dropout.

Pytorch: evaluar el modelo

```
myNN.eval()
test_loss = 0
test_acc = 0

for X_batch, y_batch in test_loader:
        y_pred = myNN(X_batch)
        loss = myNN_loss(y_pred, y_batch.unsqueeze(1))
        acc = binary_acc(y_pred, y_batch.unsqueeze(1))
        test_loss += loss.item()
        test_acc += acc.item()
```

Pytorch: predecir/clasificar/generar

- Validar el modelo sobre datos desconocidos
- Predicción: datos para los cuales no se conoce el valor a predecir.
- Clasificación: datos para los cuales no se conoce la clase a la que pertenecen.
- Generación: Crear nuevas imágenes, datos tabulares, series temporales, etc.