PyTorch

Última actualización, 11/07/2022.

Bibliografía en Información

- Aladdin Persson - YouTube

Índice

Anexo A: Terminología útil

Aquí se reúnen las palabras, términos, conceptos etc. claves a tener en cuenta a modo de recordatorio. No tiene ningún tipo de orden.

- **La letra ‘x’ se asocia a la variable independiente**, lo que usamos para hacer predicciones, por ejemplo imágenes. Mientras que la **letra ‘y’ se asocia a la variable dependiente**, lo que se denominan etiquetas y es nuestro objetivo obtener una predicción que tenga una alta probabilidad de parecerse a dicha ‘y’, un ejemplo de etiquetas pueden ser los nombres de las imágenes que permiten clasificar razas de perros.
- **weigth** = pesos: valores aleatorios con los que se inicializan a las neuronas, estos parámetros son fundamentales para determinar el tipo de funcionamiento de una red.
- w(tamaño input, número de neuronas)
- **bias** = sesgo:
    - b(1, número de neuronas)
- Un **set de datos de puede dividir** en:
    - Set de entrenamiento (train set).
    - Set de desarrollo o validación (dev set o validation set).
    - Set de pruebas (test set)
- Función de pérdida (**Loss function**): función que mide el rendimiento del modelo en uno de los ejemplos del set de entrenamiento.
- Función de coste (**Cost function**): función que mide el rendimiento del modelo en todos los ejemplos del set de entrenamiento. Sería la media de cada una de las funciones de pérdidas de cada ejemplo del set de entrenamiento.
- **Dataset**: una colección que devuelve un tuple de su variable independiente y dependiente para un solo elemento.

- **DataLoader**: un iterador que proporciona un flujo de grupos reducidos (mini-batches), donde cada grupo reducido es un tuple de un lote de variables independientes y un lote de variables dependientes.
- **one-hot encoding**: vectores de 0's con el tamaño del número de clases que tenga el dataset, cada categoría representa una posición en el vector por lo que si la imagen contiene algún elemento de alguna clase el vector tendrá un 1 en la posición del vector que corresponderá a su clase.
- Es importante saber que **una clasificación** pretende predecir una clase o categoría mientras que un modelo de **regresión** intenta predecir 1 o más cantidades numéricas.

Capítulo 0: Útiles

0.1. Guardar y cargar un modelo

Vamos a definir 2 funciones, una se encargará de guardar el estado del modelo y del optimizador pero podríamos guardar otros parámetros del modelo. Estas funciones son:

```
def guardar_checkpoint(checkpoint, filename = "checkpoint.pth.tar"):

  print("Guardando estado del modelo")
  torch.save(checkpoint, filename)

def cargar_checkpoint(checkpoint, modelo, optimizador):

  print("Cargando checkpoint")
  modelo.load_state_dict(checkpoint['estado_diccionario'])
  optimizador.load_state_dict(checkpoint['optimizador'])
```

"checkpoint.pth.tar" sería el nombre del fichero en el que se encuentra almacenados los parámetros guardados.

A la hora de definir los hiper-parámetros es recomendable declarar otra variable para indicar si queremos cargar un modelo o no:

```
# Hiperparámetros
num_clases = 10  # Para CIFAR-10 tenemos 10 clases
clasesbatch_size = 64
learning_rate = 1e-2
num_epocas = 40

# Cargar modelo
cargar_modelo = True
```

El bucle de entrenamiento podría ser similar al siguiente:

```
cont = 0

for epoca in range(num_epocas):

    correcto = 0
    perdidas = []

    # Para todas las epocas pares vamos a ir guardando el estado del modelo, por ejempo,
    # pero podríamos utilizar otra condición
    if epoca % 2 == 0:
```

```
        checkpoint = {'estado_diccionario' : modelo.state_dict(),
                      'optimizador' : optimizador.state_dict()}
        guardar_checkpoint(checkpoint)

    for batch_idx, (x, y) in enumerate(train_loader):

        x = x.to(device = device)
        y = y.to(device = device)

        if (cont == 0):

            print(x.shape)
            cont += 1

        # forward
        scores = modelo(x)
        loss = func_perdida(scores, y)

        perdidas.append(loss.item())

        # Backpropagation
        # Zero your gradients for every batch!
        # Ponemos el gradiente a 0 para cada batch para que no almacene las derivadas
        # de las anteriores capas
        optimizador.zero_grad()
        loss.backward()

        # Descenso del gradiente o pasos del algoritmo ADAM
        optimizador.step()

        #correcto += (scores == y).float().sum()
        _,pred_label = torch.max(scores, dim = 1)
        correcto += (pred_label == y).float().sum()

    # Podemos ajustar el learning rate conforme vamos entrenando el modelo,
    # inicialmente el modelo
    # requerirá pasos más grandes para aprender con mayor velocidad, conforme
    # se encuenre a un mínimo
    # los pasos deben reducirse con la intención de que no se aleje de dicho mínimo.
    # En este caso cada 8 épocas estaríamos reduciendo el learning rate 10
    #if epoca % 8 == 0:
    #
    #  learning_rate /= 10
    #  optimizador.param_groups[0]['lr'] = learning_rate

    # Usamos ahora el scheduler
    media_perdida = sum(perdidas) / len(perdidas)
    scheduler.step(media_perdida)

    print(
          f"""Epoca [{epoca+1}/{num_epocas}]
          \tPrecision {((100 * correcto) / len(train_dataset))}
          \tCoste {media_perdida}"""
    )
```

0.2. Utilizar un scheduler para el learning rate

En ocasiones durante el entrenamiento la función de pérdida se puede quedar estancada y el modelo, a pesar de seguir entrenando, no mejora. Realizando modificaciones en el learning rate podemos conseguir que el modelo mejore. En concreto, en algunos papers como el de VGG o GoogleLeNet utilizan un scheduler en el que reducen el learning rate en un factor de 10 **(learning_rate_actual = learning_rate_anterior * 0.1)** al reducir el learning rate conforme se avanza en el aprendizaje podemos mejorar el accuracy y reducir la función de pérdida.

Añadir el scheduler cuando definimos la función de pérdida y el optimizador

```
# Loss function, para clasificaciones de una sola etiqueta (y) como en este caso podemos utilizar Cross Entro
py Loss
func_perdida = nn.CrossEntropyLoss()

# No es recomendable utilizar Adam para entrenar VGG debido a la cantidad de parámetros que hay
# Vamos a utilizar los mismos parámetros que se nombran en el paper original
# optimizador = optim.Adam(params = modelo.parameters(), lr = learning_rate)
optimizador = optim.SGD(params = modelo.parameters(), lr = learning_rate,
                        momentum = 0.9)

# Podemos utilizar el lr_scheduler para ir reduciendo (en este caso) el learning rate conforme se avanza
# en el aprendizaje del modelo, en este caso se ha configurado con un factor de reduccion de 0.1 (si lr = 0.
1, nuevo_lr = 0.1 * 0.1 = 0.01)
# patience en el numero de epocas que podemos dejar sin que se mejores el modelo para esperar a reducir el le
arning rate
# Verbose = True, permite mostrar el cambio del learning rate realizado
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer = optimizador, mode = 'min',
                                                 factor = 0.1, patience = 7,
                                                 verbose = True)
```

Podemos fijarnos del punto anterior “0.1. Guardar y cargar un modelo” en el bucle empleado para el entrenamiento que estamos utilizando un scheduler que da pasos para según la función de coste.