S03.1: Aprendizaje Profundo I

Redes Neuronales y *Deep Learning*

Dr. Juan Bekios Calfa

Magíster en Inteligencia Artificial

UNIVERSIDAD ADOLFO IBÁÑEZ

Información de Contacto

- Juan Bekios Calfa
  - email: juan.bekios@edu.uai.cl, juan.bekios@ucn.cl
  - Web page: http://jbekios.ucn.cl
  - Teléfono: 235(5162) - 235(5125)

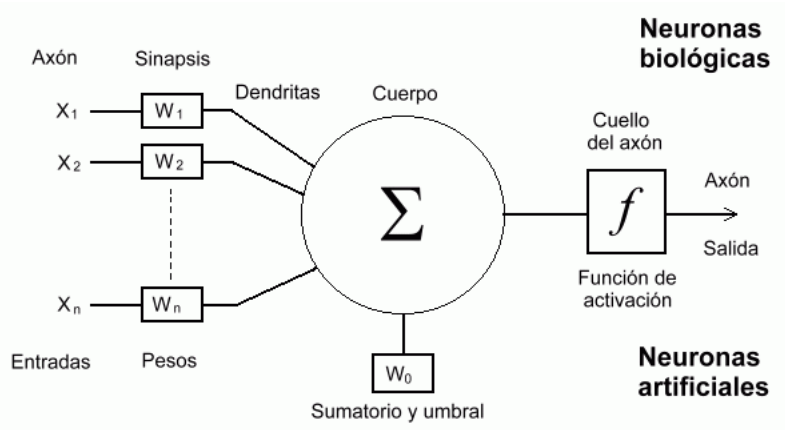
UNIVERSIDAD ADOLFO IBÁÑEZ

Contenidos

UNIVERSIDAD ADOLFO IBÁÑEZ

Contenidos

Modelo de una neurona simple



https://www.um.es/LEQ/Atmosferas/Ch-VI-3/F63s4p3.htm

Cálculo del gradiente del error

Buscamos cuanto varía el error de la función de pérdida ($\mathcal{L}$) con respecto al cambio de sus parámetros w (pesos sinápticos) y b bias (sesgo).

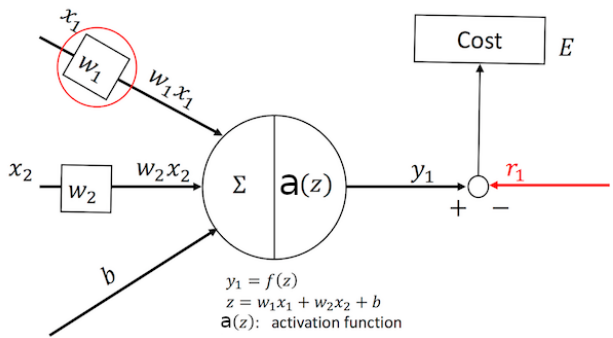
$$\frac{\partial \mathcal{L}(P)}{\partial P}, \; donde \; P \; son \; los \; parámetros$$

Eso significa que para calcular el cambio del error con respecto debemos calcular:

$$\frac{\partial \mathcal{L}(w)}{\partial w} = ?$$

$$\frac{\partial \mathcal{L}(b)}{\partial b} = ?$$

Funciones cálculo de una neurona



Funciones cálculo de una neurona

Si consideramos una red neuronal de L capas, las funciones a calcular en la última capa son:

- **Suma ponderada**: $z^L = w^L x + b^L$.
- **Función de activación**: $a(z^L)$.
- **Función de pérdida** (error en la predicción): $\mathcal{L}(a(z^L))$

¿Cómo calculamos el gradiente de la función de pérdida?

Para calcular el gradiente de $\frac{\partial \mathcal{L}(w)}{\partial w}$ y $\frac{\partial \mathcal{L}(b)}{\partial b}$ se utiliza la **regla de la cadena**:

$$\frac{\partial \mathcal{L}(w^L, a^L, z^L)}{\partial w^L} = \frac{\partial \mathcal{L}(a^L)}{\partial a^L} \frac{\partial a^L(z^L)}{\partial z^L} \frac{\partial z^L(w^L, x^L, b^L)}{\partial w^L}$$

$$\frac{\partial \mathcal{L}(w^L, a^L, z^L)}{\partial w^L} = \frac{\partial \mathcal{L}(a^L)}{\partial a^L} \frac{\partial a^L(z^L)}{\partial z^L} \frac{\partial z^L(w^L, x^L, b^L)}{\partial b^L}$$

Error cuadrático medio (última capa)

$$\mathcal{L}(a_j^L) = \frac{1}{2}\sum_j (y_i - a_j^L)^2$$

La derivada parcial de la función de pérdida con respecto a la función de activación será:

$$\frac{\partial \mathcal{L}(a^L)}{\partial a^L} = (a_j^L - y_i)$$

Función de activación

$$a^L(z^L) = \frac{1}{1 + e^{-z^L}}$$

La derivada parcial de la función de activación con respecto a la suma ponderada será:

$$\frac{\partial a^L(z^L)}{\partial z^L} = a^L z^L (1 - a^L(z^l))$$

Suma ponderada

$$z^L = \sum_i a_i^(L-1) w_i^L + b^L$$

La derivada parcial de la suma ponderada con respecto a los parámetros (w^L y b^L):

$$\frac{\partial z^L}{\partial w^L} = a_i^{L-1}$$

$$\frac{\partial z^L}{\partial b^L} = 1$$

Cálculo de gradientes para la función de pérdida

$$\frac{\partial \mathcal{L}(w^L)}{\partial a^L} = \frac{\partial \mathcal{L}}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial w^L}$$

Si consideramos que:

$$\delta^L = \frac{\partial \mathcal{L}}{\partial a^L}\frac{\partial a^L}{\partial z^L}$$

Las gradientes quedan:

$\frac{\partial \mathcal{L}}{\partial w^L} = \delta^L a_i^{L-1}$ y $\frac{\partial \mathcal{L}}{\partial b^L} = \delta^L$

Cálculo de gradientes capa interna (l)

$$\mathcal{L}(a^L(w^L a^{L-1}(w^{L-1}a^{L-2} + b^{L-1}) + b^L))$$

Sus derivadas parciales serían:

$$\frac{\partial \mathcal{L}(w^L)}{\partial a^L} = \frac{\partial \mathcal{L}}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial a^{L-1}}\frac{\partial a^{L-1}}{\partial z^{L-1}}\frac{\partial z^{L-1}}{\partial w^{L-1}}$$

$$\frac{\partial \mathcal{L}(w^L)}{\partial a^L} = \frac{\partial \mathcal{L}}{\partial a^L}\frac{\partial a^L}{\partial z^L}\frac{\partial z^L}{\partial a^{L-1}}\frac{\partial a^{L-1}}{\partial z^{L-1}}\frac{\partial z^{L-1}}{\partial b^{L-1}}$$

Algoritmo de *Backpropagation*

1. Cómputo del error de la última capa:

$$\delta^L = \frac{\partial \mathcal{L}}{\partial a^l} \frac{\partial a^l}{\partial z^l}$$

2. Repropagamos el error a la capa anterior:

$$\delta^{L-1} = w^l \delta^l \frac{\partial a^{l-1}}{\partial z^{l-1}}$$

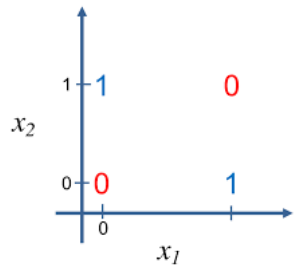
3. Calculamos las derivadas de la capa usando el error:

$$\frac{\partial \mathcal{L}}{\partial w^L} = \delta^{l-1} a_i^{l-2} \text{ y } \frac{\partial \mathcal{L}}{\partial b^L} = \delta^{l-1}$$

UNIVERSIDAD ADOLFO IBÁÑEZ

Contenidos

Problema XOR



Problema XOR

En PyTorch, una red neuronal se construye a partir de módulos. Los módulos pueden contener otros módulos, y una red neuronal también se considera un módulo en sí misma. La plantilla básica de un módulo es la siguiente:

```
class MyModule(nn.Module):

    def __init__(self):
        super().__init__()
        # Incialización de un módulo

    def forward(self, x):
        # Cálculo de la predicción del módulo
        pass
```

Problema XOR: Clasificador simple

En PyTorch, una red neuronal se construye a partir de módulos. Los módulos pueden contener otros módulos, y una red neuronal también se considera un módulo en sí misma. La plantilla básica de un módulo es la siguiente:

```
class ClasificadorSimple(nn.Module):

    def __init__(self, num_inputs, num_hidden, num_outputs):
        super().__init__()
        # Construir la red neuronal
        self.linear1 = nn.Linear(num_inputs, num_hidden)
        self.act_fn = nn.Tanh()
        self.linear2 = nn.Linear(num_hidden, num_outputs)

    ...
```

Problema XOR: Clasificador simple

```
class ClasificadorSimple(nn.Module):

    def __init__(self, num_inputs, num_hidden, num_outputs):
        super().__init__()
        # Initialize the modules we need to build the network
        self.linear1 = nn.Linear(num_inputs, num_hidden)
        self.act_fn = nn.Tanh()
        self.linear2 = nn.Linear(num_hidden, num_outputs)

    def forward(self, x):
        # Perform the calculation of the model to determine the prediction
        x = self.linear1(x)
        x = self.act_fn(x)
        x = self.linear2(x)
        return x
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Instanciar el clasificador simple

```
model = ClasificadorSimple(num_inputs=2, num_hidden=4,
    num_outputs=1)
# Imprimir el módulo y submódulos
print(model)
```

```
ClasificadorSimple(
  (lineal1): lineal (in_features = 2, out_features = 4, bias = verdadero)
  (act_fn): Tanh()
  (lineal2): lineal (in_features = 4, out_features = 1, bias = verdadero)
)
```

Instanciar el clasificador simple

```python
for name, param in model.named_parameters():
    print(f"Parameter {name}, shape {param.shape}")
```

```
Parameter linear1.weight, shape torch.Size([4, 2])
Parameter linear1.bias, shape torch.Size([4])
Parameter linear2.weight, shape torch.Size([1, 4])
Parameter linear2.bias, shape torch.Size([1])
```

Instanciar el clasificador simple

PyTorch también proporciona algunas funcionalidades para cargar los datos de entrenamiento y prueba de manera eficiente, resumidas en el paquete **torch.utils.data**.

```
1 import torch.utils.data as data
```

El paquete de datos define dos clases que son la interfaz estándar para manejar datos en PyTorch: **data.Dataset**, y **data.DataLoader**.
La **dataset** de datos proporciona una interfaz uniforme para acceder a los datos de prueba/entrenamiento.
El **data loader** se asegura de cargar y apilar de manera eficiente los puntos de datos del conjunto de datos en lotes durante el entrenamiento.

Dataset class

La clase de conjunto de datos resume la funcionalidad básica de un conjunto de datos de forma natural. Para definir un conjunto de datos en PyTorch, simplemente especificamos dos funciones: __getitem__, y __len__.

- La función **get-item** tiene que devolver el-ésimo punto de datos en el conjunto de datos.
- La función **len** devuelve el tamaño del conjunto de datos.

Para el conjunto de datos XOR, podemos definir la clase de conjunto de datos de la siguiente manera:

UNIVERSIDAD ADOLFO IBÁÑEZ

Dataset class

```
class XORDataset(data.Dataset):

    def __init__(self, size, std=0.1):
        ...

    def __len__(self):
        # Número de datos (puntos)
        ...

    def __getitem__(self, idx):
        # Obtener un dato específico indicado por idx
        ...
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Dataset class

```
class XORDataset(data.Dataset):

    def __init__(self, size, std=0.1):
        super().__init__()
        self.size = size
        self.std = std
        self.generar_puntos_xor()
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Dataset class (generar_puntos_xor())

```
class XORDataset(data.Dataset):

    def generar_puntos_xor(self):
        data = torch.randint(low=0, high=2, size=(self.size, 2), dtype=torch.
            float32)
        label = (data.sum(dim=1) == 1).to(torch.long)
        # Agregar ruido a los datos
        data += self.std * torch.randn(data.shape)

        self.data = data
        self.label = label
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Dataset class (len y getitem)

```
1  class XORDataset(data.Dataset):
2
3      def __len__(self):
4          self.data.shape[0], or self.label.shape[0]
5          return self.size
6
7      def __getitem__(self, idx):
8          data_point = self.data[idx]
9          data_label = self.label[idx]
10         return data_point, data_label
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Crear base de datos

```
dataset = XORDataset(size=200)
print("Size of dataset:", len(dataset))
print("Data point 0:", dataset[0])
```

```
Size of dataset: 200
Data point 0: (tensor([0.9632, 0.1117]), tensor(1))
```

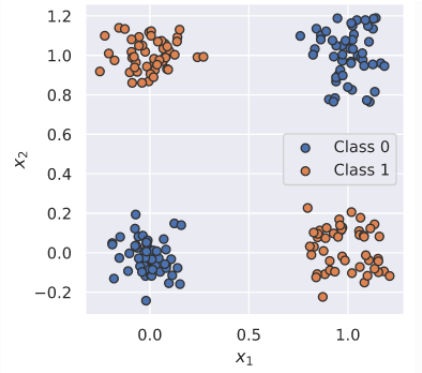
Visualizar los datos

```
def visualize_samples(data, label):
    if isinstance(data, torch.Tensor):
        data = data.cpu().numpy()
    if isinstance(label, torch.Tensor):
        label = label.cpu().numpy()
    data_0 = data[label == 0]
    data_1 = data[label == 1]

    plt.figure(figsize=(4,4))
    plt.scatter(data_0[:,0], data_0[:,1], edgecolor="#333", label="Class 0")
    plt.scatter(data_1[:,0], data_1[:,1], edgecolor="#333", label="Class 1")
    plt.title("Conjunto de datos de ejemplo")
    plt.ylabel(r"$x_2$")
    plt.xlabel(r"$x_1$")
    plt.legend()
```

Gráfica XOR

```
1 visualize_samples(dataset.data, dataset.label)
2 plt.show()
```



DataLoader class

La clase **torch.utils.data.DataLoader** representa un iterador de Python sobre un conjunto de datos con soporte para procesamiento por lotes automático, carga de datos multiproceso y muchas más funciones.

- **batch_size**: Número de muestras a apilar por lote.
- **shuffle**: si es True, los datos se devuelven en orden aleatorio. Esto es importante durante el entrenamiento para introducir la estocasticidad.
- **num_workers**: Número de subprocesos a utilizar para la carga de datos. El valor predeterminado, 0, solo un proceso.
- **pin_memory**: si es verdadero, el cargador de datos copiará los tensores en la memoria anclada de CUDA antes de devolverlos.
- **drop_last**: si es Verdadero, el último lote se elimina en caso de que sea más pequeño que el tamaño de lote especificado.

UNIVERSIDAD ADOLFO IBÁÑEZ

Instanciar un dataloader

```
data_loader = data.DataLoader(dataset, batch_size=8, shuffle=True)
```

```
print("Entrada de datos", data_inputs.shape, "\n", data_inputs)
print("Etiquetas de los datos", data_labels.shape, "\n", data_labels)
```

```
Entradas de datos torch.Size([8, 2])
 tensor([[-0.0890, 0.8608],
        [ 1.0905, -0.0128],
        [0,7967, 0,2268],
        [-0.0688, 0.0371],
        [0,8732, -0,2240],
        [-0.0559, -0.0282],
        [0,9277, 0,0978],
        [ 1.0150, 0.9689]])
Etiquetas de datos torch.Size([8])
 tensor([1, 1, 1, 0, 1, 0, 1, 0])
```

Optimización

Después de definir el modelo y el conjunto de datos, es hora de preparar la optimización del modelo. Durante el entrenamiento, realizaremos los siguientes pasos:

- Obtener un lote del cargador de datos.
- Obtener las predicciones del modelo para el lote.
- Calcule la pérdida en función de la diferencia entre predicciones y etiquetas.
- Backpropagation: calcula los gradientes de cada parámetro con respecto a la pérdida.
- Actualice los parámetros del modelo en la dirección de los gradientes.

Optimización (Módulo de Pérdida)

Podemos calcular la pérdida de un lote simplemente realizando algunas operaciones de tensor, ya que se agregan automáticamente al grafo de cálculo. Por ejemplo, para la clasificación binaria, podemos usar *Binary Cross Entropy* (*BCE*), que se define de la siguiente manera:

$$\mathcal{L}_{BCE} = -\sum_i [y_i \log x_i + (1 - y_i) \log(1 - x - I)]$$

```
loss_module = nn.BCEWithLogitsLoss()
```

Descenso del gradiente estocástico)

el paquete **torch.optim** que tiene implementados los optimizadores más populares.

```
optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
```

El optimizador proporciona dos funciones útiles: optimizer.step(), y optimizer.zero$_grad()$.

Entrenamiento

Finalmente, estamos listos para entrenar nuestro modelo. Como primer paso, creamos un conjunto de datos un poco más grande y especificamos un DataLoader con un tamaño de lote más grande.

```
train_dataset = XORDataset(size=2500)
train_data_loader = data.DataLoader(train_dataset,
   batch_size=128, shuffle=True)
```

Datos en GPU

Si el modelo es muy grande es recomendable cargarlo a GPU.

```
model.to(device)
```

```
ClasificadorSimple(
  (lineal1): lineal (in_features = 2, out_features = 4, sesgo = verdadero)
  (act_fn): Tanh()
  (lineal2): lineal (in_features = 4, out_features = 1, sesgo = verdadero)
)
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Programando entrenamiento

```python
def train_model(model, optimizer, data_loader, loss_module, num_epochs=100):
    # Set model to train mode
    model.train()

    # Training loop
    for epoch in tqdm(range(num_epochs)):
        for data_inputs, data_labels in data_loader:
            ## Step 1: Seleccionar dispositivo donde se procesaron los datos
            data_inputs = data_inputs.to(device)
            data_labels = data_labels.to(device)
            ## Step 2: Correr el modelo sobre los datos
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1)
            ## Step 3: Calcula la pérdida
            loss = loss_module(preds, data_labels.float())
            ## Step 4: Calcular backpropagation
            # Antes ejecutar el cálculo de gradiente, asegurarse que están en cero
            optimizer.zero_grad()
            # Ejecutar backpropagation
            loss.backward()
            ## Step 5: Actualizar los parámetros
            optimizer.step()
```

Guardar el modelo

model.state_dict() contiene los parámetros aprendidos.

```
state_dict = model.state_dict()
print(state_dict)
```

```
OrderedDict([('linear1.weight', tensor([[-2.6034, -3.3293],
        [ 1.9773, -2.4074],
        [-2.5968, -1.5909],
        [-0.5714, -0.8096]], device='cuda:0')), ('linear1.bias', tensor([ 1.4459, -1.3992,  2.9883, -0.1376],
             device='cuda:0')), ('linear2.weight', tensor([[-4.4624,  3.0882,  4.4031, -0.1372]], device='cuda
             :0')), ('linear2.bias', tensor([-1.6854], device='cuda:0'))])
```

Guardar el modelo

model.state_dict() contiene los parámetros aprendidos.

```
orch.save(state_dict, "mi_modelo.tar")
```

```
# Cargar el modelo
state_dict = torch.load("our_model.tar")

# Crear un nuevo modelo y cargar su estado
new_model = ClasificadorSimple(num_inputs=2, num_hidden=4, num_outputs=1)
new_model.load_state_dict(state_dict)
```

Evaluación

Una vez que hemos entrenado un modelo, es hora de evaluarlo en un conjunto de prueba extendido. Como nuestro conjunto de datos consta de puntos de datos generados aleatoriamente, primero debemos crear un conjunto de prueba con un cargador de datos correspondiente.

```
test_dataset = XORDataset(size=500)
test_data_loader = data.DataLoader(test_dataset, batch_size=128, shuffle=False,
    drop_last=False)
```

Como métrica utilizaremos la **tasa de acierto**:

$$acc = \frac{Predicciones\ correctas}{Todas\ las\ predicciones} = \frac{TP + TN}{TP + TN + FP + FN}$$

Implementación para evaluación del modelo

```python
def eval_model(model, data_loader):
    model.eval() # Set model to eval mode
    true_preds, num_preds = 0., 0.
    with torch.no_grad(): # Desactivar los gradientes del código
        for data_inputs, data_labels in data_loader:

            data_inputs, data_labels = data_inputs.to(device), data_labels.to(
                device)
            preds = model(data_inputs)
            preds = preds.squeeze(dim=1)
            preds = torch.sigmoid(preds)
            pred_labels = (preds >= 0.5).long() # Binarizar las predicciones

            true_preds += (pred_labels == data_labels).sum()
            num_preds += data_labels.shape[0]
    acc = true_preds / num_preds
    print(f"Tasa de acierto del modelo: {100.0*acc:4.2f}%")
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Ejecutar la evaluación del modelo

```
eval_model(model, test_data_loader)
Precisión del modelo: 100,00%
```

UNIVERSIDAD ADOLFO IBÁÑEZ

Referencias

- **Pattern Recognition and Machine Learning.** Christopher M. Bishop. Springer. 2006.
- **The Elements of Statistical Learning: Data Mining, Inference, and Prediction.** Trevor Hastie, Robert Tibshirani, Jerome Friedman. Springer. 2016.
- **Practical Data Science: Deep learning**. J. Zico Kolter. `http://www.datasciencecourse.org/slides/deep_learning.pdf`
- **A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way**. Sumit Saha. `https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53`

UNIVERSIDAD ADOLFO IBÁÑEZ

¿Preguntas?