Inteligencia Artificial

PyTorch

Prof. Wladimir Rodríguez

wladimir.rodriguez@outlook.com

Departamento de Computación

¿Qué es PyTorch (www.pytorch.org)?

PyTorch es una librería de código abierto del lenguaje Python, desarrollada por Facebook en 2016 para ser usada en el ámbito del aprendizaje automático.

A los investigadores de aprendizaje automático les encanta usar PyTorch. Y a partir de febrero de 2022, PyTorch es la libreria de aprendizaje profundo más utilizado en Papers With Code (https://paperswithcode.com/trends), un sitio web que realizar un seguimiento de los trabajos de investigación de aprendizaje automático y los repositorios de código adjuntos.

PyTorch también ayuda a encargarse de muchas cosas, como la aceleración de GPU (haciendo que su código se ejecute más rápido) detrás de escena.

Así que puedes concentrarte en manipular datos y escribir algoritmos y PyTorch se asegurará de que funcione rápido.

Tensores

Al igual que un vector puede considerarse como una matriz, o una lista, de escalares (números ordinarios como 1, 2 y PI), y las matrices pueden considerarse matrices de vectores, un tensor puede considerarse como un matriz de matrices. Entonces un tensor es realmente solo una matriz n-dimensional. Resulta, como veremos en los ejemplos de codificación, que esta arquitectura tiene mucho sentido cuando se trabaja con el aprendizaje automático.

En TensorFlow, los tensores se describen mediante una unidad de dimensionalidad conocida como orden. El orden de Tensor no es lo mismo que el orden de la matriz. El orden del tensor es el número de dimensiones del tensor.

Un tensor de orden dos es lo que típicamente pensamos como una matriz, un tensor de orden uno es un vector. Para un tensor de orden dos, puede acceder a cualquier elemento con la sintaxis `t[i, j]` . Para un tensor de orden tres, los elementos de direccionan con `t[i, j, k]` .

Rank	Math entity	Python example
0	Scalar (magnitude only)	`s = 483`
1	Vector (magnitude and direction)	`v = [1.1, 2.2, 3.3]`
2	Matrix (table of numbers)	`m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
3	3-Tensor (cube of numbers)	`t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]`
n	n-Tensor (you get the idea)	`....`

Los tensores son similares a los `ndarrays` de NumPy, con la adición de que los tensores también se pueden usar en una GPU para acelerar la computación.

In [2]:
```
import torch
import torch.optim as optim
import torch.nn as nn
from torchviz import make_dot
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
from time import time
%matplotlib inline
torch.__version__
```

Out[2]:
```
'1.11.0'
```

Construya una matriz de 5x3, sin inicializar:

In [3]:
```
x = torch.empty(5, 3)
x
```

Out[3]:
```
tensor([[9.1837e-39, 4.6837e-39, 9.9184e-39],
        [9.0000e-39, 1.0561e-38, 1.0653e-38],
        [4.1327e-39, 8.9082e-39, 9.8265e-39],
        [9.4592e-39, 1.0561e-38, 1.0010e-38],
        [1.0653e-38, 9.9184e-39, 1.0653e-38]])
```

Construya una matriz inicializada al azar:

In [4]:
```
x = torch.rand(5, 3)
x
```

Out[4]:
```
tensor([[0.3149, 0.4504, 0.1149],
        [0.1886, 0.4378, 0.1414],
        [0.0667, 0.5420, 0.6295],
        [0.6473, 0.5284, 0.0769],
        [0.4299, 0.5514, 0.9434]])
```

Construya una matriz llena de ceros y de `dtype long` :

In [5]:
```
x = torch.zeros(5, 3, dtype=torch.long)
x
```

Out[5]:
```
tensor([[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]])
```

Construya un tensor directamente de los datos:

In [6]:
```
x = torch.tensor([5.5, 3])
x
```

Out[6]:
```
tensor([5.5000, 3.0000])
```

o crear un tensor basado en un tensor existente. Estos métodos reutilizarán las propiedades del tensor de entrada, a menos que el usuario proporcione nuevos valores

In [7]:
```
x = x.new_ones(5, 3, dtype=torch.double)    # los métodos new_* toman tamaños
print(x)
x = torch.rand_like(x, dtype=torch.float)   # cambiar dtype!
x                                           # el resultado tiene el mismo tamaño
```

```
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], dtype=torch.float64)
```

Out[7]:
```
tensor([[0.6074, 0.4311, 0.3353],
        [0.9136, 0.8167, 0.6141],
        [0.4315, 0.7922, 0.9843],
        [0.6327, 0.3538, 0.0382],
        [0.8936, 0.3679, 0.0344]])
```

Obtener su tamaño:

In [8]:
```
x.size()
```

Out[8]:
```
torch.Size([5, 3])
```

Nota

> `torch.Size` es de hecho una tupla, por lo que admite todas las operaciones de tupla.

Operaciones

Existen múltiples sintaxis para las operaciones. En el siguiente ejemplo, veremos la operación de adición.

Adición: sintaxis 1

In [9]:
```
y = torch.rand(5, 3)
x + y
```

Out[9]:
```
tensor([[1.3422, 0.5997, 0.5574],
        [1.3826, 1.0366, 0.9593],
        [0.9793, 1.1387, 1.1828],
        [0.8921, 0.8445, 0.9560],
        [1.5759, 1.3017, 0.4394]])
```

Adición: sintaxis 2

In [10]:
```
torch.add(x, y)
```

Out[10]:
```
tensor([[1.3422, 0.5997, 0.5574],
        [1.3826, 1.0366, 0.9593],
        [0.9793, 1.1387, 1.1828],
        [0.8921, 0.8445, 0.9560],
        [1.5759, 1.3017, 0.4394]])
```

Adición: proporcionar un tensor de salida como argumento

In [11]:
```
resultado = torch.empty(5, 3)
torch.add(x, y, out=resultado)
resultado
```

Out[11]:
```
tensor([[1.3422, 0.5997, 0.5574],
        [1.3826, 1.0366, 0.9593],
        [0.9793, 1.1387, 1.1828],
        [0.8921, 0.8445, 0.9560],
        [1.5759, 1.3017, 0.4394]])
```

Adición: en sitio

In [12]:
```
y.add_(x)
y
```

Out[12]:
```
tensor([[1.3422, 0.5997, 0.5574],
        [1.3826, 1.0366, 0.9593],
        [0.9793, 1.1387, 1.1828],
        [0.8921, 0.8445, 0.9560],
        [1.5759, 1.3017, 0.4394]])
```

Nota

> Cualquier operación que cambie un tensor en el lugar se le coloca el postfijo _. Por ejemplo: *x.copy_(y)*, *x.t_()*, cambiará *x*.

¡Puede usar la indexación NumPy estándar!

In [13]:
```
x[:, 1]
```

Out[13]:
```
tensor([0.4311, 0.8167, 0.7922, 0.3538, 0.3679])
```

Cambio de tamaño: si desea cambiar el tamaño / remodelar el tensor, puede usar `torch.view` :

In [14]:
```
x = torch.rand(4, 4)
y = x.view(16)
z = x.view(-1, 8)
print(x.size(), y.size(), z.size())
```

```
torch.Size([4, 4]) torch.Size([16]) torch.Size([2, 8])
```

Si tiene un tensor de un elemento, use `.item()` para obtener el valor como un número de Python

In [15]:
```
x = torch.randn(1)
print(x)
x.item()
```

```
tensor([-0.7317])
```

Out[15]:
```
-0.731715977191925
```

Leer mas tarde:

> Aquí (https://pytorch.org/docs/stable/torch.html) se describen más de 100 operaciones de tensor, que incluyen transposición, indexación, corte, operaciones matemáticas, álgebra lineal, números aleatorios, etc.

Puente NumPy

Convertir un Tensor Torch en una matriz NumPy y viceversa es muy fácil.

El Tensor Torch y la matriz NumPy compartirán sus ubicaciones de memoria subyacentes (si el Tensor Torch está en la CPU), y cambiar una cambiará la otra.

Convertir un Tensor Torch en una matriz NumPy

In [16]:
```
a = torch.ones(5)
a
```

Out[16]:
```
tensor([1., 1., 1., 1., 1.])
```

In [17]:
```
b = a.numpy()
b
```

Out[17]:
```
array([1., 1., 1., 1., 1.], dtype=float32)
```

Vea cómo la matriz numpy cambió de valor.

In [18]:
```
a.add_(1)
print(a)
print(b)
```

```
tensor([2., 2., 2., 2., 2.])
[2. 2. 2. 2. 2.]
```

Conversión de matriz NumPy a Tensor Torch

Vea cómo cambiar la matriz NumPy cambió el Tensor Torch automáticamente

In [19]:
```
a = np.ones(5)
b = torch.from_numpy(a)
np.add(a, 1, out=a)
print(a)
print(b)
```

```
[2. 2. 2. 2. 2.]
tensor([2., 2., 2., 2., 2.], dtype=torch.float64)
```

Todos los tensores en la CPU excepto un `CharTensor` admiten la conversión a NumPy y viceversa.

Tensores CUDA

Los tensores se pueden mover a cualquier dispositivo utilizando el método `.to` .

In [20]:
```
# ejecutemos esta celda solo si CUDA está disponible
# Usaremos objetos ``torch.device`` para mover los tensores dentro y fuera de la GPU
if torch.cuda.is_available():
    device = torch.device("cuda")          # un objeto dispositivo CUDA
    y = torch.ones_like(x, device=device)  # crear directamente un tensor en la GPU
    x = x.to(device)                       # o usar la cadena ``.to("cuda")``
    z = x + y
    print(z)
    print(z.to("cpu", torch.double))
```

Un problema de regresión simple

Vamos a comenzar a utilizar `PyTorch` con un problema sencillo de regresión lineal con una sola característica x.

$$y = a + bx + \epsilon$$

Generacion de datos

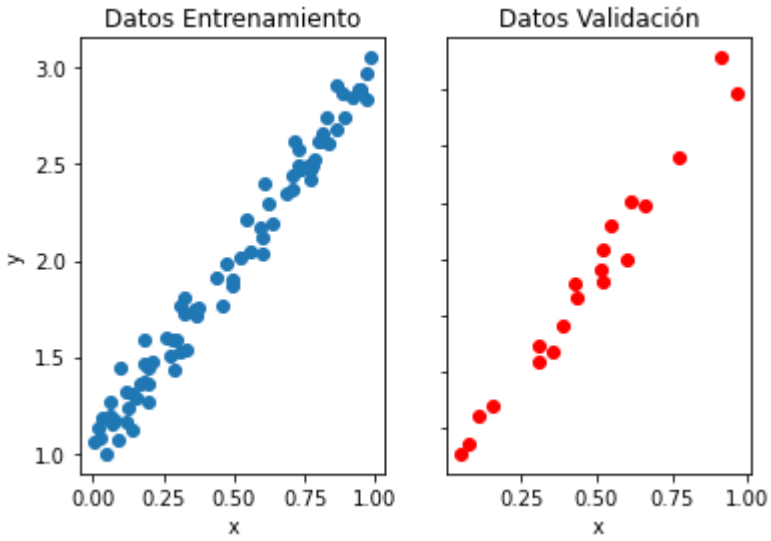
Comencemos generando algunos datos sintéticos: comenzamos con un vector de 100 puntos para nuestra característica x y creamos nuestras etiquetas usando $a = 1$, $b = 2$ y algo de ruido gaussiano.

A continuación, dividamos nuestros datos sintéticos en conjuntos de entrenamiento y validación, barajando la matriz de índices y utilizando los primeros 80 puntos barajados para el entrenamiento.

In [21]:
```
# Generación de Datos
np.random.seed(42)
x = np.random.rand(100, 1)
y = 1 + 2 * x + .1 * np.random.randn(100, 1)

# Barajar los indices
indices = np.arange(100)
np.random.shuffle(indices)

# Usar los primeros 80 indices aleatorios para entrenamiento
indices_entrenamiento = indices[:80]
# Usar los indices restantes para validación
indices_validacion = indices[80:]

# Generar conjuntos de entrenamiento y validación
x_entrenamiento, y_entrenamiento = x[indices_entrenamiento], y[indices_entrenamiento]
x_validacion, y_validacion = x[indices_validacion], y[indices_validacion]
```

Visualización de los conjuntos de datos

In [22]:
```
figura, axs = plt.subplots(1, 2)
axs[0].scatter(x_entrenamiento, y_entrenamiento)
axs[0].set(xlabel='x', ylabel='y')
axs[0].set_title('Datos Entrenamiento')
axs[1].scatter(x_validacion, y_validacion, c='red')
axs[1].set(xlabel='x', ylabel='y')
axs[1].set_title('Datos Validación')
for ax in axs.flat:
    ax.label_outer()
```



Cargando datos, dispositivos y CUDA

Para convertir las matrices Numpy a tensores Pytorch usaremos `from_numpy` . Sin embargo, este método devuelve un tensor de CPU.

Si queremos utilizar la GPU podemos usar el método `to()` . Envía un tensor a cualquier dispositivo que especifique, incluida la GPU (denominada `cuda` o `cuda:0`).

Para hacer que el código use la CPU en caso de que no tengamos disponible una GPU se puede usar `cuda.is_available()` para averiguar si tiene una GPU a su disposición y configura tu dispositivo en

consecuencia.

También puede convertirlo fácilmente a una precisión menor (flotante de 32 bits) utilizando `float()` .

In [23]:
```
dispositivo = 'cuda' if torch.cuda.is_available() else 'cpu'

# Nuestros datos estaban en matrices Numpy, pero tenemos que transformarlos en los tensores
# y luego los enviamos al dispositivo elegido
x_entrenamiento_tensor = torch.from_numpy(x_entrenamiento).float().to(dispositivo)
y_entrenamiento_tensor = torch.from_numpy(y_entrenamiento).float().to(dispositivo)

x_validacion_tensor = torch.from_numpy(x_validacion).float().to(dispositivo)
y_validacion_tensor = torch.from_numpy(y_validacion).float().to(dispositivo)
# Aquí podemos ver la diferencia: observe que .type() es más útil
# ya que también nos dice DÓNDE está el tensor (device)
print(type(x_entrenamiento), type(x_entrenamiento_tensor), x_entrenamiento_tensor.type())
```

```
<class 'numpy.ndarray'> <class 'torch.Tensor'> torch.FloatTensor
```

Si comparamos los tipos de ambas variables, obtendrá lo que esperaría: `numpy.ndarray` para la primera y `torch.Tensor` para la segunda.

El tipo de dato de un tensor de GPU sería `torch.cuda.FloatTensor` .

También podemos dar la vuelta, volviendo los tensores a matrices Numpy, usando `numpy()` . Debería ser fácil como `x_train_tensor.numpy()` pero ...

> TypeError: no se puede convertir el tensor CUDA en numpy. Use `Tensor.cpu()` para copiar primero el tensor a la memoria del host.

Desafortunadamente, Numpy no puede manejar los tensores de GPU ... primero debe hacer que sean tensores de CPU usando `cpu()` .

Creando Parámetros

¿Qué distingue un tensor utilizado para datos, como los que acabamos de crear, de un tensor utilizado como parámetro / peso (entrenable)?

Los últimos tensores requieren el cálculo de sus gradientes, por lo que podemos actualizar sus valores (los valores de los parámetros, es decir). Para eso es necesario el argumento `require_grad = True` . Le dice a PyTorch que queremos que calcule gradientes para nosotros.

In [45]:
```
#Podemos crear tensores regulares y enviarlos al dispositivo (como hicimos con nuestros dat
a = torch.randn(1, dtype=torch.float).to(dispositivo)
b = torch.randn(1, dtype=torch.float).to(dispositivo)
# y ENTONCES los fijamos como que requieren gradientes ...
a.requires_grad_()
b.requires_grad_()
print(a, b)
```

```
tensor([-0.3114], requires_grad=True) tensor([1.4280], requires_grad=True)
```

Primero enviamos nuestros tensores al dispositivo y luego usamos el método `require_grad_()` para establecer su require_grad en True en su lugar.

```
TERCERO
tensor([0.5771], device='cuda:0', requires_grad=True)
tensor([1.6813], device='cuda:0', requires_grad=True)
```

En PyTorch, cada método que termina con un guión bajo (_) realiza cambios in situ, lo que significa que modificarán la variable subyacente.

In [25]:
```
# Podemos especificar el dispositivo en el momento de la creación. ¡RECOMENDADO!
torch.manual_seed(42)
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=dispositivo)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=dispositivo)
print(a, b)
```

```
tensor([0.3367], requires_grad=True) tensor([0.1288], requires_grad=True)
```

Autograd

Autograd es el paquete de diferenciación automática de PyTorch. Gracias a ello, no necesitamos preocuparnos por las derivadas parciales, regla de la cadena ni nada por el estilo.

Entonces, ¿cómo le decimos a PyTorch que haga lo suyo y calcule todos los gradientes? Para eso sirve el método `backward()`.

¿Recuerdas el punto de partida para calcular los gradientes? Fue la función de pérdida, ya que calculamos sus derivadas parciales con respecto a nuestros parámetros. Por lo tanto, debemos invocar el método `backward()` desde la variable Python correspondiente, como `perdida.backward()`.

¿Qué pasa con los valores reales de los gradientes? Podemos inspeccionarlos mirando el atributo `grad` de un tensor.

Si revisa la documentación del método, indica claramente que los gradientes se acumulan. Entonces, cada vez que usamos los gradientes para actualizar los parámetros, necesitamos poner a cero los gradientes después. Y para eso sirve `zero_grad()`.

Entonces, abandonemos el cálculo manual de gradientes y usemos los métodos `backward()` y `zero_grad()` en su lugar.

Optimizador

Utilizamos uno de los optimizadores de PyTorch (https://pytorch.org/docs/stable/optim.html), como SGD o Adam.

Un optimizador toma los parámetros que queremos actualizar, la tasa de aprendizaje que queremos usar (¡y posiblemente muchos otros hiperparámetros también!) Y realiza las actualizaciones a través de su método `step()`.

Además, ya no necesitamos poner a cero los gradientes uno por uno. ¡Solo invocamos el método `zero_grad()` del optimizador y eso es todo!

No se deje engañar por el nombre del optimizador: si utilizamos todos los datos de entrenamiento a la vez para la actualización, como lo estamos haciendo en el código, el optimizador está realizando un descenso de gradiente por lotes, a pesar de su nombre.

Pérdida

Ahora abordamos el cálculo de pérdidas. Como se esperaba, PyTorch nos cubrió una vez más. Hay muchas funciones de pérdida (https://pytorch.org/docs/stable/nn.html#loss-functions) para elegir, dependiendo de la tarea en cuestión. Como el nuestro es una regresión, estamos utilizando la pérdida del error cuadrático medio (MSE).

Tenga en cuenta que `nn.MSELoss` *en realidad crea una función de pérdida para nosotros, NO es la función de pérdida en sí. Además, puede especificar un método de reducción para aplicar, es decir, cómo desea agregar los resultados para puntos individuales: puede promediarlos* `(reduction='mean')` *o simplemente sumarlos* `(reduction='sum')` .

Luego usamos la función de pérdida creada más adelante, para calcular la pérdida dadas nuestras predicciones y nuestras etiquetas.

Modelo

En PyTorch, un modelo está representado por una clase normal de Python que hereda de la clase `Module` .

Los métodos más fundamentales que necesita implementar son:

- `__init __(self)` : define las partes que componen el modelo, en nuestro caso, dos parámetros, a y b.

> Sin embargo, no está limitado a definir parámetros ... los modelos también pueden contener otros modelos (o capas) como sus atributos, por lo que puede anidarlos fácilmente. Veremos un ejemplo de esto en breve también.

- `forward(self, x)` : realiza el cálculo real, es decir, genera una predicción, dada la entrada x.

> Sin embargo, NO debe llamar al método `forward(x)` . Debe llamar a todo el modelo en sí, como en el `modelo(x)` para realizar un pase hacia adelante y predecir la salida.

Construyamos un modelo adecuado (pero simple) para nuestra tarea de regresión. Debe tener un aspecto como este:

In [75]:
```
class RegresionLinealManual(nn.Module):
    def __init__(self):
        super().__init__()
        # Para hacer que "a" y "b" sean parámetros reales del modelo, necesitamos envolverl
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Calcula las salidas / predicciones
        return self.a + self.b * x
```

En el método `__init__` , definimos nuestros dos parámetros, a y b, usando la clase `Parameter()` , para decirle a PyTorch que estos tensores deben considerarse parámetros del modelo del que son un atributo.

¿Por qué debería importarnos eso? Al hacerlo, podemos usar el método `parameter()` de nuestro modelo para recuperar un iterador sobre todos los parámetros del modelo, incluso aquellos parámetros de modelos anidados, que podemos usar para alimentar nuestro optimizador (¡en lugar de crear una lista de parámetros

nosotros mismos!).

Además, podemos obtener los valores actuales para todos los parámetros utilizando el método `state_dict()` de nuestro modelo.

> IMPORTANTE: necesitamos enviar nuestro modelo al mismo dispositivo donde están los datos. Si nuestros datos están hechos de tensores de GPU, nuestro modelo también debe "vivir" dentro de la GPU.

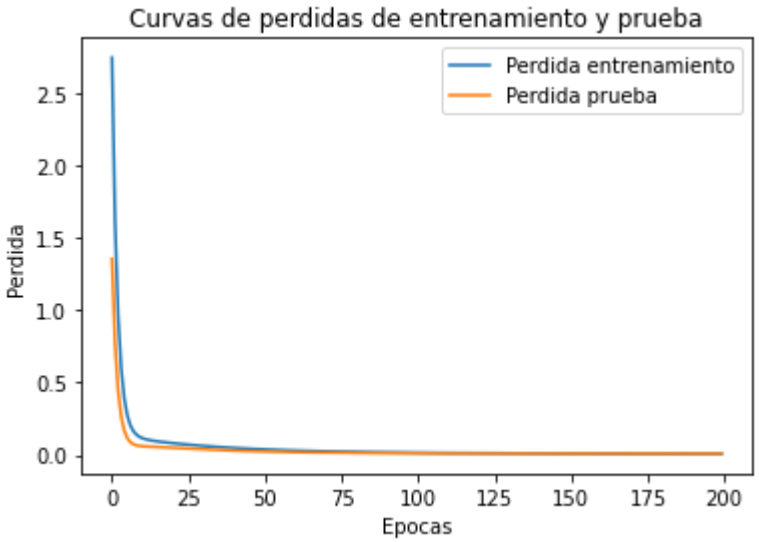
Podemos usar todos estos útiles métodos para cambiar nuestro código, que debería verse así:

In [77]:
```
torch.manual_seed(42)

# Crear lista vacias para guardar los valores de las perdidas
perdidas_entrenamiento = []
perdidas_prueba = []
conteo_epocas = []

# Ahora podemos crear un modelo y enviarlo de inmediato al dispositivo
modelo_0 = RegresionLinealManual().to(dispositivo)
# También podemos inspeccionar sus parámetros usando su state_dict
print(modelo_0.state_dict())

# Fijar hiperparámetros
tasa_aprendizaje = 1e-1
n_epocas = 200

# Definir la función de perdida
perdida_fn = nn.MSELoss(reduction='mean')

#Defimir el optimizador a utilizar
optimizador = optim.SGD(modelo_0.parameters(), lr=tasa_aprendizaje)

for epoca in range(n_epocas):
    ### Entrenamiento

    # Poner el modelo en modo de entrenamiento (este es el estado predeterminado de un mode
    modelo_0.train()

    # 1. Propagar hacia adelante los datos de entrenamiento usando el método forward() dent
    y_sombrero = modelo_0(x_entrenamiento_tensor)

    # 2. Calcule la pérdida (qué tan diferentes son las predicciones de nuestros modelos co
    perdida = perdida_fn(y_entrenamiento_tensor, y_sombrero)

    # 3. Cero los gradientes del optimizador
    optimizador.zero_grad()

    # 4. Propagación hacia atrás
    perdida.backward()

    # 5. Realizar paso de optimización
    optimizador.step()

    ### Prueba

    # Colocar el modelo en modo evaluación
    modelo_0.eval()

    with torch.inference_mode():
        # 1. Propagar hacia adelante los datos de prueba
        prediccion_prueba = modelo_0(x_validacion_tensor)

        # 2. Calcular la perdida sobre la data de prueba
        perdida_prueba = perdida_fn(prediccion_prueba, y_validacion_tensor)

        conteo_epocas.append(epoca)
        perdidas_entrenamiento.append(perdida.detach().numpy())
        perdidas_prueba.append(perdida_prueba.detach().numpy())

        # Imprimir las perdidas cada 10 epocas
        if epoca % 10 == 0:
```

```
            print(f"Epoca: {epoca} | MSE Perdida Entrenamiento: {perdida:.4f} | MSE Perdida

print(modelo_0.state_dict())
```

```
OrderedDict([('a', tensor([0.3367])), ('b', tensor([0.1288]))])
Epoca: 0 | MSE Perdida Entrenamiento: 2.7475 | MSE Perdida Prueba: 1.3556
Epoca: 10 | MSE Perdida Entrenamiento: 0.1146 | MSE Perdida Prueba: 0.0593
Epoca: 20 | MSE Perdida Entrenamiento: 0.0803 | MSE Perdida Prueba: 0.0482
Epoca: 30 | MSE Perdida Entrenamiento: 0.0613 | MSE Perdida Prueba: 0.0372
Epoca: 40 | MSE Perdida Entrenamiento: 0.0473 | MSE Perdida Prueba: 0.0289
Epoca: 50 | MSE Perdida Entrenamiento: 0.0370 | MSE Perdida Prueba: 0.0229
Epoca: 60 | MSE Perdida Entrenamiento: 0.0294 | MSE Perdida Prueba: 0.0185
Epoca: 70 | MSE Perdida Entrenamiento: 0.0238 | MSE Perdida Prueba: 0.0154
Epoca: 80 | MSE Perdida Entrenamiento: 0.0197 | MSE Perdida Prueba: 0.0132
Epoca: 90 | MSE Perdida Entrenamiento: 0.0166 | MSE Perdida Prueba: 0.0116
Epoca: 100 | MSE Perdida Entrenamiento: 0.0144 | MSE Perdida Prueba: 0.0105
Epoca: 110 | MSE Perdida Entrenamiento: 0.0127 | MSE Perdida Prueba: 0.0097
Epoca: 120 | MSE Perdida Entrenamiento: 0.0115 | MSE Perdida Prueba: 0.0092
Epoca: 130 | MSE Perdida Entrenamiento: 0.0106 | MSE Perdida Prueba: 0.0088
Epoca: 140 | MSE Perdida Entrenamiento: 0.0099 | MSE Perdida Prueba: 0.0086
Epoca: 150 | MSE Perdida Entrenamiento: 0.0094 | MSE Perdida Prueba: 0.0085
Epoca: 160 | MSE Perdida Entrenamiento: 0.0091 | MSE Perdida Prueba: 0.0084
Epoca: 170 | MSE Perdida Entrenamiento: 0.0088 | MSE Perdida Prueba: 0.0083
Epoca: 180 | MSE Perdida Entrenamiento: 0.0086 | MSE Perdida Prueba: 0.0083
Epoca: 190 | MSE Perdida Entrenamiento: 0.0085 | MSE Perdida Prueba: 0.0083
OrderedDict([('a', tensor([1.0523])), ('b', tensor([1.9127]))])
```

In [78]:
```
# Graficar las curvas de perdidas
plt.plot(conteo_epocas, perdidas_entrenamiento, label="Perdida entrenamiento")
plt.plot(conteo_epocas, perdidas_prueba, label="Perdida prueba")
plt.title("Curvas de perdidas de entrenamiento y prueba")
plt.ylabel("Perdida")
plt.xlabel("Epocas")
plt.legend();
```



Red Neuronal para reconocer los digitos de MNIST

In [35]:
```
# Definir una transformación para normalizaz la data
transformacion = transforms.Compose([transforms.ToTensor(),
                                     transforms.Normalize((0.5,), (0.5,)),
                                     ])

# Bajar y cargar la data de entrenamiento
datos_entrenamiento = datasets.MNIST('../datos/MNIST_data/', download=True, train=True, tra
datos_validacion = datasets.MNIST('../datos/MNIST_data/', download=True, train=False, trans
cargador_entrenamiento = torch.utils.data.DataLoader(datos_entrenamiento, batch_size=64, sh
cargador_validacion = torch.utils.data.DataLoader(datos_validacion, batch_size=64, shuffle=
```

Explorar la data

Veamos la forma de las imágenes y las etiquetas.

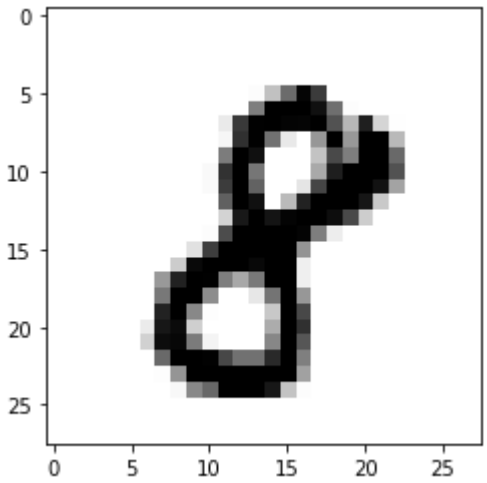
In [36]:
```
dataiter = iter(cargador_entrenamiento)
imagenes, etiquetas = dataiter.next()
print(type(imagenes))
print(imagenes.shape)
print(etiquetas.shape)
```

```
<class 'torch.Tensor'>
torch.Size([64, 1, 28, 28])
torch.Size([64])
```

La forma de las imágenes es, `torch.Size([64,1,28,28])` , lo que sugiere que hay 64 imágenes en cada lote y cada imagen tiene una dimensión de 28 x 28 píxeles. Del mismo modo, las etiquetas tienen forma de `torch.Size([64])` . Las 64 imágenes deben tener 64 etiquetas respectivamente.

Vamos a mostrar una imagen del conjunto de entrenamiento, por ejemplo, la primera.

In [37]:
```
plt.imshow(imagenes[0].numpy().squeeze(), cmap='gray_r');
```

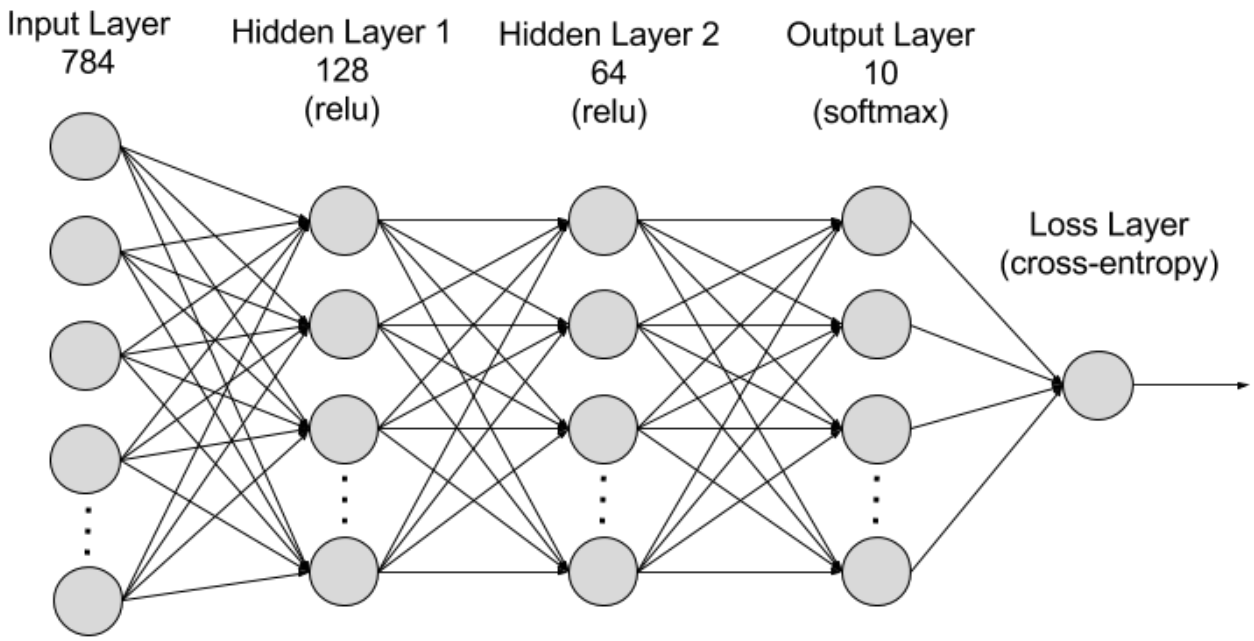


Esto generará una cuadrícula de imágenes en un orden aleatorio.

In [38]:
```
figura = plt.figure()
num_de_imagenes = 60
for index in range(1, num_de_imagenes + 1):
    plt.subplot(6, 10, index)
    plt.axis('off')
    plt.imshow(imagenes[index].numpy().squeeze(), cmap='gray_r')
```

Definir La Red Neuronal

Construiremos la siguiente red, ya que puede ver que contiene una capa de entrada (la primera capa), una capa de salida de diez neuronas (o unidades, los círculos) y dos capas ocultas en el medio.



El módulo `torch.nn` de PyTorch nos permite construir la red anterior de manera muy simple.

In [79]:
```
# Detalles de las capas de la red neuronal
tamaño_entrada = 784
tamaño_ocultas = [128, 64]
tamaño_salida = 10

# Construir la red neuronal
modelo_1 = nn.Sequential(nn.Linear(tamaño_entrada, tamaño_ocultas[0]),
                      nn.ReLU(),
                      nn.Linear(tamaño_ocultas[0], tamaño_ocultas[1]),
                      nn.ReLU(),
                      nn.Linear(tamaño_ocultas[1], tamaño_salida),
                      nn.LogSoftmax(dim=1))
print(modelo_1)
```

```
Sequential(
  (0): Linear(in_features=784, out_features=128, bias=True)
  (1): ReLU()
  (2): Linear(in_features=128, out_features=64, bias=True)
  (3): ReLU()
  (4): Linear(in_features=64, out_features=10, bias=True)
  (5): LogSoftmax(dim=1)
)
```

El `nn.Secuential` envuelve las capas en la red. Hay tres capas lineales con activación ReLU (una función simple que permite el paso de valores positivos, mientras que los valores negativos se modifican a cero). La capa de salida es una capa lineal con activación `LogSoftmax` porque este es un problema de clasificación.

Técnicamente, una función `LogSoftmax` es el logaritmo de una función `Softmax` como su nombre lo dice y se ve así, como se muestra a continuación.

$$LogSoftMax(x_i) = log\left(\frac{exp(x_i)}{\sum_j exp(x_i)}\right)$$

A continuación, definimos la pérdida de probabilidad logarítmica negativa. La cual es útil para entrenar un problema de clasificación con C clases. Juntas, `LogSoftmax()` y `NLLLoss()` actúan como la pérdida de entropía cruzada como se muestra en el diagrama de arquitectura de red anterior.

Además, debe preguntarse por qué tenemos 784 unidades en la primera capa. ¡Bueno! Es porque aplanamos cada imagen antes de enviarla dentro de la red neuronal. (28 x 28 = 784)

Entrenamiento de la Red Neuronal

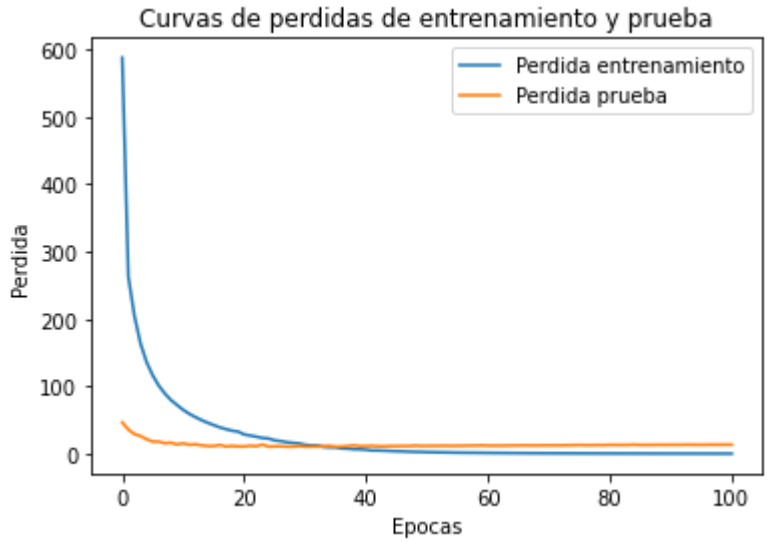
Aquí es donde sucede la magia real. Su red neuronal itera sobre el conjunto de entrenamiento y actualiza los pesos. Hacemos uso de `torch.optim` , que es un módulo proporcionado por PyTorch para optimizar el modelo, realizar el descenso de gradiente y actualizar los pesos mediante la propagación hacia atrás. Por lo tanto, en cada época (número de veces que iteramos sobre el conjunto de entrenamiento), veremos una disminución gradual en la pérdida de entrenamiento.

In [80]:
```
perdida_fn = nn.NLLLoss()
optimizador = optim.SGD(modelo_1.parameters(), lr=0.003, momentum=0.9)
tiempo_0 = time()
epocas = 101

# Crear lista vacias para guardar los valores de las perdidas
perdidas_entrenamiento = []
perdidas_prueba = []
conteo_epocas = []

for epoca in range(epocas):
    ### Entrenamiento

    # Poner el modelo en modo de entrenamiento (este es el estado predeterminado de un mode
    modelo_1.train()

    perdida_actual = 0
    for imagenes, etiquetas in cargador_entrenamiento:
        # Aplanar las imagenes MNIST a un vector de longitud 784
        imagenes = imagenes.view(imagenes.shape[0], -1)
        # 1. Propagar hacia adelante los datos de entrenamiento usando el método forward()
        salida = modelo_1(imagenes)

        # 2. Calcule la pérdida (qué tan diferentes son las predicciones de nuestros modelo
        perdida = perdida_fn(salida, etiquetas)

        # 3. Colocar a cero los gradientes del optimizador
        optimizador.zero_grad()

        # 4. Propagación hacia atrás
        perdida.backward()

        # 5. Realizar paso de optimización
        optimizador.step()

        perdida_actual += perdida

    ### Prueba

    # Colocar el modelo en modo evaluación
    modelo_1.eval()

    with torch.inference_mode():
        perdida_prueba_actual = 0
        for imagenes, etiquetas in cargador_validacion:
            # Aplanar las imagenes MNIST a un vector de longitud 784
            imagenes = imagenes.view(imagenes.shape[0], -1)
            # 1. Propagar hacia adelante los datos de entrenamiento usando el método forwar
            salida = modelo_1(imagenes)
             # 2. Calcule la pérdida (qué tan diferentes son las predicciones de nuestros m
            perdida = perdida_fn(salida, etiquetas)

            perdida_prueba_actual += perdida

        conteo_epocas.append(epoca)
        perdidas_entrenamiento.append(perdida_actual.detach().numpy())
        perdidas_prueba.append(perdida_prueba_actual.detach().numpy())

        # Imprimir las perdidas cada 10 epocas
        if epoca % 10 == 0:
            print(f"Epoca: {epoca} | Perdida Entrenamiento: {perdida_actual:.6f} | Perdida
```

```
print("\nTiempo Entrenamiento (en minutos) =",(time()-tiempo_0)/60)
```

```
Epoca: 0 | Perdida Entrenamiento: 587.560974 | Perdida Prueba: 46.810242
Epoca: 10 | Perdida Entrenamiento: 65.367882 | Perdida Prueba: 15.768169
Epoca: 20 | Perdida Entrenamiento: 28.928211 | Perdida Prueba: 11.162314
Epoca: 30 | Perdida Entrenamiento: 13.577231 | Perdida Prueba: 11.026869
Epoca: 40 | Perdida Entrenamiento: 6.312559 | Perdida Prueba: 11.668453
Epoca: 50 | Perdida Entrenamiento: 2.725103 | Perdida Prueba: 12.009416
Epoca: 60 | Perdida Entrenamiento: 1.608665 | Perdida Prueba: 12.460061
Epoca: 70 | Perdida Entrenamiento: 1.073707 | Perdida Prueba: 12.794599
Epoca: 80 | Perdida Entrenamiento: 0.814414 | Perdida Prueba: 13.552369
Epoca: 90 | Perdida Entrenamiento: 0.630292 | Perdida Prueba: 13.660367
Epoca: 100 | Perdida Entrenamiento: 0.523771 | Perdida Prueba: 13.843933

Tiempo Entrenamiento (en minutos) = 43.16070965528488
```

In [81]:
```
# Graficar las curvas de perdidas
plt.plot(conteo_epocas, perdidas_entrenamiento, label="Perdida entrenamiento")
plt.plot(conteo_epocas, perdidas_prueba, label="Perdida prueba")
plt.title("Curvas de perdidas de entrenamiento y prueba")
plt.ylabel("Perdida")
plt.xlabel("Epocas")
plt.legend();
```



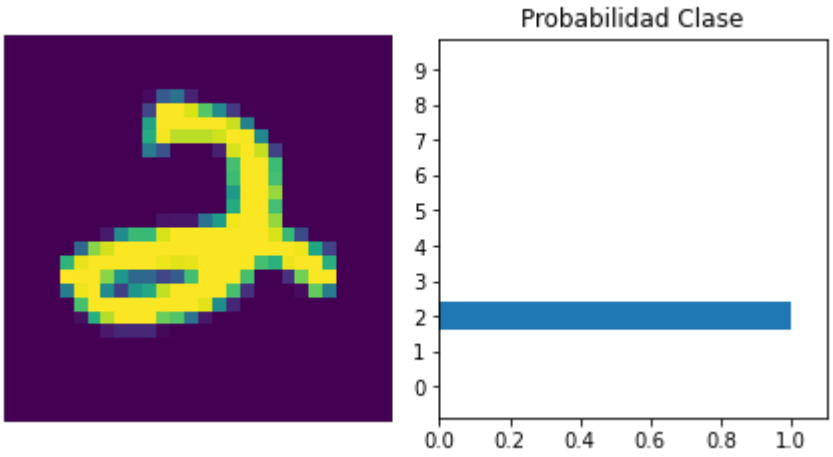
Evaluación del Modelo

Ya casi hemos terminado con nuestro trabajo. El modelo está listo, pero primero tenemos que evaluarlo. Crear una función de utilidad ver_clasificacion() para mostrar la imagen y las probabilidades de clase que se predijeron.

In [82]:
```
def ver_clasificacion(imagen, ps):
    ''' Función para ver una imagen y sus clases predichas.
    '''
    ps = ps.data.numpy().squeeze()

    fig, (ax1, ax2) = plt.subplots(figsize=(6,9), ncols=2)
    ax1.imshow(imagen.resize_(1, 28, 28).numpy().squeeze())
    ax1.axis('off')
    ax2.barh(np.arange(10), ps)
    ax2.set_aspect(0.1)
    ax2.set_yticks(np.arange(10))
    ax2.set_yticklabels(np.arange(10))
    ax2.set_title('Probabilidad Clase')
    ax2.set_xlim(0, 1.1)
    plt.tight_layout()
```

In [85]:
```
imagenes, etiquetas = next(iter(cargador_validación))

imagen = imagenes[0].view(1, 784)

modelo_1.eval()

with torch.inference_mode():
    logps = modelo_1(imagen)

# Salida de la red son probabilidades logarítmicas,
# se necesita tomar el exponente para obtener probabilidades
ps = torch.exp(logps)
probab = list(ps.numpy()[0])
print("Dígito Predicho =", probab.index(max(probab)))
ver_clasificacion(imagen.view(1, 28, 28), ps)
```

```
Dígito Predicho = 2
```



Probar el modelo con los datos de validación

In [86]:
```
conteo_correcto, conteo_total = 0, 0
for imagenes,etiquetas in cargador_validación:
    for i in range(len(etiquetas)):
        imagen = imagenes[i].view(1, 784)
        # Apagar gradientes para acelerar esta parte
        with torch.no_grad():
            logps = modelo_1(imagen)

        # Salida de la red son probabilidades logarítmicas,
        # se necesita tomar el exponente para obtener probabilidades
        ps = torch.exp(logps)
        probab = list(ps.numpy()[0])
        etiqueta_predicha = probab.index(max(probab))
        etiqueta_verdadera = etiquetas.numpy()[i]
        if(etiqueta_verdadera == etiqueta_predicha):
            conteo_correcto += 1
        conteo_total += 1

print("Número de Imágenes Probadas =", conteo_total)
print('Número de Imágenes Correctas =', conteo_correcto)
print("\nExactitud del Modelo =", (conteo_correcto/conteo_total))
```

```
Número de Imágenes Probadas = 10000
Número de Imágenes Correctas = 9806

Exactitud del Modelo = 0.9806
```

In [ ]: