Redes Neuronales para Lenguaje Natural

2024

Grupo de Procesamiento de Lenguaje Natural
Instituto de Computación

Redes Neuronales Completamente Conectadas

- Neurona artificial, descenso por gradiente
- Redes multicapa, perceptrón multicapa, backpropagation
- Sobreajuste, hiperparámetros, representación de texto
- **Implementación**

Introducción PyTorch

Librerías

- PyTorch
  - Primitivas para deep learning
  - Tensores, datasets y dataloaders
  - Grafo de computación (diferenciación automática)
  - Modelos, tipos de capas, entrenamiento, cargar/guardar
  - Funciones de activación, loss functions, optimizers
- PyTorch-NLP (torchnlp)
  - Datasets
  - Helpers, pretrained word embeddings
  - Funcionalidades para NLP (Ej. LockedDropout)

Tensores

Tensores

- Generalizan escalares, vectores y matrices
  - Escalares, tensores de rango (u orden) 0
  - Vectores (rango 1), matriz (rango 2)
  - Matriz multidimensional, rango superior a 2 (Ej. Kxnxm)
- El tamaño (shape o size) de un tensor son los grados de libertad en cada dimensión
  - Ej. tensor de rango 3 de tamaño 10x15x300
- PyTorch provee manejo de tensores (como numpy)

Crear tensores con PyTorch

```
z = torch.zeros(5, 3)
print(z)
print(z.dtype)
```

Out:

```
tensor([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]])
torch.float32
```

```
i = torch.ones((5, 3), dtype=torch.int16)
print(i)
```

Out:

```
tensor([[1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1],
        [1, 1, 1]], dtype=torch.int16)
```

Crear tensores con PyTorch (2)

A partir de datos:

```
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

A partir de un tensor de numpy:

```
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

A partir de otro tensor:

```
x_ones = torch.ones_like(x_data) # retains the properties of x_data
print(f"Ones Tensor: \n {x_ones} \n")

x_rand = torch.rand_like(x_data, dtype=torch.float) # overrides the datatype of
x_data
print(f"Random Tensor: \n {x_rand} \n")
```

Crear tensores con PyTorch (3)

```
torch.manual_seed(1729)
r1 = torch.rand(2, 2)
print('A random tensor:')
print(r1)

r2 = torch.rand(2, 2)
print('\nA different random tensor:')
print(r2) # new values

torch.manual_seed(1729)
r3 = torch.rand(2, 2)
print('\nShould match r1:')
print(r3) # repeats values of r1 because of re-seed
```

Out:

```
A random tensor:
tensor([[0.3126, 0.3791],
        [0.3087, 0.0736]])

A different random tensor:
tensor([[0.4216, 0.0691],
        [0.2332, 0.4047]])

Should match r1:
tensor([[0.3126, 0.3791],
        [0.3087, 0.0736]])
```

Atributos de un tensor

```
tensor = torch.rand(3,4)

print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
```

Out:

```
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
```

```
# We move our tensor to the GPU if available
if torch.cuda.is_available():
    tensor = tensor.to("cuda")
```

Operaciones con tensores

```
ones = torch.ones(2, 3)
print(ones)

twos = torch.ones(2, 3) * 2 # every element is multiplied by 2
print(twos)

threes = ones + twos       # addition allowed because shapes are similar
print(threes)              # tensors are added element-wise
print(threes.shape)        # this has the same dimensions as input tensors

r1 = torch.rand(2, 3)
r2 = torch.rand(3, 2)
# uncomment this line to get a runtime error
# r3 = r1 + r2
```

Out:

```
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[2., 2., 2.],
        [2., 2., 2.]])
tensor([[3., 3., 3.],
        [3., 3., 3.]])
torch.Size([2, 3])
```

Operaciones con tensores (2)

```
r = (torch.rand(2, 2) - 0.5) * 2 # values between -1 and 1
print('A random matrix, r:')
print(r)

# Common mathematical operations are supported:
print('\nAbsolute value of r:')
print(torch.abs(r))

# ...as are trigonometric functions:
print('\nInverse sine of r:')
print(torch.asin(r))

# ...and linear algebra operations like determinant and singular value
decomposition
print('\nDeterminant of r:')
print(torch.det(r))
print('\nSingular value decomposition of r:')
print(torch.svd(r))

# ...and statistical and aggregate operations:
print('\nAverage and standard deviation of r:')
print(torch.std_mean(r))
print('\nMaximum value of r:')
print(torch.max(r))
```

Operadores in-place

- Cambian el valor del tensor sin crear una copia (reducción del uso de memoria)
- Se denotan con el sufijo “_” en el nombre
  - Ej. add_(x) (in-place addition), t_() (in-place transpose)

```
print(f"{tensor} \n")
tensor.add_(5)
print(tensor)
```

Out:

```
tensor([[1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.],
        [1., 0., 1., 1.]])

tensor([[6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.],
        [6., 5., 6., 6.]])
```

Definir hardware de ejecución

- El hardware por defecto es CPU (set_default_device)
- o utilizar cuda (GPU) en caso de haber una disponible
  - o MPS (Metal Performance Shaders framework, MacOS)

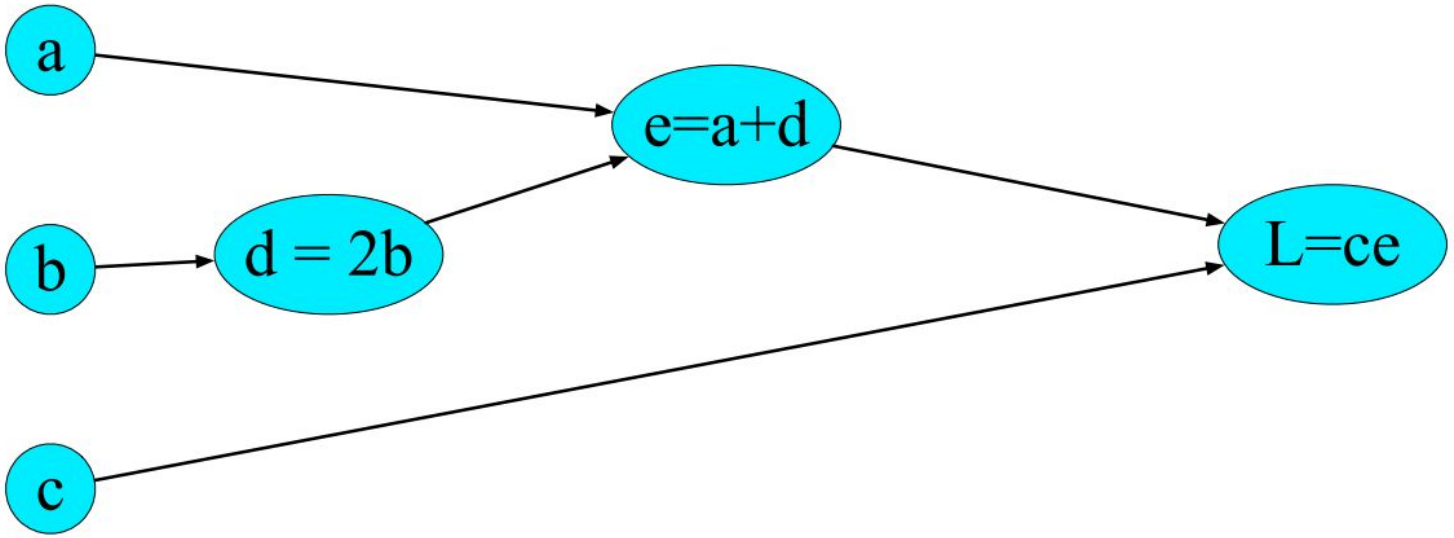
```
device = (
    "cuda"
    if torch.cuda.is_available()
    else "mps"
    if torch.backends.mps.is_available()
    else "cpu"
)
print(f"Using {device} device")
```

Autograd (diferenciación automática)

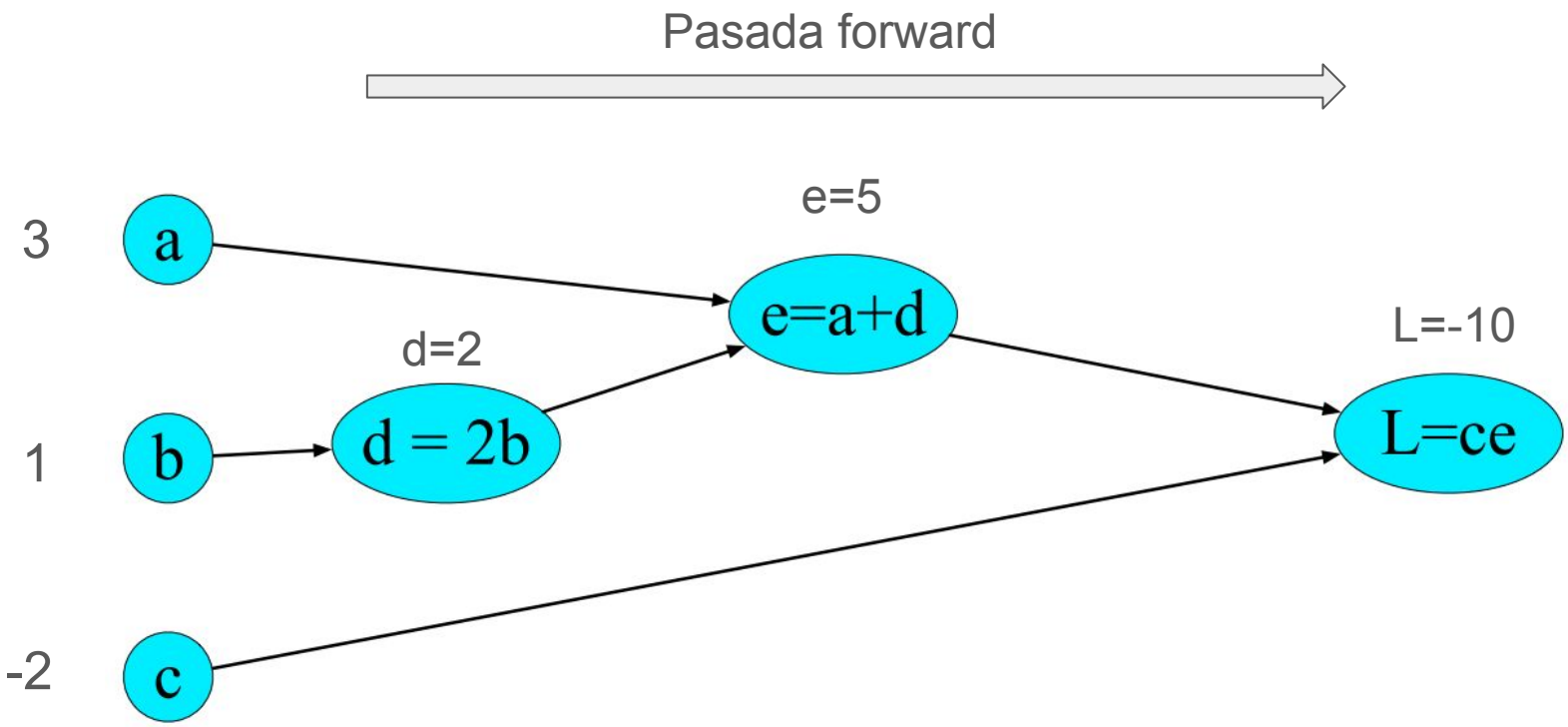
Grafo de computación

Ejemplo simple: $$L(a, b, c) = c(a + 2b)$$

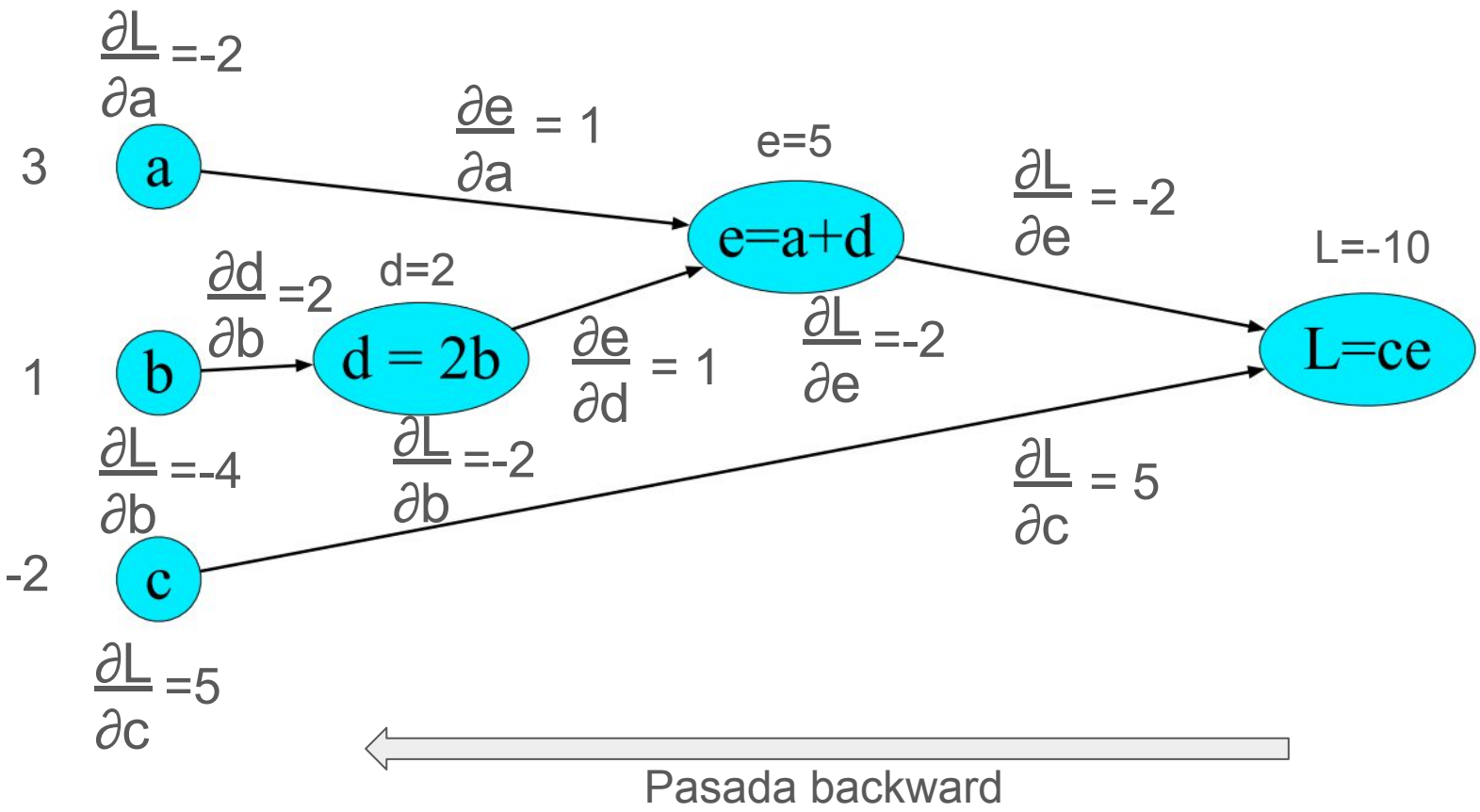
$$d = 2 * b$$
$$e = a + d$$
$$L = c * e$$



Grafo de computación



Grafo de computación



Diferenciación automática

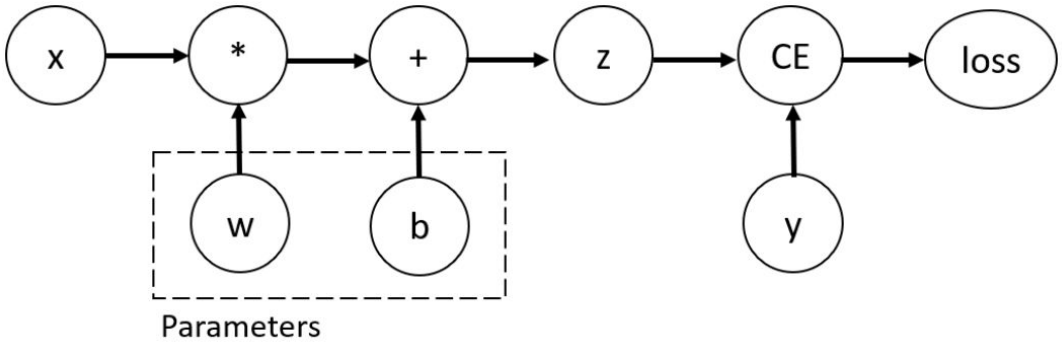
```
import torch

x = torch.ones(5)  # input tensor
y = torch.zeros(3)  # expected output
w = torch.randn(5, 3, requires_grad=True)
b = torch.randn(3, requires_grad=True)
z = torch.matmul(x, w)+b
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)
```

El parámetro requires_grad indica que podemos computar el gradiente de ese tensor. Puede cambiar su valor luego de la inicialización con:

```
x.requires_grad_(True)
```

El grafo de computación se crea dinámicamente



Computando el gradiente

```
loss.backward()
print(w.grad)
print(b.grad)
```

Out:

```
tensor([[0.3313, 0.0626, 0.2530],
        [0.3313, 0.0626, 0.2530],
        [0.3313, 0.0626, 0.2530],
        [0.3313, 0.0626, 0.2530],
        [0.3313, 0.0626, 0.2530]])
tensor([0.3313, 0.0626, 0.2530])
```

Deshabilitar la actualización del gradiente

```
z = torch.matmul(x, w)+b
print(z.requires_grad)

with torch.no_grad():
    z = torch.matmul(x, w)+b
print(z.requires_grad)
```

Para utilizar un modelo ya entrenado (performance)

Datasets y Dataloaders

Datasets y Dataloaders

- Dos clases para encapsular la lógica del manejo de datos
  - **Dataset**
    - Contiene los ejemplos (entradas y salidas)
  - **Dataloader**
    - Encapsula a un Dataset para accederlo

Custom Dataset

```
import os
import pandas as pd
from torchvision.io import read_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None,
target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label
```

Dataloader

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Definición de la Red

Definición del modelo

```
class NeuralNetwork(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits
```

Módulo nn de pytorch

```
from torch import nn
```

Entrada: Matriz de 28x28

Instanciar un modelo

```
model = NeuralNetwork().to(device)
print(model)
```

Out:

```
NeuralNetwork(
  (flatten): Flatten(start_dim=1, end_dim=-1)
  (linear_relu_stack): Sequential(
    (0): Linear(in_features=784, out_features=512, bias=True)
    (1): ReLU()
    (2): Linear(in_features=512, out_features=512, bias=True)
    (3): ReLU()
    (4): Linear(in_features=512, out_features=10, bias=True)
  )
)
```

Invocar un modelo instanciado

```
X = torch.rand(1, 28, 28, device=device)
logits = model(X)
pred_probab = nn.Softmax(dim=1)(logits)
y_pred = pred_probab.argmax(1)
print(f"Predicted class: {y_pred}")
```

Out:

```
Predicted class: tensor([7], device='cuda:0')
```

Entrenamiento

Entrenamiento de una red

- Definir e instanciar modelo
- Definir e instanciar función de pérdida y optimizer

```
model = NeuralNetwork().to(device)
```

```
loss_fn = nn.CrossEntropyLoss()
```

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

Entrenamiento de una red (train loop)

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    # Set the model to training mode - important for batch normalization and
dropout layers
    # Unnecessary in this situation but added for best practices
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Entrenamiento de una red (test loop)

```python
def test_loop(dataloader, model, loss_fn):
    # Set the model to evaluation mode - important for batch normalization and
dropout layers
    # Unnecessary in this situation but added for best practices
    model.eval()
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    # Evaluating the model with torch.no_grad() ensures that no gradients are
computed during test mode
    # also serves to reduce unnecessary gradient computations and memory usage for
tensors with requires_grad=True
    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss:
{test_loss:>8f} \n")
```

Aplicación

Caso de aplicación

- Vamos a construir un clasificador de sentimiento sobre reviews de películas
  - Binario: positivo o negativo
- Con una red neuronal completamente conectada
  - Usando Bag of Words de la entrada y reduciendo la dimensionalidad con SVD

Bag of Words y SVD (sklearn)

```
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.decomposition import TruncatedSVD

#...

vectorizer_bow = CountVectorizer(analyzer='word', ngram_range=(1, 2))
vectorizer_bow = vectorizer_bow.fit(x)

x_bow = vectorizer_bow.transform(x)



# SVD dimensionality reduction

vectorizer_svd = TruncatedSVD(n_components=500, n_iter=10, random_state=432)
vectorizer_svd = vectorizer_svd.fit(x_bow)

x_bow_svd = vectorizer_svd.transform(x_bow)
```

Definición del modelo

Usando el módulo nn :

```
class Net(nn.Module):

  def __init__(self):
    super(Net, self).__init__()
    self.fc1 = nn.Linear(500, 200)
    self.fc2 = nn.Linear(200, 1)

  def forward(self, x):
    x = self.fc1(x)
    x = F.relu(x)
    x = self.fc2(x)
    x = F.sigmoid(x)
    return x
```

Instanciar y ejecutar:

```
# instanciar y ejecutar
my_net = Net()
y = my_net(some_input)
```

Función de pérdida y optimizer

```
import torch.optim as optim

criterion = nn.BCELoss()
optimizer = optim.SGD(my_net.parameters(),
                      lr=.001, momentum=.9)
```

Entrenamiento

```
for epoch in range(20):

    running_loss = 0.0
    for i, data in enumerate(train_dataloader):

        # Obtener inputs y labels, data es una lista de [inputs, labels]
        inputs, labels = data

        # Resetear los gradientes de los parámetros
        optimizer.zero_grad()

        # forward + backward + optimizar
        outputs = my_net(inputs).squeeze(1)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # Imprimir estadísticas
        running_loss += loss.item()
        if i % 50 == 49:    # Cada 50 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 50:.3f}')
            running_loss = 0.0
```

Evaluación

```
import torcheval.metrics as tm

precision = tm.BinaryPrecision()
recall = tm.BinaryRecall()
f1 = tm.BinaryF1Score()
accuracy = tm.BinaryAccuracy()

with torch.no_grad():
    for data in test_dataloader:
        inputs, labels = data
        outputs = my_net(inputs).squeeze(1)
        labels = labels.type(torch.int32)

        precision.update(outputs, labels)
        recall.update(outputs, labels)
        f1.update(outputs, labels)
        accuracy.update(outputs, labels)

print(f"Precision: {precision.compute()}")
print(f"Recall: {recall.compute()}")
print(f"F1: {f1.compute()}")
print(f"Accuracy: {accuracy.compute()}")
```