In [462…
```
import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Dataset, ConcatDataset
import matplotlib.pyplot as plt
from tqdm import tqdm
import torchvision.transforms.functional as F
import numpy as np
```

Ataques Adversarios

> Autor: Jorge García González (Universidad de Málaga)

> Última Actualización: 10/12/2025

> Asignatura: Programación para la Inteligencia Artificial

¿Se puede engañar a una red neuronal? Ojo. La pregunta no es si una red puede equivocarse. Eso es obvio. La pregunta es si podemos engañarla. Si podemos forzar el error. Uno podría pensar que basta con usar como entrada un dato para el que la red no fue entrenada, ¿no? Si aplico a la imagen de un ciervo una una red entrenada para reconocer gatos, perros o conejos, no es sorprendente que falle. Sin embargo, eso no tiene mucho interés. Concretemos la pregunta pues: ¿Podemos hacer que la red falle para casos en los que se espera que acierte?

Llamamos Ataque Adversarial (*Adversarial Attack*) a alterar un ejemplo de entrada de una red neuronal de manera imperceptible para un humano, pero cambiando el resultado de la red. Típicamente estos ataques se enmarcan en problemas de clasificación.

Vamos a rescatar algunas funcionalidades de otros cuadernos y a entrenar un modelo para clasificar el MNIST.

In [463…
```
device = 'cuda'
device = torch.device("cuda:0" if (device == 'cuda') and torch.cuda.is_available() else "cpu")
print(device)

DRIVE=False
if DRIVE:
  from google.colab import drive
  drive.mount('/content/drive')
  workpath = '/content/drive/MyDrive/Work/Docencia UMA/2025-2026/Programacion para la IA/data'
else:
  workpath = '/workspace/data'
```

```
cuda:0
```

In [464…
```
train_dataset = datasets.MNIST(root=workpath, train=True, download=True, transform=transforms.ToTensor())
test_dataset = datasets.MNIST(root=workpath, train=False, download=True, transform=transforms.ToTensor())
```

In [465…
```
class CudaDataset(Dataset):
  def __init__(self, dataset, device, transform = None):
    self.dataset = dataset
    self.cuda_y = []
    self.cuda_x = []
    self.device = device
    self.transform = transform

    for x, y in tqdm(self.dataset, desc="Moving to GPU"):
      self.cuda_x.append(x.to(self.device))
      self.cuda_y.append(torch.tensor(y, device=self.device))

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

  def __getitem__(self, idx):
    if self.transform is None:
      x = self.cuda_x[idx]
    else:
      x = self.transform(self.cuda_x[idx])
    return x, self.cuda_y[idx]

def split_dataset(dataset, split_share=0.9):
  """
  Devuelve dos subconjuntos del dataset. split_share define cuántos ejemplos irán al
  primer subconjunto. El resto irán al segundo.
  """
  mask_indices_to_first_subset = torch.rand(len(dataset))<=split_share
  indices_first_subset = [i for i, (_, _) in enumerate(dataset) if mask_indices_to_first_subset[i]]
  indices_second_subset = [i for i, (_, _) in enumerate(dataset) if not mask_indices_to_first_subset[i]]
```

```
  first_subset = torch.utils.data.Subset(dataset, indices_first_subset)
  second_subset = torch.utils.data.Subset(dataset, indices_second_subset)

  return first_subset, second_subset

def filter_dataset(dataset, chosen_labels):
  """
  Devuelve un subconjunto del dataset que solo contiene las etiquetas especificadas.
  """
  indices = [i for i, (_, y) in enumerate(dataset) if y in chosen_labels]
  subset = torch.utils.data.Subset(dataset, indices)
  return subset
```

In [466…
```
def show(imgs):
    if not isinstance(imgs, list):
        imgs = [imgs]
    fix, axs = plt.subplots(ncols=len(imgs), squeeze=False, figsize=(15, 15))
    for i, img in enumerate(imgs):
        img = img.detach()
        img = F.to_pil_image(img)
        axs[0, i].imshow(np.asarray(img))
        axs[0, i].set(xticklabels=[], yticklabels=[], xticks=[], yticks=[])
```

In [467…
```
train_dataset, val_dataset = split_dataset(train_dataset)

if not str(device)=="cpu":
  train_dataset = CudaDataset(train_dataset, device)
  val_dataset = CudaDataset(val_dataset, device)
  test_dataset = CudaDataset(test_dataset, device)
```

```
Moving to GPU: 100%|██████████| 53998/53998 [00:06<00:00, 8249.94it/s]
Moving to GPU: 100%|██████████| 6002/6002 [00:00<00:00, 8549.43it/s]
Moving to GPU: 100%|██████████| 10000/10000 [00:01<00:00, 8737.58it/s]
```

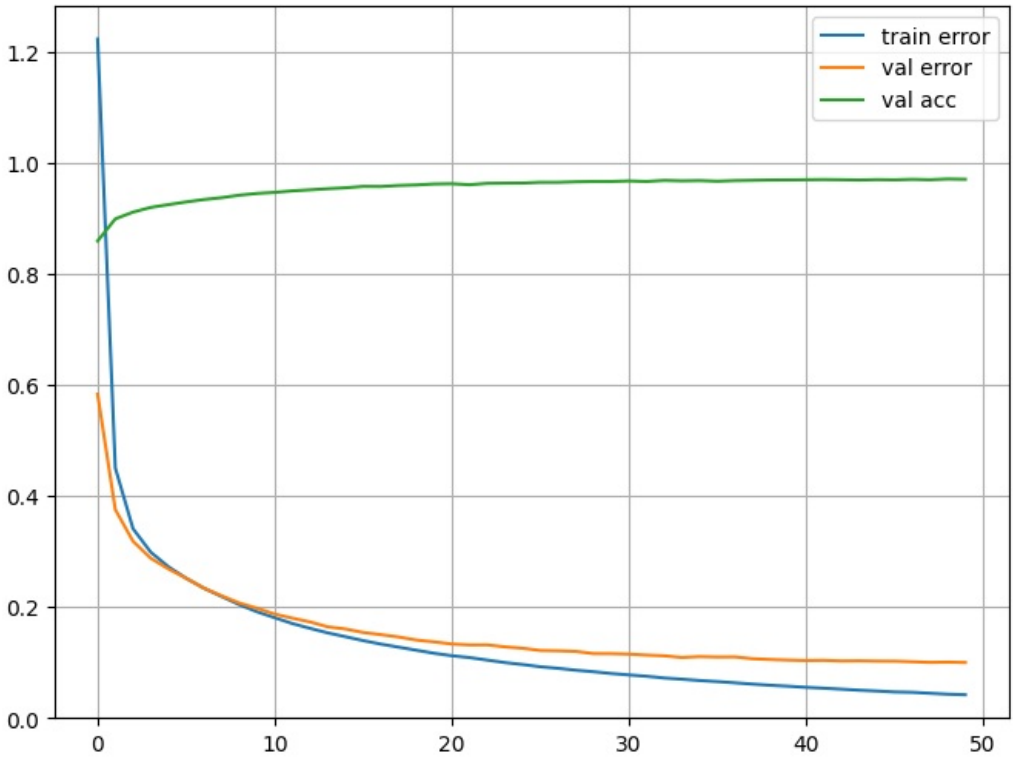
In [468…
```
def learning_loop_for_classification(train_dataloader, val_dataloader, model, epochs, loss_fn, optimizer, valida
  epoch_loss_list = []
  val_loss_list = []
  val_acc_list = []

  with tqdm(range(epochs), desc="epoch") as pbar:
    for epoch in pbar:
      steps_loss_list = []
      for x_true, y_true in train_dataloader:
        y_pred = model(x_true)                              # Forward
        optimizer.zero_grad()
        loss = loss_fn(y_pred, y_true)
        loss.backward()                                     # Backward
        steps_loss_list.append(loss.clone().detach())
        optimizer.step()
      train_loss = torch.tensor(steps_loss_list).mean()
      epoch_loss_list.append(train_loss)
      pbar.set_postfix(loss=f"{train_loss}")

      if epoch%validation_freq==0:
        val_step_loss_list = []
        val_correct_answers = 0
        val_set_size = 0
        with torch.no_grad():
          for x_val_true, y_val_true in val_dataloader:
            y_val_pred = model(x_val_true)

            loss_val = loss_fn(y_val_pred, y_val_true)
            val_step_loss_list.append(loss_val)

            correct_answers = torch.sum(y_val_true==torch.argmax(y_val_pred, dim=-1))
            val_correct_answers += correct_answers
            val_set_size += y_val_true.numel()

          val_loss = torch.tensor(val_step_loss_list).mean()
          val_loss_list.append(val_loss)
          val_acc_list.append(val_correct_answers/float(val_set_size))

  return model, epoch_loss_list, val_loss_list, val_acc_list
```

In [469…
```
learning_rate = 1e-3
epochs = 50
batch_size = 1024
loss_fn = torch.nn.CrossEntropyLoss()
validation_freq = 1

val_dataloader = DataLoader(val_dataset, batch_size=batch_size, shuffle=True)
train_dataloader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_dataset, batch_size=batch_size, shuffle=True)
```

In [470…
```
torch.manual_seed(0)

model = torch.nn.Sequential(
    torch.nn.Flatten(),
    torch.nn.Linear(28*28,64),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(64,10)
).to(device)

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


model, train_loss_list, val_loss_list, val_acc_list = learning_loop_for_classification(
    train_dataloader,
    val_dataloader,
    model,
    epochs,
    loss_fn,
    optimizer,
    validation_freq = validation_freq)
```

```
epoch: 100%|██████████| 50/50 [00:11<00:00,  4.32it/s, loss=0.04197491705417633]
```

In [471…
```
plt.figure(figsize=(8,6))
plt.plot(range(len(train_loss_list)), [l.cpu() for l in train_loss_list], label="train error")
plt.plot(range(0, validation_freq*len(val_loss_list), validation_freq), [l.cpu() for l in val_loss_list], label=
plt.plot(range(0, validation_freq*len(val_acc_list), validation_freq), [l.cpu() for l in val_acc_list], label="\

plt.legend()
plt.grid(True)
plt.ylim(bottom=0)
plt.show()
```



In [472…
```
test_correct_answers = 0
with torch.no_grad():
  for x_test_true, y_test_true in test_dataloader:

    y_test_pred = model(x_test_true)

    correct_answers = torch.sum(y_test_true==torch.argmax(y_test_pred, dim=-1))
    test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.9711999893188477
```

Un modelo sencillo, pero que consigue más de un 97% de acierto en el conjunto de test. Servirá para nuestros propósitos.

Vamos a sacar algunas imágenes del conjunto de test junto a sus etiquetas y a comprobar que efectivamente la red las clasifica bien.

In [473…
```
example_images, example_labels = test_dataset[10:25]

show(example_images)
print(example_labels)
```

```
[tensor(0, device='cuda:0'), tensor(6, device='cuda:0'), tensor(9, device='cuda:0'), tensor(0, device='cuda:0'),
tensor(1, device='cuda:0'), tensor(5, device='cuda:0'), tensor(9, device='cuda:0'), tensor(7, device='cuda:0'),
tensor(3, device='cuda:0'), tensor(4, device='cuda:0'), tensor(9, device='cuda:0'), tensor(6, device='cuda:0'),
tensor(6, device='cuda:0'), tensor(5, device='cuda:0'), tensor(4, device='cuda:0')]
```

In [474…
```
with torch.no_grad():
  pred_logits = model(torch.cat(example_images))
  pred_labels = torch.argmax(pred_logits, dim=-1)
print(pred_labels)
```

```
tensor([0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5, 4], device='cuda:0')
```

Ya tenemos un objetivo al que "atacar". Vamos a ver algunas opciones para plantear este ataque.

La idea esencial de los ataques adversarios es alterar los datos ligeramente. Añadir ruido. Bueno, vamos a probar entonces a añadir un poquito de ruido.

In [475…
```
epsilon = 0.05
perturbed_images = []
for img, label in zip(example_images, example_labels):
  perturbed_img = img + torch.randn_like(img)*epsilon
  perturbed_img = torch.clamp(perturbed_img, 0, 1)
  perturbed_images.append(perturbed_img)

show(perturbed_images)
```

¡Imperceptible!

Ahora vamos a ver qué dice nuestro modelo.

In [476…
```
with torch.no_grad():
  pred_logits = model(torch.cat(perturbed_images))
  pred_labels = torch.argmax(pred_logits, dim=-1)
print(pred_labels)
```

```
tensor([0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 9, 6, 6, 5, 4], device='cuda:0')
```

In [477…
```
test_correct_answers = 0
with torch.no_grad():
  for x_test_true, y_test_true in test_dataloader:

    perturbed_img = x_test_true + torch.randn_like(x_test_true)*epsilon
    perturbed_img = torch.clamp(perturbed_img, 0, 1)

    y_test_pred = model(perturbed_img)

    correct_answers = torch.sum(y_test_true==torch.argmax(y_test_pred, dim=-1))
    test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.9674999713897705
```

Vale... Claramente no funciona. Vamos a ser más agresivos:

In [478…
```
epsilon = 0.1
perturbed_images = []
for img, label in zip(example_images, example_labels):
  perturbed_img = img + torch.randn_like(img)*epsilon
  perturbed_img = torch.clamp(perturbed_img, 0, 1)
  perturbed_images.append(perturbed_img)

show(perturbed_images)
```

In [479]:
```
with torch.no_grad():
  pred_logits = model(torch.cat(perturbed_images))
  pred_labels = torch.argmax(pred_logits, dim=-1)
print(pred_labels)
```

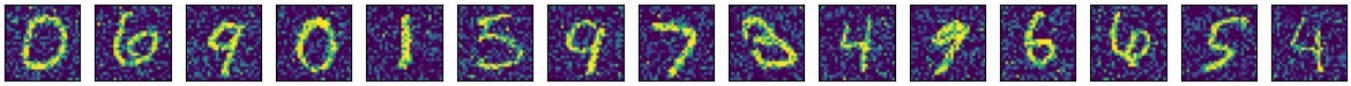
```
tensor([0, 6, 9, 0, 1, 5, 9, 7, 3, 4, 7, 6, 6, 5, 4], device='cuda:0')
```

In [480]:
```
test_correct_answers = 0
with torch.no_grad():
  for x_test_true, y_test_true in test_dataloader:

    perturbed_img = x_test_true + torch.randn_like(x_test_true)*epsilon
    perturbed_img = torch.clamp(perturbed_img, 0, 1)

    y_test_pred = model(perturbed_img)

    correct_answers = torch.sum(y_test_true==torch.argmax(y_test_pred, dim=-1))
    test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.9431999921798706
```

MUCHO más agresivos:

In [481]:
```
epsilon = 0.3
perturbed_images = []
for img, label in zip(example_images, example_labels):
  perturbed_img = img + torch.randn_like(img)*epsilon
  perturbed_img = torch.clamp(perturbed_img, 0, 1)
  perturbed_images.append(perturbed_img)

show(perturbed_images)
```



In [482]:
```
with torch.no_grad():
  pred_logits = model(torch.cat(perturbed_images))
  pred_labels = torch.argmax(pred_logits, dim=-1)
print(pred_labels)
```

```
tensor([5, 6, 5, 0, 1, 5, 9, 3, 3, 6, 8, 6, 6, 3, 5], device='cuda:0')
```

In [483]:
```
test_correct_answers = 0
with torch.no_grad():
  for x_test_true, y_test_true in test_dataloader:

    perturbed_img = x_test_true + torch.randn_like(x_test_true)*epsilon
    perturbed_img = torch.clamp(perturbed_img, 0, 1)

    y_test_pred = model(perturbed_img)

    correct_answers = torch.sum(y_test_true==torch.argmax(y_test_pred, dim=-1))
    test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.6017999649047852
```

Bien, fantástico. Hemos conseguido que un número más significativo de las clasificaciones fallen. El problema es que en las imágenes se ve claramente que han sido alteradas. Esto no cumple la condición de ser imperceptible para el ojo humano y encima no hemos conseguido que el modelo falle al predecir la mayoría.

Vamos a intentar añadir de ruido algo más inteligente.

FGSM

FGSM (*Fast Gradient Sign Method* o Método del Gradiente Rápido) es una técnica que se basa exactamente en la misma intuición en la que se basa el proceso de ajuste de las redes neuronales. El gradiente apunta en la dirección en la que una función crece. Esa idea aplicada a los pesos de la red permite que, en cada paso de un algoritmo de optimización, aumentemos o disminuyamos el valor de cada peso para

disminuir la función de pérdida.

El ataque FGSM usa esa dirección en la que la función de pérdida crece respecto a la entrada en lugar de respecto a los pesos. Dado un ejemplo (x, y) y un modelo ya entrenado m_{θ}:

$$x^{adv} = x + \epsilon s(\nabla_{x} L(m_{\theta}(x), y))$$

siendo s la función signo ($s(z)=1$ si $z\geq0$, $s(z)=-1$ si $z<0$).

Lo que implica que, para cada componente x_{i} de x:

$$x^{adv}_{i} = x + \epsilon s(\frac{\partial L(m_{\theta}(x), y)}{\partial x_{i}})$$

Dicho en lenguaje natural: FGSM suma o resta ϵ según qué opción aumenta el valor de la función de error L.

Vamos a implementarlo:

In [484…
```
def fgsm_attack(images, labels, epsilon, model, loss_fn):
  images = images.clone()
  images.requires_grad = True
  output = model(images)
  loss = loss_fn(output, labels)

  model.zero_grad()
  loss.backward()

  data_grad = images.grad.data

  adv_images = images + data_grad.sign()*epsilon

  return torch.clamp(adv_images, 0, 1)
```

Y ahora a aplicarlo:

In [485…
```
epsilon = 0.04
perturbed_images = []
for img, label in zip(example_images, example_labels):
  perturbed_img = fgsm_attack(img.unsqueeze(0), label.unsqueeze(0), epsilon, model, loss_fn)[0]
  perturbed_images.append(perturbed_img)

show(perturbed_images)
```

Con un nivel de ruido (ϵ) lo suficientemente bajo, los cambios en las imágenes son imperceptibles al ojo humano. Vamos a ver si también para el modelo que tenemos entrenado.

In [486…
```
with torch.no_grad():
  pred_logits = model(torch.cat(perturbed_images))
  pred_labels = torch.argmax(pred_logits, dim=-1)
print(pred_labels)
```

```
tensor([0, 8, 7, 0, 1, 3, 4, 3, 8, 9, 7, 5, 6, 5, 9], device='cuda:0')
```

In [487…
```
test_correct_answers = 0

for x_test_true, y_test_true in test_dataloader:

  perturbed_images = fgsm_attack(x_test_true, y_test_true, epsilon, model, loss_fn)

  pred_logits = model(perturbed_images)

  correct_answers = torch.sum(y_test_true==torch.argmax(pred_logits, dim=-1))
  test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.45969998836517334
```

Bueno, hemos conseguido que falle clasificando la mitad con un ruido imperceptible. Todavía podríamos aumentar un poco el margen de ruido sin que se apreciara y aumentar el número de fallos, pero en lugar de eso vamos a intentar depurar este proceso de añadir ruido de manera "maliciosa".

PGD

PGD (*Projected Gradient Descent* o Descenso del Gradiente Proyectado) es una versión del FGSM iterativa. En lugar de sumar o restar ϵ a cada componente de la entrada una sola vez, PGD busca un ajuste más fino dando varios pasos más pequeños con la restricción de que el ruido no supere un umbral α. Dado un ejemplo (x, y) y un modelo ya entrenado m_{θ}, el paso de iteración $t+1$ para cada componente $x_{t,i}$:

$$\Delta_{t+1,i} = x_{t,i} + \epsilon s(\frac{\partial L(m_{\theta}(x_{t}), y)}{\partial x_{t,i}})$$

$$x_{t+1,i} = min(max(\Delta_{t+1,i}, x_{0,i} - \alpha), x_{0,i} + \alpha)$$

Esencialmente PGD busca dentro del espacio de posibilidades que se le permite una versión de la imagen original que maximice el incremento del error.

Vamos a implementarlo:

In [488…
```
def pgd_attack(images, labels, alpha, epsilon, model, loss_fn, iter):
  adv_images = images.detach().clone()

  for _ in range(iter):
    adv_images.requires_grad = True
    output = model(adv_images)
    loss = loss_fn(output, labels)

    model.zero_grad()
    loss.backward()

    data_grad = adv_images.grad.data

    adv_images = adv_images + data_grad.sign()*epsilon

    eta = torch.clamp(adv_images - images, min=-alpha, max=alpha)
    adv_images = torch.clamp(images + eta, 0, 1).detach()

  return torch.clamp(adv_images, 0, 1)
```

Y vamos a ejecutarlo dándole el mismo margen de error que usaba FGSM.

In [489…
```
epsilon = 0.001
alpha = 0.04
iter = 200
perturbed_images = []
for img, label in zip(example_images, example_labels):
  perturbed_img = pgd_attack(img.unsqueeze(0), label.unsqueeze(0), alpha, epsilon, model, loss_fn, iter)[0]
  perturbed_images.append(perturbed_img)

show(perturbed_images)
```

De nuevo los cambios en las imágenes son imperceptibles para el ojo humano. Vamos a ver si para nuestro modelo hay diferencia:

In [490…
```
with torch.no_grad():
  pred_logits = model(torch.cat(perturbed_images))
  pred_labels = torch.argmax(pred_logits, dim=-1)
print(pred_labels)
```

```
tensor([0, 8, 7, 7, 3, 3, 4, 3, 8, 9, 7, 5, 6, 5, 9], device='cuda:0')
```

In [491…
```
test_correct_answers = 0

for x_test_true, y_test_true in test_dataloader:

  perturbed_images = pgd_attack(x_test_true, y_test_true, alpha, epsilon, model, loss_fn, iter)

  pred_logits = model(perturbed_images)

  correct_answers = torch.sum(y_test_true==torch.argmax(pred_logits, dim=-1))
  test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.39659997820854187
```

¡Hemos conseguido aún más fallos!

Efectivamente estas técnicas requieren de tener el modelo entrenado. Aunque no lo parezca, esto es relativamente frecuente. Hay

muchos modelos pre-entrenados que en multitud de aplicaciones se usan tal cual.

¿Se puede realizar un ataque sin conocer el modelo? Es sensiblemente más dificil, pero hay investigación al respecto. Algunas técnicas deducen las perturbaciones del conjunto de datos. Otras técncias buscan deducir las fronteras de decisión a base de prueba y error.

¿Qué podemos hacer para que nuestro modelo sea más robusto? Usar estas técnicas como regularizadores :)

In [492…
```
def learning_loop_for_classification_adversarial(train_dataloader, val_dataloader, model, epochs, loss_fn, adver
  epoch_loss_list = []
  val_loss_list = []
  val_acc_list = []
  adv_loss_list = []

  with tqdm(range(epochs), desc="epoch") as pbar:
    for epoch in pbar:
      steps_loss_list = []
      steps_adv_loss = []
      for x_true, y_true in train_dataloader:
        y_pred = model(x_true)
        optimizer.zero_grad()

        loss = loss_fn(y_pred, y_true)
        adv_x = adversarial_attack(x_true, y_true)
        adv_y_pred = model(adv_x)
        adv_loss = loss_fn(adv_y_pred, y_true)
        total_loss = loss + lambda_adversarial*adv_loss

        total_loss.backward()
        steps_loss_list.append(total_loss.clone().detach())
        steps_adv_loss.append(adv_loss.clone().detach())
        optimizer.step()
      train_loss = torch.tensor(steps_loss_list).mean()
      adv_loss = torch.tensor(steps_adv_loss).mean()
      epoch_loss_list.append(train_loss)
      adv_loss_list.append(adv_loss)
      pbar.set_postfix(loss=f"{train_loss}")

      if epoch%validation_freq==0:
        val_step_loss_list = []
        val_correct_answers = 0
        val_set_size = 0
        with torch.no_grad():
          for x_val_true, y_val_true in val_dataloader:
            y_val_pred = model(x_val_true)

            loss_val = loss_fn(y_val_pred, y_val_true)
            val_step_loss_list.append(loss_val)

            correct_answers = torch.sum(y_val_true==torch.argmax(y_val_pred, dim=-1))
            val_correct_answers += correct_answers
            val_set_size += y_val_true.numel()

          val_loss = torch.tensor(val_step_loss_list).mean()
          val_loss_list.append(val_loss)
          val_acc_list.append(val_correct_answers/float(val_set_size))

  return model, epoch_loss_list, val_loss_list, val_acc_list, adv_loss_list
```

In [493…
```
def generate_fgsm_attack(epsilon, model, loss_fn):

    def apply_fgsm_attack(images, labels):
      return fgsm_attack(images, labels, epsilon, model, loss_fn)

    return apply_fgsm_attack

def generate_pgd_attack(epsilon, model, loss_fn, alpha, iter):

    def apply_pgd_attack(images, labels):
      return pgd_attack(images, labels, alpha, epsilon, model, loss_fn, iter)

    return apply_pgd_attack
```

Vamos a probara regularizar usando el ataque FGSM durante el entrenamiento.

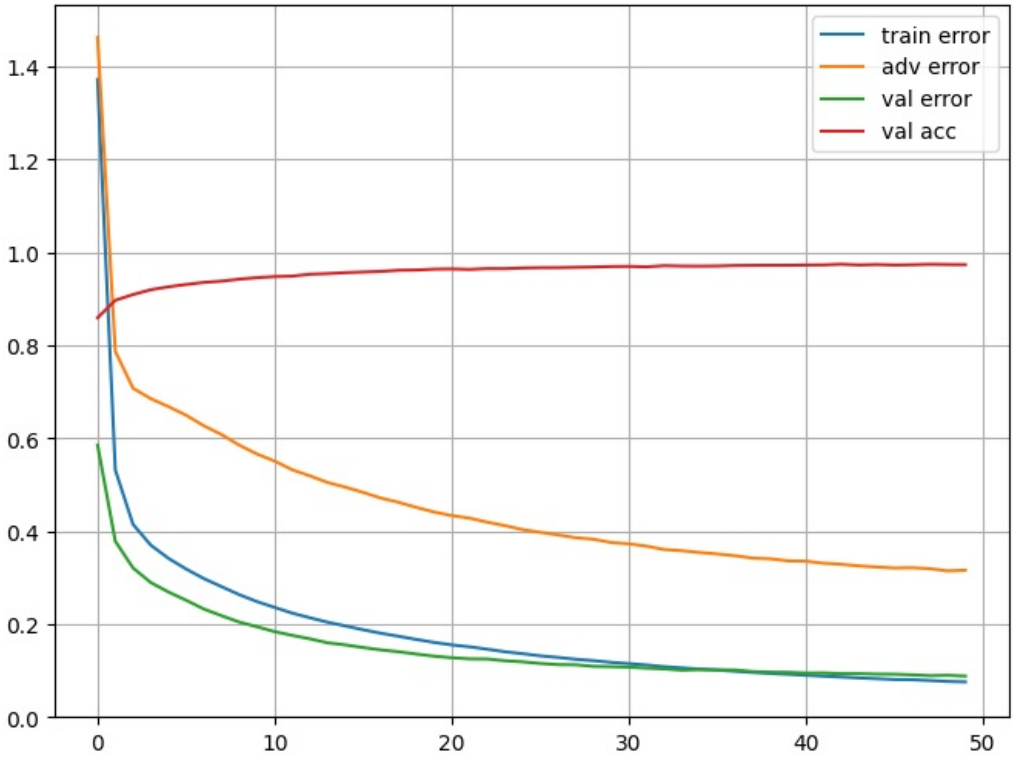
In [494…
```
torch.manual_seed(0)

model = torch.nn.Sequential(
    torch.nn.Flatten(),
    torch.nn.Linear(28*28,64),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(64,10)
```

```
).to(device)

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

epsilon = 0.04
adversarial_attack = generate_fgsm_attack(epsilon, model, loss_fn)
lambda_adversarial = 0.1

model, train_loss_list, val_loss_list, val_acc_list, adv_loss_list = learning_loop_for_classification_adversaria
    train_dataloader,
    val_dataloader,
    model,
    epochs,
    loss_fn,
    adversarial_attack,
    lambda_adversarial,
    optimizer,
    validation_freq = validation_freq)
```

```
epoch:   0%|          | 0/50 [00:00<?, ?it/s]
epoch: 100%|██████████| 50/50 [00:14<00:00,  3.35it/s, loss=0.07571763545274734]
```

In [495…

```
plt.figure(figsize=(8,6))
plt.plot(range(len(train_loss_list)), [l.cpu() for l in train_loss_list], label="train error")
plt.plot(range(len(adv_loss_list)), [l.cpu() for l in adv_loss_list], label="adv error")
plt.plot(range(0, validation_freq*len(val_loss_list), validation_freq), [l.cpu() for l in val_loss_list], label=
plt.plot(range(0, validation_freq*len(val_acc_list), validation_freq), [l.cpu() for l in val_acc_list], label="v

plt.legend()
plt.grid(True)
plt.ylim(bottom=0)
plt.show()
```



Veamos qué tal funciona para el conjunto de test tal cual:

In [496…

```
test_correct_answers = 0
with torch.no_grad():
  for x_test_true, y_test_true in test_dataloader:

    y_test_pred = model(x_test_true)

    correct_answers = torch.sum(y_test_true==torch.argmax(y_test_pred, dim=-1))
    test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.9745000004768372
```

Y ahora veamos qué tal funciona para el conjunto de test atacado con FGSM y PGD.

In [497…
```
test_correct_answers = 0

for x_test_true, y_test_true in test_dataloader:

  perturbed_images = fgsm_attack(x_test_true, y_test_true, epsilon, model, loss_fn)

  pred_logits = model(perturbed_images)

  correct_answers = torch.sum(y_test_true==torch.argmax(pred_logits, dim=-1))
  test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.878600001335144
```

In [498…
```
test_correct_answers = 0

for x_test_true, y_test_true in test_dataloader:

  perturbed_images = pgd_attack(x_test_true, y_test_true, alpha, epsilon, model, loss_fn, iter)

  pred_logits = model(perturbed_images)

  correct_answers = torch.sum(y_test_true==torch.argmax(pred_logits, dim=-1))
  test_correct_answers += correct_answers

acc = test_correct_answers/len(test_dataset)

print(f"Test acc:{acc}")
```

```
Test acc:0.8729999661445618
```

Como podemos ver, usando FGSM como regularizador no solo conseguimos robustez a los ataques de FGSM, pero también de PGD. Al regularizar usando un ataque adversario estamos haciendo algo parecido al aumentado de datos, pero usando como datos aumentados perturbaciones problemáticas para la red.

Conclusiones

Los ataques adversarios consisten en realizarle perturbaciones a una entrada para provocar el mal funcionamiento de la red para dicha entrada. Si se tiene acceso al modelo, se pueden realizar aprovechando la información del gradiente para identificar combinaciones de ruido especialmente perjudiciales. Estos ataques se pueden utilizar comor egularizadores para hacer un modelo más robusto ante ataques.