Investigación Operativa Práctica 1

Ying Ying Zhu

Parte 1

Simplex

Formato Entrada

```
examen_parcial_1_2023_2024 = """
argmax z = 0x1 + 1x2
x1 + 6x2 <= 14
0.4x1 + 0.3x2 >= 3
1x1 + 0x2  >= 1
0x1 + 1x2 >= 1
"""

examen_parcial_2_2022_2023 = """
argmax z = x1 + 2x2
-1x1+ x2 <= -1
x1 + x2 <= 2
"""
```

```python
#Codigo para sacar la matriz
for i, line in enumerate(lines):
    match = re.findall(r'x(\d+)', line)

    if line.startswith("argmax") or line.startswith("argmin"):

        line.replace("argmax", "")
        if line.startswith("argmin"):
            change_z = True
        line.replace("argmin", "")

        line = line.replace("z =", "").strip()

        parsed_line = [1]  # valor de la z


        for j in range(1, num_variables + 1):
            match = re.search(rf'([+-]?\d*\.?\d*)x{j}', line)
            coeff = float(match.group(1)) if match and match.group(1) else 1
            parsed_line.append(-coeff)

        # terminamos el reglon de las z
        parsed_line.extend([0] * (len(lines) - 1))
        parsed_line.append(0)

    else:
        # Parse restricciones: "4x1 + 2x2 <= 8" -> [0, 4, 2, 1, 0, 8]
        if "<=" in line:
            line, rhs = line.split("<=")
            rhs = float(rhs.strip())
        elif ">=" in line:
            line, rhs = line.split(">=")
            rhs = -float(rhs.strip())
            if 0 not in lineas_a_cambiar_signo:
                lineas_a_cambiar_signo.append(0)
            lineas_a_cambiar_signo.append(i)

        parsed_line = [0]  # empienzan en
```

Formato Clase

```
self.si = len(self.matriz[0]) - 2 - self.ni
self.diccionario = {f'x{i}': deepcopy(self.matriz[1:, i]) for i in range(1, self.ni + 1)}
self.diccionario.update({f's{i}': deepcopy(self.matriz[1:, self.ni + i]) for i in range(1, self.si + 1)
self.diccionario.update({'reglon_de_las_z': deepcopy(self.matriz[0][1:-1])})
self.diccionario.update({'terminos_independientes': deepcopy(self.matriz[1:, -1])})
self.diccionario.update({'restricciones': deepcopy(self.matriz[1:, 1:-1])})

self.reglon_de_las_z = deepcopy(self.matriz[0][1:-1])
self.terminos_independientes = deepcopy(self.matriz[1:, -1])
self.restricciones = deepcopy(self.matriz[1:, 1:-1])

self.base = {f's{i+1}': deepcopy(self.diccionario['s' + str(i+1)]) for i in range(self.si)}
```

No es lo más eficiente… Pero es claro

Empecemos…

```python
def resolucion(self, imprimir = False):
    # comprobar si 0 esta en la solucion, es decir si cumple las restriccione

    if np.all(self.terminos_independientes >= 0):
        return self.simplex_primal(imprimir)
    else:
        self.simplex_dual(imprimir)
        return self.simplex_primal(imprimir)
```

Simplex Dual

```python
while np.any(np.array(self.terminos_independientes) <= -1e-5) and vueltas > 0:
    vueltas -= 1
    if np.all(np.array(self.terminos_independientes) >= -1e-5) or np.all(np.isclose(self.reglon_de_las_z[:self.ni], 0, atol=1e-6)):
        break

    print('------------------------------------')
    self.iteraciones += 1
    print('Iteracion:', self.iteraciones)

    # Mascara para que los terminos independientes sean negativos
    negative_mask = self.terminos_independientes < -1e-5
    negative_values = self.terminos_independientes[negative_mask]

    if negative_values.size > 0:
        columna_salida = np.argmin(negative_values)
        columna_salida = np.where(negative_mask)[0][columna_salida]


    non_zero_mask = (np.abs(self.reglon_de_las_z) > tolerance) & (np.abs(self.restricciones[columna_salida, :]) > tolerance)

    # nos aseguramos que el reglon de laz z no valga 0 y que la restriccion no sea 0
    division_result = np.where(non_zero_mask,
                               self.reglon_de_las_z / self.restricciones[columna_salida, :],
                               np.inf)
```

Simplex Dual

```python
columna_entrada = np.argmin(abs(division_result))

if columna_entrada < self.ni:
    variable_entrada = 'x' + str(columna_entrada + 1)
else:
    variable_entrada = 's' + str(columna_entrada - self.ni + 1)

variable_salida = list(self.base.keys())[columna_salida]

if  imprimir:
    print('Variable de salida:', variable_salida)
    print('Variable de entrada:', variable_entrada)

# sacamos las columnas de la base, los nombres estan en self.base
B = self.base
B[variable_entrada] = self.diccionario[variable_entrada]
del B[variable_salida]

# Mantenemos el orden de las variables, primero las x1, x2, x3, ... y luego las s1, s2, s3, ...
B = {k: B[k] for k in sorted(B, key=self.orden_variables)}

self.base = B
```

Simplex Dual

```python
inversa_B = np.linalg.inv(np.array(list(B.values())).T)
# coeficientes reglon de las z para variables básicas
indices_base = [self.orden_variables(k) - 1 for k in B.keys()]
CB = np.array(-self.diccionario['reglon_de_las_z'][indices_base])

# Realizamos la multiplicacion

self.restricciones = np.dot(inversa_B,  self.diccionario['restricciones'])
self.terminos_independientes = np.dot(inversa_B, self.diccionario['terminos_independientes'])

self.matriz[1:, 1:-1] =deepcopy(self.restricciones)
self.reglon_de_las_z = np.dot(CB,  self.matriz[1:, 1:-1]) + self.diccionario['reglon_de_las_z']
self.matriz[0][1:-1] = deepcopy(self.reglon_de_las_z)
self.matriz[1:, -1] = deepcopy(self.terminos_independientes)

# ver en que base estamos y sacar los valores de la columna de terminos independientes

for i, k in enumerate(self.base.keys()):
    self.base[k] = deepcopy(self.diccionario[k])

self.matriz[0][-1] = np.dot(CB, self.terminos_independientes)
```

Simplex Primal

Casi lo mismo…

```
columna_entrada = np.argmin(self.reglon_de_las_z)
if columna_entrada < self.ni:
    variable_entrada = 'x' + str(columna_entrada + 1)
else:
    variable_entrada = 's' + str(columna_entrada - self.ni + 1)

# mascara para que las restricciones sean mayores a 0
positive_mask = np.abs(self.restricciones[:, columna_entrada]) > 1e-5
division_result = np.where(
    positive_mask,
    np.abs(self.terminos_independientes / self.restricciones[:, columna_entrada]),
    np.inf
)

columna_salida = np.argmin(division_result)

variable_salida = list(self.base.keys())[columna_salida]


if imprimir:
    print('Variable de entrada:', variable_entrada)
    print('Variable de salida:', variable_salida)
```

Ejemplos Simplex

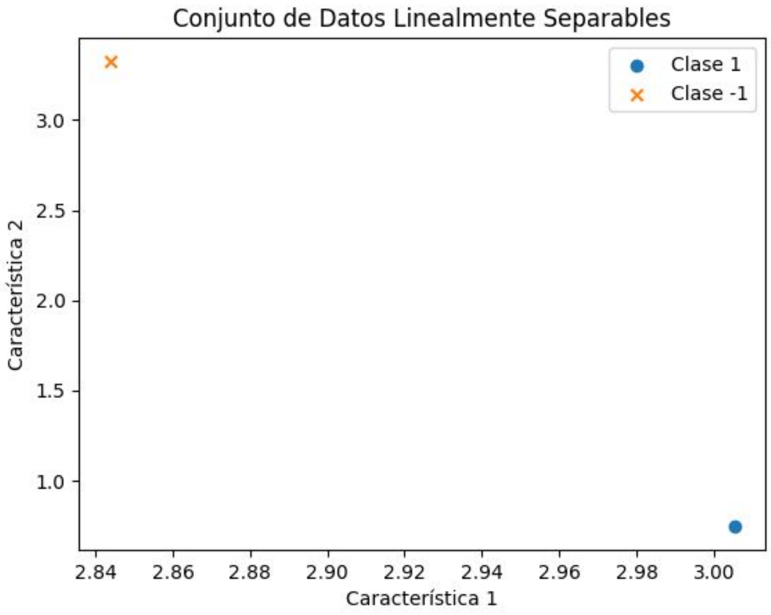
```
examen_final_2023_2024 = """
argmax z = 3x1 + 2x2
3x1 + x2 <= 10
x1 + 4x2 <= 12
"""
Simplex(examen_final_2023_2024).resolucion(True)
```

```
✓ 0.0s
-----------------------------------
Iteracion: 0
MATRIZ
[[ 1. -3. -2.  0.  0.  0.]
 [ 0.  3.  1.  1.  0. 10.]
 [ 0.  1.  4.  0.  1. 12.]]
```

```
-----------------------------------
SIMPLEX PRIMAL
-----------------------------------
Iteracion: 1
Variable de entrada: x1
Variable de salida: s1
Base: {'x1': array([3., 1.]), 's2': array([0., 1.])}
matriz:
[[ 1.     0.    -1.     1.     0.    10.   ]
 [ 0.     1.     0.333  0.333  0.     3.333]
 [ 0.     0.     3.667 -0.333  1.     8.667]]
-----------------------------------
Iteracion: 2
Variable de entrada: x2
Variable de salida: s2
Base: {'x1': array([3., 1.]), 'x2': array([1., 4.])}
matriz:
[[ 1.     0.     0.     0.909  0.273 12.364]
 [ 0.     1.    -0.     0.364 -0.091  2.545]
 [ 0.     0.     1.    -0.091  0.273  2.364]]
-----------------------------------

(np.float64(12.363636363636363),
 dict_keys(['x1', 'x2']),
 array([2.545, 2.364]))
```

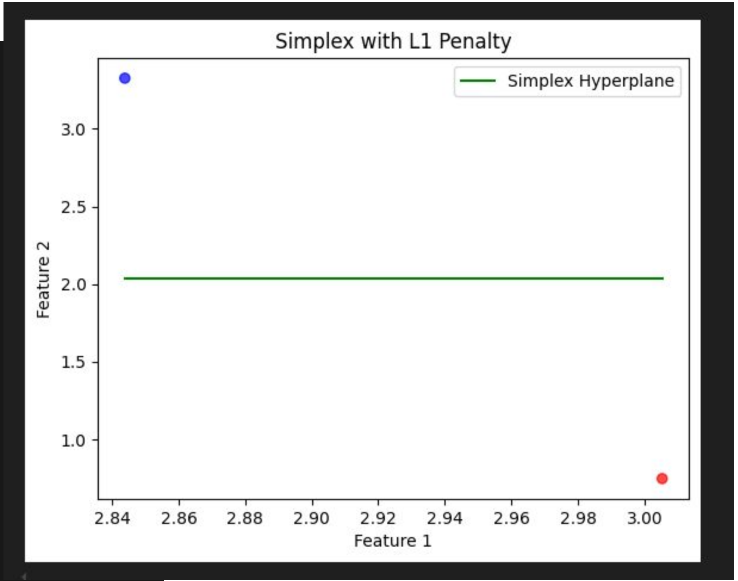
SVM

```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_blobs

# Generar un conjunto de datos linealmente separable
X, y = make_blobs(n_samples= 2, centers=2, n_features=2, random_state=0)

# Convertir las etiquetas a {-1, 1} para L1-SVM
y = 2 * y - 1

# Visualizar los datos
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], label='Clase 1', marker='o')
plt.scatter(X[y == -1][:, 0], X[y == -1][:, 1], label='Clase -1', marker='x')
plt.title('Conjunto de Datos Linealmente Separables')
plt.xlabel('Característica 1')
plt.ylabel('Característica 2')
plt.legend()
plt.show()
```

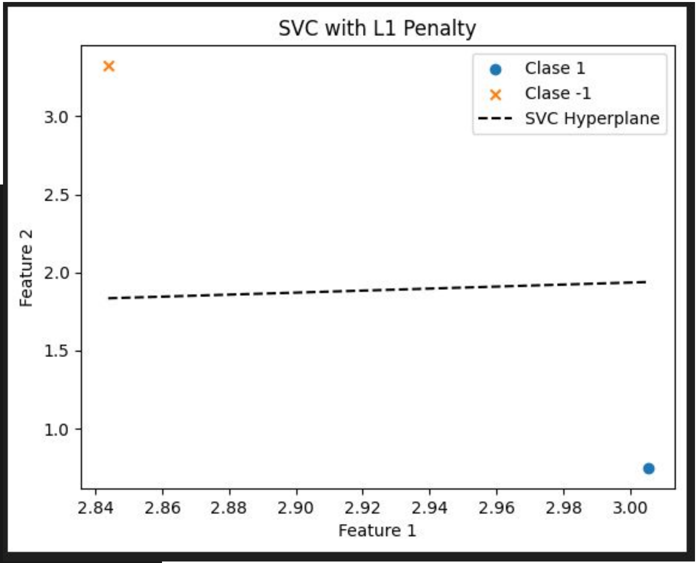


SVM SIMPLEX

```python
tabla = np.stack([X[:, 0], X[:, 1], y], axis=1)

# Definir las variables y restricciones para Simplex
restricciones = "argmin z = 1x1 + 1x2 + 0x3 \n"

for i, fila in enumerate(tabla):
    z1, z2, label = fila
    restricciones += f"{label * z1}x1 + {label * z2}x2 + {label}x3 >= {1}\n"

maximo, variables, valores = Simplex(restricciones).resolucion(True)
dicc = dict(zip(variables, valores))
dicc = {k: dicc.get(k, 0) for k in ['x1', 'x2', 'x3']}

# Calcular el hiperplano de Simplex
w1, w2, b = dicc['x1'], dicc['x2'], dicc['x3']

# Crear el gráfico
if w2 == 0:
    # Calcular x para el hiperplano de decisión (hiperplano)
    x_hyperplane = -b / w1
    plt.axvline(x=x_hyperplane, color='green', linestyle='--', label='Decision Boundary')

else:
    xx = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
    yy = (-w1 / w2) * xx - b / w2

    plt.plot(xx, yy, label='Simplex Hyperplane', color='green')
```

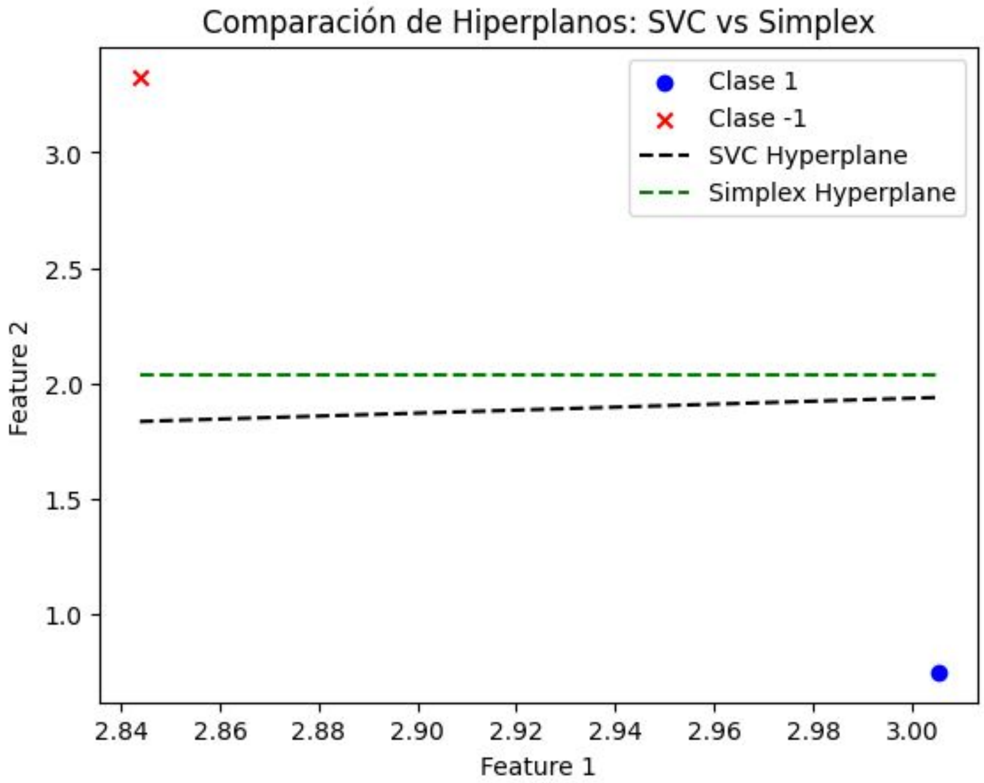


SVM SKLEARN



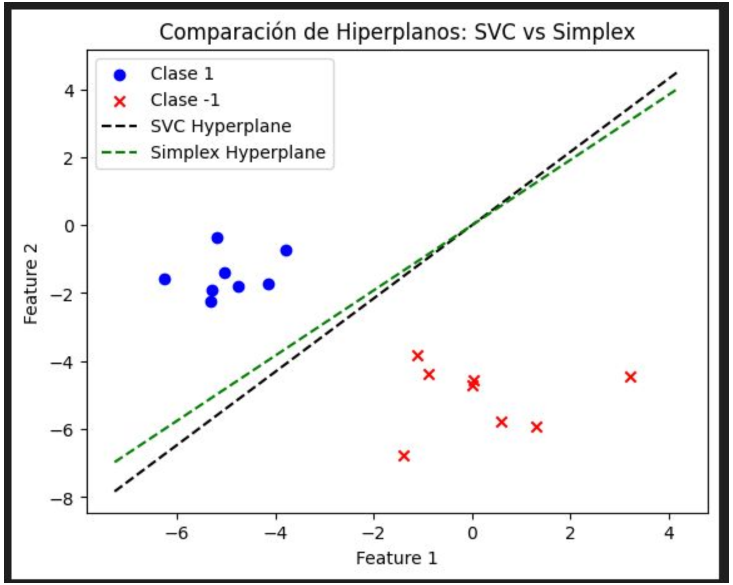

```python
from sklearn.svm import LinearSVC as SVC

# Crear un clasificador SVM con regularización L1
clf = SVC(penalty='l1')
clf.fit(X, y)

# Obtener los coeficientes y el término independiente
w = clf.coef_[0]  # Coefficients
b = clf.intercept_[0]  # Intercept

# Crear el gráfico
plt.scatter(X[y == 1][:, 0], X[y == 1][:, 1], label='Clase 1', marker='o')
plt.scatter(X[y == -1][:, 0], X[y == -1][:, 1], label='Clase -1', marker='x')

print('Coeficientes SVC', clf.coef_, 'Intercept', clf.intercept_)

if w[1] == 0:
    x_hyperplane = -b / w[0]
    plt.axvline(x=x_hyperplane, color='black', linestyle='--', label='Decision Boundary')
else:
    xx = np.linspace(X[:, 0].min(), X[:, 0].max(), 100)
    yy = (-w[0] / w[1]) * xx - b / w[1]
    plt.plot(xx, yy, 'k--', label='SVC Hyperplane')
```

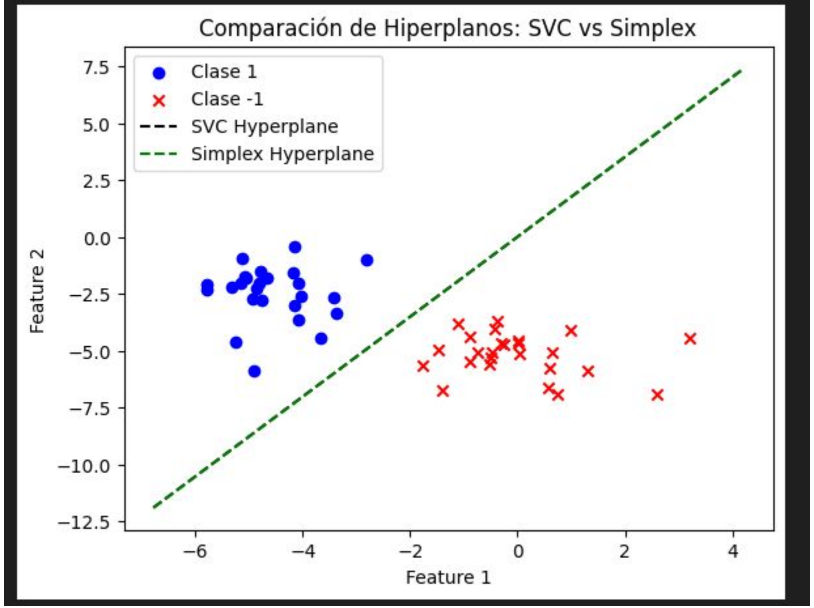
SVM Comparativa



SVM Comparativa Valor Absoluto

```
restricciones = "argmin z = 1x1 + -1x2 + 1x3 + -1x4 + 0x5\n"

for i, fila in enumerate(tabla):
    z1, z2, label = fila
    restricciones += f"{label * z1}x1 + {-label * z1}x2 + {label * z2}x3 + {-label * z2}x4 + {label}x5 >= 1\n"
```





Parte 2

Ataques Adversariales

1. Red Neuronal 95% accuracy Mnist

```
class MNISTModel(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = torch.nn.Conv2d(1, 32, 3, 1)
        self.conv2 = torch.nn.Conv2d(32, 64, 3, 1)
        self.max_pool2d = torch.nn.MaxPool2d((2, 2))
        self.dropout1 = torch.nn.Dropout2d(0.25)
        self.conv3 = torch.nn.Conv2d(64, 128, 3, 1)
        self.conv4 = torch.nn.Conv2d(128, 256, 3, 1)
        self.max_pool2d2 = torch.nn.MaxPool2d((2, 2))
        self.dropout2 = torch.nn.Dropout2d(0.5)
        self.flatten = torch.nn.Flatten()
        self.fc1 = torch.nn.Linear(4096, 128)
        self.dropout3 = torch.nn.Dropout(0.5)
        self.fc2 = torch.nn.Linear(128, 10)
```

Arquitectura empleada:

- Capas convolucionales: (entrada, filtro, tamaño, salida)
- Max Pooling: reduce el espacio de dimensión
- Dropout: apaga neuronas al azar durante el entrenamiento
- Flatten
- Linear: Capa densa

1. Red Neuronal 95% accuracy Mnist

```python
def forward(self, x):
    x = self.conv1(x)
    x = torch.relu(x)
    x = self.conv2(x)
    x = torch.relu(x)
    x = self.max_pool2d(x)
    x = self.dropout1(x)
    x = self.conv3(x)
    x = torch.relu(x)
    x = self.conv4(x)
    x = torch.relu(x)
    x = self.max_pool2d2(x
    x = self.dropout2(x)
    x = self.flatten(x)
    x = self.fc1(x)
    x = torch.relu(x)
    x = self.dropout3(x)
    output = self.fc2(x)

    return output
```

```python
# funcion de perdida y optimizador
criterion = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

CrossEntropyLoss contiene la función de activación SoftMax

Adam: método de descenso de gradiente estocástico

Activación Relu

1. Red Neuronal 95% accuracy Mnist

```
X_train = X_train.reshape(-1, 1, 28, 28)
X_test = X_test.reshape(-1, 1, 28, 28)
# normalizamos
X_train = X_train / 255
X_test = X_test / 255
```

Normalizamos el dataset u lo entrenamos con torch:

- Se pone a cero los gradientes
- Se hace la predicción
- Se calcula la pérdida y los gradientes
- Se ajusta el modelo

```
model.train()

running_loss = 0.0
for i, (data, target) in enumerate(train_loader):
    # Move data and target to GPU
    data, target = data.to(device), target.to(device)
    # zero the parameter gradients
    optimizer.zero_grad()
    # forward + backward + optimize
    output = model(data)
    loss = criterion(output, target)
    loss.backward()
    optimizer.step()

    running_loss += loss.item()

print(f'Epoch: {epoch}, Training loss: {running_loss / le
```

1. Red Neuronal 95% accuracy Mnist

Evaluamos el modelo mientras se entrena.

Early stopping

Guardar el mejor modelo

```
model.eval()
with torch.no_grad():
    test_loss = 0
    correct = 0
    for data, target in test_loader:
        data, target = data.to(device), target.to(device)

        output = model(data)
        test_loss += criterion(output, target).item()
        pred = output.data.max(1, keepdim=True)[1]
        correct += pred.eq(target.data.view_as(pred)).sum()


    test_loss /= len(test_loader.dataset)
    if test_loss < best_loss:
        best_loss = test_loss
        best_model = model.state_dict()

    print(f'Epoch: {epoch}, Test set: Average loss: {test_lo

if test_loss < best_loss:
    intentos = 10
if test_loss > best_loss:
    intentos -= 1
    if intentos == 0:
        break
```

1. Red Neuronal 95% accuracy Mnist

```python
from prettytable import PrettyTable

def count_parameters(model):
    table = PrettyTable(["Modules", "Parameters"])
    total_params = 0
    for name, parameter in model.named_parameters():
        if not parameter.requires_grad:
            continue
        params = parameter.numel()
        table.add_row([name, params])
        total_params += params
    print(table)
    print(f"Total Trainable Params: {total_params}")
    return total_params

count_parameters(model)
```

```
+--------------+------------+
|   Modules    | Parameters |
+--------------+------------+
| conv1.weight |    288     |
|  conv1.bias  |     32     |
| conv2.weight |   18432    |
|  conv2.bias  |     64     |
| conv3.weight |   73728    |
|  conv3.bias  |    128     |
| conv4.weight |   294912   |
|  conv4.bias  |    256     |
|  fc1.weight  |   524288   |
|   fc1.bias   |    128     |
|  fc2.weight  |    1280    |
|   fc2.bias   |     10     |
+--------------+------------+
Total Trainable Params: 913546
```

2. SVM Núcleo gaussiano

```python
# importamos SVM
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
from tensorflow.keras.datasets import mnist

classifier = SVC(kernel='rbf', random_state=42)

# Cargamos el dataset
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train / 255.00
x_test = x_test / 255.0

# entrenamos con mnist
classifier.fit(x_train.reshape(-1, 784), y_train)
y_pred_svm = classifier.predict(x_test.reshape(-1, 784))
print(f'Accuracy: {accuracy_score(y_test, y_pred_svm) * 100:.2f}%')
```

✓ 3m 54.9s

```
Accuracy: 97.92%
```

Kernel trick permite el uso de clasificadores lineales para problemas no lineales

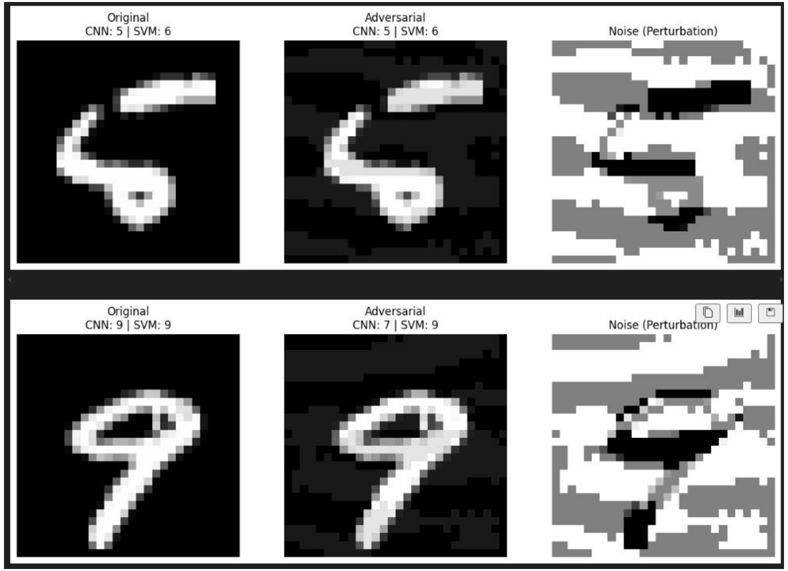
3. Ataques FGSM

```python
def fgsm_attack(image, epsilon, data_grad):
    # Obtener el signo del gradiente
    sign_data_grad = data_grad.sign()
    # Crear la imagen perturbada
    perturbed_image = image + epsilon * sign_data_grad
    # Clip values to [0, 1]
    perturbed_image = torch.clamp(perturbed_image, 0, 1)
    return perturbed_image
```

El objetivo es maximizar la pérdida. Encuentra cuánto contribuye cada pixel al valor de pérdida y agrega una perturbación en consecuencia

4. Generación de algunas imágenes adversariales

```
epsilon = 0.3
model.eval()
for i in range(total_samples):

    # Obtenemos la imagen y la etiqueta
    image = X_test[i].unsqueeze(0).to(device)
    label = y_test[i].item()

    # Predecimos la imagen original con el modelo CNN
    output = model(image)
    _, cnn_original_pred = torch.max(output, 1)
    cnn_original_pred = cnn_original_pred.item()

    # Se calcula el gradiente de la imagen
    image.requires_grad = True
    output = model(image)
    loss = criterion(output, torch.tensor([label]).to(device))
    loss.backward()
    data_grad = image.grad.data

    # Generamos la imagen adversaria
    perturbed_image = fgsm_attack(image, epsilon, data_grad)

    # Predecimos con el modelo CNN
    output = model(perturbed_image)
    _, cnn_adversarial_pred = torch.max(output, 1)
    cnn_adversarial_pred = cnn_adversarial_pred.item()

    # Predecimos con el modelo SVM
    svm_original_pred = classifier.predict(image.detach().numpy().reshape(1, 784))[0]
    svm_adversarial_pred = classifier.predict(perturbed_image.cpu().detach().reshape(1, 784))[0]

    visualize_images(image, perturbed_image, cnn_original_pred, cnn_adversarial_pred, svm_original_pred,

    aciertos += cnn_adversarial_pred == label

print(f'Aciertos: {aciertos}/{total_samples}')
```



5. Error adversariales

```python
if not os.path.exists('adversariales'):
    os.makedirs('adversariales')
    for i in range(10):
        os.makedirs(f'adversariales/{i}')

epsilon = 0.1

model.eval()

for i, (data, target) in enumerate(tqdm(test_loader)):
    data, target = data.to(device), target.to(device)

    data.requires_grad = True

    output = model(data)
    loss = criterion(output, target)

    model.zero_grad()
    loss.backward()
    data_grad = data.grad.data

    perturbed_data = fgsm_attack(data, epsilon, data_grad)

    for j in range(perturbed_data.size(0)):
        clase = target[j].item()
        save_image(perturbed_data[j], f'adversariales/{clase}/adversarial_{i * test_loader.batch_size + j}.png')
```

5. Error adversariales

```
if not os.path.exists('adversariales'):
    os.makedirs('adversariales')
    for i in range(10):
        os.makedirs(f'adversariales/{i}')

epsilon = 0.1

model.eval()

for i, (data, target) in enumerate(tqdm(test_loader)):
    data, target = data.to(device), target.to(device)

    data.requires_grad = True

    output = model(data)
    loss = criterion(output, target)

    model.zero_grad()
    loss.backward()
    data_grad = data.grad.data

    perturbed_data = fgsm_attack(data, epsilon, data_grad)

    for j in range(perturbed_data.size(0)):
        clase = target[j].item()
        save_image(perturbed_data[j], f'adversariales/{clase}/adversarial_{i * test_loader.batch_size + j}.png')
```

5. Error adversariales

```python
cnn_adversarial_preds = []
svm_adversarial_preds = []

path = 'adversariales'
# normalizar y cambiar a un canal
adversarial_dataset = torchvision.datasets.ImageFolder(path,
    transform=transforms.Compose([transforms.ToTensor(),
                transforms.Grayscale(num_output_channels=1)])
)
adversarial_loader = torch.utils.data.DataLoader(adversarial_dataset, batch_size=1000, shuffle=False)
y_test = adversarial_dataset.targets

model.eval()
for data, target in tqdm(adversarial_loader):

    data, target = data.to(device), target.to(device)

    with torch.no_grad():
        adversarial_pred_cnn = model(data).argmax(1)
        cnn_adversarial_preds.extend(adversarial_pred_cnn.cpu().numpy())

    adversarial_pred_svm = classifier.predict(data.cpu().detach().numpy().reshape(-1, 784))
    svm_adversarial_preds.extend(adversarial_pred_svm)



# Calculate the accuracy on adversarial images
cnn_adversarial_accuracy = accuracy_score(y_test, cnn_adversarial_preds)
svm_adversarial_accuracy = accuracy_score(y_test, svm_adversarial_preds)
print(f'CNN Adversarial Accuracy: {cnn_adversarial_accuracy * 100:.2f}%')
print(f'SVM Adversarial Accuracy: {svm_adversarial_accuracy * 100:.2f}%')
```

✓ 1m 1.3s

```
100%|██████████| 10/10 [01:01<00:00,  6.13s/it]
CNN Adversarial Accuracy: 84.95%
SVM Adversarial Accuracy: 88.94%
```

6. Entrena adversarialmente

```
model.eval()

cnn_preds = 0
svm_preds = 0

with torch.no_grad():
    for data, target in tqdm(test_loader_all):
        data, target = data.to(device), target.to(device)
        output = model(data)
        pred = output.argmax(1)

        svm_pred = classifier.predict(data.cpu().detach().numpy().reshape(-1, 784))

        cnn_preds += pred.eq(target).sum().item()
        svm_preds += np.sum(svm_pred == target.cpu().numpy())

cnn_accuracy = cnn_preds / len(test_loader_all.dataset)
svm_accuracy = svm_preds / len(test_loader_all.dataset)

print(f'CNN Accuracy: {cnn_accuracy * 100:.2f}%')
print(f'SVM Accuracy: {svm_accuracy * 100:.2f}%')
```

✓ 1m 56.1s

```
100%|██████████| 20/20 [01:56<00:00,  5.81s/it]
CNN Accuracy: 98.44%
SVM Accuracy: 93.69%
```

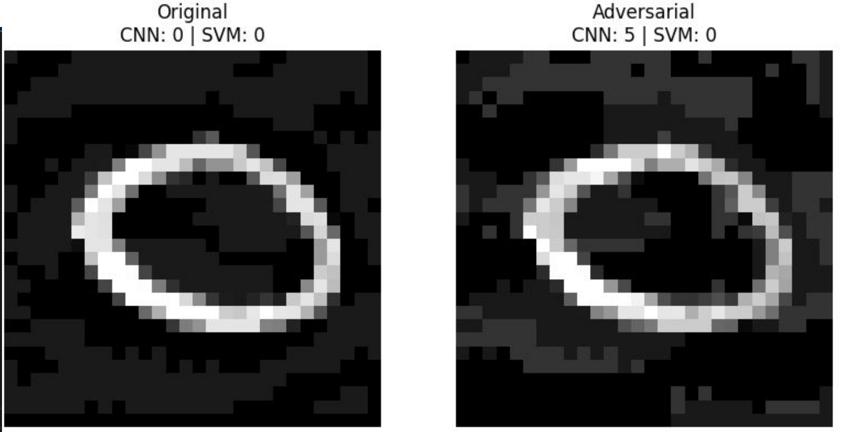
```
path = 'adversariales_train'
path_test = 'adversariales'

image_transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Grayscale(num_output_channels=1)
])
label_transform = transforms.Lambda(lambda y: torch.tensor(y, dtype=torch.int64))

adversarial_dataset = torchvision.datasets.ImageFolder(
    path,
    transform=image_transform,
    target_transform=label_transform
)

adversarial_dataset_test = torchvision.datasets.ImageFolder(
    path_test,
    transform=image_transform,
    target_transform=label_transform
)

(x_train_original, y_train_original), (x_test_original, y_test_original) = mnist.load_data()
x_train_original = x_train_original / 255.0
x_test_original = x_test_original / 255.0

x_train_original = torch.tensor(x_train_original, dtype=torch.float32).unsqueeze(1)  # Shape:
x_test_original = torch.tensor(x_test_original, dtype=torch.float32).unsqueeze(1)    # Shape:
y_train_original = torch.tensor(y_train_original, dtype=torch.int64)
y_test_original = torch.tensor(y_test_original, dtype=torch.int64)

train_original = TensorDataset(x_train_original, y_train_original)
test_original = TensorDataset(x_test_original, y_test_original)

train_dataset_all = ConcatDataset([train_original, adversarial_dataset])
test_dataset_all = ConcatDataset([test_original, adversarial_dataset_test])

train_loader_all = DataLoader(train_dataset_all, batch_size=1000, shuffle=True)
test_loader_all = DataLoader(test_dataset_all, batch_size=1000, shuffle=False)
```

6. Vs Doblemente adversariales

```python
epsilon = 0.1
model.eval()
cnn_predictions = []
svm_predictions = []
y_test = adversarial_loader.dataset.targets

model.eval()
for data, target in tqdm(adversarial_loader):
    data, target = data.to(device), target.to(device)

    data.requires_grad = True
    output = model(data)
    loss = criterion(output, target)
    model.zero_grad()
    loss.backward()
    data_grad = data.grad.data
    doubly_adversarial_data = fgsm_attack(data, epsilon, data_grad)

    with torch.no_grad():
        cnn_predictions.append(model(doubly_adversarial_data).argmax(dim=1))
        svm_predictions.append(classifier.predict(doubly_adversarial_data.view(doubly_adversarial_data.size(0), -1).cpu().numpy()))

cnn_predictions = torch.cat(cnn_predictions).cpu().numpy()
svm_predictions = np.concatenate(svm_predictions)

cnn_accuracy = np.mean(cnn_predictions == y_test)
svm_accuracy = np.mean(svm_predictions == y_test)

print(f"CNN accuracy on doubly adversarial images: {cnn_accuracy}")
print(f"SVM accuracy on doubly adversarial images: {svm_accuracy}")
```

✓ 7m 10.7s

```
100%|██████████| 60/60 [07:10<00:00,  7.18s/it]
CNN accuracy on doubly adversarial images: 0.3798833333333335
SVM accuracy on doubly adversarial images: 0.9216166666666666
```



FIN