

## ✓ Exercício 1 — Problema das Espirais

Considere o problema das espirais. Sendo a espiral 1 uma classe e a espiral 2 outra classe, gere os dados usando as seguintes equações:

- Para espiral 1:

$$x = \frac{\theta}{4} \cos(\theta), \quad y = \frac{\theta}{4} \sin(\theta), \quad \theta \geq 0$$

- Para espiral 2:

$$x = \left( \frac{\theta}{4} + 0.8 \right) \cos(\theta), \quad y = \left( \frac{\theta}{4} + 0.8 \right) \sin(\theta), \quad \theta \geq 0$$

Considere  $\theta$  assumindo **1000 valores igualmente espaçados** entre 0 e 20 radianos.

---

### Objetivo:

Solucione este problema de classificação considerando:

**a) Uma Máquina de Vetor de Suporte (SVM)**

**b) Um comitê de máquinas**, formado por:

- Uma **rede Perceptron de uma camada oculta**
- Uma **RBF**
- Uma **SVM**

## ✓ a) Uma Máquina de Vetor de Suporte (SVM)

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from sklearn.gaussian_process.kernels import RBF
from sklearn.pipeline import make_pipeline
from sklearn.ensemble import VotingClassifier
from sklearn.metrics import classification_report, ConfusionMatrixDisplay
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import LogisticRegression
from sklearn.cluster import KMeans
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.preprocessing import OneHotEncoder
```

```
from scipy.special import softmax

# Definir o intervalo de theta
theta = np.linspace(0, 20, 1000)

# Espiral 1
x1 = (theta / 4) * np.cos(theta)
y1 = (theta / 4) * np.sin(theta)
labels1 = np.zeros_like(theta)

# Espiral 2
x2 = ((theta / 4) + 0.8) * np.cos(theta)
y2 = ((theta / 4) + 0.8) * np.sin(theta)
labels2 = np.ones_like(theta)

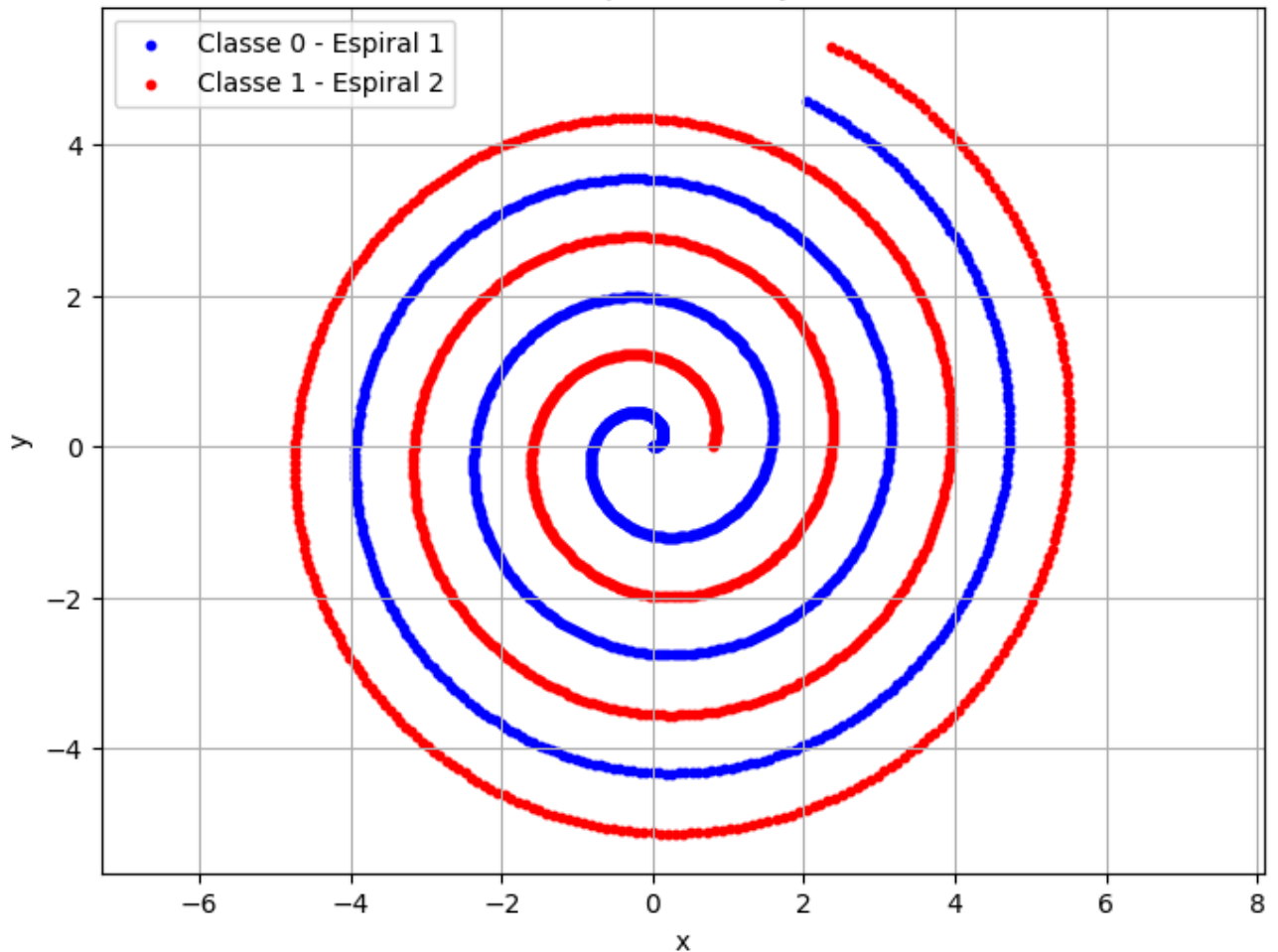
# Concatenar os dados
X = np.vstack((np.column_stack((x1, y1)), np.column_stack((x2, y2))))
y = np.concatenate((labels1, labels2))

# Criar um DataFrame
spiral_data = pd.DataFrame(X, columns=["x", "y"])
spiral_data["label"] = y

# Plotar as espirais
plt.figure(figsize=(8, 6))
plt.scatter(x1, y1, c='blue', label='Classe 0 - Espiral 1', s=10)
plt.scatter(x2, y2, c='red', label='Classe 1 - Espiral 2', s=10)
plt.title("Problema das Espirais - Conjunto de Dados")
plt.xlabel("x")
plt.ylabel("y")
plt.legend()
plt.grid(True)
plt.axis("equal")
plt.show()
```



## Problema das Espirais - Conjunto de Dados



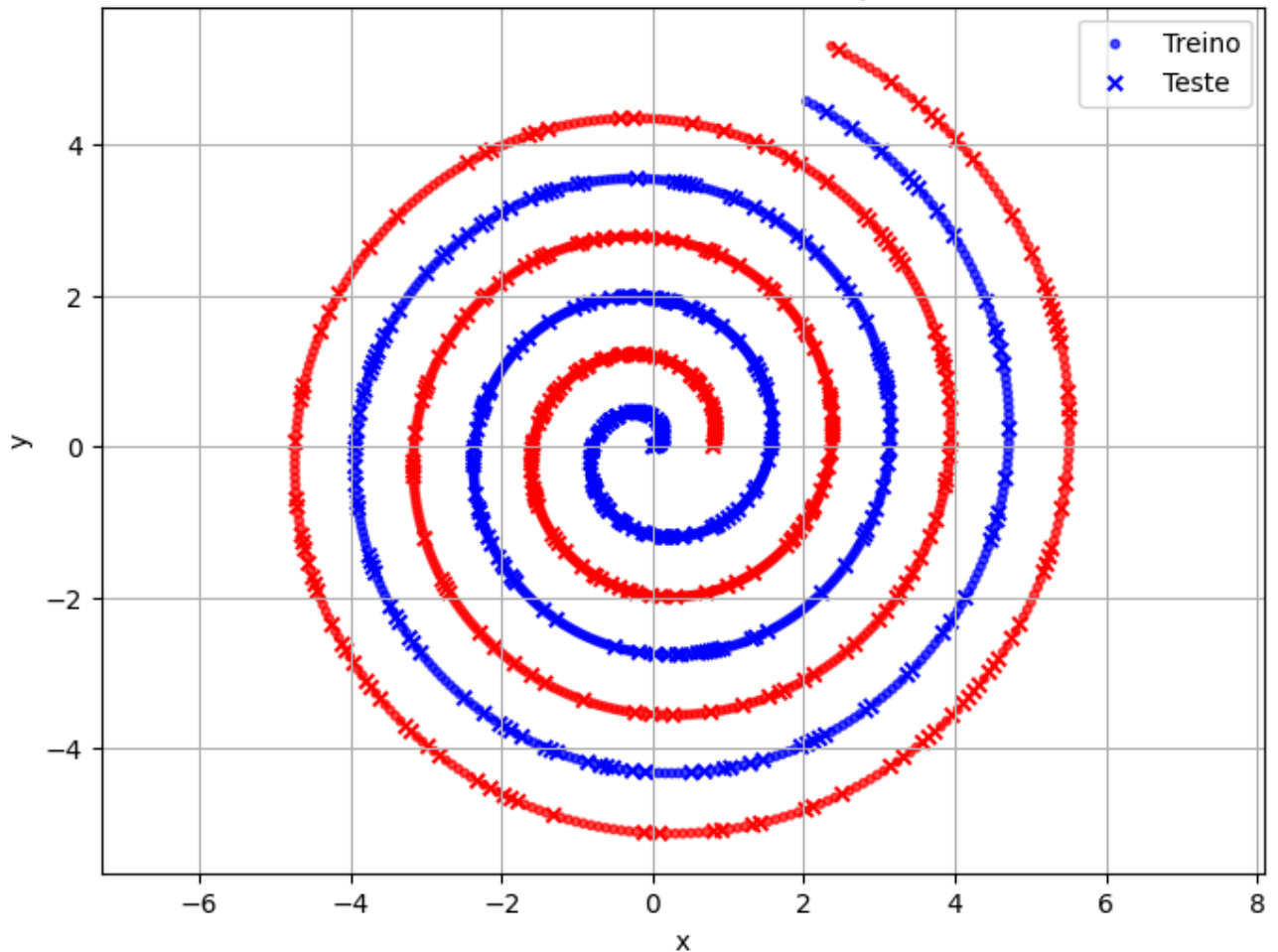
```
# Separar features e rótulos
X_data = spiral_data[["x", "y"]].values
y_data = spiral_data["label"].values

# Dividir em treino e teste
X_train, X_test, y_train, y_test = train_test_split(X_data, y_data, test_size=0.3)

plt.figure(figsize=(8, 6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap='bwr', label='Treino',
            plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='bwr', marker='x', label='
plt.title("Dados de Treino e Teste - Espirais")
plt.xlabel("x")
plt.ylabel("y")
plt.legend(["Treino", "Teste"])
plt.grid(True)
plt.axis("equal")
plt.show()
```

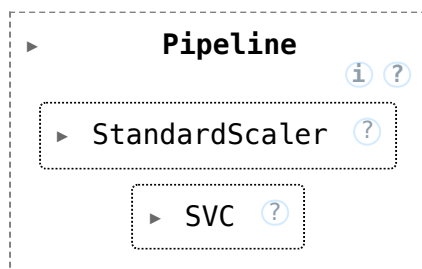


### Dados de Treino e Teste - Espirais



```
# Criar e treinar o modelo SVM com kernel RBF
svm_model = make_pipeline(
    StandardScaler(),
    SVC(kernel='rbf', C=300, gamma='auto', probability=True, random_state=42)
, memory=None)
```

```
svm_model.fit(X_train, y_train)
```



```
# Previsão e avaliação
y_pred_svm_rbf = svm_model.predict(X_test)
report_svm_rbf = classification_report(y_test, y_pred_svm_rbf, output_dict=True)
```

```
# Exibir relatório
pd_svm_rbf = pd.DataFrame(report_svm_rbf).transpose()
```

```
pd_svm_rbf = pd_svm_rbf.round(2)
print("Relatório de Classificação - SVM com Kernel RBF")
pd_svm_rbf
```



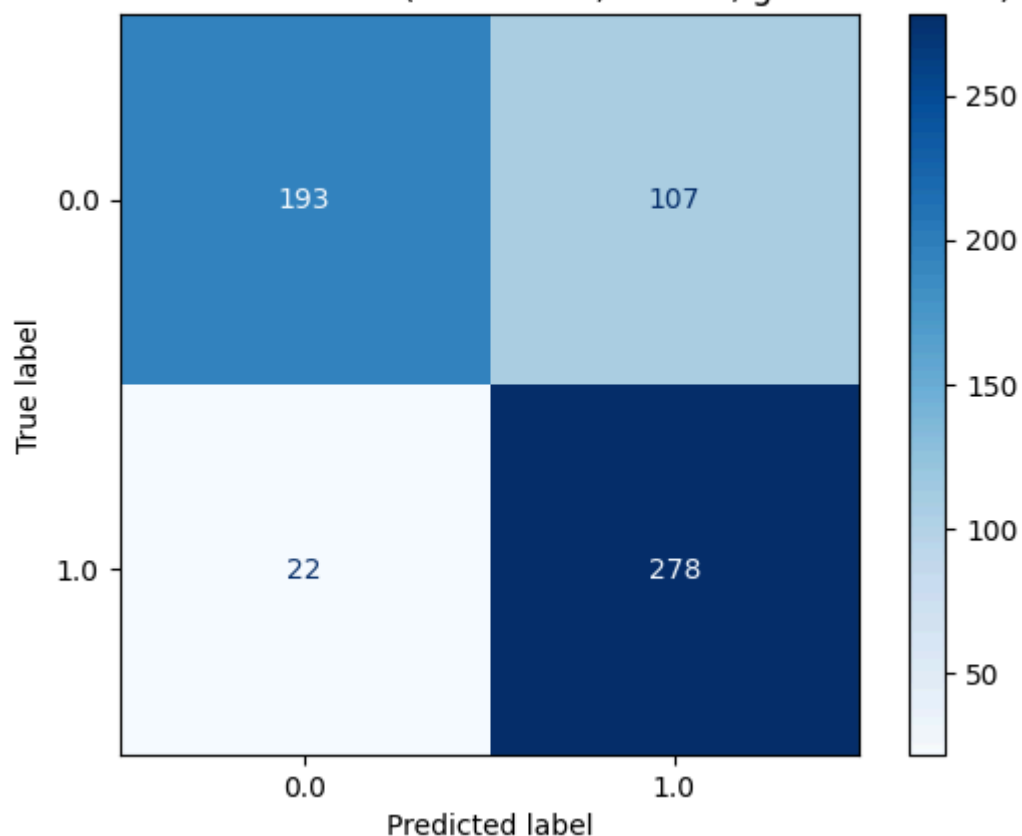
Relatório de Classificação - SVM com Kernel RBF

	precision	recall	f1-score	support
<b>0.0</b>	0.90	0.64	0.75	300.00
<b>1.0</b>	0.72	0.93	0.81	300.00
<b>accuracy</b>	0.78	0.78	0.78	0.78
<b>macro avg</b>	0.81	0.78	0.78	600.00
<b>weighted avg</b>	0.81	0.78	0.78	600.00

```
# Matriz de confusão
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svm_rbf, cmap='Blues')
plt.title("Matriz de Confusão - SVM (RBF Kernel, C=300, gamma='auto')")
plt.show()
```



Matriz de Confusão - SVM (RBF Kernel, C=300, gamma='auto')



```
# Separar pontos corretamente e incorretamente classificados
correct_idx = np.where(y_test == y_pred_svm_rbf)[0]
incorrect_idx = np.where(y_test != y_pred_svm_rbf)[0]
```

```
# Plot
plt.figure(figsize=(8, 6))
```

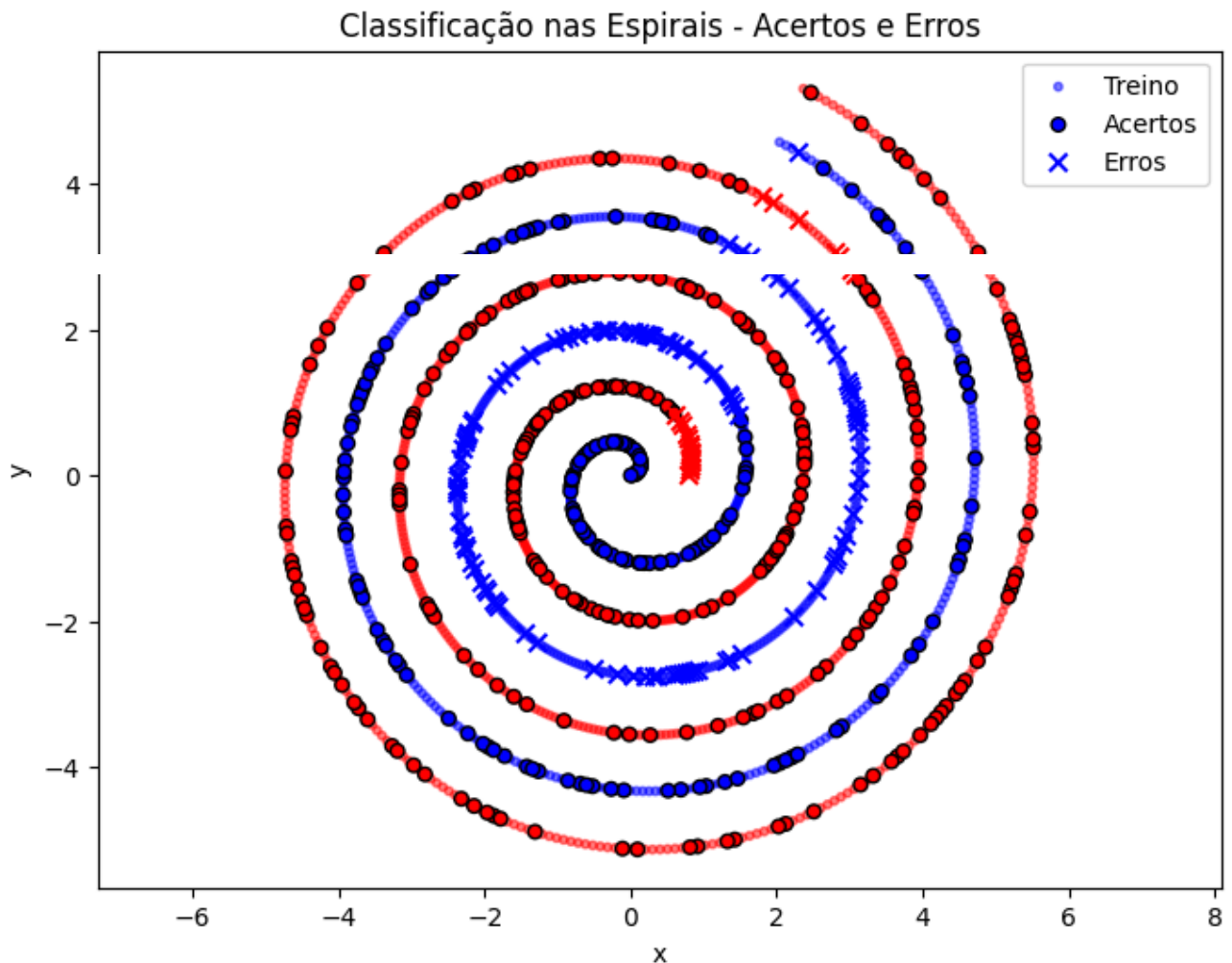
```
# Treino (sem alterações)
```

```
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap='bwr', s=10, alpha=0.5,

# Teste - Acertos
plt.scatter(X_test[correct_idx, 0], X_test[correct_idx, 1],
            c=y_test[correct_idx], cmap='bwr', marker='o', s=30, edgecolors='k',

# Teste - Erros
plt.scatter(X_test[incorrect_idx, 0], X_test[incorrect_idx, 1],
            c=y_test[incorrect_idx], cmap='bwr', marker='x', s=50, linewidths=1.5

plt.title("Classificação nas Espirais - Acertos e Erros")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(False)
plt.axis("equal")
plt.legend()
plt.show()
```



✓ **Comitê de máquinas**, formado por MPL + RBF + SVM

```
svm_model = make_pipeline(
    StandardScaler(),
```

```
SVC(kernel='rbf', C=300, gamma='auto', probability=True, random_state=42)
, memory=None)

# Criar a MLP com uma camada oculta de 10 neurônios
mlp_model = make_pipeline(
    StandardScaler(),
    MLPClassifier(hidden_layer_sizes=(10,), max_iter=1000, random_state=42)
, memory=None)

from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.cluster import KMeans
from sklearn.preprocessing import OneHotEncoder
from scipy.special import softmax
import numpy as np

class RBFNetwork(BaseEstimator, ClassifierMixin):
    def __init__(self, n_centers=10, gamma=1.0, random_state=42):
        self.n_centers = n_centers
        self.gamma = gamma
        self.random_state = random_state

    def fit(self, X, y):
        self.classes_ = np.unique(y) # Agora 100% compatível
        self.encoder_ = OneHotEncoder(sparse_output=False)
        Y_onehot = self.encoder_.fit_transform(y.reshape(-1, 1))

        kmeans = KMeans(n_clusters=self.n_centers, random_state=self.random_state)
        self.centers_ = kmeans.fit(X).cluster_centers_

        Phi = self._rbf_activation(X)
        self.output_weights_ = np.linalg.pinv(Phi) @ Y_onehot

        return self

    def _rbf_activation(self, X):
        return np.exp(-self.gamma * np.linalg.norm(X[:, np.newaxis, :] - self.centers_, axis=2))

    def predict_proba(self, X):
        Phi = self._rbf_activation(X)
        output = Phi @ self.output_weights_
        return softmax(output, axis=1)

    def predict(self, X):
        proba = self.predict_proba(X)
        return self.classes_[np.argmax(proba, axis=1)]

rbf_model = RBFNetwork(n_centers=10, gamma=1.5)

# 1. Treine separadamente cada modelo
mlp_model.fit(X_train, y_train)
```

```
svm_model.fit(X_train, y_train)
rbf_model.fit(X_train, y_train) # sem pipeline
```



▼ RBFNetwork ⓘ  
RBFNetwork(gamma=1.5)

```
# 2. Pegue as probabilidades previstas para o conjunto de teste
proba_mlp = mlp_model.predict_proba(X_test)
proba_svm = svm_model.predict_proba(X_test)
proba_rbf = rbf_model.predict_proba(X_test)
```

```
# 3. Combine as probabilidades (votação soft - média simples)
avg_proba = (proba_mlp + proba_svm + proba_rbf) / 3
```

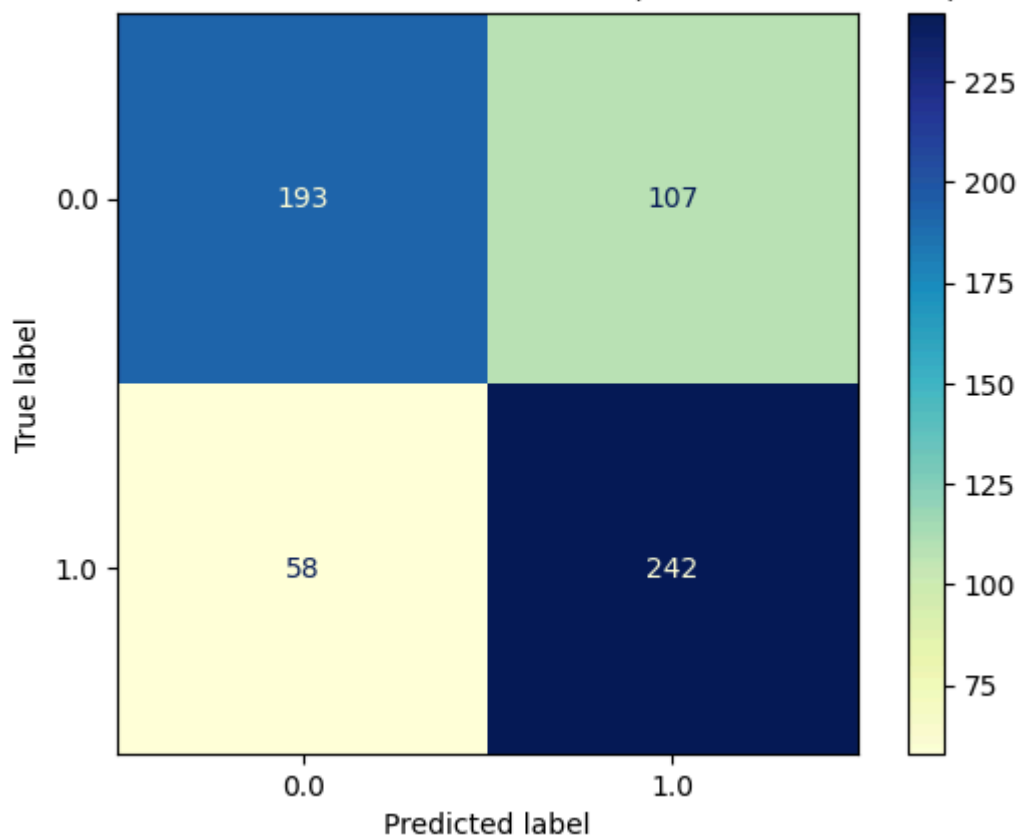
```
# 4. Pegue a classe com maior probabilidade média
y_pred_manual_committee = np.argmax(avg_proba, axis=1)
```

```
# 5. Avalie o desempenho
from sklearn.metrics import classification_report, ConfusionMatrixDisplay
```

```
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_manual_committee, cmap="YlGn")
plt.title("Matriz de Confusão - Comitê Manual (MLP + RBF + SVM)")
plt.show()
```



Matriz de Confusão - Comitê Manual (MLP + RBF + SVM)





```
report_manual_committee = classification_report(y_test, y_pred_manual_committee,  
pd_manual_committee = pd.DataFrame(report_manual_committee).transpose()  
pd_manual_committee = pd_manual_committee.round(2)  
print("Relatório de Classificação – Comitê Manual (MLP + RBF + SVM)")  
pd_manual_committee
```

➦ Relatório de Classificação – Comitê Manual (MLP + RBF + SVM)

	precision	recall	f1-score	support
<b>0.0</b>	0.77	0.64	0.70	300.00
<b>1.0</b>	0.69	0.81	0.75	300.00
<b>accuracy</b>	0.72	0.72	0.72	0.72
<b>macro avg</b>	0.73	0.72	0.72	600.00
<b>weighted avg</b>	0.73	0.72	0.72	600.00

```
# Plotar matrizes de confusão  
fig, axs = plt.subplots(1, 2, figsize=(12, 5))  
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_svm_rbf, ax=axs[0], cmap="  
axs[0].set_title("Matriz de Confusão – SVM")  
  
ConfusionMatrixDisplay.from_predictions(y_test, y_pred_manual_committee, ax=axs[1]  
axs[1].set_title("Matriz de Confusão – Comitê (MLP + RBF + SVM)")  
  
plt.tight_layout()  
plt.show()
```



Matriz de Confusão - SVM



Matriz de Confusão - Comitê (MLP + RBF + SVM)



225

```
# Separar pontos corretamente e incorretamente classificados
correct_idx = np.where(y_test == y_pred_manual_committee)[0]
incorrect_idx = np.where(y_test != y_pred_manual_committee)[0]

# Plot
plt.figure(figsize=(8, 6))

# Treino (sem alterações)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap='bwr', s=10, alpha=0.5,

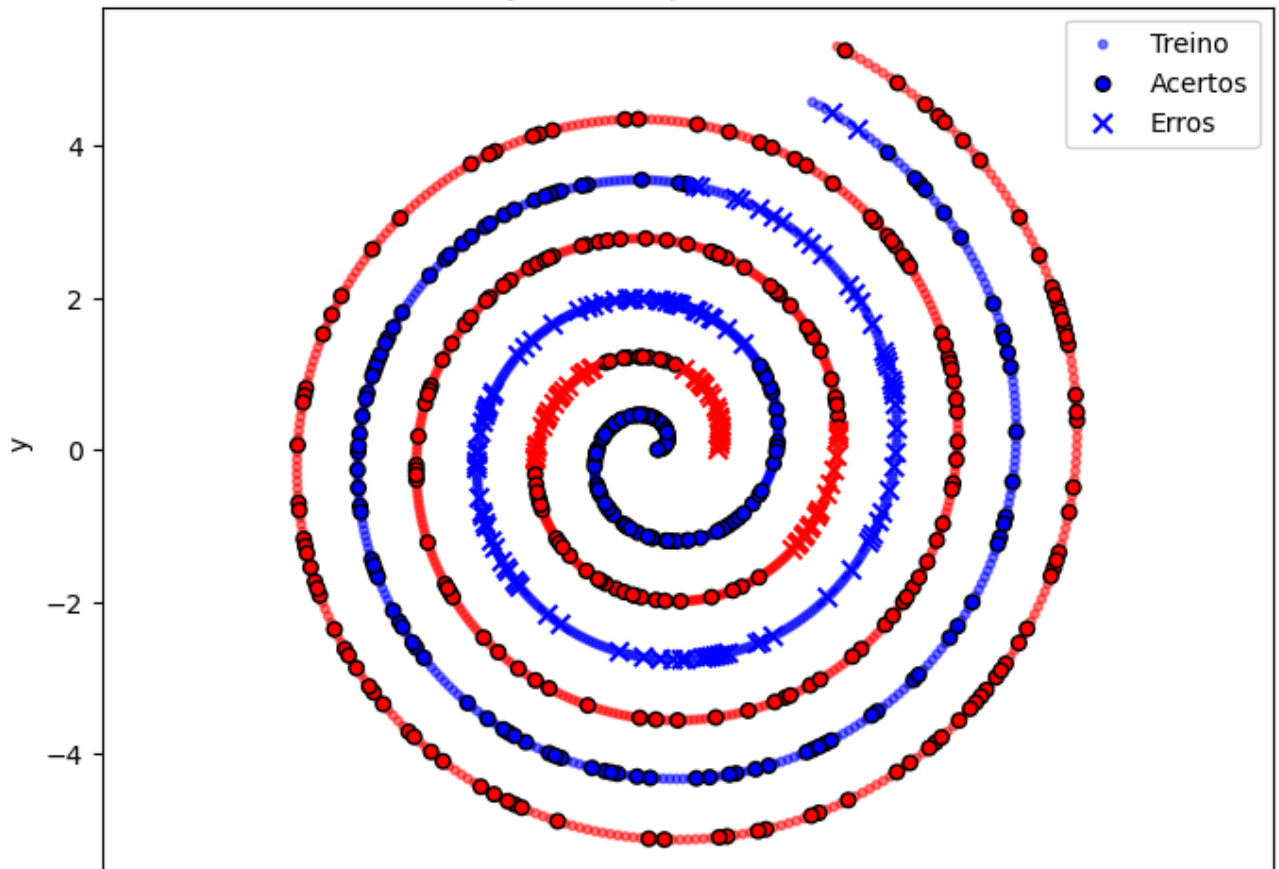
# Teste - Acertos
plt.scatter(X_test[correct_idx, 0], X_test[correct_idx, 1],
            c=y_test[correct_idx], cmap='bwr', marker='o', s=30, edgecolors='k',

# Teste - Erros
plt.scatter(X_test[incorrect_idx, 0], X_test[incorrect_idx, 1],
            c=y_test[incorrect_idx], cmap='bwr', marker='x', s=50, linewidths=1.5

plt.title("Classificação nas Espirais - Acertos e Erros")
plt.xlabel("x")
plt.ylabel("y")
plt.grid(False)
plt.axis("equal")
plt.legend()
plt.show()
```



Classificação nas Espirais - Acertos e Erros



## ✓ Exercício 2 - Classificação Imagens -> CIFAR-10

Considere uma rede deep learning convolutiva (treinada) aplicada à classificação de padrões em imagens. A base de dados considerada é a CIFAR-10 (pesquisa). A referida base de dados consiste de 60 mil imagens coloridas de 32x32 pixels, com 50 mil para treino e 10 mil para teste. As imagens estão divididas em 10 classes, a saber: avião, navio, caminhão, automóvel, sapo, pássaro, cachorro, gato, cavalo e cervo. Cada imagem possui apenas um dos objetos da classe de interesse, podendo estar parcialmente obstruído por outros objetos que não pertençam a esse conjunto. Apresente o desempenho da rede no processo de classificação usando uma matriz de confusão.

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential, load_model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Dense, Dropout, Flatten
from keras.models import Model
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.applications import VGG16

import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
```

### ✓ Load the CIFAR-10 dataset

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
```

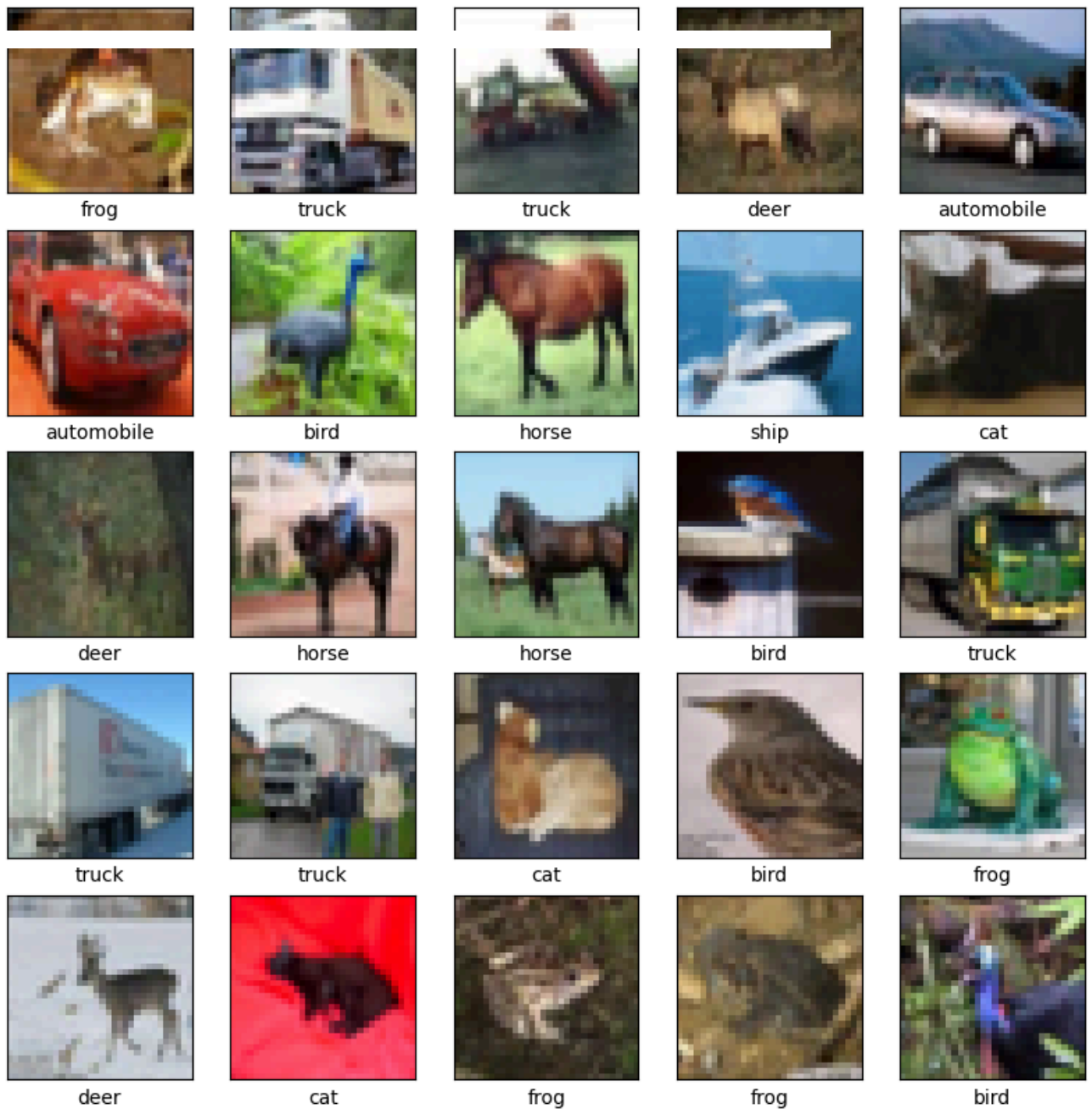
⏏ Downloading data from <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz>  
170498071/170498071 ————— 4s 0us/step

```
# verify the shape
print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')
```

⏏ x\_train shape: (50000, 32, 32, 3)  
50000 train samples  
10000 test samples

```
# transformando em strings a numeração da classe
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'h
```

```
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(5, 5, i+1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    plt.imshow(x_train[i], cmap=plt.cm.binary)
    plt.xlabel(class_names[y_train[i][0]])
plt.show()
```



```
x_train.dtype
```

```
dtype('uint8')
```

```
x_test.dtype
```

```
dtype('uint8')
```

## ✓ Raw Model

### ✓ Preprocess the data

```
#transform data in float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
```

```
# Normalization Data.
# By dividing by 255.0, you are scaling these pixel values to be between 0 and 1.
x_train /= 255.0
x_test /= 255.0
```

```
#transform y variable in categorical
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

### ✓ Define the model architecture

```
model = Sequential([
    Conv2D(32, (3, 3), activation='relu', padding='same', input_shape=(32, 32, 3)),
    BatchNormalization(),
    Conv2D(32, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.2),

    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(64, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    MaxPooling2D((2, 2)),
    Dropout(0.3),

    Conv2D(128, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
    Conv2D(128, (3, 3), activation='relu', padding='same'),
    BatchNormalization(),
```

```
MaxPooling2D((2, 2)),
Dropout(0.4),

Flatten(),
Dense(128, activation='relu'),
BatchNormalization(),
Dropout(0.5),
Dense(10, activation='softmax')
])
```

```
➦ /usr/local/lib/python3.11/dist-packages/keras/src/layers/convolutional/base_conv2d.py:100: in   
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
model.summary()
```



Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_1 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 128)	512
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147,584
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 128)	262,272
batch_normalization_6 (BatchNormalization)	(None, 128)	512
dropout_3 (Dropout)	(None, 128)	0
dense_1 (Dense)	(None, 10)	1,290

Total params: 552,874 (2.11 MB)

Trainable params: 551,722 (2.10 MB)

Non-trainable params: 1,152 (4.50 KB)

## ✓ Compile the model

```
model.compile(optimizer=Adam(learning_rate=0.001),  
              loss='categorical_crossentropy',  
              metrics=['accuracy'])
```

## ✓ Callbacks

```
checkpoint = ModelCheckpoint('model.h5', monitor='val_accuracy', save_best_only=True,  
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True))
```

## ✓ Train the model

```
history = model.fit(x_train, y_train,  
                    validation_data=(x_test, y_test),  
                    epochs=50,  
                    batch_size=64,  
                    callbacks=[checkpoint, early_stopping])
```





```

Epoch 33/50
775/782 ━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9135 - loss: 0.2472WARNING:
782/782 ━━━━━━━━━━━ 7s 9ms/step - accuracy: 0.9135 - loss: 0.2472 - val_
Epoch 34/50
782/782 ━━━━━━━━━━━ 10s 9ms/step - accuracy: 0.9135 - loss: 0.2478 - val_
Epoch 35/50
782/782 ━━━━━━━━━━━ 6s 8ms/step - accuracy: 0.9205 - loss: 0.2295 - val_
Epoch 36/50
782/782 ━━━━━━━━━━━ 7s 9ms/step - accuracy: 0.9194 - loss: 0.2295 - val_
Epoch 37/50
782/782 ━━━━━━━━━━━ 10s 9ms/step - accuracy: 0.9189 - loss: 0.2304 - val_
Epoch 38/50
781/782 ━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9210 - loss: 0.2280WARNING:
782/782 ━━━━━━━━━━━ 7s 9ms/step - accuracy: 0.9210 - loss: 0.2280 - val_
Epoch 39/50
782/782 ━━━━━━━━━━━ 7s 9ms/step - accuracy: 0.9239 - loss: 0.2193 - val_
Epoch 40/50
782/782 ━━━━━━━━━━━ 10s 9ms/step - accuracy: 0.9287 - loss: 0.2081 - val_
Epoch 41/50
782/782 ━━━━━━━━━━━ 7s 8ms/step - accuracy: 0.9249 - loss: 0.2178 - val_
Epoch 42/50
782/782 ━━━━━━━━━━━ 10s 8ms/step - accuracy: 0.9277 - loss: 0.2086 - val_
Epoch 43/50
775/782 ━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9302 - loss: 0.1985WARNING:
782/782 ━━━━━━━━━━━ 7s 9ms/step - accuracy: 0.9301 - loss: 0.1986 - val_

```

## ✓ Evaluate the model on the test set

```

test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print('\nTest accuracy:', test_acc)

```

```

⇒ 313/313 - 2s - 5ms/step - accuracy: 0.8646 - loss: 0.4333

```

```

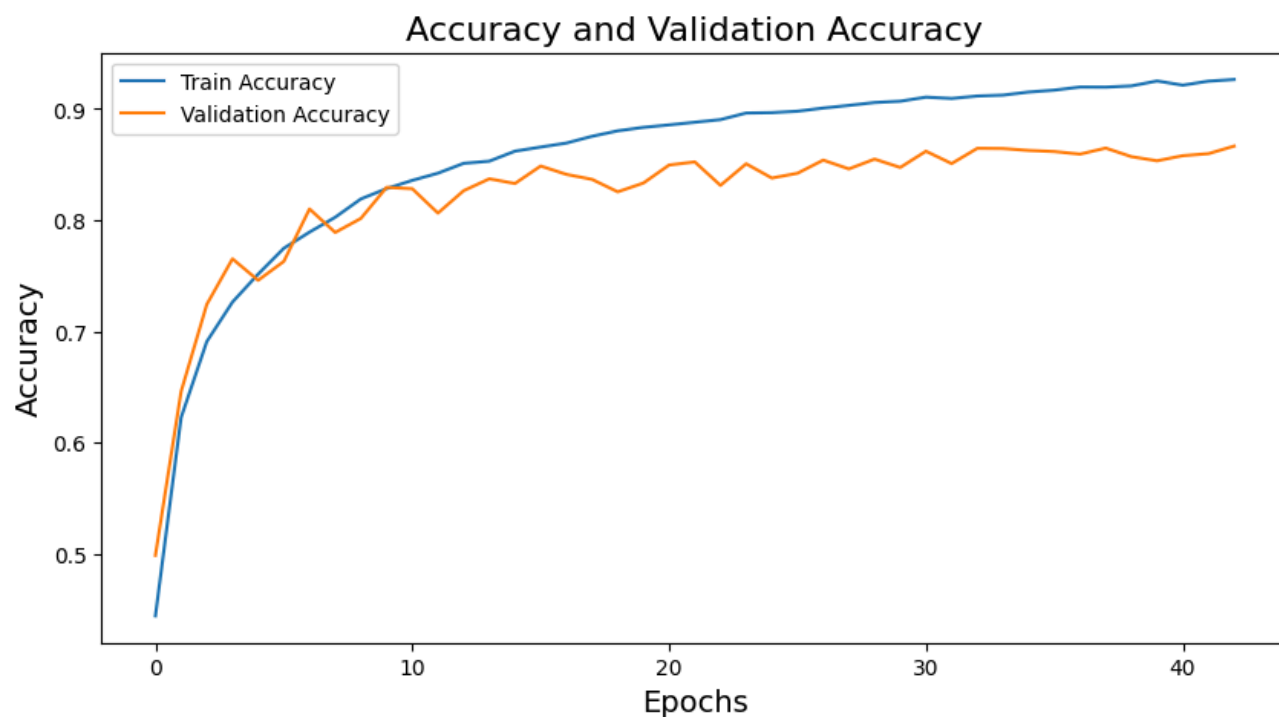
Test accuracy: 0.8646000027656555

```

```

plt.figure(figsize=(10, 5))
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy', fontsize=14)
plt.title('Accuracy and Validation Accuracy', fontsize=16)
plt.legend()
plt.show()

```



- ✓ Make predictions on the test set

```
y_pred = model.predict(x_test)
```



313/313 ————— 2s 4ms/step

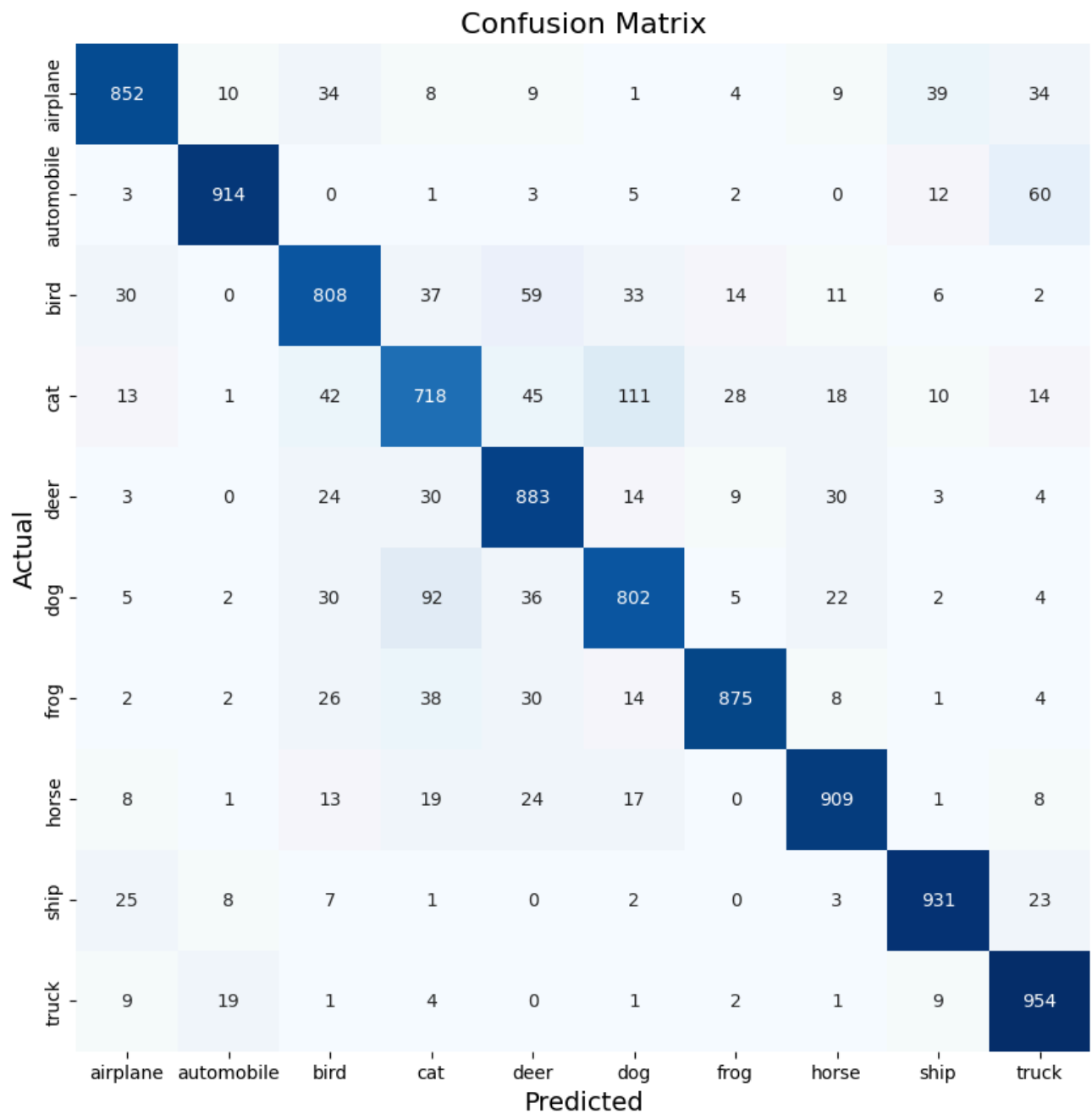
- ✓ Convert predictions from one-hot encoding to class labels

```
y_pred_classes = np.argmax(y_pred, axis=1)  
y_true_classes = np.argmax(y_test, axis=1)
```

- ✓ Create and Print the confusion matrix

```
cm = confusion_matrix(y_true_classes, y_pred_classes)
```

```
import matplotlib.pyplot as plt
plt.figure(figsize=(10, 10))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=class_names, ytick
plt.xlabel('Predicted', fontsize=14)
plt.ylabel('Actual', fontsize=14)
plt.title('Confusion Matrix', fontsize=16)
plt.show()
```



## ✓ Transfer Learning from DenseNet201

### ✓ Define the model architecture

```
# instantiate the model
vgg16 = VGG16(
    include_top=False,
    weights="imagenet",
    input_shape=(32, 32, 3),
)

vgg16.summary()
```

➔ Downloading data from <https://storage.googleapis.com/tensorflow/keras-applications/58889256/58889256> 0s 0us/step  
Model: "vgg16"

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 32, 32, 3)	0
block1_conv1 (Conv2D)	(None, 32, 32, 64)	1,792
block1_conv2 (Conv2D)	(None, 32, 32, 64)	36,928
block1_pool (MaxPooling2D)	(None, 16, 16, 64)	0
block2_conv1 (Conv2D)	(None, 16, 16, 128)	73,856
block2_conv2 (Conv2D)	(None, 16, 16, 128)	147,584
block2_pool (MaxPooling2D)	(None, 8, 8, 128)	0
block3_conv1 (Conv2D)	(None, 8, 8, 256)	295,168
block3_conv2 (Conv2D)	(None, 8, 8, 256)	590,080
block3_conv3 (Conv2D)	(None, 8, 8, 256)	590,080
block3_pool (MaxPooling2D)	(None, 4, 4, 256)	0
block4_conv1 (Conv2D)	(None, 4, 4, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 4, 4, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 4, 4, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 2, 2, 512)	0
block5_conv1 (Conv2D)	(None, 2, 2, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 2, 2, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 2, 2, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 1, 1, 512)	0

Total params: 14,714,688 (56.13 MB)  
Trainable params: 14,714,688 (56.13 MB)  
Non-trainable params: 0 (0.00 B)

```
# Congela os pesos das camadas do modelo base
for layer in vgg16.layers:
    layer.trainable = False
```

```
# Adiciona as camadas personalizadas ao modelo
x = Flatten()(vgg16.output)
x = Dense(128, activation='relu')(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
output = Dense(10, activation='softmax')(x)
```

```
# Cria o novo modelo com as camadas personalizadas
```

```
model = Model(inputs=vgg16.input, outputs=output)
```

```
# Visualiza a arquitetura do modelo  
model.summary()
```

⇒ **Model: "functional\_1"**

Layer (type)	Output Shape	Param #
input_layer_1 (InputLayer)	(None, 32, 32, 3)	0
block1_conv1 (Conv2D)	(None, 32, 32, 64)	1,792
block1_conv2 (Conv2D)	(None, 32, 32, 64)	36,928
block1_pool (MaxPooling2D)	(None, 16, 16, 64)	0
block2_conv1 (Conv2D)	(None, 16, 16, 128)	73,856
block2_conv2 (Conv2D)	(None, 16, 16, 128)	147,584
block2_pool (MaxPooling2D)	(None, 8, 8, 128)	0
block3_conv1 (Conv2D)	(None, 8, 8, 256)	295,168
block3_conv2 (Conv2D)	(None, 8, 8, 256)	590,080
block3_conv3 (Conv2D)	(None, 8, 8, 256)	590,080
block3_pool (MaxPooling2D)	(None, 4, 4, 256)	0
block4_conv1 (Conv2D)	(None, 4, 4, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 4, 4, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 4, 4, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 2, 2, 512)	0
block5_conv1 (Conv2D)	(None, 2, 2, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 2, 2, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 2, 2, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 1, 1, 512)	0
flatten_1 (Flatten)	(None, 512)	0
dense_2 (Dense)	(None, 128)	65,664
batch_normalization_7 (BatchNormalization)	(None, 128)	512
dropout_4 (Dropout)	(None, 128)	0
dense_3 (Dense)	(None, 10)	1,290

**Total params:** 14,782,154 (56.39 MB)

**Trainable params:** 67,210 (262.54 KB)

**Non-trainable params:** 14,714,944 (56.13 MB)

## ✓ Compile the model

```
# compile the model
model.compile(optimizer=Adam(learning_rate=0.001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
```

## ✓ Callbacks

```
checkpoint = ModelCheckpoint('model.h5', monitor='val_accuracy', save_best_only=True)
early_stopping = EarlyStopping(monitor='val_loss', patience=10, restore_best_weights=True)
```

## ✓ Train the model

```
history = model.fit(x_train, y_train,
                    validation_data=(x_test, y_test),
                    epochs=50,
                    batch_size=64,
                    callbacks=[checkpoint, early_stopping])
```



```

102/102 ————— 20s 14ms/step - accuracy: 0.6130 - loss: 1.1290 ·
Epoch 39/50
779/782 ————— 0s 12ms/step - accuracy: 0.6111 - loss: 1.1246WAI
782/782 ————— 11s 14ms/step - accuracy: 0.6111 - loss: 1.1246 ·
Epoch 40/50
782/782 ————— 11s 14ms/step - accuracy: 0.6102 - loss: 1.1204 ·
Epoch 41/50
782/782 ————— 21s 15ms/step - accuracy: 0.6053 - loss: 1.1370 ·
Epoch 42/50
782/782 ————— 11s 14ms/step - accuracy: 0.6050 - loss: 1.1326 ·
Epoch 43/50
782/782 ————— 20s 14ms/step - accuracy: 0.6074 - loss: 1.1407 ·
Epoch 44/50
782/782 ————— 11s 14ms/step - accuracy: 0.6133 - loss: 1.1228 ·
Epoch 45/50
782/782 ————— 11s 14ms/step - accuracy: 0.6034 - loss: 1.1322 ·
Epoch 46/50
782/782 ————— 12s 15ms/step - accuracy: 0.6061 - loss: 1.1337 ·
Epoch 47/50
782/782 ————— 20s 15ms/step - accuracy: 0.6104 - loss: 1.1236 ·
Epoch 48/50
782/782 ————— 20s 14ms/step - accuracy: 0.6110 - loss: 1.1252 ·
Epoch 49/50
782/782 ————— 21s 14ms/step - accuracy: 0.6090 - loss: 1.1281 ·
Epoch 50/50
782/782 ————— 21s 15ms/step - accuracy: 0.6072 - loss: 1.1218 ·

```

## ✓ Evaluate the model on test set

```

test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)
print('\nTest accuracy:', test_acc)

```

```

⇒ 313/313 - 4s - 12ms/step - accuracy: 0.6009 - loss: 1.1351

```

```

Test accuracy: 0.6008999943733215

```

```

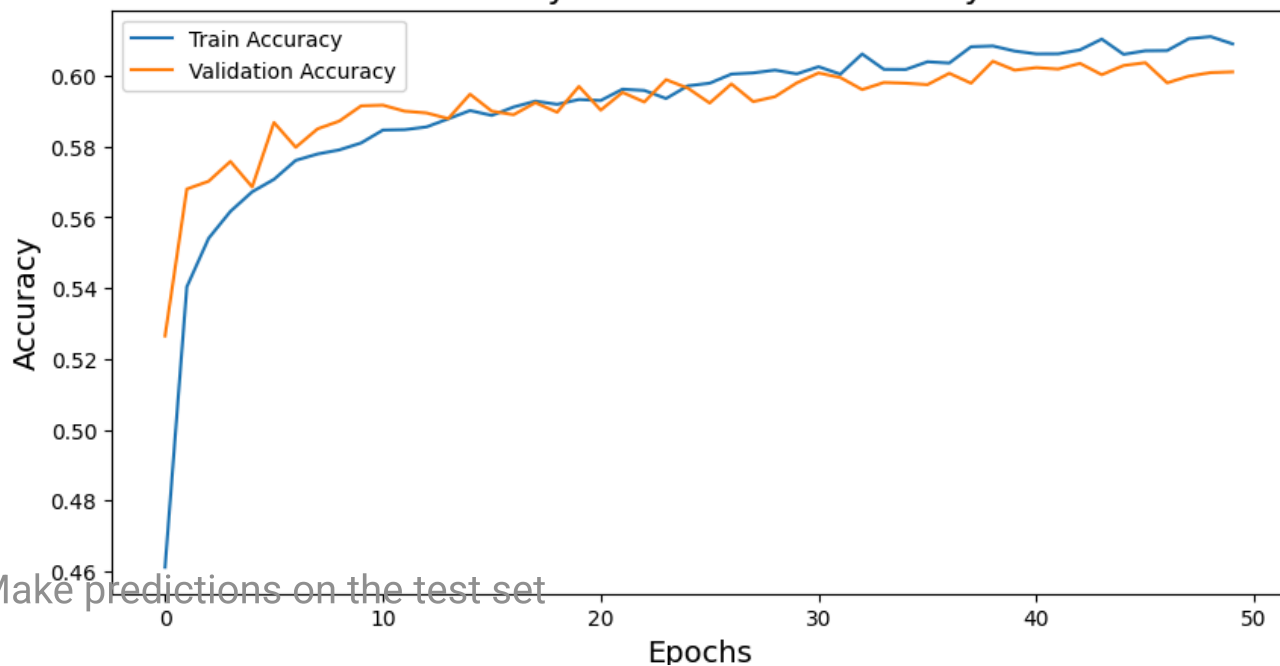
plt.figure(figsize=(10, 5))
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs', fontsize=14)
plt.ylabel('Accuracy', fontsize=14)
plt.title('Accuracy and Validation Accuracy', fontsize=16)
plt.legend()
plt.show()

```





Accuracy and Validation Accuracy



Make predictions on the test set

```
y_pred = model.predict(x_test)
```



313/313

4s 10ms/step

3) Considere quatro distribuições gaussianas,  $C_1, C_2, C_3, C_4$ , em um espaço de entrada de dimensionalidade igual a oito, isto é  $\mathbf{x} = (x_1, x_2, \dots, x_8)^t$ .

### Visualização 2D com Autoencoder

Neste notebook, vamos:

1. Gerar 4 distribuições gaussianas de 8 dimensões com médias distintas.
2. Treinar uma rede **autoencoder** para reduzir essas 8 dimensões para 2.
3. Visualizar os dados codificados em 2D.

a) Todas as nuvens de dados formadas têm variâncias unitárias, mas centros ou vetores média são diferentes e dados por  $\mathbf{m}_1 = (0, 0, 0, 0, 0, 0, 0, 0)^t$ ,  $\mathbf{m}_2 = (4, 0, 0, 0, 0, 0, 0, 0)^t$ ,  $\mathbf{m}_3 = (0, 0, 4, 0, 0, 0, 0, 0)^t$ ,  $\mathbf{m}_4 = (0, 0, 0, 0, 0, 0, 0, 4)^t$ .

**Resposta:** Cada uma das distribuições gaussianas tem variância unitária e diferentes médias dadas por:

- $\mathbf{m}_1 = (0, 0, 0, 0, 0, 0, 0, 0)^T$
- $\mathbf{m}_2 = (4, 0, 0, 0, 0, 0, 0, 0)^T$
- $\mathbf{m}_3 = (0, 0, 4, 0, 0, 0, 0, 0)^T$
- $\mathbf{m}_4 = (0, 0, 0, 0, 0, 0, 0, 4)^T$

b) Utilizar uma rede de autoencoder para visualizar os dados em duas dimensões.

**Resposta:** Foi utilizada uma rede autoencoder para reduzir a dimensionalidade dos dados para duas dimensões (2d), ou seja, uma rede neural que aprenda a codificar os vetores de 8 dimensões em um espaço de 2 dimensões e consiga reconstruir os dados originais a partir disso.

c) O objetivo é visualizar os dados de dimensão 8 em um espaço de dimensão 2.

**Resposta:** O objetivo principal é visualizar os dados originalmente em 8 dimensões em um novo espaço bidimensional (2D), preservando suas estruturas e relações.

d) Apresente os dados neste novo espaço.

**Resposta:** Os dados foram apresentados no novo espaço bidimensional, mostrando as quatro classes de forma visualmente separável, onde cada ponto representa um vetor codificado e cores distintas indicam cada uma das classes  $C_1, C_2, C_3, C_4$ .

```

import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# **Fixar semente para reprodutibilidade**
np.random.seed(42)
torch.manual_seed(42)

```

## Criar as quatro distribuições gaussianas

```

means = [
    np.array([0, 0, 0, 0, 0, 0, 0, 0]),
    np.array([4, 0, 0, 0, 0, 0, 0, 0]),
    np.array([0, 0, 4, 0, 0, 0, 0, 0]),
    np.array([0, 0, 0, 0, 0, 0, 0, 4])
]

num_samples_per_class = 500
X, y = [], []

# Gerar amostras com variância unitária
for i, mean in enumerate(means):
    cov = np.eye(8)
    samples = np.random.multivariate_normal(mean, cov, num_samples_per_class)
    X.append(samples)
    y.append(np.full(num_samples_per_class, i))

X = np.vstack(X)
y = np.concatenate(y)

# Converter para tensores

X_tensor = torch.tensor(X, dtype=torch.float32)

```

## Definir Autoencoder

```

class Autoencoder(nn.Module):
    def __init__(self):
        super(Autoencoder, self).__init__()
        self.encoder = nn.Sequential(
            nn.Linear(8, 4),
            nn.ReLU(),
            nn.Linear(4, 2)
        )
        self.decoder = nn.Sequential(
            nn.Linear(2, 4),
            nn.ReLU(),
            nn.Linear(4, 8)
        )

```

```

def forward(self, x):
    z = self.encoder(x)
    x_recon = self.decoder(z)
    return x_recon

# Instanciar modelo
model = Autoencoder()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=1e-3)

# Treinamento
epochs = 200
for epoch in range(epochs):
    optimizer.zero_grad()
    outputs = model(X_tensor)
    loss = criterion(outputs, X_tensor)
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 20 == 0:
        print(f"Epoch [{epoch+1}/{epochs}], Loss: {loss.item():.4f}")

```

## Obter representações em 2D

```

with torch.no_grad():
    encoded = model.encoder(X_tensor).numpy()

```

## Visualizar

```

plt.figure(figsize=(8, 6))
colors = ['red', 'green', 'blue', 'orange']
labels = ['C1', 'C2', 'C3', 'C4']

for i in range(4):
    plt.scatter(encoded[y == i, 0], encoded[y == i, 1],
                label=labels[i], alpha=0.6, color=colors[i])

plt.title('Visualização 2D com Autoencoder')
plt.xlabel('Dimensão 1')
plt.ylabel('Dimensão 2')
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()

```



#### Questão 4

**a-)** Utilizando uma rede NARX no caso uma rede neural perceptron de múltiplas camadas com realimentação global para fazer a predição de um passo, isto é  $\hat{x}(n + 1)$ , da série temporal.

**Resposta:** Utilizando uma rede NARX (rede neural perceptron de múltiplas camadas com realimentação global), foi feita a predição de um passo da série temporal  $\hat{x}(n + 1)$ . A rede foi treinada com os dados da série  $x(n)$  gerados pela equação:

$$x(n) = v(n) + \beta v(n - 1)v(n - 2), \quad \beta = 0,5$$

com  $v(n) \sim \mathcal{U}(0, 1)$ .

**b-)** Repetir o problema utilizando uma rede LSTM. Avalie o desempenho das redes recorrentes mostrando a curva da série temporal, a curva de predição e a curva do erro de predição  $e(n + 1) = x(n + 1) - \hat{x}(n + 1)$ .

**Resposta:** O mesmo problema foi repetido utilizando uma rede LSTM e as curvas obtidas foram:

- **Curva da série temporal:** sequência real de  $x(n)$ .
- **Curva de predição:** saída da rede  $\hat{x}(n + 1)$ .
- **Curva do erro de predição:**  $e(n + 1) = x(n + 1) - \hat{x}(n + 1)$ .

---

Gerar a série temporal  $x(n)$

$x(n) = v(n) + \beta v(n-1)v(n-2)$  com  $\beta = 0.5$   $\beta=0.5$ , onde  $v(n)$  é ruído branco gaussiano (média 0, variância 1).

```
import numpy as np
N = 1000
beta = 0.5
np.random.seed(42)
v = np.random.randn(N)
x = np.zeros(N)
for n in range(2, N):
    x[n] = v[n] + beta * v[n-1] * v[n-2]
```

Preparar os dados

a) Para NARX: Usamos os valores passados como entradas:  $x(n-1), x(n-2)$   
→ prever  $x(n+1)$ .

```
def preparar_dados_narx(x, atrasos=2):
    X, Y = [], []
    for i in range(atrasos, len(x)-1):
        X.append([x[i-1], x[i-2]])
        Y.append(x[i+1])
    return np.array(X), np.array(Y)
```

b) Para LSTM: Mesmo conjunto, mas com shape 3D: [samples, timesteps, features].

```
def preparar_dados_lstm(x, atrasos=2):
    X, Y = [], []
    for i in range(atrasos, len(x)-1):
        X.append([x[i-2]], [x[i-1]]])
        Y.append(x[i+1])
    return np.array(X), np.array(Y)
```

Dividir treino/teste e normalizar os dados python Copiar Editar

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler

# Preparar os dados
X_narx, y_narx = preparar_dados_narx(x)
X_lstm, y_lstm = preparar_dados_lstm(x)

# Divisão
X_train_narx, X_test_narx, y_train_narx, y_test_narx = train_test_split(X_narx, y_narx)
X_train_lstm, X_test_lstm, y_train_lstm, y_test_lstm = train_test_split(X_lstm, y_lstm)

# Normalização
scaler = MinMaxScaler()
X_train_narx = scaler.fit_transform(X_train_narx)
X_test_narx = scaler.transform(X_test_narx)

X_train_lstm = scaler.fit_transform(X_train_lstm.reshape(-1, 2)).reshape(-1, 2, 1)
X_test_lstm = scaler.transform(X_test_lstm.reshape(-1, 2)).reshape(-1, 2, 1)
```

Treinar e testar os modelos

a) NARX (simulada com MLP)

LSTM python Copiar Editar

```
from tensorflow.keras.layers import LSTM

model_lstm = Sequential([
    LSTM(10, input_shape=(2, 1)),
    Dense(1)
])
```

```
model_lstm.compile(optimizer=Adam(learning_rate=0.01), loss='mse')
model_lstm.fit(X_train_lstm, y_train_lstm, epochs=50)
```

Avaliação e gráfico python Copiar Editar

```
import matplotlib.pyplot as plt

# Predição
y_pred_narx = model_narx.predict(X_test_narx).flatten()
y_pred_lstm = model_lstm.predict(X_test_lstm).flatten()

erro_narx = y_test_narx - y_pred_narx
erro_lstm = y_test_lstm - y_pred_lstm

# Plot
plt.figure(figsize=(14, 10))
plt.subplot(3, 1, 1)
plt.plot(y_test_narx, label='Real')
plt.plot(y_pred_narx, label='Predição NARX')
plt.title('Predição da Série Temporal – Rede NARX')
plt.legend()

plt.subplot(3, 1, 2)
plt.plot(y_test_lstm, label='Real')
plt.plot(y_pred_lstm, label='Predição LSTM')
plt.title('Predição da Série Temporal – Rede LSTM')
plt.legend()

plt.subplot(3, 1, 3)
plt.plot(erro_narx, label='Erro NARX')
plt.plot(erro_lstm, label='Erro LSTM')
plt.title('Erro de Predição')
plt.legend()

plt.tight_layout()
plt.show()
```





## ✓ Rede LSTM para previsão de Fechamento de uma Ação

## ✓ Importando Bibliotecas necessarias para Deep Learning e Mercado Financeiro


```
import yfinance as yf
import pandas as pd
import numpy as np
from sklearn.preprocessing import MinMaxScaler
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense, Dropout
```

## ✓ Download dos Dados e visualização do dataset

```
# Baixar dados
ticker = 'AAPL'
df = yf.download(ticker, start="2018-01-01", end="2023-12-31")
df = df[['Open', 'High', 'Low', 'Close', 'Volume']]
```


 [\*\*\*\*\*100%\*\*\*\*\*] 1 of 1 completed

```
df.head()
```



Price	Open	High	Low	Close	Volume
Ticker	AAPL	AAPL	AAPL	AAPL	AAPL
Date					
2018-01-02	39.933986	40.436212	39.722768	40.426823	102223600
2018-01-03	40.490187	40.964251	40.356418	40.419781	118071600
2018-01-04	40.492539	40.710798	40.384586	40.607536	89738400
2018-01-05	40.703747	41.156687	40.612220	41.069855	94640000
2018-01-08	40.917320	41.213022	40.818749	40.917320	82271200

```
df.info()
```



```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1509 entries, 2018-01-02 to 2023-12-29
Data columns (total 5 columns):
#   Column              Non-Null Count  Dtype
---  -
0   (Open, AAPL)         1509 non-null   float64
1   (High, AAPL)         1509 non-null   float64
2   (Low, AAPL)          1509 non-null   float64
3   (Close, AAPL)        1509 non-null   float64
4   (Volume, AAPL)       1509 non-null   int64
dtypes: float64(4), int64(1)
memory usage: 70.7 KB
```

## ✓ Normalização dos dados

```
# Escalador geral para features
scaler_full = MinMaxScaler()
df_scaled = pd.DataFrame(scaler_full.fit_transform(df), columns=df.columns, index=df.index)

# Escalador dedicado para o target
scaler_target = MinMaxScaler()
df_scaled['Close'] = scaler_target.fit_transform(df[['Close']])
```

## ✓ Criando sequencias temporais multivariaveis

```
def create_multivariate_sequences(dataframe, window_size=60, target_column='Close'):
    data = dataframe.values
    target_index = dataframe.columns.get_loc(target_column)
    X, y = [], []
    for i in range(window_size, len(data)):
        X.append(data[i - window_size:i])
        y.append(data[i, target_index])
    return np.array(X), np.array(y)

window_size = 60
X, y = create_multivariate_sequences(df_scaled, window_size=window_size, target_column='Close')
```

## ✓ Dividindo os dados em Treino e Teste

```
# Dividir treino/teste
split = int(len(X) * 0.8)
X_train, X_test = X[:split], X[split:]
y_train, y_test = y[:split], y[split:]
```

## ✓ Criando modelo LSTM Muti variaveis

```
model = Sequential([
    LSTM(128, return_sequences=True, input_shape=(X_train.shape[1], X_train.shape[2])),
    Dropout(0.2),
    LSTM(64),
    Dropout(0.2),
    Dense(1)
])
```

➔ /usr/local/lib/python3.11/dist-packages/keras/src/layers/rnn/rnn.py:200: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer when it already has one of these arguments. Call arguments passed: super().\_\_init\_\_(\*\*kwargs)

```
model.summary()
```

➔ Model: "sequential\_3"

Layer (type)	Output Shape	Param #
lstm_6 (LSTM)	(None, 60, 128)	68,608
dropout_6 (Dropout)	(None, 60, 128)	0
lstm_7 (LSTM)	(None, 64)	49,408
dropout_7 (Dropout)	(None, 64)	0
dense_3 (Dense)	(None, 1)	65

Total params: 118,081 (461.25 KB)  
 Trainable params: 118,081 (461.25 KB)  
 Non-trainable params: 0 (0.00 B)

## ✓ Compilando e Treinando o Modelo

```
model.compile(optimizer='adam', loss='mean_squared_error')
model.fit(X_train, y_train, epochs=30, batch_size=32, verbose=1)
```

➔ 37/37 ————— 5s 97ms/step - loss: 0.0025  
 Epoch 3/30  
 37/37 ————— 4s 96ms/step - loss: 0.0022  
 Epoch 4/30  
 37/37 ————— 4s 111ms/step - loss: 0.0022  
 Epoch 5/30  
 37/37 ————— 5s 96ms/step - loss: 0.0021  
 Epoch 6/30  
 37/37 ————— 5s 98ms/step - loss: 0.0024  
 Epoch 7/30  
 37/37 ————— 7s 155ms/step - loss: 0.0024

```

37/37 ----- 3s 97ms/step - loss: 0.0019
Epoch 10/30
37/37 ----- 5s 97ms/step - loss: 0.0021
Epoch 11/30
37/37 ----- 7s 134ms/step - loss: 0.0017
Epoch 12/30
37/37 ----- 4s 98ms/step - loss: 0.0023
Epoch 13/30
37/37 ----- 4s 97ms/step - loss: 0.0022
Epoch 14/30
37/37 ----- 6s 116ms/step - loss: 0.0018
Epoch 15/30
37/37 ----- 4s 98ms/step - loss: 0.0017
Epoch 16/30
37/37 ----- 6s 137ms/step - loss: 0.0017
Epoch 17/30
37/37 ----- 4s 98ms/step - loss: 0.0017
Epoch 18/30
37/37 ----- 4s 101ms/step - loss: 0.0015
Epoch 19/30
37/37 ----- 5s 142ms/step - loss: 0.0015
Epoch 20/30
37/37 ----- 4s 98ms/step - loss: 0.0018
Epoch 21/30
37/37 ----- 4s 99ms/step - loss: 0.0020
Epoch 22/30
37/37 ----- 7s 140ms/step - loss: 0.0018
Epoch 23/30
37/37 ----- 9s 98ms/step - loss: 0.0019
Epoch 24/30
37/37 ----- 5s 138ms/step - loss: 0.0016
Epoch 25/30
37/37 ----- 4s 97ms/step - loss: 0.0015
Epoch 26/30
37/37 ----- 4s 99ms/step - loss: 0.0015
Epoch 27/30
37/37 ----- 5s 139ms/step - loss: 0.0015
Epoch 28/30
37/37 ----- 9s 98ms/step - loss: 0.0014
Epoch 29/30
37/37 ----- 5s 138ms/step - loss: 0.0015
Epoch 30/30
37/37 ----- 4s 98ms/step - loss: 0.0015
<keras.src.callbacks.historyv.History at 0x7cfe46c4f090>

```

## ✓ Plot das Previsões

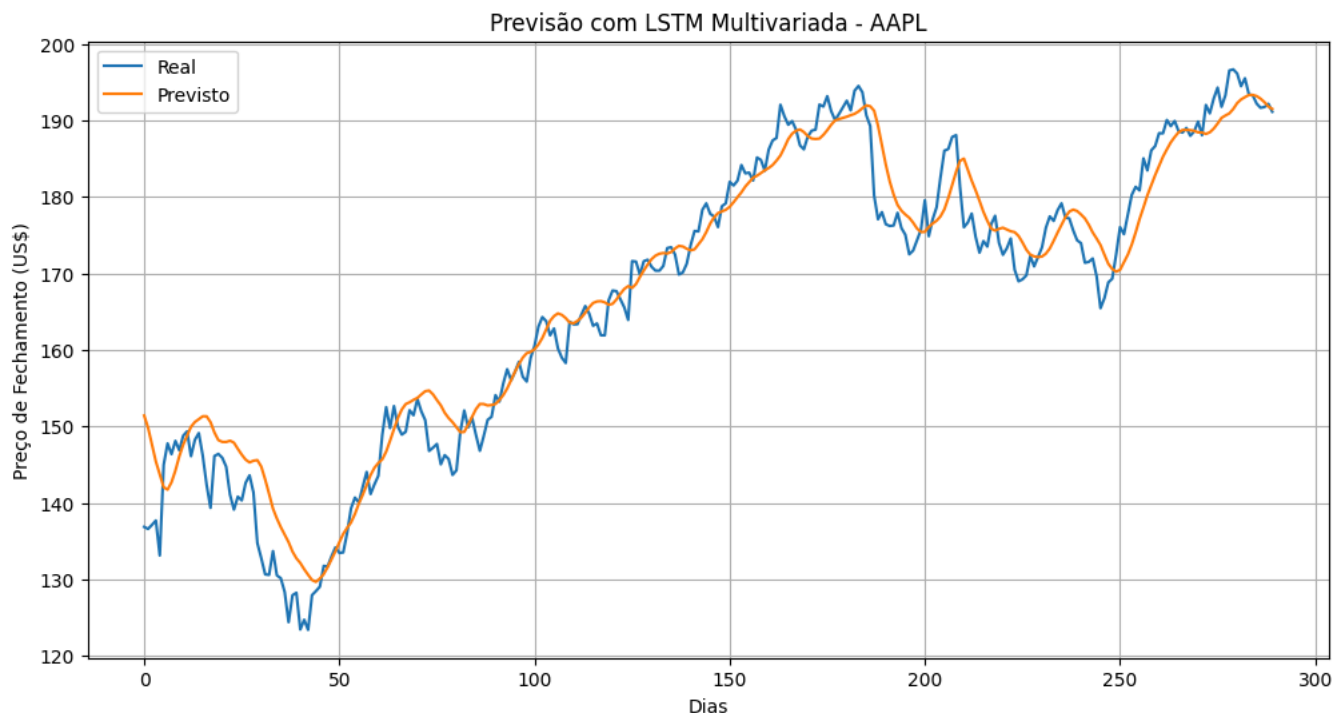
```

# Prever
y_pred_scaled = model.predict(X_test)
y_pred = scaler_target.inverse_transform(y_pred_scaled)
y_test_rescaled = scaler_target.inverse_transform(y_test.reshape(-1, 1))

# Plotar
plt.figure(figsize=(12, 6))
plt.plot(y_test_rescaled, label='Real')
plt.plot(y_pred, label='Previsto')
plt.title(f'Previsão com LSTM Multivariada - {ticker}')
plt.xlabel('Dias')
plt.ylabel('Preço de Fechamento (US$)')
plt.legend()
plt.grid(True)
plt.show()

```

10/10 1s 66ms/step

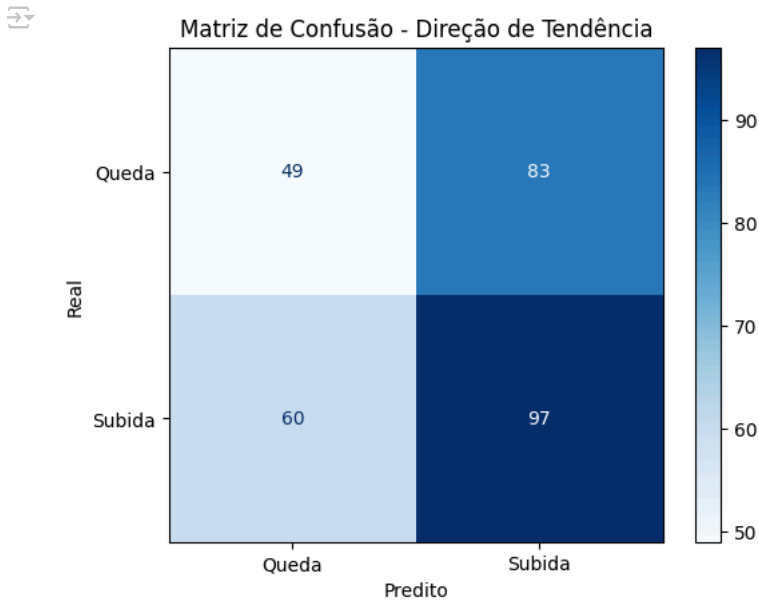


## ✓ Prever na Realidade

```
def prever_proximo_valor_multivariado(df_original, model, window_size, scaler_full, scaler_target):  
    df_recent = df_original[-window_size:]  
    df_scaled_recent = scaler_full.transform(df_recent)  
    entrada = df_scaled_recent.reshape((1, window_size, df_scaled_recent.shape[1]))  
  
    pred_scaled = model.predict(entrada)  
  
    return scaler_target.inverse_transform(pred_scaled)[0, 0]  
  
previsao_proximo_dia = prever_proximo_valor_multivariado(df, model, window_size, scaler_full, scaler_target)  
print(f"Previsão para o próximo dia: US$ {previsao_proximo_dia:.2f}")
```

1/1 0s 64ms/step  
Previsão para o próximo dia: US\$ 191.18

```
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay  
  
# 1. Converter previsões e valores reais em arrays unidimensionais  
y_pred_flat = y_pred.flatten()  
y_real_flat = y_test_rescaled.flatten()  
  
# 2. Calcular direções (1 = subida, 0 = descida ou igual)  
def calcular_direcao(series):  
    return (np.diff(series) > 0).astype(int)  
  
direcao_real = calcular_direcao(y_real_flat)  
direcao_pred = calcular_direcao(y_pred_flat)  
  
# 3. Calcular matriz de confusão  
cm = confusion_matrix(direcao_real, direcao_pred)  
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=["Queda", "Subida"])  
  
# 4. Exibir  
disp.plot(cmap='Blues')  
plt.title("Matriz de Confusão - Direção de Tendência")  
plt.xlabel("Predito")  
plt.ylabel("Real")  
plt.grid(False)  
plt.show()
```



```
# Acurácia de direção: % de vezes em que a predição acertou a direção do movimento
acertos = np.sum(direcao_real == direcao_pred)
total = len(direcao_real)
acuracia_direcao = acertos / total

print(f"✅ Acurácia de direção: {acuracia_direcao * 100:.2f}% ({acertos}/{total})")
```

✅ Acurácia de direção: 50.52% (146/289)

Na teoria, temos um modelo vencedor -> Accuracia > 50%

## 2. Gráfico com Setas de Tendência

As **tendências reais** e **preditas** em um mesmo gráfico. Usaremos setas verticais com cores:

- Azul para tendência **real**
- Vermelho para tendência **predita**

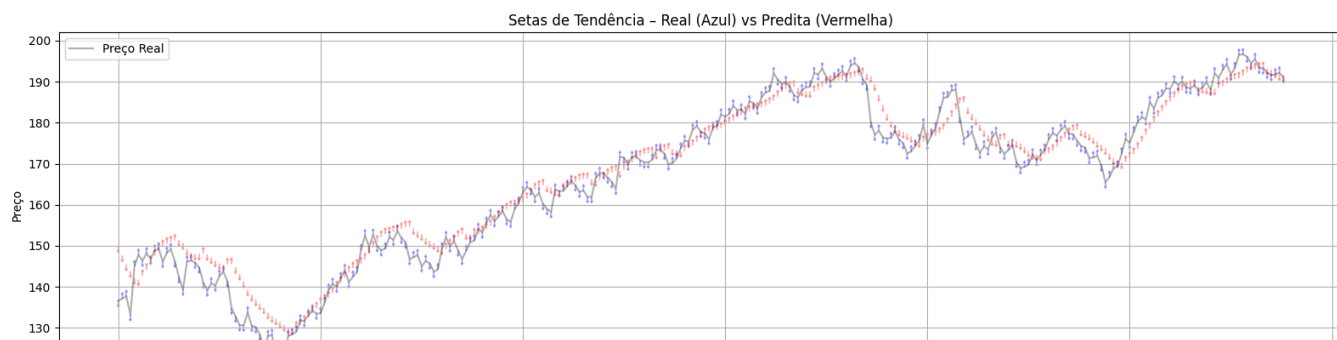
```
plt.figure(figsize=(16, 5))
dias = np.arange(len(direcao_real))

# Plot base da ação real
plt.plot(y_real_flat[1:], label='Preço Real', color='black', alpha=0.3)

# Setas de tendência real
for i, direcao in enumerate(direcao_real):
    cor = 'blue' if direcao == 1 else 'blue'
    dy = 1.0 if direcao == 1 else -1.0
    plt.arrow(i, y_real_flat[i+1], 0, dy, color=cor, head_width=0.5, head_length=0.5, alpha=0.3)

# Setas de tendência predita
for i, direcao in enumerate(direcao_pred):
    cor = 'red' if direcao == 1 else 'red'
    dy = 1.0 if direcao == 1 else -1.0
    plt.arrow(i, y_pred_flat[i+1], 0, dy, color=cor, head_width=0.5, head_length=0.5, alpha=0.3)

plt.title("Setas de Tendência – Real (Azul) vs Predita (Vermelha)")
plt.xlabel("Dias")
plt.ylabel("Preço")
plt.grid(True)
plt.legend(["Preço Real"])
plt.tight_layout()
plt.show()
```



6. Com o Deep Learning Generativos existem dois modelos que são:

### **Redes Neurais Adversárias:**

As Redes Neurais Adversárias (GAN - Generative Adversarial Network) são arquiteturas de redes neurais compostas por duas redes que são posicionadas uma contra a outra, o condiz com o conceito de redes adversárias. Ian Goodfellow e outros pesquisadores da Universidade de Montreal introduziram o conceito de GANs, que são redes neurais recentes na área de Deep Learning.

Uma das principais características dessas redes neurais é que elas podem reproduzir qualquer distribuição de dados durante o aprendizado, como geração de imagens, músicas, fala e prosa, ou seja, elas atuam, em certo sentido, no lado artístico.

O funcionamento das redes GANs funciona da seguinte forma: existem duas redes neurais, a rede geradora que gera nova instância de dados através de um vetor de números aleatórios e retorna elementos sintéticos que podem ser imagens, vozes ou música e a rede discriminadora, que recebe os elementos gerados pelo gerador realiza a autenticação desses elementos, comparando elas com os elementos retirados do conjunto de dados real.

Esse processo é iterativo, pois, à medida que o discriminador verifica que os elementos sintetizados são falsos e diferentes dos elementos reais, o gerador vai retornando elementos sintéticos cada vez mais próximos dos reais.

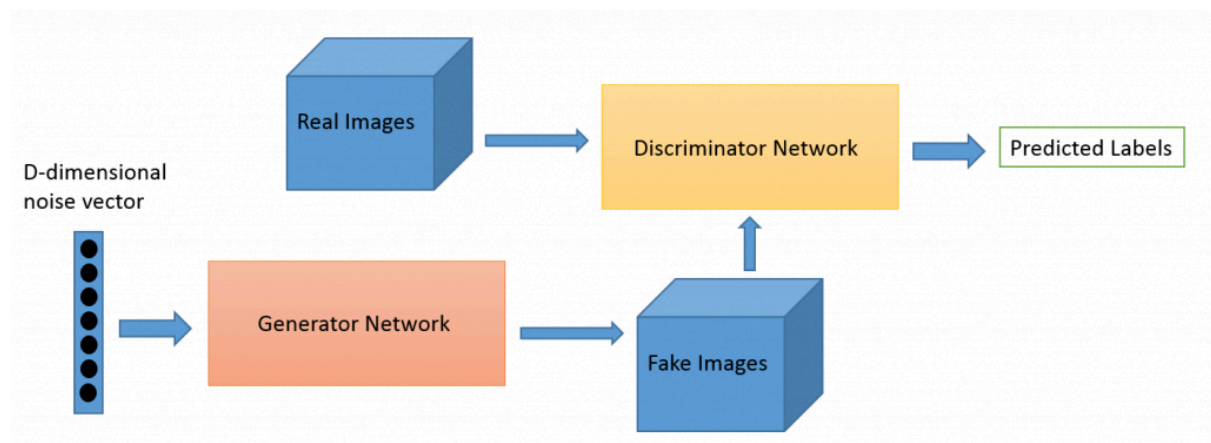


Figura 1: esquema mostrando o funcionamento das redes GAN.

Fonte: Data Science Academy.

As principais aplicações das redes GAN são: geração de vozes sintéticas de pessoas (dublagem via Inteligência artificial), geração de imagens, detecção de fraudes, preenchimento de informações faltantes e geração de modelos 3D a partir de dados 2D.

### **Autoencoders Variacionais:**



Em termos gerais, os autoencoders são técnicas de aprendizado não supervisionado para redes neurais com a tarefa de aprendizado de representação e a principal intenção dessas técnicas de aprendizado é copiar as suas entradas para suas saídas. Os autoencoders são compostos por encoders, que compactam a entrada em uma representação de espaço latente e os decoders que recebem a função de entrada compactada e os reconstroem.

As principais aplicações dos autoencoders são: remoção de ruídos, redução de dimensionalidade para visualização de dados e reconhecimento de imagens com Redes Neurais Convolucionais.

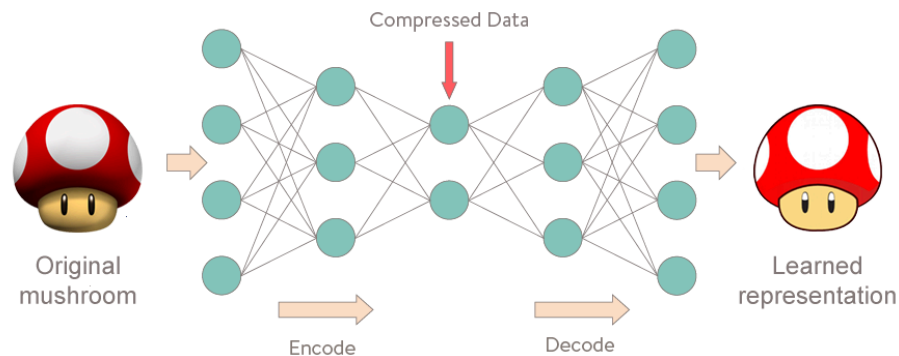


Figura 2: Esquemático apresentando o processo envolvendo Autoencoders.

Fonte: Data Science Academy.

Os autoencoders variacionais, por sua vez, possuem uma distribuição de codificações regularizada durante o treinamento com a intenção de garantir boas propriedades no seu espaço latente, permitindo assim gerar novos dados.

7.

As redes YOLO (*You Only Look Once* - Você Olha Apenas uma Vez) são redes neurais usadas para detectar objetos em uma determinada cena ou imagem. A principal característica desse tipo de rede neural é que ela diferencia das redes de detecção de objetos tradicionais pelo fato das redes YOLO serem de estágio único, enquanto as demais redes neurais são de dois estágios (no primeiro estágio as redes geram propostas de regiões de interesse e no segundo estágio essas regiões são classificadas). As redes YOLO realizam esses dois estágios simultaneamente.

A principal vantagem dessas redes neurais é que elas são bastante rápidas e eficientes, principalmente em aplicações em tempo real e o objetivo dessas redes de detecção de objetos em uma imagem é realizar a detecção de forma mais precisa possível.

O processo de detecção funciona da seguinte forma: em uma determinada imagem com alguns objetos visualizados, as redes retornam caixas delimitadoras e um mapa de probabilidades para cada célula que compõem a imagem. Um exemplo desse processo é apresentado na figura 3.

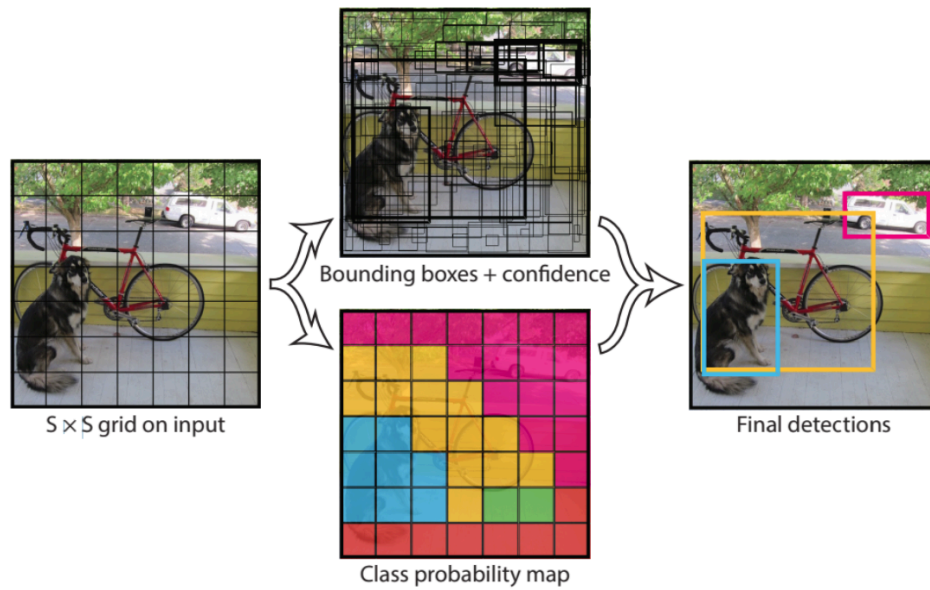


Figura 3: Imagem apresentando o processo de detecção de objetos em cena.  
Fonte: Redmon et. al (2016).

Sites pesquisados:

<https://aws.amazon.com/pt/what-is/gan/>

<https://www.deeplearningbook.com.br/introducao-as-redes-adversarias-generativas-gans-generative-adversarial-networks/>

<https://www.deeplearningbook.com.br/introducao-aos-autoencoders/>

<https://www.deeplearningbook.com.br/variational-autoencoders-vaes-definicao-reducao-de-dimensao-espaco-latente-e-regularizacao/>

<https://arxiv.org/pdf/1506.02640>