Feed-forward Artificial Neural Networks

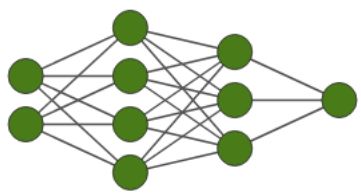
```
import numpy as np
import jax.numpy as jnp
import jax
import matplotlib.pyplot as plt
import time
```

Learning the XOR function

Let us consider the XOR ("exclusive or") function. The truth table of the XOR function is:

```
inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
outputs = np.array([[0], [1], [1], [0]])
```

We consider a fully connected ANN with the following architechture and with $\tanh(\cdot)$ as activation function:



The input of the network are the truth values of the two inputs (0 = False, 1 = True). The output is the **likelihood** (the "probability") of the truth value of the result. Hence, we want the output to be in $[0, 1]$. For this reason, we apply the activation function also to the last layer, and we rescale the output between 0 and 1.

The architecture is defined by the following **hyperparameters**:

```
n1, n2, n3, n4 = 2, 4, 3, 1
```

Define the parameters (weights and biases) of the ANN. Initialize the weights sampling from a standard normal distribution, and initialize to zero the biases.

Define both the weights and the biases as a 2-dimensional `jax` array (you can define them in `numpy` and then convert them).

```
# SOLUTION-BEGIN
np.random.seed(0)  # for reproducibility

W1 = np.random.randn(n2, n1)
b1 = np.zeros((n2, 1))
W2 = np.random.randn(n3, n2)
b2 = np.zeros((n3, 1))
W3 = np.random.randn(n4, n3)
b3 = np.zeros((n4, 1))
params = [W1, b1, W2, b2, W3, b3]
params = [jnp.array(p) for p in params]
# SOLUTION-END
```

Define a function that implements the ANN:

```
def ANN(x, params):
  ...
```

with inputs:

- `x`: num_samples by 2 `numpy` array
- `params`: list of the parameters and with output

- `y`: num_samples by 1 `numpy` array

Then, test the function on `inputs` and `params`.

```
# SOLUTION-BEGIN
def ANN(x, params):
    W1, b1, W2, b2, W3, b3 = params
    layer1 = x.T
    layer2 = jnp.tanh(W1 @ layer1 - b1)
    layer3 = jnp.tanh(W2 @ layer2 - b2)
    layer4 = jnp.tanh(W3 @ layer3 - b3)
    layer4 = (1.0 + layer4) / 2.0
    return layer4.T


ANN(inputs, params)
# SOLUTION-END
```

Print the truth table associated with the ANN.

```
for input in inputs:
  print(f"{input[0]} XOR {input[1]} --> {ANN(input.reshape((1, -1)), params)}")
```

Define a function that computes the quadratic loss on a set of data:

```
def loss_quadratic(x, y, params):
  ...
```

with inputs:

- `x`: num_samples by 2 `numpy` array
- `y`: num_samples by 1 `numpy` array
- `params`: list of the parameters and with output
- `loss`: scalar

Then, test the function on `inputs`, `outputs` and `params`.

```
# SOLUTION-BEGIN
def loss_quadratic(x, y, params):
    error = ANN(x, params) - y
    return jnp.sum(error * error)


loss_quadratic(inputs, outputs, params)
# SOLUTION-END
```

Define a function that computes the cross-entropy loss on a set of data:

$$-\sum_i y_i \log(y(x_i; \theta)) + (1 - y_i) \log(1 - y(x_i; \theta)) $$$$-\sum_i y_i \log(y(x_i; \theta)) + (1 - y_i) \log(1 - y(x_i; \theta)) $$

```
def loss_crossentropy(x, y, params):
  ...
```

with inputs:

- `x`: num_samples by 2 `numpy` array
- `y`: num_samples by 1 `numpy` array
- `params`: list of the parameters and with output
- `loss`: scalar

Then, test the function on `inputs`, `outputs` and `params`.

```
# SOLUTION-BEGIN
def loss_crossentropy(x, y, params):
    y_app = ANN(x, params)
    return -jnp.sum(y * jnp.log(y_app) + (1 - y) * jnp.log(1 - y_app))
```

```
loss_crossentropy(inputs, outputs, params)
# SOLUTION-END
```

Using the JIT module of JAX, compile both the losses defined above and their gradients w.r.t. the parameters.

```
# SOLUTION-BEGIN
loss_quad_jit = jax.jit(loss_quadratic)
grad_quad_jit = jax.jit(jax.grad(loss_quadratic, argnums=2))
loss_Xent_jit = jax.jit(loss_crossentropy)
grad_Xent_jit = jax.jit(jax.grad(loss_crossentropy, argnums=2))
# SOLUTION-END
```

Implement the GD (gradient descent) method, with learning rate 0.1 and train the ANN, using either the quadratic or the cross-entropy loss function. Collect the training history of both loss functions at each training epoch and plot them.

Finally, print the truth table associated with the ANN and compute its accuracy.

```
grad = grad_quad_jit
#grad = grad_Xent_jit
learning_rate = 1e-1

history_quad = list()
history_Xent = list()

# SOLUTION-BEGIN
t0 = time.time()
for epoch in range(2000):
    grads = grad(inputs, outputs, params)
    for i in range(len(params)):
        params[i] -= learning_rate * grads[i]
    history_quad.append(loss_quad_jit(inputs, outputs, params))
    history_Xent.append(loss_Xent_jit(inputs, outputs, params))

print(f"elapsed time: {time.time() - t0}")
plt.semilogy(history_quad, label="quadratic")
plt.semilogy(history_Xent, label="X-entropy")
plt.legend()
# SOLUTION-END
```

```
for input in inputs:
  print(f"{input[0]} XOR {input[1]} --> {ANN(input.reshape((1, -1)), params)}")
```

Compute the accuracy of the ANN

```
# SOLUTION-BEGIN
TP = ANN(np.array([[0, 1], [1, 0]]), params)
TN = 1 - ANN(np.array([[0, 0], [1, 1]]), params)
accuracy = (sum(TP) + sum(TN)) / 4
print(f"accuracy: {accuracy[0] * 100:1.2f}%")
# SOLUTION-END
```

Generalize the process

We test the ANN on a dataset made by a large circle containing a smaller circle in 2d. This is a simple toy dataset to visualize clustering and classification algorithms.

```
from sklearn.datasets import make_circles
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

X, y = make_circles(n_samples=300, factor=0.5, noise=0.1)
y = y.reshape(-1, 1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train.flatten())
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test.flatten(), marker="x")
```

Ingredients

- `init_layer_params(key, in_dim, out_dim)`: given a random key and the in/out shape of a layer, it initializes and return the weighs and biases of the layer
- `init_mlp_params(key, layer_sizes)`: given a random key and a list of layer sizes, it initializes and return the list of weighs and biases pairs of the layer
- `sigmoid(x)`: $1/(1+e^{-x})$
- `forward(params, x)`: the evaluation of the neural network with parameters `params` with input `x`
- `binary_cross_entropy(params, x, y)`: cross-entropy loss (as we did before)
- `update(params, x, y, lr)`: given the parameters `params` (list of pairs of weight, bias), update them using (stochastic) gradient descent using data `x` and `y` (assume they are a mini-batch) with learnig rate `lr`

```
# SOLUTION-BEGIN
def init_layer_params(W_key, in_dim, out_dim):
    W = jax.random.normal(W_key, (in_dim, out_dim))
    b = jnp.zeros((out_dim,))
    return W, b

def init_mlp_params(key, layer_sizes):
    keys = jax.random.split(key, len(layer_sizes) - 1)
    params = []
    for i in range(len(layer_sizes) - 1):
        params.append(init_layer_params(keys[i], layer_sizes[i], layer_sizes[i + 1]))
    return params

def sigmoid(x):
    return 1 / (1 + jnp.exp(-x))

def forward(params, x):
    for (W, b) in params[:-1]:
        x = jnp.tanh(x @ W + b)
    W, b = params[-1]
    logits = x @ W + b
    return sigmoid(logits)

def binary_cross_entropy(params, x, y):
    preds = forward(params, x)
    eps = 1e-7
    loss = -jnp.mean(y * jnp.log(preds + eps) + (1 - y) * jnp.log(1 - preds + eps))
    return loss

@jax.jit
def update(params, x, y, lr):
    grads = jax.grad(binary_cross_entropy)(params, x, y)
    new_params = jax.tree_util.tree_map(lambda p, g: p - lr * g, params, grads)
    return new_params
# SOLUTION-END
```

Training

Initialize a neural network with layers `[2, 16, 16, 1]` and train it with SDG

```
key = jax.random.PRNGKey(0)
layer_sizes = [2, 16, 16, 1]
params = init_mlp_params(key, layer_sizes)
learning_rate = 0.01
epochs = 5000
batch_size = 64

# SOLUTION-BEGIN
num_batches = X_train.shape[0] // batch_size
for epoch in range(epochs):
    perm_key, key = jax.random.split(key)
    perm = jax.random.permutation(perm_key, X_train.shape[0])
    X_shuffled = X_train[perm]
    y_shuffled = y_train[perm]

    for i in range(num_batches):
        start = i * batch_size
        end = start + batch_size
```

```
            xb, yb = X_shuffled[start:end], y_shuffled[start:end]
            params = update(params, xb, yb, learning_rate)

        if epoch % 100 == 0:
            loss = binary_cross_entropy(params, X_train, y_train)
            print(f"Epoch {epoch}, Loss: {loss:.4f}")
    # SOLUTION-END
```

Compute the accuracy and confusion matrix for the predictions of the neural network

```
from sklearn.metrics import confusion_matrix

# SOLUTION-BEGIN
y_pred = forward(params, X_test)
acc = jnp.mean((y_pred > 0.5) == y_test)
print(f"Test accuracy: {acc:.3f}")
print(confusion_matrix(y_test, (y_pred > 0.5)))
# SOLUTION-END
```

Visualize the results

```
import numpy as np
# Create a meshgrid over the input space
x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
xx, yy = np.meshgrid(np.linspace(x_min, x_max, 300),
                     np.linspace(y_min, y_max, 300))
grid = np.column_stack([xx.ravel(), yy.ravel()])

# Predict over the grid
Z = forward(params, grid)
Z = Z.reshape(xx.shape)

# Plot decision boundary and data points
plt.figure(figsize=(6, 6))
plt.contourf(xx, yy, Z, levels=[0, 0.5, 1], alpha=0.4, cmap='coolwarm')
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train.flatten(), cmap='coolwarm', s=20, edgecolor='k')
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test.flatten(), cmap='coolwarm', s=20, marker="x")
plt.title(f"Decision boundary (accuracy={acc:.3f})")
plt.xlabel("x1")
plt.ylabel("x2")
plt.show()
```

Support Vector Regression (SVR) Hands-on Exercise

In this exercise, we will implement and train a Support Vector Regression model using JAX and evaluate its performance on synthetic data. We will go through the steps of data generation, model training, and evaluation.

Step 1: Import Libraries

We will begin by importing the necessary libraries. JAX will be used for numerical computations, NumPy for array manipulations, and Matplotlib for visualization.

```
import jax
import jax.numpy as jnp
import numpy as np

from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Step 2: Define the SVR Class

We will create a class `SVR` to encapsulate the SVR model. The class will include methods for calculating the loss, training the model, and making predictions.

Loss Function

The loss function for SVR is defined as (soft-margin):

$$L(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \max(0, |f(x_i) - y_i| - \epsilon)$$

Where:

- w are the model parameters.
- λ is the regularization parameter.
- ϵ is the epsilon-insensitive loss threshold.
- $f(x_i)$ is the predicted value for input x_i.
- y_i is the actual target value.

Let's define the `SVR` class.

```
class SVR:
    def __init__(self, epsilon=0.1, lmbda=1.0):
        self.epsilon = epsilon
        self.lmbda = lmbda
        self.w = None

    def loss(self, params, X, y):
        # SOLUTION-BEGIN
        predictions = X.reshape((-1, self.n_features)) @ params[:-1] + params[-1]

        # Compute epsilon-insensitive loss
        epsilon_loss = jnp.maximum(0, jnp.abs(predictions - y) - self.epsilon)

        # Regularization term (L2 norm of w)
        reg_term = self.lmbda * jnp.sum(params**2)

        # Total loss
        return reg_term + jnp.mean(epsilon_loss)
        # SOLUTION-END

    def train(self, X, y, lr=1e-2, max_iter=1000):
        # SOLUTION-BEGIN
        _, self.n_features = X.shape

        # Initialize weights and bias
        self.w = jnp.zeros(self.n_features + 1)

        # Solve optimization problem
        grad_fn = jax.grad(self.loss, argnums=0)

        @jax.jit
        def step(w):
            return w - lr * grad_fn(w, X, y)

        for _ in range(max_iter):
            self.w = step(self.w)
        # SOLUTION-END

    def predict(self, X):
        # SOLUTION-BEGIN
        return X.reshape((-1, self.n_features)) @ self.w[:-1] + self.w[-1]
        # SOLUTION-END
```

Step 3: Generate Synthetic Data

Next, we will generate synthetic data for regression. We will create a linear relationship with added Gaussian noise.

- We will generate $n_samples$ data points.
- The true relationship will follow the equation $y = mx + c$.
- Gaussian noise will be added to simulate real-world data.

```
np.random.seed(0)  # For reproducibility
m = 2.5  # Slope
c = 1.0  # Intercept
n_samples = 100

# Generate X values uniformly distributed between 0 and 10
```

```
X = np.random.uniform(0, 10, size=(n_samples, 1))

# Create the line (y = mx + c) and add Gaussian noise
noise = np.random.normal(0, 1, size=(n_samples, 1))  # Gaussian noise
y = m * X + c + noise  # Add noise to the true line
y = y.flatten()
```

Step 4: Split the Data

Now, we will split the dataset into training and testing sets. This allows us to evaluate the model's performance on unseen data.

We will use an 80-20 split for training and testing.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

X_train = jnp.array(X_train)
X_test = jnp.array(X_test)
y_train = jnp.array(y_train)
y_test = jnp.array(y_test)
```

Step 5: Train the SVR Model

We will now create an instance of the `SVR` class and train the model using the training data.

```
svr = SVR(epsilon=1.0, lmbda=0.1)
svr.train(X_train, y_train)
```

Step 6: Make Predictions

After training, we will use the model to make predictions on both the training and testing datasets.

```
y_pred_train = svr.predict(X_train)
y_pred_test = svr.predict(X_test)
```

Step 7: Evaluate the Model

We will evaluate the model's performance using the Mean Squared Error (MSE), which is defined as:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Where:

- y_i is the actual value.
- $\hat{y}_i$ is the predicted value.
- n is the number of samples.

```
mse_train = jnp.mean((y_train - y_pred_train) ** 2)
mse_test = jnp.mean((y_test - y_pred_test) ** 2)
print(f"Train MSE: {mse_train:.4f}")
print(f"Test MSE: {mse_test:.4f}")
```

Step 8: Visualize the Results

Finally, we will visualize the training data, test data, and the model's predictions. This will help us understand how well our model has captured the underlying relationship.

```
plt.figure(figsize=(10, 6))

# Plot training data
plt.scatter(X_train, y_train, color="blue", label="Training data")
# Plot test data
plt.scatter(X_test, y_test, color="green", marker="x", label="Test data")
```

```
# Plot the prediction line (SVR model)
x_range = jnp.linspace(0, 10, 100)
y_pred_line = svr.predict(x_range)
plt.plot(x_range, y_pred_line, color="red", label="SVR prediction")
plt.fill_between(
    x_range,
    y_pred_line - svr.epsilon,
    y_pred_line + svr.epsilon,
    label="SVR tube",
    color="r",
    alpha=0.1,
)

plt.xlabel("X")
plt.ylabel("y")
plt.title("SVR on Synthetic Data with Gaussian Noise")
plt.legend()
plt.show()
```

Support Vector Machine (SVM)

We will change the previous code to implement a linear Support Vector Machine in primal formulation

Step 2: Define the SVM Class

We will create a class `SVM` to encapsulate the SVM model. The class will include methods for calculating the loss, training the model, and making predictions.

Loss Function

The loss function for SVM is defined as the hinge loss:

$$L(w) = \lambda \|w\|^2 + \frac{1}{n} \sum_{i=1}^{n} \max(0, 1 - y_i(w^T x_i + b))$$

Where:

- w are the model parameters (weights).
- b is the bias term.
- λ is the regularization parameter.
- y_i is the true label for the sample i.
- x_i is the feature vector for the sample i.
- The first term is the regularization term, and the second term is the hinge loss.

Let's define the `SVM` class.

```
class SVM:
    def __init__(self, lmbda=1.0):
        self.lmbda = lmbda
        self.w = None

    def loss(self, params, X, y):
        # SOLUTION-BEGIN
        # Compute the decision function
        print(X.shape, params.shape, y.shape)
        decision = X @ params[:-1] + params[-1]
        # Compute the hinge loss
        loss_val = jnp.maximum(0, 1 - y * decision)
        # Regularization term (L2 norm of w)
        reg_term = self.lmbda * jnp.sum(params**2)
        # Total loss
        return reg_term + jnp.mean(loss_val)
        # SOLUTION-END

    def train(self, X, y, lr=1e-2, max_iter=1000):
        # SOLUTION-BEGIN
        _, self.n_features = X.shape

        # Initialize weights and bias
```

```
        self.w = jnp.zeros(self.n_features + 1)

        # Solve optimization problem
        grad_fn = jax.grad(self.loss, argnums=0)

        @jax.jit
        def step(w):
            return w - lr * grad_fn(w, X, y)

        for _ in range(max_iter):
            self.w = step(self.w)
        # SOLUTION-END

    def predict(self, X):
        # SOLUTION-BEGIN
        # Decision function
        decision = X @ self.w[:-1] + self.w[-1]
        return jnp.sign(decision)
        # SOLUTION-END
```

Step 3: Generate Synthetic Classification Data

Next, we will generate synthetic data for classification. We will create a dataset of points in a 2D space and label them based on their coordinates.

- We will generate $n_{samples}$ data points.
- The labels will be determined by the condition $x_1 + x_2 > 10$ to classify points into two categories.

```
np.random.seed(42)  # For reproducibility
n_samples = 100

# Generate X values uniformly distributed between [0, 10]^2
X = np.random.uniform(0, 10, size=(n_samples, 2))

# Generate binary labels
y = np.where(X.sum(axis=1) > 10, 1, -1)
y = y.flatten()
```

Step 4: Split the Data

Now, we will split the dataset into training and testing sets. This allows us to evaluate the model's performance on unseen data.

We will use an 80-20 split for training and testing.

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

X_train = jnp.array(X_train)
X_test = jnp.array(X_test)
y_train = jnp.array(y_train)
y_test = jnp.array(y_test)
```

Step 5: Train the SVM Model

We will now create an instance of the `SVM` class and train the model using the training data.

```
svm = SVM(lmbda=0.001)
svm.train(X_train, y_train, lr=1e-1, max_iter=5000)
print("Loss: ", svm.loss(svm.w, X_train, y_train))
print(svm.w)
```

Step 6: Make Predictions

After training, we will use the model to make predictions on both the training and testing datasets.

```
y_pred_train = svm.predict(X_train)
y_pred_test = svm.predict(X_test)
```

Step 7: Evaluate the Model

We will evaluate the model's performance using accuracy, which is defined as the proportion of correct predictions:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

Let's calculate and print the accuracy for both the training and test sets.

```
accuracy_train = jnp.mean(y_pred_train == y_train)
accuracy_test = jnp.mean(y_pred_test == y_test)
print(f"Train Accuracy: {accuracy_train:.4f}")
print(f"Test Accuracy: {accuracy_test:.4f}")
```

Step 8: Visualize the Results

Finally, we will visualize the training data, test data, and the decision boundary of the SVM model. This will help us understand how well our model has separated the classes.

```
plt.figure(figsize=(10, 6))
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, label="Training data", marker="o")
plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, label="Test data", marker="x")

# Plot the decision boundary
t = jnp.linspace(0, 10, 1000)
xx1, xx2 = jnp.meshgrid(t, t)
xx = jnp.stack([xx1.flatten(), xx2.flatten()], axis=1)
yy = svm.predict(xx)
plt.contourf(xx1, xx2, yy.reshape(xx1.shape), alpha=0.1)
plt.xlabel("X1")
plt.ylabel("X2")
plt.title("SVM Decision Boundary on Synthetic Data")
plt.legend(loc="upper right")
plt.show()
```

Optimization with JAX: Gradient Descent and Backtracking

Laboratory Session

In this laboratory session, we will explore optimization techniques implemented with **JAX**. We will cover several optimization algorithms:

- Gradient Descent (GD)
- Gradient Descent with Backtracking Line Search
- Exact Line Search for Quadratic Functions

We will apply these methods on various benchmark functions:

- Rastrigin Function
- Ackley Function
- Quadratic Function

Let's begin by importing the rebuired packages.

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt

# Set random seed for reproducibility
key = jax.random.PRNGKey(0)
```

Helper Functions for Plotting

We define a helper function `plot_optimization_2d` to visualize optimization trajectories on a 2D contour plot. This function will:

- Plot the optimization path taken by Gradient Descent (GD)
- Plot the optimization path with Backtracking Line Search if available
- Show the function's value as a contour plot in the background.

```
def plot_optimization_2d(func, gd_path, gd_backtrack_path, title):
    x_vals = jnp.linspace(-5, 5, 50)
    y_vals = jnp.linspace(-5, 5, 50)
    X, Y = jnp.meshgrid(x_vals, y_vals)
    Z = jnp.array([[func(jnp.array([x, y])) for x in x_vals] for y in y_vals])

    fig, axs = plt.subplots(1, 2, figsize=(14, 6))
    cs = axs[0].contourf(X, Y, Z, levels=50, cmap="viridis")
    plt.colorbar(cs)
    axs[0].contour(X, Y, Z, colors="white")

    gd_path = jnp.array(gd_path)

    axs[0].plot(gd_path[:, 0], gd_path[:, 1], "r.-", label="GD")

    if gd_backtrack_path != []:
        gd_backtrack_path = jnp.array(gd_backtrack_path)
        axs[0].plot(
            gd_backtrack_path[:, 0],
            gd_backtrack_path[:, 1],
            ".-",
            color="orange",
            label="GD + backtracking",
        )
    axs[0].set_title(title)
    axs[0].set_xlabel("x")
    axs[0].set_ylabel("y")
    axs[0].set_ylim([-5, 5])
    axs[0].set_xlim([-5, 5])
    axs[0].legend()

    axs[1].semilogy([func(x) for x in gd_path], "ro-", label="GD")
    axs[1].semilogy(
        [func(x) for x in gd_backtrack_path],
        "o-",
        color="orange",
        label="GD + backtracking",
    )
    axs[1].legend()
    plt.tight_layout()
```

Defining Benchmark Functions

1. Rastrigin Function

The Rastrigin function is a **non-convex** and **multi-modal** function often used as a test for optimization algorithms. It is defined as:

$$f(x) = 10n + \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i)\right)$$

with multiple local minima.

2. Ackley Function

The Ackley function is another **non-convex** function commonly used in optimization, given by:

$$f(x) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e$$

3. Quadratic Function

The Quadratic function is **convex** and is defined as:

$$f(x) = \frac{1}{2}x^T A x + b^T x + c$$

```
# Define the benchmark functions
```

```
# 1. Rastrigin function
@jax.jit
def rastrigin(x):
    return 10 * x.size + jnp.sum(x**2 - 10 * jnp.cos(2 * jnp.pi * x)) + 1e-10


# 2. Ackley function
@jax.jit
def ackley(x):
    a = 20
    b = 0.2
    c = 2 * jnp.pi
    sum1 = jnp.sum(x**2)
    sum2 = jnp.sum(jnp.cos(c * x))
    return (
        -a * jnp.exp(-b * jnp.sqrt(sum1 / x.size))
        - jnp.exp(sum2 / x.size)
        + a
        + jnp.exp(1)
    )


# 3. Quadratic function
quadratic_A = jnp.array([[3.0, 0.5], [0.5, 1.0]])
quadratic_b = jnp.array([-1.0, 2.0])
quadratic_c = jnp.dot(quadratic_b, jnp.linalg.solve(quadratic_A, quadratic_b)) / 2


@jax.jit
def quadratic(x):
    return (
        0.5 * jnp.dot(x.T, jnp.dot(quadratic_A, x))
        + jnp.dot(quadratic_b, x)
        + quadratic_c
    )
```

Optimization Algorithms

Gradient Descent

Gradient Descent is an iterative optimization algorithm defined by:

$$x_{k+1} = x_k - \eta \nabla f(x_k)$$

where η is the learning rate.

```
def gradient_descent(grad_func, x0, lr=0.01, tol=1e-6, max_iter=1000):
    # SOLUTION-BEGIN
    x = x0
    path = [x]  # for visualization purposes only
    for _ in range(max_iter):
        grad_val = grad_func(x)
        x = x - lr * grad_val
        path.append(x)
        if jnp.linalg.norm(grad_val) < tol:
            break
    # SOLUTION-END
    return x, path
```

Gradient Descent with Backtracking Line Search

Backtracking line search modifies the learning rate η to ensure sufficient decrease.

The key idea is to start with an initial step size and iteratively reduce it until a sufficient decrease condition is met. Specifically, the backtracking process adjusts the step size tt based on the Armijo condition:

$$f(x - t\nabla f(x)) \leq f(x) - \alpha t \|\nabla f(x)\|^2$$

where:

- $f(x)$ is the objective function,
- $\nabla f(x)$ is the gradient of f at x,

- t is the step size (or learning rate) that we are adapting,
- $\alpha \in (0, 0.5)$ is a parameter controlling the required decrease,
- $\|\nabla f(x)\|^2$ represents the squared norm of the gradient, quantifying how steep the descent is at x.

The algorithm typically starts with an initial step size $t = 1$ and iteratively reduces it by multiplying it with a factor $\beta \in (0, 1)$ (often $\beta = 0.8$), until the sufficient decrease condition is satisfied. The process can be summarized as follows:

1. **Initialize** $t = 1$.
2. **While** $f(x - t\nabla f(x)) > f(x) - \alpha t\|\nabla f(x)\|^2$:
   - Update $t = \beta t$.
3. **Return** the adjusted step size t.

```
def gradient_descent_backtracking(
    func, grad_func, x0, alpha=0.3, beta=0.8, tol=1e-6, max_iter=100
):
    # SOLUTION-BEGIN
    x = x0
    path = [x]  # for visualization purposes only
    for _ in range(max_iter):
        grad_val = grad_func(x)
        t = 1.0
        # Backtracking loop
        while func(x - t * grad_val) > func(x) - alpha * t * jnp.dot(
            grad_val, grad_val
        ):
            t *= beta
        x = x - t * grad_val
        path.append(x)
        if jnp.linalg.norm(grad_val) < tol:
            break
    # SOLUTION-END
    return x, path
```

Exact Line Search for Quadratic Function

Exact line search minimizes the quadratic function

$$f(\boldsymbol{x}) = \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} + \boldsymbol{b}^T\boldsymbol{x} + c$$

in each iteration by solving for the optimal step size t^* as:

$$t^* = \frac{\nabla f(\boldsymbol{x})^T \nabla f(\boldsymbol{x})}{\nabla f(\boldsymbol{x})^T A \nabla f(\boldsymbol{x})}$$

Indeed, at iteration k, we want to minimize the (univariate) function g

$$\begin{aligned} g(t) = f\big(\boldsymbol{x}^k + t\boldsymbol{s}^k\big) &= \frac{1}{2}\big(\boldsymbol{x}^k + t\boldsymbol{s}^k\big)^T A\big(\boldsymbol{x}^k + t\boldsymbol{s}^k\big) + \boldsymbol{b}^T\big(\boldsymbol{x}^k + t\boldsymbol{s}^k\big) + c \\ &= \frac{1}{2}\big(\boldsymbol{x}^k\big)^T A\boldsymbol{x}^k + \frac{1}{2}t^2\big(\boldsymbol{s}^k\big)^T A\boldsymbol{s}^k + t\big(\boldsymbol{s}^k\big)^T A\boldsymbol{x}^k + \boldsymbol{b}^T\boldsymbol{x}^k + t\boldsymbol{b}^T\boldsymbol{s}^k + c \\ &= f\big(\boldsymbol{x}^k\big) + \frac{1}{2}t^2\big(\boldsymbol{s}^k\big)^T A\boldsymbol{s}^k + t\big(\boldsymbol{s}^k\big)^T A\boldsymbol{x}^k + t\big(\boldsymbol{s}^k\big)^T\boldsymbol{b} \\ &= f\big(\boldsymbol{x}^k\big) + \frac{1}{2}t^2\big(\boldsymbol{s}^k\big)^T A\boldsymbol{s}^k + t\big(\boldsymbol{s}^k\big)^T\Big[A\boldsymbol{x}^k + \boldsymbol{b}\Big] \end{aligned}$$

where $\boldsymbol{s}^k = -\nabla f(\boldsymbol{x}^k)$. Performing the derivative, we have

$$g'(t) = t\big(\boldsymbol{s}^k\big)^T A\boldsymbol{s}^k + \big(\boldsymbol{s}^k\big)^T\Big[A\boldsymbol{x}^k + \boldsymbol{b}\Big]$$

But, by definition of f

$$\nabla f(\boldsymbol{x}^k) = A\boldsymbol{x}^k + \boldsymbol{b}$$

Thus,

$$g'(t) = t\big(\boldsymbol{s}^k\big)^T A\boldsymbol{s}^k + \big(\boldsymbol{s}^k\big)^T \nabla f(\boldsymbol{x}^k)$$

By setting $g'(t^*) = 0$ we have

$$t^* = \frac{\nabla f(\boldsymbol{x}^k)^T \nabla f(\boldsymbol{x}^k)}{\nabla f(\boldsymbol{x}^k)^T A \nabla f(\boldsymbol{x}^k)}$$

```
def exact_line_search_quadratic(A, b, x0, tol=1e-6, max_iter=100):
    # SOLUTION-BEGIN
    x = x0
    path = [x]  # for visualization purposes only
    for _ in range(max_iter):
        grad_val = jnp.dot(A, x) + b
        t = jnp.dot(grad_val, grad_val) / jnp.dot(grad_val, jnp.dot(A, grad_val))
        x -= t * grad_val
        path.append(x)
        if jnp.linalg.norm(grad_val) < tol:
            break
    # SOLUTION-END
    return x, path
```

Testing Optimization Algorithms

Let's test our gradient descent methods on the benchmark functions, and visualize the results.

```
x0 = jnp.array([4.0, 4.0])

test_functions = [
    (rastrigin, "Rastrigin"),
    (ackley, "Ackley"),
    (quadratic, "Quadratic"),
]

for func, name in test_functions:
    grad_func = jax.jit(jax.grad(func))
    print(f"Testing on {name} function")

    # Vanilla Gradient Descent
    x_gd, path_gd = gradient_descent(grad_func, x0)
    # Gradient descent with backtracking
    x_gd_bt, path_gd_bt = gradient_descent_backtracking(func, grad_func, x0)

    plot_optimization_2d(func, path_gd, path_gd_bt, title=f"{name}")
```

Exact Line Search on Quadratic Function

Using exact line search, we aim to find the optimal path for the **Quadratic function**. This method is specific to quadratic functions and will only be applied to this case.

```
x_exact, path_exact = exact_line_search_quadratic(quadratic_A, quadratic_b, x0)
plot_optimization_2d(
    quadratic, path_exact, [], title="Quadratic Function - Exact Line Search"
)
```

Linear Regression using Stochastic Gradient Descent (SDG)

In this laboratory session, we will implement a simple linear regression model using stochastic gradient descent (SGD) to minimize the mean squared error (MSE). The dataset is synthetic, and we will use JAX for efficient computation and automatic differentiation.

Problem Setup

Given a set of data points (x_i, y_i), the goal is to learn the parameters θ_0 and θ_1 such that the model predicts $y = \theta_0 + \theta_1 x$. We will optimize these parameters using SGD to minimize the MSE loss.

Generate Synthetic Data

We will generate synthetic data where $y = 1.5x + 3$, with some added Gaussian noise. This data will be used to test our linear regression model.

Let us now generate the synthetic dataset using JAX's random number generation functions.

```
import numpy as np
np.random.seed(0)
```

```
N = 200
x_data = np.random.uniform(size=(N,)) * 10
y_data = 1.5 * x_data + 3 + np.random.normal(size=(N,))

# Show a quick preview of the generated data
plt.scatter(x_data, y_data)
```

Split into train and validation datasets

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    x_data, y_data, test_size=0.2, random_state=42
)

X_train = jnp.array(X_train)
X_test = jnp.array(X_test)
y_train = jnp.array(y_train)
y_test = jnp.array(y_test)
```

Linear Regression Model

The linear regression model is defined as:

$$y = \theta_0 + \theta_1 x$$

Where θ_0 is the intercept and θ_1 is the slope. We will define this model and a loss function to optimize it.

```
# Define the linear regression model
@jax.jit
def model(theta, x):
    # SOLUTION-BEGIN
    return theta[0] + theta[1] * x
    # SOLUTION-END
```

Mean Squared Error (MSE) Loss Function

The loss function we will use is the Mean Squared Error (MSE) between the predicted values and the true values:

$$ MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 $$$$ MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 $$

We will also use JAX's `jit` decorator to optimize this function for faster computation.

```
# Mean Squared Error loss function
@jax.jit
def mse_loss(theta, x, y):
    # SOLUTION-BEGIN
    y_pred = model(theta, x)
    return jnp.mean((y_pred - y) ** 2)
    # SOLUTION-END
```

Gradient of the Loss Function

We can compute the gradient of the MSE loss function with respect to the parameters $ \theta_0 $$ \theta_0 $ and $ \theta_1 $$ \theta_1 $ using JAX's automatic differentiation. This will allow us to update the parameters using gradient descent.

```
# Compute the gradient of the MSE loss function with JAX
# SOLUTION-BEGIN
grad_mse_loss = jax.jit(jax.grad(mse_loss))
# SOLUTION-END
```

Stochastic Gradient Descent (SGD) Update Step

In Stochastic Gradient Descent (SGD), we update the parameters using the gradient computed on a random mini-batch of the data. This allows for faster convergence compared to using the full dataset at each iteration.

We will define a function to perform a single update step of SGD.

```
# Stochastic Gradient Descent (SGD) update step, return the updated theta
@jax.jit
def sgd_update(theta, x_batch, y_batch, learning_rate):
    # SOLUTION-BEGIN
    grads = grad_mse_loss(theta, x_batch, y_batch)
    return theta - learning_rate * grads
    # SOLUTION-END
```

Stochastic Gradient Descent with Mini-batches

We will implement the full Stochastic Gradient Descent (SGD) algorithm, processing the data in mini-batches. At each epoch, we shuffle the data and update the parameters using the mini-batch gradients.

Let's now define the `stochastic_gradient_descent` function.

Fill with the following steps:

- compute and apply a permutation of the training data
- loop on the batches (a slice of size `batch_size` of the training data) and apply the SDG upate

```
# Stochastic Gradient Descent with mini-batches
def stochastic_gradient_descent(
    theta,
    training_input,
    training_labels,
    validation_input,
    validation_labels,
    learning_rate=0.01,
    epochs=100,
    batch_size=10,
    state=0,
):
    key = jax.random.PRNGKey(state)
    # Iterate over the number of epochs
    for epoch in range(epochs):
        key, subkey = jax.random.split(key)
        # SOLUTION-BEGIN
        # Shuffle data indices using JAX's random key
        perm = jax.random.permutation(subkey, len(training_input))
        # Process data in mini-batches
        for i in range(0, len(training_input), batch_size):
            batch_idx = perm[i : i + batch_size]
            x_batch = training_input[batch_idx]
            y_batch = training_labels[batch_idx]
            # Perform SGD update
            theta = sgd_update(theta, x_batch, y_batch, learning_rate)
        # Print the loss every 10 epochs
        if epoch % 10 == 0:
            loss = mse_loss(theta, validation_input, validation_labels)
            print(f"Epoch {epoch}, Loss: {loss:.4f} (test)")
        # SOLUTION-END
    return theta
```

Optimized Parameters

After running the optimization, we will obtain the optimized values of $ \theta_0 $$ \theta_0 $ and $ \theta_1 $$ \theta_1 $. These parameters define the best-fit line for the data.

Let's print the optimized parameters.

```
# Run the Stochastic Gradient Descent
# Initial guess for theta_0 (intercept) and theta_1 (slope)
theta = jnp.array([0.0, 0.0])
theta_opt = stochastic_gradient_descent(
    theta, X_train, y_train, X_test, y_test, learning_rate=0.01, epochs=100, batch_size=10
)
print(
    f"Optimized parameters: theta = [{theta_opt[0]:.2f}, {theta_opt[1]:.2f}]"
)
```

Visualizing the Results

Finally, we can plot the original data points and the regression line defined by the optimized parameters.

```
# SOLUTION-BEGIN
# After the training loop, use the optimized parameters to plot the regression line
# Generate predictions using the optimized parameters
y_pred = model(theta_opt, x_data)

# Plot the original data points
plt.scatter(X_train, y_train, label="Training points")
plt.scatter(X_test, y_test, label="Test points")

# Plot the regression line
plt.plot(x_data, y_pred, label="Fitted line", color="black", linewidth=2)

# Add labels and a legend
plt.xlabel("X")
plt.ylabel("Y")
plt.title("Linear Regression using Stochastic Gradient Descent (SDG)")
plt.legend()
plt.show()
# SOLUTION-END
```

Matrix completion and recommender systems

MovieLens data sets were collected by the GroupLens Research Project at the University of Minnesota.

This data set consists of:

- 100000 ratings (1-5) from 943 users on 1682 movies.
- Each user has rated at least 20 movies.

The `movielens.csv` file contains the full dataset. Users and items are numbered consecutively from 1. The data is randomly ordered. This is a tab separated list of

```
user id | item id | rating | timestamp
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.sparse import csr_matrix
from scipy.stats import pearsonr
```

Read the dataset from the `movielens.csv` file.

```
dataset = pd.read_csv(
    "./movielens.csv",
    sep="\t",
    header=None,
    names=["user_id", "item_id", "rating", "timestamp"],
)
dataset
```

How many movies? How many people? How many ratings?

```
n_people = np.unique(dataset.user_id).size
n_movies = np.unique(dataset.item_id).size
n_ratings = len(dataset)

print(f"{n_people} people")
print(f"{n_movies} movies")
print(f"{n_ratings} ratings")
```

Shuffle the data (see the function `np.random.shuffle`).

```
np.random.seed(1)  # for reproducibility

idxs = np.arange(n_ratings)
np.random.shuffle(idxs)
rows_dupes = dataset.user_id[idxs]
cols_dupes = dataset.item_id[idxs]
vals = dataset.rating[idxs]
```

Remove unused `user_id` or `movie_id` from `rows` and `cols`. E.g., if we have a dataset like

```
user id | item id | rating
0       |      1  |     4
0       |      2  |     5
3       |      2  |     5
```

Then we would have two empty rows in the matrix (for user 1 and 2) and one empty column (for movie 0).

```
A = [
  [0 4 5],
  [0 0 0],
  [0 0 0],
  [0 ? 5],
]
```

Thus, we want to remove empty intervals.

```
_, rows = np.unique(rows_dupes, return_inverse=True)
_, cols = np.unique(cols_dupes, return_inverse=True)

print(rows.min(), rows.max(), n_people)
print(cols.min(), cols.max(), n_movies)
```

Split the dataset into a subset of 80000 training ratings and 20000 testing ratings.

```
training_data = int(0.8 * n_ratings) # use 80% of the data as training

rows_train = rows[:training_data]
cols_train = cols[:training_data]
vals_train = vals[:training_data]
rows_test = rows[training_data:]
cols_test = cols[training_data:]
vals_test = vals[training_data:]
```

Let us denote by $\Omega$$\Omega$ the set of pairs $(i,j)$$(i,j)$ such that rating of the $i$$i$-th user on the $j$$j$-th movie is available in the training set (similarly, $\Omega_{\text{test}}$$\Omega_{\text{test}}$ is the set of testing pairs). Let us denote by $r_{ij}$$r_{ij}$ the corresponding rating.

Create a full matrix $X \in \mathbb{R}^{n \times p}$$X \in \mathbb{R}^{n \times p}$, such that:

$$ X_{i,j} = \begin{cases} r_{ij} & \text{if } (i,j) \in \Omega\\ 0& \text{otherwise} \end{cases} $$$$ X_{i,j} = \begin{cases} r_{ij} & \text{if } (i,j) \in \Omega\\ 0& \text{otherwise} \end{cases} $$

```
X_sparse = csr_matrix((vals_train, (rows_train, cols_train)), shape=(n_people, n_movies))
X_full = X_sparse.toarray()
```

Trivial recommender system

Create a trivial recommender system, based on the average rating of each user:

$$ r^{\text{pred}}_{ij} = \frac{1}{N_i} \sum_{j : (i,j) \in \Omega} r_{ij} $$$$ r^{\text{pred}}_{ij} = \frac{1}{N_i} \sum_{j : (i,j) \in \Omega} r_{ij} $$

where $N_i = card(j : (i,j) \in \Omega)$$N_i = card(j : (i,j) \in \Omega)$.

Then compute the RMSE (root mean square error):

$$ \text{RMSE} = \sqrt{\frac{1}{card(\Omega_{\text{test}})} \sum_{(i,j) \in \Omega_{\text{test}}} (r_{ij} - r^{\text{pred}}_{ij})^2} $$$$ \text{RMSE} = \sqrt{\frac{1}{card(\Omega_{\text{test}})} \sum_{(i,j) \in \Omega_{\text{test}}} (r_{ij} - r^{\text{pred}}_{ij})^2} $$

and the Pearson correlation coefficient $\rho$$\rho$ (use the function scipy.stats.pearsonr):

$$ \rho = \frac { \displaystyle\sum_{(i,j) \in \Omega_{\text{test}}} (r_{ij} - \overline{r}) (r^{\text{pred}}_{ij} - \overline{r}^{\text{pred}}) } {\sqrt{ \displaystyle\sum_{(i,j) \in \Omega_{\text{test}}} (r_{ij} - \overline{r})^2 } \sqrt{ \displaystyle\sum_{(i,j) \in \Omega_{\text{test}}} (r^{\text{pred}}_{ij} - \overline{r}^{\text{pred}})^2 }} $$$$ \rho = \frac { \displaystyle\sum_{(i,j) \in \Omega_{\text{test}}} (r_{ij} - \overline{r}) (r^{\text{pred}}_{ij} - \overline{r}^{\text{pred}}) } {\sqrt{ \displaystyle\sum_{(i,j) \in \Omega_{\text{test}}} (r_{ij} - \overline{r})^2 } \sqrt{ \displaystyle\sum_{(i,j) \in \Omega_{\text{test}}} (r^{\text{pred}}_{ij} - \overline{r}^{\text{pred}})^2 }} $$

where

$$ \begin{split} \overline{r} &= \frac{1}{card(\Omega_{\text{test}})} \sum_{(i,j) \in \Omega_{\text{test}}} r_{ij} \\ \overline{r}^{\text{pred}} &= \frac{1}{card(\Omega_{\text{test}})} \sum_{(i,j) \in \Omega_{\text{test}}} r^{\text{pred}}_{ij} \end{split} $$$$ \begin{split} \overline{r} &= \frac{1}{card(\Omega_{\text{test}})} \sum_{(i,j) \in \Omega_{\text{test}}} r_{ij} \\ \overline{r}^{\text{pred}} &= \frac{1}{card(\Omega_{\text{test}})} \sum_{(i,j) \in \Omega_{\text{test}}} r^{\text{pred}}_{ij} \end{split} $$

```
# compute trivial predictor on the training dataset
# SOLUTION-BEGIN
avg_ratings = np.empty((n_people,))
for user_id in range(n_people):
  avg_ratings[user_id] = np.mean(vals_train[rows_train == user_id])

# The vectorized version
# sum_ratings = np.bincount(rows_train, weights=vals_train, minlength=n_people)
# count_ratings = np.bincount(rows_train, minlength=n_people)
# avg_ratings = sum_ratings / count_ratings

# use the trivial predictor on the test dataset
vals_trivial = avg_ratings[rows_test]
# SOLUTION-END

errors_trivial = vals_test - vals_trivial

# print the metrics
RMSE_trivial = np.sqrt(np.mean(errors_trivial**2))
rho_trivial = pearsonr(vals_test, vals_trivial)[0]
print(f"RMSE: {RMSE_trivial:1.3f}")
print(f"rho : {rho_trivial:1.3f}")
```

Singular value truncation (SVT) based recommender system

Implement the SVT algorithm to predict the ratings of the testing set. Set a maximum number of iterations equal to 100. Print the RMSE and $\rho$$\rho$ at each iteration.

Try to calibrate the threshold to get better results.

Sketch of the SVT steps:

1. $U \Sigma V^T = A$$U \Sigma V^T = A$
2. $\displaystyle A = \sum_{k : \sigma_k < \texttt{threshold}} \sigma_k u_k v_k$$\displaystyle A = \sum_{k : \sigma_k < \texttt{threshold}} \sigma_k u_k v_k$
3. $A_{ij} = X_{ij}$$A_{ij} = X_{ij}$ for all $i, j$$i, j$ in the training dataset
4. Compute the difference w.r.t. the $A$$A$ at the previous step
5. Compute and save the test metrics

```
n_max_iter = 100
threshold = 100.0
increment_tol = 1e-6

RMSE_list = list()
rho_list = list()

A = X_full.copy()

print("Iter | Increment |  RMSE |  Corr ")
for i in range(n_max_iter):
    # SOLUTION-BEGIN
    A_old = A.copy()
    # HOMEWORK: what happens if you use rSVD here? which is a good k?
    U, s, VT = np.linalg.svd(A, full_matrices=False)
```

```
    # HOMEWORK: can you do this in a quicker way?
    # HINT: we are wasting time summing zeros
    s[s < threshold] = 0
    A = U @ np.diag(s) @ VT

    A[rows_train, cols_train] = vals_train
    increment = np.linalg.norm(A - A_old)

    vals_predicted = A[rows_test, cols_test]
    errors = vals_test - vals_predicted

    RMSE_list.append(np.sqrt(np.mean(errors**2)))
    rho_list.append(pearsonr(vals_test, vals_predicted)[0])
    # SOLUTION-END

    print(f"{i+1:04} | {increment:.3e} | {RMSE_list[-1]:1.3f} | {rho_list[-1]:1.3f}")
    if increment < increment_tol:
        break
```

Plot the history of RMSE and $\rho$$\rho$ and confront it with the trivial predictor

```
# SOLUTION-BEGIN
fig, axs = plt.subplots(2, 1, figsize=(12, 16))

axs[0].plot(RMSE_list, "o-")
axs[0].axhline(RMSE_trivial, color="red")
axs[0].legend(["RMSE", "RMSE trivial"])

axs[1].plot(rho_list, "o-")
axs[1].axhline(rho_trivial, color="red")
axs[1].legend([r"$\rho$", r"$\rho$ trivial"])
# SOLUTION-END
```

Moore-Penrose pseudo inverse

```
import numpy as np
import scipy.linalg as la
import matplotlib.pyplot as plt
```

Write a function computing the Moore-Penrose pseudo inverse, exploiting the full SVD.

$$A^\dagger = V \Sigma^{-1} U^T, \quad A = U \Sigma V^T$$$$A^\dagger = V \Sigma^{-1} U^T, \quad A = U \Sigma V^T$$

```
def my_pinv_fullSVD(A):
    # SOLUTION-BEGIN
    U, s, VT = np.linalg.svd(A)
    s[s > 0] = 1 / s[s > 0]
    Pinv = VT.transpose() @ la.diagsvd(s, A.shape[1], A.shape[0]) @ U.transpose()
    # SOLUTION-END
    return Pinv
```

Write now a function computing the Moore-Penrose pseudo inverse, exploiting the reduced SVD.

```
def my_pinv_thinSVD(A):
    # SOLUTION-BEGIN
    U, s, VT = np.linalg.svd(A, full_matrices=False)
    s[s > 0] = 1 / s[s > 0]
    Pinv =  (VT.transpose() * s) @ U.transpose()
    # SOLUTION-END
    return Pinv
```

Generate a random matrix $A$$A$ (with elements sampled from a standard Gaussian distribution) with 5 rows and 4 columns. Compute its Moore-Penrose pseudo inverse thorugh the two functions above defined, and compare the result with the function `numpy.linalg.pinv` (see Documentation).

```
A = np.random.randn(5, 4)
Apinv_numpy = np.linalg.pinv(A)
Apinv_fullSVD = my_pinv_fullSVD(A)
Apinv_thinSVD = my_pinv_thinSVD(A)
print(np.linalg.norm(Apinv_numpy - Apinv_fullSVD) / np.linalg.norm(Apinv_numpy))
print(np.linalg.norm(Apinv_numpy - Apinv_thinSVD) / np.linalg.norm(Apinv_numpy))
```

```
9.332652189948006e-17
9.332652189948006e-17
```

Compare the three implementations performances through the Google Colab magic command `%timeit`.

```
%timeit np.linalg.pinv(A)
```

```
13.4 µs ± 199 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit my_pinv_fullSVD(A)
```

```
13.4 µs ± 274 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

```
%timeit my_pinv_thinSVD(A)
```

```
10.5 µs ± 259 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Least-square regression

Consider the linear model

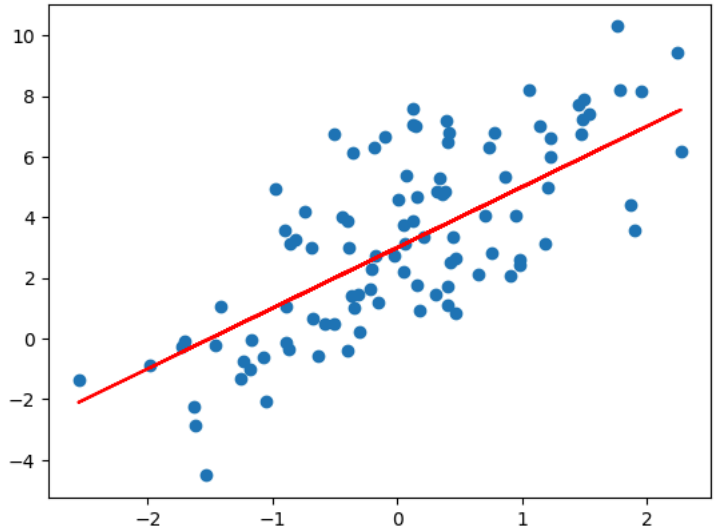
$$ y = mx + q. $$$$ y = mx + q. $$

where $m = 2$$m = 2$ and $q = 3$$q = 3$.

Generate $N = 100$$N = 100$ points $x_i$$x_i$, sampling from a standard Gaussian distribution, and the associated $y_i$$y_i$. Then, add a synthetic noise ($\epsilon_i$$\epsilon_i$) by sampling from a Gaussian distribution with zero mean and standard deviation $\sigma = 2$$\sigma = 2$. Plot the noisy data $(x_i, \tilde{y}_i)$$(x_i, \tilde{y}_i)$, where $\tilde{y}_i = y_i + \epsilon_i$$\tilde{y}_i = y_i + \epsilon_i$, in the $(x,y)$$(x,y)$ plane, together with the line $y = mx + q$$y = mx + q$.

```
m = 2.0
q = 3.0
N = 100
noise = 2.0

np.random.seed(0)

# SOLUTION-BEGIN
X = np.random.randn(N)
Y = m * X + q + noise * np.random.randn(N)
plt.scatter(X, Y)
plt.plot(X, m * X + q, color="red")
# SOLUTION-END
```

```
[<matplotlib.lines.Line2D at 0x7fde3c18c890>]
```



Using the previously implemented functions to compute the Moore-Penrose pseudo inverse, solve the least-squares problem

$$ \min_{m,q} \sum_{i=1}^N (\tilde{y}_i - (m x_i + q))^2 $$$$ \min_{m,q} \sum_{i=1}^N (\tilde{y}_i - (m x_i + q))^2 $$

and display the regression line superimposed to the noisy data and the exact model. Define

$$\Phi = [\mathbf{x}, \mathbf{1}] \in \mathbb R^{N \times 2}$$$$\Phi = [\mathbf{x}, \mathbf{1}] \in \mathbb R^{N \times 2}$$

The least square problem is

$$\Phi \mathbf{w} = \mathbf{y}$$$$\Phi \mathbf{w} = \mathbf{y}$$

With solution

$$\mathbf{w} = \Phi^\dagger \mathbf{y}$$$$\mathbf{w} = \Phi^\dagger \mathbf{y}$$

where

$$\mathbf{w} = [\hat m, \hat q]$$$$\mathbf{w} = [\hat m, \hat q]$$

Notice that in general

$$\mathbf{y}^{Test} = \Phi^{Test} \mathbf{w}$$$$\mathbf{y}^{Test} = \Phi^{Test} \mathbf{w}$$

that in our case is equivalent to

$$\mathbf{y}_i^{Test} = \hat{m} \mathbf{x}_i^{Test} + \hat{q}$$$$\mathbf{y}_i^{Test} = \hat{m} \mathbf{x}_i^{Test} + \hat{q}$$

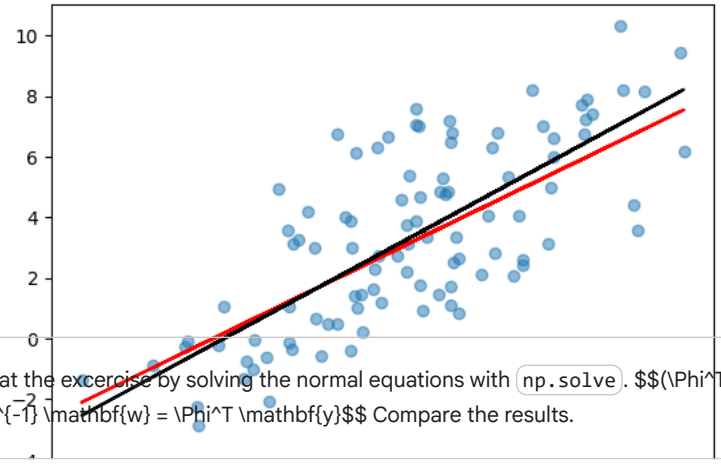
```
# SOLUTION-BEGIN
Phi = np.column_stack([X[:, np.newaxis], np.ones((N, 1))])
z = my_pinv_thinSVD(Phi) @ Y
m_hat = z[0]
q_hat = z[1]
print(f"m_hat = {m_hat:.3f}")
print(f"q_hat = {q_hat:.3f}")
# SOLUTION-END
```

```
m_hat = 2.229
q_hat = 3.150
```

Plot a scatter of the data points, the "real" linear model and the estimated linear model with SVD

```
# SOLUTION-BEGIN
plt.scatter(X, Y, alpha=0.5)
plt.plot(X, m * X + q, color="red")
plt.plot(X, m_hat * X + q_hat, color="k")
# SOLUTION-END
```

```
[<matplotlib.lines.Line2D at 0x7fde37e36cd0>]
```



Repeat the excercise by solving the normal equations with `np.solve`. $$(\Phi^T \Phi)^{-1} \mathbf{w} = \Phi^T \mathbf{y}$$$$(\Phi^T \Phi)^{-1} \mathbf{w} = \Phi^T \mathbf{y}$$ Compare the results.

```
# SOLUTION-BEGIN
z2 = np.linalg.solve(Phi.transpose() @ Phi, Phi.transpose() @ Y)
np.linalg.norm(z - z2)
# SOLUTION-END
```

```
9.930136612989092e-16
```

Ridge regression and Kernel regression

Consider the function

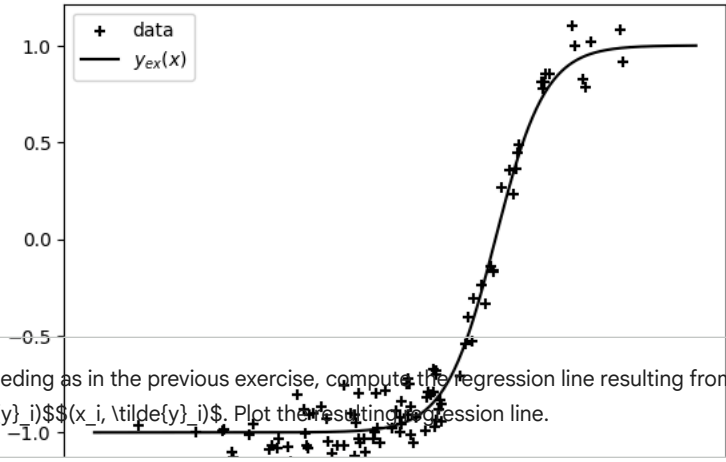
$$ y = f(x) = \tanh(2x - 1). $$$$ y = f(x) = \tanh(2x - 1). $$

Generate $N = 100$$N = 100$ points $x_i$$x_i$, sampling from a standard Gaussian distribution, and the associated $y_i$$y_i$. Then, add a synthetic noise ($\epsilon_i$$\epsilon_i$) by sampling from a Gaussian distribution with zero mean and standard deviation $\sigma = 0.1$$\sigma = 0.1$. Plot the noisy data $(x_i, \tilde{y}_i)$$(x_i, \tilde{y}_i)$, where $\tilde{y}_i = y_i + \epsilon_i$$\tilde{y}_i = y_i + \epsilon_i$, in the $(x,y)$$(x,y)$ plane.

Then, generate 1000 testing points, uniformly distributed in the interval $[-3,3]$$[-3,3]$, and display the function $y = f(x)$$y = f(x)$ in correspondence of the testing points.

```
np.random.seed(0)

# SOLUTION-BEGIN
N = 100
noise = 0.1
y_ex = lambda x: np.tanh(2 * (x - 1))

X = np.random.randn(N, 1)
Y = y_ex(X) + noise * np.random.randn(N, 1)

N_test = 1000
X_test = np.linspace(-3, 3, N_test).reshape((-1, 1))
Y_test_ex = y_ex(X_test)

plt.scatter(X, Y, marker="+", color="black", label="data")
plt.plot(X_test, Y_test_ex, color="black", label="$y_{ex}(x)$")
plt.legend()
# SOLUTION-END
```

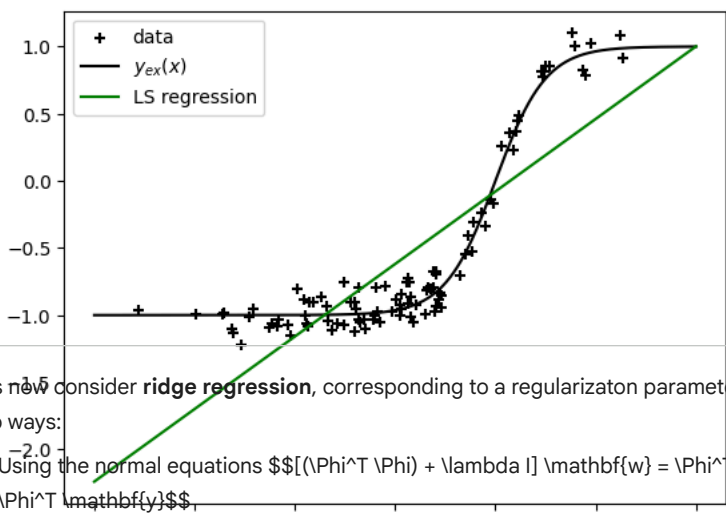
```
<matplotlib.legend.Legend at 0x7fde37e05c50>
```



Proceeding as in the previous exercise, compute the regression line resulting from the **least squares regression** of data $(x_i, \tilde{y}_i)$$(x_i, \tilde{y}_i)$. Plot the resulting regression line.

```
# SOLUTION-BEGIN
Phi = np.block([X, np.ones((N, 1))])
z = my_pinv_thinSVD(Phi) @ Y
m_hat = z[0, 0]
q_hat = z[1, 0]
print(f"m_hat = {m_hat:.3f}")
print(f"q_hat = {q_hat:.3f}")
Y_test_LS = m_hat * X_test + q_hat

plt.scatter(X, Y, marker="+", color="black", label="data")
plt.plot(X_test, Y_test_ex, color="black", label="$y_{ex}(x)$")
plt.plot(X_test, Y_test_LS, color="green", label="LS regression")
plt.legend()
# SOLUTION-END
```

```
m_hat = 0.540
q_hat = -0.619
<matplotlib.legend.Legend at 0x7fde37dbee50>
```



Let us now consider **ridge regression**, corresponding to a regularizaton parameter $\lambda = 1.0$$\lambda = 1.0$. Solve the equations in two ways:

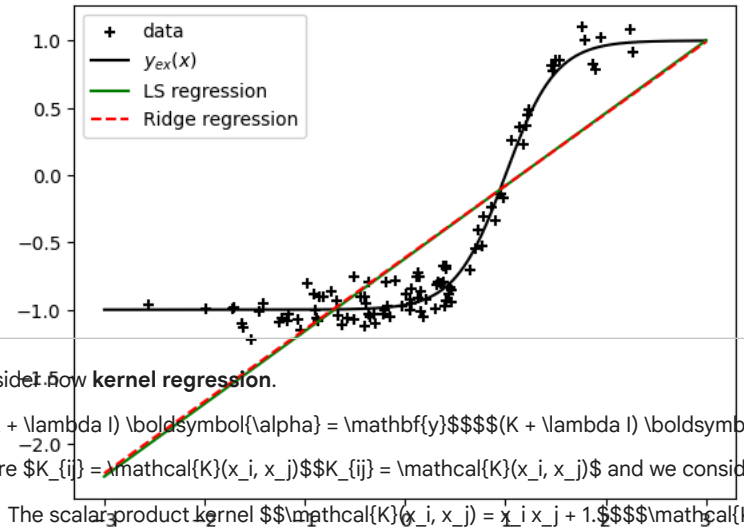
1. Using the normal equations $$[(\Phi^T \Phi) + \lambda I] \mathbf{w} = \Phi^T \mathbf{y}$$$$[(\Phi^T \Phi) + \lambda I] \mathbf{w} = \Phi^T \mathbf{y}$$,
2. Using the Woodbury identity to write the normal equations in "kernel-form" $$\mathbf{w} = \Phi^T \boldsymbol{\alpha}, \quad [(\Phi \Phi^T) + \lambda I] \boldsymbol{\alpha} = \mathbf{y}$$$$\mathbf{w} = \Phi^T \boldsymbol{\alpha}, \quad [(\Phi \Phi^T) + \lambda I] \boldsymbol{\alpha} = \mathbf{y}$$

Check that the two approaches lead to the same result. Compare the obtained regression line with the one obtained through least squares regression when changing $\lambda$$\lambda$.

```
# SOLUTION-BEGIN
lam = 1.0
PhiPhiT = Phi @ Phi.transpose()
alpha = np.linalg.solve(PhiPhiT + lam * np.eye(N), Y)
w = Phi.transpose() @ alpha
w2 = np.linalg.solve(Phi.T @ Phi + lam * np.eye(2), Phi.T @ Y)
Phi_test = np.block([X_test, np.ones((N_test, 1))])
Y_test_RR = Phi_test @ w

plt.scatter(X, Y, marker="+", color="black", label="data")
```

```
plt.plot(X_test, Y_test_ex, color="black", label="$y_{ex}(x)$")
plt.plot(X_test, Y_test_LS, color="green", label="LS regression")
plt.plot(X_test, Y_test_RR, color="red", linestyle="--", label="Ridge regression")
plt.legend()
# SOLUTION-END
```

```
<matplotlib.legend.Legend at 0x7fde3ce7f7d0>
```



Consider now **kernel regression**.

$$(K + \lambda I) \boldsymbol{\alpha} = \mathbf{y}$$$$(K + \lambda I) \boldsymbol{\alpha} = \mathbf{y}$$

Where $K_{ij} = \mathcal{K}(x_i, x_j)$$K_{ij} = \mathcal{K}(x_i, x_j)$ and we consider

1. The scalar product kernel $$\mathcal{K}(x_i, x_j) = x_i x_j + 1.$$$$\mathcal{K}(x_i, x_j) = x_i x_j + 1.$$
2. The higher-order scalar product kernel, for $q > 1$$q > 1$. $$\mathcal{K}(x_i, x_j) = (x_i x_j + 1)^q.$$$$\mathcal{K}(x_i, x_j) = (x_i x_j + 1)^q.$$
3. The Gaussian kernel, for $\sigma > 0$$\sigma > 0$. $$\mathcal{K}(x_i, x_j) = \exp\left(-\frac{(x_i - x_j)^2}{2 \sigma^2}\right).$$$$\mathcal{K}(x_i, x_j) = \exp\left(-\frac{(x_i - x_j)^2}{2 \sigma^2}\right).$$

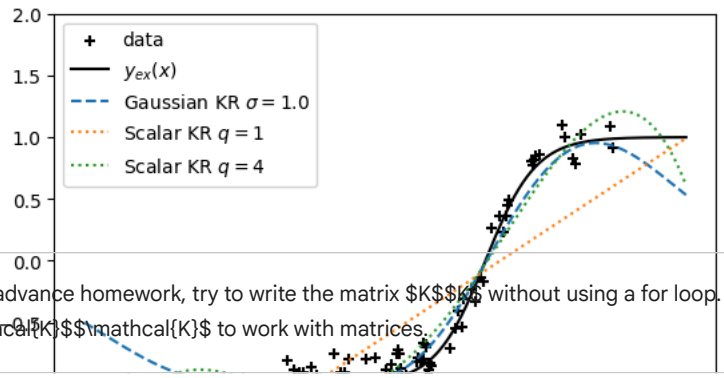
The evaluation on the test data-point $\hat{x}$$\hat{x}$ is computed as

$$\hat{\mathbf{y}} = \sum_{i = 1}^N \alpha_i \mathcal{K}(\hat{x}, x_i) = K^{Test} \boldsymbol{\alpha}$$$$\hat{\mathbf{y}} = \sum_{i = 1}^N \alpha_i \mathcal{K}(\hat{x}, x_i) = K^{Test} \boldsymbol{\alpha}$$

```
# SOLUTION-BEGIN
lam = 1.0
q = 4
sigma = 1.0


def product_kernel(x1, x2):
    return x1 * x2 + 1


def high_order_kernel(x1, x2):
    return (x1 * x2 + 1) ** q


def gaussian_kernel(x1, x2):
    return np.exp(-(((x1 - x2) / sigma) ** 2) / 2)


def kernel_regression(kernel):
    N = X.shape[0]
    K = np.empty((N, N))
    for i in range(N):
        for j in range(N):
            K[i, j] = kernel(X[i], X[j])
    alpha = np.linalg.solve(K + lam * np.eye(N), Y)
    K_test = np.empty((X_test.shape[0], N))
    for i in range(K_test.shape[0]):
        for j in range(K_test.shape[1]):
            K_test[i, j] = kernel(X_test[i], X[j])
    Y_test_KR = K_test @ alpha
    return Y_test_KR


plt.scatter(X, Y, marker="+", color="black", label="data")
plt.plot(X_test, Y_test_ex, color="black", label="$y_{ex}(x)$")
plt.plot(
```

```
    X_test,
    kernel_regression(gaussian_kernel),
    linestyle="--",
    label=f"Gaussian KR $\sigma = {sigma}$",
)
plt.plot(
    X_test,
    kernel_regression(product_kernel),
    linestyle=":",
    label=f"Scalar KR $q = 1$",
)
plt.plot(
    X_test,
    kernel_regression(high_order_kernel),
    linestyle=":",
    label=f"Scalar KR $q = {q}$",
)
plt.legend()
plt.ylim([-2, 2])
# SOLUTION-END
```

```
/tmp/ipykernel_2285756/1513242630.py:24: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and
  K[i, j] = kernel(X[i], X[j])
/tmp/ipykernel_2285756/1513242630.py:29: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and
  K_test[i, j] = kernel(X_test[i], X[j])
(-2.0, 2.0)
```

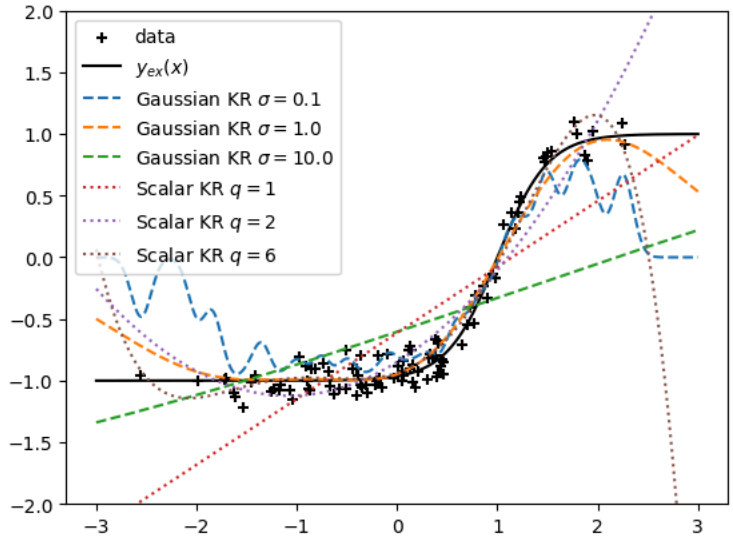


As an advance homework, try to write the matrix $K$$\mathcal{K}$ without using a for loop. Hint: you have to rewrite the kernel function $\mathcal{K}$$\mathcal{K}$ to work with matrices.

```
# SOLUTION-BEGIN
lam = 1.0

def scalar_product_kernel_q(q):
    def kernel(X1, X2):
        return (X1 @ X2.T + 1) ** q

    return kernel


def gaussian_kernel_sigma(sigma):
    def kernel(X1, X2):
        X1_sq = np.sum(X1**2, axis=1)[:, None]
        X2_sq = np.sum(X2**2, axis=1)[None, :]
        sq_dists = X1_sq + X2_sq - 2 * X1 @ X2.T
        return np.exp(-sq_dists / (2 * sigma**2))

    return kernel


def kernel_regression(kernel):
    K = kernel(X, X)
    N = X.shape[0]
    alpha = np.linalg.solve(K + lam * np.eye(N), Y)
    K_test = kernel(X_test, X)
    Y_test_KR = K_test @ alpha
    return Y_test_KR


plt.scatter(X, Y, marker="+", color="black", label="data")
plt.plot(X_test, Y_test_ex, color="black", label="$y_{ex}(x)$")
for sigma in [0.1, 1.0, 10.0]:
    plt.plot(
        X_test,
        kernel_regression(gaussian_kernel_sigma(sigma)),
        linestyle="--",
```

```
        label=f"Gaussian KR $\sigma = {sigma}$",
    )
for q in [1, 2, 6]:
    plt.plot(
        X_test,
        kernel_regression(scalar_product_kernel_q(q)),
        linestyle=":",
        label=f"Scalar KR $q = {q}$",
    )
plt.legend()
plt.ylim([-2, 2])
# SOLUTION-END
```

```
(-2.0, 2.0)
```



PageRank

We will explore how the **PageRank algorithm** works and how it relates to real-world web traffic data.

The dataset you will use is derived from the **Wikipedia category 'Machine Learning'**. Each node corresponds to a Wikipedia article, and each directed edge represents a hyperlink between two pages. Additionally, a separate file contains the **page view statistics (traffic)** for each article. The data was downloaded using Wikipedia API, for the curious the scripts are in the `datagen` folder.

Your goals are:

1. Load and inspect the dataset (`nodes.csv`, `edges.csv`, and `traffic.csv`).
2. Build a directed graph using NetworkX.
3. Compute the PageRank vector using both the NetworkX implementation and the power iteration method.
4. Compare the results of both methods and visualize the graph.
5. Compare PageRank values with real page traffic and discuss their correlation.

```
import pandas as pd
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np

# Load the data
nodes = pd.read_csv("nodes.csv")
edges = pd.read_csv("edges.csv")
traffic = pd.read_csv("traffic.csv")

print(f"Loaded {len(nodes)} nodes, {len(edges)} edges, and {len(traffic)} traffic entries")
```

Build the Graph and Transition Matrix

We first create a **directed graph** representing the hyperlink structure between Wikipedia articles. Each node corresponds to a page, and each directed edge represents a link from one page to another.

Then we construct the **transition matrix** $M$$M$ such that $M_{ij}$$M_{ij}$ represents the probability of moving from page $j$$j$ to page $i$$i$ when following a link.

```
graph = nx.DiGraph()
graph.add_nodes_from(nodes["node"])
graph.add_edges_from(edges[["source", "target"]].values)

N = len(nodes)
node_index = nodes["node"].to_dict()
node_index = {node_index[k]: k for k in node_index}

# Build the stochastic matrix M
M = np.zeros((N, N))
for u, v in graph.edges():
    if graph.out_degree(u) > 0:
        M[node_index[v], node_index[u]] = 1.0 / graph.out_degree(u)

print(f"Transition matrix M built with shape: {M.shape}")
```

The Damping Factor and Power Iteration

The **damping factor** (usually denoted as $d$$d$) models the probability that a user follows a link versus jumping to a random page. Typically, $d = 0.85$$d = 0.85$ means the user follows a link 85% of the time and jumps randomly 15% of the time.

We will:

- Compute PageRank using NetworkX's built-in implementation.
- Compute PageRank manually using the **power iteration method** applied to the Google matrix $$G = d M + \frac{1 - d}{N} 1_{N\times N}$$$$G = d M + \frac{1 - d}{N} 1_{N\times N}$$
- Compare the two results.

```
damping_factor = 0.85

# --- Compute PageRank using NetworkX ---
pagerank = nx.pagerank(graph, alpha=damping_factor)
pagerank_vec = np.array([pagerank[n] for n in nodes["node"]])
nx.set_node_attributes(graph, pagerank, "pagerank")

# --- Compute PageRank using power iteration ---
# SOLUTION-BEGIN
G_matrix = damping_factor * M + (1 - damping_factor) / N * np.ones((N, N))

p = np.ones(N) / N
tol = 1e-10
max_iter = 1000

for i in range(max_iter):
    p_next = G_matrix @ p
    p_next /= np.linalg.norm(p_next)
    if np.linalg.norm(p_next - p, 2) < tol:
        print(f"Converged after {i} iterations")
        break
    p = p_next

pagerank_powit = p / np.sum(p)
# SOLUTION-END

# Compare results
corr = np.corrcoef(pagerank_vec, pagerank_powit)[0, 1]
l1_diff = np.linalg.norm(pagerank_vec - pagerank_powit, 1)

print("\n--- PageRank Comparison ---")
print(f"Pearson correlation: {corr:.6f}")
print(f"L1 difference:       {l1_diff:.6e}")

plt.plot(pagerank_vec, 'o-')
plt.plot(pagerank_powit, 'x-')
```

Visualize the Graph

We now visualize the graph using a spring layout. Node sizes are proportional to their PageRank values.

- Larger nodes correspond to more 'important' pages.
- The layout helps highlight clusters and central pages within the category.

```
plt.figure(figsize=(12, 10))

pos = nx.spring_layout(graph, k=0.5, seed=42)

# Draw nodes
nx.draw_networkx_nodes(
    graph,
    pos,
    node_size=pagerank_vec * 5000,
    node_color="skyblue",
    alpha=0.8,
    edgecolors="gray",
)

# Draw edges
nx.draw_networkx_edges(graph, pos, arrows=True, alpha=0.4)

# Draw labels (of most important pages)optional for smaller graphs
nx.draw_networkx_labels(
    graph,
    pos,
    labels={
        n: n for n in nodes["node"] if pagerank[n] > np.quantile(pagerank_vec, 0.9)
    },
    font_color="k",
    font_size=10,
    bbox=dict(facecolor='white', edgecolor='black', boxstyle='round,pad=0.3')
)

plt.title("Wikipedia Machine Learning Graph\nNode size = PageRank", fontsize=14)
plt.axis("off")
plt.tight_layout()
plt.show()
```

Compare PageRank with Wikipedia Traffic

Finally, we compare the computed PageRank values with **real page traffic** obtained from the Wikimedia API. This allows us to see whether the theoretical importance (PageRank) corresponds to actual user visits.

- We add the PageRank value to the traffic DataFrame.
- Plot traffic vs. PageRank on a log-log scale.
- Compute and print their correlation.
- Print the 10 most important pages (highest pagerank score).

```
traffic["pagerank"] = traffic["node"].map(pagerank)

# SOLUTION-BEGIN
plt.figure(figsize=(7, 6))
plt.scatter(traffic.traffic, traffic.pagerank, alpha=0.7)
plt.xscale("log")
plt.yscale("log")
plt.xlabel("Wikipedia Traffic (log scale)")
plt.ylabel("PageRank (log scale)")
plt.title("Correlation between PageRank and Traffic")
plt.tight_layout()
plt.show()

corr_traffic = np.corrcoef(traffic.traffic, traffic.pagerank)[0, 1]
print(f"Correlation between traffic and pagerank: {corr_traffic:.3f}")

print("\nTop 10 pages by PageRank:")
print(traffic.nlargest(10, "pagerank")[["node", "pagerank", "traffic"]])
# SOLUTION-END
```

Geometric interpretation of PCA

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(0)
```

Define the unit vector $\mathbf{z}_1 = \begin{pmatrix} \cos(\theta_1) \\ \sin(\theta_1)\end{pmatrix}$$\mathbf{z}_1 = \begin{pmatrix} \cos(\theta_1) \\ \sin(\theta_1)\end{pmatrix}$ where $\theta_1 = \pi/6$$\theta_1 = \pi/6$ and a unit vector $\mathbf{z}_2 \perp \mathbf{z}_1$$\mathbf{z}_2 \perp \mathbf{z}_1$.

```
theta1 = np.pi / 6
theta2 = theta1 + np.pi / 2

z1 = np.array((np.cos(theta1), np.sin(theta1)))
z2 = np.array((np.cos(theta2), np.sin(theta2)))
```

Define the point $\mathbf{b} = \begin{pmatrix} 20 \\ 30\end{pmatrix}$$\mathbf{b} = \begin{pmatrix} 20 \\ 30\end{pmatrix}$

```
b = np.array((20, 30))
```

Generate $n = 1000$$n = 1000$ points $\mathbf{x}_i$$\mathbf{x}_i$ according to the formula:

$\mathbf{x}_i = A \mathbf{r}_i + \mathbf{b}$$\mathbf{x}_i = A \mathbf{r}_i + \mathbf{b}$

where $A$$A$ is a transformation matrix with columns $[\rho_1 \mathbf{z}_1, \rho_2 \mathbf{z}_2] $$[\rho_1 \mathbf{z}_1, \rho_2 \mathbf{z}_2] $ and the $\mathbf{r}_{i}$$\mathbf{r}_{i}$ are random vectors with independently generated componenst according to a normal distribution.

```
rho1 = 12.0
rho2 = 3.0
n_points = 1000

seeds = np.random.randn(2, n_points)
X = np.column_stack((rho1 * z1, rho2 * z2)) @ seeds + b[:, None]

X.shape
```

Plot the generated numbers in a plane, showing the directions $\mathbf{v}_1$$\mathbf{v}_1$ and $\mathbf{v}_2$$\mathbf{v}_2$, rescaled by $\rho_1$$\rho_1$ and $\rho_2$$\rho_2$, respectivaly.

```
fig, ax = plt.subplots(nrows=1, ncols=1)
ax.scatter(X[0, :], X[1, :])
ax.arrow(
    b[0] - z1[0] * rho1,
    b[1] - z1[1] * rho1,
    2 * z1[0] * rho1,
    2 * z1[1] * rho1,
    color="black",
)
ax.arrow(
    b[0] - z2[0] * rho2,
    b[1] - z2[1] * rho2,
    2 * z2[0] * rho2,
    2 * z2[1] * rho2,
    color="black",
)
ax.set_aspect("equal")
```

Perform PCA on the matrix $X$$X$:

$\boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$$\boldsymbol{\mu} = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$

$\overline{X} = X - \boldsymbol{\mu} [1,\dots,1]$$\overline{X} = X - \boldsymbol{\mu} [1,\dots,1]$

$\overline{X} = U \Sigma V^T$$\overline{X} = U \Sigma V^T$

```
# SOLUTION-BEGIN
X_mean = np.mean(X, axis=1)
U, s, VT = np.linalg.svd(X - X_mean[:, None], full_matrices=False)
# SOLUTION-END
```

Plot the first two singular vectors, rescaled by the root of the sample variance:

${\sqrt{\frac{\sigma_1^2}{(n - 1)}}}{\mathbf{u}_1}, {\sqrt{\frac{\sigma_2^2}{(n - 1)}}}{\mathbf{u}_2}$${\sqrt{\frac{\sigma_1^2}{(n - 1)}}}{\mathbf{u}_1}, {\sqrt{\frac{\sigma_2^2}{(n - 1)}}}{\mathbf{u}_2}$

```
# SOLUTION-BEGIN
u1 = U[:, 0]
u2 = U[:, 1]

r = s / np.sqrt(n_points - 1)

fig, ax = plt.subplots(nrows=1, ncols=1)
ax.scatter(X[0, :], X[1, :])
plt.arrow(
    X_mean[0] - u1[0] * r[0],
    X_mean[1] - u1[1] * r[0],
    2 * u1[0] * r[0],
    2 * u1[1] * r[0],
    color="red",
)
plt.arrow(
    X_mean[0] - u2[0] * r[1],
    X_mean[1] - u2[1] * r[1],
    2 * u2[0] * r[1],
    2 * u2[1] * r[1],
    color="red",
)
ax.set_aspect("equal")
# SOLUTION-END
```

Print and compare the real transformation directions ($\mathbf{z}_1, \mathbf{z}_2$$\mathbf{z}_1, \mathbf{z}_2$) and the estimated ones ($\mathbf{u}_1, \mathbf{u}_2$$\mathbf{u}_1, \mathbf{u}_2$)

```
# SOLUTION-BEGIN
# real transformation directions
print(z1, z2)
# estimated transformation directions
print(u1, u2)
# SOLUTION-END
```

Compute the principal components of the data:

$\Phi = U^T \overline{X}$$\Phi = U^T \overline{X}$

```
# SOLUTION-BEGIN
Phi = np.matmul(U.transpose(), X - X_mean[:, None])
# SOLUTION-END
```

Make a scatterplot of the two first principal components ($\phi_1$$\phi_1$, $\phi_2$$\phi_2$), rescaled by the root of the sample variance

```
# SOLUTION-BEGIN
fig, ax = plt.subplots(nrows=1, ncols=1)
ax.scatter(Phi[0, :] / r[0], Phi[1, :] / r[1])
ax.set_aspect("equal")
# SOLUTION-END
```

Serum proteomic pattern diagnostics

Dataset from the FDA-NCI Clinical Proteomics Program Databank [1,2].

Each column of `A` represents measurements taken from a patient. There are 216 columns representing 216 patients, out of which 121 are ovarian cancer patients and 95 are normal patients.

Each row represents the ion intensity level at a specific mass-charge value indicated in MZ. There are 2000 mass-charge values, and each row represents the ion-intensity levels of the patients at that particular mass-charge value.

The variable `grp` holds the index information as to which of these samples represent cancer patients and which ones represent normal patients.

- [1] T.P. Conrads, et al., "High-resolution serum proteomic features for ovarian detection", Endocrine-Related Cancer, 11, 2004, pp. 163-178.
- [2] E.F. Petricoin, et al., "Use of proteomic patterns in serum to identify ovarian cancer", Lancet, 359(9306), 2002, pp. 572-577.

```
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

ovariancancer_obs_path = "./ovariancancer_obs.csv"
ovariancancer_grp_path = "./ovariancancer_grp.csv"

A = np.genfromtxt(ovariancancer_obs_path, delimiter=",").transpose()
with open(ovariancancer_grp_path, "r") as fp:
    grp = np.array(fp.read().split("\n"))
grp = grp[grp != ""]
```

How many patients? How many features?

```
n_features = A.shape[0]
n_patients = A.shape[1]
print(f"{n_features} features")
print(f"{n_patients} patients")
```

How many patiens with and without cancer are there in the dataset?

```
print(f"{np.sum(grp == 'Cancer')} Cancer")
print(f"{np.sum(grp == 'Normal')} Normal")
print(f"Total: {grp.size}")
is_sane = np.where(grp == "Normal", 1, 0)
```

Choose two proteins (between 1 and 2000), and make a scatterplot of the corresponding values plotting with different colors the data corresponding to patients with and without cancer. Can you easily distinguish between patients with/without cancer?

```
protein_x = 0
protein_y = 1

plt.scatter(
    A[protein_x, grp == "Cancer"], A[protein_y, grp == "Cancer"], label="Cancer"
)
plt.scatter(
    A[protein_x, grp == "Normal"], A[protein_y, grp == "Normal"], label="Normal"
)
plt.legend()
plt.show()
```

Repeat the previous point, this time by considering three proteins.

```
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

protein_x = 1000
protein_y = 2000
protein_z = 3000

for label in ("Cancer", "Normal"):
    ax.scatter(
        A[protein_x, grp == label],
        A[protein_y, grp == label],
        A[protein_z, grp == label],
        label=label,
        marker="x",
        s=50,
    )
```

```
plt.legend()
ax.view_init(25, 20)
```

Principal Components Analysis

Perform PCA on the data. Then, plot the trend of

- the singular values $\sigma_k$$\sigma_k$
- the cumulate fraction of singular values: $\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{q} \sigma_i}$$\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{q} \sigma_i}$
- the fraction of the "explained variance": $\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{q} \sigma_i^2}$$\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{q} \sigma_i^2}$

```
# SOLUTION-BEGIN
A_mean = A.mean(axis=1)
A_bar = A - A_mean[:, None]
U, s, VT = np.linalg.svd(A_bar, full_matrices=False)

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 6))

axes[0].semilogy(s[:-1], "o-")
axes[0].set_title("Singluar values")

axes[1].plot(np.cumsum(s) / np.sum(s), "o-")
axes[1].set_title("Cumulate fraction of singular values")

axes[2].plot(np.cumsum(s**2) / np.sum(s**2), "o-")
axes[2].set_title("Explained variance")
# SOLUTION-END
```

Make a scatterplot of the first two principal components of the patients.

```
# SOLUTION-BEGIN
Phi = U[:, :2].T @ A_bar
plt.scatter(Phi[0, :], Phi[1, :], marker="o", c=(grp == "Normal"), s=50)
plt.show()
# SOLUTION-END
```

Make a scatterplot of the first three principal components of the patients.

```
# SOLUTION-BEGIN
fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")
Phi = U[:, :3].T @ A_bar
ax.scatter(Phi[0, :], Phi[1, :], Phi[2, :], marker="x", c=(grp == "Normal"), s=50)
ax.view_init(25, 20)
plt.show()
# SOLUTION-END
```

Or more compactly, and with an interactive plot

```
import plotly.express as px

px.scatter_3d(x=Phi[0, :], y=Phi[1, :], z=Phi[2, :], color=grp)
```

Visualize the protein components of the first three principal components

```
# SOLUTION-BEGIN
plt.plot(U[:, 0:3])
plt.legend(["PC1", "PC2", "PC3"])
plt.xlabel("Biomarker ID")
plt.ylabel("Importance")
# SOLUTION-END
```

The following is a generalization of the thresholding algorithm we have seen in the previous notebook for higher dimensions: instead of a vertical line, think to a plane in 3D that separates the two groups. The implementation of this classification algorithm (SVC) will be an argument of Lab04. For now, we just use this (SVC) as a tool from a library. We care only about the output training accuracy.

```
from sklearn.svm import SVC


def classify(X, labels):
    clf = SVC(kernel="linear")
    clf.fit(X.T, labels)
    y_pred = clf.predict(X.T)
    accuracy = np.mean(y_pred == labels)
    print(f"Training accuracy: {accuracy:.2f}")

classify(Phi, is_sane)
```

Handwriting recognition

Data import and visualization

Import the MNIST train dataset (https://en.wikipedia.org/wiki/MNIST_database)

```
import matplotlib.pyplot as plt
import numpy as np

# This dataset is contained in the sample data directory of Google Colab online runtimes
data = np.genfromtxt("./mnist_train_small.csv", delimiter=",")
data.shape
```

Store the data in a matrix and the labels in a vector

```
labels_full = data[:, 0]
A_full = data[:, 1:].transpose()
labels_full.shape, A_full.shape
```

Visualize the first 30 pictures with the corresponding labels

```
fig, axs = plt.subplots(ncols=10, nrows=3, figsize=(20, 6))
axs = axs.reshape((-1,))
for i in range(30):
    image_i = A_full[:, i].reshape((28, 28))
    axs[i].imshow(image_i, cmap="gray")
    axs[i].set_title(f"{labels_full[i]:.0f}")
    axs[i].axis("off")
```

Visualize the first 30 pictures corresponding the the digit "9"

```
A_filtered = A_full[:, labels_full == 9]
fig, axs = plt.subplots(ncols=10, nrows=3, figsize=(20, 6))
axs = axs.reshape((-1,))
for i in range(30):
    image_i = A_filtered[:, i].reshape((28, 28))
    axs[i].imshow(image_i, cmap="gray")
    axs[i].axis("off")
```

Classification of two digits (0 and 9)

We now focus on the digits 0 and 9 only. Build a reduced training dataset containing only images corresponding to these digits.

```
# HINT: use a bit-mask that is true when labels are 0 OR 9
#       "OR" might be a numpy operator, why don't you look up the documentation?
# SOLUTION-BEGIN
digits = (0, 9)
```

```
mask = np.logical_or(labels_full == digits[0], labels_full == digits[1])
A = A_full[:, mask]
labels = labels_full[mask]
# SOLUTION-END
```

Plot the first 30 training images.

```
# SOLUTION-BEGIN
fig, axs = plt.subplots(nrows=3, ncols=10, figsize=(20, 6))
axs = axs.reshape((-1,))
for i in range(len(axs)):
    image_i = A[:, i].reshape((28, 28))
    axs[i].imshow(image_i, cmap="gray")
    axs[i].set_title(int(labels[i]))
    axs[i].axis("off")
# SOLUTION-END
```

Compute and visualize the average of the training data

```
# SOLUTION-BEGIN
A_mean = A.mean(axis=1)
plt.imshow(A_mean.reshape((28, 28)), cmap="gray")
plt.axis("off")
# SOLUTION-END
```

Perform PCA on the data (SVD on the zero mean data). Then, plot the trend of

- the singular values $\sigma_k$$\sigma_k$
- the cumulate fraction of singular values: $\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{q} \sigma_i}$$\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{q} \sigma_i}$
- the fraction of the "explained variance": $\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{q} \sigma_i^2}$$\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{q} \sigma_i^2}$

```
# SOLUTION-BEGIN
A_bar = A - A_mean[:, None]
U, s, VT = np.linalg.svd(A_bar, full_matrices=False)

fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 6))

axes[0].semilogy(s, "o-")
axes[0].set_title("Singluar values")

axes[1].plot(np.cumsum(s) / np.sum(s), "o-")
axes[1].set_title("Cumulate fraction of singular values")

axes[2].plot(np.cumsum(s**2) / np.sum(s**2), "o-")
axes[2].set_title("Explained variance")
# SOLUTION-END
```

Visualize the first 30 principal axes (columns of $U$$U$)

```
# SOLUTION-BEGIN
fig, axs = plt.subplots(nrows=3, ncols=10, figsize=(20, 6))
axs = axs.reshape((-1,))
for i in range(len(axs)):
    image_i = U[:, i].reshape((28, 28))
    axs[i].imshow(image_i, cmap="gray")
    axs[i].axis("off")
    axs[i].set_title(f"$u_{{{i + 1}}}$")
# SOLUTION-END
```

Compute the first two principal components corresponding to the first image

```
# SOLUTION-BEGIN
print(f"1st principal component: {np.inner(A_bar[:, 0], U[:, 0])}" )
print(f"2nd principal component: {np.inner(A_bar[:, 0], U[:, 1])}" )
# SOLUTION-END
```

Define a matrix corresponding all the principal components of all the images ($U^T \bar{A}$$U^T \bar{A}$) and check the result of the previous point

```
# SOLUTION-BEGIN
A_pc = np.matmul(U.T, A_bar)
print(f"1st principal component: {A_pc[0, 0]}")
print(f"2nd principal component: {A_pc[1, 0]}")
# SOLUTION-END
```

Create a scatterplot for the first 2 principal components of all the training images, grouped by label. Consider only the first 500 pictures.

```
# naive, slow way
# for i in range(500):
#     # take the i-th sample and find the magnitude of its projection
#     # (that is, scalar prod) on the first and second principal axes
#     x = np.inner(A_bar[:, i], U[:, 0])
#     y = np.inner(A_bar[:, i], U[:, 1])
#     col = "r" if labels[i] == digits[0] else "b"
#     plt.scatter(x, y, marker="x", color=col, s=50)

# CHALLENGE: do the same of the above without a for loop
# SOLUTION-BEGIN
plt.scatter(A_pc[0, :500], A_pc[1, :500], marker="x", c=labels[:500], s=50)
# SOLUTION-END
```

Make predictions on the test dataset

Define a classifier, by fixing a threshold on the first principal component. Namely, if a sample has the first principal component smaller than the threshold, it is in the first class, else it is in the second. Then, test the classifier on the MNIST test dataset. Compute the number of true/false positive/negative for both the digits (confusion matrix) and the accuracy of the classifier.

Choose a threshold by manual inspection, and plot the vertical line representing it

```
threshold = 999
plt.scatter(A_pc[0, :500], A_pc[1, :500], marker="x", c=labels[:500], s=50)
plt.axvline(threshold, color="k", linestyle="--")
```

Load the test dataset

```
# SOLUTION-BEGIN
data_test = np.genfromtxt("./mnist_test.csv", delimiter=",")
labels_full_test = data_test[:, 0]
A_full_test = data_test[:, 1:].transpose()
labels_full_test.shape, A_full_test.shape
# SOLUTION-END
```

Extract the pictures of 0s and 9s

```
# SOLUTION-BEGIN
mask = np.logical_or(labels_full_test == digits[0], labels_full_test == digits[1])
A_test = A_full_test[:, mask]
labels_test = labels_full_test[mask]
labels_test.shape, A_test.shape
# SOLUTION-END
```

Plot the first 500 samples transformed following the **TRAINING** data principal components.

```
# SOLUTION-BEGIN
# A_mean is the one of the training dataset!
A_pc_test = U.T @ (A_test - A_mean[:, None])
plt.scatter(A_pc_test[0, :500], A_pc_test[1, :500], marker="x", c=labels_test[:500], s=50)
plt.axvline(threshold, color="k", linestyle="--")
# SOLUTION-END
```

Compute the confusion matrix, namely how many 0s are classified as 0s, how many 0s are classified as 9s, how many 9s are classified as 0s and how many 9s are classified as 9s.

```
PC_1 = A_pc_test[0, :]

# SOLUTION-BEGIN
labels_predicted = np.where(PC_1 > threshold, digits[0], digits[1])

# the operator "&" in numpy is equivalent to "np.logical_and"
true_0 = np.sum((labels_test == digits[0]) & (labels_predicted == digits[0]))
false_0 = np.sum((labels_test == digits[1]) & (labels_predicted == digits[0]))
true_1 = np.sum((labels_test == digits[1]) & (labels_predicted == digits[1]))
false_1 = np.sum((labels_test == digits[0]) & (labels_predicted == digits[1]))

print(f"true  {digits[0]}: {true_0}")
print(f"false {digits[0]}: {false_0}")
print(f"true  {digits[1]}: {true_1}")
print(f"false {digits[1]}: {false_1}")
accuracy = (true_0 + true_1) / (true_0 + true_1 + false_0 + false_1)
print(f"accuracy = {accuracy * 100:.2f} %")
# SOLUTION-END
```

You can do the same more compactly with scikit learn

```
from sklearn import metrics

cm_display = metrics.ConfusionMatrixDisplay.from_predictions(
    labels_test, labels_predicted, cmap=plt.cm.Blues, normalize="true"
)
```

Noise Removal

Load a picture as a 3-dimensional `np.array`

```
import numpy as np
from matplotlib.image import imread
import matplotlib.pyplot as plt
from scipy.ndimage import rotate

# write here the import path of the image
# test this with
# - TarantulaNebula
# - mondrian
# - mondrian (rotated)
image_path = "./mondrian.jpg"
A = imread(image_path)
# A = rotate(A, 20, reshape=False)
```

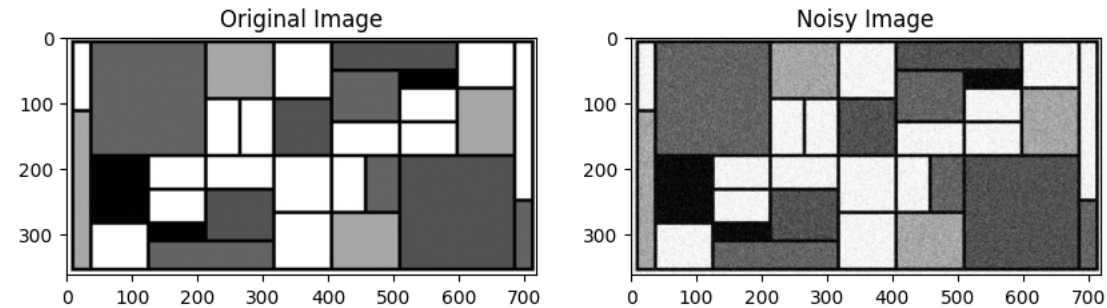
Transform the image in gray-scale and add random noise

```
X = np.mean(A, axis=2) / A.max()
gamma = 0.1
X_noisy = X + gamma * np.random.randn(*X.shape)
X_noisy = np.clip(X_noisy, 0, 1)
```

Plot the original image and the noisy one

```
fig, axs = plt.subplots(1, 2, figsize=(10, 5))
axs[0].imshow(X, cmap="gray")
axs[0].set_title("Original Image")
axs[1].imshow(X_noisy, cmap="gray")
axs[1].set_title("Noisy Image")
```

```
Text(0.5, 1.0, 'Noisy Image')
```

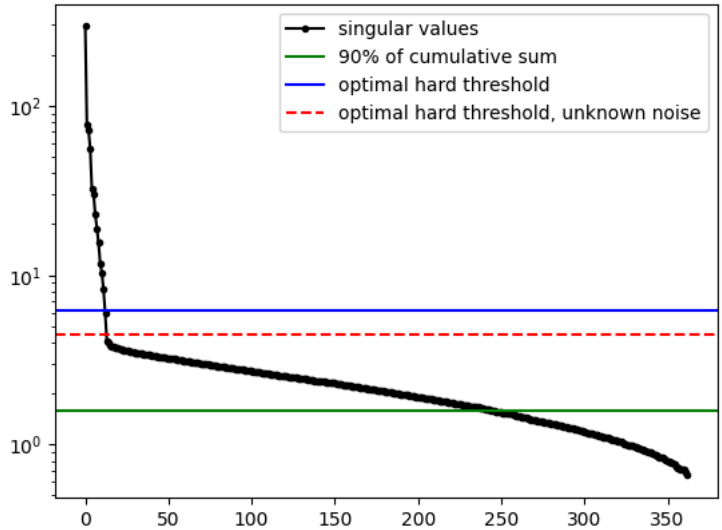


Compute and visualize the threshold for denoising with:

- optimal hard threshold $$(4 / \sqrt{3})\sqrt{n} \gamma$$$$(4 / \sqrt{3})\sqrt{n} \gamma$$
- optimal hard threshold with unknown noise (slide 7, NAML_Lect300925_v2) $$\tau = \omega(\beta) \sigma_{median}$$$$\tau = \omega(\beta) \sigma_{median}$$ $$\omega(\beta) \sim 0.56 \beta^3 - 0.95 \beta^2 + 1.82 \beta + 1.43, \quad \beta = m / n$$$$\omega(\beta) \sim 0.56 \beta^3 - 0.95 \beta^2 + 1.82 \beta + 1.43, \quad \beta = m / n$$
- 90% energy (0.9 of cumulative sum)

```
U, s, VT = np.linalg.svd(X_noisy, full_matrices=0)

# SOLUTION-BEGIN
# Truncate using optimal hard threshold
# the aspect ratio is not too distorted, we use the formula for the square
m, n = X.shape
cutoff = (4 / np.sqrt(3)) * np.sqrt(n) * gamma

# Truncate to keep 90% of cumulative sum
cumsum_thrshold = 0.90
cdS = np.cumsum(s) / np.sum(s)
r90 = np.min(np.where(cdS > cumsum_thrshold))
cutoff90 = s[r90]

# Supposing unknown noise
sigma_med = np.median(s)
beta = m / n
omega = 0.56 * beta**3 - 0.95 * beta**2 + 1.82 * beta + 1.43
cutoff_unknown = omega * sigma_med

plt.semilogy(s, "k.-", label="singular values")
plt.axhline(s[r90], color="g", label="90% of cumulative sum")
plt.axhline(cutoff, color="b", label="optimal hard threshold")
plt.axhline(
    cutoff_unknown,
    color="r",
    linestyle="--",
    label="optimal hard threshold, unknown noise",
)

plt.legend()
# SOLUTION-END
```

```
<matplotlib.legend.Legend at 0x7f97757ad2d0>
```



Compute the de-noised images with the previous thresholds

```
# SOLUTION-BEGIN
def denoise(U, s, VT, threshold):
  r = np.max(np.where(s > threshold))
  return U[:, : (r + 1)] @ np.diag(s[: (r + 1)]) @ VT[: (r + 1), :]

Xclean = denoise(U, s, VT, cutoff)
X90 = denoise(U, s, VT, cutoff90)
# SOLUTION-END
```

Plot and compare the denoised images. Plot also the error with respect to the original image.

```
# SOLUTION-BEGIN
err_clean = np.abs(X - Xclean)
err_90 = np.abs(X - X90)
err_norm_clean = np.linalg.norm(err_clean, ord="fro")
err_norm_90 = np.linalg.norm(err_90, ord="fro")

fig, axs = plt.subplots(2, 2, figsize=(10, 10))
axs[0, 0].imshow(Xclean, cmap="gray")
axs[0, 0].set_title("Cleaned Image: Hard threshold")
axs[0, 1].imshow(X90, cmap="gray")
axs[0, 1].set_title("Cleaned Image: 90% cumsum")

max_err = max([err_clean.max(), err_90.max()])
axs[1, 0].imshow(err_clean, cmap="gray", vmin=0, vmax=max_err)
axs[1, 0].set_title(f"Error: ({err_norm_clean:.2e})")
axs[1, 1].imshow(err_90, cmap="gray", vmin=0, vmax=max_err)
axs[1, 1].set_title(f"Error: ({err_norm_90:.2e})")
# SOLUTION-END
```

```
Text(0.5, 1.0, 'Error: (4.26e+01)')
```

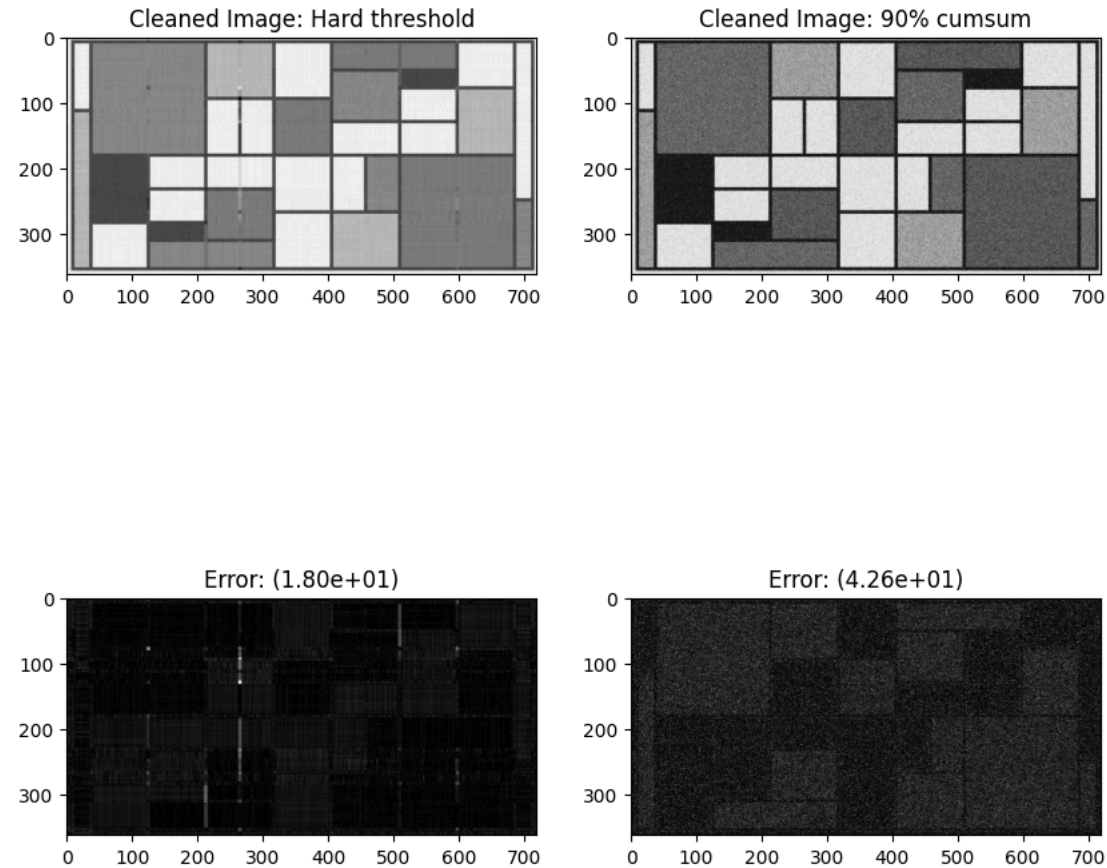


Image compression through Singular Value Decomposition

Load a picture as a 3-dimensional `np.array`

```
from matplotlib.image import imread
import matplotlib.pyplot as plt
import numpy as np
from scipy.ndimage import rotate

# write here the import path of the image
# test this with
# - TarantulaNebula
# - mondrian
# - mondrian (rotated)
image_path = "./TarantulaNebula.jpg"

A = imread(image_path)
# A = rotate(A, 20, reshape=False)
```

Visualize the picture

```
img = plt.imshow(A)
plt.axis("off")
```

```
(-0.5, 629.5, 566.5, -0.5)
```

Convert the picture to gray-scale and visualize it.

```
X = np.mean(A, axis=2)
```

```
img = plt.imshow(X)
plt.axis("off")
img.set_cmap("gray")
plt.show()
```

What is the picture size?

```
X.shape
```

```
(567, 630)
```

Perform the SVD decomposition

```
# SOLUTION-BEGIN
U, s, VT = np.linalg.svd(X, full_matrices=False)

U.shape, s.shape, VT.shape
# SOLUTION-END
```

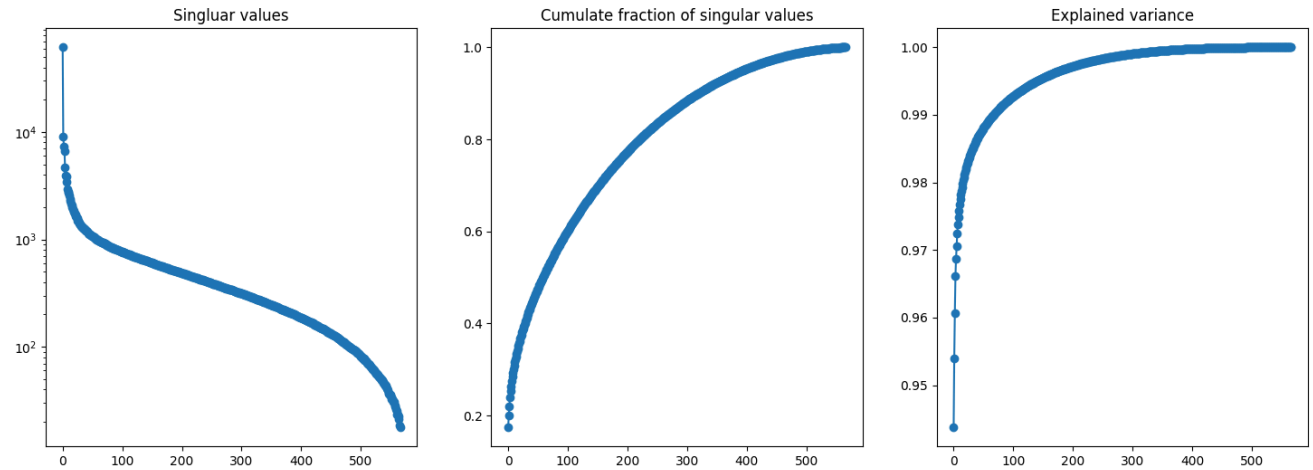
```
((567, 567), (567,), (567, 630))
```

Plot the trend of

- the singular values $\sigma_k$$\sigma_k$
- the cumulate fraction of singular values: $\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{q} \sigma_i}$$\frac{\sum_{i=1}^{k} \sigma_i}{\sum_{i=1}^{q} \sigma_i}$
- the fraction of the "explained variance": $\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{q} \sigma_i^2}$$\frac{\sum_{i=1}^{k} \sigma_i^2}{\sum_{i=1}^{q} \sigma_i^2}$

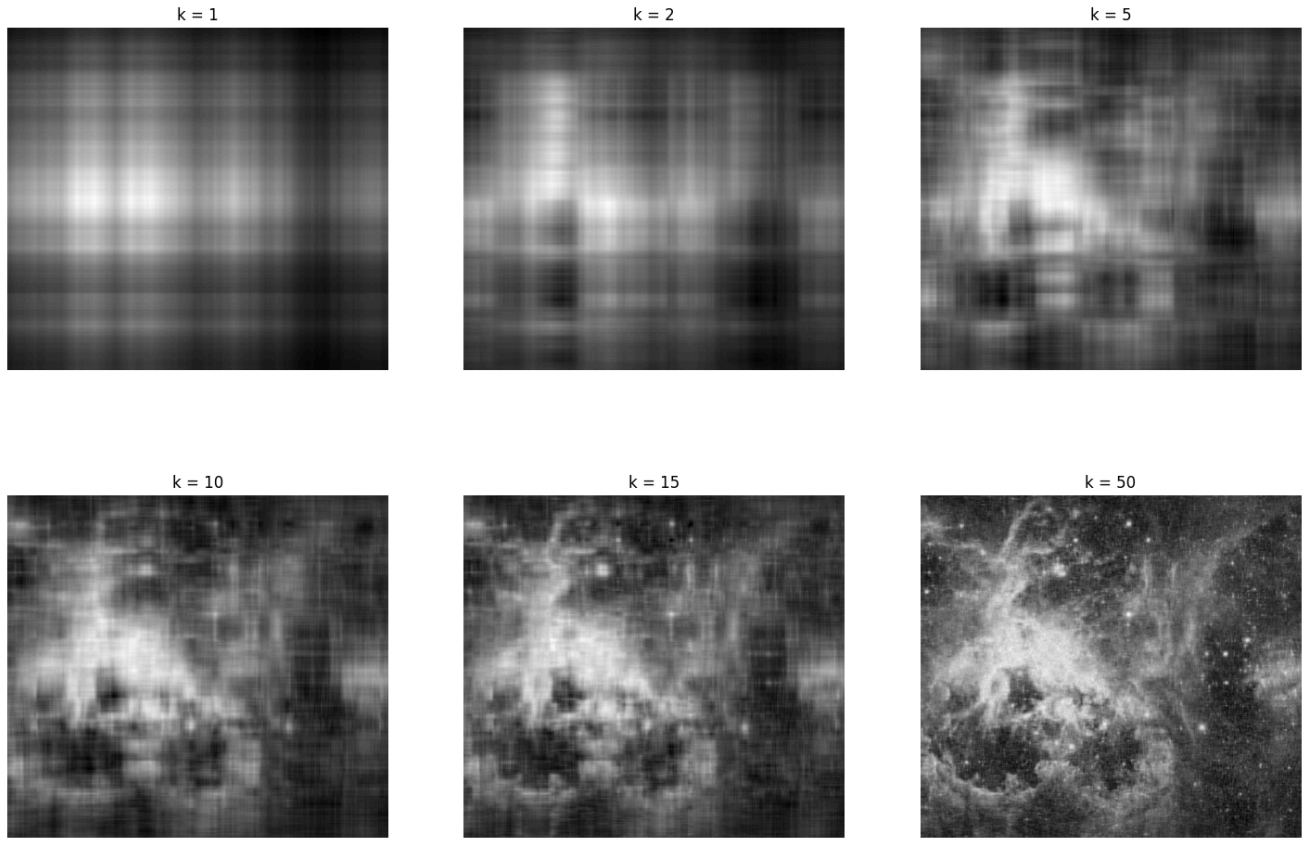
```
fig, axes = plt.subplots(nrows=1, ncols=3, figsize=(18, 6))
axes[0].semilogy(s, "o-")
axes[0].set_title("Singluar values")

axes[1].plot(np.cumsum(s) / np.sum(s), "o-")
axes[1].set_title("Cumulate fraction of singular values")

axes[2].plot(np.sqrt(np.cumsum(s**2) / np.sum(s**2)), "o-")
axes[2].set_title("Explained variance")
```

```
Text(0.5, 1.0, 'Explained variance')
```



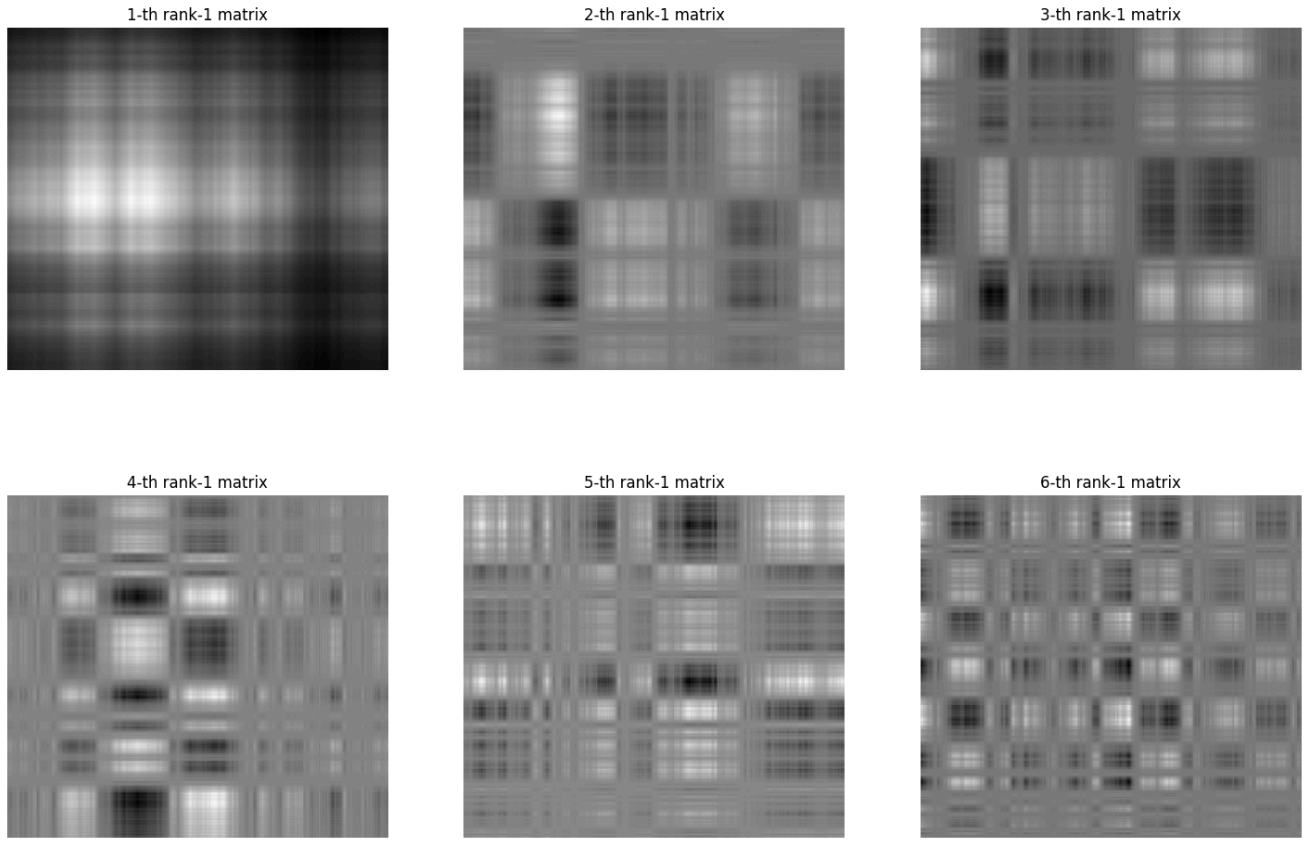
Visualize the best rank-$k$$k$ matrices ($A_k$$A_k$), for $k$$k$ = 1, 5, 10, 50, 100, 500

```
fig, axs = plt.subplots(nrows=2, ncols=3, figsize=(18, 12))
axs = axs.reshape((-1,))
idxs = [1, 2, 5, 10, 15, 50]
for i in range(len(idxs)):
    k = idxs[i]
    # SOLUTION-BEGIN
    A_k = np.matmul(U[:, :k], np.matmul(np.diag(s[:k]), VT[:k, :]))
    axs[i].imshow(A_k, cmap="gray")
    # SOLUTION-END
    axs[i].set_title(f"k = {k}")
    axs[i].axis("off")
```



Visualize the $k$$k$-th rank-1 matix, for $k$$k$ = 1,2,...,6.

```
fig, axs = plt.subplots(nrows=2, ncols=3, figsize=(18, 12))
axs = axs.reshape((-1,))
idxs = [1, 2, 3, 4, 5, 6]
for i, k in enumerate(idxs):
    # SOLUTION-BEGIN
    ukvk = np.outer(U[:, k - 1], VT[k - 1, :])
    axs[i].imshow(ukvk, cmap="gray")
    # SOLUTION-END
    axs[i].set_title(f"{k}-th rank-1 matrix")
    axs[i].axis("off")
```



Randomized SVD

Implement now a function that computes the randomized SVD of rank $k$$k$ of a generic matrix $A$$A$.

```
def randomized_SVD(A, k):
    # SOLUTION-BEGIN
    _, n = A.shape
    P = np.random.randn(n, k)
    Z = A @ P
    Q, _ = np.linalg.qr(Z)
    Y = Q.T @ A
    Uy, sy, VTy = np.linalg.svd(Y, full_matrices=False)
    U = Q @ Uy
    # SOLUTION-END
    return U, sy, VTy
```

Set $k=100$$k=100$ and compute the randomized SVD of the picture used above.

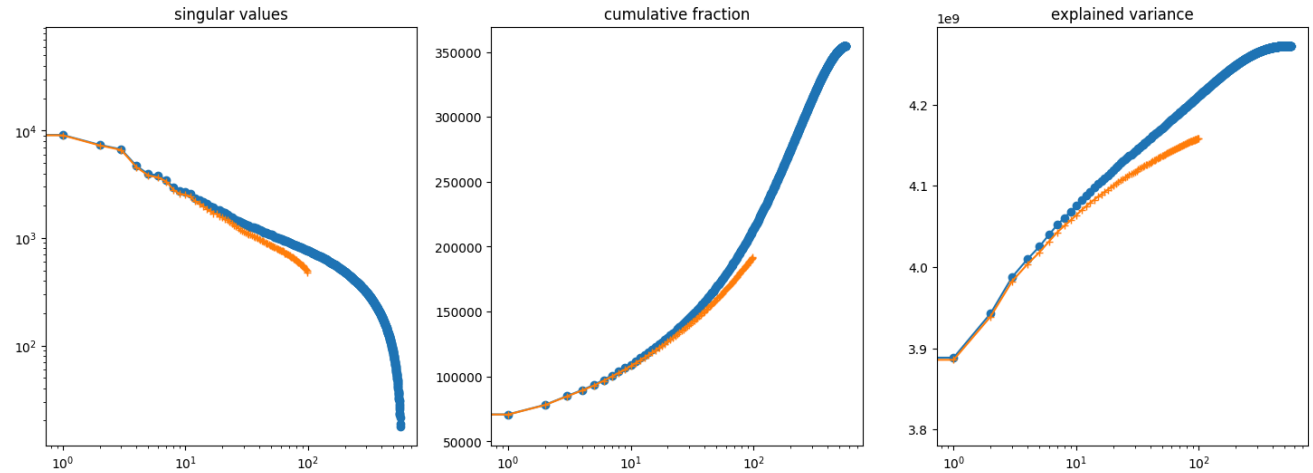
```
k = 100
# SOLUTION-BEGIN
U_rand, s_rand, VT_rand = randomized_SVD(X, k)
# SOLUTION-END
```

Plot the approximate singular values, their cumulate values and their cumulate squares, with a comparison of the ones obtained with the "exact" (i.e. non randomized) SVD.

```
fig, axs = plt.subplots(1, 3, figsize=(18, 6))
# SOLUTION-BEGIN
axs[0].loglog(s, "o-")
axs[0].loglog(s_rand, "+-")
axs[0].set_title("singular values")

axs[1].semilogx(np.cumsum(s), "o-")
axs[1].semilogx(np.cumsum(s_rand), "+-")
axs[1].set_title("cumulative fraction")
```

```
axs[2].semilogx(np.cumsum(s**2), "o-")
axs[2].semilogx(np.cumsum(s_rand**2), "+-")
axs[2].set_title("explained variance")
# SOLUTION-END
```

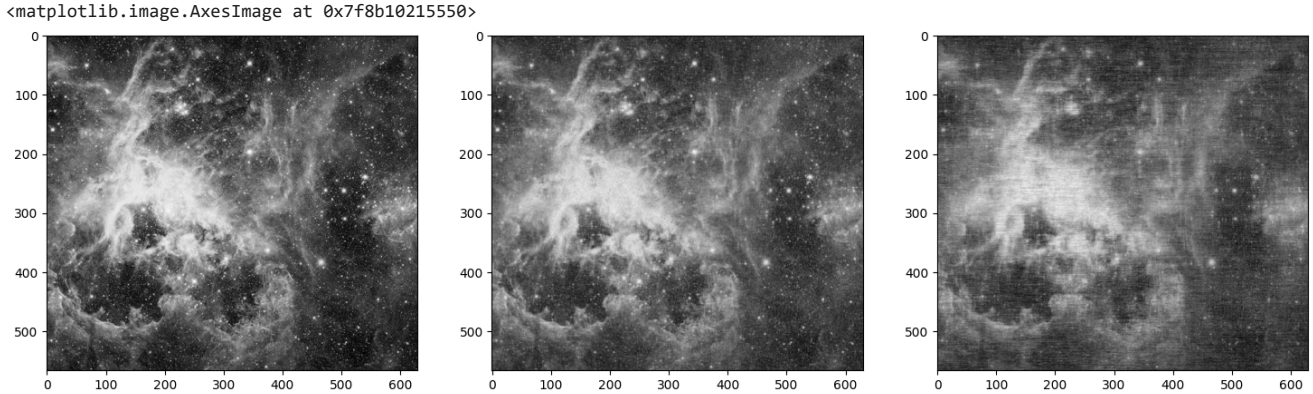
```
Text(0.5, 1.0, 'explained variance')
```



Compare the original picture with the best rank-$k$$k$ matrix obtained through SVD and the best rank-$k$$k$ matrix obtained through randomized SVD.

```
fig, axs = plt.subplots(1, 3, figsize=(18, 6))
# SOLUTION-BEGIN
axs[0].imshow(X, cmap="gray")
axs[1].imshow(U[:, :k] @ np.diag(s[:k]) @ VT[:k, :], cmap="gray")
axs[2].imshow(U_rand @ np.diag(s_rand) @ VT_rand, cmap="gray")
# SOLUTION-END
```

```
<matplotlib.image.AxesImage at 0x7f8b10215550>
```



SVD decomposition with `scipy`

Import the required modules

```
import scipy.linalg as la
import numpy as np
```

Generate a random 5x4 matrix

```
np.random.seed(0)

A = np.random.rand(5,4)
A
```

```
array([[0.5488135 , 0.71518937, 0.60276338, 0.54488318],
       [0.4236548 , 0.64589411, 0.43758721, 0.891773  ],
       [0.96366276, 0.38344152, 0.79172504, 0.52889492],
       [0.56804456, 0.92559664, 0.07103606, 0.0871293 ],
       [0.0202184 , 0.83261985, 0.77815675, 0.87001215]])
```

`<numpy|scipy>.linalg.svd`

Equivalent implementations:

- https://numpy.org/doc/stable/reference/generated/numpy.linalg.svd.html
- https://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.svd.html

Full SVD (`full_matrices = True` - default)

input:

$A \in \mathbb{R}^{m\times n}$$A \in \mathbb{R}^{m\times n}$

output:

$U \in \mathbb{R}^{m\times m}, S \in \mathbb{R}^{m\times n}, V \in \mathbb{R}^{n\times n}$$U \in \mathbb{R}^{m\times m}, S \in \mathbb{R}^{m\times n}, V \in \mathbb{R}^{n\times n}$

or, more precisely

$\boldsymbol{\sigma} \in \mathbb{R}^{q} $$\boldsymbol{\sigma} \in \mathbb{R}^{q} $

where $q = \min(m,n)$$q = \min(m,n)$.

```
U, s, VT = np.linalg.svd(A)
#U, s, VT = la.svd(A)
print('U shape: ', U.shape)
print('s shape: ', s.shape)
print('VT shape: ', VT.shape)
```

```
U shape:  (5, 5)
s shape:  (4,)
VT shape:  (4, 4)
```

Build the matrix $S$$S$

```
S = np.zeros(A.shape)
for i in range(len(s)):
    S[i, i] = s[i]
S
```

```
array([[2.64618677, 0.        , 0.        , 0.        ],
       [0.        , 0.83351254, 0.        , 0.        ],
       [0.        , 0.        , 0.70753001, 0.        ],
       [0.        , 0.        , 0.        , 0.29842614],
       [0.        , 0.        , 0.        , 0.        ]])
```

```
S = la.diagsvd(s, A.shape[0], A.shape[1])
S
```

```
array([[2.64618677, 0.        , 0.        , 0.        ],
       [0.        , 0.83351254, 0.        , 0.        ],
       [0.        , 0.        , 0.70753001, 0.        ],
       [0.        , 0.        , 0.        , 0.29842614],
       [0.        , 0.        , 0.        , 0.        ]])
```

Reconstruct the matrix $A$$A$

```
A_svd = np.matmul(U, np.matmul(S,VT))
# equivalently: A_svd = U @ S @ VT
print(f"err: {(la.norm(A - A_svd) / la.norm(A))}")
```

```
err: 4.624308861561109e-16
```

Thin SVD (`full_matrices = False`)

input:

$A \in \mathbb{R}^{m\times n}$$A \in \mathbb{R}^{m\times n}$

output:

$U \in \mathbb{R}^{m\times q}, S \in \mathbb{R}^{q\times q}, V \in \mathbb{R}^{n\times q}$$U \in \mathbb{R}^{m\times q}, S \in \mathbb{R}^{q\times q}, V \in \mathbb{R}^{n\times q}$

or, more precisely

$\boldsymbol{\sigma} \in \mathbb{R}^{q} $$\boldsymbol{\sigma} \in \mathbb{R}^{q} $

where $q = \min(m,n)$$q = \min(m,n)$.

```
U, s, VT = la.svd(A, full_matrices=False)
print('U shape: ', U.shape)
print('s shape: ', s.shape)
print('VT shape: ', VT.shape)
```

```
U shape:  (5, 4)
s shape:  (4,)
VT shape:  (4, 4)
```

Build the matrix $S$$S$

```
S = np.diag(s)
S
```

```
array([[2.64618677, 0.        , 0.        , 0.        ],
       [0.        , 0.83351254, 0.        , 0.        ],
       [0.        , 0.        , 0.70753001, 0.        ],
       [0.        , 0.        , 0.        , 0.29842614]])
```

Reconstruct the matrix $A$$A$

```
A_svd = np.matmul(U, np.matmul(S,VT))
print(f"err: {la.norm(A - A_svd) / la.norm(A)}")
```

```
err: 4.624308861561109e-16
```

A note on vectorization

Vectorization refers to the practice of replacing explicit loops with high-level mathematical operations that act on entire arrays or matrices at once. This leads to much better performance because it replaces slow Python loops with fast, optimized C and Fortran operations.

Indeed, we could be inclined to reconstruct $A_k$$A_k$ with a for loop and the explicit formula $$A_k = \sigma_1 u_1 v_1^T + ... + \sigma_k u_k v_k^T.$$$$A_k = \sigma_1 u_1 v_1^T + ... + \sigma_k u_k v_k^T.$$

Let's measure the time taken for this operation for a matrix $A$$A$ that is a bit larger

```
import time

A = np.random.rand(1000, 1500)
U, s, VT = la.svd(A, full_matrices=False)
S = np.diag(s)
```

Time the reconstruction with a for loop

```
start_time = time.time()

A_reconstructed_loop = np.zeros_like(A)
for i in range(len(s)):
    A_reconstructed_loop += s[i] * np.outer(U[:, i], VT[i, :])

loop_time = time.time() - start_time
```

Time the vectorized reconstruction using matrix multiplication

```
start_time = time.time()

A_reconstructed_matmult = U @ S @ VT

matmult_time = time.time() - start_time
```

If $S \in \mathbb R^{q \times q}$$S \in \mathbb R^{q \times q}$ diagonal (with elements on the diagonal equal to $s \in \mathbb R^{q}$$s \in \mathbb R^{q}$), then the each of $US$$US$ is equal to the element wise product of that row of $U$$U$ with $s$$s$, namely, $(US)_{i,j} = U_{i, j} s_j$$(US)_{i,j} = U_{i, j} s_j$. Indeed, $(US)_{i,j} = \sum_{k} U_{i, k} S_{k, j} = U_{i, j} S_{j, j} = U_{i, j} s_j$$(US)_{i,j} = \sum_{k} U_{i, k} S_{k, j} = U_{i, j} S_{j, j} = U_{i, j} s_j$, since $S_{k, j} = 0$$S_{k, j} = 0$ iff $k \neq j$$k \neq j$.

```
start_time = time.time()

# here we are using broadcasting to avoid the creation of a diagonal matrix
# see: https://numpy.org/doc/stable/user/basics.broadcasting.html
A_reconstructed_vectorized = (U * s) @ VT

vectorized_time = time.time() - start_time
```

We compare the results

```
print(f"Time for reconstruction using for loop: {loop_time:.6f} seconds")
print(f"Time for vectorized reconstruction: {vectorized_time:.6f} seconds")
print(f"Matmult is {loop_time / matmult_time:.1f} times faster than the loop")
print(f"Vectorized is {matmult_time / vectorized_time:.1f} times faster than the matmult")

difference = np.abs(A_reconstructed_loop - A_reconstructed_vectorized).max()
print(f"Difference between the two reconstructions: {difference:.6e}")
```

```
Time for reconstruction using for loop: 2.981118 seconds
Time for vectorized reconstruction: 0.033559 seconds
Matmult is 81.4 times faster than the loop
Vectorized is 1.1 times faster than the matmult
Difference between the two reconstructions: 5.329071e-15
```

Background Removal

We aim to remove the background from a video and be able to select only the moving parts using SVD.

We use as benchmark the real video 003 dataset from BMC 2012 [Background Models Challenge Dataset]

```
!wget http://backgroundmodelschallenge.eu/data/real/Video_003.zip
!unzip Video_003.zip
```

```
--2025-10-03 15:57:22--  http://backgroundmodelschallenge.eu/data/real/Video_003.zip
Resolving backgroundmodelschallenge.eu (backgroundmodelschallenge.eu)... 145.239.37.162, 2001:41d0:301::30
Connecting to backgroundmodelschallenge.eu (backgroundmodelschallenge.eu)|145.239.37.162|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 3279920 (3.1M) [application/zip]
Saving to: ‘Video_003.zip.1’

Video_003.zip.1     100%[===================>]   3.13M  6.00MB/s    in 0.5s

2025-10-03 15:57:23 (6.00 MB/s) - ‘Video_003.zip.1’ saved [3279920/3279920]

Archive:  Video_003.zip
   creating: Video_003/
   creating: Video_003/img/
  inflating: Video_003/img/Img_00250.bmp
  inflating: Video_003/img/Img_00251.bmp
```

```
  inflating: Video_003/img/Img_00252.bmp
  inflating: Video_003/img/Img_00350.bmp
  inflating: Video_003/img/Img_00351.bmp
  inflating: Video_003/img/Img_00352.bmp
  inflating: Video_003/img/Img_00450.bmp
  inflating: Video_003/img/Img_00451.bmp
  inflating: Video_003/img/Img_00452.bmp
  inflating: Video_003/img/Img_00550.bmp
  inflating: Video_003/img/Img_00551.bmp
  inflating: Video_003/img/Img_00552.bmp
  inflating: Video_003/img/Img_00650.bmp
  inflating: Video_003/img/Img_00651.bmp
  inflating: Video_003/img/Img_00652.bmp
  inflating: Video_003/img/Mask_00250.bmp
  inflating: Video_003/img/Mask_00251.bmp
  inflating: Video_003/img/Mask_00252.bmp
  inflating: Video_003/img/Mask_00350.bmp
  inflating: Video_003/img/Mask_00351.bmp
  inflating: Video_003/img/Mask_00352.bmp
  inflating: Video_003/img/Mask_00450.bmp
  inflating: Video_003/img/Mask_00451.bmp
  inflating: Video_003/img/Mask_00452.bmp
  inflating: Video_003/img/Mask_00550.bmp
  inflating: Video_003/img/Mask_00551.bmp
  inflating: Video_003/img/Mask_00552.bmp
  inflating: Video_003/img/Mask_00650.bmp
  inflating: Video_003/img/Mask_00651.bmp
  inflating: Video_003/img/Mask_00652.bmp
  inflating: Video_003/Video_003.avi
  inflating: Video_003/Video_003.xml
```

Import needed libraries:

```
import moviepy.editor as mpe
import numpy as np
import skimage.transform

%matplotlib inline
import matplotlib.pyplot as plt
```

Load the video and check its duration

```
video = mpe.VideoFileClip("Video_003/Video_003.avi")
video.duration
```

```
113.57
```

Let us visualize the video, and actually check its contents

```
video.subclip(0, 5).ipython_display(width=300)
```

```
Moviepy - Building video __temp__.mp4.
Moviepy - Writing video __temp__.mp4

                                                    Moviepy - Done !
Moviepy - video ready __temp__.mp4
```

0:00 / 0:05

Helper Methods

In order to work with the video, we need to implement a couple of helper methods:

- a method to tranform a color image into a gray image (SVD works with 2D matrices, thus, for sake of simplicity, we avoid repeating the same operations to each channel of the image by using a gray-scale image)
- a method to transform the video into a numpy matrix, we also downscale the video to reduce the computational cost of applying the SVD. We can unroll a frame into a single tall column. So instead of having a 2D picture, we have a column. This isn't very human-readable, but it's handy because it lets us stack the images from different times on top of one another, to put a video all into 1 matrix.

```
def create_data_matrix_from_video(clip, dims):
    number_of_frames = int(clip.fps * clip.duration)
    flatten_gray_frames = []
    for i in range(number_of_frames):
        # get_frame takes as input the time of the frame
        frame = clip.get_frame(i / clip.fps)
        # to gray scale
        gray_frame = np.mean(frame[..., :3], axis=-1).astype(int)
        # resize to reduce computational cost
        small_gray_frame = skimage.transform.resize(gray_frame, dims)
        # each frame becomes a column vector of A
        flatten_gray_frames.append(small_gray_frame.flatten())
    return np.vstack(flatten_gray_frames).T
```

We set some parameters and create the matrix

```
scale = 0.50  # Adjust scale to change resolution of image
width, height = video.size
dims = (int(height * scale), int(width * scale))
A = create_data_matrix_from_video(video, dims)
print("frame size:", dims)
print("video matrix size:", A.shape)
```

```
frame size: (120, 160)
video matrix size: (19200, 794)
```

Let's visualize the 140-th frame of the video (rescaled and decolored)

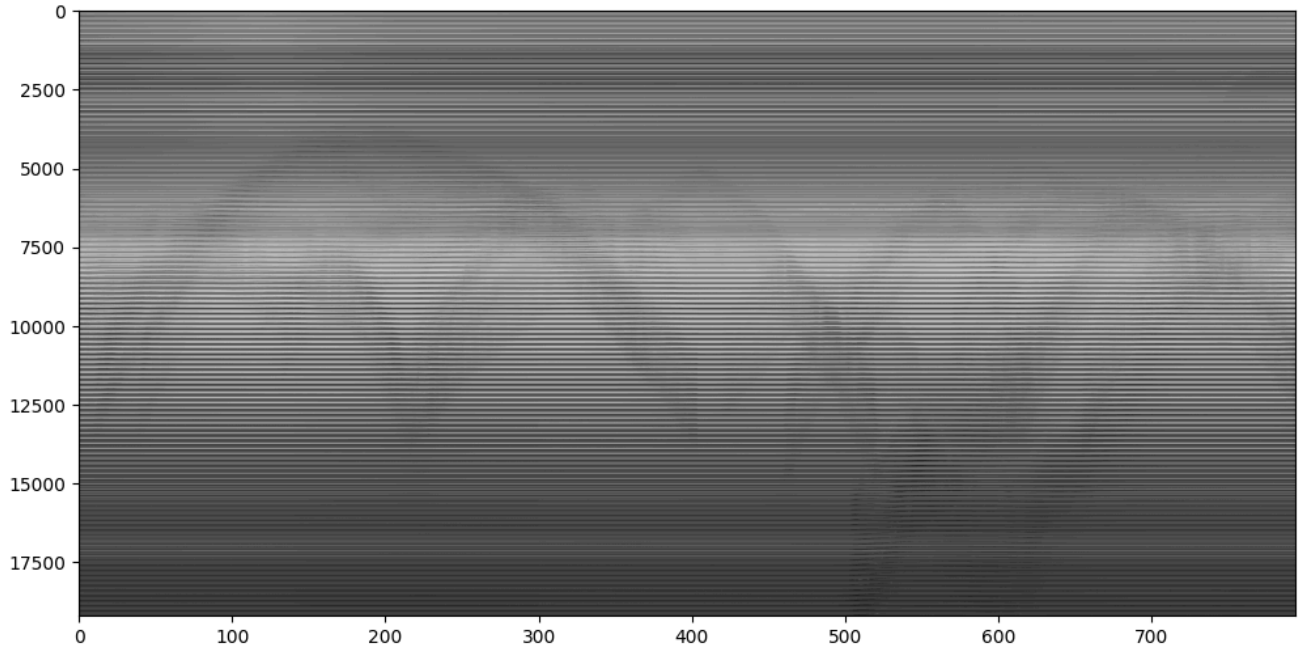
```
plt.imshow(np.reshape(A[:, 140], dims), cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7faae54fbf90>
```

What does the matrix A look like instead?

```
plt.figure(figsize=(12, 6))
plt.imshow(A, cmap="gray", aspect="auto")
```

```
<matplotlib.image.AxesImage at 0x7faae52965d0>
```



Questions: What are those wavy black lines? What are the horizontal lines?

Apply the SVD

We apply the SVD to our matrix $A$$A$

```
import time


def my_svd(XX, k):
    # HOMEWORK: replace here the svd with rSVD and comment the results
    return np.linalg.svd(XX, full_matrices=False)


t0 = time.time()
U, s, VT = my_svd(A, 10)
print(f"SVD elapsed {time.time() - t0: .2f} [s]")


U.shape, s.shape, VT.shape
```

```
SVD elapsed  1.73 [s]
((19200, 794), (794,), (794, 794))
```

Check that they are a decomposition of A

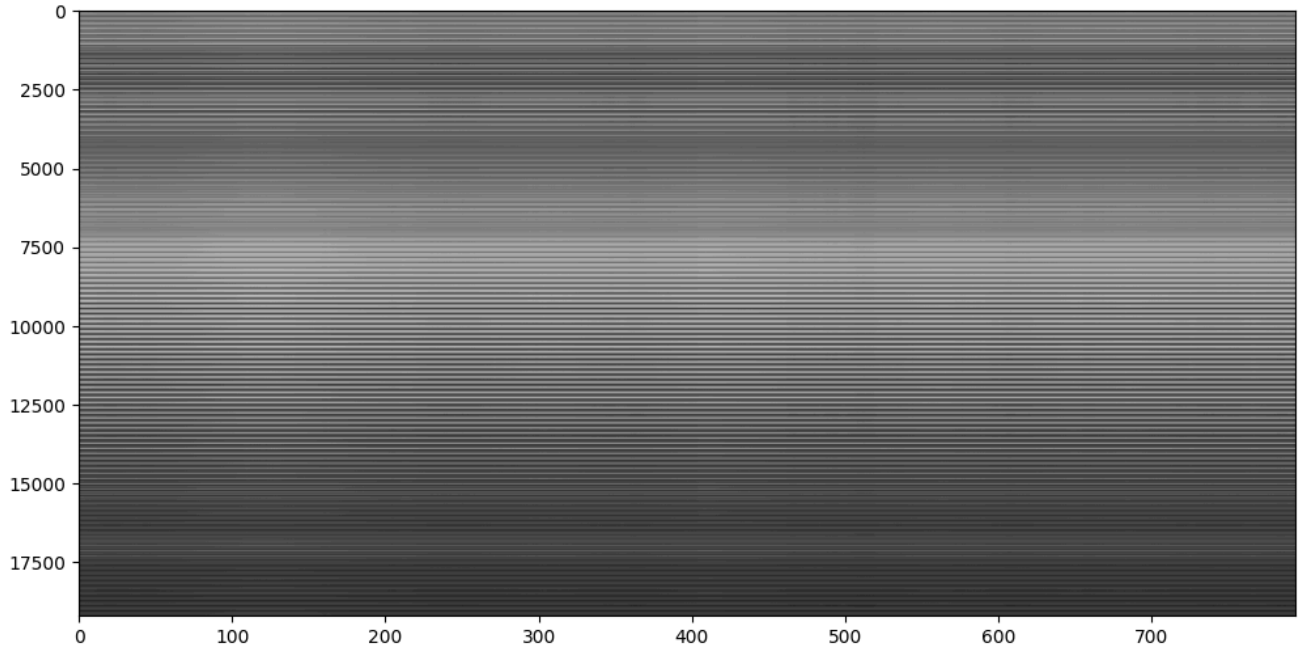
```
reconstructed_A = U @ np.diag(s) @ VT
np.allclose(A, reconstructed_A)
```

```
True
```

Since the background is more or less constant during the video (the horizontal lines of A), we suppose that the background can be reconstructed using few singular values

```
n_singular_values = 1
background = U[:, 0:n_singular_values] * s[0] * VT[0:n_singular_values, :]
plt.figure(figsize=(12, 6))
plt.imshow(background, cmap="gray", aspect="auto")
```

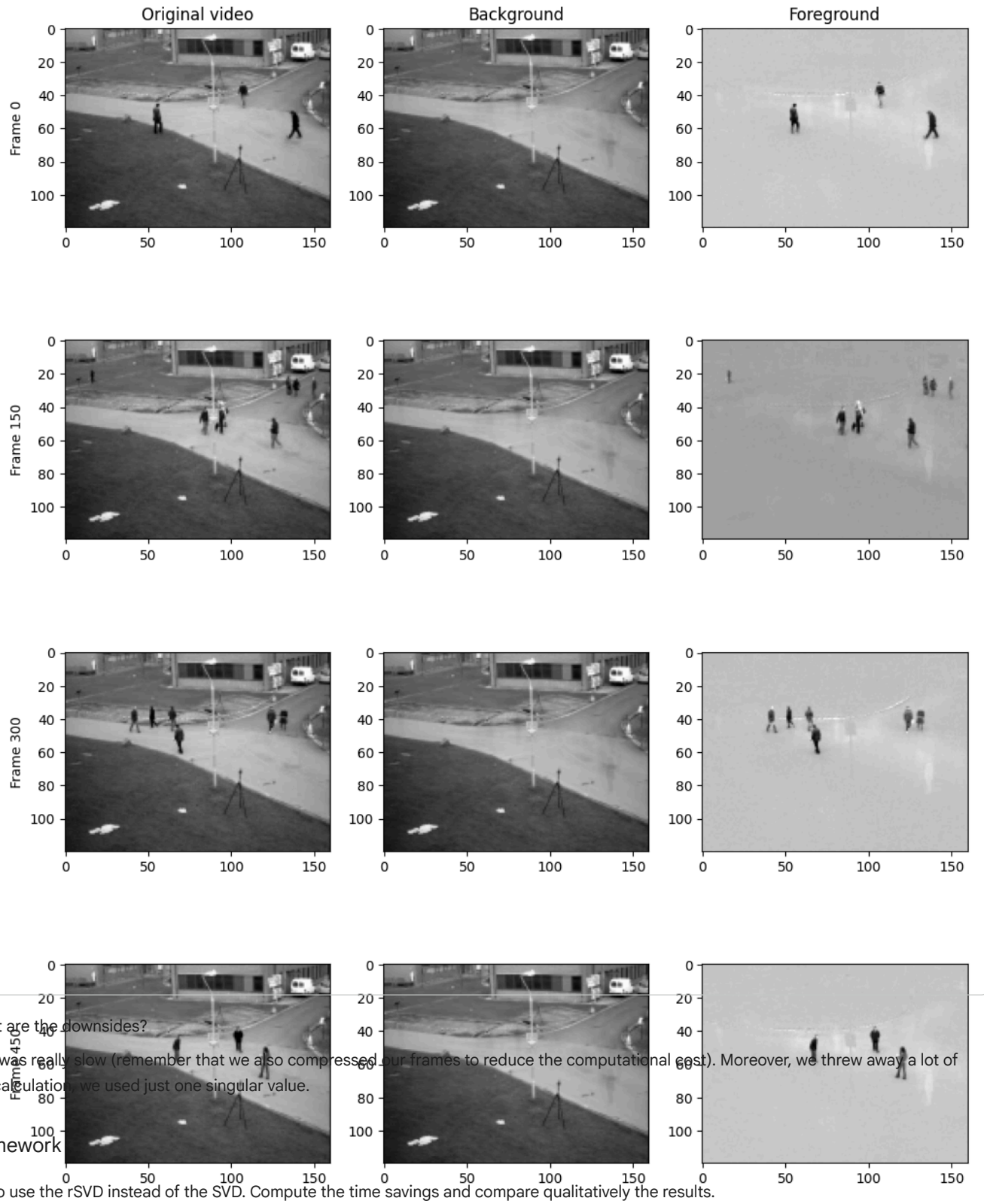
```
<matplotlib.image.AxesImage at 0x7faae5120350>
```



The SVD has done all the heavy liflting! Now we "just" have to visualize the results in a human comprehensible way.

Let's visualize the original video, the background and the foreground at some fixed instances in time.

```
def plot_frames(A, background, time_ids):
    fig, axs = plt.subplots(len(time_ids), 3, figsize=(12, 4 * len(time_ids)))
    for i, t_id in enumerate(time_ids):
        axs[i, 0].imshow(np.reshape(A[:, t_id], dims), cmap="gray")
        axs[i, 1].imshow(np.reshape(background[:, t_id], dims), cmap="gray")
        axs[i, 2].imshow(
            np.reshape(A[:, t_id] - background[:, t_id], dims), cmap="gray"
        )

        axs[i, 0].set_ylabel(f"Frame {t_id}")

        if i == 0:
            axs[0, 0].set_title("Original video")
            axs[0, 1].set_title("Background")
            axs[0, 2].set_title("Foreground")


time_ids = [0, 150, 300, 450]
plot_frames(A, background, time_ids)
```



What are the downsides?

SVD was really slow (remember that we also compressed our frames to reduce the computational cost). Moreover, we threw away a lot of our calculation, we used just one singular value.

Homework

Try to use the rSVD instead of the SVD. Compute the time savings and compare qualitatively the results.

1st order Training (optimization) methods

```
import numpy as np
import matplotlib.pyplot as plt
import time
import jax.numpy as jnp
import jax
```

Let us consider the following function

$$ f(x) = e^{-\frac{x}{10}}\sin(x) + \frac{1}{10} \cos(\pi x) $$$$ f(x) = e^{-\frac{x}{10}}\sin(x) + \frac{1}{10} \cos(\pi x) $$

defined over the interval $[0, 10]$$[0, 10]$.

```
f = lambda x: np.sin(x) * np.exp(-0.1 * x) + 0.1 * np.cos(np.pi * x)
a, b = 0, 10
```

Define a function `get_training_data` that returns a collection of `N` training samples, adding a noise sampled from a normal distribution with zero mean and standard deviation `noise`. The output should be a pair of `numpy` matrices `x, y`, of dimension `N` times 1.

```
def get_training_data(N, noise):
    np.random.seed(0)  # for reproducibility
    x = np.linspace(a, b, N)[:, None]
    y = f(x) + noise * np.random.randn(N, 1)
    return x, y
```

Plot now the function $f(x)$$f(x)$ in the considered interval, together with 100 training samples with noise magnitude equal to 0.05.

```
x_fine = np.linspace(a, b, 1000)[:, None]
plt.plot(x_fine, f(x_fine))

xx, yy = get_training_data(100, 0.05)
plt.plot(xx, yy, "o")
```

Write a function `initialize_params` that, given the input `layers_size = [n1, n2, ..., nL]`, generates the parameters associated with an ANN, having as many layers as the number of elements of `layers_size`, with as many neurons as `n1`, `n2`, etc.

Inizialize weights sampling from a standard Gaussian distribution and biases with zero values.

```
def initialize_params(layers_size):
    np.random.seed(0)  # for reproducibility
    params = list()
    for i in range(len(layers_size) - 1):
        W = np.random.randn(layers_size[i + 1], layers_size[i])
        b = np.zeros((layers_size[i + 1], 1))
        params.append(W)
        params.append(b)
    return params
```

Write a function `ANN` that implements an ANN, given the parameters `params`. Use $\tanh$$\tanh$ as activation function and do not apply the activation function to the last layer.

By convention, both the input and the output have:

- 1 sample per row
- 1 feature per column

```
def ANN(x, params):
    # layer = x.T
    layer = (2 * x.T - (a + b)) / (b - a)
    num_layers = int(len(params) / 2 + 1)
    weights = params[0::2]
    biases = params[1::2]
    for i in range(num_layers - 1):
        layer = jnp.dot(weights[i], layer) - biases[i]
        if i < num_layers - 2:
            layer = jnp.tanh(layer)
    return layer.T
```

Implement a function `loss`, that, given the input `x`, the target output `y` (i.e. the "labels") and the parameters `params`, returns the quadratic loss, defined as:

$$ \mathcal{L}(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m (y_i - \mathrm{ANN}(x_i, \boldsymbol{\theta}))^2 $$$$ \mathcal{L}(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m (y_i - \mathrm{ANN}(x_i,

\boldsymbol{\theta}))^2 $$

where $m$$m$ is the number of samples in `x`, `y` and $\boldsymbol{\theta}$$\boldsymbol{\theta}$ are the ANN parameters.

```
def loss(x, y, params):
    error = ANN(x, params) - y
    return jnp.mean(error * error)
```

Test your code, by generating the parameters associated with an ANN with two hidden layers with 5 neurons each and by computing the associated loss.

```
params = initialize_params([1, 5, 5, 1])
loss(xx, yy, params)
```

The following cell provides an helper class that allows pnline plots during the training loop. Just run the cell, we will use it later.

```
from IPython import display


class Callback:
    def __init__(self, refresh_rate=250):
        self.refresh_rate = refresh_rate
        self.fig, self.axs = plt.subplots(1, 2, figsize=(16, 8))
        self.x_fine = np.linspace(a, b, 200)[:, None]
        self.epoch = 0
        self.__call__(-1)

    def __call__(self, epoch):
        self.epoch = epoch
        if (epoch + 1) % self.refresh_rate == 0:
            self.draw()
            display.clear_output(wait=True)
            display.display(plt.gcf())
            time.sleep(1e-16)

    def draw(self):
        if self.epoch > 0:
            self.axs[0].clear()
            self.axs[0].loglog(history)
            self.axs[0].set_title("epoch %d" % (self.epoch + 1))

        self.axs[1].clear()
        self.axs[1].plot(self.x_fine, f(self.x_fine))
        self.axs[1].plot(self.x_fine, ANN(self.x_fine, params))
        self.axs[1].plot(xx, yy, "o")
```

Gradient Descent

Implement the GD method:

$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{N} \sum_{i=1}^N \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda \mathbf{g}^{(k)} \end{split} $$$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{N} \sum_{i=1}^N \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda \mathbf{g}^{(k)} \end{split} $$

where N is the number of training samples. At each iteration, append the current cost to the list `history`.

Hints:

- Use `jax.jit` to speedup the evaluation of the loss and of the gradients.
- To us the visualization callback, just initialize it outside the training loop with:

```
cb = Callback(refresh_rate = 250)
```

and after each training epoch call `cb(epoch)`, where `epoch` is the epoch index. Finally, call `cb.draw()` when training is over.

Test you code with:

- 100 training points
- noise magnitude 0.05
- two hidden layers of 5 neurons each
- 2000 epochs

Experiment different choices of learning rate $\lambda$$\lambda$, trying to maximize the performance of the algorithm.

Does it look like your ANN is struggling to learn? Try to figure out why and figure out how to fix this issue.

```
# Data
n_training_points = 100
noise = 0.05
# Hyperparameters
layers_size = [1, 5, 5, 1]
# Training options
num_epochs = 2000
learning_rate = 1e-1
########################################

xx, yy = get_training_data(n_training_points, noise)
params = initialize_params(layers_size)

loss_jit = jax.jit(loss)
grad_jit = jax.jit(jax.grad(loss, argnums=2))

history = list()
history.append(loss_jit(xx, yy, params))

cb = Callback(refresh_rate=200)

for epoch in range(num_epochs):
    # SOLUTION-BEGIN
    grads = grad_jit(xx, yy, params)
    for i in range(len(params)):
        params[i] -= learning_rate * grads[i]
    # SOLUTION-END
    history.append(loss_jit(xx, yy, params))
    cb(epoch)
cb.draw()

print("loss: %1.3e" % history[-1])
```

Stochastic Gradient Descent

Implement the SGD method:

$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda_k \mathbf{g}^{(k)} \end{split} $$$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda_k \mathbf{g}^{(k)} \end{split} $$

where $I_k$$I_k$ is the current minibatch. To select it, use the function [np.random.choice](#) with replacement.

Consider a linear decay of the learning rate:

$$ \lambda_k = \max\left(\lambda_{\textnormal{min}}, \lambda_{\textnormal{max}} \left(1 - \frac{k}{K}\right)\right) $$$$ \lambda_k = \max\left(\lambda_{\textnormal{min}}, \lambda_{\textnormal{max}} \left(1 - \frac{k}{K}\right)\right) $$

Test different choices of batch size and try to optimize the learning rate decay strategy.

```
# Data
n_training_points = 100
noise = 0.05
# Hyperparameters
layers_size = [1, 5, 5, 1]
# Training options
num_epochs = 20000
learning_rate_max = 1e-1
learning_rate_min = 2e-2
learning_rate_decay = 10000
```

```
batch_size = 10
########################################

xx, yy = get_training_data(n_training_points, noise)
params = initialize_params(layers_size)

loss_jit = jax.jit(loss)
grad_jit = jax.jit(jax.grad(loss, argnums=2))

history = list()
history.append(loss_jit(xx, yy, params))

cb = Callback(refresh_rate=250)

for epoch in range(num_epochs):
    # SOLUTION-BEGIN
    learning_rate = max(
        learning_rate_min, learning_rate_max * (1 - epoch / learning_rate_decay)
    )
    idxs = np.random.choice(n_training_points, batch_size)
    grads = grad_jit(xx[idxs, :], yy[idxs, :], params)
    for i in range(len(params)):
        params[i] -= learning_rate * grads[i]
    # SOLUTION-END

    history.append(loss_jit(xx, yy, params))
    cb(epoch)
cb.draw()

print("loss: %1.3e" % history[-1])
```

Stochastic Gradient Descent with momentum

Implement the SGD method with momentum:

$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given}, \mathbf{v}^{(0)}=\mathbf{0} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \mathbf{v}^{(k+1)} = \alpha \mathbf{v}^{(k)} - \lambda_k \mathbf{g}^{(k)}\\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} + \mathbf{v}^{(k+1)} \end{split} $$$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given}, \mathbf{v}^{(0)}=\mathbf{0} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \mathbf{v}^{(k+1)} = \alpha \mathbf{v}^{(k)} - \lambda_k \mathbf{g}^{(k)}\\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} + \mathbf{v}^{(k+1)} \end{split} $$

Test different choices of $\alpha$$\alpha$.

```
# Data
n_training_points = 100
noise = 0.05
# Hyperparameters
layers_size = [1, 5, 5, 1]
# Training options
num_epochs = 20000
learning_rate_max = 1e-1
learning_rate_min = 1e-2
learning_rate_decay = 10000
batch_size = 10
alpha = 0.9
########################################

xx, yy = get_training_data(n_training_points, noise)
params = initialize_params(layers_size)

loss_jit = jax.jit(loss)
grad_jit = jax.jit(jax.grad(loss, argnums=2))

history = list()
history.append(loss_jit(xx, yy, params))

cb = Callback(refresh_rate=250)

velocity = [0.0 for _ in range(len(params))]
for epoch in range(num_epochs):
    # SOLUTION-BEGIN
```

```
        learning_rate = max(
            learning_rate_min, learning_rate_max * (1 - epoch / learning_rate_decay)
        )
        idxs = np.random.choice(n_training_points, batch_size)
        grads = grad_jit(xx[idxs, :], yy[idxs, :], params)

        for i in range(len(params)):
            velocity[i] = alpha * velocity[i] - learning_rate * grads[i]
            params[i] += velocity[i]
        # SOLUTION-END

        history.append(loss_jit(xx, yy, params))
        cb(epoch)
    cb.draw()

    print("loss: %1.3e" % history[-1])
```

AdaGrad

Implement the AdaGrad algorithm:

$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given}, \mathbf{r}^{(0)}=\mathbf{0} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} + \mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}\\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \frac{\lambda}{\delta + \sqrt{\mathbf{r}^{(k+1)}}} \odot \mathbf{g}^{(k)} \end{split} $$$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given}, \mathbf{r}^{(0)}=\mathbf{0} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \mathbf{r}^{(k+1)} = \mathbf{r}^{(k)} + \mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}\\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \frac{\lambda}{\delta + \sqrt{\mathbf{r}^{(k+1)}}} \odot \mathbf{g}^{(k)} \end{split} $$

where $\delta = 10^{-7}$$\delta = 10^{-7}$ is a small constant. Notice that the operations in the last line should be intepreted "componentwise". test different choices of the learning rate $\lambda$$\lambda$.

```
    # Data
    n_training_points = 100
    noise = 0.05
    # Hyperparameters
    layers_size = [1, 5, 5, 1]
    # Training options
    num_epochs = 20000
    batch_size = 10
    learning_rate = 1e-1
    delta = 1e-7
    ########################################

    xx, yy = get_training_data(n_training_points, noise)
    params = initialize_params(layers_size)

    loss_jit = jax.jit(loss)
    grad_jit = jax.jit(jax.grad(loss, argnums=2))

    history = list()
    history.append(loss_jit(xx, yy, params))

    cb = Callback(refresh_rate=250)

    cumulated_square_grad = [0.0 for i in range(len(params))]
    for epoch in range(num_epochs):
        # SOLUTION-BEGIN
        idxs = np.random.choice(n_training_points, batch_size)
        grads = grad_jit(xx[idxs, :], yy[idxs, :], params)

        for i in range(len(params)):
            cumulated_square_grad[i] += grads[i] * grads[i]
            params[i] -= (
                learning_rate / (delta + jnp.sqrt(cumulated_square_grad[i])) * grads[i]
            )
        # SOLUTION-END

        history.append(loss_jit(xx, yy, params))
        cb(epoch)
    cb.draw()
```

```
print("loss: %1.3e" % history[-1])
```

RMSProp

Implement the RMSProp algorithm:

$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given}, \mathbf{r}^{(0)}=\mathbf{0} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \mathbf{r}^{(k+1)} = \rho \mathbf{r}^{(k)} + (1 - \rho)\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}\\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \frac{\lambda}{\delta + \sqrt{\mathbf{r}^{(k+1)}}} \odot \mathbf{g}^{(k)} \end{split} $$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given}, \mathbf{r}^{(0)}=\mathbf{0} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \mathbf{r}^{(k+1)} = \rho \mathbf{r}^{(k)} + (1 - \rho)\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}\\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \frac{\lambda}{\delta + \sqrt{\mathbf{r}^{(k+1)}}} \odot \mathbf{g}^{(k)} \end{split} $$

where $\delta = 10^{-7}$$\delta = 10^{-7}$ is a small constant. test different choices of the learning rate $\lambda$$\lambda$ and decay rate $\rho$$\rho$.

```
# Data
n_training_points = 100
noise = 0.05
# Hyperparameters
layers_size = [1, 5, 5, 1]
# Training options
num_epochs = 20000
batch_size = 50
learning_rate = 1e-3
decay_rate = 0.9
delta = 1e-7
########################################

xx, yy = get_training_data(n_training_points, noise)
params = initialize_params(layers_size)

loss_jit = jax.jit(loss)
grad_jit = jax.jit(jax.grad(loss, argnums=2))

history = list()
history.append(loss_jit(xx, yy, params))

cb = Callback(refresh_rate=250)

cumulated_square_grad = [0.0 for i in range(len(params))]
for epoch in range(num_epochs):
    # SOLUTION-BEGIN
    idxs = np.random.choice(n_training_points, batch_size)
    grads = grad_jit(xx[idxs, :], yy[idxs, :], params)

    for i in range(len(params)):
        cumulated_square_grad[i] = (
            decay_rate * cumulated_square_grad[i]
            + (1 - decay_rate) * grads[i] * grads[i]
        )
        params[i] -= (
            learning_rate / (delta + jnp.sqrt(cumulated_square_grad[i])) * grads[i]
        )
    # SOLUTION-END

    history.append(loss_jit(xx, yy, params))
    cb(epoch)
cb.draw()

print("loss: %1.3e" % history[-1])
```

California Housing data from 1990 US Census

In this lab we will rely on Pandas and Seaborn to inspect and visualize data.

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import time
import jax.numpy as jnp
import jax
```

Load the dataset (if you are using Google Colab, it is already in the `sample_data` folder!).

```
data = pd.read_csv("./california_housing_train.csv")
```

Data inspection

Display some basic information.

```
data.head()
```

```
data.info()
```

```
data.describe()
```

We are interested in predicting the field `median_house_value`. Plot its distribution.

```
sns.histplot(data["median_house_value"], kde=True)
```

It looks like the distribution tail has been truncated. Get rid of it to ease the prediction.

```
data = data[data["median_house_value"] < 500001]
sns.histplot(data["median_house_value"], kde=True)
```

Use a scatterplot to visualize the geograhical distribution of the houses.

```
sns.scatterplot(data=data, x="longitude", y="latitude", hue="median_house_value")
```

Look for linear correlations among features.

```
data.corr()
```

```
sns.heatmap(data.corr(), annot=True, cmap="vlag_r", vmin=-1, vmax=1)
```

```
sns.scatterplot(data=data, x="latitude", y="median_house_value")
```

Data normalization

Apply an affine transformation to the data, so that each feature has zero mean and unitary standard deviation.

```
data_mean = data.mean()
data_std = data.std()
data_normalized = (data - data_mean) / data_std

data_normalized.describe()
```

```
_, ax = plt.subplots(figsize=(16, 6))
sns.violinplot(data=data_normalized, ax=ax)
```

Train-validation split

Shuffle the data (**hint:** use the np.random.shuffle function) and split the data as follows:

- put 80% in the train dataset
- put 20% in the validation dataset

```
np.random.seed(0)  # for reproducibility
data_normalized_np = data_normalized.to_numpy()
np.random.shuffle(data_normalized_np)

fraction_validation = 0.2
num_train = int(data_normalized_np.shape[0] * (1 - fraction_validation))
x_train = data_normalized_np[:num_train, :-1]
y_train = data_normalized_np[:num_train, -1:]
x_valid = data_normalized_np[num_train:, :-1]
y_valid = data_normalized_np[num_train:, -1:]
```

ANN setup

Write a function `initialize_params` that, given the input `layers_size = [n1, n2, ..., nL]`, generates the parameters associated with an ANN, having as many layers as the number of elements of `layers_size`, with as many neurons as `n1`, `n2`, etc.

To initialize the parameters, employ the following strategy:

- Inizialize the biases with zero value.
- Inizialize the weights sampling from a Gaussian distribution with zero mean and with standard deviation $$ \sqrt{\frac{2}{n + m}} $$$$ \sqrt{\frac{2}{n + m}} $$ where $n$$n$ and $m$$m$ are the number of input and output neurons of the corresponding weights matrix (this is known as "Glorot Normal" or "Xavier Normal", see Glorot, Bengio 2010).

Other popular initializations strategies are:

- Gaussian distribution with zero mean and with standard deviation (for some constant $K$$K$) $$ \frac{K}{\sqrt{n}} $$$$ \frac{K}{\sqrt{n}} $$
- Uniform distribution $$ \left[-\sqrt{\frac{1}{n}}, \sqrt{\frac{1}{n}}\right] $$$$ \left[-\sqrt{\frac{1}{n}}, \sqrt{\frac{1}{n}}\right] $$
- Uniform distribution (this is known as "Glorot Uniform" or "Xavier Uniform") $$ \left[-\sqrt{\frac{6}{n + m}}, \sqrt{\frac{6}{n + m}}\right] $$$$ \left[-\sqrt{\frac{6}{n + m}}, \sqrt{\frac{6}{n + m}}\right] $$

```
# SOLUTION-BEGIN
def initialize_params(layers_size):
    np.random.seed(0)  # for reproducibility
    params = list()
    for i in range(len(layers_size) - 1):
        W = np.random.randn(layers_size[i + 1], layers_size[i]) * np.sqrt(
            2 / (layers_size[i + 1] + layers_size[i])
        )
        b = np.zeros((layers_size[i + 1], 1))
        params.append((W, b))
    return params


params = initialize_params([8, 5, 5, 1])
# SOLUTION-END
```

Implement a generic feedforward ANN with a function `y = ANN(x, params)', using $\tanh$$\tanh$ as activation function.

By convention, both the input and the output have:

- 1 sample per row
- 1 feature per column

```
# SOLUTION-BEGIN
activation = jnp.tanh
activation = lambda x: jnp.maximum(0.0, x)


def ANN(x, params):
    layer = x.T
    for i, (W, b) in enumerate(params):
        layer = W @ layer - b
        if i < len(params) - 1:
            layer = activation(layer)
```

```
    return layer.T


ANN(x_train, params)
# SOLUTION-END
```

Implement the quadratic (MSE) loss function `L = loss(x, y, params)`, defined as:

$$ \mathcal{L}(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m (y_i - \mathrm{ANN}(x_i, \boldsymbol{\theta}))^2 $$$$ \mathcal{L}(\mathbf{x}, \mathbf{y}, \boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^m (y_i - \mathrm{ANN}(x_i, \boldsymbol{\theta}))^2 $$

where $m$$m$ is the number of samples in $\mathbf{x}$$\mathbf{x}$, $\mathbf{y}$$\mathbf{y}$ and $\boldsymbol{\theta}$$\boldsymbol{\theta}$ are the ANN parameters.

```
# SOLUTION-BEGIN
def loss(x, y, params):
    error = ANN(x, params) - y
    return jnp.mean(error * error)
# SOLUTION-END
```

Test your code, by generating the parameters associated with an ANN with two hidden layers with 5 neurons each and by computing the associated loss.

```
# SOLUTION-BEGIN
params = initialize_params([8, 5, 5, 1])
loss(x_train, y_train, params)
# SOLUTION-END
```

Training

Gradient Descent

Implement the GD method:

$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{N} \sum_{i=1}^N \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda \mathbf{g}^{(k)} \end{split} $$$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{N} \sum_{i=1}^N \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda \mathbf{g}^{(k)} \end{split} $$

where $N$$N$ is the number of training samples. At each iteration, append the current cost to the list `history`.

Train an ANN with two hidden layers with 20 neurons each. Try to (manually) optimize the training hyperparameters.

During training, store the MSE error obtained on the train and validation sets in two lists, respectively called `history_train` and `history_valid`. Finally, plot the erros trend and diplay the final values of the errors.

Hints:

- Use `jax.jit` to speedup the evaluation of the loss and of the gradients.

```
# Hyperparameters
layers_size = [8, 20, 20, 1]
# Training options
num_epochs = 2000
lr = 1e-1
########################################

params = initialize_params(layers_size)

grad = jax.jit(jax.grad(loss, argnums=2))
loss_jit = jax.jit(loss)
grad_jit = jax.jit(grad)

n_samples = x_train.shape[0]
```

```
history_train = list()
history_valid = list()
history_train.append(loss_jit(x_train, y_train, params))
history_valid.append(loss_jit(x_valid, y_valid, params))

# SOLUTION-BEGIN
t0 = time.time()
for epoch in range(num_epochs):
    grads = grad_jit(x_train, y_train, params)
    params = jax.tree_util.tree_map(lambda p, g: p - lr * g, params, grads)

    history_train.append(loss_jit(x_train, y_train, params))
    history_valid.append(loss_jit(x_valid, y_valid, params))

# SOLUTION-END

print("elapsed time: %f s" % (time.time() - t0))
print("loss train     : %1.3e" % history_train[-1])
print("loss validation: %1.3e" % history_valid[-1])

fig, axs = plt.subplots(1, figsize=(16, 8))
axs.loglog(history_train, label="train")
axs.loglog(history_valid, label="validation")
plt.legend()
```

Stochastic Gradient Descent

Implement the SGD method:

$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda_k \mathbf{g}^{(k)} \end{split} $$$$ \begin{split} & \boldsymbol{\theta}^{(0)} \text{given} \\ & \text{for } k = 0, 1, \dots , n_{\text{epochs}} - 1\\ & \qquad \mathbf{g}^{(k)} = \frac{1}{|I_k|} \sum_{i \in I_k} \nabla_{\boldsymbol{\theta}} \mathcal{L}(x_i, y_i, \boldsymbol{\theta}^{(k)}) \\ & \qquad \boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \lambda_k \mathbf{g}^{(k)} \end{split} $$

where $I_k$$I_k$ is the current minibatch. To select it, use the function [np.random.choice](#) with replacement.

Consider a linear decay of the learning rate:

$$ \lambda_k = \max\left(\lambda_{\textnormal{min}}, \lambda_{\textnormal{max}} \left(1 - \frac{k}{K}\right)\right) $$$$ \lambda_k = \max\left(\lambda_{\textnormal{min}}, \lambda_{\textnormal{max}} \left(1 - \frac{k}{K}\right)\right) $$

Test different choices of batch size and try to optimize the learning rate decay strategy.

```
# Hyperparameters
layers_size = [8, 20, 20, 1]
# Training options
num_epochs = 1998
learning_rate_max = 1e-1
learning_rate_min = 1e-1
learning_rate_decay = num_epochs
batch_size = 1000
#######################################

params = initialize_params(layers_size)

grad = jax.jit(jax.grad(loss, argnums=2))
loss_jit = jax.jit(loss)
grad_jit = jax.jit(grad)

n_samples = x_train.shape[0]

history_train = list()
history_valid = list()
history_train.append(loss_jit(x_train, y_train, params))
history_valid.append(loss_jit(x_valid, y_valid, params))

# SOLUTION-BEGIN
t0 = time.time()
for epoch in range(num_epochs):
    lr = max(
        learning_rate_min, learning_rate_max * (1 - epoch / learning_rate_decay)
    )
```

```
        perm = np.random.permutation(n_samples)
        for i in range(0, n_samples, batch_size):
            batch_idx = perm[i : i + batch_size]
            x_batch = x_train[batch_idx]
            y_batch = y_train[batch_idx]
            grads = grad_jit(x_batch, y_batch, params)
            params = jax.tree_util.tree_map(lambda p, g: p - lr * g, params, grads)


        history_train.append(loss_jit(x_train, y_train, params))
        history_valid.append(loss_jit(x_valid, y_valid, params))
    # SOLUTION-END

    print("elapsed time: %f s" % (time.time() - t0))
    print("loss train     : %1.3e" % history_train[-1])
    print("loss validation: %1.3e" % history_valid[-1])

    fig, axs = plt.subplots(1, figsize=(16, 8))
    axs.loglog(history_train, label="train")
    axs.loglog(history_valid, label="validation")
    plt.legend()
```

Testing

Load the test dataset `sample_data/california_housing_test.csv` and use the trained model to predict the house prices of the dataset.

```
    # SOLUTION-BEGIN
    data_test = pd.read_csv("./california_housing_test.csv")
    data_test = data_test[data_test["median_house_value"] < 500001]
    data_test_normalized = (data_test - data.mean()) / data.std()
    x_test = data_test_normalized.drop("median_house_value", axis=1).to_numpy()
    Y_test = data_test["median_house_value"].to_numpy()[:, None]

    y_predicted = ANN(x_test, params)
    Y_predicted = (y_predicted * data["median_house_value"].std()) + data[
        "median_house_value"
    ].mean()
    # SOLUTION-END
```

Compare predicted prices with actual prices by means of a scatterplot.

```
    # SOLUTION-BEGIN
    test = pd.DataFrame({"predicted": Y_predicted[:, 0], "actual": Y_test[:, 0]})
    fig = sns.jointplot(data=test, x="actual", y="predicted")
    fig.ax_joint.plot([0, 500000], [0, 500000.0], "r")
    # SOLUTION-END
```

Finally, compute the RMSE (root mean square error).

```
    # SOLUTION-BEGIN
    error = Y_test - Y_predicted
    RMSE = jnp.sqrt(jnp.mean(error * error))
    print("RMSE: %0.2f k$" % (RMSE * 1e-3))
    # SOLUTION-END
```

Newton method

```
    import numpy as np
    import matplotlib.pyplot as plt
    import jax.numpy as jnp
    import jax

    # We enable double precision in JAX
    jax.config.update("jax_enable_x64", True)
```

We consider a random matrix $A \in \mathbb{R}^{n\times n}$$A \in \mathbb{R}^{n\times n}$, with $n = 100$$n = 100$ and a random vector $\mathbf{x}_{\text{ex}} \in \mathbb{R}^n$$\mathbf{x}_{\text{ex}} \in \mathbb{R}^n$. We define then $\mathbf{b} = A \, \mathbf{x}_{\text{ex}}$$\mathbf{b} = A \, \mathbf{x}_{\text{ex}}$.

```
n = 100

np.random.seed(0)
A = np.random.randn(n, n)
x_ex = np.random.randn(n)
b = A @ x_ex
```

Define the loss function

$$ \mathcal{L}(\mathbf{x}) = \| \mathbf{b} - A \, \mathbf{x} \|_2^2 $$$$ \mathcal{L}(\mathbf{x}) = \| \mathbf{b} - A \, \mathbf{x} \|_2^2 $$

```
def loss(x):
    return jnp.sum(jnp.square(A @ x - b))


print(loss(x_ex))
```

By using the `jax` library, implement and compile functins returning the gradient ($\nabla \mathcal{J}(\mathbf{x})$$\nabla \mathcal{J}(\mathbf{x})$) and the hessian ($\nabla^2 \mathcal{J}(\mathbf{x})$$\nabla^2 \mathcal{J}(\mathbf{x})$) of the loss function.

```
grad = jax.grad(loss)
hess = jax.jacfwd(jax.jacrev(loss))

loss_jit = jax.jit(loss)
grad_jit = jax.jit(grad)
hess_jit = jax.jit(hess)
```

Check that the results are correct (up to machine precision).

```
np.random.seed(0)
x_guess = np.random.randn(n)

G_ad = grad_jit(x_guess)
G_ex = 2 * A.T @ (A @ x_guess - b)
print(np.linalg.norm(G_ad - G_ex))

H_ad = hess_jit(x_guess)
H_ex = 2 * A.T @ A
print(np.linalg.norm(H_ad - H_ex))
```

- JAX enables us to compute the Hessian $\nabla^2 \mathcal{J}(\mathbf{x})$$\nabla^2 \mathcal{J}(\mathbf{x})$. This is a **function**.
- Then, we use it by evaluating the Hessian at one point $\nabla^2 \mathcal{J}(\mathbf{x})|_{\mathbf{x}_0}$$\nabla^2 \mathcal{J}(\mathbf{x})|_{\mathbf{x}_0}$, for instance when we apply the Newton's method. This is a **matrix**.
- However, when we use this matrix (the evaluation of the Hessian at a point), it rarely happens that we need the whole structure of the matrix. In most case it is sufficient to know the action that the matrix has on a vector $\mathbf{v}_0$$\mathbf{v}_0$. **Avoiding storing a full matrix greatly diminishes the cost in memory (which is quadratic).**

In other words, do we need to compute $\nabla^2 \mathcal{J}(\mathbf{x})|_{\mathbf{x}_0}$$\nabla^2 \mathcal{J}(\mathbf{x})|_{\mathbf{x}_0}$ in order to compute $\nabla^2 \mathcal{J}(\mathbf{x})|_{\mathbf{x}_0} \mathbf{v}_0$$\nabla^2 \mathcal{J}(\mathbf{x})|_{\mathbf{x}_0} \mathbf{v}_0$?

From calcolus we know that

$$ \nabla^2 \mathcal{J}(\mathbf{x}) \mathbf{v} = \nabla_{\mathbf{x}} \phi(\mathbf{x}, \mathbf{v}) $$$$ \nabla^2 \mathcal{J}(\mathbf{x}) \mathbf{v} = \nabla_{\mathbf{x}} \phi(\mathbf{x}, \mathbf{v}) $$

where

$$ \phi(\mathbf{x}, \mathbf{v}) := \nabla \mathcal{J}(\mathbf{x}) \cdot \mathbf{v} $$$$ \phi(\mathbf{x}, \mathbf{v}) := \nabla \mathcal{J}(\mathbf{x}) \cdot \mathbf{v} $$

because

$$ \left(\nabla_{\mathbf x}\phi(\mathbf x,\mathbf v) \right)_i = \frac{\partial}{\partial x_i} \left(\sum_{j=1}^n \frac{\partial \mathcal J}{\partial x_j}(\mathbf x)\, v_j \right) = \sum_{j=1}^n \frac{\partial^2 \mathcal J}{\partial x_i \partial x_j}(\mathbf x)\, v_j . $$$$ \left(\nabla_{\mathbf x}\phi(\mathbf x,\mathbf v) \right)_i = \frac{\partial}{\partial x_i} \left(\sum_{j=1}^n \frac{\partial \mathcal J}{\partial x_j}(\mathbf x)\, v_j \right) = \sum_{j=1}^n \frac{\partial^2 \mathcal J}{\partial x_i \partial x_j}(\mathbf x)\, v_j . $$

Then, in JAX we can compute $\nabla^2 \mathcal{J}(\mathbf{x})|_{\mathbf{x}_0} \mathbf{v}_0$$\nabla^2 \mathcal{J}(\mathbf{x})|_{\mathbf{x}_0} \mathbf{v}_0$ by evaluating at $(\mathbf{x}_0, \mathbf{v}_0)$$(\mathbf{x}_0, \mathbf{v}_0)$ the function $$ (\mathbf{x}, \mathbf{v}) \mapsto \nabla_{\mathbf{x}} (\nabla \mathcal{J}(\mathbf{x}) \cdot \mathbf{v}). $$$$ (\mathbf{x}, \mathbf{v}) \mapsto \nabla_{\mathbf{x}} (\nabla \mathcal{J}(\mathbf{x}) \cdot \mathbf{v}). $$

Notice that in this implementation there is never a matrix since $\mathcal{J}$$\mathcal{J}$ is a scalar field.

Implement this version of the Hessian and compare the computational performance w.r.t. the full hessian computation.

```
np.random.seed(1)
v = np.random.randn(n)

hvp_basic = lambda x, v: hess(x) @ v

# SOLUTION-BEGIN
gvp = lambda x, v: jnp.dot(grad(x), v)
hvp = lambda x, v: jax.grad(gvp, argnums=0)(x, v)

hvp_basic_jit = jax.jit(hvp_basic)
hvp_jit = jax.jit(hvp)
# SOLUTION-END

Hv_ad = hvp_jit(x_guess, v)
Hv_ex = H_ex @ v
print(np.linalg.norm(Hv_ad - Hv_ex))
```

```
%timeit hvp_basic_jit(x_guess, v)
%timeit hvp_jit(x_guess, v)
```

Implement the Newton method for the minimization of the loss function $\mathcal{L}$$\mathcal{L}$. Set a maximim number of 100 iterations and a tolerance on the increment norm of $\epsilon = 10^{-8}$$\epsilon = 10^{-8}$.

```
x = x_guess.copy()
num_epochs = 100
eps = 1e-8

for epoch in range(num_epochs):
    # SOLUTION-BEGIN
    H = hess_jit(x)
    G = grad_jit(x)
    incr = np.linalg.solve(H, -G)
    x += incr
    # SOLUTION-END

    print("============ epoch %d" % epoch)
    print("loss: %1.3e" % loss_jit(x))
    print("grad: %1.3e" % np.linalg.norm(G))
    print("incr: %1.3e" % np.linalg.norm(incr))

    if np.linalg.norm(incr) < eps:
        break

rel_err = np.linalg.norm(x - x_ex) / np.linalg.norm(x_ex)
print(f"Relative error: {rel_err:1.3e}")
```

Solve the system with `jax.scipy.sparse.linalg.cg` and use the "matrix-free" version of the Hessia

```
x = x_guess.copy()
num_epochs = 100
eps = 1e-8

for epoch in range(num_epochs):
    # SOLUTION-BEGIN
    G = grad_jit(x)
    incr, info = jax.scipy.sparse.linalg.cg(lambda y: hvp_jit(x, y), -G, tol=eps)
    x += incr
```

```
    # SOLUTION-END

    print("============ epoch %d" % epoch)
    print("loss: %1.3e" % loss_jit(x))
    print("grad: %1.3e" % np.linalg.norm(G))
    print("incr: %1.3e" % np.linalg.norm(incr))

    if np.linalg.norm(incr) < eps:
        break

rel_err = np.linalg.norm(x - x_ex) / np.linalg.norm(x_ex)
print(f"Relative error: {rel_err:1.3e}")
```

Repeat the optimization loop for the loss function

$$ \mathcal{L}(\mathbf{x}) = \| \mathbf{b} - A \, \mathbf{x} \|_2^4 $$$$ \mathcal{L}(\mathbf{x}) = \| \mathbf{b} - A \, \mathbf{x} \|_2^4 $$

```
def loss(x):
    return jnp.sum((A @ x - b) ** 4)


grad = jax.grad(loss)
hess = jax.jacfwd(jax.jacrev(loss))

loss_jit = jax.jit(loss)
grad_jit = jax.jit(grad)
hess_jit = jax.jit(hess)
```

```
x = x_guess.copy()
num_epochs = 100
eps = 1e-8

hist = [loss_jit(x)]
for epoch in range(num_epochs):
    hist.append(loss_jit(x))

    H = hess_jit(x)
    G = grad_jit(x)
    incr = np.linalg.solve(H, -G)
    x += incr

    if np.linalg.norm(incr) < eps:
        print("convergence reached!")
        break

plt.semilogy(hist, "o-")
print("epochs: %d" % epoch)
print("relative error: %1.3e" % (np.linalg.norm(x - x_ex) / np.linalg.norm(x_ex)))
```

Quasi-Newton Optimization with BFGS Update

This algorithm minimizes an objective function $ f(\mathbf{x}) $$ f(\mathbf{x}) $ using a quasi-Newton method with the BFGS update. The goal is to iteratively update the solution $ \mathbf{x} $$ \mathbf{x} $ and approximate the inverse Hessian until convergence criteria are met.

Algorithm

1. **Initialization**:
   - Set the initial guess $ \mathbf{x} = \mathbf{x}_{\text{guess}} $$ \mathbf{x} = \mathbf{x}_{\text{guess}} $.
   - Let $ \mathbf{I} $$ \mathbf{I} $ be the identity matrix, and initialize $ B^{-1} = \mathbf{I} $$ B^{-1} = \mathbf{I} $.
   - Compute the initial gradient $ \nabla f = \nabla f(\mathbf{x}_{\text{guess}}) $$ \nabla f = \nabla f(\mathbf{x}_{\text{guess}}) $.
   - Initialize the loss history: $ \text{history} = [f(\mathbf{x}_{\text{guess}})] $$ \text{history} = [f(\mathbf{x}_{\text{guess}})] $.
   - Set $ \text{epoch} = 0 $$ \text{epoch} = 0 $.

   $$ B^{-1} = \mathbf{I}, \quad \nabla f = \nabla f(\mathbf{x}_{\text{guess}}), \quad \text{history} = [f(\mathbf{x}_{\text{guess}})] $$$$ B^{-1} = \mathbf{I}, \quad \nabla f = \nabla f(\mathbf{x}_{\text{guess}}), \quad \text{history} = [f(\mathbf{x}_{\text{guess}})] $$

2. **Iterative Updates**:

- While $ \|\nabla f\| > \text{tol} $$ \|\nabla f\| > \text{tol} $ and epoch $<$$<$ max epoch:
  - Increment the epoch counter:

    $$ \text{epoch} \leftarrow \text{epoch} + 1 $$
  - Compute the search direction:

    $$ \mathbf{p} = -B^{-1} \nabla f $$
  - Perform a line search to find the step size $ \alpha $ using `sp.optimize.line_search`:

    $$ \alpha \leftarrow \text{line_search}(f, \nabla f, \mathbf{x}, \mathbf{p}) $$

    If $ \alpha $ is not found (scipy returns `None`), set $ \alpha = 10^{-8} $.
  - Update the solution vector:

    $$ \mathbf{x}_{\text{new}} = \mathbf{x} + \alpha \mathbf{p} $$
  - Compute the displacement:

    $$ \mathbf{s} = \mathbf{x}_{\text{new}} - \mathbf{x} $$
  - Update $ \mathbf{x} $:

    $$ \mathbf{x} \leftarrow \mathbf{x}_{\text{new}} $$
  - Compute the new gradient and gradient difference:

    $$ \nabla f_{\text{new}} = \nabla f(\mathbf{x}), \quad \mathbf{y} = \nabla f_{\text{new}} - \nabla f $$

    Update $ \nabla f $:

    $$ \nabla f \leftarrow \nabla f_{\text{new}} $$
  - Compute the scalar $ \rho $:

    $$ \rho = \frac{1}{\mathbf{y}^\top \mathbf{s}} $$
  - Update the inverse Hessian approximation $ B^{-1} $ using the Sherman–Morrison formula:

    $$ \mathbf{E} = \mathbf{I} - \rho \mathbf{y} \mathbf{s}^\top $$

    $$ B^{-1} \leftarrow \mathbf{E}^\top B^{-1} \mathbf{E} + \rho \mathbf{s} \mathbf{s}^\top $$
  - Append the current loss value to history: $$ \text{history.append}(f(\mathbf{x})) $$

```
max_epochs = 1000
tol = 1e-8

import scipy as sp

np.random.seed(0)

epoch = 0
x = x_guess.copy()
I = np.eye(x.size)
Binv = I
grad = grad_jit(x_guess)
history = [loss_jit(x_guess)]

while np.linalg.norm(grad) > tol and epoch < max_epochs:
    epoch += 1

    # SOLUTION-BEGIN
    # search direction
    p = -Binv @ grad

    # line search
    alpha = sp.optimize.line_search(loss_jit, grad_jit, x, p)[0]
    alpha = 1e-8 if alpha is None else alpha
    x_new = x + alpha * p

    # computing y and s
    s = x_new - x
    x = x_new
    grad_new = grad_jit(x_new)
    y = grad_new - grad
    grad = grad_new

    # Sherman-Morrison update
```

```
        rho = 1.0 / (np.dot(y, s))
        E = I - rho * np.outer(y, s)
        Binv = E.T @ Binv @ E + rho * np.outer(s, s)
        # SOLUTION-END

        history.append(loss_jit(x))

    plt.semilogy(history, "o-")
```

Regularization of ANN weights

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
import time
import jax.numpy as jnp
import jax
```

Load the Auto MPG dataset.

```
url = "http://archive.ics.uci.edu/ml/machine-learning-databases/auto-mpg/auto-mpg.data"
column_names = [
    "MPG",
    "Cylinders",
    "Displacement",
    "Horsepower",
    "Weight",
    "Acceleration",
    "Model Year",
    "Origin",
]
data = pd.read_csv(
    url, names=column_names, na_values="?", comment="\t", sep=" ", skipinitialspace=True
)
data
```

Check is there are missing entries in the dataset.

```
print(data.isna().sum())
```

Remove records with missing entries.

```
data = data.dropna()
print(data.isna().sum())
```

Data inspection

Display some basic information.

```
data.head()
```

```
data.info()
```

```
data.describe()
```

We are interested in predicting the field `MPG`, (index measuring fuel efficiency), expressed in miles per gallon (MPG), where 1 MPG = 0.354006 km/L. Plot its distribution.

```
sns.displot(data["MPG"], kde=True)
```

Look for linear correlations among data.

```
sns.heatmap(data.corr(), annot=True, cmap="vlag_r", vmin=-1, vmax=1)
```

```
sns.pairplot(data, diag_kind="kde")
```

Data normalization

Apply an affine transformation to the data, so that each feature has zero mean and unitary standard deviation.

```
data_mean = data.mean()
data_std = data.std()
data_normalized = (data - data_mean) / data_std
```

```
_, ax = plt.subplots(figsize=(16, 6))
sns.violinplot(data=data_normalized, ax=ax)
```

Train-validation split

Shuffle the data using the np.random.shuffle function and split the data as follows:

- put 80% in the train dataset
- put 20% in the validation dataset

```
data_normalized_np = data_normalized.to_numpy()
np.random.seed(0)
np.random.shuffle(data_normalized_np)

fraction_validation = 0.2
num_train = int(data_normalized_np.shape[0] * (1 - fraction_validation))
x_train = data_normalized_np[:num_train, 1:]
y_train = data_normalized_np[:num_train, :1]
x_valid = data_normalized_np[num_train:, 1:]
y_valid = data_normalized_np[num_train:, :1]

print("train set size     : %d" % x_train.shape[0])
print("validation set size: %d" % x_valid.shape[0])
```

ANN setup

Write a function `params = initialize_params(layers_size)` that initializes the parameters, given the ANN architecture. Initialize biases with zero values, and weights with a Glorot Normal initialization, i.e. sampling from a Gaussian distribution with zero mean and with standard deviation

$$ \sqrt{\frac{2}{n + m}}, $$$$ \sqrt{\frac{2}{n + m}}, $$

where $n$$n$ and $m$$m$ are the number of input and output neurons of the corresponding weights matrix.

```
def initialize_params(layers_size):
    np.random.seed(0)  # for reproducibility
    params = list()
    for i in range(len(layers_size) - 1):
        W = np.random.randn(layers_size[i + 1], layers_size[i]) * np.sqrt(
            2 / (layers_size[i + 1] + layers_size[i])
        )
        b = np.zeros((layers_size[i + 1], 1))
        params.append(W)
        params.append(b)
    return params
```

Implement a generic feedforward ANN with a function `y = ANN(x, params)`, using $ReLU$$ReLU$ as activation function.

```
activation = jax.nn.relu


def ANN(x, params):
    layer = x.T
```

```
    num_layers = int(len(params) / 2 + 1)
    weights = params[0::2]
    biases = params[1::2]
    for i in range(num_layers - 1):
        layer = weights[i] @ layer - biases[i]
        if i < num_layers - 2:
            layer = activation(layer)
    return layer.T


params = initialize_params([7, 10, 1])
ANN(x_train[:10, :], params)
```

Implement the quadratic loss (MSE) function `L = MSE(x, y, params)`.

```
def MSE(x, y, params):
    error = ANN(x, params) - y
    return jnp.mean(error * error)


params = initialize_params([7, 10, 1])
print(MSE(x_train, y_train, params))
```

Implement an $l^2$$l^2$ regularization term for the ANN weights:

$$ \mathrm{MSW} = \frac{1}{n_{weights}} \sum_{i=1}^{n_{weights}} w_i^2 $$$$ \mathrm{MSW} = \frac{1}{n_{weights}} \sum_{i=1}^{n_{weights}} w_i^2 $$

and define the loss function as

$$ \mathcal{L} = \mathrm{MSE} + \beta \, \mathrm{MSW} $$$$ \mathcal{L} = \mathrm{MSE} + \beta \, \mathrm{MSW} $$

where $\beta$$\beta$ is a suitable penalization parameter.

```
# SOLUTION-BEGIN
def MSW(params):
    weights = params[::2]
    partial_sum = 0.0
    n_weights = 0
    for W in weights:
        partial_sum += jnp.sum(W * W)
        n_weights += W.shape[0] * W.shape[1]
    return partial_sum / n_weights


def loss(x, y, params, penalization):
    return MSE(x, y, params) + penalization * MSW(params)
# SOLUTION-END

print(MSW(params))
print(loss(x_train, y_train, params, 1.0))
```

Run this cell: we will this callback to monitor training.

```
from IPython import display


class Callback:
    def __init__(self, refresh_rate=250):
        self.refresh_rate = refresh_rate
        self.fig, self.axs = plt.subplots(1, figsize=(16, 8))
        self.epoch = 0
        self.__call__(-1)

    def __call__(self, epoch):
        self.epoch = epoch
        if (epoch + 1) % self.refresh_rate == 0:
            self.draw()
            display.clear_output(wait=True)
            display.display(plt.gcf())
            time.sleep(1e-16)

    def draw(self):
```

```
        if self.epoch > 0:
            self.axs.clear()
            self.axs.loglog(history_loss_train, "b-", label="loss train")
            self.axs.loglog(history_loss_valid, "r-", label="loss validation")
            self.axs.loglog(history_MSE_train, "b--", label="RMSE train")
            self.axs.loglog(history_MSE_valid, "r--", label="RMSE validation")
            self.axs.legend()
            self.axs.set_title("epoch %d" % (self.epoch + 1))
```

Training

Train an ANN with two hidden layers with 20 neurons each, using 5000 epochs of the SGD method (with minibatch size 100) with momentum ($\alpha = 0.9$$\alpha = 0.9$). Employ a linear decay of the learning rate:

$$ \lambda_k = \max\left(\lambda_{\textnormal{min}}, \lambda_{\textnormal{max}} \left(1 - \frac{k}{K}\right)\right) $$$$ \lambda_k = \max\left(\lambda_{\textnormal{min}}, \lambda_{\textnormal{max}} \left(1 - \frac{k}{K}\right)\right) $$

with $\lambda_{\textnormal{min}} = 5e-3$$\lambda_{\textnormal{min}} = 5e-3$, $\lambda_{\textnormal{max}} = 1e-1$$\lambda_{\textnormal{max}} = 1e-1$ and decay length $K= 1000$$K= 1000$.

During training, store both the MSE error and the loss function obtained on the train and validation sets in 4 lists, respectively called:

- `history_loss_train`
- `history_loss_valid`
- `history_MSE_train`
- `history_MSE_valid`

Test different choices of the penalization parameter $\beta$$\beta$.

```
# Hyperparameters
layers_size = [7, 20, 20, 1]
penalization = 2.0
# Training options
num_epochs = 5000
learning_rate_max = 1e-1
learning_rate_min = 5e-3
learning_rate_decay = 1000
batch_size = 100
alpha = 0.9
#######################################

params = initialize_params(layers_size)

grad = jax.grad(loss, argnums=2)
MSE_jit = jax.jit(MSE)
loss_jit = jax.jit(loss)
grad_jit = jax.jit(grad)

n_samples = x_train.shape[0]

history_loss_train = list()
history_loss_valid = list()
history_MSE_train = list()
history_MSE_valid = list()


def dump():
    history_loss_train.append(loss_jit(x_train, y_train, params, penalization))
    history_loss_valid.append(loss_jit(x_valid, y_valid, params, penalization))
    history_MSE_train.append(MSE_jit(x_train, y_train, params))
    history_MSE_valid.append(MSE_jit(x_valid, y_valid, params))


dump()
cb = Callback(refresh_rate=500)

velocity = [0.0 for i in range(len(params))]
for epoch in range(num_epochs):
    learning_rate = max(
        learning_rate_min, learning_rate_max * (1 - epoch / learning_rate_decay)
    )
    idxs = np.random.choice(n_samples, batch_size)
    grads = grad_jit(x_train[idxs, :], y_train[idxs, :], params, penalization)
```

```
    for i in range(len(params)):
        velocity[i] = alpha * velocity[i] - learning_rate * grads[i]
        params[i] += velocity[i]

    dump()
    cb(epoch)
cb.draw()

print("loss (train     ): %1.3e" % history_loss_train[-1])
print("loss (validation): %1.3e" % history_loss_valid[-1])
print("MSE  (train     ): %1.3e" % history_MSE_train[-1])
print("MSE  (validation): %1.3e" % history_MSE_valid[-1])
```

We now want to to investigate more in depth the effect of the penalization parameter $\beta$$\beta$. Write a function that, given the penalization parameter, trains the ANN (with the same setting used above) and returns a dictionary containing the final values of:

- train MSE
- validation MSE
- MSW

```
# Hyperparameters
layers_size = [7, 20, 20, 1]
# Training options
num_epochs = 5000
learning_rate_max = 1e-1
learning_rate_min = 5e-3
learning_rate_decay = 1000
batch_size = 100
alpha = 0.9


def train(penalization):
    # SOLUTION-BEGIN
    params = initialize_params(layers_size)
    n_samples = x_train.shape[0]

    velocity = [0.0 for i in range(len(params))]
    for epoch in range(num_epochs):
        learning_rate = max(
            learning_rate_min, learning_rate_max * (1 - epoch / learning_rate_decay)
        )
        idxs = np.random.choice(n_samples, batch_size)
        grads = grad_jit(x_train[idxs, :], y_train[idxs, :], params, penalization)
        for i in range(len(params)):
            velocity[i] = alpha * velocity[i] - learning_rate * grads[i]
            params[i] += velocity[i]
    # SOLUTION-END
    return {
        "MSE_train": MSE(x_train, y_train, params),
        "MSE_valid": MSE(x_valid, y_valid, params),
        "MSW": MSW(params),
    }
```

Using the above defined function, store the obtained results for $\beta = 0, 0.25, 0.5, 0.75, \dots, 2$$\beta = 0, 0.25, 0.5, 0.75, \dots, 2$.

```
results = []

pen_values = np.arange(0, 2.1, 0.25)
for beta in pen_values:
    print("training for beta = %f..." % beta)
    results.append({"beta": beta, **train(beta)})

hyper_tuning_df = pd.DataFrame(results)
hyper_tuning_df
```

Plot the trend of the five quantities as functions of $\beta$$\beta$.

```
# SOLUTION-BEGIN
_, axs = plt.subplots(1, 3, figsize=(12, 6))

axs[0].plot(pen_values, hyper_tuning_df["MSE_train"], "o-")
```

```
axs[1].plot(pen_values, hyper_tuning_df["MSE_valid"], "o-")
axs[2].plot(pen_values, hyper_tuning_df["MSW"], "o-")

axs[0].set_title("MSE_train")
axs[1].set_title("MSE_valid")
axs[2].set_title("MSW")
axs[0].set_xlabel(r"$\beta$")
axs[1].set_xlabel(r"$\beta$")
axs[2].set_xlabel(r"$\beta$")
# SOLUTION-END
```

Plot the *Tikhonov L-curve*, which is - in this context - the curve "train MSE" versus "MSW".

```
# SOLUTION-BEGIN
plt.plot(hyper_tuning_df["MSW"], hyper_tuning_df["MSE_train"], "o-")
plt.xlabel("MSW")
plt.ylabel("MSE_train")
# SOLUTION-END
```

1D convolutions

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import signal
from scipy import linalg
import time
```

Consider the square wave discrete signal:

$$ \mathbf{v}_i= \begin{cases} 0, & i = 1,\dots, 50 \\ 1, & i = 51,\dots, 75 \\ 0, & i = 76,\dots, 100 \\ \end{cases} $$$$ \mathbf{v}_i= \begin{cases} 0, & i = 1,\dots, 50 \\ 1, & i = 51,\dots, 75 \\ 0, & i = 76,\dots, 100 \\ \end{cases} $$

Define the vector $\mathbf{v}$$\mathbf{v}$ and plot the corresponding signal.

```
v = np.zeros(100)
v[50:75] = 1

plt.plot(v)
```

Consider the following kernels $\mathbf{k}$$\mathbf{k}$:

```
### first kernel
k = np.ones(10) / 10

### second kernel
# k = signal.gaussian(20, std=3)
# k = k / np.sum(k)

### third kernel
# k = np.array([-1,2,-1])

plt.plot(k)
```

In what follows, we will compute the convolution between $\mathbf{v}$$\mathbf{v}$ and $\mathbf{k}$$\mathbf{k}$ in 4 different manners. At the end of each point, plot the obtained filtered (i.e. convolved) signal.

1. Toeplitz matrix

Define a Toeplitz matrix $K$$K$ associated with the kernel $\mathbf{k}$$\mathbf{k}$. Then, compute the convolution between $\mathbf{k}$$\mathbf{k}$ and $\mathbf{v}$$\mathbf{v}$, exploiting the property:

$\mathbf{k} * \mathbf{v} = K \mathbf{v}$$\mathbf{k} * \mathbf{v} = K \mathbf{v}$

```
t0 = time.time()

# SOLUTION-BEGIN
```

```
k_padded = np.zeros(len(v) + len(k) - 1)
k_padded[: len(k)] = k
K = linalg.toeplitz(k_padded, np.zeros(len(v)))
v_conv1 = np.matmul(K, v)
# SOLUTION-END

print("Execution time: %1.2e s" % (time.time() - t0))

plt.plot(v_conv1)
```

2. Direct definition (sums of products)

The convolution without "boundary layer" (i.e. the part of the vector which is identical between the standard and the circular convolutions), is given by:

$$(\mathbf{k} * \mathbf{v})_i = \sum_{j = 0}^{n_k - 1} \mathbf{k}_{n_k - j} \mathbf{v}_{i + j}$$$$(\mathbf{k} * \mathbf{v})_i = \sum_{j = 0}^{n_k - 1} \mathbf{k}_{n_k - j} \mathbf{v}_{i + j}$$

for $i = 1, \dots, n_v - n_k + 1$$i = 1, \dots, n_v - n_k + 1$.

```
t0 = time.time()

# SOLUTION-BEGIN
l_out = len(v) - len(k) + 1
v_conv2 = np.zeros(l_out)
for i in range(l_out):
    # for j in range(len(k)):
    #    v_conv2[i] += k[-j] * v[i + j]
    v_conv2[i] = np.sum(np.flip(k) * v[i : i + len(k)])
# SOLUTION-END

print("Execution time: %1.2e s" % (time.time() - t0))

plt.plot(v_conv2)
```

3. Convolution through DFT

When the two signals have the same size, we have:

$ \widehat{\mathbf{v} \circledast \mathbf{k}} = \widehat{\mathbf{v}} \odot \widehat{\mathbf{k}} $$ \widehat{\mathbf{v} \circledast \mathbf{k}} = \widehat{\mathbf{v}} \odot \widehat{\mathbf{k}} $

Where:

- $\hat{\mathbf{a}}$$\hat{\mathbf{a}}$ is the DFT of the signal $\mathbf{a}$$\mathbf{a}$
- $\circledast$$\circledast$ is the circular convolution
- $\odot$$\odot$ is the Hadamard product

When the two signals do not have the same size, the above formula can be applied simply by "zero-padding" the shorter signal (N.B: this is automatically done by the `numpy.fft.fft` command by passing the required length as second argument).

Besides the filtered signal, plot the DFT of the two signals and of their convolution.

```
from numpy.fft import ifft, fft, fftshift, fftfreq

t0 = time.time()

# SOLUTION-BEGIN
v_fft = fft(v)
k_fft = fft(k, len(v))

vk_fft = v_fft * k_fft

v_conv3 = np.real(ifft(vk_fft))
# SOLUTION-END

print("Execution time: %1.2e s" % (time.time() - t0))

# SOLUTION-BEGIN
freq = fftfreq(len(v))
```

```
fig, axs = plt.subplots(2, 2, figsize=(22, 8))
axs[0, 0].plot(fftshift(freq), fftshift(np.absolute(v_fft)))
axs[0, 0].set_title("FFT of v")
axs[0, 1].plot(fftshift(freq), fftshift(np.absolute(k_fft)))
axs[0, 1].set_title("FFT of k")
axs[1, 0].plot(fftshift(freq), fftshift(np.absolute(vk_fft)))
axs[1, 0].set_title("FFT of v * k")
axs[1, 1].plot(v_conv3)
axs[1, 1].set_title("v * k")
# SOLUTION-END
```

4. Using the command `scipy.signal.convolve`

Check out the documentation.

```
t0 = time.time()

# SOLUTION-BEGIN
v_conv4 = signal.convolve(v, k, mode="valid")
# SOLUTION-END

print("Execution time: %1.2e s" % (time.time() - t0))

plt.plot(v_conv4)
```

2D convolutions

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.image import imread
from scipy import signal
from scipy import linalg
import time
```

Consider a 2D signal $\mathbf{v}$$\mathbf{v}$ representing a picture

```
# write here the import path of the image
image_path = "./NYlibrary.png"

v = np.mean(imread(image_path), axis=2)

plt.rcParams["figure.figsize"] = [8, 8]

vmin = v.min()
vmax = v.max()
img = plt.imshow(v, cmap="gray", vmin=vmin, vmax=vmax)
plt.axis("off")
```

Consider the following kernels

```
kernel_blur1 = np.array([[1, 2, 1], [2, 4, 2], [1, 2, 1]])
kernel_blur1 = kernel_blur1 / np.sum(kernel_blur1)

kernel_blur2 = np.ones((10, 10)) / 100

kernel_edge1 = np.array([[-1, -1, -1], [-1, 8, -1], [-1, -1, -1]])
kernel_edge2 = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])
kernel_edge3 = np.array([[1, 0, -1], [0, 0, 0], [-1, 0, 1]])

kernel_sobel_horizontal = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])
kernel_sobel_vertical = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])

kernel_sharpen = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])

kernel_drunk = np.random.randn(50, 50)
```

Choose here the desired filter $\mathbf{k}$$\mathbf{k}$

```
k = kernel_blur1
```

In what follows, we will compute the convolution between $\mathbf{v}$$\mathbf{v}$ and $\mathbf{k}$$\mathbf{k}$ in 4 different manners. At the end of each point, plot the obtained filtered (i.e. convolved) signal.

1. Direct definition (sums of products)

Convolution without "boundary layers"

$$ \begin{aligned} &\text{Let } v \in \mathbb{R}^{M \times N},\quad k \in \mathbb{R}^{m \times n}.\\[4pt] &\text{Define the output size:}\qquad S_x = M - m + 1,\quad S_y = N - n + 1.\\[6pt] &\text{Define the kernel reversed for convolution:}\qquad k^\ast_{a,b} = k_{\,m-1-a,\;n-1-b}.\\[6pt] &\text{Then the convolution is:}\\[4pt] &\qquad (v * k)_{i,j} = \sum_{a=0}^{m-1}\sum_{b=0}^{n-1} k^\ast_{a,b}\; v_{\,i+a,\; j+b}, \qquad 0 \le i < S_x,\; 0 \le j < S_y. \end{aligned} $$$ \begin{aligned} &\text{Let } v \in \mathbb{R}^{M \times N},\quad k \in \mathbb{R}^{m \times n}.\\[4pt] &\text{Define the output size:}\qquad S_x = M - m + 1,\quad S_y = N - n + 1.\\[6pt] &\text{Define the kernel reversed for convolution:}\qquad k^\ast_{a,b} = k_{\,m-1-a,\;n-1-b}.\\[6pt] &\text{Then the convolution is:}\\[4pt] &\qquad (v * k)_{i,j} = \sum_{a=0}^{m-1}\sum_{b=0}^{n-1} k^\ast_{a,b}\; v_{\,i+a,\; j+b}, \qquad 0 \le i < S_x,\; 0 \le j < S_y. \end{aligned} $$

```
t0 = time.time()

# SOLUTION-BEGIN
s1 = np.array(v.shape)
s2 = np.array(k.shape)
s_out = s1 - s2 + 1
v_conv1 = np.empty(s_out)
for i in range(s_out[0]):
    for j in range(s_out[1]):
        v_conv1[i, j] = np.sum(np.flip(k) * v[i : i + s2[0], j : j + s2[1]])
# SOLUTION-END

print("Execution time: %1.2e s" % (time.time() - t0))

img = plt.imshow(v_conv1, cmap="gray", vmin=vmin, vmax=vmax)
plt.axis("off")
```

2. Convolution through DFT

Besides the filtered image, plot the DFT of the original image, of the kernel and of the filtered image.

$ \widehat{\mathbf{v} \circledast \mathbf{k}} = \widehat{\mathbf{v}} \odot \widehat{\mathbf{k}} $$ \widehat{\mathbf{v} \circledast \mathbf{k}} = \widehat{\mathbf{v}} \odot \widehat{\mathbf{k}} $

Where:

- $\hat{\mathbf{a}}$$\hat{\mathbf{a}}$ is the DFT of the signal $\mathbf{a}$$\mathbf{a}$
- $\circledast$$\circledast$ is the circular convolution
- $\odot$$\odot$ is the Hadamard product

```
from numpy.fft import ifft2, fft2, fftshift, fftfreq

t0 = time.time()

# SOLUTION-BEGIN
v_fft = fft2(v)
k_fft = fft2(k, v.shape)

vk_fft = v_fft * k_fft

v_conv2 = np.real(ifft2(vk_fft))
# SOLUTION-END

print("Execution time: %1.2e s" % (time.time() - t0))

fig, axs = plt.subplots(2, 2, figsize=(10, 8))
img = axs[0, 0].imshow(fftshift(np.log10(np.absolute(v_fft))))
fig.colorbar(img, ax=axs[0, 0])
img = axs[0, 1].imshow(fftshift((np.absolute(k_fft))))
fig.colorbar(img, ax=axs[0, 1])
img = axs[1, 0].imshow(fftshift(np.log10(np.absolute(vk_fft))))
```

```
fig.colorbar(img, ax=axs[1, 0])
axs[1, 1].imshow(v_conv2, cmap="gray", vmin=vmin, vmax=vmax)
plt.axis("off")
```

3. Using the command `scipy.signal.convolve`

Check out the documentation.

```
t0 = time.time()

# SOLUTION-BEGIN
v_conv3 = signal.convolve(v, k, mode="same")
# SOLUTION-END

print("Execution time: %1.2e s" % (time.time() - t0))

img = plt.imshow(v_conv3, cmap="gray", vmin=vmin, vmax=vmax)
plt.axis("off")
```

Let's make a comparison with the JAX low-level API: LAX

```
import jax


def convolve_same_xla_2d(v, k):
    # Reshape kernel for 2D convolution: [filter_height, filter_width, in_channels, out_channels]
    kernel = k[:, :, None, None]  # Add input and output channel dimensions
    # Reshape input for 2D convolution: [batch_size, height, width, in_channels]
    input_array = v[None, :, :, None]  # Add batch and channel dimensions

    # Calculate padding for "same" mode
    pad_height = (k.shape[0] - 1) // 2
    pad_width = (k.shape[1] - 1) // 2
    padding = [(pad_height, pad_height), (pad_width, pad_width)]  # symmetric padding

    # Perform 2D convolution with XLA
    result = jax.lax.conv_general_dilated(
        input_array,  # Input array
        kernel,  # Kernel
        window_strides=(1, 1),  # Stride of 1 for both dimensions
        padding=padding,  # "Same"-like padding
        dimension_numbers=("NHWC", "HWIO", "NHWC"),  # Data and kernel format
    )

    # Reshape the result to a 2D array
    return result[0, :, :, 0]  # Remove batch and channel dimensions


vj = jax.numpy.array(v, dtype=jax.numpy.float32)
kj = jax.numpy.array(k, dtype=jax.numpy.float32)


t0 = time.time()
v_conv4 = convolve_same_xla_2d(vj, kj)
print("Execution time: %1.2e s" % (time.time() - t0))

np.abs(v_conv4 - v_conv3).max()
```

Handwriting recognition

```
import numpy as np
import matplotlib.pyplot as plt
import jax.numpy as jnp
import jax

from flax import linen as nn
from flax.training import train_state
import optax
```

Data import and visualization

Import the MNIST train dataset (https://en.wikipedia.org/wiki/MNIST_database)

```
# This dataset is contained in the sample data directory of Google Colab online runtimes
data = np.genfromtxt("./mnist_train_small.csv", delimiter=",")
data.shape
```

Store the data in a 4-th order tensor (samples, x-pixel, y-pixel, channels) and the labels in a vector. **NOTE:** The labels are the first column of the data matrix

```
# SOLUTION-BEGIN
labels = data[:, 0]
x_data = data[:, 1:].reshape((-1, 28, 28, 1)) / 255
labels.shape, x_data.shape
# SOLUTION-END
```

Visualize the first 30 pictures with the corresponding labels

```
# SOLUTION-BEGIN
fig, axs = plt.subplots(ncols=10, nrows=3, figsize=(20, 6))
axs = axs.reshape((-1,))
for i in range(30):
    image_i = x_data[i]
    axs[i].imshow(image_i, cmap="gray")
    axs[i].set_title(int(labels[i]))
    axs[i].axis("off")
# SOLUTION-END
```

Create a one-hot representation of the labels, that is a matrix where each row corresponds to a class (i.e. a digit). the entries of the matrix are 1 if the sample corresponds to that digit, 0 otherwise.

```
# SOLUTION-BEGIN
labels_onehot = np.zeros((20000, 10))
for i in range(10):
    labels_onehot[labels == i, i] = 1
# SOLUTION-END
```

Check that the matrix has exactly one element "1" in each row.

```
# SOLUTION-BEGIN
row_sums = np.sum(labels_onehot, axis=1)
row_sums.min(), row_sums.max()
# SOLUTION-END
```

ANN training

Define the architecture of the neural network. For more details on CNNs see https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

```
class CNN(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = nn.Conv(features=32, kernel_size=(3, 3))(x)
        x = nn.relu(x)
        x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
        x = nn.Conv(features=64, kernel_size=(3, 3))(x)
        x = nn.relu(x)
        x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
        x = x.reshape((x.shape[0], -1))  # Flatten
        x = nn.Dense(features=256)(x)
        x = nn.relu(x)

        # The `softmax_cross_entropy` expects unnormalized logits.
        # There is also the `softmax_cross_entropy_with_integer_labels`
```

```
        # version that uses integers target labels.
        # If you apply a softmax first, you turn logits into probabilities, and the
        # loss might becomes numerically unstable and incorrect. Optax/JAX expects to
        # handle the softmax internally in a stable way (using logsumexp tricks).
        x = nn.Dense(features=10)(x)  # There are 10 classes in MNIST
        return x


cnn = CNN()

table = cnn.tabulate(
    jax.random.PRNGKey(0), jnp.zeros((1, 28, 28, 1)), console_kwargs={"width": 200}
)

print(table)
```

Write a function to compute the **cross entropy loss** and the **accuracy** of the model given as parameters the unormalized logits (the output of the CNN) and the one-hot encoded target value. To compute the loss you can exploit `optax.softmax_cross_entropy`, check the documentation for the details.

```
def compute_metrics(logits, labels_onehot):
    # SOLUTION-BEGIN
    loss = jnp.mean(optax.softmax_cross_entropy(logits, labels_onehot))
    accuracy = jnp.mean(jnp.argmax(logits, -1) == jnp.argmax(labels_onehot, -1))
    # SOLUTION-END
    return {"loss": loss, "accuracy": accuracy}
```

Here we define functions used for the training

```
@jax.jit
def loss_fn(params, x, y):
    # SOLUTION-BEGIN
    logits = cnn.apply({"params": params}, x)
    loss = jnp.mean(optax.softmax_cross_entropy(logits=logits, labels=y))
    # SOLUTION-END
    return loss, logits


@jax.jit
def train_step(state, x, y):
    # SOLUTION-BEGIN
    grad_fn = jax.value_and_grad(loss_fn, has_aux=True)
    (_, logits), grads = grad_fn(state.params, x, y)
    state = state.apply_gradients(grads=grads)
    metrics = compute_metrics(logits, y)
    # SOLUTION-END
    return state, metrics


def eval_model(state, dataset):
    # SOLUTION-BEGIN
    logits = state.apply_fn({"params": state.params}, dataset["image"])
    metrics = compute_metrics(logits, dataset["label"])
    # SOLUTION-END
    return metrics["loss"], metrics["accuracy"]
```

Define the function for one training epoch

```
def train_epoch(state, train_ds, batch_size, epoch, rng):
    # SOLUTION-BEGIN
    train_ds_size = len(train_ds["image"])
    steps_per_epoch = train_ds_size // batch_size

    perms = jax.random.permutation(rng, len(train_ds["image"]))
    perms = perms[: steps_per_epoch * batch_size]
    perms = perms.reshape((steps_per_epoch, batch_size))

    batch_metrics = []

    for perm in perms:
        batch = {k: v[perm, ...] for k, v in train_ds.items()}
        state, metrics = train_step(state, batch["image"], batch["label"])
```

```
        batch_metrics.append(metrics)

    training_epoch_metrics = {
        k: np.mean([metrics[k] for metrics in batch_metrics])
        for k in batch_metrics[0]
    }

    print(
        f"{epoch:04}  | "
        f"{training_epoch_metrics['loss']:.4e} | "
        f"    {training_epoch_metrics['accuracy'] * 100:.2f} | ",
        end=""
    )

    # SOLUTION-END

    return state, training_epoch_metrics
```

Prepare data by randomizing and creating the train-validation split

```
np.random.seed(0)

# SOLUTION-BEGIN
n_samples = x_data.shape[0]
perm = np.random.permutation(n_samples)
train_perc = 0.8
n_train_samples = int(train_perc * n_samples)
train_idxs = perm[:n_train_samples]
valid_idx = perm[n_train_samples:]
# SOLUTION-END

train_ds = {
    "image": jnp.array(x_data[train_idxs]),
    "label": jnp.array(labels_onehot[train_idxs], dtype=jnp.float32),
}
valid_ds = {
    "image": jnp.array(x_data[valid_idx]),
    "label": jnp.array(labels_onehot[valid_idx], dtype=jnp.float32),
}
```

Run training

```
rng = jax.random.PRNGKey(0)
rng, init_rng = jax.random.split(rng)

# Initialize lists to store metrics for graph visualization
training_losses = []
training_accuracies = []
valid_losses = []
valid_accuracies = []

# Hyperparameters
num_epochs = 10
batch_size = 64
learning_rate = 0.001

# Initialize the CNN and optimizer
# SOLUTION-BEGIN
params = cnn.init(init_rng, jnp.ones([1, 28, 28, 1]))["params"]
tx = optax.adam(learning_rate=learning_rate)
state = train_state.TrainState.create(apply_fn=cnn.apply, params=params, tx=tx)
# SOLUTION-END

print("epoch | train loss | train acc | valid loss | valid acc")
for epoch in range(1, num_epochs + 1):
    # SOLUTION-BEGIN
    # Use a separate PRNG key to permute image data during shuffling
    rng, input_rng = jax.random.split(rng)
    # Run an optimization step over a training batch
    state, train_metrics = train_epoch(state, train_ds, batch_size, epoch, input_rng)
    # Evaluate on the test set after each training epoch
    valid_loss, valid_accuracy = eval_model(state, valid_ds)
    print(
        f"{valid_loss:.4e} | " f"{valid_accuracy * 100:.2f}",
```

```
    )
    # Store metrics for graph visualization
    training_losses.append(train_metrics["loss"])
    training_accuracies.append(train_metrics["accuracy"])
    valid_losses.append(valid_loss)
    valid_accuracies.append(valid_accuracy)
    # SOLUTION-END
```

Testing

Load the dataset `sample_data/mnist_test.csv` and normalize and shape the data.

```
data_test = np.genfromtxt("./mnist_test.csv", delimiter=",")
data_test.shape
labels_test = data_test[:, 0]
x_test = data_test[:, 1:] / 255

labels_onehot_test = np.zeros((x_test.shape[0], 10))
for i in range(10):
    labels_onehot_test[labels_test == i, i] = 1.0


test_ds = {
    "image": jnp.array(x_test.reshape((-1, 28, 28, 1))),
    "label": jnp.array(labels_onehot_test),
}
```

Compute the accuracy of the classifier on this dataset.

```
# SOLUTION-BEGIN
test_loss, test_accuracy = eval_model(state, test_ds)
print(f"Loss: {test_loss:.2e}")
print(f"Accuracy: {test_accuracy * 100.:.2f}%")
# SOLUTION-END
```

Use the following script to visualize the predictions on a bunch of test images.

```
offset = 0
n_images = 40

images_per_row = 10
y_predicted = state.apply_fn({"params": state.params}, test_ds["image"])


def draw_bars(ax, y_predicted, label):
    myplot = ax.bar(range(10), (y_predicted))
    ax.set_ylim([0, 1])
    ax.set_xticks(range(10))

    label_predicted = np.argmax(y_predicted)
    if label == label_predicted:
        color = "green"
    else:
        color = "red"
    myplot[label_predicted].set_color(color)


import math

n_rows = 2 * math.ceil(n_images / images_per_row)
_, axs = plt.subplots(n_rows, images_per_row, figsize=(3 * images_per_row, 3 * n_rows))
row = 0
col = 0
for i in range(n_images):
    axs[2 * row, col].imshow(x_test[offset + i].reshape((28, 28)), cmap="gray")
    axs[2 * row, col].set_title(int(labels_test[offset + i]))
    axs[2 * row, col].axis("off")

    draw_bars(
        axs[2 * row + 1, col], jax.nn.softmax(y_predicted[i]), labels_test[offset + i]
    )
```

```
        col += 1
        if col == images_per_row:
            col = 0
            row += 1
```

Adversarial attacks

You have trained your classifier. Cool, isn't it? Let us now try to fool it.

An adversarial attack consists of an (almost imperceptible) modification of the image, aimed at fooling the classifier into making a mistake. See e.g. this article

To hack the classifier, compute the gradient of cross entropy loss funcion with respect to the input (not to the parameters!). Then, superimpose a multiple of the gradient to the original image. See e.g. this article.

Namely, follow these steps:

1. Compute the gradient of `loss_fn` with respect to the **input of the CNN (the image)**
2. Compute the perturbed image, by summing $\epsilon \text{ sign}(\nabla_x \texttt{loss_fn})$$\epsilon \text{ sign}(\nabla_x \texttt{loss_fn})$
3. Clip the result in $[0, 1]$$[0, 1]$

Visualize the original and the hacked images and the corresponding prediction of the classifier.

```
import jax
import jax.numpy as jnp
import matplotlib.pyplot as plt


# FGSM attack
def fgsm_attack(params, image, label, epsilon):
    # SOLUTION-BEGIN
    # Compute gradient of loss w.r.t. the input image
    grad_fn = jax.grad(lambda img: loss_fn(params, img, label), has_aux=True)

    gradient, _ = grad_fn(image)

    # Apply perturbation
    adv_image = image + epsilon * jnp.sign(gradient)

    # Clip back to valid pixel range
    adv_image = jnp.clip(adv_image, 0.0, 1.0)
    # SOLUTION-END
    return adv_image


# By trial-and-error I give you the information that for the following images
# an `epsilon = 0.05` is large enough to fool the CNN
epsilon = 0.05
for idx in [11, 66, 115, 244]:
    # SOLUTION-BEGIN
    x = test_ds["image"][idx : idx + 1]
    y = test_ds["label"][idx]

    # True prediction
    logits = cnn.apply({"params": state.params}, x)
    true_pred = jnp.argmax(logits, axis=-1)
    print("Original prediction:", true_pred)

    # Create adversarial example
    x_adv = fgsm_attack(state.params, x, y, epsilon)

    # Prediction on adversarial image
    logits_adv = cnn.apply({"params": state.params}, x_adv)
    adv_pred = jnp.argmax(logits_adv, axis=-1)
    print("Adversarial prediction:", adv_pred)

    # Plot
    plt.figure(figsize=(6, 3))
    plt.subplot(1, 2, 1)
    plt.title(f"Original ({true_pred})")
    plt.imshow(x[0, :, :, 0], cmap="gray")
```

```
    plt.subplot(1, 2, 2)
    plt.title(f"Adversarial ({adv_pred})")
    plt.imshow(x_adv[0, :, :, 0], cmap="gray")

    plt.show()
    # SOLUTION-END
```

```
def get_first_layer_output(cnn_module, params, input_data):
    """
    Manually applies the first convolutional layer and ReLU.
    """
    # 1. Access the first Conv layer from the parameters
    # The name of the first Conv layer is typically 'Conv_0' by default in Flax
    # if it's the first nn.Conv without an explicit name.

    # Instantiate the first Conv layer with the correct features and kernel size
    conv_layer = nn.Conv(features=32, kernel_size=(3, 3), name='Conv_0')

    # Apply the convolution using the parameters (weights and biases)
    # This requires using the `apply` method of the layer and passing the specific subset of parameters
    conv_output = conv_layer.apply({'params': params['Conv_0']}, input_data)

    # 2. Apply ReLU
    final_output = nn.relu(conv_output)

    return final_output

idx = 0
first_layer_output = get_first_layer_output(cnn, state.params, test_ds["image"][idx : idx + 1])
first_layer_output.shape

for i in range(32):
    plt.figure()
    plt.imshow(first_layer_output[0, :, :, i])
```

Handwriting recognition

```
import numpy as np
import matplotlib.pyplot as plt
import jax.numpy as jnp
import jax

from flax import linen as nn
from flax.training import train_state
import optax
```

Data import and visualization

Import the MNIST train dataset (https://en.wikipedia.org/wiki/MNIST_database)

```
# This dataset is contained in the sample data directory of Google Colab online runtimes
data = np.genfromtxt("./mnist_train_small.csv", delimiter=",")
data.shape
```

Store the data in a 4-th order tensor (samples, x-pixel, y-pixel, channels) and the labels in a vector. **NOTE:** The labels are the first column of the data matrix

```
# FILL HERE
```

Visualize the first 30 pictures with the corresponding labels

```
# FILL HERE
```

Create a one-hot representation of the labels, that is a matrix where each row corresponds to a class (i.e. a digit). the entries of the matrix are 1 if the sample corresponds to that digit, 0 otherwise.

```
# FILL HERE
```

Check that the matrix has exactly one element "1" in each row.

```
# FILL HERE
```

ANN training

Define the architecture of the neural network. For more details on CNNs see https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks

```
class CNN(nn.Module):
    @nn.compact
    def __call__(self, x):
        x = nn.Conv(features=32, kernel_size=(3, 3))(x)
        x = nn.relu(x)
        x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
        x = nn.Conv(features=64, kernel_size=(3, 3))(x)
        x = nn.relu(x)
        x = nn.avg_pool(x, window_shape=(2, 2), strides=(2, 2))
        x = x.reshape((x.shape[0], -1))  # Flatten
        x = nn.Dense(features=256)(x)
        x = nn.relu(x)

        # The `softmax_cross_entropy` expects unnormalized logits.
        # There is also the `softmax_cross_entropy_with_integer_labels`
        # version that uses integers target labels.
        # If you apply a softmax first, you turn logits into probabilities, and the
        # loss might becomes numerically unstable and incorrect. Optax/JAX expects to
        # handle the softmax internally in a stable way (using logsumexp tricks).
        x = nn.Dense(features=10)(x)  # There are 10 classes in MNIST
```