Exercise #07

IT University of Copenhagen (ITU)
Data Mining KSD (DAMIN)
(Autumn 2025)

07 October 2025

Introduction This exercise set provides hands-on experience with advanced classification techniques, focusing on Support Vector Machines (SVM) and Neural Networks (NN), including Convolutional Neural Networks (CNN). The tasks are designed to introduce you to core concepts such as linear and non-linear classification, deep learning architectures, and state-of-the-art machine learning frameworks. You will gain practical experience using widely adopted Python libraries, such as Scikit-Learn, TensorFlow, Keras, PyTorch, and PyTorch Lightning. The learning objectives for this exercise list include:

- Understand and implement Support Vector Machines (SVM) using linear and non-linear kernels.
- Explore the decision boundaries of classifiers and the effects of kernel functions on classification performance.
- Build and train simple neural networks and convolutional neural networks (CNN) for image classification tasks.
- Apply data augmentation techniques to improve deep learning model performance.
- Compare traditional model-building techniques in PyTorch with the more streamlined approach offered by PyTorch Lightning.

By the end of this exercise session, you will have developed a deeper understanding of advanced classification techniques, how to implement them in Python, and how to leverage popular machine learning frameworks to solve real-world problems.

Exercise 07.01. *Implementing SVM with Scikit-Learn* (15-20 minutes) – In this exercise, you will implement a linear Support Vector Machine (SVM) classifier using the Scikit-Learn library. SVM is a powerful supervised learning algorithm used for both classification and regression tasks. In this case, you will use SVM to classify data, training the model on one of the datasets provided by Scikit-Learn.

- **Instructions:**

- Load the dataset.
  - Use the `load_iris()` or `load_wine()` dataset from Scikit-Learn.
  - Load the dataset into a Pandas `DataFrame` for easy handling.
  - Split the dataset into training and test sets (80/20 split) using `train_test_split` from Scikit-Learn.
- Train the SVM model.
  - Create a linear SVM classifier using the `SVC(kernel='linear')` function.
  - Train the model on the training data.
- Evaluate the model.
  - Predict the labels for the test set using the trained model.
  - Calculate and print the classification metrics: accuracy, precision, recall, and f1-score.
- Hyperparameter tuning $^{(optional)}$.
  - Use `GridSearchCV` to find the optimal values for the SVM's hyperparameters, such as `C` (regularization) and `kernel`.

- **Expected Output/Questions:**
  1. What is the accuracy, precision, recall, and f1-score of your SVM model?
  2. Did you observe any improvements after hyperparameter tuning (if attempted)?
  3. How does the model perform on both the training and test sets? Are there signs of overfitting or underfitting?

Exercise 07.02. *SVM with Non-linear Kernels* (15-20 minutes) – In this exercise, you will explore the use of non-linear kernels in Support Vector Machines (SVM) to classify data that is not linearly separable. The kernel trick allows SVMs to perform transformations on the input data, enabling the classification of complex datasets. You will use the Scikit-Learn library to implement SVM with non-linear kernels.

- **Instructions:**
  - Load the dataset.
    - Use the `make_moons()` or `make_circles()` dataset from Scikit-Learn.
    - Generate a dataset with at least 500 samples and a moderate amount of noise (e.g., noise=0.2).
    - Split the dataset into training and test sets (80/20 split).
  - Train the SVM model with a non-linear kernel.
    - Create an SVM classifier with a "*Radial Basis Function (RBF) kernel*" using `SVC(kernel='rbf')`.
    - Train the model on the training data.

- Visualize the decision boundary.
  - Plot the decision boundary for the RBF kernel on the test data using `matplotlib`.
  - Compare the decision boundary with that of a "*linear SVM*" by training a linear SVM and visualizing its decision boundary.
- Evaluate the models.
  - Predict the labels for the test set using both the linear and RBF models.
  - Calculate and compare the accuracy, precision, recall, and f1-score for both models.
- Hyperparameter tuning $^{(optional)}$.
  - Use `GridSearchCV` to find the optimal parameters for the RBF kernel, such as the `C` (regularization) and `gamma` (kernel coefficient) values.

- **Expected Output/Questions:**

  1. How does the performance of the non-linear (RBF) SVM compare to the linear SVM?
  2. What is the shape of the decision boundary for the RBF kernel, and why does it differ from the linear kernel?
  3. Did hyperparameter tuning improve the performance of the RBF SVM model (if attempted)?
  4. What are the accuracy, precision, recall, and f1-score scores for both models?

Exercise 07.03. *Building a Simple Neural Network with Keras* (20-25 minutes) – In this exercise, you will build and train a simple feedforward neural network using the Keras library (available through TensorFlow). The neural network will be used for image classification on the widely-used MNIST dataset, which consists of handwritten digit images. This exercise will introduce you to fundamental concepts of neural networks, including layers, activation functions, and model evaluation.

- **Instructions:**
  - Load the dataset.
    - Use the MNIST dataset available in `keras.datasets.mnist`.
    - Split the dataset into training and test sets, normalizing the pixel values to be in the range `[0, 1]`.
  - Build the neural network.
    - Create a neural network model with one hidden layer using the Keras `Sequential()` API.
    - Use `Dense()` layers and the `ReLU` activation function for the hidden layer, with 128 neurons.

        * Add an output layer with 10 neurons (one for each class) and a `softmax` activation function.
    - Compile and train the model.
        * Compile the model using the `categorical crossentropy` loss function, `Adam` optimizer, and `accuracy` as the evaluation metric.
        * Train the model on the training set for 10 epochs with a batch size of 32.
    - Evaluate the model.
        * Evaluate the model on the test set and print the test accuracy.
        * Plot the training and validation accuracy and loss over the epochs using `matplotlib`.
    - $^{(Optional)}$ Modify the network.
        * Add a second hidden layer and retrain the model. Compare the results with the single hidden layer model.
        * Experiment with different activation functions (e.g., `sigmoid` or `tanh`).

- **Expected Output/Questions:**

    1. What is the accuracy of your neural network on the MNIST test set?
    2. How do the training and validation curves for accuracy and loss behave over the epochs? Are there any signs of overfitting or underfitting?
    3. How did adding a second hidden layer affect the model's performance (if attempted)?
    4. What was the impact of experimenting with different activation functions (if attempted)?

Exercise 07.04. *Deep Learning with Convolutional Neural Networks (CNN)* (25-30 minutes) – In this exercise, you will build and train a CNN for image classification using the CIFAR-10 dataset. CNN are particularly effective for tasks involving image data because of their ability to capture spatial hierarchies in images. You will use the Keras library to implement the CNN architecture.

- **Instructions:**
    - Load the dataset.
        * Use the CIFAR-10 dataset available in `keras.datasets.cifar10`.
        * Split the dataset into training and test sets. Normalize the pixel values to the range `[0, 1]`.
    - Build the CNN model.
        * Create a CNN model using the Keras `Sequential()` API.

    * Add two `convolutional layers` using `Conv2D()` with 32 filters and a `3x3` kernel size, followed by `ReLU` activation and `max pooling`.
    * Add a `flattening layer` to convert the 2D output of the convolutional layers into a 1D vector.
    * Add a `fully connected dense layer` with 128 neurons and `ReLU` activation.
    * Use a `softmax` output layer with 10 neurons, one for each class in the CIFAR-10 dataset.
  - Compile and train the model.
    * Compile the model using the `categorical crossentropy` loss function, the `Adam` optimizer, and accuracy as the evaluation metric.
    * Train the model for 20 epochs with a batch size of 64.
  - Evaluate the model.
    * Evaluate the model on the test set and report the test accuracy.
    * Plot the training and validation accuracy and loss over the epochs using `matplotlib`.
  - Data augmentation $^{(Optional)}$.
    * Apply `data augmentation` to the training data (e.g., rotation, flipping, zooming) using `ImageDataGenerator()` from Keras.
    * Retrain the model with augmented data and compare the performance with the original model.

- **Expected Output/Questions:**
  1. What is the test accuracy of your CNN model on the CIFAR-10 dataset?
  2. How do the training and validation curves for accuracy and loss behave over the epochs? Are there any signs of overfitting or underfitting?
  3. Did applying data augmentation improve the model's performance (if attempted)?
  4. How does the performance of your CNN compare with a basic fully connected neural network (if you have experience with that)?

Exercise 07.05. $^{(Optional)}$ *Introduction to PyTorch and PyTorch Lightning* (25-30 minutes) – In this exercise, you will build a simple neural network for image classification using the Fashion MNIST dataset. First, you will implement the network using raw PyTorch and train it manually[1]. Then, you will refactor the same model using PyTorch Lightning to observe how it simplifies the training process and improves code clarity[2]. This exercise will introduce you to PyTorch fundamentals and the benefits of using PyTorch Lightning for deep learning workflows.

[1]See `https://pytorch.org/tutorials/beginner/introyt/trainingyt.html`
[2]See `https://lightning.ai/docs/pytorch/stable/starter/introduction.html`

- **Instructions:**
  - Load the dataset.
    - Use the Fashion MNIST dataset available from `torchvision.datasets`.
    - Normalize the pixel values and split the dataset into training and test sets.
  - Build the neural network in PyTorch.
    - Define a simple neural network using the PyTorch `torch.nn.Module` class with one hidden layer (128 neurons, `ReLU` activation) and a softmax output layer for 10 classes.
    - Use `cross entropy loss` and the `Adam optimizer`.
  - Train the model manually in PyTorch.
    - Write the training loop to perform forward passes, compute the loss, back-propagate gradients, and update weights using the optimizer.
    - Train the model for 10 epochs and track the training and validation accuracy at each epoch.
  - Refactor the model using PyTorch Lightning.
    - Refactor the same network into a PyTorch Lightning module by extending the `LightningModule` class.
    - Implement the `training_step`, `validation_step`, and `configure_optimizers` methods.
    - Train the model using the PyTorch Lightning `Trainer()` class and compare the code clarity and execution speed with the manual PyTorch implementation.

- **Expected Output/Questions:**
  1. What was the test accuracy of your neural network trained with PyTorch and PyTorch Lightning?
  2. How does the manual training loop in PyTorch compare with the PyTorch Lightning implementation in terms of simplicity and readability?
  3. What are the benefits of using PyTorch Lightning, and did you observe any performance differences between the two implementations?
  4. How do the training and validation accuracy curves behave over the epochs?