

# Assignment 3

## Image classification.

IAML 2020

*Note: Please read the whole assignment carefully before starting to solve it. This includes the introductory text which may be updated for each assignment!*

In this assignment, you will experiment with different classification methods, including logistic regression, k-nearest neighbours, support vector machines and neural networks.. The purpose is to give you a practical understanding of how they compare for a specific classification task. You will use neural networks through PyTorch and other algorithms through Scikit-learn. The assignment has three parts where each part is intended for one week.:

1. **Basic models (week 1):** Set up an environment for experimentation and perform initial tests using logistic regression, support vector machines, and k-nearest neighbors.
2. **Neural networks (week 2):** Set up a training environment for neural networks using PyTorch and write an MLP (multi layer perceptron) and a CNN (convolutional neural network). You will compare them to each other and the initial models used.
3. **Tuning (week 3):** Implement a number of improvements to the neural networks. You can choose between experimenting with different optimisers, configuration of layers, regularisation techniques, or data augmentation.

For this assignment you have to write the report as a single coherent text, i.e. it should not be divided into sections based on the assignment parts. The report requirements listed for each exercise should be incorporated where appropriate. We provide a suggested structure at the end of this exercise.

On average, we expect students to use 10 hours for solving the exercises. This time is in addition to time used on lectures, reading, and exercises. This is why it is important that you use the resources available to you to get the help you need. Don't hesitate to contact us by email, on Slack, or on LearnIt - we are here to help you.

## Overview

This assignment revolves around image classification of clothes. You may already know of the ImageNet<sup>1</sup> object classification competition and dataset which has been at the forefront of much of the hype surrounding deep learning today. For this assignment, it would be entirely unrealistic to use ImageNet which contains 14 million images in approximately 20,000 classes. Instead, we use a much simpler dataset that is often used to benchmark classification algorithms and which still allows you to discover and evaluate the differences between algorithms.

<sup>1</sup> Website

## Dataset

For this assignment we use the FashionMNIST dataset to train and test the implemented algorithms. FashionMNIST is a database of clothes article images (from Zalando), consisting of  $28 \times 28$  pixel grayscale images associated with one of ten classes of clothing articles. A total of 60,000 training samples and 10,000 test samples are provided. A small sample of the images sorted by class is shown in Figure 1.



Figure 1: Sample pictures from the FashionMNIST dataset, sorted by class.

FashionMNIST is an excellent starting point since it is not too easy (as you will see) but is still small enough to allow you to train classifiers within a reasonable timeframe. It is also easy to import into your project.

- ⚠ To download the dataset, run the `downloader.py` script in the assignment materials.

## Framework

Because you will experiment with many different classification models, we provide an API for easily dealing with model initialization, training, and evaluation. The abstract class `Trainer` defines this API. You can read the details in `trainer.py`. The two subclasses `SKLearnTrainer` and `PyTorchTrainer` implement the API for the Scikit-learn and PyTorch libraries. The actual classifier (Scikit-learn algorithm or PyTorch module) is passed to the respective class constructors when instantiating objects. You have to implement the

training and evaluation functionality while the basic API for easily saving/loading trained models is provided by us.

For evaluation purposes, we provide a class `MetricLogger`, which enables easy logging and calculation of performance metrics. You will add functionality for computing metrics from logged information in part 1.





### Hand-in

You have to hand in a collection of the assignments at the end of the course. The following elements have to be included:

- **Code:** You have to hand in the complete source code used for the assignments **including any files provided by us**. Use a separate folder for each assignment. This makes it easy for us to test whether your implementations actually works. Upload the code as a **zip** archive (don't use rar).
- **Report:** You must combine your writings for each part into a single report document in **pdf** format. The reports for each part should be approximately 2-3 pages in length (normal pages of 2400 characters). Use clear formatting for assignment numbers. The report should be a single coherent text that is self-contained (no need to look at other files to understand the content) and reproducible (it must be possible for other students to redo what you did from your description).
- **Results:** Include all pictures, videos, and other files produced in a separate **zip** archive. Use a separate folder for each assignment.

### Requirements

We specify a number of requirements for the assignment - these have to be completed to pass the exam. Completed here means attempted and described in the report. However, completing each item is not a guarantee for passing the exam. The final assessment is based on an overall evaluation of your code and report. To make the requirements as clear as possible, we mark tasks with one of the following symbols:

-  The task is required.
-  You have to implement at least one of the specified tasks.
-  The task is entirely optional. You should only focus on these after you completed the other tasks.
-  The task is required for master students but is of course allowed to be attempted by bachelor students as well.

Do not expect that you will always get optimal results. Neither should you focus on optimising your code for speed unless explicitly specified. The assignments are generally open ended and we encourage you to improve your solutions as much as possible.

For questions regarding the general course requirements and differences between bachelor and master students, refer to the official course descriptions.

### *Guidelines*

We provide a few general guidelines here.

- **Be as precise as possible:** Be as precise as possible when explaining your approach, implementation, and results.
- **Reproducibility:** Your results should be reproducible from the report description alone.
- **Use the theory:** Relate your findings to theoretical concepts from the course when relevant. Don't repeat the theory unless specified, use references instead.
- **Remember that you are learning:** We don't expect you to be fluent in all the material. Solve the tasks to the best of your abilities and write what you did so others can reproduce your results from the descriptions.

**Assignment 3.1***Training and evaluating classifiers*

The first part of the assignment is about setting up a base system for training and evaluating machine learning models. You will have to do the following:

1. Implement training and evaluation methods for the `SKLearnTrainer` class in `trainers.py`.
2. Implement metric calculations in the `MetricLogger` class.
3. Create a script for training different models, including a logistic regression model, support vector machine and k-nearest neighbour model.
4. Evaluate the results using a provided Jupyter notebook (`evaluation.ipynb`).



The `SKLearnTrainer` class in `trainers.py` has already been partially implemented but lacks code for training and evaluating its model. The class defines several instance variables for the Fashion-MNIST dataset loaded as Numpy arrays. Specifically, the following instance variables are available:

- `(X_train, y_train)`: For the training images and labels.
- `(X_val, y_train)`: For the validation images and labels.
- `(X_test, y_test)`: For the testing images and labels.

Remember that fields in Python are accessed using the `self.<variable-name>` syntax, for example `self.X_train`.

**Task 1: Model training**

You first have to implement the training method as well as a script for training models.

1.  **Implement the `train()` method:** The method should train the Scikit-learn model `self.model` to the training data set. The models provide a common API with `model.fit(X, y)` being used for training. You may therefore use the method without knowing the exact model used, e.g. logistic regression or support vector machine<sup>2</sup>.
2.  **Write training script:** Create a script `train_sklearn.py`. In it, create an instance of `SKLearnTrainer` with a `LogisticRegression` model. Add calls to the `train()` and `save()` methods to train and save the result.

**Hint:** As some of the models you will use take quite a bit of time to train, it might be helpful to print status information when training.

<sup>2</sup> This is actually an example of Python's duck-typing system. The classes don't implement a common interface but just provide the `fit` method.

**Task 2: Metrics**

You will use three metrics: *accuracy*, *precision*, and *recall* to evaluate the models. Remember from Exercise 9 that these can be derived from the *confusion matrix* of a model. However, since the presented problem contains 10 classes, the matrix for this problem is  $10 \times 10$ .

An example confusion matrix is shown in Figure 2 with the predicted on the y-axis and the true class on the x-axis. The confusion matrix got its name because it reveals which classes the model is typically confused about, i.e. the classes it can't classify correctly. In the example, shirts are highly likely to be misclassified as t-shirts.

t-shirt	296	0	4	1	0	1	17	0	2	0
trouser	38	974	16	101	11	3	27	0	11	2
pullover	19	2	524	18	51	1	66	0	12	0
dress	44	11	4	693	7	0	25	0	9	0
coat	53	7	227	105	771	0	173	0	50	1
sandal	2	0	0	2	0	919	1	71	38	88
shirt	540	4	225	72	159	1	683	0	75	2
sneaker	0	1	0	3	1	45	0	907	8	50
bag	6	1	0	2	0	7	8	0	792	0
ankle boot	2	0	0	3	0	23	0	22	3	857
	t-shirt	trouser	pullover	dress	coat	sandal	shirt	sneaker	bag	ankle boot

Figure 2: Confusion matrix of sample support vector machine. The true class is on the x-axis and the predicted class on the y-axis.

The metrics can be derived from confusion matrix  $C$ , where  $C_{i,j}$  is the number of samples predicted to be class  $i$  and with ground-truth label  $j$ . Below is a description of each metric along with a formula for its calculation:

**Accuracy:** The ratio of correct predictions to the total number of test samples.

$$accuracy = \frac{\sum_{i=1}^{10} C_{i,i}}{\sum_{i=1}^{10} \sum_{j=1}^{10} C_{i,j}}$$

**Precision:** The ratio of correct predictions for a certain class to the number of predictions for that class.




$$precision_i = \frac{C_{i,i}}{\sum_{j=1}^{10} C_{j,i}}$$

**Recall:** The ratio of correct predictions for a certain class to the number of samples belonging to that class.

$$recall_i = \frac{C_{i,i}}{\sum_{j=1}^{10} C_{i,j}}$$

Note that precision and recall are vectors in multi-class classification, i.e. they describe the metric per class.

For this task, we provide a partial implementation of the class `MetricLogger` in `metrics.py`. The constructor initializes the confusion matrix and `reset()` resets the confusion matrix. `log(predicted, target)` adds the provided results to the matrix. The `one_hot` argument in the constructor is added because Scikit-learn provides numerical predictions while PyTorch provides one-hot encoded predictions.








1.  **Implement metrics:** Implement accuracy, precision, and recall as properties of `MetricLogger`.
2.  **Modify the `train` method in `SKLearnTrainer`:** The method should predict values for the validation data (`self.X_val`, `self.y_val`) and log the results using `MetricLogger` and print the calculated metrics (accuracy, precision, recall).
3.  **Implement the `evaluate` method in `SKLearnTrainer`:** The method should predict values for the test data (`self.X_test`, `self.y_test`), log the results using `MetricLogger`, and return the logger object.

**Hint:** It is much simpler to use Numpy's vectorized functions for summation. To easily get the matrix diagonal, use `np.diag`.

**Hint:** You should use `one_hot=False` for the Scikit-learn models as they return predictions as class index values.

### Task 3: Evaluating results

You should now be ready to train and evaluate different models. When running your training-script, the `save()` method automatically creates a file containing the complete model state in the `models` directory. You have to evaluate the models and produce visualisations for use in the report using the `evaluation.ipynb` notebook.

1.  **Train different models:** Train one model for each of the following algorithms:
  -  **Logistic regression:** Use `sklearn.linear_models.LogisticRegression`.
  -  **Support vector machine (linear kernel):** Use `sklearn.svm.LinearSVC`.
  -  **Support vector machine (polynomial kernel):** Use `sklearn.svm.SVC(kernel='poly')`
  -  **K nearest neighbors:** Use `sklearn.neighbors.KNeighborsClassifier`.
  -  Try variations of the above or other models you might have heard of. Use Scikit-learn's user guide for inspiration<sup>3</sup>.
2.  **Produce figures:** The provided Jupyter notebook `evaluation.ipynb` contains functionality for producing various visualisations. Add your models to the dictionary as specified and produce initial results.

**Hint:** The classes all support the `n_jobs` argument in their constructors. It sets the number of parallel processes and may result in significant speedups on multi-core machines.

<sup>3</sup> [link](#)

Note: This model might take quite a while to train with no indication of progress. Expect up to 30 minutes on a low-powered laptop.

### Task 4: Report

Remember the general report requirements presented in the overview. The following is additional requirements:

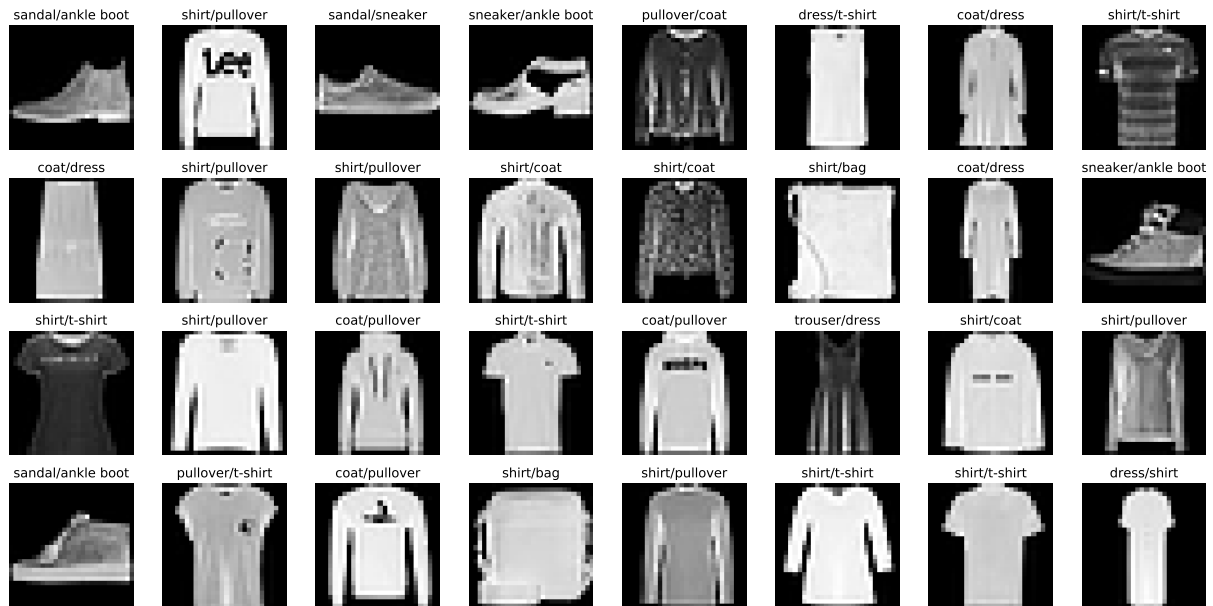





Figure 3: Sample of wrong results. The titles are formatted as <prediction>/<ground-truth>.

-  Explain the most important characteristics of each model used. Remember to be precise and add references when relevant.
-  Present and compare results for the different models.
-  Create bar-graphs for the recorded metrics (accuracy, precision, recall) and include an example confusion matrix and image grid with predictions and labels.



### Assignment 3.2

#### Neural networks

In this part of the assignment you have to implement a number of neural network architectures as well as the code for training them. Neural networks require a bit more setup than the Scikit-learn models but once you have a base implementation of model training, creating new or adapting existing networks is simple.

#### TensorBoard

In Exercise 11 you used simple print-statements to check the status when training a neural network. This is both cumbersome and less than ideal for understanding how the model is progressing. In this assignment, you will instead use TensorBoard which is a utility for visualising data. Although TensorBoard was created for TensorFlow (Google's framework for neural networks), it is supported by PyTorch as well. You will have to install it though. Simply use the following command in a terminal window after activating the `iaml` environment:

```
pip install --user tensorboard
```

TensorBoard is started from the terminal, much like Jupyter, by typing the following command:

```
tensorboard --logdir=runs
```

You may change the location of where to read logs from by changing `--logdir` but we use `runs` in this assignment because it is the standard logging location for PyTorch. You can then open TensorBoard by opening the address `localhost:6006` in a browser. The interface is shown in Figure 4.

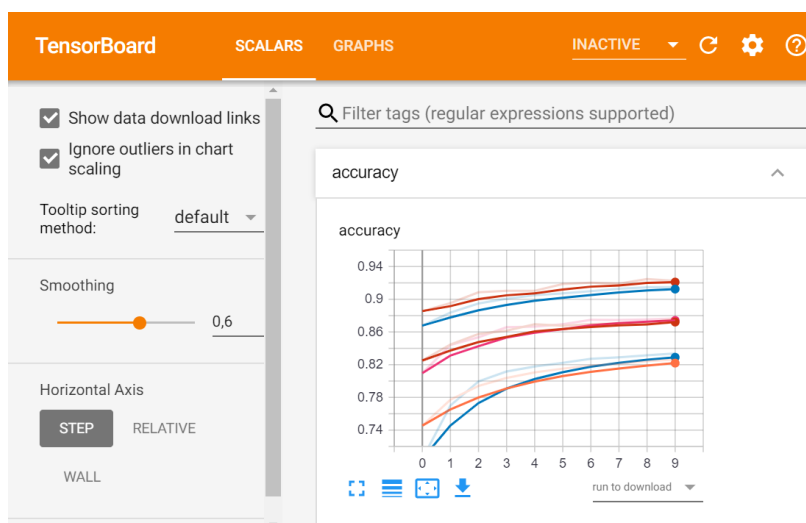


Figure 4: Screenshot of the TensorBoard interface. It shows validation accuracies for a number of models used when the assignment was created.

Task 1: Implement `PyTorchTrainer`

Your first task is to implement the `train()` method for the `PyTorchTrainer` class as you did with `SKLearnTrainer` in the previous assignment part. Again, the class loads the datasets in the constructor. This time, however, PyTorch `DataLoader` instances are used (as in Exercise 11). The following fields are provided:

- `train_data` : For the training images and labels.
- `val_data` : For the validation images and labels.
- `test_data` : For the testing images and labels.

Additionally, the constructor of `PyTorchTrainer` takes a number of extra arguments. Their use is specified in the docstring.

**Hint:** Use Exercise 11 as inspiration.



1. **⚠ Implement the `train(epochs)` method:** The method should train the PyTorch `nn.Module` in `self.model` for the specified number of epochs. For each epoch, iterate through the batches in the training data (`self.train_data`). The optimiser passed to the constructor is available as `self.optimiser`.



2. **⚠ Log metrics:** Create two `MetricLogger` objects in the `train()` method (with `one_hot=False`), one for training data and one for validation data. Log results by using the `log(prediction, label)` method.

**Hint:** Use the `reset()` method to reset the loggers between epochs.

- **⚠ Log training data results for each batch.**
- **⚠ Log validation results at the end of each epoch by iterating through the validation data `self.val_data`.**



3. **⚠ Logging to TensorBoard:** Use the method `self.logger.add_scalar(metric, scalar, step)` to log training results to TensorBoard. `metric` is a string used to identify the metric, `scalar` is the actual value, and `step` is the step value. Since scalars are shown as graphs in TensorBoard, `step` is effectively the x-axis. You have to log the following metrics:

- **⚠ Training metrics:** Log the training loss and accuracy. To avoid having too many samples, only log to TensorBoard at some fixed iteration interval (see Exercise 11 for inspiration).
- **⚠ Validation metrics:** Log the validation accuracy at the end of each epoch.



4. **⚠ Implement the `evaluate()` method:** The method should predict values for the test data `self.test_data`. Then log the results using `MetricLogger` and return the logger object.



5. **⚠ Write training script:** Create a script `train_pytorch.py`. In it, create an instance of `PyTorchTrainer` and train and save it as you did for the `SKLearnTrainer` in part 1. Use the following for the constructor parameters:

- `module`: Use the `Linear` module from Exercise 11.
- `transform`: Use `torchvision.transforms.ToTensor()`. It converts the loaded images to PyTorch tensors.
- `optimizer`: Use an instance of `torch.optim.SGD`. Use a learning rate of 0.01.
- `batch_size`: Use 128.

### Task 2: Implement modules

You now have a setup that allows simple swapping of PyTorch modules, optimisers, and preprocessing operations (transforms). In this task, you will implement both a simple multilayer perceptron (MLP) as well as a convolutional neural network (CNN).

Use `networks.py` to implement the modules. The script imports a number of useful modules for easy access to layers (`torch.nn`) and functions (`torch.nn.functional`) (imported as `F`).



1. **▲ Create MLP model:** Implement a PyTorch `nn.Module` which implements an MLP. The model should have two linear layers `nn.Linear`<sup>4</sup> and should use the ReLU activation function (`F.relu`<sup>5</sup>) for the first layer. There are a number of things to keep in mind when implementing the module:

- The input to the `forward()` method is a tensor of size  $B \times 1 \times 28 \times 28$ , where  $B$  is the batch size. To convert the images to vectors, use `torch.flatten(x, start_dim=1)`.
- As shown in Exercise 11 and explained in the lectures, the sigmoid activation of the last layer is calculated as part of the loss function to ensure numerical stability. You therefore do not need to add an activation to the second layer in the MLP.
- The input layer must have the same number of features as the image vectors (784). The output layer must have the same number of features as classes (10). The hidden layer can have an arbitrary number of features.



2. **▲ Create CNN model:** Implement a module which implements a simple CNN. The module should have two convolutional layers (`nn.Conv2d`<sup>6</sup>) and two linear layers. Use ReLU for activations. You should use a single maxpooling operation `F.max_pool2d`<sup>7</sup> after the convolutional layers have been applied. A diagram of the model is shown in Figure 5. There are a number of things to keep in mind when implementing the module:

- The convolutional and linear layers need to be defined in the module constructor, otherwise, their weights will be reset on each use.
- The output of the convolutional layers has to be flattened before it can be input to a linear layer. Do this using `torch.flatten` as described for the MLP.

<sup>4</sup> [Link to PyTorch docs.](#)

<sup>5</sup> [Link to PyTorch docs.](#)

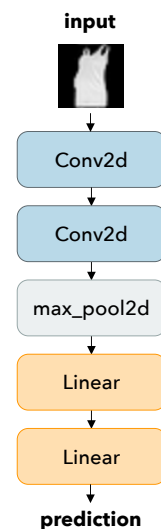


Figure 5: Diagram for the layer configuration for the CNN network.

<sup>6</sup> [Link to PyTorch docs.](#)

<sup>7</sup> [Link to PyTorch docs.](#)

- In this implementation, the convolutions change the size of the output feature maps. After a convolution, the new image size is  $s - k_{size} + 1$  where  $s$  is the width/height of the input and  $k_{size}$  is the width/height of the convolution kernel.
- You may choose the number of channels to use for the convolutions and linear layers, but they have to be equal between adjacent layers. For the first linear layer, the number of features needs to be calculated from the size of the output of the max-pool operation. Use the formula  $whc$  where  $w$  is the width,  $h$  is the height, and  $c$  is the number of channels of the output (note that max-pooling halves the width/height).



3. **Train models:** Train the models using `PyTorchTrainer` and the training script you created ( `train_pytorch.py` ). Experiment with different numbers of hidden units in the MLP and different numbers of channels in the CNN.

Train at least two versions of your CNN and MLP models. Use different values for the number of hidden units (MLP) and channels

### Task 3: Report



- Explain how MLPs and CNNs compare and when/why CNNs are better.



- Present and compare the results (the metrics) for the different models as well as the models used in part 1. Specifically, compare the MLP with the CNN and comment on the effect of varying the number of hidden units/channels.



- Create visualisations as for part 1 (you may combine the graphs).



- Discuss how the results relate to what you know of MLPs and CNNs.




- Discuss how the neural networks compare to the models used in part 1. Make an argument for which approach is best suited to the particular task studied in this assignment.


**Assignment 3.3***Tuning*

For the final assignment part you will use different tricks to improve your models. We present four possible strategies. **You must implement two of these.**

*Task 1: Experimenting with optimisers*

⚠ In this task you have to experiment with using different optimisers and evaluate their impact on model results and convergence speed. There exists a several algorithms for optimising neural networks. So far you have used stochastic gradient descent (SGD). It is possible to add terms to SGD such as momentum which may decrease the number of iterations needed for convergence. Other algorithms use more complex manipulations of the gradient to achieve faster and more stable convergence<sup>8</sup>.



 **Code:** You have to experiment with adding momentum to the SGD optimiser and using the RMSProp optimiser. Train and evaluate on your CNN network.

-  **Report:** At a high level of abstraction, explain how momentum and the RMSProp algorithm works (use references). Update your figures with the new results and comment on the impact of the optimiser changes to the metrics. Comment on how fast the different optimisers converge.

<sup>8</sup> The [youtube video](#) used for lecture 10 contains an overview of these methods. Additional information can be found on the Wikipedia pages for [momentum](#) and [RMSProp](#).

*Task 2: Experimenting with layer configurations*

⚠ The architecture of neural networks has a high impact on how well they perform certain tasks. You have already been presented with CNNs as being capable of capturing the hierarchical information in images. The depth of a CNN directly defines how many levels of a hierarchy it can represent. In this task, you have to increase the depth of your CNN implementation and experiment with the configuration of convolutional, max-pooling, and linear layers.

-  **Code:** Experiment with various layer configurations for your CNN module. Report at least two specific configurations and their results.
-  **Report:** Add the new models to your figures. Comment on the effect the number of layers has on the result metrics.

*Task 3: Regularisation*

⚠ In this task you have to experiment with techniques for regularisation. As mentioned in the [youtube video](#) used for lecture 10, regularisation covers methods that constrain the optimisation problem to counteract overfitting. You have to add dropout layers to your CNN and implement early stopping for the training algorithm.

- **Code:** Add dropout layers to your CNN. Experiment with the number of dropout layers and the  $p$  value. Implement early stopping for the `train` method in `PyTorchTrainer` - use the loss of the validation set to determine when to stop.
- **Report:** Produce new figures for the model results with regularisation. Comment on the effect of dropout and early stopping on the results.

**Hint:** PyTorch provides the layer `nn.Dropout2d` ([link](#)) for convolutional layers and `nn.Dropout` ([link](#)) for flat layers.

#### Task 4: Data augmentation

⚠ In this task you have to implement data augmentation for your CNN using PyTorch's `torchvision` library. Data augmentation involves randomly transforming training samples (using affine transformations). This effectively creates new training samples for free which can reduce overfitting effects<sup>9</sup>.

In `train_pytorch.py` you currently have a transformation that transforms the input image (in PIL format - another image library for Python) to a PyTorch tensor. You have to extend this to include random transformations. Use the function `torchvision.transforms.RandomAffine` to randomly transform the images<sup>10</sup>. You can combine multiple transformations (e.g. the affine transform and the `ToTensor` transform) using `torchvision.transforms.Compose`.

<sup>9</sup> A short introduction to the topic can be found in chapter 7.4 of *Deep Learning* ([link](#)).

<sup>10</sup> [Link to docs](#).

- **Code:** Experiment with various parameters for the random affine transformation. Train your CNN network on the transformed datasets and evaluate the results.
- **Report:** Present your results. Discuss how the data augmentation impacted the results and how it changed how fast training converges.

## Report

For this assignment, you must structure the report as a single coherent text, i.e. it should **not** be divided into sections based on the assignment parts. This is more logical since you have to compare results across individual exercises. The report requirements specified for each exercise should be incorporated in the appropriate sections. We suggest the following structure:

1. **⚠ Introduction:** Write a short introduction stating the goal of the assignment and an overview of the content.
2. **⚠ Background:** Write about the machine learning methods used in general terms. Compare the models and discuss how suited they are to the classification task.
3. **⚠ Experiments:** Write about the experiments performed and how they are evaluated.
4. **⚠ Results and evaluation:** Present and discuss the results. Consider the following:
  - **⚠** Present the results of the experiments performed in the assignment.
  - **⚠** Discuss the influence of model choice and training method on the results. Compare with the expectations presented in the background section.