Convolutional Neural Networks (CNN) have been proven effective for many types of image classification problems, where the desired output is a vector of class scores (or probability). A completely different type of problem we can tackle with a CNN is regression. In this case, the output is a high-dimensional vector. Highly non-linear mappings between input and output vectors can be trained in this way, with applications in various domains.

In this lab session, we will look at a typical example used in academics: an autoencoder CNN. We will first feed a grayscale image to our CNN, and the ideal output is the same image. The interesting feature of an autoencoder is its capacity to recover the image even when the latent space in the hidden units is low-dimensional. In other words, a compressed representation of the input images is learned. This can only work when trained on specific types of images, hence we cannot directly use this as a general image compression codec. This example is mostly useful to learn the topology of a typical regression CNN.

A second example we will look at is a denoising autoencoder. In this case, the input will be corrupted by additive noise, and the desired output is the noiseless version. We will show that with a similar architecture as before, an effective non-linear mapping between noisy/noiseless features can be learned.

1 Autoencoder CNN

In this exercise, you will build a basic autoencoder to encode/decode the MNIST dataset. This time, we will work with full-scale images of 28x28 pixels.

Step 1: Load dataset Run the code to load the dataset, split train/test and convert the data to float32 representation in the range [0-1].

Step 2: Build an autoencoder model using Keras interface In this step, you need to call the functions from the Keras interface to build the autoencoder.

- Create a `Sequential` model to start building the network.

- Add a 2D convolution layer [1] with size 3×3, 16 channels padding '`same`' and a ReLU activation function.
- Add a 2D max pooling layer [2] with size 2x2.
- Repeat this process until the resolution is reduced to $4times4$ pixels.
- Add another 2D convolution layer.
- Increase the resolution with a 2D upsampling layer [3]
- Repeat until the resolution is $16times16$ pixels. For the next convolution layer, use the '`valid`' padding to trim th result to $14times14$, then upscale to get to $28times28$.

  item Add one last convolution, with 1 channel and a sigmoid activation function.
- Compile the model with the `binary_crossentropy` loss function and `Adagrad` optimizer.

Step 3: Train the model Train the model using `model.fit` function, with `x_train` as both the input and desired output. Train for 10 training epochs with batch size of 100 samples.

Step 4: Visualize the result Complete the code to run the model on the test data, and visualize the input/output relationship for the first 10 examples.

2 Denoising autoencoder CNN

In this exercise, we will show that a similar CNN can recover an image corrupted by additive Gaussian noise.

Step 1: Load dataset and corrupt input Run the code to load the dataset and scale pixel values to [0-1]. Add Gaussian noise with $\sigma = 0.2$ to every pixel in the dataset.

[1] `https://keras.io/layers/convolutional/#conv2d`
[2] `https://keras.io/layers/pooling/#maxpooling2d`
[3] `https://keras.io/layers/convolutional/#upsampling2d`

Step 2: Build a denoising autoencoder model using Keras interface The architecture is similar to the previous exercise, but this time, you can stop reducing the image when it gets to 7×7. Use 32 channels for every convolution layer, to increase the dimensions of the feature maps.

Step 3: Train the model Train the model using `model.fit` function, with `x_train_noisy` as the input and `x_train` as the desired output. Train for 15 training epochs with batch size of 100 samples.

Step 4: Visualize the result Complete the code to run the model on the test data, and visualize the input/output relationship for the first 10 examples, and compare with the noiseless test data.