INF5400 — MACHINE LEARNING FOR IMAGE ANALYSIS
DEPARTMENT OF INFORMATICS, UNIVERSITY OF OSLO
2019

Exercise week 12: Unsupervised learning

In this exercise, you are to implement autoencoders, a variational autoencoder, and t-SNE in PyTorch.

This assignment text will tell you what to do, but not how to do it. If you are stuck, you can look at the accompanying solution proposal. If you want, you can of course experiment by changing configurations, the ones specified here are known to work, but are probably far from ideal.

Task 1: Autoencoders

In this task, you should implement a basic autoencoder, and test it out on MNIST images. You should implement three versions, a "compression autoencoder", a "denoising autoencoder", and a "sparse autoencoder" (see the lecture slides [1]) All variants can share the same configurations and network architecture, and differs only in their particularities.

The network consist of four fully-connected layers: 784 nodes in the input layer ($784 = 28 \times 28$, which is the spatial dimension of MNIST images), 128 nodes in the first hidden layer of the encoder, 32 nodes in the coding layer (or latent layer), 128 nodes in the first hidden layer of the decoder, and 784 nodes in the output layer. You can use sigmoid activation functions on all layers (including the last).

For the reconstruction loss, you can use the mean squared error between the network output and the network input. Use the Adam optimization method with a learning rate of 0.01, and train the network for 50 epochs (this should be sufficient)

[1] https://www.uio.no/studier/emner/matnat/ifi/IN5400/v19/material/week12/slides_in5400_s19_week12.pdf

using a batch size of 256. Also, remember to store checkpoints of the training progression as you should run inference on trained models.

Task 1.1: Compression autoencoder

Implement a compression autoencoder. A compression autoencoder has the characteristic bottleneck structure, and is what was explained above.

Task 1.2: Denoising autoencoder

Use your autoencoder to denoise images. You can use the same implementation as in the compression autoencoder, only that you add some noise to the input examples, and compare the reconstructed images with the corresponding images without noise. For generating noisy images, you could sample random vectors of the same size as the input image, and add them to the the input image. Normal distributed with a mean of 0 and standard deviation of 0.2 is sufficient for this demonstration (assuming the values of the input images has been scaled to be in the range of 0 to 1).

Task 1.3: Sparse autoencoder

Implement the sparse autoencoder version. For this, you need to add an additional regularizer to the cost function, otherwise the network should be the same. For the sparsity (ρ in the lecture slides) you can use a value of 0.01.

Task 2: Variational autoencoder

Implement a variational autoencoder. You can reuse most of the code used for the standard autoencoders. Remember that the encoder is used to produce the mean and standard deviation of a standard normal distribution.

As in the autoencoder, we use a simple fully connected network. The encoder should have 784 nodes in the input layer, 128 nodes in the first hidden layer, and 64 nodes in the second hidden layer. In the latent layer, use 10 nodes for the mean vector and 10 nodes for the standard deviation vector. Sample a vector of size 10 from a standard normal distribution, and use it together with the mean and standard deviation from the latent layer. This random vector of size 10 is then the input to the first hidden layer of the decoder. Use 64 nodes for the first hidden layer of the decoder, 128 nodes for the second hidden layer in the decoder, and 784 nodes for the output layer of the encoder. Use relu activations everywhere, except for the latent layer where you use the identity function, and the final layer of the decoder where you use the sigmoid activation.

For the reconstruction loss, you can use the mean of the pixelwise cross entropy between the network output and the network input. For the regularising latent loss, you can use a regularisation strength of 0.001, so that `loss = reconstruction_loss + 0.001 * latent_loss`. Use the Adam optimisation method with a learning rate

of 0.0001, and train the network for 50 epochs using a batch size of 128. Try to use a trained variational autoencoder to reconstruct the input (as with the standard autoencoders), generate new examples, and interpolate between examples.

Task 3: t-SNE

Implement t-SNE and apply it on a subset of MNIST. Below follows a brief walkthrough with pseudocode. First, load some MNIST images

```
data_loader = torch.utils.data.DataLoader(
    torchvision.datasets.MNIST(
        data_path, # TODO: Specify num_images
        train=True,
        download=True,
        transform=torchvision.transforms.Compose([torchvision.transforms.ToTensor()]),
        ),
    batch_size=num_images, # TODO: Specify num_images
    shuffle=False,
    )
images, labels = next(iter(data_loader))
images = images.view(-1, 28 * 28)
```

Then, initialise the set of map points (the 2D points corresponding to the input images) at random. Note that these are the variables that are updated each iteration of the method.

```
map_points = torch.randn(
    num_images, # TODO: Specify num_images (same as above)
    2,
    device=device, # TODO: Specify device, normally 'cpu' or 'cuda:0'
    dtype=torch.float,
    requires_grad=True,
    )
```

Then, simply run the method as long as you like (`num_iterations`).

```
for iteration in range(1, num_iterations + 1): # TODO: Specify num_iterations
    p_ij = symmetric_gauss_neighbour_probability() # TODO: Implement this
    q_ij = symmetric_student_t_neighbour_probability() # TODO: Implement this
    loss = (p_ij * (p_ij/q_ij).log()).sum()
    loss.backward()

    with torch.no_grad():
        map_points -= learning_rate * map_points.grad # TODO: Specify learning_rate
        map_points.grad.zero_()
```

Task 3.1: Implement t-SNE

Implement the functions computing `p_ij` and `q_ij`. For simplicity, use a fixed $\sigma_i = \sigma$ for all datapoints. (Implementing binary search to find an appropriate σ_i is a bit

cumbersome and is not the main focus of this exercise, but if you have time, you can of course try to implement it.)

Test your implementation with 1000 images, a learning rate of 10, and a fixed $\sigma = 10$. See `tsne_sgd_animation.gif` at the materials page for week 12 for an illustration of the first 700 steps.

Task 3.2: Substitute optimisation method

Substitute the stochastic gradient descent update with an Adam optimizer, and test it with a learning rate of 0.1 (feel free to use the pattern `optimizer = torch.optim.Adam([map_points], lr=learning_rate)`). See `tsne_adam_animation.gif` at the materials page for week 12 for an illustration of the first 700 steps.