Computational MR imaging
Laboratory 9: Machine learning in MRI and neural network architecture design

Code submission is due by 12:00 before the next Thursday lab section. Please upload your code to StudOn in a described format. Late submissions will not be accepted.

Learning objectives

- Get familiar with designing and training neural networks in Pytorch
- Examine effects of dropping data on neural network performance
- Learn about effects of over and underfitting

1. Diffusion Data Classification

The goal is to classify labels of brain tissue pixel by pixel based on quantitative diffusion parameters.

1.1. Prepare training data.

1.1.1. Load data by calling, load_data_ex1, which is the DTI data from the human connectome database.

1.1.1.1. train_data: [nSamples, nFeatures]

1.1.1.1.1. This data contains train and validation set for training

1.1.1.2. test_data: [nSamples, nFeatures]

1.1.1.2.1. This data contains test set

1.1.1.3. Each data sample contains the following entries: sample, row, column, slice, T1w, FA (Fractional Anisotropy), MD (Mean Diffusivity), AD (Azial Diffusivity), RD (Radial Diffusivity), Label (class)

1.1.1.4. The label consists of

- Label 1: Thalamus
- Label 2: Corpus callosum (CC)
- Label 3: Cortical white matter (Cortical WM)

1.1.2. Implement a method, ex1_extract_features. What features would you use for training from the data? Look at the lecture slides p.37-38.

1.1.2.1. The labels should be [0 (Thalamus), 1 (CC), 2 (Cortical WM)].

1.1.2.2. Extract features for the train and the test data from the raw data. Within the train data, it will be split into training and validation set later.

1.1.3. Implement methods, ex1_get_[nSamples,nFeatures,nLabels] to return the corresponding ouput.

1.1.4. Data preparation

1.1.4.1. Implement a method ex1_normalization that normalizes the train and test data to be in a same data range for each feature.

1.1.4.1.1. The method normalizes each feature of train and test datasets using the maximum value of the train dataset. This ensures that the same normalization factor is applied to both sets,

maintaining consistent scaling and preventing discrepancies between the two datasets.

1.1.4.1.2. Normalize the feature-extracted train and test data.

1.1.4.2. Implement a method ex1_split_train_val that splits the train data into training and validation sets.

1.1.4.2.1. Round up the number of train samples, *nTrainSamples,* by muiltiplying the total number of samples by the *train_ratio.*

1.1.4.2.2. The first nTrainSamples are used for the training set, and the remaining samples are used for the validation set.

1.1.4.2.3. Split the train data into the training and validation set

1.1.4.3. Implement a method ex1_gen_dataloader that generates a dataloader for the input_data and label_data.

1.1.4.3.1. The dtype of the input_data should be the float dtype.

1.1.4.3.2. The dtype of the label_data should be the long dtype.

1.1.4.3.3. Use data_utils.TensorDataset and data_utils.DataLoader.

1.1.4.3.4. Transfer the data to the self.device

1.1.4.3.5. Make sure the dataloader is flexible to the shuffle option.

1.1.4.3.6. Generate each dataloader for training, validation, and test set. Do not shuffle for the validation and test dataloaders.

1.2. Implement a method ex1_build_model that builds a fully connected neural network following the rules below.

1.2.1. # of the input features of the first layer: nFeatures

1.2.2. # of the input and output features of hidden layers: nHiddenFeatures

1.2.3. # of the output features of the last layer: nLabels

1.2.4. # of hidden layers: nLayers

1.2.5. Activation function: ReLU

1.2.6. Use nn.Sequential(*[]) to defind the sequential model at the end

1.2.7. Transfer the model to the self.device

1.2.8. Define the model with 100 nHiddenFeatures and 3 nLayers.

1.3. Implement methods get_loss and get_optimizer that define a loss function and an optimizer, correspondingly.

1.3.1. get_loss

1.3.1.1. loss_type can be CrossEntropyLoss or BCELoss (for Task2)

1.3.1.2. Return CrossEntropyLoss or BCELoss corresponding to loss_type

1.3.2. get_optimizer

1.3.2.1. Optimizer: Adam optimizer

1.3.3. Define a loss function and an optimizer

1.3.3.1. Learning rate for the optimizer: 0.001

1.4. Implement a method ex1_trainer that processes training and validation steps.

1.4.1. Implement a method compute_accuracy

1.4.1.1. The indices of the maximum value in each sample are predicted labels.

1.4.2. One sample of the model output is nLabels.

1.5. Train the network.
   1.5.1. Epochs: 100
   1.5.2. Plot the loss curves of training and validation.
   1.5.3. Plot the accuracy curves of training and validation

1.6. Implement a method ex1_tester that predicts the labels using the trained model. Evaluate the results of the network. Discuss about the training curves and about the test results. What could you do to improve the performance?

2. Image Quality Classification

The goal of this exercise is to classify images as either fully-sampled or under-sampled compressed sensing reconstructions.

2.1. Prepare training data.

2.1.1. Load data by calling, load_data_ex2.

2.1.1.1. x_train: training input [nTrainSamples, nSlices, nRows, nCols]

2.1.1.2. y_train: training labels [nTrainSamples]

2.1.1.3. x_val: validation input [nValSamples, nSlices, nRows, nCols]

2.1.1.4. y_val: validation labels [nValSamples]

2.1.1.5. Labels:

2.1.1.5.1. 0: Fully-sampled ground truth

2.1.1.5.2. 1: TGV-based compressed sensing

2.1.2. Training data

2.1.2.1. Implement a method ex2_gen_dataloader that generates a dataloader for the input_data and label_data.

2.1.2.1.1. The dtype of the input_data and label tensor should be the float dtype.

2.1.2.1.2. Use data_utils.TensorDataset and data_utils.DataLoader.

2.1.2.1.3. Transfer the data to the self.device

2.1.2.1.4. Make sure the dataloader is flexible to the shuffle option.

2.1.2.1.5. Generate each dataloader for trainin and validation set. Do not shuffle for the validation dataloaders.

2.2. Build CNN models as shown in the Appendix 1.

- CNN with 5 layers + fully-connected layer
- CNN with 1 layer + global average pooling
- CNN with 5 layers + global average pooling

2.3. Train CNN models.

- get_loss: BCELoss
- get_optimizer:
    - Adam optimizer
    - Learning rate: 0.001
- Epochs: 300

2.3.1. Expand the predefined compute_accuracy method to work with 1-dimention output. Think about the sigmoid is applied to the output.

2.3.2. Train CNN with 5 layers + fully-connected layer

2.3.3. Train CNN with 1 layer + global average pooling

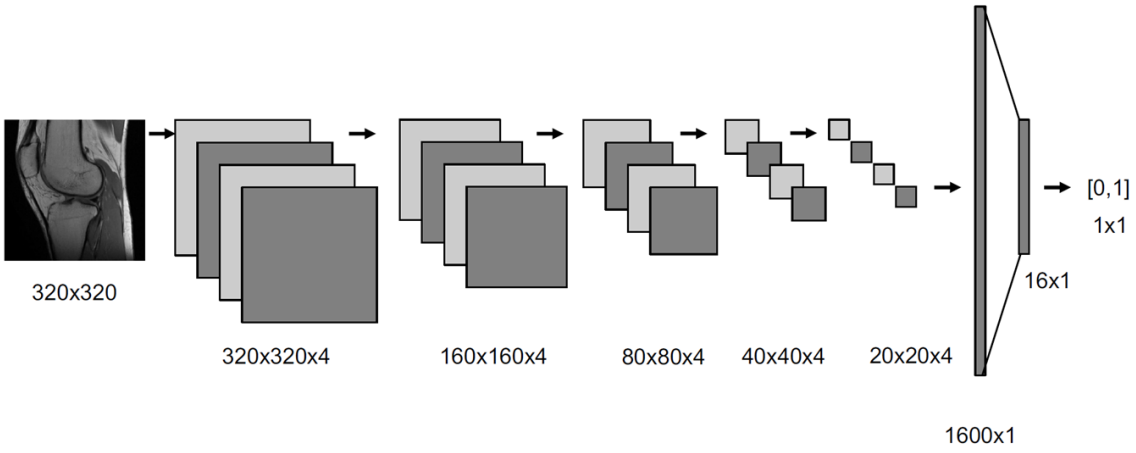
2.3.4. Train CNN with 5 layers + global average pooling

2.4. Plot loss and accuracy for each CNN models. Discuss your observations.

2.4.1. Use utils.plot() function to plot loss and accuracy.

Appendix 1

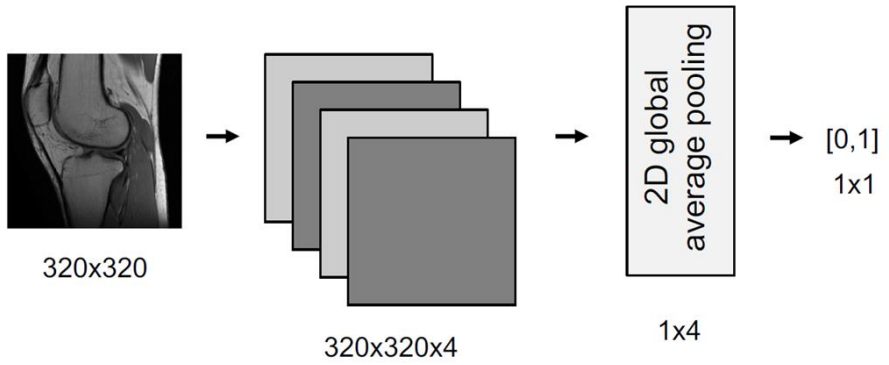
5 CNNs with Fully-connected layer



- 1 3x3x4 convolutional layer, ReLU
- 4 3x3x4 convolutional layers, ReLU, 2D MaxPooling
- Flatten
- 1600x16 fully connected layer
- 16x1 fully connected layer
- Sigmoid output

Total number of parameters:
(1x4x3x3+4) + (4x4x3x3+4)*4 + (1600x16+16) + (16x1+1) = 26265

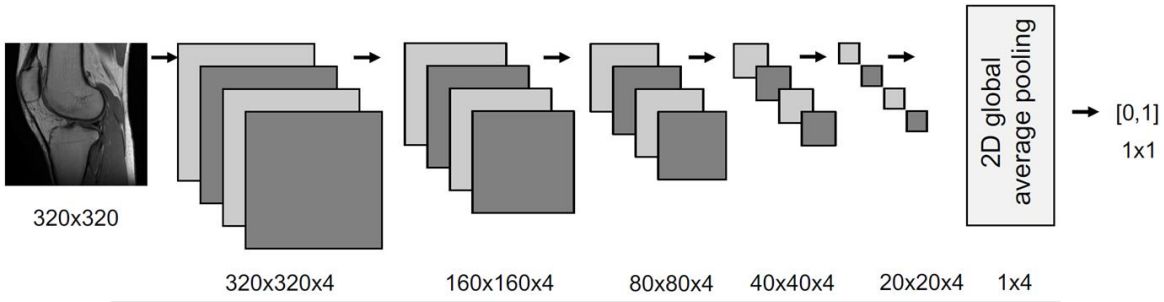
1 CNN layer + Global Average Pooling



- 1 3x3x4 convolutional layer, ReLU
- 2D Global Average Pooling
- 4x1 fully connected layer
- Sigmoid Output

Total number of parameters:
(1x4x3x3+4) + (4x1+1) = 45

5 CNN layers + Global Average Pooling



- 1 3x3x4 convolutional layer, ReLU
- 4 3x3x4 convolutional layers, ReLU, 2D MaxPooling
- 2D Global Average Pooling
- 4x1 fully connected layer
- Sigmoid output

Total number of parameters:
(1x4x3x3+4) + (4x4x3x3+4)*4 + (4x1+1) = 637