Deep Learning Assignment 8 in FS 2022

Open-Set Learning

Microsoft Forms Document:
https://forms.office.com/r/xY9sQDQdGh

Manuel Günther

Distributed: Friday, April 29, 2022
Discussed: Friday, May 6, 2022

In this assignment, we develop a network that is capable of correctly classifying known classes and at the same time rejecting unknown samples that occur during inference time. To showcase the capability, we make use of the MNIST dataset that we artificially split into known and unknown classes; this allows us to actually train a network on the data without requiring too expensive hardware.

1 Dataset

We split the MNIST dataset into 4 known classes, 4 known unknown classes (used for training) and 2 unknown unknown classes (used only for testing). While several splits might be possible, here we restrict to the following:

- Known class indexes: (4, 5, 8, 9)
- Known unknown class indexes: (0, 2, 3, 7)
- Unknown unknown class indexes: (1, 6)

Please note that, in PyTorch, class indexing starts at 0 (other than in the lecture where class indexing starts at 1).

We rely on the `torchvision.datasets.MNIST` implementation of the MNIST dataset, which we adapt to our needs. The constructor of our Dataset class takes one parameter that defines the purpose of this dataset (`"train", "validation", "test"`). The `"train"` partition uses the training samples of the known and the known unknown classes. The `"validation"` partition uses the test samples of the known and the known unknown classes. Finally, the `"test"` partition uses the test samples of the known and the unknown unknown classes.

In our implementation of the Dataset class, we need to implement two functions. First, the constructor `__init__(self, purpose)` selects the data based on our purpose. Second, the index function `__getitem__(self, n)` returns a pair $(\mathcal{X}^{[n]}, \vec{t}^{[n]})$ for the sample with the index n, where $\mathcal{X} \in \mathbb{R}^{1\times28\times28}$ with values in range $[0, 1]$ and $\vec{t} \in \mathbb{R}^O$, see below.

Since our loss function (cf. Task 5) requires our target vectors to be in vector format, we need to convert the target index $t^{[n]}$ into its vector representation $\vec{t}^{[n]}$. Particularly, we need to provide the following target vectors:

$$\begin{aligned} t^{[n]} = 4: \quad \vec{t}^{[n]} &= (1,0,0,0); & t^{[n]} = 5: \quad \vec{t}^{[n]} &= (0,1,0,0);\\ t^{[n]} = 8: \quad \vec{t}^{[n]} &= (0,0,1,0); & t^{[n]} = 9: \quad \vec{t}^{[n]} &= (0,0,0,1);\\ \text{else} \quad \vec{t}^{[n]} &= \left(\frac{1}{4},\frac{1}{4},\frac{1}{4},\frac{1}{4}\right) \end{aligned} \tag{1}$$

Task 1: Target Vectors Implement a function that generates a target vector for any of the ten different classes according to (1). The return value should be a `torch.tensor` of type float.

Test 1: Check your Target Vectors Implement a test case that checks that the target vectors for the known classes and for the unknown classes are as expected.

Task 2: Dataset Construction Write a Dataset class that derives from `torchvision.datasets.MNIST`. In the constructor, make sure that you call the `super` class constructor with the desired parameters. Afterward, the `self.data` and `self.targets` is populated with all samples and target indexes. From these, we need to sub-select the samples that fit our current purpose, and store them back to `self.data` and `self.targets`.

Task 3: Dataset Item Selection Second, we need to implement the index function of our dataset, where we need to return both the image and the target vector. The images in `self.data` are originally stored as `uint8` values in dimension $\mathbb{N}^{N\times 28\times 28}$ with values in $[0, 255]$. The targets in `self.targets` are originally stored as class indexes in dimension $\mathbb{N}^N$. Make sure that you return both in the desired format.

Test 2: Data Sets Instantiate the training dataset and the according training data loader with batch size $B = 64$. Make sure that all inputs and targets are of the desired sizes.

Task 4: Utility Function Since we need to separate known from unknown data in several tasks below, we define a utility function that takes a batch of data (of any kind) and a batch of target vectors according to (1). This function needs to return three elements: First, the samples from the batch that belong to known classes. Second, the target vectors that belong to the known classes. Finally, the samples from the batch that belong to unknown classes.

2 Loss Function and Confidence

We write our own PyTorch implementation of our loss function. Particularly, we implement a manual way to define the derivative of our loss function via `torch.autograd.Function`, which allows us to define the forward and the backward pass (the Jacobian) on our own. For this purpose, we need to implement two `static` functions into our loss. The function `forward(ctx, logits, targets)` is required to compute the loss value, and allows us to store some variables in the context for the backward pass. The `backward(ctx, result)` provides us with the result of the forward function (the loss value) as well as the context with our stored variables. Here, we need to compute the derivative of the loss with respect to both of the inputs to the forward function (which might look a bit confusing), i.e., $\frac{\partial \mathcal{J}^{\mathrm{CCE}}}{\partial \mathbf{Z}}$ and $\frac{\partial \mathcal{J}^{\mathrm{CCE}}}{\partial \mathbf{T}}$. Since the latter is not required, we can also simply return `None` for the second derivative.

Task 5: Loss Function Implementation Implement a `torch.autograd.Function` class for the adapted SoftMax function according to the equations provided in the lecture. You might want to compute the log of the network output $\ln y_o$ from the logits z_o via `torch.nn.functional.log_softmax`. Store all data required for the backward pass in the context during `forward`, and extract these from the context during `backward`.

Task 5a: Alternative Loss Function In case, the loss function is too difficult to be implemented, you can also choose to rely on PyTorch's automatic gradient computation, and simply define your loss function without backward pass. In this case, a simple function `adapted_softmax(logits, targets)` is sufficient. You can implement any variant of the categorical cross entropy loss function on top of SoftMax activations as defined in the lecture.

Task 6: Confidence Evaluation Implement the `confidence(logits, targets)` computation as provided in the lecture. Compute SoftMax confidences for the logits. Split these confidences between known and unknown classes. For samples from known classes, sum up the SoftMax confidences of the correct class. For unknown samples, sum 1 minus the maximum confidence for any of the known classes; also apply the $\frac{1}{O}$ correction for the minimum possible SoftMax confidence.

Test 3: Check Confidence Implementation Generate logit vectors and according target vectors that constitute (almost) optimal values for known and unknown samples. Make sure that the result of the `confidence` function is as expected. Note that confidence values should always be between 0 and 1, other values indicate an issue in the implementation.

1. 2D convolutional layer with Q_1 channels, kernel size 5×5, stride 1 and padding 2
2. 2D maximum pooling layer with kernel size 2×2 and stride 2
3. activation function ReLU
4. 2D convolutional layer with Q_2 channels, kernel size 5×5, stride 1 and padding 2
5. 2D maximum pooling layer with kernel size 2×2 and stride 2
6. activation function ReLU
7. flatten layer to convert the convolution output into a vector
8. fully-connected layer with the correct number of inputs and K outputs
9. fully-connected layer with K inputs and O outputs

Table 1: Network configuration of the convolutional network

3 Network and Training

We make use of the same convolutional network as utilized in Assignment 6, to which we append a final fully-connected layer with K inputs and O outputs. Additionally, we replace the tanh activation function by the better-performing ReLU function. The topology can be found in Table 1. However, instead of relying on the `torch.nn.Sequential` class, we need to define our own network class which we need to derive from `torch.nn.Module` – since our network has two outputs. We basically need to implement two methods in our network. The constructor `__init__(self, Q1, Q2, K)` needs to call the base class constructor and initialize all required layers of our network. The `forward(self, x)` function then passes the input through all of our layers, and returns both the deep features (extracted at the first fully-connected layer) and the logits (extracted from the second fully-connected layer).

Task 7: Network Definition Implement a network class including the layers as provided in Table 1. Implement both the constructor and the forward function. Instantiate the network with $Q_1 = 32, Q_2 = 32, K = 20, O = 4$.

Task 8: Training and Validation Loop Instantiate the validation set and an according data loader. Instantiate an SGD optimizer with an appropriate learning rate (the optimal learning rate might depend on your loss function implementation and can vary between 0.1 and 0.00001). Implement the training and validation loop for 10 epochs (you can also train for 100 epochs if you want). Compute the training set confidence during the epoch. At the end of each epoch, also compute the validation set confidence measure. Print both training set and validation set confidence scores to console.

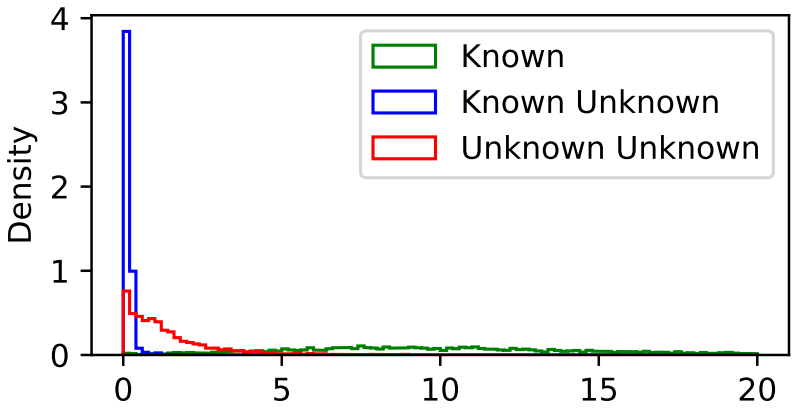

Figure 1: Magnitude plots for samples of known, known unknown and unknown unknown validation/test samples.

4 Evaluation

For evaluation, we test two different things. First, we check whether our intuition was correct and the training helped to reduce deep feature magnitudes of unknown samples while maintaining magnitudes for known samples. It is also interesting to see whether there is a difference between samples of the known unknown classes that were seen during training, and unknown unknown classes that were not. For this purpose, we extract the deep features for the validation and test sets, compute their magnitudes, and plot them in a histogram.

The second evaluation computes Correct Classification Rates (CCR) and False Positive Rates (FPR) for a given confidence threshold $\tau = 0.98$ (based on your training results, you might want to vary this threshold). For the known samples, we compute how often the correct class was classified with a confidence over threshold. For unknown samples, we assess how often one of the known classes was predicted with a confidence larger than τ.

Task 9: Feature Magnitude Plot Extract deep features for validation and test set samples, and compute their magnitudes. Split them into known, known unknown and unknown unknown samples. Plot a histogram for each of the three types of samples. Note that the minimum magnitude is 0, and the maximum magnitude can depend on your network training success. A possible solution can be found in Figure 1.

Task 10: Classification Evaluation Compute the CCR and the FPR of the test set for a confidence threshold of $\tau = 0.98$. Print both to console. A good network can achieve a CCR of $> 90\%$ for an FPR $< 10\%$.