Ex6_Regularization2

November 23, 2024

1 Exercise Sheet 6 - Regularization 2

In this exercise we will take a look at simple but effective methods used to combat overfitting when learning neural networks. We will use a very small subset of the FashionMNIST dataset to artificially induce overfitting, train and evaluate our model. Finally we will look at how to incorporate early stopping and how adding noise to our data makes our model more robust.

[13]:
```
from typing import Callable
from functools import partial

import torch
from torch import nn
from torch.optim import Optimizer, SGD

from torchvision.transforms import v2

#from solution import *
from utils import *

DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
DEVICE
```

[13]:
```
'cpu'
```

2 Training configuration

Throughout this notebook we will use the following training configuration:

[14]:
```
TRAIN_SET_SIZE = 200
VAL_SET_SIZE = 1000
EPOCHS = 500
HIDDEN_DIMS = [64, 32]
LR = 0.05
BATCH_SIZE = 32
```

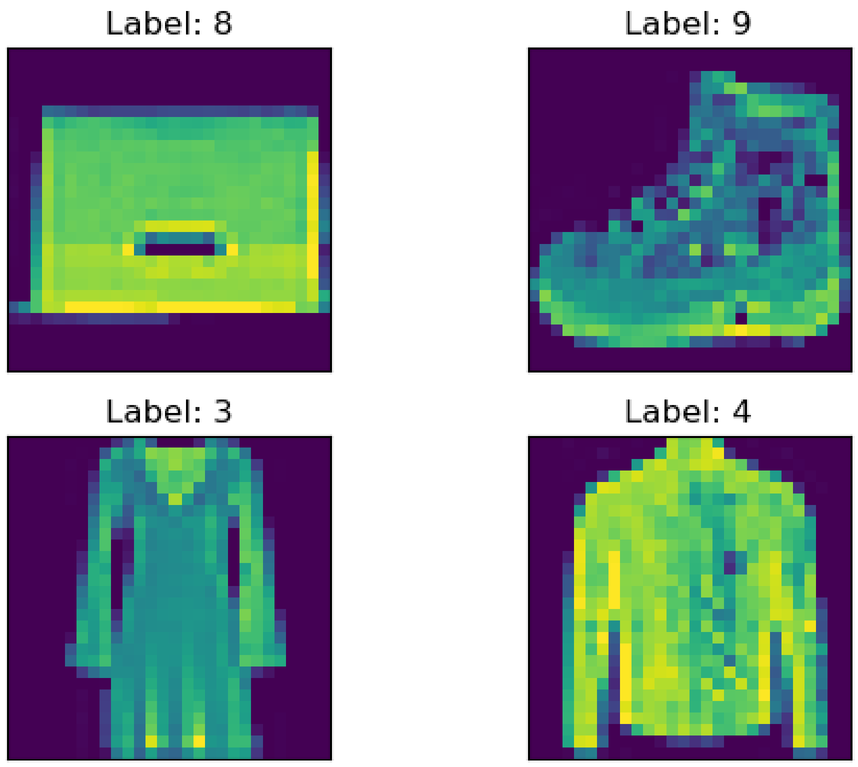
3 Dataset (FashionMNIST)

As mentioned before, we will use the FashionMNIST dataset. This dataset behaves exactly the same as the standard MNIST dataset (grayscale images with height and width of 28 pixels, 10 classes (0-9), train set with 60k samples and test set with 10k samples) with the only difference being the depicted images. While MNIST shows handwritten digits, FashionMNIST shows 10 different types of clothing. Example images are shown after execution of the following cell.

[15]:
```
train_set, val_set, test_set = get_fashion_mnist_subset(TRAIN_SET_SIZE,␣
  ↪VAL_SET_SIZE, transforms=v2.ToTensor())

train_loader = DataLoader(train_set, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(val_set, batch_size=BATCH_SIZE)
test_loader = DataLoader(test_set, batch_size=BATCH_SIZE)

visualize_first_4(train_loader)
```

```
/Users/yilongwang/anaconda3/lib/python3.10/site-
packages/torchvision/transforms/v2/_deprecated.py:42: UserWarning: The transform
`ToTensor()` is deprecated and will be removed in a future release. Instead,
please use `v2.Compose([v2.ToImage(), v2.ToDtype(torch.float32,
scale=True)])`.Output is equivalent up to float precision.
  warnings.warn(
```

```
Shape of images is torch.Size([32, 1, 28, 28])
```



4 Create model

First, we need a function to create a model that is able to classify FashionMNIST data. The model takes in inputs of the shape (batch_size x 1 x 28 x 28) and outputs a 10-dimensional vector. It should first flatten the input images, then apply a given number of linear layers to it, and finally map to a 10-dimensional vector which will be used to predict which type of clothing it is.

4.1 Task 1a) (20 P)

Complete the missing code in the following function. The function takes a list of hidden dimensions which correspond to the dimensionality of the linear layers. The input dimension of 28 x 28 and the output dimension of 10 should be hardcoded into the model. There should be as many layers as hidden dimensions, plus a final output layer. As an activation function we will use the ReLU activation.

[43]:
```
def create_model(hidden_dims: list[int]):  # TODO: explain in docstring that␣
 ↪input is written without brackets
    """Create a model that works for classifying the FahsionMNIST dataset."""
    # --------------------------------------------------
    # TODO: Replace by your code
    # --------------------------------------------------
    #model = solution_1a_create_model(hidden_dims)

    input_dim = 28*28
    output_dim = 10
    layers = []

    layers.append(nn.Flatten())
    layers.append(nn.Linear(input_dim,hidden_dims[0]))
    layers.append(nn.ReLU())

    for i in range(len(hidden_dims)-1):
        layers.append(nn.Linear(hidden_dims[i],hidden_dims[i+1]))
        layers.append(nn.ReLU())

    layers.append(nn.Linear(hidden_dims[-1],output_dim))
    model = nn.Sequential(*layers)


    # --------------------------------------------------

    model.to(DEVICE)
    return model
```

5 Train, evaluate and save models

5.1 Task 2a) (20 P)

Write a function that trains a model for one epoch / one iteration through the train dataset. Make sure to zero the gradients before each step and to apply the optimizers functions to train the model. This is a repetition from last exercise, so check there in case you are unsure.

[17]:
```python
def train_one_epoch(model, dataloader, optimizer):
    # --------------------------------------------------
    # TODO: Replace by your code
    # --------------------------------------------------
    #avg_loss = solution_2a_train_one_epoch(model, dataloader, optimizer)

    loss_fn = nn.CrossEntropyLoss()
    model.train()
    losses = []

    for X, y in dataloader:
        pred = model(X)
        loss = loss_fn(pred, y)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        losses.append(loss.item())

    avg_loss = sum(losses) / len(losses)

    # --------------------------------------------------
    return avg_loss
```

5.2 Task 2b) (20 P)

Write a function that iterates through a dataloader, and outputs the average loss and accuracy. Make sure that no gradient computation is triggered, e.g. use torch.no_grad(). Furthermore, make sure that your model in evaluation mode and to switch back afterwards. This will be important for the last task where we are adding Dropout layers to our neural network. For more information see this StackOverflow question.

[31]:
```python
def validate(model: nn.Module, dataloader: DataLoader) -> tuple[float, float]:
    # --------------------------------------------------
    # TODO: Replace by your code
    # --------------------------------------------------
    #avg_loss, avg_accuracy = solution_2b_evaluate(model, dataloader)

    loss_fn = nn.CrossEntropyLoss()
    losses = []
    correct_predict = 0
    number_samples = 0

    model.eval()
    with torch.no_grad():
        for X,y in dataloader:
            pred = model(X)
            loss = loss_fn(pred, y)
            losses.append(loss)

            _,predicted = torch.max(pred,1)
            correct_predict += sum(predicted == y)
            number_samples += len(y)



    avg_loss = sum(losses) / len(dataloader)      # number of batch
    avg_accuracy =  correct_predict / number_samples

    # --------------------------------------------------
    return avg_loss, avg_accuracy
```

Here we define our complete training function. It simply iterates for `n_epochs` epochs through the training dataset, evaluates after each epoch on the validation dataset, and finally returns an array with the train and validation losses for each epoch. An important feature of this training function is that it can take a function as an argument (that's the `callback` argument) which gets the model, the current epoch, the array of train losses up until this point, the array of validation losses until this point and the last validation accuracy. The follwing tasks will partly consist of writing functions that we will pass into the training function.

[19]:
```
def train(
        model: nn.Module,
        train_loader: DataLoader,
        val_loader: DataLoader,
        optimizer: Optimizer,
        n_epochs: int,
        callback: Callable[[nn.Module, int, list[float], list[float], float],␣
 ↪None]
    ) -> tuple[list[float], list[float]]:
    train_losses = []
    val_losses = []
    for epoch in range(n_epochs):
        train_loss = train_one_epoch(model, train_loader, optimizer)
        val_loss, val_acc = validate(model, val_loader)

        train_losses.append(train_loss)
        val_losses.append(val_loss)

        callback(model, epoch, train_losses, val_losses, val_acc)

    return train_losses, val_losses
```

5.3 Task 2c) (10 P)

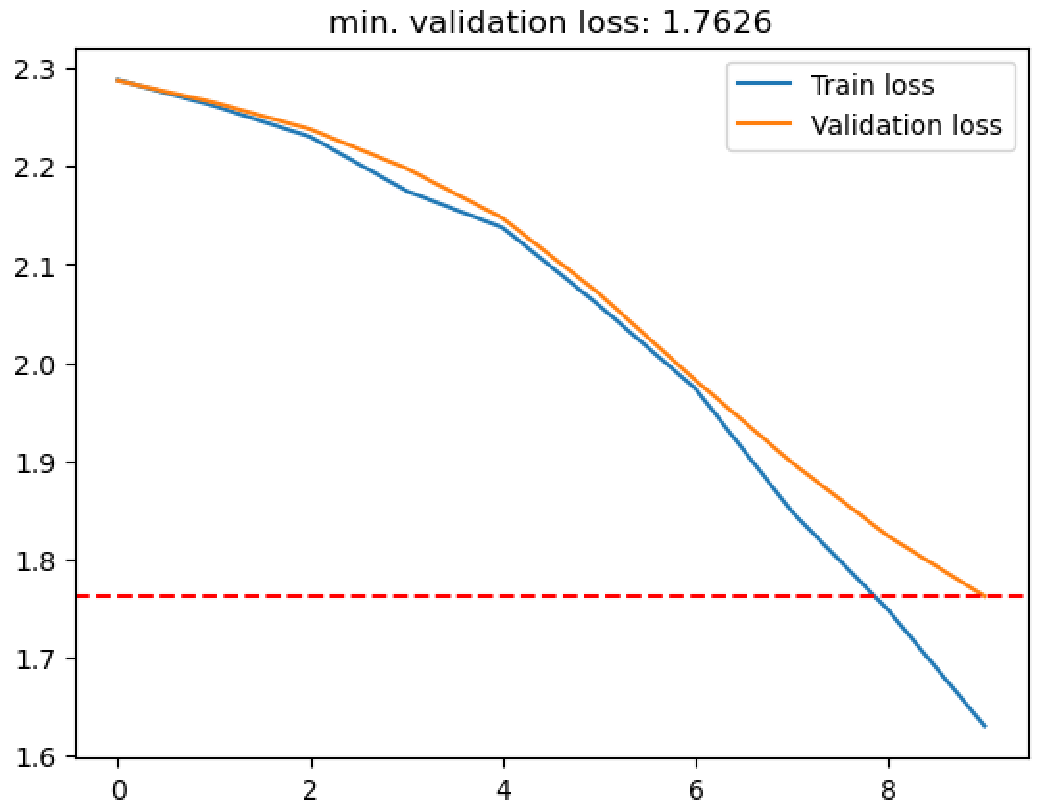
In this task you should simply write a function that we can pass into the training function above that prints the current stats every `n` epochs. This function shouldn't return anything.

[ ]:
```
def print_loss_every_n_epochs(
        model: nn.Module,
        epoch: int,
        train_losses: list[float],
        val_losses: list[float],
        val_acc,
        n: int) -> None:
    # --------------------------------------------------
    # TODO: Replace by your code
    # --------------------------------------------------
    #solution_2c_print_loss_every_n_epochs(
    #    model, epoch, train_losses, val_losses, val_acc, n)

    if epoch % n == 0:
        print(f"[EPOCH {epoch}] Train loss: {train_losses[-1]:.4f}, Validation␣
 ↪loss: {val_losses[-1]:.4f}, Validation accuracy: {val_acc:.4f}")


    # --------------------------------------------------
```

Next we will train a model. We will use the above defined hyperparameters, and a simple SGD optimizer. Furthermore we will print the training stats every epoch.

[48]:
```
model = create_model(HIDDEN_DIMS)
optimizer = SGD(model.parameters(), lr=LR)

train_losses , val_losses = train(model,
                                  train_loader,
                                  val_loader,
                                  optimizer,
                                  n_epochs=10,
                                  callback=partial(print_loss_every_n_epochs,␣
 ↪n=1))

plot_train_and_val_loss(train_losses, val_losses, title=f"min. validation loss:␣
 ↪{min(val_losses):.4f}")
```

```
[EPOCH 0] Train loss: 2.2876, Validation loss: 2.2869, Validation accuracy:
0.0860
[EPOCH 1] Train loss: 2.2613, Validation loss: 2.2647, Validation accuracy:
0.1570
[EPOCH 2] Train loss: 2.2298, Validation loss: 2.2375, Validation accuracy:
0.2680
[EPOCH 3] Train loss: 2.1747, Validation loss: 2.1976, Validation accuracy:
0.2890
[EPOCH 4] Train loss: 2.1372, Validation loss: 2.1469, Validation accuracy:
0.2880
[EPOCH 5] Train loss: 2.0582, Validation loss: 2.0703, Validation accuracy:
0.3490
[EPOCH 6] Train loss: 1.9735, Validation loss: 1.9822, Validation accuracy:
0.3060
[EPOCH 7] Train loss: 1.8486, Validation loss: 1.8985, Validation accuracy:
0.3180
[EPOCH 8] Train loss: 1.7486, Validation loss: 1.8237, Validation accuracy:
0.2990
[EPOCH 9] Train loss: 1.6308, Validation loss: 1.7626, Validation accuracy:
0.2920
```



6 Early stopping

Early stopping is the most simple thing to prevent your final model to overfit: you simply track train and test errors and use a model checkpoint before your model started to overfit.

6.1 Task 3a) (10 P)

Implementing early stopping is pretty straight forward. Simply save your model each time you've reached a new best validation loss, and otherwise don't. To reduce clutter in your filesystem you can simply override the saved model each time. Saving and loading PyTorch models is described in this guide. The function to load models is given.

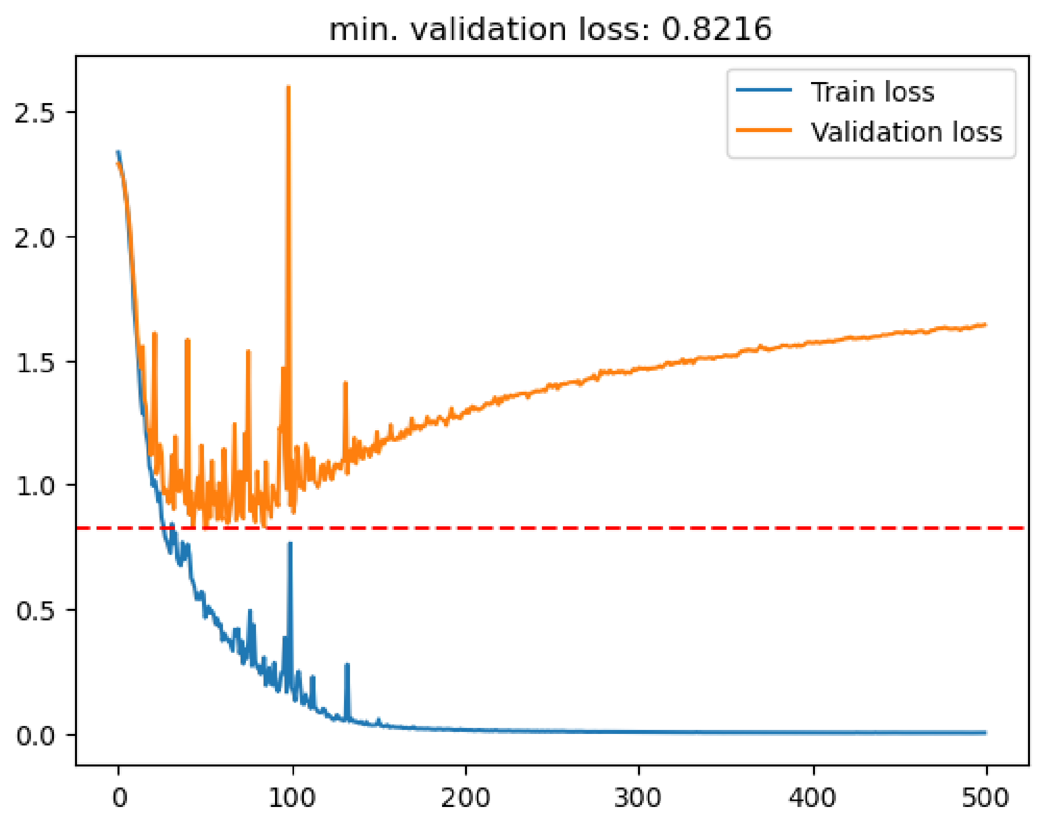
```
[49]: def save_model_if_improved(model, epoch, train_losses, val_losses, val_acc,␣
       ↪filename):
          # ---------------------------------------------------
          # TODO: Replace by your code
          # ---------------------------------------------------
          #solution_3b_save_model_if_improved(model, epoch, train_losses, val_losses,␣
       ↪val_acc, filename)
          if epoch == 0 or val_losses[-1] < min(val_losses[:-1]):
              torch.save(model.state_dict(), filename)
              print(f"[EPOCH {epoch}] Model saved with validation loss:␣
       ↪{val_losses[-1]:.4f}")

          # ---------------------------------------------------

      def load_model(model, model_file_path: str) -> nn.Module:
          model.load_state_dict(torch.load(model_file_path, weights_only=True))
          return model
```

This time we will train our model for more epochs, to show how it overfits. Your implemented function should stop saving the model once the model starts to overfit.

```
[50]: model = create_model(HIDDEN_DIMS)
      optimizer = SGD(model.parameters(), lr=LR)
      train_losses, val_losses = train(model,
                                       train_loader,
                                       val_loader,
                                       optimizer,
                                       n_epochs=EPOCHS,
                                       callback=partial(save_model_if_improved,␣
       ↪filename="early_stopping.pth"))

      plot_train_and_val_loss(train_losses, val_losses, title=f"min. validation loss:␣
       ↪{min(val_losses):.4f}")
```

```
[EPOCH 0] Model saved with validation loss: 2.2879
[EPOCH 1] Model saved with validation loss: 2.2717
```

```
[EPOCH 2] Model saved with validation loss: 2.2525
[EPOCH 3] Model saved with validation loss: 2.2251
[EPOCH 4] Model saved with validation loss: 2.1877
[EPOCH 5] Model saved with validation loss: 2.1337
[EPOCH 6] Model saved with validation loss: 2.0731
[EPOCH 7] Model saved with validation loss: 2.0093
[EPOCH 8] Model saved with validation loss: 1.8895
[EPOCH 9] Model saved with validation loss: 1.8101
[EPOCH 10] Model saved with validation loss: 1.7338
[EPOCH 11] Model saved with validation loss: 1.6002
[EPOCH 12] Model saved with validation loss: 1.5239
[EPOCH 13] Model saved with validation loss: 1.4691
[EPOCH 15] Model saved with validation loss: 1.3410
[EPOCH 16] Model saved with validation loss: 1.3153
[EPOCH 17] Model saved with validation loss: 1.2191
[EPOCH 18] Model saved with validation loss: 1.2152
[EPOCH 19] Model saved with validation loss: 1.1173
[EPOCH 22] Model saved with validation loss: 1.0433
[EPOCH 26] Model saved with validation loss: 0.9634
[EPOCH 28] Model saved with validation loss: 0.9589
[EPOCH 29] Model saved with validation loss: 0.9258
[EPOCH 32] Model saved with validation loss: 0.9004
[EPOCH 41] Model saved with validation loss: 0.8788
[EPOCH 43] Model saved with validation loss: 0.8306
[EPOCH 50] Model saved with validation loss: 0.8216
```



We now evaluate our model on the test set, to get a better estimate of the generalization error. As you can see we've named the model

[51]:
```
pretrained_model = create_model(HIDDEN_DIMS)
load_model(pretrained_model, model_file_path="early_stopping.pth")
test_loss, test_accuracy = validate(pretrained_model, test_loader)

print(f"Early stopping model achieved test loss of {test_loss:.4f} and accuarcy␣
  ↪{test_accuracy:.4f}")
# Early stopping model achieved test loss of 0.7912 and accuarcy 0.7292
```

```
Early stopping model achieved test loss of 0.8331 and accuarcy 0.6804
```

7 Dropout

Dropout is a technique where neurons are randoml set to zero, which introduces noise into the network, and helps to generalize better. Details are described in this blogpost.

7.1 Task 4a) (20 P)

Implement a function similar to the one above where we build a standard feed forward network. But now, after each activation layer, add a nn.Dropout() layer with the dropout probability specified in the parameter p.

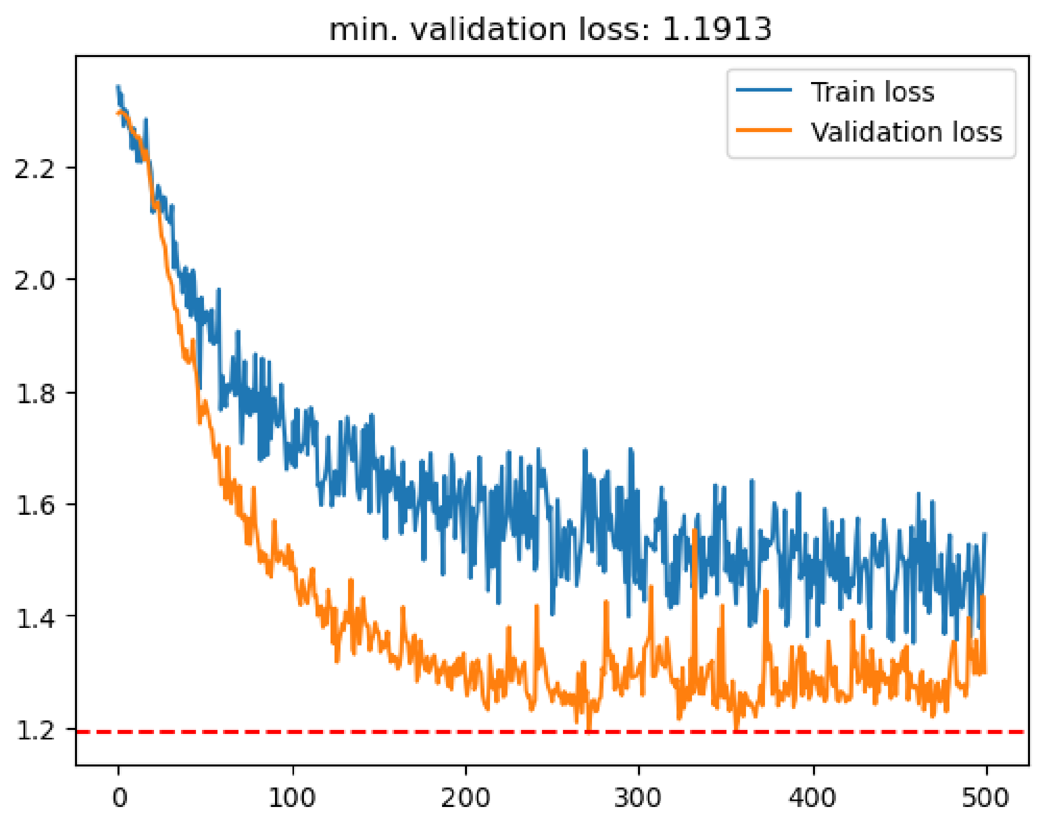
[52]:
```
def create_model_with_dropout(hidden_dims: list[int], p: float):
    # --------------------------------------------------
    # TODO: Replace by your code
    # --------------------------------------------------
    # model = solution_4a_create_model_with_dropout(hidden_dims, p)

    input_dim = 28*28
    output_dim = 10
    layers = []

    layers.append(nn.Flatten())
    layers.append(nn.Linear(input_dim,hidden_dims[0]))
    layers.append(nn.ReLU())
    layers.append(nn.Dropout(p))

    for i in range(len(hidden_dims)-1):
        layers.append(nn.Linear(hidden_dims[i],hidden_dims[i+1]))
        layers.append(nn.ReLU())
        layers.append(nn.Dropout(p))

    layers.append(nn.Linear(hidden_dims[-1],output_dim))
    model = nn.Sequential(*layers)

    # --------------------------------------------------

    model.to(DEVICE)
    return model
```

Next we will train a neural network with the exact same parameters as above, only that we now have added dropout layers.

[56]:
```
DROPOUT = 0.8

model = create_model_with_dropout(HIDDEN_DIMS, p=DROPOUT)
optimizer = SGD(model.parameters(), lr=LR)

train_losses, val_losses = train(model,
                                train_loader,
                                val_loader,
                                optimizer,
                                n_epochs=EPOCHS,
                                callback=partial(save_model_if_improved,␣
 ↪filename="dropout.pth"))

plot_train_and_val_loss(train_losses, val_losses, title=f"min. validation loss:␣
 ↪{min(val_losses):.4f}")
```

```
[EPOCH 0] Model saved with validation loss: 2.2941
[EPOCH 3] Model saved with validation loss: 2.2938
[EPOCH 4] Model saved with validation loss: 2.2930
[EPOCH 5] Model saved with validation loss: 2.2860
[EPOCH 6] Model saved with validation loss: 2.2844
[EPOCH 7] Model saved with validation loss: 2.2699
[EPOCH 8] Model saved with validation loss: 2.2613
[EPOCH 9] Model saved with validation loss: 2.2609
[EPOCH 10] Model saved with validation loss: 2.2541
[EPOCH 11] Model saved with validation loss: 2.2501
[EPOCH 13] Model saved with validation loss: 2.2402
[EPOCH 14] Model saved with validation loss: 2.2247
[EPOCH 15] Model saved with validation loss: 2.2115
[EPOCH 18] Model saved with validation loss: 2.1847
[EPOCH 19] Model saved with validation loss: 2.1679
[EPOCH 20] Model saved with validation loss: 2.1383
[EPOCH 21] Model saved with validation loss: 2.1260
[EPOCH 24] Model saved with validation loss: 2.1037
[EPOCH 25] Model saved with validation loss: 2.0761
[EPOCH 26] Model saved with validation loss: 2.0665
[EPOCH 27] Model saved with validation loss: 2.0568
[EPOCH 28] Model saved with validation loss: 2.0213
[EPOCH 29] Model saved with validation loss: 2.0042
[EPOCH 30] Model saved with validation loss: 1.9977
[EPOCH 31] Model saved with validation loss: 1.9882
[EPOCH 32] Model saved with validation loss: 1.9554
[EPOCH 33] Model saved with validation loss: 1.9451
[EPOCH 34] Model saved with validation loss: 1.9446
[EPOCH 35] Model saved with validation loss: 1.9035
[EPOCH 37] Model saved with validation loss: 1.8804
[EPOCH 38] Model saved with validation loss: 1.8578
[EPOCH 40] Model saved with validation loss: 1.8511
[EPOCH 41] Model saved with validation loss: 1.8504
[EPOCH 44] Model saved with validation loss: 1.8470
[EPOCH 45] Model saved with validation loss: 1.8302
[EPOCH 46] Model saved with validation loss: 1.8007
[EPOCH 47] Model saved with validation loss: 1.7420
[EPOCH 53] Model saved with validation loss: 1.7357
[EPOCH 54] Model saved with validation loss: 1.7319
[EPOCH 55] Model saved with validation loss: 1.6990
[EPOCH 56] Model saved with validation loss: 1.6817
[EPOCH 59] Model saved with validation loss: 1.6333
[EPOCH 62] Model saved with validation loss: 1.6078
[EPOCH 65] Model saved with validation loss: 1.6002
[EPOCH 69] Model saved with validation loss: 1.5801
[EPOCH 71] Model saved with validation loss: 1.5749
[EPOCH 72] Model saved with validation loss: 1.5713
[EPOCH 74] Model saved with validation loss: 1.5280
```

```
[EPOCH 76] Model saved with validation loss: 1.5267
[EPOCH 81] Model saved with validation loss: 1.5185
[EPOCH 82] Model saved with validation loss: 1.4953
[EPOCH 86] Model saved with validation loss: 1.4754
[EPOCH 88] Model saved with validation loss: 1.4690
[EPOCH 103] Model saved with validation loss: 1.4474
[EPOCH 104] Model saved with validation loss: 1.4375
[EPOCH 105] Model saved with validation loss: 1.4189
[EPOCH 115] Model saved with validation loss: 1.3997
[EPOCH 118] Model saved with validation loss: 1.3790
[EPOCH 124] Model saved with validation loss: 1.3503
[EPOCH 126] Model saved with validation loss: 1.3179
[EPOCH 157] Model saved with validation loss: 1.3176
[EPOCH 161] Model saved with validation loss: 1.3051
[EPOCH 172] Model saved with validation loss: 1.2829
[EPOCH 184] Model saved with validation loss: 1.2798
[EPOCH 190] Model saved with validation loss: 1.2746
[EPOCH 192] Model saved with validation loss: 1.2683
[EPOCH 201] Model saved with validation loss: 1.2581
[EPOCH 210] Model saved with validation loss: 1.2579
[EPOCH 211] Model saved with validation loss: 1.2436
[EPOCH 212] Model saved with validation loss: 1.2373
[EPOCH 213] Model saved with validation loss: 1.2325
[EPOCH 238] Model saved with validation loss: 1.2319
[EPOCH 264] Model saved with validation loss: 1.2101
[EPOCH 271] Model saved with validation loss: 1.1913
```



```
[58]: pretrained_model = create_model_with_dropout(HIDDEN_DIMS, DROPOUT)
      load_model(pretrained_model, "dropout.pth")
      test_loss, test_accuracy = validate(pretrained_model, test_loader)

      print(f"Dropout model achieved test loss of {test_loss:.4f} and accuracy of␣
       ↪{test_accuracy:.4f}")
      #Dropout model achieved test loss of 1.0192 and accuracy of 0.6272
```

```
Dropout model achieved test loss of 1.2078 and accuracy of 0.4824
```