dl-assignment-5

October 5, 2024

1 Deep Learning — Assignment 5

Fifth assignment for the 2024 Deep Learning course (NWI-IMC070) of the Radboud University.

Names: Andrew Schroeder and Fynn Gerding

Group: 17

Instructions: * Fill in your names and the name of your group. * Answer the questions and complete the code where necessary. * Keep your answers brief, one or two sentences is usually enough. * Re-run the whole notebook before you submit your work. * Save the notebook as a PDF and submit that in Brightspace together with the `.ipynb` notebook file. * The easiest way to make a PDF of your notebook is via File > Print Preview and then use your browser's print option to print to PDF.

Note:

- The models in this assignment take a while to train. It is faster on a GPU (e.g., on Google Colab), but still doable on a CPU. Plan ahead to leave enough time to analyse your results.

1.1 Objectives

In this assignment you will 1. Implement an LSTM module from scratch. 2. Use the built-in LSTM module from PyTorch. 3. Compare fully connected and recurrent neural networks in an experiment. 4. Experiment with data augmentation.

1.2 Import libraries

[3]:
```
%matplotlib inline
import csv
import glob
import re
import time
import numpy as np
import scipy.io
import scipy.signal
import matplotlib.pyplot as plt
import torch
```

```
from IPython import display

# fix the seed, so outputs are exactly reproducible
torch.manual_seed(12345);

# Use the GPU if available
def detect_device():
    if torch.cuda.is_available():
        return torch.device("cuda")
    elif torch.backends.mps.is_available():
        return torch.device("mps")
    else:
        return torch.device("cpu")
device = detect_device()
```

1.3 5.1 Dataset: Atrial fibrillation classification on ECG recordings (1 point)

In this assignment we will work with data from the PhysioNet Computing in Cardiology Challenge 2017 to classify atrial fibrillation in electrocardiograms (ECGs). Atrial fibrillation is an abnormal heart rhythm, which can be recognized as irregular patterns in ECG recordings.

(a) Download the training dataset from the challenge website and extract the files.

[4]:
```
#!mkdir -p data
#!wget -c -O data/training2017.zip https://physionet.org/files/challenge-2017/1.
 ↪0.0/training2017.zip
#!cd data/ ; unzip -qo training2017.zip
```

The dataset consists of a number of recordings and corresponding labels. We use a subset of the dataset that includes only the samples with a normal rhythm (label N or class 0) and those with atrial fibrillation (label A or class 1).

(b) Run the code to load the data.

[5]:
```
class ECGDataset(torch.utils.data.Dataset):
    # labels: 'N', 'A', 'O'
    def __init__(self, directory, max_length=18286, class_labels=('N', 'A',␣
 ↪'O')):
        super().__init__()
        self.class_labels = class_labels
        self.load_data(directory, max_length)

    def load_data(self, directory, max_length):
        label_map = {}
        with open('%s/REFERENCE.csv' % directory, 'r') as f:
            for line in csv.reader(f):
                label_map[line[0]] = line[1]

        samples = []
```

```
        lengths = []
        labels = []

        for file in sorted(glob.glob('%s/*.mat' % directory)):
            subject_id = re.match(r'.+(A[0-9]+).mat', file)[1]
            label = label_map[subject_id]
            if label not in self.class_labels:
                # skip this label
                continue
            mat_data = scipy.io.loadmat(file)
            sample = mat_data['val'][0]
            if len(sample) < 4000:
                # skip short samples
                continue
            samples.append(np.pad(sample, (0, max_length - len(sample))))
            lengths.append(len(sample))
            labels.append(self.class_labels.index(label_map[subject_id]))

        # concatenate
        samples = np.vstack(samples)
        lengths = np.stack(lengths)
        labels = np.stack(labels)

        # convert to PyTorch tensors
        self.samples = torch.tensor(samples, dtype=torch.float32)
        self.lengths = torch.tensor(lengths, dtype=torch.long)
        self.labels = torch.tensor(labels, dtype=torch.long)

    @property
    def class_proportions(self):
        return torch.mean((torch.arange(len(self.class_labels))[None, :] ==
                           self.labels[:, None]).to(torch.float), axis=0)

    def __getitem__(self, index):
        l = self.lengths[index]
        x = self.samples[index, :l]
        y = self.labels[index]
        return x, y

    def __len__(self):
        return self.samples.shape[0]


data = ECGDataset('data/training2017', class_labels=('N', 'A'))
```

The recordings have different lengths (between 30 to 60 seconds). There are more "normal" recordings than recordings that show atrial fibrillation.

(c) Print some statistics of the data.

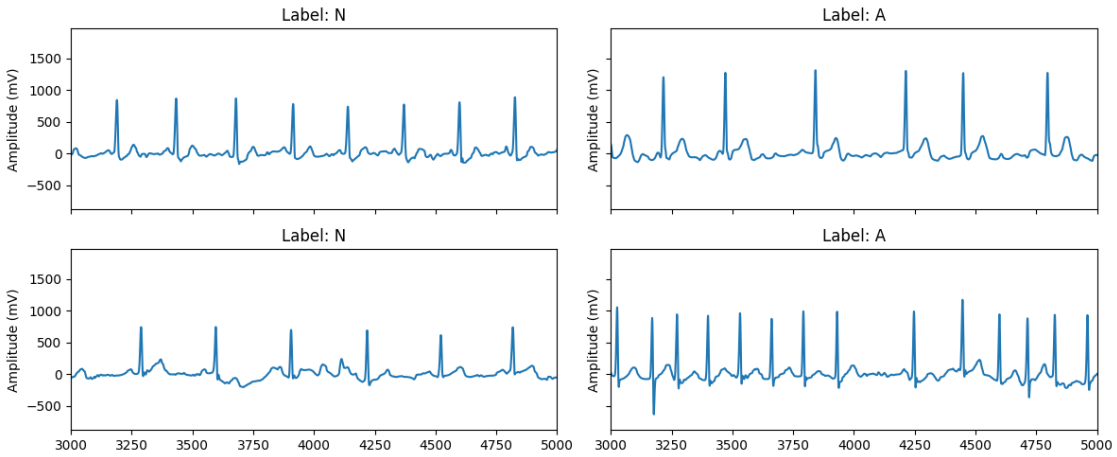
[6]:
```
print('Number of examples: %d' % len(data))
print()
print('Minimum length: %d' % torch.min(data.lengths))
print('Median length:  %d' % torch.median(data.lengths))
print('Maximum length: %d' % torch.max(data.lengths))
print()
print('Class distribution:', data.class_proportions.numpy())
```

```
Number of examples: 5622

Minimum length: 4004
Median length:  9000
Maximum length: 18286

Class distribution: [0.87709    0.12290999]
```

Each example has a 1D vector that represents the ECG measurement over time.

(d) Run the code to plot two recordings from each class.

[7]:
```
fig, axes = plt.subplots(nrows=2, ncols=2, sharex=True, sharey=True,␣
 ↪figsize=(12, 5))
for i, idx in enumerate([0, 3, 1, 4]):
    x, y = data[idx]
    ax = axes[i // 2][i % 2]
    ax.plot(x)
    ax.set_title('Label: %s' % data.class_labels[y])
    ax.set_xlim(3000, 5000)
    ax.set_ylabel('Time (frames)')
    ax.set_ylabel('Amplitude (mV)')
plt.tight_layout()
```



(e) The class distribution in this dataset is quite unbalanced. What consequences could this have? (1 point)

Normal readings make up about 87% of the total samples so with guessing we can already get close to 90% accuracy. Additionally if we don't distribute the A class samples properly between train and validation we could have issues during training. False negatives for the unhealthy class may lead to more fatal outcomes so we may want to weight mistakes in this form more heavily by modifying the loss function.

1.3.1 Feature extraction

To simplify our poblem a bit, we will convert the 1D ECG signals to spectrograms. A spectrogram is a summary of the frequencies in small windows of the recording. These features will make it easier to train a classification model.

(f) Run the code to compute the spectrograms.

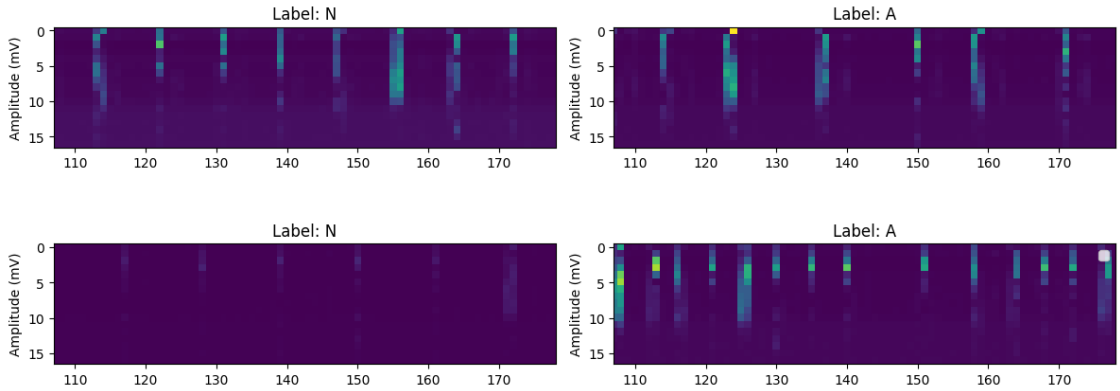
[8]:
```
class ECGSpectrumDataset(ECGDataset):
    NPERSEG = 32
    NOVERLAP = 32 // 8

    def __init__(self, *args, **kwargs):
        # initialize the original dataset to load the samples
        super().__init__(*args, **kwargs)
        # compute and store the spectrograms to replace the samples
        self.compute_spectrum()

    def compute_spectrum(self):
        """
        Replaces the samples in this dataset with spectrograms.
        """
        f, t, Sxx = scipy.signal.spectrogram(self.samples.numpy(),␣
↪scaling='spectrum',
                                             nperseg=self.NPERSEG,␣
↪noverlap=self.NOVERLAP)
        # normalize the measurements for each frequency
        Sxx = Sxx - np.mean(Sxx, axis=(0, 2), keepdims=True)
        Sxx = Sxx / np.std(Sxx, axis=(0, 2), keepdims=True)
        # replace the existing samples in the dataset with the computed␣
↪spectrograms
        self.samples = torch.tensor(Sxx.transpose(0, 2, 1))
        # recompute the length of each samples to account for the number of␣
↪windows
        self.lengths = (self.lengths - self.NPERSEG) // (self.NPERSEG - self.
↪NOVERLAP)

data_spectrum = ECGSpectrumDataset('data/training2017', class_labels=('N', 'A'))
```

(g) Plot the spectrograms for the four samples from the previous plot.

[46]:
```
plt.figure(figsize=(12, 5))
for i, idx in enumerate([0, 3, 1, 4]):
    x, y = data_spectrum[idx]
    plt.subplot(2, 2, i + 1)
    plt.imshow(x.T)
    plt.title('Label: %s' % data.class_labels[y])
    # show roughly the same segments as in the previous plot
    plt.xlim(3000 // 28, 5000 // 28)
    plt.ylabel('Amplitude (mV)')
plt.tight_layout()
plt.legend()
```

```
/tmp/ipykernel_5533/3557826403.py:11: UserWarning: No artists with labels found
to put in legend.  Note that artists whose label start with an underscore are
ignored when legend() is called with no argument.
  plt.legend()
```

[46]:
```
<matplotlib.legend.Legend at 0x730850f30ec0>
```



The spectrogram data has 17 frequency bins for each window. We will use these as our input features. We normalized the data for each frequency to zero mean, unit variance.

(h) Print the statistics of the spectrum dataset and check the shape of the first sample.

[10]:
```
print('Minimum length: %d' % torch.min(data_spectrum.lengths))
print('Median length:  %d' % torch.median(data_spectrum.lengths))
print('Maximum length: %d' % torch.max(data_spectrum.lengths))
print()
print('Mean value:         %f' % torch.mean(data_spectrum.samples))
print('Standard deviation: %f' % torch.std(data_spectrum.samples))
print()
# print the shape of the first sample
x, y = data_spectrum[0]
print('Shape of first sample:', x.shape)
```

```
Minimum length: 141
Median length:  320
Maximum length: 651

Mean value:         0.000327
Standard deviation: 1.003635

Shape of first sample: torch.Size([320, 17])
```

1.3.2 Splitting training and validation sets

We will split our dataset in separate training and validation sets (80% – 20%).

(i) Run the code to create a random split.

[11]:
```
train_samples = int(0.8 * len(data))
val_samples = len(data) - train_samples
data_train_original, data_val_original = torch.utils.data.
 ↪random_split(data_spectrum, (train_samples, val_samples))

print('data_train:', len(data_train_original))
print('data_val:  ', len(data_val_original))
```

```
data_train: 4497
data_val:   1125
```

1.3.3 Creating a balanced dataset by resampling

As you have seen, the dataset contains far more normal recordings than recordings with atrial fibrillation. We will create a balanced dataset by including multiple copies of the atrial fibrillation samples.

In this assignment we will also use a balanced validation set. This is something you may or may not want to do in practice, because it means that your validation set is no longer representative of the test data. The advantage is that the accuracy on a balanced validation set is easier to compare with the accuracy on the training set.

(j) Run the code to create balanced training and validation sets.

[12]:
```
def balance_dataset(dataset):
    # collect labels from the source dataset
    labels = torch.zeros((len(dataset),), dtype=torch.long)
    for i, (x, y) in enumerate(dataset):
        labels[i] = y
    indices = torch.arange(len(dataset), dtype=torch.long)

    unique_labels = np.unique(labels.numpy())

    # count the number of samples per class
    n = [torch.sum((labels == label).to(torch.long)).item()
```

```
         for label in unique_labels]

    # perhaps the dataset is already balanced?
    if len(np.unique(n)) == 1:
        return dataset

    print('Samples per class:', n)
    for i, label in enumerate(unique_labels):
        # we will add more samples unless every class has the same number of␣
 ↪samples
        while n[i] < max(n):
            extra_samples = max(n) - n[i]
            print('- Repeating %d samples for class %d' % (extra_samples,␣
 ↪label))

            # take a random subset of samples from this class
            idxs = torch.where(labels == label)[0]
            idxs = idxs[torch.randperm(idxs.shape[0])]
            idxs = idxs[:extra_samples]

            # add these indices to the list
            indices = torch.cat((indices, idxs))
            n[i] += len(idxs)

    # return the subset as a new torch dataset
    return torch.utils.data.Subset(dataset, indices)

print('Balancing the training set')
data_train = balance_dataset(data_train_original)
print('Balancing the validation set')
data_val   = balance_dataset(data_val_original)
```

```
Balancing the training set
Samples per class: [3952, 545]
- Repeating 3407 samples for class 1
- Repeating 2862 samples for class 1
- Repeating 2317 samples for class 1
- Repeating 1772 samples for class 1
- Repeating 1227 samples for class 1
- Repeating 682 samples for class 1
- Repeating 137 samples for class 1
Balancing the validation set
Samples per class: [979, 146]
- Repeating 833 samples for class 1
- Repeating 687 samples for class 1
- Repeating 541 samples for class 1
- Repeating 395 samples for class 1
```

```
- Repeating 249 samples for class 1
- Repeating 103 samples for class 1
```

[13]:
```
print(len(data_train))
print(len(data_val))
```

```
7904
1958
```

1.3.4 Splitting recordings into chunks

The recordings in our dataset all have different lengths and are generally quite long. To simplify training, we will split them into smaller chunks of 40 time steps each. This means that each recording will have multiple chunks in the dataset.

(k) Run the code to create the pre-chunked dataset.

[14]:
```
class ChunkedDataset(torch.utils.data.TensorDataset):
    def __init__(self, source_dataset, chunk_size=40):
        super().__init__()
        self.make_chunks(source_dataset, chunk_size)

    def make_chunks(self, source_dataset, chunk_size):
        all_x, all_y = [], []
        for x, y in source_dataset:
            for chunk in range(x.shape[0] // chunk_size):
                offset = chunk * chunk_size
                all_x.append(x[offset:offset + chunk_size])
                all_y.append(y)
        self.tensors = (torch.stack(all_x), torch.tensor(all_y))

chunked_data_train = ChunkedDataset(data_train_original)
chunked_data_val = ChunkedDataset(data_val_original)

# rebalance to compensate for any differences in length
chunked_data_train = balance_dataset(chunked_data_train)
chunked_data_val   = balance_dataset(chunked_data_val)

print('chunked_data_train:', len(chunked_data_train))
print('chunked_data_val:  ', len(chunked_data_val))
```

```
Samples per class: [33999, 4806]
- Repeating 29193 samples for class 1
- Repeating 24387 samples for class 1
- Repeating 19581 samples for class 1
- Repeating 14775 samples for class 1
- Repeating 9969 samples for class 1
- Repeating 5163 samples for class 1
- Repeating 357 samples for class 1
Samples per class: [8427, 1314]
```

```
- Repeating 7113 samples for class 1
- Repeating 5799 samples for class 1
- Repeating 4485 samples for class 1
- Repeating 3171 samples for class 1
- Repeating 1857 samples for class 1
- Repeating 543 samples for class 1
chunked_data_train: 67998
chunked_data_val:   16854
```

1.3.5 Preparing data loaders

As in the previous assignments, we will use the PyTorch DataLoader class to divide our datasets in minibatches.

(l) Run the code to create the data loaders. Look at the shape of the first minibatch.

[15]:
```
batch_size = 192
chunked_loaders = {
    'train': torch.utils.data.DataLoader(chunked_data_train, shuffle=True,␣
 ↪batch_size=batch_size),
    'val':   torch.utils.data.DataLoader(chunked_data_val,␣
 ↪batch_size=batch_size),
}

# print the x and y shapes for one minibatch
for x, y in chunked_loaders['train']:
    print(x.shape, y.shape)
    break
```

```
torch.Size([192, 40, 17]) torch.Size([192])
```

1.4 5.2 Implementing an LSTM (5 points)

Time series data such as the ECG recordings are a good target for recurrent neural network.

The class below implements an RNN layer in PyTorch, using the equations discussed in the lecture. See also section 9.4.2 of the Dive into Deep Learning book.

(a) Read through the code to see how the RNN works.

[16]:
```
class RNN(torch.nn.Module):
    """
    A simple recurrent neural network layer.

    Parameters:
       num_inputs:  scalar, the number of inputs to this module
       num_hiddens: scalar, the number of hidden units

    Input and output: see the forward function.
    """
```

```
    def __init__(self, num_inputs, num_hiddens):
        super().__init__()
        self.num_inputs = num_inputs
        self.num_hiddens = num_hiddens
        self.initialize_parameters()

    def initialize_parameters(self):
        """
        Initializes the parameters of the RNN module.

        This initializes the bias vector b_h and weight matrices W_xh and W_hh.
        """
        # Helper function that constructs two weight matrices and a bias vector
        def triple():
            return (torch.nn.Parameter(torch.normal(0, 0.01, size=(self.
↪num_inputs, self.num_hiddens))),
                    torch.nn.Parameter(torch.normal(0, 0.01, size=(self.
↪num_hiddens, self.num_hiddens))),
                    torch.nn.Parameter(torch.zeros(size=(self.num_hiddens,))))

        # parameters for the rnn
        self.W_xh, self.W_hh, self.b_h = triple()

    def forward(self, inputs):
        """
        Computes the forward pass of the RNN module.

        Input:
           inputs:  a tensor of shape (samples, steps, input features)
                    giving the input for each sample at each step

        Output:
           outputs: a tensor of shape (samples, steps, hidden features)
                    providing the hidden values at the end of each step
           state:   a tuple (hiddens,)
                    the state of the RNN at the end of the last step,
                    with hiddens a tensor of shape (samples, hidden_features)
        """
        batch_size = inputs.shape[0]

        # initialize state
        state = (torch.zeros(size=(batch_size, self.num_hiddens),
                             dtype=inputs.dtype, device=inputs.device),)

        # run steps
        outputs = []
        for step in range(inputs.shape[1]):
```

```
            state = self.one_step(inputs[:, step], state)
            outputs.append(state[0])

        # concatenate outputs
        outputs = torch.stack(outputs, axis=1)
        return outputs, state

    def one_step(self, x, state):
        """
        Run a single step of the RNN module.

        Input:
           x:     a tensor of shape (samples, input features)
                  giving the input for each sample at the current step
           state: a tuple (hiddens,)
                  the state of the RNN at the end of the previous step,
                  with hiddens a tensor of shape (samples, hidden_features)
        """
        # extract current state
        (h,) = state

        # new hidden state
        h = torch.tanh(x @ self.W_xh + h @ self.W_hh + self.b_h)

        # return the state
        return (h,)

    def __repr__(self):
        return ('RNN(num_inputs=%d, num_hiddens=%d)' %
                (self.num_inputs, self.num_hiddens))


# quick sanity check
rnn = RNN(3, 5)
print(rnn)
print('Parameters:')
for name, param in rnn.named_parameters():
    print(' - %s:' % name, tuple(param.shape))

assert rnn(torch.randn(2,10,3))[0].shape == torch.Size([2, 10, 5]), "The shape␣
 ↪of the output is incorrect"
assert len(rnn(torch.randn(2,10,3))[1]) == 1, "The hidden state should have 1␣
 ↪element"
assert rnn(torch.randn(2,10,3))[1][0].shape == torch.Size([2, 5]), "The shape␣
 ↪of the state is incorrect"
```

```
RNN(num_inputs=3, num_hiddens=5)
```

```
Parameters:
 - W_xh: (3, 5)
 - W_hh: (5, 5)
 - b_h: (5,)
```

The design of the LSTM module is more complex than that of the RNN, but it follows a similar pattern of looping over all steps in the input. You can use the RNN implementation as a basis for an LSTM module.

(b) Implement the LSTM module below. (5 points)

The equations and code on the slides, or in d2l 10.1 can provide some inspiration. Be aware that the d2l book uses `(steps, samples, ...)` instead of `(samples, steps, ...)` as the shapes for the input and output variables, so you probably cannot copy code directly. Use the RNN implementation above and adapt this to the LSTM equations from the book.

[35]:
```python
import torch.nn as nn

class LSTM(torch.nn.Module):
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.num_inputs = num_inputs
        self.num_hiddens = num_hiddens

        init_weight = lambda *shape: nn.Parameter(torch.randn(*shape) * sigma)
        triple = lambda: (init_weight(num_inputs, num_hiddens),
                          init_weight(num_hiddens, num_hiddens),
                          nn.Parameter(torch.zeros(num_hiddens)))
        self.W_xi, self.W_hi, self.b_i = triple()  # Input gate
        self.W_xf, self.W_hf, self.b_f = triple()  # Forget gate
        self.W_xo, self.W_ho, self.b_o = triple()  # Output gate
        self.W_xc, self.W_hc, self.b_c = triple()  # Input node

    def forward(self, inputs, H_C = None):
        batch_size, seq_len, _ = inputs.shape
        if H_C is None:
            # Initial state with shape: (batch_size, num_hiddens)
            H = torch.zeros((inputs.shape[0], self.num_hiddens), device=inputs.
 ↪device)
            C = torch.zeros((inputs.shape[0], self.num_hiddens), device=inputs.
 ↪device)
        else:
            H, C = H_C
        outputs = []
        for t in range(seq_len):
            x = inputs[:, t, :]
            I = torch.sigmoid(torch.matmul(x, self.W_xi) + torch.matmul(H, self.
 ↪W_hi) + self.b_i)
```

```
            F = torch.sigmoid(torch.matmul(x, self.W_xf) + torch.matmul(H, self.
 ↪W_hf) + self.b_f)
            O = torch.sigmoid(torch.matmul(x, self.W_xo) + torch.matmul(H, self.
 ↪W_ho) + self.b_o)
            C_tilde = torch.tanh(torch.matmul(x, self.W_xc) + torch.matmul(H,␣
 ↪self.W_hc) + self.b_c)
            C = F * C + I * C_tilde
            H = O * torch.tanh(C)
            outputs.append(H)
            #outputs.append(H.unsqueeze(0))

        #outputs = torch.cat(outputs, dim=0)
        #outputs = outputs.transpose(0, 1)
        return outputs, (H, C)

    def __repr__(self):
        return ('LSTM(num_inputs=%d, num_hiddens=%d)' %
                (self.num_inputs, self.num_hiddens))


# quick sanity check
lstm = LSTM(3, 5)
print(lstm)
for name, param in lstm.named_parameters():
    print(' - %s:' % name, tuple(param.shape))

assert len(list(lstm.parameters())) == 12, "There should be 12 parameters in an␣
 ↪LSTM layer"
assert len(lstm(torch.randn(2,10,3))) == 2, "LSTM should return a tuple␣
 ↪(output, state)"
assert len(lstm(torch.randn(2,10,3))[1]) == 2, "The hidden state should have 2␣
 ↪elements, h and c"
assert lstm(torch.randn(2,10,3))[1][0].shape == torch.Size([2, 5]), "The shape␣
 ↪of the hidden state is incorrect"
assert lstm(torch.randn(2,10,3))[1][1].shape == torch.Size([2, 5]), "The shape␣
 ↪of the hidden state is incorrect"
```

```
LSTM(num_inputs=3, num_hiddens=5)
 - W_xi: (3, 5)
 - W_hi: (5, 5)
 - b_i: (5,)
 - W_xf: (3, 5)
 - W_hf: (5, 5)
 - b_f: (5,)
 - W_xo: (3, 5)
 - W_ho: (5, 5)
 - b_o: (5,)
```

- W_xc: (3, 5)
- W_hc: (5, 5)
- b_c: (5,)

1.5 5.3 Defining the training loop

As last week, we need to define some functions to run the train the models.

The code is essentially the same as in assignment 4.

(a) Run the code to define the functions.

[18]:
```
def accuracy(pred_y, true_y):
    # Computes the mean accuracy.
    if pred_y.shape[1] == 1:
        # binary classification
        correct = (pred_y[:, 0] > 0).to(true_y.dtype) == true_y
    else:
        # multi-class classification
        correct = pred_y.argmax(dim=1) == true_y
    return int(correct.sum()) / len(true_y)

class Metrics:
    """Accumulate mean values of one or more metrics."""
    def __init__(self, n):
        self.count = 0
        self.sum = (0,) * n
    def add(self, count, *values):
        self.count += count
        self.sum = tuple(s + count * v for s,v in zip(self.sum,values))
    def mean(self):
        return tuple(s / self.count for s in self.sum)

def evaluate(net, test_loader, loss_function=torch.nn.CrossEntropyLoss(),␣
 ↪device=device):
    """
    Evaluate a model on the given dataset.
    Return loss, accuracy
    """
    with torch.no_grad():
        net.eval()
        metrics = Metrics(2)
        for x, y in test_loader:
            x = x.to(device)
            y = y.to(device)
            pred_y = net(x)
            loss = loss_function(pred_y, y)
            acc = accuracy(pred_y, y)
            metrics.add(len(y), loss.item(), acc)
```

```
        return metrics.mean()
```

[19]:
```
class Plotter:
    """For plotting data in animation."""
    # Based on d2l.Animator
    def __init__(self, xlabel=None, ylabel=None, legend=None, xlim=None,
                 ylim=None, xscale='linear', yscale='linear',
                 fmts=('-', 'm--', 'g-.', 'r:'), nrows=1, ncols=1,
                 figsize=(3.5, 2.5)):
        """Defined in :numref:`sec_utils`"""
        # Incrementally plot multiple lines
        if legend is None:
            legend = []
        self.fig, self.axes = plt.subplots(nrows, ncols, figsize=figsize)
        if nrows * ncols == 1:
            self.axes = [self.axes, ]
        # Use a function to capture arguments
        def config_axes():
            axis = self.axes[0]
            axis.set_xlabel(xlabel), axis.set_ylabel(ylabel)
            axis.set_xscale(xscale), axis.set_yscale(yscale)
            axis.set_xlim(xlim),     axis.set_ylim(ylim)
            if legend:
                axis.legend(legend)
            axis.grid()
        self.config_axes = config_axes
        self.X, self.Y, self.fmts = None, None, fmts

    def add(self, x, y):
        # Add multiple data points into the figure
        if not hasattr(y, "__len__"):
            y = [y]
        n = len(y)
        if not hasattr(x, "__len__"):
            x = [x] * n
        if not self.X:
            self.X = [[] for _ in range(n)]
        if not self.Y:
            self.Y = [[] for _ in range(n)]
        for i, (a, b) in enumerate(zip(x, y)):
            if a is not None and b is not None:
                self.X[i].append(a)
                self.Y[i].append(b)
        self.axes[0].cla()
        for x, y, fmt in zip(self.X, self.Y, self.fmts):
            self.axes[0].plot(x, y, fmt)
        self.config_axes()
```

```
        display.display(self.fig)
        display.clear_output(wait=True)
```

[20]:
```
def train(net, data_loaders, num_epochs, lr=0.001, optimizer=torch.optim.Adam,␣
↪device=device):
    """
    Train a network on the given data set.
    After every epoch compute validation loss and accuracy.
    """
    net.to(device)
    train_loader = data_loaders['train']
    num_batches = len(train_loader)
    optimizer = optimizer(net.parameters(), lr=lr)
    loss_function = torch.nn.CrossEntropyLoss()
    plotter = Plotter(xlabel='epoch', xlim=[1, num_epochs],
                      legend=['train loss', 'train acc', 'val loss', 'val acc'])
    start_time = time.time()
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples
        net.train()
        metrics = Metrics(2)
        for i, (x, y) in enumerate(train_loader):
            optimizer.zero_grad()
            x = x.to(device)
            y = y.to(device)
            pred_y = net(x)
            loss = loss_function(pred_y, y)
            loss.backward()
            optimizer.step()
            with torch.no_grad():
                acc = accuracy(pred_y, y)
                metrics.add(len(y), loss.item(), acc)
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                train_loss, train_acc = metrics.mean()
                plotter.add(epoch + (i + 1) / num_batches, (train_loss,␣
↪train_acc, None, None))
        val_loss, val_acc = evaluate(net, data_loaders['val'],␣
↪loss_function=loss_function, device=device)
        plotter.add(epoch + 1, (None, None, val_loss, val_acc))
    train_loss, train_acc = metrics.mean()
    train_time = time.time() - start_time
    print(f'train loss {train_loss:.3f}, train acc {train_acc:.3f}, '
          f'val loss {val_loss:.3f}, val acc {val_acc:.3f}')
    print(f'{metrics.count * num_epochs / train_time:.1f} examples/sec '
          f'on {str(device)}')
```

1.6 5.4 Constructing some networks (5 points)

In the next experiments you will train different network architectures to see how they perform on the ECG dataset.

The input to all networks has the shape (samples, time steps, features) = (mb_size, 40, 17). The output should be a two features, shape (mb_size, 2), that will be used in a cross-entropy loss function. (The networks should not include the final softmax activation function.)

Some simple baselines: * `FullyConnectedNet`: A simple fully connected network that takes all features. * `MeanSpectrumNet`: A fully connected network that works on the mean spectrum over all time steps.

A convolutional network: * `ConvNet`: This network does a convolution over the time steps, using the 17 input features as channels.

Some recurrent models: * `RNNNet`: A recurrent network with a simple RNN module. * `LSTMNet`: A recurrent network with a more advanced LSTM module. * `TorchLSTMNet`: The same model, but using the PyTorch implementation of the LSTM.

1.6.1 FullyConnectedNet

(a) Check the implementation of the following baseline architecture:

- Linear layer: network inputs to 512 units followed by a ReLU.
- Linear layer: 512 to 256 units followed by a ReLU.
- Linear layer: 256 to the network output.

[21]:
```
class FullyConnectedNet(torch.nn.Module):
    def __init__(self, inputs, outputs=2):
        super().__init__()

        # by defining these layers here, they are included in the
        # parameters() list of this module, so they can be trained
        self.net = torch.nn.Sequential(
            torch.nn.Flatten(),
            torch.nn.Linear(inputs, 512),
            torch.nn.ReLU(),
            torch.nn.Linear(512, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, outputs)
        )

    def forward(self, x):
        # x shape: (samples, steps, inputs)
        return self.net(x)

net = FullyConnectedNet(40 * 17)
print(net)
```

```
FullyConnectedNet(
```

```
  (net): Sequential(
    (0): Flatten(start_dim=1, end_dim=-1)
    (1): Linear(in_features=680, out_features=512, bias=True)
    (2): ReLU()
    (3): Linear(in_features=512, out_features=256, bias=True)
    (4): ReLU()
    (5): Linear(in_features=256, out_features=2, bias=True)
  )
)
```

1.6.2 MeanSpectrumNet

(b) Check the implementation of the following baseline architecture:

- Compute the mean spectrum (mean over the steps dimension).
- Linear layer: network inputs to 128 units followed by a ReLU.
- Linear layer: 128 to 64 units followed by a ReLU.
- Linear layer: 64 to the network output.

[22]:
```python
class MeanSpectrumNet(torch.nn.Module):
    def __init__(self, inputs=17, outputs=2):
        super().__init__()

        self.net = torch.nn.Sequential(
            torch.nn.Linear(inputs, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, outputs),
        )

    def forward(self, x):
        # x shape: (samples, steps, inputs)
        # compute the mean over all steps
        x = torch.mean(x, axis=1)
        return self.net(x)

net = MeanSpectrumNet()
print(net)
```

```
MeanSpectrumNet(
  (net): Sequential(
    (0): Linear(in_features=17, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=64, bias=True)
    (3): ReLU()
    (4): Linear(in_features=64, out_features=2, bias=True)
  )
)
```

1.6.3 ConvNet

(c) Complete the implementation of the following architecture: (1 point)

Convolution over the steps, using frequencies as channels: * 1D-convolution: network inputs to 32 channels, kernel size 3, ReLU. * Average pooling: 2. * 1D-convolution: 32 to 64 channels, kernel size 3, ReLU. * Average pooling: 2. * 1D-convolution: 64 to 128 channels, kernel size 3, ReLU. * `AdaptiveAvgPool1d(1)`: Compute the mean for each channel over all steps. * Flatten. * Linear layer: 128 to the network output.

[23]:
```
import torch.nn as nn

class ConvNet(torch.nn.Module):
    def __init__(self, inputs=1, outputs=2):
        super().__init__()

        self.net = torch.nn.Sequential(
            # 1D-convolution: network inputs to 32 channels, kernel size 3, ReLU
            nn.Conv1d(in_channels=inputs, out_channels=32, kernel_size=3,␣
 ↪padding=1),
            nn.ReLU(),

            # Average pooling: kernel size 2
            nn.AvgPool1d(kernel_size=2),

            # 1D-convolution: 32 to 64 channels, kernel size 3, ReLU
            nn.Conv1d(in_channels=32, out_channels=64, kernel_size=3,␣
 ↪padding=1),
            nn.ReLU(),

            # Average pooling: kernel size 2
            nn.AvgPool1d(kernel_size=2),

            # 1D-convolution: 64 to 128 channels, kernel size 3, ReLU
            nn.Conv1d(in_channels=64, out_channels=128, kernel_size=3,␣
 ↪padding=1),
            nn.ReLU(),

            # AdaptiveAvgPool1d(1): Compute the mean for each channel over all␣
 ↪steps
            nn.AdaptiveAvgPool1d(1),

            # Flatten
            nn.Flatten(),

            # Linear layer: 128 to the network output
            nn.Linear(128, outputs)
        )
```

```
    def forward(self, x):
        # x shape: (samples, steps, inputs)
        # swap the steps and inputs dimensions, so we can convolve over
        # the steps use the frequencies as channels
        x = x.transpose(2, 1)
        return self.net(x)

net = ConvNet()
```

1.6.4 RNNNet

(d) Check the implementation of the following architecture:

- RNN: network input to 128 hidden units.
- Use the final hidden state from the RNN.
- Linear layer: 128 to 128 units followed by a ReLU.
- Linear layer: 128 to the network output.

[24]:
```
class RNNNet(torch.nn.Module):
    def __init__(self, inputs=17, outputs=2):
        super().__init__()

        self.rnn = RNN(inputs, 128)
        self.linear = torch.nn.Sequential(
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, outputs)
        )

    def forward(self, x):
        # x shape: (samples, steps, inputs)
        out, (h,) = self.rnn(x)

        # use the final RNN hidden state as input
        # for the fully connected part
        return self.linear(h)

net = RNNNet()
print(net)
```

```
RNNNet(
  (rnn): RNN(num_inputs=17, num_hiddens=128)
  (linear): Sequential(
    (0): Linear(in_features=128, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=2, bias=True)
  )
)
```

1.6.5 LSTMNet

(e) Implement the following architecture: (see RNNNet for an example) (2 points)

- LSTM: network input to 128 hidden units.
- Use the final hidden state from the LSTM.
- Linear layer: 128 to 128 units followed by a ReLU.
- Linear layer: 128 to the network output.

[25]:
```
class LSTMNet(torch.nn.Module):
    def __init__(self, inputs=17, outputs=2):
        super().__init__()

        self.lstm = LSTM(inputs, 128)
        self.linear = torch.nn.Sequential(
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, outputs)
        )

    def forward(self, x):
        # x shape: (samples, steps, inputs)
        output = self.lstm(x)
        out, (h,c) = output
        # use the final LSTM hidden state as input
        # for the fully connected part
        return self.linear(h)

net = LSTMNet()
print(net)
```

```
LSTMNet(
  (lstm): LSTM(num_inputs=17, num_hiddens=128)
  (linear): Sequential(
    (0): Linear(in_features=128, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=2, bias=True)
  )
)
```

1.6.6 TorchLSTMNet

Implementing your own modules can be fun and good learning experience, but it is not always the most efficient solution. The built-in LSTM implementation from PyTorch is much faster than our own version.

(f) Implement a network similar to LSTMNet using the PyTorch torch.nn.LSTM module. (2 points)

[26]:
```
class TorchLSTMNet(torch.nn.Module):
    def __init__(self, inputs=17, outputs=2):
        super().__init__()

        self.rnn = torch.nn.LSTM(inputs, 128, batch_first=True)
        self.linear = torch.nn.Sequential(
            torch.nn.Linear(128, 128),
            torch.nn.ReLU(),
            torch.nn.Linear(128, outputs)
        )

    def forward(self, x):
        # x shape: (samples, steps, inputs)
        out, (h,c) = self.rnn(x)

        # use the final LSTM hidden state as input
        # for the fully connected part
        return self.linear(h.squeeze(0))

net = TorchLSTMNet()
print(net)
```

```
TorchLSTMNet(
  (rnn): LSTM(17, 128, batch_first=True)
  (linear): Sequential(
    (0): Linear(in_features=128, out_features=128, bias=True)
    (1): ReLU()
    (2): Linear(in_features=128, out_features=2, bias=True)
  )
)
```

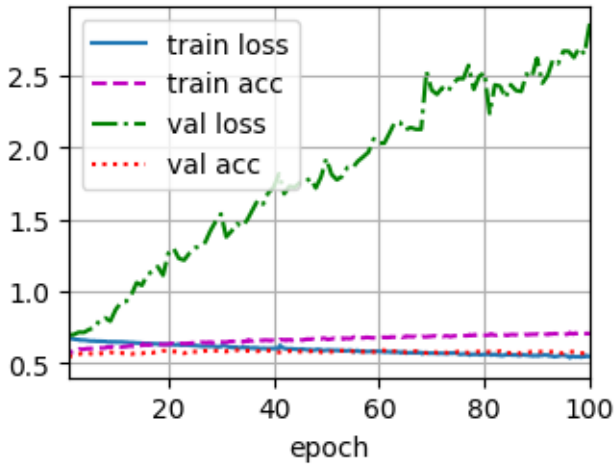
1.7 5.5 Experiments

(a) Train the models on the chunked dataset.

You may change the parameters as you see fit, but document the changes you make.

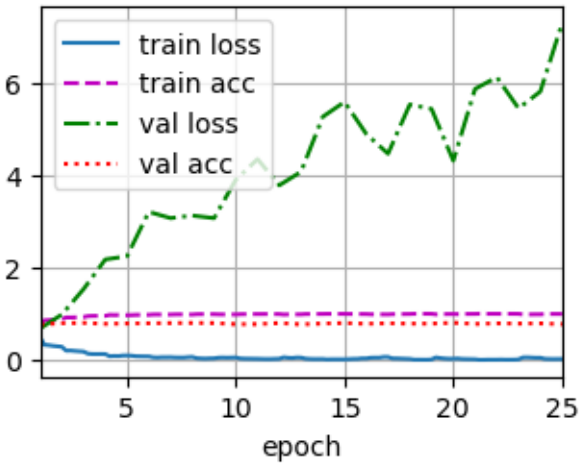
[36]:
```
train(MeanSpectrumNet(), chunked_loaders, num_epochs=100, lr=0.001)
```

```
train loss 0.548, train acc 0.709, val loss 2.864, val acc 0.581
23695.0 examples/sec on cpu
```



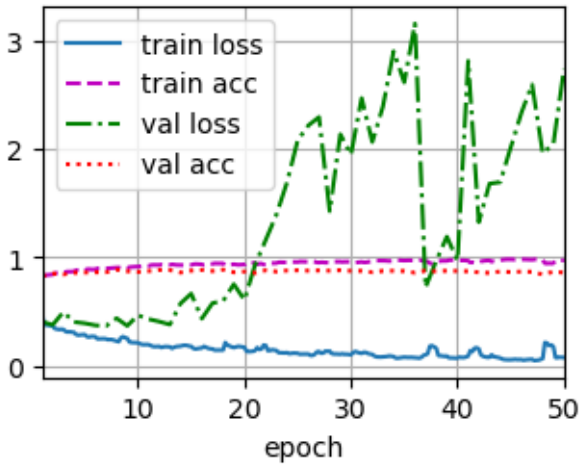
[39]:
```
train(FullyConnectedNet(40 * 17), chunked_loaders, num_epochs=25, lr=0.001)
```

```
train loss 0.018, train acc 0.995, val loss 7.289, val acc 0.774
23240.2 examples/sec on mps
```



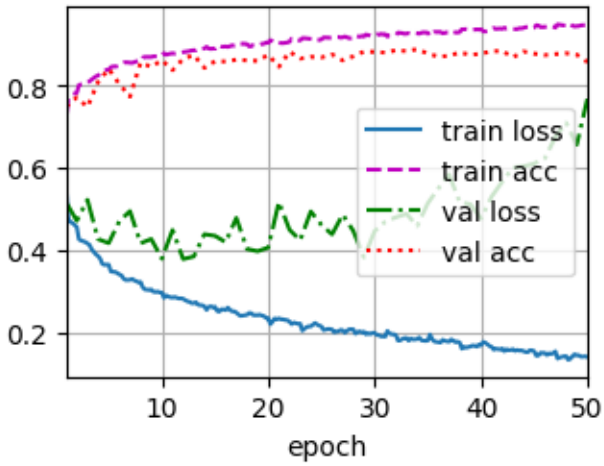
[40]:
```
train(ConvNet(17), chunked_loaders, num_epochs=50, lr=0.01)
```

```
train loss 0.075, train acc 0.974, val loss 2.744, val acc 0.859
16146.7 examples/sec on mps
```



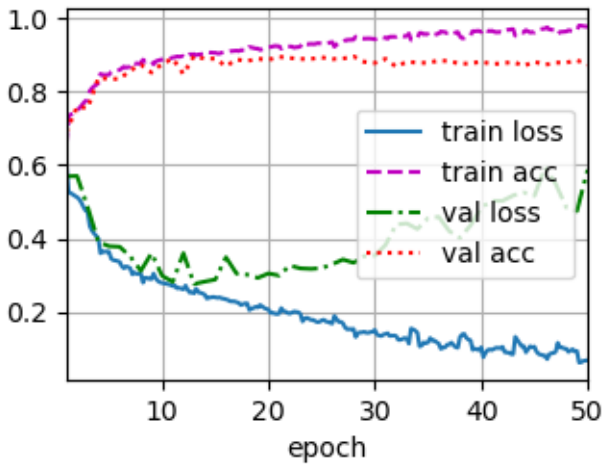
[37]:
```
train(ConvNet(17), chunked_loaders, num_epochs=50, lr=0.001)
```

```
train loss 0.140, train acc 0.948, val loss 0.768, val acc 0.857
12848.3 examples/sec on cpu
```



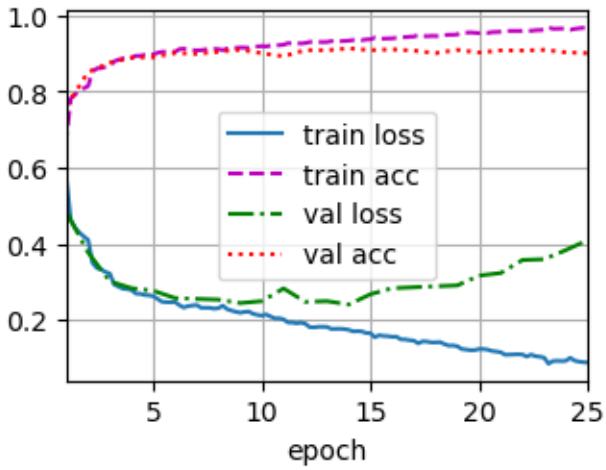
[41]:
```
train(RNNNet(17), chunked_loaders, num_epochs=50, lr=0.001)
```

```
train loss 0.068, train acc 0.977, val loss 0.587, val acc 0.874
6025.4 examples/sec on mps
```



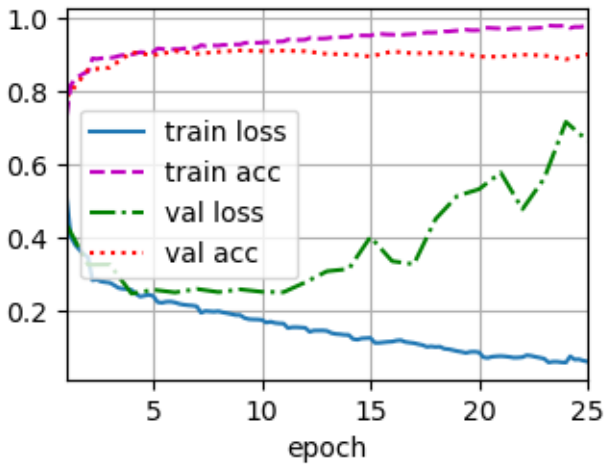
[31]:
```
train(LSTMNet(17), chunked_loaders, num_epochs=25, lr=0.001)
```

```
train loss 0.087, train acc 0.969, val loss 0.409, val acc 0.902
1496.5 examples/sec on mps
```



[49]:
```
train(TorchLSTMNet(17), chunked_loaders, num_epochs=25, lr=0.001)
```

```
train loss 0.063, train acc 0.978, val loss 0.666, val acc 0.901
14193.8 examples/sec on mps
```



1.8 5.6 Discussion (11 points)

(a) Briefly discuss and compare the performance of the models in your experiments. Which worked best and why? (2 points)

TODO: Your answer here.

- *MeanSpectrumNet*: This network performs quite poorly on both the training accuracy (0.709) and the validation accuracy (0.581). Additionally the validation loss rises continuously. I expect that taking the mean spectrum (mean over the steps dimension) is not benefiting the network. This is because we are losing information in the time dimension by just taking the average, and this lost information may be crucial to determining if the signal is coming from a healthy heart or not. Additionally, our network is relatively large given the size of the problem (17 input features and 2 outputs but with our first layer at 128 inputs). This is also what is likely causing the overfitting behavior where the validation loss keeps increasing.
- *FullyConnectedNet*: This network performs better than MeanSpectrumNet, likely because we are not losing important temporal data by taking the mean over the time dimension. Additionally the size of the first layer is roughly the same size as the input features - 512 vs 680 respectively. This explains the better performance of this network, with train accuracy = 0.99, and validation accuracy = 0.774. Given that the validation loss keeps rising the network still seems to be overfitting. The validation loss curve keeps rising because the network makes more less confident predictions over time as it overfits more and more.
- *ConvNet*: The ConvNet worked really well with a training accuracy of 0.97 and a validation accuracy of 0.86, the best validation accuracy so far. However we may still be overfitting, though not to the same extent as the previous networks since the valiation loss seems to jump up and down beyond epoch 30. It's possible that the initial learning rate of 0.01 is too high and the network is overshooting the optimal value. I created a second simulation with a learning rate of 0.001 to see if the validation loss will stabilize. Indeed the lower learning rate seems to have stabilised the validation loss (), though near epoch 40 we begin to increase, indicating an even smaller learning rate may be necessary.
- *RNNNet*: The RNN net also does very well with training accuracy of 0.97 and validation accuracy of 0.87 - this seems to be the most natural network architecture to use since they

are designed to handle time series data and thus can capture intricacies of the data that the simpler architectures cannot. However, we might expect an LSTM to perform even better given that it can keep track of data in the past better and does not forget as quickly.

- *(Torch)LSTMNet*: Sure enough the (Torch)LSTMNet's perform the best with train accuracy around 0.97 and validation accuracy around 0.9 - again this is expected since these models are built to handle time series data but are also preferable to a standard RNN because they don't suffer from vanishing and exploding gradients.

(b) Why do some of those models generalize better than others? (2 points)

As stated in the previous answer, it seems that MeanSpectrumNet peforms quite poorly because it averages across all the timesteps and loses important temporal data that could be important in identifying the condition. For the FullyConnectedNet, it appears to be overfitting very quickly, probably because the learning rate is too high and because there are too many parameters given the size of the input. Further, while it does not lose the temporal data like the MeanSpectrumNet, it also does not consider the order of the signals in time. Somewhat surprisingly the ConvolutionNet generalizes quite well, and this may be due to the fact that compared to a fully connected layer, convolutional layers typically have far fewer parameters, which would reduce the likelihood of overfitting. We see that the ConvNet performs roughly as well as the RNN when we set the learning rate to 0.001. RNN generalizes even better, likely because it considers the time order of the signal which may be important in classification. The (Torch)LSTMNet networks both generalized the best, not only because they have the advantage of an RNN, but also because through additional gates we preserve long-term dependencies and overcome vanishing and exploding gradients.

(c) How does your LSTM implementation compare with the PyTorch implementation? (1 point)

They perform roughly the same, however our LSTM implementation actually performs slightly better, with a validation accuracy of 0.902 vs the torch accuracy of 0.901 - but this is probably due to random noise. However the torch implementation is much faster, it processes 14193.8 examples/sec bs 1496.5 examples/sec in our implementation, almost 10 times as fast! This would indicate that the torch library has been highly optimised.

(d) Your RNN model probably didn't work well. Why is that model more difficult to train than the LSTM? (1 point)

Our RNN model actually performed decently well, but is harder to train than the LSTM because it cannot handle long-term dependencies as well. In the vanilla RNN architecture we can easily forget information at the start of the sequence if the sequence is very long. With an LSTM that has the forget, input, and output gates along with the memory cell internal state, the network can remember long term information.

(e) The convolutional network and the LSTM in these experiments both work on the time dimension. What is an advantage of the convolutional network over the LSTM? (1 points)

The convolutional neural network has the advantage of being able to process all time steps in parallel, while the LSTM must do its training sequentially, making the ConvNet much quicker (assuming you have a CPU or GPU that can perform the operations in parallel). This shows in the statistics of the runs - the ConvNet is about ten times quicker processing 12848 examples/sec compared to the 1496 examples/sec of the LSTMNet.

(f) What is an advantage of the LSTM over a convolutional network? (1 points)

LSTMS were built in part to better handle long range dependencies, whereas ConvNets use local receptive fields and can capture more local patterns. Additionally, because LSTMs process data sequentially, they can better capture important information related to time order of events. LSTMs may also perform better on non-stationary data, or where the data distribution changes over time, whereas ConvNets tend to assume stationary data.

(g) For reasons of speed, we used a fairly small window of 40 time steps. Suppose that we would make this window much larger. How do you think this would affect each model? (2 points)

FullyConnectedNet: If the number of timesteps increased drastically, rather than having 40×17 inputs we would have say 400×17 inputs to the first fully conencted layer with 512 nodes. In this case our network would likely underfit the model as it would not have enough parameters to capture the complexity.

MeanSpectrumNet: Likely noething would change in this network since we are taking the mean over time. It would likely have a similar performance.

ConvNet: This network would of course train more slowly, and may have lower validation accuracy as it might struggle to handle longer term dependencies without additional convolutional layers to increase it's receptive field. Expect degraded performance.

RNNNet: Again expect degraded validation and training accuracy and longer training times because the network may struggle to handle the really long term dependencies that may be important.

LSTMNet: Training time would likely increase substantially, however validation and training accuracy likely would be unaffected as these models excel with longer sequences of data.

(h) One of the difficulties with recurrent networks is that inputs from early steps are quite far away from the final result. How would you suggest to reduce that problem? (1 point)

Using LSTMs or GRUs are one way to keep relevant information from the past. Adding residual connections may also help with long term dependencies and stable gradients.

1.9 5.7 Data augmentation

Especially if your dataset is small, data augmentation can help to improve the performance of your network.

We have an easy way to add some data augmentation to the ECG dataset. In our preprocessing, we divided each recording into small chunks of 40 time steps, which we then reused in every epoch. We can add more variation to the training set by creating chunks at random positions.

The DataLoader class in PyTorch has a `collate_fn` parameter to which we can pass a function. This function is called for each minibatch in each epoch. We will use this to extract a random chunk from each sample.

The function `random_chunk_collate_fn` takes a minibatch of samples, chooses a random offset for each sample, extracts a small chunk at that position, and then concatenates and returns the result.

[38]:
```
def random_chunk_collate_fn(samples, window_size):
    # Take a list of tensors of (steps_i, features),
    # extract a random window of length window_size from each tensor,
    # concatenate to a tensor of shape (samples, window_size, features).
    x_batch = torch.empty((len(samples), window_size) + samples[0][0].shape[1:],
                          device=samples[0][0].device, dtype=samples[0][0].
 ↪dtype)
    y_batch = torch.empty((len(samples),),
                          device=samples[0][1].device, dtype=samples[0][1].
 ↪dtype)
    for i, (x, y) in enumerate(samples):
        # extract a random window
        offset = torch.randint(x.shape[0] - window_size, (1,))
        x_batch[i, :] = x[offset:offset + window_size]
        y_batch[i] = y
    return x_batch, y_batch
```

We construct a new DataLoader for our training set:

[39]:
```
# test to see the x and y shapes for one sample
random_chunk_loaders = {
    'train': torch.utils.data.DataLoader(data_train, shuffle=True,␣
 ↪batch_size=batch_size,
                                         collate_fn=lambda s:␣
 ↪random_chunk_collate_fn(s, window_size=40)),
    'val':   chunked_loaders['val']
}
for (x, y) in random_chunk_loaders['train']:
    print(x.shape, y.shape)
    break
```

```
torch.Size([192, 40, 17]) torch.Size([192])
```

Observe that the pre-chunked dataset was much larger than the new dataset with on-the-fly chunking. You might want to increase the number of training epochs a bit to make sure that the network sees a similar number of examples.

[40]:
```
print('Minibatches in chunked_loader[\'train\']:      ',␣
 ↪len(chunked_loaders['train']))
print('Minibatches in random_chunk_loaders[\'train\']:',␣
 ↪len(random_chunk_loaders['train']))
```

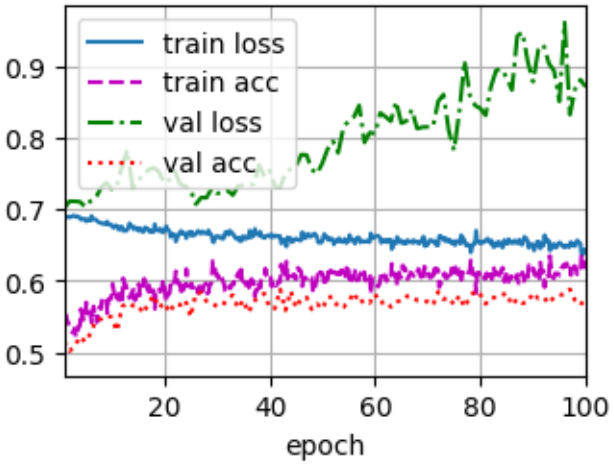
```
Minibatches in chunked_loader['train']:       355
Minibatches in random_chunk_loaders['train']: 42
```

Let's see how this data augmentation method affects the performance of your networks.

(a) Train the MeanSpectrumNet, FullyConnectedNet, ConvNet and TorchLSTMNet from the previous experiments on data from the `random_chunk_loaders`.

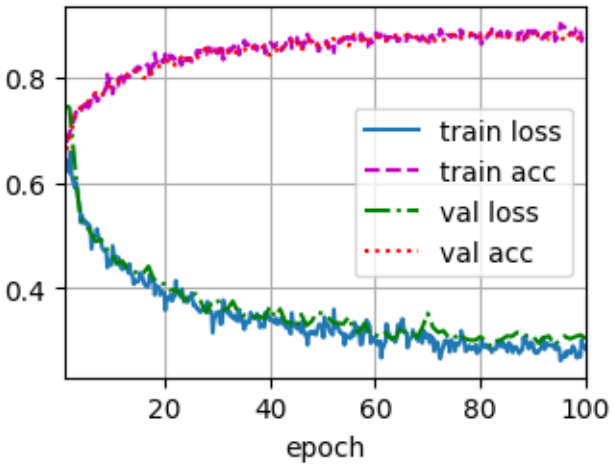
[41]:
```
train(MeanSpectrumNet(), random_chunk_loaders, num_epochs=100, lr=0.001)
```

```
train loss 0.643, train acc 0.623, val loss 0.871, val acc 0.577
3782.1 examples/sec on cpu
```



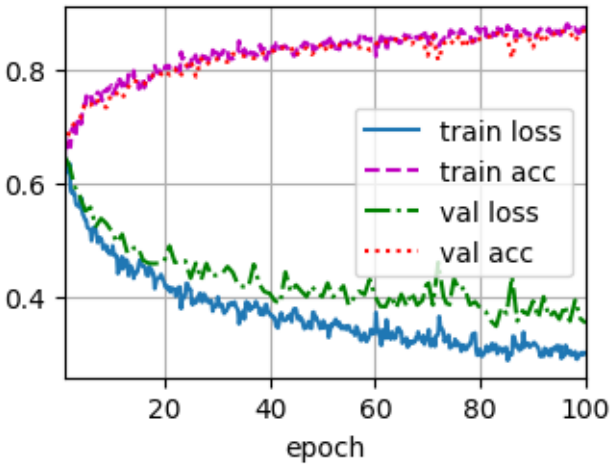
[42]:
```
train(FullyConnectedNet(40 * 17), random_chunk_loaders, num_epochs=100, lr=0.
↪001)
```

```
train loss 0.285, train acc 0.881, val loss 0.304, val acc 0.882
3481.6 examples/sec on cpu
```



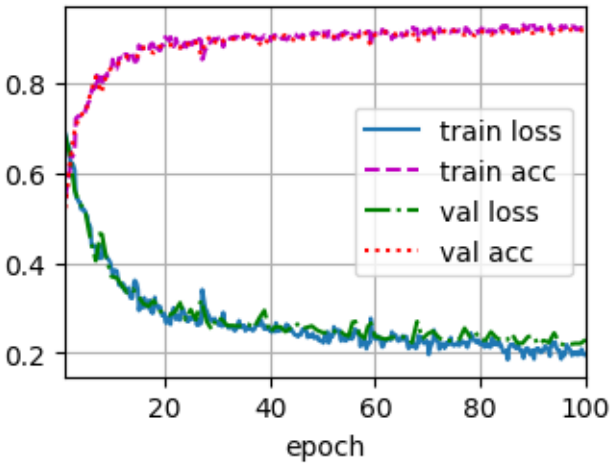
[43]:
```
train(ConvNet(17), random_chunk_loaders, num_epochs=100, lr=0.001)
```

```
train loss 0.302, train acc 0.872, val loss 0.355, val acc 0.871
3108.0 examples/sec on cpu
```



[44]:
```
train(TorchLSTMNet(17), random_chunk_loaders, num_epochs=100, lr=0.001)
```

```
train loss 0.198, train acc 0.924, val loss 0.229, val acc 0.917
1885.8 examples/sec on cpu
```



1.10 5.8 Discussion (9 points)

(a) How does the data augmentation influence the training and validation results? Can you explain this? (2 points)

Overall the data augmentation seems to improve validation accuracy and reduce overfitting across the tested networks, with the exception of the MeanSpectrumNet. In the MeanSpectrumNet, the data augmentation does seem slightly reduce the rate of overfitting, though taking the average across time on each frequency channel still leads to very bad performance as we are losing a lot of valuable data. the FullyConnectedNet is much improved because it is harder to memorise patterns

with the randomly chunked data and thus it is harder to overfit. Previously when we divided each sample in chunks of 40 timesteps and resued them every epoch, the models were seeing the same data in the same order every epoch, and thus it was possible they were learning patterns in the way data was presented. The ConvNet and the TorchLSTMNet also saw improvements, though not as significant as the FullyConnected network. Without the data augmentation these two architectures performed well, but were beginning to show signs of overfitting near the end of their training. Adding the augmented data stopped this overfitting.

(b) Why does the data augmentation affect some models more than others? (1 point)

Data augmentation did not make a huge difference in the performance of the MeanSpectrumNet, this is because the architecture is fundamentally incompatible with the data, and no amount of data augmentation will fix this. All the other network architectures are reasonably compatible with the data, so adding data augmentation can improve the performance in these networks by making it more difficult for the network to memorise patterns by showing data chunks in random order across epochs.

(c) Should we also do data augmentation on the validation set? Why, or why not? (1 point)

It is not recommended to do data augmentation on the validation set. The purpose of data augmentation is to improve generalization during training by presenting the network with variations of the original data so that it is more robust. The purpose of the validation set is to provide an unbiased estimage of how well the model would perform on data distributions in the real world. Thus it is important to keep the validation set as close as possible to data from the real world, and not to modify it with data augmentation, which should only be used during training.

(d) Data augmentation is often a good way to add some domain knowledge to your model. Based on your knowledge of ECGs, why is (or isn't) our augmentation method a good idea? (1 point)

The data augmentation seems to be a good idea as the models are less prone to overfitting on the training set. However, the suitability of this approach heavily depends on the size of the chunks for ECG data specifically. This is because many heart conditions are periodic and issues are detected by irregular heart beats or iregeular timing between heartbeats, and our chunks need to be large enough to capture full cardiac cycles, without being so large that they have cyclical and redundant data. 40 timesteps might be too small to capture a complete cycle and something closer to 200 to 300 might be better…

(e) Give an example of another suitable augmentation method and explain why it would work for this data. (2 points)

We could try to introduce some guassian noise on top of our signals, which would force the model to focus on the overall cardiac cycles, both the size of the signal and the spae between signals, rather than on smaller artifacts that tell us little. This would help the model generlize and reduce overfitting.

(f) Give an example of an augmentation method that might be suitable for other data but would probably not work here. Explain why. (2 points)

Magnitude warping changes the magnitude of the signal at each step, and could be applied to other time series data where the magnitude is important for classification. However, for the problem of

atrial fibirillation, the magnitude of the signal does not seem to tell us much about the problem, rather it is the interval between spikes in the signal that suggests a problem.

1.11 The end

Well done! Please double check the instructions at the top before you submit your results.

This assignment has 31 points. Version 231feba / 2024-10-03