Deep Learning — Assignment 3

Third assignment for the 2023 Deep Learning course (NWI-IMC070) of the Radboud University.

Names:

Group:

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.

Objectives

In this assignment you will

1. Experiment with convolutional neural networks
2. Train a convolutional neural network on a speech dataset
3. Investigate the effect of dropout and batch normalization
4. Define and train a residual neural network

Required software

If you haven't done so already, you will need to install the following additional libraries:

- `torch` and `torchvision` for PyTorch,
- `d2l`, the library that comes with Dive into deep learning book,
- `python_speech_features` to compute MFCC features.

All libraries can be installed with `pip install`.

In [ ]:
```
%matplotlib inline
import os
import numpy as np
import matplotlib.pyplot as plt
from d2l import torch as d2l
import torch
from torch import nn
from scipy.io import wavfile

# Fix the seed, so outputs are exactly reproducible
torch.manual_seed(12345)
```

3.1 Convolution and receptive fields (9 points)

We will first define some helper functions to plot the receptive field of a node in a network.

In [ ]:
```
def show_image(img, title=None, new_figure=True):
    if new_figure:
        plt.figure(figsize=(5, 5))
    im = plt.imshow(img, interpolation='none', aspect='equal', cmap='gray')
    ax = plt.gca();

    # plot pixel numbers and grid lines
    ax.set_xticks(np.arange(0, img.shape[1], 1))
    ax.set_yticks(np.arange(0, img.shape[0], 1))
    ax.set_xticklabels(np.arange(0, img.shape[1], 1))
    ax.set_yticklabels(np.arange(0, img.shape[0], 1))
    ax.set_xticks(np.arange(-.5, img.shape[1], 1), minor=True)
    ax.set_yticks(np.arange(-.5, img.shape[0], 1), minor=True)
    ax.grid(which='minor', color='gray', linestyle='-', linewidth=1.5)

    # hide axis outline
    for spine in ax.spines.values():
        spine.set_visible(False)

    if title is not None:
        plt.title(title)

# set all weights in the network to one,
# all biases to zero
def fill_weights_with_ones(network):
    for name, param in network.named_parameters():
        if 'weight' in name:
            param.data = torch.ones_like(param.data)
        elif 'bias' in name:
            param.data = torch.zeros_like(param.data)
    return network

def compute_receptive_field(network, input_size=(15, 15), binary=True):
    assert isinstance(network, torch.nn.Sequential), 'This only works with t
    for layer in network:
        if not isinstance(layer, (torch.nn.Conv2d, torch.nn.AvgPool2d)):
```

```
            raise Exception('Sorry, this visualisation only works for Conv2c

    # initialize weights to ones, biases to zeros
    fill_weights_with_ones(network)

    # find the number of input and output channels
    input_channels = None
    output_channels = None
    for layer in network:
        if isinstance(layer, torch.nn.Conv2d):
            if input_channels is None:
                # first convolution layer
                input_channels = layer.in_channels
            output_channels = layer.out_channels
    if input_channels is None:
        input_channels = 1

    # first, we run the forward pass to compute the output shape give the ir

    # PyTorch expects input shape [samples, channels, rows, columns]
    x = torch.zeros(1, input_channels, *input_size)
    x.requires_grad = True

    # forward pass: apply each layer in the network
    y = x
    y.retain_grad()
    ys = [y]
    for layer in network:
        y = layer(y)
        # keep track of the intermediate values so we can plot them later
        y.retain_grad()
        ys.append(y)

    # second, we run the backward pass to compute the receptive field

    # create gradient input: zeros everywhere, except for a single pixel
    y_grad = torch.zeros_like(y)
    # put a one somewhere in the middle of the output
    y_grad[0, 0, (y_grad.shape[2] - 1) // 2, (y_grad.shape[3] - 1) // 2] = 1

    # compute the gradients given this single one
    y.backward(y_grad)

    # receptive field is now in the gradient at each layer
    receptive_fields = []
    for y in ys:
        # the gradient for this layer shows us the receptive field
        receptive_field = y.grad
        if binary:
            receptive_field = receptive_field > 0
        receptive_fields.append(receptive_field)
    return receptive_fields

def plot_receptive_field(network, input_size=(15, 15), binary=True):
    receptive_fields = compute_receptive_field(network, input_size, binary)
```

```
    # plot the gradient at each layer
    plt.figure(figsize=(4 * len(receptive_fields), 4))
    for idx, receptive_field in enumerate(receptive_fields):
        plt.subplot(1, len(receptive_fields), idx + 1)
        # the last element of ys contains the output of the network
        if idx == len(receptive_fields) - 1:
            plot_title = 'output (%dx%d)' % (receptive_field.shape[2], recep
        else:
            plot_title = 'layer %d input (%dx%d)' % (idx, receptive_field.sh
        # plot the image with the receptive field (sample 0, channel 0)
        show_image(receptive_field[0, 0], new_figure=False, title=plot_title
        if not binary:
            plt.colorbar(fraction=0.047 * receptive_field.shape[0] / recepti

def receptive_field_size(network, input_size=(15, 15), binary=True):
    receptive_fields = compute_receptive_field(network, input_size, binary)
    return torch.count_nonzero(torch.flatten(receptive_fields[0][0,0]))
```

Using these functions, we can define a network and plot the receptive field of a pixel in the output.

(a) Run the code to define a network with one 3×3 convolution layer and plot the images.

In [ ]:
```
net = torch.nn.Sequential(
    torch.nn.Conv2d(1, 1, kernel_size=(3, 3)),
)
plot_receptive_field(net, input_size=(15, 15))
```

Read these images as follows:

- On the left, you see the input size of the network (here: 15 x 15 pixels) and the receptive field for one pixel in the output.
- On the right, you see the output size of the network (here: 13 x 13 pixels).

To visualize the receptive field of this network, we used the following procedure:

- We selected one pixel of the output (shown as the white pixel in the center in the image on the right).
- We computed the gradient for this pixel and plotted the gradient with respect to the input (the image on the left).
- This shows you the receptive field of the network: the output for the pixel we selected depends on these 9 pixels in the input.

(b) Use this method to plot the receptive field of a pixel in the output of a convolution layer with a kernel size of 5×5. (1 point)

In [ ]:
```
# TODO Plot the receptive field of a 5x5 convolution.
```

If you look at the result, you will see that two things have changed: the receptive field and the output size.

(c) How do the receptive field size and the output size depend on the kernel size? Give a formula. (1 point)

TODO Your answer here.

Counting the number of parameters

In the previous question, you saw how the receptive fields of a 3x3 convolution differs from a 5x5 kernel convolution. But this is not the only difference: there is also a difference in the number of parameters in the network.

We can count the number of parameters in the network by computing the number of elements (e.g., the weights and biases in a convolution kernel) in the parameter list of the PyTorch network.

We'll define a small helper function to do this:

In [ ]:
```
def print_parameter_count(network):
    # sum the number of elements in each parameter of the network
    count = sum([param.data.numel() for param in network.parameters()])
    print('%d parameters' % count)
```

(d) Use the function to count the number of parameters for a 3x3 convolution.

In [ ]:
```
net = torch.nn.Sequential(
    torch.nn.Conv2d(1, 1, kernel_size=(3, 3)),
)
plot_receptive_field(net, input_size=(15, 15))
print_parameter_count(net)
```

(e) Do the same to count the number of parameters for a 5x5 convolution. (1 point)

In [ ]:
```
# TODO count the number of parameters of a 5x5 convolution
```

(f) Explain the results by showing how to *compute* the number of parameters for the 3x3 and 5x5 convolutions. (1 point)

TODO Your answer here.

For these computations we used convolution layers with one input and one output channel.

We can also compute the results for a layer with a different number of channels.

(g) Define a network with a 5x5 convolution, 2 input channels and 3 output channels. Print the number of parameters.

In [ ]:
```
# TODO count the number of parameters of a 5x5 convolution with 2 input char
```

(h) Show how to compute the number of parameters for this case. **(1 point)**

TODO Your answer here.

Preserving the size of the input image

The PyTorch documentation for `torch.nn.Conv2d` describes the parameters that you can use to define a convolutional layer. We will explore some of those parameters in the next questions.

In the previous plot, you may have noticed that the output (13x13 pixels) was slightly smaller than the input (15x15 pixels).

(i) Define a network with a single 3x3 convolutional layer that produces an output that has the same size as the input. **(1 point)**

Use 1 input and 1 output channel.

In [ ]:
```
# TODO Define a network with a 3×3 kernel size that takes a 15×15 input imag
#      and produces a 15×15 output image.
```

(j) Define a network with a single 5x5 convolutional layer that preserves the input size. **(1 point)**

In [ ]:
```
# TODO Define a network with a 5×5 kernel size that takes a 15×15 input imag
#      and produces a 15×15 output image.
```

Play around with some other values to see how this parameter behaves.

Multiple layers

As you have just seen, one way to increase the size of the receptive field is to use a larger convolution kernel. But another way is to use more than one convolution layer.

(k) Define a network with two 3x3 convolutions, preserving the image size. Show the receptive field and the number of parameters. **(1 point)**

For this visualisation, do not use any activation functions, and use 1 channel everywhere.

In [ ]:
```
# TODO define a network with two 3x3 convolutions
net = torch.nn.Sequential()
print(net)
plot_receptive_field(net, input_size=(15, 15))
print_parameter_count(net)
```

Since we now have two layers, the visualization shows an extra image. From right to left, we have:

- Right: the output size and a single active pixel.
- Middle: the receptive field for the single output pixel between the first and second convolution.
- Left: the receptive field for the single output pixel in the input image.

We have now tried two ways to increase the receptive field size: increasing the kernel size, and using multiple layers.

(l) Compare the number of parameters required by the two options. Which one is more parameter-efficient? (1 point)

TODO Your answer here.

3.2 Variations on convolution (8 points)

Pooling

We can also increase the size of the receptive field by using a pooling layer.

(a) Construct a network with a 3x3 convolution (preserving the input size) followed by a 2x2 average pooling. Plot the receptive field and print the number of parameters. (1 point)

Use 1 input and 1 output channel.

In [ ]:
```
# TODO define a network with two 3x3 convolutions
net = torch.nn.Sequential()
print(net)
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
```

(b) Explain the number of parameters in this convolution + pooling network. (1 point)

TODO Your answer here.

Dilation

A third option to increase the receptive field is *dilation*.

(c) Define a network with 3x3 convolution with dilation that preserves the input size. (1 point)

In [ ]:
```
# TODO define a network with one 3x3 convolution with dilation
net = torch.nn.Sequential()
# the output should also be 15x15 pixels
print(net)
plot_receptive_field(net, input_size=(15, 15))
print_parameter_count(net)
```

(d) Explain how dilation affects the receptive field. (1 point)

TODO Your answer here.

Using strides

By default, convolution layers use a stride of 1.

(e) Change the network to use a stride of 2 and plot the result. (1 point)

In [ ]:
```
# TODO increase the stride to 2
net = torch.nn.Sequential(
    torch.nn.Conv2d(1, 1, kernel_size=(3, 3), padding=(1, 1)),
)
print(net)
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
```

(f) Explain the new output size and compare the result with that of pooling. (1 point)

TODO Your answer here.

(g) Explain how the stride affects the receptive field of this single convolution layer. (1 point)

TODO Your answer here.

(h) Explain the number of parameters for this network. (1 point)

TODO Your answer here.

3.3 Combining layers (7 points)

As you have seen, there are multiple ways to increase the receptive field. You can make interesting combinations by stacking multiple layers.

Let's try a few ways to make networks with a large receptive field. For each of the questions in this section:

- Create a network where a pixel in the output has a 9x9 receptive field.
- Use 3 input channels and 3 output channels in every layer.
- In convolution layers, try to preserve the input size as much as possible.

(a) Make a network with a single convolution that satisfies the above conditions. **(1 point)**

In [ ]:
```
# TODO
net = ...
print(net)
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
assert receptive_field_size(net) == 9*9, "Receptive field of output pixel sh
```

Many popular network architectures use a sequence of 3x3 convolutions.

(b) Use only 3x3 convolutions. **(1 point)**

In [ ]:
```
# TODO
net = ...
print(net)
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
assert receptive_field_size(net) == 9*9, "Receptive field of output pixel sh
```

(c) Use a 2x2 average pooling layer in combination with one or more 3x3 convolutions. **(1 point)**

In [ ]:
```
# TODO
net = ...
print(net)
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
assert receptive_field_size(net) == 9*9, "Receptive field of output pixel sh
```

(d) Copy the previous convolution + pooling network and replace the pooling layer with a strided convolution layer. **(1 point)**

In [ ]:
```
# TODO
net = ...
print(net)
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
assert receptive_field_size(net) == 9*9, "Receptive field of output pixel sh
```

(e) Construct a network with exactly two 3x3 convolutions. Use dilation to get a receptive field of 9x9 pixels. **(1 point)**

In [ ]:
```
# TODO
net = ...
print(net)
plot_receptive_field(net, input_size=(14, 14))
print_parameter_count(net)
assert receptive_field_size(net) == 9*9, "Receptive field of output pixel sh
```

(f) For each of the methods, list the number of layers, the number of parameters, and the size of the output of the network:

Method	Layers	Parameters	Output size
One 9x9 convolution	1	732	14x14
Many 3x3 convolutions			
With pooling			
With strided convolution			
With dilation			

(g) Compare the methods in terms of the number of parameters. **(1 point)**

TODO Your answer here.

(h) Compare the methods in terms of the output size. How much downsampling do they do? **(1 point)**

TODO Your answer here.

3.4 Padding in very deep networks (2 points)

Without padding, the output of a convolution is smaller than the input. This limits the depth of your network.

(a) How often can you apply a 3x3 convolution to a 15x15 input image?

In [ ]:
```
# find the maximum number of layers
number_of_times = 25

# create a 15x15 input
x = torch.zeros(1, 1, 15, 15)
print('input size: %dx%d' % (x.shape[2], x.shape[3]))

# create a 3x3 convolution
conv = torch.nn.Conv2d(1, 1, kernel_size=(3, 3))

for n in range(number_of_times):
    # apply another convolution
```

```python
    x = conv(x)
    print('layer %d, output size: %dx%d' % (n + 1, x.shape[2], x.shape[3]))
```

Earlier in this assignment, you have used padding to address this problem. This seems ideal.

(b) Copy the previous code, add some padding, and show that we can now have an infinite number of layers.

(We are computer scientists and not mathematicians, so for the purpose of this question we'll consider 'infinite' to be equal to 25.)

In [ ]:
```python
# TODO Your code here.
```

(c) Does it really work like this? Have a look at the following experiment.

- We simulate a convolution network with 25 convolution layers, with 3x3 kernels and the right amount of padding.
- We set the weights to 1/9 (so that the sum of the 3x3 kernel is equal to 1) and set the bias to zero.
- We give this network a 15x15-pixel input filled with ones.
- We plot the output of layers 5, 10, 15, 20, and 25.

In [ ]:
```python
# create a 15x15 input filled with ones
x = torch.ones(1, 1, 15, 15)

# create a 3x3 convolution
conv = torch.nn.Conv2d(1, 1, kernel_size=(3, 3), padding=(1, 1))

# set weights to 1/9 (= sum to one), bias to zero
conv.weight.data = torch.ones_like(conv.weight.data) / 9
conv.bias.data = torch.zeros_like(conv.bias.data)

plt.figure(figsize=(10, 2))
for n in range(1, 26):
    # apply another convolution
    x = conv(x)
    # print('layer %d, output size: %dx%d' % (n + 1, x.shape[2], x.shape[3]))
    if n % 5 == 0:
        plt.subplot(1, 5, n // 5)
        plt.imshow(x[0, 0].detach().numpy(), cmap='gray')
        plt.axis('off')
        plt.title('layer %d' % n)
```

(d) Explain the pattern that we see in the output of the final layers. How does this happen, and what does this mean for our very deep networks? (2 points)

TODO Your answer here.

3.5 Spoken digits dataset (4 points)

Time for some practical experiments. The d2l book uses a dataset of images as a running example (FashionMNIST). In this assignment we will investigate CNNs in a completely different domain: speech recognition.

The dataset we use is the free spoken digits dataset, which can be found on https://github.com/Jakobovski/free-spoken-digit-dataset. This dataset consists of the digits 0 to 9, spoken by different speakers. The data comes as .wav files.

(a) Use the commands below (or a similar tool) to download the dataset. You can also use `git clone` to clone the repository mentioned above.

In [ ]:
```
#! mkdir -p free-spoken-digit-dataset
#! wget -O - https://github.com/Jakobovski/free-spoken-digit-dataset/archive
```

Below is a function to load the data. We pad/truncate each sample to the same length. The raw audio is usually stored in 16 bit integers, with a range -32768 to 32767, where 0 represents no signal. Before using the data, it should be normalized. A common approach is to make sure that the data is between 0 and 1, between -1 and 1, or zero-mean unit-variance. Not all of these work well on this data, so later on, if your network doesn't seem to learn anything: try a different method to see if that works better.

(b) Update the below code to normalize the data to a reasonable range.

(1 point)

In [ ]:
```
samplerate = 8000
def load_waveform(file, size = 6000):
    samplerate, waveform = wavfile.read(file)
    # Take first 6000 samples from waveform. With a samplerate of 8000 that
    # Pad with 0s if the file is shorter
    waveform = np.pad(waveform,(0,size))[0:size]
    # Normalize waveform
    # TODO: Your code here.
    return waveform
```

The following code loads all .wav files in a directory, and makes it available in a pytorch dataset.

(c) Load the data into a variable `data`.

In [ ]:
```
class SpokenDigits(torch.utils.data.Dataset):
    def __init__(self, data_dir):
        digits_x = []
        digits_y = []
        for file in os.listdir(data_dir):
```

```
            if file.endswith(".wav"):
                waveform = load_waveform(os.path.join(data_dir, file))
                label = int(file[0])
                digits_x.append(waveform)
                digits_y.append(label)
        # convert to torch tensors
        self.x = torch.from_numpy(np.array(digits_x, dtype=np.float32))
        # add an extra dimension to represent the "channels" (we start with
        self.x = self.x.unsqueeze(1)
        self.y = torch.from_numpy(np.array(digits_y))

    def __len__(self):
        return len(self.x)

    def __getitem__(self, idx):
        return self.x[idx], self.y[idx]

# TODO: Your code here.
```

(d) Describe the dataset: how many samples are there, how many features does each sample have? How many classes are there? (1 point)

In [ ]:
```
# TODO Your answer here.
```

Here is code to play samples from the dataset to give you an idea what it "looks" like.

Note: If this step doesn't work in your notebook, then you can ignore it.

In [ ]:
```
from IPython.display import Audio
def play(sample):
    print(f'Label: {sample[1]}')
    return Audio(sample[0][0].numpy(), rate=samplerate)
play(data[0])
```

Before continuing, we split the dataset into a training and a test set.

In [ ]:
```
train_prop = 2/3
train_count = int(len(data) * train_prop)
train, test = torch.utils.data.random_split(data, [train_count, len(data) -
```

The code above uses 2/3 of the data for training.

(e) Discuss an advantage and disadvantage of using more of the data for training. (2 points)

TODO Your answer here.

Finally, we split the data into batches:

In [ ]:
```
data_params = {'batch_size': 32}
train_iter = torch.utils.data.DataLoader(train, **data_params)
```

```
test_iter  = torch.utils.data.DataLoader(test,  **data_params)
```

3.6 One-dimensional convolutional neural network (8 points)

We will now define a network architecture. We will use a combination of convolutional layers and pooling. Note that we use 1d convolution and pooling here, instead of the 2d operations used for images.

(a) Complete the network architecture, look at the d2l book chapter 7 and chapter 8 for examples. (2 points)

In [ ]:
```
def build_net():
    return torch.nn.Sequential(
        nn.Conv1d(1, 4, kernel_size=5), nn.ReLU(),
        nn.AvgPool1d(kernel_size=2, stride=2),
        # TODO: Add three more convolutional layers, ReLU layers and pooling
        #       doubling the number of channels each time
        # TODO: Your code here.
        nn.Flatten(),
        nn.Linear(11872, 128), nn.ReLU(),
        nn.Linear(128, 64), nn.ReLU(),
        nn.Linear(64, 10))
```

(b) The first fully connected layer has input dimension 11872, where does that number come from? (1 point)

TODO Your answer here.

Hint: think about how (valid) convolutional layers and pooling layers with stride affect the size of the data.

(c) How many parameters are there in the model? I.e. the total number of weights and biases. (1 point)

In [ ]:
```
# TODO: Compute the number of parameters
# Hint: use net.parameters() and param.nelement()
```

(d) Suppose that instead of using convolutions, we had used only fully connected layers, while keeping the number of features on each hidden layer the same. How many parameters would be needed in that case approximately? (1 point)

TODO Your answer here.

The FashionMNIST dataset used in the book has 60000 training examples. How large is our training set? How would the difference affect the number of epochs that we need? Compare to chapter 7.6 and chapter 8.1 of the book.

(e) How many epochs do you think are needed? (1 point)

In [ ]:
```
lr, num_epochs = 0.01, 10 # TODO: change
```

We will use the code from (a previous edition of) the d2l book to train the network. In particular, the `train` function, defined in chapter 7.6. This function is reproduced below:

In [ ]:
```
def train(net, train_iter, test_iter, num_epochs, lr, device = d2l.try_gpu()
    """Train a model with a GPU (defined in Chapter 6)."""
    print('training on', device)
    net.to(device)
    optimizer = torch.optim.SGD(net.parameters(), lr=lr)
    loss = nn.CrossEntropyLoss()
    animator = d2l.Animator(xlabel='epoch', xlim=[1, num_epochs],
                            legend=['train loss', 'train acc', 'test acc'])
    timer, num_batches = d2l.Timer(), len(train_iter)
    for epoch in range(num_epochs):
        # Sum of training loss, sum of training accuracy, no. of examples
        metric = d2l.Accumulator(3)
        net.train()
        for i, (X, y) in enumerate(train_iter):
            timer.start()
            optimizer.zero_grad()
            X, y = X.to(device), y.to(device, torch.long)
            y_hat = net(X)
            l = loss(y_hat, y)
            l.backward()
            optimizer.step()
            with torch.no_grad():
                metric.add(l * X.shape[0], d2l.accuracy(y_hat, y), X.shape[0
            timer.stop()
            train_l = metric[0] / metric[2]
            train_acc = metric[1] / metric[2]
            if (i + 1) % (num_batches // 5) == 0 or i == num_batches - 1:
                animator.add(epoch + (i + 1) / num_batches,
                             (train_l, train_acc, None))
        test_acc = d2l.evaluate_accuracy_gpu(net, test_iter)
        animator.add(epoch + 1, (None, None, test_acc))
    print(f'loss {train_l:.3f}, train acc {train_acc:.3f}, '
          f'test acc {test_acc:.3f}')
    print(f'{metric[2] * num_epochs / timer.sum():.1f} examples/sec '
          f'on {str(device)}')
```

(f) Now train the network.

In [ ]:
```
train(build_net(), train_iter, test_iter, num_epochs=75, lr=0.02)
```

(g) Did the training converge? (2 point)

If the training has not converged, maybe you need to change the number of epochs and/or the learning rate.

Hint: This is a non-trivial problem, so your network might take some time to learn. Don't give up too quickly, it might take 50-100 epochs before you see any significant changes in the loss curves.

TODO: Document the runs that you have performed and thir results in the table below.

Experiment	epochs	lr	train accuracy	test accuracy	converged?
experiment 1	1234	1234			

3.7 Questions and evaluation (6 points)

(a) Does the network look like it is overfitting or underfitting? Explain how see this. (1 point)

TODO Your answer here.

(b) Is what we have here a good classifier? Could it be used in a realistic application? Motivate your answer. (1 point)

TODO: discuss your answer

(c) Do you think there is enough training data compared to the dimensions of the data and the number of parameters? Motivate your answer. (1 point)

TODO Your answer here.

(d) How could the classifier be improved? Give at least 2 suggestions. (1 point)

TODO Your answer here.

(e) The free spoken digits datasets has recordings from several different speakers. Is the test set accuracy a good measure of how well the trained network would perform for recognizing digits spoken by a new, unknown speaker? And if not, how could that be tested instead? (2 points)

TODO Your answer here.

3.8 Variations (8 points)

One way in which the training might be improved is with dropout or with batch normalization.

(a) Make a copy of the network architecture from 3.6a below, and add dropout. (1 point)

Hint: see chapter 8.1 for an example that uses dropout.

In [ ]:
```
def build_net_dropout():
    return ...  # TODO: your network here

train(build_net_dropout(), train_iter, test_iter, num_epochs=200, lr=0.02)
```

(b) How does dropout change the results? Does this match what you saw on the simple network last week? (1 point)

TODO Your answer here.

(c) Make a copy of the original network architecture, and add batch normalization to all convolutional and linear layers. (1 point)

Hint: see chapter 8.5 for an example.

In [ ]:
```
def build_net_batchnorm():
    return ...  # TODO: your network here

train(build_net_batchnorm(), train_iter, test_iter, num_epochs=15, lr=0.02)
```

(d) How does batch normalization change the results? Does this match what you saw on the simple network last week? (1 point)

TODO Your answer here.

Residual network

We can also try to use a residual network. The book has code for a 2d resnet in Chapter 8.6.

(e) Copy the `Residual` module here, and adapt it for 1d convolutions. Use a kernel size of 5 for the convolution layers. (2 points)

Use residual blocks each containing two convolutional layers.

In [ ]:
```
# TODO: Residual class here
```

(f) Make a copy of the network architecture from 3.6a, and replace the convolutions with residual blocks. (1 point)

In [ ]:
```python
def build_resnet():
    return ...  # TODO: your network here

# TODO train(resnet, train_iter, test_iter, num_epochs)
```

(g) How do residual connections change the results? **(1 point)**

TODO Your answer here.

3.9 Feature extraction (5 points)

Given enough training data a deep neural network can learn to extract features from raw data like audio and images. However, in some cases it is still necessary to do manual feature extraction, in particular when working with smaller datasets like this one. For speech recognition, a popular class of features are MFCCs.

Here is code to extract these features. You will need to install the `python_speech_features` first.

In [ ]:
```python
from python_speech_features import mfcc

def load_waveform_mfcc(file, size = 6000):
    samplerate, waveform = wavfile.read(file)
    waveform = np.pad(waveform,(0,size))[0:size] / 32768
    return np.transpose(mfcc(waveform, samplerate))
```

(a) Implement a variation of the dataset that uses these features. **(2 points)**

In [ ]:
```python
class SpokenDigitsMFCC(torch.utils.data.Dataset):
    # TODO: Your code here.
    ...

data_mfcc = SpokenDigitsMFCC(data_dir) # TODO: your data directory here
train_count_mfcc = int(len(data_mfcc) * train_prop)
train_mfcc, test_mfcc = torch.utils.data.random_split(data_mfcc, [train_cour
train_iter_mfcc = torch.utils.data.DataLoader(train_mfcc, **data_params)
test_iter_mfcc  = torch.utils.data.DataLoader(test_mfcc,  **data_params)

assert next(iter(train_iter_mfcc))[0].shape == torch.Size([data_params['batc
```

The MFCC features will have 13 channels instead of 1 (the `unsqueeze` operation is not needed).

(b) Inspect the shape of the data, and define a new network architecture that accepts data with this shape. **(1 point)**

```python
def build_net_mfcc():
    # TODO: Your code here.
    ...
```

(c) Train the network with the MFCC features. **(1 point)**

```python
# TODO: Your code here.
```

(d) What would be needed to get a fully neural network approach to work as well as MFCC features? **(1 point)**

TODO Your answer here.

The end

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

This assignment has 57 points.

Version 6717cb8 / 2023-09-15