Recurrent Neural Network Example - Classifying Handwritten Digits

This example shows how a recurrent neural network (RNN) can classify handwritten digits. We use the MNIST dataset with 60000 examples of 28×28 images. This task is usually solved by giving the whole image into a neural network with an input layer of dimension $28 \cdot 28 = 784$. We however split the image into a sequence where each step corresponds to a row with the length 28.

You can find a implementation with Tensorflow for this task here:
https://github.com/aymericdamien/TensorFlow-Examples/blob/master/notebooks/3_NeuralNetworks/recurrent_network.ipynb

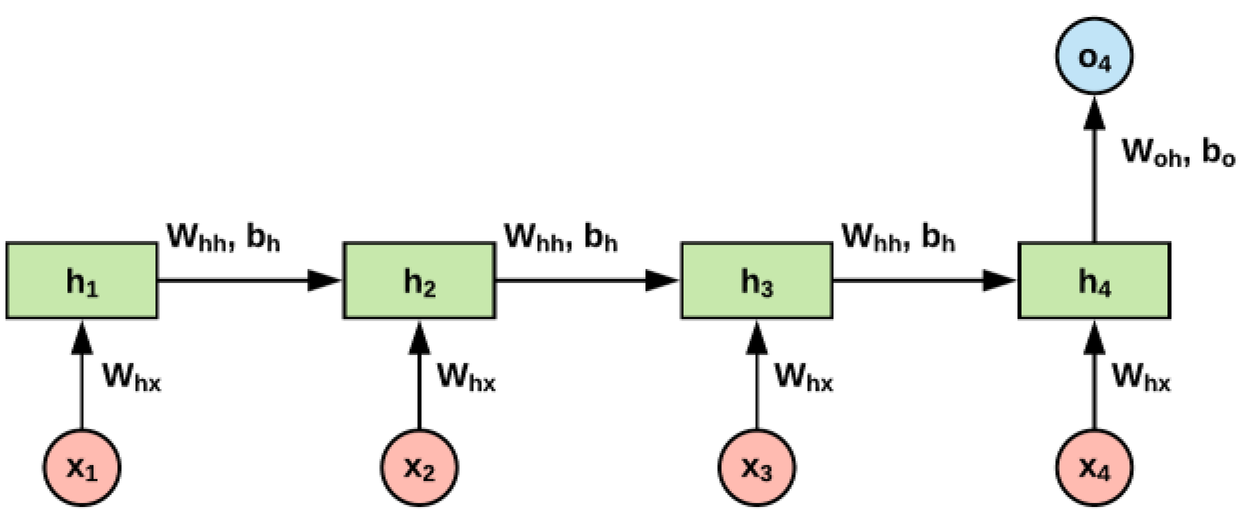
MNIST Dataset

The MNIST dataset contains 60000 examples for training and 10000 examples for testing. The digits have been size-normalized and centered in a fixed-size image (28×28 pixels) with values from 0 to 1.

More information: http://yann.lecun.com/exdb/mnist/

Many to One

For each image the RNN receives a sequence of lenght 28 (28 rows). Each input has 28 features (28 pixels per row). The task is to classify what digit is shown in the image after the whole sequence has been given to the RNN. Therefore, we choose a *many-to-one* input-output relation.



In [1]:

```
%matplotlib notebook
import matplotlib.pyplot as plt
import matplotlib.animation
import matplotlib.pyplot as plt

import torch
from torchvision import datasets, transforms  # This package includes the MNIST dataset
```

We load the training and test datasets and create a DataLoader object, respectively.

In [2]:

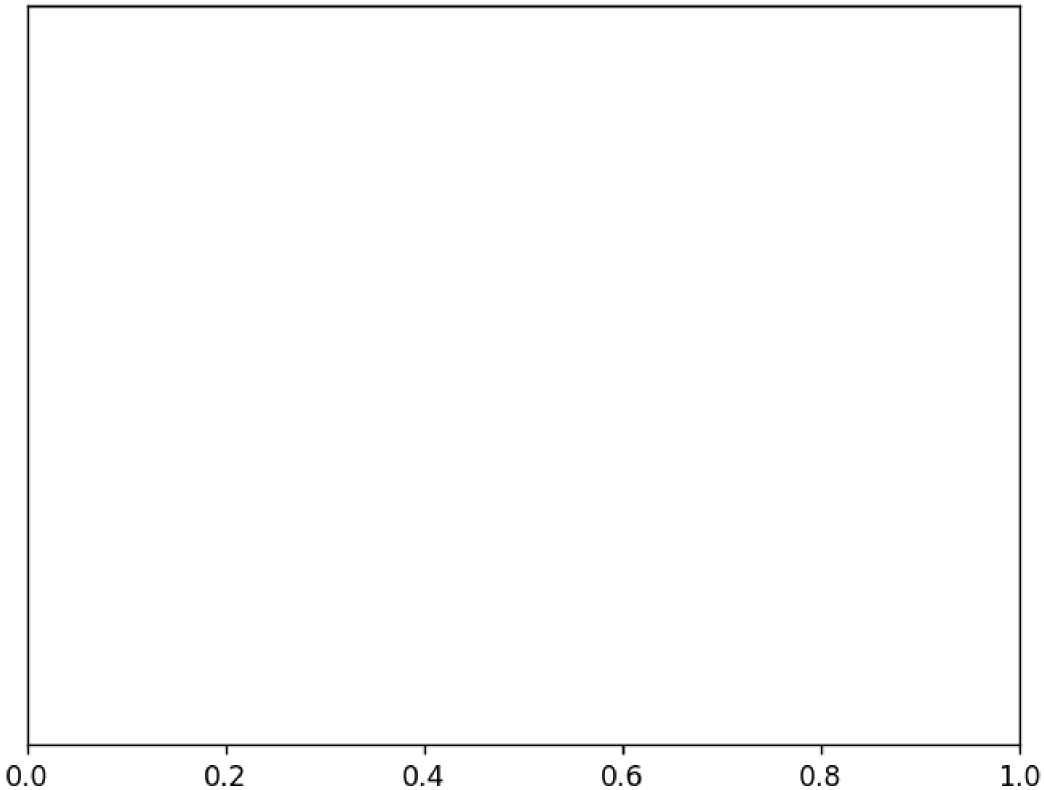
```
mnist_train = datasets.MNIST('./data', train=True, download=True, transform=transforms.ToT
mnist_test = datasets.MNIST('./data', train=False, download=True, transform=transforms.ToT

batch_size = 100
train_loader = torch.utils.data.DataLoader(mnist_train, batch_size=batch_size, shuffle=Tru
test_loader = torch.utils.data.DataLoader(mnist_test, batch_size=batch_size, shuffle=True)
```

To get a better understanding what the RNN 'sees' we take an example image and visualize the sequence of rows.

Which digit is shown here?

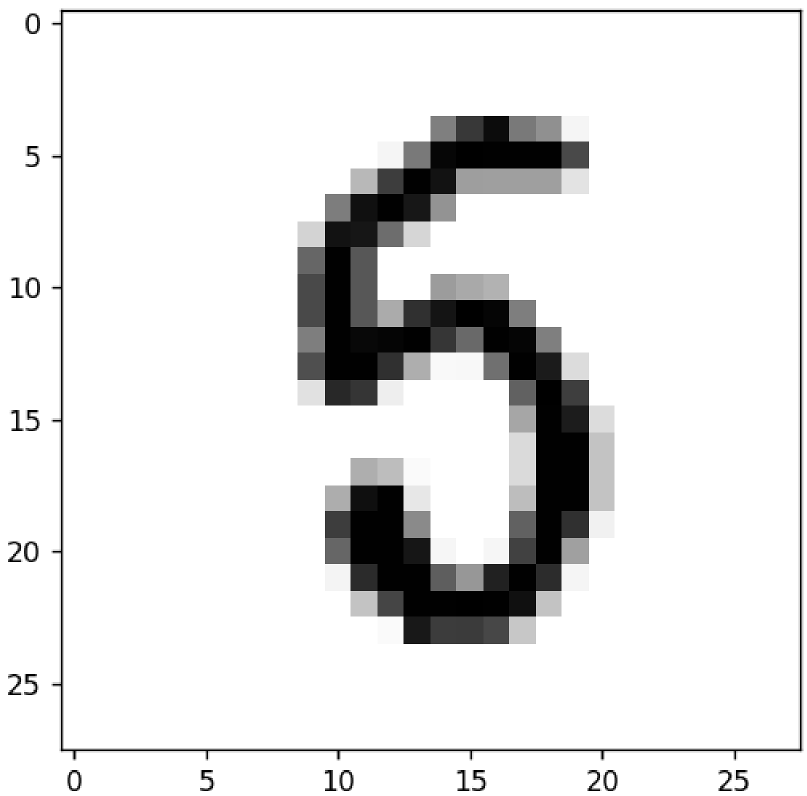
In [3]:

```
fig1 = plt.figure()
ax1 = fig1.gca()
ax1.tick_params(left=False, labelleft=False)

# Meaning of the indices (the three zeros)
# 1. Take the image, not the label
# 2. Random image in the batch
# 3. First channel (there is only one since the MNIST contains only grayscale images)
example_image = next(iter(train_loader))[0][torch.randint(batch_size - 1, (1,)).item(), 0]

def function_for_animation(frame_index):
    ax1.clear()
    image = ax1.imshow(example_image[frame_index:frame_index+1, :], animated=True, cmap='g
    ax1.set_title('t = '+str(frame_index))
    plt.draw()
    return image,

ani = matplotlib.animation.FuncAnimation(fig1, function_for_animation, interval=150, frame
```



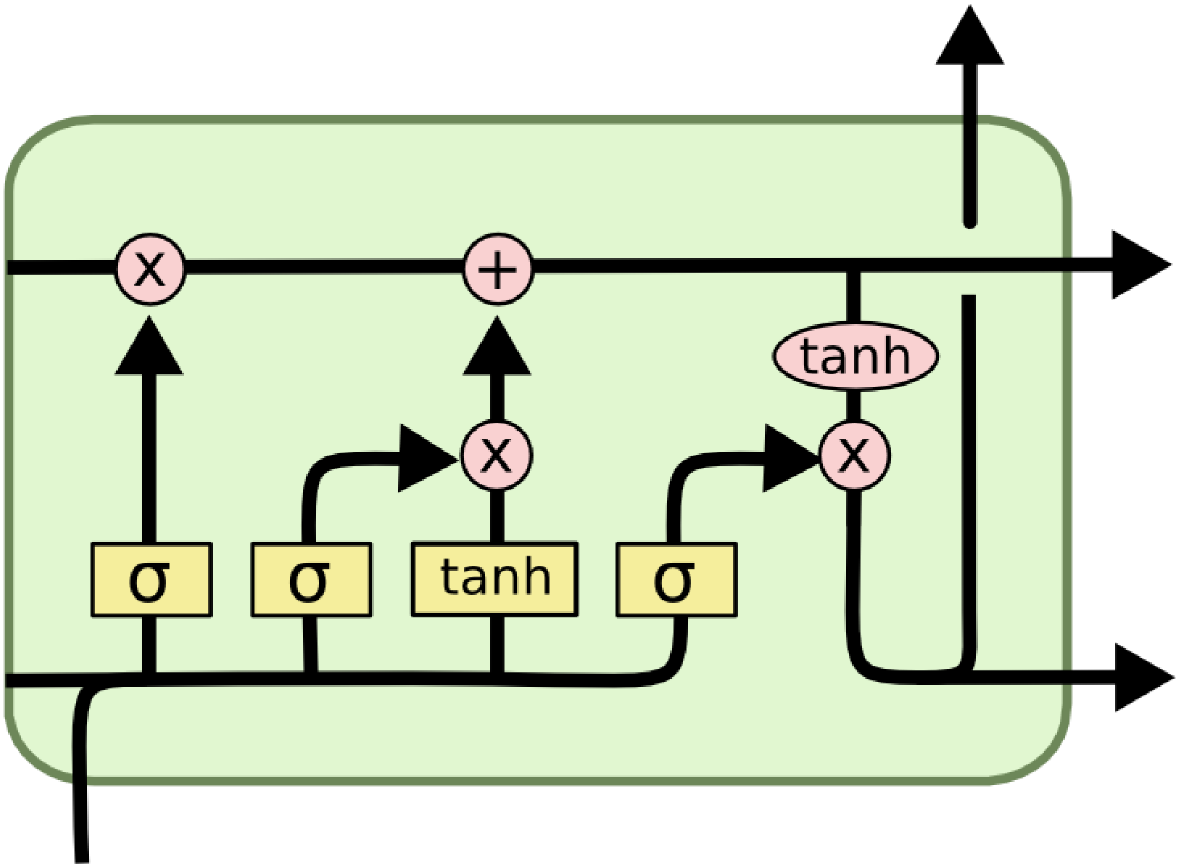
Here is the solution:

In [4]:
```
plt.figure()
plt.imshow(example_image, cmap='gray_r')
plt.show()
```



RNN with LSTM Architecture

We choose a RNN with LSTM cells.



The input and output sizes are fixed but you can try different sizes for the hidden layer. Both, the hidden state h_t and the cell state c_t will have this size!

In [5]:
```
input_size = 28  # Each image: 28x28 -> Sequence of length 28 and input dimension of 28
output_size = 10  # Digits 0-9
hidden_size = 20
```

The architecture of the model is set with the $\_\_init\_\_()$ method. To complete the model we also have to implement the forward pass.

In [6]:
```
class LSTMnist(torch.nn.Module):

    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMnist, self).__init__()

        # LSTM layer
        self.lstm = torch.nn.LSTM(input_size=input_size, hidden_size=hidden_size, batch_fi

        # The linear layer that maps from hidden state space to digits
        self.hidden2digits = torch.nn.Linear(in_features=hidden_size, out_features=output_

    def forward(self, sequence_input):

        # Remove the channel dimension of images
        sequence_squeezed = torch.squeeze(sequence_input, dim=1)
```

```
        # Give the batch with sequences to the LSTM layer
        lstm_out, (h_last, c_last) = self.lstm(sequence_squeezed)

        # Use the last hidden states of the batch to generate non-normalized predictions
        # Meaning of the index 0: Use the last hidden state of the first layer. (Our netwo
        logits = self.hidden2digits(h_last[0])

        # Apply a softmax function to generate probabilites for the classes
        # Apply the logarithm to prepare for the negative log likelihood loss
        digits = torch.nn.functional.log_softmax(logits, dim=1)
        return digits

# Create an instance of the model
model = LSTMnist(input_size=input_size, hidden_size=hidden_size, output_size=output_size)
```

To train the model we have to create an optimizer and choose the model parameters we want to train. We use a gradient descent methode and decide to train all model parameters. Try different learning rates and observe the effect on the training later!

In [7]:

```
learning_rate = 1e-1
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
```

For classification tasks a common loss function is the negative log likelihood. Explaining this loss function is out of the scope of this notebook. But you can find a good introduction here: https://ljvmiranda921.github.io/notebook/2017/08/13/softmax-and-the-negative-log-likelihood/

In [8]:

```
loss_fn = torch.nn.NLLLoss()
```

The following to functions from https://pytorch.org/tutorials/beginner/basics/optimization_tutorial.html perform training and testing loops for one epoch.

In [9]:

```
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

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

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")

def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size
```

```
    print(f"Test Error: \n Accuracy: {(100*correct):>0.1f}%, Avg loss: {test_loss:>8f} \n'
    return 100*correct, test_loss
```

To plot the training progress we store the accuracy and loss for each epoch.

Task: Vary the following parameters and observe how the training speed and reached accuracy change:

- Size of hidden state
- Batch size
- Learning rate
- Epochs

In [10]:

```
epochs = 10
accs, losses = [], []

# Check the initial performance of our model
acc, loss = test_loop(test_loader, model, loss_fn)
accs.append(acc)
losses.append(loss)

# This is the actual training
for t in range(epochs):
    print(f"Epoch {t+1}\n-------------------------------")
    train_loop(train_loader, model, loss_fn, optimizer)
    acc, loss = test_loop(test_loader, model, loss_fn)
    accs.append(acc)
    losses.append(loss)
print("Done!")
```

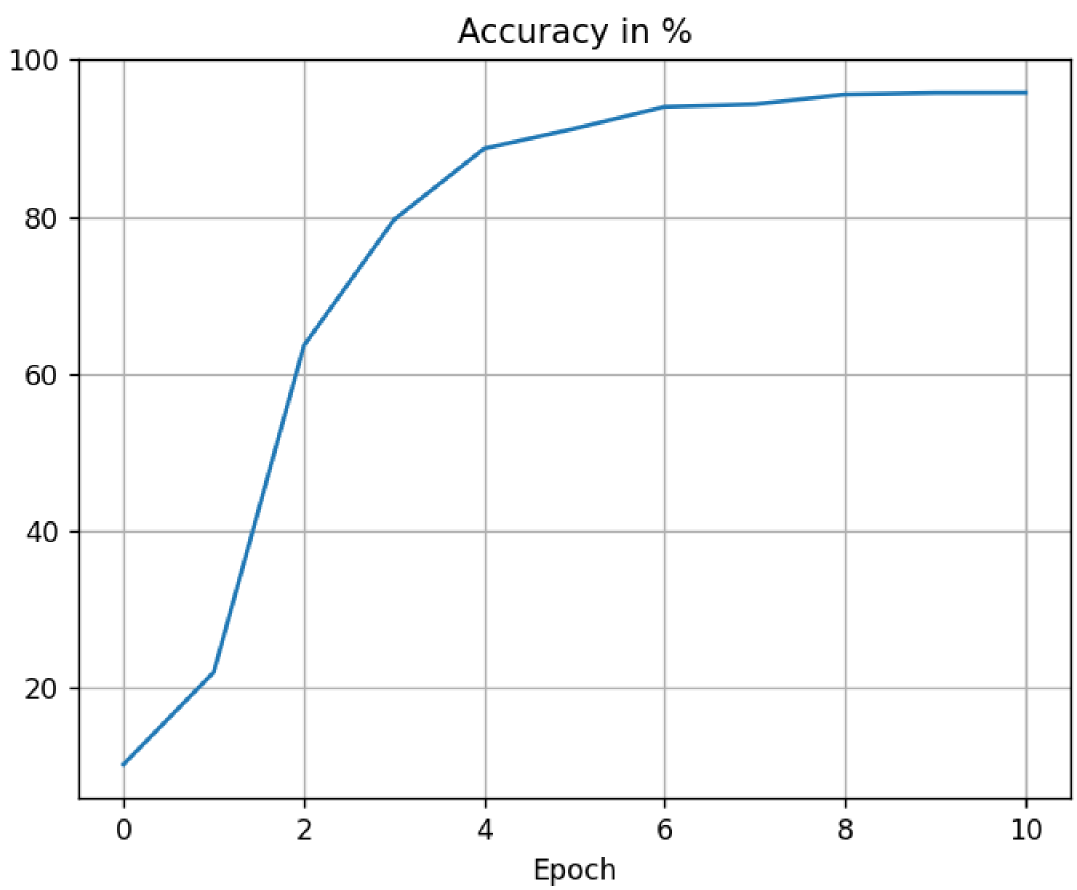
```
Test Error:
 Accuracy: 10.3%, Avg loss: 2.305458

Epoch 1
-------------------------------
loss: 2.308536  [    0/60000]
loss: 2.306280  [10000/60000]
loss: 2.285214  [20000/60000]
loss: 2.292117  [30000/60000]
loss: 2.273114  [40000/60000]
loss: 2.252304  [50000/60000]
Test Error:
 Accuracy: 22.0%, Avg loss: 2.178243

Epoch 2
-------------------------------
loss: 2.212975  [    0/60000]
loss: 1.967329  [10000/60000]
loss: 1.911477  [20000/60000]
loss: 1.620422  [30000/60000]
loss: 1.669103  [40000/60000]
loss: 1.388489  [50000/60000]
Test Error:
 Accuracy: 63.6%, Avg loss: 1.203279

Epoch 3
-------------------------------
loss: 1.261693  [    0/60000]
loss: 1.127985  [10000/60000]
loss: 0.824673  [20000/60000]
loss: 0.697006  [30000/60000]
loss: 0.660924  [40000/60000]
loss: 0.671037  [50000/60000]
Test Error:
```

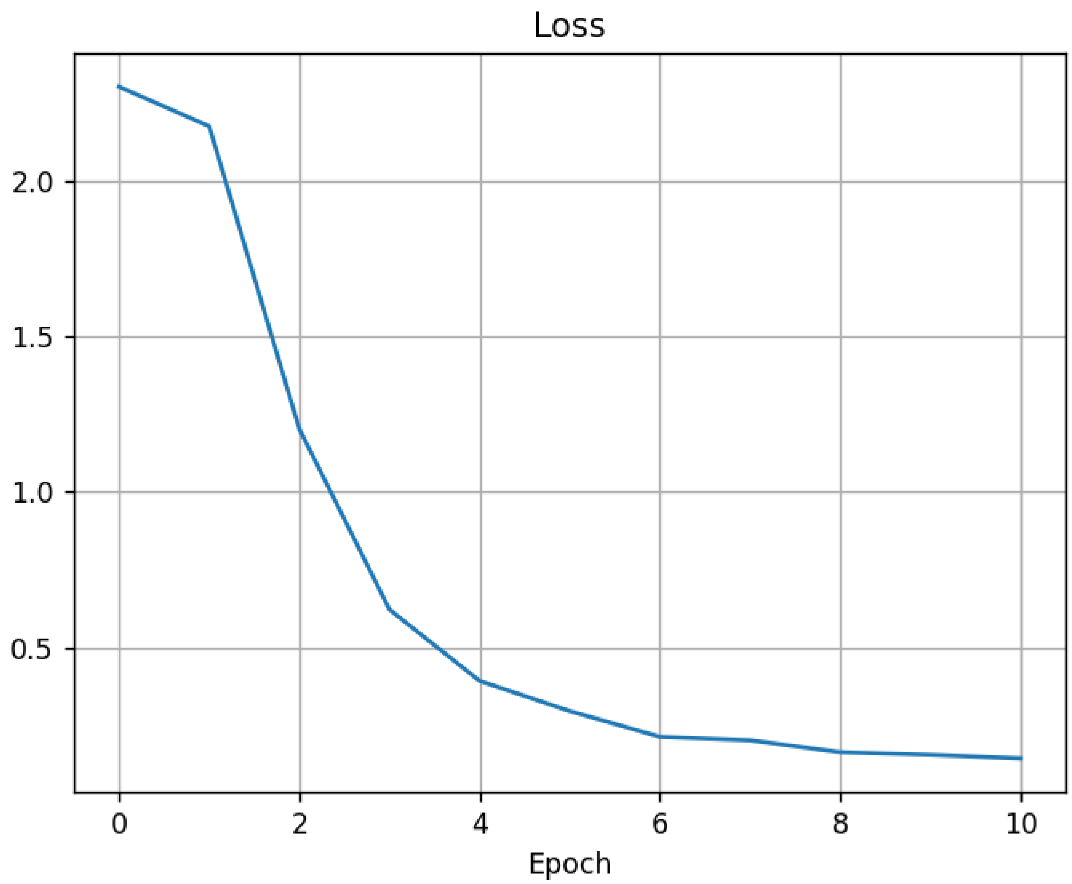
```
 Accuracy: 79.6%, Avg loss: 0.621753

Epoch 4
-------------------------------
loss: 0.654890  [    0/60000]
loss: 0.531381  [10000/60000]
loss: 0.382463  [20000/60000]
loss: 0.397134  [30000/60000]
loss: 0.640287  [40000/60000]
loss: 0.420287  [50000/60000]
Test Error:
 Accuracy: 88.7%, Avg loss: 0.392096

Epoch 5
-------------------------------
loss: 0.256009  [    0/60000]
loss: 0.564381  [10000/60000]
loss: 0.437374  [20000/60000]
loss: 0.308749  [30000/60000]
loss: 0.340025  [40000/60000]
loss: 0.207222  [50000/60000]
Test Error:
 Accuracy: 91.2%, Avg loss: 0.294883

Epoch 6
-------------------------------
loss: 0.285851  [    0/60000]
loss: 0.219533  [10000/60000]
loss: 0.254446  [20000/60000]
loss: 0.212754  [30000/60000]
loss: 0.331613  [40000/60000]
loss: 0.286068  [50000/60000]
Test Error:
 Accuracy: 94.0%, Avg loss: 0.212051

Epoch 7
-------------------------------
loss: 0.363312  [    0/60000]
loss: 0.336478  [10000/60000]
loss: 0.233131  [20000/60000]
loss: 0.459911  [30000/60000]
loss: 0.310702  [40000/60000]
loss: 0.135460  [50000/60000]
Test Error:
 Accuracy: 94.3%, Avg loss: 0.200740

Epoch 8
-------------------------------
loss: 0.291892  [    0/60000]
loss: 0.094340  [10000/60000]
loss: 0.201371  [20000/60000]
loss: 0.243912  [30000/60000]
loss: 0.163135  [40000/60000]
loss: 0.110989  [50000/60000]
Test Error:
 Accuracy: 95.6%, Avg loss: 0.162596

Epoch 9
-------------------------------
loss: 0.169522  [    0/60000]
loss: 0.353933  [10000/60000]
loss: 0.129430  [20000/60000]
loss: 0.126860  [30000/60000]
loss: 0.262140  [40000/60000]
loss: 0.152267  [50000/60000]
Test Error:
```

```
 Accuracy: 95.8%, Avg loss: 0.154179

Epoch 10
-------------------------------
loss: 0.161732  [    0/60000]
loss: 0.143557  [10000/60000]
loss: 0.199714  [20000/60000]
loss: 0.181475  [30000/60000]
loss: 0.201337  [40000/60000]
loss: 0.125356  [50000/60000]
Test Error: 
 Accuracy: 95.8%, Avg loss: 0.142538 

Done!
```

In [11]:

```python
plt.figure()
plt.plot(accs)
plt.title('Accuracy in %')
plt.xlabel('Epoch')
plt.grid()

plt.figure()
plt.plot(losses)
plt.title('Loss')
plt.xlabel('Epoch')
plt.grid()
```





To see the result we load random images from the test dataset and classify them with the trained RNN.

Task:

1. Restart the notebook and play all cells up to the training step. Change the number of epochs to 1 but do not start the training!
2. Run the following cell to see the performance of the model.
3. Train for 1 epoch and go back to step 2. of this task

The performance of the RNN should increase with each epoch.

In [12]:

```
example_loader = torch.utils.data.DataLoader(mnist_test, batch_size=1, shuffle=True)

fig2 = plt.figure()
ax2 = fig2.gca()

def function_for_animation(frame_index):
    ax2.clear()

    example_iter = iter(example_loader)
    example_image = next(example_iter)
    image = example_image[0][0, 0]

    image = ax2.imshow(image, animated=True, cmap='gray_r')

    with torch.no_grad():
        pred = model(example_image[0])
        digit = torch.argmax(pred).item()
    ax2.set_title('digit: '+str(digit))

    plt.draw()
    return image,
```

```
ani = matplotlib.animation.FuncAnimation(fig2, function_for_animation, interval=1500, fran
```