TDT76: Deep Learning

Assignment: Neural music composer

1 Introduction

In this assignment you will be asked to construct neural network models that are capable of learning the dynamics of a given music dataset. You will do so both with and without taking the composer of each musical piece into consideration. Your system should also be able to generate new songs (that is, keep playing after being given a starting-point), and it should be possible to modulate the generative process in the style of a given composer.

This assignment counts for 50% of your score from TDT76. At the day of the exam, you will first demo your system, while answering questions about, e.g., code structure, implementation ideas, and modelling choices. After the demo, you will continue with the more theoretical part of the exam that will decide the final 50% of your score.

This document describes the task you will have to solve, the dataset your solution will work with, and some Python-code we have made available for you to simplify some common tasks. Make sure to read the whole document before starting on your code. Note that we do not discuss theoretical issues in this document. Please confer the teaching material for clarifications if required.

2 Problem description

You will get access to a number of musical pieces (that we for simplicity will term "songs" in the following) composed by a set of classical composers. The songs were originally downloaded from `http://www.piano-midi.de/` in MIDI-format, but have been preprocessed for your benefit and are now available in so-called *piano-roll* format. In this format, see Figure 1, a song is represented by a binary matrix $\boldsymbol{X}$, that we for simplicity now will consider as series of binary vectors, $\{\boldsymbol{x}_1, \ldots, \boldsymbol{x}_T\}$; in your code you can get from the matrix to the vectors by simple slicing. Here, $\boldsymbol{x}_t$ signifies which notes (or rather keys of a piano) that are played at time t. Therefore, $\boldsymbol{x}_t$ is a binary vector of length K, with K equal to the number of piano-keys. Now $\boldsymbol{x}_{t,i}$, element i of that vector, tells us whether or not key i is pressed at time t, with the definition that $\boldsymbol{x}_{t,i} = 1$ if the key is pressed, and 0 otherwise.

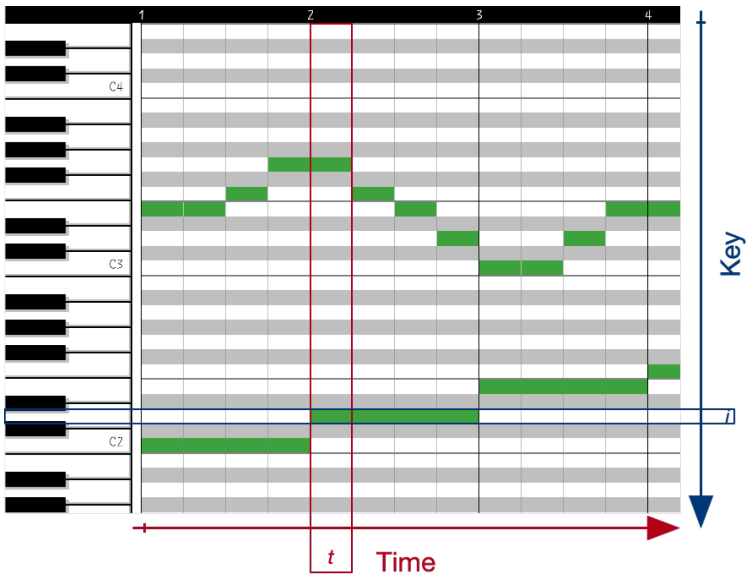


Figure 1: Piano-roll: At a specific point in time t, we have a binary vector $\boldsymbol{x}_t$, where $\boldsymbol{x}_{t,i}$ (which corresponds to element i at time t) is 1 if key i is pressed, and 0 otherwise. In this specific configuration, key i is pressed at time t, hence $\boldsymbol{x}_{t,i} = 1$.

Notice that it is possible to encode music where several keys are pressed at the same time ($\boldsymbol{x}_{t,i} = \boldsymbol{x}_{t,j} = 1$ for some $i \neq j$), and to have a single note running for several time-steps ($\boldsymbol{x}_{t,i} = 1 \; \forall t \in [t_0, t_0 + k]$ with $k > 0$). In total, this means that at each point in time t, the music played is represented by a K-dimensional binary vector, $\boldsymbol{x}_t \in \{0,1\}^K$, and this will serve as the input to your system. Furthermore, each song is of a specific length T. This length will vary between the songs, and you can recognize the end of a piece by the distinct `EndOfFile`-marker, defined so that $\boldsymbol{x}_{T,i} = 1, \; \forall i \in \{0, \ldots, K-1\}$.

Associated with each song is also a set of metadata. In this assignment the metadata is restricted to only the composer of the song. The information is given as a string, and the values are in a limited set (“`bach`”, “`brahms`”, “`debussy`”, . . .).

You will start working with the music-data *without* taking the metadata into consideration in Section 2.1. This will result in a neural composer that we will call the `Generalist` in the following. Afterwards, you will improve the `Generalist` to also take the metadata into account in Section 2.2, resulting in a neural composer we call the `Specialist`.

2.1 A simple neural composer

We will start by disregarding the metadata, and use the full dataset to learn a model that captures the dynamics of the songs. This will result in the the `Generalist`, which is a simple neural composer. You will need to use a recurrent neural network (RNN) for this task; you

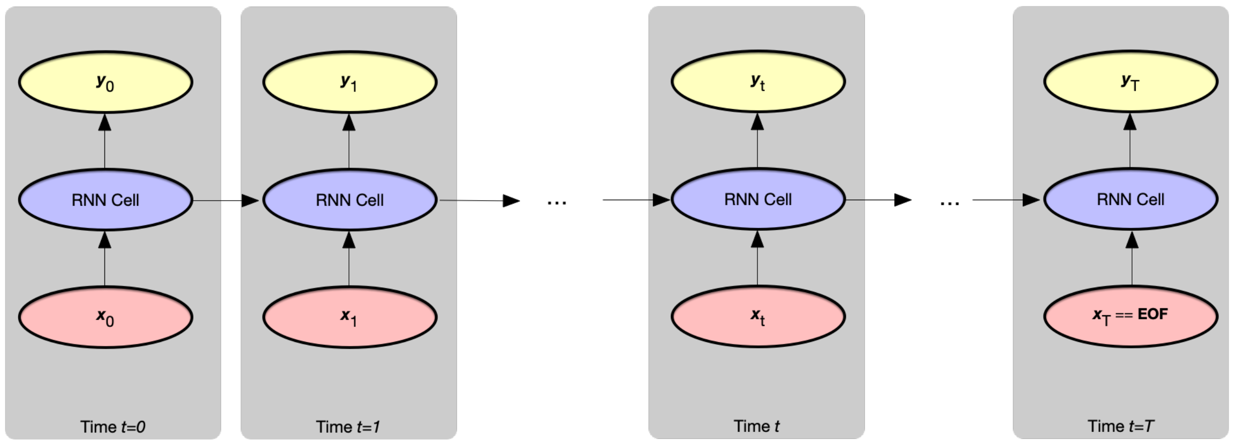


Figure 2: A general RNN structure: Input at time t is denoted $\boldsymbol{x}_t$, generating output $\boldsymbol{y}_t$. The process is terminated as soon as the `EndOfFile`-marker is read as input.

are free to choose any internal model architecture, either a well-known one (LSTM, GRU, ...), or something you construct yourself. Make sure to read the full document before deciding on your internal RNN structure.

Consider the RNN structure in Figure 2. Here we use $\boldsymbol{x}_t$ to denote input to the RNN at time t, and use $\boldsymbol{y}_t$ to signify the output at the same time. As input to the RNN we will use the piano-roll representation of each song, hence the input at any time-point is a binary vector of length K. The output $\boldsymbol{y}_t$ will also be a binary vector of length K, hence the input and output spaces are equal.

2.1.1 Training the neural composer

The main idea for learning the RNN is to teach the system at time t, when given the input $\boldsymbol{x}_t$, to produce an output $\boldsymbol{y}_t$ that is identical to the *next input*, $\boldsymbol{x}_{t+1}$. Thereafter, $\boldsymbol{x}_{t+1}$ is used as input at time $t+1$, and at that time-point we want the output $\boldsymbol{y}_{t+1} = \boldsymbol{x}_{t+2}$, and so on. The process is terminated as soon as the `EndOfFile`-marker is used as input. The training process is depicted in Figure 3.

To build your model, you need to do at least the following tasks:

- Construct the RNN model. You get to decide on internal architecture, size of hidden layer(s), activation functions, etc. You can use any deep learning library that you want, including PyTorch, "pure" Tensorflow, Keras, and so on.
- Feed all songs through the system as outlined above, and learn the parameterization of the model. You get to decide on the loss-function used during learning, the

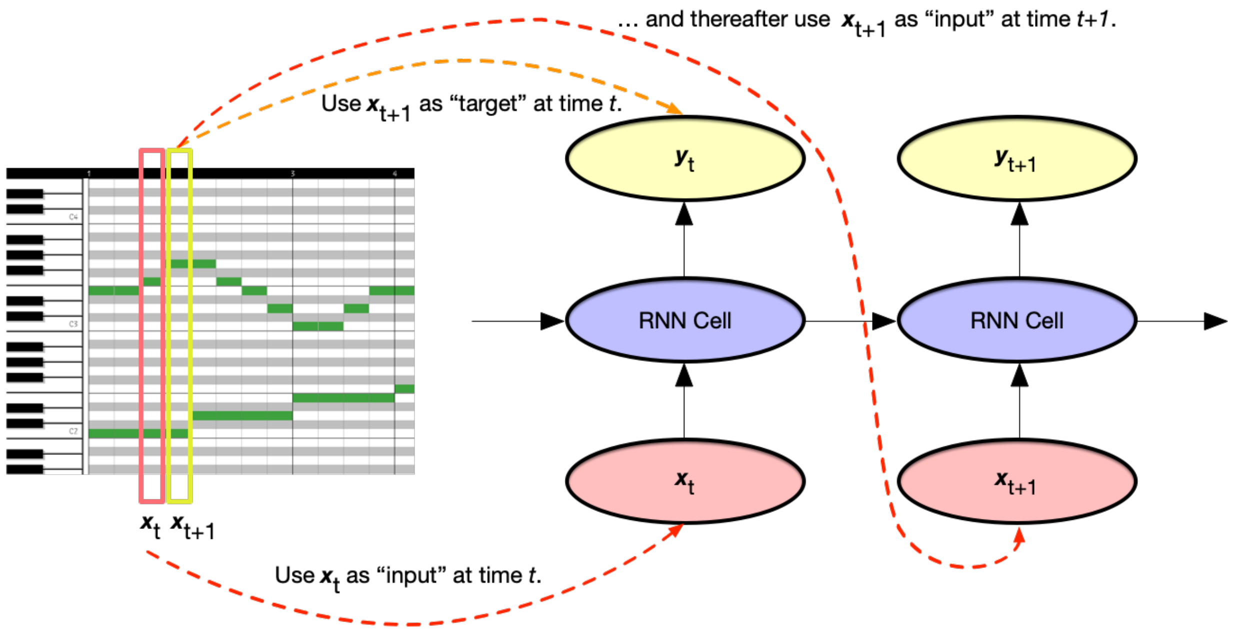

… and thereafter use x_{t+1} as "input" at time $t+1$.
Use x_{t+1} as "target" at time t.
y_t
y_{t+1}
RNN Cell
RNN Cell
x_t
x_{t+1}
x_t x_{t+1}
Use x_t as "input" at time t.

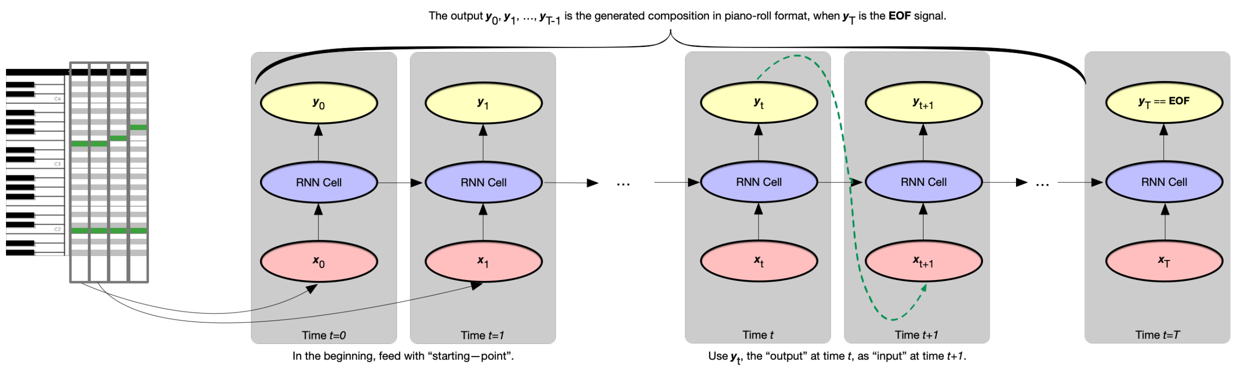
Figure 3: While *training* your RNN, the input fed to the song will be the piano-roll-representations at each point in time. The requested *output* at time t is the piano-roll representation at the next time-point, $\boldsymbol{x}_{t+1}$. Notice the two different roles each vector from the dataset takes: It serves as the desired output at one point in time, then as the input in the next time-slice.

optimization algorithm, the learning-rate scheme, etc.

2.1.2 Generating music

The top-level process for generating music is shown in Figure 4. The system is initialized by a few time-steps of an incomplete piano-roll , where we call the number of time-steps in the initialization τ. For the input of the system, $\boldsymbol{x}_t$, we will use data from the initialization for as long as they are available, i.e., when $t < \tau$. Starting from $t = \tau$, we do not have data to feed into the RNN. The key idea now is to use the generated piano-roll at time t, $\boldsymbol{y}_t$, as input at time $t + 1$, i.e., $\boldsymbol{x}_{t+1} := \boldsymbol{y}_t$. The process is terminated as soon as an `EndOfFile`-marker is generated, that is when $\boldsymbol{y}_T$ is all ones. In this mode we also store the generated outputs until `EndOfFile`, $\{\boldsymbol{y}_0, \boldsymbol{y}_1, \ldots, \boldsymbol{y}_{T-1}\}$. Recall that $\boldsymbol{y}_t \in \{0, 1\}^K$, and we will therefore interpret the sequence of outputs as the generated piece of music.

To use your model to generate music, you need to do at least the following tasks:



The output $y_0, y_1, ..., y_{T-1}$ is the generated composition in piano-roll format, when y_T is the **EOF** signal.
y_0
y_1
y_t
y_{t+1}
y_T == EOF
RNN Cell
...
x_0
x_1
x_t
x_{t+1}
x_T
Time t=0
Time t=1
Time t
Time t+1
Time t=T
In the beginning, feed with "starting—point".
Use y_t, the "output" at time t, as "input" at time t+1.

Figure 4: In *composer-mode*, your system starts with some given initialization, then feeds output at time t as input to time $t+1$. The process ends when the system outputs an `EndOfFile`-marker.

- Feed your trained RNN with an "initialization" that is read from file. Afterwards you need to feed with generated music in terms of the $\boldsymbol{y}_t$-values.
- Recognize when the composed piece is finished; it is done at time $T-1$ if and only if $\boldsymbol{y}_T$ is all ones. If your song goes on for more than 100 time-steps you can terminate even if your system was unable to generate the `EndOfFile`-marker.
- Capture the outputs $\{\boldsymbol{y}_0, \ldots, \boldsymbol{y}_{T-1}\}$, and play the generated music. Note that there is code made available for you to ouput a piano-roll representation through your computer's sound system.
- Refine the model you have created until the generated music is of sufficient quality.

2.2 The specialized neural composer

We will now proceed by helping the RNN to leverage information about the composer of each song. This part of the assignment will further develop the `Generalist`-solution you have made in Section 2.1, hence you should not start on this part before you are satisfied with the results that you obtained there.

The underlying assumption for the `Generalist` is that all the songs in the dataset share a common mathematical structure that can be learned by an RNN-model. The idea we now pursue for the `Specialist` composer is that song structure *may differ between the composers.* For instance, while Bach may be known for the strong logical structure found in many of his works, Mozart is often characterized as more "playful". We want the `Specialist` to be able to learn the peculiarities of each composer during learning, and also to compose music in the style of all the different composers represented in the dataset.

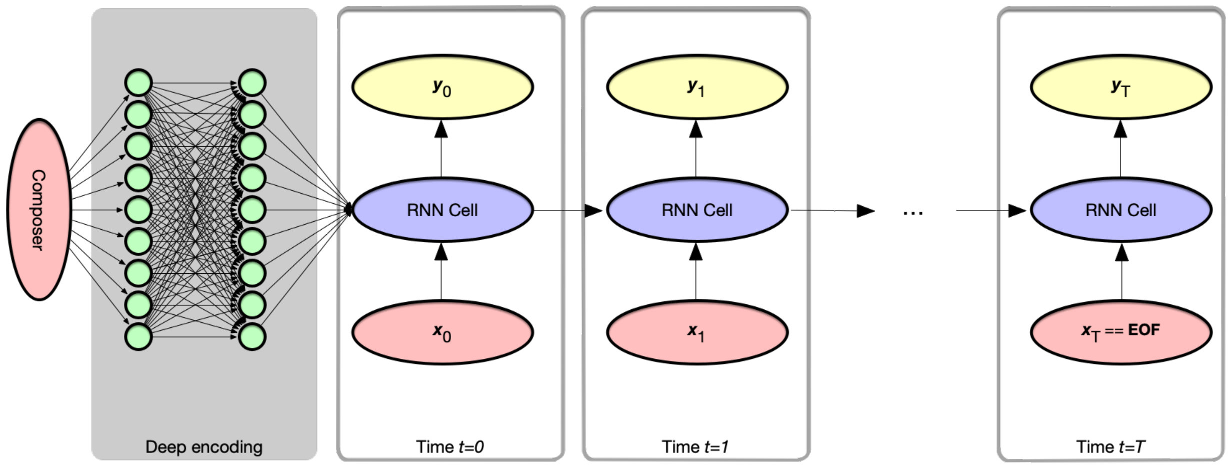


Figure 5: A general RNN structure with context-aware initialization of the first RNN-cell's internal state, compare to Figure 2.

An obvious albeit naïve approach to obtain this would be to build separate models for each composer. If we follow this strategy we select songs by, e.g., Bach, and use only this subset of the data to build the `Generalist`, resulting in a neural composer adhering to Bach's style. While this strategy would produce some result, the approach would disregard much information in the available data (e.g., that pieces composed by Debussy are songs too, and show information about musical compositions that to some extent surely is relevant also when wanting to learn about Bach's musical work). In general terms, the problem we are faced with is that the data from each sub-group (composer) is limited, so we want to leverage information from similar domains (other composers) during learning.

A high-level description of our approach is given in Figure 5. The main difference in this layout compared to Figure 2 is the introduction of a deep encoding of the metadata, which is then used as the hidden state description of the first RNN cell (corresponding to $t = 0$). The other parts of the structure, and the way they are used, are identical to the `Generalist`'s setup. You can choose the representation of the input (the composer) yourself. Furthermore, you are free to choose the architecture of the deep neural network that produces the deep encoding of the composer information (in Figure 5 we have two layers of nine nodes each, but this is not reflected in an actual implementation), and their activation functions. Obviously, the encoding that is generated must be usable as the state of the first RNN-cell, hence must have the correct dimensionality to fit there.

To learn the `Specialist` we will utilize the `Generalist` you have already created (with the parameters you found during the learning) as a starting-point. This gives us a good "initial guess" for the RNN parameters when we want to learn the `Specialist`'s behaviour, because the RNN already represents "typical" musical structure. The initialization happen inside the red box named "Phase 1" in Figure 6, where piano-roll vectors are fed into the system

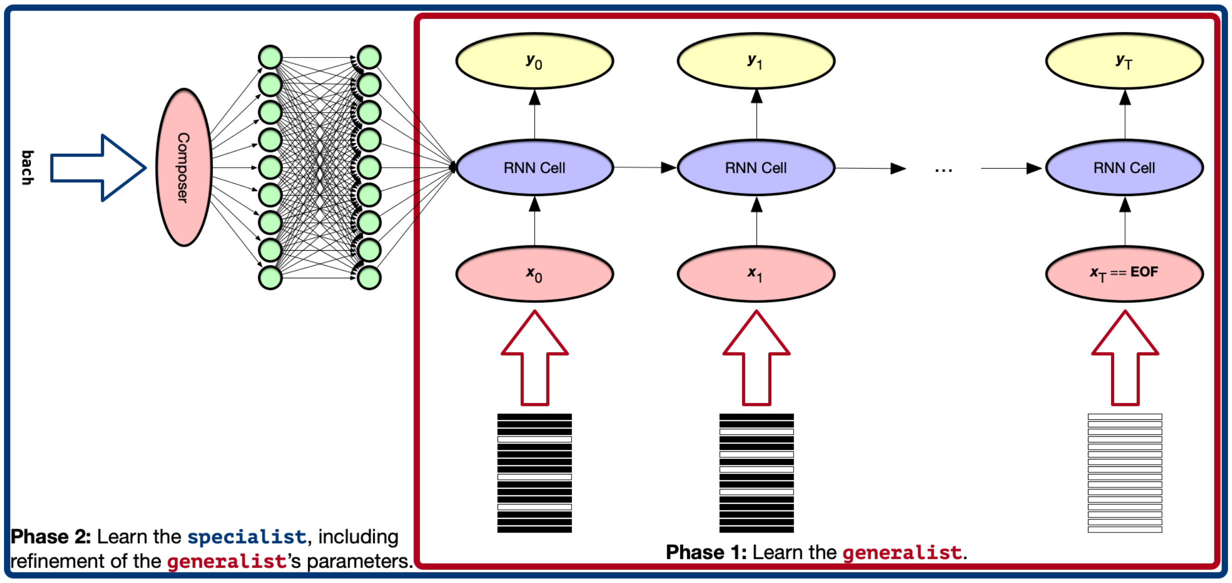

bach
Composer
y_0
y_1
y_T
RNN Cell
RNN Cell
RNN Cell
...
x_0
x_1
x_T == EOF
Phase 2: Learn the specialist, including refinement of the generalist's parameters.
Phase 1: Learn the generalist.

Figure 6: It is beneficial to think of the learning as a process in two phases: First learn a decent RNN model (red box, as done before) without considering the metadata, then refine that representation in light of the information from the metadata (blue box).

as described in the previous section. Now we want to refine the model by taking the composer of each song into account, and move to the full model (blue box termed "Phase 2" in Figure 6). To learn the `Specialist`, we use both the metadata *as well as* the piano-roll data. The learning conducted in this phase will both learn an embedding for the metadata and refine the RNN model. To avoid "forgetting" the information about musical structure encoded in the RNN at the outset of Phase 2, we recommend that you are careful when selecting the learning-rate scheme here.

To build your specialized composer, you need to do at least the following tasks:

- Construct the deep encoding model. You get to decide on internal architecture, size of hidden layer(s), activation functions, regularization, etc. Use the same RNN architecture as you did for the `Generalist`. The full architecture must be defined so that it supports end-to-end learning.
- Feed all songs through the system as outlined above, and learn the parameterization of the model. Make sure to feed in the correct composer for each song.
- Ensure that the `Specialist` is able to compose music in the style of the different composers by generating results and playing them out load.

3 Dataset

The dataset you will use is made available through the course webpage, where each song is represented by a single file. The data originates from “The classical piano midi-page”, `http://www.piano-midi.de/`, but has been converted into piano-roll format.

The data repository has two sub-folders: `training` and `helpers`. The files under `training` are used to train the system. The traing dataset has subfolder with different sampling rate (1 Hz, 2 Hz and 5Hz), each signalized by a respective suffix. To test the system you should use the helper functions to select a initial part of a song and generate a certain output. For example, if your model is in PyTorch (stored in the variable `model`), and you loaded the data using out auxiliary function for the dataset of sampling rate 5 (stored in the variable `data`) you can test your system to generate a song from the first song of the dataset calling `gen_music_seconds(model,data[0],composer=0,fs=5,gen_seconds=20,init_seconds=5)`. This will generate 20 seconds of music using the first 5 seconds of the first song of the dataset. If you are not using PyTorch and just want to generate a snippet with a certain length from your piano roll matrix you can call `test_piano_roll(pianoroll_matrix,n_seconds,fs=5)`.

We advise you to spend some time to get to know the data: Look at plots of the data, generate some statistics (lengths, variability, etc.), play some of the music, and so on. Insights into the available data may help you to come up with a sufficiently (but not overly) flexible model that is both able to learn from the data and to generate new interesting pieces of music.

4 Implementation tricks

4.1 Supporting material

We have supplied a Python-file to help do some common tasks related to the data-formats you will work with. It is called `datapreparation.py`, and is found using a link that will be provided at the course webpage. The file supplies the following functions:

- Converting between MIDI and piano-roll:
  - `piano_roll_to_pretty_midi` takes a piano-roll matrix and converts it into a in instance of the `pretty_midi.PrettyMIDI` class.
  - `midfile_to_piano_roll` reads a MIDI file and creates a piano-roll matrix, that is saved as a CSV file.
  - `piano_roll_to_mid_file` goes the other way around: from piano roll representation and save as a MIDI file.

  - `midfile_to_piano_roll_ins` does the same as `midfile_to_piano_roll`, but selects the sounds from a single instrument only.

- Loading piano-roll files:
  - `load_all_dataset` loads all `.csv` files in a given folder, and returns a list of matrixes.
  - `load_all_dataset_names` looks at all `.csv` files in a given folder, and returns a list of strings extracted from the file-names. The list will signify the **composers** of each song.

- Information about the files:
  - `get_max_length` returns the maximum length of the files (in terms of the number of time-steps the piano-roll consists of).
  - `get_numkeys` returns the number of keys used by each file.

- Inspecting piano-roll files:
  - `visualize_piano_roll`: Make a nice plot showing which key is active at what time.
  - `embed_play_v1` plays a piano-roll representation through your computer's sound system. This function expects you to work through a jupyter notebook or similar. An alternative to play a piano-roll is to use `piano_roll_to_mid_file` first, and then use your favourite MIDI player on the generated file.

We will also supply you with a `requirements.txt` file to help set up your Python environment.

We have also a Python-file called `dataset.py`, to load the dataset into PyTorch tensors. To load the whole dataset you just call `pianoroll_dataset_batch(root_folder)`, with the folder with the csv of the dataset you want to load. After that you can just access an individual song by using common indexing or iteration.

4.2 How to "seed" the hidden state

Usually when seeding the initial state of an RNN we will either get a random or zero initial assignment, which generally means that the this vector is not learned. This is sufficient for the `Generalist`. However, for the `Specialist` we want to be able to learn the initial state of the RNN and back-propagate the gradient through, so that we also can learn the weights for the encoding of the metadata, i.e, the composer of the song.

In the code-snippet below we show how this can be solved in PyTorch. We simply amend the `forward` method of the the `Specialist` class to have a conditional for the hidden layer.

The `forward` method is triggered every time the `Specialist` module is called (e.g., `Specialist (input_sequence, composer_index)`), and is also used during back-propagation.

In the code example we use GRU cells, and each application of the GRU on an input sequence expects two tensors as input: the input sequence itself and an initial hidden state. It will return the output tensor and the final hidden state tensor, where the latter also allows us to propagate the hidden state forward via multiple calls to the GRU if we update the hidden state accordingly. We want to make sure that the the first call to the GRU for a given song queries the metadata-embedding network and assigns the result to the hidden state. For the following calls, we should make sure that the hidden state returned from the *previous* call to the GRU is passed to the function as the new hidden state in *this* call. In the code bellow, we ensure this behaviour by using an optional argument to set the hidden state of the GRU cell. The default value is `None`, and if it is indeed received as `None` by the method we will call the metadata embedding network to get its value. This allows us to embed the metadata information in the first call, and propagate the generated hidden states in the subsequent calls. Overall, the architecture is now end-to-end learnable.

```python
class Specialist(nn.Module):
    def __init__(self, input_size, hidden_size, num_tags, n_layers=2):
        super(Specialist, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.num_tags = num_tags
        self.n_layers = n_layers

        self.tags_embedding = nn.Embedding(num_tags,hidden_size)
        self.notes_encoder = nn.Linear(in_features=input_size,
                                       out_features=hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, n_layers)
        self.notes_decoder = nn.Linear(in_features=hidden_size,
                                       out_features=hidden_size)
        self.output= nn.Sigmoid()

    def forward(self, input_sequence, tag, hidden=None):
        if(hidden is None):
            hidden = self.tags_embedding(tag)
            hidden = torch.cat( (hidden for i in range(self.n_layers)))

        output, hidden = self.gru(self.notes_encoder(input_sequence),hidden)
        output = self.output(self.notes_decoder(output))
        return output,hidden

```

The above code implements a “bare-bones” solution that should work, but not necessarily work well. Please do not just copy it off directly, but first look at how it works, then consider ways to make improvements!

5 Delivery and demonstration

The Administration at IDI/your department will give you a time and place to show up for the oral exam. Make sure to be there early. Before your time-slot starts you must send us your source-code, a list of requirements (external libraries that we need to install to get your code to run), and – if relevant – a list of sources you have utilized to come up with your implementation. Send the email to both `helge.langseth@ntnu.no` and `eliezer.souza.silva@ntnu.no`. The work described in this documented is designed for a single student working alone, and is to be seen in light of the regulations for exams at NTNU. If there is a suspicion of copying, we will use plagiarism-checking on the submitted source code, and if a large overlap exists between your code and the code of some other student(s), you will fail the course. The same punishment is given to both the source and the copier, so make sure to not share details of your code with anyone.

When the oral exam starts you will be asked to demonstrate your code on your own computer, **so do bring one!** You may be asked to demo both the `Generalist` and the `Specialist`. While we do not expect your system to compose pieces of music that would make Mozart envious, we expect some musical structure to your generated output, so make sure that your system provides results at a level that can be seen as a proof of concept. After the demo we will discuss, e.g., modelling choices and implementation ideas. We will talk about the decisions you have made, and may ask questions about, for instance, the effect of making some modelling choice compared to alternatives. This part of the exam is to gauge your "hands-on ability" when it comes to building deep learning systems. Credit is given for (sufficiently) good results, modelling and implementation choices, and we are looking for your practical understanding regarding what your choices enatil for this particular dataset. The more theoretical insights you have gathered throughout the course will be evaluated in a second session following after the demonstration.