Sifakis Emmanouil, csdp1288 (28/04/2024)

Problem 1. *(Conditional generation of names).*
In this exercise, you will extend the AR model presented in Tutorial 5 to the case of conditional generation of names. For each country, you will assign a one-hot encoding and pass it as an extra argument -implemented via vector concatenation- to the RNN. Write a program that takes as input a country name and returns a list of 10 names. Repeat the same for the LSTM and GRU architectures as well as for the Transformer architecture (from Tutorial 6).

Solution

→ RNN ←

Firstly, the description of the implementation, training, and generation process for the RNN network will be provided. This model is meticulously crafted to enable effective sequence generation. Inherited from 'nn.Module', it's initialized with input size, hidden size, and output size, allowing flexibility in adjusting capacity for diverse data. Key linear layers ('i2h', 'i2o', 'o2o') handle input-to-hidden, input-to-output, and hidden-to-output transformations. A dropout layer (p=0.1) mitigates overfitting by randomly dropping connections. Subsequently, softmax ensures valid probability distributions over vocabulary characters. During the forward pass, inputs, hidden state, and category are concatenated and passed through linear layers. Hidden state is updated via 'i2h', and output computed by concatenating hidden state and 'i2o' output. Initialization of the hidden state as zeros in 'initHidden' ensures consistency across training examples, enhancing training stability.

During the training in each iteration, a random training example consisting of a category, input line, and target line is selected. This randomness helps the model learn from diverse examples and generalize better to unseen data.For each character in the input line, the negative log likelihood loss ('NLLLoss') is computed. This loss function is particularly suited for sequence generation tasks as it penalizes deviations from the target sequence. Gradients are computed based on the computed loss and subsequently backpropagated through the network. This process enables the model to understand the impact of its parameters on the loss and adjust them accordingly. Finally, model parameters are updated using gradient descent, nudging them towards values that minimize the loss and improve the model's performance. That way the RNN model undergoes a process of continuous refinement, gradually improving its ability to generate coherent and contextually relevant sequences

Regarding the generation of names conditioned to the origin country the RNN generation process unfolds as described below. The core of name generation lies in the 'sample' function, which is responsible for generating a single name given a starting letter and a Country category. This function starts by initializing the hidden state, for coherence in the generated sequence. Subsequently, it enters a loop where it iteratively predicts the next character based on the current hidden state and the previously generated characters. This iterative process continues until either the maximum length for a name is reached or an end-of-sequence token is predicted, indicating the completion of the name. Throughout this process, predictions are made using the trained RNN model, which outputs probabilities for each character in the vocabulary. The next character is then selected based on these probabilities, with higher

probabilities favoring more likely characters. Once the name generation is complete, the generated name is returned as the output, ready for further processing or display.

The resulting loss from training the RNN network with a learning rate of 5e-4, 256 layers, and 1e5 iterations conditioned to countries is presented below. The model seems to work well and is also able to learn the relationships between the letters of each name for each country. The loss seems to converge around 2.2.

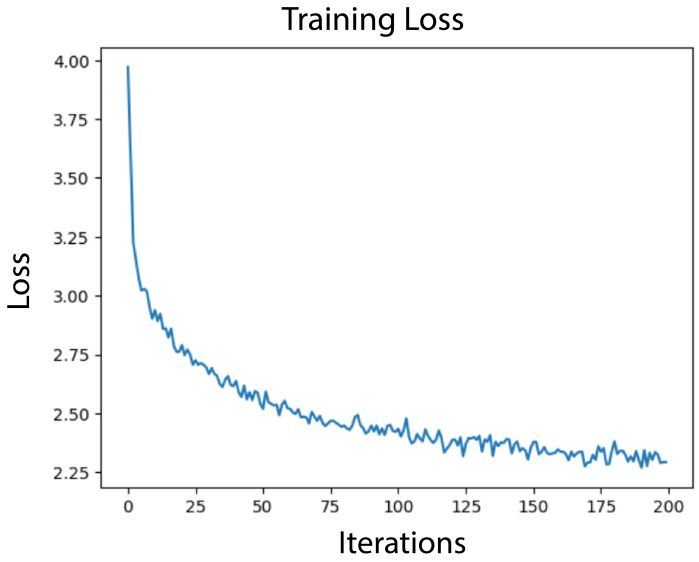

Figure 1: Training Loss plot for RNN Network

For Greece and Russia, generated names starting from letters A,B,C,D,E,F,G,K,M,N (10 names) are as follows:

Surnames starting from
ABCDEFGKMN

GREEK:	RUSSIAN:
Allis	Allankovsky
Ballos	Balloskiv
Caris	Chantov
Dostilos	Dandellov
Eallas	Eallovev
Fallas	Fariski
Garinaki	Gariskiv
Kalis	Kallov
Mallos	Malinov
Naskilos	Nakinov

Figure 2: Generation (RNN) of different names conditioned on Greek and Russian names

The countries of origin selected for Greece and Russia are chosen because their names are more distinct, allowing us to evaluate the model's performance effectively. (The additional generated names for different countries are provided in the notebook.) Overall, the model appears to make reasonable predictions, generating surnames that resemble real names in both

languages. In the case of Greek surnames exhibit patterns consistent with Greek phonetics and naming conventions described in the dataset as well, as well as for Russian surnames. However, it's noted that the model's performance could potentially benefit from further refinement, such as increasing the number of layers with now important improvements. One possible reason for this observation could be the complexity of the underlying patterns in the data. Additionally, other factors such as the quality and diversity of the training data, the choice of hyperparameters, and the specific characteristics of the task may also influence the model's performance. The detailed implementation of this RNN model is presented in the code implementation notebook : **CS673_ex1.ipynb**

→ **LSTM** ←

The LSTM network implementation for the name generation task closely resembles that of the RNN, with adjustments to accommodate the LSTM cell's architecture. First, it combines the category of the name with the input letter and then, it passes this combined information through the LSTM layer. After that, it uses a linear layer and softmax to get the final output, which is the predicted next letter. The model also initializes its hidden state to start the process. By considering both the category and the letters, the model can generate names that match specific categories while capturing patterns in the data over time.

Other than those details the training loop and the generation of the names is the same with the RNN implementation. The detailed implementation of this LSTM model is presented in the code implementation notebook : **CS673_ex1.ipynb** Below is the loss plot over the iterations of the LSTM model. Although the general plot may not have the typical smooth curve but it reaches a lower loss value compared to the RNN and converges around 1.7. The learning rate in this case is 5e-3 with 64 layers and 1e5 iterations.

Figure 3: Training Loss plot for LSTM Network

In this case, name generation demonstrates promising results. The generated names capture the distinctive characteristics of names from specific countries. Below, I present

the generated names for Greek and Russian categories as before. (The additional generated names for different countries are provided in the notebook.)

Figure 4: Generation (LSTM) of different names conditioned on Greek and Russian names

I increased the learning rate and reduced the number of layers in the RNN configuration because the initial setup didn't yield the expected results. Initially, although the training progressed as anticipated, the generated names remained largely unchanged except for the first letter (some kind of overfitting). However, with a higher learning rate and fewer layers, the generation process appeared to improve significantly.

→ GRU ←

Incorporating the GRU architecture introduces a notable departure from LSTM, offering a more simplified recurrent unit structure. While LSTM employs separate memory cells and input/output gates, GRU combines the forget and input gates into a single update gate, potentially making it more efficient in handling long-term dependencies in sequential data. This architectural shift may influence the model's capacity to capture intricate patterns and relationships during training. However, despite these differences, the fundamental processes of training and generation remain consistent with previous implementations using LSTM and RNN. The plot depicts the training loss over iterations for a GRU model with 64 layers and a learning rate of 5e-3 across 1e5 iterations. Similar to the previous LSTM implementation, the plot shows a somewhat jagged curve, but the loss converges around 1.6, which is the best out of the three models.

As with the LSTM implementation, adjustments were made to the layer size and learning rate in the GRU model to improve name generation. This fine-tuning helped mitigate potential overfitting issues and enhanced the model's performance in generating names.

In the table above, the generation results for the GRU model (Greek and Russian) are presented. While the model effectively captures dependencies and relationships between letters indicative of the country's conditions, it appears to exhibit signs of overfitting to specific categories. This is evident in the Russian category, where the endings of the generated names remain consistent. This suggests that the model may have become too specialized or biased towards certain patterns within the training data. The detailed implementation of this GRU

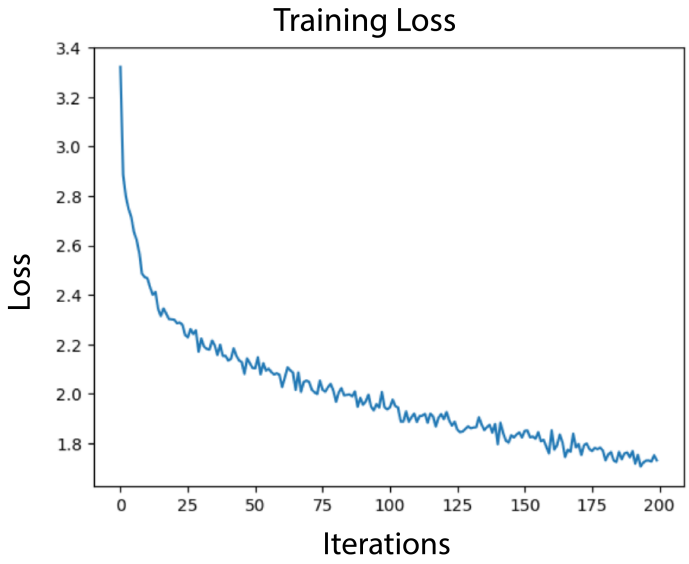

Figure 5: Training Loss plot for GRU Network

Surnames starting from
ABCDEFGKMN

GREEK:	RUSSIAN:
Alaminis	Adamin
Baris	Barinov
Chris	Charov
Demakis	Duminov
Essinis	Eshenkov
Feris	Farinov
Garrakos	Garinov
Kalos	Kalovov
Marrakos	Marimov
Namis	Nakherkov

Figure 6: Generation (GRU) of different names conditioned on Greek and Russian names

model is presented in the code implementation notebook : **CS673_ex1.ipynb**

→ **Transformer** ←

Due to time constraints, I was unable to submit the transformer implementation.

Problem 2. *(From numbers to words).*
(a) *Using num2word function num2words create a dataset of numeral - sentence pairs. For instance, a sample is (1033, "one thousand thirty three"). Use numbers up to 109 while the size of the dataset will be 105.*
(b) *Implement and then train an GRU that converts a number to its sentence analog. Plot the training loss as a function of iterations. Report the accuracy on unseen numbers. Report also what is the result when a number outside the training range is given (e.g, 101001001001 or 1001001001001).*
(c) *Repeat (b) but now using the Transformer architecture.*

Solution

→ **GRU** ←

The GRU (Gated Recurrent Unit) model is a recurrent neural network (RNN) type particularly suitable for sequence-to-sequence tasks (Tutorial 5) such as converting numbers to words. It consists of several components that collectively enable it to process input sequences and generate corresponding word representations effectively. The beginning input to the GRU model comprises sequences of numbers represented as word indices. Before being passed to the GRU layer, these indices undergo transformation into dense vectors using an embedding layer. This layer plays a crucial role in converting each word index into a dense representation within a continuous vector space. In the implementation, the embedding layer is defined with a specific embedding dimension, which is set to 256. Following the embedding process, the sequences are fed into the GRU layer. This layer processes the input sequentially, updating its hidden state at each time step.

The GRU architecture features recurrent units with gating mechanisms, enabling it to capture long-range dependencies within input sequences while addressing problems like the vanishing gradient issue. In the implementation, the GRU layer is defined with a specific hidden dimension (256), determining the size of the hidden state of the GRU cells. Subsequently, the output of the GRU layer at each time step is passed through a fully connected (linear) layer. This layer serves to map the hidden representations learned by the GRU cells to the vocabulary space. Its output size matches the size of the vocabulary, as it predicts the probability distribution across all possible words in the vocabulary. In the implementation, the fully connected layer is configured to output logits for each word in the vocabulary.

During training, the GRU model learns to map input sequences (numbers represented as word indices) to their corresponding word representations. Firstly, the model utilizes the Cross Entropy Loss as the loss function, which measures the dissimilarity between the predicted word distribution and the actual word indices in the training data. This loss function encourages the model to assign high probabilities to correct words and low probabilities to incorrect words. For optimization, the Adam optimizer is employed to update the parameters of the GRU model during training. The training loop involves iterating over the training data in batches using a DataLoader. For each batch, the model computes a forward pass to obtain the predicted word distributions. The loss between the predicted distributions and the actual word indices in the batch is then calculated. Backpropagation is performed to compute the gradients of the loss with respect to the model parameters, followed by updating the model parameters using the optimizer's update rule. To monitor

the training progress and assess model performance, the training loss is plotted over epochs. This visualization aids in understanding how the training loss decreases over time, indicating the model's learning progress, and detecting any anomalies or issues during training, such as overfitting. Below the plot 2 of the loss for the 10 epochs is presented along with the losses on each epoch.

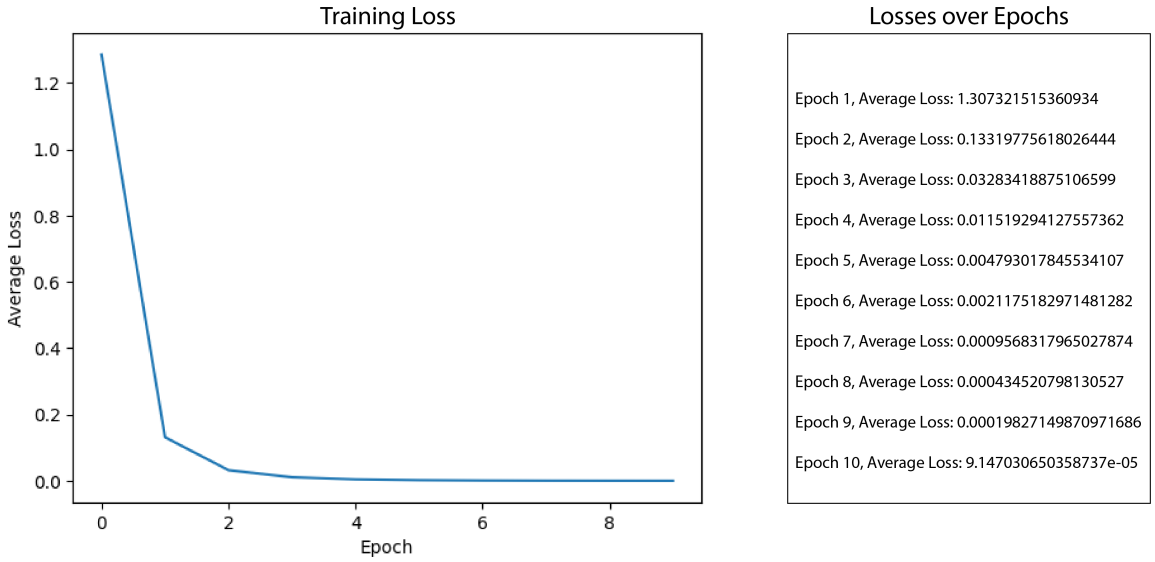

Figure 7: Training Loss plot and Loss over Epochs for GRU Network

The above GRU model's training progress over 10 epochs demonstrates a significant reduction in average loss, indicating effective learning. Starting from an average loss of 1.307, the model swiftly converges, with subsequent epochs showing a sharp decline in loss values. By the final epoch, the average loss drops to a lower value of approximately 9.147e-05. This rapid decrease in loss signifies the model's ability to accurately predict word distributions for input numbers, showcasing its effectiveness in learning and improving over time. The accuracy for generating sentences representing the input words that are withing the training range is: 99.7%

(The code implementation of this section of the exercise is in the **CS673_ex2.ipynb** notebook)

During inference post-training, the model creates word representations for numbers absent in the training data. Tokenizing the input number and converting it into word indices, the model passes it through for representation. For words not in the vocabulary, a substitution strategy is applied. Using the 'difflib' library, the model finds the most similar word in the vocabulary. Absent words are then replaced with their closest matches. For example, "billion" may be substituted with "million."

While this may lead to incorrect estimations, it maintains correct word relations. For instance, if "billion" is replaced with "million," "one billion" could become "one million." Though the absolute value may be inaccurate, word relationships remain consistent. This approach ensures coherent representations for out-of-range numbers, leveraging semantic

similarity for reasonable estimations, preserving sentence structure and coherence. For this reason the accuracy even for the absent number-words remain high : 98.3%

→ **Transformer** ←

In this implementation, the Transformer model is utilized as an alternative to the previously discussed GRU architecture. The input sequences, representing numbers as word indices, are embedded into dense vectors using a layer configured with a specific embedding dimension, consistent with the GRU model.

In contrast to recurrent layers, the Transformer model comprises multiple Transformer encoder layers. Each encoder layer operates independently, processing the input sequence using self-attention mechanisms and feed-forward neural networks. The incorporation of self-attention allows the model to effectively weigh the importance of different words in the input sequence, facilitating the capture of long-range dependencies. Additionally, each encoder layer incorporates residual connections and layer normalization to ensure stable training. Stacking multiple Transformer encoder layers enables the model to capture intricate patterns within the input sequences. After processing through the Transformer encoder layers, the output passes through a fully connected layer, mapping learned representations to the vocabulary space and predicting the probability distribution over all words, mirroring the GRU model.

During the training phase, the Transformer model undergoes a similar training process to the GRU model, with some distinctions. The Cross Entropy Loss function is utilized as the loss function, while the Adam optimizer is employed to update the parameters of the model during the training process. Although the training loop entails iterating over the training data in batches, the forward pass through the Transformer model differs from the GRU model, as it involves multiple Transformer encoder layers instead of recurrent layers. After obtaining the predicted word distributions from the output layer, the loss is calculated, and the model parameters are updated through backpropagation and the optimizer. The training losses for the Transformer model exhibit a significant decrease over 10 epochs as it is shown below 3.

The Transformer model achieves slightly lower accuracies than the GRU model for both numbers within and outside the training range, with accuracies of approximately 98.76% and 96.9%, respectively, compared to 99.7% and 98.3% for the GRU model. This difference in performance may be attributed to the complexities of the Transformer's architecture, which relies on self-attention mechanisms, compared to the simpler recurrent connections of the GRU model. Despite these variations, both models demonstrate strong capabilities in converting numbers to words, suggesting that the choice between them depends on factors such as task requirements and computational resources. (the above loss values may not e exactly the same in relation to the notebook since I run it locally for the final screenshot)

(The code implementation of this section of the exercise is in the **CS673_ex2.ipynb** notebook)

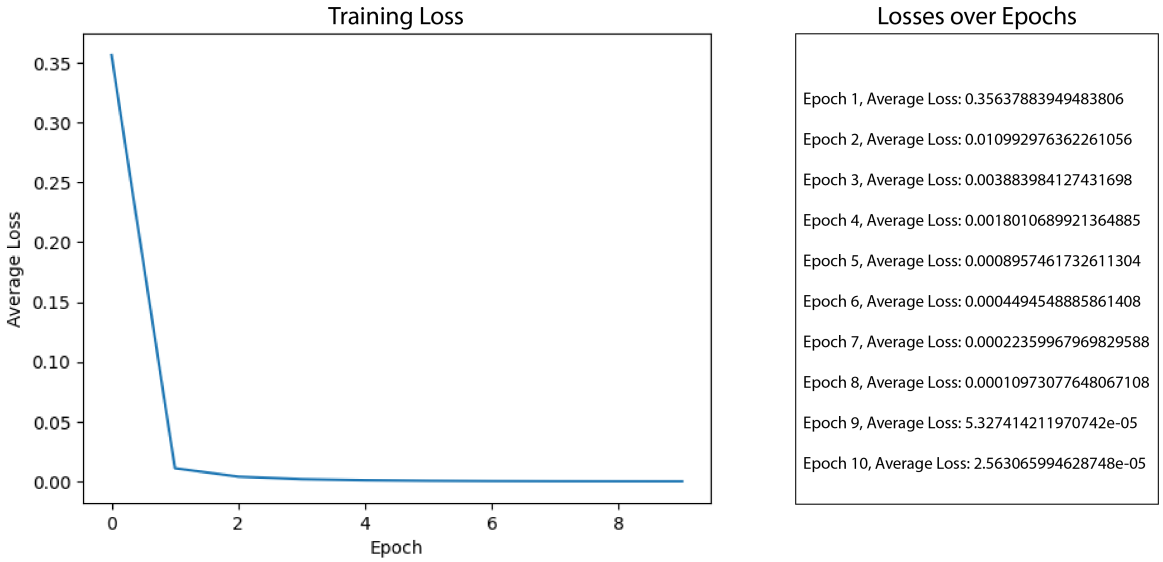

Figure 8: Training Loss plot and Loss over Epochs for Transformer Network

Problem 3. *(Music Transformer).*
Go to MusicTransformer-Pytorch and download the Music Transformer. Train on MAESTRO dataset and then generate new melodies for various instruments (eg, piano, base, etc.)

Solution

The provided notebook loads MIDI files from the dataset and preprocesses them into a format suitable for training the Music Transformer model. This involves converting MIDI files into a numerical representation. The encoded MIDI sequences are then saved as pickle files for efficient loading during training. The provided code utilizes the MAESTRO dataset v2 for training the Music Transformer model. However, one can also use a custom dataset, such as the Groove Drum Dataset that I have chosen, by following specific instructions provided within the notebook. The notebook includes dedicated cells that guide users through the necessary changes required to import and preprocess both the Maestro and custom datasets.

The training loop iterates over the dataset for a specified number of epochs, dividing it into batches using PyTorch's DataLoader class. For each batch, the model performs forward propagation to compute predicted outputs, followed by calculating the loss using cross-entropy. Backpropagation is then executed to update the model's parameters (weights and biases). Adam optimization, a variant of SGD, is applied to minimize the loss by adjusting parameter learning rates based on gradient moments.

The implementation of the MusicTransformer in PyTorch is based on the generic Transformer architecture introduced in PyTorch 1.2.0 (1.11.0 in colab). It utilizes an encoder-decoder architecture, where only the decoder is employed for music generation. Positional encoding is added to input embeddings to convey sequence position information. The self-attention mechanism enables the model to capture global dependencies efficiently. Feedforward neural networks within transformer blocks perform nonlinear transformations on input

data. Multi-head attention allows the model to focus on different parts of the input sequence simultaneously. Layer normalization and residual connections stabilize training and facilitate gradient flow. The output is generated autoregressively using the decoder of the Transformer architecture. At each time step, the model predicts the next token in the output sequence based on the tokens generated so far. This process continues iteratively until a predefined maximum sequence length is reached or a special end-of-sequence token is generated. The output is generated probabilistically, with the model producing a probability distribution over the vocabulary of musical tokens at each time step. During inference, sampling techniques such as greedy decoding or temperature-based sampling can be employed to select the next token based on the probability distribution. This allows the model to generate diverse and musically coherent sequences while maintaining control over the level of randomness in the generated output.

In the provided notebook, epochs and batch size are crucial parameters tailored to the limitations of the Google Colab environment. The model in my case is trained for 10 epochs, a significant reduction from the recommended 300 epochs, due to time and resource constraints. This compromise allows for a shorter training duration within the available resources. Additionally, a batch size of 3 , instead of the advised 2, is chosen potentially slowing down training progress. Despite these limitations, the model can still learn to some extent, although it may not achieve the same level of performance as with longer training durations and larger batch sizes. This can be confirmed be the final achieved accuracy of my train in relation to the original one as seen below. The original notebook achieves a 0.1968 eval loss and 0.420 eval accuracy

Evaluation Metrics on MAESTRO Dataset

- **Avg eval loss:** 2.6577585379282636
- **Avg eval acc:** 0.29212039791875416

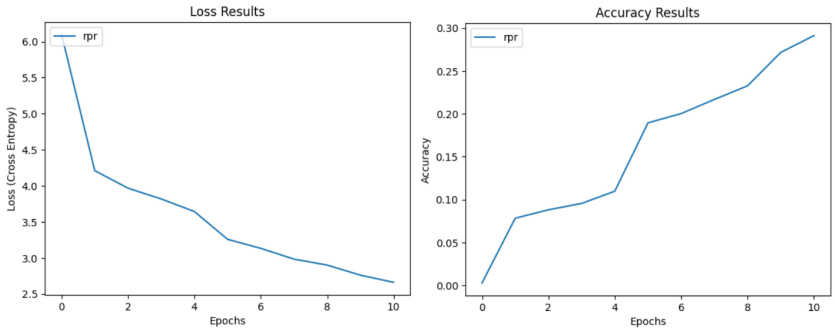

Figure 9: Training Loss and Accuracy plot over Epochs for Maestro Dataset

Evaluation Metrics on DRUM Dataset

- **Avg eval loss:** 1.8977752923965454
- **Avg eval acc:** 0.46842953426445405

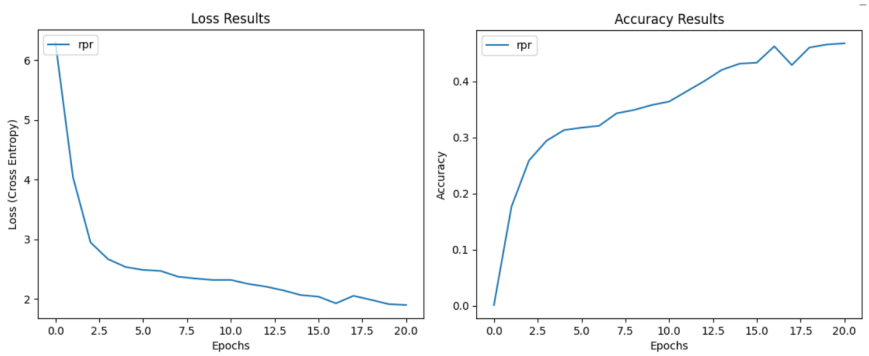

Figure 10: Training Loss and Accuracy plot over Epochs for Drum Dataset

The conversion from MIDI to WAV in the provided notebook is managed by FluidSynth, a library used for synthesizing MIDI data into audio. This process relies on SoundFont files (.sf2) to define the timbre and characteristics of the synthesized audio. In the notebook, two SoundFont files are employed. Firstly, the Piano SoundFont (.sf2) is utilized for generating audio samples from MIDI sequences trained on datasets related to pianos, such as MAESTRO. This SoundFont file contains digital samples of piano sounds, encompassing different notes, dynamics, and articulations. In cases involving custom MIDI datasets like the Groove Drum Dataset, in my case, a specific Drum SoundFont (.sf2) is required to produce audio samples with drum-like sounds. During the conversion process, the MIDI sequences generated by the MusicTransformer model are passed into FluidSynth along with the chosen SoundFont file. FluidSynth then synthesizes the MIDI data, applying the instrument samples and characteristics defined in the SoundFont, ultimately producing audio samples in the WAV format.

In the provided code we can see that the model produces two files before the final audio generation (.wav file), "primer.mid" and "rand.mid". These files fulfill distinct roles in the MIDI sequence generation process. "primer.mid" acts as a guiding seed, providing an initial musical motif or idea to influence the MusicTransformer model's output. By specifying "primer.mid," one can steer the style and structure of the generated sequences, shaping them to align with the primer's characteristics. When the MIDI sequences are generated based on primer.mid, the specified SoundFont file (usually piano.sf2 or drums.sf2 in the notebook) is utilized, as described before. Similarly, if the MIDI sequences are generated randomly from rand.mid, the same SoundFont file (e.g., piano.sf2) is used during the generation process. Consequently, the synthesized audio maintains a piano-like timbre consistent with the characteristics defined in the SoundFont. The output.wav and rest files for both MAESTRO and Custom dataset are provided.

For the Maestro dataset, the output audio .wav files successfully capture the essence of piano playing, reflecting the characteristics and nuances of the trained piano sequences. How-

ever, as the generation process progresses, the audio output tends to become more random, deviating from the structured patterns observed in piano performances. This randomness suggests that the model's ability to generate coherent and musically meaningful sequences diminishes over time. It's possible that with additional training epochs, the model could learn to produce more consistent and structured output, aligning more closely with the original piano compositions in the dataset.

After training on a custom drum dataset for 20 epochs with the provided drumset soundfont, the generated audio initially reflects some rhythmic patterns. However, as generation progresses, it becomes increasingly random, losing coherence. Jazz swing styles in the dataset exacerbate this, adding extra randomness to the outputs. Despite capturing some essence of drum performances initially, maintaining coherence over time proves challenging, suggesting the need for further training or model adjustments.

(The code implementation of this section of the exercise is in the **CS673_ex3.ipynb** notebook and the rest files like the Custom dataset and checkpoints in order to import them in drive and just generate outputs using suitable soundfonts are in the following **MEGA** link link so that the submitted zip is not large. If, for some reason, the link doesn't work, please ask me to resend it.)

(In the included files a have the checkpoint for the best loss training in order to be used for generation)