Homework Assignment 3

COGS 185: Advanced Machine Learning Methods

Due: May 16, 2024 11:59 PM PDT
Instructions: Answer the questions below, attach your code, and insert figures to create a PDF file; submit your file via Gradescope. You may look up the information on the Internet, but you must write the final homework solutions by yourself.
Late policy: 5% of the total received points will be deducted on the first day past due. Every 10% of the total received points will be deducted for every extra day thereafter.

Introduction

The aim of this homework is to experiment on different methods for probabilistic reasoning over time. We will implement Hidden Markov Model (HMM) and Recurrent Neural Networks (RNN), to infer the future and generate articles.

Datasets

For Task 1, we use INTC dataset fetched from Yahoo Finance to train a HMM. Task 3 will train a character-level RNN with the Tinyshakespeare dataset. Table 1 and Table 2 list the description and sources of the 2 datasets.

1 (40 points) Task 1: Hidden Markov Model

In this task, we go through the official tutorials of hmmlearn to get a better understanding of HMMs. `http://hmmlearn.readthedocs.io/en/latest/tutorial.html#multiple-sequences`

	# features	Data source
INTC at Yahoo finance	2 arrays of 125 floating numbers	NasdaqGS Real Time Price(Currency in USD) of Intel Corporation (INTC). `https://finance.yahoo.com/quote/INTC?p=INTC`
Description	Yahoo Finance provided an API to fetch the stock market of trading tickers. Although it is somehow no longer available, there is another fixed library can do the same fetching `https://pypi.org/project/fix-yahoo-finance/`. In this assignment, we have already dumped INTC trading history from Jan. 1st, 2017 to May 30th, 2017 in a pickle file "my_quotes.obj" on the teaching website. We will use the first 100 examples in two attributes "Close" and "Volume" to train a HMM, and then use the trained model to infer market trends of the next 15 days.	

Table 1: INTC dataset description.

	# features	Data source
Tiny-shakespeare	1,075,394 characters	The tinyshakespeare dataset was provided by Andrej Karpathy `https://github.com/karpathy/char-rnn/tree/master/data/tinyshakespeare`
Description	Tinyshakespeare was provided by Andrej Karpathy in the project Char-RNN on GitHub. It is a subset of Shakespeare's works and contains 203,863 words. Including punctuation, spaces, and carriage returns, there are totally 1,075,394 characters. In this assignment, we will use up all characters to train an char-level RNN and then use it to generate articles.	

Table 2: Tinyshakespeare dataset description.

Install hmmlearn
$ pip install hmmlearn –user
More information regarding to the API of hmmlearn can be found via
`http://hmmlearn.readthedocs.io/en/latest/api.html`

1.1 Generate sample from GaussianHMM

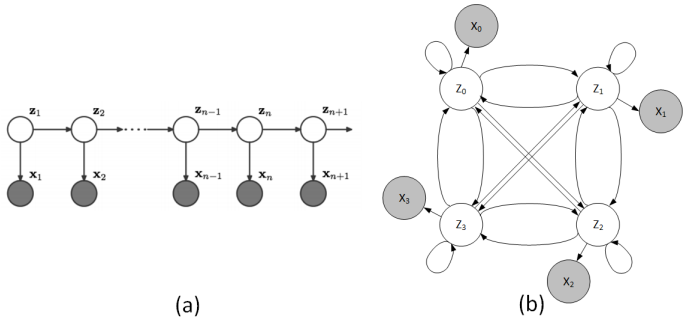


Figure 1: (a) a left-right HMM with n states. (b) a HMM with 4 states.

Figure 1 shows two kind of HMMs. You might have seen the left-right HMM in sub-figure (a) before, but we are working on a HMM with 4 states as shown in figure 1(b). Instead of having an emission matrix to define the behavior of observations, a GaussianHMM draws samples from multiple Gaussian distributions. We use the same setting on the official tutorial to build a GaussianHMM, ghmm0.

$$startprob = np.array([0.6, 0.3, 0.1, 0.0]) \tag{1}$$

$$\begin{aligned} transmat = np.array(\quad & [[0.7, 0.2, 0.0, 0.1], \\ & [0.3, 0.5, 0.2, 0.0], \\ & [0.0, 0.3, 0.5, 0.2], \\ & [0.2, 0.0, 0.2, 0.6]]) \end{aligned} \tag{2}$$

$$means = np.array(\quad \begin{matrix} [[0.0, 0.0], \\ [0.0, 11.0], \\ [9.0, 10.0], \\ [11.0, -1.0]]) \end{matrix} \tag{3}$$

$$covars = 0.5 * np.tile(np.ones([1, 2]), (4, 1)), \tag{4}$$

where *startprob* defines the prior probability distribution of the four hidden states, *transmat* stands for the transmission matrix among the states, and the Gaussian distribution for emission is defined with mean and covariance matrices in *means* and *covars*, respectively.

In your report, please answer the following three questions:

1. Use GaussianHMM(.) to build a HMM with 4 components, and then draw samples of length $L = 15$ three times from the HMM.

   $\mathbf{x}_0$, $\mathbf{z}_0$ = ghmm0.sample(L)

   Print out the three pairs of ($\mathbf{x}_0$, $\mathbf{z}_0$). Explain why each feature in a $\mathbf{x}_i$ is a tuple and can be considered as a planer coordinate. In other words, why $\mathbf{x}_0$ is in shape 15-by-2?

2. Pick up one of the three generated samples, draw the state transition on the figure 1(b). In your report, copy and past the figure 1(b) and draw the state transitions with BLUE arrows to show the path from which the ghmm0 gets the concerned sample.

3. Pick up one generated sample and use

   $\mathbf{POSTERIOR}_0$ = ghmm0.predict_proba($\mathbf{x}_0$)

   to perform forward-backward algorithm and get the posteriors. In your report, print out the $\mathbf{POSTERIOR}_0$ and explain which element is which posterior probability ($P(\mathbf{z}_{(0,i)}|\mathbf{x}_{(0,0)}, \mathbf{x}_{(0,1)} \dots)$).

1.2 Learn another HMM from Samples

Pick up one of the generated samples from subtask 1.1 to train another GuassianHMM, ghmm1, with 4 components through 500 iterations. The training could be done by a sklearn-like fit function.

ghmm1.fit($\mathbf{x}_0$)

Note you might get some deprecation warning about the covariance_type. We simply ignore them for now.

```
import warnings
warnings.filterwarnings('ignore')
```

In your report, answer the following questions:

1. Could you get exact the same parameters as that in ghmm0? Explain the reason whichever you can or cannot get an identical model in your report.

2. Concatenate all 3 generated samples from previous subtask, train ghmm1 on the concatenated samples.

   $\mathbf{x}_{con}$ = **np.concatenate**([$\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$])
   lengths$_{con}$ = **np.array**([**len**($\mathbf{x}_0$), **len**($\mathbf{x}_1$), **len**($\mathbf{x}_2$)])
   ghmm1$_{con}$**.fit**($\mathbf{x}_{con}$, **lengths**$_{con}$)

   Print out the four matrices (*startprob*, *transmat*, *means*, and *covars*) after the training. Are these four matrices more similar with the four of ghmm0?

3. Predict the states with the two GaussianHMMs.

   $\mathbf{z}_1$ = **ghmm0.predict**($\mathbf{x}_0$); $\mathbf{z}_2$ = **ghmm1.predict**($\mathbf{x}_0$)

   Can you get the same sequence of state transitions? ($\mathbf{z}_0 == \mathbf{z}_1 == \mathbf{z}_2$)?
   Hint: the new state encodings might be different, you are supposed to convert the encodings and inspect them carefully to tell whether the three sequences are the same.

1.3 Inference for Stock Market Prediction

In this subtask, we use the first 100 examples of INTC to train a 3-component GaussianHMM (ghmm2), and then use it to predict the future trend of INTC. (WARNING: This assignment is NOT responsible for any loss of real investigation in the market.)

Preprocessing
In the pickle file "my_quotes.obj", there is only one object with 4 attributes: Open, High, Low, Close, Adj Close, Volume. Here, we use Close(closing prices) and Volume as the raw data for the subtask. The total number of examples is 125. We can load the object by

```
quotes = pickle.load(open('my_quotes.obj','r'))
```

Instead of working on absolute closing prices, we use **numpy.diff(.)** to obtain the differences between successive closing prices.

```
diff_c = np.diff(quotes.Close)
```

The we stack the **diff_c** with the scaled volumes to get a binomial dataset.

```
binom1 = np.column_stack([diff_c[:100], quotes.Volume[1:101]/3e7])
```

Note that the **binom1** is a numpy array of shape 100-by-2.

Train the GaussianHMM
Create a GaussianHMM(ghmm2) with 3 components and set the covariance_type to be "diag". Train ghmm2 on **binom1**. Once the training is finished, get the most possible path of hidden states.

```
states = ghmm2.predict(binom1)
```

Visualization
Visualize the transition of hidden states with regard to the closing prices and volumes. Please strictly follow the naming of variables to make the skeleton work for you.

Predict Future Trends
Predict the trends of closing prices and volumes in the successive 15 days after the first 100 days. We have used one shift along the horizontal axis to make the two curves distinguishable by human eyes. In the plotted figure, the curve after day 100 is the predicted trend. The smaller the gap between the two curves, the better the trend-prediction.
Hint: The predicted trend depends on numpy's random seed as well. Please try to modify the seed for 10 times and pick up the best trend with the corresponding random seed.

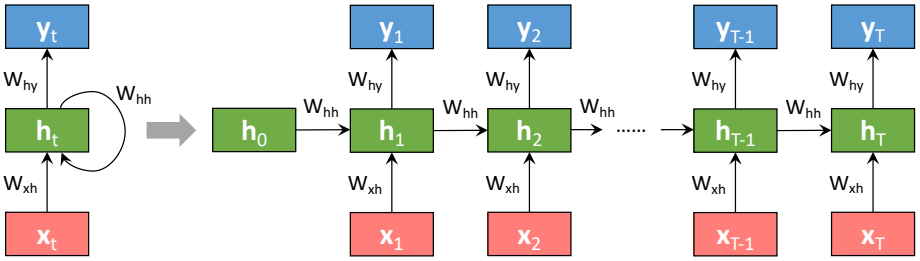
In your report, provide:

1. The four matrices of ghmm2: *startprob*, *transmat*, *means*, and *covars*.
2. The figures plotted with the given visualization code snippet. With your imagination, explain the meanings of the figures. For example, one of your hidden state might always be responsible for sharp droops of the closing prices.
3. The figures plotted with the given trend-prediction code snippet. Is your ghmm2 good at predicting the future trends? How can you improve the prediction? Use a longer history of the closing prices and volumes to train the HMM? Increase the number of hidden states? Or change the trend-prediction based on a totally different idea?
   Hint: This is an open question, and there are many papers or tutorials on-line for the market prediction for your reference. You are not required to modify

the given code snippet to implement your idea. Simply describe a method to improve the trend-prediction is enough.

2 (10 points) Task 2: Simulate Recurrent Neural Network by Hand

In this section, we will use a recurrent neural network (RNN) to process input sequence $\{\mathbf{x}_t\} = \{\mathbf{x}_1, \mathbf{x}_2 ..., \mathbf{x}_{T-1}, \mathbf{x}_T\}$ where $\mathbf{x}_t \in \mathbb{R}^m$ and predict output sequence $\{\mathbf{y}_t\} = \{\mathbf{y}_1, \mathbf{y}_2 ..., \mathbf{y}_{T-1}, \mathbf{y}_T\}$ where $\mathbf{y}_t \in \mathbb{R}^n$. The RNN's structure is shown below:



The left part of the figure is a common illustration of RNN, which includes three parameters $\mathsf{W}_{\text{xh}} \in \mathbb{R}^{l \times m}, \mathsf{W}_{\text{hh}} \in \mathbb{R}^{l \times l}, \mathsf{W}_{\text{hy}} \in \mathbb{R}^{n \times l}$ and a hidden state $\mathbf{h}_t \in \mathbb{R}^l$. It includes two steps:

1. Compute hidden state: $\mathbf{h}_t = \text{ReLU}(\mathsf{W}_{\text{xh}}\mathbf{x}_t + \mathsf{W}_{\text{hh}}\mathbf{h}_{t-1})$.
2. Predict the output: $\mathbf{y}_t = \mathsf{W}_{\text{hy}}\mathbf{h}_t$.

The right part of the figure shows the "unrolled" version of the RNN, which is the real computing path of the network:

1. In the initial stage, you have the initial hidden state $\mathbf{h}_0$.
2. Then, after feeding the first input $\mathbf{x}_1$ of the sequence, RNN will compute the current hidden state $\mathbf{h}_1 = \text{ReLU}(\mathsf{W}_{\text{xh}}\mathbf{x}_1 + \mathsf{W}_{\text{hh}}\mathbf{h}_0)$ where $\text{ReLU}(z) = max(0, z)$.
3. Afterwards, the RNN will compute the output $\mathbf{y}_1 = \mathsf{W}_{\text{hy}}\mathbf{h}_1$ and enter next round.
4. In the next round, the RNN will generate hidden state $\mathbf{h}_2 = \text{ReLU}(\mathsf{W}_{\text{xh}}\mathbf{x}_2 + \mathsf{W}_{\text{hh}}\mathbf{h}_1)$ and the output $\mathbf{y}_2 = \mathsf{W}_{\text{hy}}\mathbf{h}_2$. Eventually, the RNN will compute for T rounds and predict the whole output sequence $\{\mathbf{y}_t\} = \{\mathbf{y}_1, \mathbf{y}_2 ..., \mathbf{y}_{T-1}, \mathbf{y}_T\}$.

Assume $m = 1, n = 1, l = 2$. We are given the initial hidden state $\mathbf{h}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$, the parameters $\mathsf{W}_{\text{hh}} = \begin{bmatrix} 1 & -1 \\ 2 & -3 \end{bmatrix}$, $\mathsf{W}_{\text{xh}} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$ and $\mathsf{W}_{\text{hy}} = \begin{bmatrix} -2 & 3 \end{bmatrix}$, and an input sequence $\{\mathbf{x}_1 = 3, \mathbf{x}_2 = 5\}$. Please calculate the values of the output $\mathbf{y}_1$ and $\mathbf{y}_2$ using the unrolled version of the RNN.

3 (50 points) Task 3: RNN for Periodic Function Modeling

Set up a GPU Environment
Follow the instructions (GPU Cluster setting) on Piazza to setup a GPU environment (DataHub). We will use PyTorch that requires GPU compute.

In Task 2, you obtained some experience about the forward procedure of RNN. In this section, you will train an RNN for periodic function modeling. Assume we have a periodic function $f(t) : \mathbb{R} \to \mathbb{R}$ to model:

$$f(t) = \sin(at + b)$$

We focus on modeling the sequence $\{f(t)\} = \{f(1), f(2)..., f(T-1), f(T)\}$. Specifically, our aim is to train a network to predict the next value of the function $f(t+1)$ given current value of the function $f(t)$ and previous hidden state $\mathbf{h}_{t-1}$. Thus, after training, if we have initial value of function $f(1)$ and initial hidden state $\mathbf{h}_0$, we can iteratively generate the whole sequence $\{f(t)\}$. The following steps are used for sequence generation:

1. We use the current value of function $f(t)$ as input $\mathbf{x}_t \in \mathbb{R}^1$.
2. Based on previous hidden state $\mathbf{h}_{t-1}$ and input $\mathbf{x}_t$, we can calculate the current hidden state $\mathbf{h}_t = \tanh(\mathsf{W}_{\text{xh}}\mathbf{x}_t + \mathsf{W}_{\text{hh}}\mathbf{h}_{t-1})$ and predict the output: $\mathbf{y}_t = \mathsf{W}_{\text{hy}}\mathbf{h}_t \in \mathbb{R}^1$.
3. We will use the current output $\mathbf{y}_t$ as next value of function $f(t+1)$ and perform the above steps again.

In the evaluation stage, if we have initial hidden state $\mathbf{h}_0$, initial value of function $f(1)$ and trained weight $\mathsf{W}_{\text{xh}}, \mathsf{W}_{\text{hh}}, \mathsf{W}_{\text{hy}}$, we can utilize the above steps to generate the whole sequence $\{f(1), f(2), ..., f(T)\}$. Besides, if we have a partial sequence $\{f(1), f(2), ..., f(t-1)\}$, we can also use the RNN to generate $\{f(t), f(t+1), ..., f(T)\}$ to complete the sequence.

In the training stage, we are given the sequence $\{f(1), f(2), ..., f(T)\}$. During training, we keep Step 1 and 2 above. Since we have the ground-truth for $f(t+1)$ during training, we will compute the squared error $\mathcal{L}_t = (f(t+1) - \mathbf{y}_t)^2$ for all $t \in \{1, 2, ..., T-1\}$ and optimize the average loss $\mathcal{L} = \frac{1}{T-1}\sum_{t=1}^{T-1}\mathcal{L}_t$. Note that $f(t+1) - \mathbf{y}_t$ is actually a scalar operation since $\mathbf{y}_t \in \mathbb{R}^1$ can be regarded as a scalar.

In PyTorch implementation, you only need to use `nn.RNNCell()` to perform $\mathbf{h}_t = \tanh(\mathsf{W}_{\text{xh}}\mathbf{x}_t + \mathsf{W}_{\text{hh}}\mathbf{h}_{t-1})$ and `nn.Linear()` to perform $\mathbf{y}_t = \mathsf{W}_{\text{hy}}\mathbf{h}_t$. In the implementation, the input $\mathbf{x}_t$ and output $\mathbf{y}_t$ are both 1-dim vectors. The hidden state $\mathbf{h}_t \in \mathbb{R}^{100}$, which means there are 100 hidden neurons (i.e. hidden size = 100).

Note:

1. In the description above, we ignore the bias terms, while in PyTorch implementation, both `nn.RNNCell()` and `nn.Linear()` have bias terms at default. You can simply use the default setting of `nn.RNNCell()` and `nn.Linear()`.

Download the skeleton code `Task3_Skeleton.ipynb` from the course website. Complete the skeleton code. In order to get full marks, your report should include:

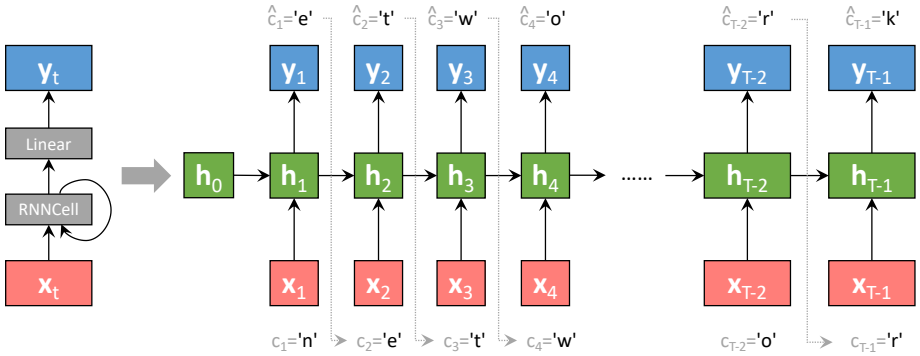
1. Your code.

2. Training loss curve.
3. A figure of generated sequence.

Hint: You may find the official PyTorch tutorial helpful before implementing the network.

4 (15 points) Bonus Task 1: Train a Char-RNN

In this section, you will train a special RNN called Char-RNN (i.e. Character RNN) on language modeling task. Assume we have a character sequence $\{c_t\} = \{c_1, c_2..., c_{T-1}, c_T\}$ where $c_t \in A$ and A is the set of an alphabet (e.g. the character sequence can be $\{c_t\} = \{$'n','e','t','w','o','r','k'$\}$, which is used in the illustration below). Our aim is to train a network to predict the next character c_{t+1} given current character c_t and previous hidden state $\mathbf{h}_{t-1}$. Thus, after training, if we have initial character c_1 and initial hidden state $\mathbf{h}_0$, we can iteratively generate the whole sequence $\{c_t\}$:



The right part of the figure shows the "unrolled" version of the Char-RNN. Assume $|A| = K$, then we can represent the characters in alphabet A into K classes $\{0, 1..., K-2, K-1\}$. Then, we have the following steps for sequence generation:

1. We convert the input character c_t into the one-hot encoding $\mathbf{x}_t \in \mathbb{R}^K$ of the class of the input character.
2. Based on previous hidden state $\mathbf{h}_{t-1}$ and one-hot input $\mathbf{x}_t$, we can calculate the current hidden state $\mathbf{h}_t = \tanh(\mathsf{W}_{\text{xh}}\mathbf{x}_t + \mathsf{W}_{\text{hh}}\mathbf{h}_{t-1})$ and predict the output: $\mathbf{y}_t = \mathsf{W}_{\text{hy}}\mathbf{h}_t$.
3. Afterwards, we can use the softmax function to obtain the probability vector $\mathbf{p}_t = \text{Softmax}(\mathbf{y}_t) = \{p_0, ..., p_{K-1}\} \in \mathbb{R}^K$. Then, we will sample the class $\hat{k}$ from the probability vector $\mathbf{p}$ and find the corresponding character in alphabet A as predicted character $\hat{c}_t$. Note that here we do not use $\hat{k} = \arg\max_k p_k$ (i.e. the index with maximum probability) as prediction since it will reduce the diversity of generated sequence.
4. We will use the current predicted character $\hat{c}_t$ as next input character c_{t+1} and perform the above steps again.

In the evaluation stage, if we have initial hidden state $\mathbf{h}_0$, initial input character c_1 and trained weight $\mathsf{W}_{\text{xh}}, \mathsf{W}_{\text{hh}}, \mathsf{W}_{\text{hy}}$, we can utilize the above steps to generate the whole sequence $\{c_1, c_2, ..., c_T\}$.

In the training stage, we are given the sequence $\{c_1, c_2, ..., c_T\}$. During training, we keep Step 1 and 2 above and calculate the probability vector $\mathbf{p}_t$ in Step 3. Since we have the ground-truth for c_{t+1} during training, we will compute the cross-entropy loss $\mathcal{L}_t = \text{CrossEntropy}(\mathbf{p}_t, c_{t+1})$ for all $t \in \{1, 2, ..., T-1\}$ and optimize the average loss $\mathcal{L} = \frac{1}{T-1}\sum_{t=1}^{T-1}\mathcal{L}_t$.

The left part of the figure shows the actual network in PyTorch implementation. You only need to use `nn.RNNCell()` to perform $\mathbf{h}_t = \tanh(\mathsf{W}_{\text{xh}}\mathbf{x}_t + \mathsf{W}_{\text{hh}}\mathbf{h}_{t-1})$ and `nn.Linear()` to perform $\mathbf{y}_t = \mathsf{W}_{\text{hy}}\mathbf{h}_t$. In the implementation, we have the size of alphabet $K = 100$, which means one-hot input $\mathbf{x}_t$ and output $\mathbf{y}_t$ are both 100-dim vectors. The hidden state $\mathbf{h}_t \in \mathbb{R}^{100}$, which means there are 100 hidden neurons (i.e. hidden size = 100).

Note:

1. In the description above, we ignore the bias terms, while in PyTorch implementation, both `nn.RNNCell()` and `nn.Linear()` have bias terms at default. You can simply use the default setting of `nn.RNNCell()` and `nn.Linear()`.
2. The training process may need 50 - 60 minutes to run.

Download the skeleton code `Bonus_Task1_Skeleton.ipynb` from the course website. Complete the skeleton code. In order to get full marks, your report should include:

1. Your code.
2. Training loss curve.
3. A sample of generated sequence.

Hint: You may find the official PyTorch tutorial helpful before implementing the network.

5 (10 points) Bonus Task 2: Transfer Learning with BERT

NOTE: This part is not runnable on Datahub. It is recommended to use Google Colab for this assignment. Change runtime type to GPU in order to maximize the experiment speed.

BERT (Bidirectional Encoder Representations from Transformers) is a recent language representation model which achieved state-of-the-art results for a wide range of natural language processing tasks such as question answering, language inference, etc. The training objectives of BERT is to predict a blank (word vector) from its surrounding (left and right) context. This is similar to recurrent neural networks (RNN) training we have seen before. However, RNN training involves prediction from left context. After it is trained to predict unlabelled text, BERT is able to achieve strong results on various downstream tasks at hand by fine-tuning with just one additional output layer.

In this task, we aim to train a predictor of sarcasm on news headlines. This is done by fitting a linear classifier (i.e., logistic regression) on top of features extracted by BERT encoder. Each datapoint consists of a text, e.g. "after careful consideration, bush recommends oil drilling" and a label `is_sarcastic`. Original version of News Headlines Dataset For Sarcasm Detection contains around 30k datapoints. We have made the size of the dataset to be $\frac{1}{10}$ of original size to allow for a faster training.
Download the skeleton code `Bonus_Task2_Skeleton.ipynb` from the course website. In order to get full marks, your report should include:

1. Train accuracy and Validation accuracy vs. training epoch
2. Best train accuracy and validation accuracy vs. various learning rates
3. Your code.