Hong Kong University of Science and Technology
COMP 4211: Machine Learning
Spring 2025

Programming Assignment 2
Due: 11 April 2025, Friday, 11:59pm

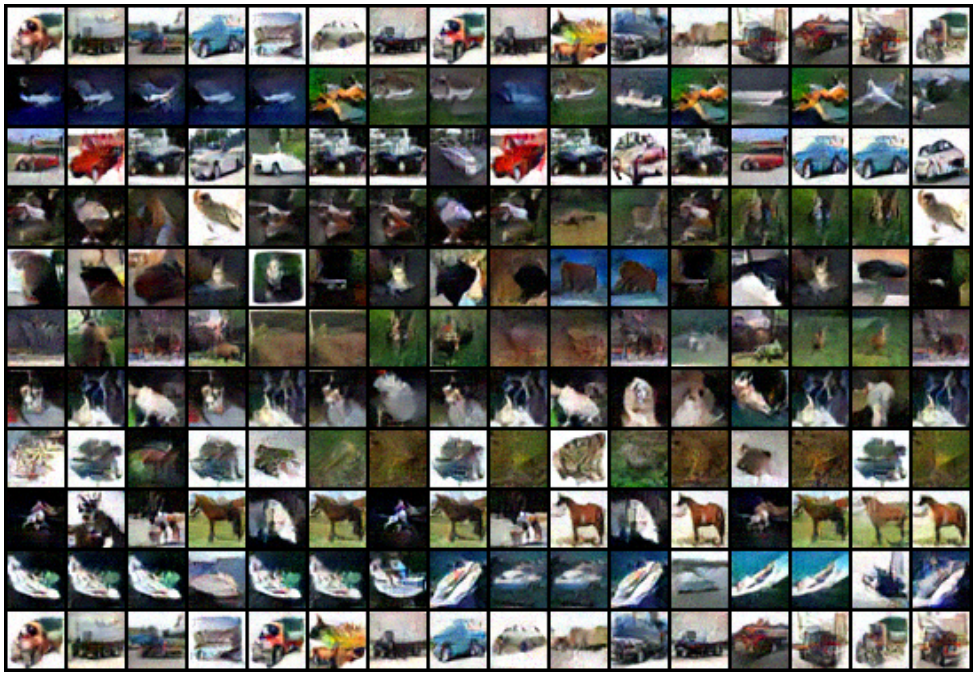

Figure 1: Images generated from a JEM model, to be implemented later in this assignment.

1 Introduction

This assignment is designed to provide students with a hands-on experience with implementation and training of a neural network model for a classification task. Afterwards, the model will be re-applied for a different kind of training, resulting in a type of generation model known as Energy-Based Model (EBM). Students will be introduced to some basic ideas of these models, along with a step-by-step guide to implementing a training routine for an EBM.

2 Objectives

In this programming assignment, students are expected to learn to:

- Utilize the TensorFlow and Keras modules to construct and train a neural network model.
- Implement a custom training routine for TensorFlow models to solve classification tasks.
- Implement a training routine for a non-standard training task, including the application of gradient calculation (`tf.GradientTape`) to variables other than model parameters.
- Analyze and comment on the performance of neural network models.

3 Tasks

This assignment is divided into two main parts: the coding part and the written report. Both parts are required for completing this assignment and will contribute to your final grade.

3.1 Coding Part

In the coding part of this assignment, you will complete the following tasks:

1. **Standard Classification Model**: Start by implementing a standard classification model using TensorFlow and Keras. This model should be trained on the CIFAR-10 dataset. You will focus on building a standard discriminative classifier, training it on the CIFAR-10 images, and evaluating its accuracy on the test set.

2. **Joint Energy-based Model (JEM)**: Re-purpose your classification model by training it as a Joint Energy-based Model (JEM). In this task, you will implement the energy-based modeling aspect to handle both classification and generation tasks. This includes:

    - Implementation of MCMC sampling of image data from the model.
    - Implementation of a Replay Buffer to store and retrieve past training samples.
    - Re-training the model using the JEM training routine to learn the joint distribution of images and labels.

All code must be submitted in a Jupyter Notebook format containing all the running results. Provide comments and explanations throughout your code to demonstrate your understanding of each step. A skeleton code has been provided to guide you in implementing these tasks, with specific coding segments labeled as $[C_n]$.

3.2 Written Report

In the written report, you are expected to:

- Answer questions related to the implementation and procedures of the training tasks.
- Present your training and testing results for the classification model and the JEM model.

Each question in the report is denoted by $[Q_n]$.

3.3 (Optional) Bonus Tasks

To encourage further experimentation and understanding on the concepts introduced in this assignment, there are several optional bonus tasks for self-exploration, such as:

- Further understanding of the mechanism behind JEM, the model trained in Part 2 of this assignment.
- Experimentation of functions introduced in Part 2, including data generation using the JEM model.
- Further training of the models to achieve optimal performance.

These tasks are not required for completing this assignment, but will provide bonus marks for in case some points are lost in the main sections. Attempts to these tasks should be done mainly as part of your written report, with relevant code implementations and screenshots provided to prove your completion of the tasks.

Each bonus task is denoted by $[B_n]$.

Note: Please refer to Section 10 for submission guidelines and Section 12 for academic integrity policies.

4 Environment Setup

All the necessary libraries are imported in the given Jupyter Notebook. Please refrain from importing additional libraries.

We recommend that all of the coding task be done on Google Colab using GPU as hardware accelerator. The option can be found in the arrow menu next to Connect → Change runtime type. Do not use CPU for training/testing or else the execution time will be extremely long.

Note that Google Colab has a (hidden) quota for GPU usage, after which you will not be able to use the GPU in your runtime for a while. If that happens, you have three options:

- Wait until the (undetermined) timeout is over. You are therefore advised to start the assignment as soon as possible so that you have enough time to finish the required tasks.
- Pay for Google Colab Pro to have more access to their GPU.
- Run the Jupyter Notebook offline in your local machine. This is not recommended as we are unable to provide support for any technical issues in your computer.

  Here are the library settings verified to be working (but not required):

  - `tensorflow==2.17.0`
  - `cuda==12.2`

5 Part 1: Classification Task

This part of the assignment focuses on implementing a CNN model to solve a classification task. In this part, you will construct an image classification model, implement a simple training routine and test the model for its test-time performance.

5.1 Dataset Description

The dataset used in this assignment is the CIFAR-10 dataset, an image dataset containing tiny color images (32×32), classified into 10 class labels of different objects (e.g., animals or vehicles). This dataset is commonly used to evaluate the performance of small-scale classification models.

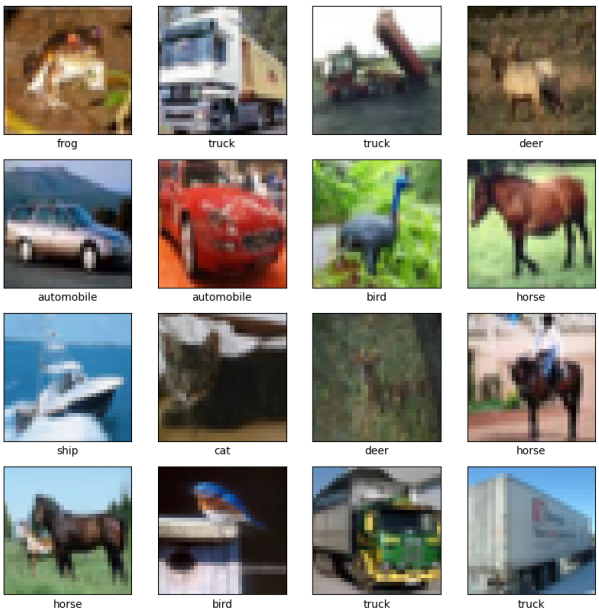

Figure 2: Sample Images from the CIFAR-10 Dataset

[Q_1] Showcase some samples from the training dataset, with at least 3 images per class. Show each image with its corresponding class label.

The code for downloading and preprocessing the images has been provided in the Jupyter notebook.

5.2 Model

For the classification model we will use the Wide ResNet architecture (WRN), a variant of the ResNet model that focuses on width (number of channels) over depth (number of layers). The particular configuration of the model to be used is called WRN-16-4:

Layer Type	Configuration	Output Size
2D Convolution	16 channels, 3×3, same padding	Input Size
ResNetBlock (see 5.2.1)	64 channels, no downsampling	Input Size
ResNetBlock	64 channels, no downsampling	Input Size
ResNetBlock	128 channels, with downsampling	Input Size / 2
ResNetBlock	128 channels, no downsampling	Input Size / 2
ResNetBlock	256 channels, with downsampling	Input Size / 4
ResNetBlock	256 channels, no downsampling	Input Size / 4
LeakyReLU		Input Size / 4
Global Average Pooling	256 channels	(Single Tensor)
Dense	(number of classes) channels	(Single Tensor)

Table 1: Structure of the WRN-16-4 Classfication Model

The model takes in an image (assumed to be of size 32×32) and outputs a single vector containing the logits for the class labels.

Learn more about Wide ResNet here: `https://arxiv.org/pdf/1605.07146`

5.2.1 ResNetBlock

A ResNet consists of multiple "stages" of processing, where the feature map is progressively downsampled in size and widened in the number of channels. Each stage consists of a stack of ResNet **blocks**, usually with the same number of output channels, and the first block in each stage also performing downsampling (See Table 1).

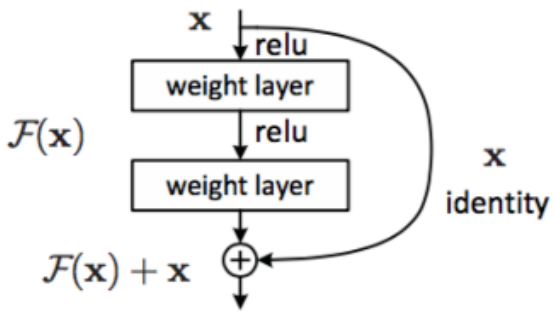

Figure 3: ResNet block to be implemented in the model. Note that this differs slightly in structure from the one shown in the lecture slides.

Some details on the implementation of the ResNet block in this model:

- **Convolution Layers:** The $\mathcal{F}(\mathbf{x})$ part of this model consists of 4 layers: 2 LeakyReLU activations (default setting), and 2 3×3 Convolution, with the `'same'` padding.

  The number of output channels (`out_channels`) should be the same for both Conv2D layers, with the LeakyReLU passed **before** each convolution.

  Note: This implementation does not use any normalization layer, unlike the original proposed model.

- **Skip Connection:** The skip connection ($+\mathbf{x}$ part) should be an identity function (i.e., the original input, unchanged) if $\mathbf{x}$ and $\mathcal{F}(\mathbf{x})$ have the same dimensionality (shape).

  If the dimensionalities are different, then the skip connection should use a 1×1 convolution mapping from (`in_channels`) to (`out_channels`) channels, with **no bias**. Then the output should look like $\mathcal{F}(\mathbf{x}) + \text{skip}(\mathbf{x})$.

- **Downsampling:** Downsampling, if needed, is handled using **strides**. Set `strides = 2` on the second convolution layer to perform downsampling.

  Since this modifies the $\mathcal{F}(\mathbf{x})$ dimensionality, the same stride setting should also be added in the skip connection convolution.

[C_1] Create a Keras Layer implementing the ResNet Block.

[Q_2] [C_2] Create a Keras Model implementing the WRN-16-4 Model. Report the number of trainable parameters.

Tip: The WRN model can be implemented with the Keras `Sequential` function, using the ResNet Blocks.

5.3 Training Routine

For this task, you will implement a custom training step for the model, which a skeleton is provided for you as `part1_train_step()`. The training step function takes in an optimizer (`optim`), a classification `model`, a batch of image `data` and `label`.

The training step function should consist of the following steps:

1. Convert the label into a one-hot encoding, and augment the image data $\mathbf{x}$ by adding Gaussian noise of standard deviation `sigma` (default 0.03).

2. With a GradientTape:

   (a) Compute the logits for the image batch using the model.
   (b) Compute the categorical Cross Entropy loss by comparing the logits with the ground truth.

3. Obtain the gradients of the loss with respect to the model's trainable variables from the gradient tape.

4. Apply the gradients using the optimizer. This is where the model learns by updating its weights based on the computed gradients.

5. Return the loss obtained, as a dictionary. These will be displayed during the training process bar in real time.

   You are allowed to return different values for monitoring purposes, but you must map the key `'loss'` to the total loss.

[Q_3] What is the use of adding Gaussian noise to the image data?

[C_3] Implement the training step function `part1_train_step()` given the above instructions.

5.3.1 Training Loop (Given)

The main training loop function has been provided to you (`train_loop_1()`). This function takes in a `model`, an `optimizer`, a `train_step` function, the number of `epochs`, the interval between weight saves.

It performs the following:

- Load the data batches and pass the corresponding arguments to the supplied `train_step` function.
- Obtain the loss from the training step, accumulating them and reporting the current loss using the `tqdm` progress bar.
- Report the average loss in each epoch, and optionally, the test accuracy of the model.

  You can uncomment the "Test Accuracy" section to track your model's performance, provided you have implemented 5.4.
- Save the model every few epochs for restoration if something goes wrong during training.

[Q_4] Train your model with the above task for at least 20 epochs. Report the loss for at least the first and last epochs of training.

5.4 Testing the Model

[C_4] Implement a testing function `evaluate_accuracy` that evaluates the accuracy of the model in classifying a given dataset.

[Q_5] Report the accuracy of the model trained in the previous section, tested against the **test** dataset. Showcase at least 4 misclassified samples (along with the predicted and ground truth label).

5.5 Regularization

Regularization is widely used to avoid the overfitting issues in the training of neural networks.

[C_5] Choose **ONE** regularization method, such as **Dropout** or **Batch Normalization**, and implement it in the neural network you have trained. For Dropout, refer to Section 2.4 in `https://arxiv.org/pdf/1605.07146`. For Batch Normalization, refer to `https://www.neuralception.com/objectdetection-batchnorm/`. Modify the model class `ResBlockRE`, train your modified model on the same task for at least 20 epochs, and test it on the **test** dataset as before.

[Q_6] Report the accuracy of the modified model when tested against the **test** dataset. Compare the performance with and without regularization, and briefly describe your findings and explanations.

6 Part 2: Joint Energy-Classifier Training

Have you ever wondered what a classification model considers as the "most likely" image for a certain class? Can you perform "gradient descent" on images to find the most "bird"-like image, for example?

This strategy of data generation is known as Energy-Based Model (EBM), where models are trained so that they model the "energy" (related to likelihood) of an input space from which data samples can be drawn.

In this part, you will implement a variant of Energy-Based Model called Joint Energy-based Model (JEM), which interprets a logit-based classification model as an EBM so that both classification and generation can be achieved with a single model.

This part will only provide a minimal description of the functions and procedures needed. For more information, you can refer to supplementary material in Section 8 and the original paper proposing this algorithm here: `https://arxiv.org/pdf/1912.03263`

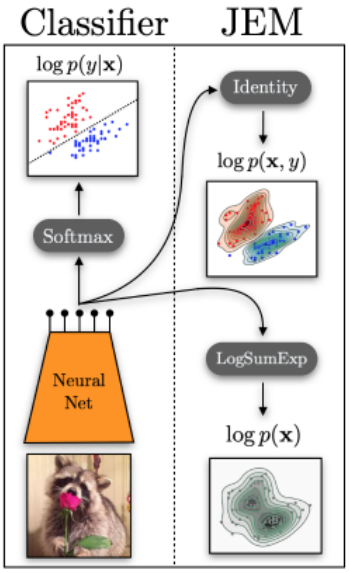

Figure 4: Joint Energy-based Model. Source: `https://arxiv.org/pdf/1912.03263`

6.1 Simplified Description

As Figure 4 suggests, there are three ways to interpret or process the output logits $f_\theta(\mathbf{x})$:

1. **Softmax**: This is the usual classifier interpretation, as used in Part 1. It denotes $p(\mathbf{x}|y)$, the "Probability of $\mathbf{x}$ being of class y"

2. **Identity**: JEM interprets this as the (negative) "energy" of the pair $(\mathbf{x}, y)$. It is an indicator of the "Likelihood of $(\mathbf{x}, y)$ being a realistic data". In particular, $E_\theta(\mathbf{x}, y) := -f_\theta(\mathbf{x})[y]$, where $f_\theta(\mathbf{x})[y]$ is the logit for the label y.

   This is just like a different, energy-based interpretation for likelihood that we typically used as the loss function.

3. **LogSumExp**: This is the (negative) "energy" of $\mathbf{x}$ itself, i.e. an indicator of the "Likelihood of $\mathbf{x}$ being a realistic input" (realistic image in our case).

   LogSumExp refers to $\log \sum_y \exp(f_\theta(\mathbf{x})[y])$, and $E_\theta(\mathbf{x}) := -\text{LogSumExp}(f_\theta(\mathbf{x}))$.

For the purpose of this assignment, you are not required to understand why it is like this theoretically, instead you can just understand "energy" $E_\theta(\mathbf{x}, y)$ (when label is given) or $E_\theta(\mathbf{x})$ (when label is not given) as a score (lower is better) for how realistic the input is. And this is going to be the loss function to optimize when we want to sample images from the model, for example, we want to find the image that minimizes the energy. If you do want to understand better on the theoretical principles of JEM (recommended), we encourage you to read through Section 8.

[C_6] Implement `energy()`. Given model (f_θ), a batch of data ($\mathbf{x}$), and optional batch of corresponding labels (y), return:

- If y is supplied, $E_\theta(\mathbf{x}, y)$ (interpretation 2).
- Otherwise, $E_\theta(\mathbf{x})$ (interpretation 3).

Tip: Since LogSumExp is a surprisingly common operation in machine learning, it has its own built-in function in TensorFlow: `tf.reduce_logsumexp()`

This function will help us in training this JEM model, as well as the sampling process.

[Q_7] Run the `visualize_energy()` function to visualize $E_\theta(\mathbf{x}, y)$ and $E_\theta(\mathbf{x})$ for real image, noise image, and grey image from the classification model in Part 1. Report the obtained visualization grid.

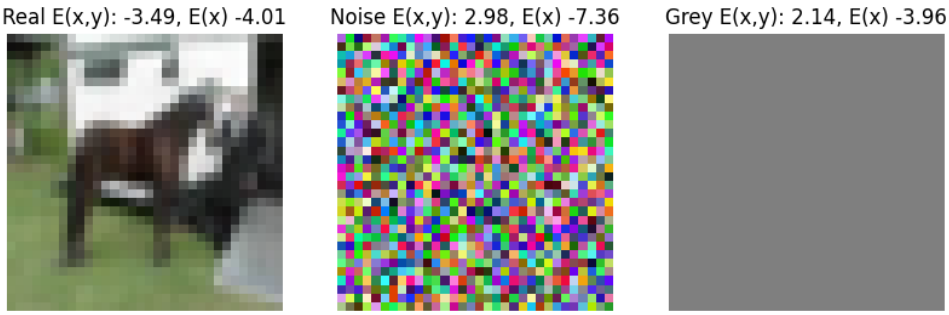

Figure 5: Example of the Energy visualization from the classification model

6.2 Sampling x

As mentioned, sampling $\mathbf{x}$ (generating image) can be done using a gradient descent method known as Stochastic Gradient Langevin Dynamics (SGLD):

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \frac{\partial E_\theta(\mathbf{x}_t)}{\partial \mathbf{x}_t} + \epsilon$$

Compare this to SGD learned in class:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{\partial L(\mathbf{w}_t; \mathcal{S})}{\partial \mathbf{w}_t}$$

The difference between these two procedures are:

- The input is being optimized instead of the model weights.
- SGLD adds $\epsilon \sim \mathcal{N}(0, \sigma^2)$, a Gaussian noise of a predetermined standard deviation σ. Here adding noise ensures proper exploration of the energy landscape, which is critical for both optimization stability and generative diversity.

Usually this process is repeatedly applied for some $T = 80$ times, starting from a random distribution (e.g., Uniform noise). By repeatedly "optimizing" for "likelihood" (negative energy), this process can sample more data considered realistic by the model, effectively working as a generative model.

[C_7] Implement `sampling_step()`. Given a `model` (i.e., f_θ), a `data` (i.e. $\mathbf{x}_t$), an optional `label` (i.e., y), `step_size` (α), `noise_amp` (σ), compute and return $\mathbf{x}_{t+1}$ (i.e., step 2 to 4 in the procedure above).

The skeleton code is given in the notebook. Here are some tips:

- You need to use the `energy()` to calculate $E_\theta(x_t)$ and its gradient. The individual energies should be aggregated using **sum** for a correct calculation of the gradient.
- To calculate the gradient, we use the same `tf.GradientTape()` method used in 5.3, but with respect to `data`. The first 2 lines have been given to you.
- While it is still possible to use an optimizer (SGD), the gradient update can be applied directly to the data, which is simpler in this case.
- (Given) To ensure that the sample is in the valid pixel space, we bound the values within $[0, 1]$ after the gradient update.

6.2.1 Replay Buffer

To train a JEM, one also needs to sample some new $\mathbf{x}'$ to calculate the loss. Since calculating gradients 80 times per batch takes too much time, JEM instead uses a **Replay Buffer** (`SampleBuffer` in the code), which stores past sampled data and reuses them for further optimization:

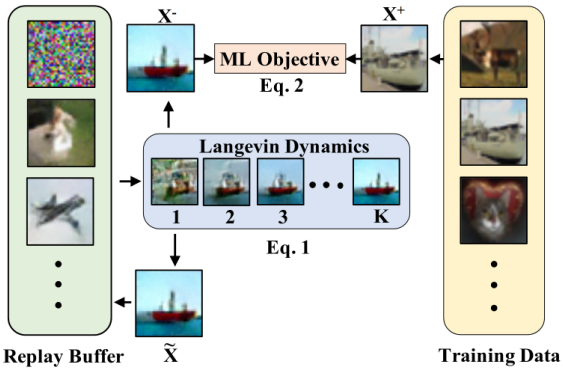

Figure 6: Replay Buffer in energy training.
Source: `https://arxiv.org/pdf/1903.08689`

[C_8] Implement the `SampleBuffer` class, which needs to support the following:

- Initialization (given): Stores some parameters for use in other methods.
- `add_to_buffer`: Append sample data and labels to a limited-size buffer (`self.buffer`). If the number of buffered samples exceeds `max_samples`, remove the oldest samples (First in, first out).
- `sample_from_buffer`: Return (`n_samples`) samples such that each sample has a `p_new` chance to be generated from uniform noise (data from 0 to 1, label from 0 to `(n_class - 1)`), or taken from the buffer otherwise.
  (This is different from generating exactly `n_samples * p_new` new samples)

  Also handle the special case where there is nothing in the buffer, in which case just generate all samples from the uniform distribution.

  Return a tuple of (images, labels).

This buffer allows the sampling of random data using a lower T (e.g., 20), speeding up the training process at the expense of a less accurate loss calculation.

[Q_8] Run the `visualize_buffer_samples()` with `p_new=0.5` and report the visualization.

6.3 JEM Training

With the tools above, we can finally start training the JEM model.

Simply put, JEM training differs from pure classification in these two steps:

1. A non-dataset $\mathbf{x}'$ is to be sampled, using the `SampleBuffer` and `sample_step()` described above.
2. An extra loss term is added using the `energy()` of real data $\mathbf{x}$ and sampled data $\mathbf{x}'$, on top of the existing CrossEntropy loss for classification.

The step-by-step detail is described in the coding task below.

6.3.1 JEM Training Step

[C_9] Implement `part2_train_step()`. This has almost the same function signature as the Part 1 version in Section 5.3, but has an extra `sb` argument which stands for `SampleBuffer`. It should consist of the following steps:

1. Convert the label into a one-hot encoding, and augment the image data $\mathbf{x}$ by adding Gaussian noise of standard deviation `sigma` (default 0.03).
2. Sample images and labels $(\mathbf{x}', y')$ from the SampleBuffer `sb`. The number of these new samples should be equal to the batch size.
3. Run `sample_step()` 20 times using $\mathbf{x}'$ and y' to obtain an "optimized" $\mathbf{x}'_+$.
4. Add $(\mathbf{x}'_+, y')$ back to the SampleBuffer.
5. With a GradientTape:

(a) Compute the logits for the $\mathbf{x}$ and $\mathbf{x}'_+$ respectively, using the model.
(b) Calculate the following loss using cross entropy (like in Part 1, use only $\mathbf{x}$) and 2 `energy()` values:

$$\mathcal{L} = \text{CrossEntropy}(y, \mathbf{x}) + E_\theta(\mathbf{x}) - E_\theta(\mathbf{x}'_+)$$

6. Follow the rest of Section 5.3 from Step 3.

A skeleton code `part2_train_step()` is provided for you.

6.3.2 Execution

While Section 6.2.1 introduced optimizations to the training, this algorithm is still considerably slower than pure classifier training. Due to potential time/quota limitation concerns, we only require training to prove that your implementation can lead to successful/stable training. Bonus marks will be given for further experimentation and explorations, including successful generation using the trained model.

[Q_9] Train the second model by calling `train_loop_2()` for at least 2 epochs. Report the loss for each epoch. **Do not reuse** the weights in Part 1.

[Q_{10}] Report the accuracy of the JEM model after your training. How does it compare with the model in Part 1 (with the same amount of training)? Try to describe and explain your observations.

Some tips on JEM training:

- If you have to save/load weights between sessions, save the buffer as well for better training results.
- Unfortunately, JEM training has a chance to fail, even if set up correctly. If you observe your loss to diverge to a very high magnitude, check the following:
  - Whether you trained on a reinitialized model or not
  - Whether the signs of the energy and loss are correct
  - Whether there exists some hidden scaling of values that may make the result incorrect. This may be caused by incorrect aggregation (sum vs. mean), implicit broadcasting, etc.
  - Whether the hyperparameters match with the default settings:

Hyperparameter	Default
Adam Learning Rate	10^{-4}
Step Size	1
Sampling Noise σ	0.01
$\mathbf{x}'$ Sampling Step T	20
Input Noise σ	0.03
Buffer Size	10000
Buffer p_{new}	0.05

  Otherwise, it is still possible that you are just unlucky and have to retry.

  If you have further problems with training, please ask in Piazza as soon as possible.

7 Bonus Part

The questions in this section are optional and designed to provide additional scoring opportunities and help students explore more on the JEM model. The maximum possible score for this assignment remains 100 points. If you lose points in Part 1 and 2, the bonus points from this section can help recover your score up to the maximum.

7.1 Exploration

These questions require some extra testing with the JEM model and some of its procedures. If the relevant coding tasks or assumptions are not fulfilled, attempts to these questions may not be graded.

7.1.1 Sample Step

[B_1] What happens if you call `sample_step()` using the model trained in Part 1? Try generating samples from pure noise as described in Section 6.2 and visualize your results.

[B_2] What happens if you put a negative value as the step size, and a real sample as the data? What if you try to classify the resulting image (after a few repetitions) with the same model? Visualize and report your findings.

7.1.2 Data Generation/Sampling

These tasks assumes you have successfully trained a JEM model for an extended period, to be able to generate new images. These may take longer than a single GPU quota in Colab.

To get new image samples, you have two options:

1. Sample from the SampleBuffer you obtained from the training, with `p_new = 0`.
2. Generate from pure noise, using the method in Section 6.2.

[B_3] Showcase a batch (at least 32) samples of generated image, either from scratch or from the SampleBuffer returned from training (or both).

Just sampling for random $\mathbf{x}$ might not be enough. What if you want to find best quality images, or just the most "bird"-like image as mentioned in the beginning of Part 2? You can also use the JEM model as a metric, where the lower energy $E_\theta(\cdot)$ indicates a higher-quality image according to the model.

[B_4] Showcase at least the "top 16" samples for each class, ranked by the energy output by your model, and the top 16 overall using the unconditional energy $E_\theta(\mathbf{x})$. See Figure 1 for example.

[B_5] What happens if you further apply the `sampling_step()` on a real data (either with no label, correct or even wrong label)? Visualize the resulting image over a series of sampling steps and try to describe the effect.

8 Supplementary Materials: Understanding Energy and JEM

In this section, we provide additional details about EBMs for those interested in a deeper understanding.

8.1 What do the energy functions mean?

Given model parameters θ, EBMs define an energy function $E_\theta(\cdot)$ on the input space ($\mathbf{x}$, e.g., images in this assignment), such that one can derive the likelihood of some input $\mathbf{x}$ as

$$p_\theta(\mathbf{x}) = \frac{\exp(-E_\theta(\mathbf{x}))}{Z_\theta}$$

where $\exp(-E_\theta(\mathbf{x}))$ essentially serves as a "weight" for the probability defined by θ, normalized using the constant $Z_\theta = \int_{\mathbf{x}'} \exp(-E_\theta(\mathbf{x}'))d\mathbf{x}'$.

The idea of JEM comes from comparing the above to the softmax function over logits from a model $f_\theta(\cdot)$:

$$p_\theta(y|\mathbf{x}) = \frac{\exp(f_\theta(\mathbf{x})[y])}{\sum_{y'} \exp(f_\theta(\mathbf{x})[y'])}$$

They propose modelling the likelihood of $(\mathbf{x}, y)$ in a similar fashion:

$$p_\theta(\mathbf{x}, y) = \frac{\exp(f_\theta(\mathbf{x})[y])}{Z_\theta}$$

i.e., $E_\theta(\mathbf{x}, y) := -f_\theta(\mathbf{x}, y)$, the 2nd interpretation in Section 6.1.

Note that if the two models f_{θ_1} and f_{θ_2} only differ by a constant ($f_{\theta_1}(\mathbf{x}) \equiv f_{\theta_2}(\mathbf{x}) + c$), the softmax value will be identical, but the energy interpretation will be different.

This explains why a model trained with just classification loss may not have a correct energy, as shown in the energy visualization task.

If we further let $p_\theta(\mathbf{x}) = \sum_{y'} p_\theta(\mathbf{x}, y')$, we can get

$$\begin{aligned} p_\theta(\mathbf{x}) &= \frac{\sum_{y'} \exp(f_\theta(\mathbf{x})[y])}{Z_\theta} \\ &= \frac{\exp(-(-\log \sum_{y'} \exp(f_\theta(\mathbf{x})[y])))}{Z_\theta} \\ E_\theta(\mathbf{x}) &= -\log \sum_{y'} \exp(f_\theta(\mathbf{x})[y']) \end{aligned}$$

i.e., the 3rd interpretation in Section 6.1.

Finally, we train a JEM with the loss $\mathcal{L} = -\log p_\theta(\mathbf{x}, y) = -\log p_\theta(\mathbf{x}) - \log p_\theta(y|\mathbf{x})$, the negative log-likelihood of the data-label pair $(\mathbf{x}, y)$.
Considering $(-\log p_\theta(y|\mathbf{x}))$ is the Categorical Cross Entropy loss, this means the training is effectively a "joint training" on classification and generation.

8.2 Why do we need to sample x during training?

We know that $-\log p_\theta(y|\mathbf{x})$ is Cross Entropy. What about $-\log p_\theta(\mathbf{x})$?

From the definition, we know it is $E_\theta(\mathbf{x}) + \log Z_\theta$. But since Z_θ is too hard to calculate, we rely on its gradient, which resolves to

$$\begin{aligned} \nabla_\theta(-\log p_\theta(\mathbf{x})) &= \nabla_\theta \left(E_\theta(\mathbf{x}) + \log \int_{\mathbf{x}'} \exp(-E_\theta(\mathbf{x}'))d\mathbf{x}' \right) \\ &= \nabla_\theta E_\theta(\mathbf{x}) - \mathbb{E}_{p_\theta(\mathbf{x}')}[\nabla_\theta E_\theta(\mathbf{x}')] \end{aligned}$$

Which means we can approximate it by sampling $\mathbf{x}'$ from $p_\theta(\cdot)$ (defined by the model),

$$-\log p_\theta(\mathbf{x}) \approx E_\theta(\mathbf{x}) - E_\theta(\mathbf{x}')$$

Note that SGLD (in Section 6.2 is not the only way to sample $\mathbf{x}$, but it is the one used in the original JEM proposal. Subsequent research has proposed alternative samplers, but this remains the simplest to implement.

9 Some Programming Tips

As is always the case, good programming practices should be applied when coding your program. Below are some common ones but they are by no means complete:

- Using functions to structure your code and scope variables properly
- Using meaningful variable and function names to improve readability
- Using consistent styles
- Including concise but informative comments
- Using a small subset of data to test the code
- Using checkpoints to save partially trained models

For documentation and guide on using TensorFlow, refer to their official website: `https://www.tensorflow.org/api_docs/python/tf`

10 Assignment Submission

Assignment submission should only be done electronically in the Canvas course site.

There should be two files in your submission with the following naming convention required:

1. **Report** (with filename `report.pdf`): in PDF format.
2. **Source code and prediction** (with filename `code.zip`): all necessary code and running processes should be recorded into a single ZIP file. The ZIP file should include at least one notebook recording all the training and evaluation results. The data should not be submitted to keep the file size small.

When multiple versions with the same filename are submitted, only the latest version according to the timestamp will be used for grading. Files not adhering to the naming convention above will be ignored.

11 Grading Scheme

This programming assignment will be counted towards 15% of your final course grade. The maximum scores for different tasks are shown below:

Table 2: [C]: Code, [Q]: Written report

Grading Scheme	Code(62)	Report(38)
Classification Task (55)		
- [Q1] Showcase samples (+labels) from CIFAR-10 Dataset		4
- [C1] Implement ResNet Block	7	
- [C2 + Q2] Implement WRN-16-4	7	3
- [Q3] Explain the use of noise in training		2
- [C3] Implement `part1_train_step()` function	7	
- [Q4] Classification training & report loss		7
- [C4] Implement `evaluate_accuracy()` function	5	
- [Q5] Evaluate model on test dataset		5
- [C5] Implement `ResBlockRE`	4	
- [Q6] Evaluate modified model on test dataset		4
JEM Training (45)		
- [C6] Implement Energy Function	2	
- [Q7] Visualize Energy Function		3
- [C7] Implement `sampling_step()` function	8	
- [C8] Implement `SampleBuffer` class	7	
- [Q8] Visualize Sample Buffer		3
- [C9] Implement `part2_train_step()` function	15	
- [Q9] JEM (partial) training & report loss		5
- [Q10] Evaluate model on test dataset		2
Bonus Part (15)		
- [B1] Sample Step on Part 1	1	
- [B2] Negative Sample Step	1	
- [B3] Data Generation/Sampling	3	
- [B4] Best Image Sampling	2	
- [B5] Sample Step on Real Data	2	

Late submission will be accepted but with penalty.

The late penalty is deduction of one point (out of a maximum of 100 points) for every hour late after 11:59pm with no more than two days (48 hours). Being late for a fraction of an hour is considered a full hour. For example, two points will be deducted if the submission time is 01:23:34.

12 Academic Integrity

Please refer to the regulations for student conduct and academic integrity on this webpage: `https://registry.hkust.edu.hk/resource-library/academic-standards`.

While you may discuss with your classmates on general ideas about the assignment, your submission should be based on your own independent effort. In case you seek help from any person

or reference source, you should state it clearly in your submission. Failure to do so is considered plagiarism which will lead to appropriate disciplinary actions.