IIT BOMBAY

Assignment #2
Points: 80

Course: *CS 725* – Instructor: *Preethi Jyothi*
Due date: *11:59 pm, October 28, 2024*

General Instructions

- Download the file `assgmt2.tgz` from Moodle and extract the file to get a directory named `assgmt2` with all the necessary files within.
- Here are the TAs to approach for each part of the assignment:
  - Part I: Tejomay, Sabyasachi, Sona
  - Part II: Darshan, Snegha, Sabyasachi
  - Part III: Poulami, Sameer
  - Part IV: Amruta, Harsh
  - Part V: Soumen
- For your final submission, create a directory named `assgmt2` with the following internal directory structure:

```
assgmt2/
  |
  +- nn_template.py
  +- part2.pdf
  +- loss_bgd_1.png,loss_bgd_2.png,loss_bgd_3.png
  +- loss_adam.png
  +- convolution_1d_template.py
  +- template_2dconv.py
  +- part5.py [EXTRA CREDIT]
  +- kaggle.csv [EXTRA CREDIT]
```

  Compress your submission directory using the command: `tar -cvzf [rollno1]_[rollno2].tgz assgmt2` and upload this .tgz to Moodle. Make sure the filename is roll numbers of all team members delimited by "_". This submission is due on or before **11:59 pm on Oct 28, 2024**. No extensions will be entertained.
- STRICTLY FOLLOW THE SUBMISSION GUIDELINES. Any deviation from these guidelines will result in penalties.

Part I: Implement a Feedforward Neural Network (25 points)

For this problem, you will implement a feedforward neural network training algorithm from scratch. This network will have:

- An input layer
- One or more hidden layers with configurable dimensions (via `hidden_dims`) and activation functions (via `activations`)
- A single output node with a sigmoid activation and a binary cross-entropy loss

`nn_template.py` outlines the structure of the neural network with support for various activation functions and optimizers. Your task is to complete the missing portions of the code, labeled as **TODOs** in `nn_template.py`. This neural network is designed to classify data into binary classes (denoted by $0, 1$).

Firstly, implement the sigmoid, tanh functions and their derivatives. These will be used in both the forward and the backward passes of the network. Complete the missing functions `sigmoid(x)` in **TODO 1a**, `sigmoid_derivative(x)` in **TODO 1b**, `tanh(x)` in **TODO 1c** and `tanh_derivative(x)` in **TODO 1d**. `relu(x)` and `relu_derivative(x)` are already implemented. Note that the sigmoid function is $\sigma(x) = \frac{1}{1+e^{-x}}$, and its derivative is $\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$. The tanh function is $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, and its derivative is $\tanh'(x) = 1 - \tanh^2(x)$. **[2 pts]**

Next, implement the following functions within the `NN` class.

- `__init__`: Initialize the weights and biases for all layers, including the output layers. (E.g., in case of a single hidden layer, there are two sets of weights corresponding to the two affine layers between 1) the input and the hidden layer and, 2) the hidden layer and the output node). Use random initializations for all these weights by sampling from a Gaussian distribution with a mean of 0 and a standard deviation of 1. This is **TODO 2**. **[3 pts]**
- `forward`: Compute the activations for all the hidden nodes and the output node. Use the activation function corresponding to each layer as provided in the parameter `activations`. Use separate variables to compute the weighted sum of inputs coming into each node (e.g., $\mathbf{z}_1 = \mathbf{w}_1\mathbf{x} + \mathbf{b}_1$), and the output after applying the activation function to the weighted sum (e.g., $\mathbf{a}_1 = g(\mathbf{z}_1)$). This completes **TODO 3a** and **TODO 3b**. **[6 pts]**
- `backward`: In this function, you will compute gradients of the loss function with respect to all the weights and biases. Recall the expression for the binary cross-entropy loss for predictions $\hat{y}$ and target values y, given by $L(y, \hat{y})$:

  $$L(y, \hat{y}) = -[y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})]$$

  The derivative of this loss with respect to $\hat{y}$ is:

  $$\frac{\partial L}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1 - y}{1 - \hat{y}}\right)$$

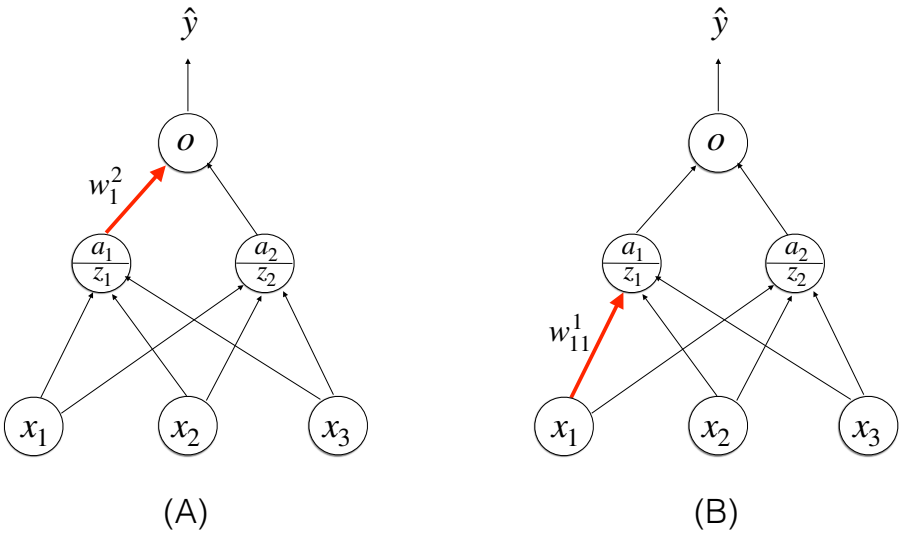

Figure 1

Let us work out all the gradient computations for a simple case with one hidden layer. Let's consider each layer at a time. First, the output layer. Consider a single weight w_1^2 in the output layer, highlighted in red in Figure 1(A) above.

$$\frac{\partial L}{\partial w_1^2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o} \cdot \frac{\partial (w_1^2 a_1 + w_2^2 a_2 + b_2)}{\partial w_1^2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o} \cdot a_1$$

where $\hat{y} = \sigma(o)$, hence $\frac{\partial \hat{y}}{\partial o} = \sigma(o)(1 - \sigma(o))$ and b_2 is the (scalar) bias weight corresponding to the output node. Gradients for all the other weights in the output layer can be similarly computed.
Next, the hidden layer. Consider a single weight w_{11}^1 highlighted in red in Figure 1(B) above.

$$\begin{aligned}
\frac{\partial L}{\partial w_{11}^1} &= \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o} \cdot \frac{\partial (w_1^2 a_1 + w_2^2 a_2 + b_2)}{\partial w_{11}^1} \\
&= \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o} \cdot w_1^2 \left(\frac{\partial a_1}{\partial w_{11}^1} \right) + 0 \\
&= \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial o} \cdot w_1^2 \cdot \sigma(z_1)(1 - \sigma(z_1)) \cdot x_1
\end{aligned}$$

since $a_1 = \sigma(z_1)$. Gradients for all the other weights in the hidden layer can be similarly computed.

Note that the above expressions are gradients for a single example. You will need to sum the gradients across all training examples.

Additionally, for the case of multiple hidden layers, you inductively propagate the gradients backward through the network using the chain rule (as discussed in class). These computations are part of **TODO 4a** and **TODO 4b**. **[12 pts]**

- `step_bgd`: Implement the code for a vanilla batch gradient descent update within this function using a static learning rate. This corresponds to a `gd_flag` value of 1. This will complete **TODO 5a**. Update the `weights` and `biases` of the affine layers by performing a gradient descent step, using their corresponding gradients, `delta_weights` and `delta_biases`, respectively. **[2 pts]**

`train`: Over `num_epochs`, the function `train` calls the forward and backward passes and computes gradients for all the training examples in a batch (stored in `X` and `y`) and updates the weights using an optimizer. The training loop also calculates train and test losses after every epoch.

If your implementation is correct, your code will converge in less than 30 epochs using `batch_size` = 100 and your test accuracy will be a perfect 1.0. During evaluation, we will also check your code on a new dataset.

Part II: Explainability in Neural Networks (20 points)

All modern neural networks, despite exhibiting remarkable performance, lack interpretability or explainability [a]. In this section, we will aim to design neural networks for two toy datasets that are both optimal (in not needing more layers than required for perfect separability) and interpretable.

You are given two binary classification datasets shown in Figure 2 and Figure 3. Points within the blue regions are labeled 1 and the rest are labeled 0. Assume that points on the boundaries are labeled 1. Design an optimal neural network for each dataset that perfectly classifies the points, and provide written explanations of what each neuron is aiming to do.

Note the following points when designing your neural network:

1. Your solution has to be a simple feed-forward neural network, with minimum number of hidden layers. (NOTE: *Partial points will be awarded for non-optimal but correct solutions that have more than the minimum number of layers.*)
2. Include an explanation of what each neuron in the hidden layers is doing.
3. Inputs to the NN are x_1 and x_2. No transformations to the inputs are allowed.
4. Assume the following threshold-based activation function $g(x;T)$ for all the neurons in your neural network:

$$g(x;T) = \begin{cases} 1 \text{ if } x \geq T \\ 0 \text{ otherwise} \end{cases}$$

5. No skip and self connections are allowed. That is, neurons in the i^{th} layer are only connected to neurons in the $(i-1)^{\text{th}}$ layer.

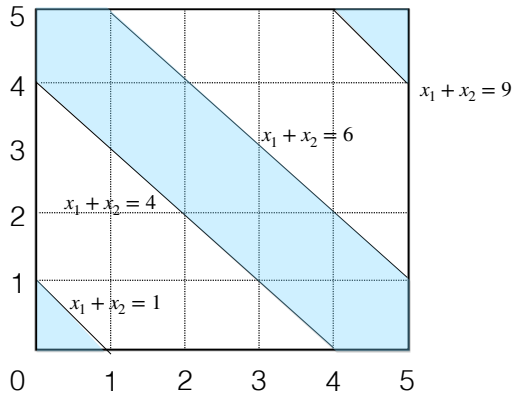

Figure 2: Bands of blue **[8 pts]**

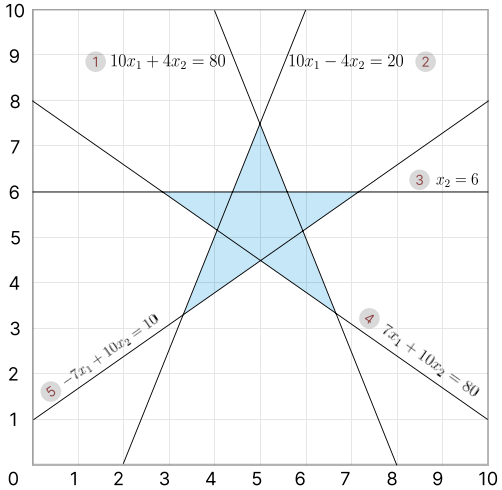

Figure 3: Catch the star **[12 pts]**

Here's a sample problem and its solution to illustrate what we expect in your answer. Note that you need to submit one pdf file titled `part2.pdf` that contains drawings of both neural networks and explanations for the hidden neurons. Feel free to use your code written in Part I to validate your answers. However, **DO NOT** submit any of these files. We will be grading solely based on your written solutions in the pdf.

[a] There is an entire sub-field of deep learning that focuses on explainability. Here's a survey paper.

Part III: Comparing Optimizers (5 points)

In this part, you will compare different variants of Gradient Descent (GD).

(A) Vanilla GD with exponentially decaying learning rate. You have already implemented vanilla GD with a static learning rate in Part I. Modify this implementation to include an exponentially decaying learning rate:

$$\eta_t = \eta_0 e^{-\lambda t}$$

where η_0 is the initial learning rate, t is the epoch number and λ is the decay constant.

(B) GD with momentum. Next, implement **GD with Momentum** whose update rule is modified to include a velocity term:

$$v_t = \beta v_{t-1} + (1-\beta)\nabla_w L(w_{t-1})$$

$$w_t \leftarrow w_{t-1} - \eta v_t$$

where β is the momentum parameter, $0 \leq \beta < 1$. We generally use g_t to represent $\nabla_w L(w_{t-1})$.

(C) Adam optimizer. As we learned in class, the **Adam optimizer** combines the benefits of both momentum and adaptive learning rates. The update rules for Adam are:

$$v_t = \beta v_{t-1} + (1-\beta) g_t$$

$$s_t = \gamma s_{t-1} + (1-\gamma) g_t \odot g_t$$

$$\hat{s}_t \leftarrow \frac{s_t}{1-\gamma^t} \quad \hat{v}_t \leftarrow \frac{v_t}{1-\beta^t}$$

$$w_t \leftarrow w_{t-1} - \frac{\eta}{\sqrt{\hat{s}_t} + \epsilon} \odot \hat{v}_t$$

In `nn_template.py`, within `step_bgd`, implement the two above-mentioned variants (A) and (B) of GD corresponding to `gd_flag` values of 2 and 3, respectively. These are **TODO 5b** and **TODO 5c**, respectively. For Adam, add your implementation to the function `step_adam`, which is **TODO 6**. **[4 pts]**

Compare loss plots. Once you have implemented all the variants, plot the loss curves for vanilla GD, GD with exponentially decaying learning rate, GD with momentum and Adam using the function `plot_loss` that is already implemented in `nn_template.py` file. Include the plots `loss_bgd_1.png`, `loss_bgd_2.png`, `loss_bgd_3.png`, `loss_adam.png` in your submission. Observe the convergence speed and final loss values achieved by each variant. **[1 pts]**

Part IV: De-convoluting Convolutions (30 points)

1D Convolutions. The 1D convolution in machine learning is actually the discrete convolution between two functions f and g, defined on the set of integers, $\mathbb{Z}$. The convolution of f and g is defined as:

$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

In the context of machine learning, since we deal with finite length arrays, the sum above is computed only for those indices that are within the bounds of the arrays. (For indices that are outside the bounds, say m which is greater than the length of array f, the corresponding discrete function at that point is taken to be zero). We will call f above as the “input" of the convolution, and g as the “kernel" or “filter" that is performing the convolution.

Padding. Let us quickly walk you through the concept of padding. Depending on what we want the output size to be, we can pad the input of the convolution (see Figure 4).

- Full padding implies that the input is padded to the largest possible extent to overlap with the kernel.
- Valid padding implies that the kernel is always positioned inside the input.
- Same padding is used to obtain an output of the same size as the input.

Imagine a 1×3 kernel and a 1×7 input array in Figure 4, to get the 1D analogues of padding.

Stride. Next, let us look at stride. Stride of a convolution operation dictates how many steps the kernel moves at each step. Stride is typically used to reduce the dimensions of the input; see Figure 5. Note that the stride and kernel size are two independent parameters. If the kernel size in Figure 5 was 4, and assuming valid padding, the output size would be 2×2.
Thus, if f and g are arrays of length L_f and L_g respectively, then the length of the convolution with stride 1 and full padding of f and g is $L_f + L_g - 1$.

(A) 1D Conv. Implement a function `Convolution_1D` in the file `convolution_1d_template.py` that takes in two 1-dimensional numpy arrays, and computes the 1D convolution of the two arrays, with a given stride and ‘valid’ or ‘full’ as possible padding modes. **[6 pts]**

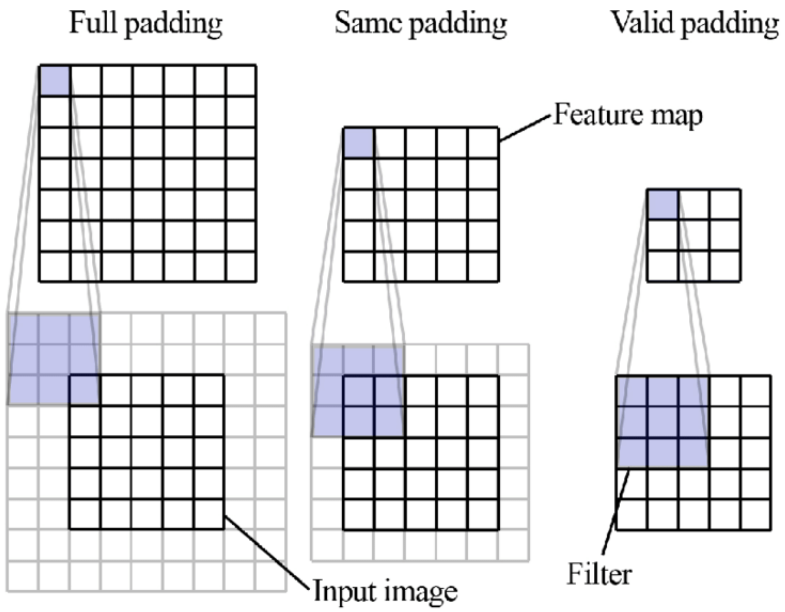

Figure 4: Full, same and valid padding

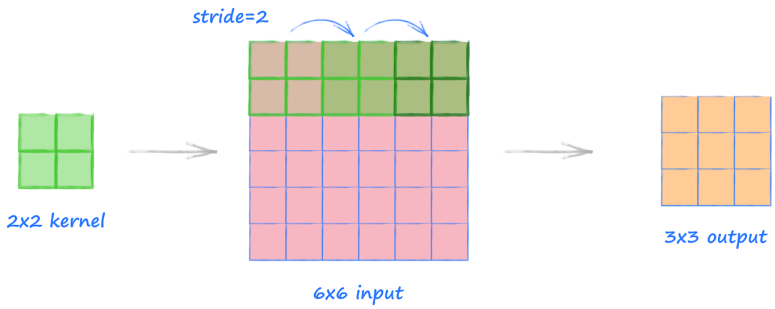

Figure 5: Illustration of a convolution with stride 2

(B) Two dice. You are given two dice, an n faced die and an m faced die. Both dice are biased, that is, the probabilities of landing on each face are not equal. You are given the probability mass functions of the two dice as two numpy arrays. For example, consider a 6-faced die A and a 3-faced die B with the following probability mass functions:

$$p_A = [0.1, 0.2, 0.3, 0.1, 0.1, 0.2]$$
$$p_B = [0.3, 0.4, 0.3]$$

The two dice are rolled together. Let us try to compute the probability that the sum of the faces of the two dice is equal to k. As a concrete example, for dice A and B above, let us try to compute the probability that the sum of the faces is equal to 7. The possibilities and their associated probabilities are:

$$(6_A, 1_B) : 0.2 \times 0.3 = 0.06$$
$$(5_A, 2_B) : 0.1 \times 0.4 = 0.04$$
$$(4_A, 3_B) : 0.1 \times 0.3 = 0.03$$

We can sum these to get the probability that the sum of the faces is equal to 7, which is $0.06 + 0.04 + 0.03 = 0.13$.

For another example, consider the probability that the sum of the faces is 2. The only possibility in this case is $(1_A, 1_B)$, which has a probability of $0.1 \times 0.3 = 0.03$.

Given the probability mass functions of the two dice, complete the function `probability_sum_of_faces` in the file `convolution_1d_template.py` that computes the probability mass function of the sum of the faces of the two dice. The function should take in two numpy arrays, `p_A` and `p_B`, which represent the probability mass functions of the two dice, and output a numpy array which represents the probability mass function of the sum of the faces of the two dice. In our example above, the output of the function for dice A and B would be: `np.array([0.03, 0.1 , 0.2 , 0.21, 0.16, 0.13, 0.11, 0.06])` *Hint: Use an appropriate call to* `Convolution_1D` *for this problem.* **[3 pts]**

2D convolutions. We now move to 2D convolutions. In this setting, a 2-dimensional kernel K moves over a 2-dimensional input image, and extracts useful features from the image. Convolution could also be used to downsample the image (i.e., make it smaller in dimensions) and keep only higher-level features. We will explore some of the applications of a convolution filter applied to an image. For this question, `pip install opencv-python`. Use only `numpy` and the `helper.py` functions in your code.

Note that we only want to code the forward pass of a 2D convolution, with a given kernel; no backpropagation is to be implemented. All new code will go into the file `template_2dconv.py`.

(C) Apply filter to image. Consider a function `apply_filter_to_patch` that takes a square image patch as input, applies the filter to the patch and returns a single pixel value. For example, if the filter performs average pooling, then this function would return a single value, which is the average of the pixels in the patch. Based on this information, write a function `movePatchOverImg` that takes any image, a filter size (of odd dimensions), and a filter (in the form of the `apply_filter_to_patch` function) as input and applies the filter over every square patch (of size equal to filter) of the image with stride 1. Your `movePatchOverImg` function must convert the image to grayscale. Use padding such that `movePatchOverImg` returns an output image, which is of the same dimensions as the input image. Note that it is mandatory that you only use the `numpy` library in your code. **[4 pts]**

(D) Edge detection. 2D convolution can be used for detecting features of images. These include vertical edges, horizontal edges, and all edges. An edge is an abrupt change in the intensity of the image. The convolution of the image with certain kernels can enhance only this abrupt change, thus detecting the edge. See Figure 6. Using only our helper functions `load_image` and `save_image` and the `numpy` library, write three functions `detect_horizontal_edge`, `detect_vertical_edge` and `detect_all_edges` which use different kernels (any size) to detect horizontal, vertical and all edges respectively. These functions must take a square image patch as input, apply the kernel to the patch and return a single pixel value. Pass these to your `movePatchOverImg` function with the provided image and save the output. **[6 pts]**

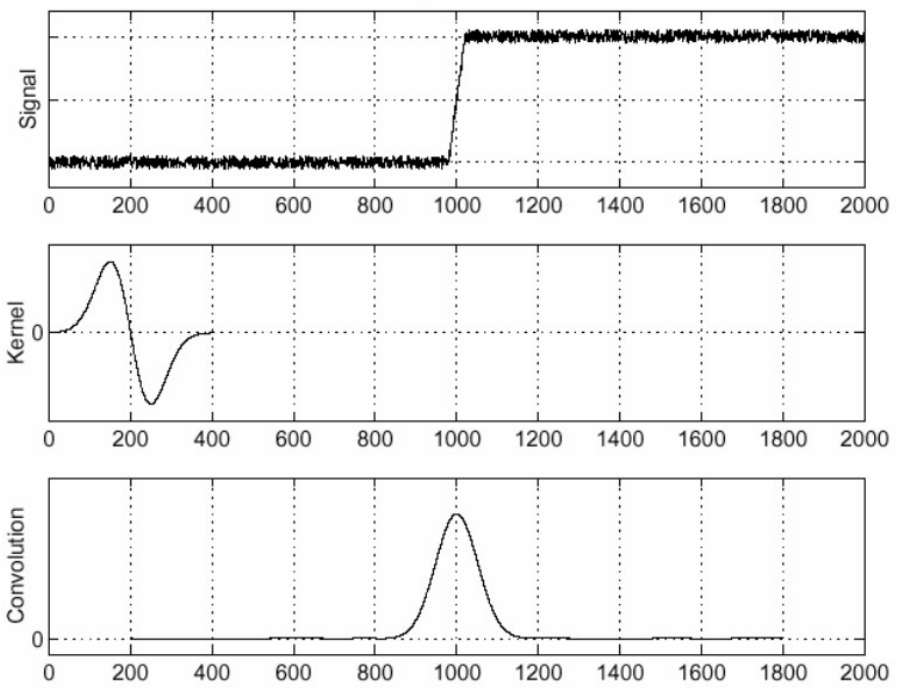

Figure 6: Illustration of edge detection using convolution.

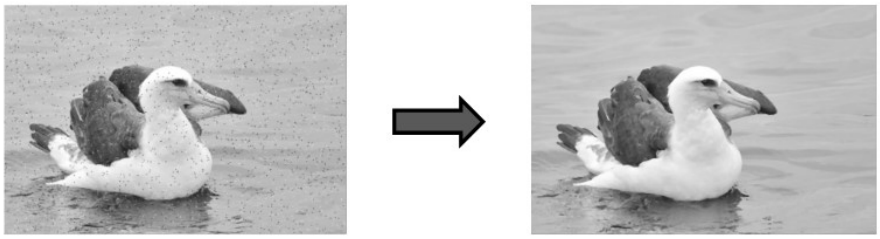

Figure 7: Illustration of noise removal.

(E) Image Denoising. Various filters can be used to pick up high and low frequency components of an image. Thus, they can be used for image denoising. This is because noise in images is usually high frequency and certain filters can be used to remove this high frequency component. You are given an image with noise added to it, `noisycutebird.png`. Using only our helper functions `load_image` and `save_image` and the `numpy` library, write a function `remove_noise` to remove this noise while maintaining the sharpness of the image edges. See Figure 7. This function must take a square image patch as input, apply the function to the patch and return a single pixel value. Pass these to your `movePatchOverImg` function with the provided image and save the resulting image. **[2 pts]**

(F) Unsharp masking. Unsharp masking is a technique of creating the illusion of a sharp image by enhancing the intensity of the edges of the image. A mask of the edges is created and added to the image. We will do this using Gaussian blur, which is 2D convolution of an image with a Gaussian kernel.

$$gaussian(x,y) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(x-\frac{\text{size}-1}{2})^2 + (y-\frac{\text{size}-1}{2})^2}{2\sigma^2}\right)$$

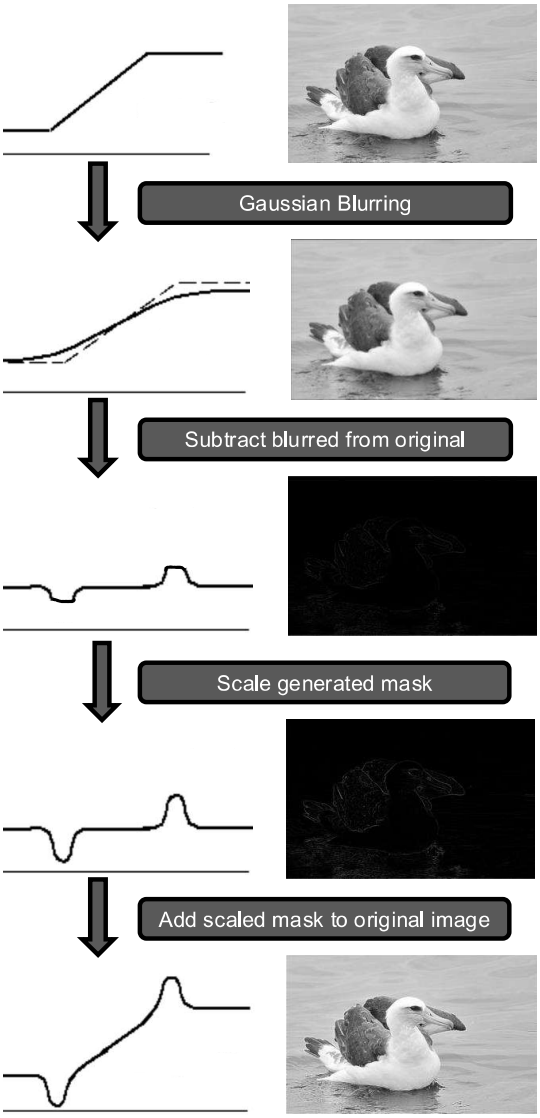

Figure 8: Illustration of unsharp masking

1. Write a function `create_gaussian_kernel` that takes as input a filter size (odd number) and a σ, and returns a square Gaussian kernel of that size. Ensure that kernel values add up to 1. **[2 pts]**

2. Write a function `gaussian_blur` to apply Gaussian blur to an image. This function must take a square image patch as input, generate a Gaussian kernel of size 25 and σ 1.0 using `create_gaussian_kernel` and apply the kernel to the patch, returning a single pixel value. **[2 pts]**

3. Write a function `unsharp_masking` that takes as input an image and a scaling factor and performs unsharp masking using the steps illustrated in Figure 8, to return the "sharp" image. Your `unsharp_masking` function should convert the image to grayscale. Use your `movePatchOverImg` function with your `gaussian_blur` function and a kernel size of 25. Take measures to prevent overflow of array values. Perform unsharp masking on the given image with an appropriate scaling factor and save the output image. **[5 pts]**

Use only our helper functions `load_image` and `save_image` and the `numpy` library. For all the tasks in this part, your saved output images should be grayscale.

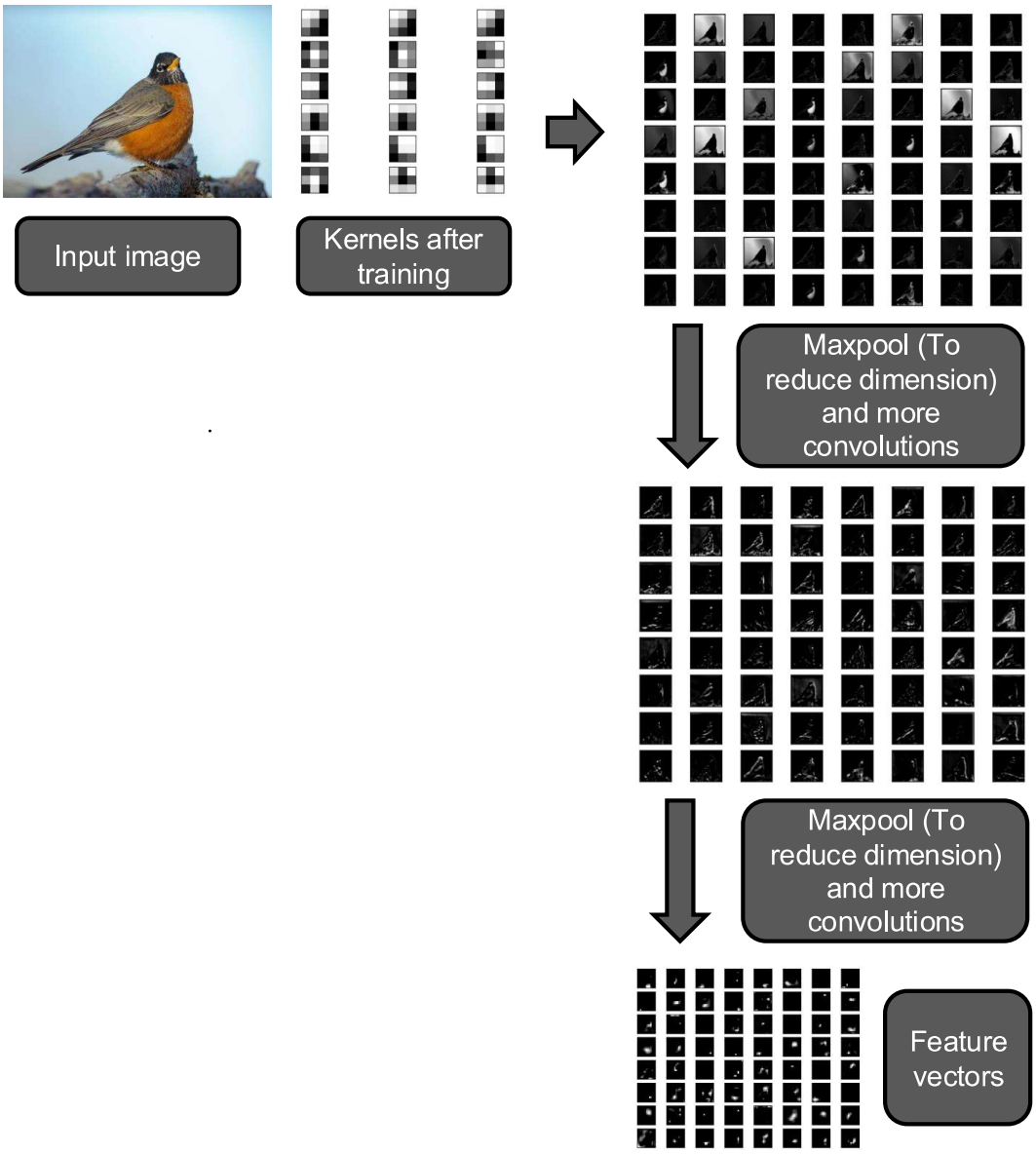

Figure 9: Illustration of how CNNs work

Expository note about CNNs. You have seen how different convolutional kernels are used to extract different features from images. This exact principle lays the foundation of convolutional neural networks (CNNs). Multiple kernels are used to create feature maps from an image. Max-pooling is then used to reduce the dimensions of these feature maps. Convolution is applied on these outputs. The process is repeated over and over, until low dimension feature vectors are obtained for the input image. These feature vectors can now be passed through a feed-forward neural network, for any downstream task on the input images. This process is illustrated in Figure 9. As a CNN is trained, during backpropagation, the kernel weight values are updated in such a way, that the relevant feature maps are extracted for the given downstream task. For more information, see here.

Kaggle Competition: Extra Credit (5 points)

Challenge: Classification from Image Features.

Overview. In this Kaggle competition, you will solve a realistic classification task to predict a class label (0 to 99) based on a set of visual features extracted from the publicly available ImageNet dataset. The final model will be evaluated on a test dataset via Kaggle, and test performance will be measured using **Accuracy**.
Competition link: You can join the competition on Kaggle: IIT Bombay CS 725 Assignment 2 (Autumn 2024). Please sign up on Kaggle using your IITB LDAP email ID, with your Kaggle "Display Name" set to the roll number of any member in your team. This is important for us to identify you on the leaderboard.

Dataset description. You are given three CSV files:

- `train.csv`: This file contains the training data with 64 features and a corresponding target label for each entry.
- `test.csv`: This file contains the test data with 64 features but without the target label.
- `sample.csv`: This file contains the submission format with predicted label for the test data. You will have to submit such a file with your test predictions.

Each row in the data files represent an instance with the following columns:

- ID: A unique identifier for each data point.
- `feature_0, feature_1, ..., feature_63`: The 64 features extracted from the dataset.
- `label`: The target label for each data point (only in `train.csv`).

Task description. Implement a classification model for the given problem. You are free to use any of the predefined neural network layers from PyTorch or other ML libraries with any choice of optimizers and regularization techniques. You do not need to stick to the code you've written in Part I. Tune the hyperparameters on a held-out set from `train.csv` to achieve best model performance on the test set. Predict the target label on the test dataset.

Evaluation. The performance of your model will be evaluated based on the classification accuracy calculated on the test dataset (automatically via Kaggle). Your model will be evaluated on the provided test set, where a random 50% of the examples are marked as *private* and the remaining are *public*. The final evaluation will be based on the private part of the test set, which will be revealed via the private leaderboard after the competition concludes.

Submission. Submit your source file named `part5.py` and a CSV file `kaggle.csv` with your predicted label for the test dataset, following the format in `sample.csv`. This is an extra credit problem. Top-scoring performers on the "Private Leaderboard" (with a fairly relaxed threshold determined after the deadline passes) will be awarded up to 5 extra points.